Sketch-Based Rendering from Few Strokes in Orthographic Views

Yassin Bayoumy, Emil Hodzic-Santor, Aly Khedr

April 13th 2023

Introduction

Our project details the creation of a sketch-based interface that allows users to quickly and intuitively create 3D models from 2D sketches and then export these models as a .obj file, which can be used in applications such as 3D printing.

Creating 3D models is generally very time-consuming and artistically demanding. As a result, 3D modelling has become less accessible to a large number of people. While most people cannot create highly detailed 3D models, nearly everyone knows how to draw simple sketches; even those that can't draw well. Creating sketch-based 3D modelling interfaces and improving on existing ones helps make 3D modelling more accessible to the general public.

The proposed sketch-based interface will be implemented using a variety of blending and deformation techniques to allow the creation of 3D models using simple 2D line sketches. In each of the following sections, one of the important methods involved in creating our program is described. We have broken down each section into three parts - the *objective* (what was required and why), the *method and outcomes* (what we did to implement it, and how it works), and the *obstacles* (any challenges or consequences we encountered, and how we solved them).

Requirements

The implementation of our program was created by adapting the 'CPSC 589 3D Boilerplate' program on Windows in C++, using OpenGL. In particular, our program makes heavy use of the `glm` library.

Specific usage instructions of our program are covered in the `README.md` file that comes with the program code.

Stroke Capture

Objectives: Before any 3D modeling takes place, we first need a method of capturing strokes of a user's cursor. It is important that the strokes are accurate to the user's input, efficient, and modifiable. For each angle that the user is drawing in, the program should capture the input of the user as if it was being drawn on the plane that the user is facing (ex. if the user is viewing the XY plane, points should be drawn in XY plane, meaning the z-coordinate of each point should be 0).

Method and Outcomes: To capture our intended strokes, we take the current mouse input coordinates, which will always be comprised of just an X and a Y component and apply the following algorithm:

1. Multiply our input by a 'perspective multiplier' that is equal to the the tangent of half of the FOV (in our case this is $\pi/4$) times the camera's radius.

2. Subtract the camera's radius from this value, essentially pushing back these points to the plane that we are drawing on.

3. Apply the inverse of the view matrix to this points to get the desired point.

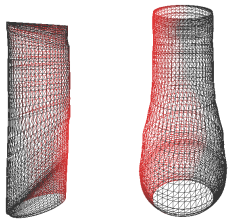
This method provided our program with the one point of input for each frame of the program. While this is accurate, it is neither efficient to store nor easy to modify. To reduce the number of points to store, we apply the reverse Chaikin algorithm, as suggested in class and 'Sketch Based Modeling with Few Strokes' [1]. This method obtains the control points of the curve as if it was a B-Spline curve, undoing one iteration of a traditional Chaikin algorithm. We apply two iterations of the reverse Chaikin to our points, as we found this was generally a good mix between having a decently small number of control points, while maintaining the shape of the original curve.

With these control points, we found a quadratic B-Spline curve (has to be quadratic - since that corresponds to the reverse Chaikin algorithm), which is a close approximation of the initial sketch. For efficiency, we store the control points in a new class called `Mesh`, as these points will be what define the initial curves of the mesh.

In an editing mode, we allow the user to click and drag each of the control points. Using the locally modifiable property of B-Spline curve, the movement of these points results in changes only where the B-Spline curve is defined. Additionally, we allow the user to reduce/increase the number of control points for more detailed/less detailed control of the curve they drew. This simply applies the Chaikin algorithm and reverse Chaikin algorithm to increase and decrease detail, respectively.

Obstacles: Since the above stroke capturing method would store a new point every frame, we quickly found that the density of points in the line was inconsistent. Depending on the speed that the user would draw the stroke, the density of points in the line would greatly differ. This density differed from line to line but, more importantly, also differed at different sections within the same line. Applying the Reverse Chaikin and BSpline algorithms to the line did not resolve this density problem.

