Entities#intersect_with demystified

SLBaumgartner
February 2014

Entities#intersect_with is one of the most poorly documented methods in the SketchUp Ruby API. The doc entry for this method is vague, overly terse, gives arguments needlessly confusing names, and misstates the return value! This is unfortunate, because intersect_with is actually an extremely powerful method. Performing the same operations using Ruby code instead of this compiled C method would be extremely slow. It can be employed to do some marvelous things, but it can also create massive confusion if you don't understand how it works.

So here is my version of how it should have been documented, followed by discussion of how to use it:

Basic documentation

edge_arr = ents.intersect_with(recurse, ents_trans, dest_ents, dest_trans, hidden, with_ents)

where

- edge_arr is an Array of Edges created by the intersection. It will be empty if none were found (contrary to the API docs, the method never returns nil)
- ents is an Entities collection, one set to be intersected
- recurse is a Boolean
- ents_trans is a Transformation associated with ents
- dest_ents is an Entities collection where results will be added
- dest_trans is a Transformation associated with dest_ents
- hidden is a Boolean
- with_ents is an Entity, comma-separated list of Entities, Array of Entities, Group, or ComponentInstance - the other set to be intersected.

Entities#intersect_with creates Edges corresponding to the intersections of Faces in ents with Faces in with_ents. The resulting Edges are added to Entities collection dest_ents and are also returned in an Array. This returned Array eliminates the need to search dest_ents for the new Edges, which can be a difficult task if dest_ents was not originally empty. Note that intersect_with only creates Edges. If you want associated Faces, you will need to follow up by invoking Edge#find_faces on them.

If recurse is true, Faces from Groups or ComponentInstances nested inside ents will be tested recursively; if false, they will be ignored. Groups or ComponentInstances nested inside with_ents are always included regardless of the value of recurse[1].

[1] I don't know whether this is by design or is a bug.

Note, also, that the method behaves as if these nested collections were exploded (recursively) in each collection before testing against the other one. It does not find intersections among nested Entities within the same argument collection and does not place the Edges it finds into nested collections in dest_ents. There is no mechanism provided to limit the nesting depth that will be recursed; it is all or nothing.

If hidden is true, hidden Geometry in ents and with_ents will be included, otherwise hidden Geometry is ignored. This applies equally whether individual Entities in the collections were hidden or if the object that owns the Entities collection (a Group or ComponentInstance) was hidden entirely. It also applies regardless of whether the layer containing the Entities is hidden.

Discussion

Because intersect_with must test each Face from ents against each Face from with_ents, the computational effort grows as the product of the number of Faces in the two collections (computer science n-squared). For large, complex Entities collections it can take a considerable amount of time. You can reduce the time by culling out Entities that clearly can't intersect, which requires effort proportional to the number of Entities (computer science n). Depending on the complexity of the test and the numbers of Entities, doing this culling first in Ruby code and then intersecting the reduced collections could be faster than letting the C code of intersect_with deal with the original multiplicative count.

To find meaningful intersections, the locations of all of the Vertices of the Entities must be expressed in the same coordinate system during the calculation. This is an area where the general operation of the method affords power that will only occasionally be needed in practice but can create very strange results if it is not understood. Since you will not be able to follow the operation of intersect_with unless you thoroughly understand SketchUp's concepts of coordinate system, transformation, context, and parent, I will start by discussing them.

A *context* is a collection of objects that are simultaneously accessible for editing in SketchUp. For example, the model as a whole is one context. If you have created a Group or ComponentInstance, it defines a new context nested within the model context. In the model context you can manipulate the Group as a unit, but you cannot alter or edit any of its internal Entities. When you open a Group for edit, all the Entities in its context are then available to edit, but Entities outside the context are not. While the Group is open for editing, the Group's context is called the *active context* and its Entities collection is the model's *active entities*.

The context in which another context is nested is called its *parent*. SketchUp contexts can be recursively nested as deeply as you wish. The model is the ultimate root parent of all other contexts. When you define a Group or ComponentInstance, its context has as parent the context that was open at that time.

Every context has a local coordinate system in which the locations of all its Entities are specified. The topic of coordinate systems and transformations could fill another entire essay, so for brevity I'll stick to just the essential ideas. In a nutshell, if you say something is located at the point [x, y, z], a coordinate system is what gives meaning to those three values. The values express distances to the location from a particular origin point using a particular scale along three particular axis directions. The origin, scale, and axes are what constitute a coordinate system. A transformation is what converts those three location values into their equivalents in another coordinate system with potentially different origin, scale, and axes. It is important to understand that a single location can be described using different triples of values in different coordinate systems.

