PLUGINS ENVIRONMENT TO EXTEND *Bezierspline.rb*

PROGRAMMING MANUAL

Credits and acknowledgments to:

- ***Carlos António dos Santos Falé**, for the Cubic Bezier algorithm (cubicbezier.rb, March 2005), which I incorporated as an extension to my macro and is used to illustrate this manual*

Foreword

While designing the script ***bezierspline.rb***, I realized that the mathematical algorithm could be isolated and abstracted from the rest of the environment, which is finally just dealing with the integration in Sketchup of the creation and the edition mode of your curve, seen as a set of points.

Actually, all we need is a method to compute

- a new set of points, i.e. the resulting curve (`crvpts`),
- from a set of Control Points (`ctrlpts`)
- with some parameters, such as `precision`

In other terms, if you have such a method behaving as

```
crvpts = compute_my_own_curve ctrlpts, precision
```

…then it should be possible to integrate it as an extension to *bezierspline.rb*.

Of course, as I did not want that the master macro *bezierspline.rb* had to be modified when integrating new algorithms, the extension must respect some naming conventions, as we'll see later on.

Also, as your curve algorithm may require a few more parameters than just a single 'Precision' factor, the extension environment provides ways to prompt the user for extra values.

Requirement: you need to have the script ***bezierspline.rb*** in your Sketchup Plugins. If you intend to offer multi-language support in your extension (which is always a good idea), you can use another macro-library I designed, ***LibTraductor.rb***.

Test utility: I did not publish one, but you can have a look at ***BZ__CubicBezier.rb***, which is precisely implemented as an extension, or at ***BZ__Divider.rb***, which is a test macro dividing a polyline into equal segments.

Compatibility: the extension environment has been tested with **Sketchup v5 and v6** (Pro and free versions, English and recent v6 French) on Windows XP and Windows Vista. I don't know however if it works on Mac

1. How to design an extension to *Bezierspline.rb*

1) Naming Conventions and Programming Model

In the simplest form, you just need to provide a **unique method to compute the curve** from a set of control points and some extra parameters. This method must be encapsulated in a Ruby module, itself written in a script file located in the Sketchup Plugins directory.

The following naming convention must be respected:

1) **The file name must begin with BZ__**, for instance, ***BZ__CubicBezier.rb***
2) **Within the file name, a particular constant <FILE NAME>__LIST must be defined and contain an array of one or several module names where you have your extensions**. The module names are defined as strings, with the right case, as coded in your file. For instance

   ```
   BZ__CUBICBEZIER__LIST = ["BZ__CubicBezier"]
   ```

   Normally you specify the list as an **array of strings**, but it works if you pass only a simple string. (for instance `BZ__CUBICBEZIER__LIST = "BZ__CubicBezier"`).
3) You must have a module with the specified name, in which you have defined:
   - **Your computing method, with the method name `bz_compute_curve`,** implementing the required interface (see below for specifications)
   - **A set of constants `BZ_xxx`** that specify the behavior of your extension

Here is a complete illustration, taken from file ***BZ__CubicBezier.rb***:

```
BZ__CUBICBEZIER__LIST = "BZ__CubicBezier"

module BZ__CubicBezier

   BZ_TYPENAME = "BZ__CubicBezier"
   BZ_MENUNAME = ["Cubic Bezier curve",
                  "|FR|Courbe de Bezier spline cubique"]
   BZ_PRECISION_MIN = 1
   BZ_PRECISION_MAX = 20
   BZ_PRECISION_DEFAULT = 7
   BZ_CONTROL_MIN = 3
   BZ_CONTROL_MAX = 200
   BZ_CONTROL_DEFAULT = 999
   BZ_CONVERT = "BZ__Polyline"
   BZ_LOOPMODE = -2

   def BZ__CubicBezier.bz_compute_curve(pts, precision)
      …..
      #Code to compute the resulting mathematical curve as an array of point <crvpts>
      …..
      crvpts          #Return value of the method is the array of points computed
   end

   ….. #Rest of your code

end #module BZ__CubicBezier
```

In general you'll make the top method `bz_compute_curve` fairly short, just calling another method in your module (or elsewhere) that does the actual mathematical calculation.

