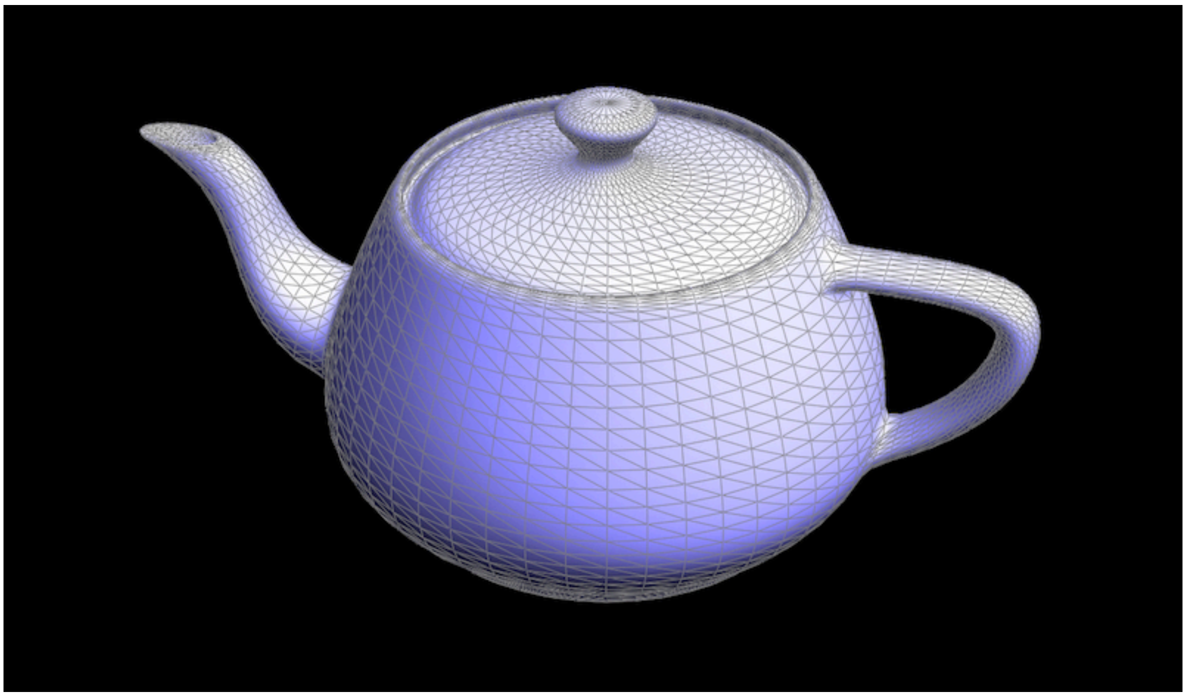
# ASSIGNMENT 4
# MeshEdit



In this assignment you will explore a subset of the geometric topics covered in lecture. You will build Bezier curves and surfaces using de Casteljau's algorithm, manipulate half-edge meshes, implement Loop subdivision, and write shaders for your own meshes! When you are finished, you will have a tool that allows you to load and edit basic COLLADA mesh files that are now used by many major modeling packages and real time graphics engines

## Logistics

### Deadline

The deadline for the entire assignment is Monday, November 18th at 11:55pm.

### Before you get started...

As you go through the assignment, refer to the write-up guidelines and deliverables section at the bottom of this page. It is recommended that you accumulate deliverables into sections in your webpage write-up as you work through the project. Please consult this article on how to build the assignment.

### Getting started

You can clone the assignment from GitLab using the command

A debugging note: Check out this page on how to set up *gdb* to work with recent OS X releases. Please use *gdb* to debug, particularly if you get a seg fault that you can't track! Turn on debug symbols by toggling this line at the top of the *CMakeLists.txt* file in the root directory:

```
option(BUILD_DEBUG     "Build with debug settings"    ON)
```

You'll have to rerun *cmake* to update the Makefiles after doing this.

## Project structure

The project has 5 parts, divided into 3 sections, worth a total of 100 possible points. Some require only a few lines of code, while others are more substantial.

**Section I: Bezier Curves and Surfaces**

- Part 1: Bezier curves with 1D de Casteljau subdivision
- Part 2: Bezier surfaces with separable 1D de Casteljau subdivision (15 pts)

**Section II: Loop Subdivision of General Triangle Meshes**

- Part 3: Average normals for half-edge meshes
- Part 4: Loop subdivision for mesh upsampling

**Section III: Shaders (Bonus!!)**

- Part 5: Fun with shaders

## Using the GUI

When you have successfully built your code, you will get an executable named `meshedit` in the build directory. The `meshedit` executable takes exactly one argument from the command line. You may load a single COLLADA file by specifying its path. For example, to load the example file *dae/quadball.dae* from your build directory:

```
./meshedit ../dae/quadball.dae
```

