CS 184 Assignment 2

Hector Li

July 2025

0 Overview

Special note: The PDF of this write-up and the original png files used in this document are all available on the website https://sigmundr6.github.io.

In this assignment, we investigate different geometric algorithms for modeling and rendering manifolds in computer graphics. The first section of this assignment (Part 1 and Part 2) explores Bézier patches, including mathematical motivation, implementation details, and picture examples. Part 1 introduces two dimensional Bézier curves and de Casteljau's algorithm. Part 2 applies de Casteljau's algorithm to three dimensional control points to get a smooth surface. We are using the Utah teapot extensively as an example to show the outcome of our functions.

However, even though Bézier curves and patches are C^∞ on their own, the manifold becomes hard to manage when the number of control points increases. If we group control points into different sets and compute piecewise Bézier curves or patches, the smoothness will depend on the arrangement of the control points. In addition, the recursive stacks also grow fast as the number of control point grows, preventing us from modeling complex surfaces before stack overflow. Moreover, to render a Bézier patch, we still need to sample the two coefficients used to generate the patch, defeating one major advantage of the continuous polynomial functions of Bézier patches.

Therefore, in the second section (Part 3, 4, 5 and 6), we discuss the more widely adopted mesh structure for modeling manifolds today. Part 3 is relatively self-contained. It introduces a way to compute normal vectors on a point weighted by the areas of the faces incident on the vertex. The weighted normal vectors can be used to compute shading. Part 4 and 5 explore the implementation detail of edge flips and edge splits, two useful mesh operations that will become instrumental when we realize the Loop subdivision algorithm in Part 6.

One most unexpected discovery upon finishing this assignment is how computationally expensive modeling 3D manifolds, especially through mesh operations is, despite the efficiency of C++ handling pointers. Since the number of objects grow exponentially in the Loop subdivision, one can expect doing multiple rounds of upsampling is unsustainable, but the responsiveness of the GUI slows drastically even after just three rounds on some simple mesh structures, when modeling real objects requires a much higher mesh count.

It is entirely possible that with better planning, we can reduce the number of iterations in our Loop subdivision implementation or make the algorithm more optimized for caches. However, since the computational complexity grows exponentially, upsampling still remains unsustainable.

1 Part 1

Given a set of control points

$$S = \{P_1, \cdots, P_n \in \mathbb{R}^m\}$$

for some $m, n \in \mathbb{Z}_+$ and $t \in (0, 1)$, we can connect the points to get $(n-1)$ line segments. On each of these line segments, we perform linear interpolation with the coefficient t, and we have a new set S' consisting of points

$$\begin{aligned} P_1' &= (1-t)P_1 + tP_2, \\ P_2' &= (1-t)P_2 + tP_3, \\ &\cdots\cdots, \\ P_{n-1} &= (1-t)P_{n-1} + tP_n. \end{aligned}$$

For each step, we reduce the number of control points by 1, and when the set consists of only one point P_t°, we call P_t° a point of the Bézier curve with the set of control points S and the coefficient t. If we vary t, these points will form a curve. It is not obvious, but mathematicians have proven that the curve generated in this way is equivalent to the sum of Bernstein polynomials, so it is C^∞. Note that the curve is also well defined when $t = 0$ (the points collapse to P_1 and $t = 1$ (the points collapse to P_n). However, if we divide control points into multiple subsets and use piecewise Bézier curves, the smoothness of the curve resulting from de Casteljau's algorithm will depend on how control points are laid out. Another thing to note is that even though we use linear interpolations to find the points, the eventual curve is only guaranteed to pass through the first and the last control point in the set.

In our C++ program, we implement de Casteljau's algorithm by defining a recursive function `BezierCurve::evaluateStep` that takes in a vector of control points `std::vector<Vector2D>&`. With `t` given as a class member, we can return the vector if the vector contains only one element as the base case and construct a vector of new control points using the algorithm described above for the non-base cases. Note this function only carries out one step of the algorithm and reduces the number of control points by 1 when there are more than 1 control points. To return a point of the Bézier curve with the given control points and the coefficient `t`, we need to call this function repeatedly until there is only one control point left, which is the point on the curve.

Below, we have included some graphs of producing a Bézier curve on six control points and how the curve varies when we move the control points change the interpolation coefficient t.

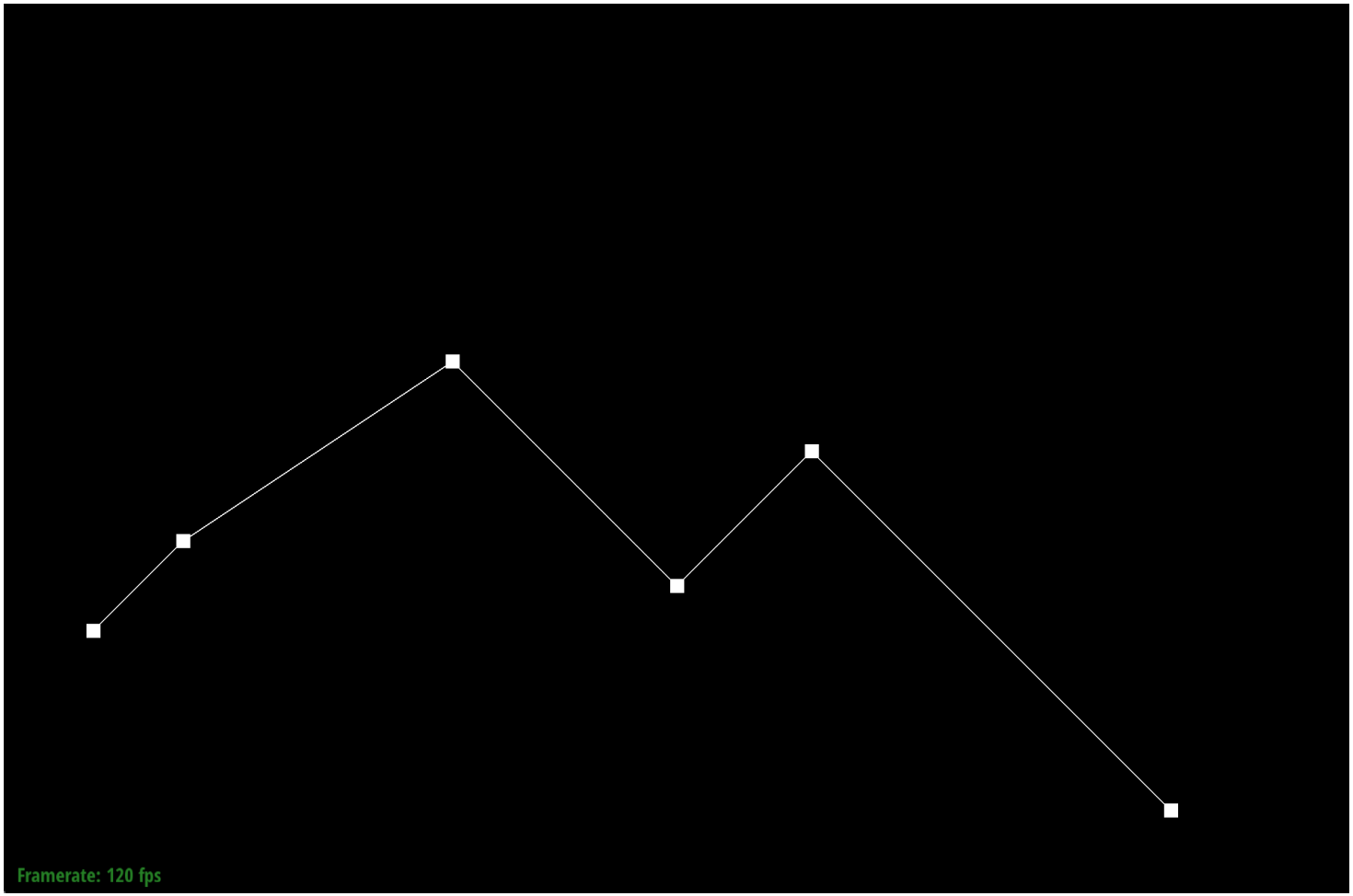


Figure 1: Step 0: Six control points.

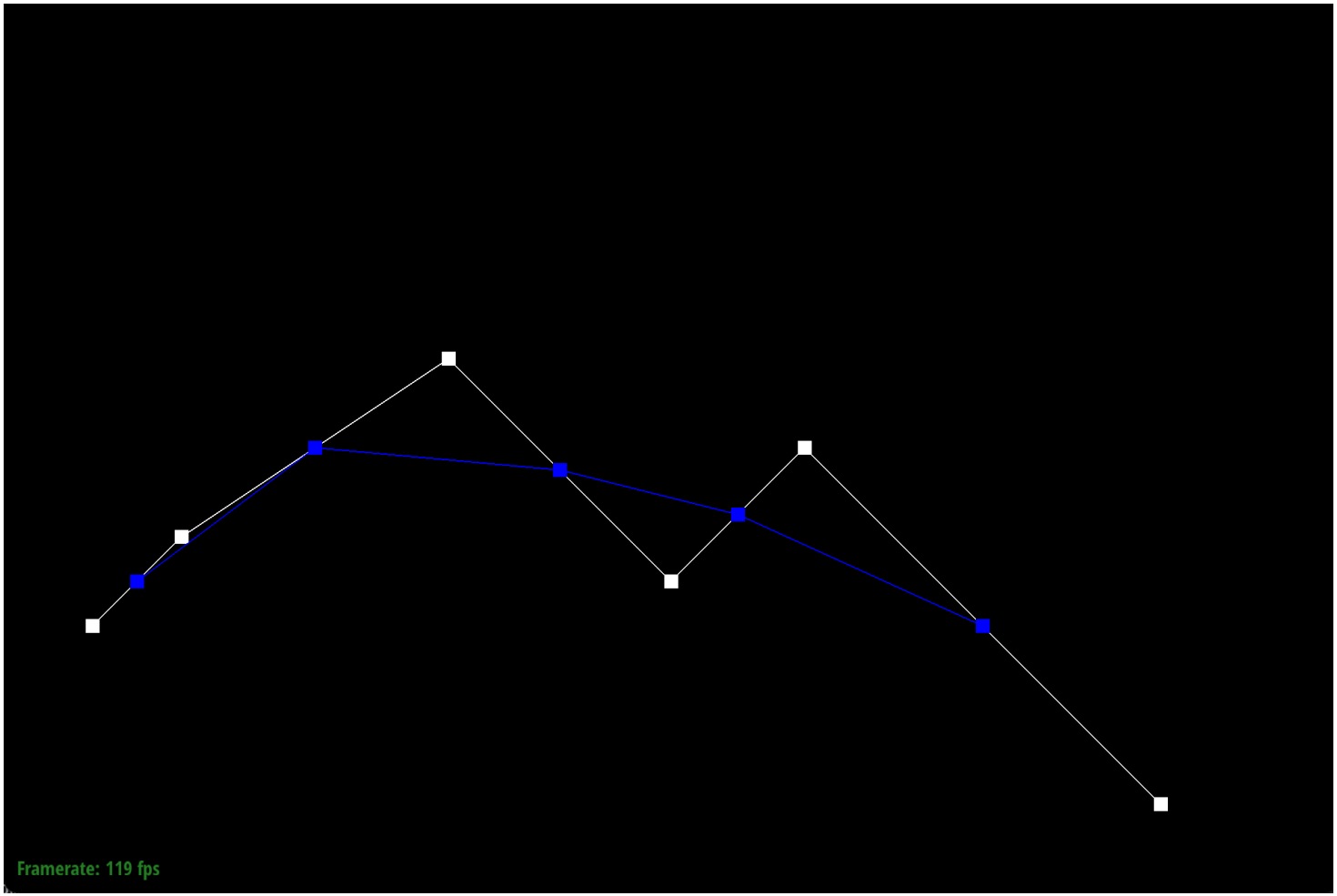


Figure 2: Step 1: Five temporary control points.

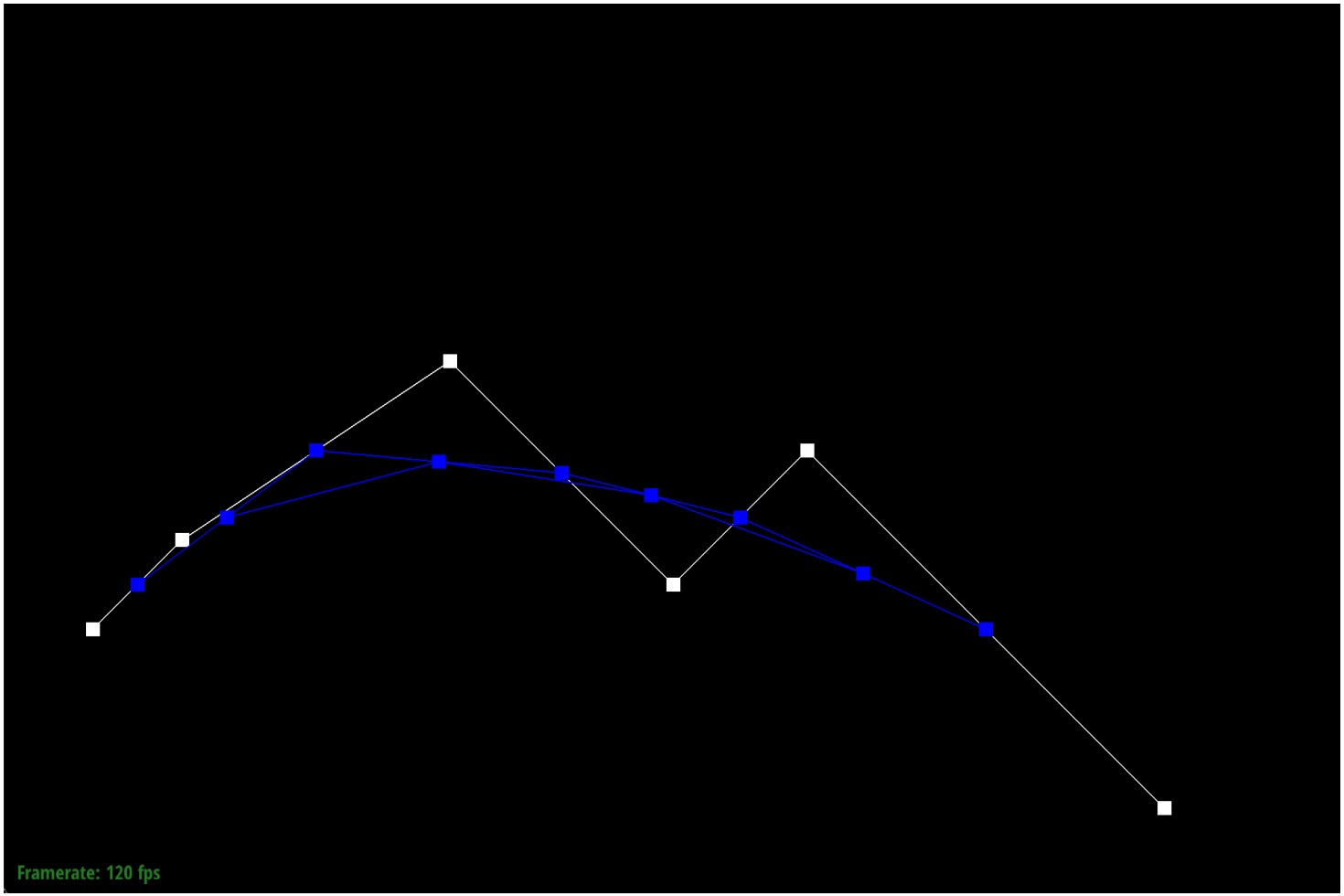


Figure 3: Step 2: Four temporary control points.

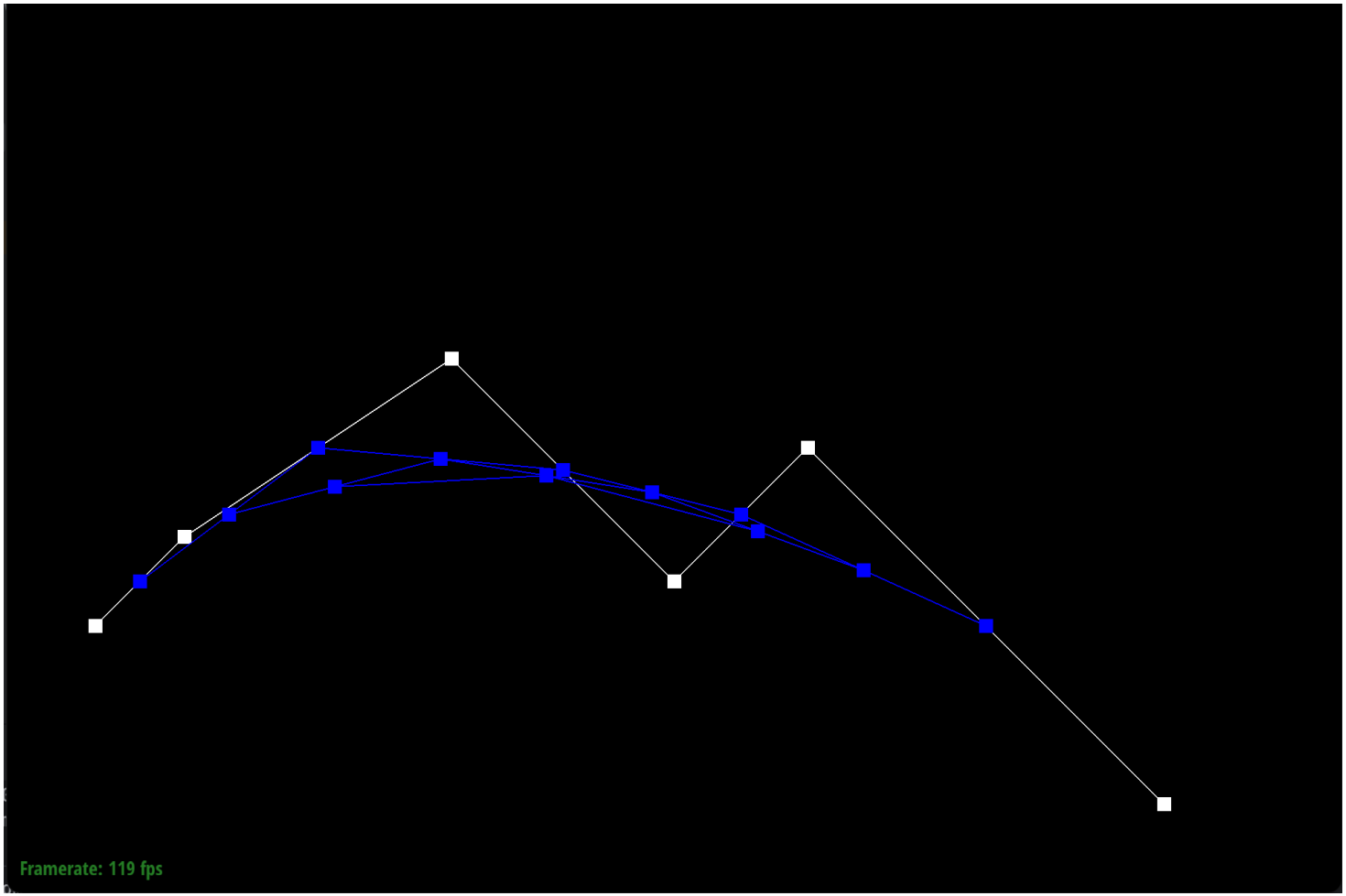


Figure 4: Step 3: Three temporary control points.

