PL Research: Examples

John Loeber
April 28, 2015

The following are examples of use of the image-editing language as I envision it. With no syntactical conventions defined as of yet, all of the code is pseudocode, so as to convey the workflow by which one might manipulate images.

The application/IDE will have three components: a graphical display, a data panel that shows the current *state* of all visual elements, and a code panel in which the user may enter high-level expressions to programmatically manipulate groups of images. Any change made in any one of these three components can be propagated to the other components. This propagation would be done automatically by the program so as to ensure that all three components are always synchronized. (Imagine a 'propagate' button that the user pushes when they've finished making a small change.) The code panel also acts as a *history* of all changes made in the program. Commands entered in the code panel all fall into two categories:

1. *Discarded Code* – code that is executed once, i.e. changes the data, and thereafter is "commented out," remaining in the window only for historical purposes.

2. *Persistent Code* – code that remains active for all time. For example, variable declarations are persistent.

Question: the history could become large for big projects due to a large volume of discarded code, making the little persistent code difficult to read/manage. Should the history go into a separate panel/tab?

Without further ado, I present how a user might do some exemplary tasks programmatically. I denote the content of the data panel with **DATA** and the content of the code panel with **CODE**. I denote comments with #.

Example 1.

Suppose the user wants to place three squares on the canvas in the top-left corner, and then space the squares horizontally with 20px between them. Here's one way of doing this:

First step: initalize the canvas.

DATA
```
canvas = (x:1500,y:500)
```
CODE

Second step: propagate changes in data to the code. (The changes to code are made automatically as a consequence of the 'propagate' feature.)

DATA
```
canvas = (w:1500,h:500)
```
CODE
```
# initialize canvas (w:1500, h:500)
```

Third step: write code to generate the squares. We write a template for all the squares, then generate three copies of it.

DATA
```
canvas = (w:1500,h:500)
```
CODE
```
# initialize canvas (w:1500, h:500)
template1 = square(w:20, h:40, x:0, y:0, color:blue)
generate(template1, 3)
```

Fourth step: propagate changes in code to the data. The changes to data are made automatically as a consequence of the 'propagate' function. Note that `generate` is discarded automatically – we only need it once – whereas `template1` is persistent, as we might use it again.

DATA
```
canvas = (w:1500,h:500)
element1 = square(w:20, h:40, x:0, y:0, color:blue)
element2 = square(w:20, h:40, x:0, y:0, color:blue)
element3 = square(w:20, h:40, x:0, y:0, color:blue)
```
CODE
```
# initialize canvas (w:1500, h:500)
template1 = square(w:20, h:40, x:0, y:0, color:blue)
#generate(template1, 3)
```

Fifth step: group the three squares, and then call a spacing function on them.

DATA
```
canvas = (w:1500,h:500)
element1 = square(w:20, h:40, x:0, y:0, color:blue)
element2 = square(w:20, h:40, x:0, y:0, color:blue)
element3 = square(w:20, h:40, x:0, y:0, color:blue)
```
CODE
```
# initialize canvas (w:1500, h:500)
template1 = square(w:20, h:40, x:0, y:0, color:blue)
#generate(template1, 3)
group1 = group(element1,element2,element3)
xspacing(group1,px:20)
```

Note: as I was writing the above, I realized that it might be easier for a group to be yielded directly by the `generate` statement. To be decided/engineered!

Sixth step: hit the “propagate” button to propagate the changes. Note that the variable declaration of group1 is persistent, and calling the spacing command is discarded.

DATA
```
canvas = (w:1500,h:500)
element1 = square(w:20, h:40, x:0, y:0, color:blue)
element2 = square(w:20, h:40, x:40, y:0, color:blue)
element3 = square(w:20, h:40, x:80, y:0, color:blue)
```
CODE
```
# initialize canvas (w:1500, h:500)
template1 = square(w:20, h:40, x:0, y:0, color:blue)
#generate(template1, 3)
group1 = group(element1,element2,element3)
#xspacing(group1,px:20)
```

And the user is done. Note that though the final product is nine lines, the user only wrote five of them, which seems like a reasonable demand of the user.

Let’s take this a little bit further. Suppose the user decided that the 20px spacing didn’t look good, and wanted to space all elements evenly horizontally. This could be achieved with a single command in code: `xspacing(group1)` where the missing arguments default to an even spacing across the canvas.

