CPSC 526 project: Finbook

David Williams-King

April 10, 2013

1 Introduction

There is an online tool called Flipbook [3] that lets users create animations frame by frame (in the style of physical flip books). The site is quite popular, with new flipbooks updated several times every hour. However, the interface is quite primitive: users can draw individual lines, undo the latest operations, and clone the last frame to create a new one (or else draw the new frame from scratch). Making new frames that are similar to previous ones is challenging.

In this project, I created a program similar to Flipbook, called Finbook[1]. The interface is better in one main way: I implemented some techniques[2] from "As-rigid-as-possible shape manipulation" [6] to allow users to create new frames based on old ones more easily.

Figure 1: Finbook logo.

The current version of Finbook is v1.5.4, the fifth public release. It is available online at `<http://elfery.net/finbook-1.5>`, and the code is released under an open source 3-clause BSD license.

2 Related Work

As mentioned, the main technique used by Finbook is as-rigid-as-possible deformation [6]. There are several existing implementations of this available [10] [11]; I referred most heavily to Ryan Schmidt's C++ implementation [11] when writing my own (see below for details).

There has been previous work on rigid deformation of hand-drawn cartoons [12]. The method is designed for fully coloured cartoon characters, and consists of two steps: pushing vertices so as to minimize changes in colour (i.e. distance metric in colour space), and secondly a rigid mesh deformation similar to our method. This work seems quite comprehensive and generates impressive results, but it may be overkill for simple non-filled-in sketch-based objects.

2.1 Libraries and algorithms

Finbook is written in JavaScript (with HTML and CSS), which is an advantage over the Flash implementation of Flipbook: JavaScript is more portable and tends to run faster, particularly with modern JS engines like Google's v8 [5].

Standard JavaScript libraries used:

- numeric.js [9] for linear algebra operations;
- jQuery [7] for general convenience;
- jCanvas [4] for a jQuery-compatible HTML 5 canvas;
- and Bootstrap [13] for user-interface elements.

In addition to as-rigid-as-possible deformation, Finbook uses a few smaller algorithms:

- Delaunay Triangulation, using a public domain implementation by J. T. L [8];
- Point-in-polygon determination (even-odd rule algorithm), implemented by me and released into the public domain;
- Convex hull calculation, using the quickhull algorithm [2], implemented by me based on [14], and released into the public domain;
- Barycentric coordinate calculation.

These algorithms are described in context below.

3 How Finbook Works

3.1 Turning sketches into meshes

If you want to apply as-rigid-as-possible deformation to sketches the user draws on the screen, there is one main problem. A sketch is not a three-dimensional mesh, but rather a curve, and the curve must be converted to a

[1] The name comes from the fact that fins are rather like flippers.

[2] The first two of three steps; thus, I didn't implement their exact technique, but rather a deformation method which does produce reasonable results.

mesh. The deformation paper does talk about curves (using 2D Laplacian curve editing), but doesn't mention how to handle multiple curves in one object, like legs and arms. Also, I wanted to leave open the possibility of supporting coloured-in regions in the future. So I wanted to use mesh deformation.

Another issue is that sketches the user draws have a large number of control-points. When they move the mouse slowly, strokes with a few hundred points would not be uncommon. One way to handle this issue is to use a reverse subdivision scheme (which I have successfully done in the past), which has the additional benefit of letting the user smooth their drawings. But the method used by Finbook does not actually care about large numbers of points (the mesh density is decoupled from the sketch point density).

Converting a sketch to a mesh can be done if you already have a polygon representation or a convex outline. So given the user's sketches, I simply find the convex hull of the points and treat that as an initial polygon. (I wrote a quickhull [2] convex hull implementation, which is reminiscient of quicksort, based on [14]. I've released my quickhull code into the public domain.) Then it makes sense to use some kind of Delaunay triangulation. There is a port of the poly2tri [1] sweep-line algorithm to JavaScript, but I didn't realize that initially and so came up with a different mechanism.

I generate a lot of random points and make sure they are inside the convex hull by using an even-odd rule algorithm (see Figure 2; this is the other part of my code I've released public domain, since it's reusable). Then, I run a constrained Delaunay triangulation method using these points, union the points on the convex hull. (I found a very fast sub-quadratic implementation of this in the public domain [8], so I used that rather than write my own.) The triangulation finds the best construction of non-overlapping triangles that use all of the points. See Figure 3 for the result.

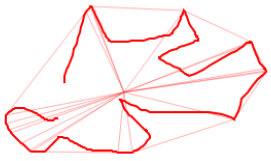

Figure 2: Finding the convex hull using an even-odd rule algorithm (similar to winding number raycasting).

This gives a mesh, which can be used in as-rigid-as-possible deformation. The mesh is not ideal, consisting of random points and capturing more space than it needs to, but we will see later that these are minor issues.

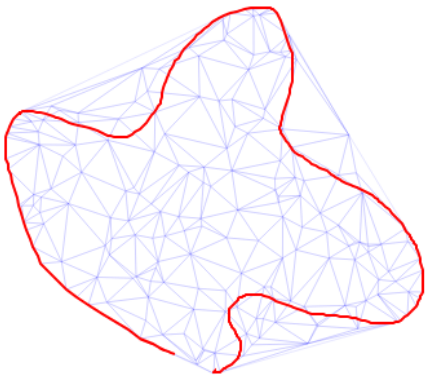

Figure 3: Delaunay triangulation using points on the convex hull, and random points that lie within the hull.

3.2 Applying deformations to sketches

Suppose we've determined a mesh using Delaunay Triangulation as described above, and then applied as-rigid-as-possible deformations to that mesh. Now we have a deformed mesh, but we really wanted deformed user sketches. So Finbook takes every input point in the user's sketch, and warps it according to the deformed mesh. The first step is to find the barycentric coordinates of each point in terms of the original mesh, and which triangle the point lies in. (This is currently a linear search, which is inefficient, but the runtime is dominated by the matrix operations in as-rigid-as-possible.) Then the corresponding triangles are found in the deformed mesh, and the barycentric coordinates are turned back into Euclidean coordinates using the deformed triangles. Finally, straight lines are drawn between the deformed sketch-points (under the assumption that the deformation is relatively linear at small scales).

The deformed vertices are cached for each particular deformed mesh, so that per-frame display time won't be affected. As mentioned earlier, it would be possible to reduce the number of points through reverse subdivision schemes, or speed up the linear barycentric-coordinate search with a 2D spatial data structure, but these don't appear to be major runtime bottlenecks.

3.3 As-Rigid-As-Possible Deformation

The heart of the program is as-rigid-as-possible deformation, which takes an input mesh with some vertices constrained to different positions, and creates an output mesh which satisfies the constraints and is deformed "rigidly". The algorithm supplies a closed form for the output mesh, i.e. it is deterministic and fully reversible and will not explode when collapsed to a singularity, etc.

The algorithm has three main steps.

1. Orientation: first, an error metric is minimized which

allows rotation and uniform scaling of every triangle[3].

2. Scaling: secondly, the error between each edge and its original edge length is minimized; in other words, the triangles are scaled to try to match their original dimensions (finding a triangle similar to the original).

3. Fitting[4]: this takes individually scaled triangles and merges their vertices together, minimizing a global error function. (Each triangle may "pull" a specific vertex to different places, which must be resolved.)

3.3.1 Step 1: Orientation

The first step is relatively straightforward to derive from the original paper. I did it by hand on paper. You start by writing each vertex of a triangle in terms of the others:

$$v_2^f = v_0 + x_{01}(v_1 - v_0) + y_{01}R_{90}(v_1 - v_0)$$
$$v_1^f = v_2 + x_{20}(v_0 - v_2) + y_{20}R_{90}(v_0 - v_2)$$
$$v_0^f = v_1 + x_{12}(v_2 - v_1) + y_{12}R_{90}(v_2 - v_1)$$

Now, in two dimensions a counter-clockwise rotation of 90 degrees is just $x' = y, y' = -x$, so we have

$$v_{2x}^f = v_{0x} + x_{01}(v_{1x} - v_{0x}) + y_{01}(v_{1y} - v_{0y})$$
$$v_{2y}^f = v_{0y} + x_{01}(v_{1y} - v_{0y}) + y_{01}(-v_{1x} + v_{0x})$$
$$v_{1x}^f = v_{2x} + x_{20}(v_{0x} - v_{2x}) + y_{20}(v_{0y} - v_{2y})$$
$$v_{1y}^f = v_{2y} + x_{20}(v_{0y} - v_{2y}) + y_{20}(-v_{0x} + v_{2x})$$
$$v_{0x}^f = v_{1x} + x_{12}(v_{2x} - v_{1x}) + y_{12}(v_{2y} - v_{1y})$$
$$v_{0y}^f = v_{1y} + x_{12}(v_{2y} - v_{1y}) + y_{12}(-v_{2x} + v_{1x})$$

Then write the error metric

$$\sum_i E_{v_i} = \sum_i [(v_{ix}^f - v_{ix})^2 + (v_{iy}^f - v_{iy})^2]$$

and substitute out the occurrences of v_{2x}^f and v_{2y}^f using the above formulae. Shuffle terms and solve for $(v_{0x}^f, v_{0y}^f, v_{1x}^f, v_{1y}^f)$. In the actual program, write down the matrix, invert it and find these four variables, then solve for the v_2^f variables after that.