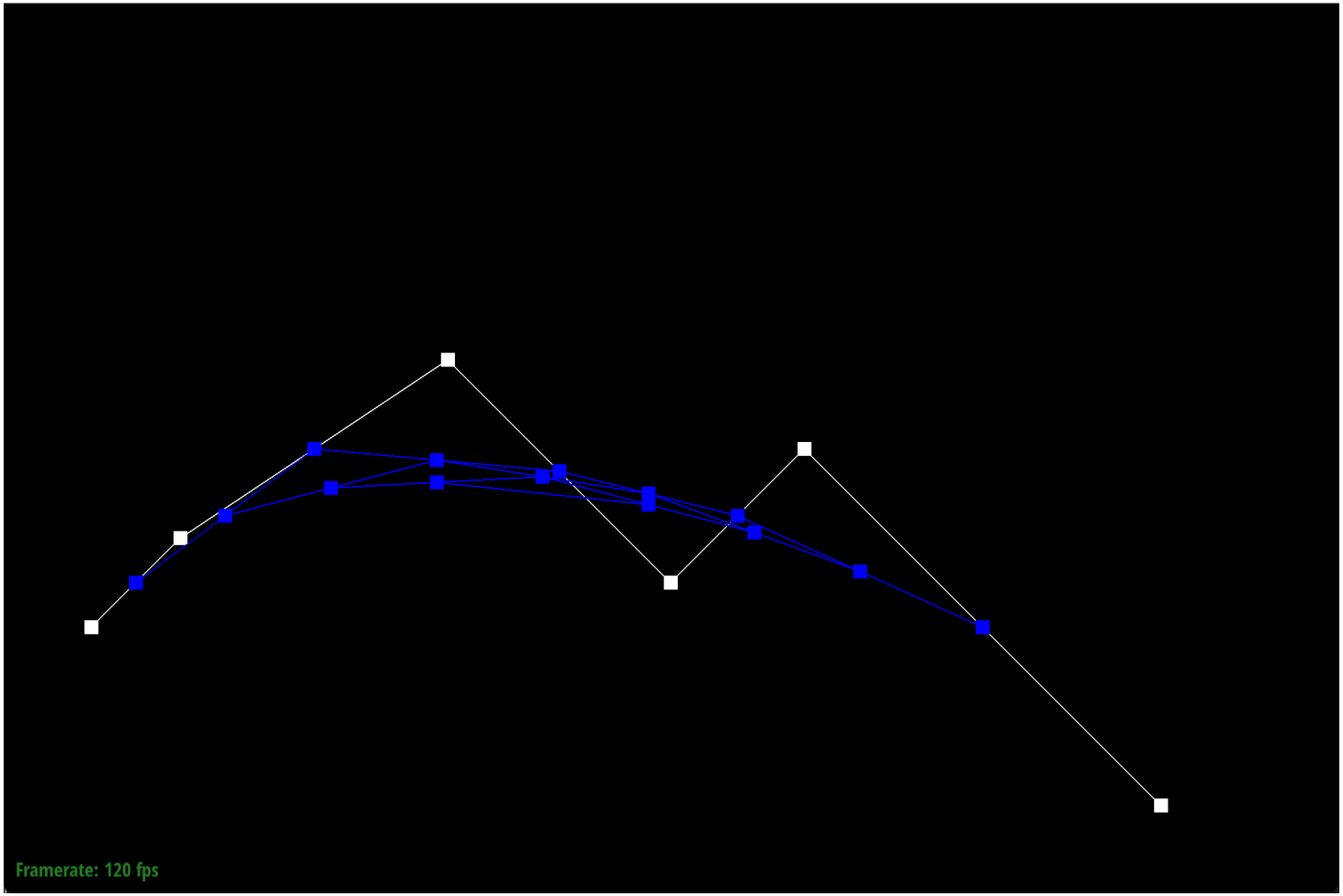


Figure 5: Step 4: Two temporary control points.

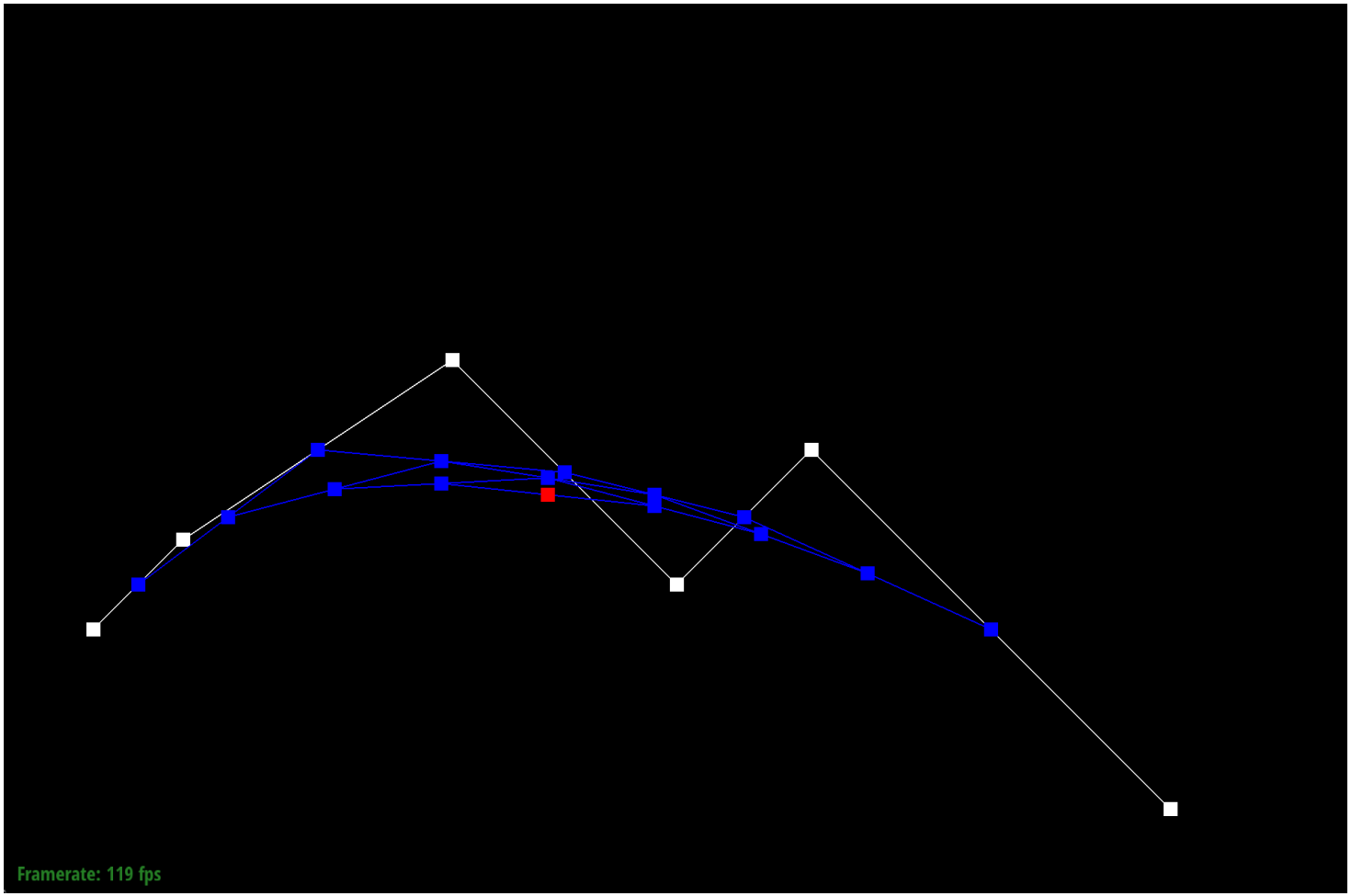


Figure 6: Step 5: Point on the Bézier curve.

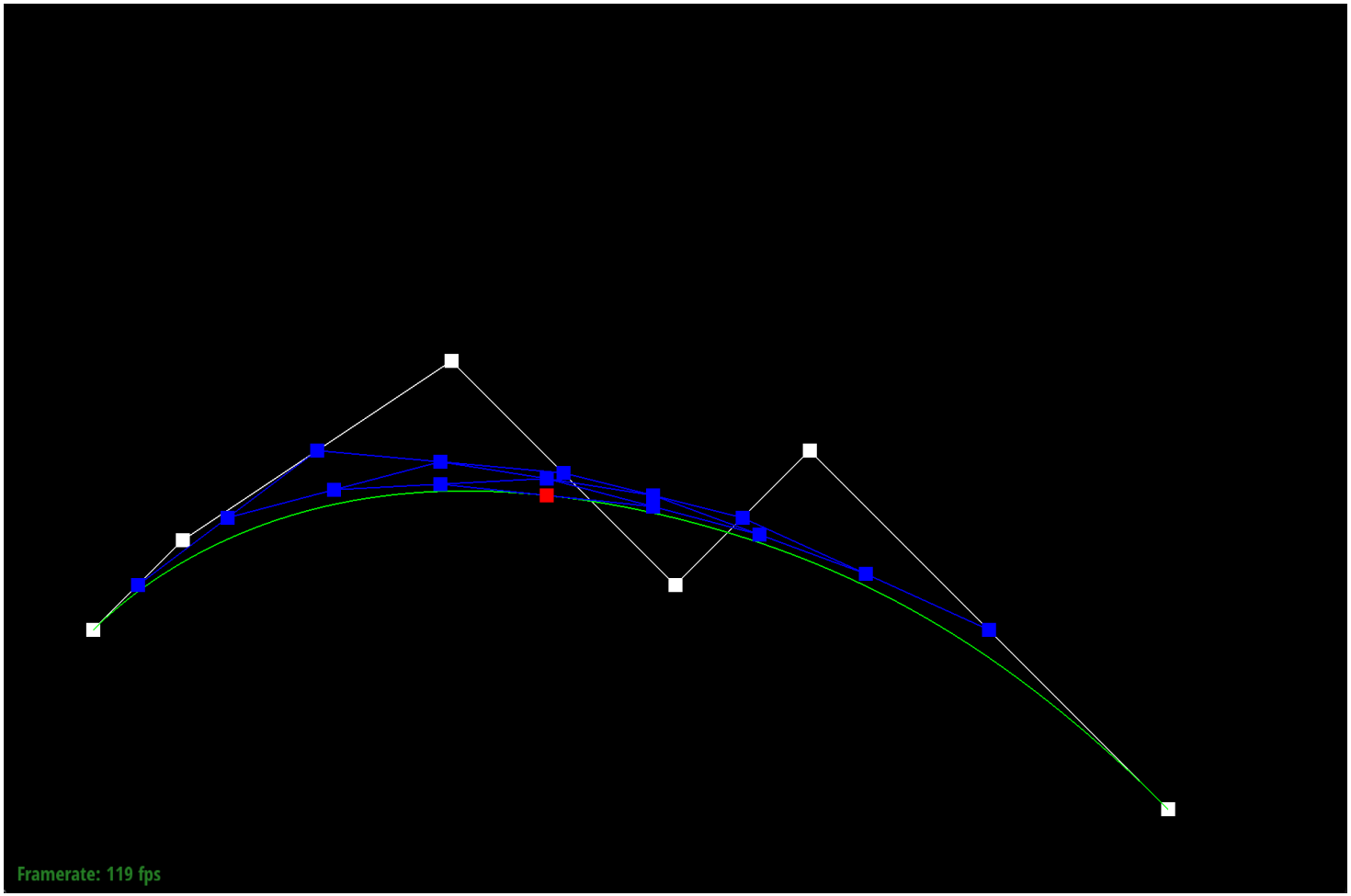


Figure 7: Full Bézier curve.

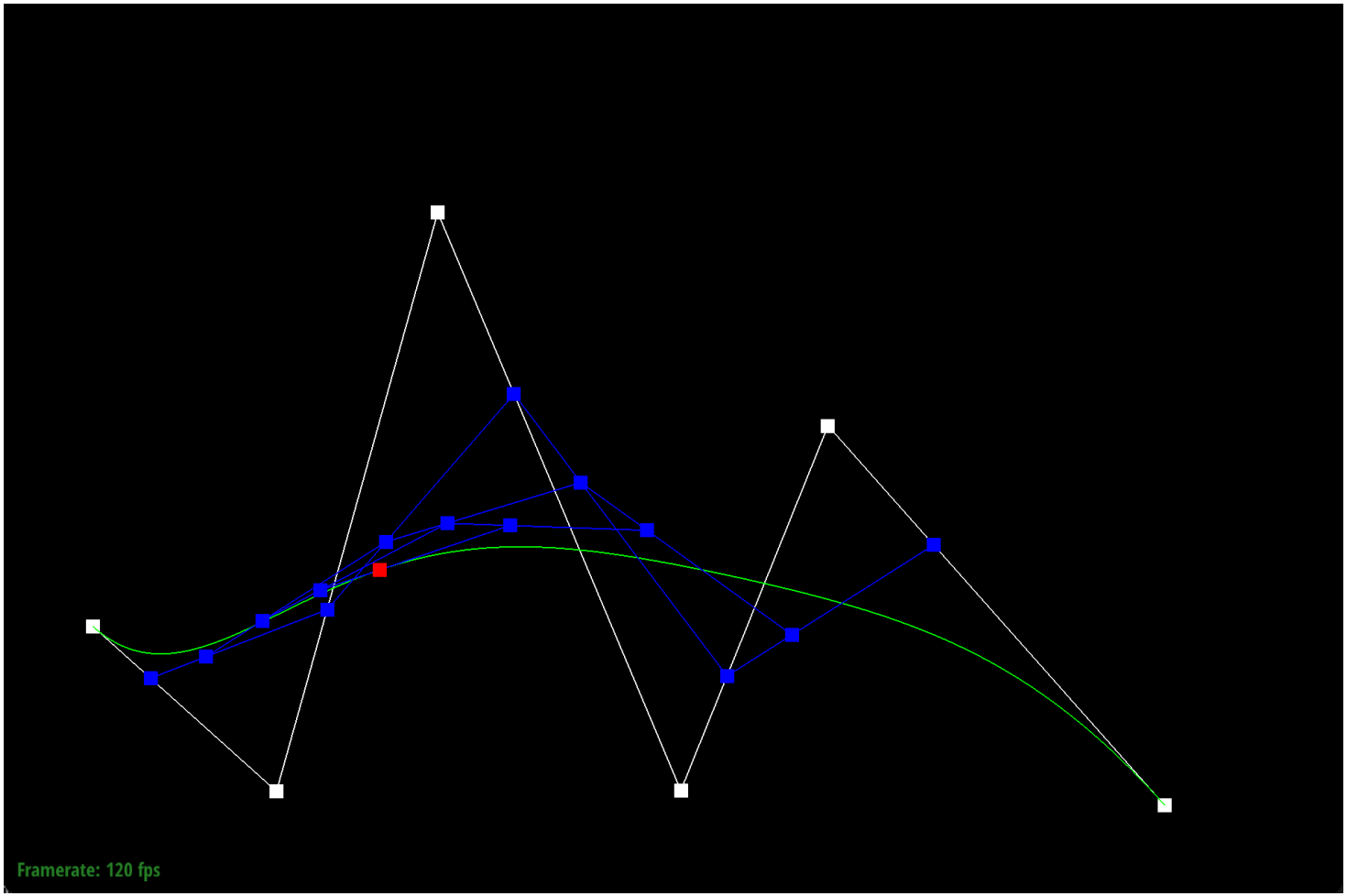


Figure 8: Bézier curve with six different control points and a different t.

2 Part 2

Given a set of sets of control points

$$S = \{T_1, T_2, \cdots, T_n\}$$

where $T_1, \cdots, T_n$ each have m control points in $\mathbb{R}^s$ for some $m, n, s \in \mathbb{Z}_+$, and some fixed $u, v \in (0, 1)$, we can apply the recursive de Casteljau algorithm described in Section 1 to each T_i for $i = 1, \cdots, n$ with the interpolation coefficient u. After $(m-1)$ rounds of recursion, each set T_i of control points will produce one point $P^\circ_{iu} \in \mathbb{R}^s$ for $i = 1, \cdots, n$. Note that $Q^\circ_{iu} \in \mathbb{R}^s$ is a point on the Bézier curve generated by control point set T_i with interpolation coefficient u.

Now we want to apply the recursive de Casteljau algorithm on the set of these new control points with the interpolation coefficient v. After $n-1$ rounds of recursion, we will have a point P°_{uv} on the Bézier curve with the control points $P^\circ_{1u}, \cdots, P^\circ_{nu}$. If we vary u and v, these points will form a patch of surface. Mathematicians have also proven that the surface generated in this way is equvalent to the sum of the products of Bernstein polynomials in u and Bernstein polynomials in v, so the patch of surface is C^∞. However, the smoothness of piecewise Bézier patch depends upon how the control points are laid out. Note that the surface is well defined when $u = 0$ or $u = 1$ or $v = 0$ or $v = 1$ since we are using the same recursive method in Section 1.

In our C++ program, we implement de Casteljau's algorithm by defining the recursive function `BezierPatch::evaluateStep` that takes in control points in `std::vector<Vector3D>&` and the interpolation coefficient `t`. Since de Casteljau's algorithm works in $\mathbb{R}^m$ for any $m \in \mathbb{Z}_+$, the implementation detail is almost exactly the same as the function we described in Section 1. Namely, we can return the vector if the vector contains only one element as the base case and construct a vector of new control points by taking linear interpolations with coefficient `t` between the consecutive control points. This function also carries out only one step in the algorithm and reduces the number of control points by 1 when there are more than 1 control points. We then define a function `BezierPatch::evaluate1D` that takes in same arguments as `evaluateStep`, but calls `evaluateStep` repeatedly until there is only one control point left and return it, which is the desired point on the Bézier curve. With `evaluate1D`, we can define `BezierPatch::evaluate` that takes in two linear interpolation coefficients `u` and `v`. The function performs `evaluate1D` with the coefficient `u` on each element of the two dimensional vector class member `controlPoints` and then performs `evaluate1D` with the coefficient `v` on the points we get. Therefore, with fixed $(u, v) \in [0, 1]^2$, we have a point on the Bézier patch defined by the control points. We can then sample `u` and `v` to get the whole patch of surface.

Below we include a picture of the Utah teapot made with our Bézier patch method.

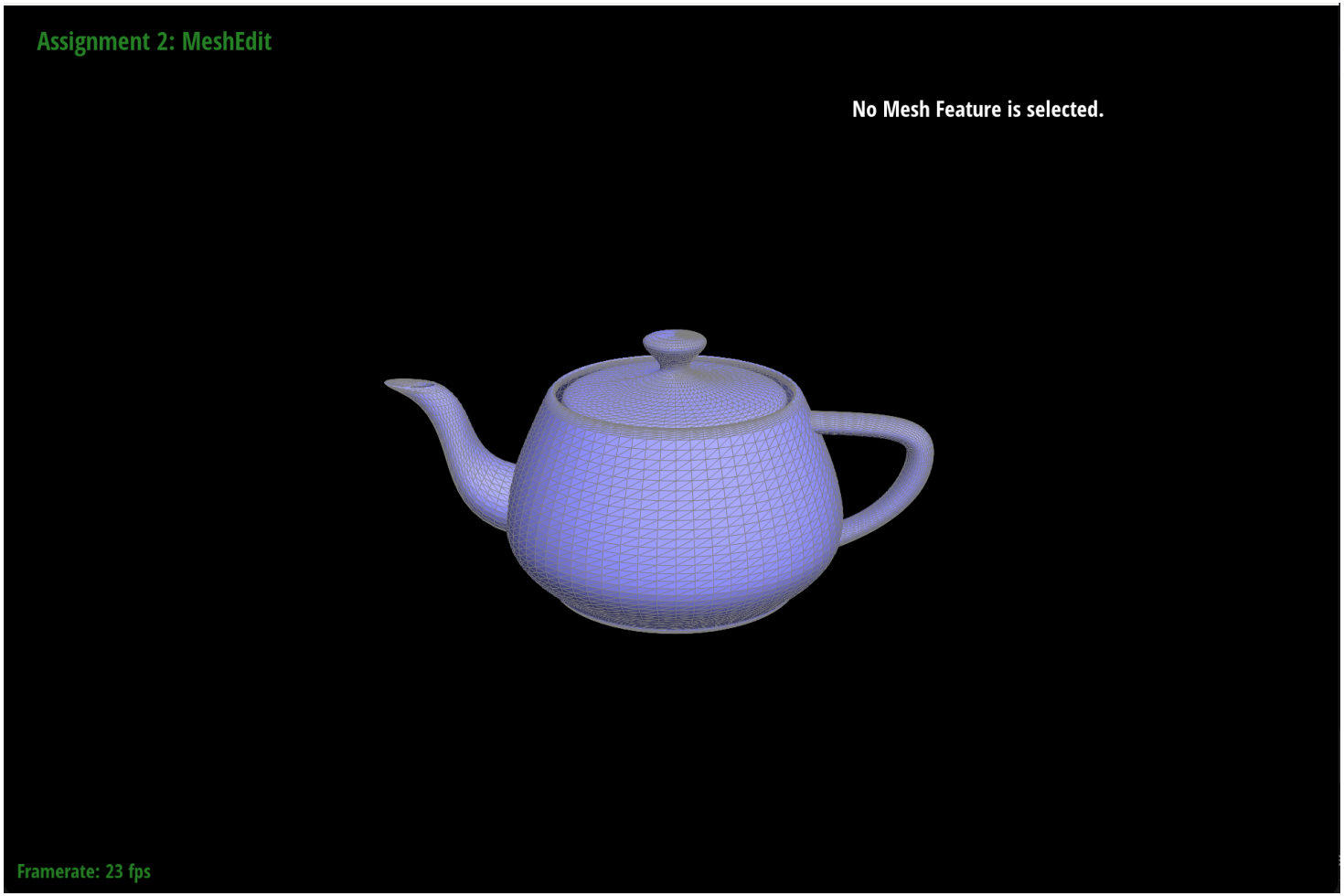


Figure 9: The Utah teapot made with the Bézier patch.

3 Part 3

In our C++ program, for any given vertex, we can use the `halfedge` method to get a pointer `h` (wrapped as an iterator in the implementation) to the halfedge originating from this vertex. Since all of our implementations are done on pointers wrapped as iterators rather than on the objects themselves for memory efficiency, we will just use the terminology of objects to mean the pointers to these objects to simplify the discussion unless otherwise noted.

To store the normal vector and area of each triangle on the vertex, we can make `std::vector<Vector3D> normals` and `std::vector<float> area`. With a halfedge originating from the vertex, we can get the three vertices `p0, p1, p2` in the counterclockwise orientation with `h->vertex()->position`, `h->next()->vertex()->position`, and `h->next()->next()->vertex()->position`. We can then compute the cross product `Vector3D N = cross(p1 - p0, p2 - p0)`. We can add `N.norm() * 0.5f` to the `area` vector and `N.unit()` to the `normals` vector.

