

Bezigon Technical Reference

Versions 4.2.0 and 2.2.0 · 16 October 2026

About this document

Bezigon computes the Bezier handles of a path drawn with a FreeHand-style pen: the user clicks nodes, and the library decides where every handle goes. This document covers both versions of the library, the rules they follow, and how to build them into PhotoLine.

	4.2.0	2.2.0
Node types	Corner, Smooth, Tangent	Corner, Smooth
Library size	bezigon.cpp 515 lines, bezigon.h 169	bezigon.cpp 283 lines, bezigon.h 138
Extra functions	segment_is_straight, handle_axis	none
Corner and Smooth handles	identical to 2.2.0	identical to 4.2.0

On any path made only of Corner and Smooth nodes the two versions produce byte-for-byte the same handles. This was checked on 300,000 random paths and on every case in the permutation tables. Choose 4.2.0 if PhotoLine should offer Tangent nodes; choose 2.2.0 if it should not, since it is about half the size.

Two companion documents show every case as a picture: **Permutations - Corner, Smooth and Cusp** (both versions) and **Permutations - Tangent** (4.2.0 only).

Glossary

The same things have had several names during development. This document, the code and the permutation tables use these.

Term	Meaning
Node	A point on the path, with a position, a type and two handles.
Corner	A node placed with a plain click. Also called <i>sharp</i> . The path may turn abruptly here.
Smooth	A node placed with a modifier click. The path flows through it; its two handles are collinear.
Tangent	4.2.0 only. FreeHand's <i>connector point</i> . Each handle continues the chord on the other side of the node, so only its length is free.
Cusp	Not a fourth type: a Corner whose cusp flag is set because it was converted from a Smooth (or Tangent) node, either by the user or by a modifier close. Its handles lie along the chords to its neighbours and move independently.
Plain corner	A Corner that is not a cusp. It continues the arc of a Smooth neighbour. It comes only from a plain click.
In-handle, out-handle	The in-handle faces the previous node, the out-handle the next one. Both are stored as offsets from the node.
Chord	The straight line between two adjacent nodes.
Segment	The cubic Bezier drawn between two adjacent nodes.
Manual handle	A handle the user has dragged, or one a Tangent kept on becoming a cusp. No rule recomputes it.

Term	Meaning
Two-node loop	A closed path of exactly two nodes. Each node has the other on both sides.
Out-and-back	A path whose end node sits exactly on its start node, or any two nodes either side of a node that sit on one point.
PC, MC	Plain click and modifier click, as used in the permutation tables.

Files and requirements

File	Needed by PhotoLine	Contents
bezigon.h	yes	The whole public interface.
bezigon.cpp	yes	The rules. No other dependencies.
main.cpp	no	The demo: window, mouse, keyboard, drawing (EGE library).
demo_scenes.h	no	4.2.0 only. Example paths for the demo's keys 1 to 7, also used by the tests.
test_bezigon.cpp	no	The test program. Links against bezigon.cpp only.

- C++11 or later. Built and tested with g++ and clang under C++11 and C++20. Not yet built with MSVC.
- Standard library only: <vector> and <cmath>.
- Everything is in namespace bezigon. The version is in the macro BEZIGON_VERSION.
- No global state, no exceptions thrown, no I/O. Each computation allocates a few temporary arrays with one entry per node.

The data

Node

4.2.0:

```
struct Node
{
    double    x = 0.0, y = 0.0;
    NodeType type = NodeType::Corner;    // Corner, Smooth or Tangent
    bool      cusp      = false;
    bool      manual_in = false;
    bool      manual_out = false;
    double    hx_in  = 0.0, hy_in  = 0.0;
    double    hx_out = 0.0, hy_out = 0.0;
};
```

2.2.0 is the same except that type is a bool smooth:

```
struct Node
{
    double x = 0.0, y = 0.0;
    bool   smooth = false;
    bool   cusp   = false;
    bool   manual_in = false;
    bool   manual_out = false;
    double hx_in  = 0.0, hy_in  = 0.0;
    double hx_out = 0.0, hy_out = 0.0;
};
```

Field	Meaning
x, y	Position. Any units, any origin.
type / smooth	What the user chose. Draw the node's marker from this.
cuspid	Corner nodes only. True if the corner came from converting a curved node. A cuspid's handles lie along its chords; a plain corner continues its neighbours' arcs. Set by set_node_type / set_node_smooth and by a modifier close. make_node never sets it.
manual_in, manual_out	True once the user has dragged that handle. A manual handle is never recomputed, except that on a Tangent its direction still follows the rule and only its length is kept.
hx_in, hy_in	Offset from the node to its incoming control point.
hx_out, hy_out	Offset from the node to its outgoing control point.

