

Application and Math Concepts of Procedural Elements for Computer Graphics

ASCII 3D Renderer in Pure C

Geovanne Gallinati

February 13, 2026

GitHub Repository

Contents

1	Homogeneous Coordinates	2
1.1	Row Vector Convention	3
1.2	House Vertices	3
2	Matrix Operations	4
3	3D Transformation Matrices	5
3.1	Translation	5
3.2	Scaling	6
3.3	Rotation About the X Axis	6
3.4	Rotation About the Y Axis	7
3.5	Rotation About the Z Axis	7
4	Composite Transformations	8
4.1	Rotation About an Arbitrary Point	8
4.2	Numerical Example	9
5	3D to 2D Projection	9
5.1	Orthographic (Parallel) Projection	10
5.2	Perspective Projection	10
5.3	Viewport Mapping	11
6	Surface Normal and Cross Product	12
6.1	Example: Front Face	13
7	Hidden Surface Removal	13
7.1	Back-Face Culling	13
7.2	Painter's Algorithm	14
8	Line Rasterization: DDA Algorithm	14

9 Polygon Filling: Scan-Line Algorithm	16
10 Diffuse Lighting: Lambert's Law	17
10.1 Mapping to ASCII	18
11 Complete Rendering Pipeline	19
12 Reference	21

1 Homogeneous Coordinates

Mathematics

Points in **homogeneous coordinates** are defined as 4-component vectors. A 3D point (x, y, z) is represented as:

$$\mathbf{P} = [x \ y \ z \ w] \quad (1)$$

where w is the *homogeneous coordinate*. The corresponding Cartesian point is:

$$(X, Y, Z) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w} \right) \quad (2)$$

By convention, we use $w = 1$ for points. In perspective projections, w takes values other than 1, requiring *perspective division*.

C Implementation `matrix.h` / `matrix.c`

The data types that represent homogeneous coordinates:

```

1 typedef struct {
2     double m[4][4];
3 } Mat4;
4
5 typedef struct {
6     double v[4]; /* v[0]=x, v[1]=y, v[2]=z, v[3]=w */
7 } Vec4;
8
9 typedef struct {
10    double x, y, z;
11 } Vec3;
```

Creating a point with $w = 1$ (default) and with explicit w :

```

1 Vec4 vec4_create(double x, double y, double z)
2 {
3     Vec4 v;
4     v.v[0] = x;
5     v.v[1] = y;
6     v.v[2] = z;
7     v.v[3] = 1.0; /* w = 1 by default */
8     return v;
9 }
10
11 Vec4 vec4_create_w(double x, double y, double z, double w)
12 {
13     Vec4 v;
14     v.v[0] = x; v.v[1] = y;
15     v.v[2] = z; v.v[3] = w;
16     return v;
17 }
```

1.1 Row Vector Convention

Mathematics

Row vectors with post-multiplication:

$$\mathbf{P}' = \mathbf{P} \cdot \mathbf{M} \quad (3)$$

where \mathbf{P} is a 1×4 row vector and \mathbf{M} is a 4×4 matrix. This is **opposite** to OpenGL's convention ($\mathbf{P}' = \mathbf{M} \cdot \mathbf{P}$).

The translation terms are placed in the **last row** of the matrix.

C Implementation `matrix.c: mat4_transform_point`

The row-vector \times matrix multiplication $\mathbf{P}' = \mathbf{P} \cdot \mathbf{M}$:

```

1 void mat4_transform_point(const Mat4 *M, const Vec4 *p, Vec4 *result
  )
2 {
3     Vec4 temp;
4     int j, i;
5     for (j = 0; j < 4; j++)
6     {
7         temp.v[j] = 0.0;
8         for (i = 0; i < 4; i++)
9         {
10             temp.v[j] += p->v[i] * M->m[i][j];
11         }
12     }
13     *result = temp;
14 }
```

Notice: the inner loop computes $P'_j = \sum_{i=0}^3 P_i \cdot M_{ij}$, which is exactly the row-vector convention $\mathbf{P}' = \mathbf{P} \cdot \mathbf{M}$.

