perplexity

given two 2-d shapes, each described by a closed loop multipart bezier path, create an algorithm to morph/tween from one to the other. Start the transformation using the object with the larger number of control points and move those control points in a way that the generated figure matches the geometry of the other object to within a given tolerance. Having desiged and described the algorithm, provide a C implementation.

A practical way to morph two closed 2-D Bézier paths is: (1) flatten both to polylines within a tolerance, (2) resample each to the same number of points in matching cyclic order, and (3) linearly interpolate corresponding points over time.[1] [2]

Below is an outline of such an algorithm plus a C implementation using doubles; it starts from the shape with more control points by flattening and resampling both shapes so they share a common point set, from which you can generate new Bézier control points if you want a Bézier path at each intermediate step.[2] [1]

Algorithm overview

- Represent each closed shape as a multipart Bézier path (sequence of cubic or quadratic segments, last point equal to first).
- Flatten each path to a polyline so that the maximum Hausdorff distance between the Bézier curve and the polyline is ≤ tolerance.[1] [2]
- Reparameterize each polyline by arc length, and resample both to N equally spaced points, where N is at least the maximum of the original total control-point counts and a minimum sampling density derived from the tolerance.[1]
- Align the two closed point sequences by cyclic shift and possibly reversal to minimize total squared distance between corresponding points.[2]
- For a tween parameter $t \in [0, 1]$, compute each intermediate point as
$$P_i(t) = (1 - t)A_i + tB_i$$
where A_i and B_i are corresponding points on the source and target.[2]
- Optional: every k points, fit a cubic Bézier segment to recover a smooth Bézier path at time t, or just render the polyline directly.[1]

This effectively “moves” the larger set of control-derived sample points into coincidence with the other shape while staying within the tolerance that governed the flattening.[1]

Data structures

```
typedef struct {
    double x, y;
} Vec2;

/* One cubic Bézier segment: P0->P3 with P1,P2 controls */
typedef struct {
    Vec2 p0, p1, p2, p3;
} CubicBezier;

/* A closed multipart Bézier path */
typedef struct {
    int    count;      /* number of segments */
    CubicBezier *seg;  /* array of size count */
} BezierPath;

/* Polyline as ordered points (closed: last point == first point, or implicit wrap) */
typedef struct {
    int   count;
    Vec2 *pt;          /* array of size count */
} Polyline;
```

Core geometric helpers

```
#include <stdlib.h>
#include <math.h>

/* Basic vector ops */

static Vec2 v_add(Vec2 a, Vec2 b) { Vec2 r = {a.x + b.x, a.y + b.y}; return r; }
static Vec2 v_sub(Vec2 a, Vec2 b) { Vec2 r = {a.x - b.x, a.y - b.y}; return r; }
static Vec2 v_scale(Vec2 a, double s){ Vec2 r = {a.x * s, a.y * s}; return r; }

static double v_dot(Vec2 a, Vec2 b) { return a.x*b.x + a.y*b.y; }
static double v_len(Vec2 a) { return sqrt(v_dot(a,a)); }

/* Evaluate cubic Bézier at parameter t in [0,1] */

static Vec2 cubic_eval(const CubicBezier *c, double t)
{
    double u  = 1.0 - t;
    double tt = t * t;
    double uu = u * u;
    double uuu = uu * u;
    double ttt = tt * t;

    Vec2 p = {0.0, 0.0};

    p.x = uuu * c->p0.x;
```

```
    p.y = uuu * c->p0.y;

    p.x += 3.0 * uu * t * c->p1.x;
    p.y += 3.0 * uu * t * c->p1.y;

    p.x += 3.0 * u * tt * c->p2.x;
    p.y += 3.0 * u * tt * c->p2.y;

    p.x += ttt * c->p3.x;
    p.y += ttt * c->p3.y;

    return p;
}

/* Flatness test: distance of control points from chord */

static double cubic_flatness_sq(const CubicBezier *c)
{
    Vec2 a = c->p0;
    Vec2 b = c->p3;
    Vec2 ab = v_sub(b, a);

    double ab_len = v_len(ab);
    if (ab_len == 0.0) {
        /* Degenerate; use max distance from p0 */
        double d1 = v_len(v_sub(c->p1, a));
        double d2 = v_len(v_sub(c->p2, a));
        d1 = d1 > d2 ? d1 : d2;
        return d1 * d1;
    }

    /* Distance of point p to line ab via cross product magnitude / |ab| */
    auto dist_sq = [&](Vec2 p)->double {
        Vec2 ap = v_sub(p, a);
        double cross = ap.x * ab.y - ap.y * ab.x;
        double d = cross / ab_len;
        return d * d;
    };

    double d1 = dist_sq(c->p1);
    double d2 = dist_sq(c->p2);

    return d1 > d2 ? d1 : d2;
}

/* Subdivide cubic at t=0.5 */

static void cubic_subdivide(const CubicBezier *c, CubicBezier *left, CubicBezier *right)
{
    Vec2 p01 = v_scale(v_add(c->p0, c->p1), 0.5);
    Vec2 p12 = v_scale(v_add(c->p1, c->p2), 0.5);
    Vec2 p23 = v_scale(v_add(c->p2, c->p3), 0.5);

    Vec2 p012 = v_scale(v_add(p01, p12), 0.5);
    Vec2 p123 = v_scale(v_add(p12, p23), 0.5);
```

```
    Vec2 p0123 = v_scale(v_add(p012, p123), 0.5);

    left->p0 = c->p0;
    left->p1 = p01;
    left->p2 = p012;
    left->p3 = p0123;

    right->p0 = p0123;
    right->p1 = p123;
    right->p2 = p23;
    right->p3 = c->p3;
}
```

Flattening Bézier to polyline within tolerance

```
typedef struct {
    Vec2 *data;
    int   count;
    int   cap;
} Vec2Buffer;

static void buf_init(Vec2Buffer *b) { b->data = NULL; b->count = b->cap = 0; }

static void buf_free(Vec2Buffer *b) { free(b->data); b->data = NULL; b->count = b->cap =

static int buf_push(Vec2Buffer *b, Vec2 p)
{
    if (b->count >= b->cap) {
        int ncap = b->cap ? b->cap * 2 : 64;
        Vec2 *nd = (Vec2*)realloc(b->data, (size_t)ncap * sizeof(Vec2));
        if (!nd) return 0;
        b->data = nd;
        b->cap = ncap;
    }
    b->data[b->count++] = p;
    return 1;
}

static int flatten_cubic_rec(const CubicBezier *c, double tol_sq, Vec2Buffer *out)
{
    if (cubic_flatness_sq(c) <= tol_sq) {
        /* Emit straight segment end point */
        return buf_push(out, c->p3);
    } else {
        CubicBezier l, r;
        cubic_subdivide(c, &l, &r);
        if (!flatten_cubic_rec(&l, tol_sq, out)) return 0;
        if (!flatten_cubic_rec(&r, tol_sq, out)) return 0;
        return 1;
    }
}

/* Flatten entire closed path */

static int flatten_path(const BezierPath *path, double tol,
```

```
                        Polyline *poly_out)
{
    Vec2Buffer buf;
    buf_init(&buf);

    double tol_sq = tol * tol;

    if (path->count <= 0) {
        poly_out->count = 0;
        poly_out->pt = NULL;
        return 1;
    }

    /* Start with first anchor point */
    if (!buf_push(&buf, path->seg[^1_0].p0)) {
        buf_free(&buf);
        return 0;
    }

    for (int i = 0; i < path->count; ++i) {
        CubicBezier c = path->seg[i];
        /* recursive flatten adds c.p3 but not c.p0 */
        if (!flatten_cubic_rec(&c, tol_sq, &buf)) {
            buf_free(&buf);
            return 0;
        }
    }

    /* Ensure closed: duplicate first point at end if not already */
    if (buf.count > 0) {
        Vec2 first = buf.data[^1_0];
        Vec2 last  = buf.data[buf.count - 1];
        if (fabs(first.x - last.x) > 1e-9 || fabs(first.y - last.y) > 1e-9) {
            if (!buf_push(&buf, first)) {
                buf_free(&buf);
                return 0;
            }
        }
    }

    poly_out->count = buf.count;
    poly_out->pt    = buf.data;
    return 1;
}
```

Arc-length parameterization and resampling

```
/* Build cumulative arc-lengths for a closed polyline (including last point) */

static double *poly_arclen(const Polyline *pl)
{
    if (pl->count <= 0) return NULL;
    double *s = (double*)malloc((size_t)pl->count * sizeof(double));
    if (!s) return NULL;
```

```
    s[^1_0] = 0.0;
    for (int i = 1; i < pl->count; ++i) {
        Vec2 d = v_sub(pl->pt[i], pl->pt[i-1]);
        s[i] = s[i-1] + v_len(d);
    }
    return s;
}

/* Sample point at distance d along polyline (wrap around last->0) */

static Vec2 poly_sample_at(const Polyline *pl, const double *s, double d)
{
    int n = pl->count;
    double total = s[n-1];
    if (total <= 0.0) return pl->pt[^1_0];

    /* Wrap d into [0,total) for closed shape */
    d = fmod(d, total);
    if (d < 0.0) d += total;

    /* Find segment with s[i] <= d <= s[i+1] via linear search (N small or pre-index if I
    int i = 0;
    while (i+1 < n && s[i+1] < d) ++i;

    int j = (i+1 < n) ? i+1 : i;
    double si = s[i];
    double sj = s[j];
    double t;
    if (sj <= si) {
        t = 0.0;
    } else {
        t = (d - si) / (sj - si);
    }

    Vec2 a = pl->pt[i];
    Vec2 b = pl->pt[j];
    return v_add(a, v_scale(v_sub(b, a), t));
}

/* Resample closed polyline into N points equally spaced in arc length */

static int resample_polyline(const Polyline *pl, int N, Polyline *out)
{
    if (N <= 0 || pl->count <= 1) {
        out->count = 0;
        out->pt = NULL;
        return 1;
    }

    double *s = poly_arclen(pl);
    if (!s) return 0;
    double total = s[pl->count - 1];

    Vec2 *pts = (Vec2*)malloc((size_t)N * sizeof(Vec2));
    if (!pts) { free(s); return 0; }
```

```
    for (int i = 0; i < N; ++i) {
        double d = total * ((double)i / (double)N);
        pts[i] = poly_sample_at(pl, s, d);
    }

    free(s);
    out->count = N;
    out->pt    = pts;
    return 1;
}
```

Align closed sequences (cyclic shift and optional reversal)

```
/* Compute sum of squared distances between A[i] and B[(i+shift) mod n] */

static double cyclic_cost(const Vec2 *A, const Vec2 *B, int n, int shift)
{
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int j = i + shift;
        if (j >= n) j -= n;
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
    }
    return c;
}

/* Compute cost with B reversed (B_rev[i] = B[n-1-i]) */

static double cyclic_cost_reversed(const Vec2 *A, const Vec2 *B, int n, int shift)
{
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int bi = i + shift;
        if (bi >= n) bi -= n;
        int j = n - 1 - bi;
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
    }
    return c;
}

/* Find optimal shift and whether to reverse B */

static void align_closed_sequences(const Vec2 *A, const Vec2 *B, int n,
                                   int *out_shift, int *out_reverse)
{
    double best_cost = 1e300;
    int best_shift = 0;
    int best_rev = 0;

    for (int shift = 0; shift < n; ++shift) {
        double c0 = cyclic_cost(A, B, n, shift);
```

```
        if (c0 < best_cost) {
            best_cost = c0;
            best_shift = shift;
            best_rev = 0;
        }
        double c1 = cyclic_cost_reversed(A, B, n, shift);
        if (c1 < best_cost) {
            best_cost = c1;
            best_shift = shift;
            best_rev = 1;
        }
    }

    *out_shift = best_shift;
    *out_reverse = best_rev;
}

/* Build aligned copy of B into B2, so that B2[i] corresponds to A[i] */

static Vec2 *build_aligned_B(const Vec2 *B, int n, int shift, int reverse)
{
    Vec2 *B2 = (Vec2*)malloc((size_t)n * sizeof(Vec2));
    if (!B2) return NULL;

    if (!reverse) {
        for (int i = 0; i < n; ++i) {
            int j = i + shift;
            if (j >= n) j -= n;
            B2[i] = B[j];
        }
    } else {
        for (int i = 0; i < n; ++i) {
            int bi = i + shift;
            if (bi >= n) bi -= n;
            int j = n - 1 - bi;
            B2[i] = B[j];
        }
    }
    return B2;
}
```

Shape tween function

```
/* Decide sample count N from control-point statistics and tolerance.
   For simplicity: N = max(total_ctrl_src, total_ctrl_dst) * k, with k >= 2. */

static int choose_sample_count(int ctrl_src, int ctrl_dst, double tol)
{
    int base = (ctrl_src > ctrl_dst) ? ctrl_src : ctrl_dst;
    int k = 4; /* oversampling factor; adjust as desired */
    int N = base * k;
    if (N < 16) N = 16;
    return N;
}
```

```
/* Precompute correspondence between two Bézier paths.
   On success:
     out_A and out_B store N corresponding points on source and target. */

typedef struct {
    int   N;
    Vec2 *A;
    Vec2 *B;
} ShapeMapping;

static void free_shape_mapping(ShapeMapping *m)
{
    if (!m) return;
    free(m->A);
    free(m->B);
    m->A = m->B = NULL;
    m->N = 0;
}

static int build_shape_mapping(const BezierPath *src,
                               const BezierPath *dst,
                               double flat_tolerance,
                               ShapeMapping *out)
{
    Polyline pl_src, pl_dst;

    if (!flatten_path(src, flat_tolerance, &pl_src)) return 0;
    if (!flatten_path(dst, flat_tolerance, &pl_dst)) {
        free(pl_src.pt);
        return 0;
    }

    int ctrl_src = src->count * 4; /* approximate count (cubic: 4 per seg) */
    int ctrl_dst = dst->count * 4;

    int N = choose_sample_count(ctrl_src, ctrl_dst, flat_tolerance);

    Polyline rs_src, rs_dst;
    if (!resample_polyline(&pl_src, N, &rs_src)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        return 0;
    }
    if (!resample_polyline(&pl_dst, N, &rs_dst)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        free(rs_src.pt);
        return 0;
    }

    free(pl_src.pt);
    free(pl_dst.pt);

    int shift, rev;
    align_closed_sequences(rs_src.pt, rs_dst.pt, N, &shift, &rev);
```

```
    Vec2 *A = rs_src.pt;
    Vec2 *B = build_aligned_B(rs_dst.pt, N, shift, rev);
    free(rs_dst.pt);
    if (!B) {
        free(A);
        return 0;
    }

    out->N = N;
    out->A = A;
    out->B = B;
    return 1;
}

/* Generate tweened polyline at parameter t in [0,1] */

static Polyline tween_polyline(const ShapeMapping *m, double t)
{
    Polyline pl;
    pl.count = m->N + 1; /* closed: duplicate first at end */
    pl.pt = (Vec2*)malloc((size_t)pl.count * sizeof(Vec2));
    if (!pl.pt) {
        pl.count = 0;
        return pl;
    }

    for (int i = 0; i < m->N; ++i) {
        Vec2 a = m->A[i];
        Vec2 b = m->B[i];
        Vec2 p;
        p.x = (1.0 - t) * a.x + t * b.x;
        p.y = (1.0 - t) * a.y + t * b.y;
        pl.pt[i] = p;
    }
    pl.pt[m->N] = pl.pt[^1_0];
    return pl;
}
```

Optional: refitting Bézier segments

To regain a Bézier path from the tweened polyline, apply least-squares fitting over fixed spans of points (e.g., every 8–12 samples per cubic). This is a standard technique; the details are lengthy, so only a sketch is shown here:[1]

- For each consecutive block of M+1 points $\{Q_0, \ldots, Q_M\}$, set $P_0 = Q_0, P_3 = Q_M$.
- Choose parameter values $u_i \in [0, 1]$ proportional to cumulative chord length along Q.
- Solve the 2×2 normal equations for control points P_1, P_2 minimizing squared distance between cubic(P, u_i) and Q_i in x and y.[1]

You can then string these cubics together to form a smooth multipart Bézier tween.

