SUMMON 1.6 Manual

Matt Rasmussen
April 6, 2007

Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology

rasmus@mit.edu

Contents

1 Introduction

1.1 What is SUMMON

SUMMON is a python extension module that provides rapid prototyping of 2D visualizations. By heavily relying on the python scripting language, SUMMON allows the user to rapidly prototype a custom visualization for their data, without the overhead of a designing a graphical user interface or recompiling native code. By simplifying the task of designing a visualization, users can spend more time on understanding their data.

SUMMON was designed with several philosophies. First, recompilation should be avoided in order to speed up the development process. Second, design of graphical user interfaces should also be avoided. Designing a good interface takes planning and time to layout buttons, scrollbars, and dialog boxes. Yet a poor interface is very painful to work with. Even when one has a good interface, rarely can it be automated for batch mode. Instead, SUMMON relies on the python terminal for most interaction. This allows the users direct access to the underlining code, which is more expressive, and can be automated through scripting.

Lastly, SUMMON is designed to be fast. Libraries already exist for accessing OpenGL in python. However, python is relatively slow for real-time interaction with large visualizations (trees with 100,000 leaves, matrices with a million non-zeros, etc.). Therefore, all real-time interaction is handled with compiled native C++ code. Python is only executed in the construction and occasional interaction with the visualization. This arrangement provides the best of both worlds.

1.2 Features

Listed below is a short summary of the features offered in this version of SUMMON.

- Python module extension
- Fast OpenGL graphics
- Drawing arbitrary points, lines, polygons
- Binding inputs (keyboard, mouse, hotspots) to any python function
- SVG output (also PNG with ImageMagick)
- cross-platform (windows, linux)

2 Installing SUMMON

The latest version of SUMMON can be obtained from http://people.csail.mit.edu/rasmus/summon/. Download the *.tar.gz archive and unzip it with the command:

```
tar zxvf summon-1.6.tar.gz
```

Before running or compiling SUMMON, the following libraries are required:

- python 2.4 (or greater)
- GL
- GLUT
- SDL (for threading)

2.1 Compiling SUMMON

SUMMON can be installed using the standard distutils (http://docs.python.org/inst/inst.html). For example, in the `summon-1.6` directory run:

```
python setup.py install
```

To install SUMMON in another location other than in `/usr` use:

```
python setup.py install --home=<another directory prefix>
```

2.2 Configuring SUMMON

SUMMON expects to find a configuration file called `summon_config.py` somewhere in the python path. Distutils installs a default module installed in your python path. To customize SUMMON with your own key bindings and behavior, you can write your own `summon_config.py` file. Just be sure it appears in your python path somewhere *before* SUMMON default configuration file. Alternatively, you can create a configuration file `.summon_config` in your home directory. The configuration file is nothing more than a python script that calls the SUMMON function `set_binding` in order to initialize the default keyboard and mouse bindings.

3 Using SUMMON

SUMMON can be used a stand-alone program and as a module in a larger python program. The stand-alone version is installed in `PREFIX/bin/summon` and is called from the command line as follows:

```
usage: summon [python script]
```

On execution, SUMMON opens an OpenGL window and evaluates any script that it is given in the python engine. After evaluation, the SUMMON prompt should appear which provides direct access to the python engine. Users should be familar with the python language in order to use SUMMON.

The SUMMON prompt acts exactly like the python prompt except for the OpenGL window and the appearence of a builtin module called `summon`. All of the commands needed to interact with the visualization are within the `summon` module.

To learn how to use SUMMON, example scripts have been provided in the `summon/examples/` directory. Examples of full fledged visualizations, SUMMATRIX and SUMTREE, are also given in the `summon/bin/` directory. Their example input files are given in `summon/examples/summatrix/` and `summon/examples/sumtree/`, respectively.

3.1 Example Script

For an introduction to the basic commands of SUMMON, let us walk through the code of the first example. To begin, change into the `summon/examples/` directory and open up `example1.py` in a text editor. Also use SUMMON to execute the example with following command.

```
$ summon example1.py
```

The visualization should immediately appear in your OpenGL window. The following controls are available:

left mouse button	scroll
right mouse button	zoom (down: zoom-out, up: zoom-in)
Ctrl + right mouse button	zoom x-axis
Shift + right mouse button	zoom y-axis
arrow keys	scroll
Shift + arrow keys	scroll faster
Z	zoom in
z	zoom out
h	home (make all graphics visible)
Ctrl + l	toggle anti-aliasing
p	output SVG of the current view
Ctrl + p	output PNG of the current view
Ctrl + x	show/hide crosshair
q	close window

