Rendering

- Rendering: computing all pixel color values to represent a 3D scene.
- Shading model (illumination model): method of modeling how lights interact with objects in a scene.
- Rendering hierarchy: rendering methods using different shading models that approximate physical lighting process at different levels of details, leading to different levels of realism.
  - Wireframe: draw edges only, hidden lines may and may not be removed.
  - Flat shading: each face is shaded uniformly, i.e. drawn with the same color.
  - Smooth shading: color changes smoothly on a surface using an interpolation technique.
  - Special effects: shadowing, texture mapping.
Lights (from either point light source or ambient light source) that reach the surface of an object can be:
- absorbed by the object (converted into heat)
- reflected and scattered from the surface (surface is visible through reflected lights)
- transmitted into and through the surface (translucent and transparent objects)

The surface material determines how much (fraction) light (of different wavelengths) is reflected and transmitted.
Reflections

- Important vectors:
  - \( s \): light direction; \( m \): surface normal;
  - \( v \): viewing direction (sight vector).
- Light reflection: combination (sum) of 
diffuse and specular reflections.
- The part of the reflected light that reaches
  the "eye" determines the color of the pixel.
  - Diffuse reflection: scattering of reflected
    lights in all directions equally, generating dull
    surfaces and more uniform lighting.
  - Specular reflection: highly directional
    mirror-like reflection, generating highlights
    and shiny surface.

Diffuse reflection

- The diffuse component of the light reflection:
  \[
  I_d = I_s \rho_d \cos(\theta) = I_s \rho_d \frac{\vec{s} \cdot \vec{m}}{||m||}
  \]
  where:
  - \( I_s \): light intensity (or one component of the light color)
  - \( \rho_d \): diffuse reflection coefficient. constant for each face
    or object.
- A more general form:
  \[
  I_d = I_s \rho_d f(d) \max\left(\frac{\vec{s} \cdot \vec{m}}{||m||}, 0\right)
  \]
  i.e.
  \[
  I_d = 0 \text{ if } \vec{s} \cdot \vec{m} < 0
  \]
  \( f(d) \): distance attenuation, e.g.
  \[
  f(d) = \frac{1}{ad^2 + bd + c}
  \]
Specular reflection

- Phong reflection model (Supported by OpenGL) provides simple highlights and plastic-like appearance.
- More complex specular effects such as shiny metallic appearance can only be modeled by global illumination model, e.g. ray tracing.
- Phong model: The reflected light is the strongest in the perfect reflection direction (mirror reflection). Its strength diminishes as the reflection angle deviates from the perfect reflection angle.
Phong reflection model

- Specular reflection intensity:

$$I_{sp} = I_s \rho_s \cos^f (\phi) = I_s \rho_s \left( \frac{\bar{r} \cdot \bar{v}}{||\bar{r}||^2} \right)^f$$

where $$r = -s + 2 \frac{s \cdot \bar{m}}{||\bar{m}||^2} \bar{m}$$

if $$\bar{r} \cdot \bar{v} < 0, I_{sp} = 0$$

$$\rho_s$$: specular reflection coefficient, assumed to be constant for each face or object.

$$f$$: shininess coefficient, larger $$f$$ represents more shiny material. Normally, $$1 \leq f \leq 200$$, and $$f_{mirror} = \infty$$

Specular reflection is directional, i.e. it depends on the viewing vector $$\bar{v}$$.
Halfway vector

- Phong model is expensive.
- Halfway vector: a cheaper alternative (by Jim Blinn) that avoids computing the reflection vector.

\[
\vec{h} = \vec{s} + \vec{v}
\]

\[
I_{wp} = I_s \rho_s \cos^f(\beta) = I_s \rho_s \left( \frac{\vec{h} \cdot \vec{m}}{\|\vec{h}\|} \right)^f
\]
Ambient light and global illumination

- **Local illumination (Phong model):** the shading of a point only depends on lights that directly reach the point, and the local geometry and material properties. Local illumination can be directly applied in a graphics pipeline (as in OpenGL).

- **Global illumination:** the shading of a point is affected by the light reflections of all other points (objects) in the scene. It involves solving a global rendering equation, and therefore cannot be directly applied in a graphics pipeline.

- In local illumination, points that are not directly reachable by a light source is not shaded (i.e. black). Ambient light is designed to provide a uniform light for all points in the scene, regardless of their physical locations.

- **Ambient reflection** = \( I_a \rho_a \)

  \( \rho_a \): ambient reflection coefficient; \( I_a \): ambient intensity
Combining light contributions

✓ The total amount of light that reaches the eye from a point:
\[ I = I_a \rho_a + I_d \rho_d \cdot L + I_s \rho_s \cdot P^f \]
where \[ L = \max(\frac{\vec{s} \cdot \vec{m}}{|\vec{s}|}, 0) \]
\[ P = \max(\frac{\vec{h} \cdot \vec{m}}{|\vec{h}|}, 0) \]
✓ In practice, \( I_d \) and \( I_s \) are normally the same.
✓ Multiple light sources: \[ I = I_a \rho_a + \sum_{j=1}^{n} (I'_a \rho'_a L + I'_s \rho'_s P^f) \]
✓ Color components:
\[ I_r = I_{ar}' \rho_{ar} \cdot L + I_{av} \rho_{av} \cdot P^f \]
\[ I_g = I_{ag}' \rho_{ag} + I_{sg} \rho_{sg} \cdot L + I_{sg} \rho_{sg} \cdot P^f \]
\[ I_b = I_{ab}' \rho_{ab} + I_{sb} \rho_{sb} \cdot L + I_{sb} \rho_{sb} \cdot P^f \]

<table>
<thead>
<tr>
<th>Material</th>
<th>metallic ( \rho_{mm} )</th>
<th>diffuse ( \rho_{dd} )</th>
<th>specular ( \rho_{sp} )</th>
<th>exponent ( f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black</td>
<td>0.00</td>
<td>0.01</td>
<td>0.50</td>
<td>32</td>
</tr>
<tr>
<td>Plastic</td>
<td>0.00</td>
<td>0.01</td>
<td>0.50</td>
<td>27.8974</td>
</tr>
<tr>
<td>Brass</td>
<td>0.238442</td>
<td>0.780392</td>
<td>0.982157</td>
<td>25.6</td>
</tr>
<tr>
<td>Bronze</td>
<td>0.2125</td>
<td>0.714</td>
<td>0.966048</td>
<td>76.8</td>
</tr>
<tr>
<td>Chrome</td>
<td>0.25</td>
<td>0.4</td>
<td>0.774597</td>
<td>12.8</td>
</tr>
<tr>
<td>Copper</td>
<td>0.19125</td>
<td>0.37038</td>
<td>0.256777</td>
<td>51.2</td>
</tr>
<tr>
<td>Gold</td>
<td>0.24725</td>
<td>0.75164</td>
<td>0.659281</td>
<td>9.84615</td>
</tr>
<tr>
<td>Pewter</td>
<td>0.10586</td>
<td>0.427451</td>
<td>0.3333</td>
<td>51.2</td>
</tr>
<tr>
<td>Silver</td>
<td>0.19225</td>
<td>0.59754</td>
<td>0.360273</td>
<td>89.6</td>
</tr>
<tr>
<td>Polished Silver</td>
<td>0.23425</td>
<td>0.2775</td>
<td>0.723911</td>
<td>89.6</td>
</tr>
</tbody>
</table>
Shading and OpenGL pipeline

- Shading is applied when objects are projected from 3D to 2D, carrying the shaded colors onto the 2D vertices.

```
BEGIN (GL_POLYGON)
for (i=0; i < 3; i++) {
  glNormal3f (norm[i].x, norm[i].y, norm[i].z);
  glVertex3f (pt[i].x, pt[i].y, pt[i].z);
}
END (
```

- The normal of each vertex is supplied by the user and will be transformed along with the vertex for shading computation.

Light sources in OpenGL

- Creating a light source: up to 8 light sources (GL_LIGHT0, ..., GL_LIGHT7) can be created.
  
```
glLightf (GL_LIGHT0, para_name, parameter);
glEnable (GL_LIGHTING);
glEnable (GL_LIGHT0);
```
- para_name = GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR:
  
  parameter = light color (RGBA), e.g. (I_a, I_d, I_s, I_b, 1)
- When para_name = GL_POSITION, parameter is the location of a point light: (x, y, z, w). If w = 0, it is a parallel light.
- Light sources have default values.