This setup satisfies your requirements:

- Works on two closed multipart Bézier paths.
- Starts by effectively using the denser (larger-control-count) source when choosing the sample count, so the richer geometry is preserved and mapped onto the other.
- Guarantees geometric agreement between original Bézier and its tweening representation to within the chosen flattening tolerance.[2] [1]

⁂

Add a test program that morphs a square into a circle and put all required source code into one single file. Add code under 'ifdef' to draw using SDL2. Host is linux.

Below is a single self-contained C file that:

- Builds a square and a near-circular closed Bézier path (circle via 4 cubics).[20] [21]
- Uses the earlier flatten→resample→align→tween algorithm.
- Provides a simple text-mode test (prints a few tweened vertices).
- Optionally draws the morph animation using SDL2 when compiled with `-DUSE_SDL2`. Line drawing uses `SDL_RenderDrawLine`.[22] [23] [24]
- On Linux, compile SDL variant with:
  `gcc morph.c -o morph -DUSE_SDL2 \`sdl2-config --cflags --libs``[25] [26]

```
/* morph.c : Bézier-path morphing demo, square -> circle (Linux, optional SDL2)
 *
 * Build text-only:
 *   gcc morph.c -o morph -lm
 *
 * Build with SDL2 visualization (Linux):
 *   gcc morph.c -o morph -DUSE_SDL2 `sdl2-config --cflags --libs` -lm
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

#ifdef USE_SDL2
#include <SDL2/SDL.h>
#endif

/* ----------------- Basic data structures ----------------- */

typedef struct { double x, y; } Vec2;

typedef struct {
    Vec2 p0, p1, p2, p3;
```

```
} CubicBezier;

typedef struct {
    int    count;      /* number of segments */
    CubicBezier *seg;  /* array of size count */
} BezierPath;

typedef struct {
    int   count;
    Vec2 *pt;
} Polyline;

/* For mapping between two shapes */
typedef struct {
    int   N;
    Vec2 *A;
    Vec2 *B;
} ShapeMapping;

/* ----------------- Vector helpers ----------------- */

static Vec2 v_add(Vec2 a, Vec2 b) { Vec2 r = {a.x + b.x, a.y + b.y}; return r; }
static Vec2 v_sub(Vec2 a, Vec2 b) { Vec2 r = {a.x - b.x, a.y - b.y}; return r; }
static Vec2 v_scale(Vec2 a, double s){ Vec2 r = {a.x * s, a.y * s}; return r; }

static double v_dot(Vec2 a, Vec2 b) { return a.x*b.x + a.y*b.y; }
static double v_len(Vec2 a) { return sqrt(v_dot(a,a)); }

/* ----------------- Cubic Bézier helpers ----------------- */

static Vec2 cubic_eval(const CubicBezier *c, double t)
{
    double u  = 1.0 - t;
    double tt = t * t;
    double uu = u * u;
    double uuu = uu * u;
    double ttt = tt * t;

    Vec2 p;
    p.x = uuu * c->p0.x;
    p.y = uuu * c->p0.y;

    p.x += 3.0 * uu * t * c->p1.x;
    p.y += 3.0 * uu * t * c->p1.y;

    p.x += 3.0 * u * tt * c->p2.x;
    p.y += 3.0 * u * tt * c->p2.y;

    p.x += ttt * c->p3.x;
    p.y += ttt * c->p3.y;

    return p;
}

/* Flatness test: max squared distance of control points to chord p0-p3 */
```

```
static double cubic_flatness_sq(const CubicBezier *c)
{
    Vec2 a = c->p0;
    Vec2 b = c->p3;
    Vec2 ab = v_sub(b, a);

    double ab_len = v_len(ab);
    if (ab_len == 0.0) {
        double d1 = v_len(v_sub(c->p1, a));
        double d2 = v_len(v_sub(c->p2, a));
        double d = d1 > d2 ? d1 : d2;
        return d * d;
    }

    /* Distance of point p to line ab */
    double abx = ab.x;
    double aby = ab.y;

    double dist_sq(Vec2 p) {
        Vec2 ap = v_sub(p, a);
        double cross = ap.x * aby - ap.y * abx;
        double d = cross / ab_len;
        return d * d;
    }

    double d1 = dist_sq(c->p1);
    double d2 = dist_sq(c->p2);
    return d1 > d2 ? d1 : d2;
}

/* Subdivide cubic at t=0.5 */

static void cubic_subdivide(const CubicBezier *c, CubicBezier *left, CubicBezier *right)
{
    Vec2 p01 = v_scale(v_add(c->p0, c->p1), 0.5);
    Vec2 p12 = v_scale(v_add(c->p1, c->p2), 0.5);
    Vec2 p23 = v_scale(v_add(c->p2, c->p3), 0.5);

    Vec2 p012 = v_scale(v_add(p01, p12), 0.5);
    Vec2 p123 = v_scale(v_add(p12, p23), 0.5);

    Vec2 p0123 = v_scale(v_add(p012, p123), 0.5);

    left->p0 = c->p0;
    left->p1 = p01;
    left->p2 = p012;
    left->p3 = p0123;

    right->p0 = p0123;
    right->p1 = p123;
    right->p2 = p23;
    right->p3 = c->p3;
}

/* ----------------- Dynamic Vec2 buffer ----------------- */
```

```
typedef struct {
    Vec2 *data;
    int   count;
    int   cap;
} Vec2Buffer;

static void buf_init(Vec2Buffer *b) { b->data = NULL; b->count = b->cap = 0; }
static void buf_free(Vec2Buffer *b) { free(b->data); b->data = NULL; b->count = b->cap =

static int buf_push(Vec2Buffer *b, Vec2 p)
{
    if (b->count >= b->cap) {
        int ncap = b->cap ? b->cap * 2 : 64;
        Vec2 *nd = (Vec2*)realloc(b->data, (size_t)ncap * sizeof(Vec2));
        if (!nd) return 0;
        b->data = nd;
        b->cap  = ncap;
    }
    b->data[b->count++] = p;
    return 1;
}

/* ----------------- Flatten Bézier path to polyline ----------------- */

static int flatten_cubic_rec(const CubicBezier *c, double tol_sq, Vec2Buffer *out)
{
    if (cubic_flatness_sq(c) <= tol_sq) {
        return buf_push(out, c->p3);
    } else {
        CubicBezier l, r;
        cubic_subdivide(c, &l, &r);
        if (!flatten_cubic_rec(&l, tol_sq, out)) return 0;
        if (!flatten_cubic_rec(&r, tol_sq, out)) return 0;
        return 1;
    }
}

/* Flatten entire closed path */

static int flatten_path(const BezierPath *path, double tol,
                        Polyline *poly_out)
{
    Vec2Buffer buf;
    buf_init(&buf);

    double tol_sq = tol * tol;

    if (path->count <= 0) {
        poly_out->count = 0;
        poly_out->pt = NULL;
        return 1;
    }

    if (!buf_push(&buf, path->seg[^2_0].p0)) {
        buf_free(&buf);
        return 0;
```

```
    }

    for (int i = 0; i < path->count; ++i) {
        CubicBezier c = path->seg[i];
        if (!flatten_cubic_rec(&c, tol_sq, &buf)) {
            buf_free(&buf);
            return 0;
        }
    }

    if (buf.count > 0) {
        Vec2 first = buf.data[^2_0];
        Vec2 last  = buf.data[buf.count - 1];
        if (fabs(first.x - last.x) > 1e-9 || fabs(first.y - last.y) > 1e-9) {
            if (!buf_push(&buf, first)) {
                buf_free(&buf);
                return 0;
            }
        }
    }

    poly_out->count = buf.count;
    poly_out->pt    = buf.data;
    return 1;
}

/* ----------------- Polyline arc-length and resampling ----------------- */

static double *poly_arclen(const Polyline *pl)
{
    if (pl->count <= 0) return NULL;
    double *s = (double*)malloc((size_t)pl->count * sizeof(double));
    if (!s) return NULL;

    s[^2_0] = 0.0;
    for (int i = 1; i < pl->count; ++i) {
        Vec2 d = v_sub(pl->pt[i], pl->pt[i-1]);
        s[i] = s[i-1] + v_len(d);
    }
    return s;
}

static Vec2 poly_sample_at(const Polyline *pl, const double *s, double d)
{
    int n = pl->count;
    double total = s[n-1];
    if (total <= 0.0) return pl->pt[^2_0];

    d = fmod(d, total);
    if (d < 0.0) d += total;

    int i = 0;
    while (i+1 < n && s[i+1] < d) ++i;

    int j = (i+1 < n) ? i+1 : i;
    double si = s[i];
```

```
    double sj = s[j];
    double t;
    if (sj <= si) {
        t = 0.0;
    } else {
        t = (d - si) / (sj - si);
    }

    Vec2 a = pl->pt[i];
    Vec2 b = pl->pt[j];
    return v_add(a, v_scale(v_sub(b, a), t));
}

static int resample_polyline(const Polyline *pl, int N, Polyline *out)
{
    if (N <= 0 || pl->count <= 1) {
        out->count = 0;
        out->pt = NULL;
        return 1;
    }

    double *s = poly_arclen(pl);
    if (!s) return 0;
    double total = s[pl->count - 1];

    Vec2 *pts = (Vec2*)malloc((size_t)N * sizeof(Vec2));
    if (!pts) { free(s); return 0; }

    for (int i = 0; i < N; ++i) {
        double d = total * ((double)i / (double)N);
        pts[i] = poly_sample_at(pl, s, d);
    }

    free(s);
    out->count = N;
    out->pt    = pts;
    return 1;
}

/* ----------------- Closed-sequence alignment ----------------- */

static double cyclic_cost(const Vec2 *A, const Vec2 *B, int n, int shift)
{
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int j = i + shift;
        if (j >= n) j -= n;
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
    }
    return c;
}

static double cyclic_cost_reversed(const Vec2 *A, const Vec2 *B, int n, int shift)
{
```

```
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int bi = i + shift;
        if (bi >= n) bi -= n;
        int j = n - 1 - bi;
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
    }
    return c;
}

static void align_closed_sequences(const Vec2 *A, const Vec2 *B, int n,
                                   int *out_shift, int *out_reverse)
{
    double best_cost = 1e300;
    int best_shift = 0;
    int best_rev = 0;

    for (int shift = 0; shift < n; ++shift) {
        double c0 = cyclic_cost(A, B, n, shift);
        if (c0 < best_cost) {
            best_cost = c0;
            best_shift = shift;
            best_rev = 0;
        }
        double c1 = cyclic_cost_reversed(A, B, n, shift);
        if (c1 < best_cost) {
            best_cost = c1;
            best_shift = shift;
            best_rev = 1;
        }
    }

    *out_shift = best_shift;
    *out_reverse = best_rev;
}

static Vec2 *build_aligned_B(const Vec2 *B, int n, int shift, int reverse)
{
    Vec2 *B2 = (Vec2*)malloc((size_t)n * sizeof(Vec2));
    if (!B2) return NULL;

    if (!reverse) {
        for (int i = 0; i < n; ++i) {
            int j = i + shift;
            if (j >= n) j -= n;
            B2[i] = B[j];
        }
    } else {
        for (int i = 0; i < n; ++i) {
            int bi = i + shift;
            if (bi >= n) bi -= n;
            int j = n - 1 - bi;
            B2[i] = B[j];
        }
```

```
    }
    return B2;
}

/* ----------------- Shape mapping and tweening ----------------- */

static void free_shape_mapping(ShapeMapping *m)
{
    if (!m) return;
    free(m->A);
    free(m->B);
    m->A = m->B = NULL;
    m->N = 0;
}

static int choose_sample_count(int ctrl_src, int ctrl_dst, double tol)
{
    (void)tol;
    int base = (ctrl_src > ctrl_dst) ? ctrl_src : ctrl_dst;
    int k = 4;
    int N = base * k;
    if (N < 32) N = 32;
    return N;
}

static int build_shape_mapping(const BezierPath *src,
                               const BezierPath *dst,
                               double flat_tolerance,
                               ShapeMapping *out)
{
    Polyline pl_src, pl_dst;

    if (!flatten_path(src, flat_tolerance, &pl_src)) return 0;
    if (!flatten_path(dst, flat_tolerance, &pl_dst)) {
        free(pl_src.pt);
        return 0;
    }

    int ctrl_src = src->count * 4;
    int ctrl_dst = dst->count * 4;

    int N = choose_sample_count(ctrl_src, ctrl_dst, flat_tolerance);

    Polyline rs_src, rs_dst;
    if (!resample_polyline(&pl_src, N, &rs_src)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        return 0;
    }
    if (!resample_polyline(&pl_dst, N, &rs_dst)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        free(rs_src.pt);
        return 0;
    }
```

```
    free(pl_src.pt);
    free(pl_dst.pt);

    int shift, rev;
    align_closed_sequences(rs_src.pt, rs_dst.pt, N, &shift, &rev);

    Vec2 *A = rs_src.pt;
    Vec2 *B = build_aligned_B(rs_dst.pt, N, shift, rev);
    free(rs_dst.pt);
    if (!B) {
        free(A);
        return 0;
    }

    out->N = N;
    out->A = A;
    out->B = B;
    return 1;
}

static Polyline tween_polyline(const ShapeMapping *m, double t)
{
    Polyline pl;
    pl.count = m->N + 1;
    pl.pt = (Vec2*)malloc((size_t)pl.count * sizeof(Vec2));
    if (!pl.pt) {
        pl.count = 0;
        return pl;
    }

    double omt = 1.0 - t;
    for (int i = 0; i < m->N; ++i) {
        Vec2 a = m->A[i];
        Vec2 b = m->B[i];
        Vec2 p;
        p.x = omt * a.x + t * b.x;
        p.y = omt * a.y + t * b.y;
        pl.pt[i] = p;
    }
    pl.pt[m->N] = pl.pt[^2_0];
    return pl;
}

/* ----------------- Shape construction: square & circle ----------------- */

/* Unit square centered at origin, side length 2 (from (-1,-1) to (1,1)) */

static BezierPath make_square(void)
{
    BezierPath path;
    path.count = 4;
    path.seg = (CubicBezier*)malloc(4 * sizeof(CubicBezier));

    double s = 1.0;

    /* Use straight-line cubics: p1 and p2 on the edges */
```

```
    /* Bottom: (-s,-s) -> ( s,-s) */
    path.seg[^2_0].p0.x = -s; path.seg[^2_0].p0.y = -s;
    path.seg[^2_0].p1.x = -s; path.seg[^2_0].p1.y = -s;
    path.seg[^2_0].p2.x =  s; path.seg[^2_0].p2.y = -s;
    path.seg[^2_0].p3.x =  s; path.seg[^2_0].p3.y = -s;

    /* Right: (s,-s) -> (s, s) */
    path.seg[^2_1].p0.x =  s; path.seg[^2_1].p0.y = -s;
    path.seg[^2_1].p1.x =  s; path.seg[^2_1].p1.y = -s;
    path.seg[^2_1].p2.x =  s; path.seg[^2_1].p2.y =  s;
    path.seg[^2_1].p3.x =  s; path.seg[^2_1].p3.y =  s;

    /* Top: (s,s) -> (-s,s) */
    path.seg[^2_2].p0.x =  s; path.seg[^2_2].p0.y =  s;
    path.seg[^2_2].p1.x =  s; path.seg[^2_2].p1.y =  s;
    path.seg[^2_2].p2.x = -s; path.seg[^2_2].p2.y =  s;
    path.seg[^2_2].p3.x = -s; path.seg[^2_2].p3.y =  s;

    /* Left: (-s,s) -> (-s,-s) */
    path.seg[^2_3].p0.x = -s; path.seg[^2_3].p0.y =  s;
    path.seg[^2_3].p1.x = -s; path.seg[^2_3].p1.y =  s;
    path.seg[^2_3].p2.x = -s; path.seg[^2_3].p2.y = -s;
    path.seg[^2_3].p3.x = -s; path.seg[^2_3].p3.y = -s;

    return path;
}

/* Circle approximation via 4 cubic Béziers (radius 1, center at origin).
   Control-point distance k = 4*(sqrt(2)-1)/3 ≈ 0.5522847498. [web:36][web:39] */

static BezierPath make_circle(void)
{
    BezierPath path;
    path.count = 4;
    path.seg = (CubicBezier*)malloc(4 * sizeof(CubicBezier));

    double r = 1.0;
    double k = 4.0 * (sqrt(2.0) - 1.0) / 3.0;

    /* Start at (r,0) and go CCW in four 90° arcs */
    /* 0 -> 90° */
    path.seg[^2_0].p0.x =  r; path.seg[^2_0].p0.y = 0.0;
    path.seg[^2_0].p1.x =  r; path.seg[^2_0].p1.y =  r * k;
    path.seg[^2_0].p2.x =  r * k; path.seg[^2_0].p2.y =  r;
    path.seg[^2_0].p3.x =  0.0; path.seg[^2_0].p3.y =  r;

    /* 90° -> 180° */
    path.seg[^2_1].p0 = path.seg[^2_0].p3;
    path.seg[^2_1].p1.x = -r * k; path.seg[^2_1].p1.y =  r;
    path.seg[^2_1].p2.x = -r;     path.seg[^2_1].p2.y =  r * k;
    path.seg[^2_1].p3.x = -r;     path.seg[^2_1].p3.y =  0.0;

    /* 180° -> 270° */
    path.seg[^2_2].p0 = path.seg[^2_1].p3;
    path.seg[^2_2].p1.x = -r;     path.seg[^2_2].p1.y = -r * k;
    path.seg[^2_2].p2.x = -r * k; path.seg[^2_2].p2.y = -r;
```

```
    path.seg[^2_2].p3.x =  0.0;   path.seg[^2_2].p3.y = -r;

    /* 270° -> 360° */
    path.seg[^2_3].p0 = path.seg[^2_2].p3;
    path.seg[^2_3].p1.x =  r * k; path.seg[^2_3].p1.y = -r;
    path.seg[^2_3].p2.x =  r;     path.seg[^2_3].p2.y = -r * k;
    path.seg[^2_3].p3.x =  r;     path.seg[^2_3].p3.y =  0.0;

    return path;
}

/* ----------------- Simple drawing helpers ----------------- */

static void free_polyline(Polyline *pl)
{
    if (!pl) return;
    free(pl->pt);
    pl->pt = NULL;
    pl->count = 0;
}

#ifdef USE_SDL2
static void draw_polyline_sdl(SDL_Renderer *ren, const Polyline *pl,
                              int w, int h)
{
    if (pl->count < 2) return;

    /* Map logical coords [-1,1]x[-1,1] to screen space */
    for (int i = 0; i < pl->count - 1; ++i) {
        Vec2 a = pl->pt[i];
        Vec2 b = pl->pt[i+1];

        int x1 = (int)((a.x * 0.5 + 0.5) * (double)w);
        int y1 = (int)((-a.y * 0.5 + 0.5) * (double)h);
        int x2 = (int)((b.x * 0.5 + 0.5) * (double)w);
        int y2 = (int)((-b.y * 0.5 + 0.5) * (double)h);

        SDL_RenderDrawLine(ren, x1, y1, x2, y2);
    }
}
#endif

/* ----------------- Main test program ----------------- */

int main(int argc, char **argv)
{
    (void)argc; (void)argv;

    BezierPath square = make_square();
    BezierPath circle = make_circle();

    ShapeMapping map;
    double flat_tol = 0.002; /* flattening tolerance */

    if (!build_shape_mapping(&square, &circle, flat_tol, &map)) {
        fprintf(stderr, "Failed to build shape mapping\n");
```

```
        free(square.seg);
        free(circle.seg);
        return 1;
    }

    printf("Mapping built with N=%d correspondence points.\n", map.N);

    /* Print a few sample tween vertices to stdout */
    for (int step = 0; step <= 4; ++step) {
        double t = (double)step / 4.0;
        Polyline pl = tween_polyline(&map, t);
        printf("t = %.2f, first 5 vertices:\n", t);
        int limit = pl.count < 5 ? pl.count : 5;
        for (int i = 0; i < limit; ++i) {
            printf("  (%.3f, %.3f)\n", pl.pt[i].x, pl.pt[i].y);
        }
        free_polyline(&pl);
    }

#ifdef USE_SDL2
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        fprintf(stderr, "SDL_Init failed: %s\n", SDL_GetError());
        free_shape_mapping(&map);
        free(square.seg);
        free(circle.seg);
        return 1;
    }

    int width = 640;
    int height = 640;
    SDL_Window *win = SDL_CreateWindow("Bezier Morph: Square -> Circle",
                                       SDL_WINDOWPOS_CENTERED,
                                       SDL_WINDOWPOS_CENTERED,
                                       width, height, SDL_WINDOW_SHOWN);
    if (!win) {
        fprintf(stderr, "SDL_CreateWindow failed: %s\n", SDL_GetError());
        SDL_Quit();
        free_shape_mapping(&map);
        free(square.seg);
        free(circle.seg);
        return 1;
    }

    SDL_Renderer *ren = SDL_CreateRenderer(win, -1,
                                           SDL_RENDERER_ACCELERATED |
                                           SDL_RENDERER_PRESENTVSYNC);
    if (!ren) {
        fprintf(stderr, "SDL_CreateRenderer failed: %s\n", SDL_GetError());
        SDL_DestroyWindow(win);
        SDL_Quit();
        free_shape_mapping(&map);
        free(square.seg);
        free(circle.seg);
        return 1;
    }
```

```
    int running = 1;
    double t = 0.0;
    double speed = 0.5; /* cycles per second */

    Uint32 last_ticks = SDL_GetTicks();

    while (running) {
        SDL_Event ev;
        while (SDL_PollEvent(&ev)) {
            if (ev.type == SDL_QUIT) running = 0;
            if (ev.type == SDL_KEYDOWN) {
                if (ev.key.keysym.sym == SDLK_ESCAPE) running = 0;
            }
        }

        Uint32 now = SDL_GetTicks();
        double dt = (double)(now - last_ticks) / 1000.0;
        last_ticks = now;

        t += speed * dt;
        if (t > 1.0) t -= 1.0;

        Polyline pl = tween_polyline(&map, t);

        SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
        SDL_RenderClear(ren);

        SDL_SetRenderDrawColor(ren, 255, 255, 255, 255);
        draw_polyline_sdl(ren, &pl, width, height);

        SDL_RenderPresent(ren);

        free_polyline(&pl);
    }

    SDL_DestroyRenderer(ren);
    SDL_DestroyWindow(win);
    SDL_Quit();
#endif

    free_shape_mapping(&map);
    free(square.seg);
    free(circle.seg);

    return 0;
}
```

