Submission Title
Graphics: Shape Generation

Contact Information
Richard Rasala: rasala@ccs.neu.edu
Viera Proulx: vkp@ccs.neu.edu
College of Computer & Information Science
Northeastern University
Boston MA 02115

Problem Statement
One of the basic problems of computer graphics is to draw a smooth curve that passes through or interpolates a sequence of vertices. This problem divides into sub-problems based on two questions:

- Is the curve closed or open?
- Are the tangents at the vertices computed automatically by an algorithm or are they tweakable by the user?

Java provides the foundation for the solution to these problems by supplying the code to specify a cubic curve within a Bézier arch as shown in the diagram below.

Given two endpoints (shown in green) and two control points (shown in red), there is a unique cubic curve joining the endpoints whose tangent segments in the Bézier arch (shown in red line segments) up to a standard factor of 3 determined by derivative formulas. It is well known in the graphics literature that the cubic curve can be computed by a very efficient recursive subdivision of the arch and that no polynomial formulas need to be evaluated. Java provides an implementation of this path algorithm.

The problem to be addressed here is how to build longer smooth curves that are made up of cubic segments specified by such arches and that take advantage of what Java provides. The text Java 2D Graphics by Jonathan Knudsen is a reference for what is built into Java.

Solution Overview
The interpolation problem is to define a smooth curve that passes through a given set of vertex points. A sample solution to this problem is given in the diagram below in which the given 10 vertex points are shown in green and the smooth closed curve is shown in black.

Let us now consider what such a solution involves and what options must be provided to the programmer.

A fundamental decision is whether the tangents at the vertices are computed automatically by an algorithm or are tweakable by the program or the end user. JPT defines two classes to deal with this decision: AutomaticShape and TweakableShape. These classes extend the class BaseShape which contains the common data structures and methods.

The easy case is TweakableShape since JPT can assume that all vertex and tangent data is supplied externally. We therefore focus on AutomaticShape. In this case, it makes a difference if the curve is closed or open. If the curve is open, then the tangents at the curve endpoints remain freely selectable by the caller.

Therefore, to define a strategy to automatically compute tangents, it is necessary to provide two methods:

```java
// closed curves
public float[][] makeTangents(float[][] vertex)

// open curves
public float[][] makeTangents(float[][] vertex, float[][] endTangent)
```

These methods are specified in the interface Tangent.Strategy. Here, the vertex array is N x 2, the end tangent array is 2 x 2, and the tangent array that is returned is N x 2. We choose to use arrays rather than more object-oriented data structures since it is easier to express the mathematical algorithms in this format.

In the graphics community, there is no agreement on a single best strategy to automatically compute tangents. That is the reason we define an open interface Tangent.Strategy --- to allow for many possible implementations. In the Tangent class, we provide two sample strategies, the Bézier Strategy and the Chord Strategy, that are illustrated in the diagrams below.
In the diagrams, the curve is shown is black, the polygon defined by the vertex array is shown in green, and the frame defined by the successive Bézier arches is shown in red. You will notice that there are subtle differences in the arch structures and that this translates into subtle differences in the curves.

The Bézier Strategy is defined by a mathematical uniqueness property. It turns out that given an array of vertex points, there is a unique piecewise cubic curve through the points that has continuous first and second derivatives. The Bézier Strategy produces the tangents for this unique curve. In the reference cited below, we published a very efficient algorithm for computing these tangents. The essence of the algorithm is to compute the tangent at a vertex by a sum of successive chords that are multiplied by appropriate coefficients. The coefficients decrease by a factor of approximately 4 in each step so in practice very few chords are actually needed. All of this mathematics is, of course, completely encapsulated.

The Chord Strategy is simpler. The tangent at a vertex is simply a fixed factor multiplied by the chord joining the two adjacent vertices. The sample with factor $1/3$ is known as the Catmull-Rom spline and also the Cardinal Spline with tension 1. The sample with factor $1/6$ is the Cardinal Spline with tension 1/2.

Some people in the graphics community argue in favor of the Chord Strategy because of local control, that is, the tangent at a vertex can only be influenced by the position of the two adjacent vertices. On the other hand, some people prefer the smoother feel of the curves produced by the Bézier Strategy.

Let us now discuss how this translates into the construction of automatic shapes. The `AutomaticShape` class has 7 constructors that permit the caller to provide up to 6 items of information.

- A vertex array of form `float[][]` and size N x 2.
- An end tangent array of form `float[][]` and size 2 x 2 or `null` if no end tangents are needed.
- A `TangentStrategy` object. This may be one of the strategies in the `Tangent` class or be user defined. If `null` then the default strategy is the Bézier Strategy.
- A `PathStrategy` object. This is one of the options:
  - `Path.POLYGON`
  - `Path.BEZIER_CUBIC`
- A `ClosureMode` object. This one of the options:
  - `ClosureMode.CLOSED`
  - `ClosureMode.OPEN`
- A `WindingRule` object. This is one of the options:
  - `WindingRule.WIND_NONZERO`
  - `WindingRule.WIND_EVEN_ODD`

Note that, due to the fourth setting, an `AutomaticShape` may be used to produce either a polygon or a cubic curve spline.

In the example we used for illustration, the construction of the shape with the Bézier Strategy is done as follows:

```java
float[][] data = new float[][] {
    { 100,  25 }, { 150,  50 },
    { 200,  25 }, { 250,  50 },
    { 300,  25 }, { 300, 100 },
    { 250,  75 }, { 200, 100 },
    { 150,  75 }, { 100, 100 }
};
AutomaticShape shape = new AutomaticShape
    (data, null, null, Path.BEZIER_CUBIC);
```

This example shows how easy it is to define a cubic shape. Only the minimum amount of information is required of the caller.

So far we have discussed curved shapes in which at least the first derivatives are continuous at the vertex points. To create shapes with sharp corners or even breaks, we use the `append` methods in the class `Path`. These methods require:

- An existing `GeneralPath` object as the start path or `null` if a new path is to be created.
- An array `Shape[]` of shapes to append to one another.
- Either one `boolean` to determine whether to connect the successive shapes or an array `boolean[]` to make the connection choice on a per shape basis.

Below we show the result of appending 4 shapes, first connected and then not connected. The use of fill emphasizes the difference between making connections or not.

![Example of appended shapes with and without connections](image)

Experience with the Solution
The shape generation tools have only recently been added to JPT so we do not have extensive classroom testing. However, we have created the Kaleidoscope case study on the web site and much sample code that uses these tools. In particular, we have sample programs that illustrate the creation and editing of open curved shapes.

We have found it a real pleasure to create shapes by focusing only on the relevant geometry as expressed in the `AutomaticShape` family of constructors. We believe that faculty and students will have the same experience and will appreciate the ability to do rich graphics easily.

The rendering of shapes, images, and text will be discussed in the separate submission on `Paintable` and `MutablePaintable`.

API Documentation & Related Materials
The main JPT site to access documentation, code, and the jpt.jar:

http://www.ccs.neu.edu/jpt/

The cubic spline interpolation algorithms are described in: