Normal Vector

For a triangle ABC, $N = (B - A) \times (C - A)$ (Select 2 edges counter-clockwise, and normalize)

Vector Distance

For two points $P = (x_1, y_1, z_1), Q = (x_2, y_2, z_2)$,

$$\|\overrightarrow{PQ}\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}.$$

For $\mathbf{u}, \mathbf{v} \in \mathbb{R}^3$, $\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z = \|\mathbf{u}\| \|\mathbf{v}\| \cos\theta$.
Use: angle between vectors, projection, parallelism.

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}$$

gives a vector orthogonal to both $\mathbf{u}, \mathbf{v}$. **Use:** normal computation, orientation.

Common Transformation Matrices

Translation: `glTranslatef`(t_x, t_y, t_z); Scaling by (s_x, s_y, s_z):

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about x, y, z axis: `glRotatef`$(\theta, x?, y?, z?)$

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

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

Perspective (simple form):$\rightsquigarrow$ Initialize $\to$ register callback functions $\to$ enter glutMainLoop() $\to$ wait for events $\to$ call corresponding callbacks $\to$ program exits when closed.

Primitive Drawing

- **glBegin(mode) ... glEnd()** Define a sequence of vertices for drawing. `mode` = GL_POINTS, GL_LINES, GL_TRIANGLES, GL_QUADS, etc. Note: only draw convex, planar polygons.

```
glBegin(GL_TRIANGLES);
    glColor3f(1,0,0); glVertex3f(0,0,0);
    glColor3f(0,1,0); glVertex3f(1,0,0);
    glColor3f(0,0,1); glVertex3f(0,1,0);
glEnd();
```

- **glColor3f(r,g,b)** Set current drawing color (RGB, 0–1).
- **glVertex2f(x,y), glVertex3f(x,y,z)** Specify a vertex in 2D or 3D.

State Management

- **glLoadIdentity()** Reset current matrix to identity (no transform).
- **glClear(mask)** Clear buffers. Common: `GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT`.
- **glFlush()** Force execution of all issued OpenGL commands (mainly for single-buffered mode).

Window / Display Setup

- **glutInit(&argc, argv)** Initialize GLUT.
- **glutInitDisplayMode(flags)** Select display mode, e.g. `GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH`.
- **glutDisplayFunc(func)** Register display callback (called whenever window redraws).
- **glutMainLoop()** Enter GLUT event loop (program stays here).

Shading

- **glShadeModel(mode)** `GL_FLAT` = uniform color per primitive, `GL_SMOOTH` = interpolated colors.

Matrices & Viewing

- **glMatrixMode(mode)** Select current matrix stack: `GL_MODELVIEW` or `GL_PROJECTION`.
- **GL_MODELVIEW** Holds modeling + viewing transforms.
- **GL_PROJECTION** Holds projection (camera lens) transforms.
- **gluOrtho2D(left,right,bottom,top)** 2D orthographic projection.
- **glOrtho(l,r,b,t,n,f)** 3D orthographic projection volume.
- **glFrustum(l,r,b,t,n,f)** Perspective frustum volume.
- **gluPerspective(fov,aspect,n,f)** Perspective projection with vertical field-of-view.
- **gluLookAt(eye,center,up)** Set camera at `eye`, looking at `center`, with `up` direction.

Depth & Culling

- **glEnable(cap)** Turn on a feature. E.g. `GL_DEPTH_TEST`, `GL_CULL_FACE`.
- **glDepthMask(flag)** Enable/disable writing to depth buffer. Useful for transparency.
- **glCullFace(mode)** Choose which faces to cull: `GL_BACK`, `GL_FRONT`, `GL_FRONT_AND_BACK`.
- **GL_DEPTH_TEST** Feature that discards fragments hidden behind others in depth.

