Snap*!* Graphics: Computational Photography for Introductory Computer Science Curriculum

Michael Ball
UC Berkeley
Department of EECS
michael@cs10.org

ABSTRACT

Computational Photography, and image processing are topics which are generally reserved for upper division, or even graduate level computer science students. This paper explores whether it is possible to introduce the basic techniques of image modification to students with no prior programming experience through a visual programming language called Snap*!* which is currently being used in many "CS0" style classrooms.

Categories and Subject Descriptors

K. Computing Milieux, **K.3** [**COMPUTERS AND EDUCATION**], **K.3.2** Computer and Information Science Education, **Subjects:** Curriculum; Computer science education

General Terms

Documentation, Design, Experimentation, Human Factors.

Keywords

Computational photography, image processing, visual data, Snap*!*, CS0, curriculum, visual programming languages, graphics

1. INTRODUCTION

Snap*!* is a web based, visual programming language inspired my MIT's Scratch.[1][2] Like Scratch, Snap*!* supports a full range of programming projects, with an emphasis on using "sprites" as the main metaphor. Every sprite has at least one costume, and while there are a range of visual "effects" built into Snap*!*, they are limited and there is no customization and direct way to modify a sprite's costume in code. The project extends Snap*!*'s built in abilities by enabling direct access to the pixel data as well as creating some built in functions for common image processing operations such as image convolutions and warps.

2. Why Snap*!*

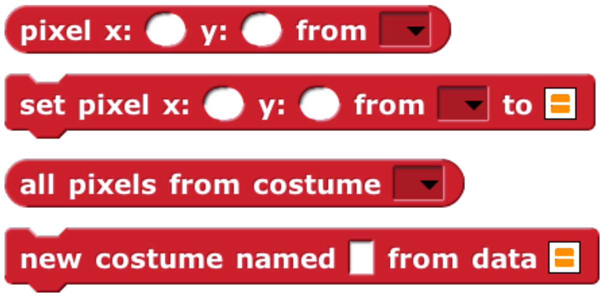
Snap*!* is currently in use at UC Berkeley as part of the CS10 class, "The Beauty and Joy of Computing", BJC.[3] The BJC project is also in place at dozens of high schools across the United States, and will soon be a course on edX.org for anyone to join.[4][5] Snap*!*'s ability for students to create any custom function, as well employ techniques like higher-order functions (lambda) means that students can easily extend the given inputs to do just about anything that are interested in. This enables Snap! to have a relatively small set of built in functions, and allows students to have more opportunities to write their own code. Many[1] students have reported that they enjoy working on projects which have visually distinctive results, so this work will hopefully extend the projects which are available to students.

3. Design Approach

3.1 Basic Primitive Operations

The goal was to provide a few basic primitive blocks which would enable complete manipulation of images. I settled on four blocks which are related to each other. These blocks are:

- Get Single Pixel from a costume
- Set a single pixel in a costume to the colors in a list
- Return a nested array of all pixels in a costume
- Create a new costume from an array of pixels



3.1.1 Colors

The goal of this project is to abstract away some of the details with regards to image manipulation. In JavaScript, all images always have RGBA color channels, so this is how Snap*!* represents colors internally. For the time being, these colors can be represented as a direct color object, or an array of four values.

Ultimately, this setup seems to be far from perfect. While an array of four values makes the matrix operations and primitive blocks simple to implement for this project, I believe it would be much better to simply treat each image as a 2-D array of "colors" and then architect the primitive functions to follow this format. Color objects are more visual and would hopefully have less cognitive load than trying to figure out which RGBA values represent which color.

[1] "Many" is a woefully unscientific term, and I currently don't have much data on the subject as to an exact quantity. However, 6 semesters of TA-ing for CS10, as well as many conversations with high school teachers and professors confirm the facts that graphics are cool.

