CS184/284A Spring 2025 Homework 2 Write-Up

Names:

Link to webpage: (TODO) https://cal-cs184.github.io/hw-webpages-su25-Luke-liyy/hw2/
Link to GitHub repository: (TODO) chttps://cal-cs184.github.io/hw-webpages-su25-Luke-liyy/

Overview

This assignment walks through several cornerstone algorithms in computer graphics for handling curves, surfaces, and triangle meshes. I implemented six major components:

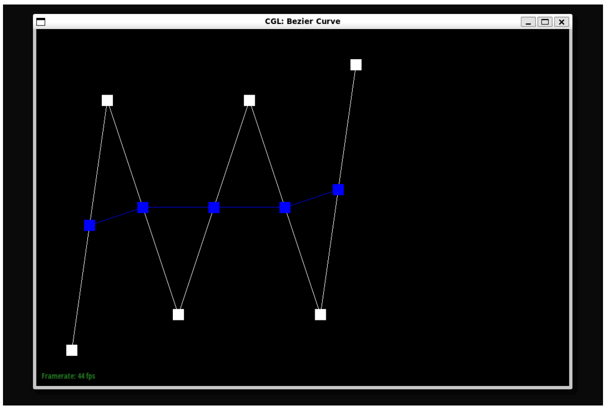
- **Bézier curves and surfaces**: Implemented 1-D de Casteljau evaluation for Bézier curves and extended it to two dimensions to evaluate any point on a bicubic Bézier patch.
- **Half-edge data structure**: Built a full half-edge mesh representation and utilities for navigating and modifying connectivity.
- **Area-weighted vertex normals**: Produced smooth per-vertex normals by summing adjacent face normals weighted by triangle area.
- **Local topology edits**: Added reversible edge flip and edge split operations that keep all half-edge pointers consistent.
- **Loop subdivision**: Completed the four-pass Loop subdivision pipeline to upsample arbitrary manifold meshes and analysed failure cases on the Utah teapot.

Section I: Bezier Curves and Surfaces

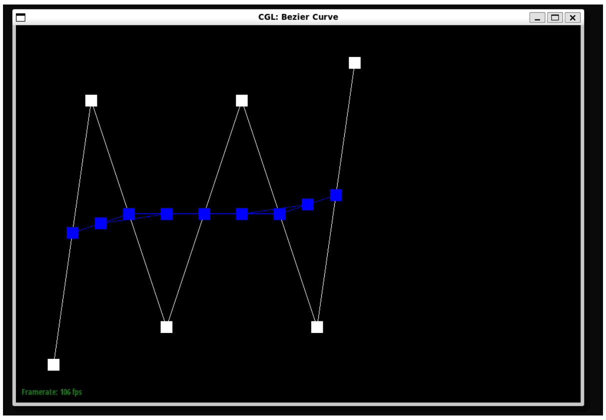
Part 1: Bezier curves with 1D de Casteljau subdivision

de Casteljau evaluates a Bézier curve at a parameter $t \in [0, 1]$ by repeated linear interpolation of its control points.

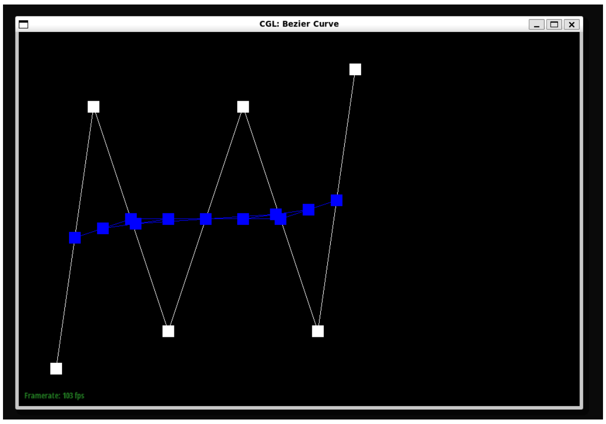
For a control-point list $\{P_0, \ldots, P_n\}$, define $P_i^{(1)} = (1-t)P_i^{(0)} + tP_{i+1}^{(0)}$ for $i = 0 \ldots n-1$.
Repeat on that list until a single point $P_0^{(n)}$ remains; this point $B(t)$ lies on the curve.