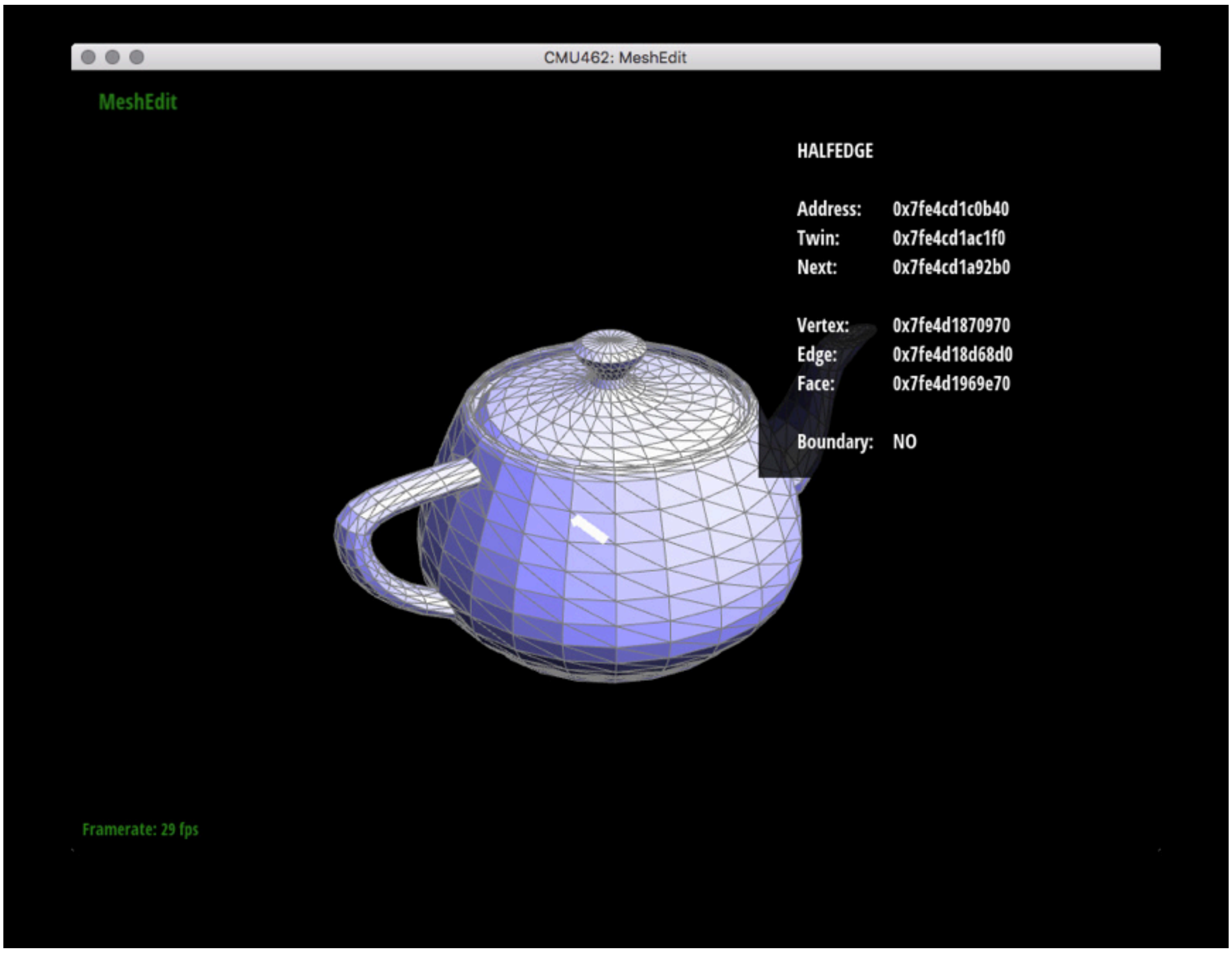
In Section I Part I, you will be able to load Bezier curves by running a command such as:

```
./meshedit ../bzc/curve1.bzc
```

In Section I Part II, you will be able to load Bezier surfaces by running a command such as:

```
./meshedit ../bez/teapot.bez
```

When you first run the application, you will see a picture of a mesh made of triangles. The starter code that you must modify is drawing this mesh. The editor already supports some basic functionality, like moving vertices around in space, which you can do by just clicking and dragging on a vertex. You can also rotate the camera by right-clicking and dragging (or dragging on the background), and zoom in and out using the scroll wheel or multi-touch scrolling on a trackpad. Hitting the spacebar will reset the view. As you move the cursor around the screen, you'll notice that mesh elements (faces, edges, and vertices) under the cursor get highlighted. Clicking on one of these elements will display some information about the element and its associated data.



HALFEDGE

Address: 0x7fe4cd1c0b40
Twin: 0x7fe4cd1ac1f0
Next: 0x7fe4cd1a92b0

Vertex: 0x7fe4d1870970
Edge: 0x7fe4d18d68d0
Face: 0x7fe4d1969e70

Boundary: NO

In this assignment, you will add additional functionality to the program that allows you to modify the mesh in a variety of ways. Each of these methods is exposed through the viewer. There are two basic types of operations. Each operation will be executed with a key press.

1. Local flip (**F**) and split (**S**) operations, which modify the mesh in a small neighborhood around the currently selected mesh element.
2. Loop subdivision (**U**), which refines and smooths the entire mesh.

Here is the full specification on keyboard controls for the GUI:

| Command | Key |
|---|---|
| Flip the selected edge | F |
| Split the selected edge | S |
| Upsample the current mesh | U |
| Toggle information overlay | I |
| Toggle center points | P |

| | |
|---|---|
| Select the next halfedge | `N` |
| Select the twin halfedge | `T` |
| Switch to GLSL shaders | `W` |
| Switch between GLSL shaders | `0-9` |
| Toggle using area-averaged normals | `Q` |
| Recompile shaders | `R` |
| Reset camera to default position | `SPACE` |
| Edit a vertex position | (click and drag on vertex) |
| Rotate camera | (click and drag on background, or right click) |

Note that each COLLADA file may contain multiple mesh objects; more generally, a COLLADA file describes a **scene graph** (much like SVG) that is a hierarchical representation of all objects in the scene (meshes, cameras, lights, etc.), as well as their coordinate transformations. Global resampling methods will be run on whichever mesh is currently selected.

