✠ Unit length vector: $\hat{v} = \frac{v}{|v|} = \frac{v}{||v||_2}$

✠ Dot product:

$$u.v = \langle u, v \rangle = |u||v| \cos\theta$$

results in a scalar. Equal to 0 when orthogonal; equal to 1 when parallel and unit vectors.

✠ The **cross-product** is computed as:

$$\begin{aligned} a \times b &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_i & a_j & a_k \\ b_i & b_j & b_k \end{vmatrix} \\ &= \begin{vmatrix} a_j & a_k \\ b_j & b_k \end{vmatrix} \mathbf{i} - \begin{vmatrix} a_i & a_k \\ b_i & b_k \end{vmatrix} \mathbf{j} + \begin{vmatrix} a_i & a_j \\ b_i & b_j \end{vmatrix} \mathbf{k} \end{aligned}$$

✠ Parametric equation of a ray: $P(t) = P_0 + t(P_1 - P_0)$

✠ Points to Parametric equation: $P(t) = (x - x_1)(y_2 - y_1) - (y - y_1)(x_2 - x_1)$

✠ Equation of sphere: $(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2$

✠ Rotation followed by translation:

$$R.T = \begin{bmatrix} R_1 & R_2 & R_3 & 0 \\ R_4 & R_5 & R_6 & 0 \\ R_7 & R_8 & R_9 & 0 \\ T_1 & T_2 & T_3 & 1 \end{bmatrix}$$

✠ Rotation around the 3 axis:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_x(\theta) = \begin{bmatrix} 1 & 0 & & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

✠ **Specular Reflection** is where the direction incoming light (the incident ray), and the direction of outgoing light reflected (the reflected ray) make the same angle with respect to the surface normal.

✠ **Diffuse reflection** is the reflection of light from an uneven or granular surface such that an incident ray is seemingly reflected at a number of angles.

✠ Parametric equation from COP to the center of a pixel:

$$\begin{aligned} P(t) &= (x(t), y(t), z(t)) \\ &= (t[x_{\min} + f_{\text{width}}(i + 0.5)], \;\; t[y_{\min} + f_{\text{height}}(j + 0.5)], \;\; d - td) \end{aligned}$$

$$f_{\text{width}} = (x_{\max} - x_{\min})/M$$

✠ **Ray casting** is the use of ray-surface intersection tests. If we intersect a ray with a sphere at the origin:

$$\begin{aligned} x^2 + y^2 + z^2 &= r^2 \\ (tX)^2 + (tB)^2 + d^2 - 2td^2 + t^2d^2 &= r^2 \\ t^2(X^2 + B^2 + d^2) + 2t(-d^2) + (d^2 - r^2) &= 0 \\ At^2 + 2Bt + C &= 0 \end{aligned}$$

If the discriminant gives positive real roots, the line intersects with the sphere. The smallest value of t would give the distance to the sphere.

✠ **Ambient Light** is constant throughout scene and cheap approximation to global illumination.

✠ **Diffuse reflector** scatters light (equally) in all directions. Incoming direction of light is still important because it decides the reflected intensity equals $\cos\theta = n.l$, where n is the normal of the surface and l is the direction of incident light. This is called **Lambert's law**.

✠ Local illumination equation:

$$I_r = k_a I_a + \sum_{j=1}^{M} I_{i,j} \left(k_d(n.l_j) + k_s(n.h_j)^m\right) \tag{1}$$

I_r is intensity radiating from object.
n is the normal to the surface of the object.
I_a is ambient light, and k_a is the proportion of ambient light reflected.
$I_{i,j}$ is the incoming intensity of light j.
l_j is the vector (direction) of light j. k_d is the proportion of diffuse light reflected (diffuse reflection coefficient).
h_j is the normalized direction vector for specular reflection from light j. k_s is the proportion of

specular light reflected. m, the shininess, is the power of the light - more m: smaller hightlight; low m: highlight is blurred. This is the **Phong Color Model** used for modelling glossy surfaces.

Remember this equation is run for each channel separately to give $I_{r,\text{red}}, I_{g,\text{green}}, I_{b,\text{blue}}$. This is because the intensities I and reflectance parameters k are wavelength dependent quantities.

