6.837: Computer Graphics Fall 2012

Programming Assignment 1: Curves and Surfaces

Due September 26th at 8:00pm.

In this assignment, you will be implementing splines and swept surfaces to model interesting shapes. The primary goal of this assignment is to introduce you to splines and coordinate systems. Upon successful completion of this assignment, you will be rewarded with a powerful tool for modeling 3D shapes.

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

You may also wish to look at some cool results from a previous year's class. The remainder of this document is organized as follows.

1 Getting Started

The sample solution `a1soln` is included in the starter code distribution. You may need to change the its permissions to be able to run it (type `chmod u+x a1soln` at the Athena prompt).

First, load up `core.swp` (type `./a1soln swp/core.swp`). Note that this is a different way of reading the contents of a file than the method used in assignment zero. In that assignment, we used the `<` operator to put the contents of the file into standard input. Now, we will tell the program the filename as a command line argument, and the program will open and read the file itself (the starter code does this already for you). If you look at `main(...)`, you'll see that an `ifstream` object is created from the filename given on the command line, and the curves files are read from this stream. In the function `parseFile` (in `parse.cpp`), the curve data is read from the file into data structures that the code you write will use. 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 the display of local coordinate frames using the `c` key, and you can toggle the display of control points using the `p` key.

Then, load up `weird.swp`. This displays a generalized cylinder. You can toggle the display mode of the surface with the `s` key. By default, it starts with smooth shading, but you can also turn the surface off to get a better look at the curves, or render a wireframe surface with normals.

Try viewing some of the other SWP files to get an idea of the sorts of curves and surfaces that these techniques can generate.

2 Summary of Requirements

This section summarizes the core requirements of this assignment. You will find more details regarding the implementation of these requirements later in this document. Note that, for this and most future assignments, you are free to start your implementation from scratch (in C++). However, the provided starter code implements a number of these requirements already, so we encourage you to either start from that or read through the files to see the implementation.

2.1 File Input

This is fully implemented in the starter code. Your program must read in a specific file format (SWP) that allows users to specify curves (Bezier, B-Spline, and circles) and surfaces (surfaces of revolution and generalized cylinders). Many SWP files are provided in the `swp` subdirectory. The specification of the file format is given in the header file of the provided parser (`parse.h`). The starter code distribution also includes a number of SWP files that your code must be able to read and process (if you implement the assignment from scratch and file input is broken, you will receive no credit for the assignment). Note that, for grading, we may be using SWP files that are *not* included with the starter code.

2.2 User Interface

This is fully implemented in the starter code. Your program must provide functionality to rotate the model using mouse input and select the display mode of the curves and surfaces. If you choose to implement the assignment from scratch, play with the sample solution and support equivalent functionality. Again, if you choose to implement the assignment from scratch and it fails to support the user interface requirements, you will receive no credit.

2.3 Curves (45% of grade)

Your program must be able to generate and display piecewise cubic Bezier and B-spline curves. In addition, it must correctly compute local coordinate frames along the curve (as described elsewhere in 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. If you use the starter code, you can fill in the `evalBezier` and `evalBspline` functions in `curve.cpp` and the display will be handled for you.

You will receive 30% of the total number of points if you implement one type of curve correctly, and 15% for the other type (since you can use one curve type to implement the other using a simple transformation).

2.4 Surfaces (50% of grade)

Your program must be able to generate and display surfaces of revolution (of a curve on the xy-plane around the y-axis) and generalized cylinders. It must compute surface normals correctly. To demonstrate this in your program, you should have two modes of display. One mode should display the surface is drawn in wire-frame mode with the vertex normals drawn pointing outwards from the surface. The other should display the surface with smooth shading. The display functions are already implemented for you in the starter code. To generate surfaces, you can fill in the `makeSurfRev` and `makeGenCyl` functions in `surf.cpp`. Note that, for generalized cylinders, 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.

You will receive 20% of the total number of points for mesh generation of each type of surface (surface of revolution and generalized cylinders). For proper computation and display of normals, you will receive 10%.