- Distance attenuation: 
  
  \[ f(d) = 1/(k_c + k_1 D + k_2 D^2) \]
  
  the coefficients may be defined in glLightf ()
OpenGL Lighting Model

- Lighting model: general rules in lighting.
- Ambient light
  \[ \text{GLfloat amb[]} = \{0.2, 0.3, 0.1, 1.0\}; \]
  \[ \text{glLightModelfv (GL\_LIGHT\_MODEL\_AMBIENT, amb);} \]
- Local viewpoint: specify whether to compute shading using the true viewing direction vector \( (v) \) or a constant \( (0,0,1) \).
  \[ \text{glLightModeli (GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, GL\_TRUE);} \]

Polygon sides

- Front face is defined by counter clockwise (CCW) vertex order while looking from the camera. The other side is the back face.
- \[ \text{glLightModeli (GL\_LIGHT\_MODEL\_TWO\_SIDES, GL\_TRUE);} \]
  (normals will be reversed when draw back faces.)
Moving light sources

- Light sources are treated the same as vertices, i.e. they are subject to ModelView transformations.
- Lights moving with camera:
  - `glMatrixMode (GL_MODELVIEW);` `glLoadIdentity ();`
  - `glLightfv (GL_LIGHT0, GL_POSITION, pos);`
  - `gluLookAt (...):` `draw object;`
- Lights moving independent of camera:
  - `glMatrixMode (GL_MODELVIEW);` `glLoadIdentity ();`
  - `glPushMatrix ();`
  - `glRotate d(...) ;` `glTranslated (...) ;`
  - `glLightfv (GL_LIGHT0, GL_POSITION, pos);`
  - `glPopMatrix ();`
  - `gluLookAt (...);` `draw object;`

Material properties in OpenGL

- Reflection coefficients can be separately defined for each color component in OpenGL.
- `glMaterialf (face, para_name, parameter);`
  - `face:` `GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`
  - `para_name:` `GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_SHININESS, GL_EMISSION.``
- OpenGL illumination model (green and blue are similar):
  \[ I_r = I_{mr} \rho_a + \sum_{j=1}^{n} f(d)(I_{ar} \rho_{ar} + I_{dr} \rho_{dr} L + I_{sr} \rho_{sr} P) \]
Polygon filling

- Polygon filling: filling pixels within a polygon. Pixels are normally filled scanline by scanline, and left to right.
  
  ```
  for (y = ymin; y <= ymax; y++) {
    find xmin and xmax for this scanline;
    for (x = xmin; x <= xmax; x++)
      find and fill the color for this pixel (x,y);
  }
  ```

- Pixel colors are determined by the polygon vertices and their normals.
- Flat shading uses the same normal for all vertices of a polygon.
- Smooth shading uses normals computed from the actual surface that polygon is approximating.

Surface normals

- Planar surface:
  - $Ax+By+Cz+D=0$ : $\vec{n} = (A, B, C)$
  - Polygon: $\vec{n} = (p_2 - p_1) \times (p_3 - p_2)$
- General implicit surface $f(x,y,z)=0$ :
  $$\vec{n} = \text{grad}(f) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$$
- Parametric surface $S(u,v) = (x(u,v), y(u,v), z(u,v))$
  $$\vec{n} = \frac{\partial S}{\partial u} \times \frac{\partial S}{\partial v}$$
- Example: a sphere surface
  $$S(u,v) = (r \cos(u) \cos(v), r \cos(u) \sin(v), r \sin(u))$$
Flat shading and smooth shading

- Shading model:
  - `glShadeModel (GL_FLAT)` or `glShadeModel (GL_SMOOTH)`
- Flat shading: Uses the same color for all pixels in a polygon. The color is pre-computed (i.e. outside the polygon filling loop) using the normal of the first vertex of the polygon.
  - Edges are visible;
  - Specular highlights are not well rendered.
- Smooth shading: Smooth color changes within a polygon. Edges are “smoothed out”. There are two typical smoothing shading methods: Gourand shading and Phong shading.
Gourand shading

• Compute the color for each vertex of the polygon, using an illumination model, and then interpolate the colors at the vertices to generate colors for all filling pixels.

```
compute colors at all vertices of the polygon;
for (each scanline QR)
    compute colors at Q and R by color linear interpolation on edges AB and CD.
    for (each pixel, P, on the scanline segment QR)
        compute color at P by linear interpolation on edge QR
```

Bilinear interpolation of colors:

\[
I_0 = (1-u) \cdot I_A + u \cdot I_B, \quad u = \frac{(x-t)}{(x_0-x_0)}
\]

\[
I_1 = (1-v) \cdot I_C + v \cdot I_D, \quad v = \frac{(y-t)}{(y_0-y_0)}
\]

\[
I_p = (1-t) \cdot I_0 + t \cdot I_1, \quad t = \frac{(z-t)}{(z_0-z_0)}
\]
Phong shading

- Compute the normal vector at each pixel within the polygon by bilinear interpolation of normal vectors at polygon vertices. And then compute the color the pixel using an illumination model.

\[
\begin{align*}
\text{for (each scanline QR)} & \quad \text{compute normals at Q and R by linear interpolation of normals at the end points of edges AB and CD}; \\
& \quad \text{for (each pixel, P, on the scanline segment QR); } \\
& \quad \text{compute normal at P by linear interpolation on edge QR; } \\
& \quad \text{compute color of P by an illumination model; }
\end{align*}
\]

Bilinear interpolation of normals:
\[
\begin{align*}
\vec{m}_Q &= (1-u) \cdot \vec{m}_A + u \cdot \vec{m}_B, \quad u = \frac{Q_x - x_Q}{x_A - x_Q} \\
\vec{m}_R &= (1-v) \cdot \vec{m}_C + v \cdot \vec{m}_D, \quad v = \frac{R_x - x_R}{x_C - x_R} \\
\vec{m}_P &= (1-t) \cdot \vec{m}_Q + t \cdot \vec{m}_R, \quad t = \frac{P_y - y_Q}{y_R - y_Q}
\end{align*}
\]

Gouraud shading vis Phong shading

- Gouraud shading is faster than Phong shading since shading computation is only performed at the vertices.
- Phong shading presents higher image quality since shading computation is done at each pixel.
- Highlights are not as well rendered in Gouraud shading as in Phong shading since color interpolation removes some highlights within the polygons.
- OpenGL implements Gouraud shading.
- Problems with concave polygons (both methods):
Hidden line and hidden surface removal

- Hidden line removal: wireframe display

- Hidden surface removal: depth order in surface rendering.

- Algorithms: depth sort (painter's), raycasting, space subdivision (Warnock, BSP, Octree), Z-buffer.

- OpenGL solution: Z-buffer (depth buffer)
Depth buffer (Z-buffer) approach

Depth buffer (Z-buffer): a framebuffer-like memory storing the depth value of the closest point to each pixel. During display, the \((i,j)\) location of Z-buffer stores a value \(d(i,j)\) representing the closest depth value so far at pixel location \((i,j)\). \(d(i,j)\) will be replaced if a closer point is drawn later to the same pixel.

\[
\text{for (each point } P \text{ to be drawn)} \\
\quad \text{compute the projected pixel location } (i,j); \\
\quad \text{if } (d(i,j) > \text{depth } (P)) \\
\quad \quad d(i,j) = \text{depth } (P); \quad \text{draw } P;
\]

Depth computation

The \(z\) value of a point after viewport transformation is scaled into \([0,1]\), i.e. \(0\) on the near plane, and \(1\) on the far plane. The \(d(i,j)\) values are initialized to \(1\).

Planar polygon with equation: \(ax+by+cz+q=0\)

if \((c=0)\) ignore the polygon; else

for (each scanline \(QR\))

\[d = -(ax+by+q)/c;\]

for (each subsequent pixel on the scanline)

\[d = d - a/c;\]

Non-planar polygon: bilinear interpolation:

\[
\begin{align*}
&d_Q = (1-u) \cdot d_A + u \cdot d_B, \quad u = \left(\frac{y_P-y_0}{y_N-y_0}\right) \\
&d_R = (1-v) \cdot d_C + v \cdot d_D, \quad v = \left(\frac{x_P-x_0}{x_N-x_0}\right) \\
&d_P = (1-t) \cdot d_Q + t \cdot d_R, \quad t = \left(\frac{y_P-y_R}{y_N-y_R}\right)
\end{align*}
\]
Depth buffer in OpenGL

✓ OpenGL implementation

```c
glutInitDisplayMode (GLUT_DEPTH);
 glEnable (GL_DEPTH_TEST);
 glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

✓ Pros & Cons:
- Simple to implement
- Can handle any type of object
- Less sensitive to scene complexity
- Requires large (high-speed) memory
- Precision in depth is limited by Z-buffer bit depth

Depth sorting: painter’s algorithm

✓ Priority list
- An object space algorithm
- It builds a depth priority list of all objects (polygons) based on their distances to the viewpoint; and then draws the objects in a reverse priority order.

