CS-C3100 Computer Graphics, Fall 2018

Lehtinen / Aarnio, Kemppinen, Ollikainen

Programming Assignment 2: Curves and Surfaces

Due Sun Oct 7th at 23:59.

In this assignment, you will be implementing spline curves and mesh subdivision. These are powerful building blocks for animation, 3D modeling and other tasks. The recommended extra credit work applies your spline curves to additional ways of modeling 3D objects: swept surfaces and generalized cylinders.

This assignment is considerably more challenging than the intro assignment, and the majority of your time will be spent thinking about the theory rather than how to write code. Start early so you have time for debugging, thinking about the issues that come up, and asking for help.

Requirements (maximum 10 p) ***on top of which you can do extra credit***

1. **Evaluating Bezier curves (2 p)**
2. **Evaluating B-spline curves (2 p)**
3. **Subdividing a mesh to smaller triangles (2 p)**
4. **Computing positions for the new vertices (2 p)**
5. **Smoothing the mesh by repositioning old vertices (2 p)**

1 Getting Started

The sample solution `example.exe` is included in the starter code distribution. Notice that we have added some buttons to the right of the window.

First, load up `core.swp` by clicking the 'Load SWP' button on the right. For this assignment, parsing the files is already done for you.

The file `core.swp` contains four curves. Their control points are shown in yellow, and the resulting curves are shown in white. You can toggle what is drawn with the UI buttons. Click the 'Draw normals' button to see the local coordinate system at points of a curve.

Subdivision surfaces are loaded with the 'Load OBJ' button. (Previously loaded curves will stay visible, and can only be hidden by restarting the application.) Press the 'Refine subdivision' and 'Coarsen subdivision' buttons to change the subdivision level. Play with the different models to see what happens. Note that boundary checks are not yet implemented in the model solution, so the `patch.obj` subdivision does not work properly.

The SWP files under the `srev` and `gcyl` subdirectories are for extra credit work. In addition to splines, these files contain sweep curves and generalized cylinders based on those splines. `example.exe` can draw them; click the 'Draw surface' button to show or hide the generated surface.

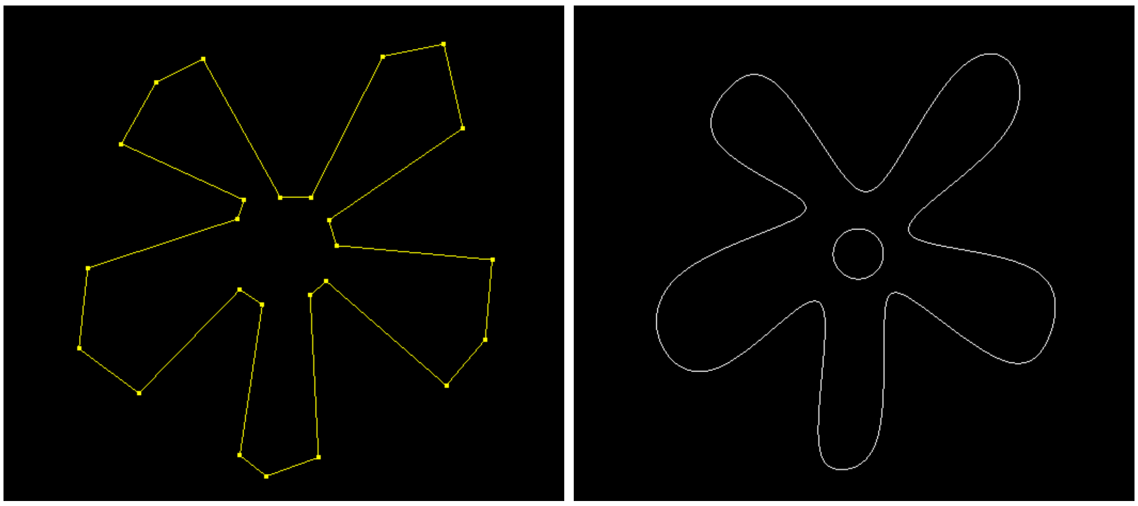
Now, build the starter project and launch it. Load the SWP file `circles.swp`. The starter code is fully functional on this example as-is. If you click the 'Draw normals' button, you'll also notice that the coordinate frames are computed for you. The scene can be rotated, panned and zoomed using the mouse and its buttons; try it out.

This assignment reuses old code, and the rendering code is very different from the intro assignment. The code you see here is old, "immediate mode" OpenGL. Please note that you don't have to touch the rendering code in this assignment, and all the remaining assignments will again use modern style OpenGL.

2 Implementing Curves

Your first task is to implement piecewise cubic Bezier splines and B-splines. Given an array of control points, you are to generate a set of points that lie on the spline curve (which can then be used to draw the spline by connecting them with line segments). For instance, the control points shown in the picture on the left will result in the piecewise uniform cubic B-spline on the right (in this example, the first three control points are identical to the last three, which results in a closed curve).

The starter code already renders the control points for you. You will need to evaluate the spline individually and build a list of vertices per each curve.



Computing points on the spline is sufficient if the only goal is to draw it. For other applications of splines, such as animation and surface modeling, you might need to generate additional information (see extra credit section).

R1 Bezier curves (2 p)

Implement the evaluation of Bezier curve points (function `evalBezier`). Start by writing the `coreBezier` function in `curve.cpp`, which is the basis for evaluating one 4-point bezier curve at small steps, generating sequences of vertices that can then be used for drawing short straight lines on the screen, ultimately making it look like a smooth line.

In the `evalBezier` function, you generate points along the whole chained-together Bezier curve. Because we chain the individual Beziers together by sharing the last control point of the preceding curve with the first control point of the next curve, this function must get $3n + 1$ vertices as input ($n \geq 1$). I.e., the first curve segment is defined by control points $\{p_1, p_2, p_3, p_4\}$, the second by $\{p_4, p_5, p_6, p_7\}$, etc. You need to loop over the input control points and feed the proper subsets to the `coreBezier` function. You need to string the results of `coreBezier` together in the output vector of tessellated points.

The sample function `evalCircle` shows how to use the `CurvePoint` and `Curve` classes. You'll only need to fill in the V member of the `CurvePoints`; the normals and others are for extra credit.

See the slides on spline curves for more in-depth information about the underlying math!

R2 B-splines (2 p)

Evaluate a B-spline curve in `evalBspline`. As you already have the code for evaluating points on a Bezier curve from R1, it's best to just change the basis from B-spline to Bezier, and use the code you already wrote. This is the benefit of generality we've talked about in class!

Similar to `evalBezier`, you should loop over the input control points. Remember that B-splines build the curve with a sliding window that only increases by one at each round; the first segment is built from control points $\{p_1, p_2, p_3, p_4\}$, but differing from the above, the second uses control points $\{p_2, p_3, p_4, p_5\}$, etc.

For each such subset, put the four control points in a 4x4 matrix and apply the B-spline-to-Bezier conversion matrix (that you know how to build). Then read off the new control points, now in the Bernstein basis, off the columns of the resulting matrix, and pass them on to `evalBezier`.