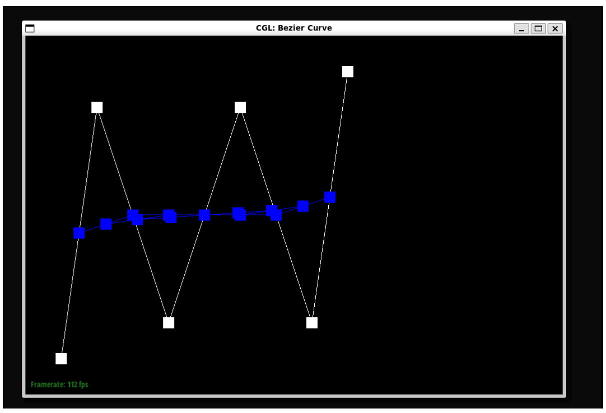
1



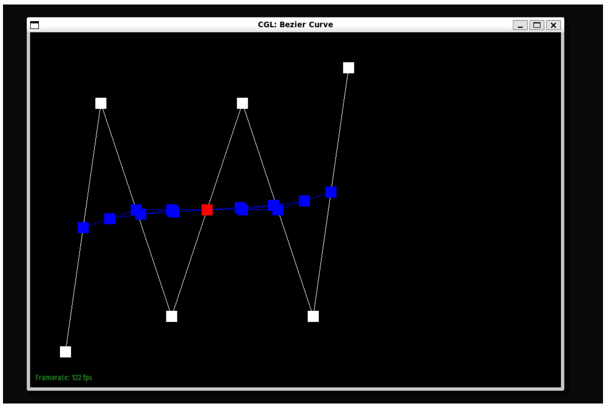
2



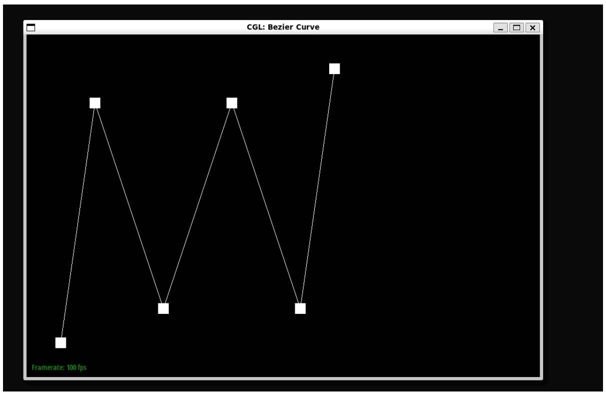
3



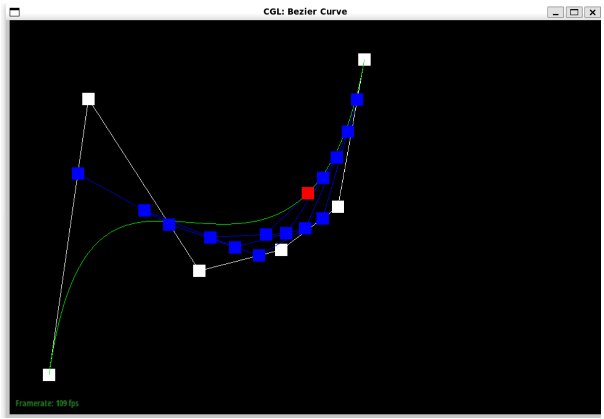
4



5



control-point



mocp

Part 2: Bezier surfaces with separable 1D de Casteljau

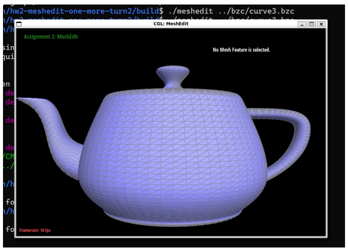
Algorithm Overview

1. For a bicubic Bézier surface patch you have a 4 × 4 grid of control points $P_{ij},\, i,j \in \{0,\ldots,3\}$.
2. Fix v and, for each of the four rows $(P_{0j}, P_{1j}, P_{2j}, P_{3j})$, apply the 1-D de Casteljau procedure with parameter u to collapse the row to a single point $Q_j(u)$.
3. These four intermediate points form a Bézier curve in the v-direction. Treat them as control points and run the 1-D algorithm again, now with

parameter v, to obtain the final surface position $S(u, v)$.

Code Implementation

- `evaluateStep`: Performs a single linear-interpolation layer on a 1-D list of points.
- `evaluate1D`: Repeatedly calls `evaluateStep` until the control polygon is collapsed to a single point.
- `evaluate`: Combines both dimensions to compute the final surface point.



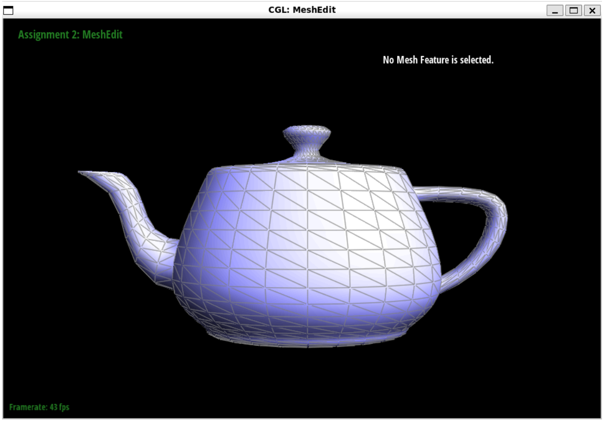
teapot

Section II: Triangle Meshes and Half-Edge Data Structure

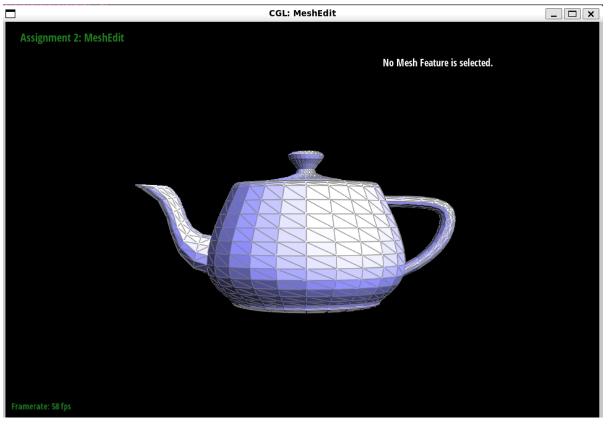
Part 3: Area-weighted vertex normals

How the area-weighted vertex normal is computed:

1. Loop around the one-ring of faces starting from an outgoing half-edge `hStart = halfedge()`. Repeatedly follow `h = h->twin()->next()` to move counter-clockwise until returning to `hStart`. This visits every face that contains the vertex.
2. For each face, let `p0 = h->vertex->position`, `p1 = h->next->vertex->position`, `p2 = h->next->next->vertex->position`. Compute `faceNormal = cross(p1 - p0, p2 - p0)`. Since its length equals twice the face area, summing these vectors automatically performs area weighting: `nSum += faceNormal`.
3. After the loop, if `nSum` is non-zero, return `nSum.unit()`; otherwise return the zero vector unchanged.



