CREATIVE GRAPHICAL CODING VIA PIPELINED PIXEL MANIPULATION

Dale E. Parson
Kutztown University of PA
parson@kutztown.edu

ABSTRACT

Creative coding is the act of computer programming intended to create aesthetic artifacts in one or more digital media such as graphical images, animated videos, computer games, or musical performances. Visual artists and musicians use computers to compose, to render, and to perform. Algorithms remain as important as they are for any computer program, but their intent is to inspire, or at least to entertain, in contrast to more utilitarian applications of algorithms. This paper outlines the software structures and aesthetic perspectives of two novel algorithms for the creative manipulation of pixels in the Processing language. The first algorithm focuses on manipulating a digital canvas after it has been painted by a set of animated virtual paintbrushes. The metaphor is visual memory, where a digital canvas remembers what was painted during prior animated frames, making manipulation of those memories accessible to the artist. The second algorithm focuses on using an image-processing pipeline to analyze and fragment copies of a photographic image for use in screening the original image. The inspiration is visual processing by the brain, in which different areas of the brain handle different aspects of stimuli arriving via the optic nerves before integrating them into a composite image.

KEYWORDS

computer graphics, creative coding, generative art, image processing, Processing language

1. Introduction

Digital art has been around for decades [1]. The advent of powerful graphical processing units (GPUs) and sophisticated programming environments has pushed the creation of this art into the aesthetic mainstream among professionals, academics, and students.

The author has been collaborating with electronic musicians and digital artists for the last 25 years, with collaborations increasing in frequency and sophistication during the last ten. Despite being a computer science professor, a full 70% of the author's students in spring 2018 are majors in Kutztown University's new Applied Digital Arts program offered by the Art and Art History Department. The work that informs this paper runs the gamut from teaching introductory animated graphics programming through exhibiting in multimedia exhibitions and concerts. This is a very exciting and energetic time to be involved in the creation of new modes of artistic expression.

This paper provides a high-level overview of the technical and aesthetic aspects of two generative compositional algorithms. Emphasis is on images rather than words, with links to four animated video recordings.

2. Related work

The author's involvement in creative graphical coding dates back to using the Logo language dialect of LISP [2-4] and its body-referential turtle geometry [5] coming out of the MIT Media Lab in the 1970s and 80s. Current work uses the Processing framework that also had its beginning in the MIT Media Lab [6,7]. This framework includes powerful and efficient graphics, animation, image-processing, and computer audio code libraries, and an integrated development environment / debugger oriented towards artists, with language bindings available for Java, Javascript, Python, and Java on the Android mobile operating system [8]. Programming reported in this paper is in Java.

The OpenGL Shading Language (GLSL) is a C-like programming language targeted to GPUs that could accomplish some of the results described here for Processing [9,10]. The author has stayed with Processing implementation techniques because of familiarity and the desire to explain these techniques to Processing-savvy students. GLSL would likely be more efficient for some of the mechanisms of Section 4.

Interactive and time-based digital art is a rapidly growing field [11]. The author is in ongoing collaborations with digital artists and musicians too wide ranging to list here. The current work derives primarily from spending time exploring these digital media in depth, as opposed to external influences.

3. Graphical Canvas as Short-Term Memory

The algorithm of Section 3 focuses on manipulating a digital canvas after it has been painted by a set of animated virtual paintbrushes. The metaphor is visual memory, where a digital canvas remembers what was painted during prior animated frames, making manipulation of those memories accessible to the artist. Section 3.1 gives an overview of the main aspects of a Processing *sketch* (Processing's name for a program), along with an overview of the specific algorithm. Section 3.2 explores aesthetic aspects. Using software paintbrushes is inspired in part by Adobe Photoshop [12], Adobe Illustrator [13], and similar 2D illustration programs, although those programs are used for static image capture and manipulation, not for live, interactive animation or creative coding.

3.1 PImage and pixel manipulation in Processing

Figure 1 is a UML (Unified Modeling Language [14]) class diagram showing the author's Processing sketch *ShapePaintEcho* and its relationships to Processing library classes. Every Processing sketch takes the form of a subclass of library class PApplet, which is where Processing supplies its seemingly global variables such as *width* and *height* (width and height of the display window in pixels), and *pixels*, which is the 1D Java array housing a RGBA integer pixel value (Red, Green, Blue, and Alpha, Alpha being a measure of opacity) at each element; the pixels array has (width*height) elements, one for each display pixel. PApplet also supplies a large number of seemingly global functions, examples here being *loadPixels*() that downloads hardware display buffer pixel values into the pixels array, *updatePixels*() that copies the pixels array into the display buffer, and *createImage*() that creates a photo-like PImage object of a specified (width*height) pixel size. PApplet supplies many conventional drawing functions such as *background*(), *ellipse*(), *rect*(), and *triangle*() that do not require direct manipulation of pixels by the sketch. Figure 1 focuses on pixel manipulation functions because pixel manipulation is the focus of this paper.