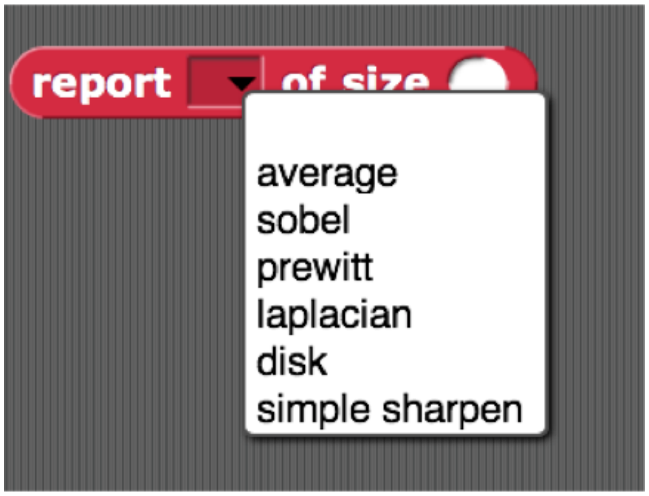
However, I always intend to keep the option for integer representations available because this is a necessity for many algorithms.

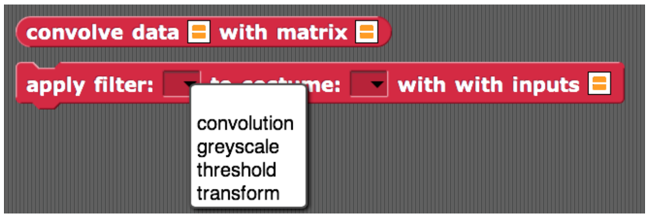
3.2 A “Starter Kit” of Functions

While, these few primitives would be all students need to implement even advanced operations such as 2D convolutions, I’ve chosen to build in a handful of examples and functions which enable students to get started playing around more quickly. Spending time learning how a convolution works at the first step would take away from the ability to learn some higher level algorithms. Additionally, implementing these functions natively in JavaScript has a speed advantage over implementing the same functions in Snap*!* Long term, there is a lot of flexibility as to what should and shouldn’t be included in a default set of functions presented to students. Snap*!* has support for “libraries” like many languages so this provides a good way to extend the capabilities in the future, or to better organize the functions given to students.

3.3 Examples

These are a two other built in functions, which enable advanced image processing. The report function is similar in some ways to `fspecial()` in MATLAB. However, there is a current issue that the block as a static size parameter which does not make sense from options like the Sobel operator, but is necessary for the average filter or the disk filter.




convolve data with matrix
apply filter: to costume: with with inputs
convolution
greyscale
threshold
transform

The second block here applies these filters directly to the selected costume element. The greyscale and threshold filters are some simple examples, but there could be others. In general, the layers of abstraction shown here is a prototype that needs further refinement. Eventually, I would expect the second function to be removed and some of the functionality transferred to a block which reports a new image matrix since this better encourages functional programming paradigms. (The reason for this block is noted below).

4. IMPLEMENTATION

All the work for this project is an extension of the current Snap*!* environment and codebase. (Both codebases are open source and on Github. [6][7]).

JavaScript has no built-in library nor methods for traditional matrix math. It does support nested arrays, however, all image data is drawn using the HTML5 Canvas element [8]. The canvas element stores images as a 1-D array of length 4*width*height, representing each pixel as 4 elements that are R, G, B and A values stored as unsigned 8-bit integers. (Using integers is actually helpful for students are they don’t have to worry about decimal operations or floating point errors.)

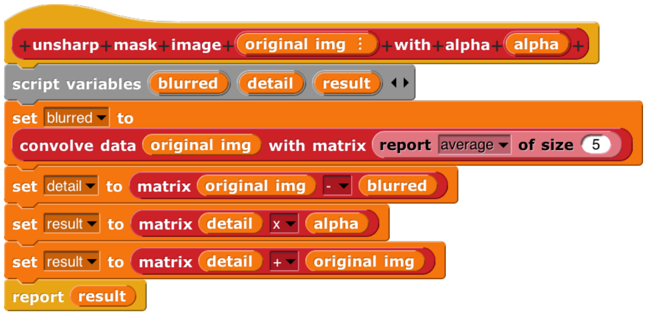
Due to this, I implemented as much of the work to do the image manipulation simply using for-loops over the pixels in each array. This works alright in the case of convolution. However, I ran into a fair bit of trouble implementing a full inverse warp method, which turned out to be rather slow and buggy. Fortunately, the Canvas specification has a `setTransform()` method, which applies a 6-point affine transformation to the image data. This method is relatively fast base it is natively implemented by all recent browsers. (Unfortunately, this method also occasionally had some problems that cause a transformation not to appear. This still needs to be tracked down, but I believe this is an issue with Snap*!* and the not the underlying implementation of the Canvas.)