This program uses the standard four-cubic approximation for a circle and SDL2’s basic renderer API for line drawing under Linux, with typical compilation flags obtained from `sdl2-config`.[21] [26] [22] [25] [20]

⁂

That worked. Now, the two 2-D paths are actually in independent planes. The inputs remain the same 2-D paths but are accompanied by corresponding transforms from the x,y plane to the plane they are in. modify the code to output an accompanying transform for each step in the morphing transformation such that the morphed extrusion from the starting path to the final path describes a 3d solid. The ultimate aim of this code is to perform 'lofting' in OpenSCAD between a starting shape and an end shape. Have the SDL option render a perspective wireframe drawing of all the steps along the extruded 3D shape. Again, output the entire program. Note: add '-lm' library to the end of the command-line for compiling the source. Add the compile command at the top of the source file as a comment. use the name 'loft' rather than 'morph' throughout.

```
/* Compile (Linux):
 *   gcc loft.c -o loft -DUSE_SDL2 `sdl2-config --cflags --libs` -lm
 * or without SDL2:
 *   gcc loft.c -o loft -lm
 *
 * This program:
 *   - Builds two 2D Bézier paths (square, circle)
 *   - Loft-maps them between two arbitrary 3D planes
 *   - Outputs tweened cross-sections and per-step transforms
 *   - Optionally renders a perspective wireframe of the loft in SDL2
 *
 * Perspective projection follows a simple pinhole camera model. [web:55]
 * Lofting idea matches multi-slice interpolation between profiles, as in
 * typical loft operations. [web:59][web:63]
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#ifdef USE_SDL2
#include <SDL2/SDL.h>
#endif
```

```
/* ----------------- Basic math types ----------------- */

typedef struct { double x, y; } Vec2;
typedef struct { double x, y, z; } Vec3;

/* Orthonormal frame for a plane:
 * point P = origin + u * x_axis + v * y_axis
 * x_axis, y_axis, normal are unit and orthogonal.
 */
typedef struct {
    Vec3 origin;
    Vec3 x_axis;
    Vec3 y_axis;
    Vec3 normal;
} PlaneFrame;

/* 4x4 column-major matrix (OpenGL-style) */
typedef struct {
    double m[^3_16];
} Mat4;

/* ----------------- 2D vector helpers ----------------- */

static Vec2 v2_add(Vec2 a, Vec2 b){ Vec2 r = {a.x + b.x, a.y + b.y}; return r; }
static Vec2 v2_sub(Vec2 a, Vec2 b){ Vec2 r = {a.x - b.x, a.y - b.y}; return r; }
static Vec2 v2_scale(Vec2 a, double s){ Vec2 r = {a.x * s, a.y * s}; return r; }
static double v2_dot(Vec2 a, Vec2 b){ return a.x*b.x + a.y*b.y; }
static double v2_len(Vec2 a){ return sqrt(v2_dot(a,a)); }

/* ----------------- 3D vector helpers ----------------- */

static Vec3 v3_add(Vec3 a, Vec3 b){ Vec3 r = {a.x + b.x, a.y + b.y, a.z + b.z}; return r;
static Vec3 v3_sub(Vec3 a, Vec3 b){ Vec3 r = {a.x - b.x, a.y - b.y, a.z - b.z}; return r;
static Vec3 v3_scale(Vec3 a, double s){ Vec3 r = {a.x * s, a.y * s, a.z * s}; return r; }
static double v3_dot(Vec3 a, Vec3 b){ return a.x*b.x + a.y*b.y + a.z*b.z; }
static Vec3 v3_cross(Vec3 a, Vec3 b){
    Vec3 r = {a.y*b.z - a.z*b.y,
              a.z*b.x - a.x*b.z,
              a.x*b.y - a.y*b.x};
    return r;
}
static double v3_len(Vec3 a){ return sqrt(v3_dot(a,a)); }
static Vec3 v3_norm(Vec3 a){
    double L = v3_len(a);
    if (L == 0.0) return a;
    return v3_scale(a, 1.0 / L);
}

/* ----------------- 4x4 matrices ----------------- */

static Mat4 mat4_identity(void)
{
    Mat4 M;
    for (int i = 0; i < 16; ++i) M.m[i] = 0.0;
    M.m[ 0] = 1.0;
```

```
    M.m[ 5] = 1.0;
    M.m[^3_10] = 1.0;
    M.m[^3_15] = 1.0;
    return M;
}

static Mat4 mat4_mul(Mat4 A, Mat4 B)
{
    Mat4 R;
    for (int c = 0; c < 4; ++c) {
        for (int r = 0; r < 4; ++r) {
            double v = 0.0;
            for (int k = 0; k < 4; ++k) {
                v += A.m[k*4 + r] * B.m[c*4 + k];
            }
            R.m[c*4 + r] = v;
        }
    }
    return R;
}

static Vec3 mat4_mul_point(Mat4 M, Vec3 p)
{
    double x = p.x, y = p.y, z = p.z;
    double rx = M.m[^3_0]*x + M.m[^3_4]*y + M.m[^3_8]*z  + M.m[^3_12];
    double ry = M.m[^3_1]*x + M.m[^3_5]*y + M.m[^3_9]*z  + M.m[^3_13];
    double rz = M.m[^3_2]*x + M.m[^3_6]*y + M.m[^3_10]*z + M.m[^3_14];
    double rw = M.m[^3_3]*x + M.m[^3_7]*y + M.m[^3_11]*z + M.m[^3_15];
    if (rw != 0.0) {
        rx /= rw; ry /= rw; rz /= rw;
    }
    Vec3 r = {rx, ry, rz};
    return r;
}

/* Build transform from 2D plane coordinates (u,v,0,1) to 3D space.
 * P = origin + u * x_axis + v * y_axis
 */
static Mat4 plane_to_world(const PlaneFrame *pf)
{
    Mat4 M = mat4_identity();

    /* columns 0,1,2 are basis; column 3 is translation */
    M.m[^3_0]  = pf->x_axis.x;
    M.m[^3_1]  = pf->x_axis.y;
    M.m[^3_2]  = pf->x_axis.z;

    M.m[^3_4]  = pf->y_axis.x;
    M.m[^3_5]  = pf->y_axis.y;
    M.m[^3_6]  = pf->y_axis.z;

    M.m[^3_8]  = pf->normal.x;
    M.m[^3_9]  = pf->normal.y;
    M.m[^3_10] = pf->normal.z;

    M.m[^3_12] = pf->origin.x;
```

```
    M.m[^3_13] = pf->origin.y;
    M.m[^3_14] = pf->origin.z;

    return M;
}

/* ----------------- Cubic Bézier path structs ----------------- */

typedef struct {
    Vec2 p0, p1, p2, p3;
} CubicBezier2D;

typedef struct {
    int           count;
    CubicBezier2D *seg;
} BezierPath2D;

typedef struct {
    int   count;
    Vec2 *pt;
} Polyline2D;

/* ----------------- Cubic Bézier helpers (2D) ----------------- */

static Vec2 cubic2_eval(const CubicBezier2D *c, double t)
{
    double u  = 1.0 - t;
    double tt = t * t;
    double uu = u * u;
    double uuu = uu * u;
    double ttt = tt * t;

    Vec2 p;
    p.x = uuu * c->p0.x;
    p.y = uuu * c->p0.y;

    p.x += 3.0 * uu * t * c->p1.x;
    p.y += 3.0 * uu * t * c->p1.y;

    p.x += 3.0 * u * tt * c->p2.x;
    p.y += 3.0 * u * tt * c->p2.y;

    p.x += ttt * c->p3.x;
    p.y += ttt * c->p3.y;

    return p;
}

static double cubic2_flatness_sq(const CubicBezier2D *c)
{
    Vec2 a = c->p0;
    Vec2 b = c->p3;
    Vec2 ab = v2_sub(b, a);

    double ab_len = v2_len(ab);
    if (ab_len == 0.0) {
```

```
        double d1 = v2_len(v2_sub(c->p1, a));
        double d2 = v2_len(v2_sub(c->p2, a));
        double d = d1 > d2 ? d1 : d2;
        return d * d;
    }

    double abx = ab.x;
    double aby = ab.y;

    double dist_sq(Vec2 p){
        Vec2 ap = v2_sub(p, a);
        double cross = ap.x * aby - ap.y * abx;
        double d = cross / ab_len;
        return d * d;
    }

    double d1 = dist_sq(c->p1);
    double d2 = dist_sq(c->p2);
    return d1 > d2 ? d1 : d2;
}

static void cubic2_subdivide(const CubicBezier2D *c,
                             CubicBezier2D *left,
                             CubicBezier2D *right)
{
    Vec2 p01  = v2_scale(v2_add(c->p0, c->p1), 0.5);
    Vec2 p12  = v2_scale(v2_add(c->p1, c->p2), 0.5);
    Vec2 p23  = v2_scale(v2_add(c->p2, c->p3), 0.5);

    Vec2 p012 = v2_scale(v2_add(p01, p12), 0.5);
    Vec2 p123 = v2_scale(v2_add(p12, p23), 0.5);

    Vec2 p0123 = v2_scale(v2_add(p012, p123), 0.5);

    left->p0 = c->p0;
    left->p1 = p01;
    left->p2 = p012;
    left->p3 = p0123;

    right->p0 = p0123;
    right->p1 = p123;
    right->p2 = p23;
    right->p3 = c->p3;
}

/* ----------------- Dynamic 2D Vec buffer ----------------- */

typedef struct {
    Vec2 *data;
    int   count;
    int   cap;
} Vec2Buffer;

static void buf2_init(Vec2Buffer *b){ b->data = NULL; b->count = b->cap = 0; }
static void buf2_free(Vec2Buffer *b){ free(b->data); b->data = NULL; b->count = b->cap =
```

```
static int buf2_push(Vec2Buffer *b, Vec2 p)
{
    if (b->count >= b->cap) {
        int ncap = b->cap ? b->cap * 2 : 64;
        Vec2 *nd = (Vec2*)realloc(b->data, (size_t)ncap * sizeof(Vec2));
        if (!nd) return 0;
        b->data = nd;
        b->cap  = ncap;
    }
    b->data[b->count++] = p;
    return 1;
}

/* ----------------- Flatten 2D path to polyline ----------------- */

static int flatten_cubic2_rec(const CubicBezier2D *c,
                              double tol_sq,
                              Vec2Buffer *out)
{
    if (cubic2_flatness_sq(c) <= tol_sq) {
        return buf2_push(out, c->p3);
    } else {
        CubicBezier2D l, r;
        cubic2_subdivide(c, &l, &r);
        if (!flatten_cubic2_rec(&l, tol_sq, out)) return 0;
        if (!flatten_cubic2_rec(&r, tol_sq, out)) return 0;
        return 1;
    }
}

static int flatten_path2(const BezierPath2D *path,
                         double tol,
                         Polyline2D *poly_out)
{
    Vec2Buffer buf;
    buf2_init(&buf);
    double tol_sq = tol * tol;

    if (path->count <= 0) {
        poly_out->count = 0;
        poly_out->pt = NULL;
        return 1;
    }

    if (!buf2_push(&buf, path->seg[^3_0].p0)) {
        buf2_free(&buf);
        return 0;
    }

    for (int i = 0; i < path->count; ++i) {
        CubicBezier2D c = path->seg[i];
        if (!flatten_cubic2_rec(&c, tol_sq, &buf)) {
            buf2_free(&buf);
            return 0;
        }
    }
```

```
    if (buf.count > 0) {
        Vec2 first = buf.data[^3_0];
        Vec2 last  = buf.data[buf.count - 1];
        if (fabs(first.x - last.x) > 1e-9 || fabs(first.y - last.y) > 1e-9) {
            if (!buf2_push(&buf, first)) {
                buf2_free(&buf);
                return 0;
            }
        }
    }

    poly_out->count = buf.count;
    poly_out->pt    = buf.data;
    return 1;
}

/* ----------------- Polyline arc-length and resampling ----------------- */

static double *poly2_arclen(const Polyline2D *pl)
{
    if (pl->count <= 0) return NULL;
    double *s = (double*)malloc((size_t)pl->count * sizeof(double));
    if (!s) return NULL;
    s[^3_0] = 0.0;
    for (int i = 1; i < pl->count; ++i) {
        Vec2 d = v2_sub(pl->pt[i], pl->pt[i-1]);
        s[i] = s[i-1] + v2_len(d);
    }
    return s;
}

static Vec2 poly2_sample_at(const Polyline2D *pl,
                            const double *s,
                            double d)
{
    int n = pl->count;
    double total = s[n-1];
    if (total <= 0.0) return pl->pt[^3_0];

    d = fmod(d, total);
    if (d < 0.0) d += total;

    int i = 0;
    while (i+1 < n && s[i+1] < d) ++i;

    int j = (i+1 < n) ? i+1 : i;
    double si = s[i];
    double sj = s[j];
    double t;
    if (sj <= si) t = 0.0;
    else t = (d - si) / (sj - si);

    Vec2 a = pl->pt[i];
    Vec2 b = pl->pt[j];
    return v2_add(a, v2_scale(v2_sub(b, a), t));
```

```
}

static int resample_polyline2(const Polyline2D *pl,
                              int N,
                              Polyline2D *out)
{
    if (N <= 0 || pl->count <= 1) {
        out->count = 0;
        out->pt = NULL;
        return 1;
    }

    double *s = poly2_arclen(pl);
    if (!s) return 0;

    Vec2 *pts = (Vec2*)malloc((size_t)N * sizeof(Vec2));
    if (!pts) { free(s); return 0; }

    double total = s[pl->count - 1];
    for (int i = 0; i < N; ++i) {
        double d = total * ((double)i / (double)N);
        pts[i] = poly2_sample_at(pl, s, d);
    }

    free(s);
    out->count = N;
    out->pt    = pts;
    return 1;
}

/* ----------------- Closed sequence alignment ----------------- */

static double cyclic_cost2(const Vec2 *A, const Vec2 *B, int n, int shift)
{
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int j = i + shift;
        if (j >= n) j -= n;
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
    }
    return c;
}

static double cyclic_cost2_rev(const Vec2 *A, const Vec2 *B,
                               int n, int shift)
{
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int bi = i + shift;
        if (bi >= n) bi -= n;
        int j = n - 1 - bi;
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
```

```
    }
    return c;
}

static void align_closed2(const Vec2 *A, const Vec2 *B, int n,
                          int *out_shift, int *out_reverse)
{
    double best_cost = 1e300;
    int best_shift = 0;
    int best_rev = 0;

    for (int shift = 0; shift < n; ++shift) {
        double c0 = cyclic_cost2(A, B, n, shift);
        if (c0 < best_cost) {
            best_cost = c0;
            best_shift = shift;
            best_rev   = 0;
        }
        double c1 = cyclic_cost2_rev(A, B, n, shift);
        if (c1 < best_cost) {
            best_cost = c1;
            best_shift = shift;
            best_rev   = 1;
        }
    }
    *out_shift   = best_shift;
    *out_reverse = best_rev;
}

static Vec2 *build_aligned2(const Vec2 *B, int n,
                            int shift, int reverse)
{
    Vec2 *B2 = (Vec2*)malloc((size_t)n * sizeof(Vec2));
    if (!B2) return NULL;

    if (!reverse) {
        for (int i = 0; i < n; ++i) {
            int j = i + shift;
            if (j >= n) j -= n;
            B2[i] = B[j];
        }
    } else {
        for (int i = 0; i < n; ++i) {
            int bi = i + shift;
            if (bi >= n) bi -= n;
            int j = n - 1 - bi;
            B2[i] = B[j];
        }
    }
    return B2;
}

/* ----------------- Shape mapping (2D) ----------------- */

typedef struct {
    int   N;
```

```
    Vec2 *A;
    Vec2 *B;
} ShapeMapping2D;

static void free_shape_mapping2(ShapeMapping2D *m)
{
    if (!m) return;
    free(m->A);
    free(m->B);
    m->A = m->B = NULL;
    m->N = 0;
}

static int choose_sample_count2(int ctrl_src, int ctrl_dst, double tol)
{
    (void)tol;
    int base = (ctrl_src > ctrl_dst) ? ctrl_src : ctrl_dst;
    int k = 4;
    int N = base * k;
    if (N < 32) N = 32;
    return N;
}

static int build_shape_mapping2(const BezierPath2D *src,
                                const BezierPath2D *dst,
                                double flat_tol,
                                ShapeMapping2D *out)
{
    Polyline2D pl_src, pl_dst;

    if (!flatten_path2(src, flat_tol, &pl_src)) return 0;
    if (!flatten_path2(dst, flat_tol, &pl_dst)) {
        free(pl_src.pt);
        return 0;
    }

    int ctrl_src = src->count * 4;
    int ctrl_dst = dst->count * 4;

    int N = choose_sample_count2(ctrl_src, ctrl_dst, flat_tol);

    Polyline2D rs_src, rs_dst;
    if (!resample_polyline2(&pl_src, N, &rs_src)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        return 0;
    }
    if (!resample_polyline2(&pl_dst, N, &rs_dst)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        free(rs_src.pt);
        return 0;
    }

    free(pl_src.pt);
    free(pl_dst.pt);
```

```
    int shift, rev;
    align_closed2(rs_src.pt, rs_dst.pt, N, &shift, &rev);

    Vec2 *A = rs_src.pt;
    Vec2 *B = build_aligned2(rs_dst.pt, N, shift, rev);
    free(rs_dst.pt);
    if (!B) {
        free(A);
        return 0;
    }

    out->N = N;
    out->A = A;
    out->B = B;
    return 1;
}

/* Tween 2D polyline at parameter t */