In the model, the "root" coordinate system is the familiar set of red, green and blue model Axes with their origin and unit scale. Within a Group, locations are in the Group's local coordinates, relative to its own axes, origin, and scale. The Group's transformation specifies how to map locations from the Group's local coordinates into its parent's coordinates. A Component's Entities are actually owned by the associated ComponentDefinition and expressed in its local coordinates. But, unlike a Group, each ComponentInstance carries its own Transformation that tells how the ComponentDefinition's coordinates are mapped into the ComponentInstance's parent's coordinates.

The definition of intersect_with includes Transformations associated with ents and dest_ents, but nothing for with_ents. Therefore, it must necessarily work with the position locations of with_ents verbatim. The implication is that the calculation of intersections must take place in the coordinate system used by with_ents' locations, that is, the coordinate system of these Entities' parent. If you like, you could think that intersect_with implicitly uses an identity transformation for with_ents, though that has the same effect as not bothering to transform with_ents.

If with_ents is an Entity or Array of Entities, its locations are already given in its parent's coordinates. If with_ents is a Group or ComponentInstance, the Group or CI has a transformation by which its Entities are transformed to the coordinates of the parent of the Group or CI. The intersect_with method applies this transformation implicitly, allowing you to pass a Group or ComponentInstance as well as a single Entity or Array of Entities. While intersecting, all of them employ the coordinate system of the parent from which they were obtained. In particular, this means that they all *must* have the same parent else strange Edges will result, though I don't know whether SketchUp tests for this. Below I will refer to with_ents' parent's system as the *intersection coordinates*.

So, if the calculation is performed in intersection coordinates, the Entities from ents must be transformed to intersection coordinates first. That is the function of the ents_trans argument. The method applies this transformation to all locations in each Entity of ents before testing for intersections with Faces from with_ents. It

creates a new Edge, still in intersection coordinates, for each intersection that it finds.

This usage of ents_trans implies that it maps ents' local coordinates to intersection coordinates. This will naturally be true if ents belongs to an object with the same parent as with_ents. But you must be careful if you snatch ents from an object nested in some other parent! I'll say more about this later.

Finally, intersect_with transforms the new Edges using the *inverse of dest_trans* and adds them to dest_ents and the returned array. Why the inverse? Because dest_ents also came from some parent. The method needs to put the new Edges back into the same coordinates as the rest of that context, which might not be the same as the intersection coordinates!

To transform in the opposite direction, you use the inverse of a given Transformation. Rather than making you generate the inverse yourself, which would be potentially confusing and easy to forget, the API expects you to pass the transform that converts dest_ents coordinates to intersection coordinates, and it internally does the inverse for you. And once again you need to be careful if dest_ents' object was nested in a different parent than with_ents.

The API intentionally separated the transformation arguments from ents and dest_ents to give you more power over how these Entities collections are interpreted. They did not provide the same flexibility with respect to with_ents, possibly because doing so would add nothing to the power of the method (I'll explain why later) and would risk even greater confusion to developers.

To make all this more concrete, here's a basic example. Suppose ents is the Entities collection from a Group of interest whose parent is the model. Further suppose that with_ents is another Group you created in the model, and that you want to add the intersection Edges into the model's own Entities as loose Geometry. Since with_ents' Group comes directly from the model context, the intersection coordinates are the same as the model's coordinates. So, ents_trans must be the Transformation of the ents Group because that is what places the ents Group's Entities into the model coordinate system (the model is ents' Group's parent). Dest_ents is the model's Entities collection, and since the intersection coordinates are already model coordinates, dest_trans must be an identity Transformation.

```
ents = Sketchup.active_model.entities
g1 = ents.add_group
e1 = g1.entities
t1 = g1.transformation
…add Entities to e1
g2 = ents.add_group
e2 = g2.entities
… add Entities to e2
```

```
t2 = g2.transformation
idtr = Geom::Transformation.new #no arguments means create identity trans

int_arr = e1.intersect_with(false, t1, ents, idtr, false, g2)
```

What if you want to merge the intersection into either of the Groups? To do so, simply use that Group's Entities as dest_ents and the Group's Transformation as dest_trans:

```
int_arr = e1.intersect_with(false, t1, e1, t1, false, g2) # results added to g1
or
int_arr = e1.intersect_with(false, t1, e2, t2, false, g2) # results added to g2
```

Remember that intersect_with is only adding Edges, so even if these Edges could define new Faces in g1 or g2, the Faces will not have been created. There is no possibility of recursively generating new intersections with new Faces implied by the Edges created by the first intersection. One pass with the initial Geometry is all you get!