The code is fairly straightforward for most examples. In the Snap*!*-Graphics repository, all the code is under the `graphics/` subdirectory. In particular, the files `objects.js` and `threads.js` contain the bulk of the modifications to support this project.

4.1 Graphics Projects In Snap*!*

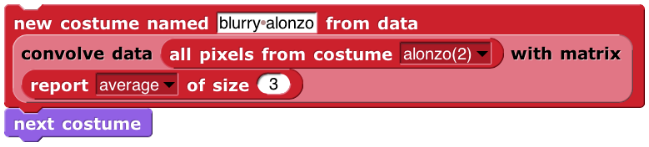
Here are a couple small examples to show how easy it is to create some simple effects using the tools provided. This first block is an unsharp mask. The second shows repeated blurring of a costume. It would not be too hard to extend the repeated blurring into a set of blocks that are able create a Gaussian stack. (The code for generating a Gaussian matrix still needs to be integrated into Snap*!*)

Unsharp Mask:



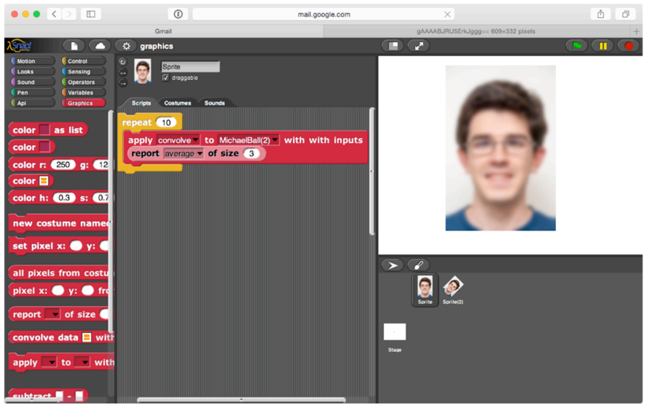
```
+unsharp+mask+image+ (original img ⋮) +with+alpha+ (alpha) +
script variables (blurred) (detail) (result) ◂▸
set blurred to
  convolve data (original img) with matrix (report average of size 5)
set detail to matrix (original img) - (blurred)
set result to matrix (detail) x (alpha)
set result to matrix (detail) + (original img)
report (result)
```

Single Blur:



```
new costume named blurry•alonzo from data
  convolve data (all pixels from costume alonzo(2)) with matrix
    (report average of size 3)
next costume
```

Repeated Blurring:



```
repeat 10
    apply convolve to MichaelBall(2) with with inputs
        report average of size 3
```

5. FUTURE WORK

There is a lot of work that still needs to be done to make these functions ready for students to use. By far the largest problem is the lack of error handling and robustness to user-mistakes. This is particularly a problem as sometimes too large amounts of data can cause browsers to crash, but also because students who are just starting to program love to break things.

Right now, the use of an affine warp is tired directly to manipulating an existing costume, I would like to move this functionality to a separate block which allows a user to warp an input image to a new output of a specific size.

Finally, speed and memory use are of particular concerns. These need to be monitored and tested significantly before the graphics system gets wider use as it is likely not to perform very well on older hardware.

Outside the scope of this project, I plan to continue working on those tools over the next few months. The goal is to have a basic example curriculum written that will work for students by the summer offering of CS10.

6. ACKNOWLEDGMENTS

Thanks to Professor Efros and our GSIs for an amazing class and a great semester!

7. REFERENCES

[1] Brian Harvey and Jens Mönig, http://snap.berkeley.edu/

[2] Scratch, http://scratch.mit.edu/

[3] Dan Garcia, Brian Harvey, Michael Ball, et all. http://cs10.org/

[4] The Beauty and Joy of Computing, http://bjc.berkeley.edu/

[5] BJC1.x, on edX, http://bjc.link/bjc1x

[6] Snap! main source code repository Snap—Build-Your-Own-Blocks on GitHub, https://github.com/jmoenig/Snap--Build-Your-Own-Blocks

[7] Snap!-Graphics, code repository for this project on GitHub: https://github.com/cycomachead/snap-graphics

[8] HTM5 Canvas Specification, W3C Working Draft: http://www.w3.org/TR/2dcontext/