CSC4140 Assignment V

Computer Graphics

March 22, 2022

Geometry

This assignment is 9%(with 2 extra credit) of the total mark.

Strict Due Date: 11:59PM, April 02nd, 2022

Student ID:

Student Name:

This assignment represents my own work in accordance with University regulations.

Signature:

Overview

Congrats. In the last assignment, you have make a great achievement to realize a pipeline of rasterization. This time, you will dig deeply into the Geometry! Basically, this project has 2 parts and 6 small tasks, worth a total 110 points (20 extra points can be added if you completed a fancy additional work, but no more than 9% for this time). The tasks are as follows:

- **Part I Bezier Curves and Surfaces (25 points)**
  - Bezier curves with 1D de Casteljau subdivision (10 points)
  - Bezier surfaces with separable 1D de Casteljau (15 points)
- **Part II Bezier Curves and Surfaces (65 points)**
  - Area-weighted vertex normals (10 points)
  - Edge flip (15 points)
  - Edge split (15 points + 6 extra)
  - Loop subdivision for mesh upsampling (25 points + 7 +7 extra)

Running the Executable

The general command for running the executable *meshedit* is the following:

```
./meshedit <PATH_TO_FILE>
```

Note that *meshedit* expects different file formats for different parts of this assignment, as detailed below:

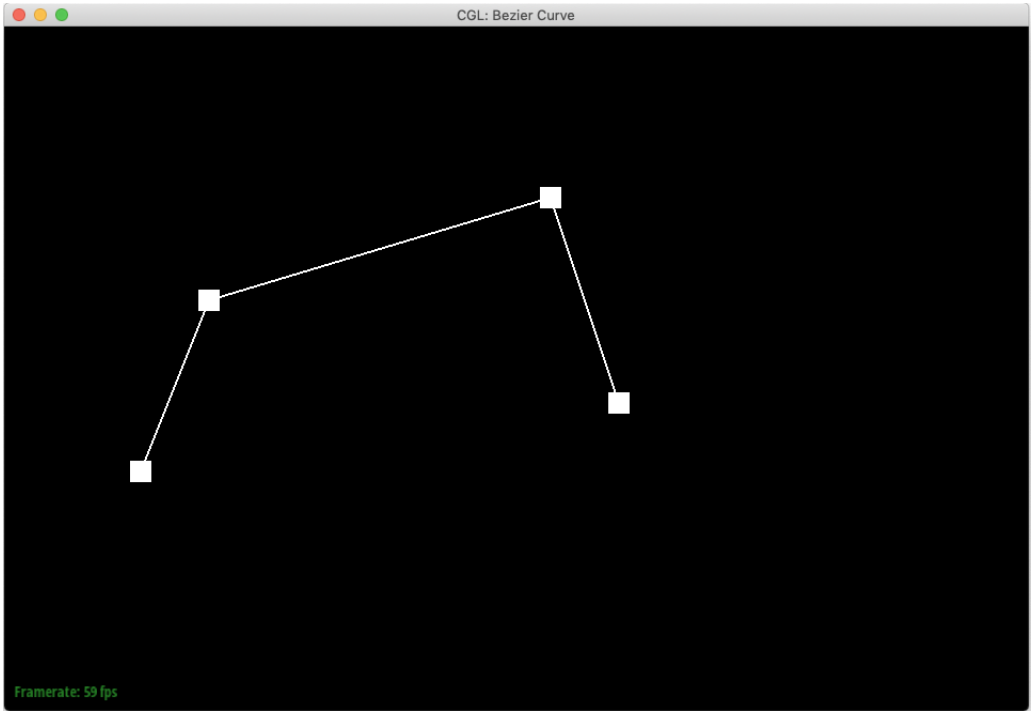
```
./meshedit ../bzc/curve1.bzc  // E.g., Part I
./meshedit ../*/*.bez  //  E.g., Part I-2
./meshedit ../*/*.dae // E.g., Part II & III
```

Using the Graphical User Interface (GUI)

The following details how to use the GUI provided by the executable *meshedit*. You may want to skim through them on your first read-through; and read them in more details when you have implemented parts of this assignment and are ready to use the GUI.

Part I - 1

When you run *meshedit* with a Bezier curve (.bzc) for Section I Part I, you will see control points rendered on screen, as shown below.



For this part only, the GUI is different. Below is the full specification on keyboard controls.

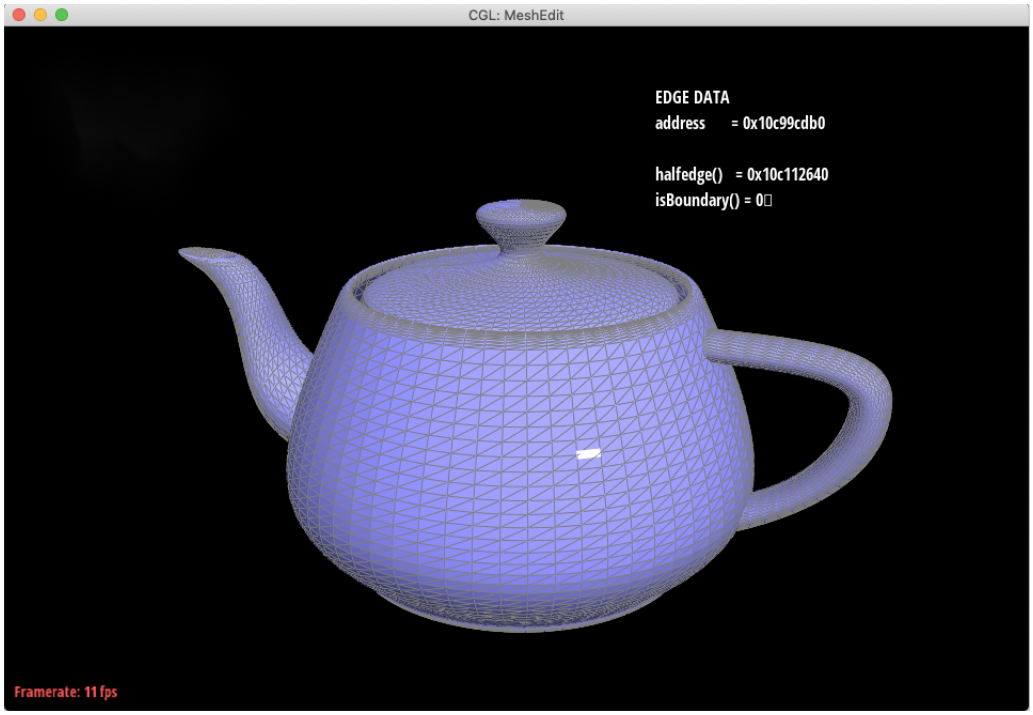
Key	Action
E	Perform one call to $BezierCurve :: evaluateStep(...)$ and cycle through levels
C	Toggle whether or not the fully evaluated Bezier curve is drawn to the screen