2) The constants parameters

A number of constants must be defined in your module so that the master macro *bezierspline.rb* knows how to handle your curve in Sketchup. Here are the conventions, with an example in italic:

```
BZ_TYPENAME = "BZ__CubicBezier"
```

This is the **unique code to characterize your curve**, defined **as a string**. It is **language independent** and will not appear explicitly to the Sketchup user. It must also be unique within the whole community of extensions to *bezierspline.rb*, because it is actually stored in the Sketchup object curve itself (so that it can be edited at a later stage). So I would encourage you to use a naming convention to make it unique (best is simply the module name) and avoid types like "MyCurve", which could be thought of by anybody else.

```
BZ_MENUNAME = ["Cubic Bezier curve",
               "|FR|Courbe de Bezier spline cubique"]
```

This is the **string** that will appear in the Sketchup menu "Draw", as well as in the contextual menus or other dialog interactions. It is the one visible to the end-user. It is a good idea to encode it for supporting **multi-language** (see manual on *LibTraductor.rb*).

```
BZ_CONTROL_MIN = 3
BZ_CONTROL_MAX = 200
BZ_CONTROL_DEFAULT = 999
```

These 3 parameters define the limits and default values for the number of control points that the end-user can enter in Sketchup when creating the curve, and when editing it later on. They must be of **type Positive Integer >= 2**

There are a few conventions:

- if `BZ_CONTROL_MIN` = 2, then your computation method will be called as the end-user enters the second control point. You should therefore be careful to support it in your calculation method.
- if `BZ_CONTROL_DEFAULT` = **999**, then you indicate that by default the curve will be created in 'Open-ended' mode, that is, in sequential order.

In the case of Cubic Bezier curves, you need at least 3 control points (otherwise, it would just be a segment), and I considered that you can comfortably do what you need with a maximum of 200 control points. It is also more natural to draw them in Open-Ended mode.

```
BZ_PRECISION_MIN = 1
BZ_PRECISION_MAX = 20
BZ_PRECISION_DEFAULT = 7
```

These parameters define the limits and default values for the Precision parameter, which normally determines the level of details when drawing the curve in Sketchup. It is dependent on your algorithm. Although they are **usually numeric**, there is actually no constraint on the type of this parameter, as long as it supports comparison with <=>.

If both min and max are equal, the user cannot change the Precision parameter.

`BZ_CONVERT = "BZ__Polyline"`

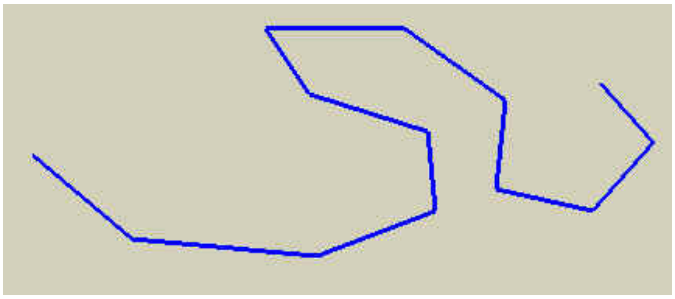
This parameter is a list of curve *typenames* that can be converted into your curve type. They are specified as a **list of strings separated by comma or semi-column**.

The principle of the conversion is to take the vertices of the origin curve and use them as control points to compute the destination curve.

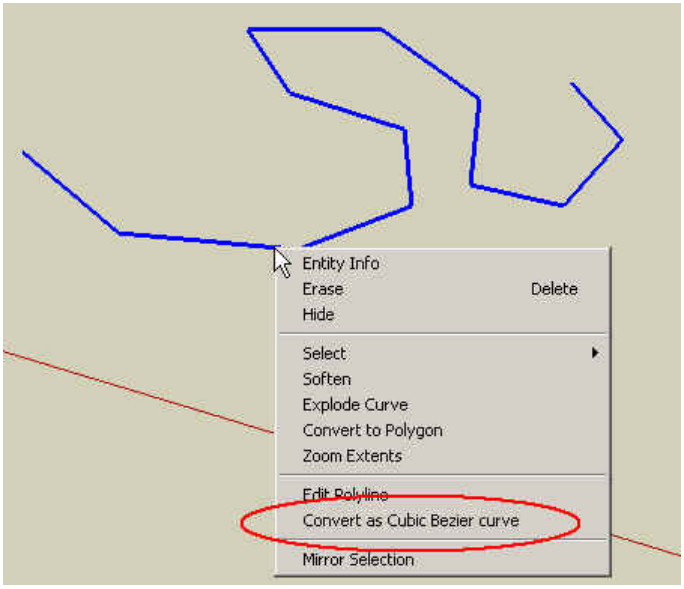
Remember that the *typename* of the curve is actually stored within the curve by *bezierspline.rb*.