Then we can access another halfedge originating from the vertex clockwise to our last halfedge by setting `h = h->twin()->next()`. From each outgoing halfedge, we repeat the process described above to add elements to the `area` vector and `N.unit()` to the `normals` vector. Then we go to the next outgoing halfedge by taking the twin and the next, add elements to the vectors, and repeat the process until we are back at `h`. We can use the `std::accumulate` method to compute the sum of the areas and compute the weighted normal with `normal += normals[i] * area[i] / area_sum` with `i` being the indices of the vector `normals`. We return the `Vector3D` object `normal` as the desired weighted normal vector.

Below we have included the pictures of the Utah teapot shading with and without vertex normals.

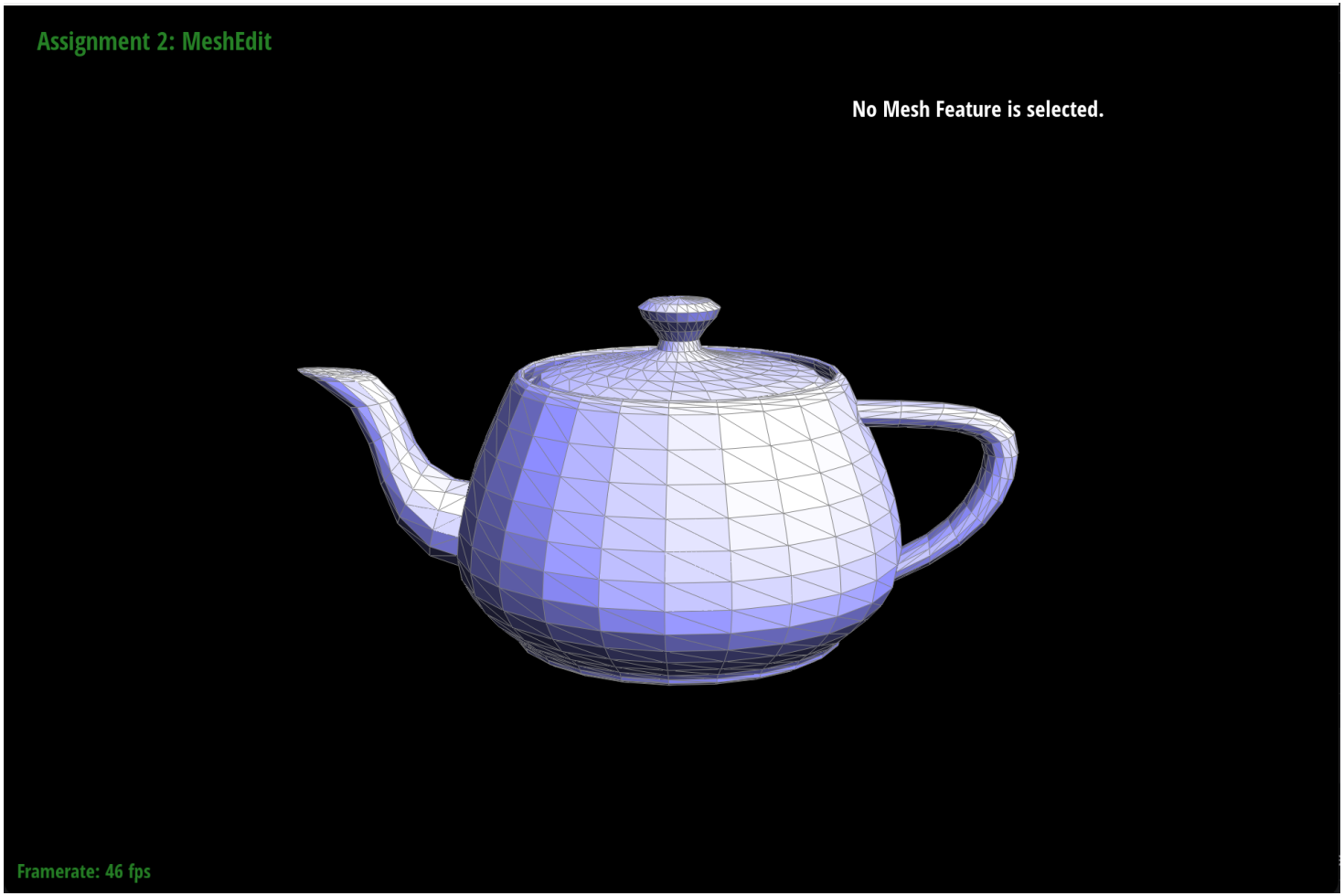


Figure 10: The Utah teapot shading without vertex normals.

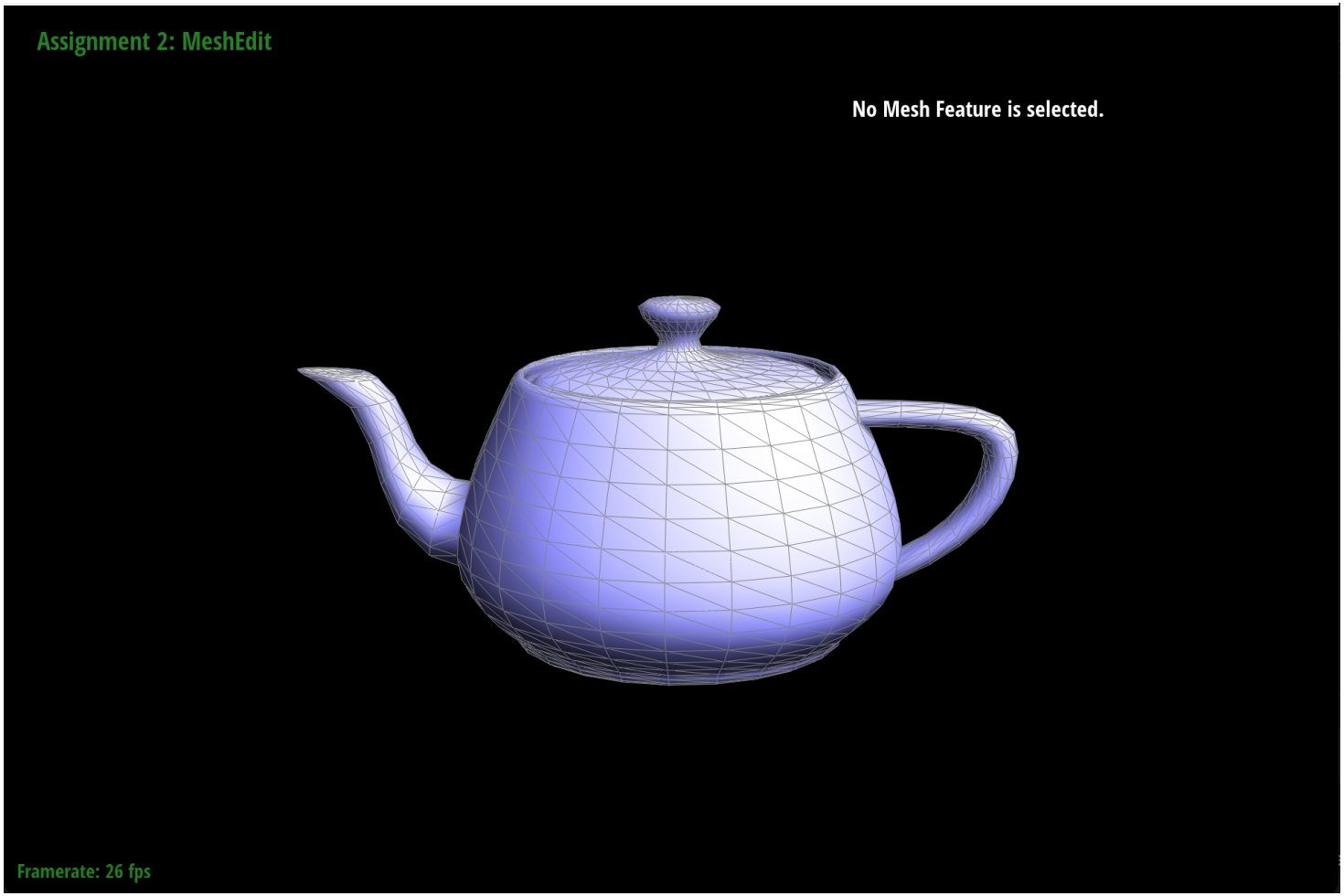
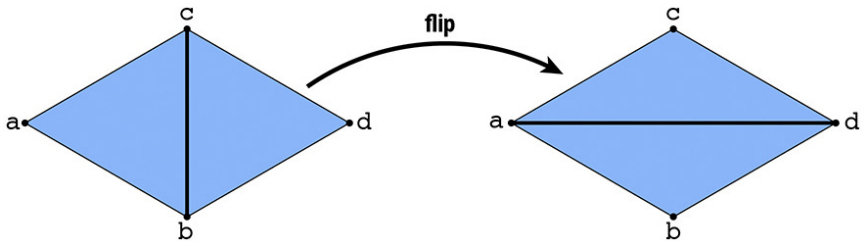


Figure 11: The Utah teapot shading with vertex normals.

4 Part 4

For the edge flip function in our C++ program, we are given an edge as the argument. We want to check whether the edge is on the boundary. If it is on the boundary, we return the original edge without flipping anything since we do not want to destroy the mesh structure by flipping a boundary edge.



When the edge given is not on the boundary, we use the diagram above to help us understand the operation. For the edge flip, it is sufficient to consider only the objects inside these two triangles since the objects outside have their pointers unaffected. We define the halfedges `ab, bd, dc, ca`, `e` for *bc* and `re` (reverse e) for *cb* by traversing the halfedge of the edge `e0` given with twin and next methods. Then we define the edges `abu, bdu, dcu, cau` (undirected), vertices `a, b, c, d`, and faces `f1` (the upper face) and `f2` (the lower face). The edge flip is achieved by setting the vertex of `e` to `a` and the vertex of `re` to `d`. We then carefully update `next, twin, vertex, edge, face` pointers of each halfedge, and `halfedge` pointers of each vertex, edge, and face. Then we return the edge `e0` with its pointers reassigned. There is not any special debugging trick except being extra careful about the pointer assignment and not relying on the auto-filling feature of the IDE. We did not use the provided `Halfedge::setNeighbors` method, but groups the pointer assignment clauses of each object into blocks for easy debugging.

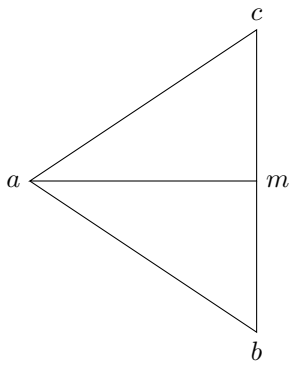
Below we have included pictures of the Utah teapot before and after the edge flip.

Figure 12: The Utah teapot before edge flips.

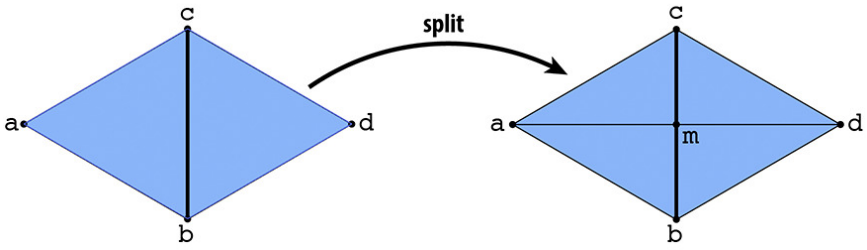
Figure 13: The Utah teapot after the edge flips. Note the diamond shape.

5 Part 5

For the edge split method in our C++ program, we are given an edge `e0` as the argument. We need to consider two different situations depending on whether `e0` is on the boundary.



When the edge is on the boundary, we use the diagram above to help us understand the operation. For the edge split, we need to consider all objects inside the triangle and all objects related to the edge we are splitting. We take the halfedge originating from `b` inside the triangle and call it `bm` (even though it is still *bc* at this point). We take its twin and call the halfedge `cm` (the choice is not arbitrary, because choosing `mb` would make the original `next` pointer pointing at *cb* hard to maintain). We define the existing halfedges `ab, bm, ca, cm`, the existing edges `abu, bcu, cau`, the existing vertices `a,b,c` and the existing faces `f1` and `bf` (boundary face). Then we define the new vertex `m` whose position is set at `(b->position + c->position) * 0.5f`, new halfedges `am, ma, mc, mb`, new edge `amu`, and new face `f2`. We want to make `f1` the lower face and `f2` the upper face. Then we carefully update `next, twin, vertex, edge, face` pointers of each halfedge, and `halfedge` pointers of each vertex, edge, and face. One thing to note is that we have to update `mb->next()` to `cm->next()` before changing the pointers of `cm`, since the halfedge `cm->next()` is outside this diagram. After correctly reassigning the pointers, we return the vertex `m`.



When the edge is not on the boundary, we use the diagram above to help us understand the operation. Unlike the boundary case, we just need to consider all objects inside the two triangles. We take the halfedge originating from `b` inside the left triangle and call it `bm` (it is still *bc* at this point). This time

we call its twin `mb`. We define the existing halfedges `ab, bd, dc, ca, bm, mb`, the existing edges `abu, bdu, dcu, cau, bmu`, the existing vertices `a, b, c, d`, and the existing faces `f1, f2`. Then we define the new vertex `m` whose position is set at `(b->position + c->position) * 0.5f`, new halfedges `cm, mc, am, ma, md, dm`, new edges `amu, mdu, mcu`, and new faces `f3, f4`. We want to make `f1, f2, f3, f4` to be the lower-left, lower-right, upper-left, and upper-right faces respectively. Then we carefully update `next, twin, vertex, edge, face` pointers of each halfedge, and `halfedge` pointers of each vertex, edge, and face. After correctly reassigning the pointers, we return the vertex `m`.

Again, there is not any special debugging trick except being extra careful about the pointer assignment and not relying on the auto-filling feature of the IDE. We did not use the provided `Halfedge::setNeighbors` method, but groups the pointer assignment clauses of each object into blocks for easy debugging. Since edge split function requires adding new objects and rearranging more pointers, arranging code into blocks makes it much easier to read.

Below we have included pictures of the Utah teapot before any operation, after some edge splits, and after a combination of both edge splits and edge flips. We have also included pictures of the Beetle car frame to show how the boundary case is handled properly by our program.

Figure 14: The Utah teapot before any operation.

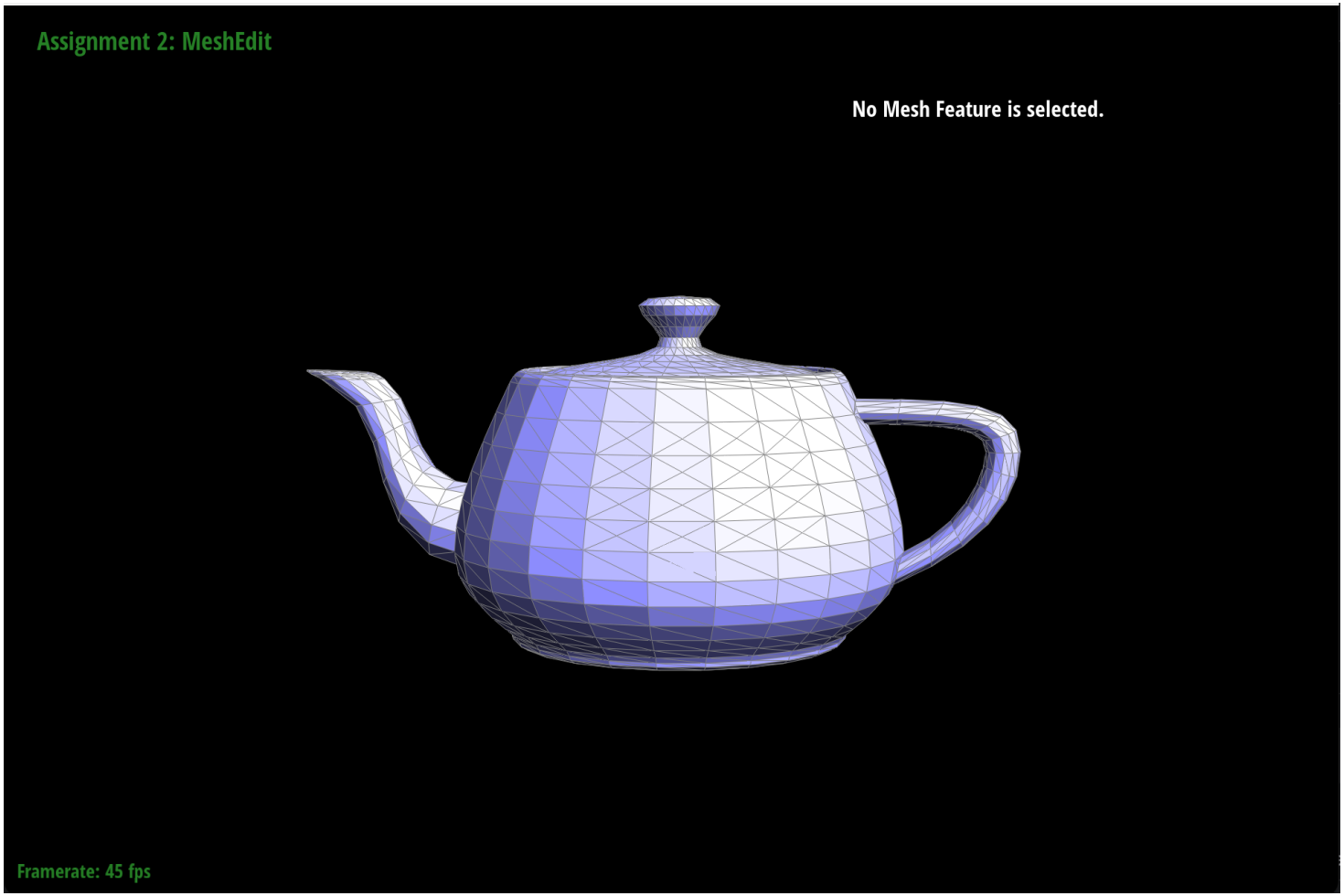


Figure 15: The Utah teapot after some edge splits.

Figure 16: The Utah teapot after a combination of both edge splits and edge flips.

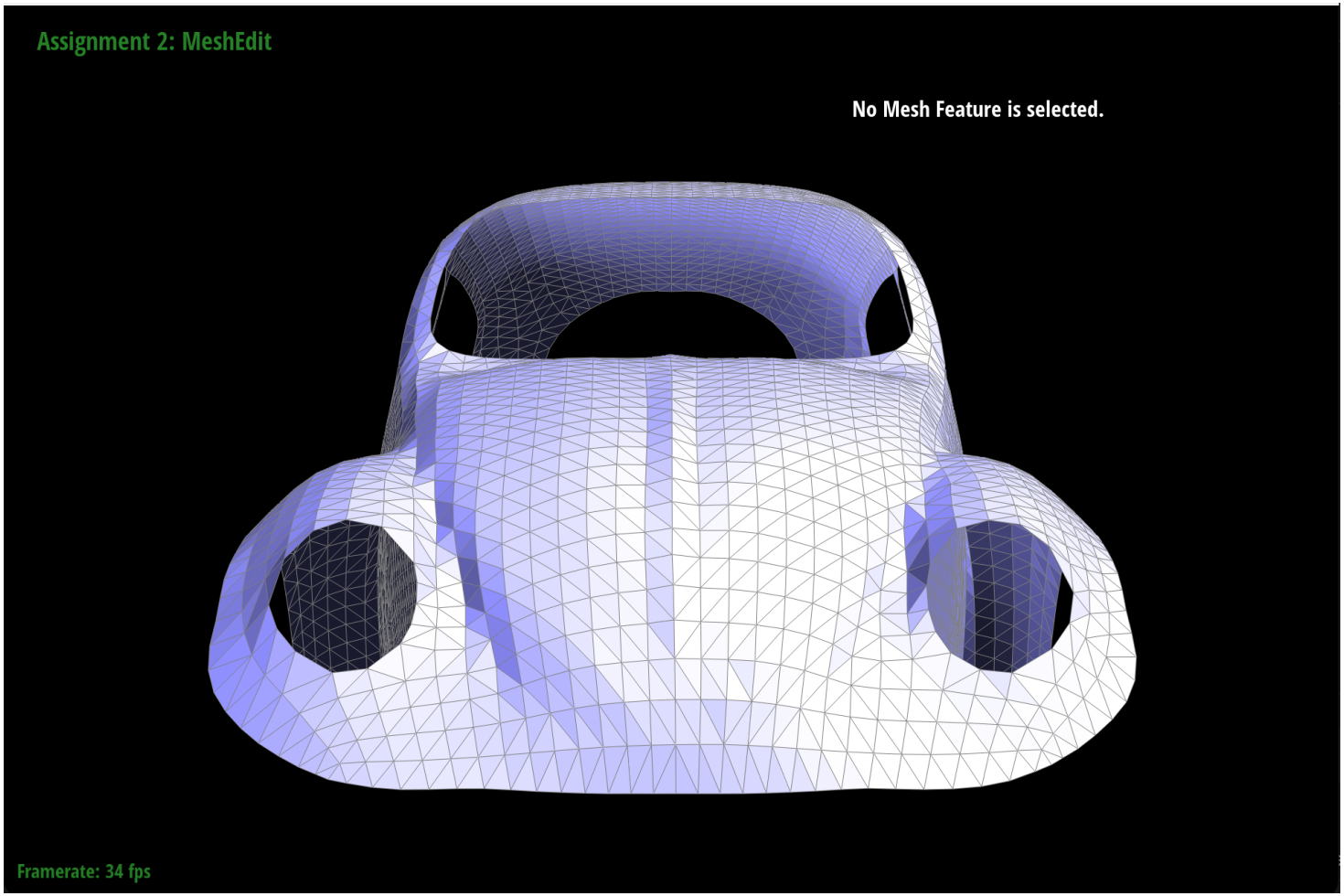


Figure 17: The Beetle car frame before any operation.