Processing programmers write what appear to be global variables and functions, much like the C subset of C++, albeit in Java for Processing. Processing generates class wrapper code, here for class ShapePaintEcho, as a subclass of PApplet. Sketch programmers can use library variables and functions residing in PApplet without the knowledge or syntactic trappings of classes. Acquiring the language is simpler than the usual practice of writing Java classes right from the start. Sketch programmers can define inner classes when useful for modeling graphical entities. Originally the library class PApplet derived from Java's Applet class, although Processing 3 eliminates that class derivation.

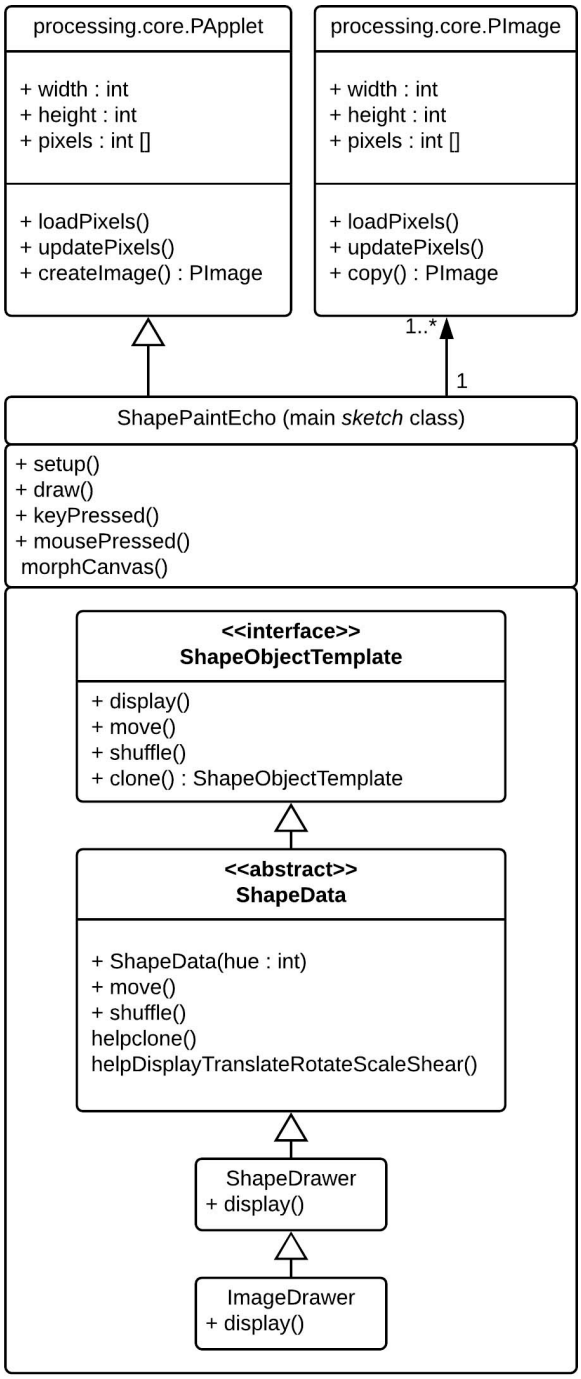


Figure 1: Class diagram for using a canvas as memory

Sketch ShapePaintEcho supplies the *setup*(), *draw*(), *keyPressed*() and *mousePressed*() functions of Figure 1. Function *morphCanvas*() is specific to this sketch, while the others are known to Processing. The Processing run-time framework arranges to call the sketch class' setup() function once, at the start of sketch execution, as the sketch's "main" function; setup() sets the final (width*height) size of the graphical window, along with other fixed and mutable sketch properties. Processing then schedules the sketch's draw() function to run periodically at the *frameRate*; the sketch can adjust the default

frameRate of 60 frames per second during execution. Periodic invocation of draw() is the basis of animation. Typically, draw() invokes *background*() to erase the previous frame's image and fill the display with a background color. Subsequent draw() code supplied by the sketch programmer creates shapes, loads and plots image files, performs geometric transformations, and clips the display when needed. Processing calls sketch functions *mousePressed*() and *keyPressed*() upon entry of mouse and keyboard data, respectively.

ShapePaintEcho defines one nested Java interface and three nested classes. Interface ShapeObjectTemplate specifies the four diagrammed functions for displaying, moving, shuffling (randomizing location and other properties), and cloning (copying) a graphical object modeled by a subclass of ShapeObjectTemplate. Abstract class ShapeData supplies some data fields and helper functions used by subclasses, and concrete classes ShapeDrawer and ImageDrawer model and display various shapes and images specific to this sketch. Using classes to model animated graphical entities gives students intuitive, immediate feedback to reinforce learning the object-oriented structuring concepts of inheritance and polymorphism.

A basic sketch would draw() and move() shape objects, shuffle() for re-initialization, and clone() for copying, on objects that store graphical state such as object location and color. That is a partial description of this sketch. What is novel about ShapePaintEcho is that its draw() function does not erase the prior frame upon each invocation. Instead, the keyPressed() and mousePressed() functions provide means for the user to manipulate the canvas of images painted onto the display by prior invocations of draw(). What follows is simplified pseudo-code for morphCanvas() as called near the start of the draw() function:

1. Use loadPixels() to copy the display's pixels from the previous draw() frame into the pixels array.
2. Create a new, blank PImage object with the same (width*height) as the display and copy the just-loaded display's pixels array into it.
3. Perform graphical rotation, scaling, and shearing (scale shape width as a function of height or vice versa) on this PImage object as specified by prior user input via keyPressed() and mousePressed().
4. Plot the PImage as just another graphical object on the current display. Scaling the prior frame's PImage up causes clipping of its former outer areas. Scaling the prior frame's PImage down fills using background color pixels (typically black) into the display periphery. The morphed PImage over-writes the previous display with its modified copy.

