Stencil: Brief introduction to model, language and Explore

Joseph A. Cottam

April 30, 2010

1 Quick Start

If you are itching to just jump in, run ”java -jar Stencil.jar” from the command line in the directory this document was found in. If you get lost, read this document. If you’re still lost, feel free to contact me at jcottam@indiana.edu.

2 Introduction

Welcome to the super-alpha documentation for Stencil. Things are always changing with Stencil, 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.

3 Model

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

3.1 System

Stencil is geared towards creating visualizations of dynamic data (e.g. data that changes while its being examined) based on declarative descriptions. Stencil creates a swing panel component that accepts data streams, effectively separating the hosting program from any details about operation scheduling, internal data representation, graphics libraries, etc. The resulting component asks for two things from a host program: some data and a space to draw on (i.e. to be included in a display).

3.2 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.

Data being feed into a stencil component from the outside word must have a special type of tuple: SourcedTuple. Sourced tuples a pair (Source, Values). Source indicates which stream the data belongs to. Values is a tuple that matches the description given in the stencil program for the indicated stream.

3.3 Visuals

There are two primary visual concepts: Layers and Glyphs. Glyphs are visual primitives, like lines or circles. Stencil is currently 2D + layering (sometimes called 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.

Glyphs are grouped into layers. Layers group glyphs that are derived in a similar fashion from the incoming data. Each layer may only include on type of glyph. 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.

3.4 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.

Many operators are included in stencil. Unfortunately, no good catalog of operators is included. The closest thing is the .yml files in the source code. The .yml files also include the tuple prototypes and default specializers. Generally speaking, this will indicate what parameters can be set on an operator and what the results will look like. There are exceptions to every rule though as some operators produce different tuples based on the specializer and some (through my own negligence) do not provide default values through their default specializer.

4 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.

Stencil is now fully case sensitive (this was not the case in earlier versions). By convention, keywords are lower case, operators are camel case (with initial capital), facets are camel case (with initial lower-case) and key names are lower case. Tuple fields follow two conventions, depending on where they come from; glyph fields are upper case while all other tuples tend to start with a lower case letter and are camel case there-after. Exceptions exist, so be careful (sorry...working on finding a convention that works).

Stencil programs are conventionally laid in a general-to-specific/early-to-late style, as follows (deviations from this listing may be made, as noted below)

Preamble: Imports, Stream declarations, Order declarations, Canvas descriptor. These must appear in the order specified here, but any element except the stream declarations may be omited.

Streams/Layers: Description of stream transformations and layers to store results in. These may be intermixed (and mixed with operators), but streams tend to be given first.

Operators: ”Pure” stencil operators are defined entirely within stencil

Pythons: Python operator descriptions. These tend to be long, so they are put last.

4.1 Imports

Imports are currently not required since all modules are imported by default.

4.2 Data Description

stream <name> (<field>+)
All stencil programs must consume external data. Therefore, stencil programs start with a declaration of external data sources. The data descriptor is is started the **stream** keyword and includes a name and a tuple prototype. Incoming data must conform to the give prototype (e.g. it must have the same number of fields).

4.3 Order Declaration

order $< group >$**+** Order declarations indicate prioritization between external streams. Groups are of the form $(< name > (|| < name >)+)$. The || indicates that the listed streams have the same priority and should be loaded round-robin style. If a group contains are all finite, it may be given priority with the form $< group >>< group >$. The group on the left of the $>$ will be loaded in its entirety before any stream on the right.

4.4 Canvas

Canvas descriptor is used to control the background color and setup guides. The guide system is complex, and not described in this document. Please refer to the examples given in the AutoGuide directory and published papers.

4.5 Stream Definition

stream <name> (<field>+)
Creates a new internal stream and opens a context for data consumption. Stream names must be unique relative to other streams and layers. The field list is a prototype of the tuple produced by this stream definition.