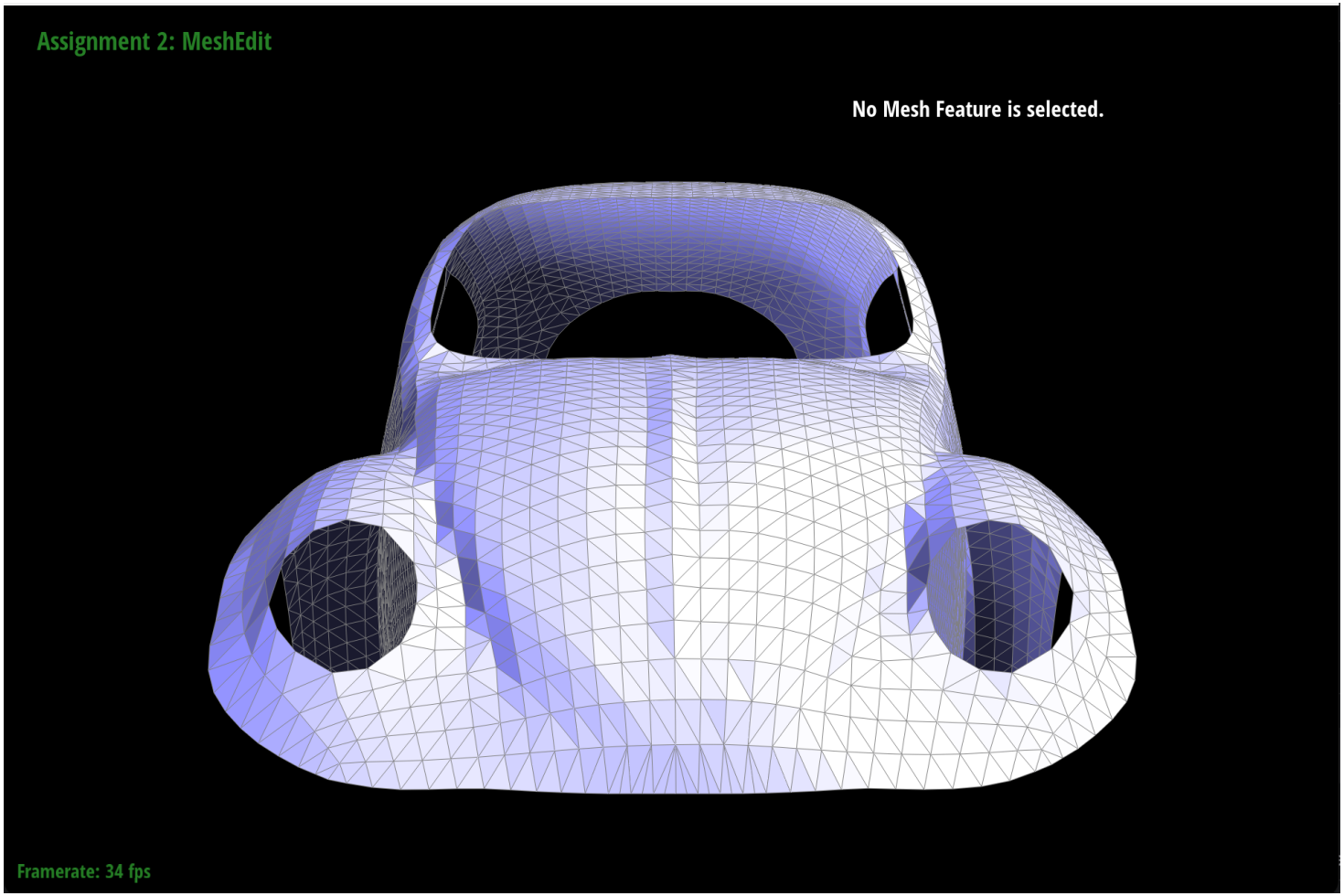


Figure 18: The Beetle car frame after some edge splits on the boundary.

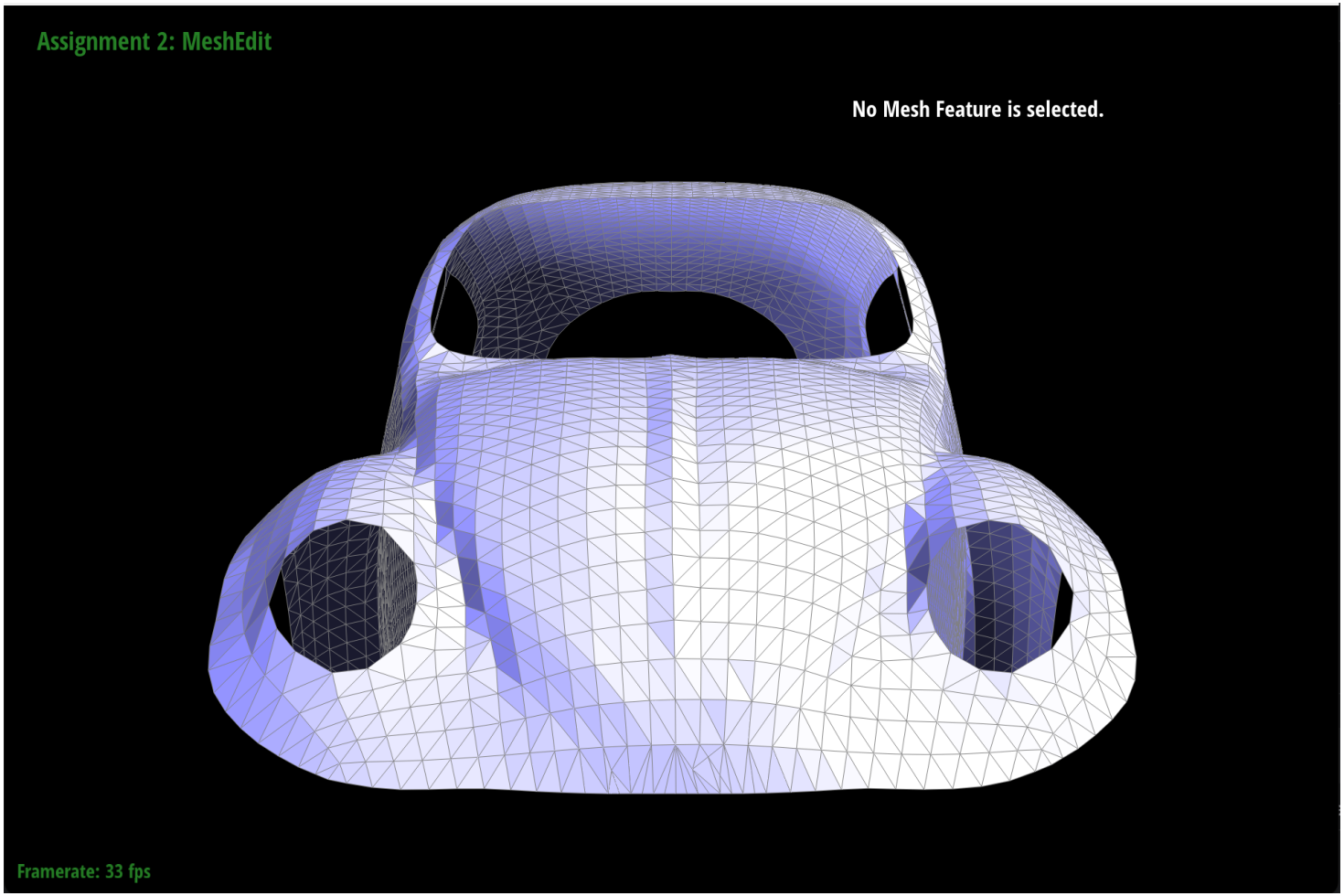
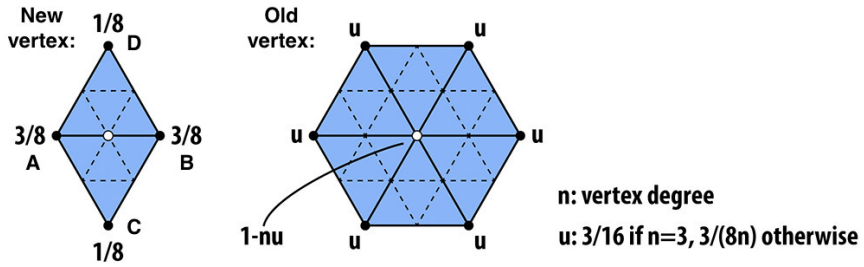


Figure 19: The Beetle car frame after a combination of both edge splits and edge flips near the boundary.

6 Part 6

The original Loop subdivision method updates the positions of the old and new vertices after splitting the edges. However, as we have seen in discussion sessions, after N rounds of Loop subdivision for some $N \in \mathbb{Z}_+$, there are 4^N times the number of triangles in the original mesh. Traversing the new mesh and finding the correct vertices to update becomes complicated and computationally intensive. In our C++ program, we implement an equivalent version of Loop subdivision in the function `MeshResampler::upsample` while slightly changing the order of computations. Unlike Catmull-Clark subdivision that uses new vertices to update the positions of the old vertices, Loop subdivision only uses the existing vertices to compute the new positions of the old and new vertices. Therefore, we can create an instance variable `newPosition` associated with each vertex and edge (which is used to store the new position of the new vertices). We can use the diagram below to help us understand the vertex update process.



We first iterate through all edges in the mesh to compute the positions of new vertices. For a given edge, we can use `halfedge` method to find a halfedge `h1` associated with this edge, and use `twin` method to find the other. If either of the halfedge is on the boundary, then the edge is also on the boundary, we set the `newPosition` value to the midpoint of the two vertices of the edge. If the edge is not on the boundary, with `next` method, we can get traverse all the vertices and assign the weight according to the left diagram to compute the value of `newPosition`.

Next we iterate through all vertices in the mesh to compute the positions of old vertices. For a given vertex, we can use `halfedge` method to find a `start` halfedge originating from the vertex. If the vertex is on the boundary, we want to want to find two halfedges `h` and `h2` originating from the vertex with h on the boundary and `h2->twin()` on the boundary. We can iterate through the outgoing halfedges in the clockwise rotation with the `twin()->next()` combination. Once we find the desired h and h2, we can use twin method to get the neighboring vertices on the boundary and assign them $\frac{1}{8}$ weight each $\frac{3}{4}$ weight to the vertex itself, and store the computed `newPosition` value to the vertex. If the vertex is not on the boundary, we can iterate through the outgoing halfedges in the same pattern with `twin` and `next` methods and then add all the neighboring vertices to a vector `nv`. We can then compute the `newPosition` value using the weight on the right diagram.

Once we finish computing the new positions for the old and new vertices,

we want to perform the edge split on every edge. However, in a single iteration over the edges, we do not want to split the edges that have already been split. Therefore, we introduce another instance variable `isNew` to track whether a vertex or an edge was added after an edge split. However, as we can see from the diagram in Section 5, each edge split on the boundary introduces two new edges whole the two edges that were originally part of the old edge are not marked new. However, tracking these flags inside the `upsample` function does not seem easy. It seems more natural to add these flags when we reassign pointers in the `splitEdge` function. One problem with setting up the flag in the `splitEdge` function is that we sometimes want to split some edges first before performing the Loop subdivision, and the flags generated in that process can interfere with the `upsample` function. Fortunately, the issue is easy to fix. We can just manually set all `isNew` flags to false when iterating through the vertices and edges to compute `newPosition` values. With the flags correctly set up, we will check the two vertices of each edge and only perform an edge split when neither vertex is new. This process is also error-prone, since we are adding new edges in the midst of iterating through edges, and any minor mistake in previous code can access stale pointers and cause segmentation fault or the GUI crashing. Organizing pointer reassignment clauses into blocks to improve readability in previous sections can help spot incorrect pointer assignment faster.

After splitting the edges, we can make sure that new edges only connect to the new vertices by iterating through all edges in the mesh and flip only the new edges whose two vertices have different `isNew` values (new edges connecting an old vertex and a new vertex). We also want to set all `isNew` flags to false on the edges in this iteration for future rounds of Loop subdivision.

In the final step, we iterate through all vertices in the mesh, update their positions to `newPosition` and set all `isNew` flags to false.

Below we have included pictures of different rounds of Loop subdivisions on the torus. As we can see, sharp corners and edges are smoothed out as the number of rounds increases. As a comparison, we pre-split the right sharp edge of the torus into many smaller edges. We can see that the subdivision algorithm has a much harder time to smooth out the sharp edge, and the impact of the original "sharpness" is still visible at Round 4. We can see that pre-splitting some edges does reduce the smoothing effect on sharp edges of Loop subdivision.

We have also included pictures of different rounds of Loop subdivisions on the cube. As we can see, the cube becomes slightly asymmetric after repeated subdivisions. Upon a closer look, we can see that the asymmetry mainly happens at the vertices of the original cube. When we check the original cube, we can see that some vertices have degree 4 and some vertices have degree 5, and the mesh does not have high symmetry. These vertices will become extraordinary vertices where the smoothness of the surface will be lower. We perform edge split on every diagonal across the face, so that after the splitting, the mesh is mirror symmetric, radial symmetric, and inversion symmetric. Ideally we want every vertex to have degree 6, but it is not possible due to Euler's formula for polyhedra. As we increase the rounds of Loop subdivision, we can see our preprocessed cube is less asymmetric, and mirror symmetry, radial symmetry,

and inversion symmetry are preserved. One reason why our preprocessed cube is more symmetric is that we introduced more symmetry from the beginning. These symmetric structures are preserved under Loop subdivision. In addition, we brought the degrees of vertices closer to 6 in general. Mathematicians have been able to prove that we can achieve C^2 smoothness on vertices with degree 6.

Since we implemented the boundary situation for both `splitEdge` and `upSample` methods, our program supports Loop subdivision on surfaces with boundaries. We have included pictures of different rounds of Loop subdivision on the Beetle car frame.

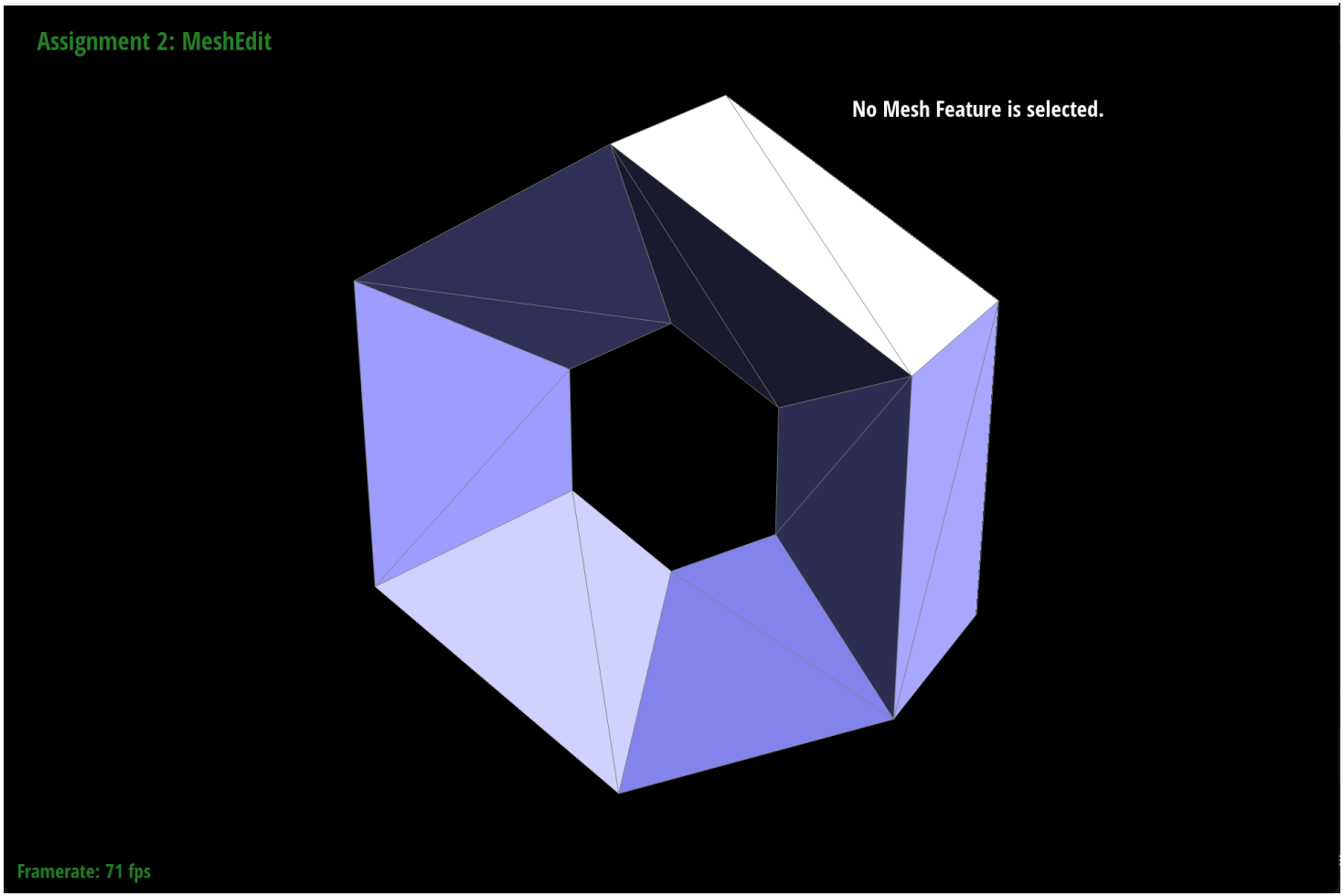


Figure 20: Torus before Loop subdivision.

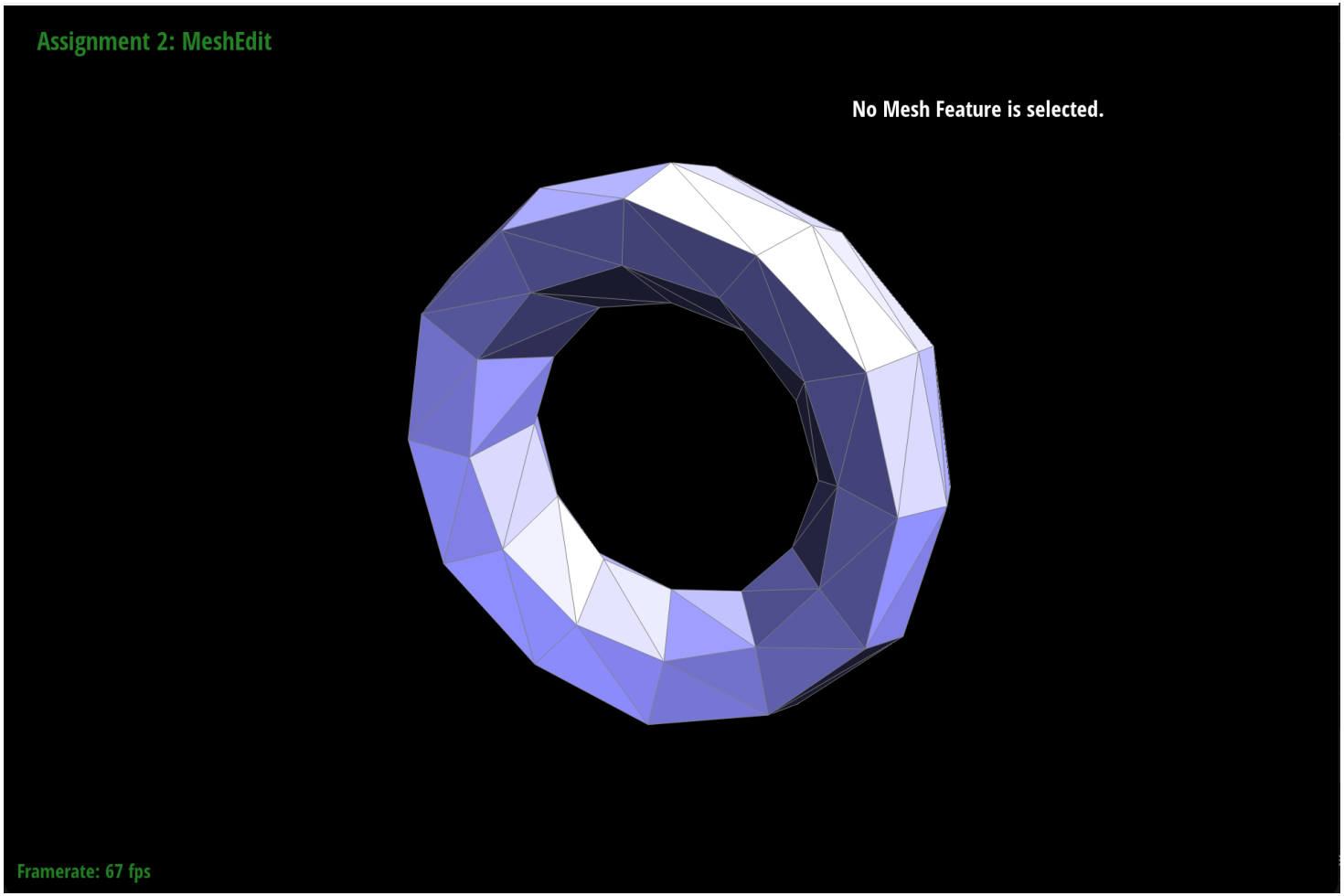


Figure 21: Torus after Loop subdivision Round 1.

