CS184/284A Spring 2025 Homework 2 Write-Up

Names: Armen Hanissian, Aidan Gould

Link to webpage: https://cal-cs184-student.github.io/hw-webpages-armen-aidan-1/hw2/index.html
Link to GitHub repository: https://github.com/cal-cs184-student/hw-webpages-armen-aidan-1

Overview

This is a report for Homework 2 of CS 284. Here, we explore two ideas key to 3D graphics and design: Bézier surfaces for realistic smooth curves and triangle mesh representations for 3D geometries. Combining these tools with lighting models creates realistic virtual renderings and the ability to manipulate them.

It was surprising to us how simple the algorithms are in practice and how it is possible to traverse the elements of the mesh in any order when doing global mesh editing operations. We frequently interact with complex meshes in our own research in computational engineering, so it was interesting to finally see under the hood of a mesh's data structure.

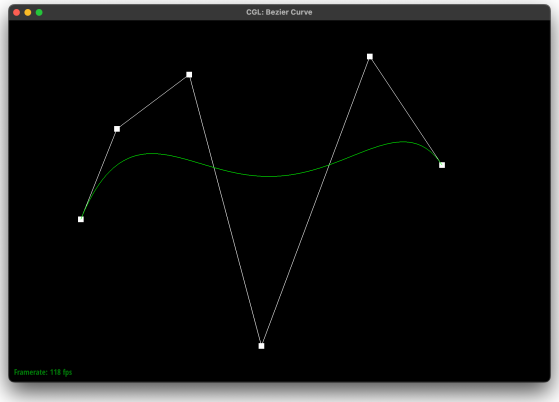
Section I: Bézier Curves and Surfaces

Part 1: Bézier curves with 1D de Casteljau subdivision

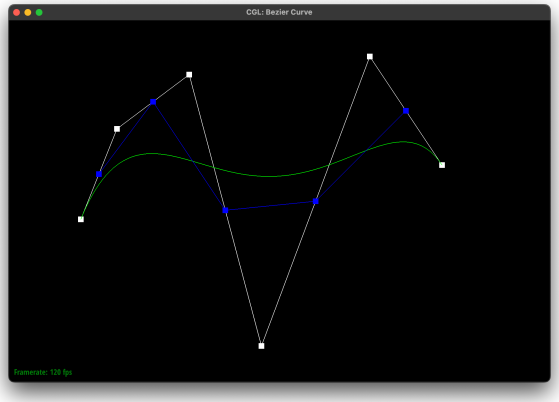
De Casteljau's Algorithm is a recursive algorithm used to evaluate a Bézier curve at the point corresponding to parameter t from its control points. It works by recursively splitting the sides of the curve's control polygon by the ratio given by t, then connecting these split points to form a lower-order control polygon until arriving at a single point.

We implemented this by writing the function `evaluateStep`,

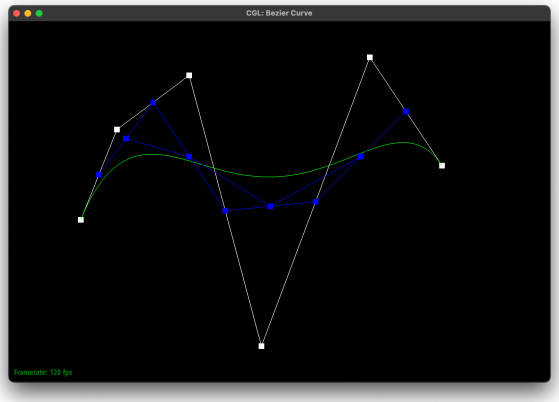
which takes a vector of 2-D control points as input and returns the control points for the following level of de Casteljau subdivision. It works by pushing the `lerp` of the points p_i and p_{i+1} for $i = 1, 2, \ldots, n-1$ onto the vector returned by the function. This function is called recursively within `render()` and `drawCurve()` to draw the loaded Bézier curve to the GUI.