An offset of (0, 0) means no handle. Only a side with no neighbour (the ends of an open path), and in 4.2.0 a Tangent's straight side, have none. Every other side carries a handle, lying along its chord when it has nothing else to follow.

make_node

```
Node make_node(double x, double y, NodeType type); // 4.2.0
Node make_node(double x, double y, bool smooth); // 2.2.0
```

A new node as created by a click. It sets position and type only; handles are filled in by compute_handles.

Close

```
enum class Close { Open, Plain, Modifier };
```

Plain and Modifier record how the path was closed: by a plain or a modifier click on the start node.

Path

```
class Path
{
public:
    std::vector<Node> nodes;
    Close close() const; // read only
};
```

The node list is public and the host edits it freely. The close state can only be read: it is set by close_path alone, because a modifier close also changes the start node, and doing both in one call means the two can never disagree. To reopen a path, copy its nodes into a new Path.

Options

```
struct Options { double handleFraction = 0.39; };
```

Each automatic handle is handleFraction times the length of its own segment. 0.39 is the measured FreeHand value. Every function that computes handles takes an Options, defaulted.

Functions

Every function ignores an out-of-range index and changes nothing. None of them throws.

close_path

```
bool close_path(Path& path, bool withModifier);
```

Closes the path and returns true, or returns false and changes nothing if there are fewer than two nodes. With the modifier, the start node becomes a Corner; if it was Smooth (or Tangent) it becomes a cusp. Call `compute_handles` afterwards.

Two nodes close into a loop of two segments. A plain close of two Smooth nodes draws an ellipse.

set_node_type (4.2.0) and set_node_smooth (2.2.0)

```
void set_node_type (Path& path, int index, NodeType type, const Options& = Options());  
void set_node_smooth(Path& path, int index, bool smooth, const Options& = Options());
```

Changes a node's type and recomputes. Always use these rather than assigning the field, because they also decide what happens to the handles:

- **To Smooth (or Tangent):** the node's handles are reset to the automatic ones and its cusp flag is cleared.
- **To Corner:** the node becomes a cusp, however it was drawn. Its handles lie along its chords. In 4.2.0 a Tangent's own handles survive the change, as manual handles; nothing else does.
- A type equal to the current one is ignored.

So any node toggles between smooth and the same cusp every time. A plain corner, which follows its neighbours' arcs, comes only from a plain click.

compute_handles

```
void compute_handles(Path& path, const Options& opts = Options());
```

Recomputes every handle that is not manual. Call it after any change to the node list: a node added, moved, inserted or deleted, a type changed by hand, a manual flag cleared, and after `close_path`. Calling it twice gives the same result as calling it once.

It is cheap. Measured on 4.2.0 with mixed node types: 4 microseconds for 100 nodes, 41 for 1,000, 0.5 ms for 10,000 and 6.5 ms for 100,000 (g++ -O2, one core). Calling it on every mouse move is fine.

drag_handle

```
void drag_handle(Path& path, int index, bool isIn, double x, double y,  
const Options& opts = Options());
```

Moves one handle tip to the absolute point (x, y), marks that handle manual, and recomputes, so neighbours that depend on it follow at once.

Node	What happens to the other handle
Smooth	Turned to stay collinear, keeping its length, and marked manual too (unless it has zero length).
Corner, cusp	Untouched. Each handle moves on its own.
Tangent (4.2.0)	Only the handle's length is taken from (x, y); its direction stays on the Tangent's axis. A handle can only be created where <code>handle_axis</code> gives a direction.

segment_is_straight (4.2.0)

```
bool segment_is_straight(const Path& path, int index);
```

True if the segment from node `index` to the next is drawn as a straight line, because both its handles lie on the chord. Export such a segment as a line. False if no segment starts at `index`. In 2.2.0 the host can make the same test: a handle lies on its chord when its cross product with the chord is zero and its dot product is between 0 and the chord length.

handle_axis (4.2.0)

```
bool handle_axis(const Path& path, int index, bool isIn, double& ax, double& ay);
```

For a Tangent node, the unit direction its handle on that side must point. False if the node is not a Tangent or has no neighbour to take the direction from. The demo uses it to decide which handle a Ctrl-drag pulls out.

The rules

Every automatic handle is `handleFraction` (0.39) times the length of its own segment, so only its direction needs deciding.