✠ Using **Shadow feelers** we can detect if an object is in shadow from a particular light. This is done by seeing if l_j intersects any other objects in the scene. This introduces the term S_j in the local illumination equation (Equation 1).

✠ Recursive **ray tracing** is just done by ray-casting secondary rays recursively. Remember there needs to be a termination depth. *Ray tracing* simulates specular-specular reflection.

✠ Figure 1 shows R which is the reflected ray shot to recursively ray trace.

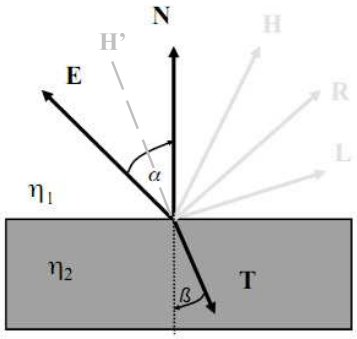

Figure 1: Reflection model

We need to compute an equation for R with the constraints $\alpha = \beta$; and all vectors are normalized:

$$
\begin{aligned}
aR + bE = N \;\Rightarrow\; aR + aE &= N && \text{since the contribution will be the same} \\
1.\sin(\alpha) &= a\sin(\pi - 2\alpha) && \text{from the sine law} \\
\sin(\alpha) &= a\sin(\pi - 2\alpha) \\
\sin(\alpha) &= a\sin(2\alpha) \\
\sin(\alpha) &= 2a\sin(\alpha)\cos(\alpha) \\
\therefore R &= -E + 2cos(\alpha)N \\
R &= -E + 2(N.E)N
\end{aligned}
$$

Now the recursive ray tracing equation is:

$$I_r = I_{\text{local}} + k_r I_{r'}$$

where I_{local} was computed in Equation 1, $I_{r'}$ is computed recursively by shooting the r' recursive ray (following direction R), and k_r is the coefficient of reflectance (usually set equal to k_s).

✠ **Snell's law** describes the interaction of rays with a transmissive surface (transparent).

$$\frac{\sin\alpha}{\sin\beta} = \frac{\eta_2}{\eta_1} = \eta_{21}$$

α is angle with the surface from the viewing point; η_1 is the index of refraction for the surface on which the viewing ray is incident. We have this equation for the transmissive ray:

$$T = -\eta_{12}E + N(\eta_{12}.\cos\alpha - \sqrt{1+\eta_{12}^2.(\cos^2\alpha - 1)}$$

Negative root here would mean total internal reflection. Now the transmissive ray is shot recursively just like the reflected ray:

$$I_r = I_{\text{local}} + k_r I_{r'} + k_t I_{t'}$$

✠ Transmissive surfaces can be illuminated from behind in the direction H' as shown in Figure 1. This would give the specular component for transmissive surfaces $(k_t(h'.n)^m)$:

$$H' = \frac{E - \eta_{21}L}{\eta_{21} - 1}$$

where L is the light coming through the surface.

✠ **General Camera** coordinates system: **VRP** - where the camera is
VPN - where the camera points
VUV - which way is up for camera

✠ **UVN** coordinates system: **VRP** - origin of VC (view-coordinates)
VPN - Z-axis (N-axis) of VC system
VUV - Y-axis (V-axis) of VC system
X-axis (U-axis) for left-handed VC system

✠ How to map points from World coordinate (WC) to View coordinate (VC) system:

1. $n = \frac{VPN}{|VPN|}$
2. $u = \frac{n \times VUV}{|n \times VUV|}$

3. $v = u \times n$
4. This forms the rotation matrix: $\begin{bmatrix} u & v & n \end{bmatrix}$
5. Hence the complete mapping matrix that takes you from WC to VC (taking objects to VC):

$$M = \begin{bmatrix} u_1 & v_1 & n_1 & 0 \\ u_2 & v_2 & n_2 & 0 \\ u_2 & v_2 & n_2 & 0 \\ -\langle q, u \rangle & -\langle q, v \rangle & -\langle q, n \rangle & 1 \end{bmatrix}$$

where q is the VRP point (bringing this point to the origin forms this equation). Also the matrix that takes you from VC to WC (taking rays to WC):

$$M^{-1} = \begin{bmatrix} R^T & 0 \\ q & 1 \end{bmatrix}$$

✠ A planar polygon (face) is defined as:

$$[p_0, p_1, \ldots, p_{n-1}, p_n], \qquad p_0 \equiv p_n$$

Equation of plane:

$$l(x, y, z) = ax + by + cz - d = 0$$

remember $n = (a, b, c)$ is the normal, and $d = n.p_0$.

