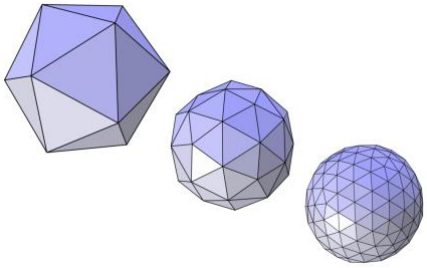
Subdivision Surfaces (Loop Subdivision) Final Project EXPECTATIONS, GUIDE, AND HINTS by Dylan Rowe

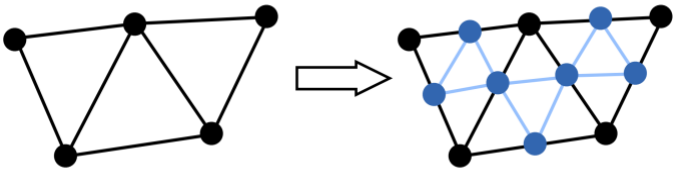
In this choice for the final project, you'll be implementing *Loop subdivision*, a surface subdivision method named after its creator, Charles Loop. While you can choose to implement it in the way that makes the most sense to you, there are some hints and ideas below that will likely help with some common confusions/complications along the way.



The Basics

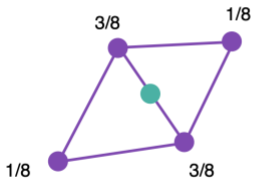
To understand the basic idea behind Loop subdivision, we recommend looking at the slides on surfaces; however, we include a brief summary here.

In Loop subdivision, we desire to subdivide a triangle mesh in a way that changes both the *topology* (connectivity) and *geometry* of the mesh, causing it to appear smoother. Topologically, each triangle is split into 4 subtriangles as below:

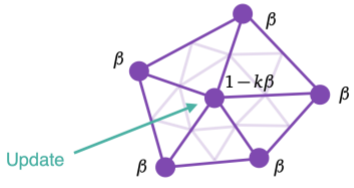


Geometrically, we assign new weights to both the new vertices (which are created along old edges), and the old vertices. These weights may not be directly intuitive, and the mathematics for deriving them are just beyond the scope of this class, but rest assured that they are based in reason:

- The new points have position given by weighted average of the neighboring vertices.



- Then, the old points position are updated by the following weighted average of the (old) neighboring points



- k = valence (in this picture k=5)
- $\beta = \begin{cases} 3/(8k) & k > 3 \\ 3/16 & k = 3 \end{cases}$

This is the basic idea behind Loop subdivision: we create new vertices, faces, and edges, and then assign new weights to every vertex using some update rule. In your final project report, you should show images of both a simple icosahedron mesh at 0, 1, and 2 iterations of Loop subdivision, and some other triangle mesh of your choice at 0, 1, and 2 iterations of Loop subdivision. Below, we give some hints that will help you implement this in a cleaner way.

Bonus/Extra Credit

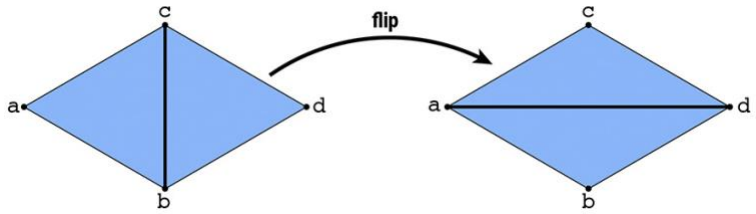
While bonus points/extra credit will generally be assigned for excellent exposition and demonstration as in the other projects, one idea to increase your chance for bonus points is to adapt your viewer to have a nice user interface, for example an interface that allows a user to split/flip edges by clicking them (this will likely help you debug your implementation as well, but isn't necessary). Another idea is to implement another standard type of subdivision beyond Loop subdivision, e.g. Catmull-Clark subdivision.

Hints:

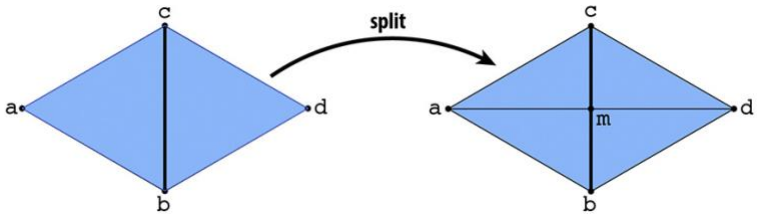
- We recommend beginning from your code submitted in Homework 2. While this code doesn't have lighting, it gives a simple mesh viewing framework that will allow you to examine and display meshes you operate on.
- One idea is to load the base mesh from a file (e.g. .obj), subdivide it a given number of times, and then insert the final geometry data into the geometry spreadsheet before sending it to be rendered by OpenGL.
- The **Halfedge mesh** is a data structure that is crucial to playing with mesh topology and geometry. It reduces traversing meshes and describing local operations on their elements to simple pointer following. A description of the basic halfedge mesh data structure is provided in the slides on surfaces mentioned above, but below we also recommend some extensions of this structure that make Loop subdivision easier.
- For your project, you'll likely want to write code which parses .obj files and converts them to your halfedge mesh data structure. This stackoverflow post says more about how one can construct edges and halfedges from just a description of faces and vertices.
- For our course mesh viewer, normals are typically assigned to vertices, rather than faces. However, **arbitrary meshes may not have vertex normal vectors assigned at all**. One simple idea to assign normals to vertices (area-weighted average) is to first get the normal vector of each face using the cross product, and then assign vertex normals using a convex combination where the weights of the combination are based on the corresponding face's area.
    - Explicitly, if a vertex's incident faces f_i have unit normals n_i and areas a_i, the vertex should have normal vector $normalise(\sum_i \frac{a_i}{\sum_j a_j} n_i)$. Use the halfedge data structure to get the normals, areas, and incident faces.
- In general, the properties of a 3D mesh are usually described as **topology** or **geometry**. The topology of a mesh is determined entirely by connectivity, i.e. face/edge/vertex incidence relations. The geometry of a mesh is determined by the locations of its

vertices. Notice that one can change the geometry of a mesh without changing the topology; this is key to understanding the following algorithm.

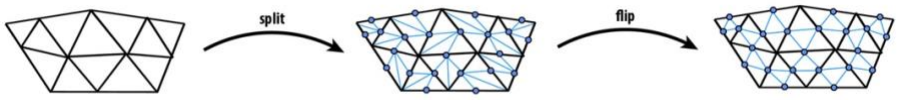
- It will likely be helpful to calculate the positions of the new and old vertices **before you update the topology of the mesh**. In general, this will allow you to use the old mesh topology to calculate the necessary quantities before your mesh gets more complicated to traverse. Thus, the order you'd perform the updates after following this hint is: (1) Calculate new vertex positions without actually updating positions in the mesh (2) Update the topology of the mesh (3) Assign each vertex its new position which was calculated in step 1.
- We recommend implementing the following two mesh operations (images courtesy of CS 184/284 at Berkeley), as they will make it much easier to create the necessary mesh. When designing these, it is best to draw out the operation on paper, giving each Point, Edge, Face, and Halfedge its own variable name. Then, note how **every** pointer changes, and implement this in code. It is very difficult to debug this kind of thing, so please be careful, and if something is wrong after implementation, return to your drawings and make sure **every** pointer is correct.
    - **Flip**, which takes in an edge pointer and "flips" it in the mesh:



    - and **Split**, which takes an edge pointer and "splits" it in the mesh:



- After implementing these two operations, we can create the new topology quickly, by **splitting every edge**, and then **flipping every <u>new</u> edge that connects a new and old vertex**:



Naively iterating through the edges in the mesh will likely lead to ruin here, since you will run into an infinite loop if the edges you generate from splitting are also inserted into the std::vector you're iterating over, so you will likely need to think of a better way to do this.

- In addition to Points, Faces, and Halfedges, it may be convenient to hold another class of object called **Edges**. Each Edge holds a pointer to one of its Halfedges, and each Halfedge holds a pointer to its parent Edge; this will allow you to manage the new points along old edges in a simpler and more compact way. The mesh class will also hold an std::vector containing all edges in the mesh.
- In addition to adding the Edge class, it may be beneficial to give both **is_new** (which says if an element is new or old) and **new_pos** (which gives the new position of either the old vertex or new vertex along an edge) variables to both Edges and Points; this way, you can keep track of whether a Point is new or old, as well as hold the new position of a vertex you're inserting on some edge.
- Debugging the flip and split operations is **very difficult**, and can be a true test of patience. We highly recommend drawing a diagram of the two triangles before and after the operation, labelling the faces, edges, and vertices, and explicitly assigning pointers for every single object in play.
- One practical idea to show a wireframe of the edges in your mesh for debugging purposes is to use the line

  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

  before the .draw() call in your OpenGL code, and the line

  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

  after (the second command simple returns the glPolygonMode to the default, in case you want to switch back and forth between the modes).
- If you wish to insert new .obj files into your program, be sure that they are **triangle meshes without boundary edges**. If you do not deal with boundary edges in your implementation, edge flips and splits will likely lead to errors; additionally, if there are quadrilaterals in your mesh, Loop subdivision will not know how to deal with them. There are many different ways to convert general polygon meshes into triangle meshes using standard free mesh editing programs like Blender; we recommend looking for tutorials online.