After returning from morphCanvas(), draw() display()s and move()s the current set of ShapeData modeling objects, i.e., the paintbrushes. "Paintbrush" object construction, elimination, and randomization (shuffle()ing) are under the control of keyboard commands. Thus draw() is essentially a two-step operation: 1) Load, manipulate, and plot the pixels of the previous frame; 2) plot the current shapes on top of the canvas morphed from the previous frame.

Given the repeated copy and mutate operations of morphCanvas(), the "background" plotted by this function is a recursive visual function that incorporates not only the previous draw() frame, but also its predecessor, etc., up to the limits of display resolution and clipping. Hence the metaphor of canvas-as-memory, because the canvas remembers all previous shape-object-display steps, up to those limits.

3.2 Aesthetics of manipulating a canvas as memory[1]

The illustrations on the next page show four static screen shots of live user interaction with ShapePaintEcho. The first, labeled *The Heart of the Machine*, shows 8 shapes – 5 rectangles, 2 triangles, and a quadrilateral – being plotted in the current 60th-of-a-second frame. The user has used the mouse and keyboard commands to direct morphCanvas() to scale-down repeatedly the prior frames, moving their apparent trails towards the center. Discontinuities in some of the trails indicate movement by their shapes in prior frames, although most appear to be immobile, with only canvas-trail image processing occurring. The trails blur because scaling results in misalignment of original pixel locations with their new, scaled locations. Scaling pixel coordinates is inexact, requiring lossy interpolation and averaging of location and color information [15]. Canvas rotations of other than 90 and 180 degrees also result in pixel misalignment, as do shearing, causing aging images on the canvas to cloud increasingly over time. Static image editing tools such as Adobe Photoshop do not finalize pixel interpolation until the user has signalled completion of a series of geometric transforms, thereby reducing error. Also, a Photoshop user can select the interpolation algorithm to use on image resizing. The issue for ShapePaintEcho is not so much a matter of Processing's pixel interpolation algorithms, as it is a matter of the frequency of lossy geometric transforms. At a frameRate of 60 frames per second, there can be 60 cumulatively-lossy transforms in a second, with more if there is a combination of lossy scaling, rotation, and shearing. Misalignment accumulates rapidly. This image clouding is not a problem for sketches like ShapePaintEcho, however. It is simply an aspect of the artistic medium to be utilized by a digital artist. The author added a feature to a later version of the sketch that allows a user to keep copies of an original shape in a list

[1] The reader is encouraged to view the 2-minute video recording and the 4-minute video recording at https://drive.google.com/open?id=1OHNdA_lNwbuGO86tHa2JMGkUhUzCY-1 and https://drive.google.com/open?id=14hBoJVw_3ogh2pci-hXQT7phBs6wJrfY to get a better sense of the interactive, animated execution of ShapePaintEcho than is possible with a static set of images.

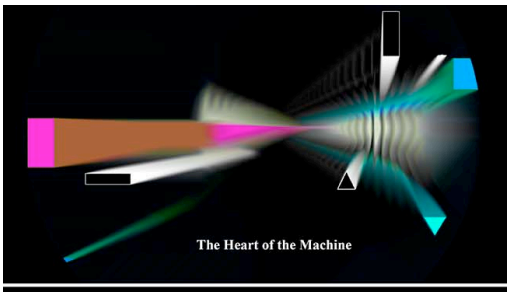
for exact re-plotting in subsequent frames. The artistic effect of keeping temporal object histories can be useful, but it is entirely different from the canvas-based morphing that remains the aesthetic keystone of ShapePaintEcho.

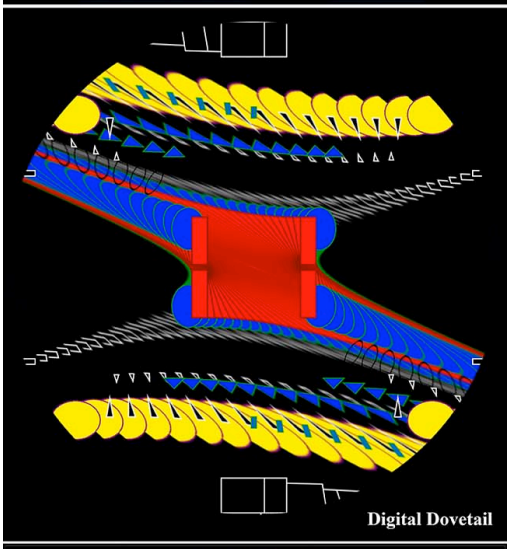
The compositional intent for *The Heart of the Machine* is the set of digital veins and arteries connecting into a virtual heart, with the morphing canvas serving as the pumping heart.