The statement `BZ_CONVERT = "BZ__Polyline"` just says that any Sketchup curve of type "`BZ__Polyline`" can be converted into a Cubic Bezier curves.

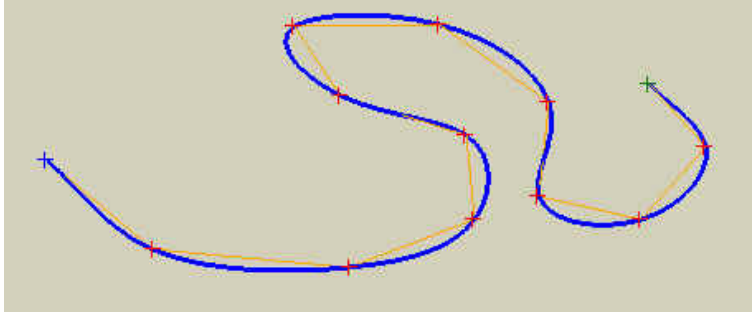
For instance here is a Polyline curve:



When you select the curve and show the contextual menu, you will have a menu item allowing you to convert the Polyline into a Cubic Bezier curve.



Here is the result, which you can further edit to move, add, delete the control points, or change the Precision parameter.



If you pass '*' in the list, then any Sketchup curve could be converted into your curve. This is the case for Polyline curves, which are defined with `BZ_CONVERT = "*"`. I did not do it directly for Cubic Bezier curves, as you usually need to prepare the curve so that you get a nicer result. So the rational order of things implies 2 steps:

Any Sketchup curve → Polyline → Cubic Bezier Curve

For convenience, I made the typename specified as regular expression, so that you rely on some naming conventions to offer conversion to several curve types.

BZ_LOOPMODE

This parameter indicates whether you support Loop mode for your mathematical curve. The valid values are:

`BZ_LOOPMODE = 0` **NO support of loop mode** (or your curve is already a loop)

`BZ_LOOPMODE = 1` **Support of loop mode with a single segment**

`BZ_LOOPMODE >= 2` **Support of loop mode with a 'nice' closure**. The value of the constant `BZ_LOOPMODE` indicates the default number of points for the loop closure (which can be changed by the user in the VCB, by typing a new number followed by 'c'). This is the value that will be passed to your calculation method in the argument `loop`. Note that if `BZ_LOOPMODE = 2`, then, the default value will be the Precision parameter.

Whenever `BZ_LOOPMODE` is positive, it is the responsibility of the computing method to close the loop, based on the value of argument `loop`.

Bezierspline.rb supports anyway a **built-in method to close curves via a segment or via a 'nice' loop**. In such case, you need to pass the constant as a negative number, and you don't have to bother with loops in your calculation method:

`BZ_LOOPMODE = -1` **loop mode with a single segment** is provided in standard

`BZ_LOOPMODE <= -2` loop mode with a 'nice' closure is provided by *bezierspline.rb*. Actually, I have implemented it as a portion of Bezier classic curve, as this curve has the good property to respect tangency at start- and end-point. The absolute value of `BZ_LOOPMODE` will be passed to your calculation method in the argument `loop`. Note that if `BZ_LOOPMODE = -2`, then, the default value will be the Precision parameter.