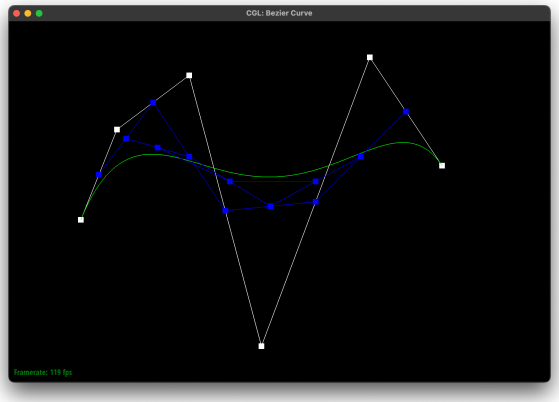
No subdivisions



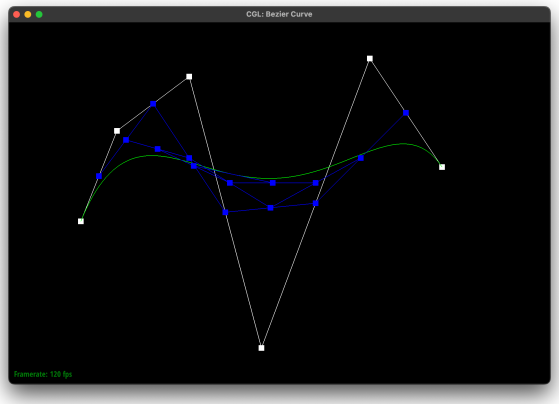
One subdivision



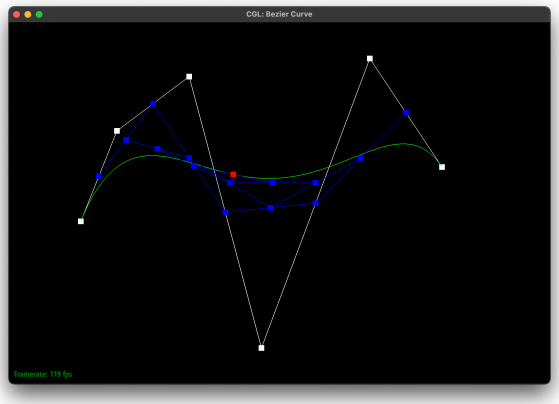
Two subdivisions



Three subdivisions

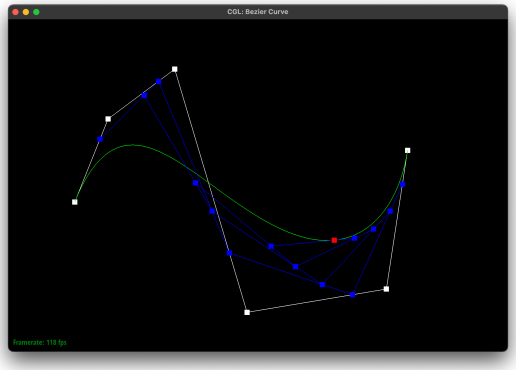


Four subdivisions



Five subdivisions

De Casteljau's algorithm for a quintic Bézier curve.

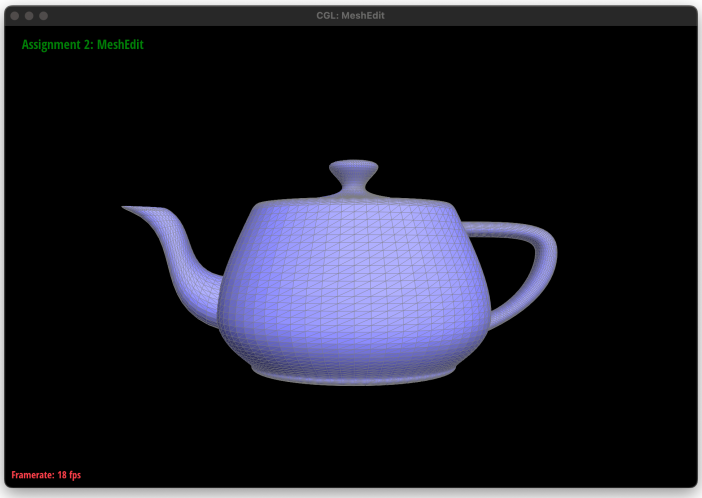


Modified Bézier curve

Part 2: Bézier surfaces with separable 1D de Casteljau

The De Casteljau algorithm extends to Bézier surfaces by evaluating an n-by-n grid of control points in two dimensions at the parameters u, v to form a 2-D surface in 3-D space. Each row of the matrix of control points is evaluated at the parameter u using the 1D de Casteljau subdivision, resulting in n three-dimensional points. These are used as the control points for a final 1D de Casteljau subdivision evaluated at the parameter v, resulting in a single three dimensional point lying on the surface.

To implement this, we completed the function `evaluate1D()` which works similarly to `evaluateStep()` except that it returns the full evaluation of the Bézier curve at the parameter t as a 3D point. This works by recursively calling `evaluateStep` until a single point is returned. The primary function `evaluate()` evaluates a Bézier surface at the parameters u, v. It works by evaluating the rows of the control point matrix at the parameter u using `evaluate1D`, pushing these onto a vector of points `std::vector upts`, then returning the output of `evaluate1D` with `upts` as the input control points and v as the parameter to evaluate at.



Teapot rendered from .bez file.

Section II: Triangle Meshes and Half-Edge Data Structure

Part 3: Area-weighted vertex normals

The algorithm for the weighted vertex normals starts with edge associated with the vertex making the call (v_0). Using next() twice, we collect the positions of the two other vertices which make up the first triangle face, calling them v_1 and v_2. Then the weighted normal vector is calculated as

$$N_{face} = \frac{(v_1 - v_0) \times (v_2 - v_0)}{2}$$

The cross product makes a normal vector whose magnitude is equal to the area of the parallelogram defined by the two vectors. Half of that area makes the magnitude equal that of the corresponding triangle instead. The next face is then found using twin(). The normal from each face is summed and the result is normalized. The results below show a much smoother teapot shading with vertex normals turned on!