1. $l(x, y, z) = 0$ - point on plane
2. $l(x, y, z) > 0$ - point in positive half-space (i.e. in the direction of the normal)
3. $l(x, y, z) < 0$ - point in negative half-space (i.e. opp. to the direction of the normal)
4. direction of normal decided by the way points that were selected

✠ **Polyhedra** is a collection of connected polygons (given that each vertex joins 3 or more edges): $\#Vertics - \#Edges + \#Faces = 2$

✠ Representations of Polyhedra:

1. Exhaustive faces list each having its point sets
2. **Indexed Face Set**: Has Vertex array, and a Face array having pointers (indexes) to the vertex array.
3. **Winged edge data structure** (WE): Vertex list; Edge list (having vertex pairs); Face list (having edge lists). Each vertex (ring) has links to next and previous vertices and to the edge ring; each face (ring) contains links to the next and previous faces and to the edge ring; each

edge (ring) has links to the next and previous edge and also links to the neighboring vertices and faces as shown in Figure 2.

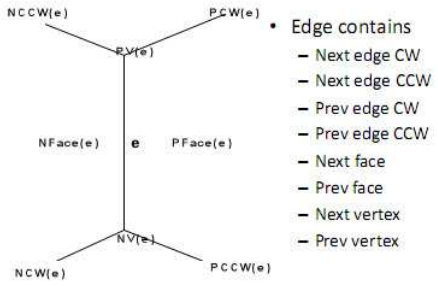

Figure 2: Winged Edge data structure

✠ We can build a *WE* by: (1) traversing each face CCW, building edges along the way; (2) link the next and previous vertices and faces; (3) link previous CCW edge (other edges will be filled in while traversing other faces).

✠ The **Barycentric Combination** is just a weighted sum of points, where the sum of weights is 1 ($\sum_{i=1}^{n} a_i = 1$):

$$p = \sum_{i=1}^{n} a_i p_i$$

Barycentric combination for a line is: $p(t) = (1-t)p_t + tp_2$ *Note that for a point p is inside a convex polygon iff* $0 \leq a_i, \forall i$

✠ *Barycentric coordinates* are preserved under affine transformation.

✠ **Intersection of rays with polygons**. First step is to find if the ray ($r(t) = q_0 + td_r$, d_r is the direction vector) intersects the plane of the polygon: $n.p_0 - d = 0$. This would be the case if $n.d_r \neq 0$. Now the intersection point is given by:

$$t_{\text{int}} = \frac{d - n.q_0}{n.d_r}$$
$$p_{\text{int}} = q_0 + t_{\text{int}} d_r$$

Methods to find if point inside polygon or not:

1. **Winding number test**: Point is inside polygon if sum of angles subtended from all adjacent pairs of vetrices is $\pm 2\pi$ and 0 if outside.

2. **Infinite Ray test**: Shoot ray and count +1 for crossing edge CCW, and -1 for crossing edge in CW. If result is 0 then point is outside.
3. **Half-space test**: Point is inside if it is in the negative half space of all the edges on the polygon.
4. **Triangle inside/outside**: Given the 3 points A, B, C on the plane, point p is inside if:

$$\begin{bmatrix} A & B & C \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} p \\ 1 \end{bmatrix}$$

and $a_i \geq 0, \quad \forall i$

We can also project the problem in 2D space by throwing away the axis whose coefficient in the plane equation is the maximum (i.e. we are trying to minimize the angle between the plane normal and the normal of the selected 2D axis)

✠ **Scene Graphs** are formed using *directed acyclic graphs* where the links are tranformations - and the root of the graph corresponds to the *world coordinates*.

✠ Each object in the *scene graph* is modeled in it own local coordinate system.

✠ An object's local transformation (LT) maps it from its own local coordinates to its parents local coordinates. The concatenation of all LT matrices above a node is called the CTM (current transformation matrix). So, lets say, the CTM for the upper arm would be $T_U T_S T_B$.

If an object has two parents it will occur at two locations.