After completing this part, the camera animation system (example in `swp/campath/sponza.swp`) works partially – the position is correctly animated, but the camera looks in a constant direction. Fixing this is an extra credit assignment.

3 Loop Subdivision Surfaces

In this part of the assignment, you are to implement triangle mesh subdivision using the Loop scheme. Algorithms like this are the core building blocks of all modern modeling systems!

The basic idea is to subdivide each triangle in the input mesh into four new triangles, and to carefully compute positions for the new vertices *and the old vertices* in a way that results in a smooth surface when we keep subdividing. In the limit, we get a surface that is actually mathematically smooth almost everywhere. Choosing the appropriate ways of computing the positions is deep and fascinating, but fortunately, implementing a subdivision scheme does not require one to understand why. (Phew.) While you should be able to fulfill the requirements using this document only, we *heartily* recommend you check out Denis Zorin and his colleagues' SIGGRAPH course notes on subdivision to find out more. You'll find that Loop's algorithm is a popular, but by no means only choice of subdivision algoritm, for instance.

You will find the code you need to complete in `Subdiv.cpp`. The class `MeshWithConnectivity` is the one where the heavy lifting happens.

3.1 Debug tools

If you run into problems with for example indexing the triangles when generating the one-ring (see R5 description,) it might help to use the provided debug highlight functionality to see what your code does. If you press `alt` while you have a model loaded, the debug code first finds the vertex your mouse is currently pointing at, then runs the one-ring generation code only for that one vertex. When LoopSubdivision is running in this debug mode, the `debugPass` boolean will be set to true. You can then push vertex indices into the `highlightIndices` vector to highlight those vertices as large red points in the GL view. The solution executable demonstrates this by highlighting the one-ring around the chosen vertex. You can naturally customize what happens in the debug pass if you want to debug some other aspect of the subdivision algorithm.

3.2 Warmup: Connectivity Information

In subdivision algorithms, it is important that vertices are shared between adjacent triangles. This is easiest if the position for each vertex (as well as its color, normal, etc.) are only stored once, and the triangles refer to these tables by indices. In `MeshWithConnectivity`, the positions, normals, and colors are stored in the member vectors `positions`, `normals`, and `colors`.

Now, we specify each triangle as a sequence of three indices into a vertex table, which would look something like shown in Figure 2, left. We store the index information in the `indices` member of `MeshWithConnectivity`, defined as `std::vector<Vec3i> indices`. So, each triangle has a 3-component vector of indices that describes the vertices. The positions of the three vertices of the ith triangle are then simply `positions[indices[i][0]]`, `positions[indices[i][1]]`, and `positions[indices[i][2]]`.

Furthermore, we will need to know which triangles are connected to which other triangles. This is called *connectivity information* or *mesh topology.* In order to fulfill the requirements, you need to understand and use the simple data structure described below, but its complete implementation is given to you in the starter code.

First, we define the three edges of a triangle as follows. For each triangle,

1. The first edge e_0 runs between vertices i_0 and i_1,
2. The second edge e_1 runs between vertices i_1 and i_2,
3. The third edge e_2 runs between vertices i_2 and i_0.

Note that since we always specify the vertices of triangles in consistent (either clockwise or counterclockwise, but not mixed) order, these edges are directed.

Now, with this definition in hand, information about neighboring triangles can be stored in another table of three integers per triangle, as shown in Figure 2, middle. In our starter code, this is the `neighborTris` vector. The table simply says, for each triangle, what is the index of the triangle on the other side of each of the edges, which are numbered as above. Take a moment to convince yourself that the indexing of the vertices, triangles, edges, and neighbors you see in the two figures indeed makes sense. If a triangle does not have a neighbor on the other side of some edge, we denote it with a -1. You will not have to worry about it to fulfill requirements, though.

This is not quite enough information yet. In addition to the index of the neighboring triangle, we encode, for each edge of each triangle, what is the index of the corresponding edge of the neighboring triangle. For instance, the first edge of triangle 1 consists of vertices $4, 6$. (Verify this in the index table!) But in the neighboring triangle 7 – verify that the first entry for triangle 1 in the neighbor table is indeed 7 – the corresponding edge $6, 4$ is not the first, but the third edge! Verify also this from the index table. In general, there is no way to be sure what these correspondences are except by looking at the actual data, so we encode this in a *third* array of three integers per triangle called *neighbor edges*, and we put a 2 in the first position for triangle 1. This table encodes the information "for each edge in the mesh, what is the index of the corresponding edge in the neighboring triangle." In the starter code, this is the `neighborEdges` vector. Together, neighbor triangle and neighbor edge indices encode all we need to know about connectivity on the mesh.

Figure 3 shows a simple triangle mesh. The first step of the Loop algorithm is to insert a new vertex in the middle of each *edge* in the input mesh.

3.3 Starter code

The starter code has all the structure you need; you just have to fill in the proper handling of a few things, all in the `LoopSubdivision` function. The code already fills in all the

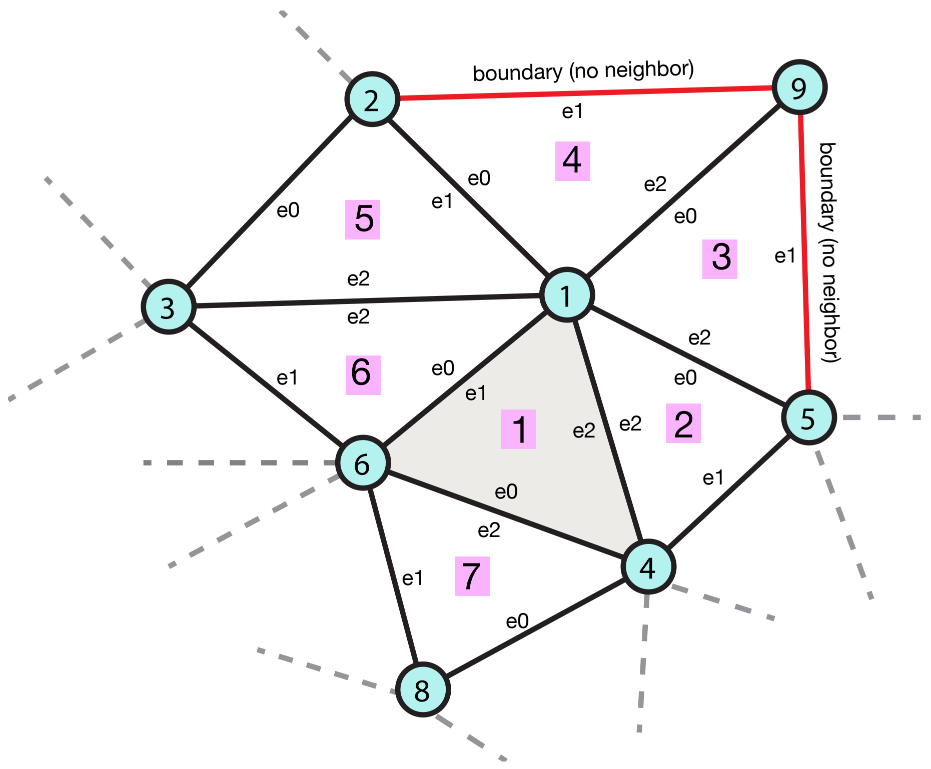


Figure 1: Encoding mesh connectivity. Vertices are shown as light blue circles, triangle numbers are shown in pink boxes. The labels e0, e1, e2 denote the ordering of the vertices within each triangle. See text for details.