1.2 House Vertices

Mathematics

The house model uses 10 vertices in homogeneous coordinates. Examples:

$$\mathbf{V}_0 = [-1 \quad -1 \quad 1 \quad 1] \quad (\text{front-left-bottom}) \quad (4)$$

$$\mathbf{V}_8 = [0 \quad 2 \quad 1 \quad 1] \quad (\text{front roof peak}) \quad (5)$$

C Implementation `geometry.c: geometry_init_house`

```

1 void geometry_init_house(Model *model)
2 {
3     /* Front face (z = +1) */
4     model->vertices[0] = vec4_create(-1.0, -1.0, 1.0);
5     model->vertices[1] = vec4_create( 1.0, -1.0, 1.0);
6     model->vertices[2] = vec4_create( 1.0,  1.0, 1.0);
7     model->vertices[3] = vec4_create(-1.0,  1.0, 1.0);
8
9     /* Back face (z = -1) */
10    model->vertices[4] = vec4_create(-1.0, -1.0, -1.0);
```

```

11 model->vertices[5] = vec4_create( 1.0, -1.0, -1.0);
12 model->vertices[6] = vec4_create( 1.0,  1.0, -1.0);
13 model->vertices[7] = vec4_create(-1.0,  1.0, -1.0);
14
15 /* Roof peaks */
16 model->vertices[8] = vec4_create( 0.0,  2.0,  1.0);
17 model->vertices[9] = vec4_create( 0.0,  2.0, -1.0);
18
19 model->num_vertices = HOUSE_NUM_VERTICES;
20 /* ... face definitions follow ... */
21 }

```

Each call to `vec4_create(x,y,z)` produces $[x, y, z, 1]$.

2 Matrix Operations

Mathematics

The **identity matrix** **I** satisfies $\mathbf{P} \cdot \mathbf{I} = \mathbf{P}$:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6)$$

Matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$:

$$C_{ij} = \sum_{k=0}^3 A_{ik} \cdot B_{kj} \quad (7)$$

C Implementation `matrix.c`

```

1 void mat4_identity(Mat4 *m)
2 {
3     memset(m->m, 0, sizeof(m->m));
4     m->m[0][0] = 1.0;
5     m->m[1][1] = 1.0;
6     m->m[2][2] = 1.0;
7     m->m[3][3] = 1.0;
8 }
9
10 void mat4_multiply(const Mat4 *A, const Mat4 *B, Mat4 *result)
11 {
12     Mat4 temp;
13     int i, j, k;
14     for (i = 0; i < 4; i++)
15     {
16         for (j = 0; j < 4; j++)
17         {
18             temp.m[i][j] = 0.0;
19             for (k = 0; k < 4; k++)
20             {
21                 temp.m[i][j] += A->m[i][k] * B->m[k][j];
22             }
23         }
24     }
25 }

```

```

23     }
24     }
25     mat4_copy(&temp, result);
26 }

```

The triple nested loop directly implements $C_{ij} = \sum_k A_{ik}B_{kj}$. A temporary matrix `temp` is used so that `result` can safely alias A or B.

3 3D Transformation Matrices

There are five fundamental transformations using 4×4 matrices that operate on row vectors in homogeneous coordinates.

3.1 Translation

Mathematics

The translation matrix shifts a point by (t_x, t_y, t_z) :

$$\mathbf{T}(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \quad (8)$$

Verification:

$$\mathbf{P}' = \mathbf{P} \cdot \mathbf{T} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = \begin{bmatrix} x + t_x & y + t_y & z + t_z & 1 \end{bmatrix} \quad (9)$$

The terms t_x, t_y, t_z are in the last row because of the row vector convention.

C Implementation `transform.c: transform_translation`

```

1 void transform_translation(double tx, double ty, double tz, Mat4 *
  out)
2 {
3     mat4_identity(out);
4     out->m[3][0] = tx;    /* last row, column 0 */
5     out->m[3][1] = ty;    /* last row, column 1 */
6     out->m[3][2] = tz;    /* last row, column 2 */
7 }

```

Starts from identity and places t_x, t_y, t_z in row 3 (the last row), matching the mathematical definition exactly.

3.2 Scaling

Mathematics

The scaling matrix relative to the origin:

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (10)$$

Result: $\mathbf{P}' = [s_x x \quad s_y y \quad s_z z \quad 1]$

C Implementation transform.c: transform_scale

```
1 void transform_scale(double sx, double sy, double sz, Mat4 *out)
2 {
3     mat4_identity(out);
4     out->m[0][0] = sx;    /* scale x */
5     out->m[1][1] = sy;    /* scale y */
6     out->m[2][2] = sz;    /* scale z */
7 }
```

The diagonal elements M_{00}, M_{11}, M_{22} hold the scale factors.

3.3 Rotation About the X Axis

Mathematics

Following the right-hand rule, a positive rotation of θ about X rotates Y toward Z :

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11)$$

Verification with $\theta = 90$: $(0, 1, 0) \rightarrow (0, 0, 1)$:

$$\begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \quad (12)$$

C Implementation transform.c: transform_rotation_x

```
1 void transform_rotation_x(double theta, Mat4 *out)
2 {
3     double c = cos(theta);
4     double s = sin(theta);
5     mat4_identity(out);
6     out->m[1][1] = c;    /* cos(theta) at [1][1] */
7     out->m[1][2] = s;    /* sin(theta) at [1][2] */
8     out->m[2][1] = -s;   /* -sin(theta) at [2][1] */
9 }
```

```

9     out->m[2][2] = c;    /* cos(theta) at [2][2] */
10 }

```

The four modified elements correspond exactly to the 2×2 rotation sub-matrix in rows/columns 1–2 of the mathematical definition.