✠ Spherical coordinates are given as:

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \tan\phi &= y/x \\ \cos\theta &= z/r \end{aligned}$$

✠ Rotation around an arbitrary axis $[p_0, p_1]$:

1. Translate p_0 to origin giving p_1^*.
2. Get spherical coordinates of p_1^*: (r, ϕ, θ).

3. Rotate about Z by $-\phi$.
4. Rotate about Y by $-\theta$.
5. Perform the actual rotation about Z.
6. Reverse the first 4 steps.

✠ Ray tracing is speeded-up by tracing rays to vertices of the polygons and filling the space in between (**Rasterization**). Instead of tracing rays:

1. Project all polygons (vertices) onto the viewplane.
2. Transform projected 2D vertices into display coordinates.

For this we need to define:

1. **View plane** is a plane orthogonal to VPN onto which the scene is projected.
2. **View plane distance** is the distance from the VRP to the view plane (along the N-axis).
3. **Type of projection** - perspective: all rays converge to COP; parallel (orthographic): parallel rays going in direction of projection, DOP remain parallel (where the COP is at $-\infty$). Note that COP is defined by the offset from VRP.
4. **View plane window** is the window on the view plane parallel to U and V axes.
5. **Front and back clipping planes**

✠ **Canonical Frame** is the fundamental arrangement to which all projection is done. It has these properties:

1. COP/DOP - (0,0,-1).
2. View plane parallel to UV plane.
3. View plane bounded by ± 1.
4. View plane distance $= 1$.

✠ How to get to canonical perspective:

1. Move the view plane to UV plane (such that $n = 0$):
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -d & 1 \end{bmatrix}$$
2. Translate COP to N-axis:
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -c_x & -c_y & 0 & 1 \end{bmatrix}$$
3. Change View volume to a regular pyramid:
$$\begin{bmatrix} 2D/dx & 0 & 0 & 0 \\ 0 & 2D/dy & 0 & 0 \\ -px/dx & -py/dy & 1 & 0 \\ -(px/dx)D & -(py/dy)D & 0 & 1 \end{bmatrix}$$
$D = d - c_z, \quad dx = U_2 - U_1, \quad dy = V_2 - V_1, \quad px = U_1 + U_2 - 2c_x, \quad py = V_1 + V_2 - 2c_y$, where U_1, U_2 and V_1, V_2 are viewplane boundaries.
4. Scaling:
$$\begin{bmatrix} 1/D & 0 & 0 & 0 \\ 0 & 1/D & 0 & 0 \\ 0 & 0 & 1/D & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiply all four matrices to get the Q matrix. Now we can get the transformation matrix:
$$T = \underbrace{M}_{\text{WC to VC}} \quad \underbrace{Q}_{\text{VC to canonical VC}}$$

✠ The final step is to transform into canonical parallel space (which incorporates perspective projection):
$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
which gives us the point $\left(\frac{x}{z+1}, \frac{y}{z+1}, \frac{z}{z+1}\right)$. From now the projection onto the X-Y plane is obtained by simply ignoring the Z component. If you also want to incorporate the front and back clipping plane:
$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{D_{\max}+1}{D_{\max}-D_{\min}} & 1 \\ 0 & 0 & -\frac{D_{\min}(D_{\max}+1)}{D_{\max}-D_{\min}} & 1 \end{bmatrix}$$

✠ **Clip** polygons for numerical stability; avoid drawing stuff behind the eye; avoid wasting time on object outside image boundary. In short clipping is the process of finding the exact part of the polygon inside the viewing volume.

✠ Analytical clipping:

1. **Sutherland-Hodgman Algo.**: Clipping from each boundary edge in turn.
   (1) Iterate over all polygon edges $((P_i, P_{i+1}))$ for each boundary,
   (2) if entering - output $(\mathbf{p}, P_{i+1})$,
   (3) if exiting - output $(\mathbf{p})$,
   (4) if totally outside then output nothing,
   (5) if totally inside output (P_{i+1}).
   In all cases **p** is the point of intersection with the edge. This can be extended to 3D by using 6 planes rather than 4 lines (intersection computed with edge to a plane). In 3D normals of planes will point to the interior, and we need to reject (cull) a point when $N.p < d$.
2. **Weiler-Atherton Algo.**: Used when we have convex polygons (in which case Sutherland-Hodgman will return degenerate edges). It produces separate polygons for each visible section:
   (1) Compute intersection points with all edges. Classify them as "leaving" or "entering".
   (2) Create CW loop of polygon vertices; also create a separate CW loop of boundary vertices.
   (3) Incorporate the intersection points with connections to all lists.
   (4) Start with entering vertex: if you encounter a leaving vertex, shift to clip boundaries loop; if you encounter an entering vertex, shift to polygon loop; ends when you arrive to the entering vertex you started from.
   (5) Repeat till no entering vertices left.

