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:
  \[ I = f(d)(k_a I_a + k_d \sum (I_{L_j} (\hat{n} \cdot \hat{L}_j)) + k_s \sum (I_{E_j} (\hat{g} \cdot \hat{E}_j)')) + k_i I_s + k_I, \]
  - The first three terms are local shading results.
  - \( I_s \) and \( I_t \) 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

Recursive Ray-tracing algorithm

\[
\text{Is is computed recursively at each point through the "r" ray; It is computed recursively at each point through the "p" ray.}
\]
Recursive Ray-tracing algorithm (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 I_s=I_t=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 \( A_i F_j = A_j F_i \)
- We have: \( B_j = E_i + \rho \sum B_i F_i \)
- Or:
  \[
  \begin{pmatrix}
  1 - \rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1s} \\
  -\rho_2 F_{11} & 1 - \rho_2 F_{22} & \cdots & -\rho_2 F_{2s} \\
  \vdots & \vdots & \ddots & \vdots \\
  -\rho_s F_{11} & -\rho_s F_{22} & \cdots & 1 - \rho_s F_{ss}
  \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_s \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_s \end{pmatrix}
  \]
- \( \rho \) 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

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

- Solid angle: \( \omega = A/r^2 \), 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: \( d\omega = dA/r^2 \)
  \[
  dA = (\sin \varphi) \cdot d\varphi \cdot d\theta \cdot r^2 \\
  d\omega = (\sin \varphi) \cdot d\varphi \cdot d\theta
  \]
Computing Form Factors

Consider the solid angle from a differential surface \( dA_i \):

\[
d\omega = (\cos \varphi_j / r^2) \cdot dA_j
\]

From \( I_j = dB_i / (\cos \varphi_j \cdot d\omega) \), \( B_j = \pi \cdot I_j \), \( dB_i = I_j \cdot \cos \varphi_i \cdot \cos \varphi_j \cdot dA_j / r^2 \)

we have: \( dB_j \cdot dA_i = (B_j \cdot \cos \varphi_i \cdot \cos \varphi_j) / (\pi \cdot r^2) \cdot dA_j \cdot dA_j \)

therefore: \( F_{dA_i, dA_j} = dB_i \cdot dA_i / (B_j \cdot dA_j) = (\cos \varphi_i \cdot \cos \varphi_j) / (\pi \cdot r^2) \cdot dA_j \)

\( F_{A_i \rightarrow A_j} = F_{ij} = 1 / A \int_A \int_A (\cos \varphi_i \cdot \cos \varphi_j) / (\pi \cdot r^2) \cdot H_{ij} \cdot dA_i dA_j \)

where:

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

Other properties of form factors

\[
A_i \cdot F_{ij} = A_j \cdot F_{ji}
\]

\[
\sum_{j=1}^{a} F_{ij} = 1 \quad F_{ii} = 0 \quad (\text{if all patches are planar or context})
\]
The Hemisphere

- For a given differential area $dA_i$ of patch-i, computing $F_{dA_i-A_j}$ 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, $F_{dA_i-A_j}$ 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.

\[ F_y = \frac{1}{A_j} \int_{A_i} \left( \int_{A_j} \frac{(\cos \phi_i \cdot \cos \phi_j)}{(\pi \cdot r^2) \cdot H_y} \cdot dA_j \right) dA_i \]

\[ = \int_{A_j} \frac{(\cos \phi_i \cdot \cos \phi_j)}{(\pi \cdot r^2) \cdot H_y} \cdot dA_j \]
For each cell $P$, a form factor increment can be pre-computed:

$$\Delta F_p = (\cos \phi_i \cdot \cos \phi_j) / (\pi \cdot r^2) \cdot \Delta A$$

$\Delta A$ is the area of the cell.

$\phi_i$ 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 $P$.

$\phi_j$ is the angle between this vector and the normal of the cell.

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 $P$ has coordinates $(x, y, z)$ in the hemicube local coordinate system.

- **For top face:**
  \[ r = \sqrt{x^2 + y^2 + 1} \quad \cos \varphi_i = \cos \varphi_j = 1/r \]
  \[ \Delta F_p = \Delta A/\left(\pi (x^2 + y^2 + 1)^2\right) \]

- **For YZ side face (other side faces are similar):**
  \[ r = \sqrt{y^2 + z^2 + 1} \quad \cos \varphi_i = z/r, \quad \cos \varphi_j = 1/r \]
  \[ \Delta F_p = z \cdot \Delta A/\left(\pi (y^2 + z^2 + 1)^2\right) \]

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:**
  \[ B_c = 2B_1 - B_o \]
- **Scan conversion and bilinear interpolation (Gouraud shading)**
Special Effect: Billboarding