Tri#	i_0	i_1	i_2
1	4	6	1
2	1	5	4
3	1	9	5
4	1	2	9
5	3	2	1
6	1	6	3
7	4	8	6

Vertex indices

Tri#	n_0	n_1	n_2
1	7	6	2
2	3	...	1
3	4	-1	2
4	5	-1	3
5	...	4	6
6	1	...	5
7	...	...	1

Neighbor triangles

Tri#	ne_0	ne_1	ne_2
1	2	0	2
2	2	...	2
3	2	-1	0
4	1	-1	0
5	...	0	2
6	1	...	2
7	...	...	0

Neighbor edges

Figure 2: Storing triangles as triplets of vertex indices. See Figure 1 for illustration.

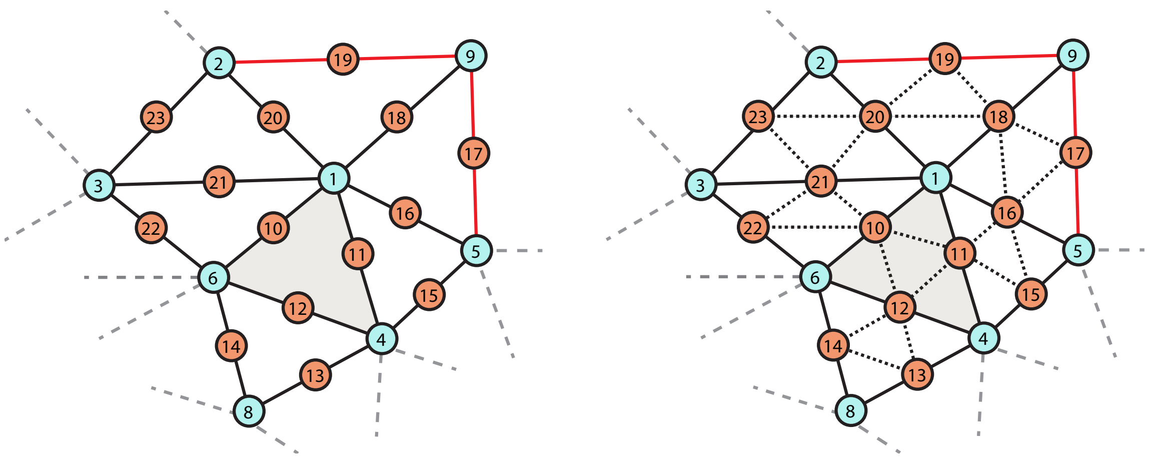


Figure 3: **Left:** In each round of Loop subdivision, we insert new vertices (red) in the midpoints of all edges of the input mesh. **Right:** We then simply replace all triangles of the input mesh with four new ones built from the old and newly-inserted vertices.

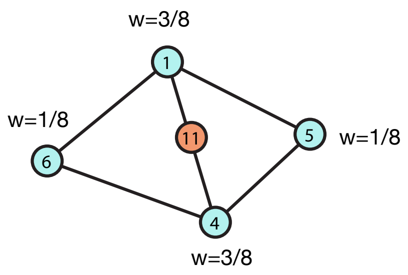
connectivity data so that you get the picture. It's missing the generation of new faces based on the new indices, though.

As mentioned above, the class `MeshWithConnectivity` is everything you need here. The main app supplies the original mesh to it, and asks it to give back the subdivided mesh. The needed raw information is unpacked to the member variables of the class.

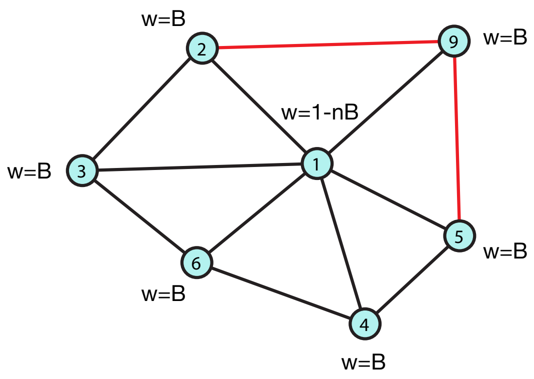
R3 Rebuilding the triangles (2 p)

Your first job is to build the new triangle vertex indices at the end of the function according to the figure 3. After completing that successfully, you should see that each triangle is getting replaced by four smaller ones when increasing the subdivision level in the UI (use the wireframe mode!), but the shape of the object is not changed. That you will fix in the next sections.

Also, while the new positions are being created, you'll need to build a mapping from the incoming edges to the new vertices (the red circles in 3). From that mapping, you then read the indices when doing the rebuild below. For this, we use the `std::map` data structure, which associates *keys* to *values* in a way that allows fast insertion and lookup. By examining the code you'll note that we use a pair of ints as the key to identify edges. (Obviously, edges are uniquely determined by the indices of their two endpoints). The two loops in the beginning of `MeshWithConnectivity::LoopSubdivision()` go over each triangle and each edge of each triangle of the input mesh; this means we actually iterate over each edge twice, in opposite orders. To get around this, we identify the edge by their endpoints in sorted order, hence the max/min expressions when computing `edge`. Next we check if we've seen this edge already, and only if we haven't we place a new vertex in the middle of the edge and append it to the end of the vertex data vectors using `push_back()`. Your job is to insert the index of the newly-created vertex



Weights for computing the positions for the new (odd) vertices. The position for vertex 11 is $p_{11} = \frac{3}{8}p_1 + \frac{3}{8}p_4 + \frac{1}{8}p_6 + \frac{1}{8}p_5$.



Weights for computing positions for old (even) vertices. The new position p'_1 of vertex 1 depends on all the $n = 6$ vertices that are connected to it through edges. We use the formulas $B = \frac{3}{8n}$ when $n > 3$, and $B = \frac{3}{16}$ when $n = 3$. Then, $p'_1 = (1 - nB)p_1 + B(p_3 + p_2 + p_9 + \ldots)$

Figure 4: The magic in subdivision surfaces happens when we compute positions for the newly-inserted vertices *and also the old vertices*. **Left:** Computing the new position for a newly-inserted vertex. Its position is a linear combination of the positions of the two endpoints of the edge it was inserted into, and the two vertices on the opposite sides of the triangles the original edge was part of. Finding the appropriate vertices is easy using the connectivity structure described above. **Right:** The new positions for the old vertices only depend on the old vertices. Again, the position is a linear combination of the positions. To find the weights for vertex 1, we must use our connectivity structure to loop around all the other vertices connecting to it. (This set is called the *one-ring* or *1-ring*.) Note that in a general closed, watertight triangle mesh, you can have any number $n \geq 3$ connecting vertices! In this example, $n = 6$. Looping around to find the connecting vertices is fortunately pretty simple using our connectivity structure. We then compute the weight B, which is the same for all vertices, multiply the positions of the 1-ring vertices by it, and finally add the old position, weighted by $1 - nB$, and we're done.

into `new_vertices` so that we keep track of its index properly (and don't try to insert it another time). You can use the array index operator [] for doing this. This is super easy, and just introduces you to what is happening.