```
Projection plane
  (3)
  (4)
  (5)
(2)
(1)
```

Drawing order: (5), (4), (3), (2), (1)
**Painter’s algorithm**

1. Compute bounding box for each polygon
2. Build a preliminary depth priority list using the maximum depth, \(d_{\text{max}}\), of each polygon. (The first in the list has the \(d_{\text{max}}\) farthest from the viewer)
3. For (each polygon \(P\) in the list)
   - let \(Q\) be the next polygon in the list;
   - if \((d_{\text{min}}(P) \geq d_{\text{max}}(Q))\) draw \(P\);
   - else
     - let \(S = \{Q: d_{\text{min}}(P) > d_{\text{max}}(Q)\}\);
     - apply a priority test sequence on all polygons in \(S\);
     - if (All polygons in \(S\) pass the test) draw \(P\);
     - else interchange \(P\) and \(Q\), go back to (3);
   - If (the same \(P\) and \(Q\) need interchange again later)
     - cyclical overlapping or intersection exists, split \(P\) along the plane of \(Q\), remove original \(P\) from the list, insert the two new polygons into the list, start the algorithm again.

**Priority test sequence**

- Are bounding boxes of \(P\) and \(Q\) disjoint in \(X\) or \(Y\)?
  
  \[P_{x_{\text{max}}} < Q_{x_{\text{min}}} \text{ or } P_{x_{\text{min}}} > Q_{x_{\text{max}}} \text{ or } P_{y_{\text{max}}} < Q_{y_{\text{min}}} \text{ or } P_{y_{\text{min}}} > Q_{y_{\text{max}}}\]

- Is \(P\) wholly on the far side of the plane of \(Q\)?
  
  Let \(p(x,y,z)=Ax+By+Cz+D=0\) be the plane equation of \(Q\). If \(p()\) have the same sign as \(\text{sign}(-D)\) for all vertices of \(P\), then \(P\) is wholly on the far side of \(Q\).

- Is \(Q\) wholly on the near side of the plane of \(P\)?
  
  Test is similar.

- \(Q\) passes the test if any of the tests answers yes.
Cyclical overlap

- Cyclical priority order:

- How to split a polygon by a plane?

Raycasting Algorithm

- Image space algorithm. It determines the visibility for each pixel by casting a ray, connecting the viewpoint and the center of the pixel, into the scene. The first intersection the ray has is the visible point for the pixel.

- A simple algorithm:

```plaintext
for (each pixel) {
    construct a ray;
    for (each object in the scene)
        compute and store the ray-object intersection points, on a point list;
    if (the point list is not empty)
        determine the closest point and its shading color, and set the pixel color;
    else set background color;
}
```
Optimization for raycasting

- **Bounding box**: Minimum and maximum coordinates of the object in X, Y, and Z directions.
- Testing ray intersection with bounding box may not be efficient: need transformation of ray or object.
- **Bounding sphere**: The minimum sphere enclosing the object. It can be directly computed from bounding box (not a tight bound), or by numerical minimization.
- Testing ray intersection with a sphere is easy: computing distance from the center of the sphere to the ray.

\[ d = \sqrt{\left| P_0 - C \right|^2 - \left( (C - P_0) \cdot \vec{V} / \left| \vec{V} \right|^2 \right)} \]

Optimization for raycasting (2)

- Clustering: Building a hierarchy of bounding spheres for groups of objects. Each ray will traverse the hierarchy tree in a depth-first order, but skip subtrees whose bounding boxes or spheres do not intersect the ray.
- Priority sorting: Pre-sort objects in priority order, and compute the ray-object intersection according to priority order - avoid unnecessary intersection computation.
- Spatial subdivision: hierarchical space subdivision. Only the objects that intersect the subspaces that the ray passes through need to be computed.

- Ray-polygon intersection
  1. Ray-plane intersection
  2. Inside test
Raycasting algorithm using a clustering hierarchy

for (each ray) {
    for (each cluster sphere on the hierarchy tree) {
        perform a bounding sphere test;
        if (the ray intersects the sphere) {
            if (the sphere is an object sphere, i.e. leaf node) {
                place the object on the active object list;
            } else {
                check child clusters;
            }
        } else {
            skip this subtree;
        }
    }
    if (active object list is empty) {
        display background color;
    } else {
        for (each active object) {
            compute ray-object intersection, if any, and place the intersection point(s) on the intersection list;
            if (intersection list is empty) {
                display background color;
            } else {
                determine the nearest intersection point, and display the shaded color at this point;
            }
        }
    }
}

BSP (Binary Space Partitioning)

- **Basic ideas:**
  - Suitable for highly complex but static scenes, with high rendering rate and rapidly changing view point (e.g. flight simulation).
  - Preprocess of the scene in the WCS such that the depth order for any view point can be easily obtained.
  - Based on the fact that object space can be divided into two half spaces by each polygon.
BSP tree

- A binary tree can be constructed by recursively subdividing the object space into halfspaces. Each halfspace is recursively subdivided by one of the polygons within the halfspace, until there is only one (or zero) polygon left in the halfspace. This BSP tree is not unique for a given scene.
- When the dividing plane intersects a polygon, the polygon will be split into two, one in each halfspace.
- To check which halfspace a polygon belongs to, a simple substitution with each polygon vertex to the dividing plane equation, \(Ax+By+Cz+D=0\), is sufficient. If the results are all positive or 0, the polygon is in "+halfspace", if all negative, it's in "-halfspace", otherwise splitting is necessary.
- A good BSP tree will have minimal splits.

BSP construction algorithm

```c
BSPtree BSPConstruct (polylist) {
  if (polylist is empty) return null;
  else
    SelectPolygon (polylist, root);
    backlist=null; frontlist=null;
    for (each remaining polygon, poly, on polylist)
      if (poly is in front of root)  AddtoBSPtree (polygon, frontlist);
      else if (poly is behind root) AddtoBSTree (polygon, backlist);
      else
        SplitPolygon (polygon, root, frontpart, backpart);
        AddtoBSPtree (frontpart, frontlist);
        AddtoBSPtree (backpart, backlist);
    lefttree = BSFConstruct (frontlist);
    righttree = BSFConstruct (backlist);
    return combine (lefttree, root, righttree);
}
```
BSP tree display

- The display algorithm
  
  \[\text{DisplayBSP} (\text{BSPtree}) \{\]
  
  \[\text{if (BSPtree is not empty)}\]
  
  \[\text{if (viewpoint is in front of root)}:\]
  
  \[\text{DisplayBSP (BSPtree.right)};\]
  
  \[\text{display root.polygon;}\]
  
  \[\text{DisplayBSP (BSPtree.left)};\]
  
  \[\text{else}\]
  
  \[\text{DisplayBSP (BSPtree.left)};\]
  
  \[\text{display root.polygon;}\]
  
  \[\text{DisplayBSP (BSPtree.right)};\]
  
  \[\}\]
  
- SelectPolygon (): testing with different polygons as roots. Experiments show that a few (5-6) tests are normally sufficient.

Octree

- Object space subdivision: recursively subdivide a cubic domain of the object space into 8 equal sized cubes (octants or voxels). Octants are numbered from 0 to 7 in a left-right, bottom-up and front-back order.

- Octree nodes: root represents the original object space, each level of the octree represents the corresponding level of subdivision, and leaf nodes represent octants that do not require further subdivision (based on certain criteria).

- Nodes are labels: F (full), E (empty), P (partially full, gray).

- A simple subdivision criterion
  
  \[\text{if ((node.type == 'P') \&\& (node.size > minimal_size) \&\& (node is above the minimal complexity level))}\]
  
  \[\text{then subdivide;}\]

- Minimal complexity level: the level of object complexity that is considered easy to solve. (e.g. only one polygon remains)
Octree display

Octree is constructed in WCS. The display order is determined based on the viewing direction.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Display order</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>7, 6, 5, 4, 3, 2, 1, 0</td>
</tr>
<tr>
<td>&lt;0</td>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>6, 7, 4, 5, 2, 3, 0, 1</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>&lt;0</td>
<td>&gt;=0</td>
<td>5, 4, 7, 6, 1, 0, 3, 2</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>&gt;=0</td>
<td>&lt;0</td>
<td>3, 2, 1, 0, 7, 6, 5, 4</td>
</tr>
<tr>
<td>&lt;0</td>
<td>&lt;0</td>
<td>&gt;=0</td>
<td>4, 5, 6, 7, 0, 1, 2, 3</td>
</tr>
<tr>
<td>&lt;0</td>
<td>&gt;=0</td>
<td>&lt;0</td>
<td>2, 3, 0, 1, 6, 7, 4, 5</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>&lt;0</td>
<td>&lt;0</td>
<td>1, 0, 3, 2, 5, 4, 7, 6</td>
</tr>
<tr>
<td>&lt;0</td>
<td>&lt;0</td>
<td>&lt;0</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
</tbody>
</table>

Octree display (2)

Display algorithm

for (each given viewing direction)

build an index for octree display order: index[0...7];

displayOctree (root);

PROCEDURE displayOctree (octNode)
{
    if (octNode is a leaf node)
        if (octNode is not empty)
            display octNode;
    else
        for (i = 0 ... 7)
            displayOctree (octNode.children[index[I]]);
}


Blending

- When blending is enabled, a color drawn into a pixel (source color) will be blended with the color that is already in that pixel (destination color): `glEnable(GL_BLEND);`
- Alpha value of a color is often used as the opacity of the color, i.e. low alpha value means high transparency.
- Blending is carried out through a blending function:
  
  $$\text{dstColor} = \text{srcFactor} \times \text{srcColor} + \text{dstFactor} \times \text{dstColor}$$

- `glBlendFunc(srcFactor, desFactor);`
- `srcFactor, dstFactor`: `GL_ZERO, GL_ONE, GL_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_COLOR, GL_ONE_MINUS_DST_ALPHA, ...`
- Linear blending: `dstFactor=(1-srcAlpha), srcFactor=srcAlpha
  
  $$C = (1-\alpha) C + \alpha C_i$$`

Transparency

- Simple transparency:
  
  `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`

- Using transparency with a depth buffer (for opaque objects):
  First draw all opaque objects with depth buffer enabled; then use `glDepthMask` to set the depth buffer to read-only. It allows to transparent objects to be drawn without wrongly set the depth buffer, but still detect transparent objects that are behind opaque objects (thus should not draw).
Texture Mapping

- Enhancing rendering realism by adding surface textures to faces of mesh objects.
- Texture image: a function \( f(s, t) : [0,1] \times [0,1] \rightarrow D \) where \( D \) is the color domain. It is often represented as a 2D image (bitmap), but can also be computed through a procedure or a mathematical function.
- Bitmap texture:
  - Texel: each pixel unit of the texture image.
  - Can be either a 2D array of texels or 1D array texels in a row-major order.
- Procedural texture: The texture function \( f(s, t) \) is computed through a procedure (e.g. fake sphere).
A texel in an $n \times m$ texture image:

- Access
  - Nearest neighbor:
    \[
    f(s, t) = \text{texture}([(\text{int})(s \cdot (n-1) + 0.5)]([(\text{int})(t \cdot (m-1) + 0.5)])
    \]
  - Bilinear interpolation

\[
\begin{align*}
&u = s \cdot (n-1) - (\text{int})(s \cdot (n-1)), \\
v = t \cdot (m-1) - (\text{int})(t \cdot (m-1)), \\
C_q &= (1-u) \cdot C_a + u \cdot C_b, \\
C_r &= (1-v) \cdot C_a + v \cdot C_b,
\end{align*}
\]
Mapping texture to surface

- Each surface point has a corresponding texture coordinate, \((s,t)\), within the texture image.
- In OpenGL, each polygon vertex can have a texture coordinate specified by:
  
  `glTexCoord2f (s,t)`

- Texture coordinate is a state variable (similar to normal)