## Getting Acquainted with the Starter Code

Before you start, here is some basic information on the structure of the starter code. **Your code for all parts except shading will be contained inside *student_code.cpp***.

For Bezier curves and surfaces (Section I), you'll be filling in member functions of the `BezierCurve` and `BezierPatch` classes, declared in *bezierCurve.** and *bezierPatch.**. We have put dummy definitions for all the `BezierCurve` and `BezierPatch` functions you need to implement inside *student_code.cpp*, where you'll be writing your code.

For half-edge meshes (Section II), you'll be filling in member functions of the `HalfedgeMesh` class, declared in *halfEdgeMesh.**. We have put dummy definitions for all the half-edge functions you need to modify inside *student_code.cpp*, where you'll be writing your code.

For shaders (Section III), you'll be editing the *shader/frag* file.

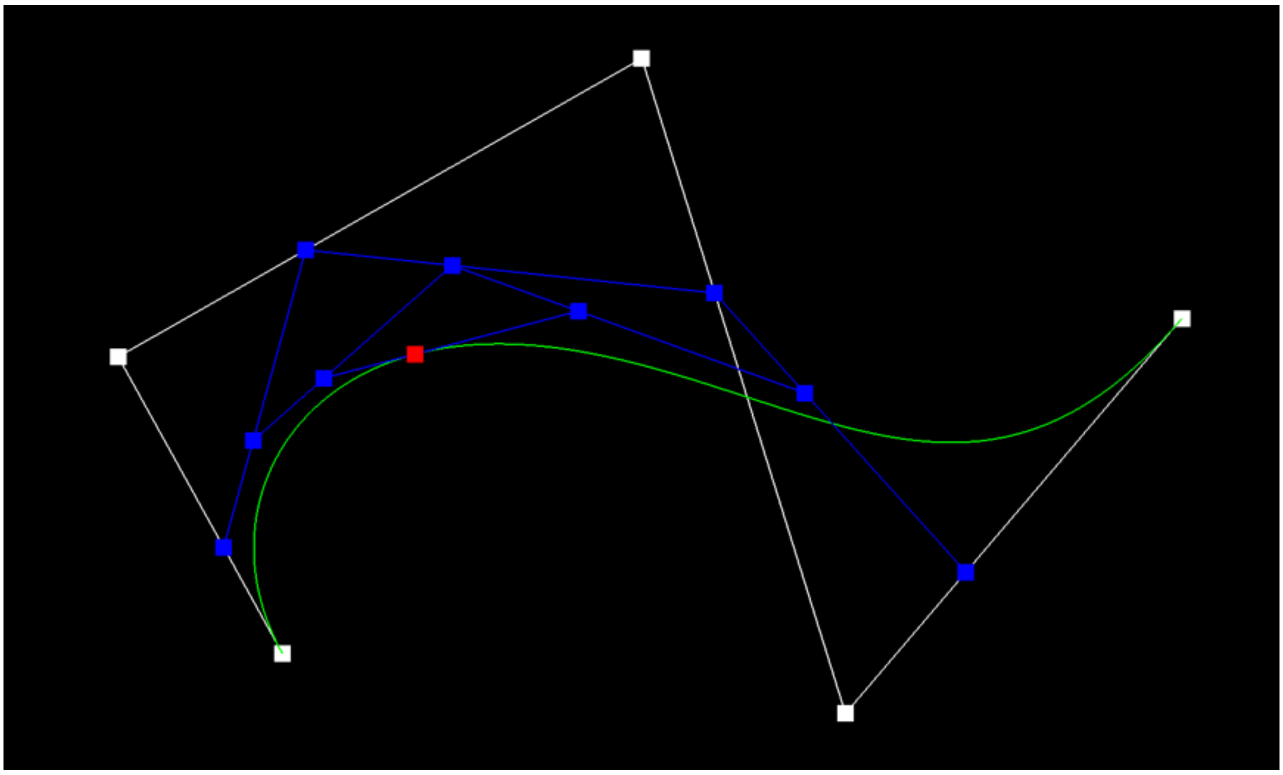
Before you attempt Section II, you'll want to consult lectures 10 and 11 as a half-edge refresher and then also read this in-depth article with more detail about the half-edge data structure and its implementation in the starter code.

## Section I: Bezier Curves and Surfaces

In Section I, we take a closer look at Bezier curves and surfaces. In computer graphics, Bezier curves and surfaces are parametric curves and surfaces that are frequently used to model smooth and indefinitely scalable curves and surfaces.

A Bezier curve of degree n is defined by (n + 1) control points. Similarly, a Bezier surface of degree (n, m) is defined by (n + 1)(m + 1) control points.

A Bezier curve is a parametric curve defined by a single parameter, t, which ranges between 0 and 1. Similarly, a Bezier surface is a parametric surface defined by two parameters, u and v. Using de Casteljau's algorithm, we can evaluate these parametric curves and surfaces for any given set of parameters.



## Part 1: Bezier curves with 1D de Casteljau subdivision

**Relevant lectures: 9**

In Part 1, we will work with generic Bezier curves, though we will look at cubic Bezier curves as our motivating example and extend the concept from there to support any order Bezier curve.