2.5 Artifact (5% of grade)

Finally, you are to use your program to create at least two complex geometric models, which are substantially different from those distributed with the assignment. They can model real-life objects or they can be abstract forms. Also note that the SWP file format supports multiple surfaces, so you may wish to exploit this functionality.

In addition to the two or more SWP files, you should submit screenshots of these shapes in either PNG (preferred), JPEG, or GIF format. To take a screenshot on an Athena system, type `add graphics; /mit/graphics/bin/xv`, hit the *Grab* button, and read the directions. Alternatively, you may create your artifact by exporting your surfaces and rendering them using other software. The sample code provided can export your surfaces to OBJ files by passing in a prefix as the second argument on the command line. It will produce with one file per named surface with the prefix prepended (e.g., `./a1 swp/weird.swp foo`, will create `foo_weird.obj`).

3 Starter Code

To build the starter code, type `make`. This will compile a barely-working version of the application into `a1`. Go ahead and run this on `swp/circles.swp`. The starter code, as-is, is fully functional on this example. If you press `c`, you'll also notice that the coordinate frames are computed for you.

If you try and load any of the other SWP files, the program will complain (and often verbosely; try running the program on `swp/core.swp`). Your job is to replace these complaints with real code. In particular, all the code you should need to write can be placed in `curve.cpp` and `surf.cpp` (the starter code was generated from the solution code by deleting parts from these files only).

You can see what functions you need to implement, as well as what they should do, in those files and the associated header files `curve.h` and `surf.h`. We also encourage you to take a look at the other parts of the code, especially the functions that draw the curves and surfaces. These functions give a very good idea of how to access and modify the various data structures that are used.

The starter code uses the `vecmath` vector library extensively. It provides classes such as `Vector3f`, `Vector4f` and `Matrix4f`, and all sorts of helpful functions that operate on vectors and matrices (not only stuff like addition and multiplication, but dot products, cross products, and so on). Look through some of the starter code to see how it is used, and also look at the individual header files in `/mit/6.837/public/include/vecmath` (it's not too long). The source code is also available on Athena via `git` by typing:

```
git clone /mit/6.837/public/vecmath.git
```

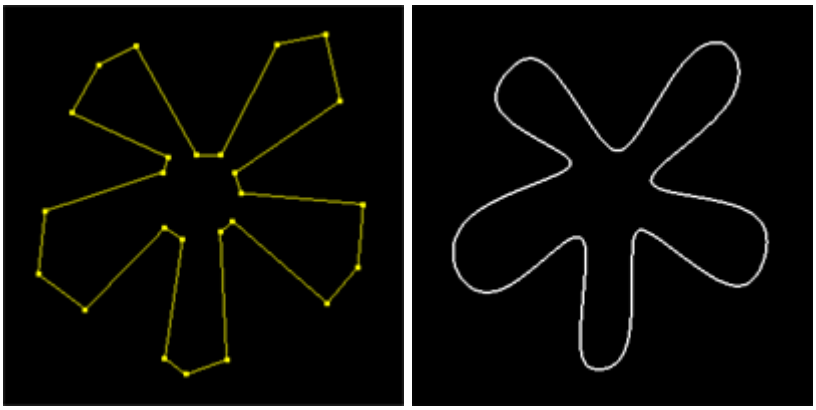
This code also uses the `vector` object from the C++ Standard Template Library. There are plenty of examples in the starter code that demonstrate how they're used, but if you'd like to have a gentler introduction, there is a tutorial here and a full (but less comprehensible) description here.

Note: If you get compile errors, specifically 'error: GL/glut.h: No such file or directory', type:

```
sudo aptitude install freeglut3 freeglut3-dev
```

4 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).



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.

4.1 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 infection 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.

4.2 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):

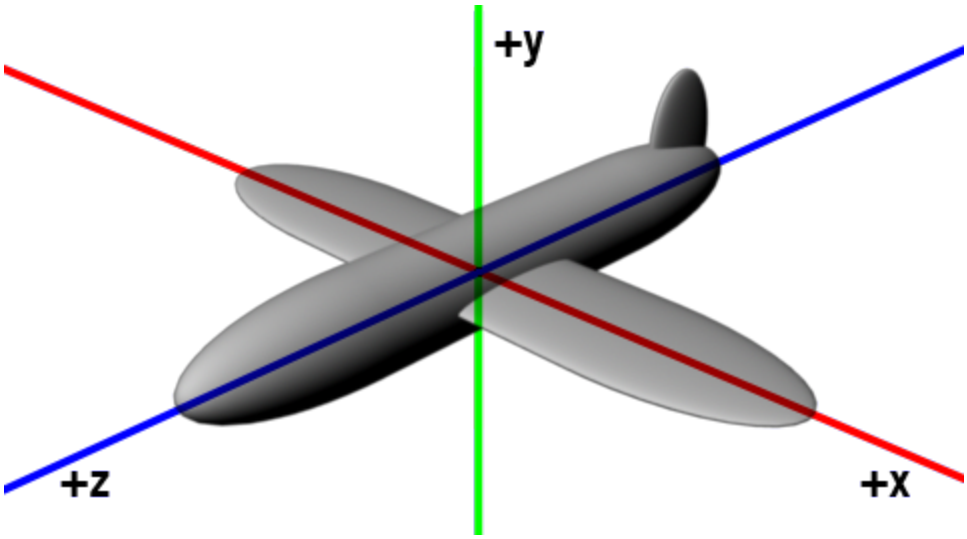
$$\begin{aligned}\mathbf{V}_i &= \mathbf{q}(t_i) \\ \mathbf{T}_i &= \mathbf{q}'(t_i).\text{normalized}()\end{aligned}$$

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:

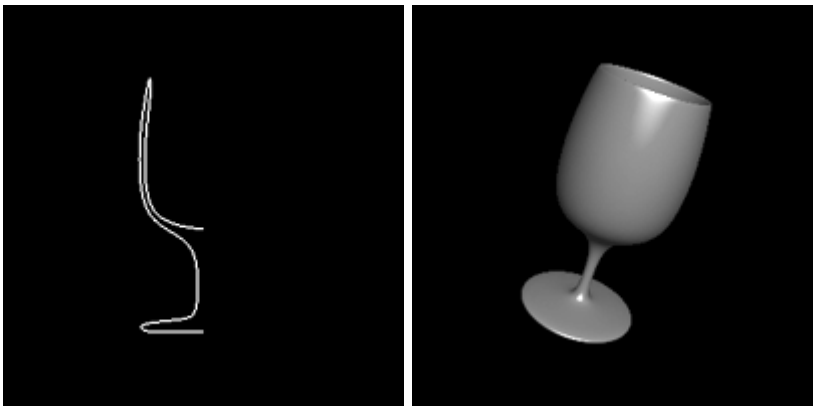
$$\mathbf{M} = \begin{bmatrix} \mathbf{N} & \mathbf{B} & \mathbf{T} & \mathbf{V} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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.

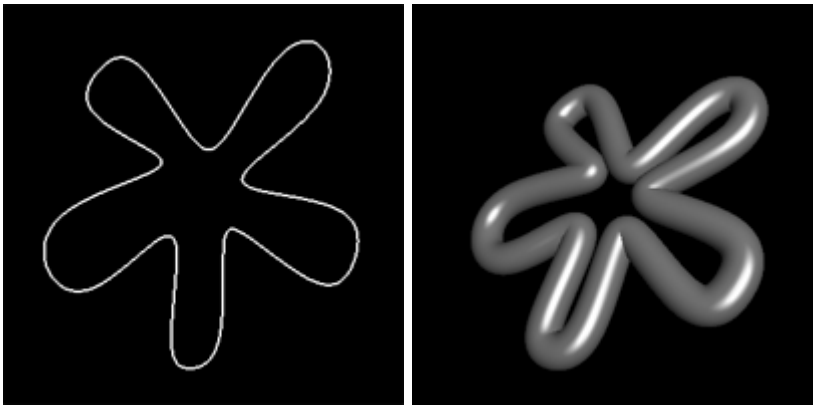
5 Implementing Surfaces

In this assignment, you will be using the curves that you generate 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.



5.1 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.

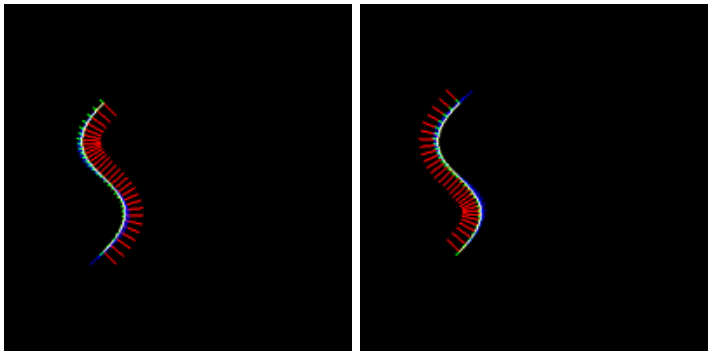
Suppose that you have already evaluated the control points along the profile curve. Then, you can generate the *vertices* of the surface by simply duplicating the evaluated curve points at evenly sampled values of rotation. This should be your first task during the implementation process.

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 **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 **M**, its *normal* should be transformed by the *inverse transpose* of the top-left 3×3 submatrix of **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).

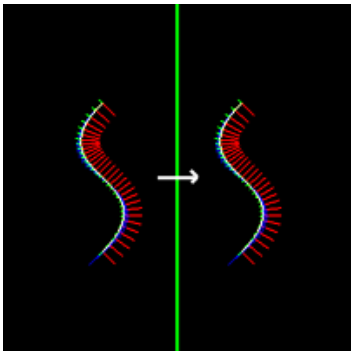
Another thing that you'll need to worry about is the orientation of the normals. For OpenGL to perform proper lighting calculations, the normals need to be facing *out* of the surface. So, you can't just blindly rotate the normals of any curve and expect things to work.

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.



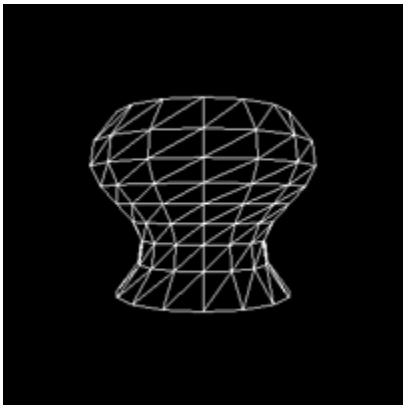
In this assignment, we will assume that, for two-dimensional curves, the normals will always point to the *left* of the direction of travel (the direction of travel is indicated by the blue lines). In other words, the image on the left is correct. This is to be consistent with the fact that, when you're 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.