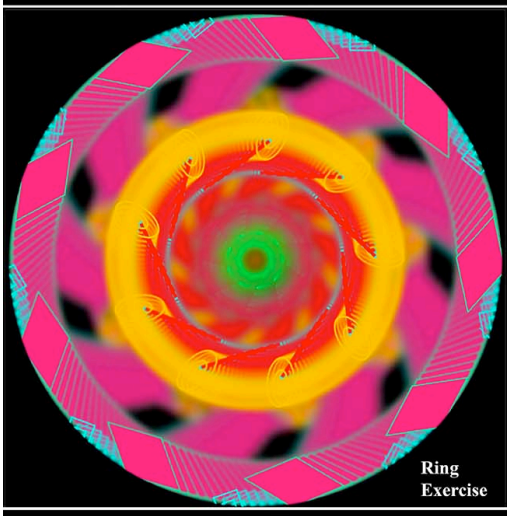
The second image called *Digital Dovetail* grows from an exact, non-lossy, 180-degree cumulative rotation of the canvas. There is no scaling of this canvas, with lossless trails contributed by the un-erased movement of the shapes. It symbolizes transition from approximation at the outer boundaries to precision at the center.

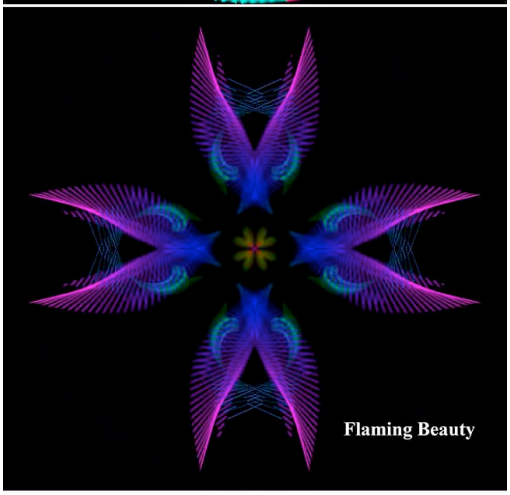
The *Ring Exercise* shows the cumulative results of graphical object movement with interactive, lossy canvas rotation and repeated up-and-down scaling. The second band from the outer edge has been scaled up and down by the user multiple times, hence the fuzziness when compared to the adjoining circular layers.

The final image shows the effects of a paintbrush patterned after the Chinese *I Ching* [16] character *Flaming Beauty*, without fill – two solid, parallel lines with one broken line between them – with the canvas receiving lossless reflections around the X and Y center axes and 90 degree rotations. This set of images gives some idea of the expressive range of ShapePaintEcho. The reader is strongly encouraged to view the two videos linked in Footnote 1 to get a more complete sense of the interactive capabilities of this sketch.









4. A Multithreaded Pipeline for Image Processing

The algorithm of section 4 focuses on using an image-processing pipeline to analyze and fragment copies of a photographic image for use in screening the original image. The inspiration is visual processing by the brain, in which different areas of the brain handle different aspects of stimuli arriving via the optic nerves before integrating them into a composite image.

4.1 Infrastructure for image pipeline stages

Figure 2 is a UML class diagram showing the author's Processing sketch *PhotoMontage* and the author's reusable Java library package *PixelVisitor*. The complexity of this class diagram is due to the fact that, unlike ShapePaintEcho, PhotoMontage and similar sketches operate on individual pixel data at the level of Java code. ShapePaintEcho uses optimized Processing library classes and functions to copy display pixels into a PImage object, rotate-translate-shear that PImage object, and then plot it. ShapePaintEcho manipulates individual pixels at the Java-sketch level in only a few places. In

contrast, much of the work of PhotoMontage is manipulation of individual pixels. Performing this work in the Processing display thread slows the effective frameRate down to the point of uselessness. Typical tests of the algorithms discussed here run in the 5-frames-per-second range while attempting a frameRate of 60 because of the computational intensity of the task. ShapePaintEcho discussed in Section 3, in contrast, has no difficulty maintaining a 60-frames-per-second frame rate. The solution for PhotoMontage is to use multi-threading to achieve an interactive frameRate of 60-frames-per-second as explained in this section. Code was run on a 2.6 GHz Intel Core i7 with 8 hardware threads (4 dual-threaded cores) running Mac OS X 10.9.5 and Processing 3.3.6.

The classes tagged <<*active*>> in Figure 2 run in different threads from the Processing display thread that invokes draw() at the frameRate. They offload the work of individual pixel manipulation from this Processing thread, working in parallel. Active class *ImageLoader* has the cyclic job of loading the next JPEG or PNG image file in a sequence from the file system into a PImage object, invoking a set of multithreaded filters on one or more copies of this PImage object, and then passing the resulting array of PImage objects to the Processing thread via a thread-safe java.util.concurrent.SynchronousQueue object called *loaded* in Figure 2. There is also a thread-safe java.util.concurrent.CopyOnWriteArrayList object called *fnames* that allows both the Processing and ImageLoader threads to access the list of image files. ImageLoader provides an image-processing pipeline, preparing the next PImage array from the image file + filters while Processing's thread manipulates and displays the previously pipelined/filtered array of PImage objects derived from an image file.