To get started, take a look at *bezierCurve.h* and examine the protected variables defined within the class. Let's briefly go over the purpose of ones you will primarily be concerned with:

- `std::vector<Vector2D> controlPoints` : A vector containing exactly control points that define the Bezier curve. This vector is initialized with the control points of the Bezier curve file passed in.
- `std::vector< std::vector<Vector2D> > evaluatedLevels` : A 2D vector containing the evaluated points at each level of subdivision when applying de Casteljau's algorithm. Initially, it contains a single vector representing the starting level containing just the original control points. You should add additional levels with their respective "intermediate control points" every time `evaluateStep` is called. For example, a cubic Bezier curve should have 4 total levels, where the first level contains the original control points and the final level contains just a single evaluated point that represents $B(t)$ (the Bezier curve evaluated at t).
- `float t` : A parameter varying between 0.0 and 1.0 at which to evaluate the Bezier curve.

Implement `evaluateStep` , which looks at the control points from the most recently evaluated leveland performs de Casteljau's algorithm to compute the next level of intermediate control points. These intermediate control points at each level should be appropriately stored into the member variable `evaluatedLevels` , which is used by the program to render the Bezier curve.

### Implementation Notes

- `evaluatedLevels` is seeded with the original control points of the Bezier curve, so you should already have a "most recent level" to look at and use to compute your first set of intermediate control points.
- **DO NOT** use or modify the variable `eval_level` defined in `bezierCurve.h` . This is used internally for rendering the curve.
- `std::vector` is similar to Java's `ArrayList` class; you should use `std::vector` 's `push_back` method to add elements, which is analogous to Java's `ArrayList` 's `append` method.

Recall from lecture that de Casteljau's algorithm gives us the following recursive step that we can repeatedly apply to evaluate a Bezier curve:

`evaluateStep` should return immediately if the Bezier curve has already been completely evaluated at t (i.e. `evaluateStep` has already been called enough times to completely solve for the point at B(t)).

Check your implementation by running the program with this syntax: `./meshedit <path to .bzc file>` :

```
./meshedit ../bzc/curve1.bzc
```

*bzc/curve1.bzc* is a cubic Bezier curve, whereas *bzc/curve2.bzc* is a degree-4 Bezier curve. Feel free to play around with higher-order Bezier curves by creating your own *bzc* files.

**Using the GUI: Part 1**

For this Part only, the GUI is a little different. There are two keyboard commands:

- **E**: Perform one call to `evaluateStep` ; will cycle through the levels once fully evaluated
- **C**: Toggles whether or not the entirely evaluated Bezier curve is drawn to the screen

Step through the evaluation of $B(t)$ by repeatedly pressing **E** to verify your implementation is correct. Toggle the Bezier curve using **C** to check that the curve correctly follows from its control points.

Besides these keyboard commands, there are some neat controls with your mouse:

- **Click and drag** the control points to move them and see how your Bezier curve (and all intermediate control points) changes accordingly
- **Scroll** to move the evaluated point along the Bezier curve and see how the intermediate control points move along with it; this is essentially varying $t$ between 0.0 and 1.0

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

Relevant lectures: 9

In Part 2, we will work only with cubic Bezier surfaces.

To get started, take a look at *bezierPatch.h* and examine the class definition. In this part, you will be working with:

- `std::vector< std::vector<Vector3D> > controlPoints` : A 2D vector representing a 4x4 grid of control points that define the cubic Bezier surface. This variable is initialized with all 16 control points.
- `Vector3D evaluate(double u, double v) const` : You will fill this function in, which evaluates the Bezier curve at parameters (u, v). In mathematical terms, it computes B(u, v).
- `Vector3D evaluate1D(std::vector<Vector3D> points, double t) const` : An optional helper function that you might find useful to implement to help you with your implementation of `evaluate` . Given an array of 4 points that lie on a single curve, evaluates the curve at parameter t using 1D de Casteljau subdivision.

Implement `evaluate` , which completely evaluates the Bezier surface at parameters u and v. Unlike Part 1, you will not perform just a single step at a time -- you will instead completely evaluate the Bezier surface using the specified parameters. This function should return that final evaluated point.

Use the following algorithm to repeatedly apply separable 1D de Casteljau's algorithm in both dimensions to evaluate the Bezier surface:

```
For each row i:
  Let q(i, u) := apply de Casteljau's algorithm with parameter u to the i-th row of control points

Let p(u, v) := apply de Casteljau's algorithm with parameter v to all q(i, u)
Return p(u, v)
```

If your implementation is correct, you should see a teapot by running the following command:

```
./meshedit ../bez/teapot.bez
```

## Section II: Loop Subdivision of General Triangle Meshes

In Section I, we dealt with Bezier curves and surfaces, parametric functions that were defined by a set of control points. Through de Casteljau's algorithm, we performed subdivision that allowed us to evaluate those functions.

With Bezier curves, we performed 1D subdivision, and evaluation steps were relatively simple since adjacent control points were trivially retrieved from a 1D vector. With Bezier surfaces, we applied the same 1D subdivision concepts in both dimensions, and evaluation steps were a little more complicated, but adjacent control points were still easily retrieved from the 2D grid of control points.

What about generic triangle meshes? Here, the only rules are that our mesh is made up of triangles and that the triangles connect to each other through their edges and vertices. There is no constraining 2D grid as before with Bezier surfaces. As a result, adjacency is unfortunately no longer trivial. Enter the half-edge data structure, a powerful and popular data structure commonly used to store mesh entities and their connectivity information.

**Note:** Before diving into Section II, be sure to first read this article to help you navigate the `HalfedgeMesh` class, which you will use extensively in the next section of the project.

### Part 3: Average normals for half-edge meshes

**Relevant lecture: 7**

For Part 3, make sure you understand the code given for a `printNeighborPositions` function as well.

In this part, you will implement the `Vertex::normal` function inside *student_code.cpp*. This function returns the area-weighted average normal vector at a vertex, which can then be used for more realistic local shading compared to the default flat shading technique. This slide depicts this graphically.

In order to compute this value, you will want to use a `HalfedgeIter` to point to the `Halfedge` you are currently keeping track of. A `HalfedgeIter` (analogously `VertexIter` , `EdgeIter` , and `FaceIter` ) is essentially a pointer to a `Halfedge` (respectively `Vertex` , `Edge` , and `Face` ), in the sense that you will use `->` to dereference its member functions. Also, you can test whether two different iterators point to the same object using `==` , and you can assign one iterator to point to the same thing as another using `=` (this will NOT make the pointed-to objects have the same value, just as with pointers!).

**Technical implementation caveat**: For this part only, you're implementing a `const` member function, which means you need to use `HalfedgeCIter` s instead of `HalfedgeIter` s. These merely promise not to change the values of the things they point to.

The relevant member functions for this task are `Vertex::halfedge()` , `Halfedge::next()` and `Halfedge::twin()` . You will also need the public member variable `Vector3D Vertex::position` .

How you might use these to begin implementing this function:

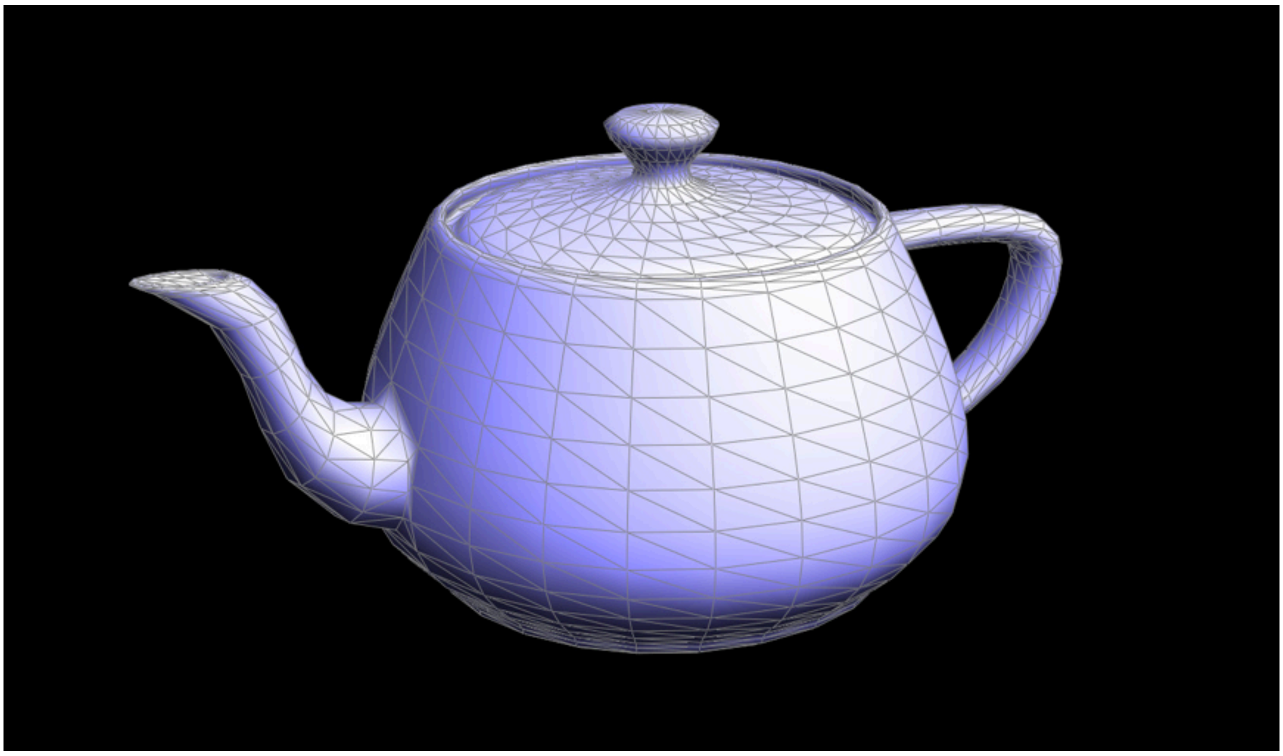
```
Vector3D n(0,0,0); // initialize a vector to store your normal sum
HalfedgeCIter h = halfedge(); // Since we're in a Vertex, this returns a halfedge
                              // pointing _away_ from that vertex
h = h->twin(); // Bump over to the halfedge pointing _toward_ the vertex.
               // Now h->next() will be another edge on the same face,
               // sharing the central vertex.
```