with vertex normals

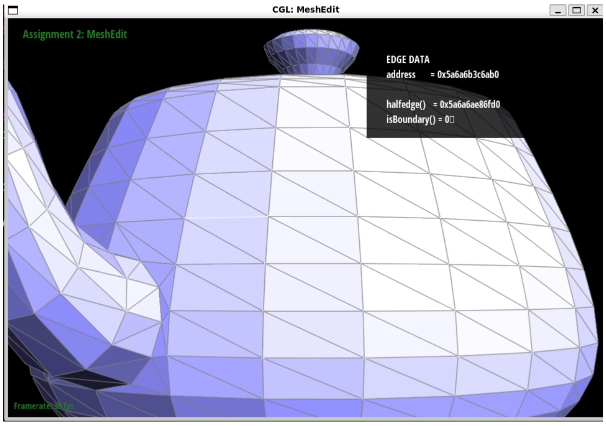


without vertex normals

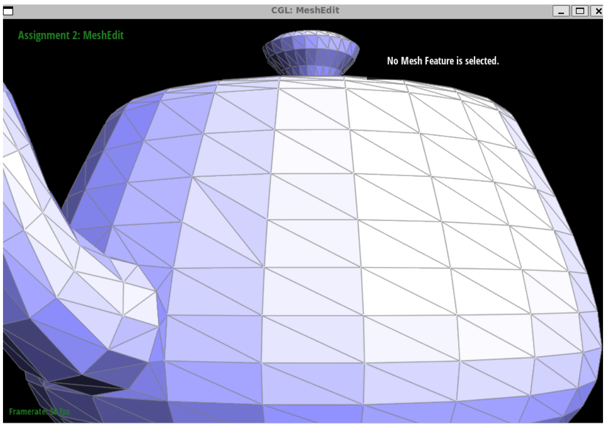
Part 4: Edge flip

Edge-flip procedure:

1. **Boundary guard** – Exit immediately if the target edge is on the boundary, because it has only one incident triangle.
2. **Cache handles** – Collect the six half-edges of the two incident faces (`h0…h5`), their four vertices (`v0…v3`), the two faces (`f0, f1`), and the edge itself (`e0`).
3. **Redirect the diagonal** – Change the start vertices of the two half-edges that form the diagonal. After this single step the geometric diagonal is already (`v2, v3`).
4. **Re-wire topology** – With six `setNeighbors` calls assign `next`, `twin`, `face`, `edge`, and `vertex` for each half-edge so that
   - new face `f0′` has ring `v2 → v3 → v1` (`h0, h5, h2`)
   - new face `f1′` has ring `v3 → v2 → v0` (`h1, h3, h4`)
5. **Update anchors** – Pick one interior half-edge per element as its handle and return the original edge handle (`e0`); its identity is unchanged even though it now spans the new vertices (`v2, v3`).



before flip



after flip

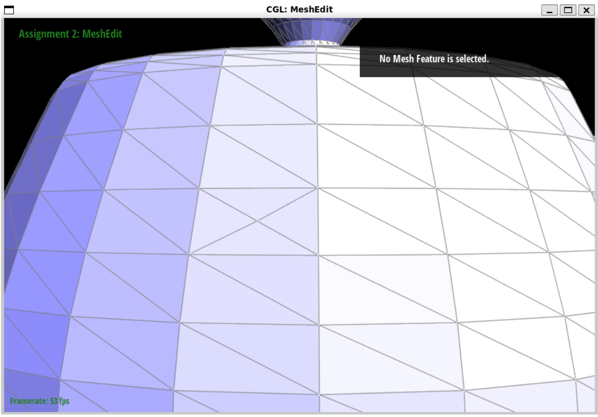
Part 5: Edge split

Edge-split implementation in a nutshell:

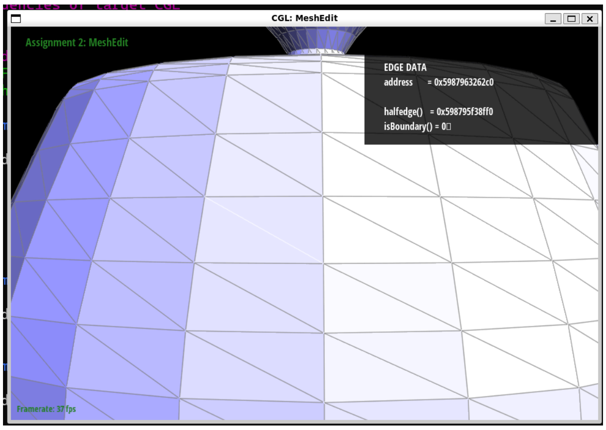
1. **Collect the old neighborhood** – First cache the six half-edges that form the two incident triangles of the target edge `e0` (`h0` - `h5`), their four corner vertices `v0` - `v3`, and the two faces `f0, f1`.
2. **Insert the midpoint vertex and three brand-new edges** – Allocate a new vertex `vm` at the average of the endpoints ((`v0+v1`)/2). Create three fresh edges (`eA, eB, eC`) with their twin half-edges (`h6` - `h11`) and set the `isNew` flag for later Loop subdivision passes.
3. **Re-wire half-edge connectivity**
   - *Twins & edges*: pair all new half-edges as twins and hook them to their edge objects.
   - *Next rings*: rebuild the `next` cycles so that each of the four resulting triangles has a consistent CCW loop.
   - *Vertices & faces*: redirect the `vertex()` and `face()` pointers, ensuring every element stores one of its half-edges.
4. **Update top-level handles** – Point each original vertex's `halfedge()` to an outgoing half-edge that still references it, and set `vm->halfedge()` to the half-

edge that runs along the old edge direction (required by the spec).