Interface *PixelArrayVisitor* is the keystone of custom library package *PixelVisitor*. Each PixelArrayVisitor object reads an entire 2D PImage processed in a previous pipelined stage and writes a single 1D row of pixels for a resultant (width*height) PImage, with *width* pixels per row. Pixel width and height can number in the thousands, and in principle it is possible to run *height* PixelArrayVisitor active objects in a parallel worker-thread pool in order to accelerate pixel processing.

Abstract library class *PixelArrayVisitorHelper* stores the data fields shown, and class *PixelArrayVisitorContainer* stores a row of pixels produced by a PixelArrayVisitor object. Library interface PixelArrayVisitorFactory specifies a factory method for constructing an application-specific PixelArrayVisitor object.

The application classes derived from library classes of PixelVisitor in Figure 2 are PixelArrayVisitorColorSorter, derived from PixelArrayVisitor, and PixelArrayVisitorColorSorterFactory, a class for manufacturing PixelArrayVisitorColorSorter objects.

PixelArrayVisitorColorSorter is an active class that is at the heart of the image-processing algorithm of PhotoMontage. A PixelArrayVisitorColorSorter thread visits a row of pixels in an incoming PImage and produces 1-of-8 derived rows, one for each of the following colors: black, red, green, blue, yellow, cyan, magenta, and white. ImageLoader uses Processing's *posterize* filter [17] to determine the dominant color out of these 8 for each of the pixels in the original PImage row; each PixelArrayVisitorColorSorter worker thread creates a row for its color-of-8 with only those color-dominated pixels from the original image, with all other pixels in its output row being transparent (i.e., with an alpha value of 0). The opaque pixels in each of the 8 resulting, color-specific rows are identical to the pixels in the original, non-processed PImage. The 8 rows are 8 maps of per-pixel dominant color-of-8 from the original PImage. 8 PixelArrayVisitorColorSorter threads produce 8 PixelArrayVisitorContainer objects; a PixelArrayMapReducer object of Figure 2 combines all rows so produced into a collection of 8 complete PImage objects, one per dominant color.

PixelArrayMapReducer, which runs within the ImageLoader thread, implements a map-reduce algorithm on pixel manipulation in the original, functional-language sense of map-reduce. It uses a cached thread pool from library class java.util.concurrent.Executors to execute parallel worker threads as PixelArrayVisitorColorSorter objects. A cached thread pool is demand-driven in its number of threads, up to the number of available contexts (hardware threads, 8 for this laptop). PixelArrayVisitorColorSorter provides the mapping from an input PImage row to 1-of-8 output PixelArrayVisitorContainer objects, one per row-color combination. When all threads running as PixelArrayVisitorColorSorter objects have completed their work, class PixelArrayMapReducer reduces their returned rows into an array of 8 PImage objects. The ImageLoader thread then sends this PImage array to the Processing thread via the SynchronousQueue *loaded*. Figure 3 is a UML activity diagram that outlines the steps discussed in the current section.

