ShaDe for SketchUp©

Advanced User's Guide

Author	Manuela Ruiz Montiel
Date	April 24, 2012
Version of ShaDe	2.3

Contents

0. Introduction 3
1. The rules 5
2. The shapes 6
3. The grammar 13
4. The execution of the grammar onto a design 14
5. Saving all together: the project 16

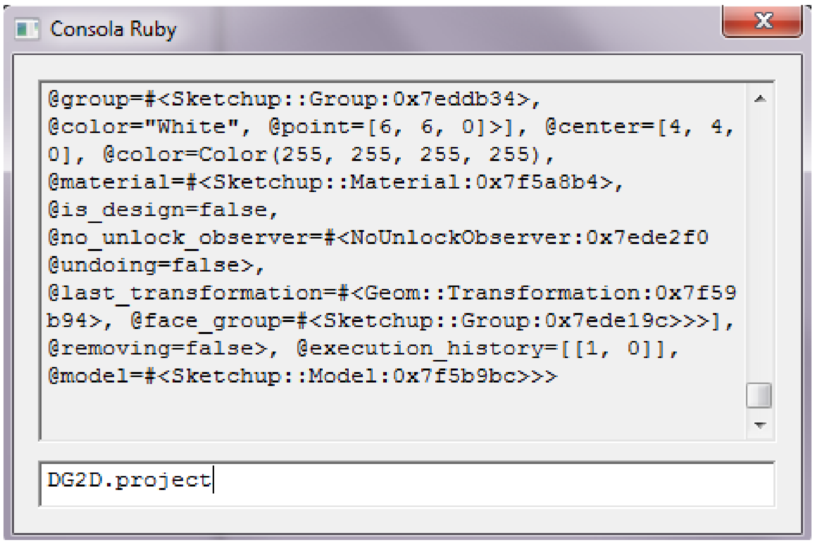
0. Introduction

Google SketchUp is a 3D modelling software that combines a tool-set with an intelligent drawing system. Along with its Ruby API, it can be extended and customized to satisfy particular and concrete needs, that may go from automation of common tasks to the creation of personal drawing tools. This is managed through the creation of Ruby scripts that will work as plugins.

ShaDe for Sketchup is one of them. The aim of this plugin is to offer the user a way of creating, editing and experimenting 2D shape grammars .

Provided certain knowledge about the formalism of shape grammars, the purpose of this document is to offer a complete, detailed technical reference about the internal logic structure of the plugin: what are its different entities, how do we represent them and, of course, how can them be manipulated.

There are two main ways of manipulating the plugin structures. The first and obvious way, aimed to the standard users, is through the possibilities of the SketchUp (SU in advance) graphic interface. A manual on this topic is covered in the document **ShaDe - User's Guide**. The second way, aimed to advanced users, is by entering commands into the Ruby console that comes available with SketchUp. If we do not see it, we can open it through the **Window -> Ruby Console** option.



```
@group=#<Sketchup::Group:0x7eddb34>,
@color="White", @point=[6, 6, 0]>], @center=[4, 4,
0], @color=Color(255, 255, 255, 255),
@material=#<Sketchup::Material:0x7f5a8b4>,
@is_design=false,
@no_unlock_observer=#<NoUnlockObserver:0x7ede2f0
@undoing=false>,
@last_transformation=#<Geom::Transformation:0x7f59
b94>, @face_group=#<Sketchup::Group:0x7ede19c>>>],
@removing=false>, @execution_history=[[1, 0]],
@model=#<Sketchup::Model:0x7f5b9bc>>>
```

```
DG2D.project
```

Screenshot 1: The Ruby Console

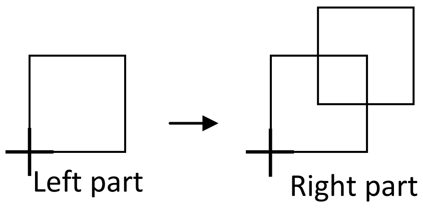
In the next sections, all the different entities of the plugin will be explained with a table in which the information below is included:

1. **What** is the entity
2. The main features or **hypothesis** made about the entity
3. **How** do we **represent** it
4. **What** can we **do** with it?

Note that this document is not a code reference, that is, not every methods and attributes of every class are included. Actually, it offers a general understanding of the code and the methods we can use to manipulate the entities directly from the RC. For a complete code reference[1] please see the HTML documentation.