static Polyline2D tween_polyline2(const ShapeMapping2D *m, double t)
{
    Polyline2D pl;
    pl.count = m->N + 1;
    pl.pt = (Vec2*)malloc((size_t)pl.count * sizeof(Vec2));
    if (!pl.pt) {
        pl.count = 0;
        return pl;
    }
    double omt = 1.0 - t;
    for (int i = 0; i < m->N; ++i) {
        Vec2 a = m->A[i];
        Vec2 b = m->B[i];
        Vec2 p;
        p.x = omt * a.x + t * b.x;
        p.y = omt * a.y + t * b.y;
        pl.pt[i] = p;
    }
    pl.pt[m->N] = pl.pt[^3_0];
    return pl;
}

/* ----------------- Loft: planes + 3D cross-sections ----------------- */

typedef struct {
    int   count;
    Vec3 *pt;
} Polyline3D;

static void free_polyline2(Polyline2D *pl)
{
    if (!pl) return;
    free(pl->pt);
    pl->pt = NULL;
    pl->count = 0;
}
```

```
static void free_polyline3(Polyline3D *pl)
{
    if (!pl) return;
    free(pl->pt);
    pl->pt = NULL;
    pl->count = 0;
}

/* Given 2D cross-section p2 and a per-step plane, map to 3D */
static Polyline3D lift_polyline_to_plane(const Polyline2D *pl,
                                         const PlaneFrame *plane)
{
    Polyline3D out;
    out.count = pl->count;
    out.pt = (Vec3*)malloc((size_t)out.count * sizeof(Vec3));
    if (!out.pt) {
        out.count = 0;
        return out;
    }

    Mat4 T = plane_to_world(plane);
    for (int i = 0; i < pl->count; ++i) {
        Vec2 q = pl->pt[i];
        Vec3 p = {q.x, q.y, 0.0};
        out.pt[i] = mat4_mul_point(T, p);
    }
    return out;
}

/* ----------------- Camera and projection for SDL wireframe ----------------- */

/* Simple camera: position, forward, up; right = forward x up */
typedef struct {
    Vec3 pos;
    Vec3 forward;
    Vec3 up;
} Camera;

static Mat4 camera_view_matrix(const Camera *cam)
{
    Vec3 f = v3_norm(cam->forward);
    Vec3 r = v3_norm(v3_cross(f, cam->up));
    Vec3 u = v3_cross(r, f);

    Mat4 V = mat4_identity();

    V.m[^3_0] = r.x; V.m[^3_4] = r.y; V.m[^3_8]  = r.z;
    V.m[^3_1] = u.x; V.m[^3_5] = u.y; V.m[^3_9]  = u.z;
    V.m[^3_2] = -f.x;V.m[^3_6] = -f.y;V.m[^3_10] = -f.z;

    Vec3 t;
    t.x = -v3_dot(r, cam->pos);
    t.y = -v3_dot(u, cam->pos);
    t.z =  v3_dot(f, cam->pos);
```

```
    V.m[^3_12] = t.x;
    V.m[^3_13] = t.y;
    V.m[^3_14] = t.z;

    return V;
}

/* Simple perspective projection, symmetric frustum. [web:55][web:58] */
static Mat4 perspective_matrix(double fov_y_deg,
                               double aspect,
                               double z_near,
                               double z_far)
{
    double fov_rad = fov_y_deg * (M_PI / 180.0);
    double f = 1.0 / tan(fov_rad * 0.5);
    Mat4 P;
    for (int i = 0; i < 16; ++i) P.m[i] = 0.0;

    P.m[^3_0]  = f / aspect;
    P.m[^3_5]  = f;
    P.m[^3_10] = (z_far + z_near) / (z_near - z_far);
    P.m[^3_11] = -1.0;
    P.m[^3_14] = (2.0 * z_far * z_near) / (z_near - z_far);
    return P;
}

/* Project 3D point to normalized device coordinates (-1..1) */
static int project_point(const Mat4 *VP, Vec3 p,
                         double *x_ndc, double *y_ndc, double *z_ndc)
{
    Vec3 q = mat4_mul_point(*VP, p);
    /* W already divided in mat4_mul_point */
    if (q.z <= 0.0) return 0; /* behind camera */
    *x_ndc = q.x;
    *y_ndc = q.y;
    *z_ndc = q.z;
    return 1;
}

#ifdef USE_SDL2
static void draw_polyline3d(SDL_Renderer *ren,
                            const Mat4 *VP,
                            const Polyline3D *pl,
                            int w, int h)
{
    if (pl->count < 2) return;

    for (int i = 0; i < pl->count - 1; ++i) {
        Vec3 a = pl->pt[i];
        Vec3 b = pl->pt[i+1];
        double ax, ay, az;
        double bx, by, bz;
        if (!project_point(VP, a, &ax, &ay, &az)) continue;
        if (!project_point(VP, b, &bx, &by, &bz)) continue;

        int sx1 = (int)((ax * 0.5 + 0.5) * (double)w);
```

```
        int sy1 = (int)((-ay * 0.5 + 0.5) * (double)h);
        int sx2 = (int)((bx * 0.5 + 0.5) * (double)w);
        int sy2 = (int)((-by * 0.5 + 0.5) * (double)h);

        SDL_RenderDrawLine(ren, sx1, sy1, sx2, sy2);
    }
}
#endif

/* ----------------- Shape construction: square & circle in 2D ----------------- */

/* Square in local 2D coords, side length 2, centered at origin */
static BezierPath2D make_square2(void)
{
    BezierPath2D path;
    path.count = 4;
    path.seg = (CubicBezier2D*)malloc(4 * sizeof(CubicBezier2D));
    double s = 1.0;

    /* Straight-line cubics along edges */
    path.seg[^3_0].p0.x = -s; path.seg[^3_0].p0.y = -s;
    path.seg[^3_0].p1.x = -s; path.seg[^3_0].p1.y = -s;
    path.seg[^3_0].p2.x =  s; path.seg[^3_0].p2.y = -s;
    path.seg[^3_0].p3.x =  s; path.seg[^3_0].p3.y = -s;

    path.seg[^3_1].p0.x =  s; path.seg[^3_1].p0.y = -s;
    path.seg[^3_1].p1.x =  s; path.seg[^3_1].p1.y = -s;
    path.seg[^3_1].p2.x =  s; path.seg[^3_1].p2.y =  s;
    path.seg[^3_1].p3.x =  s; path.seg[^3_1].p3.y =  s;

    path.seg[^3_2].p0.x =  s; path.seg[^3_2].p0.y =  s;
    path.seg[^3_2].p1.x =  s; path.seg[^3_2].p1.y =  s;
    path.seg[^3_2].p2.x = -s; path.seg[^3_2].p2.y =  s;
    path.seg[^3_2].p3.x = -s; path.seg[^3_2].p3.y =  s;

    path.seg[^3_3].p0.x = -s; path.seg[^3_3].p0.y =  s;
    path.seg[^3_3].p1.x = -s; path.seg[^3_3].p1.y =  s;
    path.seg[^3_3].p2.x = -s; path.seg[^3_3].p2.y = -s;
    path.seg[^3_3].p3.x = -s; path.seg[^3_3].p3.y = -s;

    return path;
}

/* Circle approx via 4 cubics, radius 1, center at origin. [web:36][web:39] */
static BezierPath2D make_circle2(void)
{
    BezierPath2D path;
    path.count = 4;
    path.seg = (CubicBezier2D*)malloc(4 * sizeof(CubicBezier2D));

    double r = 1.0;
    double k = 4.0 * (sqrt(2.0) - 1.0) / 3.0;

    path.seg[^3_0].p0.x =  r; path.seg[^3_0].p0.y = 0.0;
    path.seg[^3_0].p1.x =  r; path.seg[^3_0].p1.y =  r * k;
    path.seg[^3_0].p2.x =  r * k; path.seg[^3_0].p2.y =  r;
```

```
    path.seg[^3_0].p3.x =  0.0;   path.seg[^3_0].p3.y =  r;

    path.seg[^3_1].p0 = path.seg[^3_0].p3;
    path.seg[^3_1].p1.x = -r * k; path.seg[^3_1].p1.y =  r;
    path.seg[^3_1].p2.x = -r;     path.seg[^3_1].p2.y =  r * k;
    path.seg[^3_1].p3.x = -r;     path.seg[^3_1].p3.y =  0.0;

    path.seg[^3_2].p0 = path.seg[^3_1].p3;
    path.seg[^3_2].p1.x = -r;     path.seg[^3_2].p1.y = -r * k;
    path.seg[^3_2].p2.x = -r * k; path.seg[^3_2].p2.y = -r;
    path.seg[^3_2].p3.x =  0.0;   path.seg[^3_2].p3.y = -r;

    path.seg[^3_3].p0 = path.seg[^3_2].p3;
    path.seg[^3_3].p1.x =  r * k; path.seg[^3_3].p1.y = -r;
    path.seg[^3_3].p2.x =  r;     path.seg[^3_3].p2.y = -r * k;
    path.seg[^3_3].p3.x =  r;     path.seg[^3_3].p3.y =  0.0;

    return path;
}

/* ----------------- Example plane frames for demo ----------------- */

/* Bottom plane: XY plane at z=0 */
static PlaneFrame make_bottom_plane(void)
{
    PlaneFrame pf;
    pf.origin.x = 0.0; pf.origin.y = 0.0; pf.origin.z = 0.0;
    pf.x_axis.x = 1.0; pf.x_axis.y = 0.0; pf.x_axis.z = 0.0;
    pf.y_axis.x = 0.0; pf.y_axis.y = 1.0; pf.y_axis.z = 0.0;
    pf.normal.x = 0.0; pf.normal.y = 0.0; pf.normal.z = 1.0;
    return pf;
}

/* Top plane: rotated and translated plane */
static PlaneFrame make_top_plane(void)
{
    PlaneFrame pf;

    /* Example: center at (0,0,4), normal tilted */
    pf.origin.x = 0.0;
    pf.origin.y = 0.0;
    pf.origin.z = 4.0;

    Vec3 n = {0.2, 0.4, 0.87};
    pf.normal = v3_norm(n);

    /* Choose x_axis roughly along world X and orthogonalize */
    Vec3 tmp = {1.0, 0.0, 0.0};
    Vec3 x = v3_sub(tmp, v3_scale(pf.normal, v3_dot(tmp, pf.normal)));
    if (v3_len(x) < 1e-6) {
        tmp.x = 0.0; tmp.y = 1.0; tmp.z = 0.0;
        x = v3_sub(tmp, v3_scale(pf.normal, v3_dot(tmp, pf.normal)));
    }
    pf.x_axis = v3_norm(x);
    pf.y_axis = v3_norm(v3_cross(pf.normal, pf.x_axis));
```

```
    return pf;
}

/* Interpolate two plane frames linearly in origin and axes, renormalizing. */
static PlaneFrame lerp_plane(const PlaneFrame *a,
                             const PlaneFrame *b,
                             double t)
{
    double omt = 1.0 - t;
    PlaneFrame pf;
    pf.origin.x = omt * a->origin.x + t * b->origin.x;
    pf.origin.y = omt * a->origin.y + t * b->origin.y;
    pf.origin.z = omt * a->origin.z + t * b->origin.z;

    pf.x_axis.x = omt * a->x_axis.x + t * b->x_axis.x;
    pf.x_axis.y = omt * a->x_axis.y + t * b->x_axis.y;
    pf.x_axis.z = omt * a->x_axis.z + t * b->x_axis.z;

    pf.y_axis.x = omt * a->y_axis.x + t * b->y_axis.x;
    pf.y_axis.y = omt * a->y_axis.y + t * b->y_axis.y;
    pf.y_axis.z = omt * a->y_axis.z + t * b->y_axis.z;

    pf.normal.x = omt * a->normal.x + t * b->normal.x;
    pf.normal.y = omt * a->normal.y + t * b->normal.y;
    pf.normal.z = omt * a->normal.z + t * b->normal.z;

    pf.x_axis = v3_norm(pf.x_axis);
    pf.y_axis = v3_norm(pf.y_axis);
    pf.normal = v3_norm(pf.normal);

    return pf;
}

/* ----------------- Main: loft demo ----------------- */

int main(int argc, char **argv)
{
    (void)argc; (void)argv;

    /* Build 2D paths */
    BezierPath2D path_bottom = make_square2();
    BezierPath2D path_top    = make_circle2();

    /* Build shape mapping in 2D */
    ShapeMapping2D map2;
    double flat_tol = 0.002;
    if (!build_shape_mapping2(&path_bottom, &path_top, flat_tol, &map2)) {
        fprintf(stderr, "Failed to build 2D shape mapping\n");
        free(path_bottom.seg);
        free(path_top.seg);
        return 1;
    }

    printf("2D loft mapping built with N=%d correspondence points.\n",
           map2.N);
```

```
    /* Define planes for bottom and top shapes */
    PlaneFrame plane_bottom = make_bottom_plane();
    PlaneFrame plane_top    = make_top_plane();

    /* Number of loft steps (cross-sections) including endpoints */
    int steps = 16;
    if (steps < 2) steps = 2;

    /* Generate loft cross-sections and transforms */
    Polyline3D *sections = (Polyline3D*)calloc((size_t)steps,
                                               sizeof(Polyline3D));
    if (!sections) {
        free_shape_mapping2(&map2);
        free(path_bottom.seg);
        free(path_top.seg);
        return 1;
    }

    printf("Loft transforms (origin and axes) for each step:\n");
    for (int k = 0; k < steps; ++k) {
        double t = (double)k / (double)(steps - 1);
        PlaneFrame pf = lerp_plane(&plane_bottom, &plane_top, t);

        /* Log transform: origin and normal (sufficient for OpenSCAD plane) */
        printf("step %2d t=%.3f: origin=(%.3f,%.3f,%.3f) normal=(%.3f,%.3f,%.3f)\n",
               k, t,
               pf.origin.x, pf.origin.y, pf.origin.z,
               pf.normal.x, pf.normal.y, pf.normal.z);

        /* Tween 2D shape */
        Polyline2D pl2 = tween_polyline2(&map2, t);

        /* Lift into 3D plane */
        sections[k] = lift_polyline_to_plane(&pl2, &pf);

        free_polyline2(&pl2);
    }

#ifdef USE_SDL2
    /* SDL2 visualization: perspective wireframe of all cross-sections. */
    if (SDL_Init(SDL_INIT_VIDEO) != 0) {
        fprintf(stderr, "SDL_Init failed: %s\n", SDL_GetError());
        for (int k = 0; k < steps; ++k) free_polyline3(&sections[k]);
        free(sections);
        free_shape_mapping2(&map2);
        free(path_bottom.seg);
        free(path_top.seg);
        return 1;
    }

    int width = 800, height = 800;
    SDL_Window *win = SDL_CreateWindow("Loft: square -> circle (3D wireframe)",
                                       SDL_WINDOWPOS_CENTERED,
                                       SDL_WINDOWPOS_CENTERED,
                                       width, height,
                                       SDL_WINDOW_SHOWN);
```

```
    if (!win) {
        fprintf(stderr, "SDL_CreateWindow failed: %s\n", SDL_GetError());
        SDL_Quit();
        for (int k = 0; k < steps; ++k) free_polyline3(&sections[k]);
        free(sections);
        free_shape_mapping2(&map2);
        free(path_bottom.seg);
        free(path_top.seg);
        return 1;
    }

    SDL_Renderer *ren = SDL_CreateRenderer(win, -1,
                                           SDL_RENDERER_ACCELERATED |
                                           SDL_RENDERER_PRESENTVSYNC);
    if (!ren) {
        fprintf(stderr, "SDL_CreateRenderer failed: %s\n", SDL_GetError());
        SDL_DestroyWindow(win);
        SDL_Quit();
        for (int k = 0; k < steps; ++k) free_polyline3(&sections[k]);
        free(sections);
        free_shape_mapping2(&map2);
        free(path_bottom.seg);
        free(path_top.seg);
        return 1;
    }

    Camera cam;
    cam.pos.x = 6.0; cam.pos.y = 5.0; cam.pos.z = 6.0;
    Vec3 target = {0.0, 0.0, 2.0};
    cam.forward = v3_norm(v3_sub(target, cam.pos));
    cam.up.x = 0.0; cam.up.y = 0.0; cam.up.z = 1.0;

    Mat4 V = camera_view_matrix(&cam);
    Mat4 P = perspective_matrix(60.0,
                                (double)width / (double)height,
                                0.1, 50.0);
    Mat4 VP = mat4_mul(P, V);

    int running = 1;
    while (running) {
        SDL_Event ev;
        while (SDL_PollEvent(&ev)) {
            if (ev.type == SDL_QUIT) running = 0;
            if (ev.type == SDL_KEYDOWN) {
                if (ev.key.keysym.sym == SDLK_ESCAPE) running = 0;
            }
        }

        SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
        SDL_RenderClear(ren);

        SDL_SetRenderDrawColor(ren, 160, 160, 160, 255);
        for (int k = 0; k < steps; ++k) {
            draw_polyline3d(ren, &VP, &sections[k], width, height);
        }
```

```
        /* Draw vertical "rails" between corresponding vertices to see loft */
        SDL_SetRenderDrawColor(ren, 80, 255, 80, 255);
        for (int i = 0; i < map2.N; ++i) {
            for (int k = 0; k < steps - 1; ++k) {
                Vec3 a = sections[k].pt[i];
                Vec3 b = sections[k+1].pt[i];
                double ax, ay, az, bx, by, bz;
                if (!project_point(&VP, a, &ax, &ay, &az)) continue;
                if (!project_point(&VP, b, &bx, &by, &bz)) continue;
                int sx1 = (int)((ax * 0.5 + 0.5) * (double)width);
                int sy1 = (int)((-ay * 0.5 + 0.5) * (double)height);
                int sx2 = (int)((bx * 0.5 + 0.5) * (double)width);
                int sy2 = (int)((-by * 0.5 + 0.5) * (double)height);
                SDL_RenderDrawLine(ren, sx1, sy1, sx2, sy2);
            }
        }

        SDL_RenderPresent(ren);
    }

    SDL_DestroyRenderer(ren);
    SDL_DestroyWindow(win);
    SDL_Quit();
#else
    printf("SDL2 not enabled; build with -DUSE_SDL2 for wireframe view.\n");
#endif

    for (int k = 0; k < steps; ++k) free_polyline3(&sections[k]);
    free(sections);
    free_shape_mapping2(&map2);
    free(path_bottom.seg);
    free(path_top.seg);

    return 0;
}
```

