SKETCHUP RUBY API

About the Evolutions of the API for Script writers

10 Oct 2013

by Fredo6

This short note is a contribution to the Sketchup Unconference in Barcelona. Overall, I observe that the new versions of Sketchup are putting a good focus on the extensibility of Sketchup via plugins, in particular by making the programing framework more robust and better documented. This is positive.

There are surely plenty of evolutions that can be considered to integrate Sketchup into other applications, extend it to perform useful functions in the professional world and become itself a more powerful 3D design tool. Such major evolutions should for instance deal with:

- ***More advanced integration with the web for data and user interaction**, via available techniques popular for Internet programming. Web dialogs are noticeably unexploited today.*
- ***Inclusion of a Nurbs library (and other primitives like Delaunay triangulations)** to be used by plugins. There are open-source packages available and a native support in Ruby mapping a C-library would be great to achieve acceptable performance and also standardize Sketchup around 3D-packages selected by Trimble.*

However, to remain on solid earth, I will concentrate on some simple evolutions of the Sketchup Ruby API, which I think can be achieved in coming versions.

1. Plugin Management

EWH and the Sketchucation Plugin Store are progressively taking over and resolving the pain point of script installation. There remain a few details however:

a) **Persistent references to menu**: this goes along with the effort put by Trimble on guidelines for writing plugins (naming conventions, mandatory namespacing via modules, …). This is needed to organize the Plugin menus by author.

b) **An official reference environment for managing plugin ancillary files**: several plugins generate or use other files and there is no convention for their 'best' and 'safest' locations. At least SU14 has defined a root directory in the <user> environment where to store plugins and temporary files.

2. Interactive Tool environment

This is about the Sketchup::Tool class and methods, which manages the GUI and interactions with the user:

a) **View.draw_image**: draw an png / jpg image on the viewport, with possibility of adjusting size. In many cases, this would allow to have dynamic button palette (instead of drawing with open GL primitives) and to be more illustrative in the GUI.

b) **View.draw GL_POLYGON**: today, this method only works for **convex** polygons, in 2D or 3D. So, if you want to draw a general polygon (like for instance, mimic a face), you have to make a decomposition into triangles. A native C method would be faster and easier to use. Possibly introduce another code so that GL_POLYGON remains simple and fast.

c) **VCB and alpha characters**: A large number of characters are ignored passed through to Sketchup when they are assigned to a shortcut (so the tool is switched). This limits the formalism that can be used with the VCB. The ideal implementation would be that **all alpha characters are accepted and kept at VCB level whenever the user has already typed a digit or an arithmetic sign**.

d) **Standard cursors**: it would be useful that Sketchup ships with some predefined cursors, such as Hourglass, various types of arrows, etc… This will help to have a uniform design across plugins. Possibly, a superposition of png cursors would also be helpful (for instance to add a small + to a cursor).

e) **Painting edges**: If you draw over an edge of the model (via view.draw), for instance to highlight an edge in color, you don't have a clear overwriting of the edge. The workaround is to slightly offset the segment in 3D so that it is located 'in front' of the edge. This implies some code which may impact performance when drawing a large number of lines. It would be good to have a method (coded in C from the Ruby code below) that would be much faster.

```
#Calculate the point slightly offset to cover the edges
def G6.small_offset(view, pt, pix=1)
        pt2d = view.screen_coords pt
        ray = view.pickray pt2d.x, pt2d.y
        vec = ray[1]
        size = view.pixels_to_model pix.abs, pt
        size = -size if pix < 0
        pt.offset vec, -size
end
```

f) **Interrupt calculation**: not sure if this is feasible, but it would be extremely useful if SU could provide a mechanism to be notified of a user interruption during a long calculation. This could be based on a couple of methods:

   – **notify_interrupt()**: standard method of the Sketchup::Tool class to be invoked when there is an interruption.

   – **UI.interrupt?(tool)**: to be called by the calculation routine, at certain statement [since calculation are often composed of smaller blocks of calculation]. This method just checks if the user has typed a key or clicked in the viewport. If so, it will call <notify_interrupt> for the specified tool argument. Return true if interruption requested, false otherwise.

g) **view.draw_point**: correct the side effects and provide more marks

h) **View.draw_text**: drawing of text is currently limited. We would need:
   - to set the text color
   - to control the font size
   - to calculate the length of a given text (for justification), unless we also get a method to draw a text with justification in a reference box, possibly multi-line.

i) **Where is the mouse**: a simple method that can be called from any place in the code of the Tool class, returning the coordinate of the mouse.

j) **Redo notification**: currently, the onCancel method is fired whether the user has done an Undo or a Redo. In some cases it does matter to know which action was done and this is not trivial to find out from Ruby.

