Application and Math Concepts of Procedural Elements for Computer Graphics

ASCII 3D Renderer in Pure C

David F. Rogers, 2nd edition

Geovanne Gallinati

February 13, 2026

GitHub Repository

Contents

1 Homogeneous Coordinates **3**
1.1 Row Vector Convention 4
1.2 House Vertices 4

2 Matrix Operations **5**

3 3D Transformation Matrices **6**
3.1 Translation 6
3.2 Scaling 7
3.3 Rotation About the X Axis 7
3.4 Rotation About the Y Axis 8
3.5 Rotation About the Z Axis 8

4 Composite Transformations **9**
4.1 Rotation About an Arbitrary Point 9
4.2 Numerical Example 10

5 3D to 2D Projection **11**
5.1 Orthographic (Parallel) Projection 11
5.2 Perspective Projection 11
5.3 Viewport Mapping 12

6 Surface Normal and Cross Product **13**
6.1 Example: Front Face 14

7 Hidden Surface Removal **14**
7.1 Back-Face Culling 14
7.2 Painter's Algorithm 15

8 Line Rasterization: DDA Algorithm **16**

9 Polygon Filling: Scan-Line Algorithm **17**

10 Diffuse Lighting: Lambert's Law **18**
10.1 Mapping to ASCII 19

11 Complete Rendering Pipeline **20**
11.1 Code to Chapter Correspondence 22

12 Reference **22**

1 Homogeneous Coordinates

Mathematics

Rogers (Chapters 2 and 5) defines points in **homogeneous coordinates** as 4-component vectors. A 3D point (x, y, z) is represented as:

$$\mathbf{P} = \begin{bmatrix} x & y & z & w \end{bmatrix} \tag{1}$$

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

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

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:

```c
typedef struct {
    double m[4][4];
} Mat4;

typedef struct {
    double v[4];   /* v[0]=x, v[1]=y, v[2]=z, v[3]=w */
} Vec4;

typedef struct {
    double x, y, z;
} Vec3;
```

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

```c
Vec4 vec4_create(double x, double y, double z)
{
    Vec4 v;
    v.v[0] = x;
    v.v[1] = y;
    v.v[2] = z;
    v.v[3] = 1.0;   /* w = 1 by default */
    return v;
}

Vec4 vec4_create_w(double x, double y, double z, double w)
{
    Vec4 v;
    v.v[0] = x;   v.v[1] = y;
    v.v[2] = z;   v.v[3] = w;
    return v;
}
```

1.1 Row Vector Convention

Mathematics

Rogers uses row vectors with post-multiplication:

$$\mathbf{P}' = \mathbf{P} \cdot \mathbf{M} \tag{3}$$

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 $\mathbf{P}' = \mathbf{P} \cdot \mathbf{M}$:

```
void mat4_transform_point(const Mat4 *M, const Vec4 *p, Vec4 *result
    )
{
    Vec4 temp;
    int j, i;
    for (j = 0; j < 4; j++)
    {
        temp.v[j] = 0.0;
        for (i = 0; i < 4; i++)
        {
            temp.v[j] += p->v[i] * M->m[i][j];
        }
    }
    *result = temp;
}
```

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:

$$\mathbf{V}_0 = \begin{bmatrix} -1 & -1 & 1 & 1 \end{bmatrix} \quad \text{(front-left-bottom)} \tag{4}$$

$$\mathbf{V}_8 = \begin{bmatrix} 0 & 2 & 1 & 1 \end{bmatrix} \quad \text{(front roof peak)} \tag{5}$$

