CS 184: Computer Graphics and Imaging, Spring 2020

Project 2: Mesh Editor

Nico Deshler, CS184

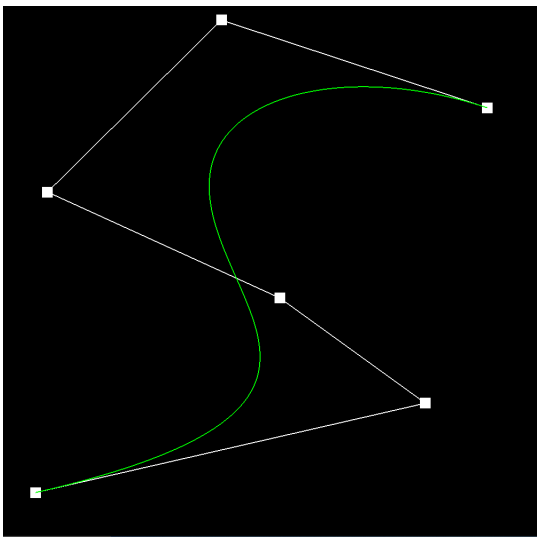
Overview

In this project we were tasked with implementing the De Casteljau Algorithm for Bezier Curves/Surfaces, and implementing elementary mesh operations (i.e. edge-flip, edge-split, and upsampling) for a Mesh Editor using the halfedge data structure. Succesful implementation of these features is central to processing geometry in computer graphics. Through this project I consolidated my understanding of Bezier curves/surfaces and became fluent with mesh traversal using the halfedge data structure. The images and results shown in this write-up demonstrate the variety of geometric objects and processing schemes that can be acheived using the features implemented herein.

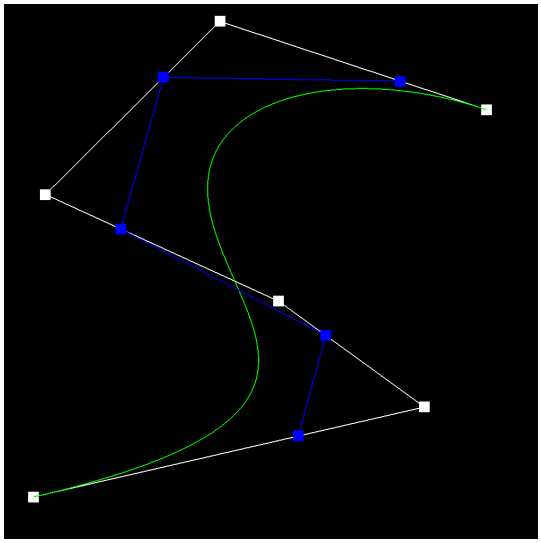
Section I: Bezier Curves and Surfaces

Part 1: Bezier curves with 1D de Casteljau subdivision

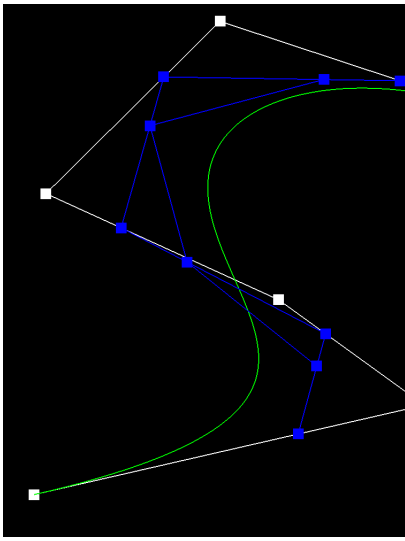
The de Casteljau Algorithm is a method for computing the position of a point on a Bezier Curve parametrized by the scalar t which can vary from 0 to 1. Given a list of n control points p_1 ... p_n we can define points on bezier curve by recursively performing linear interpolations between the points and evaluating each line segment at the fractional length t. Mathematically, the recursive step can be defined as p_i' = (1-t)p_i + tp_i where the set of primed coordinate points is length n-1. In this implementation, we define a subroutine for the BezierCurve class called BezierCurve::EvaluateStep() which takes in a list of points and a parameter t and performs a single step of the recursion defined previously, returning a list of linearly-interpolated points. This elementary function is later used as a helper for complete descriptions of Bezier Curves and Surfaces. The image sequence below shows the progression of the De Casteljau algorithm updates on list of 6 points for a fixed value of t to demonstrate how the algorithm converges to a single point on the Bezier curve. The green line defines the true Bezier Curve and is displayed for reference. The blue line segments apparent in iterations 1-5 show the linear interpolations at each step. The red square in Image 5 indicates the final point on the Bezier curve that the algorithm converges to for a fixed value of t. Note that it lies directly on the green reference curve indicating the algorithm's fidelity. The final image below shows the De Casteljau algorithm operating on a list of slightly perturbed points and a different t value.