With the new vertex data at hand, build the new triangle list at the bottom of the file. For each old triangle, you create four new ones, according to the figure. The old indices are given in the triplet `even`. Build the similar vector of 3 ints `odd` using the information from `new_vertices`. For this, you will need to query `new_vertices` to get the index of each new vertex added to the middle of each edge of the old triangle. You can use `make_pair`, similar to the construction of `edge` earlier in the same method, to build the corresponding pairs of ints (i.e. edges) for each edge, and query the map with the array index operator [], as you know you are always going to find the right values. When you've fetched the indices of the odd (new) vertices, remove the last line of the loop from the starter code (which just outputs a single triangle with the old vertices), and replace it with four `push_back()`s that properly build the new triangles out of the odd and even indices you now have at hand.

R4 Positions for odd (new) vertices (2 p)

Now that you can see the results of the planar subdivision, let's actually do it using Loop's method! The idea is to weigh the nodes such that the surface gets smoother and smoother as the level of subdivision increases. Figure 4 illustrates what the weights are. For each new vertex, there is exactly four surrounding vertices to take into account.

In the first nested loops in `MeshWithConnectivity::LoopSubdivision()`, you are going to compute the proper weighted average of the four relevant input vertices instead of merely placing the new vertices into the middle of the original edge like the starter code does. To do this, you are going to need the connectivity information.

Again, the nested loops iterate over each edge of each triangle. At each iteration, the variables `v0` and `v1` are the endpoints of the edge we are looking at. We merely need two more vertices: the ones on the opposite sides of the triangles on both sides of the edge. Getting the first one is easy: we are at triangle i and the edge that starts from its jth vertex; you know perfectly well how to get the one thats two steps ahead of the start vertex (look at the beginning of the inner loop). Getting the one on the other side of the edge is almost as simple. Use the connectivity structure to get the index of the triangle on the other side and the index of the edge that corresponds to the current one; then you can just walk around the other triangle two steps like you just did in this current triangle.

Once you've found the indices of all the relevant vertices, compute new positions, normals, and colors using the weights from the figure instead of the simple midpoint given in the starter code. Careful thinking with pen and paper helps here; draw lots of arrows. Use the weights 3/8 for `v0` and `v1`.

Handling mesh boundaries is not needed for the requirement, but your code must not crash when encountering a boundary. Simply checking neighbor indices against -1, and setting the fourth vertex position to 0 will be enough. Doing boundary handling properly is an extra, and can be found in the recommended section.

R5 Repositioning the even (old) vertices (2 p)

After implementing R3 and R4, the supplied icosahedron looks like a spikeball. The old vertices must also be repositioned, again as in the figure 4. You have to loop through the neighbors of the vertex in question. By looking at the neighbor information illustrated in figure 1, this is surprisingly easy. You have to use the neighbor triangle and edge information to walk to the next vertex.

The second loop given in the starter code in `MeshWithConnectivity::LoopSubdivision()` (after the nested ones) will find you a vertex `v0` *and an edge* emanating from that vertex — the jth edge of triangle i. Clearly, `indices[i][(j+1)%3]`, the other end of the jth edge, will get you a vertex that's connected to `v0`. Great, you've found one already!

The mental model that will lead you to the goal is simple. The idea, now, is to loop around all the other edges that emanate from the vertex `v0` by jumping from triangle to triangle until you hit the one you started at. It turns out that this is really simple to do using the connectivity information we have stored; when you've done this properly, you have a simple loop that starts from triangle i and edge j, updates the current triangle and edge by walking the connectivity, and keeps doing this until the current triangle loops back to i. *If you find your code containing conditionals or other complex logic, you've gone astray! The end result is a really simple while loop.*

Be careful with indexing! You *will* need a pen and lots of paper, but in the end, this is just one loop in about ten lines of code, and a few lines for the original position. It might also help to utilize the provided debug highlighting functionality. See the debug tools section above and code comments for more details.

Again, handling mesh boundaries properly is not needed for this requirement, but your code must not crash on triangle edges. It's enough to simply check neighbor triangle indices against -1, and leave the vertex at its original position at this stage. If you want to do proper handling for mesh boundaries, you can find an extra credit for it in the recommended section.

4 Extra Credit

As always, you are free to do other extra credit work than what's listed here. We'll be fair with points, but if you want to attempt something grand, better talk to us beforehand.

4.1 Recommended

- Proper handling of boundaries for Loop subdivision (3p). You can find information on how to do this in Denis Zorin's SIGGRAPH course notes and Matt Fisher's simple Loop tutorial You can use the provided `patch.obj` for trying this out. We will update the solution exe to do the right thing as well.

- Local coordinate frames on the curve (1 p). Compute local coordinate frames along the curve (as described towards the end of this document) and display them. You should render the curve in white, and the **N**, **B**, and **T** vectors in red, green, and blue, respectively. See more about this in the appendix.

  The camera animation system (example in `sponza.swp`) uses the local frame as its orientation so instead of looking in a constant direction, the camera now looks forward along its path. Note that you need to toggle the camera path orientation mode on your UI to see this.

- Surfaces of revolution (see appendix) (3 p). Generate and display surfaces of revolution (of a curve on the xy-plane around the y-axis) and compute the surface normals correctly. The starter code includes support for this, it's the 'Draw curve' button. The solution implements this, so you can check how the results should look like.

  To get you motivated, here is an image that was generated using Maya from one of the swept surfaces that your program will create.

- Generalized cylinders (appendix) (3 p). Sweep the spline as the outline of a cylinder along another curve. You are not required to precisely match the results of the sample solution, as it includes a somewhat arbitrary choice of the initial coordinate frame. Again, this is included in the model solution, and some SWPs include generalized cylinders.

- Other subdivision schemes (? p). Catmull-Clark, Doo-Sabin, etc. Make it possible to change the scheme on the fly, and maybe also to draw the different ones simultaneously to compare them.