As a result of having an inconsistent density of points throughout and between lines, When two lines were being used to create a rotational blending surface, it would result in some abnormal meshes that would bulge out. We noticed that these results were inconsistent and that two similar sets of lines could produce different results depending on the speed at which they were drawn. An example of these anomalies is pictured below:



To resolve this, we used some linear interpolation to ensure a consistent density of points when a line is being drawn. We defined a 'pointEpsilon' parameter that we set to 0.1. This defines the distance of neighbouring points on a line. Afterwards, on every frame when drawing a line, we would linearly interpolate between the last point in the line being drawn and the current cursor position, then insert points that are a 'pointEpsilon' distance apart. This ensured that every point in a line was always 'pointEpsilon' distance away from its neighbouring points and resolved the density problem we encountered.

Rotational Blending

Objectives: Using the two curves that were drawn by the user, we wish to create an object that interpolates the two curves, but exists in all three dimensions rather than just the two which the curves were sketched in.

Method and Outcomes: In order to take two curves drawn by a user and create a mesh, we used the rotational blending technique that was presented in lecture. To briefly recap, that is:

1. Given the two co-planar, user created curves, denoted $q_l(u)$ and $q_r(u)$, a center curve is found as $c(u) = \frac{1}{2}q_l(u) + \frac{1}{2}q_r(u)$.
2. For any value u_i we find a disc $t_{u_i}(v)$ that is centered at $c(u_i)$ and passes through $q_l(u_i)$ and $q_r(u_i)$. Starting with the unit circle that is defined in the axes of the current up direction (we call this 'up') and the vector that the camera makes with the origin (we call this 'out'), the disc is obtained with the following procedure:
   (a) Scale the unit circle by $\frac{1}{2}|q_r(u) - q_l(u)|$ (the size of the diameter between the two user created curves).
   (b) Rotation about the 'out' axis by the angle that $q_r(u) - q_l(u)$ makes with the 'up' axis.
   (c) Translation by $c(u_i)$.

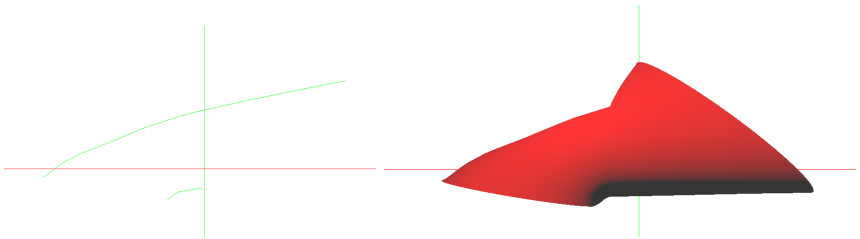
We then create a triangular mesh that joins the discs together, and we close the mesh with a flat surface at the top and bottoms of the mesh.

Obstacles: For the most part, the process of rotational blending worked exactly as planned. However, there were some cases that we needed to account for to ensure that it worked every time. First was the case that the user would draw one line going in one direction (say left to right), and another line going the opposite direction (right to left).

If our program simply iterated through the values of u in the curves $q_l(u)$ and $q_r(u)$, the resulting mesh would be created by drawing discs that go across the object, with an axis that is much smaller than the object itself. This results in a mesh that intersects itself, and in almost all cases that this occurs, it is not intended by the user.

To fix this, we wrote a quick algorithm that checked the endpoints of each curve and found which ends of the curve were closest to each other. If the end points that were closest to each other already corresponded to both being the beginning or the end points of both curves, we leave as is. Otherwise, we reverse one of the curves so that this correspondence is true.

Another case we experience is seen below, where in some cases the surface would leave the bounds that were drawn, either becoming much wider or much thinner than intended.



This error was a consequence of calculating the angle θ using the dot product method, that is find θ between vectors a and b as:

$$\theta = arccos\left(\frac{a \cdot b}{||a|| \cdot ||b||}\right)$$

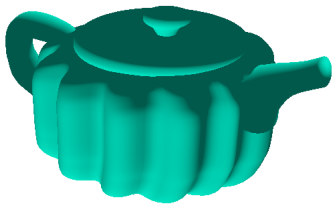
The issue with this method was that it did not count for orientation/direction. The reason that the left and right side of the object are rendered the same is that the dot product method finds the angle between the leftmost and rightmost disc to be the same, except it is measuring one of these angles from the positive y axis, and one from the negative y axis.