C Implementation `geometry.c: geometry_init_house`

```
void geometry_init_house(Model *model)
{
    /* Front face (z = +1) */
    model->vertices[0] = vec4_create(-1.0, -1.0,  1.0);
    model->vertices[1] = vec4_create( 1.0, -1.0,  1.0);
    model->vertices[2] = vec4_create( 1.0,  1.0,  1.0);
    model->vertices[3] = vec4_create(-1.0,  1.0,  1.0);

    /* Back face (z = -1) */
    model->vertices[4] = vec4_create(-1.0, -1.0, -1.0);
```

```
    model->vertices[5] = vec4_create( 1.0, -1.0, -1.0);
    model->vertices[6] = vec4_create( 1.0,  1.0, -1.0);
    model->vertices[7] = vec4_create(-1.0,  1.0, -1.0);

    /* Roof peaks */
    model->vertices[8] = vec4_create( 0.0,  2.0,  1.0);
    model->vertices[9] = vec4_create( 0.0,  2.0, -1.0);

    model->num_vertices = HOUSE_NUM_VERTICES;
    /* ... face definitions follow ... */
}
```

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

2 Matrix Operations

Mathematics

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

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

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

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

C Implementation `matrix.c`

```
void mat4_identity(Mat4 *m)
{
    memset(m->m, 0, sizeof(m->m));
    m->m[0][0] = 1.0;
    m->m[1][1] = 1.0;
    m->m[2][2] = 1.0;
    m->m[3][3] = 1.0;
}

void mat4_multiply(const Mat4 *A, const Mat4 *B, Mat4 *result)
{
    Mat4 temp;
    int i, j, k;
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < 4; j++)
        {
            temp.m[i][j] = 0.0;
            for (k = 0; k < 4; k++)
            {
                temp.m[i][j] += A->m[i][k] * B->m[k][j];
            }
```

```
            }
        }
        mat4_copy(&temp, result);
}
```

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

Rogers (Chapter 5) defines 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):

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

Verification:

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

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

C Implementation `transform.c: transform_translation`

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

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:

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

Result: $\mathbf{P}' = \begin{bmatrix} s_x x & s_y y & s_z z & 1 \end{bmatrix}$

C Implementation `transform.c: transform_scale`

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

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:

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

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

$$\begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 \end{bmatrix} \tag{12}$$

C Implementation `transform.c: transform_rotation_x`

```c
void transform_rotation_x(double theta, Mat4 *out)
{
    double c = cos(theta);
    double s = sin(theta);
    mat4_identity(out);
    out->m[1][1] =  c;    /* cos(theta) at [1][1] */
    out->m[1][2] =  s;    /* sin(theta) at [1][2] */
    out->m[2][1] = -s;    /* -sin(theta) at [2][1] */
```

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

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:

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

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

C Implementation `transform.c: transform_rotation_y`

```
void transform_rotation_y(double theta, Mat4 *out)
{
    double c = cos(theta);
    double s = sin(theta);
    mat4_identity(out);
    out->m[0][0] =  c;    /* cos(theta) at [0][0] */
    out->m[0][2] = -s;    /* -sin(theta) at [0][2] */
    out->m[2][0] =  s;    /* sin(theta) at [2][0] */
    out->m[2][2] =  c;    /* cos(theta) at [2][2] */
}
```

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:

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

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

C Implementation `transform.c: transform_rotation_z`

```c
void transform_rotation_z(double theta, Mat4 *out)
{
    double c = cos(theta);
    double s = sin(theta);
    mat4_identity(out);
    out->m[0][0] =  c;   /* cos(theta) at [0][0] */
    out->m[0][1] =  s;   /* sin(theta) at [0][1] */
    out->m[1][0] = -s;   /* -sin(theta) at [1][0] */
    out->m[1][1] =  c;   /* cos(theta) at [1][1] */
}
```

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

4 Composite Transformations

Mathematics

Rogers (Chapter 5) emphasizes that in the row vector convention, the **multiplication order** is the **application order**:

$$\mathbf{M}_{composite} = \mathbf{M}_1 \cdot \mathbf{M}_2 \cdot \mathbf{M}_3 \cdots \tag{15}$$

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**: $\mathbf{T}(c_x, c_y, c_z)$