To verify your implementation is correct, you can repeatedly press E to cycle through each level of the de Casteljau subdivision. You can press C to toggle the Bezier curve to check if it is generated correctly based on the control points. You can also use your mouse to:

- **Click and drag** the control points to move them and see how your Bezier curve, along with 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 I - 2 and Onwards

When you run *meshedit* from Part I-2 and onwards, you will see a triangle mesh rendered on screen, as shown below.



As you hover the cursor around the screen, you will notice that mesh elements (half-edges, vertices, edges,, and faces) under the cursor are highlighted in purple or white, such as an edge in the image above. You can click on one of these elements to select it and the GUI will display some information about the element and its associated data.

Below is the full specification on keyboard controls.

Key	Action
Q	Toggle using vertex normals (Task 3)
F	Flip the selected edge (Task 4)
S	Split the selected edge (Task 5)
L	Upsample the current mesh (Task 6)
N	Select the next half-edge
T	Select the twin half-edge
W	Toggle wireframe
I	Toggle information overlay
$SPACE$	Reset camera to default position
0 - 9	Switch between GLSL shaders
R	Recompile shaders

You can also use your mouse to:

- **Click and drag a vertex** to change its position.
- **Click and drag background** or **right click and drag anywhere** to rotate the camera.

- **Scroll** to adjust the camera zoom.

You will implement area-weighted vertex normals $\boxed{Q}$, local edge flip $\boxed{F}$, local edge split $\boxed{S}$, and loop subdivision $\boxed{L}$ in Task 3, Task 4, Task 5, and Task 6, respectively. Therefore, these four key commands will do nothing until you have completed their respective part.

Starter Code Structure

Before you start, here is some basic information on starter code structure.

For Bezier curves and surfaces (Part I), you will be filling in member functions of the *BezierCurve* and *BezierPatch* classes, defined in *bezierCurve.h* and *bezierPatch.h*.

For triangle meshes (Part II), you will be filling in member functions of the *Vertex*, *HalfedgeMesh*, and *MeshResampler* class, defined in *halfEdgeMesh.h*.

We have put dummy definitions for all the functions you will need to modify within *student_code.cpp*. You will implement all tasks of this assignment in this file!

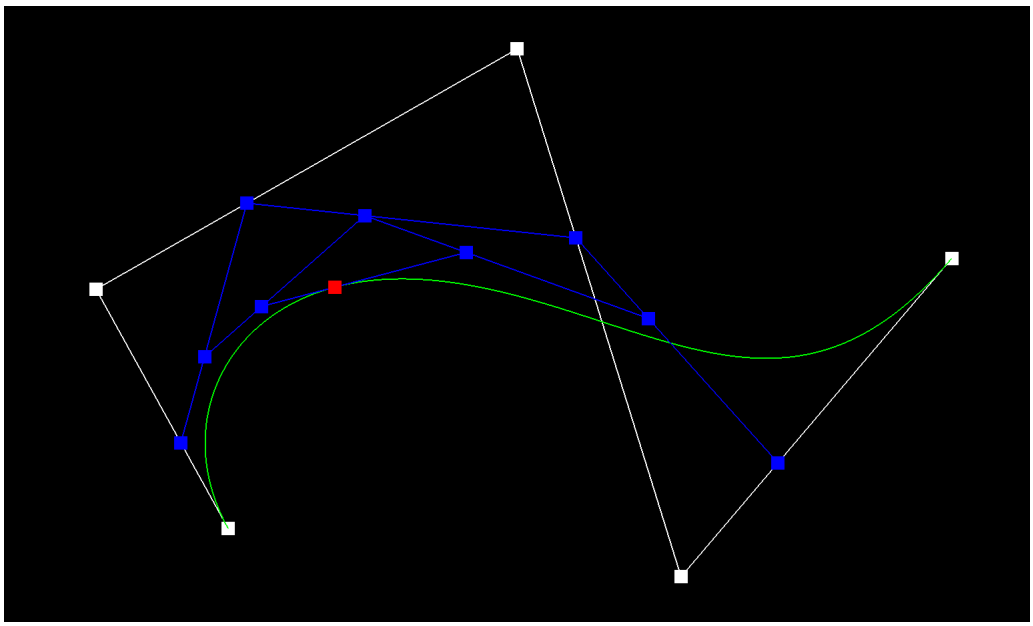


Figure 1: Bezier Curve

1 Part I: Bezier Curves and Surfaces (25 points)

In computer graphics, Bezier curves and surfaces are frequently used to model smooth and infinitely scalable curves and surfaces, such as in Adobe Illustrator (a curve drawing program) and in Blender (a surface modeling program).

A Bezier curve of degree n is defined by (n + 1)(n+1) control points. It is a parametric curve based on a single parameter t, ranging between 0 and 1.

Similarly, a Bezier surface of degree $(n, m)(n, m)$ is defined by $(n+1) \times (m+1)(n+1) \times (m+1)$ control points. It is a parametric surface based on two parameters uu and vv, both ranging between 0 and 1.