⁂

That code worked. Modify it now to output (on stdout) OpenSCAD source code that will generate the same 3D object.

The loft can be expressed as a single OpenSCAD `polyhedron(points, faces)` built from all cross-section slices and “rails” between them. Below is a complete replacement for `loft.c` that: [44] [45] [46]

- Keeps the previous lofting logic.
- Adds an OpenSCAD writer that emits a `polyhedron()` with all vertices and faces.[47] [44]
- Still offers the SDL2 wireframe view when built with `-DUSE_SDL2`.

Compile:

```
gcc loft.c -o loft -DUSE_SDL2 `sdl2-config --cflags --libs` -lm
# or without SDL2:
gcc loft.c -o loft -lm
```

Running `./loft > loft.scad` produces OpenSCAD source defining the loft via `polyhedron`.[45] [44]

```
/* Compile (Linux):
 *   gcc loft.c -o loft -DUSE_SDL2 `sdl2-config --cflags --libs` -lm
 * or without SDL2:
 *   gcc loft.c -o loft -lm
 *
 * Run and redirect to OpenSCAD:
 *   ./loft > loft.scad
 *
 * The generated OpenSCAD describes the lofted solid as a polyhedron()
 * built from multiple interpolated cross-sections ("slices"). [web:70][web:72][web:77]
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#ifdef USE_SDL2
#include <SDL2/SDL.h>
#endif

/* ----------------- Basic math types ----------------- */

typedef struct { double x, y; } Vec2;
typedef struct { double x, y, z; } Vec3;

/* Plane frame: P = origin + u*x_axis + v*y_axis */
typedef struct {
    Vec3 origin;
    Vec3 x_axis;
    Vec3 y_axis;
    Vec3 normal;
} PlaneFrame;

/* 4x4 column-major matrix */
typedef struct {
    double m[^4_16];
} Mat4;

/* ----------------- 2D/3D helpers ----------------- */

static Vec2 v2_add(Vec2 a, Vec2 b){ Vec2 r = {a.x + b.x, a.y + b.y}; return r; }
static Vec2 v2_sub(Vec2 a, Vec2 b){ Vec2 r = {a.x - b.x, a.y - b.y}; return r; }
static Vec2 v2_scale(Vec2 a, double s){ Vec2 r = {a.x * s, a.y * s}; return r; }
static double v2_dot(Vec2 a, Vec2 b){ return a.x*b.x + a.y*b.y; }
static double v2_len(Vec2 a){ return sqrt(v2_dot(a,a)); }

static Vec3 v3_add(Vec3 a, Vec3 b){ Vec3 r = {a.x + b.x, a.y + b.y, a.z + b.z}; return r;
```

```
static Vec3 v3_sub(Vec3 a, Vec3 b){ Vec3 r = {a.x - b.x, a.y - b.y, a.z - b.z}; return r;
static Vec3 v3_scale(Vec3 a, double s){ Vec3 r = {a.x * s, a.y * s, a.z * s}; return r; }
static double v3_dot(Vec3 a, Vec3 b){ return a.x*b.x + a.y*b.y + a.z*b.z; }
static Vec3 v3_cross(Vec3 a, Vec3 b){
    Vec3 r = {a.y*b.z - a.z*b.y,
              a.z*b.x - a.x*b.z,
              a.x*b.y - a.y*b.x};
    return r;
}
static double v3_len(Vec3 a){ return sqrt(v3_dot(a,a)); }
static Vec3 v3_norm(Vec3 a){
    double L = v3_len(a);
    if (L == 0.0) return a;
    return v3_scale(a, 1.0 / L);
}

/* ----------------- 4x4 matrices ----------------- */

static Mat4 mat4_identity(void)
{
    Mat4 M;
    for (int i = 0; i < 16; ++i) M.m[i] = 0.0;
    M.m[ 0] = 1.0;
    M.m[ 5] = 1.0;
    M.m[^4_10] = 1.0;
    M.m[^4_15] = 1.0;
    return M;
}

static Mat4 mat4_mul(Mat4 A, Mat4 B)
{
    Mat4 R;
    for (int c = 0; c < 4; ++c) {
        for (int r = 0; r < 4; ++r) {
            double v = 0.0;
            for (int k = 0; k < 4; ++k) {
                v += A.m[k*4 + r] * B.m[c*4 + k];
            }
            R.m[c*4 + r] = v;
        }
    }
    return R;
}

static Vec3 mat4_mul_point(Mat4 M, Vec3 p)
{
    double x = p.x, y = p.y, z = p.z;
    double rx = M.m[^4_0]*x + M.m[^4_4]*y + M.m[^4_8]*z  + M.m[^4_12];
    double ry = M.m[^4_1]*x + M.m[^4_5]*y + M.m[^4_9]*z  + M.m[^4_13];
    double rz = M.m[^4_2]*x + M.m[^4_6]*y + M.m[^4_10]*z + M.m[^4_14];
    double rw = M.m[^4_3]*x + M.m[^4_7]*y + M.m[^4_11]*z + M.m[^4_15];
    if (rw != 0.0) {
        rx /= rw; ry /= rw; rz /= rw;
    }
    Vec3 r = {rx, ry, rz};
    return r;
```

```
}

/* Build transform from 2D plane coords (u,v) to 3D space. */
static Mat4 plane_to_world(const PlaneFrame *pf)
{
    Mat4 M = mat4_identity();

    M.m[^4_0]  = pf->x_axis.x;
    M.m[^4_1]  = pf->x_axis.y;
    M.m[^4_2]  = pf->x_axis.z;

    M.m[^4_4]  = pf->y_axis.x;
    M.m[^4_5]  = pf->y_axis.y;
    M.m[^4_6]  = pf->y_axis.z;

    M.m[^4_8]  = pf->normal.x;
    M.m[^4_9]  = pf->normal.y;
    M.m[^4_10] = pf->normal.z;

    M.m[^4_12] = pf->origin.x;
    M.m[^4_13] = pf->origin.y;
    M.m[^4_14] = pf->origin.z;

    return M;
}

/* ----------------- 2D Bézier path ----------------- */

typedef struct {
    Vec2 p0, p1, p2, p3;
} CubicBezier2D;

typedef struct {
    int           count;
    CubicBezier2D *seg;
} BezierPath2D;

typedef struct {
    int   count;
    Vec2 *pt;
} Polyline2D;

/* Cubic Bézier 2D eval/flatness/subdivide */

static Vec2 cubic2_eval(const CubicBezier2D *c, double t)
{
    double u  = 1.0 - t;
    double tt = t * t;
    double uu = u * u;
    double uuu = uu * u;
    double ttt = tt * t;

    Vec2 p;
    p.x = uuu * c->p0.x;
    p.y = uuu * c->p0.y;
```

```
    p.x += 3.0 * uu * t * c->p1.x;
    p.y += 3.0 * uu * t * c->p1.y;

    p.x += 3.0 * u * tt * c->p2.x;
    p.y += 3.0 * u * tt * c->p2.y;

    p.x += ttt * c->p3.x;
    p.y += ttt * c->p3.y;

    return p;
}

static double cubic2_flatness_sq(const CubicBezier2D *c)
{
    Vec2 a = c->p0;
    Vec2 b = c->p3;
    Vec2 ab = v2_sub(b, a);

    double ab_len = v2_len(ab);
    if (ab_len == 0.0) {
        double d1 = v2_len(v2_sub(c->p1, a));
        double d2 = v2_len(v2_sub(c->p2, a));
        double d = d1 > d2 ? d1 : d2;
        return d * d;
    }

    double abx = ab.x;
    double aby = ab.y;

    double dist_sq(Vec2 p){
        Vec2 ap = v2_sub(p, a);
        double cross = ap.x * aby - ap.y * abx;
        double d = cross / ab_len;
        return d * d;
    }

    double d1 = dist_sq(c->p1);
    double d2 = dist_sq(c->p2);
    return d1 > d2 ? d1 : d2;
}

static void cubic2_subdivide(const CubicBezier2D *c,
                             CubicBezier2D *left,
                             CubicBezier2D *right)
{
    Vec2 p01  = v2_scale(v2_add(c->p0, c->p1), 0.5);
    Vec2 p12  = v2_scale(v2_add(c->p1, c->p2), 0.5);
    Vec2 p23  = v2_scale(v2_add(c->p2, c->p3), 0.5);

    Vec2 p012 = v2_scale(v2_add(p01, p12), 0.5);
    Vec2 p123 = v2_scale(v2_add(p12, p23), 0.5);

    Vec2 p0123 = v2_scale(v2_add(p012, p123), 0.5);

    left->p0 = c->p0;
    left->p1 = p01;
```

```
    left->p2 = p012;
    left->p3 = p0123;

    right->p0 = p0123;
    right->p1 = p123;
    right->p2 = p23;
    right->p3 = c->p3;
}

/* ----------------- Vec2 dynamic buffer and flattening ----------------- */

typedef struct {
    Vec2 *data;
    int   count;
    int   cap;
} Vec2Buffer;

static void buf2_init(Vec2Buffer *b){ b->data = NULL; b->count = b->cap = 0; }
static void buf2_free(Vec2Buffer *b){ free(b->data); b->data = NULL; b->count = b->cap =

static int buf2_push(Vec2Buffer *b, Vec2 p)
{
    if (b->count >= b->cap) {
        int ncap = b->cap ? b->cap * 2 : 64;
        Vec2 *nd = (Vec2*)realloc(b->data, (size_t)ncap * sizeof(Vec2));
        if (!nd) return 0;
        b->data = nd;
        b->cap  = ncap;
    }
    b->data[b->count++] = p;
    return 1;
}

static int flatten_cubic2_rec(const CubicBezier2D *c,
                              double tol_sq,
                              Vec2Buffer *out)
{
    if (cubic2_flatness_sq(c) <= tol_sq) {
        return buf2_push(out, c->p3);
    } else {
        CubicBezier2D l, r;
        cubic2_subdivide(c, &l, &r);
        if (!flatten_cubic2_rec(&l, tol_sq, out)) return 0;
        if (!flatten_cubic2_rec(&r, tol_sq, out)) return 0;
        return 1;
    }
}

static int flatten_path2(const BezierPath2D *path,
                         double tol,
                         Polyline2D *poly_out)
{
    Vec2Buffer buf;
    buf2_init(&buf);
    double tol_sq = tol * tol;
```

```
    if (path->count <= 0) {
        poly_out->count = 0;
        poly_out->pt = NULL;
        return 1;
    }

    if (!buf2_push(&buf, path->seg[^4_0].p0)) {
        buf2_free(&buf);
        return 0;
    }

    for (int i = 0; i < path->count; ++i) {
        CubicBezier2D c = path->seg[i];
        if (!flatten_cubic2_rec(&c, tol_sq, &buf)) {
            buf2_free(&buf);
            return 0;
        }
    }

    if (buf.count > 0) {
        Vec2 first = buf.data[^4_0];
        Vec2 last  = buf.data[buf.count - 1];
        if (fabs(first.x - last.x) > 1e-9 || fabs(first.y - last.y) > 1e-9) {
            if (!buf2_push(&buf, first)) {
                buf2_free(&buf);
                return 0;
            }
        }
    }

    poly_out->count = buf.count;
    poly_out->pt    = buf.data;
    return 1;
}

/* ----------------- Polyline arclength/resampling ----------------- */

static double *poly2_arclen(const Polyline2D *pl)
{
    if (pl->count <= 0) return NULL;
    double *s = (double*)malloc((size_t)pl->count * sizeof(double));
    if (!s) return NULL;
    s[^4_0] = 0.0;
    for (int i = 1; i < pl->count; ++i) {
        Vec2 d = v2_sub(pl->pt[i], pl->pt[i-1]);
        s[i] = s[i-1] + v2_len(d);
    }
    return s;
}

static Vec2 poly2_sample_at(const Polyline2D *pl,
                            const double *s,
                            double d)
{
    int n = pl->count;
    double total = s[n-1];
```

```
    if (total <= 0.0) return pl->pt[^4_0];

    d = fmod(d, total);
    if (d < 0.0) d += total;

    int i = 0;
    while (i+1 < n && s[i+1] < d) ++i;

    int j = (i+1 < n) ? i+1 : i;
    double si = s[i];
    double sj = s[j];
    double t;
    if (sj <= si) t = 0.0;
    else t = (d - si) / (sj - si);

    Vec2 a = pl->pt[i];
    Vec2 b = pl->pt[j];
    return v2_add(a, v2_scale(v2_sub(b, a), t));
}

static int resample_polyline2(const Polyline2D *pl,
                              int N,
                              Polyline2D *out)
{
    if (N <= 0 || pl->count <= 1) {
        out->count = 0;
        out->pt = NULL;
        return 1;
    }

    double *s = poly2_arclen(pl);
    if (!s) return 0;

    Vec2 *pts = (Vec2*)malloc((size_t)N * sizeof(Vec2));
    if (!pts) { free(s); return 0; }

    double total = s[pl->count - 1];
    for (int i = 0; i < N; ++i) {
        double d = total * ((double)i / (double)N);
        pts[i] = poly2_sample_at(pl, s, d);
    }

    free(s);
    out->count = N;
    out->pt    = pts;
    return 1;
}

/* ----------------- Alignment and mapping ----------------- */

static double cyclic_cost2(const Vec2 *A, const Vec2 *B, int n, int shift)
{
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int j = i + shift;
        if (j >= n) j -= n;
```

```
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
    }
    return c;
}

static double cyclic_cost2_rev(const Vec2 *A, const Vec2 *B,
                               int n, int shift)
{
    double c = 0.0;
    for (int i = 0; i < n; ++i) {
        int bi = i + shift;
        if (bi >= n) bi -= n;
        int j = n - 1 - bi;
        double dx = A[i].x - B[j].x;
        double dy = A[i].y - B[j].y;
        c += dx*dx + dy*dy;
    }
    return c;
}

static void align_closed2(const Vec2 *A, const Vec2 *B, int n,
                          int *out_shift, int *out_reverse)
{
    double best_cost = 1e300;
    int best_shift = 0;
    int best_rev = 0;

    for (int shift = 0; shift < n; ++shift) {
        double c0 = cyclic_cost2(A, B, n, shift);
        if (c0 < best_cost) {
            best_cost = c0;
            best_shift = shift;
            best_rev   = 0;
        }
        double c1 = cyclic_cost2_rev(A, B, n, shift);
        if (c1 < best_cost) {
            best_cost = c1;
            best_shift = shift;
            best_rev   = 1;
        }
    }
    *out_shift   = best_shift;
    *out_reverse = best_rev;
}

static Vec2 *build_aligned2(const Vec2 *B, int n,
                            int shift, int reverse)
{
    Vec2 *B2 = (Vec2*)malloc((size_t)n * sizeof(Vec2));
    if (!B2) return NULL;

    if (!reverse) {
        for (int i = 0; i < n; ++i) {
            int j = i + shift;
```

```
            if (j >= n) j -= n;
            B2[i] = B[j];
        }
    } else {
        for (int i = 0; i < n; ++i) {
            int bi = i + shift;
            if (bi >= n) bi -= n;
            int j = n - 1 - bi;
            B2[i] = B[j];
        }
    }
    return B2;
}

typedef struct {
    int   N;
    Vec2 *A;
    Vec2 *B;
} ShapeMapping2D;

static void free_shape_mapping2(ShapeMapping2D *m)
{
    if (!m) return;
    free(m->A);
    free(m->B);
    m->A = m->B = NULL;
    m->N = 0;
}

static int choose_sample_count2(int ctrl_src, int ctrl_dst, double tol)
{
    (void)tol;
    int base = (ctrl_src > ctrl_dst) ? ctrl_src : ctrl_dst;
    int k = 4;
    int N = base * k;
    if (N < 32) N = 32;
    return N;
}

static int build_shape_mapping2(const BezierPath2D *src,
                                const BezierPath2D *dst,
                                double flat_tol,
                                ShapeMapping2D *out)
{
    Polyline2D pl_src, pl_dst;

    if (!flatten_path2(src, flat_tol, &pl_src)) return 0;
    if (!flatten_path2(dst, flat_tol, &pl_dst)) {
        free(pl_src.pt);
        return 0;
    }

    int ctrl_src = src->count * 4;
    int ctrl_dst = dst->count * 4;

    int N = choose_sample_count2(ctrl_src, ctrl_dst, flat_tol);
```

```
    Polyline2D rs_src, rs_dst;
    if (!resample_polyline2(&pl_src, N, &rs_src)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        return 0;
    }
    if (!resample_polyline2(&pl_dst, N, &rs_dst)) {
        free(pl_src.pt);
        free(pl_dst.pt);
        free(rs_src.pt);
        return 0;
    }

    free(pl_src.pt);
    free(pl_dst.pt);

    int shift, rev;
    align_closed2(rs_src.pt, rs_dst.pt, N, &shift, &rev);

    Vec2 *A = rs_src.pt;
    Vec2 *B = build_aligned2(rs_dst.pt, N, shift, rev);
    free(rs_dst.pt);
    if (!B) {
        free(A);
        return 0;
    }

    out->N = N;
    out->A = A;
    out->B = B;
    return 1;
}

static Polyline2D tween_polyline2(const ShapeMapping2D *m, double t)
{
    Polyline2D pl;
    pl.count = m->N + 1;
    pl.pt = (Vec2*)malloc((size_t)pl.count * sizeof(Vec2));
    if (!pl.pt) {
        pl.count = 0;
        return pl;
    }
    double omt = 1.0 - t;
    for (int i = 0; i < m->N; ++i) {
        Vec2 a = m->A[i];
        Vec2 b = m->B[i];
        Vec2 p;
        p.x = omt * a.x + t * b.x;
        p.y = omt * a.y + t * b.y;
        pl.pt[i] = p;
    }
    pl.pt[m->N] = pl.pt[^4_0];
    return pl;
}
```

```
/* ----------------- Loft: lifting 2D slices to 3D ----------------- */

typedef struct {
    int   count;
    Vec3 *pt;
} Polyline3D;