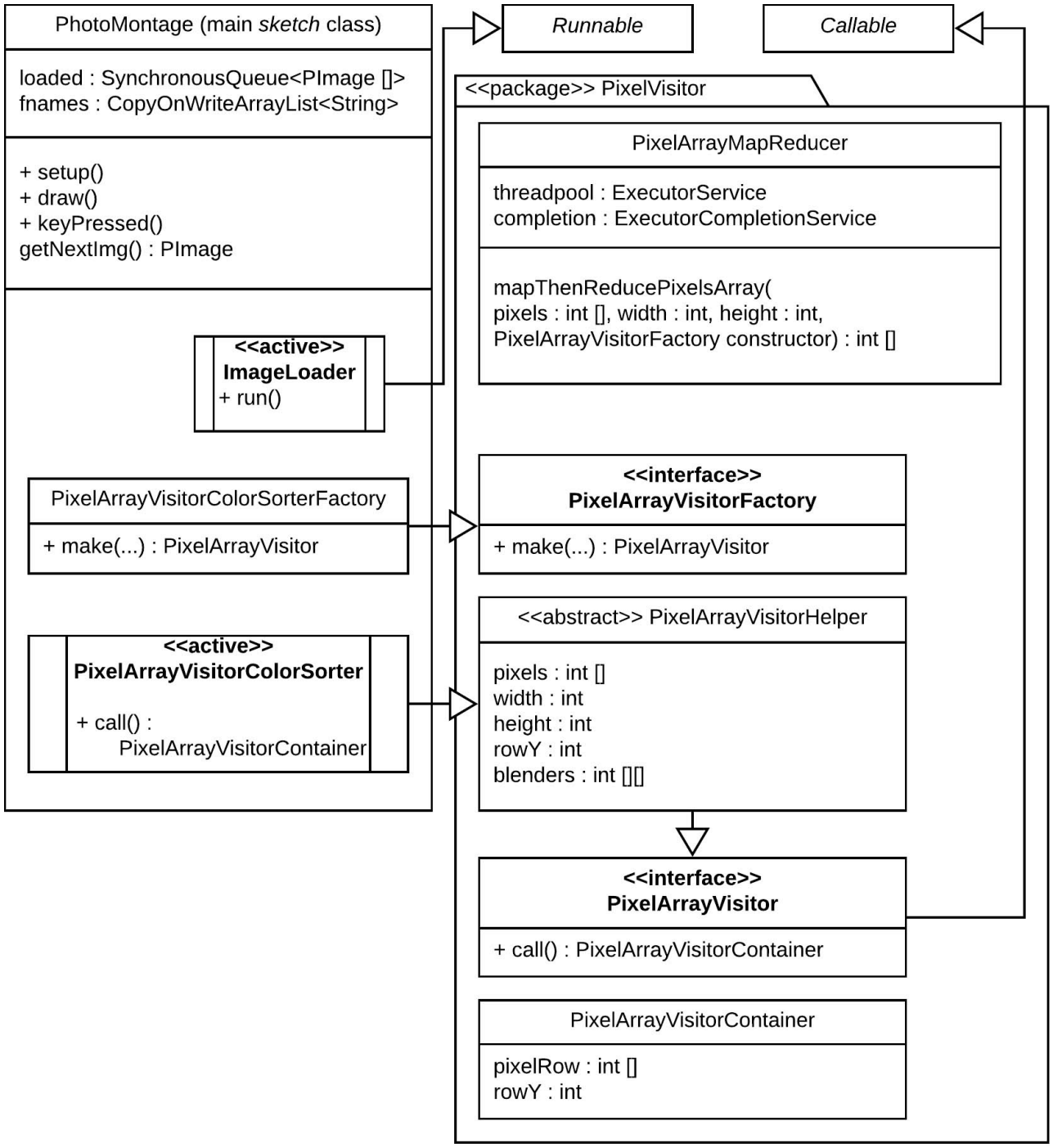


Figure 2: Class diagram for a multithreaded image processing pipeline

Processing's thread within the draw() function plots the returned original PImage as the primary image, after which it plots the 8 color-dominated PImages (with transparent pixels for other-than-dominant-color pixels) at various rotational angles as directed by the user. These 8 overlays act as curtains or veils over the original image. Successive invocations of draw() gradually reduce the opacity of these curtain overlays from 100% to 0%, exposing the original image behind them. PhotoMontage's draw() also extrudes and fragments the curtains at the beginning of displaying an image, gradually reducing the extrusion and fragmentation as it reduces curtain opacity, such that the curtain appears more like the original image as it unveils the original image through increased transparency. When the curtain reaches 100% transparency (i.e., 0% opacity), draw()