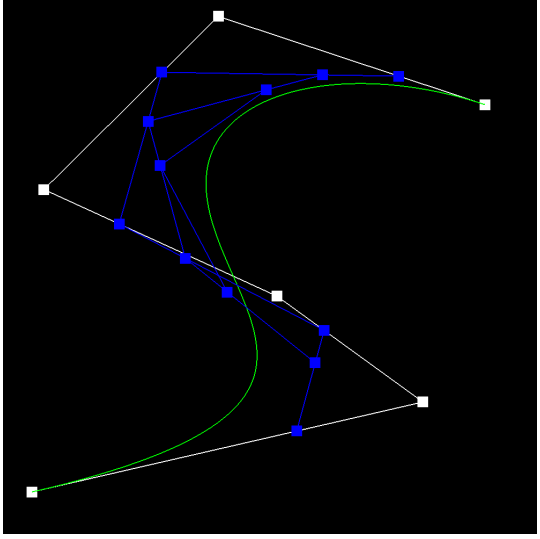
Itertion 0.



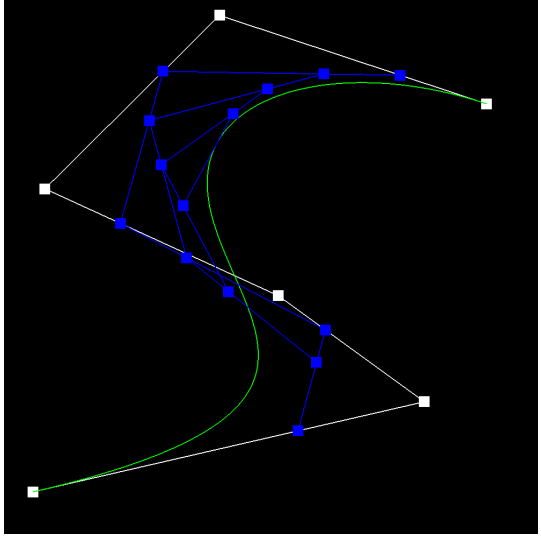
Itertion 1.



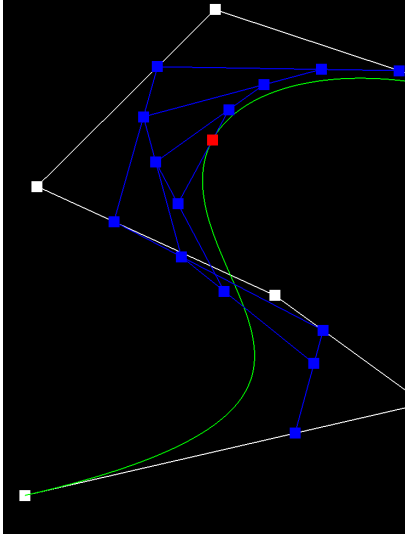
Itertion 2.



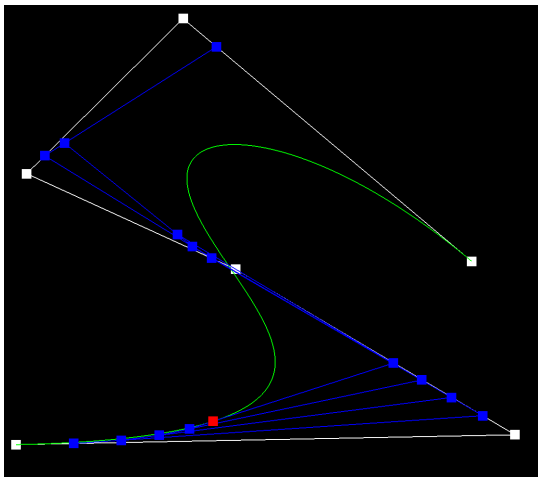
Itertion 3.



Itertion 4.