static void free_polyline2(Polyline2D *pl)
{
    if (!pl) return;
    free(pl->pt);
    pl->pt = NULL;
    pl->count = 0;
}

static void free_polyline3(Polyline3D *pl)
{
    if (!pl) return;
    free(pl->pt);
    pl->pt = NULL;
    pl->count = 0;
}

static Polyline3D lift_polyline_to_plane(const Polyline2D *pl,
                                         const PlaneFrame *plane)
{
    Polyline3D out;
    out.count = pl->count;
    out.pt = (Vec3*)malloc((size_t)out.count * sizeof(Vec3));
    if (!out.pt) {
        out.count = 0;
        return out;
    }

    Mat4 T = plane_to_world(plane);
    for (int i = 0; i < pl->count; ++i) {
        Vec2 q = pl->pt[i];
        Vec3 p = {q.x, q.y, 0.0};
        out.pt[i] = mat4_mul_point(T, p);
    }
    return out;
}

/* ----------------- Camera/projection (for SDL view) ----------------- */

typedef struct {
    Vec3 pos;
    Vec3 forward;
    Vec3 up;
} Camera;

static Mat4 camera_view_matrix(const Camera *cam)
{
    Vec3 f = v3_norm(cam->forward);
    Vec3 r = v3_norm(v3_cross(f, cam->up));
    Vec3 u = v3_cross(r, f);
```

```
    Mat4 V = mat4_identity();

    V.m[^4_0] = r.x; V.m[^4_4] = r.y; V.m[^4_8]  = r.z;
    V.m[^4_1] = u.x; V.m[^4_5] = u.y; V.m[^4_9]  = u.z;
    V.m[^4_2] = -f.x;V.m[^4_6] = -f.y;V.m[^4_10] = -f.z;

    Vec3 t;
    t.x = -v3_dot(r, cam->pos);
    t.y = -v3_dot(u, cam->pos);
    t.z =  v3_dot(f, cam->pos);

    V.m[^4_12] = t.x;
    V.m[^4_13] = t.y;
    V.m[^4_14] = t.z;

    return V;
}

static Mat4 perspective_matrix(double fov_y_deg,
                               double aspect,
                               double z_near,
                               double z_far)
{
    double fov_rad = fov_y_deg * (M_PI / 180.0);
    double f = 1.0 / tan(fov_rad * 0.5);
    Mat4 P;
    for (int i = 0; i < 16; ++i) P.m[i] = 0.0;

    P.m[^4_0]  = f / aspect;
    P.m[^4_5]  = f;
    P.m[^4_10] = (z_far + z_near) / (z_near - z_far);
    P.m[^4_11] = -1.0;
    P.m[^4_14] = (2.0 * z_far * z_near) / (z_near - z_far);
    return P;
}

static int project_point(const Mat4 *VP, Vec3 p,
                         double *x_ndc, double *y_ndc, double *z_ndc)
{
    Vec3 q = mat4_mul_point(*VP, p);
    if (q.z <= 0.0) return 0;
    *x_ndc = q.x;
    *y_ndc = q.y;
    *z_ndc = q.z;
    (void)z_ndc;
    return 1;
}

#ifdef USE_SDL2
static void draw_polyline3d(SDL_Renderer *ren,
                            const Mat4 *VP,
                            const Polyline3D *pl,
                            int w, int h)
{
    if (pl->count < 2) return;
```

```
    for (int i = 0; i < pl->count - 1; ++i) {
        Vec3 a = pl->pt[i];
        Vec3 b = pl->pt[i+1];
        double ax, ay, az;
        double bx, by, bz;
        if (!project_point(VP, a, &ax, &ay, &az)) continue;
        if (!project_point(VP, b, &bx, &by, &bz)) continue;
        int sx1 = (int)((ax * 0.5 + 0.5) * (double)w);
        int sy1 = (int)((-ay * 0.5 + 0.5) * (double)h);
        int sx2 = (int)((bx * 0.5 + 0.5) * (double)w);
        int sy2 = (int)((-by * 0.5 + 0.5) * (double)h);
        SDL_RenderDrawLine(ren, sx1, sy1, sx2, sy2);
    }
}
#endif

/* ----------------- Shape construction: square & circle ----------------- */

static BezierPath2D make_square2(void)
{
    BezierPath2D path;
    path.count = 4;
    path.seg = (CubicBezier2D*)malloc(4 * sizeof(CubicBezier2D));
    double s = 1.0;

    path.seg[^4_0].p0.x = -s; path.seg[^4_0].p0.y = -s;
    path.seg[^4_0].p1.x = -s; path.seg[^4_0].p1.y = -s;
    path.seg[^4_0].p2.x =  s; path.seg[^4_0].p2.y = -s;
    path.seg[^4_0].p3.x =  s; path.seg[^4_0].p3.y = -s;

    path.seg[^4_1].p0.x =  s; path.seg[^4_1].p0.y = -s;
    path.seg[^4_1].p1.x =  s; path.seg[^4_1].p1.y = -s;
    path.seg[^4_1].p2.x =  s; path.seg[^4_1].p2.y =  s;
    path.seg[^4_1].p3.x =  s; path.seg[^4_1].p3.y =  s;

    path.seg[^4_2].p0.x =  s; path.seg[^4_2].p0.y =  s;
    path.seg[^4_2].p1.x =  s; path.seg[^4_2].p1.y =  s;
    path.seg[^4_2].p2.x = -s; path.seg[^4_2].p2.y =  s;
    path.seg[^4_2].p3.x = -s; path.seg[^4_2].p3.y =  s;

    path.seg[^4_3].p0.x = -s; path.seg[^4_3].p0.y =  s;
    path.seg[^4_3].p1.x = -s; path.seg[^4_3].p1.y =  s;
    path.seg[^4_3].p2.x = -s; path.seg[^4_3].p2.y = -s;
    path.seg[^4_3].p3.x = -s; path.seg[^4_3].p3.y = -s;

    return path;
}

/* Circle via 4 cubics. [web:36][web:39] */
static BezierPath2D make_circle2(void)
{
    BezierPath2D path;
    path.count = 4;
    path.seg = (CubicBezier2D*)malloc(4 * sizeof(CubicBezier2D));
```

```
    double r = 1.0;
    double k = 4.0 * (sqrt(2.0) - 1.0) / 3.0;

    path.seg[^4_0].p0.x =  r; path.seg[^4_0].p0.y = 0.0;
    path.seg[^4_0].p1.x =  r; path.seg[^4_0].p1.y =  r * k;
    path.seg[^4_0].p2.x =  r * k; path.seg[^4_0].p2.y =  r;
    path.seg[^4_0].p3.x =  0.0;   path.seg[^4_0].p3.y =  r;

    path.seg[^4_1].p0 = path.seg[^4_0].p3;
    path.seg[^4_1].p1.x = -r * k; path.seg[^4_1].p1.y =  r;
    path.seg[^4_1].p2.x = -r;     path.seg[^4_1].p2.y =  r * k;
    path.seg[^4_1].p3.x = -r;     path.seg[^4_1].p3.y =  0.0;

    path.seg[^4_2].p0 = path.seg[^4_1].p3;
    path.seg[^4_2].p1.x = -r;     path.seg[^4_2].p1.y = -r * k;
    path.seg[^4_2].p2.x = -r * k; path.seg[^4_2].p2.y = -r;
    path.seg[^4_2].p3.x =  0.0;   path.seg[^4_2].p3.y = -r;

    path.seg[^4_3].p0 = path.seg[^4_2].p3;
    path.seg[^4_3].p1.x =  r * k; path.seg[^4_3].p1.y = -r;
    path.seg[^4_3].p2.x =  r;     path.seg[^4_3].p2.y = -r * k;
    path.seg[^4_3].p3.x =  r;     path.seg[^4_3].p3.y =  0.0;

    return path;
}

/* ----------------- Plane frames ----------------- */

static PlaneFrame make_bottom_plane(void)
{
    PlaneFrame pf;
    pf.origin.x = 0.0; pf.origin.y = 0.0; pf.origin.z = 0.0;
    pf.x_axis.x = 1.0; pf.x_axis.y = 0.0; pf.x_axis.z = 0.0;
    pf.y_axis.x = 0.0; pf.y_axis.y = 1.0; pf.y_axis.z = 0.0;
    pf.normal.x = 0.0; pf.normal.y = 0.0; pf.normal.z = 1.0;
    return pf;
}

static PlaneFrame make_top_plane(void)
{
    PlaneFrame pf;
    pf.origin.x = 0.0;
    pf.origin.y = 0.0;
    pf.origin.z = 4.0;

    Vec3 n = {0.2, 0.4, 0.87};
    pf.normal = v3_norm(n);

    Vec3 tmp = {1.0, 0.0, 0.0};
    Vec3 x = v3_sub(tmp, v3_scale(pf.normal, v3_dot(tmp, pf.normal)));
    if (v3_len(x) < 1e-6) {
        tmp.x = 0.0; tmp.y = 1.0; tmp.z = 0.0;
        x = v3_sub(tmp, v3_scale(pf.normal, v3_dot(tmp, pf.normal)));
    }
    pf.x_axis = v3_norm(x);
    pf.y_axis = v3_norm(v3_cross(pf.normal, pf.x_axis));
```

```
    return pf;
}

static PlaneFrame lerp_plane(const PlaneFrame *a,
                             const PlaneFrame *b,
                             double t)
{
    double omt = 1.0 - t;
    PlaneFrame pf;
    pf.origin.x = omt * a->origin.x + t * b->origin.x;
    pf.origin.y = omt * a->origin.y + t * b->origin.y;
    pf.origin.z = omt * a->origin.z + t * b->origin.z;

    pf.x_axis.x = omt * a->x_axis.x + t * b->x_axis.x;
    pf.x_axis.y = omt * a->x_axis.y + t * b->x_axis.y;
    pf.x_axis.z = omt * a->x_axis.z + t * b->x_axis.z;

    pf.y_axis.x = omt * a->y_axis.x + t * b->y_axis.x;
    pf.y_axis.y = omt * a->y_axis.y + t * b->y_axis.y;
    pf.y_axis.z = omt * a->y_axis.z + t * b->y_axis.z;

    pf.normal.x = omt * a->normal.x + t * b->normal.x;
    pf.normal.y = omt * a->normal.y + t * b->normal.y;
    pf.normal.z = omt * a->normal.z + t * b->normal.z;

    pf.x_axis = v3_norm(pf.x_axis);
    pf.y_axis = v3_norm(pf.y_axis);
    pf.normal = v3_norm(pf.normal);

    return pf;
}

/* ----------------- OpenSCAD polyhedron output ----------------- */

/* Build point list and face list for polyhedron based on slices.
 * - S slices (steps), each slice has N_pts points.
 * - Points indexed as idx = slice * N_pts + pt.
 * - Faces: bottom cap, top cap, and side quads split into triangles. [web:70][web:72][we
 */
static void emit_openscad_polyhedron(const Polyline3D *sections,
                                     int steps,
                                     int N_pts)
{
    int total_points = steps * N_pts;

    /* Header and points */
    printf("// Auto-generated loft from C program\n");
    printf("// steps=%d, points_per_slice=%d\n\n", steps, N_pts);
    printf("points = [\n");
    for (int s = 0; s < steps; ++s) {
        for (int i = 0; i < N_pts; ++i) {
            int idx = s * N_pts + i;
            Vec3 p = sections[s].pt[i];
            printf("  [% .6f, % .6f, % .6f]%s\n",
                   p.x, p.y, p.z,
```

```
                    (idx == total_points-1) ? "" : ",");
        }
    }
    printf("];\n\n");

    printf("faces = [\n");

    /* Bottom cap (slice 0): fan around point 0. */
    for (int i = 1; i < N_pts-1; ++i) {
        printf("  [ %d, %d, %d ],\n", 0, i, i+1);
    }

    /* Top cap (slice steps-1): fan, but reverse winding. */
    int base_top = (steps-1) * N_pts;
    for (int i = 1; i < N_pts-1; ++i) {
        printf("  [ %d, %d, %d ],\n",
               base_top, base_top + i+1, base_top + i);
    }

    /* Side faces between slices s and s+1.
     * For each segment i..i+1, form quad: (s,i)->(s,i+1)->(s+1,i+1)->(s+1,i)
     * Then split quad into two triangles. [web:72][web:77]
     */
    for (int s = 0; s < steps-1; ++s) {
        int base0 = s * N_pts;
        int base1 = (s+1) * N_pts;
        for (int i = 0; i < N_pts-1; ++i) {
            int a = base0 + i;
            int b = base0 + i+1;
            int c = base1 + i+1;
            int d = base1 + i;
            printf("  [ %d, %d, %d ],\n", a, b, c);
            printf("  [ %d, %d, %d ],\n", a, c, d);
        }
    }

    printf("];\n\n");
    printf("polyhedron(points = points, faces = faces, convexity = 10);\n");
}

/* ----------------- Main: loft + OpenSCAD output + optional SDL ----------------- */

int main(int argc, char **argv)
{
    (void)argc; (void)argv;

    BezierPath2D path_bottom = make_square2();
    BezierPath2D path_top    = make_circle2();

    ShapeMapping2D map2;
    double flat_tol = 0.002;
    if (!build_shape_mapping2(&path_bottom, &path_top, flat_tol, &map2)) {
        fprintf(stderr, "Failed to build 2D shape mapping\n");
        free(path_bottom.seg);
        free(path_top.seg);
        return 1;
```

```
    }

    PlaneFrame plane_bottom = make_bottom_plane();
    PlaneFrame plane_top    = make_top_plane();

    int steps = 16;
    if (steps < 2) steps = 2;

    Polyline3D *sections = (Polyline3D*)calloc((size_t)steps,
                                               sizeof(Polyline3D));
    if (!sections) {
        free_shape_mapping2(&map2);
        free(path_bottom.seg);
        free(path_top.seg);
        return 1;
    }

    for (int k = 0; k < steps; ++k) {
        double t = (double)k / (double)(steps - 1);
        PlaneFrame pf = lerp_plane(&plane_bottom, &plane_top, t);
        Polyline2D pl2 = tween_polyline2(&map2, t);
        sections[k] = lift_polyline_to_plane(&pl2, &pf);
        free_polyline2(&pl2);
    }

    /* Emit OpenSCAD source describing the loft as a polyhedron. [web:70][web:72] */
    if (sections[^4_0].count > 1) {
        emit_openscad_polyhedron(sections, steps, sections[^4_0].count);
    }

#ifdef USE_SDL2
    /* Optional SDL2 visualization: perspective wireframe of all slices. */
    if (SDL_Init(SDL_INIT_VIDEO) == 0) {
        int width = 800, height = 800;
        SDL_Window *win = SDL_CreateWindow("Loft wireframe",
                                           SDL_WINDOWPOS_CENTERED,
                                           SDL_WINDOWPOS_CENTERED,
                                           width, height,
                                           SDL_WINDOW_SHOWN);
        if (win) {
            SDL_Renderer *ren = SDL_CreateRenderer(win, -1,
                                                   SDL_RENDERER_ACCELERATED |
                                                   SDL_RENDERER_PRESENTVSYNC);
            if (ren) {
                Camera cam;
                cam.pos.x = 6.0; cam.pos.y = 5.0; cam.pos.z = 6.0;
                Vec3 target = {0.0, 0.0, 2.0};
                cam.forward = v3_norm(v3_sub(target, cam.pos));
                cam.up.x = 0.0; cam.up.y = 0.0; cam.up.z = 1.0;

                Mat4 V = camera_view_matrix(&cam);
                Mat4 P = perspective_matrix(60.0,
                                            (double)width / (double)height,
                                            0.1, 50.0);
                Mat4 VP = mat4_mul(P, V);
```

```
                int running = 1;
                while (running) {
                    SDL_Event ev;
                    while (SDL_PollEvent(&ev)) {
                        if (ev.type == SDL_QUIT) running = 0;
                        if (ev.type == SDL_KEYDOWN &&
                            ev.key.keysym.sym == SDLK_ESCAPE) running = 0;
                    }

                    SDL_SetRenderDrawColor(ren, 0, 0, 0, 255);
                    SDL_RenderClear(ren);

                    SDL_SetRenderDrawColor(ren, 160, 160, 160, 255);
                    for (int k = 0; k < steps; ++k) {
                        draw_polyline3d(ren, &VP, &sections[k], width, height);
                    }

                    SDL_SetRenderDrawColor(ren, 80, 255, 80, 255);
                    int N_pts = sections[^4_0].count;
                    for (int i = 0; i < N_pts; ++i) {
                        for (int k = 0; k < steps - 1; ++k) {
                            Vec3 a = sections[k].pt[i];
                            Vec3 b = sections[k+1].pt[i];
                            double ax, ay, az, bx, by, bz;
                            if (!project_point(&VP, a, &ax, &ay, &az)) continue;
                            if (!project_point(&VP, b, &bx, &by, &bz)) continue;
                            int sx1 = (int)((ax * 0.5 + 0.5) * (double)width);
                            int sy1 = (int)((-ay * 0.5 + 0.5) * (double)height);
                            int sx2 = (int)((bx * 0.5 + 0.5) * (double)width);
                            int sy2 = (int)((-by * 0.5 + 0.5) * (double)height);
                            SDL_RenderDrawLine(ren, sx1, sy1, sx2, sy2);
                        }
                    }

                    SDL_RenderPresent(ren);
                }

                SDL_DestroyRenderer(ren);
            }
            SDL_DestroyWindow(win);
        }
        SDL_Quit();
    }
#endif

    for (int k = 0; k < steps; ++k) free_polyline3(&sections[k]);
    free(sections);
    free_shape_mapping2(&map2);
    free(path_bottom.seg);
    free(path_top.seg);

    return 0;
}
```