3.4 Rotation About the Y Axis

Mathematics

Positive rotation rotates Z toward X :

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos \theta & 0 & -\sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (13)$$

This is the main rotation in the animation: the house rotates continuously about the vertical Y axis.

C Implementation transform.c: transform_rotation_y

```

1 void transform_rotation_y(double theta, Mat4 *out)
2 {
3     double c = cos(theta);
4     double s = sin(theta);
5     mat4_identity(out);
6     out->m[0][0] = c;    /* cos(theta) at [0][0] */
7     out->m[0][2] = -s;   /* -sin(theta) at [0][2] */
8     out->m[2][0] = s;    /* sin(theta) at [2][0] */
9     out->m[2][2] = c;    /* cos(theta) at [2][2] */
10 }

```

The 2×2 rotation sub-matrix occupies rows/columns 0 and 2, skipping row/column 1 (the Y axis is unchanged). Note the negative sign is at position $[0][2]$, matching the mathematical formula.

3.5 Rotation About the Z Axis

Mathematics

Positive rotation rotates X toward Y :

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (14)$$

Note: The position of $-\sin \theta$ differs between the three rotations, following the cyclic rule $X \rightarrow Y \rightarrow Z \rightarrow X$.

C Implementation transform.c: transform_rotation_z

```

1 void transform_rotation_z(double theta, Mat4 *out)
2 {
3     double c = cos(theta);
4     double s = sin(theta);
5     mat4_identity(out);
6     out->m[0][0] = c;    /* cos(theta) at [0][0] */
7     out->m[0][1] = s;    /* sin(theta) at [0][1] */
8     out->m[1][0] = -s;   /* -sin(theta) at [1][0] */
9     out->m[1][1] = c;    /* cos(theta) at [1][1] */
10 }

```

The 2×2 sub-matrix occupies rows/columns 0–1; row/column 2 (Z axis) is unchanged.

4 Composite Transformations

Mathematics

In the row vector convention, the **multiplication order** is the **application order**:

$$\mathbf{M}_{\text{composite}} = \mathbf{M}_1 \cdot \mathbf{M}_2 \cdot \mathbf{M}_3 \cdots \quad (15)$$

where \mathbf{M}_1 is applied first, then \mathbf{M}_2 , etc.

4.1 Rotation About an Arbitrary Point

Mathematics

To rotate about a center $\mathbf{C} = (c_x, c_y, c_z)$ instead of the origin:

1. **Translate** center to origin: $\mathbf{T}(-c_x, -c_y, -c_z)$
2. **Rotate**: $\mathbf{R}_y(\theta)$
3. **Translate back**: $\mathbf{T}(c_x, c_y, c_z)$

$$\mathbf{M} = \mathbf{T}(-c_x, -c_y, -c_z) \cdot \mathbf{R}_y(\theta) \cdot \mathbf{T}(c_x, c_y, c_z) \quad (16)$$

C Implementation transform.c: transform_rotate_y_around

```

1 void transform_rotate_y_around(double theta,
2     double cx, double cy, double cz, Mat4 *out)
3 {
4     Mat4 t1, r, t2, temp;
5
6     /* Step 1: Translate center to origin */
7     transform_translation(-cx, -cy, -cz, &t1);
8
9     /* Step 2: Rotate about Y */
10    transform_rotation_y(theta, &r);
11

```

```

12  /* Step 3: Translate back */
13  transform_translation(cx, cy, cz, &t2);
14
15  /* Composite: M = T1 * R * T2 */
16  mat4_multiply(&t1, &r, &temp);
17  mat4_multiply(&temp, &t2, out);
18 }

```

The three steps from the mathematical definition map directly to three function calls, composed via two matrix multiplications. The same pattern is used for `transform_rotate_x_around`.

4.2 Numerical Example

Mathematics

The geometric center of the house is $\mathbf{C} = (0, 0.4, 0)$. To rotate 30 about Y around the center:

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -0.4 & 0 & 1 \end{bmatrix}, \mathbf{R}_y = \begin{bmatrix} 0.866 & 0 & -0.5 & 0 \\ 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \mathbf{T}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0.4 & 0 & 1 \end{bmatrix} \quad (17)$$

Since $c_x = c_z = 0$, the composite reduces to the pure rotation matrix.