Itertion 5.



Bezier Curve for perturbed points with a new t value.

Part 2: Bezier surfaces with separable 1D de Casteljau subdivision

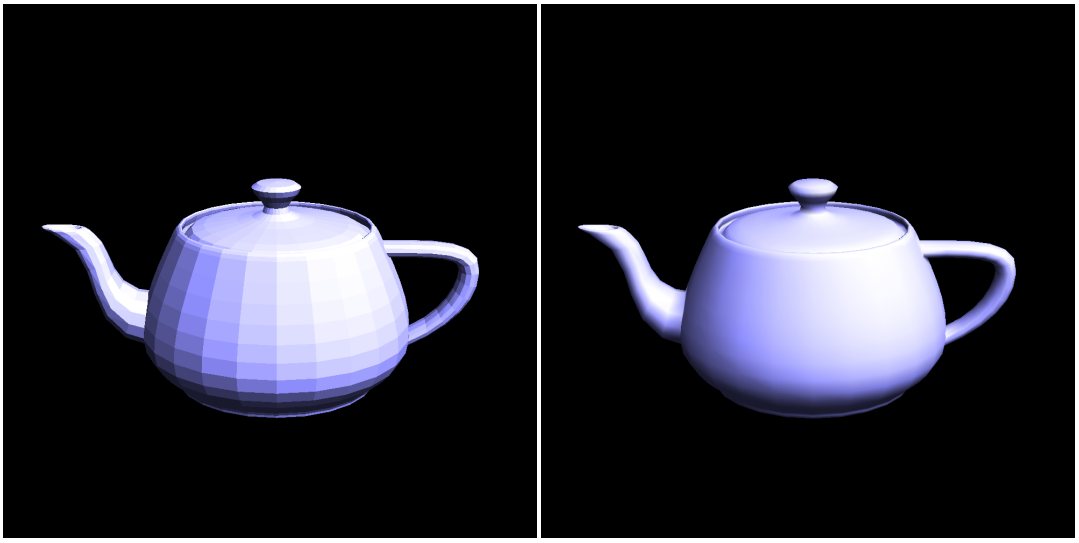
The 1D De Casteljau Algorithm extends naturally to 2D Bezier surfaces. To do so, it must be possible to define the surface using an assortment of 1D Bezier curves. Given a collection of m Bezier *curves* in 3D space each defined with n control points and paramtertized by a scalar u (e.g. f_1(u),f_2(u),...f_m(u)), we can define a Bezier *surface* as the locus of points reached by a 'sliding' Bezier curve g_u(v) parametrized by v that is defined through the control points p_1 = f_1(u), p_2 = f_2(u), ...,p_m = f_m(u) for a fixed u. Now letting u vary, we have a function of 2 coordinates, g_u(v) \longrightarrow g_(u,v)g(uv). To evaluate a point g(u,v) on such surfaces using the De Casteljau Algorithm, we define a function that takes in an mxn 2D vector of control points and the parametric coordinates (u,v). We first execute the 1D De Casteljau algorithm described in Part 1 along the rows of the 2D vector, which results in a set of m Bezier curves evaluated at u. Then using these resulting m points, we evaluate the 'sliding' Bezier curve at v with the De Casteljau Algorithm. The image below demonstrates how complex geometries (hard edges, beveled surfaces, discontinuities, etc.) can be created with a patchwork of bezier surfaces.

A teapot described using Bezier Surfaces.