In part I, you will use de Casteljau algorithm to evaluate Bezier curves and surfaces for any given set of control points and parameters, such as the Beizer curve below. In the image, white squares are the given control points, blue squares are the intermediate control points evaluated at the given parameter by de Casteljau algorithm, and the red square is the final evaluated point that lies on the Bezier curve.

1.1 Task 1: Bezier Curves with 1D de Casteljau Subdivision (10 points)

In Task 1, you will implement Bezier curves. To get started, take a look at *bezierCurve.h* and examine the member variables defined in the class. Specifically, you will primarily be working with the following:

- *std::vector<Vector2D> controlPoints:* A *std::vector* of original control points that define the Bezier curve, initialized from input Bezier curve file (*.bzc*).
- *float t*: A parameter at which to evaluate the Bezier curve, ranging between 0 to 1.

Recall from lecture that de Casteljau algorithm gives us the recursive step below:

Given n (possibly intermediate) control points $p_1, ..., p_n$ and the parameter t, we can use linear interpolation to compute the $n-1$ intermediate control points at the parameter t in the next subdivision level, $p'_1, ..., p'_{n-1}$, where

$$p'_i = lerp(p_i, p_{i+1}, t) = (1-t)p_i + tp_{i+1} \tag{1}$$

By applying this step recursively, we eventually arrive at a final, single point and this point, in fact, lies on the Bezier curve at the given parameter t.

You need to implement this recursive step within *BezierCurve::evaluateStep(...)* in *student_ code.cpp*. This function takes as input a *std::vector* of 2D points and outputs a *std::vector* of intermediate control points at the parameter tt in the next subdivision level. Note that the parameter t is a member variable of the *BezierCurve* class, which you have access to within the function. **Each call to this function should performs only one step of the algorithm, i.e., one level of subdivision.**

Implementation Notes *std::vector* is similar to Java's *ArrayList* class. You should use the *push_ back(...)* method to add elements to a *std::vector*. This is analogous to the *append(...)* method of *ArrayList*. You can view this page for more information on *push_ back(...)*.

Sanity Check To check if your implementation is likely correct, you can run *meshedit* with a Bezier curve file (*.bzc*) and view the generated Bezier curve on screen. Refer back here for a list of controls only for Part I Task 1.

For example, you can run the following command:

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

where *bzc/curve1.bzc* is a cubic Bezier curve. The provided *bzc/curve2.bzc* is a degree-4 Bezier curve. Feel free to create your own *.bzc* files and explore other Bezier curves.

Complete the given functions: Task 1

```
BezierCurve::evaluateStep(...)
```

For Your Write-Up: Task 1

- Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.
- Take a look at the provided *.bzc* files and create your own Bezier curve with 6 control points of your choosing. Use this Bezier curve for your screenshots below.
- Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point. Press $\boxed{E}$ to step through. Toggle $\boxed{C}$ to show the completed Bezier curve as well.
- Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter t via mouse scrolling

1.2 Task 2: Bezier Surfaces with Separable 1D de Casteljau (15 points)

In Task 2, you will adapt what you have implemented for Bezier curves to Bezier surfaces. To get started, take a look at *bezierPatch.h* and examine the member variables defined in the class. Specifically, you will primarily be working with the following:

- *std::vector<std::vector<Vector3D>> controlPoints*: A 2D *std::vector* that has $n \times n$ grid of original control points. *controlPoints* is of size nn and each inner *std::vector*, i.e., *controlPoints[i]*, contains n control points at the ith-row. For example, *controlPoints[0]* and *controlPoints[n-1]* each have n control points at the bottom row and the top row, respectively.

Recall from lecture that separable 1D de Casteljau algorithm works as the following:

The inputs to the algorithm are (1) a $n \times n$ grid of original control points, P_{ij} , where i and j are row and column index, and (2) the two parameters u and v.

We first consider that each **row** of n control points, $P_{i0}, ..., P_{i(n-1)}$, define a Bezier curve parameterized by u. Just as in Task 1, we can use the recursive step to evaluate the final, single point P_i on this Bezier curve at the parameter u. We then consider that P_i for all n rows define a Bezier curve parameterized by v. (These n points lie roughly along a column. This lecture slide may help you visualize.) We can again use the recursive step from Task 1 to evaluate the final, single point P on this Bezier curve at the parameter v. This point P, in fact, lies on the Bezier surface at the given parameter u and v.

You need to implement the following functions in *student_ code.cpp*:

- *BezierPatch::evaluateStep(...)*: Very similar to *BezierCurve::evaluateStep(...)* in Task 1, this recursive function takes as inputs a *std::vector* of 3D points and a parameter t. It outputs a *std::vector* of intermediate control points at the parameter t in the next subdivision level.
- *BezierPatch::evaluate1D(...)*: This function takes as inputs a *std::vector* of 3D points and a parameter t. It outputs directly **the final, single point** that lies on the Bezier curve at the parameter t. This function does not output intermediate control points. You may want to call *BezierPatch::evaluateStep(...)* inside this function.
- *BezierPatch::evaluate(...)*: This function takes as inputs the parameter u and v and outputs the point that lies on the Bezier surface, defined by the $n \times n$ *controlPoints*, at the parameter u and v. Note that *controlPoints* is a member variable of the *BezierPatch* class, which you have access to within this function.

Sanity Check To check if your implementation is likely correct, you can run *meshedit* with a Bezier curve file (*.bez*) and view the generated Bezier curve on screen. For example, you can run the following command:

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

Complete or use the given functions.

```
BezierPatch::evaluateStep(...)
BezierPatch::evaluate1D(...)
BezierPatch::evaluate(...)
```

For Your Write-Up: Task 2

- Briefly explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces.

- Show a screenshot of *bez/teapot.bez* (**not** *.dae*) evaluated by your implementation.

2 Part II: Triangle Meshes and Half-Edge Data Structure (65 points)

Note:

- In Part II, you will be working extensively with the half-edge data structure, as well as the provided *HalfedgeMesh* class, which implements the data structure.
- Before diving into this part, we **recommend** that you read this introduction to half-edge data strucutre.
- We also highly, highly recommend that you carefully read through this primer on how to navigate meshes using the *HalfedgeMesh* class. The primer will help you get familiar with the provided *HalfedgeMesh* class quickly and present many examples, complete with code, that you may find very helpful for this section!
- Finally, we encourage you to read and understand the documentation at the beginning of *halfedgeMesh.h*, which provides supplementary information to the primer.

In Part I, you have implemented Bezier surfaces, which are parametric surfaces defined by a 2D grid of control points. As discussed in lecture, we can also represent surfaces using triangle meshes. While Bezier surfaces are better at representing smooth surfaces than triangle meshes and require less memory, Bezier surfaces are much more difficult to render directly. (Can you reason why this is the case?) In fact, Bezier surfaces are often converted into triangle meshes before being rendered to screen.

As a result, triangle meshes are sometimes the preferred way to represent 3D geometric models in computer graphics. One way to store a triangle mesh is as a list of vertices and a list of triangles indexing the vertices, illustrated in this lecture slide. Unfortunately, such simple data structure does not allow for meaningful traversal of meshes required by many geometry processing tasks. For example, to find all triangles neighbouring a given triangle, we must iterate through the entire list of triangles, which can be prohibitively expensive if the mesh has millions of triangles.

To help answer the neighbouring triangle query and many others more efficiently, the half-edge data structure explicitly stores the connectivity information among mesh elements, i.e., vertices, edges, and faces. Throughout this section, you will use the half-edge data structure to implement a few commonly used geometric operations; and we hope you will get to appreciate the simplicity and flexibility of this data structure.

2.1 Task 3: Area-Weighted Vertex Normals (10 points)

In Part 3, you will implement area-weighted normal vectors at vertices. Recall from lecture that these vertex normals can be used for Phong shading, which provides better shading for smooth surfaces than flat shading. To get started, take a look at the *Vertex* class defined in *halfedgeMesh.h*, along with the very helpful comments within the class. In short, a *Vertex* object encapsulates a single mesh vertex and has member variables such as the following (the list is not exhaustive):

- *Vector3D position*: The 3D coordinate of this vertex.
- *HalfedgeIter& halfedge*: A reference to the half-edge rooted at this vertex.
- *HalfedgeCIter halfedge*: The same half-edge as above, except this variable is *const* (i.e., constant) and cannot be modified.

To compute an area-weighted normal at a given vertex, you should use the half-edge data structure to iterate through faces (triangles) incident to the vertex, i.e., faces that have the given vertex as one of its vertices. For each such face, you weight its normal by its area. Finally, you normalize the sum of all area-weighted normals.

You need to implement this part in *Vertex::normal()* in *student_code.cpp*. This function takes no input and outputs the area-weighted vertex normal. Since *normal()* is a member function of the *Vertex* class, you have access to the class member variables, such as *Vector3D position*, within the function. To understand how to iterate through faces incident to a vertex, you may find the example *printNeighborPositions(...)* in this primer helpful. (Hint: A face in the half-edge data structure only points to its associated half-edge. To compute the area and normal of a face, what you really need are its three vertices instead of the face itself. A normal of a face is defined as a vector perpendicular to the surface at a given point. The cross product of two vectors along the face would return a third vector perpendicular to the two vectors.)

Implementation Notes For Task 3 only, you are implementing a *const* (i.e., constant) member function, which requires that no code within this function modifies any member variables. This just means that you should use *HalfedgeCIter* and not *HalfedgeIter* in *Vertex::normal()*.

For more information on the differences between *Halfedge **, *HalfedgeIter*, and *HalfedgeCIter*, check out this short article on Iterators vs Pointers.

You may also find this CGL Vector Documentation helpful for implementing any operations between vectors.

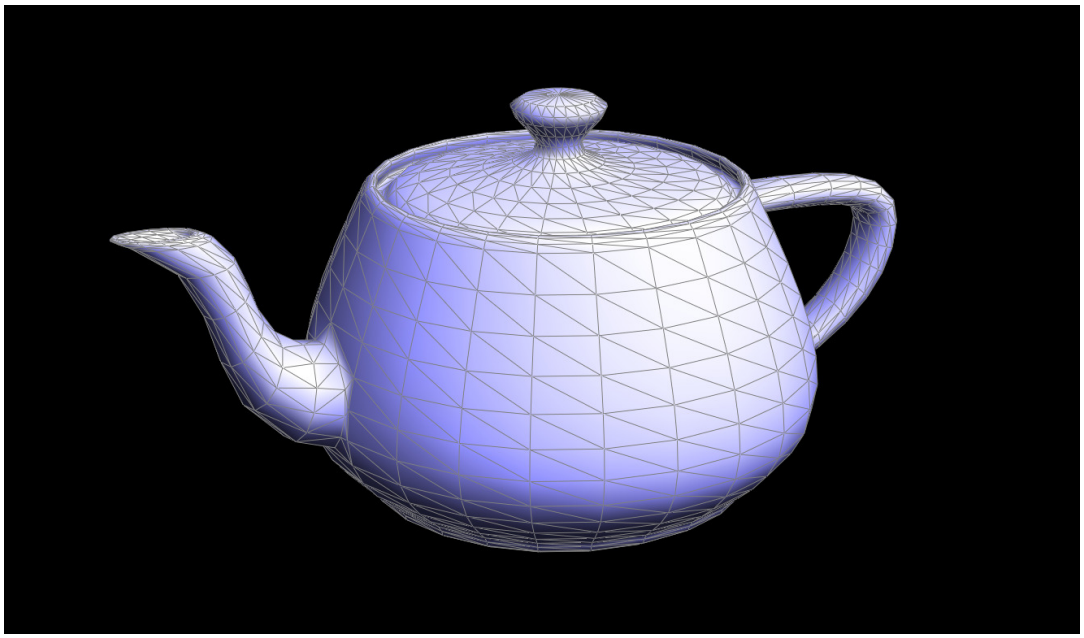
Sanity Check To check if your implementation is likely correct, you can run *meshedit* with a COLLADA mesh file (*.dae*). Once the model is loaded, press $\boxed{Q}$ to toggle the area-averaged

normals, which calls **Vertex::normal()** you have implemented, and the shading of the model should become smoother and no longer flat.