Applying to vertex $\mathbf{V}_1 = [1, -1, 1, 1]$:

$$\mathbf{V}'_1 = [1.366 \quad -1 \quad 0.366 \quad 1] \quad (18)$$

C Implementation `geometry.c: geometry_compute_center`

The center \mathbf{C} used in the rotation is computed as the average of all vertices:

```

1 void geometry_compute_center(Model *model)
2 {
3     double cx = 0, cy = 0, cz = 0;
4     int i;
5     for (i = 0; i < model->num_vertices; i++)
6     {
7         cx += model->vertices[i].v[0];
8         cy += model->vertices[i].v[1];
9         cz += model->vertices[i].v[2];
10    }
11    model->center_x = cx / model->num_vertices;
12    model->center_y = cy / model->num_vertices;
13    model->center_z = cz / model->num_vertices;
14 }

```

This yields $\mathbf{C} = (0, 0.4, 0)$ for the 10-vertex house model.

5 3D to 2D Projection

There are two types of projection.

5.1 Orthographic (Parallel) Projection

Mathematics

The simplest projection, which discards the z coordinate:

$$\mathbf{P}_{ortho} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (19)$$

Result: $x_{screen} = x$, $y_{screen} = y$.

Parallel lines remain parallel; no depth effect.

C Implementation `projection.c`: orthographic branch

```

1 void projection_init_ortho(Projection *proj, int vp_width,
2                             int vp_height, double scale)
3 {
4     proj->type = PROJ_ORTHOGRAPHIC;
5     proj->d = 0.0;
6     proj->vp_width = vp_width;
7     proj->vp_height = vp_height;
8     proj->scale = scale;
9     proj->aspect_fix = 2.0;
10 }
11
12 /* Inside projection_project(): */
13 else /* orthographic */
14 {
15     x2d = x; /* simply copy x */
16     y2d = y; /* simply copy y, discard z */
17 }
```

The orthographic case directly assigns x and y , ignoring z entirely.

5.2 Perspective Projection

Mathematics

The perspective projection is defined with the eye at $(0, 0, d)$ and projection plane at $z = 0$:

$$\mathbf{P}_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{d} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (20)$$

After multiplication and **perspective division**:

$$x_{screen} = \frac{x}{1 - z/d}, \quad y_{screen} = \frac{y}{1 - z/d} \quad (21)$$

Geometric interpretation: Distant objects (z more negative, $1 - z/d$ larger) are divided by a larger number, making them smaller.

Singularity: When $z = d$, the denominator is zero (point at the eye). We guard against this with $w > 0.001$.

C Implementation `projection.c: projection_project`

```

1  int projection_project(const Projection *proj, const Vec4 *p3d,
2                          double *screen_x, double *screen_y)
3  {
4      double x = p3d->v[0];
5      double y = p3d->v[1];
6      double z = p3d->v[2];
7      double x2d, y2d;
8
9      if (proj->type == PROJ_PERSPECTIVE)
10     {
11         double w = 1.0 - z / proj->d;  /* w = 1 - z/d */
12         if (w <= 0.001)
13         {
14             return 0;  /* singularity guard */
15         }
16         x2d = x / w;  /* perspective division */
17         y2d = y / w;
18     }
19     else
20     {
21         x2d = x;
22         y2d = y;
23     }
24     /* ... viewport mapping follows ... */
25     return 1;
26 }

```

The perspective division $x/(1 - z/d)$ is computed explicitly. The guard $w \leq 0.001$ prevents division by zero at the singularity.

5.3 Viewport Mapping**Mathematics**

After projection, coordinates are mapped to the terminal grid:

$$col = x_{2d} \cdot scale \cdot f_{aspect} + \frac{width}{2} \quad (22)$$

$$row = \frac{height}{2} - y_{2d} \cdot scale \quad (23)$$

where $f_{aspect} \approx 2.0$ corrects for terminal characters being taller than wide, and Y is inverted (terminal y increases downward).

C Implementation `projection.c: viewport mapping`

```

1      *screen_x = x2d * proj->scale * proj->aspect_fix
2                  + proj->vp_width / 2.0;
3      *screen_y = proj->vp_height / 2.0
4                  - y2d * proj->scale;

```

This is the final part of `projection_project()`. The `aspect_fix = 2.0` stretches x to compensate for tall characters. The subtraction in y inverts the axis.

