Sketch-Based Modeling in Augmented Reality

Nicole Feng

CS 174, June 2019

Abstract

The aim of this project is to extend sketch-based modeling into augmented reality (AR). In contrast to traditional computer-aided design (CAD) where modeling is done via a set of predefined primitives, sketch-based modeling turns 2D sketches into 3D models. Using sketches as input results in a more intuitive interface and a lower learning curve for the user, and is useful for the earlier stages of design where the user may not know exactly where in 3D space they want objects to go, but perhaps want to quickly explore a 3D rendition of a rough sketch. For this project, I use the Teddy system (Igarashi et al. 1999), which uses an inflation-based approach, to create meshes from sketches. The result is an Android app, called "TeddyAR", developed using the Unity game engine and Google ARCore augmented reality SDK, that turns user-drawn 2D sketches into triangular meshes that they can then place in their current context.

The project implementation can be found at https://github.com/nzfeng/TeddyAR. Presentation material can be found at https://drive.google.com/drive/folders/1j1L6u3O_p7O8mB1ARDwWcoRzWMH7Eegw?usp=sharing. Questions can be directed to nzfeng@caltech.edu.

Mathematical Representation

The original *Teddy* paper states the following recipe for inflation:

1. Take a closed 2D polygon as input.
    - Resample and make all edges some unit length.
2. Determine the chordal axis of the polygon.
    - Triangulate the polygon using constrained Delaunay triangulation (CDT).
    - Obtain the chordal axis by connecting the midpoints of internal edges.
    - Prune insignificant branches using fanning procedure.
3. Inflate the mesh.
    - Elevate each vertex on the spine by a distance proportional to the average distance between the vertex and the boundary vertices directly connected to it.
    - Convert each internal non-spinal edge to a quarter oval, and elevate along with the spine.
    - Sew together the neighboring elevated edges.
    - Copy the elevated mesh to the other side to make the mesh closed and symmetric.

The general idea is to find the "skeleton" of the drawn polygon, and "inflate" it along this local axis of symmetry to produce a closed, symmetric object.

Figure 1: A visualization of the idea behind Teddy's inflation algorithm. (I apologize for the rightmost image, that's supposed to be my artist's rendition of a 3D bean shape.)

The Teddy algorithm tends to result in blobby, free-form shapes.

Below I describe the rationale and role of each step.

Constructing the polygon spine

Loosely speaking, the polygon "skeleton" is the "frame" on which all of the "flesh" will hang. Intuitively, it represents the local axis of symmetry. One way to think about this is if we try to estimate the shape of the polygon by fiting a bunch of circles into the interior; the points at which a circle is tangent to the polygon boundary can be considered the "symmetric" points, while centers of the circles would form the skeleton "axis of symmetry." This is the motivation for the *medial axis transform* (MAT).

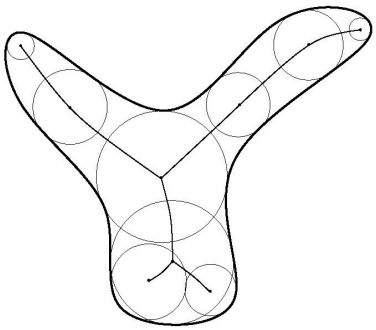


Figure 2: Example of constructing the medial axis transform. Figure from here.

The medial axis of an object is the set of all points having more than one closest point on the object's boundary. We can also define it in terms of *maximal discs*, which are circles contained in the interior of the object that are tangent to its boundary at at least 2 points. (The medial axis transform technically also includes the associated radii of the maximal discs.) However, the MAT tends to be sensitive to small variations in the boundary, result in many spurious branches, and take a long time to compute. In addition, the MAT may not exist for polygons.

Instead of considering the midpoints of diameters (centers of circles) along with half the lengths of diameters (the radii of circles) of the maximal discs, we could also consider the midpoints of chords along with half the lengths of those

chords. This is the motivation for the *chordal axis transform* (CAT). Specifically, a *maximal chord of tangency* of a maximal disc is a chord that

1. connects two points of tangency (of the disc with the shape boundary)
2. at least one of the two arcs subtended by the chord as no points of tangency

The second condition can be thought of as ensuring that the location of a chordal axis point is determined by its two inducing boundary points alone, so that the parts of the skeleton that reflect features of the shape actually tend to lie close to those features.