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

C Implementation `transform.c: transform_rotate_y_around`

```c
void transform_rotate_y_around(double theta,
    double cx, double cy, double cz, Mat4 *out)
{
    Mat4 t1, r, t2, temp;

    /* Step 1: Translate center to origin */
    transform_translation(-cx, -cy, -cz, &t1);

    /* Step 2: Rotate about Y */
    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:

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

Since $c_x = c_z = 0$, the composite reduces to the pure rotation matrix.
Applying to vertex $\mathbf{V}_1 = [1, -1, 1, 1]$:

$$\mathbf{V}_1' = \begin{bmatrix} 1.366 & -1 & 0.366 & 1 \end{bmatrix} \tag{18}$$

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

Rogers (Chapter 5) presents two types of projection.

5.1 Orthographic (Parallel) Projection

Mathematics

The simplest projection, which discards the z coordinate:

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

Result: $x_{screen} = x, \quad y_{screen} = y$.
Parallel lines remain parallel; no depth effect.

C Implementation `projection.c`: orthographic branch

```c
void projection_init_ortho(Projection *proj, int vp_width,
                           int vp_height, double scale)
{
    proj->type = PROJ_ORTHOGRAPHIC;
    proj->d = 0.0;
    proj->vp_width = vp_width;
    proj->vp_height = vp_height;
    proj->scale = scale;
    proj->aspect_fix = 2.0;
}

/* Inside projection_project(): */
    else  /* orthographic */
    {
        x2d = x;    /* simply copy x */
        y2d = y;    /* simply copy y, discard z */
    }
```

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

5.2 Perspective Projection

Mathematics

Rogers defines the perspective projection with the eye at $(0, 0, d)$ and projection plane at $z = 0$:

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

After multiplication and **perspective division**:

$$x_{screen} = \frac{x}{1 - z/d}, \qquad y_{screen} = \frac{y}{1 - z/d} \tag{21}$$

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`

```c
int projection_project(const Projection *proj, const Vec4 *p3d,
                       double *screen_x, double *screen_y)
{
    double x = p3d->v[0];
    double y = p3d->v[1];
    double z = p3d->v[2];
    double x2d, y2d;

    if (proj->type == PROJ_PERSPECTIVE)
    {
        double w = 1.0 - z / proj->d;  /* w = 1 - z/d */
        if (w <= 0.001)
        {
            return 0;  /* singularity guard */
        }
        x2d = x / w;  /* perspective division */
        y2d = y / w;
    }
    else
    {
        x2d = x;
        y2d = y;
    }
    /* ... viewport mapping follows ... */
    return 1;
}
```

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

5.3 Viewport Mapping

Mathematics

After projection, coordinates are mapped to the terminal grid:

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

$$row = \frac{height}{2} - y_{2d} \cdot scale \tag{23}$$

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

```c
    *screen_x = x2d * proj->scale * proj->aspect_fix
              + proj->vp_width / 2.0;
    *screen_y = proj->vp_height / 2.0
```

```
                    - 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

Rogers (Chapter 7) defines the **face normal** 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):

$$\mathbf{E}_1 = \mathbf{V}_2 - \mathbf{V}_1, \qquad \mathbf{E}_2 = \mathbf{V}_3 - \mathbf{V}_1 \tag{24}$$

$$\mathbf{N} = \mathbf{E}_1 \times \mathbf{E}_2 \tag{25}$$

Component expansion:

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

C Implementation `matrix.c`: `vec3_cross` and `vec3_sub`

```
Vec3 vec3_sub(Vec3 a, Vec3 b)
{
    Vec3 result;
    result.x = a.x - b.x;     /* E = V2 - V1 */
    result.y = a.y - b.y;
    result.z = a.z - b.z;
    return result;
}

Vec3 vec3_cross(Vec3 a, Vec3 b)
{
    Vec3 result;
    result.x = a.y * b.z - a.z * b.y;    /* Nx */
    result.y = a.z * b.x - a.x * b.z;    /* Ny */
    result.z = a.x * b.y - a.y * b.x;    /* Nz */
    return result;
}
```