1 Model–View and 3D Representation (Algorithms & Data Structures)

Model–View: Frames, Matrices, and Hierarchies

- **Model–View matrix** M: concatenates object–to–world (model) and world–to–camera (view) transforms. Typical composition (right-multiplied): $M \leftarrow M\,T(\mathbf{t})\,R(\theta, \hat{\mathbf{a}})\,S(s_x, s_y, s_z)$. *Order matters* (non-commutative).
- **Matrix stack**: use `glPushMatrix`/`glPopMatrix` to delimit local frames. Transform state between pushes affects all descendants; popping restores the parent frame.
- **Hierarchical transforms**: model parts as a tree (e.g., torso $\to$ arm $\to$ forearm $\to$ hand). Each node stores a local transform; the global pose is the product along the path from root to the node. Animations vary local parameters over time.
- **Orientation/winding**: triangle order encodes front face (right-hand rule). Choose CCW or CW consistently; configure the API (e.g., `glFrontFace(GL_CCW)`) and optional back-face culling.

3D Object Representations

- **Independent faces**: each face stores three vertex (x, y, z). *Pros*: simplest. *Cons*: duplicated vertices; no explicit adjacency/topology.
- **Indexed face set (triangle list)**: a *vertex array* of unique points plus a *face array* of integer triplets (indices). *Pros*: shared vertices, compact, GPU-friendly. *Cons*: still lacks explicit edge/face adjacency.
- **Adjacency-enriched meshes**: store connectivity to enable fast traversal and editing.
  - *Per-vertex/face adjacency lists*: neighbors of a vertex; neighboring faces of a face.
  - *Half-edge / winged-edge* (canonical for robust geometry): each directed edge stores pointers to its origin vertex, twin, next/prev in the face, and incident face. *Pros*: $O(1)$ local traversal; supports topology edits. *Cons*: higher memory, more bookkeeping.
- **Other representations** (for context): parametric patches, CSG, spatial subdivision (voxels), implicit surfaces.

Core Algorithms on Meshes

- **Vertex deduplication** (build indexed mesh): hash (x, y, z) (with epsilon tolerance) to map repeated coordinates to one index; emit faces as index triplets.
- **Face normals**: for triangle v_0, v_1, v_2,

$$\mathbf{n}_f = \frac{(v_1 - v_0) \times (v_2 - v_0)}{\|(v_1 - v_0) \times (v_2 - v_0)\|}.$$

- **Vertex normals** (smooth shading): area- or angle-weighted average of incident face normals, then renormalize.
- **Topology checks**: manifoldness (each edge has at most two incident faces), consistent winding, boundary detection (edges with one incident face), connected components (DFS/BFS over adjacency).
- **Rendering paths**:
  - Indexed draw calls (e.g., `glDrawElements`) for an indexed face set.
  - Depth buffering for visibility (preferred) vs. painter's algorithm (order-dependent).
  - Optional back-face culling to skip backfacing triangles.

Practical Tips

- Keep *geometry* (vertex positions) separate from *topology* (indices/connectivity); this avoids numeric drift and reduces memory.
- Use the matrix stack to isolate local edits: push, apply local $T/R/S$, draw child, pop.
- Be consistent with triangle winding across the asset pipeline; fix mismatches at import.

Hidden Surface Removal (HSR) + 3D BSP Cheatsheet

Context

After modeling, viewing, and perspective transforms, we must ensure polygons are drawn in correct visibility order to avoid "wrong drawing order." Two broad families exist: *object-precision* (control draw order) and *image-precision* (decide per-pixel overwrite).

Object-precision (orderly).

Depth sort with splitting (e.g., Weiler–Atherton) and **BSP trees** produce a back-to-front order for any viewpoint. Handles transparency but can be expensive to build or split polygons.

Image-precision (non-orderly).

Z-buffer keeps a depth value per pixel; simple and hardware-friendly, robust for interpenetrating geometry, but suffers from precision issues (Z-fighting) and limited transparency.

Binary Space Partitioning (BSP) in 3D

Goal: Preprocess a static polygon set into a tree so that, for any camera point p, an in-order traversal yields a correct back-to-front drawing order. Standard in classic FPS engines.

Definitions

Each node stores:

- A **splitting plane** Π : $\mathbf{n}\cdot\mathbf{x}+d=0$ coincident with a polygon P (node's polygon).
- Two child subspaces: *Back* ($\mathbf{n}\cdot\mathbf{x}+d<0$) and *Front* ($>0$).
- Polygons in the node's plane; polygons crossing Π are split into coplanar parts for the two sides.