3.3.2 Step 2: Scaling

The second step is quite a lot harder, and I spent many hours on paper and in sage trying to simplify the exact formulae. (They're relatively easy to compute but the derivative is half a page.) Eventually I gave up and looked at the Mathematica-based notebook from [11], and that made more sense (Ryan Schmidt had spent some time finding common sub-expressions and factoring them out). So my implementation is much like Ryan's implementation[5], although they are in different languages. I was going to spend more time deriving my own version from scratch, but it didn't seem worth it since this version worked, and there were other algorithms to work on.

3.3.3 Step 3: Fitting

The third step is reminiscient of the first step in terms of the matrices that are getting constructed. I got partway through implementing this and decided it wasn't really necessary. With a very sparse mesh it is important, but here is what the deformation looks like on a Finbook mesh:

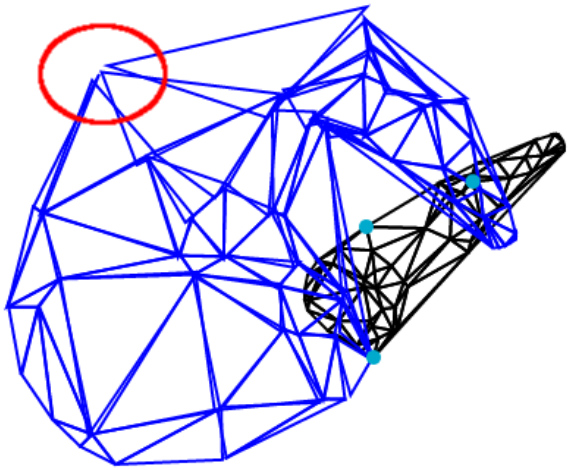

Figure 4: As-rigid-as-possible deformation without fitting (the third step). The vertex circled in red is about as bad as it gets.

This is really quite good, so rather than spend the implementation time on a step that's not so important (and the only step to involve LU decomposition – slow!), I opted for a simpler approach. I just take all the proposed vertices and average their positions. This will not find the global minima from the original problem; it behaves more like a Laplacian deformation than a rigid deformation. But I think it's just as useful for flipbook authors (at least for a system based on sketches rather than filled-in regions). It's certainly an improvement over having to redraw an entire character anew for each frame, which is what Flipbook requires.

3.3.4 Caching data

As described in the as-rigid-as-possible paper, Finbook precomputes the matrices used in steps 1 and 2 every time a constraint point is added or removed. This is by far

[3] Apparently, this corresponds to 2D Laplacian editing.
[4] The current version of Flipbook does not implement this step.
[5] Note: the license of Ryan's code is "free as in beer, you can use it in any commercial product you like", so I don't feel too guilty.

the most expensive step. Then, in the editor, the original mesh vertices are deformed according to the matrices once per frame (it's quite a fast operation for small to medium sized meshes). And when the deformed mesh changes, it is used to warp the positions of the input sketches; this is cached so that as far as the rest of the program is concerned, it is simply painting a sketch object.

4 Using Finbook

Finbook has three modes: Sketch, Deform, and Run animation. In Sketch mode, the user can draw sketches by clicking and dragging the mouse; in Deform mode, the user can take a set of sketches and apply as-rigid-as-possible deformation to them; and Run animation mode shows the frames in sequence at ten frames per second.

In sketch mode, click and drag the mouse to draw strokes. All strokes will get added to the current object by default; click on the "+" icon on the right to create a new object (then you can select one from the Objects list). An object is simply a collection of strokes that you can deform at the same time. If you make a mistake you can delete an object (with the "-" button) and start over.

In deform mode, click on at least two vertices of the current object to add constraints. Then, click and drag the constrained vertices to deform the object. You can add more constraints, or remove them with ctrl-clicking. The deformed object will replace the original in the sketch view, unless you click "Undo deform" to restore the original object.