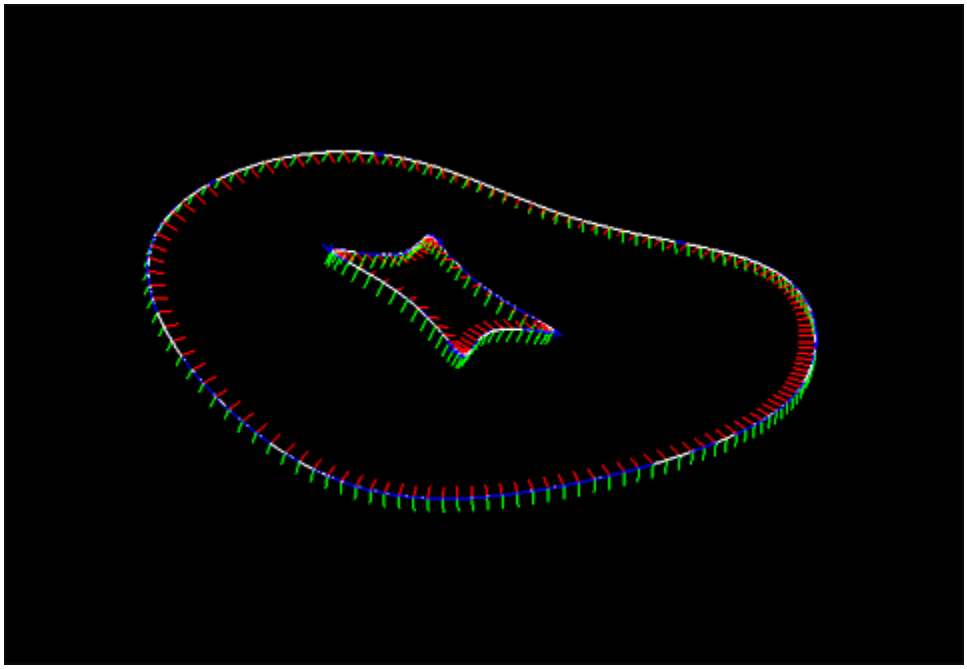
5.2 Generalized Cylinders

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.

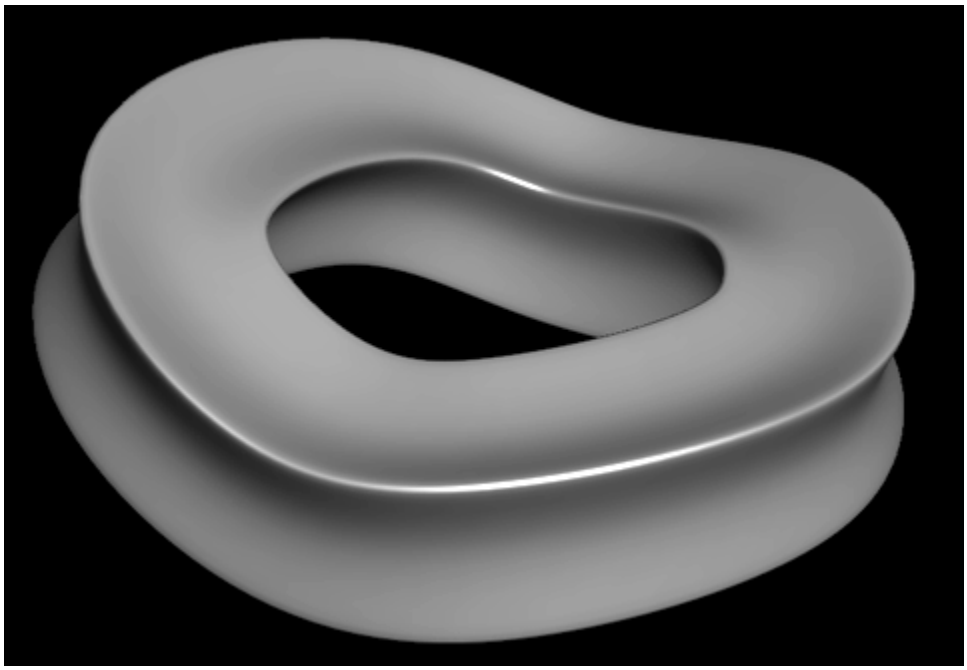
Just as with surfaces of revolution, each copy of the profile curve is independently transformed. With surfaces of revolution, we used a rotation. With generalized cylinders, we will use the coordinate system defined by the **N**, **B**, **T**, and **V** vectors of the sweep curve. To put it in the context 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.



6 Extra Credit

As with the previous assignment, the extra credits for this assignment will also be ranked *easy*, *medium*, and *hard*. However, these are all relative to the individual assignment; an easy extra credit for this assignment will probably be will be harder than an easy one from the last assignment (and we will assign credit accordingly). Furthermore, these categorizations are only meant as a rough guideline for how much they'll be worth. The actual value will depend on the quality of your implementation. E.g., a poorly-implemented medium may be worth less than a well-implemented easy. We will make sure that you get the credit that you deserve.

If you do any of these extra credits, please make sure that your program does not lose support for the SWP file format. We will be using these files to evaluate your assignment.

6.1 Easy

- Render your model more interestingly. 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.

- Implement another type of spline curve, and extend the SWP format and parser to accommodate it. 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.

- Implement another kind of primitive curve that is not a spline. 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.