To ensure some consistency in the calculation of these angles, we instead chose to compute them using the `glm::orientedAngle()` function, which measures the angles with respect to an axis. In our case, this is the viewers position.

Cross-Section Editing

Objective: As we saw in the rotational blending method, the shape is constructed from a unit circle lying in the plane of the 'up' and 'out' vectors. To

allow the user to draw more detailed and complicated shapes that do not just follow a smooth, round profile or cross-section, we allow the user to draw a new cross-section, or edit the existing cross-section, that will replace this standard unit circle. A desired example of a teapot that had a modified cross section is seen below.

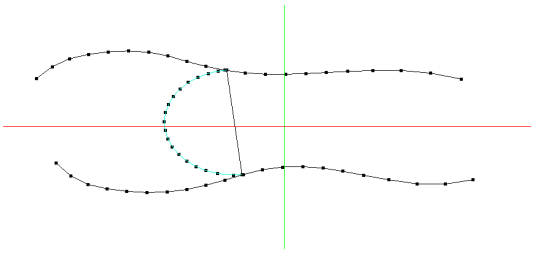


Method and Outcomes: Our method has two main methods - edit an existing curve or draw a new curve. Starting with drawing a new curve, we create an equal number of vertices on each curve that the user drew initially, and task the user with sketching a new cross-section between any vertex on one of the curves and any vertex on the other curve. To avoid any strange outputs, we only accept input that is drawn this way. If a user begins a line at one of the curves and stops holding the right mouse button before reaching the other curve, the line is canceled and deleted.

Once a line has been drawn between two points, we allow the user to edit it exactly as we did in the original curve sketching method. We draw a vector between the start and end points of our cross-section sketch, which will be our new diameter of the cross-section. Using this diameter vector, we find the point that is at the center of the diameter and use it to translate our sketch to the origin. Next, we rotate the sketch with respect to the 'up' vector so that it is no aligned with it. Finally, we mirror the sketch to obtain a closed sketch (like a circle or oval) and then scale it so that the height of the sketch is 2, corresponding to the unit circle.

To edit an existing cross-section, there are two possibilities - either we have already sketched a new cross-section, for which we can simply retrieve the points we had drawn previously. The other possibility is that we are editing the initial cross section, which is the unit circle. So, we retrieve the unit circle drawn in the necessary axes, and discard half the points to get a half-disc.

Next, we find the middle points of each curve that the user drew. These points will be where we display the cross-section. To get our cross section to this location, we scale it by a factor of $\frac{1}{2} * d$, where d is the diameter between these two points. Then we rotate it by the angle between the 'up' vector and this vector, and translate it to the center point of the vector. The resulting editable half-disc is seen below:



Once the user has applied their desired edits, the coarse points are sent to the `mesh` class to re-render the mesh.

Obstacles: Similar to cross-section editing, we initially had an issue with the rotation of the cross section line before it is sent to the `mesh` class. The user-drawn cross-section would not be oriented correctly, and would thus result in an odd looking object, far from what the user desired. However, switching to the `glm::orientedAngle()` method resolved this issue.

Additionally, introducing a cross-section that was no longer the unit circle meant that we were required to switch to calculating the normals of the vertices of the mesh via cross product, rather than taking the vector from the point on the disc and the center of the disc, which can only be done for a circular cross-section.

Profile Curve Modification

Objective: Using the original sketches and the cross-section modification allowed us to produce very accurate symmetric objects. From above, the positive z direction would mirror the negative z direction. However, not all objects have this property.

Take the example of a shoe or boot, if viewed from the top, one side of the object will have a flat edge, and one will have a gentle curve. This problem resulted in the creating of a new method, which we called the 'Pinch Method' for simplicity.

Method and Outcomes: To change the profile curve of an object, the user would need to define two new curves that are orthogonal to the to the original curves, and run parallel to the center axis that we defined in the process of rotational blending.