What about if this is also not to the user’s liking, and they really want the squares to be spaced horizontally across the canvas with proportional distances of 1/3 and 2/3 between the squares? This can be done in code in two lines, like this:
```
list1 = [1/3,2/3]
xspacing(group1,proportion:list1)
```

Where the `proportion` argument allows the user to submit a list such that `sum(list)==1`. There's a nice opportunity here for Haskell-style list comprehensions to generate such lists. At this point the textual representation of the graphic will be this:

DATA
```
canvas = (w:1500,h:500)
element1 = square(w:20, h:40, x:0, y:0, color:blue)
element2 = square(w:20, h:40, x:500, y:0, color:blue)
element3 = square(w:20, h:40, x:1000, y:0, color:blue)
```
CODE
```
# initialize canvas (w:1500, h:500)
template1 = square(w:20, h:40, x:0, y:0, color:blue)
#generate(template1, 3)
group1 = group(element1,element2,element3)
#xspacing(group1,px:20)
list1 = [1/3,2/3]
#xspacing(group1,proportion:list1)
```

At this point, the user might decide that this proportional spacing is nice, but they really want there to be a 20px margin between the squares and the left and right borders of the canvas. I'm really not sure how this *should* work, but these are some tentative ideas:

1. The user could go into the data, and change `element1` and `element2` to have `x:20` and `x:1460` respectively, then propagate those changes. (Note that it's 1460 because there's a 20px margin and the square itself is 20px wide.) Then the user could uncomment `#xspacing(group1,proportion:list1)` and then propagate the changes.

   Note that this implementation leaves many edge-cases to be considered: it would only work if the `xspacing` command were written so as to take the left- and right-most visual elements as fixed, and then to space the elements in-between according to the proportions. But this conflicts with the earlier way in which we applied the even horizontal spacing to default arguments (which also moved the rightmost element appropriately to fill the horizontal space).

2. The user could use some spaceholders and re-group visual elements. This would, of course, need a rigorous definition of spaceholders and exactly how they function. Regardless, I imagine that in the code (or perhaps in the data?) the user would enter something like
   ```
   newElement = spaceholder(w:20,h:0) + element1
   newElement2 = spaceholder(w:20,h:0) + element3
   newGroup = (newElement,newElement2,element2)
   xspacing(newGroup,proportion:list1)
   ```

3. Maybe this is a sufficiently common task to be accommodated by its own function or property. Maybe `xspacing` should have properties `marginleft` and `marginright`, so the user only has to enter:

   ```
   xspacing(group1, proportion:list1, marginleft:20, marginright:20)
   ```

   and be done with everything. This seems like the cleanest approach.

4. Above, I mentioned that maybe this should be accommodated by a function. It's possible to envision cases in which the user would have lots of groups to be spaced and wouldn't want to manually set lots of margins, so there needs to be an overarching way. It would be easy to write a function that preserves proportional spacing between elements. In practice, it might look like this:

   ```
   marginate(group1,marginleft:20,marginright:20)
   ```

Example 2.

Suppose you're working on some design project. You want to tile ten images diagonally across the canvas, lower their opacity, and add a title.

The program will have some graphical way to import images (`File` → `Open` → `...`). Suppose you use this to import your ten images and initialize your canvas as usual. The state of the program will be:

DATA

```
canvas = (w:500,h:500)
element1 = import(w:100, h:100, x:0, y:0, "/home/dogfood.jpg")
element2 = import(w:200, h:240, x:0, y:0, "/home/catfood.jpg")
element3 = import(w:400, h:405, x:0, y:0, "/home/etc.jpg")
element4 = import(w:400, h:405, x:0, y:0, "/home/etc2.jpg")
element5 = import(w:400, h:405, x:0, y:0, "/home/etc3.jpg")
element6 = import(w:400, h:405, x:0, y:0, "/home/etc4.jpg")
element7 = import(w:400, h:405, x:0, y:0, "/home/etc445.jpg")
element8 = import(w:400, h:405, x:0, y:0, "/home/etc5.jpg")
element9 = import(w:400, h:405, x:0, y:0, "/home/etc6.jpg")
element10 = import(w:400, h:405, x:0, y:0, "/home/etc800.jpg")
```

CODE

```
# initialize canvas (w:500, h:500)
# import element1(w:100,h:100,x:0,y:0,"/home/dogfood.jpg")
```

. . . (similar statements for the other elements) . . .

```
# import element10(w:400,h:405,x:0,y:0,"/home/etc800.jpg")
```

This example highlights that I have to think about the code panel and its double function as a history: in this case, just looking at eleven lines of effectively dead code just feels unclean. The history should probably be relegated to another panel (perhaps a tab from code) with little adverse impact on the functionality of the program.