Teapot without vertex normals.

Teapot with vertex normals.

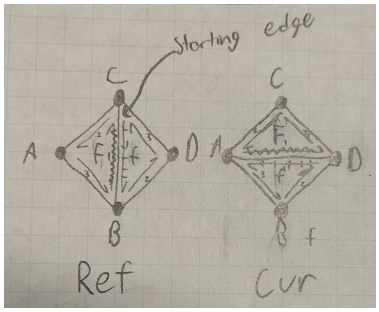
Part 4: Edge flip

The edge flip algorithm works by first collecting all of the needed iterators, then performing reassignment as necessary to make the flip. Here, following convention from continuum mechanics (we are mechanical engineers), we call the untransformed state the "current configuration" and the transformed state the "reference configuration".

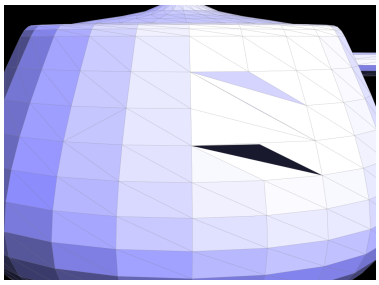
As shown below, each vertex is named A-D and the two faces are named F and f. Each half edge is named after its corresponding face in the reference configuration followed by 1-3 (for example, one half edge is named f2). The structure is traversed along the half edges using $next()$ and $twin()$ to collect all of the iterators.

Reassignment is fairly straightforward. Each object simply needs to point to its new correspondences in the current configuration. For an edge flip, most pointers are already correct so we only reassign where necessary.

Careful planning and a good naming convention meant that the code worked on the first build. We tested it by flipping many edges in the same area on both the teapot and cube test files. It will sometimes create a degenerate case where the structure was completely flattened (as expected), but it never caused an error or made holes in the mesh. We show one of our tests on the teapot mesh below.



Labeling convention for edge flip.


Teapot with a single flip on the left and many flips in an area on the right.

Part 5: Edge split

The edge split algorithm also works by first collecting all of the needed iterators, then performing reassignment as necessary to make the flip. Again, following convention from continuum mechanics, we call the untransformed state the "current configuration" and the transformed state the "reference configuration".

As shown in the figure below, each vertex is named A-D and the two faces are named w-x. Each half edge is named after its corresponding face in the current configuration followed by 1-3 (for example, one half edge is named x2). Edges are named after their corresponding faces, so we have an edge named Ewx for example. The structure is traversed along the half edges using `next()` and `twin()` in the current configuration to collect all of the preexisting iterators. All but two of the named half edges in the reference configuration are the same as in the current configuration. For the two that are not, they are reassigned after collection to their new name in the current config. New objects are also created, all named for the current config.

The reassignment code is longer for splits than flips but no more complex. Each object simply needs to point to its new correspondences in the current configuration.

Careful planning and a good naming convention meant that the code worked on the first build. We tested it by splitting

and flipping many edges in the same area on the teapot mesh. It never caused an error or made holes in the mesh. Below we show one of our split tests on the teapot mesh and another test which includes a combination of flips and splits.

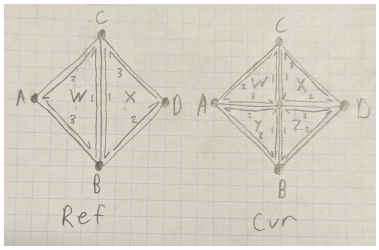
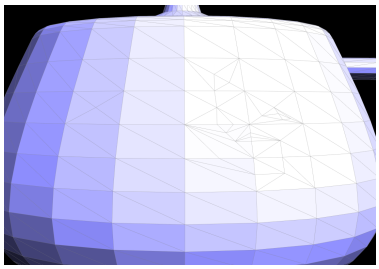
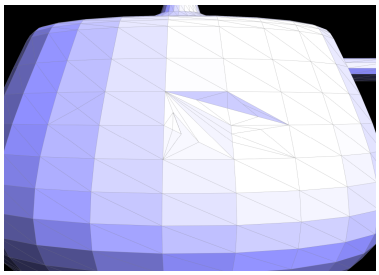


Diagram of our naming convention with the reference and current configurations shown.


Teapot with a single split on the left and many splits in an area on the right.


Teapot with many splits and flips.

Part 6: Loop subdivision for mesh upsampling

To implement loop subdivision for mesh upsampling we took the following approach:

1. **Compute new positions for all of the original vertices of the input mesh.** The updated position is a function of the vertex degree n, the original position of the vertex, and the sum of positions of the neighboring vertices, and the weighting factor $u = \frac{3}{16}$ if $n = 3$, else $u = \frac{3}{8n}$. At this point also set the vertices' `isNew` flag to `false` since they are from the original mesh.
2. **Compute the subdivision vertex positions and save them for later.** This is a weighted sum of four

vertices; two on the split edge and two orthogonal to the split edge. We access these vertices by looping through all edges in the mesh and traversing to each one via halfedge iterators. Each edge's split vertex is stored in `e->newPosition`. We also created a new flag `isOrig` which we set equal to true for these unmodified edges.

