Short livarot documentation

yafosf@users.sourceforge.net

December 2, 2003

This doc is for the C++ version of livarot. The C version is just a dumb conversion, so the C++ classes become C structures, and the C++ member functions become C functions with the first parameter (called `lui` as often as possible) being the instance of the C structure on which the function is supposed to work, *ie* "this". Also the function names are mangled when 2 or more versions of the same C++ function exist.

1 Creating polygons

Obviously, before you manipulate polygons, you have to create them.

1.1 What's that `Shape` class?

It's defined in the `Shape.h` file.

A bunch of preliminary definitions for the mathematically impaired:

- Graph: a bunch of points and a bunch of edges between pairs of points. Edges can be directed (origin of the edge → end of the edge). Points can be called vertices.
- Planar: Wwhen a graph can be drawn in a 2D plane without intersection between edges, the graph is planar.
- Degree: the degree of a vertex in a graph is the number of edges having that point as end. In the case of directed graphs, the indegree is the number of edges whose end is the vertex, and the outdegree is the number of edges whose start is the vertex.
- Cycle: in a graph, a succession of vertices $v_0 \to v_1 \to \ldots \to v_n \to v_0$ where each consecutive pair (v_i, v_{i+1}) is an edge of the graph. Usually, you request that each vertex only appears once in the cycle (exept the v_0, of course, which appears twice).

- Eulerian: a graph is eulerian if you can find a cycle in the graph where each edge is visited exactly once. A nice characterization is that a undirected graph is eulerian if and only if each vertex has even degree, and a directed graph is eulerian if and only if the indegree of each vertex is equal to its outdegree.

In case this is still fuzzy, check the Boolean library documentation at `http://www.xs4all.nl/~kholwerd/algdoc/datastructure.html`.

A `Shape` is a polygon. More precisely, it can describe a polygon by treating it as a planar eulerian graph. Note that although the `Shape` class can store any graph, most of its functions require eulerian planar graphs. A `Shape` can describe any polygon, whith or without holes, with or without touching edges,etc ...

And remember that `Shape`s are directed graphs.

1.2 What's that `Path` class?

It's defined in the `Path.h` file.

It's just a little helper class designed to store a path (surprised?), where a path is composed of the following possible elements:

- straight arcs (=segments)
- cubic bezier curves
- quadratic bezier curves, with any number of intermediary points
- elliptic arcs

Since these building blocks of a path are not all straight edges, they need to be converted in a polyline before being used to make polygon. Thus the `Path` class also contains a polyline approximation of the path it describes.

1.3 Winding numbers

A polygon splits the plane in a certain number of region. Each region has a winding number, which will be used in several places. For undirected graphs, the winding number of a point (any point of the given region) is the minimum number of edges you have to cross when going from this point to a point infinitely far away, not necessarily using a straight line. For directed graphs, edges have a left and a right side, and winding numbers are computed differently: the winding number of a point is the number of edges you have to cross from left to right minus the number of edges you have to cross right to left when going from the point to a point infinitely far away.

A fill rule is a rule according which to choose which region are "in the polygon" and which region aren't. The most used are the "odd-even" fill rule, where all regions with odd winding number are used, and the "non zero" fill rule, where all regions with winding number $\neq 0$ (positive or negative) are used. In livarot, the `ConvertToShape` function also accepts the "strictly positive" fill rule, where only the regions with strictly positive winding number are used.

An illustration of winding number can be found here: `http://developer.apple.com/documentation/Cocoa/Conceptual/DrawBasic/Concepts/bezier.html#/apple_ref/doc/uid/20000886/CJBCCDGG` (bottom of page)

1.4 How do i proceed?

You can build a polygon with different methods. The only requirement is that the `Shape` class instance you will use must describe a polygon. How you obtain that polygon is up to you ... Check the `Tester.cpp` file for examples.

1.4.1 Best method: path → shape

Nearly all the functions mentioned below are member functions of the `Path` class.

First: build a `Path`: allocate it, build the path with the `MoveTo`, `LineTo`, `CubicTo`, etc... commands

Second: transform the path in a polyline with a call to `Convert`

Third: allocate a `Shape` class instance or use a pre-existing one.

Fourth: fill the polyline with a call to `Fill` or stroke it with a call to `Stroke` (possibly with dashes). These function take a `dest` argument which is the `Shape` instance which should contain the polygon.