4.6 Layer Definition

layer <name>
Names a layer and opens a context for data consumption. When a layer is defined, a new operator is implicitly created with $find$ and $remove$ facets. Layers are comprised of consumption contexts filled with rules. To change the type of glyph on a canvas, a type indicator can optionally be included in square-brackets after the layer name. Layers may include a list of default values before any consumption context, see included examples.

4.7 Consumption Context

from <name> <rule>+
The from clause must in a layer or stream. The name indicates what stream to get values from (either an external or stencil defined stream). Each tuple in the stream will trigger each consumption context. If two (or more) consumption contexts in the same layer/stream name the same stream, all will be executed, in top-to-bottom order.

4.8 Rules

(<field>+) : <Operator>(<value>+) -> ... -> (<value>+)
(<field>+) : $*$ **<Operator>(<value>+) -> ... -> (<value>+)**
Rules are defined as sets of values (either names or literals) passed to a mapping operator. The call rule syntax is straightforward, but the assignment operation is a little different. The syntax is as follows:

1. The left-hand is the target fields in the glyph tuple. This is implicitly the most appropriate for the given context, but others may be specified (discussed later).

2. The two binding operators are static (the colon) or dynamic (the colon-star). These determine how often the values are computed. Simply, the static binding is only done once: when the source tuple is being processed off the stream. The dynamic binding is updated regularly, even when no new values are found on the stream. To accomplish this, the relevant portions of the source are associated with the resulting glyph and periodically used to re-evaluated the rule. Only the tuple used to most recently update the glyph is remembered. Dynamic binding can only be used in a layer.

3. Operator calls are specified as name and the arguments to them. These are positionally significant elements. Numbers, literal strings and tuple references may be used as arguments.

4. Pass operator (arrows) places a tuple on the results stack. The map $>>$ and fold $>-$ operators also exist but are not discussed further here.

5. Pack statement end a call chain. A pack is a list of values with no prefixing operator. Values in the pack are matched order-wise to the fields on the left of the binding operator. If there is no pack given, a default pack is produced of the required length taking the values in order from the last result (if there are insufficient results an error is produced, but like many Stencil errors its fairly difficult to decode).

Operators can optionally include a a 'specializer'. This is a set dictionary of compile-time parameters to set up the operator. Specializers look like $[< key >:< value >]$ (with more pairs given as a coma-separated list). Any value may appear in the value position, but keep in mind that no operation may be performed (these are compile-time arguments, so no interpretation is conducted).

By default, rules will 'target' the logical result of the context. In a layer, that is the glyph. In a stream, that is the result of the stream. In an operator, that is the operator result. $view$ and $canvas$ are also valid targets in stream or layer contexts to set properties of the view or canvas respectively.

Some operators are referred to with like @ $< name >$ .... This is a custom-syntax argument operator. It allows arguments to be specified that do not follow the normal comma-separated syntax. Color is specification is the most prominent example. Tuple values are referred to inside the arguments by placing them in curly-braces.

Tuple references rules are simple, but allow complex reference to values. To understand reference rules, understand that tuples are built on to a stack of results. The stack is as follows: canvas, view, $< stream >$, prefilter, local, $< results >$*. The $< stream >$ frame is referred to with the stream name included in the $from$ clause or the literal word $stream$. Other frames are referred to with the given names. Results frames can only be reference if named (see included examples, look for $-[xxxx] >$) UNLESS it is the most recent result frame. Tuple fields can be referred to by name or by index. If by index, the index is enclosed in square braces, per $[n]$. Fields in a named frame are referred to like $canvas[ref]$. The ref may be a number or a name.

Three special reference values exist. First, underscore is shorthand for [0], the default value of a tuple. Second, star refers to all values of the given tuple (or the default tuple if unqualified). These will be packed up position-wise. It may only be used with tuple with a finite prototype. Finally 'LAST' will pass the tuple itself into an operator.

