Automated Calligraphy Using Dynamics

Christopher Thompson
Department of Computer Science and Engineering
University of Washington

Abstract

We have developed a partially automated system for creating calligraphic letterforms. The system uses a simple dynamics model to generate and render realistic pen strokes. The simulation is controlled by a series of constraints specified by the user. In this report, we describe our novel approach to generating calligraphy and then evaluate its effectiveness.

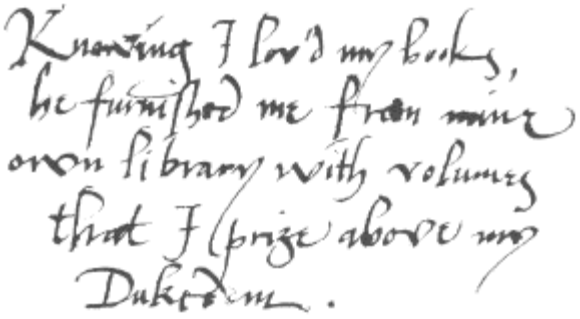
1 Introduction

Calligraphy is an art form with a long and rich history. East Asian (Chinese, Korean, and Japanese), Arabic, Hebrew, Greek, and Western cultures all have significant and distinct calligraphic traditions. In fact, there are so many different styles of calligraphy that a taxonomy of all of them would be nearly impossible. To give you some idea of the possibilities, four examples of recent Western calligraphy follow.

Source: Online calligraphy exercise book

Source: "The Pillow Book" (film, 1996)

Source: United States Postal Service



Source: “Prospero’s Books” (film, 1991)

For this project, we built a software system for creating calligraphic letterforms. The user positions skeleton letters on the screen, uses the user-interface to specify constraints, then finally clicks "Render." The program runs a dynamics simulation to create calligraphic-looking pen strokes, taking into account the mass of the pen and the friction of the pen with the paper. Finally, the program takes the output of the simulation and applies a user-specified rendering style to draw a final image.

Here are three sample images generated using my system. For more, please see the "Results" section further down.

Three rendering styles: futuristic, uncial, and pen.

2 Prior Work

The author of this report is not aware of any other system that uses a dynamics simulation to create calligraphy. A few commercial calligraphic software packages exist. One, "ByHand," claims to take into account "pen speed and direction." However, since this package creates strokes in real time, it is unlikely that it solves a physical optimization problem.

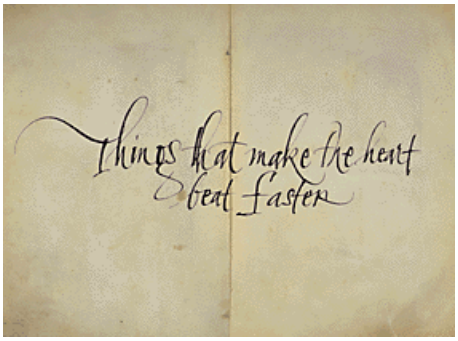
The inspiration for the novel technique described in this paper comes from Paul Haeberli's DynaDraw paint program, which modulates mouse strokes using a simple dynamics model and generates calligraphic-looking strokes. However, using DynaDraw to actually write a word is virtually impossible--the system is too difficult to control (since the user is specifying the inputs to a continuously varying simulation, a task that is even more difficult than specifying the initial values for an initial value problem). Readers can play with a Java applet implementation of DynaDraw at http://pubweb.bnl.gov/people/mcdonald/Applets/DynaDraw.html. The inspiration for our novel approach was that if there was a way to specify constraints on the dynamics simulation beforehand, and then run the simulation, the system could potentially be useful for creating calligraphy.

Witkin and Kass's "Spacetime Constraints" paper (SIGGRAPH 88) describes how to create an animation by having the animator specify a series of constraints (e.g. jump from point A and fall down at point B) and then having the computer solve a physically constrained optimization problem to generate the motion. We use a variant of their "spacetime particle" model in this project.

A related motivation for this work was Wong, Zongker, and Salesin's "Computer-Generated Floral Ornament" paper. This paper describes a system that creates complicated ornamentation using the following surprisingly elegant algorithm:

do
  find a big empty space
  draw a stroke in the space
until there are no big empty spaces left

Another interesting observation is that expert calligraphers (particularly Asian calligraphers) often plan their strokes in a global way, attempting to draw strokes that naturally fill empty spaces, or connect with other strokes. As an example, consider the following example from Peter Greenaway's film "The Pillow Book":



Notice how the descender of the letter "k" in "make" connects up with the top of the "t" in "faster," as if the calligrapher wanted to create a conceptual capital "T". Clearly, it would be very interesting to investigate how to place constraints in order to meet these types of goals. Though we did not plan on writing code to do this, our interactive system lets us experiment with different ways of placing constraints.

