Stencil: Brief introduction to model, language and the Explore application

March 31, 2012

1 Quick Start

This document reflects the version for March 2012. 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 (jcottam@indiana.edu).

2 Introduction

Welcome to the 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 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 Stencil programs.

3 Model

Before we diving into the language itself, lets take a look at 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 fed 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 most have WIDTH, HIEGHT and IMPLANT properties.

Glyphs are grouped into layers. Layers group glyphs that are derived in a similar fashion from the incoming data. Each layer may include only one 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 thought 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 (sorry). The closest thing is to look at the java source code in the stencil.modules package. Anything annotated with @Operator is an operator. (Help: An APT script to translate those annotations to webpage would be greatly appreciated.) The annotation contains a 'name' field, which is the name that can be used to call the operator from Stencil. It also probably contains a 'prototype' field, which indicates what fields its return tuple has. Any operator without a prototype in its annotation creates a tuple with a prototype based on either its instantiation arguments or that can't be effectively prototyped.

Operators are grouped into modules. Most modules are imported by default, but some are not. A list of available modules and the default imports is found in the Stencil.properties file.

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 case sensitive. Keywords are lower case, except NULL. By convention, operators are camel case (with initial capital) and facets are camel case (with initial lower-case). Field and specializer keys (more on that later) are camel case (with initial lower-case). Tuple fields follow two conventions, depending on where they come from; glyph fields are upper case while all other tuples are camel case (with initial lower-case).

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, Constants, Stream declarations, Order declarations. These must appear in the order specified here, but any element except the stream declarations may be omited.

Consumers: View/Canvas/Streams/Layers all consume streams (in 'from' blocks). Stream definitions (not declarations) and layers both produce tuples as well. These may be intermixed (and mixed with operators), but stream definitions tend to be given first.

Operators: "Pure" Stencil operators are defined entirely within Stencil

4.1 Imports

import <name>
import <name> : <prefix>
Imports are performed with the keyword **import**. This is followed by the name of the module to import. Modules can only supply new operators (data, guide and layer types are still in progress). By default, all operators in a module are placed in the root namespace. However, if colon-namespace for is used, the namespace prefix must be used to refer to any operator from the module.

4.2 Stream Declaration

stream <**name**> **(**<**field**>**+)** from <source>
All Stencil programs must consume external data. Therefore, Stencil programs usually include a list of external data sources. The data descriptor is is started the **stream** keyword and includes a name and a list of field names (called its prototype). Incoming data must conform to the give prototype (e.g. it must have the same number of fields). The **from**-segment describes where the data will be loaded from (typically a file).

4.3 Order Declaration

order $< group >$**+** Order declarations indicate prioritization between 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 streams that 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 <**name**> <**specializer**> <**from-block**>**+** Canvas descriptor. Used to setup background color, possibly in response to incoming data. The syntax for specializer and from-blocks is shared with many other entities, and thus deferred until after layer definitions are described.

4.5 View

view <**from-blocks**> View descriptor. Allows navigation in response to incoming data (zoom/pan/etc.) based on rules in the from-block.

4.6 Stream Definition

stream <**name**> **(**<**field**>**+** <**from-block**>**+)**
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. Stream definitions can include rules of the same form as layer definitions.

4.7 Layer Definition

layer <**name**> <**specializer**> <**guides**> <**defaults**> <**from-block**>**+**
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 guides, and from-blocks (consumption contexts) filled with rules. The type of the glyph in the layer is set in the specializer (the key is ”type”, but it is also the default key so it may be omitted, the default value is ”SHAPE”). Layers may include a list of default values before any consumption context, prefixed with the word ‘defaults’ but otherwise the same as a consumes block.

4.8 Guide Definitions

guide (<**type**> <**specializer**> **from** <**att**>**)* guide (**<**type**> <**specializer**> <**sample**> **from** <**att**>**)*** Guides all come under a single ‘guide’ heading that is part of the layer definition. The guide type indicates what kind of guide to create (axis, legend, trend line, etc). One or more (comma-separated) ‘from’ statements may appear to indicate which attribute of the layer the guide is to address. The sample type may be omitted, but Flex, Linear, Categorical and Log are accepted. The default is Flex which acts as a mix of Linear and Categorical based on the input data.

4.9 Consumption Context

from <**name**> <**rule**>**+**
The consumption context links rules to an input stream. It is initiaited with the keyword ’from’ (and is thus often called a ‘from-block’). After the ‘from’, the name indicates which 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.

4.10 Rules

(<field>+) : <Operator>(<value>+) -> ... -> (<value>+)
(<field>+) : $*$ **<Operator>(<value>+) -> ... -> (<value>+)**
Rules are defined as binding pairs. The left side indicates which layer attribute will receive the results of the right side (this is called a ‘binding’).

The syntax is as follows:

1. The left-hand is the target fields in the glyph tuple. This must be a defined field for the layer’s type.

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-evaluate 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, tuple references and functions (prefixed with an @) may be used as arguments.

4. Pass operator (arrows) places a tuple on the results stack.

5. Pack statements 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 it’s fairly difficult to decode).

Operators can optionally include 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). Literal values and constants can appear in as values. Keys may be omitted. In this case, the keys are then filled in from the default specializer defined by the context.

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. $prefilter$ and $local$ are also valid targets that place computations before others.

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 specification is the most prominent example. Tuple values are referred to inside the arguments by placing them in curly-braces.

Operators produce results and those results must be named. Binding a name for a result tuple is done by placing the name in square braces before the call. If no name is supplied, then the operator name is used instead. This is analogous to a ‘let’ binding. For example: ”Add(1,2,3)” results in a tuple called ‘Add’ but ”[Sum] Add(1,2,3)” results in a tuple called ”Sum”.

In general, tuple references are made using the tuple name, followed by the field name. However, there are some exceptions for convenience. Just writing the tuple name will give the tuple’s first field value. Therefore ”Sum” and ”Sum.0” are the same. To get the tuple as the value itself, use the suffix ”.ALL”. The underscore is ”the most recent value”, and will get the first value out of the most recent tuple produced.

4.11 Glyphs

All glyphs have ID, X, Y, Z, VISIBLE and most have WIDTH, HIEGHT, REGISTRATION, ROTATION and IMPLANT properties.

Registration and implant bear some describing. Registration indicates where on the resulting pixels the X/Y indicate. This is important for all non-points since X and Y pick a point, but the glyph will span many points. Valid registrations are TOP_LEFT, TOP, TOP_RIGHT, LEFT, CENTER, RIGHT, BOTTOM_LEFT, BOTTOM and

BOTTOM_RIGHT. Changing the registration will not change what X and Y report but will change where the glyph is rendered. 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 them. It will always report measuring from the bottom-left corner of the glyph, regardless of the registration. Glyphs that also support ROTATION will rotate around the registration.

IMPLANT refers to the "implantation" of the rendering, or how the rendering interacts with the zoom and pan. POINT implantation take size information in zoom-independent fashion. A circle of size 1 and implanted as ”POINT” will always take the same number of screen pixels, regardless of zoom level. In contrast is ”AREA” implantation. A circle of size 1 in ”AREA” implantation takes up one pixel at zoom 1 but 100 pixels at zoom 100. Valid values are ”AREA”, ”POINT”, ”X”, ”Y”, ”LARGEST”, ”SMALLEST” and ”LINE”. "LINE" implantation behaves like ”POINT” implantation, but it looks better for line-based glyphs.

SHAPE : (Default layer type)

- FILL_COLOR – Paint for the fill, does not have to be a solid color (any paint will do)
- PEN_COLOR – Paint for the outline
- PEN – Stroke information for the outline
- 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.

SLICE : (Fraction of a circle)

- SIZE – Radius
- START – Start angle (in degrees)
- END – End agle (in degrees)
- FILL_COLOR
- PEN_COLOR
- PEN
- Note: There is no rotation for this glyph type. Must use the start and end angle appropriately.

LINE :

- PEN
- PEN_COLOR
- X1, Y1, X2, Y2
- CAP1, CAP2 – Line caps

POLY_POINT : Poly point covers closed and opened multi-segment lines. It is a collection of points, with properties settable on each point.

- PEN
- PEN_COLOR
- X
- Y
- GROUP – Which points should this one be associated with? IDs must be unique in the layer, this lets multiple lines be kept in one layer.
- ORDER – Where in the sequence of points does this one fall? This will be sorted at render time, so arbitrary values are allowed.
- CONNECT – Should the last point in the group connect back to the first?
- SEGMENT – If this is set to false (which is the default value) for the first item in a group, then the first items visual properties are used for all segments. Otherwise the 'head' point's values are used for each segment.

PIE :

- PEN

- PEN_COLOR
- 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 :

- COLOR – Fill color of the text
- TEXT – Actual text to display
- JUSTIFY (LEFT, RIGHT, CENTER)
- FONT
- BOUNDS & LAYOUT – Readable fields, used internally for efficiency

IMAGE :

- FILE – Image file to load (png files only)

4.12 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).

4.13 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. This is useful if an operator's internal memory is needed in multiple locations. Operators typically include a list of filter/rule pairs (joined with the 'gate' operator =>). In operator context, the 'default' keyword can be supplied as a filter that matches anything. 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.

5 Examples

There are many examples in the Examples directory. Each example includes a reference image with the Stencil code and data used to generate it.

5.1 Detailed example

One of the more basic diagrams (and thus a simple Stencil) is scatter plot. The one given below has X,Y, color and trendline guides It uses a dynamic binding for X and Y so the input ranges don't need to be known but all plots will be 100px by 100px (unzoomed).

```
//Import some color functions based on Cynthia Brewer’s work
import BrewerPalettes

//Expect a tuple with six fields
stream flowers(sepalL, petalW, sepalW,  petalL, species, obs)

layer FlowerPlot
guide
  trend[sample: "SIMPLE"] from ID                     //trend guide type must be ‘from ID’
  legend[X: 15, Y: 90] Categorical from FILL\_COLOR
  axis[guideLabel: "Petal Length"] Linear from X
  axis[guideLabel: "Petal Width"] Linear from Y

from flowers
   ID: obs
   X:* Scale[min: 0, max:100](petalL)
   Y:* Scale[min: 0, max:100](petalW)
   FILL\_COLOR: BrewerColors(species) -> SetAlpha(50,_)
   REGISTRATION: "CENTER"
```

6 Known Errors and Issues

The implantation is not always correct, especially in the guides.

An experimental multi-thread, queued-loader (to hide I/O latency) sometimes cannot detect the end of file properly. This can result in an infinite loop or a dropped tuple.

Error messages from Stencil are horrible...sorry.