Fifth: the polygon may contain self-intersections, duplicate points or edges, and other problems, meaning it's not a planar eulerian graph. A call to `ConvertToShape` will put a "good" version of the polygon in another `Shape` instance, the one you called `ConvertToShape` with; for example `a->ConvertToShape(b)` will compute a clean version of `b` and put it in `a`. Now you have a nice planar eulerian graph, ready to be used for subsequent operations.

Note: a path is only one contour. if you want to build a polygon with several contours (for example a polygon with holes), you have to add each contour incrementally. The `justAdd` parameter in the `Fill` and `Stroke` function allows that:

- if `justAdd=false`, the contents of the `Shape` class instance you pass as receiver for the polygon are erased: the result is a polygon containing only the contour associated with the `Path`.
- if `justAdd=true`, the contents are not erased, so that the contour associated to the `Path` is just added to the other contours already present in the `Shape`.

In case you need to make a hole, don't forget to invert the contour direction (see the winding rules section for the reason why); the `invert` boolean is currently disabled.

Note 2: sometimes you can guarantee that a contour doesn't have self-intersections or other problems. In these case, the `ConvertToShape` is overkill; you can just use the `ForceToPolygon` function on the `Shape` containing the contour(s) to say "that `Shape` is already a planar eulerian graph". Use with caution, though. Also note that almost all functions on the `Shape`s reorder the points in lexicographic order.

1.4.2 Not best method: directly build the contents of the `Shape`

You can add points and edges directly in the `Shape` instance.

First: start with a call to `Reset()` to empty the contents of the `Shape`

Second: add the points with calls to AddPoint(). The integer returned by the function is the `ID` of the point, more precisely its index in the `pts` array.

Third: add edges with calls to `AddEdge()`. The 2 parameters to this function are the points' `ID`s of its extremities.

Fourth: just like to other method, call `ConvertToShape` if needed.

2 I have a polygon. Now what?

Congratulations! Now you can do boolean operations on it or inset or offset it or raster it in a buffer.

2.1 Boolean operations

It's possible to do union, intersection, difference or symetric difference. The function `Booleen` is pretty self-explanatory, and returns a "good" polygon (planar eulerian blablabla).

2.1.1 Insets or Offsets

That's the `MakeOffset` function's job. After a call to `MakeOffset`, the calling instance of `Shape` will contain and offset of the polygon you pass in the `of` parameter. Note that this result is not a good polygon; you need a subsequent call to `ConvertToShape` to get a good polygon (or you can work directly on the bad polygon, if needed). Also note that you need the `fill_positive` fill rule to get the correct polygon.

2.1.2 Rasterization

Buffers are 32 bits ARGB and used through the very simple class `Buffer`. Rasterization comes in different flavours, with different algorithms, different paints, and clipping. Most are more or less equivalent in terms of speed. The rasterization functions are in the `Buffer` class. The helper function are static functions in the `Buffer` class also, and take care of manipulating colors, blending pixels and runs of pixels.

Available paints:

- straight color: stored in a `std_color` structure. The structure contains the color in ARGB format and in separate components formats (both in `uint16_t` and `float` format) for quicker rasterization. Maybe i'll add some gamma correction someday.
- linear gradient: stored in a `lin_grad` structure. Can have any orientation, size, number of stops, repeat pattern. The current implementation even accepts transparency.
- radial gradient: stored in a `rad_grad` structure. Like the linear gradient, any orientation, sizes, number of stops,etc ...
- generic paint: stored in a `gen_color` structure; basically it's just a set of callbacks, so you'll have to write them. There's a function to put the "current" position to a point, to a x-coordinate, to a y-coordinate, to the next pixel in the line, and a function to retrieve the color. No particular example yet.

Available methods (all are scanline algorithms):

- `RasterShape`: computing the line's exact coverage, then the pixels coverage, then rasterizing runs of pixels. The slowest. The `quick` parameter selects a different method to move the scanline. Sadly, it's often slower than the non-quick method. Don't use.
- `RasterShapeA`: computing the line's exact coverage, by means of a set of increments/decrements, then the pixels coverage, then rasterizing runs of pixels. Faster than the previous. I think libart uses the same method.
- `RasterShapeB`: uses 4*4 supersampling and bit masks. Converts the bitmasks to the same pixel runs format as the 2 previous functions, so that the blending functions can be reused. Can be overall faster than the 2 other method, since for even-odd and non-zero fill rule, this method can deal with non-"good" polygons, thus sparing you a call to `ConvertToShape` (slow function).