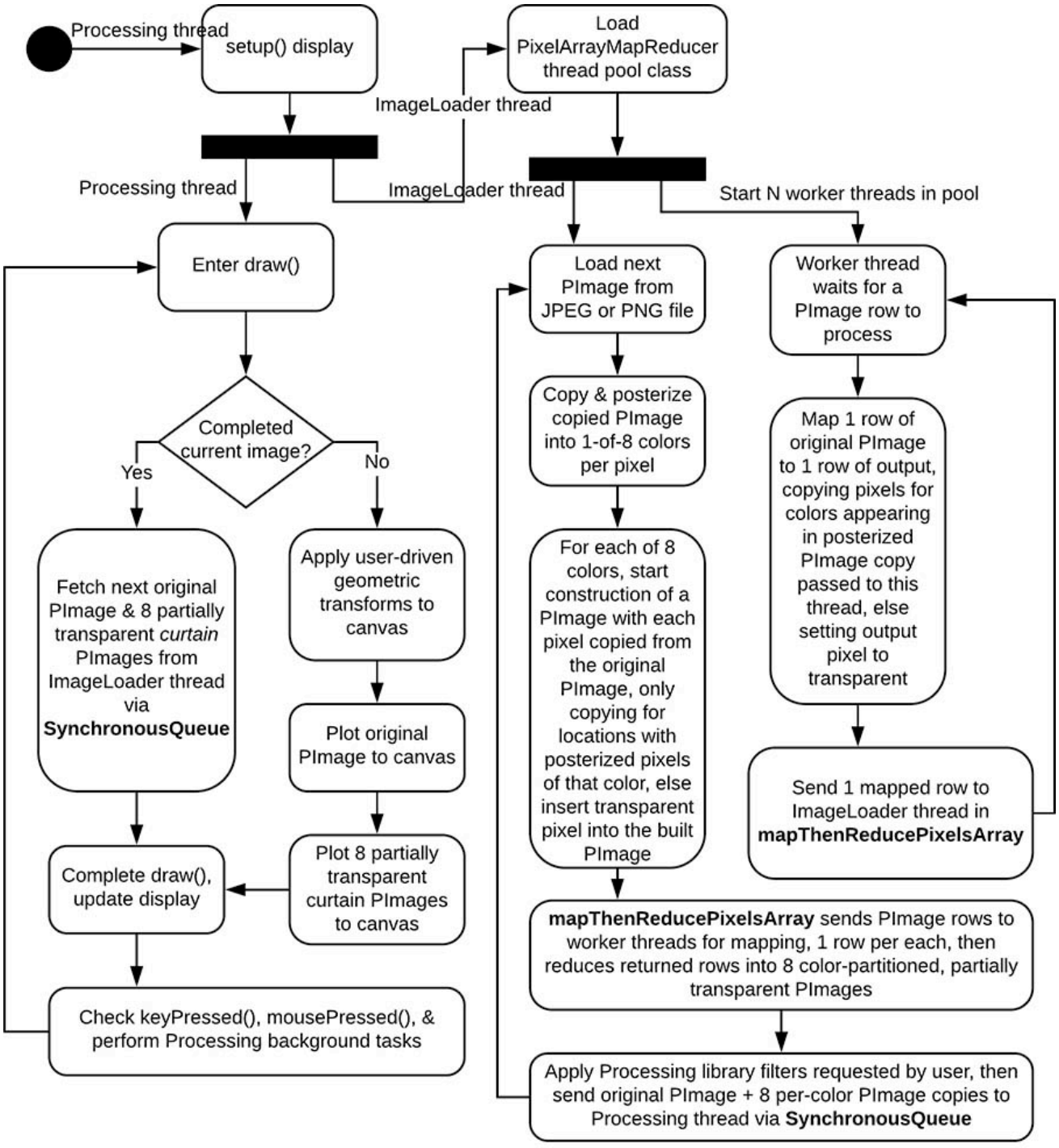


Figure 3: Activity Diagram for PhotoMontage & PixelVisitor interactions

dequeues the next array of the original + 8 color-partitioned PImages from the *loaded* queue and repeats the process. With each image, successive invocations of draw() perform an unveiling of the original image loaded from a file, using extruded, fragmented curtains made from copies of the original image, partitioned by dominant color. Animated display has the feel of assembling the original image from its compositional components, in a manner suggestive of image assembly in the brain.

4.2 Aesthetics of pixel screens as unveiling curtains

Even more than the aesthetics discussed for ShapePaintEcho, PhotoMontage is animated, interactive video in nature. The still images appearing in this section are useful for discussion purposes only, and unlike the

illustrations of Section 3, are not visual compositions in their own right. The reader is very strongly encouraged to view the 5:35 minute video *WildFlowers* and the 11 minute video *MovingArchitecture* linked in this footnote[2].

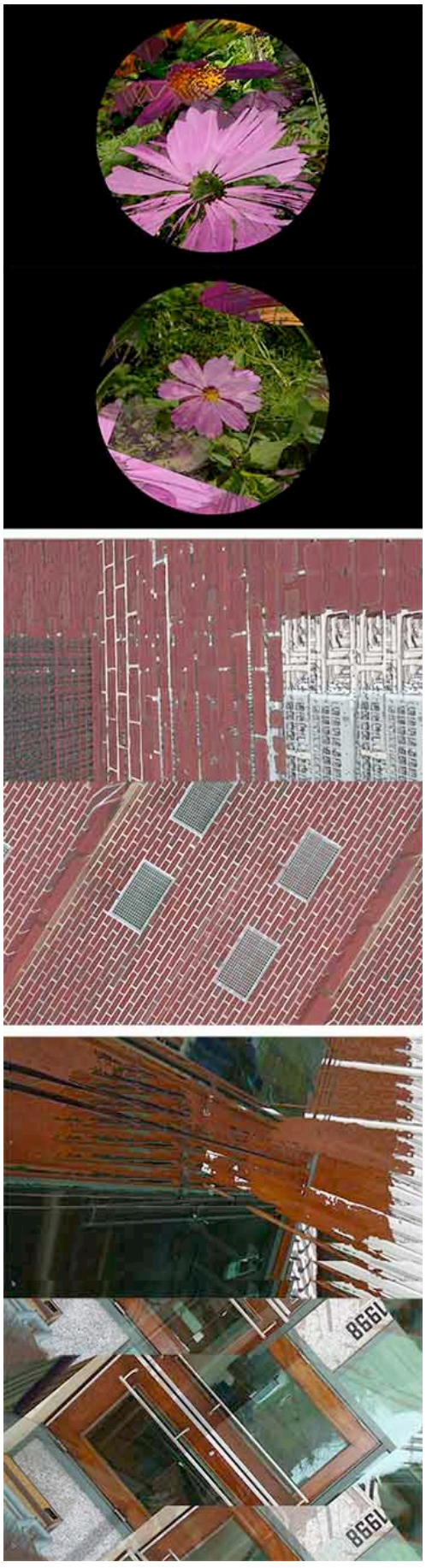
The illustrations to the right are screenshots of these two animated, interactive videos. There are three pairs of early-later screenshots, where the earlier image shows the color-partitioned, extruded, fragmented curtain with the original image underneath, and the later image shows the unveiled original image with some mostly transparent remnant of the curtain.

The two flower images at the top are the most revealing with respect to color partitioning in the curtains. Note how the yellow-dominated center of the flower (mostly the *stigma*) has separated from the magenta-dominated petals in the top image, while they appear integrated in the bottom image as in the original photograph. Smaller details of color fragmentation and extrusion are visible in the top pair of images. What is evident only when viewing the linked *WildFlowers* video is that these color-oriented curtain fragments are moving independently from each other in the video artwork. They are flying apart in the process of revealing the original image.

The next pair of images emphasizes the fact that the curtain of brickwork, mortar, and grates are scaled up from their original sizes (as are the flower curtains in the preceding pair), thereby scaling the video motion. The curtain flickers during video viewing because draw() fragments and reorders the columns of the curtain, defragmenting as it unveils the original image.