6 Surface Normal and Cross Product

Mathematics

The **face normal** is defined as the cross product of two edge vectors. For a face with vertices $\mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3$ (counter-clockwise as seen from outside):

$$\mathbf{E}_1 = \mathbf{V}_2 - \mathbf{V}_1, \quad \mathbf{E}_2 = \mathbf{V}_3 - \mathbf{V}_1 \quad (24)$$

$$\mathbf{N} = \mathbf{E}_1 \times \mathbf{E}_2 \quad (25)$$

Component expansion:

$$\begin{cases} N_x = E_{1y} \cdot E_{2z} - E_{1z} \cdot E_{2y} \\ N_y = E_{1z} \cdot E_{2x} - E_{1x} \cdot E_{2z} \\ N_z = E_{1x} \cdot E_{2y} - E_{1y} \cdot E_{2x} \end{cases} \quad (26)$$

C Implementation matrix.c: vec3_cross and vec3_sub

```

1  Vec3 vec3_sub(Vec3 a, Vec3 b)
2  {
3      Vec3 result;
4      result.x = a.x - b.x;      /* E = V2 - V1 */
5      result.y = a.y - b.y;
6      result.z = a.z - b.z;
7      return result;
8  }
9
10 Vec3 vec3_cross(Vec3 a, Vec3 b)
11 {
12     Vec3 result;
13     result.x = a.y * b.z - a.z * b.y;    /* Nx */
14     result.y = a.z * b.x - a.x * b.z;    /* Ny */
15     result.z = a.x * b.y - a.y * b.x;    /* Nz */
16     return result;
17 }
```

Each line of `vec3_cross` corresponds exactly to one component of Equation 26.

C Implementation geometry.c: geometry_face_normal

Combining subtraction and cross product to compute a face's outward normal:

```

1  Vec3 geometry_face_normal(const Model *model, int face_idx)
2  {
3      const Face *face = &model->faces[face_idx];
4      const Vec4 *v0 = &model->transformed[face->indices[0]];
5      const Vec4 *v1 = &model->transformed[face->indices[1]];
6      const Vec4 *v2 = &model->transformed[face->indices[2]];
7
8      Vec3 p0 = vec3_create(v0->v[0], v0->v[1], v0->v[2]);
9      Vec3 p1 = vec3_create(v1->v[0], v1->v[1], v1->v[2]);
10     Vec3 p2 = vec3_create(v2->v[0], v2->v[1], v2->v[2]);
11 }
```

```

12     Vec3 e1 = vec3_sub(p1, p0);    /* E1 = V2 - V1 */
13     Vec3 e2 = vec3_sub(p2, p0);    /* E2 = V3 - V1 */
14     return vec3_cross(e1, e2);      /* N = E1 x E2 */
15 }

```

The function extracts 3D coordinates from the homogeneous `Vec4`, computes the two edge vectors, and returns their cross product.

6.1 Example: Front Face

Mathematics

Front face vertices: $\mathbf{V}_0 = (-1, -1, 1)$, $\mathbf{V}_1 = (1, -1, 1)$, $\mathbf{V}_2 = (1, 1, 1)$.

$$\mathbf{E}_1 = (2, 0, 0), \quad \mathbf{E}_2 = (2, 2, 0) \quad (27)$$

$$\mathbf{N} = (0 - 0, 0 - 0, 4 - 0) = (0, 0, 4) \quad (28)$$

Normal points in $+Z$ (toward observer) \implies front-facing.

7 Hidden Surface Removal

Algorithms for determining which surfaces are visible are as follows.

7.1 Back-Face Culling

Mathematics

A face is **visible** if its normal points toward the observer. For orthographic projection with the observer at $+Z$:

$$\text{Face is \textbf{visible} if } N_z > 0 \quad (29)$$

Front face: $N_z = 4 > 0 \rightarrow$ visible.

Back face: $N_z = -4 < 0 \rightarrow$ hidden.

Limitation: Works perfectly for convex objects. For non-convex objects (e.g., house with roof), we also need the Painter's Algorithm.

C Implementation `pipeline.c`: back-face culling

```

1     for (i = 0; i < model->num_faces; i++)
2     {
3         Vec3 normal = geometry_face_normal(model, i);
4         if (normal.z > 0)    /* Nz > 0 => visible */
5         {
6             visible_faces[num_visible].face_idx = i;
7             visible_faces[num_visible].avg_z =
8                 geometry_face_avg_z(model, i);
9             num_visible++;
10        }
11    }

```

The single condition `normal.z > 0` implements Equation 29. Only faces that pass this test enter the visible list.

7.2 Painter's Algorithm

Mathematics

The Painter's Algorithm draws faces from farthest to nearest:

1. Compute the **average depth** of each visible face:

$$\bar{z}_f = \frac{1}{n} \sum_{i=1}^n z_i \quad (30)$$

2. **Sort** faces by \bar{z} in ascending order (farthest first).
3. **Rasterize** in order: closer faces overwrite farther ones.

