Overview

One of the most common complaints from users of programs like Adobe Photoshop for image manipulation, is that the vast number of largely irrelevant features hinder their ability to quickly grasp and use the application efficiently. My application, The Manipulator, includes only features relevant to image manipulation, from basic features like levels and black and white adjustment to more complex and interesting ones like polarized or optical illusion creators. Incorrect inputs.

To build my application, I used a combination of Kivy for the interface, and Pillow (a branch of PIL) for the image processing. Kivy handles the UI setup, all changes, and displaying images. Using the python Kivy API, I was able to create a more responsive interface that dynamically resizes as the user wishes. The image itself can be zoomed and panned using the mouse or multitouch, allowing easier inspection of detail. Buttons change color when clicked and revert back when released. Toggle buttons remain the changed color until either clicked again or another toggle button in the same group is clicked. This notifies the user of the current state. Wherever appropriate, I used color pickers and sliders rather that text inputs for more intuitive control. While there is a “tools” option on the toolbar, all of those functionalities also have keyboard shortcuts. The expectation is that as the user becomes more familiar with the interface, keyboard shortcuts are quicker and more fluid to use, though buttons remain for added robustness.

Important Kivy Terms

Canvas: Contains instructions for graphics and/or drawing

Widget: The fundamental “things” in Kivy are called widgets. Each contains a canvas that allows a graphical representation, but the widget itself is a set of properties or instructions. Buttons, labels, images, and more are all represented as widgets

Layout: Think of layouts as containers. The container resizes the widgets or containers it contains to fit. There are several types of containers all with different properties and default inheritance patterns. One of the two main types I used was a GridLayout. It consisted of as many rows and columns as I determined. When a new widget is added to the GridLayout, it automatically resizes all of them to fit its size. This could be overridden as necessary. The other key layout was the FlexLayout. FlexLayout contains no preset guidelines about positioning, instead the user must determine the position, size, relativity of its children manually.

Bugs & Irregularities

Many of the code design choices that seem less intuitive are a result of working around quirks, bugs, irregularities, or incompatibility within Kivy. The most obvious are described in the following paragraphs.

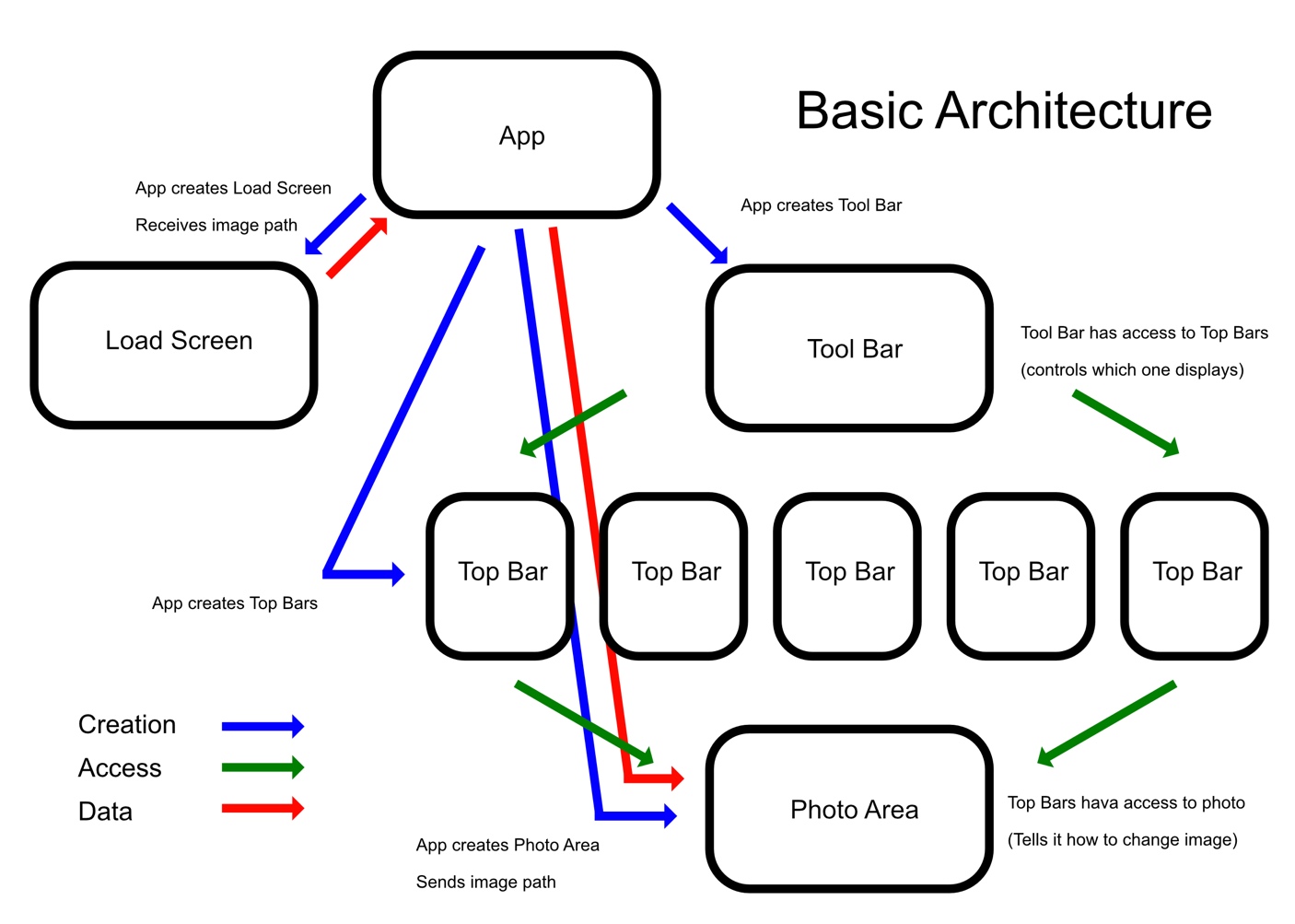
Kivy only supports displaying images from a texture (usually created from bytes), jpg or png file, or downloaded from the web. As a result, I was unable to display PIL images (created after image manipulation directly. The work around was to modify and use a portion of an experimental module of Kivy’s. The coverage revealed that this function had not been properly tested so I tested and handled errors manually. Finally, to change and display images, every time a button is clicked that corresponds to an image change, the saved “pilImage” is called and manipulated. Then the PIL image is copied and converted to pure data and back into a Kivy image data. Then the image data is converted to a Kivy texture, which can be displayed. The old Kivy image is removed and the new one displayed.