3 Novel Approach

To my knowledge the automated calligraphy approach described here is novel and has not been tried before.

Pen strokes are modeled as the trajectory of a single particle. This particle has a user-specified mass and is under the influence of a user-specified amount of friction (or, more correctly, drag). Each stroke is parameterized by t. The start and end of each stroke occur at t=0 and t=1 respectively. Let the particle's position at time t be x(t), its velocity at time t be v(t), its acceleration at time t be a(t), and the force that the hand applies to the particle at time t be f(t). The particle's mass is m, and the force due to drag at time t is d(t). The equation of motion is from physics:

$$m\mathrm{a}(t) - \mathrm{f}(t) - \mathrm{d}(t) = 0.$$

Under the influence of the constraints implied by the equation of motion, as well as other user-specified constraints (discussed below), we find a solution which minimizes:

$$\int_0^1 |\,\mathrm{f}(t)\,|^2 dt\,.$$

This optimization is performed using the CFSQP quadratic programming solver.

The system supports three types of user defined constraints:

1. *Equality Constraints*
   At t, the particle must be at a specified point on the plane.
2. *Inequality (Region) Constraints*
   At t, the particle must be within a given radius of a specified point.
3. *Weight Constraints*
   At t, the particle should tend towards a specified point. The amount of the particle's attraction to the point is controlled by a user-specified weight. These constraints result in a solution which does not perfectly minimize the integral of the square of the force.

A more rigorous description of how these constraints are implemented can be found in the next section of this report.

After it finds a solution to the optimization problem, the system renders each stroke by drawing the trajectory of the particle. The rendering style can be controlled by changing how the pen width and angle are calculated. The user controls the minimum and maximum pen width using sliders, then the user chooses a method for determining what value between the minimum and maximum to use. The options for varying the pen's width are:

- *Constant*
  The pen's width is equal to the value of the maximum width slider and never varies.
- *Global Velocity*
  The pen's width is maximal at the point on the stroke where the pen is moving the slowest, and the width is minimal at the point on the stroke where the pen is moving the fastest. Linear interpolation is used in between. This roughly emulates the way that more ink sinks into a page when you move a brush more slowly.
- *Global Velocity (Inverse)*
  The opposite of the previous option. The pen's width is maximal where the pen is moving the fastest.
- *Local Velocity*
  The difference between a user-specified threshold and the pen's velocity is subtracted from the maximum pen width to give the actual pen width.
- *Local Velocity (Inverse)*
  The opposite of the previous option. The difference is added to the minimum pen width.
- *Global Force*
- *Global Force (Inverse)*
- *Local Force*
- *Local Force (Inverse)*
  Each of these options is identical to the corresponding velocity options, except that they use force as the determining variable.

The options for varying the pen's angle are:

- *Constant*
  The pen's angle (specified by the user via a dial) is constant.
- *Velocity*
  The pen's angle is perpendicular to the velocity. The pen turns as the particle turns.
- *Velocity (with Offset)*
  The pen's angle is equal to the angle perpendicular to the velocity plus a constant offset specified by the user.
- *Force*
  Identical to the Velocity option, but using force.
- *Force (with Offset)*
  Identical to the Velocity (with Offset) option, but using force.

4 In-Depth Details

This section of the report describes how we set up the quadratic programming problem. We represent x(t) and f(t) independently. Each of these functions is discretized into n intervals (*i.e.* a sequence of n+1 samples, labeled as x_i and f_i), with a time interval of h between samples.

We represent the physics constraints by creating tableau equality constraints as follows:

$$m\frac{x_{i+1} - 2x_i + x_{i=1}}{h^2} - f_i - c_d\frac{x_i - x_{i-1}}{h} = 0$$

where c_d is a user-specified drag coefficient.

We represent the equality constraints by creating the obvious tableau equality constraints:

$$x_i - p_i = 0$$

where p_i is the user-specified location where x_i should be.

Inequality constraints are represented by creating tableau inequality constraints:

$$(x_i - p_i)^2 \le r_i^2$$

where r_i is the user-specified radius around floating control point p_i.