For example, you can run the following command:

```
./meshedit ../dae/teapot.dae
```

After you press $\boxed{Q}$, the shading of the teapot should look smooth like the image below. If your shading differs from the image, your implementation of *Vertex::normal()* is likely incorrect. One common mistake is that the computed normals point towards the interior of the model, as opposed to the exterior required by computer graphics. The oppositely oriented normals lead to very dark shading, which can be fixed by reversing the normals.



Complete or use the given functions. Task 3

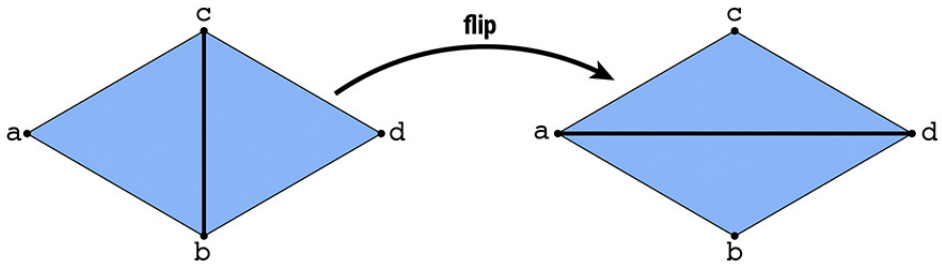
```
Vertex::normal()
```

For Your Write-Up: Task 3

- Briefly explain how you implemented the area-weighted vertex normals.
- Show screenshots of *dae/teapot.dae* (*not .bez*) comparing teapot shading with and without vertex normals. Use $\boxed{Q}$ to toggle default flat shading and Phong shading.

2.2 Task 4: Edge Flip (15 points)

In Task 4, you will implement a local remeshing operation on an edge, called a **flip**. Given a pair of triangles (a,b,c) and (c,b,d), a flip operation on their shared edge (b,c) converts the original pair of triangles into a new pair (a,d,c) and (a,b,d), as shown below:



You need to implement this part in *HalfedgeMesh::flipEdge(...)* in *student_code.cpp*. This function takes as input an *EdgeIter* to an edge that needs to be flipped, e.g., edge (b,c) above, and outputs an *EdgeIter* to the now flipped edge, e.g., edge (a,d) above.

- Draw a simple mesh, such as the pair of triangles (a,b,c) and (c,b,d) above, and write down a list of all elements, i.e., half-edges, vertices, edges, and faces, in this mesh.
- Draw the mesh after the remeshing operation and again write down a list of all elements in the now modified mesh.
- If the remeshing operation adds new elements in the modified mesh, create these new elements. An edge flip **does not** add new elements, but an edge split in Task 5 **does**.
- For **every** element in the modified mesh, set **all** of its pointers to the correct element in the modified mesh, **even if** the element being pointed to has not changed:
  - For each vertex, edge, and face, set its *halfedge* pointer.
  - For each half-edge, set its *next*, *twin*, *vertex*, *edge*, and *face* pointer to the correct element. You can use *Halfedge::setNeighbors(...)* to set all pointers of a half-edge at once.
  - We recommend setting all pointers of all elements in the modified mesh, not just the ones that have changed, because it is very easy to miss a pointer and get errors that are very difficult to debug. Once you are sure that your implementation works, you can remove the unnecessary pointer assignments if you wish.

Your implementation of *HalfedgeMesh::flipEdge(...)* should do the following:

- Never flip a boundary edge. The function should simply return immediately if either neighbouring face of the edge is on a boundary loop. Every mesh element has a useful *isBoundary()* function that returns true if the element is on the boundary.
- Perform only a constant amount of work. The cost of flipping a single edge should not be proportional to the mesh size.

- Do **not** add or delete any mesh elements. There should be exactly the same number of elements before and after the edge flip. You only need to reassign pointers.

Implementation Notes Given any mesh element, you can use the provided *check_for(...)* debugging function to check which other elements in the mesh point to it. For example, you can use this function to confirm if an element is pointed to by the right number of other elements. Please read the Debugging Aid section in this primer on the HalfedgeMesh class for more information.

Sanity Check After loading a model from a *.dae* file, you can click to select an edge and press $\boxed{F}$ to flip it. Refer back here for a list of controls for Task 2 and onwards. Sometimes, your implementation may seem to work when you flip an edge only once. We recommend that you flip an edge a few more times to be more certain that your code really works as expected. Task 6 will require you to have a working implementation of edge flips. There should be no holes in your mesh created by an edge flip. Note that you may reach a degenerate case where one triangle looks much darker than the others, this is ok.

Complete or use the given functions. Task 4

```
1 HalfedgeMesh::flipEdge(...)
```

For Your Write-Up: Task 4

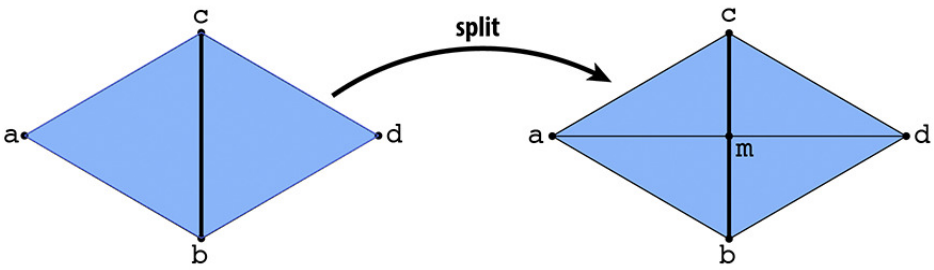
- Briefly explain how you implemented the edge flip operation and describe any interesting implementation /
- Show screenshots of the teapot before and after some edge flips.
- Write about your eventful debugging journey, if you have experienced one.