C Implementation pipeline.c and geometry.c

Average depth computation:

```

1 double geometry_face_avg_z(const Model *model, int face_idx)
2 {
3     const Face *face = &model->faces[face_idx];
4     double sum_z = 0;
5     int i;
6     for (i = 0; i < face->num_verts; i++)
7     {
8         sum_z += model->transformed[face->indices[i]].v[2];
9     }
10    return sum_z / face->num_verts;
11 }
```

Sorting by depth (ascending = farthest first):

```

1 static int compare_face_depth(const void *a, const void *b)
2 {
3     const FaceDepth *fa = (const FaceDepth *)a;
4     const FaceDepth *fb = (const FaceDepth *)b;
5     if (fa->avg_z < fb->avg_z) return -1;
6     if (fa->avg_z > fb->avg_z) return 1;
7     return 0;
8 }
9
10 /* In pipeline_render_frame(): */
11 qsort(visible_faces, num_visible,
12       sizeof(FaceDepth), compare_face_depth);
```

The C standard library `qsort` sorts by \bar{z} ; faces are then drawn in order so closer faces overwrite farther ones.

8 Line Rasterization: DDA Algorithm

Mathematics

The **DDA (Digital Differential Analyzer)** is described as follows.
A segment from (x_0, y_0) to (x_1, y_1) with $\Delta x = x_1 - x_0$, $\Delta y = y_1 - y_0$.

Choose $N = \max(|\Delta x|, |\Delta y|)$ steps so that the dominant axis advances by exactly 1 pixel per step:

$$x_{inc} = \frac{\Delta x}{N}, \quad y_{inc} = \frac{\Delta y}{N} \quad (31)$$

At each step $i = 0, 1, \dots, N$:

$$x_i = x_0 + i \cdot x_{inc}, \quad y_i = y_0 + i \cdot y_{inc} \quad (32)$$

Plot pixel at $(\text{round}(x_i), \text{round}(y_i))$. Complexity: $O(N)$ per line.

C Implementation `render.c: render_line`

```

1 void render_line(Framebuffer *fb, double x0, double y0,
2                   double x1, double y1, char c)
3 {
4     double dx = x1 - x0;                /* delta x */
5     double dy = y1 - y0;                /* delta y */
6     int abs_dx = (int)(fabs(dx) + 0.5);
7     int abs_dy = (int)(fabs(dy) + 0.5);
8     int steps = abs_dx > abs_dy ? abs_dx : abs_dy; /* N */
9
10    if (steps == 0)
11    {
12        fb_set(fb, (int)(x0 + 0.5), (int)(y0 + 0.5), c);
13        return;
14    }
15
16    double x_inc = dx / (double)steps;    /* x_inc = dx/N */
17    double y_inc = dy / (double)steps;    /* y_inc = dy/N */
18    double x = x0, y = y0;
19    int i;
20
21    for (i = 0; i <= steps; i++)
22    {
23        fb_set(fb, (int)(x + 0.5), (int)(y + 0.5), c);
24        x += x_inc;    /* x_i = x0 + i * x_inc */
25        y += y_inc;    /* y_i = y0 + i * y_inc */
26    }
27 }
```

The code follows the DDA algorithm step by step: $\text{steps} = N = \max(|\Delta x|, |\Delta y|)$; increments are $\Delta x/N$ and $\Delta y/N$; rounding is done by adding 0.5 before casting to `int`.

9 Polygon Filling: Scan-Line Algorithm

Mathematics

The **scan-line algorithm** is described as follows:

For each horizontal scan line y :

1. Find all **intersections** with polygon edges
2. **Sort** intersections by ascending x
3. **Fill** between consecutive pairs: (x_0, x_1) , (x_2, x_3) , ...

Edge/scan-line intersection for an edge from (x_1, y_1) to (x_2, y_2) :

$$x_{int} = x_1 + \frac{y - y_1}{y_2 - y_1} \cdot (x_2 - x_1) \quad (33)$$

Condition: $y \in [\min(y_1, y_2), \max(y_1, y_2))$ (half-open interval to avoid counting shared vertices twice).