In your text editor, the example `example1.py` should contain the following python code:

```
#!/usr/bin/env summon
# SUMMON examples
# example1.py - basic commands

# make summon commands available
from summon.core import *

# syntax of used summon functions
# add_group( <group> )   : adds a group of graphics to the screen
# group( <elements> )    : creates a group from several graphical elements
# lines( <primitives> )  : an element that draws one or more lines
# quads( <primitives> )  : an element that draws one or more quadrilaterals
# color( <red>, <green>, <blue>, [alpha] ) : a primitive that specifies a color


# clear the screen of all drawing
clear_groups()

# add a line from (0,0) to (30,40)
add_group(group(lines(0,0, 30,40)))

# add a quadrilateral
add_group(group(quads(50,0, 50,70, 60,70, 60,0)))

# add a multi-colored quad
add_group(group(quads(
    color(1,0,0), 100, 0,
    color(0,1,0), 100, 70,
    color(0,0,1), 140, 60,
    color(1,1,1), 140, 0)))


# add some text below everything else
add_group(group(
    text("Hello, world!",    # text to appear
         0, -10, 140, -100,    # bounding box of text
         "center", "top")))    # justification of text in bounding box

# center the "camera" so that all shapes are in view
home()
```

As you can see, the first line of the script imports all of the SUMMON functions from the `summon` module into the current environment. The first of such functions is the `clear_groups()` command. All graphics are added and removed from the screen in sets called *groups*. Groups provide a way to organize and reference graphical elements. The `clear_groups()` function removes all groups that may be on the screen.

The next line of python code in the example adds a single line to the screen. The line is created with the `lines` function, which takes a series of numbers specifying the end-point coordinates for the line. The first two numbers specify the x and y coordinates of one end-point (0,0) and the last two specify the other end-point (30,40). Next, the line is placed in a group using the `group` function which returns a group ready to be added to the screen. Lastly, the `add_group` function is called on the group. This function finally places the line on the screen. Although this may seem like a lot to type to draw a single line, in most uses several lines and other graphics are placed a group before adding them to the screen.

The next line in the example adds a quadrilateral to the screen with the `quads` command. The arguments to the `quads` function are similar to the `lines` function, except four vertices (8 numbers) are specified. Both functions can draw multiple lines and quadrilaterals (hence their plural names) by supplying more coordinates as arguments.

The third group illustrates the use of color. Color is stateful, as in OpenGL, and all vertices that appear after a color object in a group will be affected. The `color` function creates a color object. Color objects can appear within graphical elements such as `lines` and `quads` or directly inside a group. Since each vertex in this example quad has a different color, OpenGL will draw a quadrilateral that blends these colors.

Lastly, an example of text is shown. Once again the text is added to the screen using the `add_group` function. The arguments to the text function specify the text to be displayed, a bounding box specified by two opposite vertices, and then zero or more justifications ("center", "top", etc.) that will affect how the text aligns in its bounding box. There are currently three types of text: `text` (bitmap), `text_scale` (stroke), `text_clip` (stroked text that clips). The bitmap text will clip if it cannot fit within its bounding box. This is very useful in cases where the user zooms out very far and no more space is available for the text to fit. See the example `text.py` for a better illustration of the different text constructs.

The final function in the script is `home()`. `home()` causes the SUMMON window to scroll and zoom such that all graphics are visible. This is a very useful command for making sure what you have drawn is visible in the window. The command can also be execute by pressing the 'h' key. This key comes in handy when you "lose sight" of the visualization.

This is only a simple example. See the remaining scripts for examples of SUMMON's more powerful features.

3.2 Example Visualizations: SUMMATRIX and SUMTREE

In the `summon/bin/` directory are two programs, `summatrix` and `sumtree` that use summon to visualize large datasets. There programs are written in python and so can be easily extended. In my own work, I have extended the tree visualization program to integrate more closely with biological data (executing CLUSTALW and MUSCLE on subtrees, displaying GO terms, etc.). The purpose of writing visualization programs in this way, is to allow others to easily overlay and integrate their own data.

Also in both visualizations the underling data is accessable through global python variables. That means if you have a very specific question like, "How many genes in my subtree have a particular GO term?", you can quickly write a few lines of python to walk the tree and answer the question youself. It would be very difficult to anticipate all such questions during the development of a visualization. And yet when visualizing, it can become frustrating if you cannot fully interact with the data.

Example input files for both programs can be found under the `summon/examples` directory. Both programs will print their usage if run with no arguments. Here are some recommended examples:

```
$ sumtree -n olfactory-genes.tree
```

```
$ sumtree -n olfactory-genes.tree -t 10
```

```
$ summatrix -i human_mouse.imat
```

4 SUMMON Function Reference

All help information is also available from the SUMMON prompt. Use `help(command)` to get required arguments and a usage description.