4.9 Glyphs

There are a variety of glyph types. All of them have ID,X,Y, REGISTRATION and IMPLANTATION properties. The value of the IMPLANTATION indicates what properties the glyph has but cannot be set (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.

Every tuple has a REGISTRATION, X and Y. Valid registrations are TOP_LEFT, TOP, TOP_RIGHT, LEFT, CENTR, RIGHT, BOTTOM_LEFT, BOTTOM and BOTTOM_RIGHT. The X and Y report the location of the registration point. Changing the registration will not change what X and Y report but will change where the glyph is rendered (NOTE: This is a change from earlier behavior). WIDTH and HEIGHT can be used to get the width and height that the object will be drawn, sometimes it can also be used to set. It will always report measuring from the top-left corner of the glyph, regardless of the registration.

Most glyphs also support ROTATION. When rotation is present, it usually around the registration but sometimes around top-left corner. Sorry.

Indexed properties (denoted by a **.n** below) use the following conventions (the example is the X property, replace with the appropriate property name):

X.1	The first defined element in the group.
X.n	The last defined element in a group.
X.new	Add a new element to the end
X.0	Add a new element to the beginning (may not always be defined).
X.1.5	Add a new element between 1 and 2 (may not always be defined). In general, specify a fractional value indicates an insertion between the floor and the ceiling of that value.

SHAPE : (Default implantation)

- Stroke Group
- Fill Group

- SHAPE (RECTANGLE, CROSS,ELLIPSE, DIAMOND, HEXAGON, STAR, TRIANGLE_UP, TRIANGLE_DOWN, TRIANGLE_LEFT, TRIANGLE_RIGHT, NONE)
- SIZE: Number describing how large the glyph should be. Unit is in pixels, but float values are understood.

LINE :

- Stroke Group
- X.1, Y.1, X.2, Y.2

POLY_LINE :

- Stroke Group
- Xn – Set the n'th X value
- Yn – 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 Group
- Has the same properties as POLY_LINE, but the last coordinate is always connects to the first.

PIE :

- Stroke Group
- 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 should be treated as read-only. It will report as SLICE/(FIELD+SLICE)
- SLICE_COLOR
- FIELD_COLOR
- ANGLE : The start angle of the slice. Field and Slice determine the end angle

TEXT :

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

IMAGE :

- Size
- FILE_NAME – Image file to load

Fill Group : Fill issues

- FILL_COLOR Fill color or the foreground color of a fill
- PATTERN Specify the pattern, HATCH is the only implemented item right now
- PATTERN_SCALE Overall size of a pattern iteration. The tile size is derived from this value.

- PATTERN_WEIGHT Characteristic size of pattern elements
- PATTERN_BACK Background color of a pattern. Default is CLEAR.

Stroke Group : Line-based issues

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

4.10 Filtering

filter((<field> <comparison> <value>)+)

Filtering occurs in consumption contexts. Generally, filtering occurs before any layer rules are executed. However, since some conditions are complex, a $prefilter$ target may be specified for bindings. All $prefilter$ bindings will be executed before filtering. It is strongly suggested that side-effects be avoided in $prefilter$ targets, but they are permitted. Multiple filter blocks may be included in one context. Multiple tests may be included in a filter block. All tests in a filter block are combined with AND. Multiple filter blocks under a target are combined with OR.

Valid comparison operators are >, >=, <, <=, ==, !=, = (regular expression) or ! (negated regular expression). Any Java regular expression may be used. True and False are represented with "true" and "false". Complex objects are compared using the underlying .equals method when == is used (pointer equality is not possible in Stencil....I'm not sure it even makes sense).