2.3 Task 5: Edge Split (15 points + 6 extra)

In Task 5, you will implement a different local remeshing operation on an edge, called a split. Given a pair of triangles (a,b,c) and (c,b,d) , a split operation on their shared edge (b,c) inserts a new vertex mm at its midpoint and connects the new vertex to each opposing vertex aa and dd, yielding four triangles as shown below:

You need to implement this part in *HalfedgeMesh::splitEdge(...)* in *student_code.cpp*. This function takes as input an *EdgeIter* to an edge that needs to be split, e.g., edge (b,c) above, and outputs a *VertexIter* to the newly inserted vertex, e.g., vertex m above.

Because an edge split adds new mesh elements, e.g., a new vertex, two new triangles, three new edges, and etc, you will have more pointers to keep track of and may find that an edge split is a little trickier to implement than an edge flip. We encourage you to again follow our recommendation in



Task 4 to ensure that all pointers of all mesh elements point to the right elements in the modified mesh.

Your implementation of *HalfedgeMesh::splitEdge(...)* should do the following:

- Ignore requests to split boundary edges, unless you are trying for extra credit. The function can simply return immediately if either neighbouring face of the edge is on a boundary loop. Note that splitting a boundary edge does make sense, but flipping a boundary edge does not make sense. (Can you reason why this is the case?)
- Assign the position of the new vertex to the midpoint of the original edge. Remember from Task 3 that the Vertex class has a member variable *Vector3D position*.
- Perform only a constant amount of work. The cost of splitting a single edge should not be proportional to the mesh size.
- Create only as many new elements as needed. The mesh should not have any elements that is not connected to the rest of the mesh.

Extra Credit (7 points): Support edge splits for boundary edges. For this, you will need to carefully read the paragraphs on virtual boundary face right before the Iterating Over Mesh Entities section in this primer on the HalfedgeMesh class. In short, you will split the edge in half, but only split in half the face that is non-boundary.

Sanity Check After loading a model from a *.dae* file, you can click to select an edge and press $\boxed{S}$ to split it. Refer back here for a list of controls for Task 2 and onwards. To verify that your implementation is likely correct, you can flip some edges that you have split and then split some edges that you have flipped. You can also alternate between flipping and splitting edges many times in mesh regions that are nearby and far apart; and check if the mesh changes correctly. Task 6 will require you to have a working implementation of edge splits.

Complete or use the given functions. Task 5

```
1 HalfedgeMesh::splitEdge(...)
```

For Your Write-Up: Task 5

- Briefly explain how you implemented the edge split operation and describe any interesting implementation / debugging tricks you have used.
- Show screenshots of a mesh before and after some edge splits.
- Show screenshots of a mesh before and after a combination of both edge splits and edge flips.
- Write about your eventful debugging journey, if you have experienced one.
- If you have implemented support for boundary edges, show screenshots of your implementation properly handling split operations on boundary edges.

2.4 Task 6: Loop Subdivision for Mesh Upsampling (25 points)

Sometimes, we may wish to convert a coarse polygon mesh into a higher-resolution one for better display, more accurate simulation, and etc. Such conversion requires an upsampling algorithm that nicely interpolates or approximates the original mesh data.

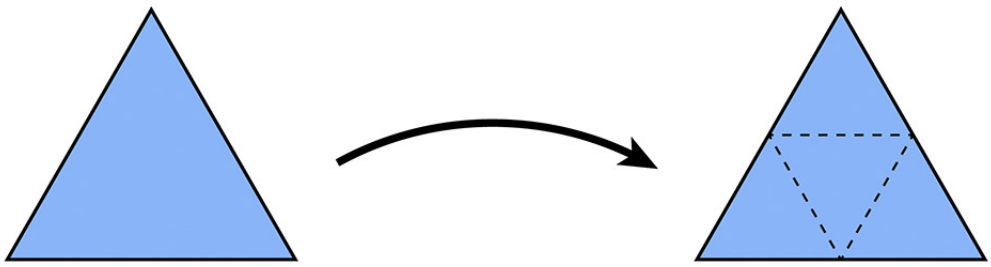
Unfortunately, the techniques we have learned to upsample 2D images, such as bilinear filtering, do not easily translate to upsampling 3D meshes. Among many reasons, a mesh often has vertices at irregular locations, as opposed to on a regular grid like an image. (Can you think of other reasons?)

In Part 6, you will implement a mesh upsampling method, called *loop subdivision*. In short, loop subdivision upsamples a mesh by (1) subdividing each of its triangles into four smaller triangles and (2) updating vertices of subdivided mesh based on some weighting scheme.

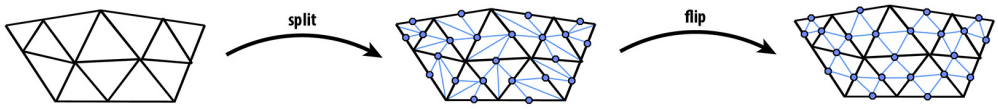
You need to implement these two steps of loop subdivision, outlined in more details below, in *MeshResampler::upsample(...)* in *student_code.cpp*. This function has no outputs and takes as input a reference to a *HalfedgeMesh* that will be subdivided.

A single loop subdivision consists of the following two steps. If we repeatedly apply these two steps, we will converge to a smooth approximation of our original mesh.

1. 4-1 subdivision Subdivide each triangle in the mesh into four by connecting edge midpoints, as shown below. This is called a 4-1 subdivision.