C Implementation `render.c: render_fill_polygon`

```

1 void render_fill_polygon(Framebuffer *fb,
2     const double *verts_x, const double *verts_y,
3     int n, char c)
4 {
5     /* ... compute ymin, ymax ... */
6
7     for (y = ymin; y <= ymax; y++) /* each scan line */
8     {
9         num_intersections = 0;
10        for (i = 0; i < n; i++)
11        {
12            j = (i + 1) % n; /* next vertex (wraps) */
13            double y1 = verts_y[i], y2 = verts_y[j];
14            double x1 = verts_x[i], x2 = verts_x[j];
15
16            /* Half-open interval test */
17            if ((y1 <= y && y2 > y) || (y2 <= y && y1 > y))
18            {
19                /* x_int = x1 + (y-y1)/(y2-y1) * (x2-x1) */
20                double t = ((double)y - y1) / (y2 - y1);
21                double x_int = x1 + t * (x2 - x1);
22                intersections[num_intersections++] = x_int;
23            }
24        }
25
26        /* Sort intersections by x (insertion sort) */
27        for (i = 1; i < num_intersections; i++) { /* ... */ }
28
29        /* Fill between pairs */
30        for (i = 0; i + 1 < num_intersections; i += 2)
31        {
32            int xstart = (int)(intersections[i] + 0.5);
33            int xend = (int)(intersections[i + 1] + 0.5);
34            int col;

```

```

35         for (col = xstart; col <= xend; col++)
36             fb->buffer[y][col] = c;
37     }
38 }
39 }

```

The half-open interval condition $(y1 \leq y \ \&\& \ y2 > y) \ || \ (y2 \leq y \ \&\& \ y1 > y)$ ensures each vertex intersection is counted exactly once. The intersection formula $x1 + t * (x2 - x1)$ directly implements Equation 33.

10 Diffuse Lighting: Lambert's Law

Mathematics

Lambertian diffuse lighting is described as follows.

Lambert's Law: the reflected intensity is proportional to $\cos \theta$ between the normal and light direction:

$$I_{diffuse} = I_d \cdot \cos \theta = I_d \cdot (\hat{\mathbf{N}} \cdot \hat{\mathbf{L}}) \quad (34)$$

where $\hat{\mathbf{N}}$ and $\hat{\mathbf{L}}$ are unit vectors:

$$\hat{\mathbf{N}} = \frac{\mathbf{N}}{|\mathbf{N}|}, \quad |\mathbf{N}| = \sqrt{N_x^2 + N_y^2 + N_z^2} \quad (35)$$

With **ambient light** I_a :

$$I = I_a + I_d \cdot \max(0, \hat{\mathbf{N}} \cdot \hat{\mathbf{L}}) \quad (36)$$

In our renderer: $I_a = 0.15$, $I_d = 0.85$, so $I \in [0.15, 1.0]$.

C Implementation `matrix.c`: normalization

Vector normalization $\hat{\mathbf{N}} = \mathbf{N}/|\mathbf{N}|$:

```

1 double vec3_length(Vec3 v)
2 {
3     return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
4 }
5
6 Vec3 vec3_normalize(Vec3 v)
7 {
8     double len = vec3_length(v);
9     Vec3 result;
10    if (len > 1e-10)
11    {
12        result.x = v.x / len;
13        result.y = v.y / len;
14        result.z = v.z / len;
15    }
16    else
17    {
18        result.x = result.y = result.z = 0.0;
19    }
20    return result;

```

```

21 }
22
23 double vec3_dot(Vec3 a, Vec3 b)
24 {
25     return a.x * b.x + a.y * b.y + a.z * b.z;
26 }

```

The guard `len > 1e-10` prevents division by zero for degenerate normals.

C Implementation `render.c: render_shade`

The complete Lambert shading computation:

```

1 char render_shade(Vec3 normal, Vec3 light_dir)
2 {
3     Vec3 n = vec3_normalize(normal);      /* N_hat */
4     Vec3 l = vec3_normalize(light_dir);   /* L_hat */
5     double dot = vec3_dot(n, l);          /* N . L */
6     if (dot < 0.0) dot = 0.0;              /* max(0, ...) */
7
8     double ambient = 0.15;                /* I_a = 0.15 */
9     double diffuse = 0.85;                /* I_d = 0.85 */
10    double intensity = ambient + diffuse * dot;
11    if (intensity > 1.0) intensity = 1.0;
12
13    int idx = (int)(intensity * (ASCII_RAMP_LEN - 1) + 0.5);
14    if (idx < 0) idx = 0;
15    if (idx >= ASCII_RAMP_LEN) idx = ASCII_RAMP_LEN - 1;
16
17    return ascii_ramp[idx];
18 }

```

Line by line, this implements $I = I_a + I_d \cdot \max(0, \hat{N} \cdot \hat{L})$ from Equation 36, then maps I to an ASCII character.