✠ Clipping in canonical parallel space is simple because we just need to check for inequalities with a single axis for each plane (remember our viewing volume is a cube now).

✠ **Scan-line coherency** says if an edge is not parallel to the scan line, the next x coordinate on the scan line will be:

$$x_i = x_{i-1} + \frac{1}{a}$$

where a is the gradient of the line.

✠ **Active edge table** (AET) for rasterization have these entries:

$$(y_2, x_1, dx)$$

where $dx = (x_2 - x_1)/(y_2 - y_1)$

1. Each row in the edge table will belong to a particular scan line. Each row is sorted by the current x value.
2. Delete edges whose y_2 is equal to the current scan line index.
3. Iterate over each scan line and increment the x value of each edge still in the table with dx.
4. Appends all new edges to the edge table whose y_1 equals the current scan line index.
5. Output spans as pair of consecutive x values still in the list.

✠ Shading in Rasterization:

1. **Gouraud Shading** (comes from Lambert's law) uses the above method extends the edge table by:
$$(y_2, x_1, dx, r_1, dr, g_1, dg, b_1, db)$$
Now increment the RGB values when moving over scan lines and linearly interpolate as we move across a scan line. Here $dr = (r_2 - r_1)/(y_2 - y_1)$. The initial values for RGB are computed on the edge vertices by analytically computing the normals. To use this method, polygons must be sufficiently small with respect to the distance from the light.
2. **Phong Shading** (comes from Phong color model) is used to get specular highlights back into the polygons. Rather than interpolating RGB values, it interpolates normals across scan line and then uses the Equation 1 to get the actual color.

✠ **Half-space based triangle rasterization**:

1. Compute projection of triangle.
2. Make bounding box around the triangle (only consider area: $x_{\min}, x_{\max}, y_{\min}, y_{\max}$).
3. Compute the line equations for each pixel. If for all $L_i(x, y) > 0$ (positive half-space test) then point inside.
4. If pixel inside, paint it with triangle color.
5. You can also sub-divide the bounding box.

✠ Depth Rasterization:

1. Directly compute z value from plane equation. This is done for each pixel.
2. Pre-order polygon (as done in visibility determination).
3. Use **Z-buffer**. Initialize it to maximum value - for each pixel in each polygon see if the depth is less than the one stored in the depth buffer. If so set the depth for that Z-buffer pixel, and set the colour value in the frame buffer.

4. Scan line depth buffering can be done using AET (when polygons don't intersect):

$$(y_2, x_1, dx, z_1, dz)$$

where $dz = (z_2 - z_1)/(y_2 - y_1)$. Each polygon is processed in a separate AET and the (1D) Z-buffer is used over each scan line. Has problems of aliasing on depth. Note that all Z value interpolation need to be done as follows to take into account perspective projection:

$$Z_{\text{int}} = \frac{1}{\frac{1}{Z_1} + s\left(\frac{1}{Z_2} - \frac{1}{Z_1}\right)}$$

✠ **Texture Mapping** modifies the diffuse component of surface. A texture is defined as a 2D array of **texels**.

✠ **Minification**: one pixel covers more than one texel.
Magnification: one texel covers more than one pixel.

✠ **Forward mapping** gives points on the texture map onto a polygon. The inverse mapping gives texture vertex for each polygon vertex. Can use barycentric coordinate to paint textures. Can also put in AET to do bilinear interpolation of texture coordinates. To correct for perspective, we interpolate points by converting them to:

$$(u', v', q) = (u/z_1, v/z_1, 1/z_1)$$

once interpolated, divide (u', v') by the interpolated $1/z$, to get (u, v)

✠ To solve under-sampling and over-sampling we perform filtering when pasting in textures. We use nearest neighbor approach; bilinear filtering (weighted average in 4 neighborhood). We can also use **Mip-mapping** which creates a pyramid of textures. We now interpolate textures using two adjacent layers in the mipmap (choice of layer where du and dv for dx and dy is closest to 1).