The final pair of images shows color partitioning, fragmentation, and extrusion of a pair of doors and surrounding windows unveiled in the second image. Both of the latter pairs appear in the *MovingArchitecture* video. PhotoMontage works especially well with high-contrast color images such as those in the *WildFlowers* video because of the color partitioning. It also works well with architectural photographic sequences such as those of *MovingArchitecture* because the rotation, fragmentation, extrusion, and unveiling of angular perspective lines in architectural photos highlights such boundary-inducing lines.

Watching these videos is very much like watching the unveiling of a scene on a stage as the curtains draw back. The viewer conjectures what scene will actually appear as the photographed image appears. Visual revelation is a key aspect of the PhotoMontage sketch. The author is also experimenting with applying it to live video input from a camera moving about a room and photographing its contents and subjects. The results of live image manipulation are very promising for future work.



---

2 https://drive.google.com/open?id=1o8UqtmIf2IZHFhzyg4mE5hCxTthi5ug7 and https://drive.google.com/open?id=1658WdCHdDNFIRuIftDbgsceOAamhfSbs

5. Conclusions and future work

Custom pixel-level manipulation of generated and photographic images has provided some very fruitful sketches in which to create interactive, animated, computer-generated video art. Much of the interactive work involves exploring visual spaces opened up by these techniques in searches for genuine art. Generate-and-test is a time-honored technique in computing that applies to this type of computer application.

The author has submitted still photographs captured from ShapePaintEcho animations to a state-wide Pennsylvania art competition in 2018. Also, one of the author's colleagues in Kutztown University's Art and Art History Department has invited the author to collaborate on artwork, based on a demonstration video of PhotoMontage, during a sabbatical leave in spring 2019. The author has joined the planning committee for Kutztown University's Applied Digital Arts major program in spring 2018. These developments are very exciting for a computer scientist in the later stages of a long career.

The author has begun exploring use of 3D graphical techniques in conjunction with the 2D techniques of ShapePaintEcho. The canvas itself becomes a curtain in the Z-plane of 0 – the virtual flat surface of the display – with animated shapes emerging in front of this virtual 2D canvas, and shapes disappearing behind it. Combining some of the techniques of PhotoMontage with ShapePaintEcho is also an area for future work. ShapePaintEcho can generate images for PhotoMontage to color-fragment, extrude, and unveil.

Finally, recursive use of morphed, scaled-down canvas PImages as graphical objects to display as multiple mobile objects in a ShapePaintEcho or PhotoMontage context is a very promising area for exploration.

The author's media software tools always involve human interaction as a key dimension. With adequately expressive controls, good human performers can always come up with inspired ideas during performance that might never occur while coding a program. It is very important to make these *visual instruments* so expressive that a good performer can surprise the coder during performance.

The author ends by noting that, after over a year of work on these sketches, the animations still often surprise me. There are times when I cannot figure out how my code is doing things that are so visually alive.

References

[1] Paul, Christiane, *Digital Art*, Thames & Hudson (publisher) *World of Art* series, 2003.

[2] Clayson, James, *Visual Modeling with Logo: A Structured Approach to Seeing* (Exploring With Logo), MIT Press, 1988.

[3] Harvey, Brian, *Computer Science Logo Style*, Second Edition, Volume 3: *Beyond Programming*, MIT Press, 1997.

[4] Papert, Seymour, *Mindstorms: Children, Computers, and Powerful Ideas*, Basic Books, 1982.

[5] Abelson, Harold and Andrea diSessa, *Turtle Geometry: The Computer as a Medium for Exploring*, MIT Press, 1981.

[6] Reas, Casey and Ben Fry, *Processing: A Programming Handbook for Visual Designers and Artists*, Second Edition, MIT Press, 2014.

[7] Shiffman, Daniel, *Learning Processing, A Beginner's Guide to Programming Images, Animation, and Interaction*, Second Edition, Morgan Kaufmann, 2015.

[8] Processing home page, https://processing.org/, link tested in January 2018.

[9] *Shaders*, https://processing.org/tutorials/pshader/, link tested January 2018.

[10] *Core Language (GLSL),* OpenGL, https://www.khronos.org/opengl/wiki/Core_Language_(GLSL) , link tested January 2018.

[11] *The Digital Art Movement*, Modern Art Insight, http://www.theartstory.org/movement-digital-art.htm, link tested January 2018.

[12] Weinmann, Elaine and Peter Lourekas, *Photoshop CC: Visual QuickStart Guide*, Peachpit Press, 2015.

[13] Weinmann, Elaine and Peter Lourekas, *Illustrator CC: Visual QuickStart Guide*, Peachpit Press, 2014.

[14] Fowler, Martin, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third Edition, Addison-Wesley, 2003.

[15] IDL interpolation methods, http://northstar-www.dartmouth.edu/doc/idl/html_6.2/Interpolation_Methods.html , link tested January 2018.

[16] Blofeld, John (translator), *I Ching, The Book of Change*, E.P. Dutton & Co., 1968.

[17] Documentation for Processing's *filter* function, https://processing.org/reference/filter_.html, link tested January 2018.