When calculating the mouse position of within the Kivy module, the built in method returns half the correct value. For all other methods relating to window and widget positioning, the values are correct. Hence, whenever Window.mouse\_pos is called, the values are manually doubled to correct for this possible bug.

Another strange issue was that after setting the initial position and size values for the Kivy image in the Scatter, position would revert to 0,0 and size to 100,100, the default new widget values. To handle this, I declared the position and size values on creation, and then once again right after by directly accessing these properties and modifying them.

Another problem was when trying to map pixel locations, after an undo, the size of the Kivy image would revert to its default (while remaining correctly displayed on the screen). Therefore, the pixel values were off for the next set of calculations. To counter this, I stored the position and size values of each correct call. Then, before each calculation, if the values had reverted to default, I changed them to the last ones. Otherwise, I updated the saved values to the current ones.

Overall Architecture



The diagram above shows the basic code architecture of my program. I created it this way to allow everything to access only what it needs (anything that creates something else also has access to it, every child also has access to its parent).

At the base of the program is the App, which is a subclass of Kivy’s app and how I start the mainloop of the program. Everything is controlled in the app, but data is never manipulated. Instead, it serves as a facilitator of sorts, sending data between parts of the program that can’t access on another. The main piece of data in the entire application is the image path itself, which is retrieved from the load screen and sent to the photo area. Later on, if a user wants to open a new image, the top bar containing the button calls on the load area. This time, the image path is sent to the top bar and to the photo area.

The tool bar is the second major controller of the application. It determines the current state of the program (by determining which of its buttons are selected) and displays the corresponding top bar. The top bars all contain buttons that trigger methods in photoarea. Those methods update the photo with the user selected changes.

Relative Pixel Location

Unlike most uses of tkinter, displaying and positioning of my widgets are not fixed, but instead are relative and inherit position and sizing from parent layouts. This means that the pixel locations of all objects must be calculated each time they’re needed from the cascading hierarchy of window to layout(s) to widget. To determine if the user has clicked the photo (if in a clickable mode like crop or selective B/W) I first calculate the current size of the window, then check if the click is in the Scatter (parent layout for photo). Since the photo is centered in the scatter, I use that as an anchor point to determine if the click is within the photo bounds. After that, I compute the x location of the current click over the width of the photo and likewise for y and height. This tells me the relative location of the click within the photo. Using that, I map to the nearest pixel within the photo to determine the user selected location. This is done because the pixel display of the photo isn’t the same as the pixels within the photo, since the photo is initially scaled to fit the screen, and later can be zoomed in and out by the user. This changes the 1 to 1 pixel matching to something that can vary drastically from instance to instance.

User Interface Design

The goal of my UI design was to keep things simple and easy to find. The inputs are all tailored toward what’s particular intuitive for that tool, and filters are grouped in an intuitive manner. For example, colors aren’t inputted through RGB, sliders and a clicakable color picer are provided. Keyboard shortcuts are present but not necessary to use the application. Instead of using a built in file dialog box, I wrote my own in the style of Kivy for better matching. All popups, sliders, and widgets are in the same style. Colors are kept gray for better color perception, with occasional blue accents.