  ```
  glBegin (GL_QUADS);
  glTexCoord2f (0.0, 0.0); glVertex3f (1.0, 2.5, 1.5);
  glTexCoord2f (0.0, 0.6); glVertex3f (1.0, 3.7, 1.5);
  glTexCoord2f (0.8, 0.6); glVertex3f (2.0, 3.7, 1.5);
  glTexCoord2f (0.8, 0.0); glVertex3f (2.0, 2.5, 1.5);
  glEnd ();
  ```

Texture Definition and Rendering

- A texture is defined by:

  ```
  glTexImage2D(GL_TEXTURE_2D, level, components, width,
              height, border, format, type, image)
  ```

  Where "image" is a 1D array of size "width x height x components" and type "type", storing the texture image with color format "format".

- For each surface point to be drawn to a pixel, its texture coordinate, \((s,t)\), is first computed, and corresponding color in the texture image, \(f(s,t)\), will then be found, and drawn to the pixel.

- Computing texture coordinate: bilinear interpolation - only work well with parallel projection.
**Perspective Interpolation**

- With perspective projection, equal-distance steps on screen space do not correspond to equal-distance steps in texture space -- Bilinear interpolation can lead to texture distortion.
- Perspective interpolation: given line segment $AB$ and its perspective projection $ab$. Let the 4th components of $a$'s and $b$'s homogeneous coordinates be $a_4$ and $b_4$. Then the texture coordinates at a point $p(f) = (1-f) \cdot a + f \cdot b$ will be:

$$s_p = \frac{(1-f) \cdot \frac{s_a}{a_4} + f \cdot \frac{s_b}{b_4}}{1 + f \cdot \frac{1}{b_4}},$$

$$t_p = \frac{(1-f) \cdot \frac{t_a}{a_4} + f \cdot \frac{t_b}{b_4}}{1 + f \cdot \frac{1}{b_4}}.$$
Interpolating z-values:

\[ Z_t = Z_1 + t(Z_2 - Z_1) = \frac{1}{Z_1} - \frac{1}{Z_1} (Z_2 - Z_1) \]

Interpolating attribute values:

\[ I_t = I_1 + t(I_2 - I_1) \]

Figure 2: The virtual camera is looking in the +z direction in the camera coordinate system. The image plane is at a distance of d in front of the camera. A, B and C are points on the primitive with attribute values \( I_1, I_2 \) and \( I_3 \) respectively, and their images on the image plane are \( a, b \) and \( c \) respectively. \( s \) and \( t \) are parameters used for linear interpolation.
Texture modulation - how texture colors are used?

- **Replace (Glowing):** use texture color to directly replace the pixel color without computing shading.
  
  ```
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
  ```

- **Modulate (Painting):** multiple the texture colors to the diffuse and ambient components in shading formula.
  
  ```
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
  L = texture(s,t)[I_dρ_d + I_aρ_aL] + ρ_pP'
  ```

- There are several other modulation modes, such as:
  
  GL_BLEND, GL_ADD, GL_DECAL

Texture parameters

- Many texture mapping parameters can be set by:
  
  ```
  glTexParameter(GL_TEXTURE_2D, pname, param)
  ```

  - **pname = GL_TEXTURE_MIN_FILTER:** minifying function options, determining how a pixel color is computed if the pixel is mapped to an area greater than one texel. (LINEAR, NEAREST, MIPMAP, etc).

  - **pname = GL_TEXTURE_MAG_FILTER:** magnification function options, determining how a pixel color is computed if the pixel is mapped to an area smaller than one texel. (NEAREST, LINEAR, etc)

  - **pname = GL_TEXTURE_WRAP_S or GL_TEXTURE_WRAP_T:** setting wrapping parameters in s or t direction. (CLAMP, REPEAT, etc.)

  - **pname = GL_TEXTURE_BORDER_COLOR, etc.**
Texture objects

- A texture object stores texture data so that it can be used later. Many texture objects can be generated, each has its own texture data and parameters.
- Generating \( n \) names (system generated positive integers) in \( \text{nameArray} \) for texture objects:
  \[ \text{glGenTextures}(n, \text{nameArray}); \]
- \( \text{glBindTexture}(\text{GL_TEXTURE_2D}, \text{name}); \)
  - Creates a texture object with the given name, and binds the texture object to the next texture data to be created using \( \text{glTexImage2D} \)
  - Make this texture object current, so that subsequent texture operations are applied to this texture object.
  - Can be called multiple times, but only the first time creates the texture object.

Other OpenGL texture functions

- Texture image files: texture file reading functions are needed to read texture images from image files. It depends on image file format, e.g. jpeg, gif, ppm, bmp, etc.
- \( \text{glEnable}(\text{GL_TEXTURE_2D}) \): Enable 2D texture mapping
- \( \text{glBindTexture}(\text{GL_TEXTURE_2D}, \text{name}); \)
  - (1) Assign a name to the next texture to be created using \( \text{glTexImage2D} \);
  - (2) bind the named texture to the current 2D texture (make active).
- \( \text{glHint}(\text{GL_PERSPECTIVE_CORRECTION_HINT}, \text{GL_NICEST}); \)
  - requests the use of perspective interpolation in texture rendering.
- Example: rotating cube with 6 texture images.
Example: rotating cube with 6 texture images

Automatically generated texture coordinates