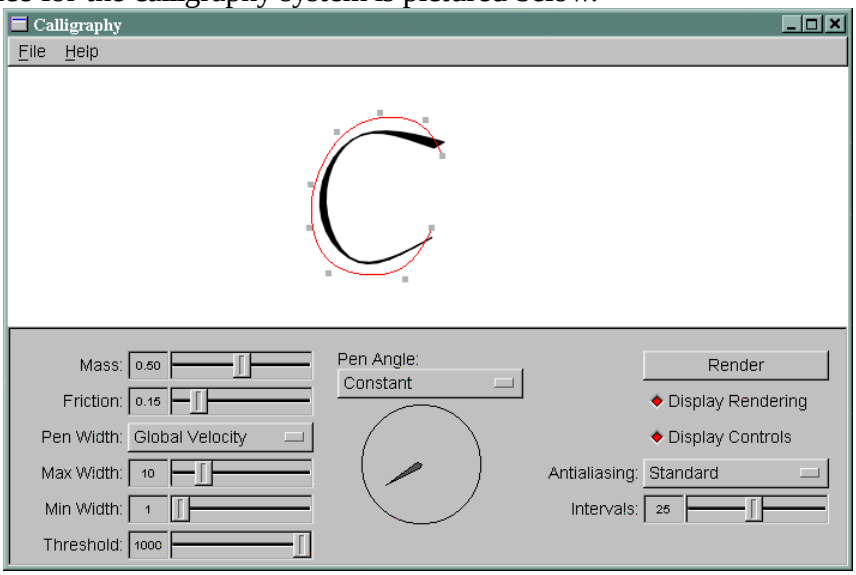
Finally, weight constraints are represented by altering the objective function to minimize:

$$h\sum_i \left| f_i^2 \right| + \sum_j w_j \left| (x_j - p_j)^2 \right|$$

where w_j is the user-specified weight of floating control point p_j.

5 The User Interface

The user interface for the calligraphy system is pictured below.



The controls at the bottom of the interface correspond in to the various parameters discussed earlier in this report. For instance, the "Min Width" slider controls the minimum pen width, and the "Pen Angle" combo box selects an option for varying the pen angle. The only controls not mentioned previously are:

- *Antialiasing*
  Controls the level of antialiasing (None, Standard, or High).
- *Intervals*
  Sets the number of intervals the solver should discretize *t* into.
- *Display Rendering*
  When checked, this causes the viewport to show the results of the last rendering.
- *Display Controls*
  When checked, this causes the viewport to show the user-interface (control points, stroke skeletons, constraints, *etc*.).

If the user clicks the right mouse button in the viewport, a pop-up menu appears. This menu contains all of the letters of the alphabet (both uppercase and lowercase). Choosing a letter causes a set of skeleton strokes defining that letter to be placed at the mouse position.

There are two pull-down menus at the top of the screen. The options are:

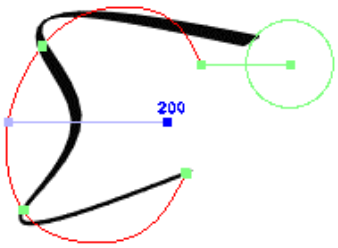
- *File -> New*
  Deletes all the skeleton strokes from the current document.
- *File -> Open*
  Displays a dialog that allows you to select a document to load. The program uses a custom file format.
- *File -> Save*
  Displays a dialog that allows you to specify a file name. Pressing OK saves the current document to that file.
- *File -> Save Image*
  Displays a dialog that allows you to specify a file name. Pressing OK saves the most recent rendering to a Windows bitmap (BMP) file.
- *Help -> About*
  Displays information about the program.

The program has two modes, Skeleton Editing mode and Constraint Editing mode. The skeleton strokes are always visible in both modes. All skeleton strokes are drawn in grey, except for the currently selected stroke(s), which is (are) drawn in red. Control points for the skeleton strokes are only visible in Skeleton Editing mode, and control points and weights for the constraints are only visible in constraint editing mode. All operations that affect the skeleton strokes are done with the middle mouse button. All operations that affect the constraints are done with the left mouse button. Therefore, switching between the two modes is simple and intuitive – to go into Skeleton Editing mode, use the middle mouse button for something. To go into Constraint Editing mode, use the left mouse button for something.

In Skeleton Editing mode, the B-spline control points are drawn in grey, except for the currently selected point, which is drawn in red. You can change the positions of control points by dragging them. You can also:

- press CTRL + middle mouse button on a skeleton stroke to add a control point
- press CTRL + middle mouse button on a control point to delete it
- press SHIFT + middle mouse button on a skeleton stroke to add it to the set of currently selected strokes.