The CAT is then defined as the set of all pairs (p, d) where either

1. p and d are the midpoint and half the length, respectively, of a maximal chord of tangency, or
2. p and d are the center and radius of a maximal disc with three maximal chords of tangency that form an acute triangle.

(The second case results in so-called junction triangles, which will be discussed below, and represent the junction of different "branches" of the polygon.)

The CAT has a number of certain properties and advantages over the MAT, although the original paper that introduces the CAT omits the proofs "for sake of brevity" and I haven't had the time to attempt a proof.

In the discrete case (i.e. a polygon as a discrete shape), maximal discs are replaced by *empty circles*, which are circles that pass through exactly three polygon vertices, and do not contain any other vertices in its interior. Maximal chords of tangency are replaced by the line that joins two non-neighboring vertices of the polygon if and only if an empty circle also passes through both of these vertices.

It turns out that this construction is equivalent to implementing a constrained Delaunay triangulation (CDT). (A Delaunay triangulation is a decomposition of a polygon into triangles such that the circumcircle of each triangle is empty.) There are many types of algorithms for implementing Delaunay triangulation. For example, one can be obtained by an edge-flipping algorithm, where we first construct any triangulation of a given set of points, and flip the common edge of two adjacent triangles to one that connects the two unshared vertices instead, until a Delaunay triangulation is obtained; or we can use an iterative algorithm, where each point is added iteratively and re-triangulating with each addition. All these algorithms rely on a fast way to determine if a point lies within a circumcircle, usually done by partitioning the space so we don't have to check every single point for each test triangle. A constrained Delaunay triangulation, where we stipulate that certain edges must be included in the triangulation, can be obtained by first obtaining a Delaunay triangulation, then adding constraints one by one. If the constraint already exists as an edge, then we don't do anything. If it is not included, we delete all triangles that are intersected by the added constraint, then re-triangulate the region.

The resulting triangulation after adding constraints may not be perfectly Delaunay. In practice, resampling the input points to make all edges some unit length before performing constrained Delaunay triangulation somewhat improves the quality of the resulting triangulation by reducing sliver triangles.

Pruning and re-triangulation

After the CDT, we obtain a triangulation that contains three types of triangles: "terminal" (2 boundary edges), "sleeve" (1 boundary edge), and "junction" (no boundary edges.) The CDT essentially decomposes the polygon into branches, whose ends are indicated by terminal triangles and whose body consist of sleeve triangles. The chordal axis is obtained by connecting the midpoints of the interior edges (and midpoints to circumcenters of junction triangles at junctions.)

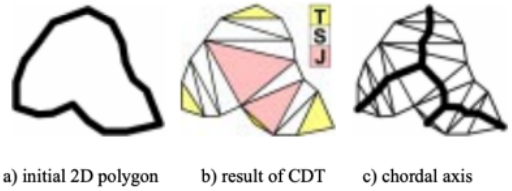


Figure 3: Example of obtaining the chordal axis of a polygon. Image from *Teddy*.

However, we then want to obtain an "inflatable-friendly" triangulation.

To that end, we remove insignificant branches and round the extremities using a "fanning" process. A branch can be considered insignificant if it is "too short", which we can measure as the ratio between the length from the end of the branch (the furthest point on its terminal triangle) and the beginning (the nearest edge on a junction triangle), and the branch width (the length of the corresponding edge on the junction triangle) is below some threshold. In addition, we want to round the ends of each branch by essentially fitting a hemispherical cap. We can combine these steps into the following process:

- Start at a terminal triangle.
- Construct a semicircle whose diameter is the interior edge.
- As long as each vertex is contained within the semicircle, consider the adjacent (sleeve) triangle, and use the interior edge opposite the current edge (the other edge of the adjacent triangle) as the diameter for the semicircle test.
- Repeat this process, expanding into progressively larger regions by merging with adjacent triangles, until either a polygon vertex lies outside the semicircle, or a junction triangle is reached.
  - If the former occurs (i.e. we get to a point where a vertex lies outside the semicircle), intuitively this means that the branch is "longer than it is wide" and there is "room" to fit a circular cap. We take midpoint of the current edge (the edge of the triangle that was absorbed last) as the center around which to create a fan of triangles.
  - If the latter occurs (i.e. we reach a junction triangle), that means we've gotten all the way to the root of the branch without having enough "room" to even fit a circular cap, and the branch is "too short." So we get rid of this spurious branch by taking the circumcenter[1] of the junction triangle as the center around which to create a fan of triangles.

[1]The original *Teddy* paper states using the centroid, but given how the chordal axis is constructed, the correct center to use should in fact be the circumcenter.

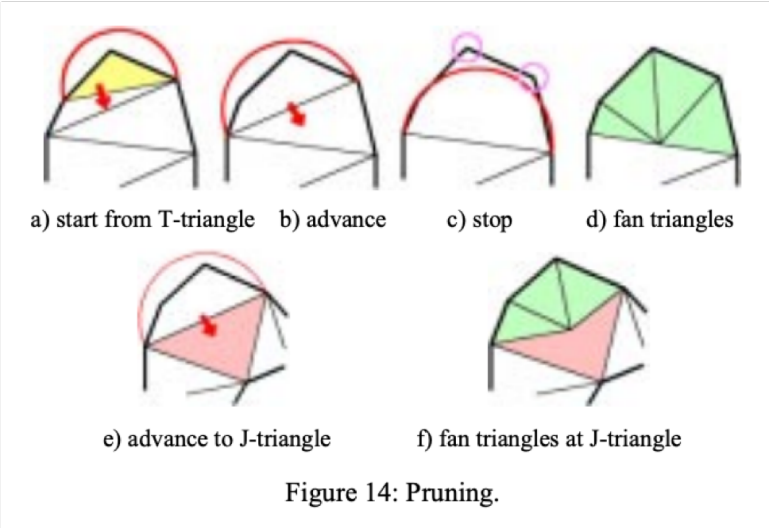


Figure 4: The pruning/fanning process. Image from *Teddy*.

The spine of the polygon is obtained by connecting the fan centers and midpoints of interior edges. After this fanning/pruning process, we then subdivide the resulting sub-polygons such that all interior edges are between the spine and the polygon boundary.

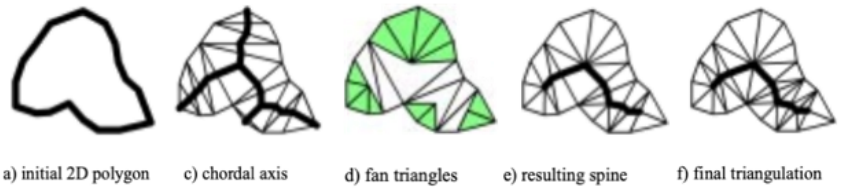


Figure 5: The original polygon → constrained Delaunay triangulation → pruned triangulation → final spine. Image from *Teddy*.

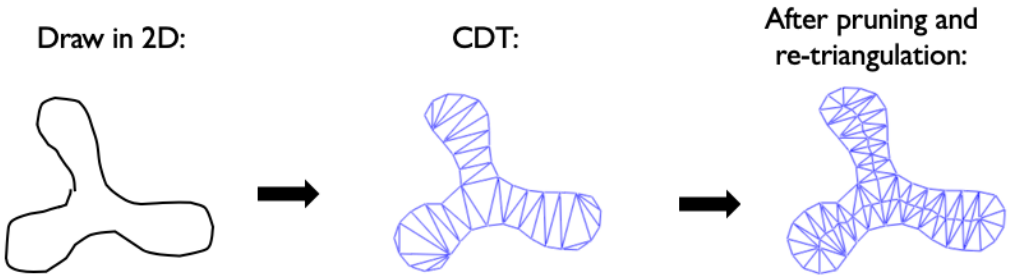


Figure 6: Another visualization of the triangulation process.

Inflating the mesh

After the final triangulation, we are left with the spine of the polygon (which can be characterized as consisting of the edges between interior vertices) and the attached triangles. To construct the mesh, first the spine is elevated. Each vertex on the spine is elevated a distance equal to the average length of its adjacent edges. In this way, wide areas of the polygon become fat regions of the mesh, and narrow areas become thin.

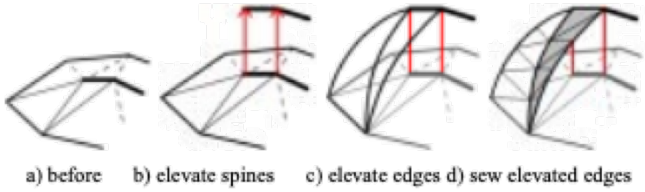


Figure 7: The inflation process. Image from *Teddy.*

Then we elevate each interior edge such that each edge approximates a quarter ellipse. (The axes of the ellipse will be the length of the edge, and the height of the attached spine vertex.) Each triangle is then sewn together using several smaller triangles.