```c
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP );
glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_GEN_T );
glEnable( GL_TEXTURE_2D );
```
**Portable Pixmap (PPM) image files**

- PPM file has the simplest file format, sometimes also called raw format.
- The file has a header followed by image data.
  - Any line starting with a "#" is a comment line
  - First line is always: "P6"
  - Second line has two numbers: the width and height of the image.
  - The next line has one single integer number: the maximum value of any RGB color component.
  - The image data is in binary form, containing the byte stream of the pixel values stored in a row by row order. Each pixel has three bytes (its RGB values).
  - fread() and fwrite() are the best ways to read and write the binary portion of the image data.

![PPM image file example](test.ppm)

**OpenGL Quadrics**

- Quadrics: mathematical surfaces represented by 2nd degree polynomial functions: cylinder, cone, sphere, etc.
  \[a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5xz + a_6yz + a_7x + a_8y + a_9z + a_{10}\]
- OpenGL quadrics (quadric object): quadrics with certain drawing properties. A quadrics (with drawing properties) can be used to draw various types of quadric surfaces such as cylinder, cone and sphere.
- Using quadrics: a quadric object is first created, assigned drawing properties, and then applied to various quadric surfaces for rendering.
  ```c
  GLUquadricObj *gluNewQuadric();
  ```
- Drawing properties: Draw style; Normal; Orientation; Texture coordinates.
Drawing properties

- Draw style: filled, wireframe, etc.
  
  `gluQuadricDrawStyle (GLUquadricObj *quadObj, GLenum style);`
  
  `style: GLU_FILL (default), GLU_LINE, GLU_POINT, GLU_SILHOUETTE.`

- Normal: how normals are generated.
  
  `gluQuadricNormals (GLUquadricObj *quadObj, GLenum normalMode);`
  
  `normalMode: GLU_NONE (no normal), GLU_FLAT (normals for polygons only), GLU_SMOOTH (normals for vertices).`

Drawing properties (2)

- Orientation: Normal pointing to outside (default) or inside.
  
  `gluQuadricOrientation (GLUquadricObj *quadObj, GLenum orientation);`
  
  `orientation: GLU_OUTSIDE (default), GLU_INSIDE.`

- Texture Coordinates: generate texture coordinates or not.
  
  `gluQuadricTexture (GLUquadricObj *quadObj, GLboolean useTextureCoords);`
  
  `useTextureCoords: GL_TRUE, GL_FALSE (default).`

- Cleaning up:
  
  `gluDeleteQuadric (GLUquadricObj *quadObj);`
Quadric surfaces

- Disk: a flat circle with possibly a hole in the middle.
  - `gluDisk (quadObj, innerRadius, outerRadius, slices, loops)`
  - `slices`, `loops`: number of subdivisions for polygon approximation.

- Partial disk:
  - `gluPartialDisk (quadObj, innerRadius, outerRadius, slices, loops, startAngle, sweepAngle)`

- Sphere:
  - `gluSphere (quadObj, radius, slices, stacks)`
  - `slices`, `stacks`: for polygon approximation.
  - Texture coordinates:
    - `s`: rotation angle around Z-axis
    - `t`: Z coordinate

Quadric surfaces (2)

- Cylinders (and cones):
  - `gluCylinder (quadObj, baseRadius, topRadius, height, slices, stacks)`
  - When either `baseRadius` or `topRadius` is 0, it becomes a cone.
  - Texture coordinates:
    - `s`: angle
    - `t`: height (Z-coordinates).
Drawing quadrics

- A quadric object is first created, assigned drawing properties, and then applied to various quadric surfaces for rendering.
  
  ```
  GLUquadricObj *gluNewQuadric();
  ```

- Draw style: `GLU_FILL` (default), `GLU_LINE`, etc.
  
  ```
  gluQuadricDrawStyle (GLUquadricObj *quadObj, GLenum style);
  ```

- Normal: `GLU_NONE`, `GLU_FLAT`, `GLU_SMOOTH`
  
  ```
  gluQuadricNormals (GLUquadricObj *quadObj, GLenum normalMode);
  ```

- Orientation: `GLU_OUTSIDE` (default) or `GLU_INSIDE`.
  
  ```
  gluQuadricOrientation (GLUquadricObj *quadObj, GLenum orientation);
  ```

- Cleaning up: `gluDeleteQuadric (GLUquadricObj *quadObj);`

Quadric surface texture mapping

- Setting quadrics texture coordinates: `GL_TRUE` or `GL_FALSE`
  
  ```
  gluQuadricTexture (GLUquadricObj *quadObj, GLenum useTextureCoords);
  ```

- Sphere: `gluSphere (quadObj, radius, slices, stacks)`
  
  slices, stacks: for polygon approximation. texture coordinates automatically generated.

- Cylinders (and cones):
  
  ```
  gluCylinder (quadObj, baseRadius, topRadius, height, slices, stacks)
  ```

  texture coordinates automatically generated
1D and 3D textures

✓ 1D texture: can be considered a 2D texture with a height 1. It is often used for drawing color bands or curves.
  
  unsigned char texture[128];
  glTexImage1D(GL_TEXTURE_1D, level, components, width, border, format, type, texture);

✓ 3D texture: can be considered a stack (depth) of 2D textures. It is often used for medical imaging visualization, solid texturing, and volume rendering.
  
  unsigned char texture[width*height*depth*3];
  glTexImage3D(GL_TEXTURE_3D, level, components, width, height, depth, border, format, type, texture);

3D texture mapping examples
Curved surface texture mapping

- Assuming a curved surface is represented by a parametric surface: \( S(u,v) = (x(u,v), y(u,v), z(u,v)) \), and approximated by a polygon mesh.
- The main task is to find the proper texture coordinates for the vertices of the polygon mesh.

![Texture function S(u,v)](image)

Cylindrical surfaces

- Parametric representation:
  \[
  \begin{align*}
  x(\theta, h) &= r \cos(\theta) \\
  y(\theta, h) &= r \sin(\theta) \\
  z(\theta, h) &= h
  \end{align*}
  \]

- Mapping function:
  \[
  [0,1] \times [0,1] \rightarrow [\theta_a, \theta_b] \times [h_a, h_b]
  \]

  \[
  \begin{align*}
  s &= (\theta - \theta_a)/(\theta_b - \theta_a) \\
  t &= (h - h_a)/(h_b - h_a)
  \end{align*}
  \]
Surfaces of revolution

- Parametric representation
  \[ x(\theta, v) = X(v) \cos(\theta) \]
  \[ y(\theta, v) = X(v) \sin(\theta) \]
  \[ z(\theta, v) = Z(v) \]

- Mapping function:
  \[ s = (\theta - \theta_a) / (\theta_b - \theta_a) \]
  \[ t = (v - v_a) / (v_b - v_a) \]

- Using enclosing cylinder: the texture is first mapped to the enclosing cylinder, and then projected back to the surface along the cylinder’s normal direction.

Spherical surfaces

- Parametric representation
  \[ x(\theta, \phi) = r \cos(\phi) \cos(\theta) \]
  \[ y(\theta, \phi) = r \cos(\phi) \sin(\theta) \]
  \[ z(\theta, \phi) = r \sin(\phi) \]

- Linear mapping:
  \[ s = (\theta - \theta_a) / (\theta_b - \theta_a) \]
  \[ t = (\phi - \phi_a) / (\phi_b - \phi_a) \]

- Triangular mapping:
  \[ (\theta, 90^\circ) \rightarrow (0.5, 1.0) \]
  \[ (\theta_a, 0.0) \rightarrow (0.0, 0.0) \]
  \[ (\theta_b, 0.0) \rightarrow (1.0, 0.0) \]
Bump Mapping

- Bump mapping simulates surface roughness by perturbing normal vectors. It works better than mapping a texture of rough surface image (Fig 8.49).
- Normal perturbation: Given a surface $P(u,v)$, with normal vectors: $n(u,v) = P_u \times P_v / |P_u \times P_v|$, the perturbed surface is:
  $$P'(u,v) = P(u,v) + \text{texture}(s(u),t(v)) \cdot n(u,v)$$
  and
  $$n'(u,v) = n + \text{texture}_u \cdot (P_u \times n) + \text{texture}_v \cdot (n \times P_v)$$

  where $\text{texture}(s,t)$ is a scalar valued perturbation function.

- The rendering process will use the original surface vertices (i.e. $P(u,v)$) and the perturbed normals (i.e. $n(u,v)$).
Perturbation function (texture)

- The perturbation function can be a mathematical function (e.g. $\sin(au)\sin(bv)$) or a sampled gray level image.
Environmental mapping

- Allowing surrounding/background scene to be seen through reflections on object surfaces (Fig 8.63).
- An environmental texture is mapped to the interior surface of a surrounding sphere or cube, and reflected back to object surfaces (Fig 8.64).
- The environment map is reflected onto the object surfaces:

  Trace a ray from eye to a point P on the object surface, determine the reflection ray, and then trace the reflection ray until it hits the surrounding sphere or cube. The color from environment map at this hit-point will be placed at point P.

Environment mapping - OpenGL solution

- A simplified solution using automatically generated texture coordinates.

  // Build the environment map as a texture image
  // Generate texture coordinates automatically
  glTexGenf (GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  glTexGenf (GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
  glEnable (GL_TEXTURE_GEN_S);
  glEnable (GL_TEXTURE_GEN_T);
  // bind the environment texture and draw the object
Mipmaps

- Using different resolutions of a texture image for different levels of detail rendering.
  - Level-0: the original texture (primary texture). The width and height need to be a power of 2 (e.g. 64x64)
  - Level-1: half size of level-0 in both s and t directions. (32x32)
  - Level-2: half size of level-1 (16x16)
  - Level-3 (8x8): level 4 (4x4): level 5 (2x2): level 6 (1x1).
- Defining mipmap:
  
  `glTexImage2D (GL_TEXTURE_2D, level, GL_RGB, ...)`
  where level = 0, 1, 2, ...

- A GLU mipmap function: generating a series of mipmaps automatically from a primary texture image.
  
  `gluBuild2DMipmaps (GL_TEXTURE_2D, ...,)`
Mipmap filters

✓ Mipmap minification filters need to be applied when using mipmap textures. E.g.:

- **GL_NEAREST_MIPMAP_LINEAR**: using nearest mipmap to the polygon resolution, and using GL_LINEAR filter.
- **GL_LINEAR_MIPMAP_NEAREST**: using linear interpolation between the two mipmaps closest to the polygon resolution, and the GL_NEAREST filter.

✓ Example:

```gl
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, 64, 64, GL_RGB, GL_UNSIGNED_BYTE, texImage);
```
**Global illumination**

- **Global illumination model:** lights that reach a point by reflection from or transmitted through other objects are also considered in shading computation.
- Most visible surface algorithms do not support global illumination.
- Graphics pipeline architecture is not suitable.
- **Ray-tracing model:** tracing the light rays coming to the eye backwards.

**Ray-tracing shading model**

- **Whitted global illumination model:**
  \[
  I = f(d)(k_a I_a + k_d \sum (I_{L_j} \hat{n} \cdot \hat{L_j}) + k_s \sum (I_{E_j} \hat{n} \cdot \hat{H_j})) + k_s I_s + k_f I_f
  \]
  - The first three terms are local shading results.
  - \(I_s\) and \(I_f\) are recursively computed at their nearest intersections using the same whitted model.
  - \(F(d)\) is a distance attenuation function.
  - A shadow ray may be used at each intersection point (pointing to a light source) for shadow test. If the shadow ray is blocked by an opaque object, this point is in shadow and the local shading will not be computed for this light source.
Ray-tracing process

Is is computed recursively at each point through the “r” ray; It is computed recursively at each point through the “p” ray.

Recursive Ray-tracing algorithm

```c
RT_trace (RT_ray ray, int depth) {
    determine the closest intersection;
    if (object_hit) {
        compute normal at intersection
        return RT_shade (object_hit, ray, intersection, normal, depth);
    }
    else    return background_color;
}