At this point, you should

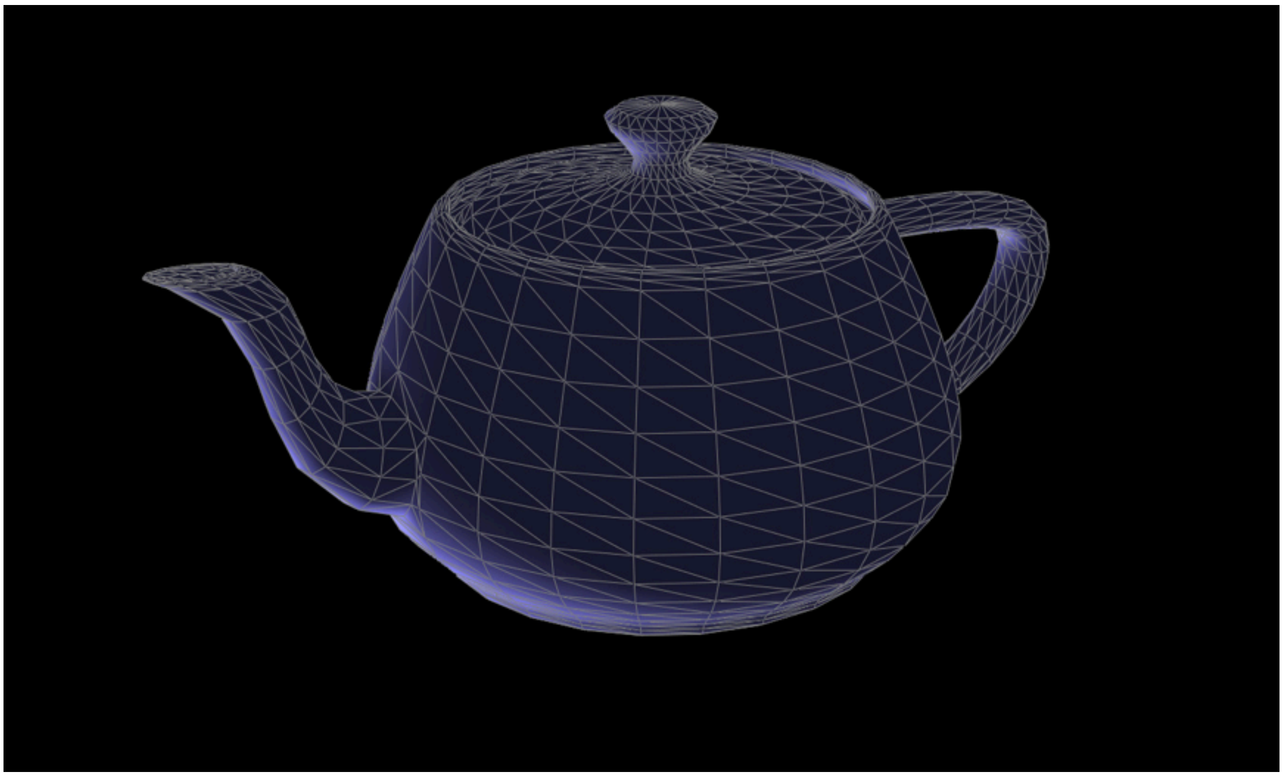
1. Save a copy of `h` 's value in another `HalfedgeCIter h_orig` .

2. Start a `while` loop that ends when `h == h_orig` .
3. Inside each loop iteration:
   - Accumulate area-weighted normal of the current face in the variable `n` . You can do this by using the cross product of triangle edges. We've defined the cross product for you, so don't re-implement it yourself! Since the cross product of two vectors has a norm equal to twice the area of the triangle they define, these vectors are *already area weighted*!
   - Once you've added in the area-weighted normal, you should advance `h` to the halfedge for the next face by using the `next()` and `twin()` functions.
4. After the loop concludes, return the re-normalized unit vector `n.unit()` .

After completing this part, load up a *dae* such as *dae/teapot.dae* and press **W** to switch to GLSL shaders and then press **Q** to toggle area-averaged normal vectors (which will call on your `Vertex::normal` function). Here's an example of what *dae/teapot.dae* should look like with correctly implemented area-averaged normals.



It should NOT look like this!



## Part 4: Loop subdivision for mesh upsampling

**Relevant lectures: 11**

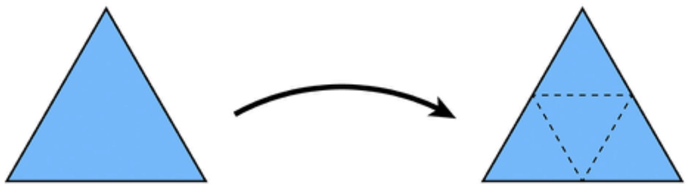
Now, we can leverage the previous two parts to make implementing the mesh topology changes in Loop subdivision very simple! In this task, you will implement the whole Loop subdivision process inside the `MeshResampler::upsample` in *student_code.cpp*.

Loop subdivision is somewhat analogous to upsampling using some interpolation method in image processing: we may have a low-resolution polygon mesh that we wish to upsample for display, simulation, etc. Simply splitting each polygon into smaller pieces doesn't help, because it does nothing to alleviate blocky silhouettes or chunky features. Instead, we need an upsampling scheme that nicely interpolates or approximates the original data. Polygon meshes are quite a bit trickier than images, however, since our sample points are generally at *irregular* locations, i.e., they are no longer found at regular intervals on a grid.

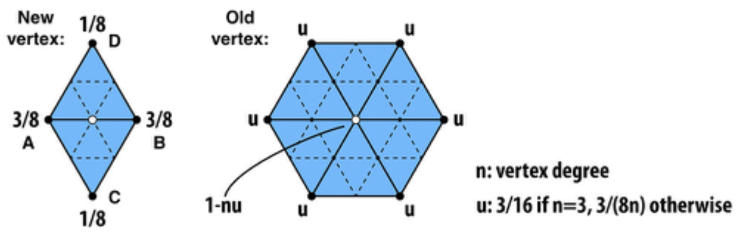
Loop subdivision consists of two basic steps:

1. Change the mesh topology: split each triangle into four by connecting edge midpoints (sometimes called "4-1 subdivision").
2. Update vertex positions as a weighted average of neighboring positions.