To perform 4-1 subdivisions over the entire mesh, you can use the edge flip and edge split operation you have implemented. Specifically, you can do the following:

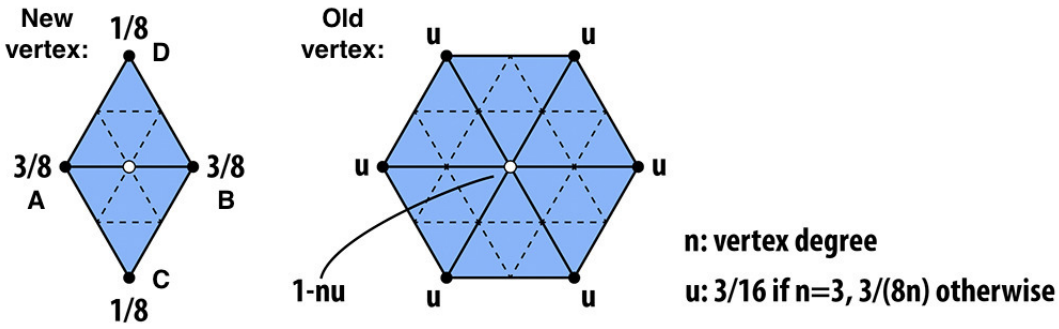


(1) Split every existing edge of the mesh in any order. (2) Flip any new edge that connects an old vertex and a new vertex.

Note: After (1), every original edge will now be represented by 2 edges. You should not flip these edges because they are already along the boundary of the 4-way subdivided triangles. In the figure above, you should only flip blue edges that connect an old vertex and new vertex. You should *not* flip any black new edges.

Update vertex positions Update vertex positions as weighted average of neighboring vertex positions. Each new vertex position can be calculated from the original vertex positions as seen in the figure below.

The figure below shows the weights we use to (1) compute the position of a newly added vertex or to (2) update the position of an existing vertex. The new vertex and old vertex are depicted as the white circle in their respective mesh.



(1) The position of a new vertex splitting the shared edge (A, B) between a pair of triangle (A, C, B) and (A, B, D) is

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

(2) The updated position of an old vertex is

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

where n is the vertex degree, i.e., number of edges incident to the vertex, u is the constant shown in the figure, *original_position* is the **original position** of the old vertex, and *original_neighbor_position_sum* is the sum of all **original positions** of the neighboring vertices.

Notes While you can implement loop subdivision exactly as described, we actually **recommend** that you update vertex positions **before** performing 4-1 subdivisions over the entire mesh. More specifically, you may want to follow the steps below:

- Step A: Compute the positions of both new and old vertices using the original mesh. We want to perform these computations before subdivision because traversing a coarse mesh is much easier than traversing a subdivided mesh with more elements.
- Step B: Subdivide the original mesh via edge splits and flips as described.
- Step C: Update all vertex positions in the subdivided mesh using the values already computed.

If you choose to follow our recommendation, you may consider using the following member variables of the *Vertex* and *Edge* class to facilitate your implementation:

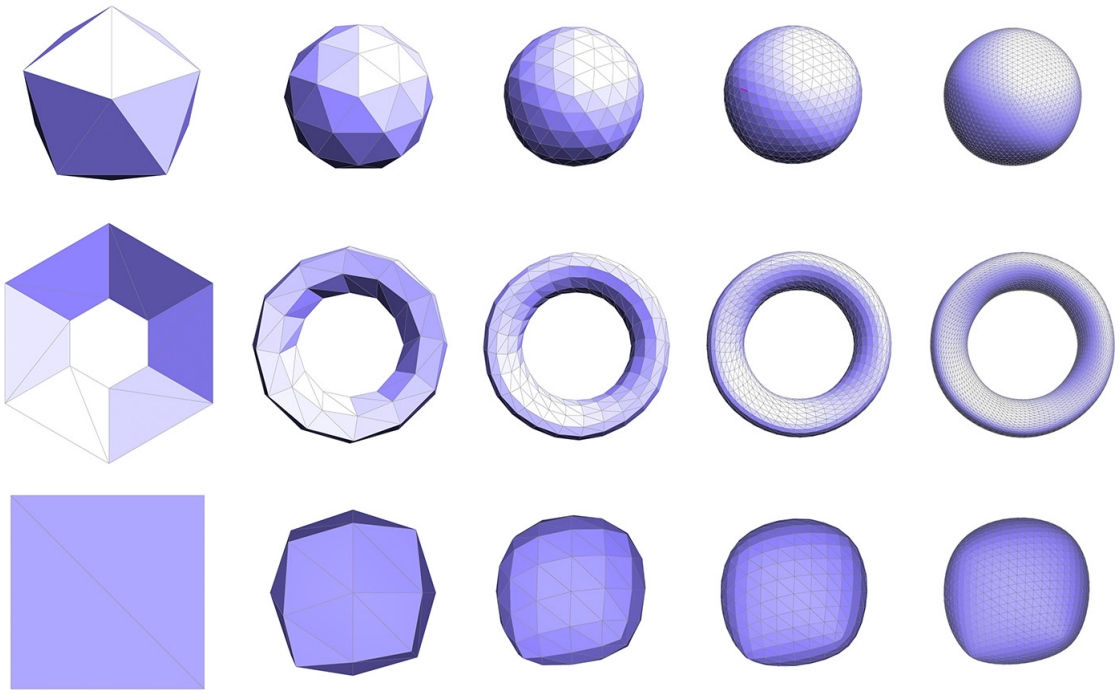
- *Vector3D Vertex::newPosition* temporarily stores the updated position of a new or old vertex, computed using the weighted average described above.
- *Vector3D Edge::newPosition* temporarily stores the position of the vertex that will ultimately be inserted at the edge midpoint.
- *bool Vertex::isNew* flags whether a vertex exists in the original mesh or is a new vertex inserted at an edge midpoint by subdivision.
- *bool Edge::isNew* flags whether an edge exists in the original mesh or is a new edge added by subdivision.

Notes: While computing the updated vertex positions in Step A, you should **not** change the value of *Vector3D Vertex::position* because the weighted average is based on vertex positions in the **original** mesh. (Please take a moment to understand why.) You will, however, need to update *Vertex::position* in Step C.