RT_shade (obj, ray, point, normal, depth) {
    color = ambient term;
    for (each light source) {
        sRay=ray to light source
        compute shadow ray testing for the light source;
        if (not blocked) compute local shading for this light source;
    }
```
Recursive Ray-tracing algorithm (cont.)

if (depth < maxDepth){ /* return if depth is too deep */
    rRay = ray in reflection direction from point;
    rColor = RT_trace (rRay, depth+1);
    color += rColor scaled by ks;
    tRay = ray in refraction direction from point;
    tColor = RT_trace (tRay, depth+1);
    color += tColor scaled by kt;
}
return color;

Extension to the recursive ray tracing algorithm

- Distance attenuation: compute distance to the first intersection, and accumulate the distance through recursion.
- Adaptive termination condition: Terminate the recursion at a point when the total shading without recursion (assuming Is=Ic=1) is less than a threshold.
- Shadow by semi-transparent objects: scale the local shading based on the opacities (alpha values) of the objects blocking the shadow ray.
- Cone / Beam tracing: Using a cone or a pyramid to cover the entire pixel area (the ray only cover the center point) - better rendering quality, and can avoid missing small objects.
- Object clustering and bounding sphere hierarchy: reducing intersection computation.
Examples
Radiosity

- Ray tracing does simulate global diffuse reflection.
- **Light Energy Conservation**: in a closed environment (enclosure), light energy emitted or reflected by each surface will be reflected or absorbed by other surfaces in the same enclosure.
- **Radiosity**: the rate at which energy leaves a surface (the sum of the rates of emission, reflection and transmission.)
- Radiosity method first determines all light interactions in an enclosure in a view independent way, and then renders the scene from different view points.
- Surfaces and light sources are treated uniformly.
- Radiosity is treated the same as color intensity in the rendering process. 
  \[ B = \pi \cdot I \]
Surface Patches and Radiosity

- Surfaces are broken into \( N \) discrete patches, each patch emits and reflects light uniformly (i.e. having a uniform radiosity).
- Assuming patches are opaque and perfectly diffuse:
  \[
  B_i = E_i + \rho_i \sum B_j (A_j F_{ji} / A_i)
  \]
  - \( B_i \): radiosity of patch-\( i \) (energy/unit time/unit area)
  - \( E_i \): rate at which light is emitted from patch-\( i \)
  - \( A_i, A_j \): areas of patch-\( i \) and patch-\( j \)
  - \( F_{ji} \): Form factor: fraction of energy leaving patch-\( j \) that arrives at patch-\( i \).
  - \( \rho_i \): Surface reflectivity (fraction of incident energy reflected from surface)

Radiosity Equation

- Since \( A_i F_{ij} = A_j F_{ji} \)
- We have: \( B_i = E_i + \rho_i \sum B_j F_{ji} \)
- Or:
  \[
  \begin{bmatrix}
  1 - \rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} & \vdots \\
  -\rho_2 F_{21} & 1 - \rho_2 F_{22} & \cdots & -\rho_2 F_{2n} & \vdots \\
  \vdots & \vdots & \ddots & \vdots & \vdots \\
  -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1 - \rho_n F_{nn} & \vdots \\
  \end{bmatrix}
  \begin{bmatrix}
  B_1 \\
  B_2 \\
  \vdots \\
  B_n \\
  \end{bmatrix}
  =
  \begin{bmatrix}
  E_1 \\
  E_2 \\
  \vdots \\
  E_n \\
  \end{bmatrix}
  \]

- \( \rho_i \) and \( E_i \) are wavelength dependent (color components)
- \( F_{ij} \) only depend on the geometry and positions of the patches, and can be pre-computed independent of viewpoint and wavelength.
- Solving radiosity equation: Gauss-Seidel iteration
Other properties of form factors

\[ A_i \cdot F_{ij} = A_j \cdot F_{ji} \]
\[ \sum_{i=1}^{n} F_{ij} = 1 \quad F_{ii} = 0 \text{ (if all patches are planar or context)} \]

Form Factors \((F_{ij})\)

- **Solid angle**: \( \omega = \frac{A}{r^2} \), where \( A \) is the area of a spherical surface intersected by the cone with the center of the sphere as apex and angle \( \omega \)
- **The differential solid angle**: \( d\omega = \frac{dA}{r^2} \)
  \[ dA = (\sin \phi) \cdot d\varphi \cdot d\theta \cdot r^2 \quad d\omega = (\sin \phi) \cdot d\varphi \cdot d\theta \]
Computing Form Factors

Consider the solid angle from a differential surface $dA_i$:

$$d\omega = (\cos \phi_j / r^2) \cdot dA_j$$

From $I_i = dB_i / (\cos \phi_i \cdot d\omega)$, $B_i = \pi \cdot I_i$, $dB_i = I_i \cdot \cos \phi_j \cdot \cos \phi_j \cdot dA_j / r^2$

we have: $dB_i \cdot dA_i = (B_i \cdot \cos \phi_i \cdot \cos \phi_j) / (\pi \cdot r^2) \cdot dA_i \cdot dA_j$

therefore: $F_{dA_i \cdot dA_j} = dB_i \cdot dA_i / (B_i \cdot dA_i) = (\cos \phi_j \cdot \cos \phi_j) / (\pi \cdot r^2) \cdot dA_i$

$$F_{dA_i \cdot dA_j} = F_{ij} = 1 / A_i \int_{A_j} (\cos \phi_j \cdot \cos \phi_j) / (\pi \cdot r^2) \cdot H_{ij} \cdot dA_i \cdot dA_j$$

where:

$$H_{ij} = \begin{cases} 1, & \text{if there is no occlusion between } dA_i \text{ and } dA_j \\ 0, & \text{otherwise} \end{cases}$$

The Hemisphere

✓ For a given differential area $dA_i$ of patch-$i$, computing $F_{dA_i \cdot dA_j}$ is equivalent to:

1) Projecting the visible part of $A_j$ onto a unit hemisphere centered at $DA_i$

2) Projecting the projected area orthographically down onto the hemisphere’s unit circle base.

3) Computing the ratio of the projected area to the unit circle area
Hemisphere and form factor

- Two patches having the same projected area on the sphere will have the same form factor
- When $A_i$ is small, $F_{dAi-Aj}$ can be used to approximate $F_{ij}$, with the center of patch-i as the center of the hemisphere
- Approximating hemisphere using hemicube

Hemicube

- A unit hemicube is used to replace hemisphere. Each face of the hemicube is divided into a number of equal-sized cells.
Computing form factor using hemicube

- Patches are clipped against the viewing frustum defined by the center of the hemicube and each of its 5 faces. The clipped patches are projected onto the hemicubes faces.

\[ F_{ij} = \frac{1}{A_j} \left( \int_{A_j} \left( \cos \varphi_i \cdot \cos \varphi_j \right) \left( \frac{\pi \cdot r^2}{H_{ij} \cdot dA_j} \right) dA_j \right) \]

\[ = \int_{A_j} \left( \cos \varphi_i \cdot \cos \varphi_j \right) \left( \frac{\pi \cdot r^2}{H_{ij} \cdot dA_j} \right) \]

Form Factor Increment

- For each cell \( P \), a form factor increment can be pre-computed:

\[ \Delta F_p = \left( \cos \varphi_i \cdot \cos \varphi_j \right) / \left( \pi \cdot r^2 \right) \cdot \Delta A \]

\( \Delta A \) is the area of the cell.
\( \varphi_i \) is the angle between the hemicube’s Z axis (normal direction of the patch) and the vector connecting the center of the hemicube and cell \( P \), and \( \varphi_j \) is the angle between this vector and the normal of the cell.
Form Factor Increment

- For each patch $j$: $F_j = \sum_{p} \Delta F_p$
- All $\Delta F_p$ are pre-computed (independent of other patches). Due to symmetry, only 1/8 of the top face cells and 1/4 of the side face cells need to be computed.

Computing $\Delta F_p$

Let cell P have coordinates $(x,y,z)$ in the hemicube local coordinate system
- For top face:
  \[
  r = \sqrt{x^2 + y^2 + 1} \quad \cos \varphi_i = \cos \varphi_j = 1/r \]
  \[
  \Delta F_p = \Delta A / (\pi(x^2 + y^2 + 1)^2) \]
- For YZ side face (other side faces are similar):
  \[
  r = \sqrt{y^2 + z^2 + 1} \quad \cos \varphi_i = z/r, \quad \cos \varphi_j = 1/r \]
  \[
  \Delta F_p = z \cdot \Delta A / (\pi(y^2 + z^2 + 1)^2) \]
Radiosity Rendering

- View independent.
- Vertex radiosity:
  \[ B_o = (B_1 + B_2 + B_3 + B_4)/4 \]
- For boundary vertex:
  \[ B'_f = B_1 + B_2 - B_o \]
- For Corner vertex:
  \[ B'_g = 2B_1 - B_o \]
- Scan conversion and bilinear interpolation (Gouraud shading)

Special Effect: Billboardig

- Using a texture mapped polygon (billboard) to simulate the rendering of complex objects (e.g. trees, particle systems, remote scenes, etc.)
- Main problem: When the camera rotates, the rotation of the billboard doesn’t appear 3D.
- Solution: Keeping the normal of the billboard polygon perpendicular to the projection plane.
  - finding two vectors on the billboard plane that will become parallel to the projection plane after the ModelView transformation.
  - Transforming the projection plane vectors by the inverse of the ModelView matrix.
Billboarding (2)

✓ Getting the current ModelView matrix:
  glGetFloatv(GL_MODELVIEW_MATRIX, M);
✓ The transpose of this matrix will be its inverse. The first two columns of the upper 3x3 submatrix form the two vectors (right vector and up vector) on the billboard plane.

\[
M = \begin{bmatrix}
m_{11} & m_{12} & m_{13} & m_{14} \\
m_{21} & m_{22} & m_{23} & m_{24} \\
m_{31} & m_{32} & m_{33} & m_{34} \\
m_{41} & m_{42} & m_{43} & m_{44}
\end{bmatrix}
\]

\[
\text{right} = \begin{bmatrix}
m_{11} \\
m_{21} \\
m_{31} \\
m_{41}
\end{bmatrix}
\]

\[
\text{up} = \begin{bmatrix}
m_{12} \\
m_{22} \\
m_{32} \\
m_{42}
\end{bmatrix}
\]

✓ Computing the billboard polygon: billboard corners
  center + up*height + right*width
  center + up*height - right*width
  center - up*height + right*width
  center - up*height - right*width

Billboarding example
Special Effect: Fog

- Use of Fog
  - Simulating natural fog.
  - Reducing scene complexity (hiding invisible objects)
  - Smooth transition of visibility
- OpenGL fog: blending a pixel color with fog color based on object distance, fog density and fog mode.
- Enabling: glEnable(GL_FOG);
- Setting fog parameters: glFog(pname, param);
  - pname can be:
    - GL_FOG_MODE: GL_LINEAR, GL_EXP, GL_EXP2
    - GL_FOG_DENSITY: density
    - GL_FOG_COLOR: Cf
    - GL_FOG_START: start
    - GL_FOG_END: end

Fog blending equations

- Blending pixel color, \( C_p \), and Fog color, \( C_f \), using a blending factor \( f \):
  \[ C = f \times C_p + (1-f) \times C_f \]
- GL_LINEAR:
  \[ f = \frac{(end - z)}{(end - start)} \]
- GL_EXP:
  \[ f = e^{(-density \times depth)} \]
- GL_EXP2:
  \[ f = e^{(-density \times depth)^2} \]
Special effect: Particle Systems

✓ A technique to simulate and render objects/phenomena that do not have clear surface definitions (fire, water, smoke, explosion, etc.)
✓ Particles are entities (e.g. points) with attributes acting independently, and collectively.
✓ Particles attributes: velocity, color, life span, etc.
✓ Particles in the same particle system have similar attributes - the system as a whole will generate a common and cumulative effect (e.g. smoke particle system, fire particle system, water particle system).
Particle attributes

- **Position**: affected by velocity. Previous positions may also be needed for tracking purpose.
- **Velocity**: affected by force/acceleration.
- **Life span**: the period from its generation (by the source) to its disappearance. It may affect many other attributes.
- **Size**: may vary during its life span.
- **Weight**: useful in calculating effects of forces
- **Representation**: visual appearance in the scene