Blending functions:

- composition/decomposion of colors in ARGB format: `Decomp` and `Recomp`. Their parameters are self-explanatory.

- blending of a pixel on another one: `Composite` and `CompositeOpaque` variant. The "opaque" version assumes the color is blended on an already opaque pixel.
- blending of runs of pixels: `RasterRun` variants (one for each possible paint).

3 From `Path` to `Shape` and back

The "tenth" release adds a function to recover a `Path` from a correct `Shape`: the `ConvertToForme()` function. There are 2 version of this function:

- the basic one:`void ConvertToForme(Path* dest);`
- the complex one:`void ConvertToForme(Path* dest,int nbP,Path* *orig);`

The difference lies in the output, *ie* the `Path` stored in `dest` upon return. In the basic case, the `Path` is just a polyline, meaning it's a set of contours. In the complex one, the function tries to reassemble segments of the contours in curve patches. For that reason, it requires the `Path` from which the segments come. To use this method, you must do the following:

- for each `Path` that will create segments in the polygon, use `ConvertWithBackData()` instead of `Convert()`. This will prepare the `Path`s and make reassembling segments possible.
- assign an ID to each `Path`, and use `Fill(dest,ID,...)` when transforming it in a polygon. This is needed to find where each segment in the resulting polygon will come from.
- do the operations you need on the polygon. Only `ConvertToShape()` and `Booleen()` preserve the information needed for the reassembly, so you must limit yourself to these 2 functions.
- call `void ConvertToForme(Path* dest,int nbP,Path* *orig);`, where `orig` is an array of size `nbP` giving the correspondance between ID and `Path`. A segment with ID k will be reassembled according to the `Path` in `orig[k]`.

4 Algorithmical remarks

4.1 Intersector

It's the usual Benley-Ottman sweepline intersector, with some rounding along the line of the Hobby algorithm; the windings are computed with a depth-first search in the graph. The rouding part is only "along the lines of", because i didn't really understand the implementation suggested by Hobby. The PolyBoolean seems truer to these suggestions, so check it out if needed. The Benley-Ottman uses an AVL tree and a binary heap queue, as it should.

That part should have a $O((n+m)ln(n+m))$ running time (where n is the number of points in the result polygon, and m the number of edges). This is quite optimal (i think), but remember that some non-"good" polygons have a "good" version where $m = n_0^2$ (n_0 being the number of points in the non-"good" original polygon).

The `Booleen` function uses the same intersector algorithm, only the winding computation and edge selection is different.

4.2 Rasterization

These functions have been written in a quite dumb way. Pixel manipulation ought to be severely optimized, vectorized or whatever. Also note that compiler ought to inline lots of the functions involved in this part.

4.3 `Path` handling

Also done very stupidly. The approximation of the bezier curves is done by recursive dichotomy. There is a maximum of 8 levels of recursion, feel free to increase it.

5 Before i leave

5.1 Rotating, translating,... polygons

Well, just apply the transformation to each element of the `pts` array of the `Shape` instance where the polygon is stored... the number of points is stored in `nbPts`.

A few caveats: if the transformation is an anti-transformation (for example: an axial symetry), all edges have to be reversed. Use the `Inverse()` function on each edge index, that is 0 to `nbEdge`-1. If the transformation is a nasty one, such as a whirl, you may end up with intersecting edges; a `ConvertToShape` will be needed. Of course, affine transformation are NOT nasty. You usually won't need to call `SortPoints` yourself, it's called by each function that requires the points in ordered form. Same thing for the `SortEdge` function, which sort the edges incident to each vertex in clockwise order.

5.2 Voronois

Are not implemented in the livarot version "eighth" and "ninth", but are in the previous ones. The `Shape` class changed slightly in the "eighth" version, and i still haven't rewritten the voronoi part.

It's back in "tenth", but still no clipping of diagrams against polygons, nor rasterization of these diagrams.

5.3 Justification

That part is just for fun. It's in livarot 'cause i used the polygon function to make justification in arbitrary shapes possible, but it has nothing to do with it... sadly, my poor understanding of how sourceforge.net file releases work had me delete that part. It's gone for now

5.4 What to include?

`LivarotDefs.h` contains the basic constants and general types. `Livarot.h` contains the includes for the appropriate classes.