Stencil: Brief introduction to model, language and explore

Joseph A. Cottam

August 28, 2009

1 Introduction

Welcome to the super-alpha documentation for Stencil. This are always changing with Stencil (being alpha software), so please update your copy of Stencil and this document regularly. The upside of such flux is that I might be able to help you with your particular problems. Feel free to contact me (jcottam@indiana.edu) if you are having problems or think a particular feature would be useful. Thank you for trying Stencil! I hope this document shows you enough to help you start making you own stencils.

If you have downloaded the JAR file, instructions on its use are in the 'Explore' section.

2 Model

Before we dive into the language itself, lets talk about the Stencil system and conceptual models.

2.1 System

Stencil is geared towards creating visualization components based on declarative descriptions. The current implementation uses an interpreter to with a stencil (e.g. program written in the Stencil language) to wrap the Piccolo scene-graph library. The resulting component asks for three things from a host program: a stencil to execute, a space to draw on (e.g. a JPanel) and some data. The interpreter constructs appropriate data structures based on the stencil and handles all the data processing coordination once data has been converted to streams of tuples. As such, the stencil component insulates the host program from the details of the graphics library and many of the requirements of the visual construction.

2.2 Data Description

Stencil components need data. All data in the Stencil system is represented as streams of tuples. A stream is a sequence of data. It may be unbounded, it may be infrequently updated, it may never contain anything! Streams contain tuples. Tuples are positionally significant groups of data. Coordinates in geometric spaces (like X,Y,Z) are a common set of tuples. The X value is always first, Y is second and Z always third. This is the interface standard for Stencil. All data must enter the system as streams of tuples.

2.3 Visuals

There are two primary visual sets: Layers and Glyphs. Glyphs are visual primitives, like lines or circles. Stencil is currently 2.5D, and thus only works in terms of 2D glyphs. Glyphs have properties that may be set, though the exact properties depends on the glyph being used. All glyphs have ID, X, Y, Z and IMPLANTATION properties. Supplemental properties can also be created for glyphs, allowing meta-data to be stored with the glyph. Glyphs of a layer with the same type will all have the same meta-data fields defined, and thus can be represented as a tuple.

Glyphs are grouped into layers. Layers group glyphs that are derived in a similar fashion from the incoming data. Layers also provide a search capability based on ID (so all glyphs on a layer must have unique IDs, but glyphs on different layers may have the same ID). Layers may be though of as a set of glyphs where each glyph has a unique ID.

2.4 Mapping and Rules

Transformation from incoming data to glyphs is performed via multi-stage mappings. Chains of mappings are called rules. All mapping operators consume tuples and produce tuples as a result. The end of a rule is typically a fragment of a glyph tuple, and the collection of rules in a layer defines how the glyphs of that layer appear.

3 Language

The Stencil language is closely based on the model described above. Many elements are directly translated, so please familiarize yourself with the model before proceeding to the particulars of the grammar.

As a general rule, Stencil is case sensitive. Keywords tend not to be, but don't rely on that...keep them in upper case! Python block names are not case sensitive, but may be in the future. Some tuples are not case sensitive, but others are. As a general rule, keep case the consistent and upper-case all stencil-defined elements. As things move forward with the grammar, this will likely become the standard.

3.1 Incoming Data

external stream <name> (<field>+)
All incoming data streams must be described at the top of the stencil. The description is initiated with the keywords external stream (someday there will be other *external* entities, thus the pair of keywords). Field names follow, providing a tuple prototype for the rules associated with the stencil. The stream names used in this section are used in the 'from' blocks of the layers.

3.2 Layer Definition

layer <name>[<glyph>]
Names a layer and opens a context for stream definitions. Layer names must be unique. When a layer is defined, a few mapping operators are implicitly created: Make, Find and MakeOrFind. These operators work with the collection of glyphs in the layer. All work as functions mapping an ID to a glyph tuple. If any of these functions is invoked in a rule on the current layer, it will set the glyph that is being modified by all rules until another such function is invoked. One of these functions must be invoked before any values can be set on a tuple in a layer. Only the 'Find' function can be safely invoked by rules in another layer. By default, when setting ID, MakeOrFind will be invoked (unless find or Make are explicitly invoked). The glyph type may be omitted, it is then assumed to be 'SHAPE'.

3.3 Filters

filter(<field> <op> <value>) : <Legend>(<value>+) -><Legend> (<value>+) ->... -><value> To determine if a set of rules should be run, a set of filters may be defined. A filter is initiated by the *filter* keyword and a test definition. Tests are created by a field, operator and value. The field is the product of the mapping. Valid fields can be found in the far right-hand side of the mapping. Valid operations can be found in the description of legends.

3.4 Rules

from <stream>
A 'from' or 'consumes' block is initiated by a statement of the stream to take values from. All rules in the block can use the fields from the stream. A from block begins with the word 'from' and continues until the next from block, layer or legend is started.

<**field**> **:** <**field**>
<**field**> **:** <**Legend**>(<**value**>**+) ->**<**Legend**> (<**value**>**+) ->... ->**<**value**>
(<**field**>**+): <**Legend**>(<**value**>**+) ->**<**Legend**>(<**value**>**+) ->... ->(**< $value$**¿+)**
<**target**> (< $field$**¿+): <**Legend**>(<**value**>**+) ->**<**Legend**>(<**value**>**+) ->... ->(**< $value$**¿+)**

Rules have two parts, a target and a mapping. The target is on the left-hand side of the colon and the mapping is on the right. By default, all rules in a layer target *glyph* (other in a layer are *view* and *canvas*). The first call in the mapping uses any of the fields in the incoming stream. After that, each subsequent call uses only values created by the prior legend. The last part of a mapping (which may be the only part of the mapping) is just a list of fields and literals; it is matched pairwise to the field list in the target portion of the rule.

3.5 Specializers

<**legend**>**[**<**values**>**](**<**value**>**+)** May legends take compile-time arguments. These are given in square braces after the legend name but before the runtime arguments. The valid arguments of a specializer are determined by the legend itself. Common specializers include a range (of the form <start> .. <end> where *n* indicates the current position in the stream, positive values indicate absolute positions in the stream and negative values indicate offsets from the current position) and argument list.

3.6 Colors (Sigils)

@<**name**>**(**<**value**>**+) @color(**<**value**>**+)** Standard form tuples are defined by *sigils*. Color is the only sigil currently implemented. A sigil can be used anywhere a value is expected. It defines a tuple that can be interpreted according to some broader context. A color is a 4-tuple (r,g,b,a), though you can create it as either r,g,b,a or r,g,b or by name or by name with alpha. Planned future sigils include date, time and in-fix math equations that yield singleton values.

3.7 Glyphs

There are a variety of glyph types. All of them have ID,X,Y and IMPLANTATION properties. The value of the IMPLANTATION can only be set once and determines what other properties the glyph has. IMPLANTATION is borrowed from Bertin and refers to the type of glyph being implanted into the canvas. Implantations and their special properties are defined below. Since many implantation share the same properties, two major property groups are defined below the implantation listing.

SHAPE : (Default node type)

- Stroke Group
- Fill Group
- SHAPE (RECTANGLE, CROSS,ELLIPSE, DIAMOND, HEXAGON, STAR, TRIANGLE_UP, TRIANGLE_DOWN, TRIANGLE_LEFT, TRIANGLE_RIGHT, NONE)
- REGISTRATION (TOP_LEFT, CENTER, TOP_CENTER)
- FILL_COLOR (background color, also used even when pattern is used)
- PATTERN (HATCH, EMPTY)
- PATTERN_WEIGHT (how heavy should non-background features be)
- PATTERN_SCALE (how scaled up)
- PATTERN_COLOR (what color should non-background features be)

LINE :

- Stroke Group
- X2, Y2

POLY_LINE :

- Stroke Group
- Size Group (query only, defines bounding box with index-free X and Y)
- X<n> – Set the n'th X value
- Y<n> – Set they n'th Y value
- You can only extend the line by one segment at a time.
- Lines are draw between adjacent X/Y coordinates

POLY :

- Stroke Group
- FILL_COLOR
- Has the same properties as POLY_LINE, but the last coordinate is always connects to the first.

PIE :

- Stroke Group (affects only the outline)
- PERCENT: How much should be filled with a slice
- FIELD: Value for the non-slice
- SLICE: Value for the slice
- Note: If field or slice is set, then percent becomes read-only and will report as SLICE/(FIELD+SLICE)
- SLICE_COLOR
- FIELD_COLOR
- PIE_STYLE (CLOCK or CENTER)

TEXT :

- Stroke Group (defines the outline of the bounding box)
- Size Group (define the text bounding box)
- ROTATION
- REGISTRATION (TOP_LEFT, CENTER, TOP_CENTER)
- TEXT – Actual text to display
- JUSTIFY (LEFT, RIGHT, CENTER)
- FONT
- FONT_SIZE
- FONT_STYLE
- FONT_COLOR

IMAGE :

- Size
- FILE_NAME – Image file to load

Size group :

- WIDTH
- HEIGHT

Stroke Group : Line-based issues

- STROKE_WEIGHT
- STROKE_COLOR
- CAP_STYLE (BUTT, ROUND, SQUARE)
- JOIN_STYLE (MITER, BEVEL, ROUND)
- MITER_LIMIT

3.8 Legends

legend <name> (<field>+) ->(<field>+)
<filter> => <rule>
Defines a legend of a give name, consuming a tuple described by the prototype and creating a new one described by the second prototype. Legend rules are similar to layer rules, put they are prefaced by a filter and have a target type of *return* (and only *return* may be used.).

Filters are boolean expressions, and must be followed by the *gate* operator '=>'. The first filter matched determines which rule is run. Fitlers are formed as (¡field¿ ¡op¿ ¡value¿). The left-hand side must come from the tuple prototype fields, the right-hand may be a field or a non-color literal (sorry, color comparison doesn't work yet). Valid operators are: <¿> <==>, <!=>, <= > (regular expression) or <! > (negated regular expression). A default case for legends may be provided by using the keyword *all* in place of the normal filter.

3.9 Python

PYTHON <name> <facet>+
Special legends may be expressed in Python. These legends step outside of the stencil model, but may be used as if they were inside since they consume and produce tuples. The basic syntax is to name a function group and a memory space. The environment is shared between facets defined in a single python group.

facet <name> (<field>+) ->(<field>+) <body>
The block syntax defines a block of code and the input/output tuple prototypes as well as the body. The input tuple will be loaded as individual variables into the the python environment. The output tuple is constructed by copying values with the given names out, after the body has been executed. The name used to call a legend is the group.block. The default block in a standard mapping (one that ends in GLYPH) will be 'main' unless otherwise specified. It is 'query' if in a FILTER mapping. Block names are not case sensitive.

A special block *Init* cannot have tuple prototypes. *Init* is called once when the stencil program begins to execute. Other special facets are *Map* (the default call if only the name given in Python is used in a regular rule) and *Query* (the default call in legend and filter context).

4 Stencil Explore (Simple Host Application)

The Stencil system is based on a component generation model (much like compiler generators). As such, a stencil component always relies on a host application to provide it with data streams and a canvas. The Stencil jar contains a simple host application for interactively building and testing stencils. This application is known as Stencil Explore.

Stencil explore is included in the JAR download (in addition to the Stencil interpreter). To launch Explore, simply invoke the jar (double click or type java -jar Stencil.jar...assuming you renamed the jar to Stencil). **However, for this to work, you will need to have the support library downloaded as well.** The contents of the support library (the jar's that Stencil depends on) need to be located in the same directory that the jar lives in. Make sure you have the right support library (it changes occasionally).

Stencil Explore is not intended as a comprehensive graphics production environment; rather, it should be regarded as a sandbox to get started in. As such, it does not expose the full range of of Stencil capabilities. Noted limitations are the ability to only receive data streams from regular-expression split-able files and the mouse (Stencil has no such limitation, it only expects streams of tuples and does not care about their source). Further, it can only present one stream of tuples to the Stencil component at a time (even though a Stencil generated component is capable of processing concurrent streams).

The two major portions of the explore application are the stencil definition (left-pane) and the source-list (right-pane). The stencil definition contains the actual stencil. The source list is a way to connect defined streams to files (or set parameters on the mouse stream, when used).

5 Examples

Annotated examples can be found in the 'examples' directory. The 'simpleLines', 'scatterPlot' and 'stocks' examples demonstrate the basic capabilities. Python integration is used heavily in the 'SeeTest' example. Automatic guide generation (not discussed here, but possible in limited circumstances) is demonstrated in the 'autoGuide' section.