Section II: Sampling

Part 3: Average normals for half-edge meshes

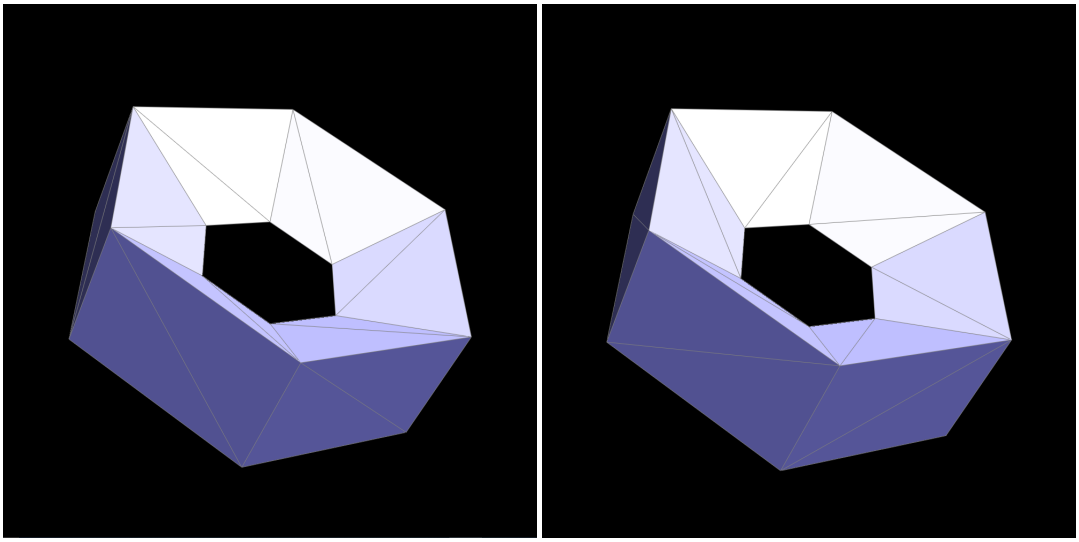
My Implementation of vertex normals made use of the halfedge data stucture and some properties of vector cross-products. Given a vertex in the mesh, I traversed the faces of the mesh surrounding this vertex, accessed the normal vectors associated with each face, and weighted these normals by the area of their corresponding face. To traverse the mesh apporpriately and access the surrounding faces, I began by retrieving the halfedge associated with the target vertex. Since each halfedge object contains a pointer to the face it is contained in, I simply had to reach every outgoing halfedge leaving the target vertex. If at an outgoing halfedge, the next outgoing halfedge can be reached with a call to twin() followed by a call to next(). In the halfedge mesh representation, the normal vectors are members of the face elements and are thus immediately accessible. However, determining the area of the face (and by extension the appropriate weighting factor for the face normal) was non-trivial. I made use of the fact that magnitude of the vector produced by the cross-product between two vectors is equal to twice the area of the triangle defined by the two argument vectors. Since each face in the mesh is triangular, I was able to determine its normal weight from this geometric relation by creating two temporary 3D vectors from the three vertices pertaining to the face and takin their cross-product. Coincidentally, it is also an interesting geometric fact that the cross-product operation itself produces a vector normal to the plane contained by the argument vectors. Hence, it was not necessary to retrieve the normal from the face element. Finally, to generate the vertex normal I performed a weighted sum of the normals of each face where the weights corresponded to the fractional area of the face. Below is side-by-side comparison of the shading benefits garnered using vertex normals.



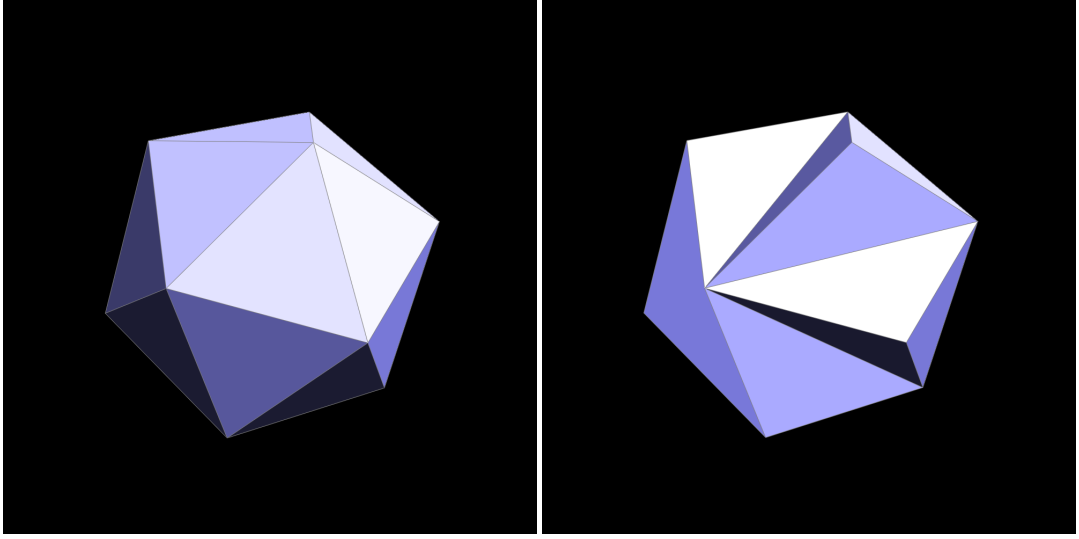
Flat shading using face normals. Phong shading using vertex normals.