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:

```
Vec3 geometry_face_normal(const Model *model, int face_idx)
{
    const Face *face = &model->faces[face_idx];
    const Vec4 *v0 = &model->transformed[face->indices[0]];
```

```
    const Vec4 *v1 = &model->transformed[face->indices[1]];
    const Vec4 *v2 = &model->transformed[face->indices[2]];

    Vec3 p0 = vec3_create(v0->v[0], v0->v[1], v0->v[2]);
    Vec3 p1 = vec3_create(v1->v[0], v1->v[1], v1->v[2]);
    Vec3 p2 = vec3_create(v2->v[0], v2->v[1], v2->v[2]);

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

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

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

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

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

7 Hidden Surface Removal

Rogers (Chapter 7) presents algorithms for determining which surfaces are visible.

7.1 Back-Face Culling

Mathematics

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

$$\text{Face is } \textbf{visible} \text{ if } N_z > 0 \tag{29}$$

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

```
    for (i = 0; i < model->num_faces; i++)
    {
        Vec3 normal = geometry_face_normal(model, i);
        if (normal.z > 0)    /* Nz > 0 => visible */
        {
            visible_faces[num_visible].face_idx = i;
            visible_faces[num_visible].avg_z =
```

```
                geometry_face_avg_z(model, i);
            num_visible++;
        }
    }
```

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:

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

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:

```c
double geometry_face_avg_z(const Model *model, int face_idx)
{
    const Face *face = &model->faces[face_idx];
    double sum_z = 0;
    int i;
    for (i = 0; i < face->num_verts; i++)
    {
        sum_z += model->transformed[face->indices[i]].v[2];
    }
    return sum_z / face->num_verts;
}
```

Sorting by depth (ascending = farthest first):

```c
static int compare_face_depth(const void *a, const void *b)
{
    const FaceDepth *fa = (const FaceDepth *)a;
    const FaceDepth *fb = (const FaceDepth *)b;
    if (fa->avg_z < fb->avg_z) return -1;
    if (fa->avg_z > fb->avg_z) return  1;
    return 0;
}

    /* In pipeline_render_frame(): */
    qsort(visible_faces, num_visible,
          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

Rogers (Chapter 2) presents the **DDA (Digital Differential Analyzer)**.
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:

$$x_{inc} = \frac{\Delta x}{N}, \qquad y_{inc} = \frac{\Delta y}{N} \tag{31}$$

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

$$x_i = x_0 + i \cdot x_{inc}, \qquad y_i = y_0 + i \cdot y_{inc} \tag{32}$$

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

C Implementation `render.c`: `render_line`

```c
void render_line(Framebuffer *fb, double x0, double y0,
                 double x1, double y1, char c)
{
    double dx = x1 - x0;                          /* delta x */
    double dy = y1 - y0;                          /* delta y */
    int abs_dx = (int)(fabs(dx) + 0.5);
    int abs_dy = (int)(fabs(dy) + 0.5);
    int steps = abs_dx > abs_dy ? abs_dx : abs_dy; /* N */

    if (steps == 0)
    {
        fb_set(fb, (int)(x0 + 0.5), (int)(y0 + 0.5), c);
        return;
    }

    double x_inc = dx / (double)steps;    /* x_inc = dx/N */
    double y_inc = dy / (double)steps;    /* y_inc = dy/N */
    double x = x0, y = y0;
    int i;

    for (i = 0; i <= steps; i++)
    {
        fb_set(fb, (int)(x + 0.5), (int)(y + 0.5), c);
        x += x_inc;        /* x_i = x0 + i * x_inc */
        y += y_inc;        /* y_i = y0 + i * y_inc */
    }
}
```

The code follows the DDA algorithm step by step: `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

Rogers (Chapters 3–4) describes the **scan-line algorithm**:
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):