To be more concrete, here are the definitions I used in the 3 curves I provided:

- Bezier Classic → `BZ_LOOPMODE <= -15`
- Cubic Bezier → `BZ_LOOPMODE <= -2`
- Polyline → `BZ_LOOPMODE <= -1`

BZ_VERTEX_MARK

This parameter allows showing the vertices of the computed curves by default when creating or editing a curve (vertices are displayed as blue dots). The end-user can indeed show or hide the marks by the toggle key F5. The valid values are:

`BZ_ VERTEX_MARK = 0` Vertex marks are NOT shown by default

`BZ_ VERTEX_MARK = 1` Vertex marks are shown by default

3) The computing method

There are 3 valid forms, depending on which level of features you implement in your extension. *bezierspline.rb* will detect which form you use and call it appropriately.

```
crvpts = modulename.bz_compute_curve ctrlpts, precision
crvpts = modulename.bz_compute_curve ctrlpts, precision, loop
crvpts = modulename.bz_compute_curve ctrlpts, precision, loop, extras
```

It must respect the following programming interface:

- Name: it must be in the form ***modulename***.`bz_compute_curve`.
- Return Value: **an array of 3D points defining the computed curve**
- Arguments:
  - `ctrlpts` is the array of control points (at least 2 points)
  - `precision` is the Precision parameter which you have your own interpretation
  - `loop` is a numeric flag, indicating whether you should:
    - Close the loop by a nice curve (`loop >= 2`)
    - Close the loop by a segment (`loop = 1`)
    - No loop closure (`loop = 0`)
  - `extras` is a hash array of extra parameters (see section 2.2):

It is very important that your method always behaves correctly and never crashes in all situations. The macro *bezierspline.rb* does not check the return value and just displays the points computed. The reason is that it would not know what else to do in such situations.

It is also very important that the calculation is fast, as your method will be called when drawing or modifying interactively the curve.

4) Default value for `BZ_xxx` constants

For convenience, I have defined defaults if a constant is not explicitly defined, but I would encourage you to **always give them definition and not rely on defaults**.

`BZ_TYPENAME` → same as module name
`BZ_MENUNAME` → same as `BZ_TYPENAME`

`BZ_PRECISION_MIN` → **0**
`BZ_PRECISION_MAX` → **0**
`BZ_PRECISION_DEFAULT` → same as `BZ_CONTROL_MIN`

`BZ_CONTROL_MIN` → **3**
`BZ_CONTROL_MAX`→ **500**
`BZ_CONTROL_DEFAULT` → same as `BZ_CONTROL_MAX`

`BZ_LOOPMODE` → **0** (no support of loop mode)
`BZ_VERTEX_MARK` → **0** (no display of vertices by default)

2. A full example with the module Polyline

The extension **Polyline** is actually integrated in the file *bezierspline.rb*, but is actually coded like a normal extension (ClassicBezier as well by the way).

Here is the full code

```
module BZ__Polyline

  BZ_TYPENAME = "BZ__Polyline"    #unique symbolic name
  BZ_MENUNAME = ["Polyline"]      #kept the same name in English and French
  BZ_PRECISION_MIN = BZ_PRECISION_MAX = 1     #no precision needed
  BZ_CONTROL_MIN = 2              #allow to have a polyline with only one segment
  BZ_CONTROL_MAX = 300            #hope it is enough!
  BZ_CONTROL_DEFAULT = 999        #open end drawing
  BZ_CONVERT = "*"                #convert any curve made of at least 2 segments
  BZ_LOOPMODE = -1                #allow closing by segment done by bezierspline.rb

  #the calculation method just returns the polygon of control points
  def BZ__Polyline.bz_compute_curve(pts, numseg)
    pts
  end

end #End module BZ__Polyline
```

3. More advanced features

1) Organizing modules, constants and methods

It is important that you understand how the macro *bezierspline.rb* proceeds to find out about the extensions.

a) **It first look for scripts files following the naming convention BZ__xxxx.rb** in the Plugins directory of Sketchup, as well as in a subdirectory **BZ_Dir**, if present.

The benefit of putting your extensions in the *BZ_Dir* folder is that they are not loaded automatically by Sketchup at start time, but explicitly by the macro *bezierspline.rb*.

b) **It then loads each of these files if not already loaded and look for a constant defined at file level <filename>_LIST**.

c) **It then looks for the `BZ_xxx` constants and method `bz_compute_curve`, within the strict scope of the modules specified in the list**. Actually, it does not care too much if the module is in the same file or in another file, as long as it is loaded in the Ruby environment.

It may happen that you want to **code several extensions within the same file**, but that the algorithms are close enough that you don't want to duplicate the critical code. You can do it, in several ways.