4.2 Easy

- While subdividing, assign new colors to the vertices based on how old they are to visualize how the smoothness develops. (1p)

4.3 Medium

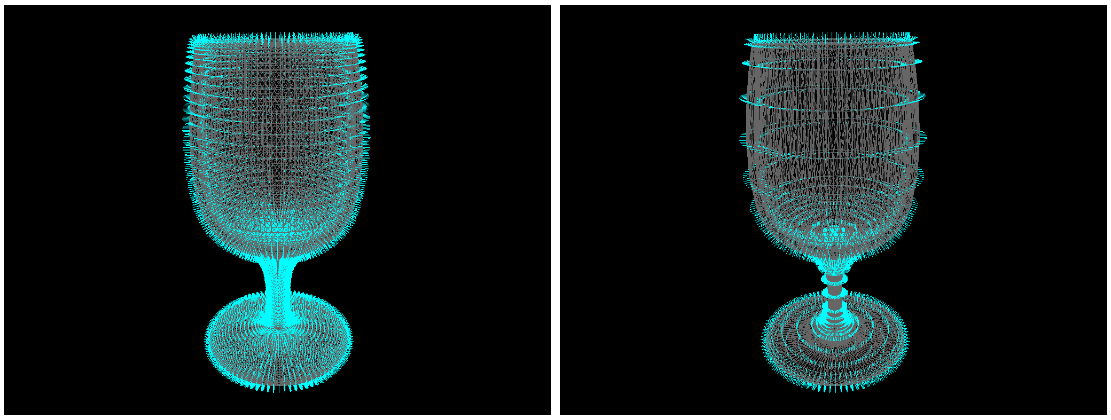
- Bezier interpolation for orientations (4p). Use Bezier splines to interpolate between orientation control points in the camera animation mode. Tthe given `sponza.swp` contains a camera path with orientations encoded as quaternions. Since only two orientations can be combined in a sensible manner at a time, you'll need to implement De Casteljau's algorithm to interpolate the control points using spherical linear interpolation (slerp) and convert the resulting quaternion to an orientation matrix.

- Implement another type of spline curve, and extend the SWP format and parser to accommodate it (3 p). We recommend Catmull-Rom splines, as they are C^1-continuous and interpolate all the control points, but feel free to try others (Bessel-Overhauser, Tension-Continuity-Bias). If you implement this extension, please make sure that your code can still read standard SWP files, since that's how we're going to grade it. Additionally, provide some SWP files that demonstrate that they work as they're supposed to.

- Isosurface extraction (5 p). Convert volume data to a triangle mesh using e.g. marching cubes: http://www.cs.toronto.edu/ jacobson/seminar/lorenson-and-cline-1987.pdf. You can find volumetric density data sets that you can use here: https://graphics.stanford.edu/data/voldata/

- Implement another kind of primitive curve that is not a spline (? p). Preferably, pick something that'll result in interesting surfaces. For instance, you might try a corkscrew shape or a Trefoil Knot. Both of these shapes would work great as sweep curves for generalized cylinders. Again, you'll probably want to extend the SWP format to support these sorts of primitives. We'll be fair and give you points based on how elaborate your primitive is.

- Adaptive step size (4 p). In this assignment, the suggested strategy for discretizing splines involves making uniform steps along the parametric curve. However, it is difficult to choose the appropriate number of steps, since the curvature of splines can vary quite a bit. In fact, *any* choice of a uniform step size will result in either too few steps at the sharpest turns, or too many in the straight regions. To remedy this situation, implement a recursive subdivision technique to adaptively control the discretization so that more samples are taken when needed. You will need to analyze the derivatives of the curve to get tight bounds.


Comparison on the wineglass object. Uniform step size (33120 faces) on the left, adaptive step size (18540 faces) on the right.

- Render your surfaces (either subdivision surface or surfaces of revolution and generalized cylinders) more interestingly (? p). Change the materials, or even better, change the material as some function of the surface. You might, for instance, have the color depend on the sharpness of the curvature. Or jump ahead of what we are covering in class and read about texture mapping, bump mapping, and displacement mapping. If you visualize curvature properly, you'll get at least 3 extra points.

4.4 Hard

- Implement piecewise Bezier and B-spline surfaces (3 p), i.e. surfaces with gaps in between. The description (and the sample code) only supports swept surfaces with C^1-continuous sweep curves. This is all that is required, but it may be desirable to sweep curves with sharp corners as well. Extend your code to handle this functionality for piecewise Bezier curves. If you do this extension, you should also extend the SWP format so that control points can be specified. Additionally, you should provide some sample SWP files to demonstrate that your technique will work.

- Cylinder curve scaling (4 p). We can extend the generality of generalized cylinders by allowing the profile curve to be controlled by other curves as well. For instance, you may wish to not only control the orientation of the profile curve using a sweep curve, but the relative scale ("thickness" of the cylinder) of the profile curve using a *scale curve*. This is more challenging than it may sound at first, since you must worry about the fact that the sweep curve may not have the same number of control points as the scale curve.

- Implement a user interface so that it is easy for users to specify curves using mouse control (5 p). A user should be able to interactively add, remove, and move control points to curves, and those curves should be displayed interactively. This application may be implemented as an extension to your assignment code, or as a standalone executable. Either way, it should be able to export SWP files. You may choose to implement a purely two-dimensional version, or, for additional credit, a three-dimensional curve editor. If you choose the latter, you should provide an intuitive way of adding and moving control points using mouse input (this is somewhat challenging, since you will be trying to specify and move three-dimensional points using a two-dimensional input device). See how the `CommonControls` object is used in the starter code to get an idea of how to add new UI-components. Provide a few sample SWP models and screenshots that you created with this UI. We might show the best ones in the lectures!

4.5 Very hard

- Subdivision surface fitting (10 p). See e.g. papers Loop Subdivision Surface Fitting by Geometric Algorithms and Fitting Subdivision Surfaces.
- Multiresolution editing of the subdivided surfaces (10 p). E.g. Mudbox and ZBrush work about like this. See Interactive Multiresolution Mesh Editing for ideas.

4.6 Tangential

- The sample code will export your surfaces to OBJ format. Files in this format can be imported into a variety of 3D modeling and rendering packages, such as Maya. This software has been used for a number of video games, feature films (including *The Lord of the Rings*), and television shows (including, believe it or not, *South Park*). You may wish to make your artifact more interesting by rendering it in this software, or other 3D-modeling software. Since the course isn't about learning software tools, you won't receive extra credit for this. However, your artifact will look amazing when we post the images on the web for everyone to see. The image at the beginning of this document was generated using this software.

4.7 Research-grade extra

Umbra Software challenge: Filtering and filling in 3D laser scans 3D laser scanning devices such as LIDAR produce clouds of 3D points that represent the shape of the space being scanned. There are no triangles, just points! As they shoot laser pulses out from a central location, it's simple to see why the density of the points drops with increasing distance from the scanner; furthermore, the locations of the points is not exact and some noise remains. And clearly, locations that are not visible to the scanner don't get any points. (This is why large locations require multiple scans that are aligned and fused into one.)

In this assignment, download complex 3D scans from a suitable repository, such as this, write code for visualizing the point clouds interactively, and see what you can do in order to 1) remove noise and 2) fill in more points in areas of low density. You're free (and are encouraged!) to consult research literature on finding potential approaches to try. You can also try to implement techniques for generating triangle meshes from the point clouds.