Note: In practice, any polygon can be chosen to define the split; repeated recursively until leaves contain simple geometry. :contentReference[oaicite:4]index=4

Construction (static scene)

1. If set S of polygons is empty: return null.
2. Choose a polygon $A\in S$ and define its plane Π_A.
3. Partition every $Q\in S\setminus\{A\}$ against Π_A:
   - If entirely in back/front half-space, send to that child set.
   - If straddling Π_A, split Q into Q_{back} and Q_{front}.
   - If coplanar with Π_A, attach to the node (commonly to a coplanar list).
4. Recurse on back set to build `node->back`; recurse on front set for `node->front`.

This preprocessing is view-independent and reused for all camera positions. :contentReference[oaicite:5]index=5

Rendering (any viewpoint p)

For a node with plane $\Pi:\mathbf{n}\cdot\mathbf{x}+d=0$:

1. Compute $s=\mathbf{n}\cdot p+d$.
2. If $s>0$ (camera in front half-space):
   `draw(node->back)` → `draw(node->coplanar)` → `draw(node->front)`
3. If $s<0$ (camera in back half-space):
   `draw(node->front)` → `draw(node->coplanar)` → `draw(node->back)`

This is exactly a view-dependent in-order traversal that yields back-to-front order. Transparency becomes straightforward (alpha-blend as you go). :contentReference[oaicite:6]index=6 **Build BSP:**

1. Choose A as root; split B and C against Π_A. They are not straddling ($z=0$), so both go fully to either front ($z>0$) or back ($z<0$) depending on their extents. Suppose both lie in *front* ($z>0$) for illustration $\Rightarrow$ `root->back = null`, `root->front` contains $\{B,C\}$.
2. Recurse on `front`: choose B next, partition C by Π_B into C_{front} ($x>0$) and C_{back} ($x<0$).
3. Continue until sets become simple leaves.

Render for a viewpoint. Let $p=(2,2,2)$:

$$\mathbf{n}_A\cdot p+d_A=2>0\Rightarrow \text{at root, draw Back, then } A, \text{ then Front.}$$

Since `Back` is empty: draw A (the $z=0$ polygon). Descend to `Front` (node B). With p:

$$\mathbf{n}_B\cdot p+d_B=2>0\Rightarrow \text{draw Back subtree of } B, \text{ then } B, \text{ then Front subtree.}$$

Those subtrees contain the parts of C (C_{back} at $x<0$ and C_{front} at $x>0$). The emitted sequence is a correct back-to-front order for this p. If we move to p', the traversal order flips accordingly—*without rebuilding the tree.* :contentReference[oaicite:7]index=7

Pros / Cons (at a glance)

- **Pros:** Single precomputation supports *all* viewpoints; good for transparency ordering; widely used in classic game engines (static maps). :contentReference[oaicite:8]index=8
- **Cons:** Expensive splits; not ideal for dynamic/moving geometry; preprocessing time and potential growth in polygon count. :contentReference[oaicite:9]index=9

OpenGL Hooks (use with BSP or Z-buffer)

- Depth buffer: `glEnable(GL_DEPTH_TEST); glClear(GL_COLOR_BUFFER_BIT GL_DEPTH_BUFFER_BIT); glDepthMask(GL_TRUE);` (clear each frame). :contentReference[oaicite:10]index=10
- Back-face culling: `glEnable(GL_CULL_FACE); glCullFace(GL_BACK);` Use consistent winding (`glFrontFace(GL_CCW)`). Test via $\mathbf{V}\cdot\mathbf{N}>0$. :contentReference[oaicite:11]index=11

This emits a back-to-front order; swap the order to front-to-back if you prefer early-Z culling with an opaque Z-buffer pass. :contentReference[oaicite:12]index=12

2 Projection

Families of Projections

- **Orthographic (parallel)**: Projectors are parallel; depth is discarded (e.g., take (x,y)). Preserves sizes along projector direction; good for CAD/engineering drawings; less realistic.
- **Perspective**: Projectors meet at a center of projection; distant objects appear smaller (foreshortening).