✠ **Bump mapping** alters the surface normal to give effects of texture. This is done by computing normal from the partial derivatives of the texture.

✠ **Displacement mapping** is used to actually move surface point. This is done before any visibility calculations (to induce shadows).

✠ **Environment maps** is used to simulate reflection by using the direction of the reflected ray to index a spherical texture map at infinity.

✠ **OpenGL** is an API which defines a mechanism to specify images in the frame buffer. It is run

by tracking values in state tables. The pipeline has (1) Vertex assemly, (2) vertex operations, (3) Primitive assemly, (4) Primitive operations, (5) Rasterization, (6) Fragment operations (fragment lighting, texture mapping, etc.). Lastly everything is pushed to the frame buffer.

✠ **Vertex shaders** are used for custom lighting, transforms, displament models etc.. On the other hand **Pixel shaders** are used for per-pixel lighting, etc.

✠ **Radiosity** is used to model specular-specular light interactions. This is done using the **Radiance Equation** (essentially all lighting solutions in CG are approximations to the *Radiance Equation*)

✠ **Flux** or **Radiant Energy**, denoted by Φ, is the rate of energy flowing through a surface per unit time. (watts - W).

✠ The total light input to a volume should be equal to the total output or absorbed. Input:

1. **Emission** - emitted from within the volume.
2. **Inscattering** - light flows in the volume from outside.

Ouput:

1. **Streaming** - is light flowing through without any interaction with the volume.
2. **Outscattering** - light interacting with volume and then being reflected out.
3. **Absorption** - light being absorbed in the volume

✠ $\Phi(p, \omega)$ denotes the flux at $p \in V$ (in volume), in the direction ω. It can be written in terms of total light input = output. Complete knowledge of this would give the solution to the illumination problem. We make a few simplifying assumptions: (1) wavelength independence, (2) time invariance, (3) light transport in vacuum (non-participating medium).

✠ The **Radiance equation** computes the radiance from a point on a surface in a direction ω.

$$L(p,\omega) = \underbrace{L_e(p,\omega)}_{\text{emitted radiance}} + \underbrace{\int \underbrace{f(p,\omega_i,\omega)}_{\text{BRDF for scaling}} \underbrace{L(p^*,-\omega_i)}_{\text{radiance arriving from } p^*} \cos\theta_i d\omega_i}_{\text{reflected radiance from all other surfaces}} \tag{2}$$

where $L(p, \omega)$ is the radiance at point p on the surface along direction ω. The equation essentially models view-independent global illumination.

✠ **Radiance** L is the flux that passes through or is emitted from a particular area, and falls within a given solid angle in a specified direction. Measured in watts per steradian per square metre ($W.sr^{-1}.m^{-2}$). Remember radiance is constant along a ray!

✠ Solid angle $d\omega_b = \frac{dB \cos\theta_B}{r^2}$, where dB is the area of a patch, and r is the distance between the patches.

✠ **Radiosity** is flux per unit area that *radiates* from a surface: $B = d\Phi/dA$. **Irradiance** on the other hand is the flux per unit area that arrives at a surface (given by E).

✠ The **bidirectional reflectance distribution function (BRDF)** relates radiance to irradiance (units 1/sr):

$$f(p, \omega_i, \omega_r) = \frac{dL}{dE}$$

It essentially defines the look of the surface. It is usually computed by relying on the ideal types: (1) perfectly specular, (2) perfectly diffuse (ρ/π), (3) glossy reflection. Properties of BRDF include:

1. $f(p, \omega_i, \omega_r) \geq 0$
2. Energy conservation: $\int_\Omega f(\omega_i, \omega_r) d\mu(\omega_r) \leq 1 \quad \forall \omega_i$
3. $f(\omega_i, \omega_r) = f(\omega_r, \omega_i)$

✠ **Radiant intensity** is a measure of the intensity of electromagnetic radiation. It is defined as power per unit solid angle - watts per steradian ($W.sr^{-1}$).

✠ Ray tracing is an image space algo., radiosity is an object space algo. Radiosity is a view independent solution and remains constant as long as light sources don't move.