This assignment is vague and open-ended on purpose. Be sure to explain your approach, rationale, and references along with your results! Umbra Software will host an excursion to the top 5 students. You retain complete ownership of your code; your score will be determined by course staff; you are in no obligation to show your code or results to Umbra. *Due to the open-ended nature of this assignment, you can turn your solution along with a future assignment round — you do not have to submit with Round 2. Preferably, write a description of your solution up in a separate PDF file that contains text and images detailing your solution.*

5 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Visual Studio 2017. Comment out any functionality that is so buggy it would prevent us seeing the good parts. Check that your `README.txt` (which you hopefully have been updating throughout your work) accurately describes the final state of your code.
- Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all your code, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive.
- Sanity check: look inside your ZIP archive. Are the files there? (Better yet, unpack the archive into another folder, and see if you can still open the solution, compile the code and run.)

Submit your archive in MyCourses folder "Assignment 2".

A Appendix: Curve normals

Computing points on the spline is sufficient if the only goal is to draw it. For other applications of splines, such as animation and surface modeling, it is necessary to generate more information.

We'll go over these details first in two dimensions and then go up to three.

A.1 Two Dimensions

Consider a simple example. Suppose we wanted to animate a car driving around a curve $\mathbf{q}(t)$ on the xy-plane. Evaluating $\mathbf{q}(t)$ tells us where the car should be at any given time, but not which direction the car should be pointing. A natural way to choose this direction is with the first derivative of curve: $\mathbf{q}'(t)$.

To determine the appropriate transformation at some t, we introduce some shorthand notation. First, we define the position $\mathbf{V}$ as:

$$\mathbf{V} = \mathbf{q}(t)$$

We define the unit tangent vector $\mathbf{T}$ as:

$$\mathbf{T} = \mathbf{q}'(t)/||\mathbf{q}'(t)||$$

Then we define the normal vector $\mathbf{N}$ as a unit vector that is orthogonal to $\mathbf{T}$. One such vector that will satisfy this property is:

$$\mathbf{N} = \mathbf{T}'/||\mathbf{T}'||$$

Then, if we assume that the car is pointed up its positive y-axis, the appropriate homogeneous transformation can computed as:

$$\mathbf{M} = \begin{bmatrix} \mathbf{N} & \mathbf{T} & \mathbf{V} \\ 0 & 0 & 1 \end{bmatrix}$$

Unfortunately, there is a problem with this formulation. Specifically, if $\mathbf{q}$ has an inflection point, the direction of the normal will flip. In other words, the car will instantaneously change orientation by 180 degrees. Furthermore, if the car is traveling along a straight line, $\mathbf{N}$ is zero, and the coordinate system is lost. This is clearly undesirable behavior, so we adopt a different approach to finding an $\mathbf{N}$ that is orthogonal to $\mathbf{T}$.

We introduce a new vector orthogonal to $\mathbf{T}$ that we'll call $\mathbf{B}$. This is known as the *binormal*, and we'll arbitrarily select it to be in pointing in the positive z-direction. Given $\mathbf{B}$, we can compute the appropriate normal as:

$$\mathbf{N} = \mathbf{B} \times \mathbf{T}$$

Note that $\mathbf{N}$ will be unit and orthogonal to both $\mathbf{B}$ and $\mathbf{T}$, because $\mathbf{B}$ and $\mathbf{T}$ are orthogonal unit vectors. This may not seem like the most intuitive method to compute the desired local coordinate frames in two dimensions, but we have described it because it generalizes quite nicely to three.

A.2 Three Dimensions

To find appropriate coordinate systems along three dimensions, we can't just use the old trick of selecting a fixed binormal $\mathbf{B}$. One reason is that the curve might turn in the direction of the binormal (i.e., it may happen that $\mathbf{T} = \mathbf{B}$). In this case, the normal $\mathbf{N}$ becomes undefined, and we lose our coordinate system. So here is the solution that we suggest.

We will rely on the fact that we will be stepping along $\mathbf{q}(t)$ to generate discrete points along the curve. In other words, we might step along $\mathbf{q}(t)$ at $t_1 = 0$, $t_2 = 0.1$, and so on. At step i, we can compute the following values (just as in the two-dimensional case):

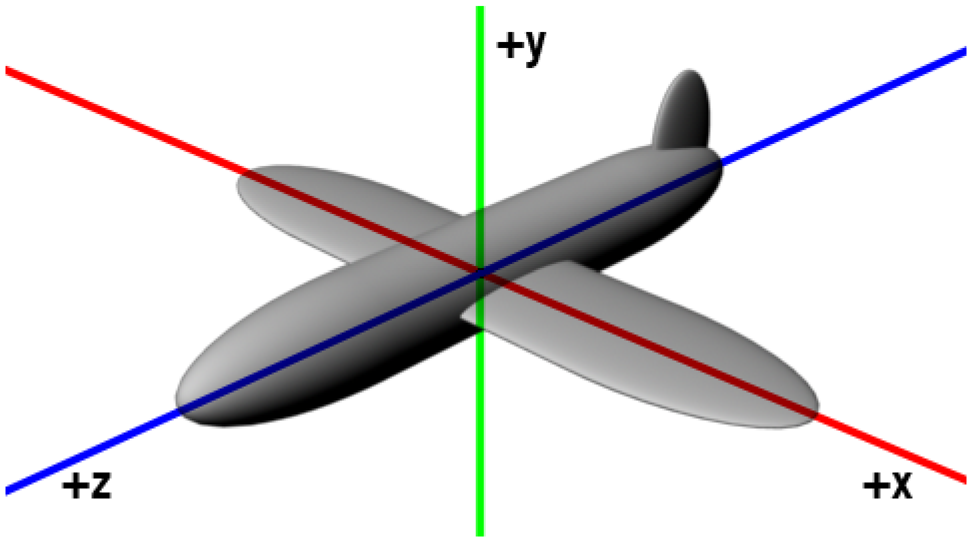
$$\begin{aligned}\mathbf{V}_i &= \mathbf{q}(t_i)\\ \mathbf{T}_i &= \mathbf{q}'(t_i).\text{normalized}()\end{aligned}$$

To select $\mathbf{N}_i$ and $\mathbf{B}_i$, we use the following recursive update equations:

$$\begin{aligned}\mathbf{N}_i &= (\mathbf{B}_{i-1} \times \mathbf{T}_i).\text{normalized}()\\ \mathbf{B}_i &= (\mathbf{T}_i \times \mathbf{N}_i).\text{normalized}()\end{aligned}$$

In these equations, `normalized()` is a method of `Vector3f` that returns a unit-length copy of the instance (i.e., `v.normalized()` returns $\mathbf{v}/||\mathbf{v}||$). We can initialize the recursion by selecting an arbitrary $\mathbf{B}_0$ (well, almost arbitrary; it can't be parallel to $\mathbf{T}_1$). This can then be plugged into the update equations to choose $\mathbf{N}_1$ and $\mathbf{B}_1$. Intuitively, this recursive update allows the the normal vector at t_i to be as close as possible to the normal vector at t_{i-1}, while still retaining the necessary property that the normal is orthogonal to the tangent.