[1] In addition, for a reference on how to deal with SU objects, please visit http://code.google.com/intl/es-ES/apis/sketchup/docs/index.html

1. The rules

Rule	
What is it?	This entity represents a rule of the current grammar. A graphical representation of a rule can be the following: 
Main Features or Hypothesis	• Rules are general, that is, they substitute the left shape with what it is on its right part • The left shape of a rule cannot be empty • They have always a cross in both parts, that points the origin relative to the respective shape. This allows to identify the spatial relation between the left and right shapes.
How is it represented?	The main attributes of a rule are the following: • **rule_id**: an internal rule id • **alpha**: a **LabelledShape** representing the left shape • **beta**: a **LabelledShape** representing the right shape
How can we get it?	By using the method **ShadeUtils.get_rule(rule_idx)**. If the specific index is out of range, a message appears in the RC. We can get rules by means of other methods. Please see the **Grammar** table
What can we do with it?	Given a **LabelledShape** called **shape**, we can change the rule parts: • **rule.alpha = shape** • **rule.beta = shape**

2. The shapes

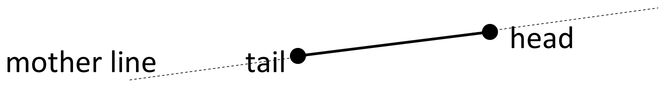
LabelledShape	
What is it?	This entity represents a **labelled shape.**
Main Features or Hypothesis	• Shapes are always created in the coordinates origin. Later, we can change its transformations by means of the SketchUp tools • The shapes are made of **Segments** and **Labelled points.** • The shapes we are going to deal with must always have at least **three** distinguishable points, in order to compute triangles. A distinguishable point is either a labelled point or an intersection point, that is, a point shared by two or more segments. In fact, we can consider an intersection point as an special kind of label, with the value "Intersection" • The segments are always **maximal lines** • The segments and the points are stored inside "ordered collections of ordered collections". This means that we cluster the segments or the points according to keys: • In the case of segments, they are clustered according to their line descriptors. All segments sharing the same line descriptor are a cluster. See **LineDescriptor** to understand how the line descriptor keys are ordered • In the case of points, they are clustered according to their labels. See **Label** to understand how the label keys are ordered Then, the clusters themselves are also ordered w.r.t. ordering relations established over segments and points. See **Segment** for more information
How is it represented?	The main attributes of a general shape are the following: • **s:** a collection of segments, divided by layers. For example, the segments in Layer0 are stored in s[“Layer0”] • **p:** a collection of labelled points, also divided by layers, as s
How can we get it?	See **RuleLabelledShape** and **CurrentLabelledShape**
What can we do with it?	Given a LabelledShape entity, let us name it **shape**, we can manipulate it with the following methods through the RC: • shape.add_label(point, value, layer): adds a **Label** on the specified point with the specified value, in the specified layer • shape.refresh_from_info(edges, points_and_materials): given an array of edges (SketchUp objects) and an array of pairs [point, material] (SketchUp objects), updates the

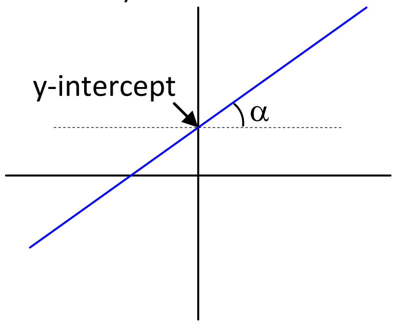
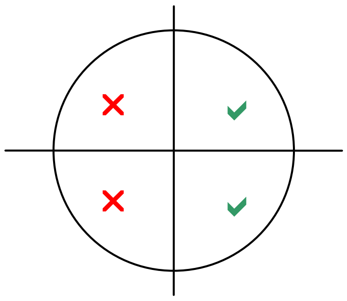
	internal representation of the shape, erasing the previous one • shape.shape_expression(other_shape, exp_type, focus): krishnamurti's algorithm for computing shape expressions: exp_type specifies the expression type, that is: "Intersection", "Union", "Difference", "Subshape" or "Equal". “focus” can take the values "Segments" or "Points", and it tells the algorithm to focus in the segments or in the points to perform the operation. • shape.save(path): Saves the shape in the specified path, into a .txt file • shape.load(path): Loads the shape from the specified path, from a .txt file • shape.paint(): Paints the shape in the SketchUp canvas