✠ Estimated Radiosity quantity stored with the surface - and works only for energy exchange in diffuse surfaces. We change Equation 2 to:

$$L(p, \omega) = L_e(p, \omega) + \int f(p, \omega_i, \omega) L(p^*, -\omega_i) \overbrace{G(p, p^*)}^{\text{geometric term for dist./orient.}} \underbrace{V(p, p^*)}_{\text{binary visibility term}} dA(p^*) \quad (3)$$

Since we are thinking of perfect diffuse surfaces, we can ignore the BRDF:

$$B_p = E_p + \rho_p \int B_{p^*} \underbrace{G'(p, p^*) V(p, p^*)}_{\text{form factor}} dA(p^*) \quad (4)$$

where G' includes a $1/\pi$ factor, ρ_p is the reflectance coefficient. Remember the relationship between radiance and radiosity is:

$$B(p) = \pi L_{\text{diffuse}}(p)$$

We further convert this problem to a discrete representation:

$$B_i = E_i + \rho_i \sum_{j=1}^{n} F_{ij} B_j$$

shows that the radiosity from path i is equal to radiosity directly emitted from i, plus the proportion of irradiance that it reflects out again. The incoming energy is the sum of the irradiance received from each patch j but modified by the proportion of energy from such a patch that arrives at i (determined by form factor). The form factor F_{ij} tells the fraction of energy leaving patch i which arrives at patch j. These form factors follow the law of conservation: $\sum_{j=1}^{n} F_{ij} = 1$ and reciprocity: $F_{ij}A_i = F_{ji}A_j$.

1. Divide scene into small patches (mesh). Each patch is associated with total energy leaving the surface (Radiosity).
2. Use gauss-seidel relaxation to compute radiosity patch by patch:

$$\begin{bmatrix} B_1^{(k+1)} \\ B_2^{(k+1)} \\ \vdots \\ B_n^{(k+1)} \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix} + \begin{bmatrix} \rho_1 F_{11} & \rho_1 F_{12} & \dots & \rho_1 F_{1n} \\ \rho_2 F_{21} & \rho_2 F_{22} & \dots & \rho_1 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_n F_{n1} & \rho_n F_{n2} & \dots & \rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1^{(k)} \\ B_2^{(k)} \\ \vdots \\ B_n^{(k)} \end{bmatrix}$$

3. How to compute the form factor:

$$F_{dA_i dA_j} = \frac{\cos\alpha_i \cos\alpha_j}{\pi r^2} V(dA_i, dA_j)$$
$$F_{ij} \approx \int_{A_j} \frac{\cos\alpha_i \cos\alpha_j}{\pi r^2} dA_j$$

   Here the **Nusselt Analogy** says that the form factor is equal to the proportion of projected area on circular base. The **Hemi-cube Approximation** uses a lookup table on a hemi-cube to get pre-computed delta form-factors. By projecting all patches on this hemi-cube, the sum of all delta form factors gives the final form factor:

$$\Delta F_q = \frac{1}{\pi(x^2 + y^2 + 1)^2}$$

4. Ray-casting can be used for radiosity computation by taking a sample of points on the source patch, and trace and sum contributions to the receiver patch.

Since the full solution can be slow we follow the **Shooting** method:

1. Set $B_i = \Delta B_i = E_i$ where i is an emittor patch.
2. Sort $A_i \Delta B_i$ into descending order.
3. For each patch i compute the form factor F_{ij}
4. For each patch $j \neq i$, compute:

$$(B_j \text{ due to } B_i) = \rho_j B_i F_{ij} \frac{A_i}{A_j}$$

5. Add the component computed to ΔB_j.
6. Also add to B_j.
7. Set $\Delta B_i = 0$

unshot energy average $U = \frac{\sum_{i=0}^{n} \Delta B_i A_i}{\sum_{i=0}^{n} A_i}$
average reflectance $\rho_{\text{avg}} = \frac{\sum_{i=0}^{n} \Delta A_i \rho_i}{\sum_{i=0}^{n} A_i}$
infinite interreflections $R = \frac{1}{1-\rho_{\text{avg}}}$
ambient $\rightarrow R.U$
display $B_i = B_i + \rho_i$Ambient

✠ **Umbra** the part of the shadow that sees no light.
Penumbra the part of the shadow that receives some light.