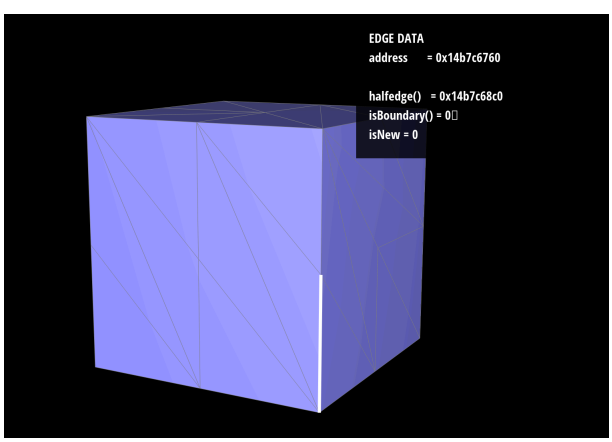
3. **Split the edges in the mesh.** To get this part to work we introduced a flag `isOrig` which will return `false` for an edge that has been split or was generated by a split. Looping through all edges, we first store an iterator to the subsequent edge, split the edge if `isOrig == true`, flag the returned vertex `v->isNew`, and update the `isNew` and `isOrig` flags accordingly for the four subdivision edges. `isNew` is set to `false` for the two segments of the original edge and `true` for the newly generated edges, which cut across the subdivided triangle. Also, copy `e->newPosition` into the newly created vertex, so that we can easily access it when assigning vertex positions.
4. **Flip new edges connecting a new vertex and an old vertex.** Looping through edges, check if `e->isNew`, grab its vertices `v1` and `v2`, and flip the edge if `v1->isNew != v2->isNew`. This is the logical equivalent to XOR, which will return false if both vertices are old or are new.
5. **Assign the original vertices and subdivision vertices their new positions.** Looping through all vertices, simply set each vertex's position equal to `v->newPosition` which we assigned in steps 1 and 3.

Debugging

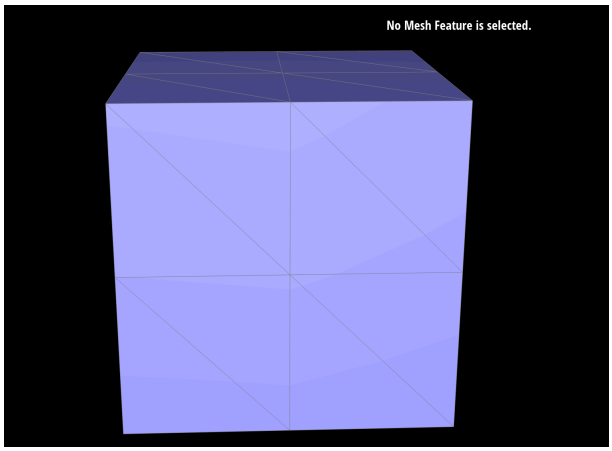
Implementing loop subdivision was not as painless of a process as implementing edge flipping or splitting.

One of the first issues we encountered was endless recursion during the edge splitting step. One thought was that saving the last edge iterator of the original mesh (e.g. `EdgeIter e_end = mesh.edgesEnd()`) would allow us to iterate through the original edges and stop before iterating through new edges. However, new edges are not appended in order, so this approach did not work. There was no obvious solution with the class variables provided, so we decided to include a new flag variable `isOrig` to flag edges that were split or produced by a split, which allowed us to divide the original edges without infinite recursion.

We then visually analyzed the mesh by running the code through steps 3 and 4:



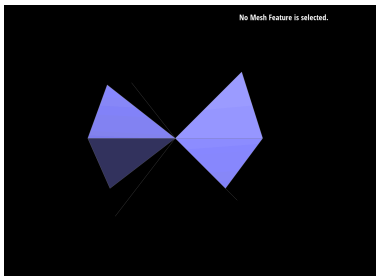
Split edges, before flipping



After flipping

Verifying edge splits and flips.

After visually verifying the mesh, we ran the full function including the vertex updates. The result was incorrect as it appeared that many vertices were erroneously moved to position $\{0, 0, 0\}$:

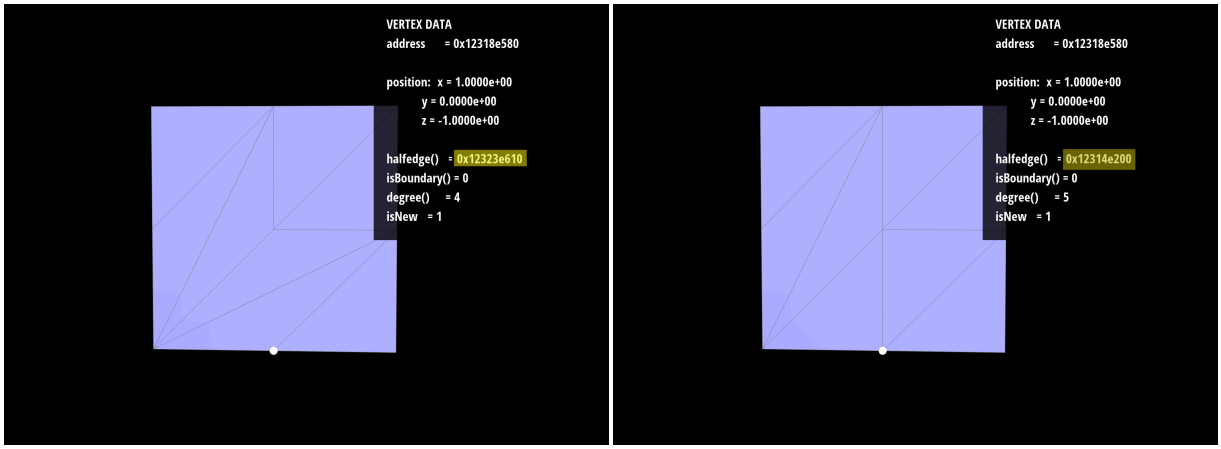


Faulty vertex position updates.

In step 5, we were initially trying to access the new positions of split vertices from the split edges (i.e. from `e\_split->newPosition`) but there was no reliable way to find the edge which stored `newPosition` since the split vertex's halfedge changes after the flip in step 4. So `v->halfedge()->edge()->newPosition` in most cases would return $\{0, 0, 0\}$ because the vertex's halfedge would often change to one associated with a new edge, which had no `newPosition` data.

We added the edges' `newPosition` to the debugging GUI to determine if there was a pattern to accessing the proper edge from the changed vertex halfedge. With no obvious solution to access the updated positions this way, we tried assigning both segments of the original split edge `newPosition` and then for each new vertex in step 5, visit all of its edges until you find one that is old (i.e. `e->isNew = false`) and then pulling the `newPosition` from here. This

worked but didn't feel right. This was less efficient than the (in hindsight) obvious solution to store the `newPosition` in the split vertex's `v->newPosition` immediately following the split operation. This both simplified and sped up our implementation of loop subdivision.



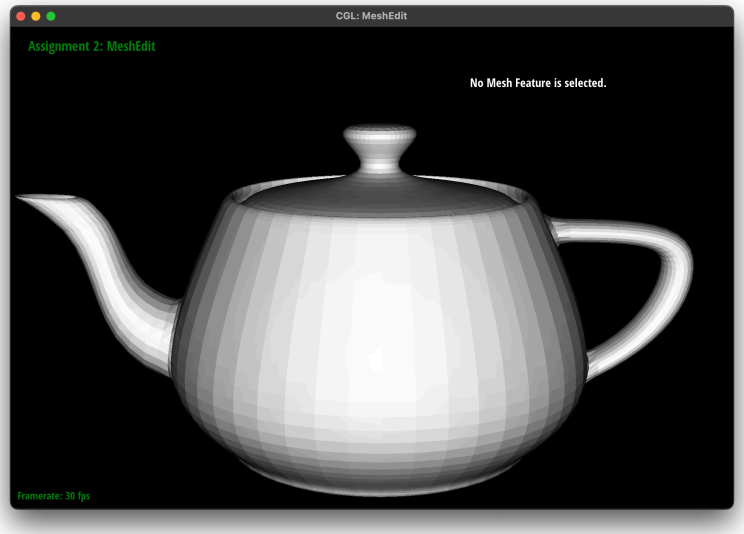
`v->halfedge() = 0x12323e610` `v->halfedge() = 0x12314e200`

The split vertex's halfedge changes after the edge flip operation.

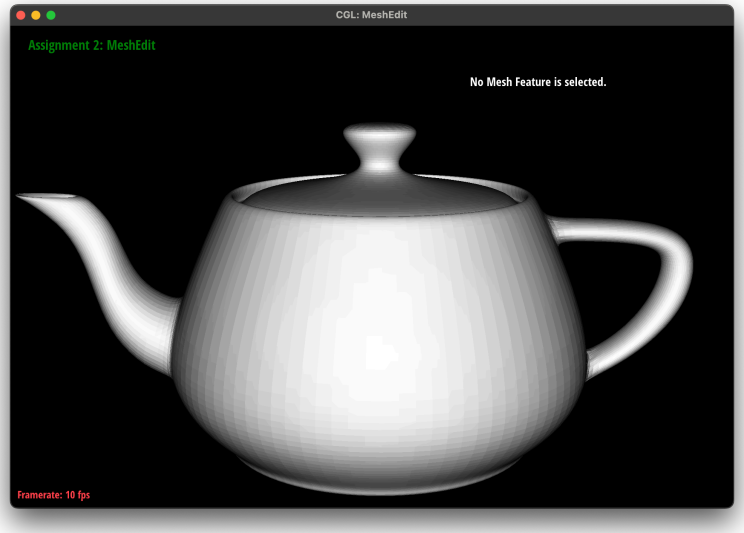
Task 6 Results

The predominant effect of loop subdivision is the smoothing of sharp edges, which are visible in coarsely tessellated geometry. Both intentional edges and corners (such as the corners of a cube) and unintentional edges (due to the coarse sampling of a smooth surface) are smoothed by loop subdivision. For geometry with primarily smooth features, like the teapot, loop subdivision creates a visually appealing and realistic geometry:

No subdivision



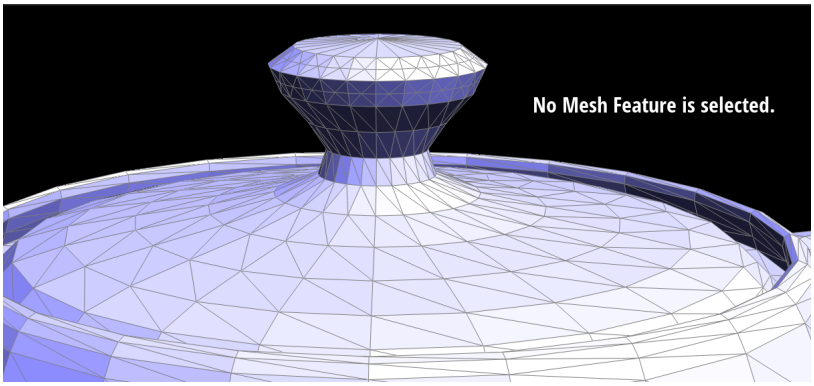
One level of subdivison



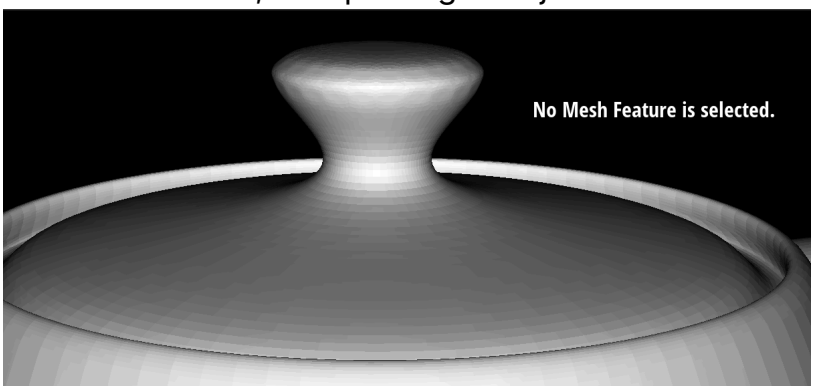
Two levels of subdivison

Loop subdivision smooths the sharp edges of the teapot.

Suppose that we intend to model the sharp crease in the knob on the teapot lid and it should be visible in our loop subdivided geometry. Without modification to the input mesh, this crease will disappear after two levels of subdivision. If we split the edges near this crease before subdividing, we can mitigate the effects of excessive smoothing:



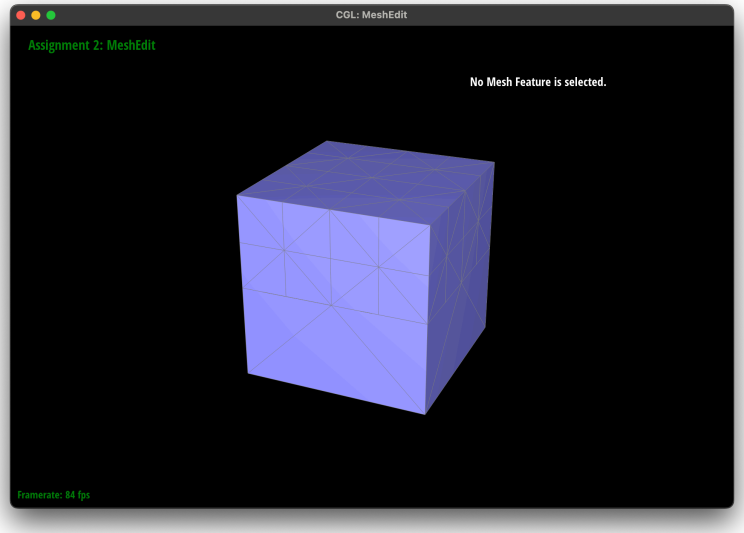
Before subdivision, we split edges adjacent to the crease