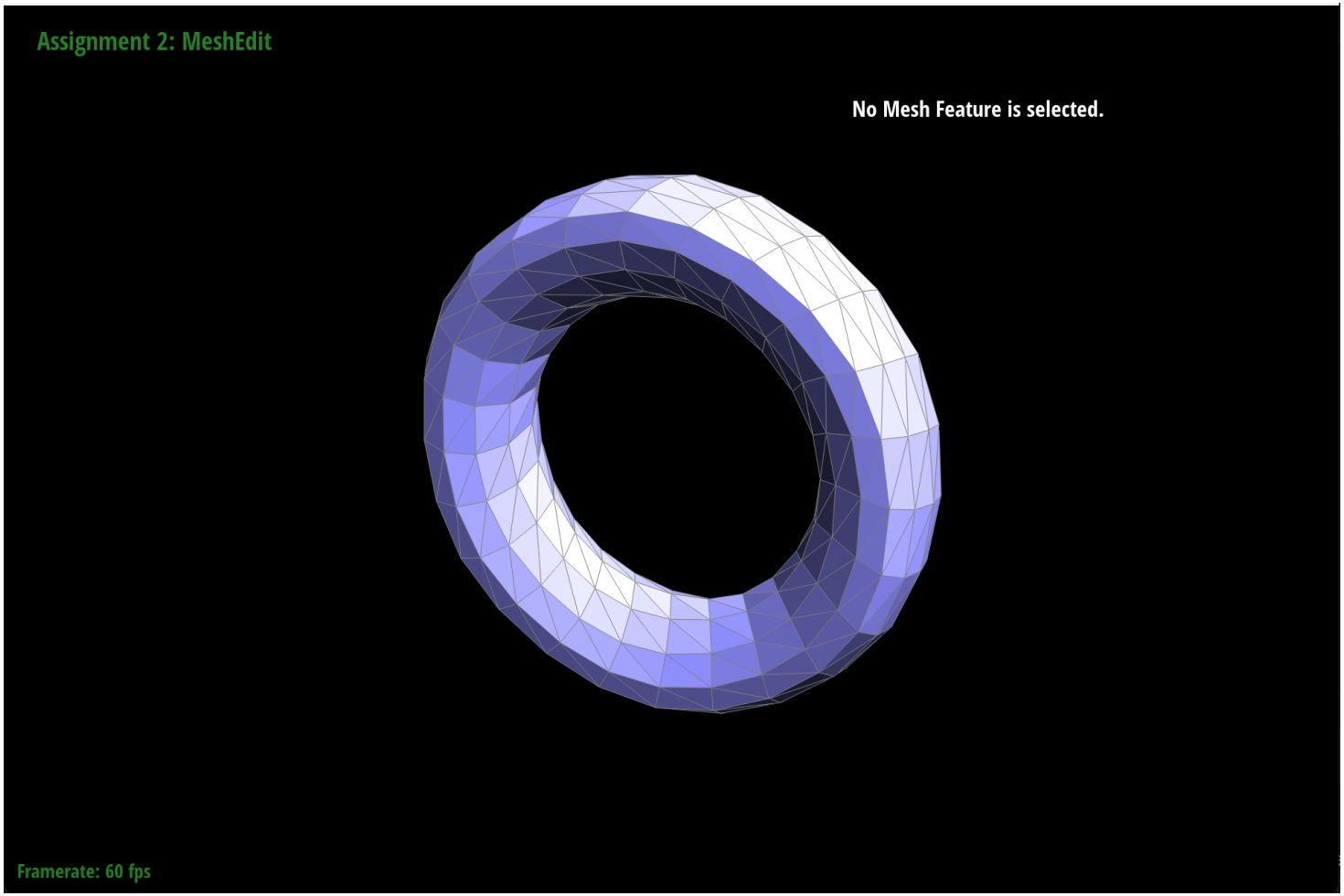


Figure 22: Torus after Loop subdivision Round 2.

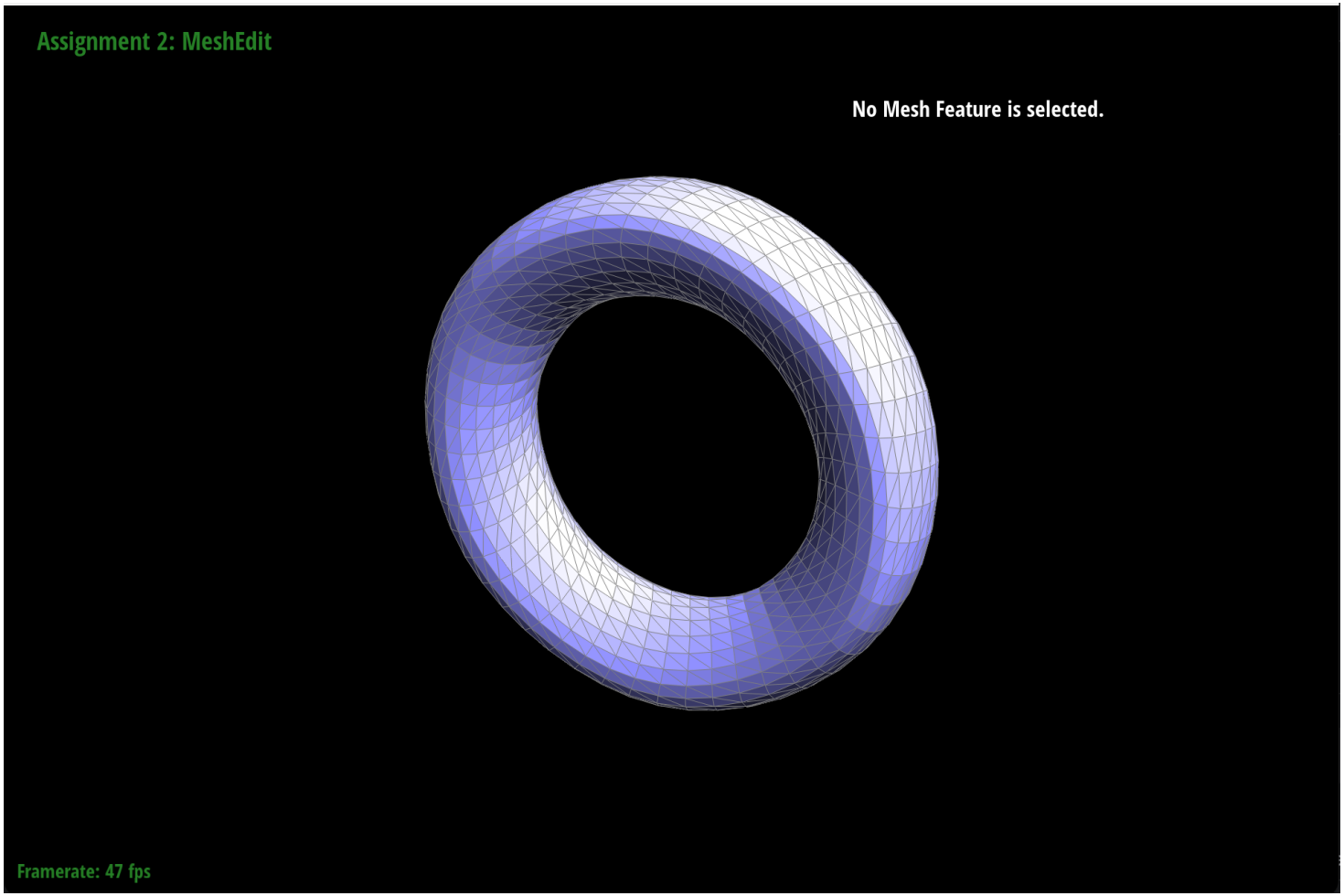


Figure 23: Torus after Loop subdivision Round 3.

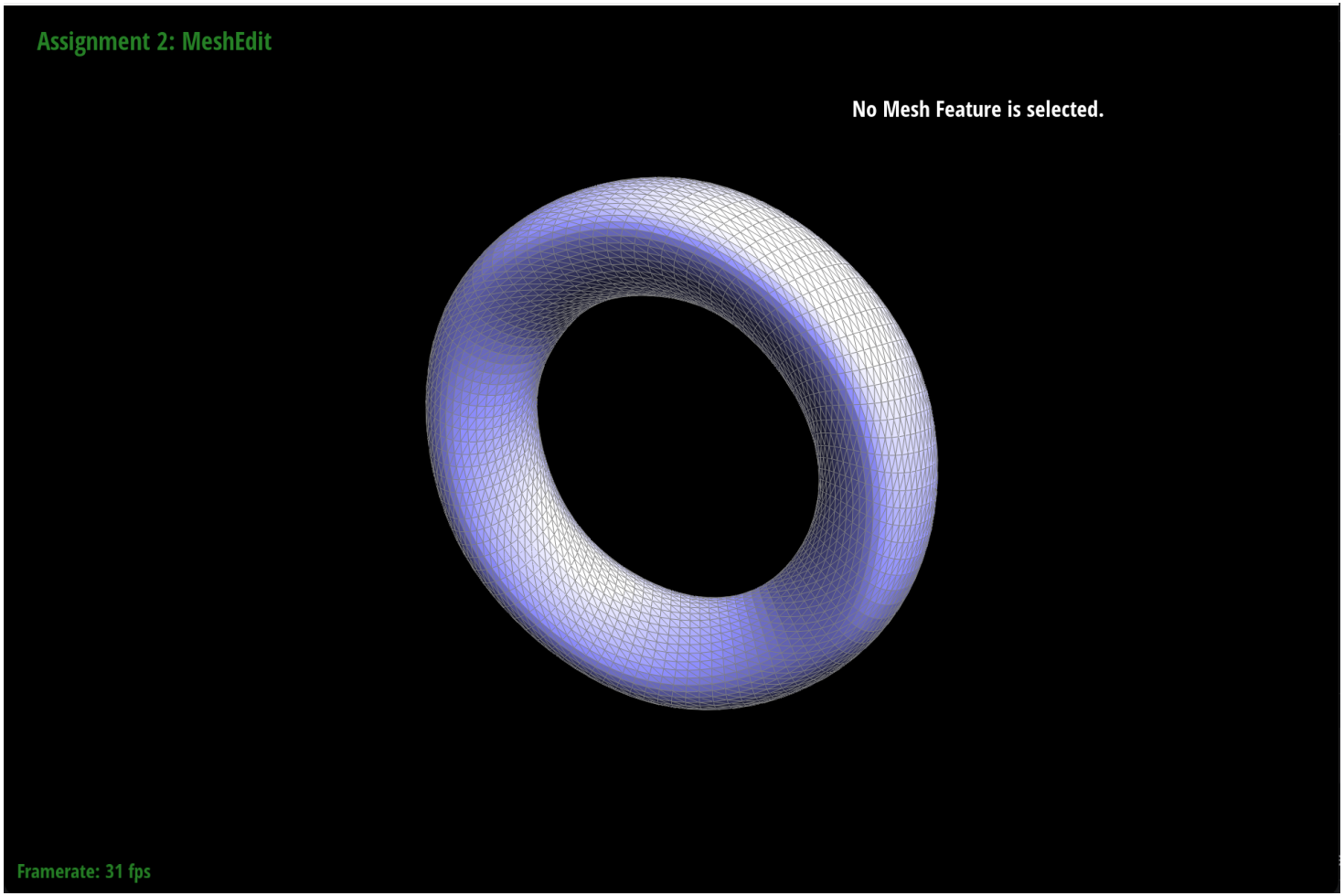


Figure 24: Torus after Loop subdivision Round 4.

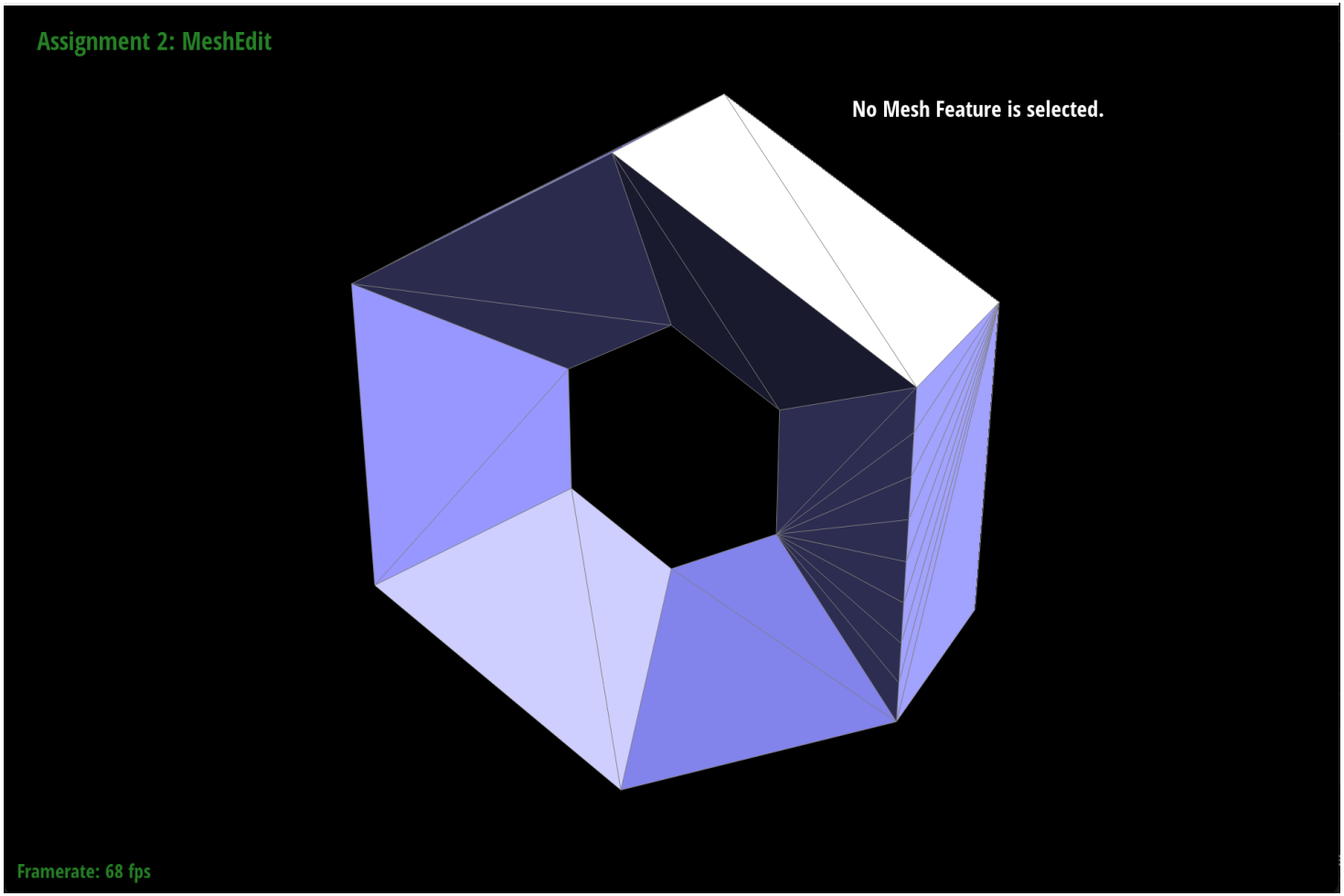


Figure 25: Torus before Loop subdivision. The right sharp edge is pre-splitted for comparison.

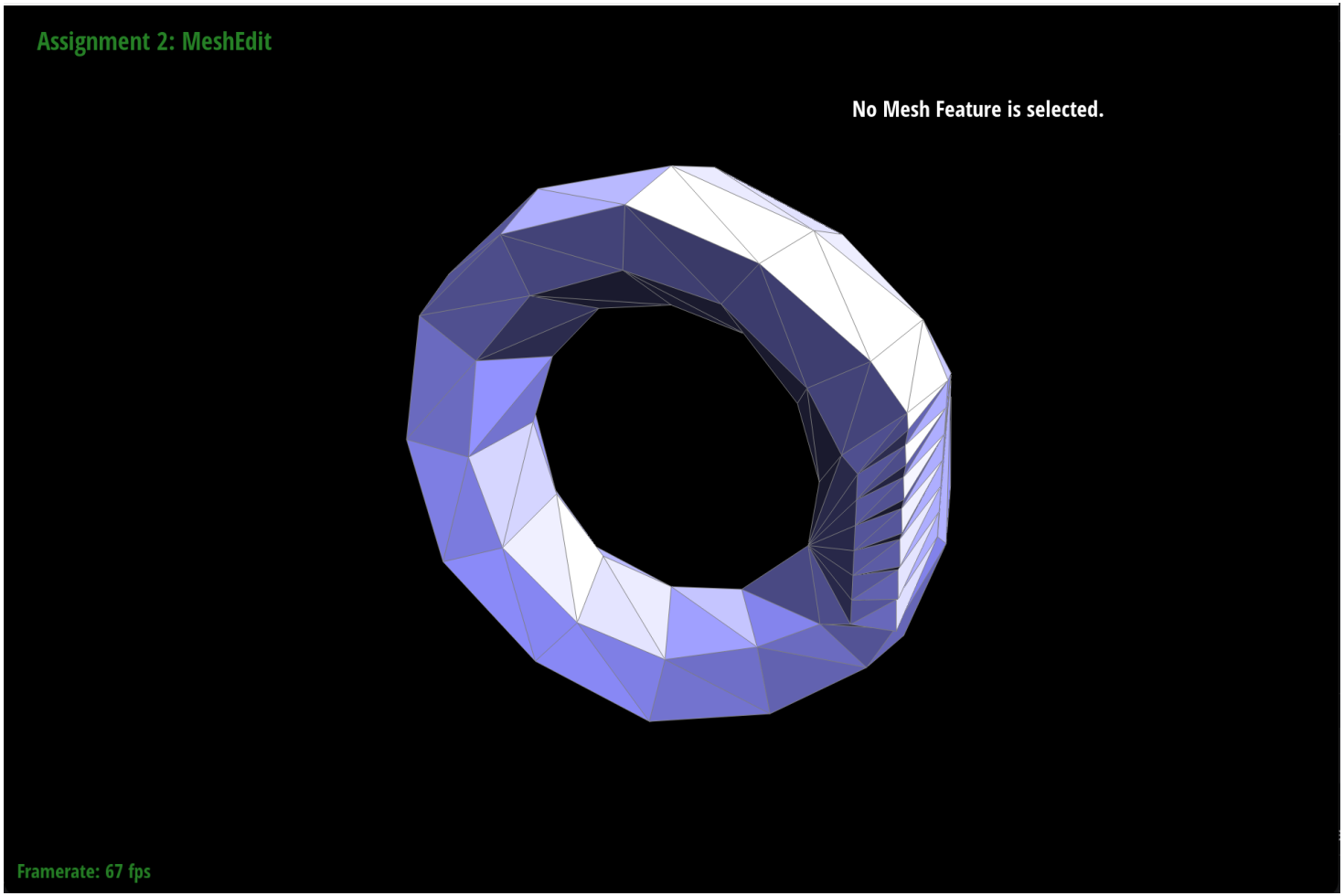


Figure 26: Torus after Loop subdivision Round 1. The right sharp edge is pre-splitted for comparison.