✠ **Soft-shadows** (only possible with area light sources) are produced by shooting multiple shadow rays.

✠ **Fake-shadows** produced by projecting the object onto the ground. Gives no inter-object shadows.

✠ **Shadow maps**: (1) Use light source as the view point and compute the Z-buffer - giving the shadow Z-buffer; (2) from the view point, before we scan-convert an object, we transform its point to the light-space and only paint the pixel if the depth less than or equal to the one given in the Z-buffer.

Shadow map filtering can be done to give smooth shadows by doing an averaging (percentage closer filter) across a set of neighboring pixels.

✠ **Shadow Volume** is the volume of space occluded by an object infront of the light. Each silhouette edge of the polygon object gives a shadow plane. By shooting a ray to our point of interest we just need to count the front-facing and back-facing shadow planes the ray crosses. If their difference is 0 then the point is not in shadow. Done in two steps:

1. Find the plane equations for all shadow planes.
2. At run-time determine the shadow plane count per pixel. Here the *stencil buffer* can be used for counting the number of SV planes crossed (stencil buffer is set iff Z-buffer test passed):
   (a) render scene into RGB and Z-buffer.
   (b) render shadow polygons with stencil-buffer (do the increment and decrement count).
   (c) Re-render the scene with lighting off by only rendering pixels where the stencil-buffer is non-zero.

✠ Sampling methods for *Soft shadows*:

1. *Soft shadows* can be rendered by dividing the area light source into multiple point light sources (randomly placed) and rendering hard shadows with them.
2. *Soft shadows* can be done by illumination maps where shadows are precomputed - hence in this method we cannot changee the lighting condition.

Analytical methods for *Soft shadows*:

1. **Extremal Shadow boundaries** compute the planes for penumbra and umbra. If we write these planes into object space, using scan-conversion we can find the contribution of light at each point.
2. **Discontinuity meshing** subdivides the plane into discontinuity points and computes illumination intensity at these points. Use quadratice approximation in areas between the points. Places where EV and EEE surfaces (from aspect graphs) intersect the polygon define critical points where shadows visually change.

✠ **Visibility Culling** needs to be used for speed-up and accurate rendering. One solution is simply to use the Z-buffer.

✠ **Back-face culling** is used to decide which polygons not to render. If COP (c_x, c_y, c_z) then if $l(c_x, c_y, c_z) > 0$ COP is in front.

In projection space, the polygon plane should have a negative z-component n to be visible. For this we need to compute the plane equation in projection space:

$$q.\left(T^{-1}.\begin{bmatrix} a \\ b \\ c \\ -d \end{bmatrix}\right) = q.m = 0$$

Just look if the z-component of m is negative.

✠ How to order the polygons in order of their depth

1. **Z-sort** Sort back-to-front according to the z-axis values at mid-points.
2. **Depth-sort (Newell et al.)** It says that P can be rendered before Q if any of these conditions are satisfied:
   (a) Z-extent of Q is wholly infront of P.
   (b) Y-extent or X-extent of Q does not overlap with P.
   (c) All points on P lie on the opposite side of Q than the COP.
   (d) All points on Q lie on the same side of P as the COP.
   (e) If the projections on the XY plane don't overlap (run 2D polygon overlap tests)
3. **Schumacker** uses an object space method. Works on the idea if you can define a plane between two objects, then the polygon lying in the same half-space as the viewpoint will have higher priority. Builds a tree by dividing the whole space into partitioning planes.
4. **Binary Space Partitioning Trees (BSP)**: Build tree independent of viewpoint:
   (a) Choose arbitrary polygon and make it root of the tree (partition the space where the direction of plane normal decides the front side).
   (b) Perform this recursively (left side of tree is front).
   (c) Coplanar polygons remain at the same node.
   (d) Split planes if partition falls across them.

   This can also be done iteratively, by inserting polygons into the tree. Traverse tree from a given view point to get visibility ordering (back-to-front):
   (a) If viewpoint in front of the plane then traverse back first, render the current node, and then front.
   (b) If viewpoint in back of the plane then traverse front first, render the current node, and then back.

   BSPs can also be used for hierarchical ordering of spaces, representation of polygons or polyhedra.