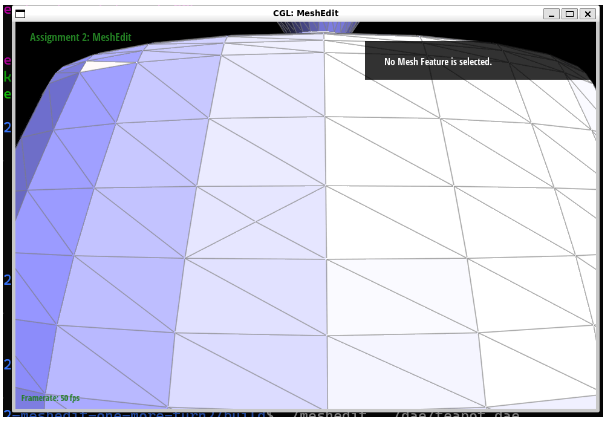
5. **Return the new vertex** – The function finally returns `vm`, giving callers a convenient handle to the midpoint.



5-split



5



5-split-flip

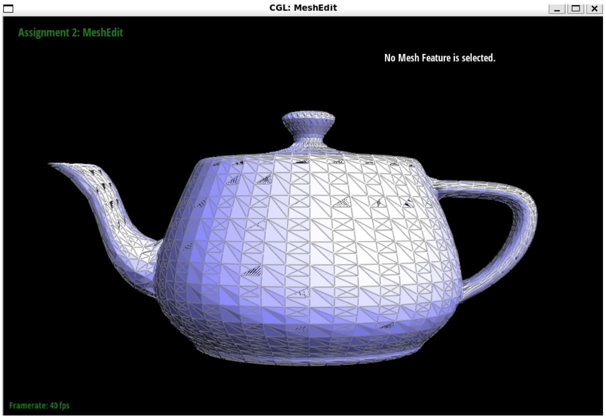
Part 6: Loop subdivision for mesh upsampling (not finished)

Loop-Subdivision Implementation (brief overview)

1. **Pass 1 – pre-compute new positions without touching the mesh topology**
   - For every old vertex, store `newPosition` using the standard Loop masks:

        - *interior vertex* → weighted average of itself and all one-ring neighbours ($u = \frac{3}{16}$ for valence 3, $u = \frac{3}{8n}$ otherwise);
        - *boundary vertex* → $\frac{3}{4}v + \frac{1}{8}(n_0 + n_1)$.
    - For every edge, store its `newPosition` as either
        - the midpoint (boundary edge), or
        - the Loop edge rule $\frac{3}{8}(A+B) + \frac{1}{8}(C+D)$ (interior).
2. **Pass 2 – topology refinement**
    - Iterate over the original edge list once (captured by `num_original_edges`).
    - Split each edge; the newly created midpoint vertex inherits the edge's pre-computed `newPosition`.
    - Tag elements created by splitting as `isNew = true` so we can recognise them later.
3. **Pass 3 – edge flips for odd/even pattern**
    - Traverse all edges: if an edge is *new* and its two incident vertices do not share the same "newness", flip it.
    - This produces the canonical Loop connectivity where every old–new edge becomes the diagonal of the "kite".
4. **Pass 4 – final geometry update**
    - Overwrite the position of every vertex with its buffered `newPosition`, completing the geometric smoothing.

loop



pressL

When Loop subdivision was applied to the Utah teapot, two distinct failure modes emerged. The first was geometric distortion: after the initial subdivision the surface erupted into spikes and cross-shaped slivers, and although further refinement was possible, the artefacts grew progressively worse. The second was a topological breakdown: no proper 4-to-1 splits appeared and a second subdivision attempt drove the program into an infinite loop or outright crash.