Smooth

The path follows the tangent of the circle through the previous node, this node and the next. A missing neighbour drops out of the formula, so an end node of an open path points along its one chord.

If the previous and next nodes are the same point there is no such circle. In a two-node loop the node points square to the chord, which draws an ellipse, as Xara does. Anywhere else nothing special happens: each handle lies along its own chord, as in FreeHand, until the nodes are moved apart.

Corner

Each side is decided on its own. If the neighbour on that side is Smooth, the handle continues its arc: the neighbour's direction reflected across the chord. Otherwise the handle lies along the chord, which draws the side as a straight line and still gives the user something to pull.

A Corner never follows a Tangent, even when the user has pulled a handle out of it: the pulled handle bends only the Tangent's own end of the segment. Nor does it follow a handle dragged on another Corner.

Cusp

A Corner made by converting a curved node. It follows none of its neighbours: each handle lies along its own chord, so the node is a point with a straight approach on each side. In 4.2.0 a Tangent's own handles survive the conversion.

Tangent (4.2.0)

Each handle continues the chord on the other side of the node: the out-handle points along *node minus previous*, the in-handle along *node minus next*. At an end of an open path the handle points along its own chord instead. The two handles are therefore not collinear in general, and a Tangent between two curves makes a notch.

A side has a handle only if the neighbour on that side already curves the segment (it is Smooth, or has a dragged handle facing this node), or if the user pulled one out. A handle on a straight segment would bend it, so toward a plain Corner the segment stays straight. All configurations are tabulated in *Permutations - Tangent*.

The modifier close

The start node is made by two clicks, and each of its handles behaves as if its click ended an open path. Closing with the modifier turns the start node into a Corner, a cusp if it was curved. As long as it is a Corner, its in-handle points straight back at the last node, whatever that node is doing; its out-handle follows the Corner or cusp rule.

Two-node loops

Two nodes can be closed. Each then has the other on both sides, which is the one case where a Smooth node takes the perpendicular. The results:

Nodes	Plain close	Modifier close
Smooth + Smooth	An ellipse.	A teardrop: the start node becomes a cusp.
Corner + Smooth	An ellipse: the corner continues the smooth node's arc.	A teardrop: the start node's in-handle reaches straight back.
Smooth + Corner	An ellipse.	A doubled straight line: the cusp and the corner both lie on the chord.
Corner + Corner	A doubled straight line.	A doubled straight line.

Coincident nodes

A neighbour sitting exactly on a node counts as absent for that side. Two nodes either side of a node that sit on one point (an out-and-back) leave that node with no circle; outside a two-node loop its handles simply lie along their chords. The library does not merge, weld or move nodes. Merging an end node into the start node is the host's decision (see *Closing by dragging onto the start node* below).

Integrating into PhotoLine

The library is a pure function from node positions and types to handles. PhotoLine keeps its own tool, drawing, hit testing and undo; it calls Bezigon at the moments listed below and reads the handles back.

Adding the code

1. Copy `bezigon.h` and `bezigon.cpp` into the PhotoLine source tree and add the `.cpp` to the build. No defines or flags are needed.
2. Include `bezigon.h` where the pen tool lives. Everything is in namespace `bezigon`.
3. Leave `main.cpp`, `demo_scenes.h` and `test_bezigon.cpp` out of the product. Keep `test_bezigon.cpp` buildable on the side: it is the quickest way to confirm a compiler or flag change has not altered the results.

Holding the path

The simplest arrangement: while the pen tool is active, it owns one `bezigon::Path`. Every user action edits that path and recomputes; PhotoLine's own path object is rebuilt from it for drawing, and the result is handed to PhotoLine's document when the tool commits.

The control points of segment i to $i+1$ are absolute positions built from the offsets:

```
P0 = (n[i].x,          n[i].y)
P1 = (n[i].x  + n[i].hx_out, n[i].y  + n[i].hy_out)
P2 = (n[i+1].x + n[i+1].hx_in, n[i+1].y + n[i+1].hy_in)
P3 = (n[i+1].x,          n[i+1].y)
```

A closed path has one more segment, from the last node back to node 0, built the same way. A two-node loop therefore has two segments, 0 to 1 and 1 to 0.