You can add new frames by making a copy of the current one (lower-right, "Copy") or creating a new frame. If you copy a deformed object, the two objects can be deformed and manipulated independently. You can copy/add frames at any point in the timeline (unlike Flipbook, which only allows frame copies to be placed after the last frame!). Use the buttons around the timeline to switch between frames. And press "Run animation" (or the play button twice) to see a loop of the animation.

Exporting animations to JSON (JavaScript Object Notation, a text file format) is supported. Currently, loading animations doesn't properly load undeformed meshes (just the deformed versions), so while you can display animations, you can't really edit them after the fact. However, the file format is fully forward compatible, storing all the data necessary for reconstruction. So current saves may be fully restorable in the future.

The pause button makes Finbook stop repainting, so that it will use virtually no CPU (for switching tabs etc).

5 Future Work

A list of features that it would be nice to have in Finbook but which I did not implement:

Figure 5: Screenshot of Finbook version 1.5.4.

- Different colours/brush sizes for drawing!
- Basic undo/redo functionality for strokes, and copy/paste/deletion of strokes within each object – instead of just manipulating objects themselves.
- Full three-step as-rigid-as-possible deformation, to see how much of a difference it really makes.
- Smarter, non-randomized triangulation with Constrained Delaunay Triangulation [1]. Or at the least, some way to control the kind of mesh that gets created. Right now Finbook sometimes creates highly detailed meshes which are quite slow to deform (i.e. the as-rigid-as-possible precomputation steps).
- Exporting ("publishing") finbooks to a website. This requires some backend work which I will probably do at some point, but which was beyond the scope of the course project.
- It would be possible to "parallelize" the numeric.js calculations to run in a different web worker thread, so that the user could interrupt it if it takes too long.

6 Conclusion

Finbook is a JavaScript animation tool that lets you draw flipbooks frame by frame. It supports rigid deformations of objects, not using the actual technique presented in [6], but a subset of that which should still be useful for flipbook authors. Deformation greatly speeds up the process of creating a frame similar to the previous one. And Finbook has a nice user interface with support for inserting arbitrary frames, grouping strokes into objects, etc. Overall, it's a very good start on a useful animation tool.

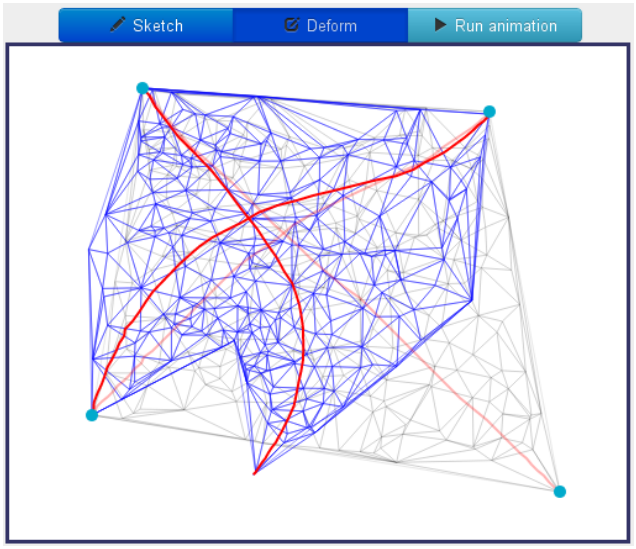

Figure 6: Showing deformation of a cross or kite.

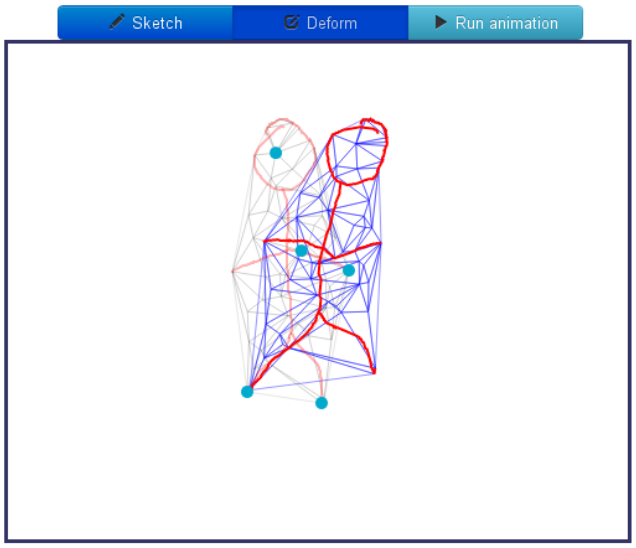

Figure 7: Deforming the position of a stick figure.