RuleLabelledShape < LabelledShape	
What is it?	This entity represents a **rule labelled shape,** that is, a labelled shape that is part of a rule**.**
Main Features or Hypothesis	• Once a shape is attached to a rule, this association cannot be changed, that is, we cannot move shapes over the set of rules
How is it represented?	The main attributes of a rule shape are the following: • **shape_id:** an internal id • **host_rule_id:** the id of the rule to which this shape belongs • **host_rule_part**: the part of the rule to which this shape belongs. It can be "Left" or "Right" • **group**: SketchUp object used to draw the shape
How can we get it?	Once we have a **Rule** object, we can access its shapes: • **rule.left** • **rule.right**
What can we do with it?	Given a RuleLabelledShape entity, let us name it **shape**, we can manipulate it with the following methods through the RC: • shape.transform(t): Given a 2x3 homogeneus matrix (translation, rotation or scaling), transform the shape according to it • shape.refresh: Erases and paints the shape • shape.refresh_from_entities(entities, transformation): Refresh the shape from the specified collection of SketchUp entities, that are affected by the specified transformation • shape.erase: Erases the shape from SketchUp canvas

CurrentLabelledShape < LabelledShape	
What is it?	This entity represents the **current labelled shape**, that is, the one that is generated by the application of rules.
Main Features or Hypothesis	• There is only one instance of this class in our plugin
How is it represented?	The main attributes of the current shape are the following: • **group**: SketchUp object used to draw the shape • **pi:** a permutation of the collection p that gives the indices of it in ascendant order of the number of labels of each type
How can we get it?	Please see **Execution**.
What can we do with it?	Given a CurrentLabelledShape entity, let us name it **shape**, we can manipulate it with the following methods through the RC: • shape.refresh • shape.erase

Point	
What is it?	It is a point of the space
Main Features or Hypothesis	The points we are going to deal with are always 2D points, that is, their third dimension is always zero
How is it represented?	When a point is needed, we can create it by means of the next code: p = Point.new(x, y)

Segment	
What is it?	It is a line segment, described by its tail point, its head point and its mother line 
Main Features or Hypothesis	• The tail is always lesser than the head, that is, tail.x < head.x **or** tail.x = head.x and tail.y < head.y (that is, we use a lexicographic ordering of the points) • If two segments are collinear, that is, they are in the same mother line, then we can compare them. This is done by the relation: SegmentA < SegmentB iff headA < tailB
How is it represented?	The main attributes of a segment are the following: • **tail**: the point (SU object) representing the tail • **head**: the point (SU object) representing the head • **line_descriptor**: the mother line on which the segment lays
What can we do with it?	Suppose we have a Segment object, let us call it **segment**, we can manipulate it with the following methods through the RC: • segment.tail=(new_tail) • segment.head=(new_head) • segment.coincident?(point): Returns true if the specified point is inside the segment • segment.overlap?(other_segment): Returns true if the segments overlap • segment.collinear?(other_segment): Returns true if the segments are collinear

LineDescriptor	
What is it?	It is the representation of a line, by the sine of the angle formed w.r.t. the x-axis and the intercepting point with the y-axis (or the x-axis if the line is vertical) 
Main Features or Hypothesis	• If the line is vertical, then the y-intercept is a x-intercept point, that is, with the x-axis • The sine that we use is always in the first or four quadrant, that is:  So the behaviour of the sine is the similar to the slope one, since the signs of the sine in these quadrants are the same of those of the slope. This is possible because of the relative ordering of the tails and heads of the segments for whom we compute the mother lines. • We order the lines with a lexicographic criterion, first comparing the sines (slopes) and then comparing the intercepts.
How is it represented?	The main attributes of a line descriptor are the following: **sine:** the sine of the angle formed w.r.t. the x-axis **intercept**: the intercepting point with the y-axis, or with the x-axis if the shape is vertical
What can we do with it?	Suppose we have a LineDescriptor object, let us call it **line_descriptor**, we can manipulate it with the following methods through the RC: line_descriptor.satisfied?(point): Returns true if the specified point satisfied the line equation