- As mentioned before, the suggested method of computing coordinate frames has a problem: if a curve is closed, the coordinate frames are not guaranteed to line up where the curve meets itself. The sample solution fixes these issues as follows. First, it detects when these errors occur (when the positions and tangents are identical but the normals are not). Then, it interpolates the rotational difference between the start and end of the curve, and adds these to the coordinate frames along the curve. Implement this solution in your code. To check your solution, you should be able to display a seamless solid for `weirder.swp`. You should also ensure that your solution does not affect the results for curves that are not closed.

- 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, and provide some SWP files that demonstrate the correctness of your implementation.

6.2 Medium

- Implement a user interface so that it is easy for users to specify curves using mouse control. 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). If you choose to implement this, you may want additional user interface widgets such as menus and sliders. We recommend Qt for this.

- 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. See Buss (Chapter VII.3) for details on one way this can be performed.

- 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 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 *birail surface*. Birail surfaces sweep a profile curve along *two* sweep curves which control the scale and orientation of the profile curve. Specifically, the two profile curves (the rails) are attached to control points on the sweep curves. The profile is the swept along both of the rails, while keeping the control points on the rails using scale and rotation.

- Implement piecewise Bezier and B-spline surfaces. 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.

6.3 Hard

- The techniques covered in this assignment are just one of many ways that modelers create shapes. Another popular technique is that of *subdivision surfaces*. At a high level, you can think of this technique as starting with a polygon mesh and defining the desired surface as the limit of a set of subdivision rules that are applied to the mesh. Implement this technique. You may find this helpful. Also note that, if you plan to take 6.839, you'll have an advantage, since this is one of the projects in that class.

6.4 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, as it is available to you on Athena. Since 6.837 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.

7 Submission Instructions

As with the previous assignment, you are to write a `README.txt` or optionally a `README.html` that answers the following questions:

- How do you compile and run your code? Provide instructions for Athena Linux.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.
- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe how you did it.
- Do you have any comments about this assignment that you'd like to share? We know this was a tough one, but did you learn a lot from it? Or was it overwhelming?

As with the previous assignment, you should submit your assignment using Stellar. Your submission should contain:

- Your source code.
- A compiled executable named `a1`.
- The `README.txt` file.
- At least two SWP files that define your new artifacts.
- Images of your artifacts in PNG, JPEG, or GIF format.

Assignment One is due September 28 at 8:00pm. We will NOT accept late submissions.

8 Helpful Hints

- Get started early. This assignment does not need to involve too much additional code (the sample solution adds about 130 lines to the starter code), but it is remarkably easy to make mathematical and logical errors that are very difficult to debug.
- Use features of the interface to debug your code. For instance, by toggling through the curve drawing modes, you can see the coordinate systems that are computed at each evaluated point and check if they make sense. If your surfaces don't look right, use the wireframe mode to check whether the normals are pointing outwards and if your triangulation is correct.
- Exploit code modularity. B-spline control points can be converted to Bezier control points via a matrix multiplication, so you can reuse your Bezier code to generate the appropriate coordinate frames. This also makes implementing other types of splines (e.g., Catmull-Rom) relatively easy.
- Look at the headers in the `vecmath` vector library. It's brief, and the library provides numerous functions that you will find indispensable. You can just call `cross`, rather than writing it yourself and worrying if you did it right.
- Study the starter code. It will give you some insight into how to access the relevant data structures, and there are numerous helpful comments throughout the code. Furthermore, you may find certain parts of code helpful. For instance, the code that draws the local coordinate frames on curves is quite similar to the code that you'll want to use for generalized cylinders.
- If you are working on your own computer and you want to use the starter code, you'll probably need to recompile the `vecmath` vector library for your system. The source code is available on Athena via `git` as mentioned earlier. However, this is the extent of support that we will provide. 6.837 exclusively supports Athena Linux; if you choose to work on your own system, you're still responsible for ensuring that your code runs on Athena.
- You can test curves and surfaces independently by using the provided circle primitives. Take a look at `tor.swp`, which draws a torus as a swept surface without using any spline curves.
- Get started early. We know that we're repeating ourselves, but we want to emphasize that this is a much more difficult project that the previous one.