10.1 Mapping to ASCII

Mathematics

The intensity $I \in [0.15, 1.0]$ is mapped to a ramp of 12 ASCII characters:

" .,:;=+*#%@\$" (index 0 to 11, dark \rightarrow bright)

$$idx = \text{round}(I \cdot 11) \quad (37)$$

Example: Light $\hat{L} = (0.46, 0.74, 0.56)$ (normalized from $(0.5, 0.8, 0.6)$). For the front face $\hat{N} = (0, 0, 1)$:

$$\hat{N} \cdot \hat{L} = 0.56, \quad I = 0.15 + 0.85 \times 0.56 = 0.626, \quad idx = \text{round}(6.89) = 7 \Rightarrow '*' \quad (38)$$

C Implementation `render.h` and `pipeline.c`

The ASCII ramp definition and light direction:

```

1  /* render.h */
2  #define ASCII_RAMP " .,:;=+*%@$"
3  #define ASCII_RAMP_LEN 12
4
5  /* pipeline.c */
6  static const Vec3 LIGHT_DIR = {0.5, 0.8, 0.6};

```

The ramp index $(\text{int})(\text{intensity} * 11 + 0.5)$ implements Equation 37.

11 Complete Rendering Pipeline

The pipeline follows this order:

1. **Define model:** vertices and faces
2. **Transform vertices:** $V' = V \cdot M$
3. **Compute normals:** $N = E_1 \times E_2$
4. **Back-face culling:** $N_z > 0 \implies \text{visible}$
5. **Painter's sort:** sort by \bar{z}
6. **Project 3D \rightarrow 2D**
7. **Rasterize:** DDA + Scan-line
8. **Lambert shading:** $I = I_a + I_d \cdot (\hat{N} \cdot \hat{L})$
9. **Display on terminal**

C Implementation pipeline.c: pipeline_render_frame (complete)

The entire pipeline in a single function:

```

1  void pipeline_render_frame(Model *model, Framebuffer *fb,
2      const Projection *proj,
3      double angle_y, double angle_x, RenderMode mode)
4  {
5      Mat4 rot_y, rot_x, composite;
6      int i;
7      FaceDepth visible_faces[HOUSE_NUM_FACES];
8      int num_visible = 0;
9
10     /* --- Step 2: Build composite transformation --- */
11     transform_rotate_y_around(angle_y,
12         model->center_x, model->center_y, model->center_z,
13         &rot_y);
14     transform_rotate_x_around(angle_x,
15         model->center_x, model->center_y, model->center_z,
16         &rot_x);
17     mat4_multiply(&rot_y, &rot_x, &composite);
18
19     /* Transform all vertices: V' = V * M */
20     for (i = 0; i < model->num_vertices; i++)
21     {

```

```

22     mat4_transform_point(&composite,
23         &model->vertices[i], &model->transformed[i]);
24 }
25
26 /* --- Steps 3-4: Normal + Back-face culling --- */
27 for (i = 0; i < model->num_faces; i++)
28 {
29     Vec3 normal = geometry_face_normal(model, i);
30     if (normal.z > 0)
31     {
32         visible_faces[num_visible].face_idx = i;
33         visible_faces[num_visible].avg_z =
34             geometry_face_avg_z(model, i);
35         num_visible++;
36     }
37 }
38
39 /* --- Step 5: Painter's sort --- */
40 qsort(visible_faces, num_visible,
41     sizeof(FaceDepth), compare_face_depth);
42
43 /* --- Steps 6-8: Project, Rasterize, Shade --- */
44 for (i = 0; i < num_visible; i++)
45 {
46     int fi = visible_faces[i].face_idx;
47     const Face *face = &model->faces[fi];
48     double px[MAX_FACE_VERTS], py[MAX_FACE_VERTS];
49     int v, valid = 1;
50
51     /* Step 6: Project 3D -> 2D */
52     for (v = 0; v < face->num_verts; v++)
53     {
54         int idx = face->indices[v];
55         if (!projection_project(proj,
56             &model->transformed[idx], &px[v], &py[v]))
57             { valid = 0; break; }
58     }
59     if (!valid) continue;
60
61     /* Step 7-8: Fill with shading */
62     if (mode == RENDER_FILLED || mode == RENDER_BOTH)
63     {
64         Vec3 normal = geometry_face_normal(model, fi);
65         char shade_char = render_shade(normal, LIGHT_DIR);
66         render_fill_polygon(fb, px, py,
67             face->num_verts, shade_char);
68     }
69
70     /* Step 7: Wireframe */
71     if (mode == RENDER_WIREFRAME || mode == RENDER_BOTH)
72     {
73         for (v = 0; v < face->num_verts; v++)
74         {
75             int next = (v + 1) % face->num_verts;
76             render_line(fb, px[v], py[v],
77                 px[next], py[next], WIRE_CHAR);
78         }
79     }

```

```
80     }  
81 }
```

Each step in the pipeline diagram corresponds to a clearly identifiable block in the code, demonstrating the direct mapping from the theory to the C implementation.

12 Reference

Rogers, David F. *Procedural Elements for Computer Graphics*. 2nd edition. McGraw-Hill, 1998.