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

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`

```c
void render_fill_polygon(Framebuffer *fb,
    const double *verts_x, const double *verts_y,
    int n, char c)
{
    /* ... compute ymin, ymax ... */

    for (y = ymin; y <= ymax; y++)  /* each scan line */
    {
        num_intersections = 0;
        for (i = 0; i < n; i++)
        {
            j = (i + 1) % n;  /* next vertex (wraps) */
            double y1 = verts_y[i], y2 = verts_y[j];
            double x1 = verts_x[i], x2 = verts_x[j];

            /* Half-open interval test */
            if ((y1 <= y && y2 > y) || (y2 <= y && y1 > y))
            {
                /* x_int = x1 + (y-y1)/(y2-y1) * (x2-x1) */
                double t = ((double)y - y1) / (y2 - y1);
                double x_int = x1 + t * (x2 - x1);
                intersections[num_intersections++] = x_int;
            }
        }

        /* Sort intersections by x (insertion sort) */
        for (i = 1; i < num_intersections; i++) { /* ... */ }

        /* Fill between pairs */
        for (i = 0; i + 1 < num_intersections; i += 2)
        {
            int xstart = (int)(intersections[i] + 0.5);
            int xend = (int)(intersections[i + 1] + 0.5);
            int col;
```

```
            for (col = xstart; col <= xend; col++)
                fb->buffer[y][col] = c;
        }
    }
}
```

The half-open interval condition `(y1 <= y && y2 > y) || (y2 <= 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

Rogers (Chapter 8) presents **Lambertian diffuse lighting**.

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

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

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

$$\hat{\mathbf{N}} = \frac{\mathbf{N}}{|\mathbf{N}|}, \qquad |\mathbf{N}| = \sqrt{N_x^2 + N_y^2 + N_z^2} \tag{35}$$

With **ambient light** I_a:

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

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

```
double vec3_length(Vec3 v)
{
    return sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
}

Vec3 vec3_normalize(Vec3 v)
{
    double len = vec3_length(v);
    Vec3 result;
    if (len > 1e-10)
    {
        result.x = v.x / len;
        result.y = v.y / len;
        result.z = v.z / len;
    }
    else
    {
        result.x = result.y = result.z = 0.0;
    }
    return result;
```

```
}

double vec3_dot(Vec3 a, Vec3 b)
{
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

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

C Implementation `render.c: render_shade`

The complete Lambert shading computation:

```
char render_shade(Vec3 normal, Vec3 light_dir)
{
    Vec3 n = vec3_normalize(normal);      /* N_hat */
    Vec3 l = vec3_normalize(light_dir);   /* L_hat */
    double dot = vec3_dot(n, l);          /* N . L */
    if (dot < 0.0) dot = 0.0;            /* max(0, ...) */

    double ambient = 0.15;    /* I_a = 0.15 */
    double diffuse = 0.85;    /* I_d = 0.85 */
    double intensity = ambient + diffuse * dot;
    if (intensity > 1.0) intensity = 1.0;

    int idx = (int)(intensity * (ASCII_RAMP_LEN - 1) + 0.5);
    if (idx < 0) idx = 0;
    if (idx >= ASCII_RAMP_LEN) idx = ASCII_RAMP_LEN - 1;

    return ascii_ramp[idx];
}
```

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:

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

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

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

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

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

The ASCII ramp definition and light direction:

```c
/* render.h */
#define ASCII_RAMP " .,:;=+*#%@$"
#define ASCII_RAMP_LEN 12

/* pipeline.c */
static const Vec3 LIGHT_DIR = {0.5, 0.8, 0.6};
```

The ramp index `(int)(intensity * 11 + 0.5)` implements Equation 37.

11 Complete Rendering Pipeline

The pipeline follows the order described by Rogers across the chapters of the book:

1. **Define model:** vertices and faces
2. **Transform vertices:** $\mathbf{V}' = \mathbf{V} \cdot \mathbf{M}$
3. **Compute normals:** $\mathbf{N} = \mathbf{E}_1 \times \mathbf{E}_2$
4. **Back-face culling:** $N_z > 0 \implies$ 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:

```c
void pipeline_render_frame(Model *model, Framebuffer *fb,
    const Projection *proj,
    double angle_y, double angle_x, RenderMode mode)
{
    Mat4 rot_y, rot_x, composite;
    int i;
    FaceDepth visible_faces[HOUSE_NUM_FACES];
    int num_visible = 0;

    /* --- Step 2: Build composite transformation --- */
    transform_rotate_y_around(angle_y,
        model->center_x, model->center_y, model->center_z,
        &rot_y);
    transform_rotate_x_around(angle_x,
        model->center_x, model->center_y, model->center_z,
        &rot_x);
    mat4_multiply(&rot_y, &rot_x, &composite);

    /* Transform all vertices: V' = V * M */
    for (i = 0; i < model->num_vertices; i++)
    {
```

```
        mat4_transform_point(&composite,
            &model->vertices[i], &model->transformed[i]);
    }

    /* --- Steps 3-4: Normal + Back-face culling --- */
    for (i = 0; i < model->num_faces; i++)
    {
        Vec3 normal = geometry_face_normal(model, i);
        if (normal.z > 0)
        {
            visible_faces[num_visible].face_idx = i;
            visible_faces[num_visible].avg_z =
                geometry_face_avg_z(model, i);
            num_visible++;
        }
    }

    /* --- Step 5: Painter's sort --- */
    qsort(visible_faces, num_visible,
          sizeof(FaceDepth), compare_face_depth);

    /* --- Steps 6-8: Project, Rasterize, Shade --- */
    for (i = 0; i < num_visible; i++)
    {
        int fi = visible_faces[i].face_idx;
        const Face *face = &model->faces[fi];
        double px[MAX_FACE_VERTS], py[MAX_FACE_VERTS];
        int v, valid = 1;

        /* Step 6: Project 3D -> 2D */
        for (v = 0; v < face->num_verts; v++)
        {
            int idx = face->indices[v];
            if (!projection_project(proj,
                &model->transformed[idx], &px[v], &py[v]))
            { valid = 0; break; }
        }
        if (!valid) continue;

        /* Step 7-8: Fill with shading */
        if (mode == RENDER_FILLED || mode == RENDER_BOTH)
        {
            Vec3 normal = geometry_face_normal(model, fi);
            char shade_char = render_shade(normal, LIGHT_DIR);
            render_fill_polygon(fb, px, py,
                face->num_verts, shade_char);
        }

        /* Step 7: Wireframe */
        if (mode == RENDER_WIREFRAME || mode == RENDER_BOTH)
        {
            for (v = 0; v < face->num_verts; v++)
            {
                int next = (v + 1) % face->num_verts;
                render_line(fb, px[v], py[v],
                    px[next], py[next], WIRE_CHAR);
            }
        }
```

```
        }
    }
```

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

11.1 Code to Chapter Correspondence

Stage	Algorithm	Rogers	Source File
Homogeneous coordinates	$[x\ y\ z\ w]$, 4×4 matrices	Ch. 2, 5	`matrix.c/h`
3D Transformations	T, S, R_x, R_y, R_z	Ch. 5	`transform.c/h`
Model definition	Vertices + CCW faces	Ch. 5	`geometry.c/h`
Projection	Orthographic / Perspective	Ch. 5	`projection.c/h`
Line rasterization	DDA	Ch. 2	`render.c`
Polygon filling	Scan-line fill	Ch. 3–4	`render.c`
Hidden surfaces	Back-face culling + Painter's	Ch. 7	`pipeline.c`
Lighting	Lambertian diffuse	Ch. 8	`render.c`
Pipeline orchestration	All stages combined	All	`pipeline.c`

12 Reference

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