4.11 Operators

operator <name> (<field>+) -> (<field>+)
<filter> => <rule>

Defines an operator of a give name, consuming a tuple described by the first prototype and producing a new tuple with the 2nd prototype. Operators typically include a list of filter/rule pairs (joined with the 'gate' operator =>). This is useful if an operator's internal memory is needed in multiple locations. Operator filters are formed like layer filters. If the filter passes, then the rule is executed. Filters are evaluated in top-to-bottom fashion. $prefilter$ rules are also permitted in operators. All $prefilter$ rules will be executed before any filters are checked. Operator rules are formed like layer mapping rules.

An alternative operator form has no rules, instead using $base < operator >$ after the 2nd tuple prototype. This indicates that the operator is an instance of the given operator. This is useful for sharing memory between operator references.

4.12 Python

python <name> <facet>+

Operators may be expressed in Python. These operators step outside of the stencil model, but may be used as if they were inside since they consume and produce tuples. All facets given will share the same memory space memory space.

facet <name> (<field>+) -> (<field>+) <body>

The facet 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. Name resolution is case sensitive.

A special block is the Init block. It will be execute once before any other blocks are. Init is specified without any prototypes.

5 Examples

Here are some examples. Many more can be found in the Examples directory. Each example in the examples directly includes a reference image with the stencils and data used to generate it.

5.1 Scatter Plots

One of the more basic diagrams (and thus a simple stencil) is scatter plot. This one assumes V1 and V2 are numeric values.

```
stream NodePositions (name, v1, v2)

layer ScatterPlot
from NodePositions
   ID:  name
   (X,Y) : (v1, v2)  /*Use V1 for X and V2 for Y*/
   SIZE: 3
   SHAPE: "RECTANGLE"
   FILL_COLOR:  @Color(GRAY)
```

A bit more complex is a scatter plot using operators to derive the layout. Assume that v1 and v2 are now names that need to be converted into numerical values.

```
stream Points (name, v1, v2, category)

layer ScatterPlot
from Points
   ID : name
   X:  XLayout(v1)
   Y: YLayout(v2)
   FILL_COLOR:  Colorize(category)
   STROKE_COLOR : @Color{CLEAR}

operator XLayout (v) -> (v)
   (ALL) => v : Index(v)

operator YLayout (v) -> (v)
   (ALL) => v : Index(v)

operator Colorize (cat) -> (C)
   (cat =˜ 'west')   => C : @Color{BLUE}
   (cat =˜ 'east')   => C : @Color{Gray40}
   (cat =˜ 'north')  => C : @Color{AQUA}
   (cat =˜ 'south')  => C : @Color{192, 202, 36, 120}
   (cat =˜ '.*')     => C : @Color{RED}      /*Just in case we missed something...*/
```

5.2 Node-Link

Basic node-link diagram, reading the nodes out of the adjacency list that also defines the links. The layout is being computed by the python function “Squarify”.

```
stream BFS(id)
stream VertexList(id1, id2)

order BFS > VertexList
```

```
layer Nodes
from BFS
   ID : ID
   REGISTRATION: "CENTER"
   FILL_COLOR: @Color{BLUE}
   (SHAPE, SIZE) : ("ELLIPSE",  10)
   (X,Y) : Ordering(ID) -> Squarify.Main(_) -> (X,Y)

layer Labels[TEXT]      /*Store text nodes here, not shape nodes*/
from BFS
   ID: id
   (X,Y): Nodes.find(ID) -> (X,Y)
   (TEXT, FONT_SIZE): (ID, 10)

layer Edges[LINE]
from VertexList
   ID: Concatenate(ID1,ID2)
   (X.1, Y.1): Nodes.find(ID1) -> (X,Y)
   (X.2, Y.2): Nodes.find(ID2) ->  (X,Y)
   STROKE_COLOR: @Color{Gray30,0.8}


operator Ordering base Index

python Squarify
facet Init {
   from math import sqrt
   from math import ceil
}
facet main (Seq) -> (X,Y) {
   seq = int(Seq)+1
   base = int(ceil(sqrt(seq)))
   X = (seq/base) * 10 *base
   Y = (seq\%base) * 10 *base
}
```