What to call for each user action

User action	Library calls	Host does
Plain click on the canvas	<code>nodes.push_back(make_node(x, y, Corner)), compute_handles</code>	Hit testing first: a click on an existing node or handle is not a new node.
Modifier click	Same, with <code>Smooth</code> (or <code>Tangent</code> in 4.2.0)	Chooses the key bindings.
Drag a node	Set <code>x, y</code> ; <code>compute_handles</code>	Never touch the handles yourself.
Click the start node	<code>close_path(path, modifier), compute_handles</code>	Decides the hit radius.
Drag the end node onto the start node	See the next section	Snap, merge and close.
Convert a node	<code>set_node_type / set_node_smooth</code>	Chooses the keys or menu.
Drag a handle tip	<code>drag_handle(path, i, isIn, x, y)</code>	Hit tests the tip.
Pull a handle out of a Tangent (4.2.0)	<code>handle_axis</code> for each side to pick one, then <code>drag_handle</code>	Decides which side the gesture means.
Release dragged handles	Clear <code>manual_in, manual_out</code> ; <code>compute_handles</code>	There is no library call for this.
Insert or delete a node	<code>Edit nodes; compute_handles</code>	See <i>Editing a closed path</i> below.
Undo, redo	Nothing	Snapshot the Path. It is a plain value type and copies safely.

Closing by dragging onto the start node

In the demo, dragging the end node onto the start node closes the path when the mouse is released, and holding the modifier at release gives a modifier close. This is host behaviour built from library calls, and PhotoLine can copy it:

1. While the end node of an open path of three or more nodes is dragged within the snap radius of the start node, set its position exactly to the start node's.
2. On release, if the end node sits exactly on the start node, remove it (`nodes.pop_back()`). It is the same point, and keeping it would leave a zero-length segment.
3. Call `close_path(path, modifierHeldAtRelease)` and `compute_handles`.

A path of three nodes closed this way becomes a two-node loop, which is the ellipse or teardrop described above. If PhotoLine does not want this gesture, nothing in the library depends on it; the click on the start node still works.

Editing a closed path

- Inserting or deleting interior nodes needs only `compute_handles`.
- Deleting node 0 of a modifier-closed path makes another node the start node, which the modifier close never touched. Rebuild instead: copy the remaining nodes into a new Path, close it again, and recompute.
- Changing the start node: copy the nodes in their new order into a new Path and close it again.

- Reversing direction: reverse the node order, swap each node's in- and out-handles and its two manual flags, and recompute. For a closed path keep node 0 first (reverse nodes 1 to n-1). The tests check that reversal gives mirrored handles on open and plain-closed paths.

Saving and loading

Save every field of every node, and the close state. To load, push the nodes, call `close_path` if the path was closed, then call `compute_handles`. Recomputing a loaded path changes nothing, because automatic handles are recomputed to the same values and manual ones are left alone.

One case needs care. A modifier `close` turns the start node into a cusp, but the user may later convert it back to Smooth. Calling `close_path(path, true)` on load would turn it into a cusp again. To restore it exactly, call `close_path` first, then copy the saved node 0 back over `nodes[0]`, then call `compute_handles`. This was checked: the restored path matches the saved one exactly.

Drawing and export

- Draw the node marker from type (or smooth). The demo draws a cusp like any Corner; PhotoLine may draw it differently, using the `cusp` flag.
- Every sharp node carries handles lying flat along its chords, so a polygon shows two per node. To reduce clutter, hide any handle for which `segment_is_straight` is true (4.2.0) or which lies on its chord (2.2.0).
- When exporting, write a straight segment as a line rather than a cubic. Its drawn shape is the same either way, but other editors will then recognise a polygon as one.

Coordinates

Any units and any origin. Every handle is a fraction of its own segment, so the results scale exactly with the drawing; the tests cover $1e-6$ to $1e12$. Y-up and y-down are both correct: mirroring the drawing mirrors the output exactly, and rotating, scaling or moving it does the same to the handles (a randomised test checks this). Moving a node, or changing its type, never changes the handles of nodes more than two places away.

Threads

There is no shared state. Different paths can be computed on different threads at once; one path must not be edited from two threads at once.

Upgrading from 4.1.0 or 2.1.1

Change	4.1.0 or 2.1.1	4.2.0 and 2.2.0
Close state	<code>Path::close</code> was a public member	Read with <code>close()</code> , set only by <code>close_path</code>
Fewest nodes to close	3	2
Cusp record	<code>born_curved</code> (4.1.0), <code>was_smooth</code> (2.1.1): how the node was drawn	<code>cusp</code> : whether the corner came from a curved node
2.2.0 only	Nodes built by hand; <code>close_path</code> returned nothing; <code>drag_handle</code> took a node	<code>make_node</code> ; <code>close_path</code> returns bool; <code>set_node_smooth</code> ; <code>drag_handle(Path&, ...)</code> recomputes

The behaviour changed in exactly four places, all agreed. Every case in both permutation documents was run against the original 4.1.0 code, and these were the only differences:

1. **Two-node loops** now exist; 4.1.0 refused to close two nodes.
2. **Out-and-back**: a Smooth node whose neighbours coincide now lies along its chords. 4.1.0 drew a perpendicular there; it now does so only in a two-node loop.
3. **Cusps**: a node drawn as a Corner, made Smooth and then a Corner again, is now a cusp. 4.1.0 returned it to the plain corner it was drawn as.
4. **Pulled Tangent handles** (4.2.0 only): a plain Corner beside a Tangent no longer continues a handle pulled out of the Tangent; it keeps its handle on its chord. In 4.1.0 the Corner's handle swung out to continue the pulled arc.

Of the Tangent configurations 4.1.0 could build, 100 of 102 give identical handles in 4.2.0. The other two are the pulled-handle change.

Checklist

- Nodes are created with `make_node`, never filled in by hand.
- `compute_handles` runs after every change to the node list.
- Types change only through `set_node_type` / `set_node_smooth`.
- Handle drags go through `drag_handle`.
- The path is closed only through `close_path`, and its return value is checked.
- A modifier-closed path is restored by closing, then copying node 0 back.
- Straight segments are exported as lines.
- `test_bezigon.cpp` passes when built with PhotoLine's compiler and flags.

Degenerate input

The library never crashes, never produces NaN or infinity and never loops, whatever it is given. Each of these cases is covered by a test.

Input	Behaviour
Empty path	Nothing happens.
One node	No handles. <code>close_path</code> refuses.
Two nodes, open	A straight segment; each node's handle lies along the chord.
Two nodes, closed	A two-node loop (see <i>The rules</i>).
A neighbour on top of a node	Counts as absent for that side.
Previous and next on one point	In a two-node loop: square to the chord. Elsewhere: handles along their chords.
Every node in one place	No handles anywhere.
Very large or very small coordinates	Tested from 1e-6 to 1e12. All tolerances scale with the local segment length.
Out-of-range index	Ignored by every function; nothing changes.

What the library does not do

It produces four numbers per node and nothing else. Drawing, hit testing, selection, marker shapes and colours, when handles are visible, snapping, merging nodes, undo and redo, saving, and key and mouse bindings are all the host's business.

Verification

Check	4.2.0	2.2.0
Named tests	79	44
Randomised property tests	18	8
Individual checks, all passing	2,150,377 (g++), 2,152,112 (clang)	1,010,577 (g++), 1,011,140 (clang)
Compilers	g++ and clang, C++11 and C++20, -Wall -Wextra -Wpedantic, no warnings	same
Address and undefined-behaviour sanitizers	clean	clean
Scripted demo input (stand-in graphics library)	41 checks pass	41 checks pass

Check counts differ between compilers only because g++ and clang evaluate function arguments in a different order, so the test generator builds different random paths from the same seed.

2.2.0 against 4.2.0. 300,000 random Corner and Smooth paths, with drags, conversions and all three close states: byte-identical output. Every case in *Permutations - Corner, Smooth and Cusp* also gives identical handles, types and flags in both.

4.2.0 against 4.1.0. Every case in both permutation documents was run against the original 4.1.0 code. The only differences are the four agreed changes listed under *Upgrading*. An earlier random comparison agreed: 131,802 ordinary paths containing Tangents were identical, and of the out-and-back paths that changed, every changed node was a Smooth or a Corner, never a Tangent.

The demo. Both demos have been run with the EGE graphics library on Windows. The screenshots in the permutation documents are drawn from the library's own output, styled like the demo. Mutation testing, which checked the tests themselves on 4.1.0 (81 of 84 deliberate faults caught), has not been repeated for 4.2.0.

Known consequences and open questions

Two-node loops containing a Tangent. They follow the rules unchanged, which gives odd shapes: a Tangent's handles continue the chord on the other side, and in a two-node loop that is the same chord, so they point away from the only neighbour. Not checked against FreeHand or Xara. PhotoLine may prefer to treat a Tangent in a two-node loop as a Corner.

A cusp straightens its own ends. Converting a node next to a Tangent into a cusp makes the segment between them straight at both ends, so the Tangent drops its handle on that side. Converting it back restores everything.

Every sharp node displays handles. See *Drawing and export* for how to hide them.

The handle fraction does not vary with the angle. 0.39 is exact for a 90-degree arc only.

Conversion discards dragged handles, except a Tangent's on becoming a cusp. This matches Xara.

Closing by drag merges the end node. That is the demo's choice, described above, not a library rule.