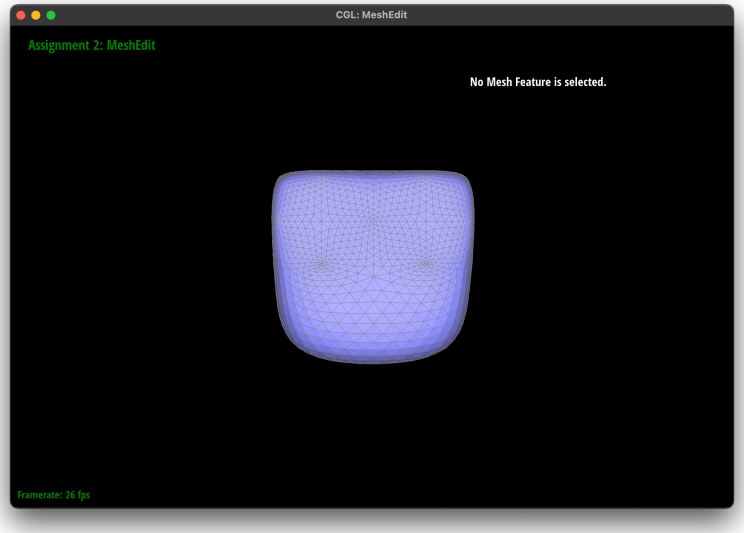
The crease is still visible even after multiple subdivisions

Mitigating excessive smoothing on the teapot lid.

We can visualize this by splitting the edges on the top half of the cube while not modifying the edges on the bottom half. Pre-splitting near the edges results in smaller radii. Our intuition is that splitting near these edges creates more extraordinary points, resulting in less smooth geometry in these regions of the mesh.

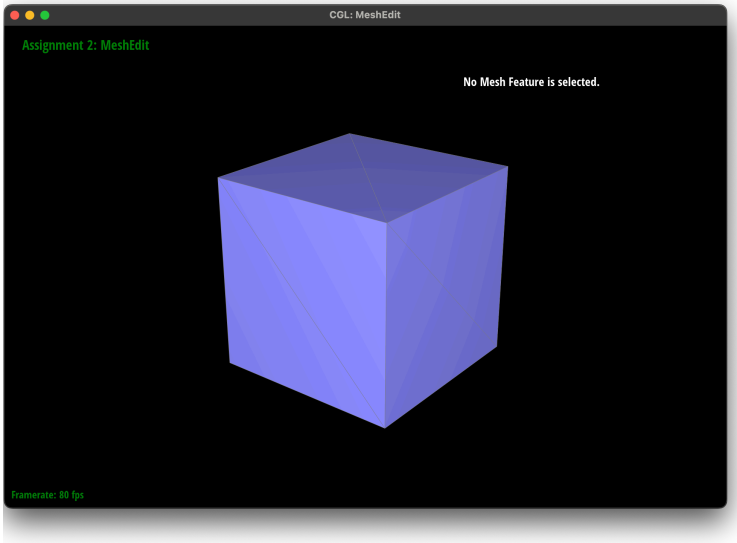


We perform multiple splits on the top half of the cube before refinement.

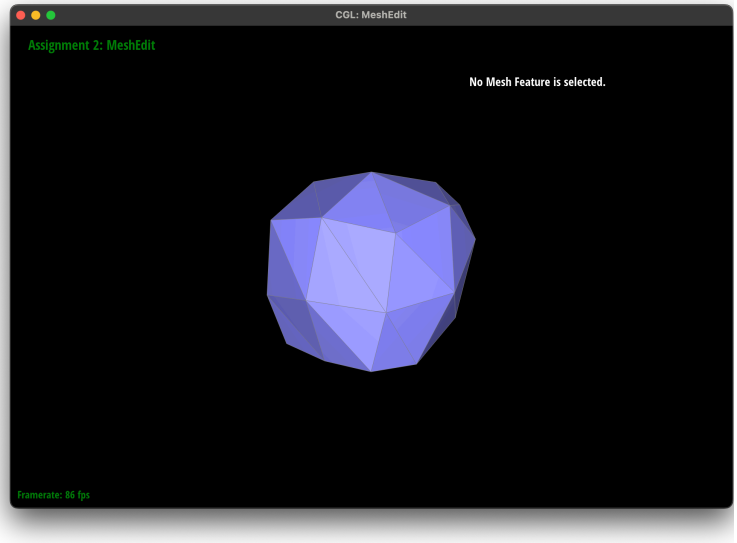


The edge radii are sharper on the top edges.
Comparing edge radii with and without edge splits.

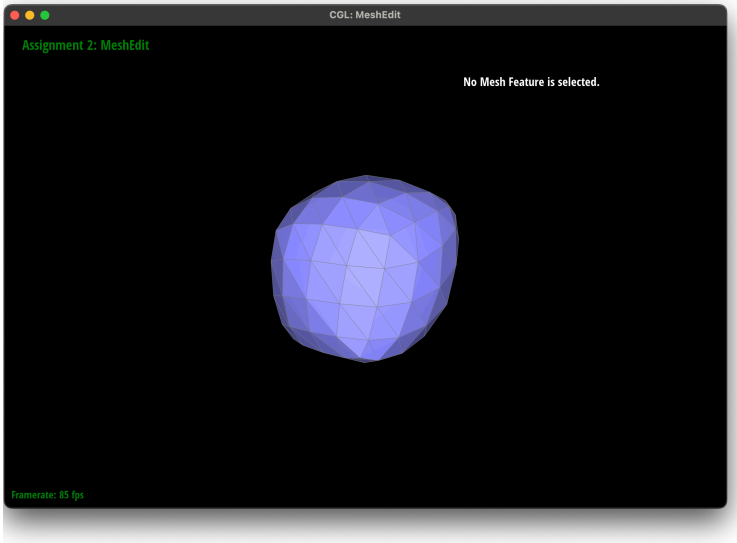
When performing loop subdivision on a cube, it tends to round sharp edges and shears the geometry into an asymmetrical shape. This is because the input mesh is not symmetrical, it is almost possible to see the diagonal edges on the faces even after three levels of subdivision. The limit surface will not be symmetrical if the input mesh is not symmetrical.