Label	
What is it?	It is just a value that can be associated with a point
Main Features or Hypothesis	The values are compared by a lexicographic ordering of its names. These names will be those of the materials of SketchUp or maybe "Intersection", if the label is going to mark an intersection point
How is it represented?	**value:** the value of the label. It can be a colour name (please see the User’s Guide to check which colour names are available) or the String "Intersection"
What can we do with it?	Suppose we have a Label object, let us call it **label**, we can manipulate it with the following methods through the RC: label.name(): returns the value name

3. The grammar

Grammar	
What is it?	A Grammar is a sorted collection of **Rule** entities plus an axiom, that is, a **LabelledShape** that will work as an initial **current shape**
Main Features or Hypothesis	A grammar always has at least one rule The axiom is by default a cloned shape of the left part of the first rule
How is it represented?	The representative attributes of a grammar are: **rules**: a sorted collection of **Rule** entities **axiom**: the initial **LabelledShape** of the **Design** **saved**: true if the current status of the grammar has been saved onto a file
How can we get it?	By using the method **ShadeUtils**.**get_grammar()**
What can we do with it?	Given a Grammar entity, let us name it **g**, we can manipulate it with the following methods through the RC: **g.add_rule(rule):** adds a **Rule** entity at the end of the rules collection **g.remove_rule(i):** removes the **Rule** in the i-th position in the rules collection **g.load(path)**: loads the grammar specified in the path, with an extension .gr2 **g.save(path)**: saves the grammar into the specified path

4. The execution of the grammar onto a design

Execution	
What is it?	Is an entity that gathers all the necessary objects in order to perform the grammar onto the **current shape**
Main Features or Hypothesis	• The **current shape** has always to comply with the present constraints
How is it represented?	The representative attributes of an execution entity are: • **grammar**: the grammar to be performed • **current_shape**: the current shape • **constraints**: present **Constraint** objects in the execution • **goals:** present **Goals** objects in the execution • **show_labels**: true if we want the labels to appear in the design • **execution_history**: triples of [rule_id, transformation, shape], ordered by application time (the first in the list are those that were first applied). Each pair means that the **Rule** with id rule_id was applied to the **shape** with the specified transformation • **file_axiom**: true iff the axiom is loaded from a file instead of being attached to the left shape of the first rule
How can we get it?	By using the method **ShadeUtils**.**get_execution()**
What can we do with it?	Given an execution, let us call it exe, we can use the following methods: • **apply_rule(rule_id, transformation):** if possible, applies the rule with id rule_id to the current shape. The transformation can be specified or empty. In the first case, the algorithm tries to apply the rule with the transformation. In the second case, it randomly choose a possible transformation that makes the current shape comply with the present constraints • **possible_transformations(rule):** Returns the possible transformations which with the specified rule can be applied • **add_constraint(constraint):** adds the specified **constraint**, just in case there is not another **constraint** of the same kind (analogously for goals) • **remove_constraint (constraint_name):** removes the constraint with the specified name (analogously for goals) • **brothers():** returns all the pairs [rule_id, transformation] than can be applied to the current shape • **apply_rule_random():** applies a random rule to the current shape • **apply_n_rules_random(n_rules,**

	show_backtracking_steps): applies a number (n_rules) of rules to the current shape, after resetting it. If the second argument is set to true, then we will see the steps that the backtracking algorithm is doing • **reset:** resets the **current shape**, that is, back to the axiom • **undo:** undoes the last execution

5. Saving all together: the project

Project	
What is it?	The Project is a wrapper for the **Execution** entity, and provides the extension of saving it into a file or loading from it. What saves is the **Grammar** along with the present **Restrictions** in the execution, all in one single file
How is it represented?	The main attributes of a Project are the following: • **execution**: the **Execution** entity wrapped by the Project • **title**: the **Project** title • **path**: the path in which the Project was previously saved • **saved**: true if the current Project has been saved onto a file
How can we get it?	By using the method **ShadeUtils**.**get_project()**
What can we do with it?	Given a project entity, let us call it p, we can manipulate it through the following methods from the RC: • **p.load(path)**: loads the project specified in the path, with an extension .prj • **p.save(path)**: saves the project into the specified path • **p.execution=(new_execution):** sets the **Execution** of the project to new_execution • **p.set_title(new_title):** sets the title of the project • **p.set_path=(new_path):** sets the path of the project, so the next time it will be saved into new_path. • **p.saved=(saved):** sets the saved status of the project to true or false