tt with the two-dimensional case, we can use these three vectors to define a local coordinate system at each point along the curve. So, let's say that we wanted to animate this airplane:



And let's say that we wanted to align the z-axis of the airplane with the tangent $\mathbf{T}$, the x-axis with the normal $\mathbf{N}$, the y-direction with the binormal $\mathbf{B}$, and have the plane at position $\mathbf{V}$. This can be achieved with the following transformation matrix:

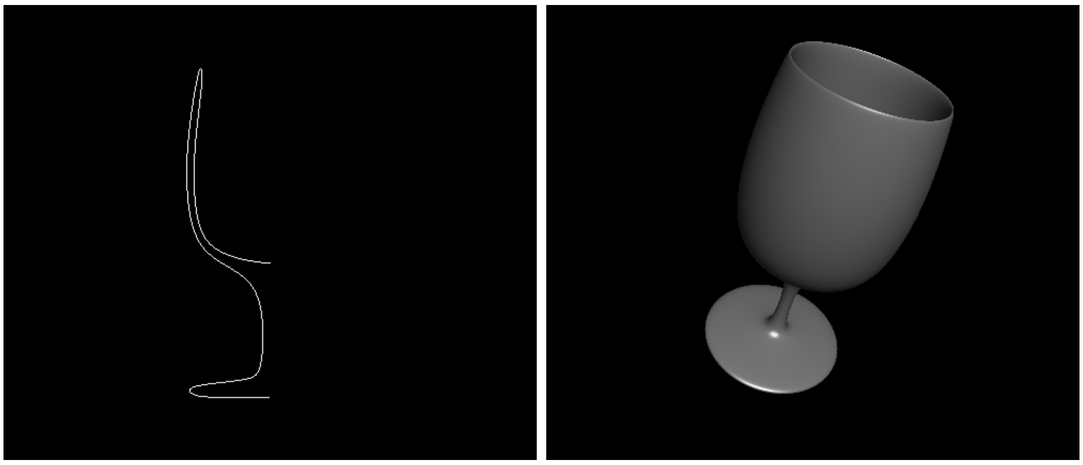
$$\mathbf{M} = \begin{bmatrix} \mathbf{N} & \mathbf{B} & \mathbf{T} & \mathbf{V} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And that summarizes one way to compute local coordinate systems along a piecewise spline. Although the recursive update method that we proposed works fairly well, it has its share of problems. For one, it's an incremental technique, and so it doesn't really give you an analytical solution for any value of t that you provide. Another problem with this technique is that there's no guarantee that closed curves will have matching coordinate systems at the beginning and end. While this is perfectly acceptable for animating an airplane, it is undesirable when implementing closed generalized cylinders.

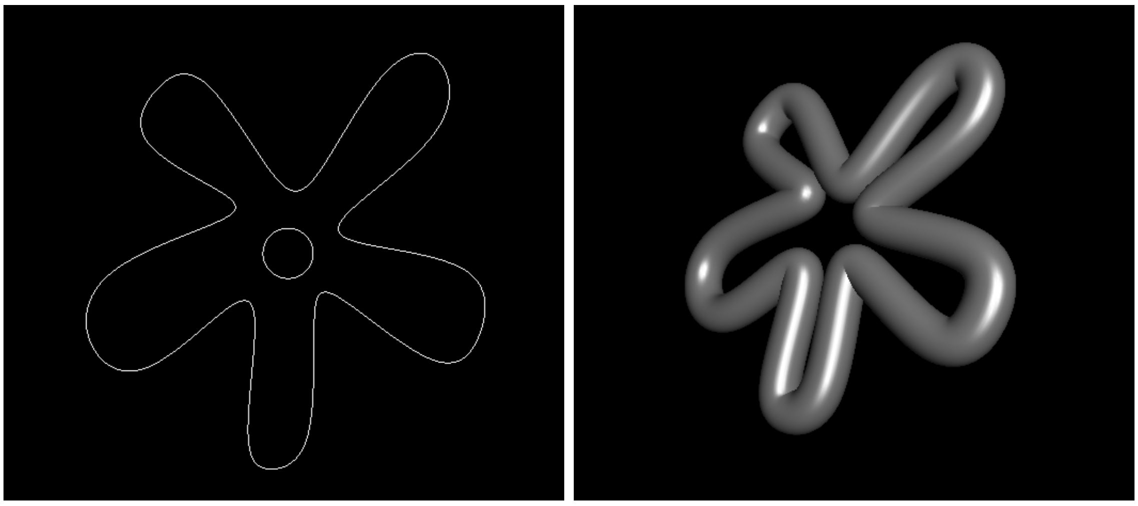
B Appendix: Implementing Surfaces

Use the curves that you generated to create swept surfaces. Specifically, the type of surfaces you are to handle are *surfaces of revolution* and *generalized cylinders*.

The following images show an example of a surface of revolution. On the left is the *profile curve* on the xy-plane, and on the right is the result of sweeping the surface about the y-axis.



The images below show an example of a generalized cylinder. On the left is what we'll call the *sweep curve*, and the surface is defined by sweeping a *profile curve* along the sweep curve. Here, the profile curve is chosen to be a circle, but your implementation should also support arbitrary two-dimensional curves.



B.1 Surfaces of Revolution

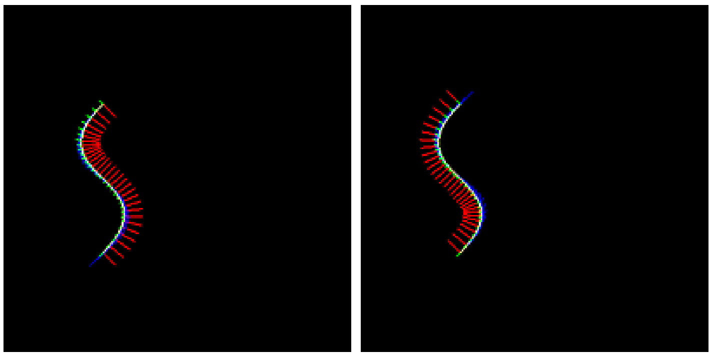
For this assignment, we define a surface of revolution as the product of sweeping a curve on the xy-plane *counterclockwise* around the positive y-axis. The specific direction of the revolution will be important, as you will soon see.

Suppose that you have already evaluated the control points along the profile curve. Then, you can generate the *vertices* of the surface by simply duplicating the evaluated curve points at evenly sampled values of rotation. This should be your first task during the implementation process.

However, vertices alone do not define a surface. As we saw from the previous assignment, we also need to define normals and faces. This is where things get a little more challenging. First, let's discuss the normals. Well, we already have normal vectors $\mathbf{N}$ from the evaluation of the curve. So, we can just rotate these normal vectors using the same transformation as we used for the vertices, right? Yes, and no. It turns out that, if we transform a *vertex* by a homogeneous transformation matrix $\mathbf{M}$, its *normal* should be transformed by the *inverse transpose* of the top-left 3×3 submatrix of $\mathbf{M}$. A discussion of why this is the case appears in the Red Book. You can take comfort in the fact that the inverse transpose of a rotation matrix is itself (since rotation is a rigid transformation).