Perspective Basics

- Canonical setup: camera at $(0,0,-f)$ looking along $+z$; image plane at $z=0$.
- Mapping: $x'=\frac{f}{z}\,x,\qquad y'=\frac{f}{z}\,y.$
- Homogeneous form: use a perspective projection matrix to map to clip space; after division by w you obtain normalized device coordinates (NDC).

$$P=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&\frac{1}{f}&1\end{bmatrix}\ \text{homogenous coord:}\ \begin{bmatrix}x\\y\\z\\\frac{z+f}{f}\end{bmatrix}=\begin{bmatrix}\frac{fx}{z+f}\\\frac{yf}{z+f}\\\frac{zf}{z+f}\end{bmatrix}$$

2.1 Model–View Matrix and Transform State

Where a vertex goes. Whenever you submit a vertex p (e.g., with `glVertex`), OpenGL draws it at

$$p'=M\,p, p''=PMp$$

Set Projection when the window resizes or the "lens" changes.
Set Model–View every frame (camera), and per object (model transforms).

```
void reshape(int w,int h){
  glViewport(0,0,w,h);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(40.0, (float)w/h, 1.0, 80.0);   // or glOrtho(...)

  glMatrixMode(GL_MODELVIEW);                    // back to MV for drawing
}

void display(){
  glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0,0,6,  0,0,0,  0,1,0);              // View part of MV

  glPushMatrix();                                 // Model part for object A
    glTranslatef(...); glRotatef(...); glScalef(...);
    drawA();
  glPopMatrix();

  glPushMatrix();                                 // Object B
    glTranslatef(...);
    drawB();
  glPopMatrix();
}
```

Changing the transformation matrix M.

- `glMatrixMode(GL_MODELVIEW)` — select the Model–View matrix stack. *Other choices:* `GL_PROJECTION`, `GL_TEXTURE`, `GL_COLOR`.
- `glLoadIdentity()` — set $M\leftarrow I$ (identity).
- `glTranslatef(tx,ty,tz)` — postmultiply: $M\leftarrow M\,T$.
- `glRotatef(angle,x,y,z)` — postmultiply: $M\leftarrow M\,R$.

Camera (View) Transform

- Build a camera frame with position $\mathbf{t}$, forward $\mathbf{w}$, up $\mathbf{u}$, and left $\mathbf{v}=\mathbf{u}\times\mathbf{w}$.
- Transform world points to this camera frame (*view* transform) before applying projection.

OpenGL Fixed-Function Pipeline Cheatsheet

- Order: **Model** → **View** → **Projection** → **Viewport**.
- View (camera):

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eyeX,eyeY,eyeZ,  cenX,cenY,cenZ,  upX,upY,upZ);
```

- Perspective (viewing frustum and clipping):

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fovy, aspect, near, far);   // or glFrustum(...)
```