Part 4: Half-edge flip

Implementing the edge-flip operation on the mesh involved careful pointer manipulation. With pen and paper, I began by tabulating the reassignments to every instance variable of every element involved in the edge-flip. This pen-and-paper starting point provided a visual reference from which the appropriate pointer manipulations were clear. After including a check that the target edge was not a boundary edge, I transcribed created temporary variables for every relevant mesh element using halfedge mesh traversal function calls. With these variables in hand, I executed the proper reassignments tabulated previously. Fortunately, I performed these steps meticulously and did not run into any bugs. Observing the example images below, the edge-flip operation by definition alters the topology of any mesh. However, it is interesting to see how in certain cases the edge-flip operation also alters the mesh geometry. In particular, if an edge-flip is performed on an edge that connects two faces that are do not reside in the same plane, the geometry of the mesh will change.



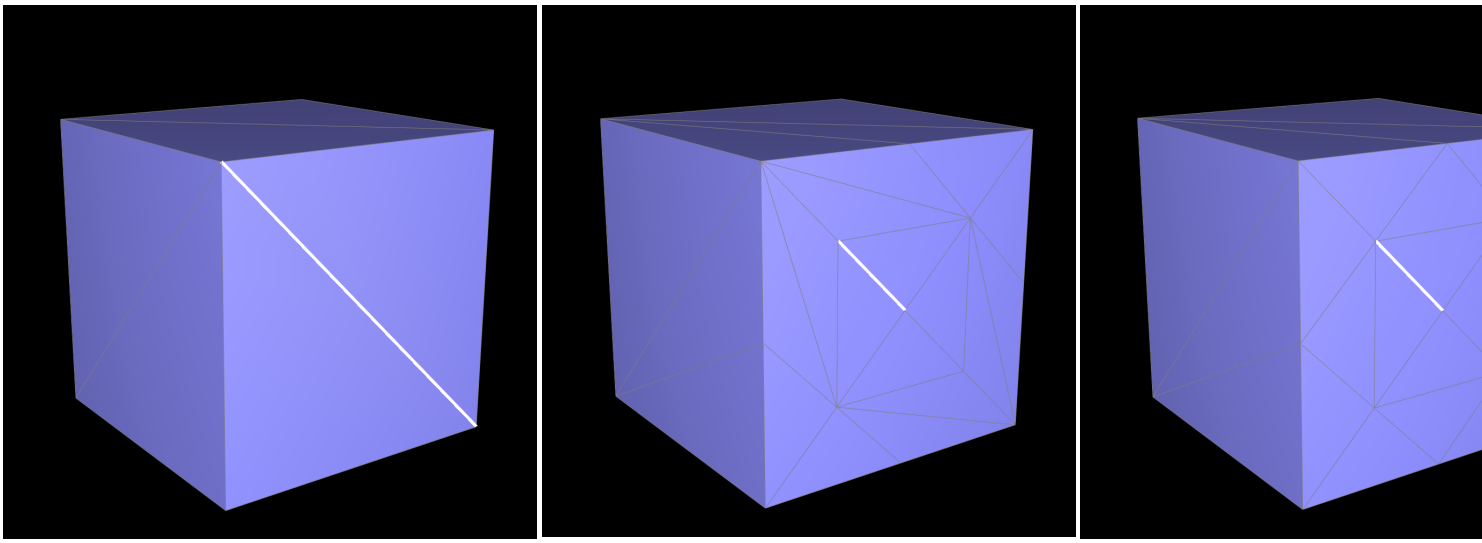
A torus. A torus after performing edge flips on in-plane diagonals.



An icosahedron. An icosahedron after performing four edge flips.

Part 5: Half-edge split

To implement the edge split operation, I took a similar approach as that described for the edge-flip operation. However, this operation requires appending 1 new vertex, 3 new edge elements, and 6 new halfedges to the mesh. After tabulating the pointer reassignments for each mesh element, I also updated the position for the new vertex (placing it at the midpoint of the target edge), and set the isNew field as true for all newly created edges and vertices (a mistake which led to extensive debugging time in my implementation of Loop subdivision).