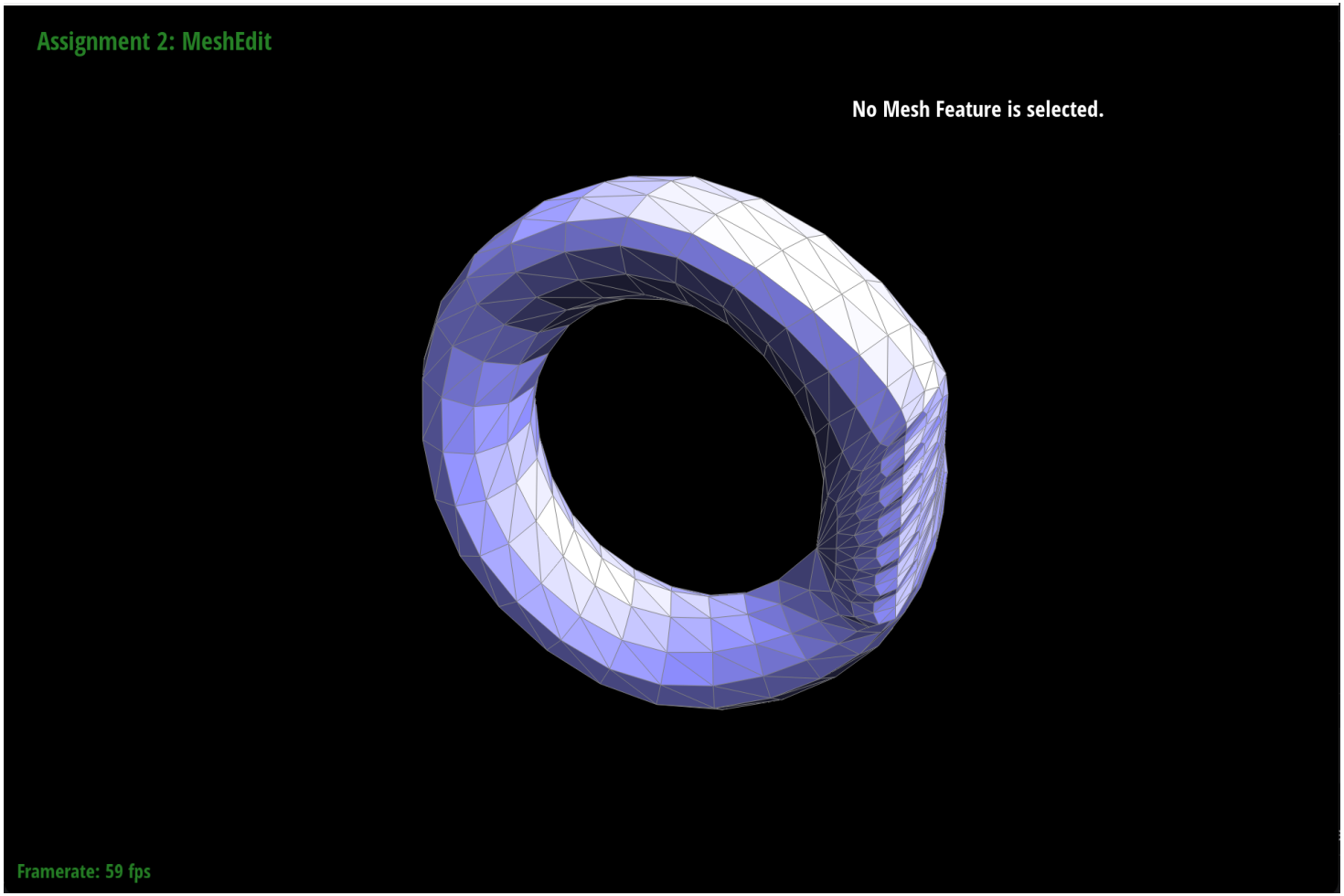


Figure 27: Torus after Loop subdivision Round 2. The right sharp edge is pre-splitted for comparison.

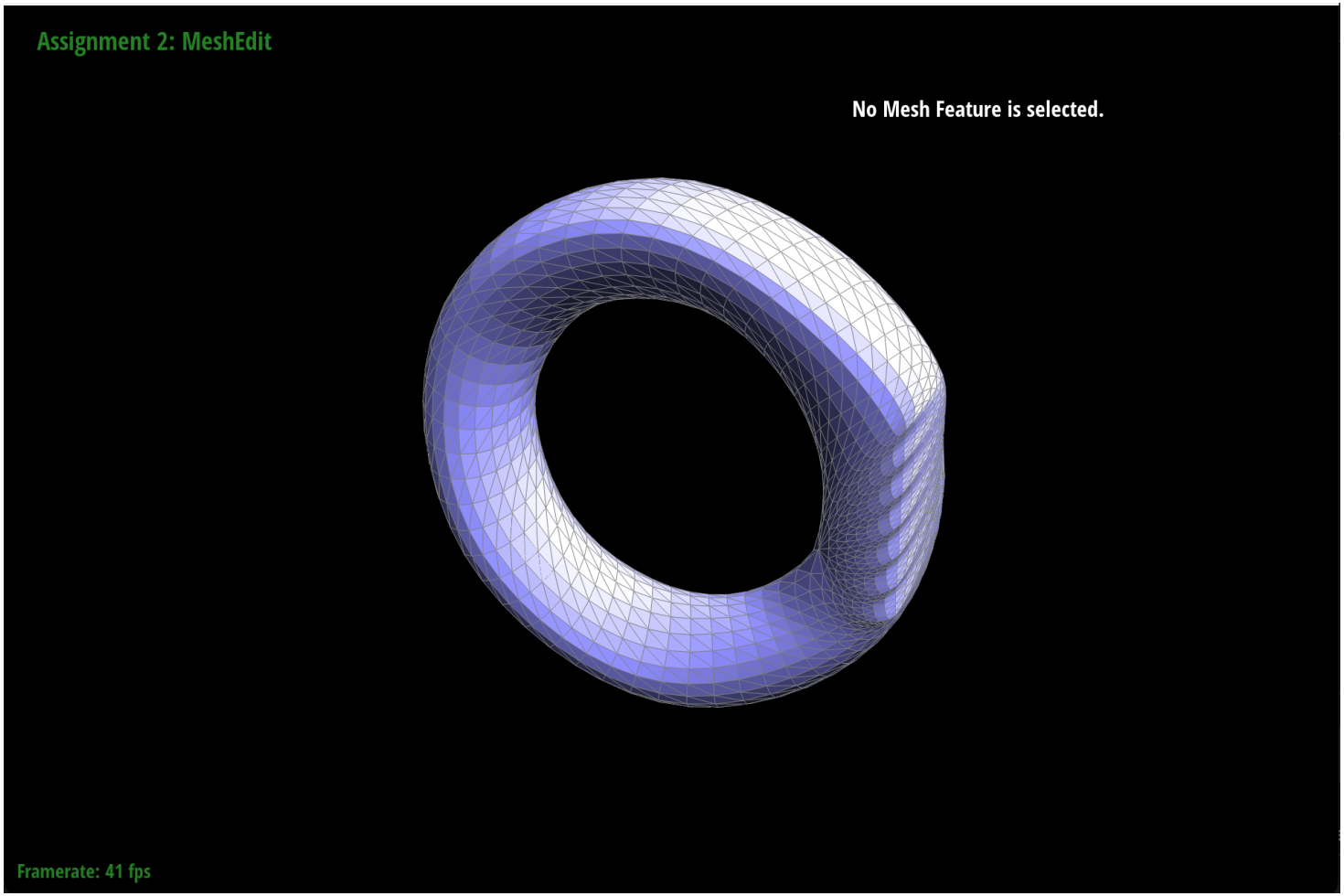


Figure 28: Torus after Loop subdivision Round 3. The right sharp edge is pre-splitted for comparison.

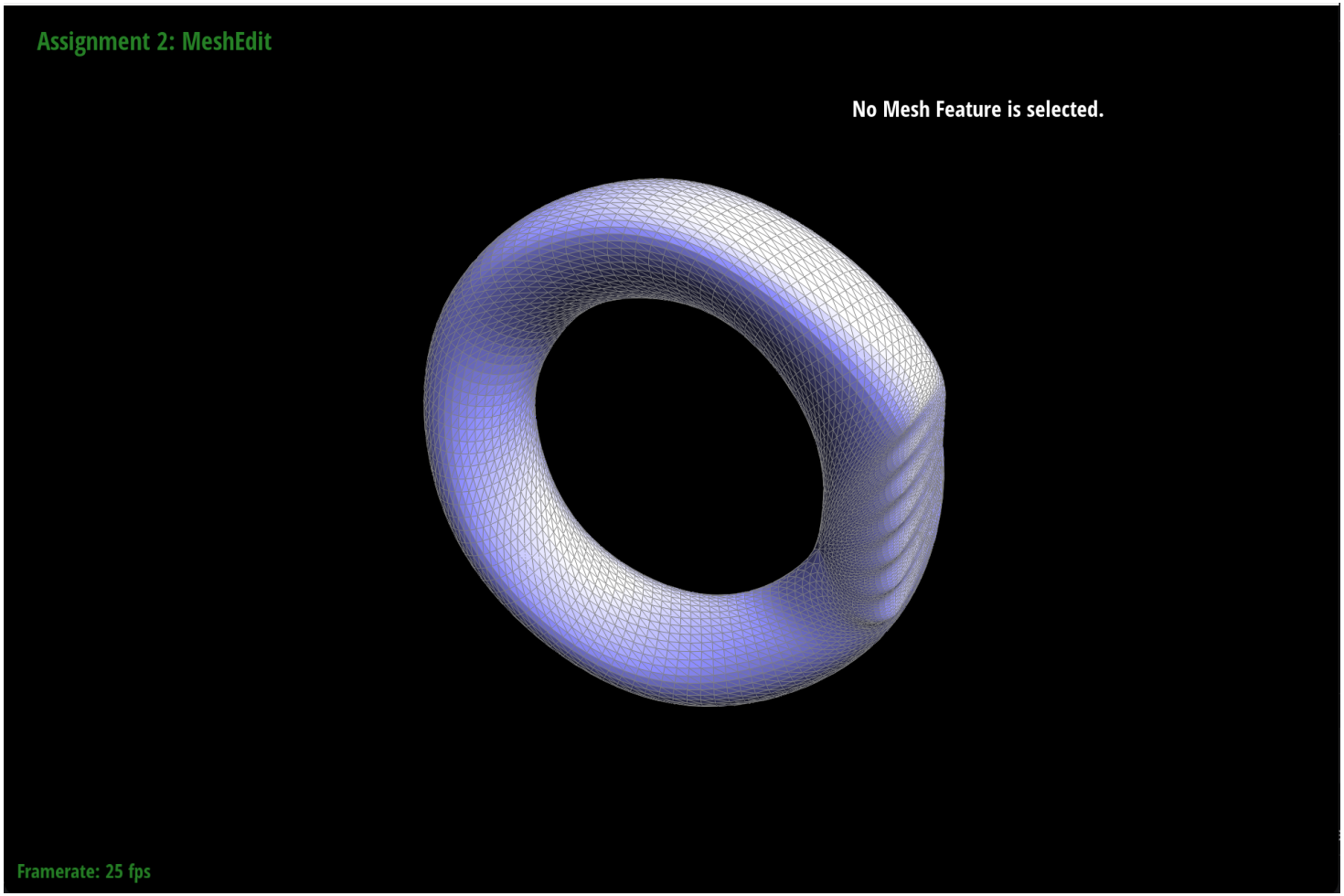


Figure 29: Torus after Loop subdivision Round 1. The right sharp edge is pre-splitted for comparison.

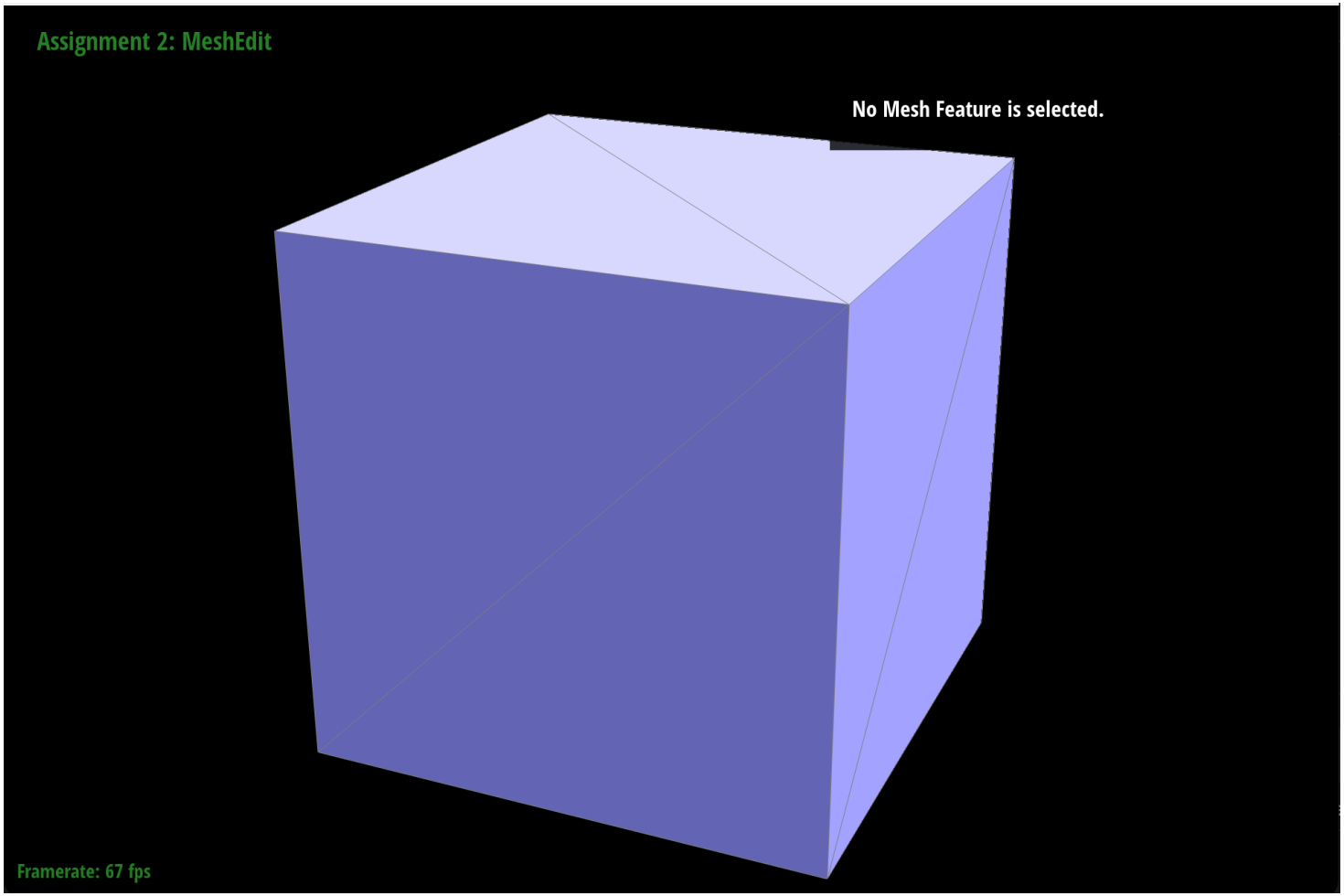


Figure 30: Original cube before Loop subdivision

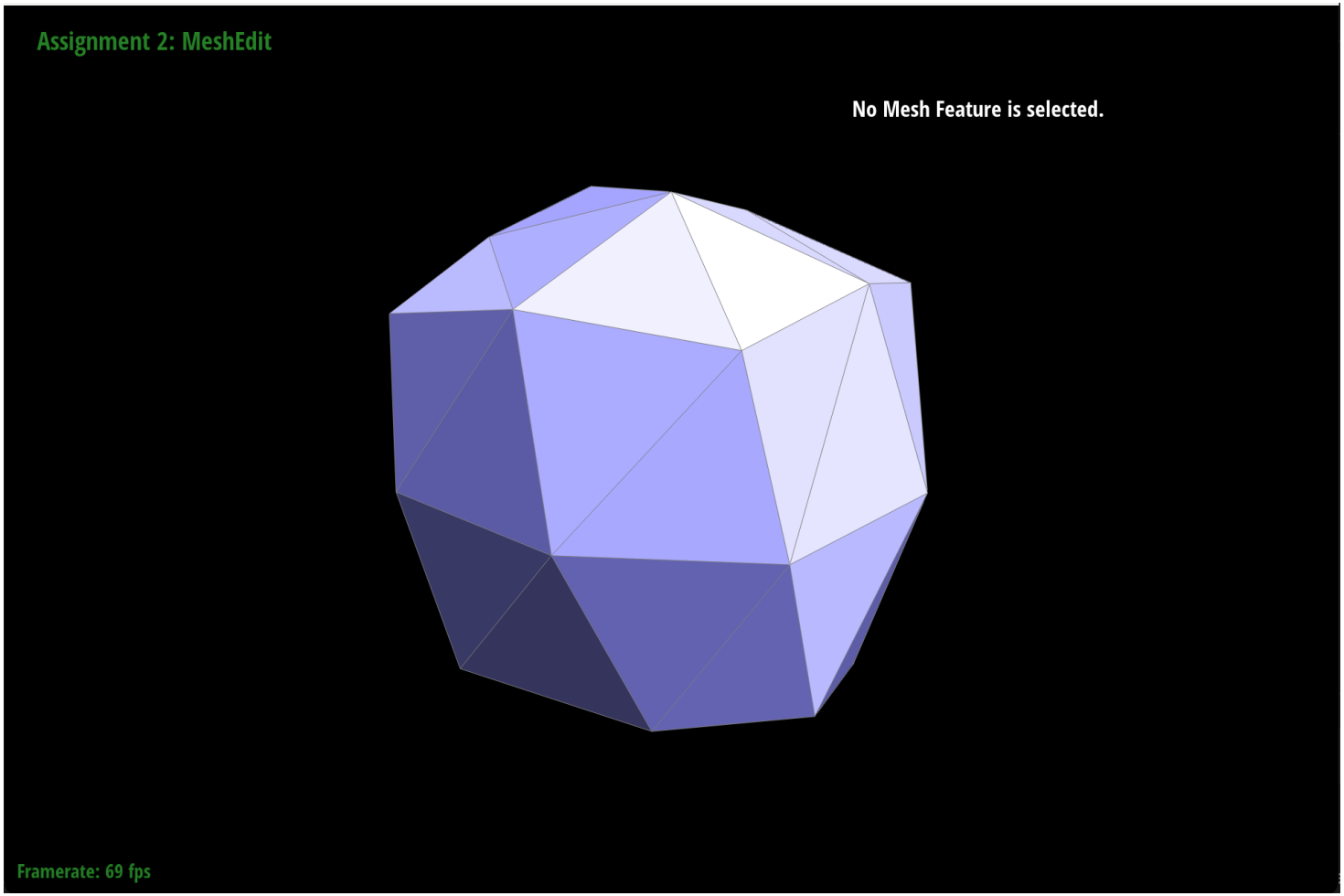


Figure 31: Original cube after Loop subdivision Round 1.

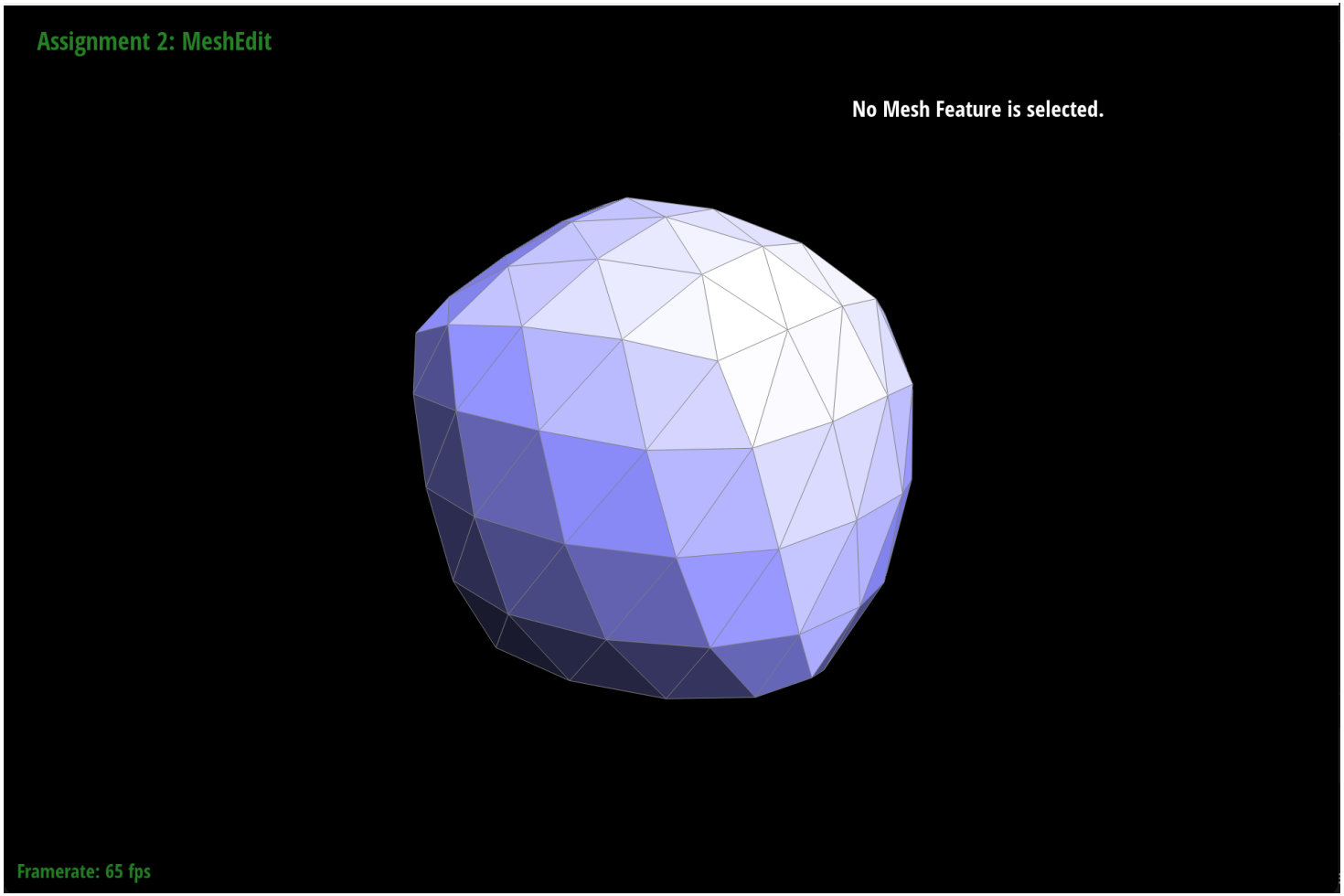


Figure 32: Original cube after Loop subdivision Round 2.

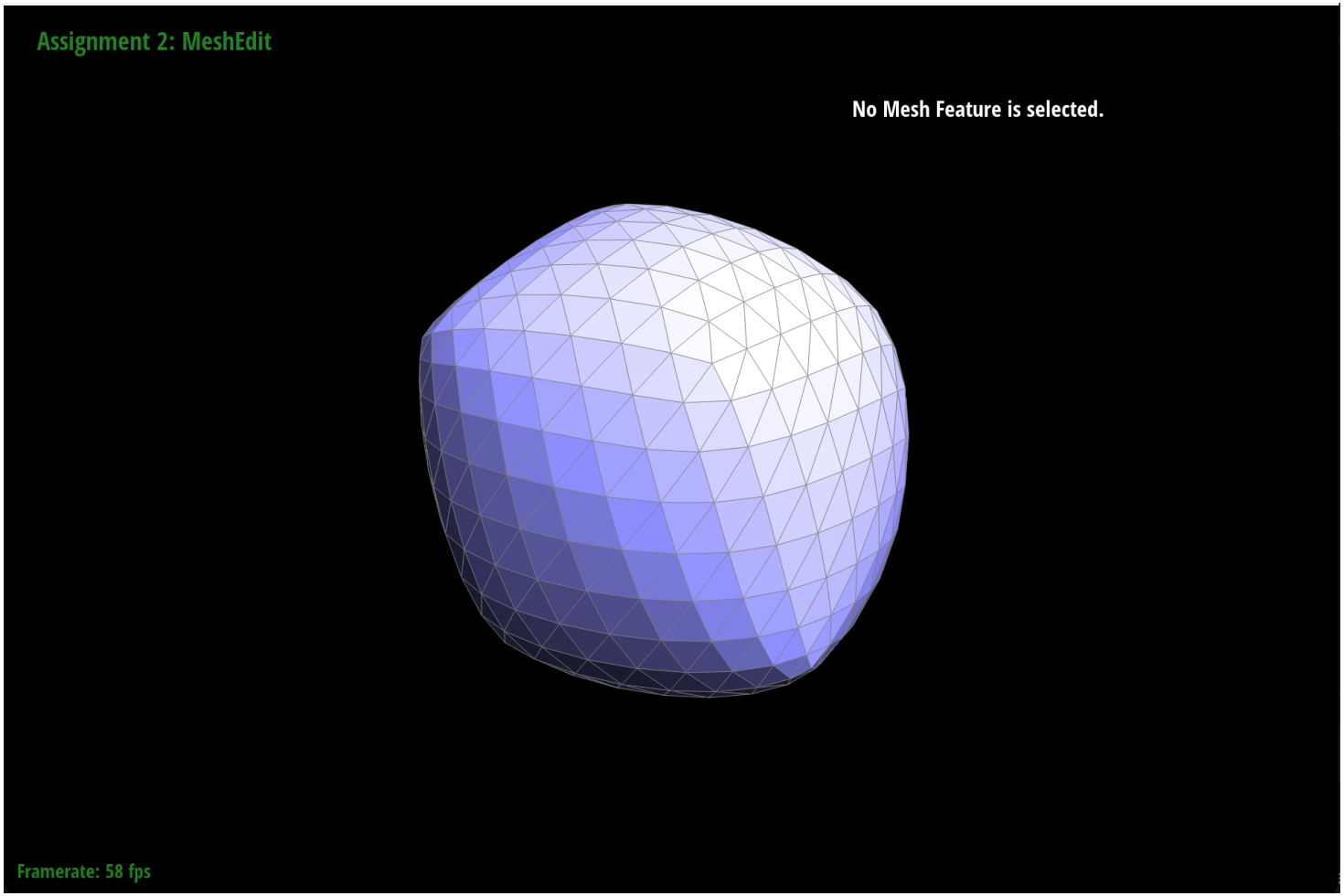


Figure 33: Original cube after Loop subdivision Round 3.

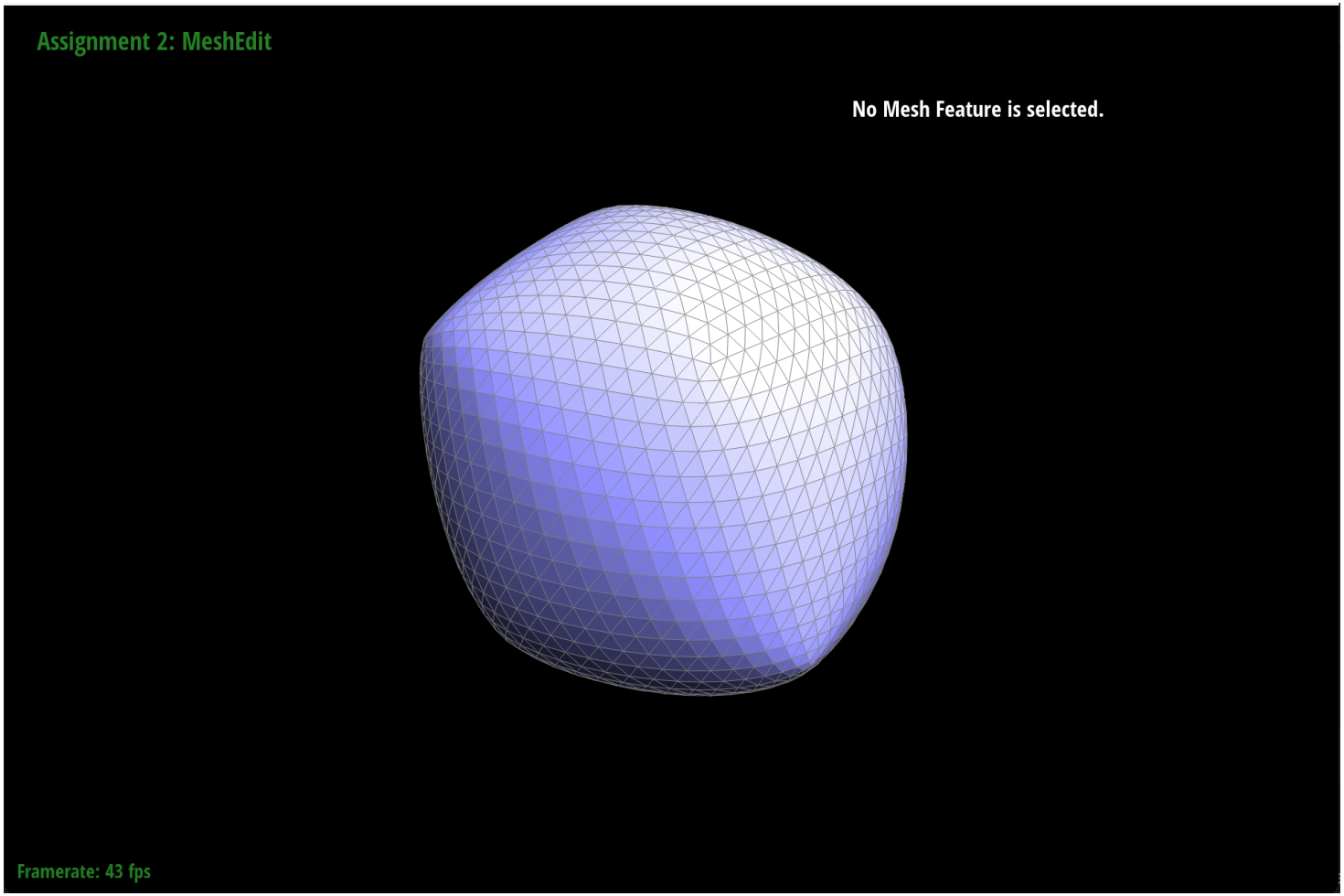


Figure 34: Original cube after Loop subdivision Round 4.

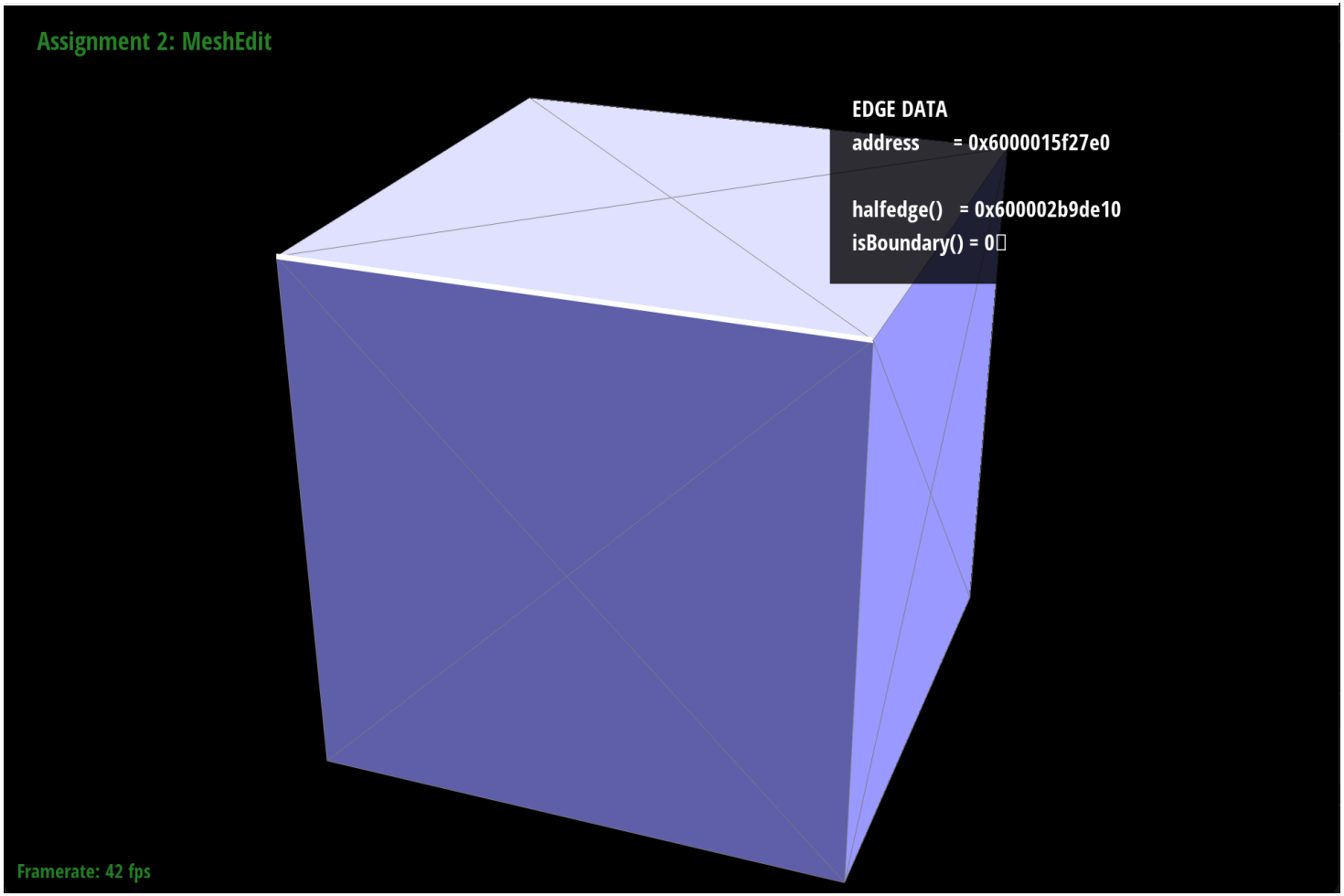


Figure 35: Preprocessed cube before Loop subdivision.

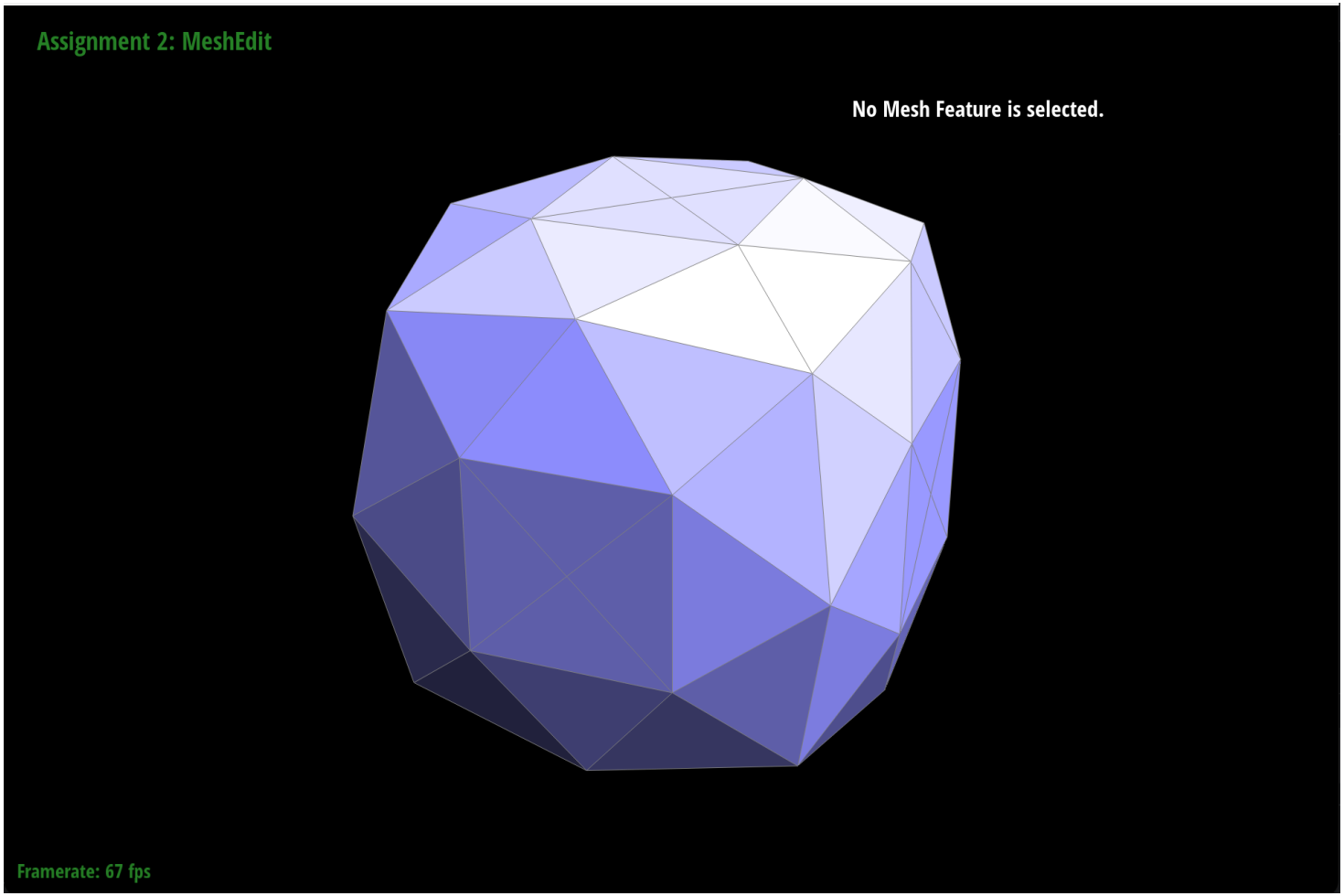


Figure 36: Preprocessed cube after Loop subdivision Round 1.

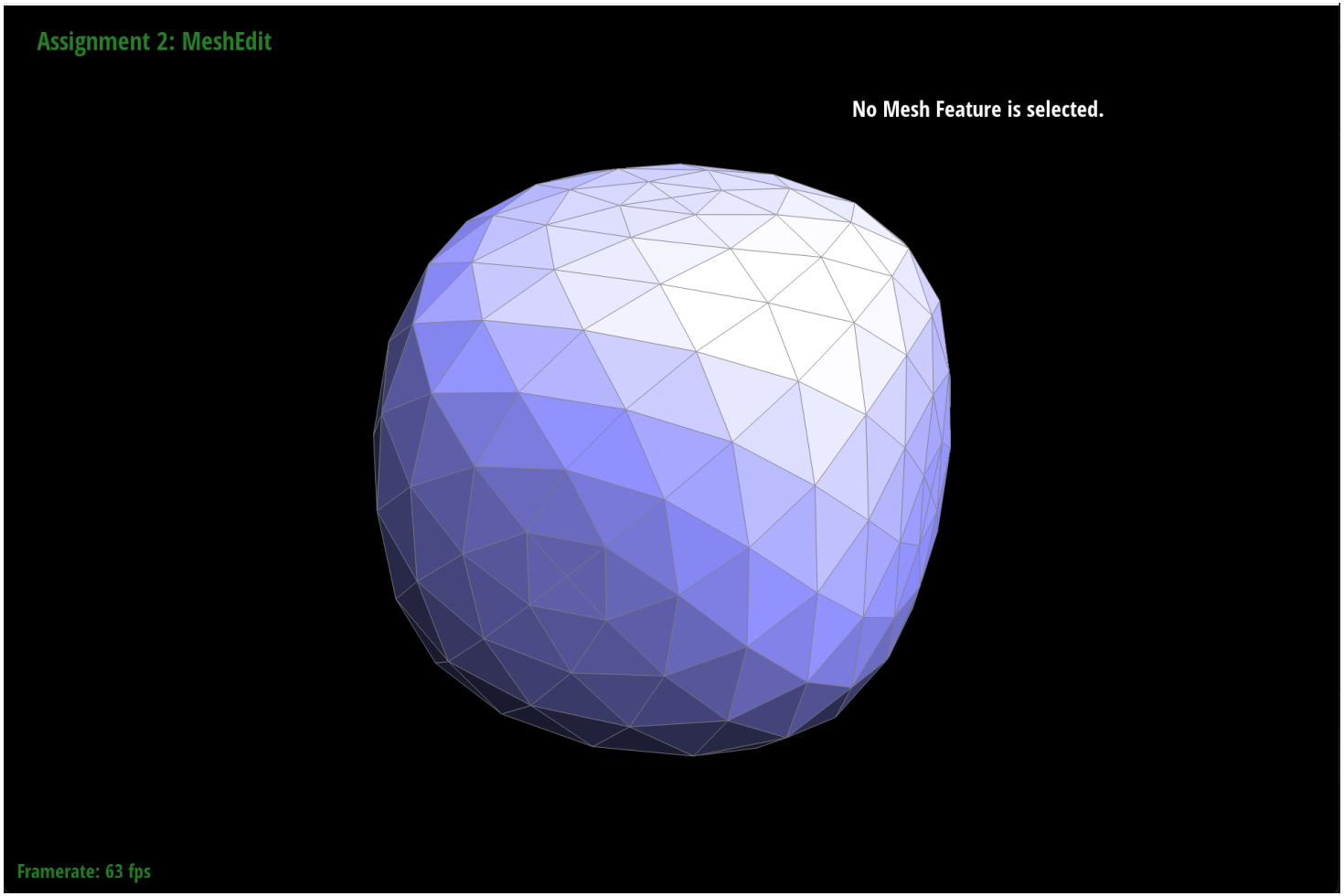


Figure 37: Preprocessed cube after Loop subdivision Round 2.