- 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:
  
  ```c
  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 =
  \[
  \begin{bmatrix}
  m_{11} & m_{12} & m_{13} & m_{14} \\
  m_{21} & m_{22} & m_{23} & m_{24} \\
  m_{31} & m_{32} & m_{33} & m_{34} \\
  m_{41} & m_{42} & m_{43} & m_{44}
  \end{bmatrix}
  \]
  
  right = \[
  \begin{bmatrix}
  m_{11} \\
  m_{21} \\
  m_{31} \\
  m_{41}
  \end{bmatrix}
  \]
  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`
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:
  \[
  f = (\text{end} - z)/(\text{end} - \text{start})
  \]

- GL_EXP:
  \[
  f = e^{-(\text{density} \times \text{depth})}
  \]

- GL_EXP2:
  \[
  f = e^{-(\text{density} \times \text{depth})^2}
  \]
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.

Fractal Dimension

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

- $d =$ fractal dimension
- $s =$ scale of the subparts
- $n =$ number of subparts used to replace a lower level subpart

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

Deterministic self-similar fractals

- We start with a simple geometric shape, called the *initiator*. Subparts of the initiator are iteratively replaced by a pattern, called the *generator*.
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**

![Diagram of pixel center and pixel coordinates]

---

**The DDA algorithm**

```plaintext
if (|x_2 - x_1| > |y_2 - y_1|) then length = |x_2 - x_1|
else length = |y_2 - y_1|
\Delta x = (x_2 - x_1) / length
\Delta y = (y_2 - y_1) / length
x = x_1 + 0.5; y = y_1 + 0.5
for (i = 1 to length) {
    set ((int)(x), (int)(y))
    x = x + \Delta x; y = y + \Delta y
}
```

![Diagram of DDA algorithm]

---

*Note: The images contain graphical representations of the pixel center and pixel coordinates, as well as the DDA algorithm.*
**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 $0 \leq \Delta y \leq \Delta x$ only)

<table>
<thead>
<tr>
<th>Floating-point version</th>
<th>Integer version</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = x_1$</td>
<td>$x = x_1$</td>
</tr>
<tr>
<td>$y = y_1$</td>
<td>$y = y_1$</td>
</tr>
<tr>
<td>$\Delta x = x_2 - x_1$</td>
<td>$\Delta x = x_2 - x_1$</td>
</tr>
<tr>
<td>$\Delta y = y_2 - y_1$</td>
<td>$\Delta y = y_2 - y_1$</td>
</tr>
<tr>
<td>$m = \Delta y / \Delta x$</td>
<td>$e' = 2 \times \Delta y - \Delta x$</td>
</tr>
<tr>
<td>$e = m - 0.5$</td>
<td>$e = m - 0.5$</td>
</tr>
<tr>
<td>for ($i = 1$ to $\Delta x$) {</td>
<td>for ($i = 1$ to $\Delta x$) {</td>
</tr>
<tr>
<td>set $(x, y)$</td>
<td>set $(x, y)$</td>
</tr>
<tr>
<td>while ($e &gt; 0$) {</td>
<td>while ($e' &gt; 0$) {</td>
</tr>
</tbody>
</table>
<pre><code>| $y = y + 1$         | $y = y + 1$ |
| $e = e - 1$         | $e' = e' - 2 \times \Delta x$ |
</code></pre>
<p>| }                    | } |
| $x = x + 1$          | $x = x + 1$ |
| $e = e + m$          | $e' = e' + 2 \times \Delta y$ |
|}                      |} |</p>

**Polygon Scan Conversion**

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

[Diagram of polygon scan conversion]
Scan line coordinate system

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

Parity scan conversion
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.

<table>
<thead>
<tr>
<th>x</th>
<th>Δx</th>
<th>Δy</th>
</tr>
</thead>
</table>

x : x coordinate of the first intersection
Δx : the increment in X from scan line to scan line
Δy : 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.

<table>
<thead>
<tr>
<th>x</th>
<th>Δx</th>
<th>Δy</th>
</tr>
</thead>
</table>

x : x coordinate of the current intersection
Δx : the X increment of this edge
Δy : the remaining number of scan lines that will intersect this edge

```c
for (each polygon edge) {
    – determine the first (highest) scan line intersected by the edge.
    – place the edge in 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 (Δy), and the edge’s X increment (Δx).
}
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) {
        Δy = Δy - 1
        if (Δy < 0) drop the edge from the active_edge_list
        x = x + Δx
    }
}
Seed filling: connectivity

- At least one pixel inside the polygon
- The algorithm fills the interior region of a polygon defined by the boundary pixels
- An 8-connected algorithm only fill 4-connected polygons.
- An 4-connected algorithm fill 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
  }
}