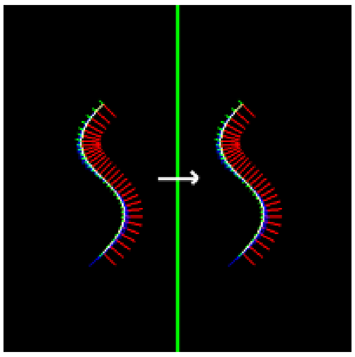
Another thing that you'll need to worry about is the orientation of the normals. For OpenGL to perform proper lighting calculations, the normals need to be facing *out* of the surface. So, you can't just blindly rotate the normals of any curve and expect things to work.

To appreciate this issue, observe the following Bezier curves. The difference between them is that the normals (the red lines) are reversed. This is the result of just reversing the order of the control points. In other words, even though the curves are the same, the normals depend on the direction of travel.



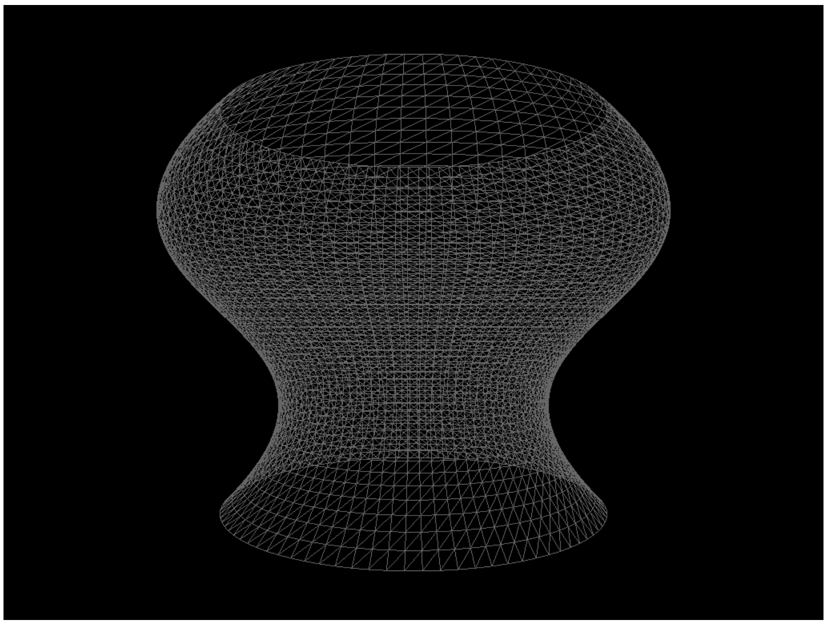
In this assignment, we will assume that, for two-dimensional curves, the normals will always point to the *left* of the direction of travel (the direction of travel is indicated by the blue lines). In other words, the image on the left is correct. This is to be consistent with the fact that, when you're drawing a circle in a counterclockwise direction, the analytical normals will always point at the origin. With this convention, you will actually want to *reverse* the orientation of the curve normals when you are applying them to the surface of revolution. Note that, if you implement two-dimensional curves as we described previously, you will automatically get this behavior.

We must also keep in mind that translating the curve may disrupt our convention. Consider what happens if we just take the curve on the left side, and translate it so that it is on the other side of the y-axis. This is shown below (the y-axis is the thick green line).



Here, the normals that were once facing towards the y-axis are now facing away! Rather than try to handle both cases, we will assume for this assignment that the profile curve is always on the *left* of the y-axis (that is, all points on the curve will have a negative x-coordinate).

So far, we have ignored the faces. Your task will be to generate triangles that connect each repetition of the profile curve, as shown in the following image. The basic strategy is to zigzag back and forth between adjacent repetitions to build the triangles.



In OpenGL, you are required to specify the vertices in a specific order. They must form a triangle in counterclockwise order (assuming that you are looking at the front of the triangle). If you generate your triangle faces backwards, your triangles will have incorrect lighting calculations, or even worse, not appear at all. So, when you are generating these triangle meshes, keep this in mind. In particular, this is where our assumption of counterclockwise rotation about the y-axis comes in handy.

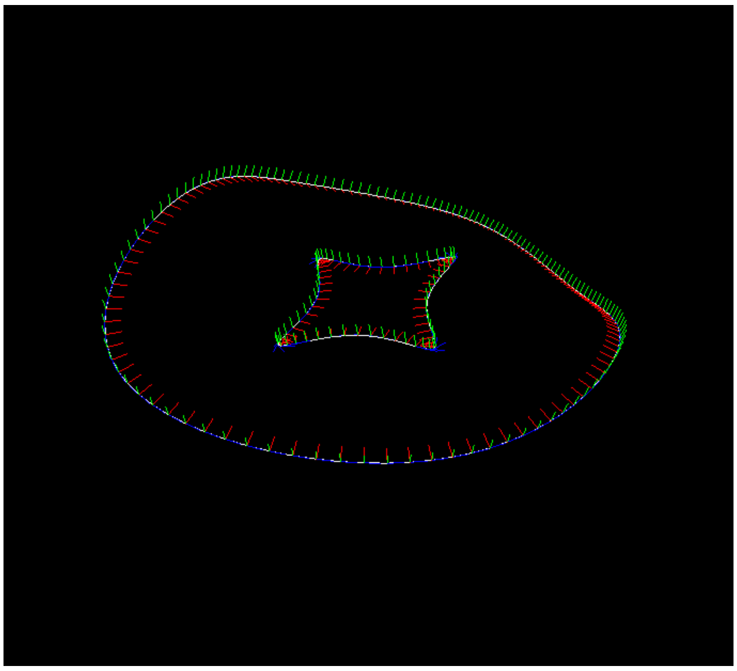
B.2 Generalized Cylinders

By now, you should be ready to implement generalized cylinders. They are very much like surfaces of revolution; they are formed by repeating a profile curve and connecting each copy of the profile curve with triangles. The difference is that, rather than sweeping the two-dimensional profile curve around the y-axis, you'll be sweeping it along a three-dimensional sweep curve.

Just as with surfaces of revolution, each copy of the profile curve is independently transformed. With surfaces of revolution, we used a rotation. With generalized cylinders, we will use the coordinate system defined by the **N**, **B**, **T**, and **V** vectors of the sweep curve. To put it in the context of a previous discussion, imagine dragging a copy of the profile curve behind an airplane that's flying along the sweep curve, thus leaving a trail of surface.

The process of selecting the correct normals and faces is very similar to how it is done for surfaces of revolution. We recommend that you write your triangle meshing code as a separate function so that it may be reused.

Here is a neat example of a generalized cylinder. First, let's see the profile curve and the sweep curve. Both of these are shown with local coordinate systems at points along the curves. Specifically, the blue lines are the **T**s, the red lines are the **N**s, and the green lines are the **B**s.



The small curve is chosen to be the profile curve, and the large one is chosen to be the sweep curve. The resulting generalized cylinder is shown below.