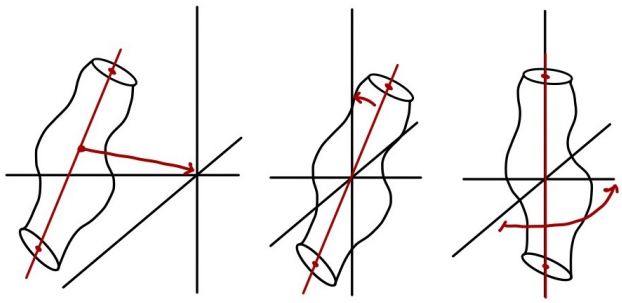
Initially we discussed drawing the profile curves exactly onto the axis of the object, but since the axis will typically be curved, this would be difficult to accomplish as the user would need to draw from multiple different angles. Instead, we opted get a simpler axis of the object by taking the the vector that ran from the first point of the axis to the last.

With this axis, which was a straight line by definition, the user would need to draw two new profile curves that ran parallel to it. This is where we ran into yet another difficulty. In some cases, the object in question would have two suitable axes to draw the profile curves, whereas in others, the object would have

only one axis from which this was possible. For our application, we wanted a consistent and repeatable process. So, we implemented to following procedure for any time a user wanted to draw new profile curves:

1. The simplified center axis of the object is found as `axis[0] - axis.back()`. Let us denote this axis as `c_axis`.
2. Find the angle between `c_axis` and the current up vector, relative to the camera's position using the `glm::orientedangle()` function, similar to how we calculated the necessary angles for rotational blending. Let us denote this angle as `theta`.
3. Rotate the entire mesh by `-theta` with respect to the position of the camera. This will align `c_axis` with the current up vector.
4. Rotate the camera by $\pi/2$ to view the mesh's axis.

This process is visualized below:



Now that the user has the mesh aligned with current up vector of the camera, and we are orthogonal to the original curves that were sketched, the user can draw the profile just as the initial curves were drawn. To redraw the mesh, there is now a second pair of curves that the program must consider. To form the mesh, we use a similar technique to rotational blending, however we now scale each disc in two directions. For each vertex along the axis of the object, the program takes the value that along each of the new profile curves that is closest to this vertex, with respect to whichever coordinate corresponds to the current up vector. For example, if the current up vector is the y direction, the program chooses the vertex along each of the profile curves that have the most similar y-coordinate to the center axis.

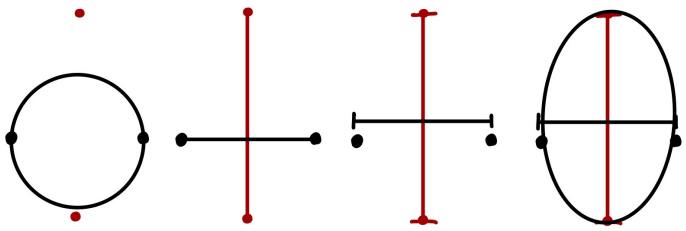
With these two corresponding vertices, the process of rotational blending can be modified to scale each disc in this new direction to match the new profile curves. The additional steps to perform this scaling are:

1. Let us denote the vector between the original curves as `diameter` and the vector between the new curves as `pdiameter`.

2. We find a midpoint for our new disc using the current 'out' direction as the component for `diameter` and original 'out' direction (which the initial curves were drawn in) as the component for `pdiameter`.

3. Translate a disc to this new point. We then scale the disc using the same method as we did to find the center, but instead scaling by $\frac{1}{2}$`glm::length(diameter)` and $\frac{1}{2}$`glm::length(pdiameter)` in their respective directions.

This process is shown below in one cross-section of the mesh, with original curves illustrated by black dots and the new profile curves illustrate by red dots.



This new disc does it's best to maintain the dimensions of both the original curves and the new profile curves. We should also note that we did not have to rotate the disc before matching the points on the profile curves. This is because the rotation of the disc does not actually change the 'up' coordinate which we are matching the coordinates with.

We have now 'stretched' or 'pinched' the mesh in the direction of the new profile curves. All we have to do is reverse of what we did to align the mesh with the axis, so that the mesh and the camera have been returned to their original states.

Obstacles: It is important to note that the program stores these profile curves in the `Mesh` class as they were drawn, not rotating them to match the object's axis. Because of this, if an existing mesh is being modified, and said mesh has existing profile curves, this process will have to be repeated to add the profile curves to any other modification that was made. That is, if a user is editing the initial curves of a mesh that already had it's profile curves edited, the mesh must be recreated with these new initial curves, and then again be aligned with the up, vector, before the profile curves are applied and then the mesh is returned.