4-1 subdivision does this to each triangle:



And the following picture depicts the correct weighting for the new averaged vertex positions:



Written out, the new position of an old vertex is

```
(1 - n*u) * original_position + u * neighbor_position_sum
```

where `n` is the number of neighboring vertices, `u` is a constant as depicted in the figure above, `original_position` is the vertex's original position, and `neighbor_position_sum` is the sum of all neighboring vertices' positions.

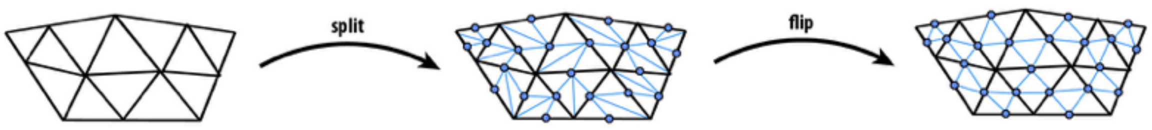
The position for a newly created vertex v that splits an edge AB connecting vertices A and B and is flanked by opposite vertices C and D across the two faces connected to AB in the original mesh will be

```
3/8 * (A + B) + 1/8 * (C + D)
```

If we repeatedly apply these two steps, we will converge to a smoothed approximation of our original mesh. In this task you will implement Loop subdivision, leveraging the split and flip operations to handle the topology changes. In particular, you can achieve a 4-1 subdivision by applying the following strategy:

1. Split every edge of the mesh in any order whatsoever.
2. Flip any new edge that touches a new vertex and an old vertex. *Note*: Every original edge will now be represented by 2 edges, you *should not* flip these edges, because they are always already along the boundary of the 4 way divided triangles. In the diagrams below, you should only flip the blue edges that connect an old and new vertex, but you should not flip any of the black new edges.

The following pictures (courtesy Denis Zorin) illustrate this idea:



#### Implementation walkthrough

For Loop subdivision, we have also provided some additional data members that will make it easy to keep track of the data you need to update the connectivity and vertex positions. In particular:

- `Vertex::newPosition` can be used as temporary storage for the new position (computed via the weighted average above). Note that you should *not* change the value of `Vertex::position` until *all* the new vertex positions have been computed -- otherwise, you are taking averages of values that have already been averaged!
- Likewise, `Edge::newPosition` can be used to store the position of the vertices that will ultimately be inserted at edge midpoints. Again, these values should be computed from the original values (before subdivision), and applied to the new vertices only at the very end. The `Edge::newPosition` value will be used for the position of the vertex that will appear along the old edge after the edge is split. We precompute the position of the new vertex before splitting the edges and allocating the new vertices because it is easier to traverse the simpler original mesh to find the positions for the weighted average that determines the positions of the new vertices.
- `Vertex::isNew` can be used to flag whether a vertex was part of the original mesh, or is a vertex newly inserted by subdivision (at an edge midpoint).
- `Edge::isNew` likewise flags whether an edge is a piece of an edge in the original mesh, or is an entirely new edge

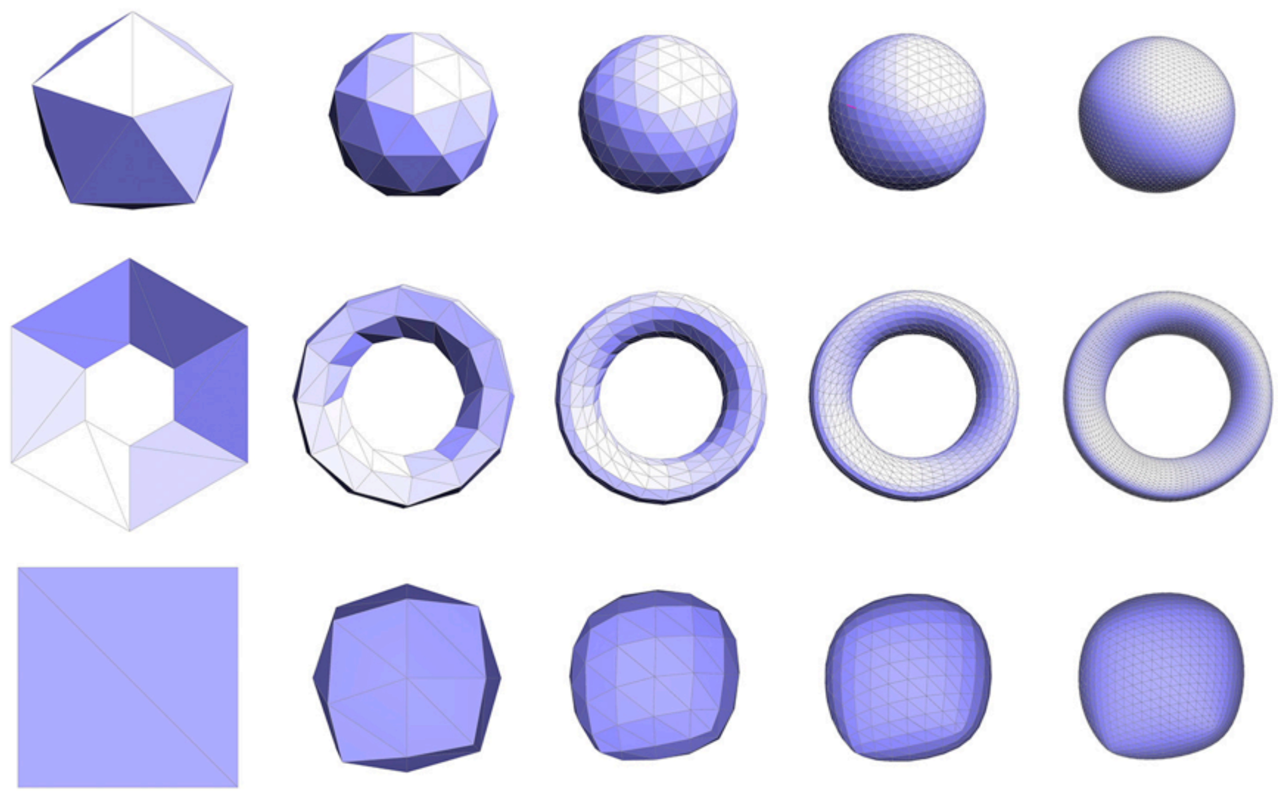
created during the subdivision step.