Next suppose you want to form intersections with a ComponentInstance rather than with a Group. This case is slightly more complicated because a ComponentInstance does not in itself contain any Entities. Rather, its associated ComponentDefinition owns the Entities, defined in the ComponentDefinition's local coordinates, and the ComponentInstance owns a Transformation that tells how to map those coordinates into the CI's parent. So, for ents you use the Entities collection from the ComponentDefinition, and for ents_trans you use the Transformation from the ComponentInstance.

```
# assume ci is the ComponentInstance of interest.
ents = ci.definition.entities
ents_trans = ci.transformation
```

Now for a first hint of the power and subtlety of intersect_with. Because you supply ents_trans (in contrast with how intersect_with assumes it should use the Group's transformation when with_ents is a Group), there does not have to be any actual ComponentInstance placed in the model using that transformation! It is the transformation that *would have been used* to place an instance in a particular location, orientation, and scale. You can, for example, generate intersections with a rotated version of a Component without actually performing the rotation in the model. You can grab a ComponentDefinition out of the model's definitions collection and intersect its Entities with objects wherever you wish in the model!

Here's another example of a powerful usage: suppose you want to slice a ComponentInstance, ci, at a particular Face, much the way a SectionCut does, and save the slice as a new ComponentInstance.

```
parent_ents = ci.parent.entities

ents = ci.definition.entities
ents_trans = ci.transformation

cut_face = parent_ents.add_face (#appropriate corners or edges…)

cut_gp = parent_ents.add_group # it would be good to give the group or ci a name
cut_inst = cut_gp.to_component
dest_ents = cut_inst.definition.entities

inst_arr = ents.intersect_with(false, ents_trans, dest_ents, ents_trans, false, cut_face)
```

Because I used ents_trans for the value of dest_trans, this method call puts the intersection Edges into dest_ents in the same locations as the actual intersections between ci and cut_face. That is, if you later drop instances of ci.definition and cut_inst.definition at the same place, they will align as in the original cut.

Now, as promised, some discussion about handling different parents. As already noted, the Transformations that you obtain directly from a Group or ComponentInstance serve to map its Entities into its parent's coordinates. What if the parent is itself a Group or ComponentInstance, i.e. if the parent's coordinates are not model coordinates? What if ents, dest_ents, and with_ents all have different parents?

What you must do in such situations is to build up the multi-stage transformations that take ents and dest_ents coordinates into the intersection coordinates, that is, with_ents parent's coordinates. This requires finding a parent context that contains all three sets of Entities. Call this the *shared root context*. Since the model is the ultimate root of all nested contexts, it is always possible to find a shared root context, though it might not be necessary to go all the way back to the model context.

To build a multi-stage Transformation, concatenate (in SketchUp Transformations, operator *) the Transformations from stage by stage through the parents until you reach the shared root context. For instance if g1 is nested within (parented by) g2, which is in turn parented by the model, then the transformation from g1 coordinates to model coordinates is

```
g1m = g2.transformation * g1.transformation
```

Due to the representation used in SketchUp, transformations concatenate right-to-left, so g1m first takes g1's locations to g2's coordinates, and then takes those to g2's parent's coordinates.

So, suppose we want to generate the intersection of g1's Entities (use them as ents), nested in g2, with g3's Entities (use them as with_ents), nested in g4, and put the results into g5's Entities (use them as dest_ents), which is itself directly in the model. The coordinates of g3 are not model coordinates, nor are they the coordinates of the parent context of g1 or g5! What to do!?

First, realize that the shared root context in this example is the model context. Also realize that g3's parent context is g4, that is, when used as with_ents, its entities will be implicitly transformed to g4's coordinates before starting the intersection. The transformations from the three required contexts to the shared root are thus:

g1r = g2.transformation * g1.transformation
g3r = g4.transformation
g5r = g5.transformation

Since when g3 is used as with_ents, we can't specify a transformation for it, we have to bring the other Entities into its parent's coordinates. Fortunately, this is easily accomplished by concatenating the inverse of its transformation to the shared root onto the other two transformations:

g3_inv = g3r.inverse

ents_trans = g3_inv * g1r
dest_trans = g3_inv * g5r

This example shows why it is really not necessary to pass a transformation for with_ents. You can manipulate ents_trans and dest_trans to achieve the same effect. Since this entire example is an advanced special case, it is better to simplify the interface for more typical situations.