If the user wishes to edit the profile curves of a mesh that does not have pre-exiting profile curves, the program does it's best to estimate these curves by taking the coordinates of the edges of the mesh that are closest to the plane the user is viewing. For a standard rotational blending surface, before any cross-section modification has happened, these points are just the points on each disc that lie at the start of the disc (`disc[0]`), and halfway through the disc (we approximate this point `disc[floor(disc.size()/2)]`).

Additional issues arose when taking into account the cross-section modification, since the width of the disc is now no longer standard to the unit circle, like the original object created via standard rotational blending would be. To account for this, we changed the `create` function in the `mesh` class to measure the distance between points across the cross section shape and then recalculate the height and width of the cross-section.

Object Selection

Objectives: To enable the editing of existing objects in an intuitive and user-friendly way, object selection was needed. This needed to be efficient to keep the user experience responsive.

Method and Outcomes: We implemented this feature using the provided object selection tutorial. The implemented method of object selection utilizes a separate frame buffer to take advantage of OpenGL's depth buffer calculations. The following steps were taken to determine which object is being hovered:

1. Create a second OpenGL frame buffer that is made up of a texture and a depth buffer.
2. Render the scene onto this new frame buffer. However, instead of rendering in colour, render in the object indices.
3. Get the current cursor position and extract the corresponding pixel's object index.

Exporting

Objectives: Once the user has created a model they are satisfied with. We wish to make this model usable by exporting it to a `.obj` file. This can then be used as part of graphics software such as a game or for 3D printing purposes.

Method and Outcomes: The export to `.obj` functionality that we implemented works as expected. Since we knew that this was a requirement of our software, we designed the `Mesh` class to use indexed geometry. On top of making rendering to OpenGL more efficient, the use of indexed geometry made the export to `.obj` functionality simple.

The `.obj` file that gets exported is comprised of a set of vertices (`v`), a set of normals (`vn`), and sets of faces (`f`). The sets of faces are grouped using the `g`, based on the different objects created in our software.

Obstacles: Since the `Mesh` class was already designed with the `.obj` format in mind, there were no problems with implementing this functionality properly.

However, there are two issues that should be mentioned:

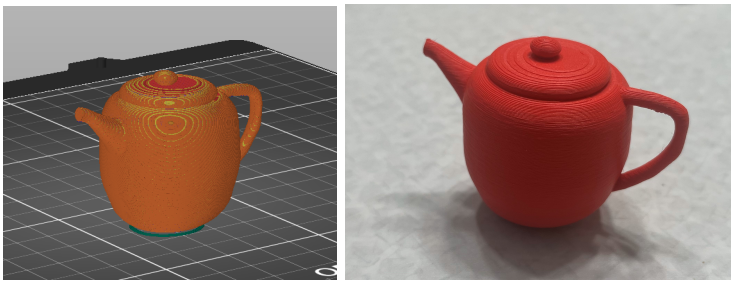
1. When exporting a large object with many vertices, the program freezes while it is exporting to the `.obj` file. While this is not detrimental to the functionality, it slightly worsens the user experience.

2. The `.obj` file location is sometimes hard to find. This was circumvented by adding a popup that told the user where to find the file after it had been exported.

3D Printing

Objectives: Now with the `.obj` file format, we wish to open our object in a 3D printing software and print it.

Method and Outcomes: In most cases, the `.obj` can easily be opened in a printing software such as Prusa3D, that slices the object and can then be printed using a 3D printer. Below is an image of a teapot we drew in the software in Prusa3D, and the resulting 3d-printed object.



Unfortunately, since our software does not support calculating mesh intersections there are cases in which we do not have immediately have a singular, watertight mesh. At times, this can lead to unintended results from the slicing software. If these results are encountered, the 3D printing software that is provided by Microsoft, *3D Builder* allows us to 'repair' these issues and either order from a third party printing company, or export the repaired mesh in a variety of file formats such as `.stl, .obj,` or `.3MF`, which are all capable of being 3D printed.