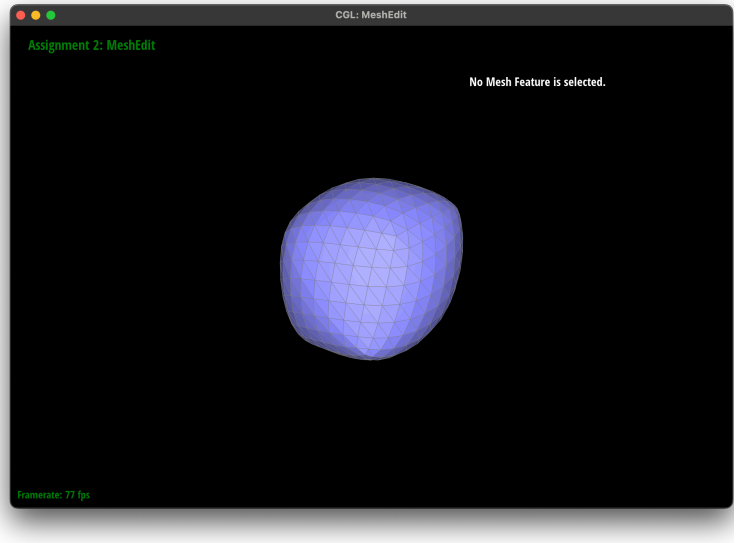
No subdivision



One level of subdivision



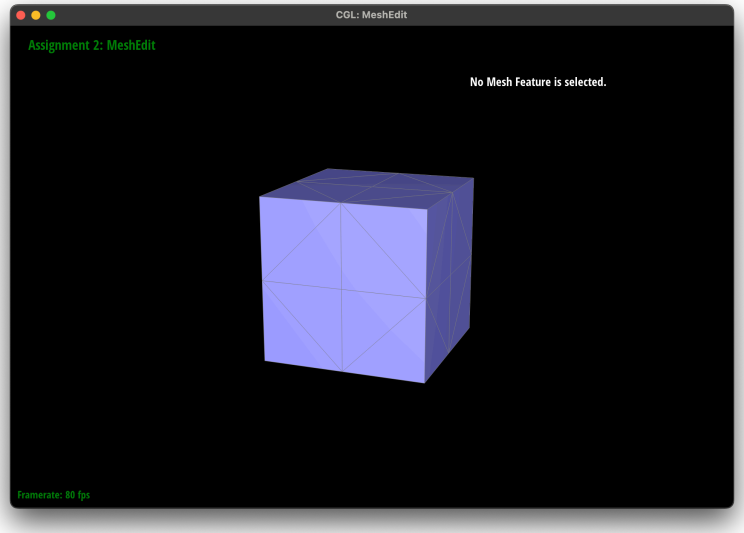
Two levels of subdivision

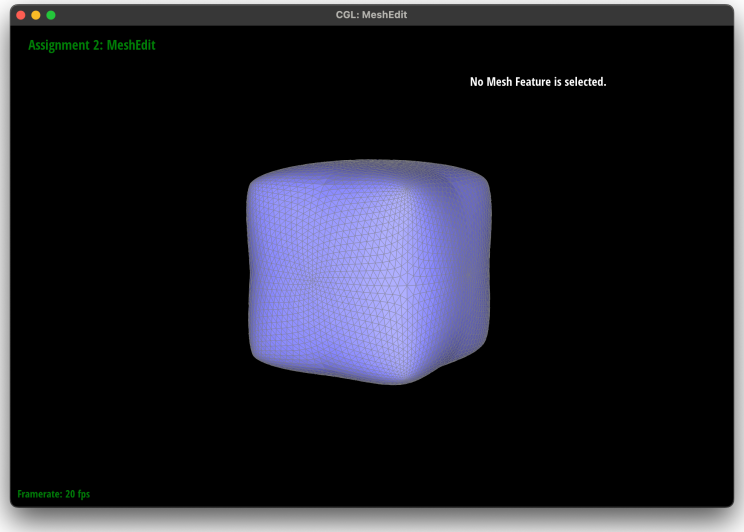


Three levels of subdivision

Loop subdividing the cube results in asymmetrical geometry.

If we pre-process the cube mesh by splitting the diagonals across each face and then flipping to get a symmetrical pattern on each face, the resulting geometry from loop subdivision is symmetrical. This is because a symmetric input mesh will result in a symmetric limit surface.





Effect of pre-processing on symmetry of subdivided geometry