Let's take the example of a file *BZ__SuperBezier.rb*, and let's assume you provide two classes of SuperBezier curves, A and B. In file *BZ__SuperBezier.rb*, you would write:

```
BZ__SUPERBEZIER_LIST = ["SuperBezier_A", "SuperBezier_B"]

Module SuperBezier_A
   BZ_TYPENAME= "SuperBezier_TypeA"
   #etc… for other BZ_xxx constants

   def SuperBezier_A.bz_compute_curve(pts, precision)
      crvpts = SuperMaths.compute pts, precision, 'A'
   end
end

Module SuperBezier_B
   BZ_TYPENAME= "SuperBezier_TypeB"
   #etc… for other BZ_xxx constants

   def SuperBezier_B.bz_compute_curve(pts, precision)
      crvpts = SuperMaths.compute pts, precision, 'B'
   end
end

Module SuperMaths
   def SuperMaths.compute(pts, precision, type)
      #your complex algorithm goes here, handling types A and B
   end
end
```

Actually, the module `SuperMaths` could well be in another file, if you have in *BZ__SuperBezier.rb* a `require` statement to load this other file.

2) Extra Parameters

In some occasions, you may need much more parameters to specify the curve computation than just the single Precision parameter.

The macro *bezierspline.rb* provides some ways to prompt the user for extra parameters, usually with a dialog box (but not always, as you could take them from a configuration file for instance or from a crystal ball!).

The important things to know are:

- **Extra parameters are defined as a hash array**. This is convenient as you can store any number of values of any type, identified by a key. This is also the format that is used by *LibTraductor* dialog boxes if you wish to use them.
- **Extra parameters are stored within the Sketchup curve**, as attribute of the Sketchup entity. This means that you can retrieve them when you Edit a curve
- **There are 3 particular moment when Extra parameters will be requested to your module**:
  - When creating a curve (mode = '**N**')
  - When converting a Sketchup curve to your curve type (mode = '**C**')
  - When the end-user want to change parameters by pressing the TAB key (mode = '**E**')

The interface to your callback must respect the following rules:

```
extras_out = modulename.bz_ask_extras mode, extras_in, ctrlpts, precision
```

where

- Name: it must be in the form ***modulename***`.bz_ask_extras`.
- Return Value: **a hash array containing the parameters or `false` if the user pressed Cancel**. In this latter case, the operation will be simply cancelled if it is a curve creation or a curve creation, and nothing will be changed if it is invoked by the end-user pressing TAB.
- Arguments:
  - `mode` is a character flag, indicating the event
    - `mode = 'N'` → at creation of the curve
    - `mode = 'C'` → when converting a curve to yours
    - `mode = 'E'` → when the end-user pressed TAB
  - `extras` is a hash array of extra parameters:
  - `ctrlpts` is the array of control points (be careful, it ma be empty when called at creation!). It is passed for convenience, in case you wish to compute the limits of some parameters.
  - `precision` is the Precision parameter which you have your own interpretation. It is also passed for convenience.

I produced an extension ***BZ__Divider*** that computes a curve with respecting equal spacing between points, while making sure that the vertices are located on the polygon of control points. For this, I needed at least to ask the user for the interval value between points. This will serve me to illustrate how you can use extra parameters.

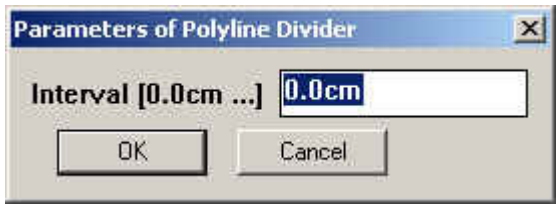
First, I wrote the method for prompting the user:

```
TTH_Title = ["Parameters of Polyline Divider",
             "|FR| Parametres du Diviseur de Polyline"]
TTH_Interval = ["Interval", "|FR| Intervalle 1"]

….

def BZ__Divider.bz_ask_extras (mode, extras, pts, precision)
   unless @dlg
      @dlg = Traductor::DialogBox.new TTH_Title
      @dlg.field_numeric "Interval", TTH_Interval, 0.cm, 0.cm, nil
   end
   return (@dlg.show extras) ? @dlg.hash_results : false
end
```

The first time it is called, it will create the dialog box object and display it with the default value (here `0.cm`). I put a lower limit to `0.cm`, but no upper limit. The initial display will thus be:



The user will then enter a value, say `25cm`.