Gallery

We have included some interesting results that our application produced:

Figure 1: A plane being constructed in the software (left), and a resulting 3D-printed plane (right)

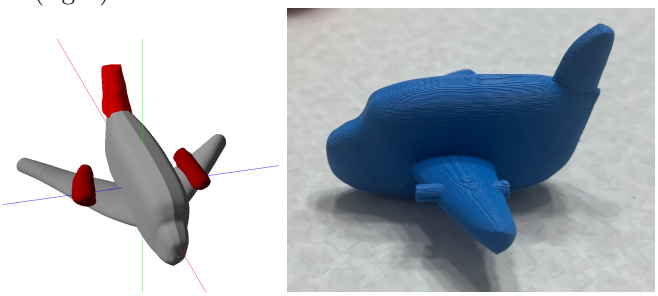


Figure 2: An application of the 'profile curve modification' technique, shown by (1) the initial angle, (2) the orthogonal angle, (3) the edited orthogonal angle, and (4) the resulting view from the initial angle.

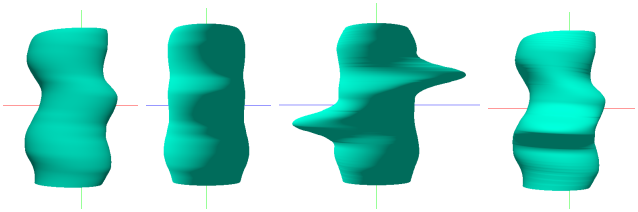
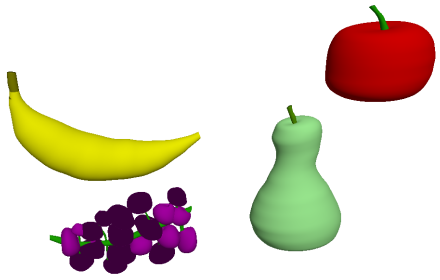


Figure 3: A selection of various fruits displayed in the same plane, showing the multiple object capability of the program.



Conclusion and Next Steps

In the end, our project was able to accomplish some of the main goals we set. With little instruction, a user can draw a sketches that can be quickly transformed into an accurate 3D rendering of their sketch. With our provided

tools for modification, the user is able to change the finer details of the object that were not defined by the initial sketch. Finally, the user has the option to export their creation in the form of a `.obj` file, and with a little help from external software (for the cases that involve object intersection), this object can be 3D printed.

To improve the results of our project, we suggest the following additions to our program could be made:

1. Object translation and rotation. This would allow to create a more detailed object in just one angle, particularly for objects that require layers of shapes (for example, a face, that requires features like a nose, mouth, or eyebrows to be in front of the face).

2. Subdivision surfaces. A subdivision scheme such as Catmull-Clark subdivision would allow the user to easily smooth the mesh to remove unwanted imperfections or details.

3. Mesh intersection. Providing a method to calculate the intersection between meshes, and combine them into one mesh would remove the need for the user to use software such as Microsoft 3D Builder in order to obtain a singular mesh that could be 3D printed. This sort of implementation may required switching much of how the rotational blending surface is calculated, as the mesh intersection methods that were introduced in class relied on the areas in question being implicitly defined, so that it is easy to determine if a point is inside or outside of it.

References

[1] Cherlin, Joseph Jacob, et al. "Sketch-Based Modeling with Few Strokes." Proceedings of the 21st Spring Conference on Computer Graphics, 2005, https://doi.org/10.1145/1090122.1090145.

[2] Olsen, Luke, et al. "Sketch-Based Mesh Augmentation." Proceedings of the 2nd Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBIM'05), 28 Aug. 2005, pp. 43–52., https://doi.org/10.2312/SBM/SBM05/043-052.

[3] Olsen, Luke. "A Taxonomy of Modeling Techniques Using Sketch-Based Interfaces." Eurographics 2008 - State of the Art Reports (STARs), no. 1017-4656, 14 Apr. 2008, pp. 39–57., https://doi.org/10.2312/egst.20081044.

[4] "The OBJ File Format – Simply Explained." All3DP, 30 Mar. 2023, https://all3dp.com/1/obj-file-format-3d-printing-cad/.

[5] Alias/Wavefront OBJ File Format, https://people.computing.clemson.edu/ dhouse/courses/405/docs/brief-obj-file-format.html