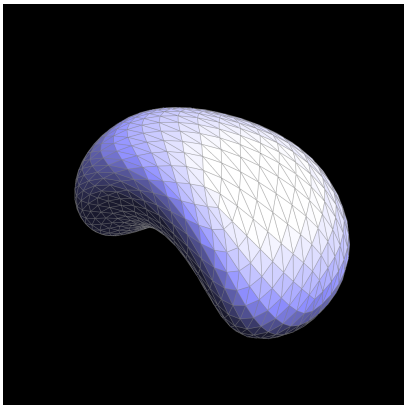
Cube face. Cube face after several splits. Split cube face after several fli

Part 6: Loop subdivision for mesh upsampling

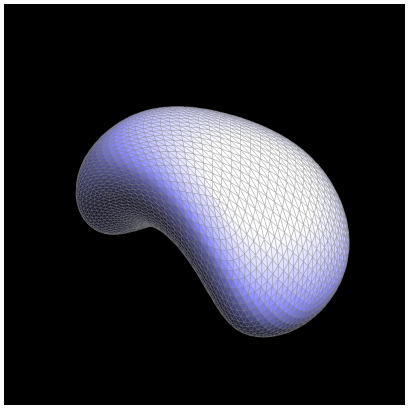
As recommended in the project spec, I began by preprocessing the weighted vertex locations for both the old and new vertices in the target mesh. For the old vertices, I stored the new locations in the newPosition field of the vertices themselves. However, storing the locations of the new vertices was a bit nuanced as they had not yet been created. Ultimately, I stored them in the newPosition field of the old mesh edges. In this way they would still be accessible after edge-splits and would be incident to the new vertices. Subsequently, I looped through all the old edges of the mesh and divided them using the edge-split operation. After toiling for some time with infinte loops caused by appending new edges to the mesh, I finally got the proper conditions set for splitting an edge (i.e. asserting that none of the vertices at the end of the edge were new). It then took me some time to convince myself that 4-1 triangle subdivision could indeed be achieved by carefully selecting which edges to flip after splitting every edge in the mesh. Once understanding the conditions for flipping an edge, I was able to update my edge-split subroutine to only set the isNew field to true for the new edges that bisected the original edge. Finally, I repositioned each vertex using the preprocessed locations identified at the beginning.

As loop supdivision entails repositioning vertices to make a more uniformly sampled mesh, sharper edges become rounded. We can think of the sharp edges as high frequency 3D content in the mesh. By enforcing mesh uniformity (i.e. making the spacing between vertices roughly consistent) we are effectively imposing a Nyquist sampling frequency. Thus loop subdivision bevels sharp edges and tends to produce the greatest quadric error after the first iteration. The images of the cube below show this effect. Moreover, comparing the second and third rows of the image sequence below, we see upsampling a cube using loop subdivision leads to two different results - One asymmetric and the other symmetric. This distinction can be attributed to the starting topology of the cube for either case. In row 2, the case where the cube geometry converges to an assymetric convex body, there are merely two triangles per face of the cube and two rotationally symmetric configurations over each cartesian axis. Contrastingly, in row 3, the case where the cube geometry converges to a more symmetric convex body, there are four triangles per cube face and four rotationally symmetric configurations over each cartesian axis. As discussed previously, a single iteration of loop subdivision produces serious quadric error

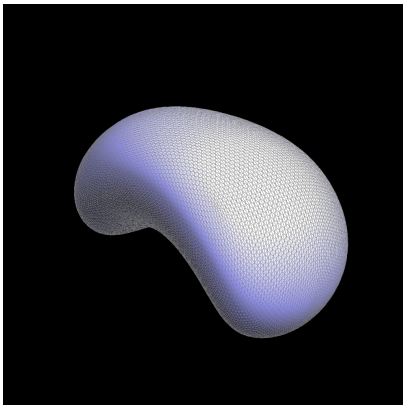
for shapes that are undersampled as upsampling the mesh via loop subdivion effectively high-pass filters the mesh. Thus since the third row cube topology contains more samples than the second row cube topology, and since all the faces of the third row cube are identical, upsampling converges to an object that is more cube-like.