4.1 SUMMON General Functions

`add_group(group)`
adds drawing groups to the current model

`assign_model(windowid, 'world'|'screen', modelid)`
assigns a model to a window

`call_proc(proc)`
executes a procedure that takes no arguments

`clear_all_bindings()`
clear all bindings for all input

`clear_binding(input)`
clear all bindings for an input

`clear_groups()`
removes all drawing groups from the current display

`close_window([id])`
closes a window

`del_model(modelid)`
deletes a model

`focus(x, y)`
focus the view on (x,y)

`get_bgcolor()`
gets background color

`get_group(groupid)`
creates a tuple object that represents a group

`get_model(windowid, ['world'|'screen'])`
gets the model id of a window

`get_models()`
gets a list of ids for all models

`get_mouse_pos('world'|'screen'|'window')`
gets the current mouse position in the requested coordinates

`get_root_id()`
gets the group id of the root group

`get_visible()`
gets visible bounding box

`get_window()`
gets the id of the current window

`get_window_name(id)`
get the name of a window

`get_window_size()`
gets current window's size

`get_windows()`
gets a list of ids for all open windows

`home()`
adjust view to show all graphics

`insert_group(groupid, group)`
inserts drawing groups under an existing group

`new_model()`
creates a new model and returns its id

`new_window()`
creates a new window and returns its id

`redraw_call(func)`

calls function 'func' on every redraw

`remove_group(groups)`
removes drawing groups from the current display

`replace_group(groupid, group)`
replaces a drawing group on the current display

`set_antialias(True|False)`
sets anti-aliasing status

`set_bgcolor(red, green, blue)`
sets background color

`set_binding(input, proc|command_name)`
bind an input to a command or procedure

`set_model(modelid)`
sets the current model

`set_visible(x1, y1, x2, y2)`
change display to contain region (x1,y1)-(x2,y2)

`set_window(id)`
sets the current window

`set_window_name(id name)`
sets the name of a window

`set_window_size(x, y)`
sets current window's size

`show_group(groupid, True|False)`
sets the visibilty of a group

`timer_call(delay, func)`
calls a function 'func' after a delay in seconds

`trans(x, y)`

translate the view by (x,y)

`version()`
prints the current version

`vertices(x, y, * more)`
creates a list of vertices

`zoom(factorX, factorY)`
zoom view by a factor

`zoomx(factor)`
zoom x-axis by a factor

`zoomy(factor)`
zoom y-axis by a factor

4.2 SUMMON Graphics

`points(* vertices|colors)`
plots vertices as points

`lines(* vertices|colors)`
plots vertices as lines

`line_strip(* vertices|colors)`
plots vertices as connected lines

`triangles(* vertices|colors)`
plots vertices as triangles

`triangle_strip(* vertices|colors)`
plots vertices as connected triangles

`triangle_fan(* vertices|colors)`
plots vertices as triangles in a fan

`quads(* vertices|colors)`

plots vertices as quads

`quad_strip(* vertices|colors)`
plots vertices as connected quads

`polygon(* vertices|colors)`
plots vertices as a convex polygon

4.3 SUMMON Primitives

`vertices(x, y, * more)`
creates a list of vertices

`color(red, green, blue, [alpha])`
creates a color from 3 or 4 values in [0,1]

4.4 SUMMON Text constructs

`text(string, x1, y1, x2, y2, ['left'|'center'|'right'], ['top'|'middle'|'bottom'])`

draws text justified within a bounding box

`text_scale(string, x1, y1, x2, y2, ['left'|'center'|'right'], ['top'|'middle'|'bottom`

draws stroked text within a bounding box

`text_clip(string, x1, y1, x2, y2, minheight, maxheight, ['left'|'center'|'right'], ['top'|'middle'|'bottom'])`
draws stroked text within a bounding box and height restrictions

4.5 SUMMON Transforms

`translate(x, y, * elements)`
translates the coordinate system of enclosed elements

`rotate(angle, * elements)`

rotates the coordinate system of enclosed elements

`flip(x, y, * elements)`
flips the coordinate system of enclosed elements over (x,y)

`scale(x, y, * elements)`
scales the coordinate system of enclosed elements

4.6 SUMMON Input specifications

`input_motion('left'|'middle'|'right', 'up'|'down', ['shift'], ['ctrl'], ['alt'])`

specifies a mouse motion input

`input_key(key, ['shift'], ['ctrl'], ['alt'])`
specifies a keyboard input

`input_click('left'|'middle'|'right', 'up'|'down', ['shift'], ['ctrl'], ['alt'])`

specifies a mouse click input

4.7 SUMMON Miscellaneous

`hotspot('over'|'out'|'click', x1, y1, x2, y2, proc)`
constructs hotspot for a region that activates a python procedure 'proc'

`hotspot_click(cannot be invoked on commandline)`
activates a hotspot with a 'click' action