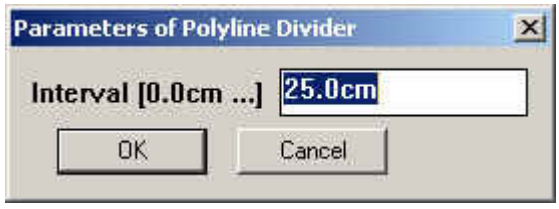
The calculation method `BZ__Divider.bz_compute_curve` must be in the complete form

```
crvpts = BZ__Divider.bz_compute_curve ctrlpts, precision, loop, extras
```

When it is called by *bezierspline.rb*, the hash array argument `extras` will contain the value of the interval: `extras = {Interval => 25.0cm}`, which you can extract by the statement:

```
interval = extras["Interval"]
```

When you later edit the curve and press TAB, you will retrieve this value and can change it.



You can design more complex dialog boxes of course, depending on your needs. I suggest you refer to the documentation about ***LibTraductor.rb***.

Now, you are absolutely not forced to prompt the user via *LibTraductor* utilities. You can use your own way, for instance with Web dialogs or whatever other mechanisms. You are not even obliged to prompt the user. *Bezierspline.rb* makes no hypothesis on what you do with your method, as long as you return a **hash array** or `false`.

However, keep in mind that *Bezierspline.rb* requires that the argument `extras` is an hash array in order to store it as an attribute of the curve (via `entity.set_attribute`)

3) Developing and testing your extensions

1) Developing your extension

When you develop, you would probably use the Ruby console in Sketchup. After modifying your script file(s), you would manually reload them by a statement such as `load 'myfile.rb'` typed in at the command prompt of the console.

Note that this will work for all changes except those changes related to:

- Module names
- The callback methods (this is why you should put the 'real' code in other methods.
- Values of BZ_xxx constants
- Adding modules for new extensions within the same file, or in another file

In such cases, you must close Sketchup and restart it. This is because *bezierspline.rb* has already loaded your module and processed the constants and method determination.

2) Common Errors

Here are the few typical errors you may encounter when coding extensions

a) Wrong case for module names

Be careful with the case for module names. For instant, if you have an extension provided as BZ__SuperBezier.rb, which contains a module `SuperBezier` then

`BZ__SUPERBEZIER_LIST = ["SuperBezier"]` ***is a Valid form***

`BZ__SuperBezier_LIST = ["SuperBezier"]` ***is also a Valid form***

But

`BZ__SUPERBEZIER_LIST = ["SUPERBEZIER "]` ***is not valid***

`BZ__SUPERBEZIER_LIST = ["SuperBEZIER "]` ***is not valid either***

b) Wrong case for constants

All constants `BZ_xxx` must all be defined in UPPERCASE. Otherwise, they would be simply ignored.

c) Several versions of the file in the Sketchup Plugins directory

Keep in mind that Sketchup automatically loads all the *.rb* files present in the Plugins directory (order is very often alphabetic, but it may depends on what each script does with `require` or `load` statements. So if you have several copies of your same module (in files respecting however the BZ__ convention), the effect may not be predictable, as:

- In principle, the *bezierspline.rb* macro would not find the extension module since the LIST constants is probably not based on the file name
- However, the module will be defined in Ruby, and so the constants.

As a result, you'll likely get a mix of your two files.

So if in your code editor you sometimes do a "Save as Copy", be careful to do it in another directory than the Sketchup Plugins folder or the BZ_Dir folder.

d) Optional arguments in `bz_compute_curve` *or* `bz_ask_extra`

In principle, you should be clear and consistent about the arguments of these callback methods.

There is anyway a check on the number of arguments in order to adapt to your form of callback, but for whatever reason, there is a bug in the interpreter concerning the `arity` method when there are optional arguments.

So be careful NOT to have optional arguments in the callbacks, such as:

```
def bz_compute_curve pts, precision, loop=0, extras=nil
```

The valid forms for callbacks are:

```
def bz_compute_curve pts, precision  #don't care with loop mode and extras
def bz_compute_curve pts, precision, loop       #don't care with extras
def bz_compute_curve pts, precision, loop, extras     #general form
```

and:

```
def bz_ask_extra mode, extras, pts, precision #single form
```