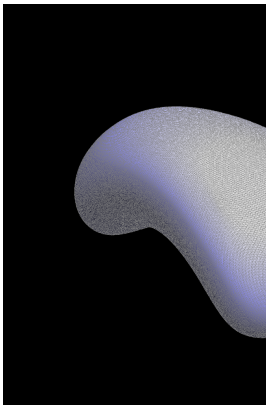
Bean - 1x Upsample.



Bean - 4x Upsample.

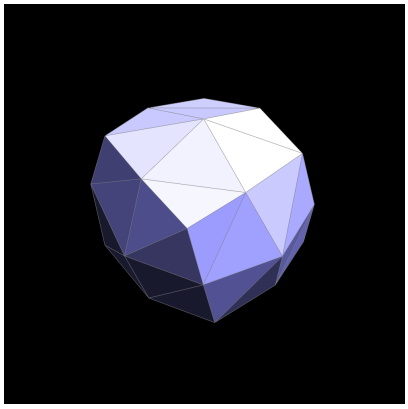


Bean - 16x Upsample.

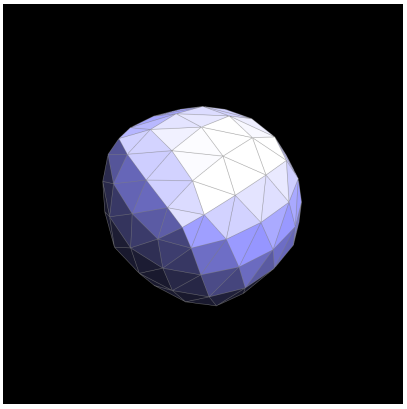


Bean - 64x Upsan

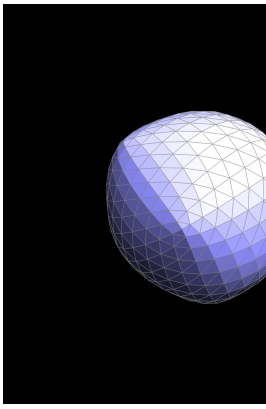
Asymmetric cube topology - 1x Upsample.



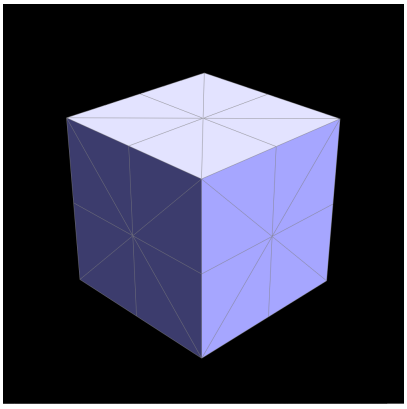
Asymmetric cube topology - 4x Upsample.



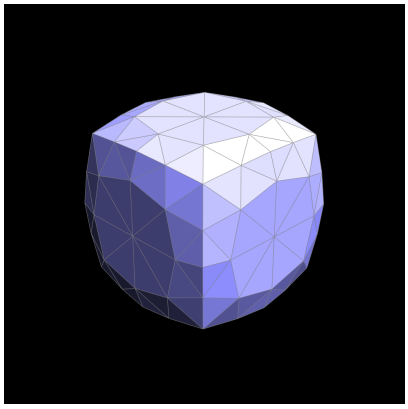
Asymmetric cube topology - 16x Upsample.



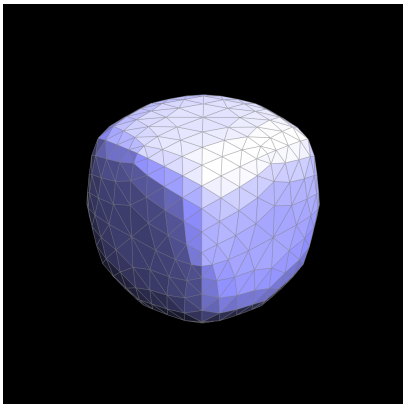
Asymmetric cube topo Upsample.



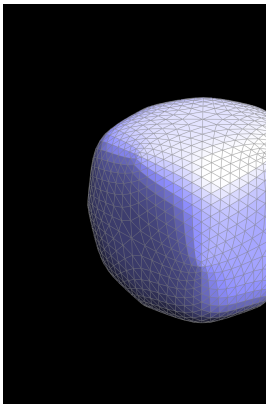
Symmetric cube topology - 1x Upsample.



Symmetric cube topology - 4x Upsample.



Symmetric cube topology - 16x Upsample.



Symmetric cube topol Upsample.