I will also need an easy way to group lots of elements (perhaps mass-selection would be most easily done via the graphical interface.) Regardless of how it's done, suppose that we group the elements in the code panel:

```
group1 = group(element1,element2,...,element10)
```

Note that I use ellipses just for display purposes, instead of the other elements. Entering ellipses would not be interpreted by the DSL.

We know that we have some images that aren't square. But to tile them diagonally, we'll want them to be square. So we write in the code panel and propagate:

```
map(group1,resize(square))
```

Now all our images are square. But they're not all the same size. We want them to be the same size. We also want them to all to fit the canvas exactly. Since the canvas has dimensions 500×500 and we have 10 square images, it's clear that we need to resize all our images to be 50×50. So:

```
map(group1,resize(w:50px,h:50px))
```

Finally, the diagonal tiling effect can be achieved by setting x- and y-spacings of 0 pixels:

```
xspacing(group1,px:0)

yspacing(group1,px:0)
```

Thus, the ending state of the program will be:

DATA

```
canvas = (w:500,h:500)
element1 = import(w:50, h:50, x:0, y:0, "/home/dogfood.jpg")
element2 = import(w:50, h:50, x:50, y:50, "/home/catfood.jpg")
element3 = import(w:50, h:50, x:100, y:100, "/home/etc.jpg")
element4 = import(w:50, h:50, x:150, y:150, "/home/etc2.jpg")
element5 = import(w:50, h:50, x:200, y:300, "/home/etc3.jpg")
element6 = import(w:50, h:50, x:250, y:250, "/home/etc4.jpg")
element7 = import(w:50, h:50, x:300, y:300, "/home/etc445.jpg")
element8 = import(w:50, h:50, x:350, y:350, "/home/etc5.jpg")
element9 = import(w:50, h:50, x:400, y:400, "/home/etc6.jpg")
element10 = import(w:50, h:50, x:450, y:450, "/home/etc800.jpg")
```

CODE

```
# initialize canvas (w:500, h:500)
# import element1(w:100,h:100,x:0,y:0,"/home/dogfood.jpg")
...(similar statements for the other elements) ...
# import element10(w:400,h:405,x:0,y:0,"/home/etc800.jpg")
group1 = group(element1,element2,...,element10)
#map(group1,resize(square))
#map(group1,resize(w:50px,h:50px))
#xspacing(group1,px:0)
#yspacing(group1,px:0)
```

There may of course be situations in which a solution to a problem is not as clean or easy as in the above examples (suppose the user wants the images to be of different dimensions, or a blank space in the middle, etc.). Note that there are effectively three use cases:

1. The user wants to apply some general pattern to all elements.
2. The user wants to apply some general pattern to some elements, and some specific/elementwise unique pattern to some other elements.
3. The user wants to do something that cannot be practically generalized.

The software and image-setting language will make it possible to operate swiftly in each of these cases:

1. It's straight-forward to group elements as appropriate, and then to use functions that operate on groups in order to achieve the desired results.
2. The general pattern can be applied to a subset of the elements again using group syntax/techniques. Note that the user should be allowed to declare groups that are intersections, unions, complements, etc. of groups. This would add significant flexibility to the capacity of functions operating on groups. Smaller patterns can also be set using group functions. Extremely particular attributes can always be set manually by editing the data.
3. Anything that can't be generalized can just be hard-coded in the data. Note that in this domain, the number of visual elements involved in creating a non-generalizable pattern must be quite small: as the number of elements increase, patterns emerge among subsets of the elements, making it possible to generalize. Also note that non-generalizable designs are an unlikely use case, anyway.

Other examples I should work on:

1. Workflow when deleting elements/refactoring the image.
2. Generating algorithmic art. This tool should be good for this purpose (though not for the *automatic* generation of algorithmic art). That is, this tool should allow artists to implement designs algorithmically and work iteratively.