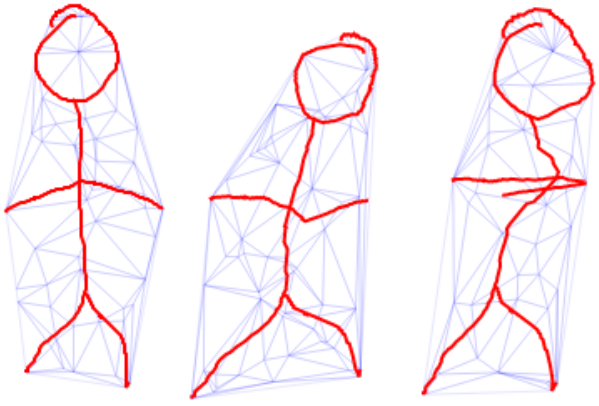

Figure 8: Several deformed versions of the same figure.

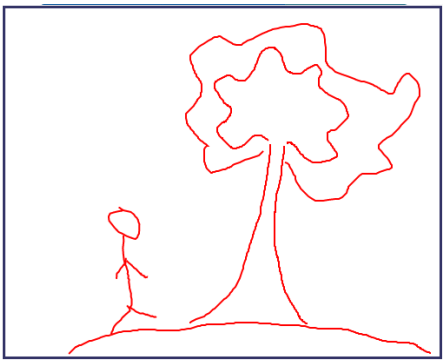

Figure 9: A walking character.

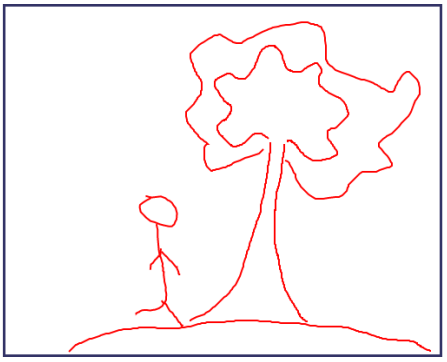

Figure 10: Deforming means redrawing is not necessary!

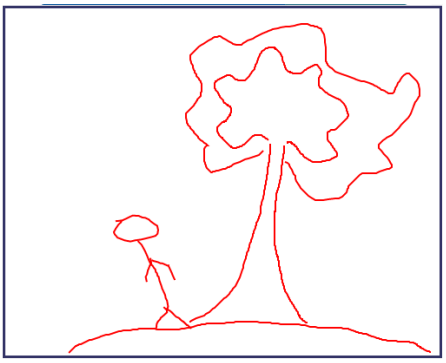

Figure 11: Looking at this tree.

Figure 12: Showing the original undeformed sketch.

References

[1] poly2tri: A 2d constrained delaunay triangulation library. <http://code.google.com/p/poly2tri/>.

[2] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[3] Benetton. Flipbook Deluxe. <http://www.benettonplay.com/toys/flipbookdeluxe/>.

[4] Caleb Evans. jCanvas. <http://calebevans.me/projects/jcanvas/>.

[5] Inc. Google. V8 javascript engine. <http://code.google.com/p/v8/>.

[6] Takeo Igarashi, Tomer Moscovich, and John F Hughes. As-rigid-as-possible shape manipulation. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1134–1141. ACM, 2005.

[7] The jQuery Foundation. jQuery. <http://www.jquery.com/>.

[8] J. T. L. Fast Delaunay Triangulation in JavaScript. <https://github.com/ironwallaby/delaunay>.

[9] Sébastien Loisel. Numeric javascript. <http://www.numericjs.com/>.

[10] Xiaofeng Mi. ARAP. <http://www.research.rutgers.edu/ xmi/files/arap.tar.gz>.

[11] Ryan Schmidt. As-Rigid-As-Possible 2D Shape Manipulation Demo. <http://www.dgp.toronto.edu/ rms/software/Deform2D/>.

[12] Daniel Sỳkora, John Dingliana, and Steven Collins. As-rigid-as-possible image registration for hand-drawn cartoon animations. In *Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering*, pages 25–33. ACM, 2009.

[13] Twitter. Bootstrap. <http://twitter.github.io/bootstrap/>.

[14] Jakob Westhoff. Calculate a convex hull – The Quickhull Algorithm. <http://westhoffswelt.de/blog/0040_quickhull_introduction_and_php_implementation.html>.