⁂

1. https://fse.studenttheses.ub.rug.nl/15641/1/thesis.pdf
2. https://www.wisdom.weizmann.ac.il/~ylipman/2015_homotopic_morphing.pdf
3. https://www.sciencedirect.com/science/article/abs/pii/S0020740323008597
4. https://www.sciencedirect.com/science/article/abs/pii/S1047320308000059
5. http://www.cad.zju.edu.cn/home/jqfeng/papers/2D shape morphing via automatic feature matching and hierarchical interpolation.pdf
6. https://www-ui.is.s.u-tokyo.ac.jp/~takeo/papers/rigid.pdf
7. https://faculty.cc.gatech.edu/~turk/my_papers/texture_transfer.pdf
8. https://dl.acm.org/doi/10.1145/3596711.3596731
9. https://www.youtube.com/watch?v=l_o1jg0BO6w
10. https://dl.acm.org/doi/10.1145/344779.344859
11. https://etd.lib.metu.edu.tr/upload/12622285/index.pdf
12. https://docsdrive.com/pdfs/ansinet/itj/2014/2425-2430.pdf
13. http://nishitalab.org/user/nis/cdrom/pg/morphing.pdf
14. http://ieeexplore.ieee.org/document/883958/
15. http://gamma.cs.unc.edu/3DMORPHING/features.html
16. https://perso.uclouvain.be/paul.vandooren/publications/SamirVLGA10.pdf
17. https://www.tandfonline.com/doi/full/10.1080/10106049.2023.2197516
18. https://www.sciencedirect.com/science/article/abs/pii/S0097849309000387
19. https://www.cs.tau.ac.il/~levin/arap.pdf
20. https://stackoverflow.com/questions/1734745/how-to-create-circle-with-bézier-curves
21. https://en.wikipedia.org/wiki/Bézier_curve
22. https://stackoverflow.com/questions/11737988/how-to-draw-a-line-using-sdl-without-using-external-libraries
23. https://freepascal-meets-sdl.net/chapter-5-drawing-primitives/
24. https://gigi.nullneuron.net/gigilabs/drawing-lines-with-sdl2/
25. https://wiki.libsdl.org/SDL2/FAQLinux
26. https://manpages.debian.org/testing/libsdl2-dev/sdl2-config.1.en.html
27. https://discourse.libsdl.org/t/sdl2-simple-sdl-render-graphics-on-unix-like-os-with-c-or-c-style-c/25016
28. https://stackoverflow.com/questions/67233475/compile-an-sdl-project-using-gcc
29. https://www.reddit.com/r/C_Programming/comments/34f2uc/printing_text_line_by_line_in_an_sdl_window/
30. https://www.youtube.com/watch?v=fUn1lUWPUWs
31. https://discourse.libsdl.org/t/3d-in-sdl/41757
32. https://forums.codeblocks.org/index.php?topic=26009.0
33. https://www.m-hikari.com/ams/ams-2017/ams-25-28-2017/p/ahmadAMS25-28-2017.pdf
34. https://www.youtube.com/watch?v=3PlkQLGUhk0
35. https://github.com/libsdl-org/SDL/issues/8244

36. https://www.mathworks.com/matlabcentral/fileexchange/6844-approximation-of-circle-using-cubic-bezier-curve?s_tid=prof_contriblnk
37. https://www.reddit.com/r/sdl/comments/17sgzjl/how_do_i_crosscompile_source_code_for_windows_on/
38. https://progbook.org/sdl.html
39. https://www.youtube.com/watch?v=9P_MAgSNZx8
40. https://openscad.org/cheatsheet/
41. https://www.cse.iitd.ac.in/~narain/courses/2402-col781/slides/04-perspective.pdf
42. https://arxiv.org/html/2406.08960v1
43. https://www.youtube.com/watch?v=6FXFmyp0aBo
44. https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Primitive_Solids
45. https://homehack.nl/openscad-polygon-and-polyhedron/
46. https://www.instructables.com/Loft-in-OpenSCAD/
47. https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/The_OpenSCAD_Language
48. https://www.reddit.com/r/openscad/comments/o68cn9/unable_to_render_polyhedron_please_help/
49. https://github.com/openscad/openscad/issues/3411
50. https://openscad.rocklinux.narkive.com/ULzklbrm/polygon-offset-function
51. https://spolearninglab.com/curriculum/lessonPlans/hacking/resources/software/3d/openscad/openscad_polygons.html