  - 3D points: often for remote scenes
  - Lines: a trace of particle trajectory
  - Texture mapped quads: most widely used. E.g. spark particles
- **Color**: may change during its life span

Particle system attributes

- **Particle list**: the list of all particles alive and the maximum number of particles it is allowed to generate
- **Position**: The point where new particles are generated. **Emission rate**: how often particles are created. Needs to keep track of time and maximum particles allowed.
- **Force**: assign a force (can be different) to each particle to produce acceleration and velocity.
- **Default particle attributes**: attributes assigned at the time a particle is created.
- **Current state**: whether to continue produce particles or not (e.g. out of view, expired time limit, etc.)
- **Blending**: blending functions used when drawing the particles
- **Representation**: representation stored in the particle system if all particles have the same representation.
Particle system effect

- Design issues in particle systems
  - Common sense. E.g. smoke rises, spark has a short life span, etc.
  - Physics. Simulating the physics of the particles will give better realistic effect.
  - Templates. Using part of the existing particle systems as templates.
  - Experiments. Trial-and-error using a basic particle system.
Special Effect: Shadows

- Hard shadow (clear boundary and even darkness): from single and focused light
- Soft shadow (with fuzzy boundary and uneven darkness): from multiple and ambient lights
- Projective shadows
- Stenciled shadow volumes

Projective shadow

- Using projection matrix to cast the vertices from the light source to the plane where the shadow will appear.
- The projection matrix is:

\[
\begin{bmatrix}
    p - a \cdot x & -a \cdot y & -a \cdot z & -a \cdot w \\
    -b \cdot x & p - b \cdot y & -b \cdot z & -b \cdot w \\
    -c \cdot x & -c \cdot y & p - c \cdot z & -c \cdot w \\
    -d \cdot x & -d \cdot y & -d \cdot z & p - d \cdot w
\end{bmatrix}
\]

where \( \text{lightPos} = (x, y, z, w) \); \( \text{planeCoefficients} = (a, b, c, d) \)
\( p = ax + by + cz + dw \)
Projective shadow (2)

- Multiplying the shadow matrix to the current ModelView matrix, and draw the object with:
  - Black color without lighting and texturing
  - Blending enabled
- Depth test should be disabled, but making sure to draw the shadow before objects in front of it.
- Restricting shadow: the shadow should be restricted to only the surface area on which the shadow is drawn. This can be accomplished using a stencil buffer
- Only hard shadows
- Multiple light sources: create shadows for each light source.
- Have to know the shadow plane (too complicated for complex environment)

Example
Stenciled shadow volume

- Allows shadowing on all surfaces of the scene.
- Shadow volume: the volume formed by the light source and the edges of the object. Shadows are generated by checking which objects are within the shadow volume.
- Checking whether a point is within the shadow volume: casting a ray from eye to the point, and count the number of intersection with the shadow volume. Increment the count when you intersect a front face and decrease the count when you intersect a back face. This can be implemented using the stencil buffer.
- OpenGL implementation: four pass rendering approach

Stenciled shadow volume steps

1. Render the scene normally using ambient and emissive lighting only.
2. Render the shadow volume to the stencil buffer (with depth write and color buffer disabled). It also counts the number of times the shadow volume is entered (and passed the depth test) using the stencil buffer.
3. Render a third pass, similar to (2), but only decrement the stencil buffer when a back face is drawn to a pixel (with depth test passed).
4. Draw the scene again at pixels with stencil value 0. This time, all normal lighting and texturing will be activated.
Fractals

- A technique used to define objects without well-defined surfaces, e.g. mountains, clouds, trees, etc.
- Fractal geometry methods: procedures rather than equations are used to model the surfaces of fractal objects.
- Two basic characteristics: infinite details, and self-similarity

Fractal generation procedures

- The procedure $F$ is iteratively applied to some elements or subparts (e.g. edges, faces, etc.) of the object to generate higher levels of details:

$$E_1 = F(E_0), E_2 = F(E_1), E_3 = F(E_2), \ldots$$

where $F$ usually replaces each element or subpart of the object with multiple scale-down subparts with some kind of transformations.
Deterministic self-similar fractals

We start with a simple geometric shape, called the \textit{initiator}. Subparts of the initiator are iteratively replaced by a pattern, called the \textit{generator}.

Fractal Dimension

A measure of "roughness, or fragmentation, of the object.

\[ d = \text{fractal dimension} \]
\[ s = \text{scale of the subparts} \]
\[ n = \text{number of subparts used to replace a lower level subpart} \]

\[ n \cdot s^d = 1 \text{ or } d = \frac{\ln(n)}{\ln(1/s)} \]
Fractal dimension (examples)

- $s = 0.5$, $n = 4$, $d = 2$
- $s = 0.5$, $n = 8$, $d = 3$

- $s = \frac{1}{6}$, $n = 18$, $d = 1.613$
- $s = \frac{1}{3}$, $n = 2$, $d = 0.633$
- $s = \frac{1}{8}$, $n = 16$, $d = 1.333$
Statistically self-similar fractals
Other fractal models

- Random walk: draw connected line segments with random directions and random lengths.

- Brownian motion fractals
  - Adding an additional parameter to the statistical distribution describing Brownian motion. It sets the fractal dimension for motion path.
  - A single Brownian path generates a fractal curve
  - A 2D array of Brownian motion path (elevations) over a planar grid (ground) generates terrain, or mountain-like models.

- Self-Squaring fractals: repeatedly applying a transformation function to points in a complex space
Line Rasterization

✓ The process of determining which pixels provide the best approximation to the desired line segment.
✓ Line rasterization algorithms need to be:
  1. Fast
  2. Accurate at the two end points of the line segment
  3. Constant in brightness (thickness) for lines of all orientations

DDA: Digital Differential Analyzer

✓ Basic idea:
  \[ Y_{i+1} = Y_i + \Delta x \cdot \frac{(y_2 - y_1)}{(x_2 - x_1)} \]
  where \( \frac{(y_2 - y_1)}{(x_2 - x_1)} \) is a constant, and \( \Delta x \) is the pixel width.
✓ Pixel center and pixel coordinates
The DDA algorithm

if \(|x_2 - x_i| > |y_2 - y_i|\) then length = |x_2 - x_i|
else length = |y_2 - y_i|

\[\Delta x = \frac{(x_2 - x_i)}{\text{length}}\]
\[\Delta y = \frac{(y_2 - y_i)}{\text{length}}\]

\(x = x_i + 0.5; \ y = y_i + 0.5\)

for (i = 1 to length) {
set ((int)(x), (int)(y))
\(x = x + \Delta x; \ y = y + \Delta y\)
}

DDA example

\(P_1 = (0, 0), \ P_2 = (7, 4)\)
Bresenham’s algorithm

✓ Basic idea: The decision of picking the closest pixel at every step can be made by only checking the sign of the error from the pixel to the line.
✓ Floating point algorithm
✓ Integer algorithm: easier for hardware implementation

---

- The algorithm (for 0 <= Δy <= Δx only)

**Floating-point version**

\[
\begin{align*}
x &= x_1 \\
y &= y_1 \\
Δx &= x_2 - x_1 \\
Δy &= y_2 - y_1 \\
m &= Δy / Δx \\
e &= m - 0.5 \\
\text{for } (i = 1 \text{ to } Δx) \{ \\
  &\text{set } (x, y) \\
  &\text{while } (e > 0) \{ \\
  &\quad y = y + 1 \\
  &\quad e = e - 1 \\
  &\} \\
  &x = x + 1 \\
  &e = e + m \\
}\end{align*}
\]

**Integer version**

\[
\begin{align*}
x &= x_1 \\
y &= y_1 \\
Δx &= x_2 - x_1 \\
Δy &= y_2 - y_1 \\
e' &= 2 \cdot Δy - Δx \\
\text{for } (i = 1 \text{ to } Δx) \{ \\
  &\text{set } (x, y) \\
  &\text{while } (e' > 0) \{ \\
  &\quad y = y + 1 \\
  &\quad e' = e' - 2 \cdot Δx \\
  &\} \\
  &x = x + 1 \\
  &e' = e' + 2 \cdot Δy \\
}\end{align*}
\]
Polygon Scan Conversion

- Polygon filling by inside test
  - Bounding box
  - Inside test

Scan line coordinate system
Parity scan conversion

- Basic steps:
  - Scanline/polygon intersection
  - Sorting of intersection points in X direction
  - Pairing intersection points for inside segments
  - Draw (setting pixels) the inside segments
  - Special cases:
    - Vertices
    - Horizontal edges

Active List polygon scan conversion

- Coherences in intersection and X sorting
  - The Y-bucket for each scan line: a linked list of edges with which the scan line is the first one to intersect.
    - \( x \): x coordinate of the first intersection
    - \( \Delta x \): the increment in X from scan line to scan line
    - \( \Delta y \): the number of scan lines the edge intersects
  - Active edge list: a linked list of all active edges of the current scan line
  - Active edge: an edge that intersects the current scan line.
    - \( x \): x coordinate of the current intersection
    - \( \Delta x \): the X increment of this edge
    - \( \Delta y \): the remaining number of scan lines that will intersect this edge
for (each polygon edge) {
    // determine the first (highest) scan line intersected by the edge.
    // place the edge into the y_bucket of this first scan line.
    // store in the y_bucket the initial X intersection value, the
    // number of scan lines intersected by the edge (dx), and the
    // edge's X increment (dx).
    
    active_edge_list = empty
    for (each scan line from the highest to the lowest) {
        for (each edge in the scan line's y_bucket) {
            insert the edge into the active_edge_list, using an
            increasing order with respect to X values.
        }
        
        extract pairs of intersection points from the active_edge_list,
        and draw the scan line segments defined by the pairs.
        
        for (each edge on the active_edge_list) {
            \( \Delta y = \Delta y - 1 \)
            
            if (\( \Delta y < 0 \)) drop the edge from the active_edge_list
            
            \( x = x + \Delta x \)
        }
    }

---

Seed filling: connectivity

- 4-connected and 8-connected boundaries

- 4-connected and 8-connected regions

8-connected

4-connected

8-connected

4-connected
Seed filling polygon

- At least one pixel inside the polygon
- The algorithm fills the interior region of a polygon defined by the boundary pixels
- A 8-connected algorithm only fills 4-connected polygons.
- An 4-connected algorithm fills both 4-connected and 8-connected polygons.
- A stack can be used to keep track of the seed expansion

The algorithm

Draw polygon boundary edges by a line rasterization algorithm
Push the seed pixel onto the stack

while (the stack is not empty) {
    - pop a point from the stack
    - set the pixel to the filling color
    - for (each of the connected (4- or 8-) pixels of the current pixel) {
        if (the color of the pixel is neither the boundary color nor the filling color)
            push the pixel onto the stack
    }
}