In Constraint Editing mode, control points for equality and inequality constraints are drawn in green, while control points for weight constraints are drawn in blue. Whenever a control point is selected, it becomes a brighter shade of green or blue (as appropriate). Control points can be moved by dragging with the left mouse button. Each constraint has two control points: the *root* control point, which is attached to a skeleton stroke and is used to determine the value of *t* that the constraint affects, and the *floating* control point, which can be positioned anywhere. The root and floating control points for each constraint are connected by a line, so that their relationship is obvious. Each weight constraint has an integer displayed above its floating control point – this is the weight assigned to the constraint. Each inequality constraint has a circle drawn around its floating control point, representing the region that this constraint affects. The following image shows all three types of constraints:



In Constraint Editing mode, you can:

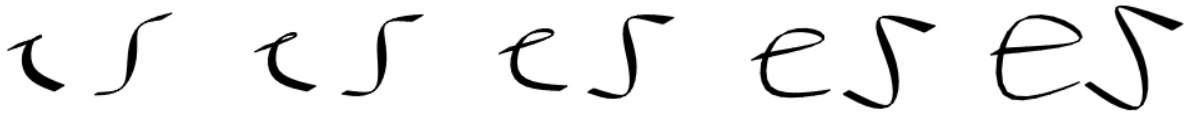
- press CTRL + left mouse button on a skeleton stroke to add a weight constraint
- press CTRL + left mouse button on a weight constraint's root point to delete it
- press SHIFT + left mouse button on a skeleton stroke to add an inequality constraint (to transform this into an equality constraint, hold down Z as described below)
- press SHIFT + left mouse button on a skeleton stroke to delete an equality/inequality constraint
- hold Z while a weight constraint floating control point is selected to decrease the constraint's weight
- hold X while a weight constraint floating control point is selected to increase the constraint's weight

- hold Z while an inequality constraint floating control point is selected to decrease the constraint's radius; when the radius becomes small enough so that you can no longer see it, the inequality constraint becomes an equality constraint
- hold X while an equality/inequality constraint floating control point is selected to increase the constraint's radius (possibly transforming an equality constraint to an inequality constraint)
- press D to delete all the constraints associated with the currently selected skeleton stroke
- press Q, W, E, or R to have the program automatically assign a set of constraints to the currently selected stroke; each of these letters uses a different algorithm to assign constraints (try them all!).

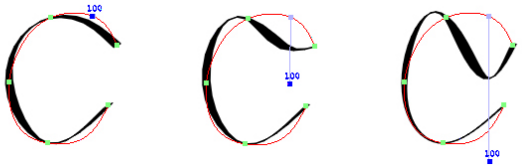
6 Results

Many different lettering styles can be produced by changing the types and locations of constraints and varying the pen parameters. Three interesting examples appeared at the beginning of this report. Here, the effects of specific parameters are examined.

Using only weight constraints tends to produce very lazy looking text. Depending on one's point of view, the text could be either "futuristic" or written by someone who is very tired. Decreasing the pen's mass or increasing the weights of the constraints creates more controlled, deliberate text. The following images show the effect of increasing the mass. Each letter has five uniformly spaced weight constraints along its skeleton stroke.

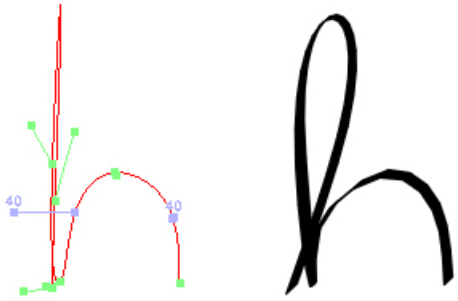


Weight constraints are also useful as a way for the user to "tug" the pen in a specific direction in an intuitive way. Since the simulation is global, this "tug" influences the whole curve shape, which works nicely to produce distortions that look natural, as shown below.



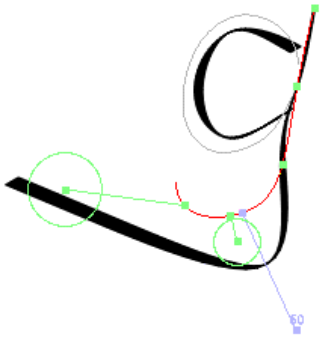
Equality constraints are a different beast. Naturally, placing many equality constraints along a skeleton stroke forces the letter to approximate its skeleton very closely. If a person is going to do this, why not just use splines and forget about dynamics? The dynamics simulation provides velocity, force, and acceleration information that can be useful to the algorithm that draws the brush strokes. In the following example, we use many equality constraints to get Helvetica-like e and s shapes, but we also force the pen angle to be perpendicular to the particle's direction of motion, and use the global velocity to control the pen width. The result is a decidely unusual look:

Equality constraints tend to be most useful when used in moderation. For instance, the following example takes advantage of the dynamics simulation to draw a nice cursive loop. The constraints are pictured on the left, and the final rendering on the right.



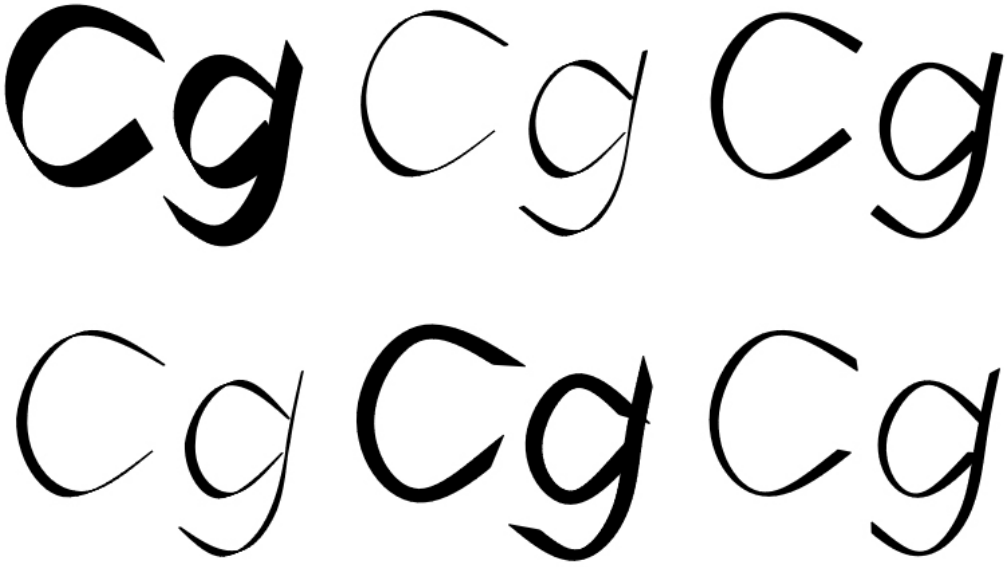
The above example also provides an opportunity to demonstrate the effect of adjusting the amount of friction. The image pictured below on the left has the least amount of friction, while the image pictured on the right has the most.

Inequality constraints can be used when you have a general idea of where a curve should go, but you would like the dynamics solver to select a realistic-looking curve. Unfortunately, inequality constraints are not very intuitive. Because of the binary nature of these constraints (the particle is either inside the specified radius or not inside), the solver will always find a solution in which the curve just touches each inequality constraint's region of influence. Weight constraints can sometimes be useful to help control the solver's solution. For instance, a weight constraint is used in the following example to position the curve of the g on the bottom side of an inequality constraint.



Here is a rendering produced using the above example's constraints (with a slightly different rendering style):

As pen parameters, using Global Velocity for the width and either a constant or Velocity (with Offset) as the angle generally produces the most pleasing calligraphic results. A few examples are pictured below.



7 Evaluation of Results

There are really two separate aspects to this project: shape and rendering style. Each can be considered separately.

The dynamics simulation was definitely good at producing curves that look as if they could have been drawn by hand. Though more work still needs to be done to determine how best to automatically create the constraints, even simple schemes for creating the constraints produce curves which are surprisingly pleasing. Also, because of the global nature of the simulation, the curves change in realistic ways when the constraints are modified by the user. While many of my sample images exhibit strong shape similarities, keep in mind that only one typeface was used as the basis for the underlying skeleton strokes. If we were to allow the user to choose a typeface and then generate skeleton strokes from the chosen face, our system could produce a wide variety of interesting types of curves.

On the other hand, the rendering style results are not entirely satisfying. While modifying the brush width based on the magnitude of the velocity produces reasonable results, such renderings only look significantly better than renderings that do not use dynamics information about half of the time. More work needs to be done to find better ways of using the dynamics information for rendering. In particular, the Global Velocity scheme has the weakness that every stroke has a point where it is at minimum width and a point where it is at maximum width, even if the stroke is very short or does not significantly change speed.

8 References

Haeberli, Paul. Dynadraw: A Dynamic Drawing Technique. http://www.sgi.com/grafica/dyna/.

Lawrence, Craig T. and Jian L. Zhou and André L. Tits. CFSQP Feasible Sequential Quadratic Programming code, version 2.5d. http://www.isr.umd.edu/Labs/CACSE/FSQP/fsqp.html.

Witkin, Andrew and Michael Kass. Spacetime constraints. *Proceedings of SIGGRAPH 88*, 159-168, 1988.

Wong, Michael T. and Douglas E. Zongker and David H. Salesin. Computer-Generated Floral Ornament. *Proceedings of SIGGRAPH 98*, 423-434, 1998.