Implementation

The app was developed for Android phones, which must be running Android 7.0 or higher for the AR features, using the Unity game engine (specifically, version 2017.4.26.) All scripts were written in C#. (In terms of the actual creation process, I first implemented a C++ version using OpenGL - also available at https://github.com/nzfeng/TeddyAR - then "translated" everything to C#.)

In order, the implementation steps are:

- Implement halfedge data structure for performing manipulations on the mesh
- Implement constrained Delaunay triangulation
- Implement drawing interface, which:
  - Takes in 2D drawing input
  - Resamples drawn curve
  - Performs CDT
  - Re-triangulate
  - Elevates the spine
  - Elevates each of the interior edges, then stitches them together
  - Construct vertex, triangle, and vertex normal arrays for the mesh.
- Have ARCore detect planes on which to place the newly created object
- Support other mesh operations.

Constrained Delaunay Triangulation was originally implemented following Sloan 1992 but was too slow, so I ended up using the poly2tri library, which is much more optimized. I implemented the other steps of the algorithm exactly as described in the previous section, via the corresponding the halfedge operations.

The augmented reality components of the app were implemented using Google ARCore, which provides plane detection and lighting estimation capabilities. Specifically, horizontal planes are detected for the user to place meshes on, and lighting estimation provides a plausible lighting of the mesh. Google does not expose the source code for its functionalities, but my guess is that lighting estimation is done by computing some average of pixel intensities across frames to estimate light direction and intensity.

All project materials and source code can be found at https://github.com/nzfeng/TeddyAR.

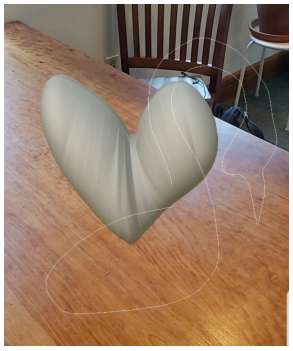
Results

Below are screenshots from the app:

(a) Drawing a heart in 2D (the white line - it might be a little hard to see.)

(b) The resulting mesh.

Figure 8: Example of drawing a heart.



(a) Drawing a swan in 2D, over the existing heart mesh.

(b) The resulting mesh.

Figure 9: Followed by drawing a swan.

The mesh is rendered as a white, opaque diffuse surface. The video demo can be found at https://drive.google.com/drive/folders/1j1L6u3O_p7O8mB1ARDwWcoRzWMH7Eegw?usp=sharing, titled "teddy.mp4".

Conclusion

Currently, the user is only able to create one mesh at a time. Unfortunately the sketch-to-mesh conversion takes 10 - 15 seconds, depending on the desired mesh resolution. Pinch-enlarge and shrink capabilities are supported (the user can use a pinching motion to scale the mesh up or down), although this operation is laggy and would be aggravating to a serious user. The main bottleneck in speed is due to assigning to mesh vertices and/or triangles, which is a copy operation, and unfortunately this must be done repeatedly if we want to update them (no direct reference.)

The goals of this project were to use AR to extend the capabilities of sketch-based modeling, and also provide example usage of Google ARCore, which is fairly new (initial release March 2018; stable release February 2019), and there has been little existing work demonstrating its use.

The original Teddy paper also proposes methods for extrusion and cutting, which would enable the user to create meshes with holes in them, for example. Additional mesh operations such as extrusion and cutting, and mesh smoothing, would allow users to create a much wider variety of objects, and encourage more creativity and interaction. There are also many other existing sketch-based modeling algorithms (in particular, non-inflation-based ones) that could be extended to AR, for example ILoveSketch or BendSketch.

Source code can be found at https://github.com/nzfeng/TeddyAR, and presentation material can be found at https://drive.google.com/drive/folders/1j1L6u3O_p7O8mB1ARDwWcoRzWMH7Eegw?usp=sharing.

References

1. Igarashi, Matsuoka, Tanaka, *Teddy: A Sketching Interface for 3D Freeform Design*, 1999.
2. Lakshman Prasad, *Morphological Analysis of Shapes*, 1997.
3. Math in Unity reference: habrador.com
4. Google ARCore API reference: https://developers.google.com/ar/reference/
5. C++ poly2tri: https://github.com/jhasse/poly2tri
6. C# poly2tri: https://github.com/MaulingMonkey/poly2tri-cs