3. Manipulating and Generating Geometry

This applies to the inspection, creation and modification of Sketchup entities (edge, faces, groups, components, etc…).

a) **Mapping of EntityID**: each entity has a unique EntityID that has the good idea to be preserved across Undo or abort operation. The issue here is that there is no direct way to retrieve the ruby object from the entityID. The only way is to traverse the model until you find an object with the right entityID. In interactive script, where you provide rollback and modify functionality, you frequently have to perform an *abort_operation* and then a recreation of geometry to reflect the new parameters.

b) **Built-in Weld**: today, to transform a sequence of edges into a curve, you need to create a group, do an *add_curve* in this group and explode the group. This is cumbersome and has the side effect that some entities (edge and adjacent faces) may change as object.

c) **Bulk methods**: these are methods that take a list of entities instead of a single one, for instance to assign edge properties, material, etc… The objective here is to increase performance.

d) **Reference between mesh and generated geometry**: the fastest method to generate large geometry of faces is undeniably ***entities.fill_from_mesh*** performed in an empty group. Typically:

 mesh = *Geom::PolygonMesh.new*

 loop to create polygons via *mesh.add_polygon*

 create a new group, via *entities.add_group*

 Create the geometry via *group.entities.fill_from_mesh mesh, …*

 The drawback of this approach is that, if you have to then make some changes to the generated geometry (edge properties, material, …), you have to rebuild a reference between the original polygons and the geometry. So far, I have noticed that the list of entities of the group maps the list of polygons in the order they have been created. So the first polygon created is the first face in *group.entities.to_a*. This is fortunate, but possibly a coincidence, so that it may not be preserved in further versions of Sketchup.

 What would be helpful is a mechanism to cross-reference the generated faces and the polygons. It could be a hash table passed to *entities.fill_from_mesh*, or any other method.

e) **Duplicate groups and make unique**: The problem has already been covered in a separate discussion: http://sketchucation.com/forums/viewtopic.php?f=180&t=52014#p470425. This is really a big issue and not easy to handle correctly from the Ruby code.

f) **SuperHide**: in interactive scripts, you may have to hide some faces and edges and draw in the viewport to simulate the new geometry (for instance in Joint Push Pull, you have to erase the face to reflect a negative offset where the face is 'pushed'). The usual method is to achieve the result is:

 - Create a dummy layer and make it non-visible
 - Change the layer of the faces or edges to this dummy layer, noting the original layer for each one

This is cumbersome and error-prone. A more direct method would be helpful.

g) **Top-level transformation for groups / components**: top-level transformations is the transformation between the actual entities embedded in groups / components and the viewport. This is useful to simulate the geometry by view.draw() in the viewport. Currently, top-level transformations can be determined easily when the group / component is picked (via InputPoint or Pickhelper transformation methods). However, in other cases, you have to traverse the whole model to reconstruct recursively the top-level transformation.

4. Inspectors, message boxes and Web Dialog boxes

This applies to the message dialog boxes, inspectors and to the web dialogs

a) **Focus on SU viewport**: it would be extremely useful to have control on the focus to the Sketchup viewport, at least to be able to force the focus on the viewport. For instance, when you want to create a companion web dialog box that is used to set interactively some parameters. When a parameter is set, you could give focus to the SU viewport by script so that the user is not misled, wondering why a shortcut does not work, not realizing that the focus is not on the SU viewport. Same situation when the script uses, or interacts with some inspectors, like the material chooser.

b) **Material chooser on Mac**: it would be good that the current material shown in the chooser is accessible via *model.materials.current()*, as it is on Windows.

c) **'new' Material**: it would be useful that when the current material is not included in the model, the <material> object returned by *model.materials.current* behaves normally and does not crash SU when you assign it to entities.

d) **More configurable Messagebox**: it would be a plus to have a form of message box where you can pass the label of buttons as arguments. Although we can use a web dialog box instead, the issue is that only native message boxes are really modal on MacOS.

e) **Dialog box for picking directory**: this is simply missing and it would be damned helpful.

f) **Browser info for web dialog boxes**: a method to get the minimum information of the browser environment (browser version, size of screens, etc…) would be helpful. This would allow to adjust the code when the HTML is generated from Ruby. Currently you have to create a web dialog and use some JavaScript to get these basic info for further usage.

g) **Synchronization on Mac for web dialog**: Also a well-documented problem, with no straightforward workaround.

5. Built-in Geometrical calculation methods

This is a call for providing more routines performing calculations in 3D space. The objective is that, if implemented in C under the cover, it would give better performance.

a) **Intersection of 2 segments**: calculate the intersection of 2 segments given by their pair of points. Return the intersection point or nil.

b) **Intersection of a line and a segment**: Return the intersection point or nil.

c) **Barycenter**: Again, just to speed up calculation. There are several cases, given a list of points:

- **Straight barycenter**: simple average of the points
- **Curve barycenter**: points are given in sequence. The average is weighted by the length of segments. This avoids that duplicated points count twice and reflect the weigth barycenter of a physical object materialized by the curve.
- **Centroid**: exact centroid of a closed loop. Normally, this should also compute the area.

d) **Area**: calculate the area of a polygon (concave or convex). Area is only available for true face objects.

e) **Multi-linear average**: typically used for interpolating a value at a given point <pt> from the values given in a set of points based on their distance to <pt>. This is a lengthy calculation, which would surely be much faster if written in C.

For instance, with a set of 2 points:

$$val = ((d1 * val2 + d2 * val1) / (d1 + d2))$$

For a set of 3 points:

$$val = ((d1*d2 * val3 + d1*d3 * val2 + d2*d3 * val1) / (d1*d2 + d1*d3 + d2*d3))$$

The general case with many points implies to calculate a lot of products and sums which is not very performant in Ruby.

f) **Intersection of a line and a triangle (or polygon) in 3D**: nothing complex, but lengthy in Ruby because the method Geom.point_in_polygon_2D only works with 2D coordinates. So you have to transform the polygon to the XY plane.

g) **Proximity**: Given a point <pt>, find the closest point to a segment and to a curve. Again, mainly for performance reasons.

h) **Geometry primitives**: basic primitives for 2D and 3D geometry would also benefit from being written in C (by Trimble or linking with a package): Bezier, spline curves, triangulation, etc….