TOPOSHAPER ISOCONTOUR

Service API

FREDO6 – 25 MAR 2015

1. Introduction

TopoShaper Isocontour is a standalone script which **calculates a terrain from a set of isocontours** (isocontour = curve of constant altitude).

Since TopoShaper **v2.1**, the algorithm can also be invoked **as a service from an external Ruby script**, including from within an interactive Tool.

The API is implemented in the module **F6_TopoShaper**, as the standalone method **F6_TopoShaper.api_isocontour_calculation**. Therefore, you should protect the invocation of the API with a section:

```
if defined?( F6_TopoShaper.api_isocontour_calculation)
  …
end
```

The method does NOT generate any geometry. It does NOT invoke a `model.start_operation`. The method does silently the calculation of the terrain from the specified contours and options, and then returns the geometry information describing the terrain surface, hull and skirt.

2. Invoking the API

a) Syntax

The calculation method takes the following form

```
results = F6_TopoShaper.api_isocontour_calculation(lst_contours, hsh_options=nil)
```

Arguments:

- `lst_contours`: a list of arrays of 3D points. Each array is therefore a single contour.
- `hsh_options` (optional): a hash array of options (*see below*)

Return value:

- `nil`, if there is no contour provided
- `a structure` containing the information about the contour or the error details when there is an error (*see next paragraph*)

b) Error protection

The API is protected against errors (`begin..rescue`). If an error happens, the return value is a structure with a single field `:error` containing the Exception. So you must analyze the results return value:

```
e = results.error
if e
    #e is the exception, e.message the error message
end
```

c) Options

The options are passed as a Hash array **hsh_options**, each option being defined by a symbol. If an option is omitted, it takes its default value.

- **:nx** ➔ grid dimension in X (i.e. number of cells in direction X) - (default **50**)
- **:ny** ➔ grid dimension in Y (i.e. number of cells in direction Y) - (default **50**)

  Note: if one of the dimensions **nx** or **ny** is omitted or passed nil, it is calculated from the other dimension.

- **:plane_normal** ➔ the 'vertical vector' for the terrain. If omitted, it is indeed taken as **Z_AXIS**

- **:option_hilltop** ➔ force hilltops and basins to be flat or round (default **round**). To make them flat, pass the value **:flat**; for round leave it **nil** or **:auto**

- **:notify_proc** ➔ optional callback method to be notified about the progress of the calculation. The calculation can take long, so this method may be useful if you wish to display the progress to the user. The callback method takes 2 arguments:
  - **time**: delta of time since the calculation started, in second
  - **message**: displayable text indicating the current step

  For instance

```
notify_proc = proc { |time, message| puts "Time = #{time} msg = #{message}" }
```

  The output to the Ruby console would be

```
Time = 0.0 msg = Analyzing Contours
Time = 0.194011 msg = Computing the Enveloppe
Time = 0.552031 msg = Generating the Grid
Time = 0.768044 msg = Determining the Zones
Time = 1.092062 msg = Interpolating Altitudes
Time = 1.265072 msg = Extrapolating Altitudes
Time = 1.282073 msg = Calculating Mesh
Time = 1.325076 msg = Calculating Boundaries
```

Example of options, where **ny** will be calculated:

```
hsh_options = { :nx => 100, :option_hilltop => :flat, :notify_proc = notify_proc }
```

d) Inspecting the results

If the operation completed with no error, the return value is a structure containing the following fields:

- **:error** ➔ an Ruby Exception if there was an error, otherwise **nil**
- **:lst_cell_info** ➔ A list of cell information describing the terrain

  Each element of the list is an array with 2 elements
  - **pts**: the ordered points of the cell (4 or 3 points)
  - **diago**: a Boolean indicating whether the cell quad should be triangulated

  The following example illustrates how to exploit the information, for instance if you wish to generate the terrain mesh (quads and triangles):

```
results.lst_cell_info.each do |cell_info|
   pts, diago = cell_info
   if diago
      face1 = entities.add_face(pts[0..2])
      face2 = entities.add_face(pts[2], pts[3], pts[0])
   else
      face = entities.add_face(pts)
   end
end
```

- **: skirt_panels** ➔ A list of quads (or triangles) describing the skirt.

  The following example illustrates how to exploit the information, for instance if you wish to generate the skirt geometry:

```
results. skirt_panels.each do |pts|
   entities.add_face(pts)
end
```

- **: hull_projection** ➔ A list of points describing the Hull projected at altitude 0.

  It is therefore a planar close loop.

  The following example illustrates how to exploit the information, for instance if you wish to draw the projected hull:

```
entities.add_curve results.hull_projection
```

- **:nx** ➔ grid dimension in X used for calculation
- **:ny** ➔ grid dimension in Y used for calculation

  Note: these parameters are in the results because they could have been calculated.

e) About the Point coordinates

The points of the terrain, skirt and hull are returned in the same axes as the coordinates of the input contours. Similarly, the optional parameter `plane_normal` must be given in these local coordinates.

It is therefore up to the calling method to perform the proper transformation from the Sketchup model geometry when specifying the points of the contours.

3. A full Example

The following example shows a full example, where the terrain is computed from a Group containing a set of Sketchup curves and then drawn as a group in the model:

```
def api_example

      #Extracting the group from selection
      selection = Sketchup.active_model.selection
      g = selection[0]
      return unless selection.length == 1 && g.instance_of?(Sketchup::Group)

      #Computing the curves
      hcurves = {}
      edges = g.entities.grep(Sketchup::Edge)
      edges.each do |edge|
            curve = edge.curve
            hcurves[curve.entityID] = curve if curve
      end
      return if hcurves.empty?

      #Computing the list of contours from the curve
      tr = g.transformation       #to get coordinates at top level
      lst_contours = hcurves.values.collect do |curve|
            curve.vertices.collect { |vx| tr * vx.position }
      end

      #Calling the API
      notify_proc = proc { |time, message| puts "Time = #{time} msg = #{message}" }
      hsh_options = { :nx => 50, :notify_proc => notify_proc }
      results = F6_TopoShaper.api_isocontour_calculation lst_contours, hsh_options

      #Testing the results
      if results.error
            puts "error #{results.error.message}"
            return
      end

      #Drawing the terrain
      model = Sketchup.active_model
      model.start_operation "API Topo"
      grp = model.active_entities.add_group
      gent = grp.entities
      results.lst_cell_info.each do |cell_info|
            pts, diago = cell_info
            if diago
                  f = gent.add_face(pts[0..2])
                  f = gent.add_face(pts[2], pts[3], pts[0])
            else
                  f = gent.add_face(pts)
            end
      end
      model.commit_operation
end
```