- Orthographic:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(left,right,bottom,top,near,far);   // 2D: gluOrtho2D(...)
```

- Objects outside the frustum are clipped; default window after projection and division is $[-1,1]^3$ in NDC.

Extras

- **Stereoscopic/3D**: render two slightly offset views (delivery via anaglyph, polarization, shutter glasses, etc.).
- Other projections (e.g., stereographic) exist for special purposes.

3 Colors

Light and Color Basics

- Visible light is a band of electromagnetic (EM) waves; what we perceive as a color is a *mixture* of EM frequencies (e.g., red $\sim 4.3\times10^{14}$ Hz, violet $\sim 7.5\times10^{14}$ Hz). White is the most "impure" (broad-mixture) color.

- A color's spectrum can be sketched by an energy–frequency curve: *dominant frequency* (peak), *brightness* (area under the curve), and *purity* (how concentrated the energy is around the dominant frequency).

Color Models (Tri-stimulus Theory)

- A **color model** chooses three descriptors (axes) to specify colors, echoing the three cone types in the human eye.
- We use three common models: **RGB** (displays), **CMY** (printing), and **HSV** (artist/user-oriented control).

RGB (Additive)

- Colors are 3D vectors $(r, g, b) \in [0, 1]^3$; adding colors is vector addition. The full set of colors forms a unit cube.
- *Additive* system: adding primaries makes the result *brighter*; complementary pairs (e.g., red–cyan, green–magenta, blue–yellow) sum to white.

CMY (Subtractive)

- Uses cyan, magenta, yellow as primaries (ink/filters).
- *Subtractive* system: stacking pigments removes light—adding primaries makes the result *darker*.

HSV (Hue–Saturation–Value)

- Constructed as a *hexcone*: project the RGB cube onto a hexagon and stack layers by brightness.
- $V = \max(R, G, B)$ (brightness), $S =$ distance from the gray axis to the color normalized by the hexagon radius (pureness), $H =$ angle around the axis (red $0°$, green $120°$, blue $240°$).
- Pure hue at $V = 1$, $S = 1$; when $S = 0$ the hue is undefined (grays).

Color Gamut and Devices

- No device or primary set reproduces *all* perceivable colors; the **gamut** is the subset achievable by a system.
- Different devices (and even extended-primary systems like RGBY or RGBCMY) have different gamuts.

Additive vs. Subtractive (Summary)

- **Additive (RGB)**: emit light; more primaries $\Rightarrow$ brighter (white at full).
- **Subtractive (CMY)**: filter/absorb light; more inks $\Rightarrow$ darker (black at full).

Illumination and Shading

Phong Illumination: $I_{phong} = I_a K_a + f_{att} I_p K_d (N \cdot L) + f_{att} I_p K_s (R \cdot V)^n$

$$\begin{bmatrix} i_r \\ i_g \\ i_b \end{bmatrix} = \begin{bmatrix} i_{ar}k_{ar} \\ i_{ag}k_{ag} \\ i_{ab}k_{ab} \end{bmatrix} \text{(Ambient)} + f_{att} I_p K_d (N \cdot L) \text{(Diffuse)} + f_{att} I_p K_s (R \cdot V)^n \text{(specular)}$$

- I_a luminance in ambient term: $[i_{ar} \quad i_{ag} \quad i_{ab}]^T$: uniform lighting on every surface in the scene
- K_a material property: $[k_{ar} \quad k_{ag} \quad k_{ab}]^T$
- K_a diffuse material property: $[k_{dr} \quad k_{dg} \quad k_{db}]^T$
- K_s specular mat. prop: $[k_{sr} \quad k_{sg} \quad k_{sb}]^T$ all material properties $\in [0, 1]$
- f_{att} distance attenuation factor (of point light source)
- I_p point light source vector
- $N \cdot L$ diffuse reflection $\propto \cos(\theta)$, N is surface normal, L is light vector
- $R \cdot V$ reflection vector, V is viewpoint vector, R is reflection vector.

$$R = 2(N \cdot L)N - L$$

- n shininess coefficient: higher $n \to$ smaller and sharper highlights $\in [1, 500]$

Multiple lightings:

$$I_{phong} = I_a K_a + \sum_i f_{att,i} I_{p,i} [K_d (N \cdot L) + K_s (R \cdot V)^n]$$

```
// Call once after you create the GL context/window.
static void setupPhongExample(bool highlight = false) {
    // ---------- Material (object) terms ----------
    // K_a: ambient reflectance (RGBA)
    GLfloat K_a[4] = { 0.20f, 0.20f, 0.20f, 1.0f };

    // K_d: diffuse/albedo reflectance (RGBA)
    GLfloat K_d[4] = { 0.70f, 0.70f, 0.75f, 1.0f }; // slightly bluish

    // K_s: specular reflectance (RGBA) - white when highlight is on
    GLfloat K_s[4] = { 0.20f, 0.20f, 0.20f, 1.0f };
    if (highlight) { K_s[0] = K_s[1] = K_s[2] = 1.0f; } // strong white specular

    // n: shininess exponent (0..128 in fixed-function OpenGL)
    GLfloat shininess = highlight ? 100.0f : 16.0f;

    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,   K_a);       //  K_a
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,   K_d);       //  K_d
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,  K_s);       //  K_s
    glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, shininess); //  n

    // ---------- Global ambient light (scene ambient) ----------
    // I_a: global ambient intensity (RGBA)
    GLfloat I_a[4] = { 0.15f, 0.15f, 0.15f, 1.0f };
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, I_a); // contributes K_a × I_a

    // Optional: two-sided lighting if you render both faces
    // glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

    // ---------- One positional light (Light 0) ----------
    // I_l: light intensities (ambient/diffuse/specular) for this light
    GLfloat L_ambient [4] = { 0.10f, 0.10f, 0.10f, 1.0f };
    GLfloat L_diffuse [4] = { 1.00f, 1.00f, 1.00f, 1.0f };
    GLfloat L_specular[4] = { 1.00f, 1.00f, 1.00f, 1.0f };

    glLightfv(GL_LIGHT0, GL_AMBIENT,  L_ambient);  // I_l (ambient)
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  L_diffuse);  // I_l (diffuse)
    glLightfv(GL_LIGHT0, GL_SPECULAR, L_specular); // I_l (specular)

    // Position (w=1 for point light, w=0 for directional/infinite)
    GLfloat lightPos[4] = { -3.0f, 4.0f, 6.0f, 1.0f }; // (x,y,z,1)
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);

    // ---------- Distance attenuation f_att(d) = 1/(k_c + k_l d + k_q d^2) ----------
    // Defaults are kc=1, kl=kq=0 (no falloff). Set your own:
    GLfloat k_c = 1.0f;   // constant term
    GLfloat k_l = 0.06f;  // linear term
    GLfloat k_q = 0.02f;  // quadratic term (most physically plausible)
    glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION,  k_c);
    glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION,    k_l);
    glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, k_q);

    // ---------- Pipeline toggles ----------
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    glShadeModel(GL_SMOOTH);   // Gouraud shading
    glEnable(GL_NORMALIZE);    // keep normals unit-length after scaling
    glEnable(GL_DEPTH_TEST);   // needed for proper 3D visibility
}
```

Spotlight: $\phi = \cos^{-1}(s \cdot v)$ where v is spotlight direction, s is vector from light source to the surface. both are unit vectors.
light intensity of spotlight at s: $I_p' = I_p (\cos\phi)^n = I_p (s \cdot v_s)^n$

Shadings

- Flat shading: `glShadeModel(GL_FLAT);`
- Gouraud shading: use vertex normal (avg of all surfaces sharing the vertex), pixels in the polygon has interpolated color from all vertices. `glShadeModel(GL_SMOOTH)`
- Phong Shading: Compute vertex normal, interpolate normal vectors on each pixels
- Toon shading: phong shading but use $N \cdot L$ and $R \cdot V$ with discrete range

Problems with Interpolated Shading with Polygonal Models

- Non-global effects
- No shadow
- Polygonal silhouette
- Orientation dependence
- Shared vertices
- Misleading vertex normals

Global vs Non-global Illumination

- **Global (Ray Tracing)**
  - More photorealistic/complex
  - Computes many types of physical light interactions
  - Slow
  - e.g., CG movies
- **Non-global (e.g., Phong Shading)**
  - **Only** considers the light source, the surface point, and the viewer directly
  - Faster but less realistic
  - e.g., no shadow/reflection
  - e.g., real-time 3D games

Fractal drawings

```
void drawRecursiveSquares(int n) {
  glPushMatrix();
  if(n==0) return;
  for(int i=0;i<4;i++) {
    glPushMatrix();glPushMatrix();
    glRotatef(90*i,0,0,1);
    glTranslatef(1,1,0);
    glScalef(0 5 0 5 0 5);glScalef(0.5,0.5,0.5);
    drawRecursiveSquares(n-1);
    glPopMatrix();
  }
  drawUnitTwoSquare();
  glPopMatrix();
}
void drawHive(int n) {
  if (n == 1) {
    drawHexagon();
    return;
  }
  glPushMatrix();
  for (int i = 0; i < 6; i++) {
    glRotated(60, 0, 0, 1);
    glPushMatrix();
    glTranslated(0, sqrt(3), 0);
    drawHives(n - 1);
    glPopMatrix();
  }
  glPopMatrix();
}
```