Examples below illustrate the correct behavior of the loop subdivision algorithm:

Extra Credit (7 points): Support meshes with boundary. You will first need to make sure that your edge split function appropriately handles boundary edges. You do not need to change your edge flip function. You will also need a different weighted average for boundary vertices. See "A Survey of Subdivision-Based Tools for Surface Modeling" for more information.

Extra Credit (7 points): Implement additional subdivision schemes. There exist many alternatives to loop subdivision. Triangle subdivision schemes include the Butterfly scheme, the



modified Butterfly scheme, and sqrt3-Subdivision. One of the most popular subdivision schemes for quadrilateral meshes is *Catmull-Clark*. There are also special subdivision schemes, such as polar subdivision for handling meshes with vertices of high degree.

Complete or use the given functions. Task 6

```
1 MeshResampler::upsample(...)
```

For Your Write-Up: Task 6

- Briefly explain how you implemented the loop subdivision and describe any interesting implementation / debugging tricks you have used.
- Take some notes, as well as some screenshots, of your observations on how meshes behave after loop subdivision. What happens to sharp corners and edges? Can you reduce this effect by pre-splitting some edges?
- Load *dae/cube.dae*. Perform several iterations of loop subdivision on the cube. Notice that the cube becomes slightly asymmetric after repeated subdivisions. Can you pre-process the cube with edge flips and splits so that the cube subdivides symmetrically? Document these effects and explain why they occur. Also explain how your pre-processing helps alleviate the effects.
- If you have implemented any extra credit extensions, explain what you did and document how they work with screenshots..

Tips

- Start early. This assignment has multiple parts and you want to manage your time accordingly. Remember that pesky bugs can waste your time like no other; and always keep in mind Hofstadter's Law.
- Task 1 and 2 should be relatively straightforward to implement once you understand Bezier curves and surfaces and de Casteljau algorithm.
- Task 3 to 6 will be more difficult to implement correctly right away since they involve managing pointers, which are exposed to you as iterators. Make sure you test these parts, especially Task 4 and 5, together on a few different meshes. While correct behaviors do not imply correct code, incorrect behaviors do imply incorrect code.
- Make sure you allocate enough time to do a good job on the write-up!

Submission

Project Write-Up Guidelines and Instructions Please follow the same overall structure as described in the deliverables section below. The goals of your write-up are for you to (1) think about and articulate what you have built and learned in your own words and (2) have a write-up of the project to take away from the class. Your write-up should include the following:

- An overview of the project, including your approach to and implementation for each of the parts, as well as what problems you have encountered and how you solved them. Strive for clarity and succinctness.
- For each part, make sure to include the results described in the corresponding Deliverables section, in addition to your explanation. If you failed to generate any results correctly, provide a brief explanation on why.
- The final, optional part seven, you have the opportunity to be creative and individual! Be sure to provide a good description on what you were going for, what you did, and how you did it, :).
- Clearly indicate any extra credit items you have completed; and provide a thorough explanation and illustration for each of them.

The write-up is one of our main methods to evaluate your work, so it is important to spend the time to do it correctly and thoroughly. Plan ahead to allocate time for the write-up well before the deadline.

Project Write-Up Deliverables and Rubric This rubric lists the basic, minimum requirements for your write-up. The content and quality of your write-up are extremely important. You should make sure to at least address all the points listed below. The extra credits are intended for students who want to challenge themselves and explore methods beyond the fundamentals. They are not worth a lot of points, so do not necessarily expect to use extra credit points to make up for lost points elsewhere.

Overview Give a high-level overview of what you have implemented in this assignment. Think about what you have built as a whole. Share your thoughts on what interesting things you have learned from completing this assignment.

Task 1

- Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.
- Take a look at the provided *.bzc* files and create your own Bezier curve with 6 control points of your choosing. Use this Bezier curve for your screenshots below.
- Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point. Press $\boxed{E}$ to step through. Toggle $\boxed{C}$ to show the completed Bezier curve as well.
- Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter t via mouse scrolling.

Task 2

- Briefly explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces.
- Show a screenshot of *bez/teapot.bez* (not *.dae*) evaluated by your implementation.

Task 3

- Briefly explain how you implemented the area-weighted vertex normals.
- Show screenshots of *dae/teapot.dae* (not *.bez*) comparing teapot shading with and without vertex normals. Use $\boxed{Q}$ to toggle default flat shading and Phong shading.* Briefly explain how you implemented the area-weighted vertex normals.
- Show screenshots of *dae/teapot.dae* (not *.bez*) comparing teapot shading with and without vertex normals. Use $\boxed{Q}$ to toggle default flat shading and Phong shading.

Task 4

- Briefly explain how you implemented the edge flip operation and describe any interesting implementation / debugging tricks you have used.
- Show screenshots of a mesh before and after some edge flips.
- Write about your eventful debugging journey, if you have experienced one.

Task 5

- Briefly explain how you implemented the edge split operation and describe any interesting implementation / debugging tricks you have used.
- Show screenshots of a mesh before and after some edge splits.
- Show screenshots of a mesh before and after a combination of both edge splits and edge flips.
- Write about your eventful debugging journey, if you have experienced one.
- If you have implemented support for boundary edges, show screenshots of your implementation properly handling split operations on boundary edges.

Task 6

- Briefly explain how you implemented the loop subdivision and describe any interesting implementation / debugging tricks you have used.
- Take some notes, as well as some screenshots, of your observations on how meshes behave after loop subdivision. What happens to sharp corners and edges? Can you reduce this effect by pre-splitting some edges?
- Load *dae/cube.dae*. Perform several iterations of loop subdivision on the cube. Notice that the cube becomes slightly asymmetric after repeated subdivisions. Can you pre-process the cube with edge flips and splits so that the cube subdivides symmetrically? Document these effects and explain why they occur. Also explain how your pre-processing helps alleviate the effects.
- If you have implemented any extra credit extensions, explain what you did and document how they work with screenshots.