Given this setup, we strongly suggest that it will be easiest to implement subdivision according to the following "recipe" (though you are of course welcome to try doing things a different way!). The basic strategy is to *first* compute the new vertex positions (storing the results in the `newPosition` members of both vertices and edges), and only *then* update the connectivity. Doing it this way will be much easier, since traversal of the original (coarse) connectivity is much simpler than traversing the new (fine) connectivity. In more detail:

1. Mark all vertices as belonging to the original mesh by setting `Vertex::isNew` to `false` for all vertices in the mesh.
2. Compute updated positions for all vertices in the original mesh using the vertex subdivision rule, and store them in `Vertex::newPosition` .
3. Compute new positions associated with the vertices that will be inserted at edge midpoints, and store them in `Edge::newPosition` .
4. Split every edge in the mesh, being careful about how the loop is written. In particular, you should make sure to iterate only over edges of the original mesh. Otherwise, you will keep splitting edges that you just created!
5. Flip any new edge that connects an old and new vertex.
6. Finally, copy the new vertex positions ( `Vertex::newPosition` ) into the usual vertex positions ( `Vertex::position` ).

If you made the requested modification to the return value of `HalfedgeMesh::splitEdge()` (see above), then an edge split will now return an iterator to the newly inserted vertex, and the halfedge of this vertex will point along the edge of the original mesh. This iterator is useful because it can be used to (i) flag the vertex returned by the split operation as a new vertex, and (ii) flag each outgoing edge as either being new or part of the original mesh. (In other words, Step 3 is a great time to set the members `isNew` for vertices and edges created by the split. It is also a good time to copy the `newPosition` field from the edge being split into the `newPosition` field of the newly inserted vertex.)

You might try implementing this algorithm in stages, e.g., *first* see if you can correctly update the connectivity, *then* worry about getting the vertex positions right. Some examples below illustrate the correct behavior of the algorithm.



## Section III: Shaders (BONUS)

## Part 5: Fun with shaders

**Relevant lecture: 7**

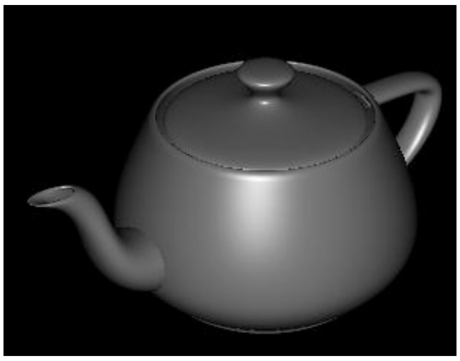
For this part, you will implement Phong shading. Take a look at these slides to review the basic concepts and variables used in shading equations.

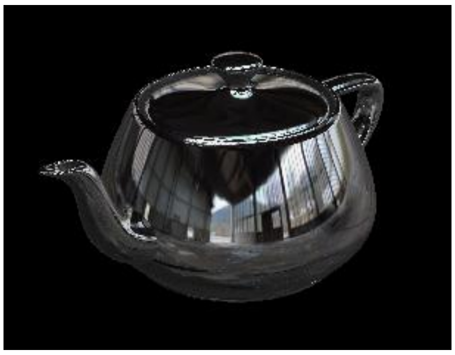
Here is a list of functions you will need to modify inside *shader/frag*:

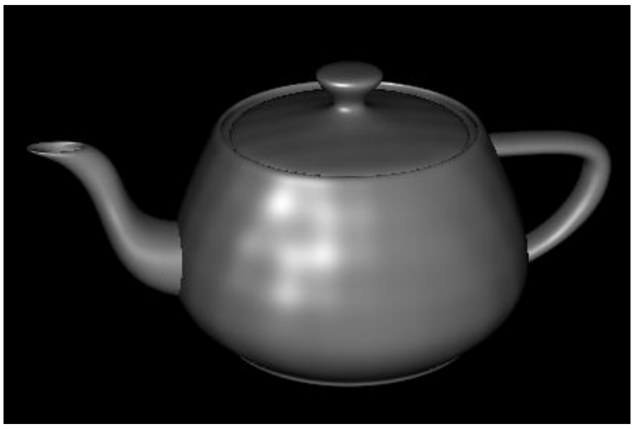
1. `shadePhong`

GLSL Tutorials: Here is a short tutorial on GLSL. It should have everything you need to know about this part of the assignment.

### Reference Images







## Submission

To be done on LMS