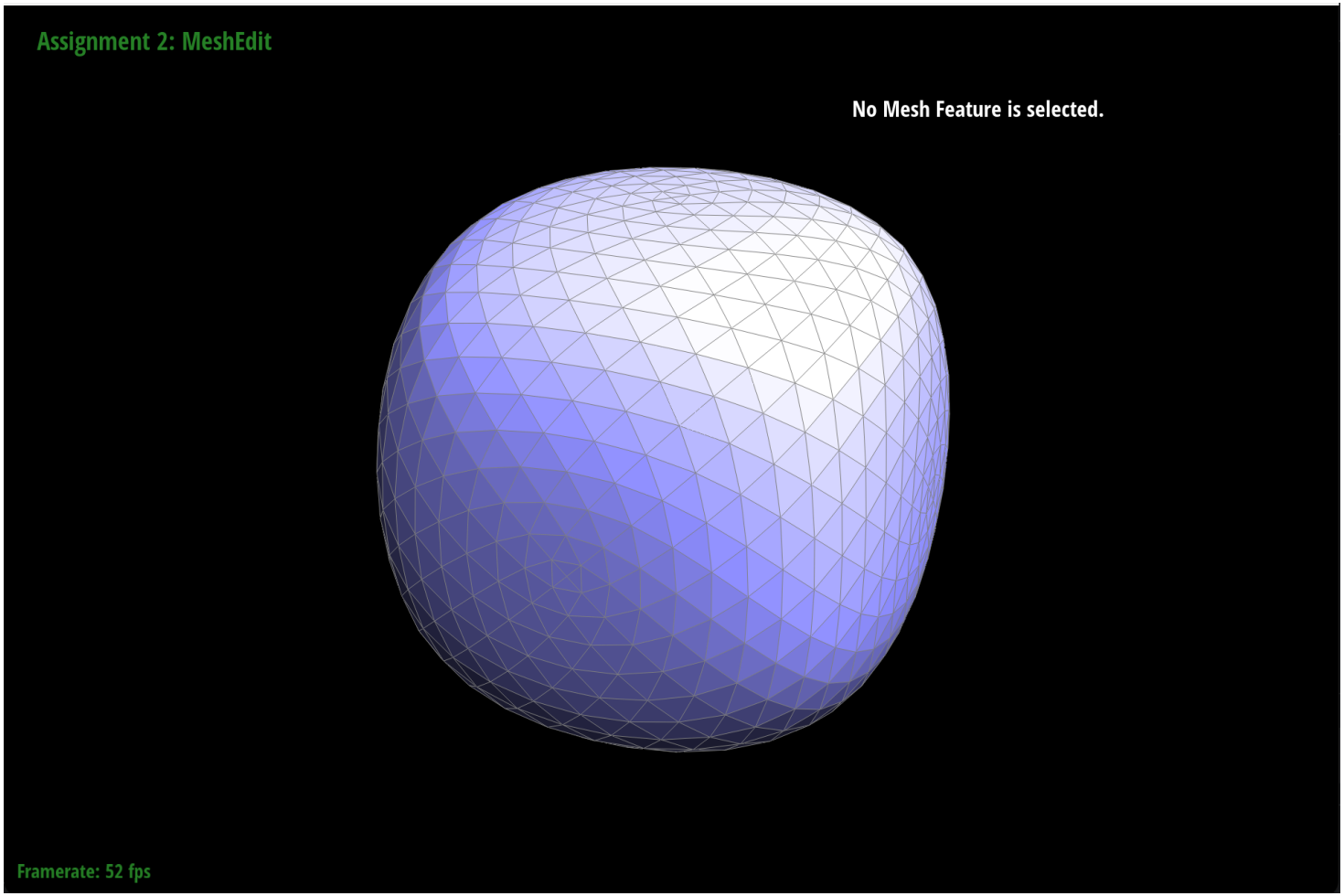


Figure 38: Preprocessed cube after Loop subdivision Round 3.

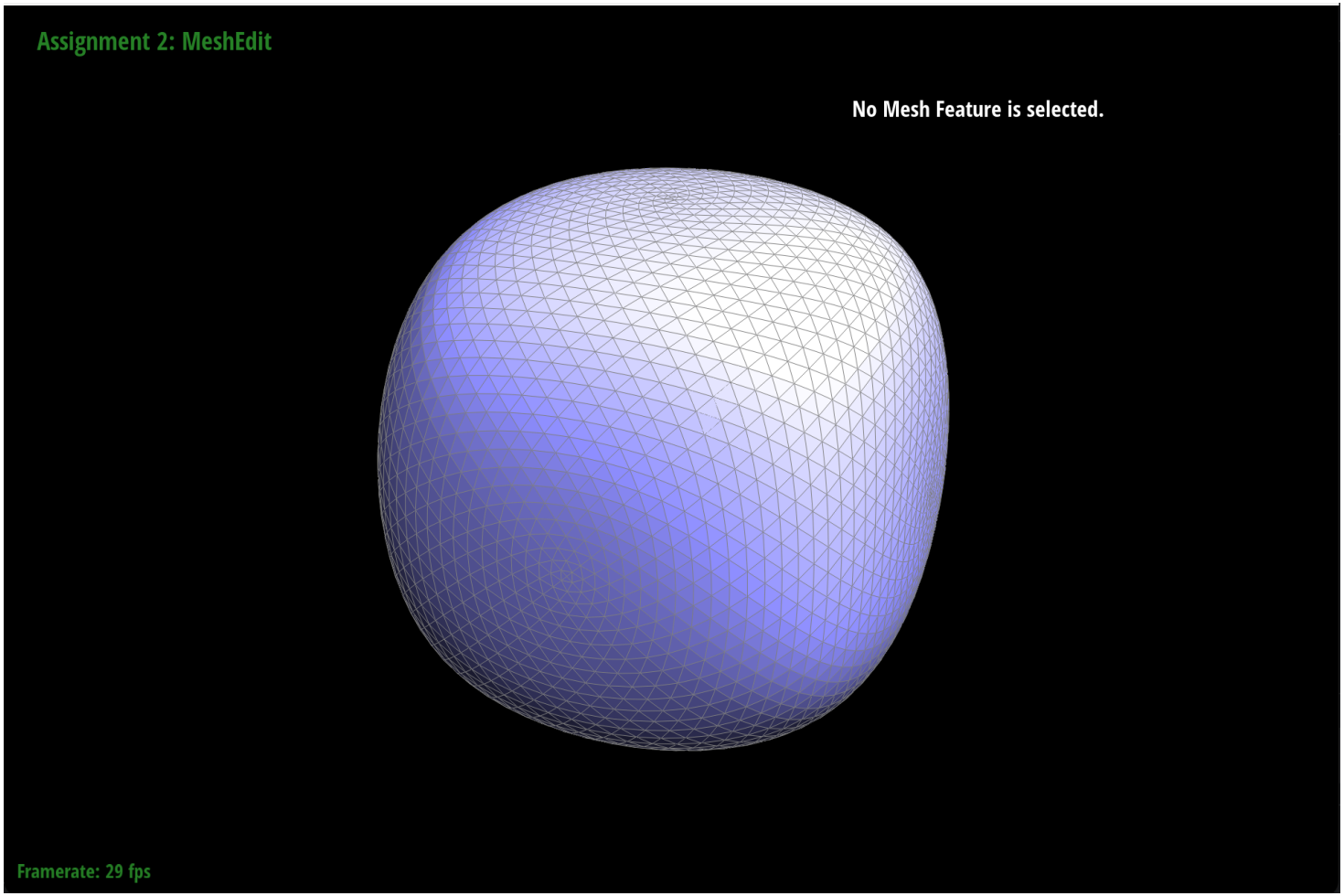


Figure 39: Preprocessed cube after Loop subdivision Round 4.

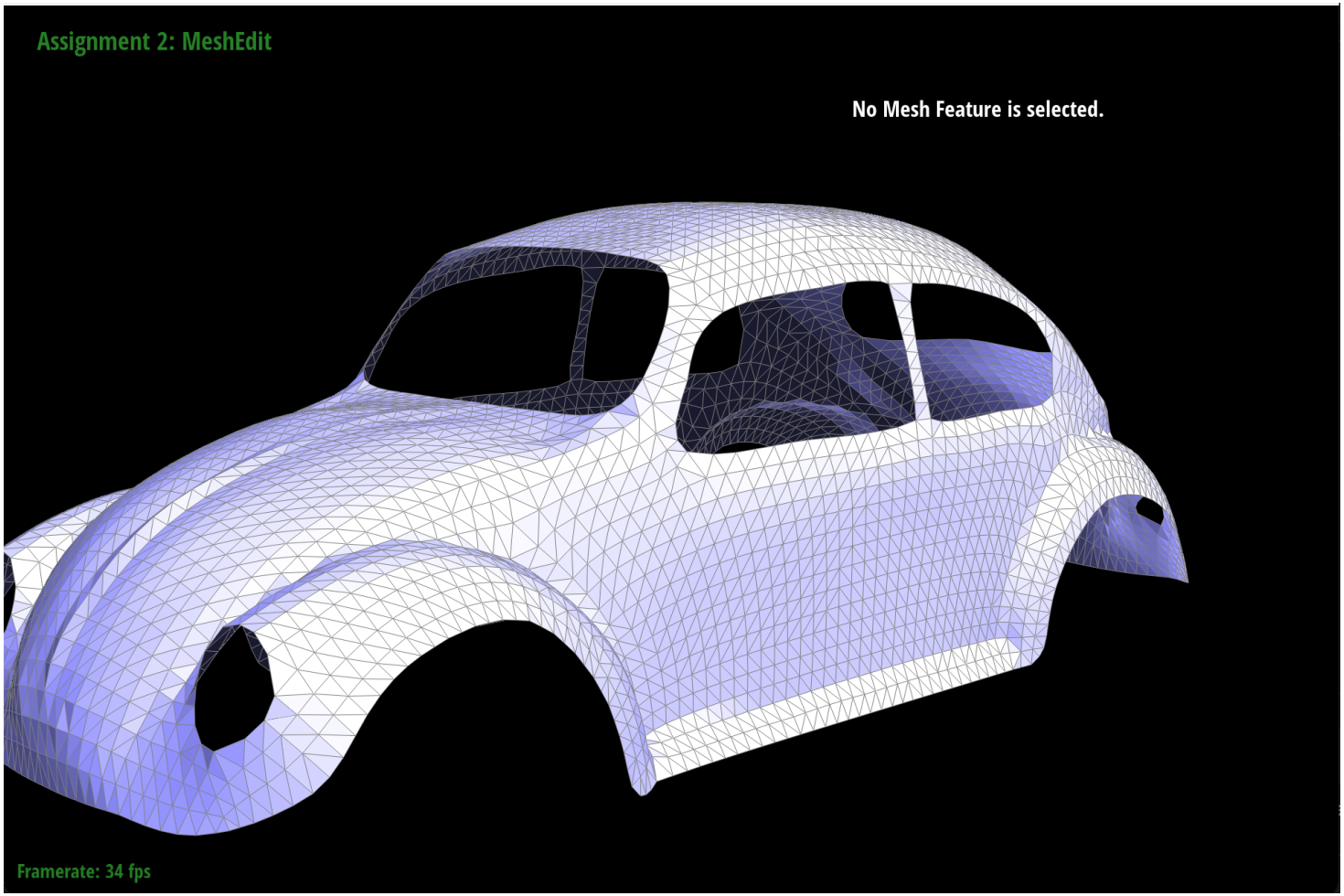


Figure 40: Beetle car frame before Loop subdivision.

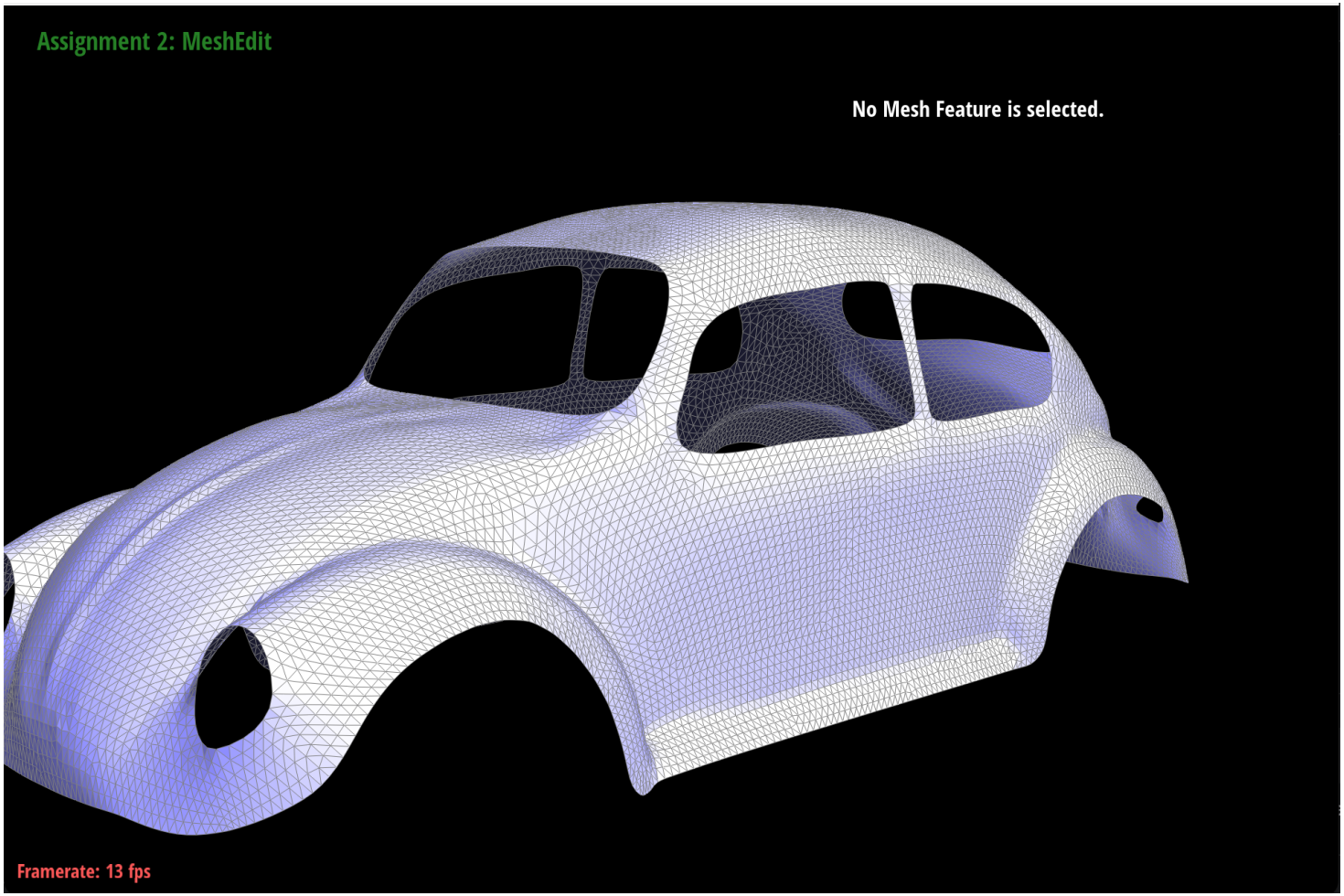


Figure 41: Beetle car frame after Loop subdivision Round 1. Upsampling is supported on the boundary.

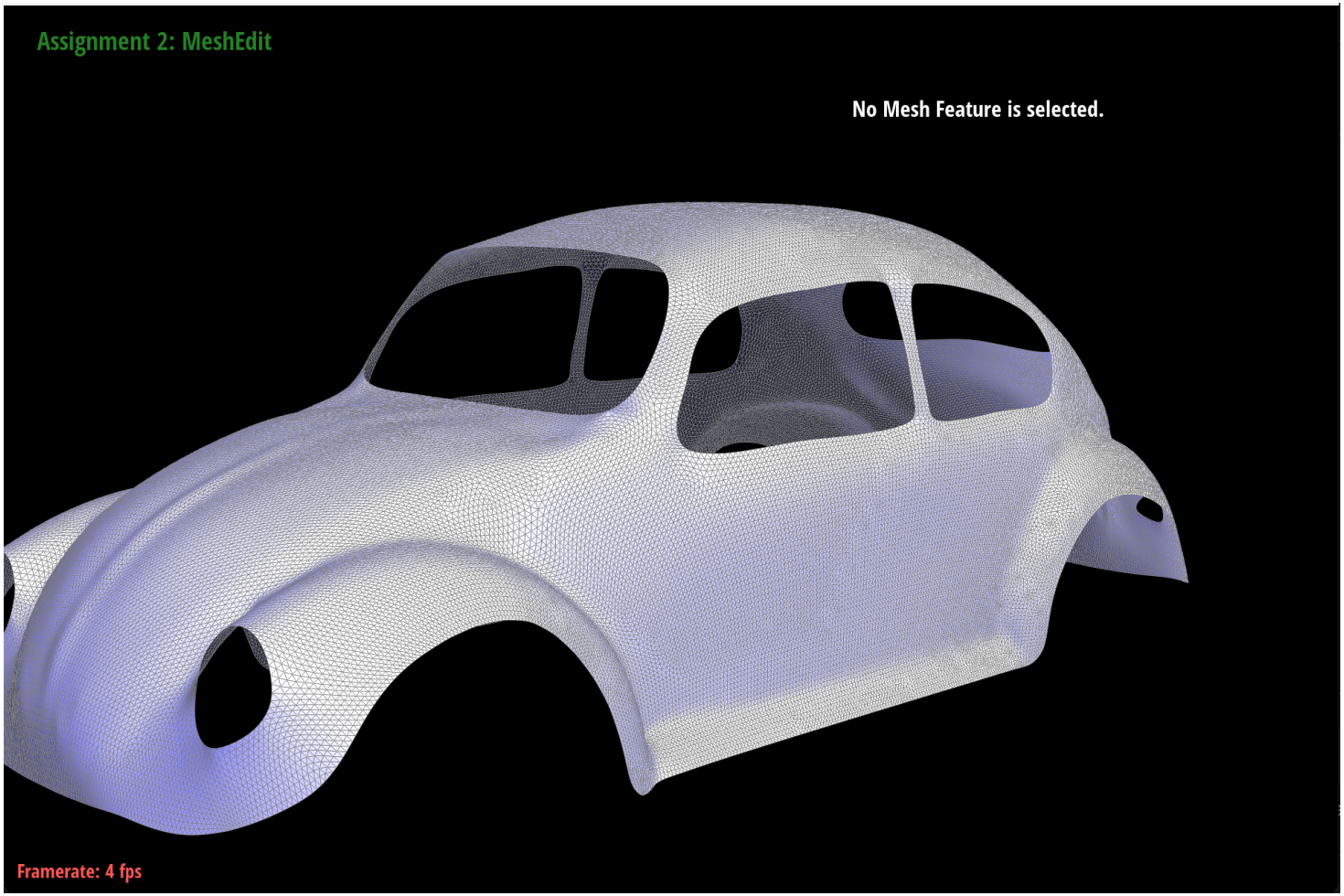


Figure 42: Beetle car frame after Loop subdivision Round 2. Upsampling is supported on the boundary.

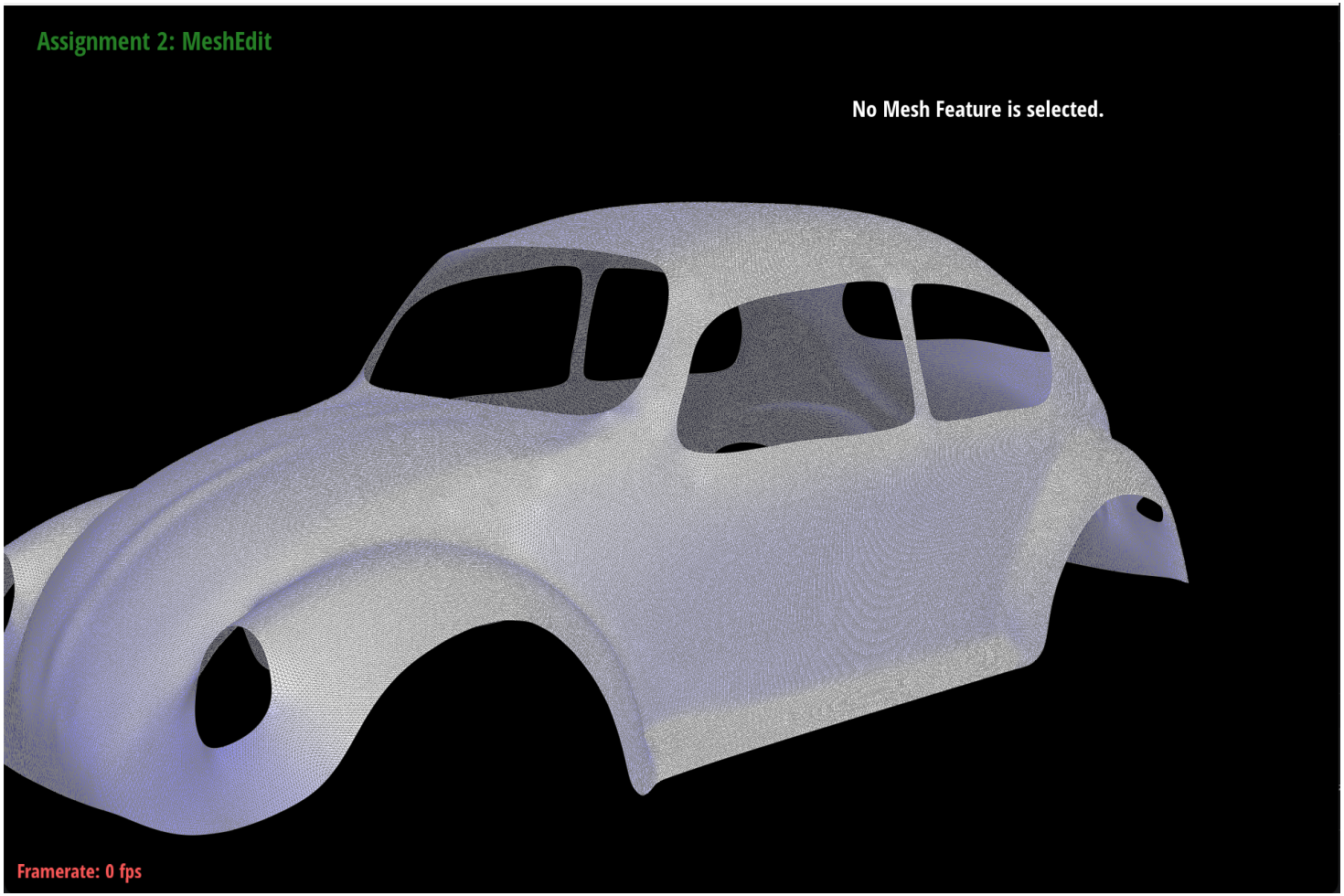


Figure 43: Beetle car frame after Loop subdivision Round 3. Upsampling is supported on the boundary.