“OCam-l-Sketch” Design Document

Kevin Lin (kl738), Jack Thompson (jwt84), and Emma Cohn (emc284)

I. System Description

Summary:
This system will be a functional programming implementation of an etch-a-sketch application with a few additional features. These features include a GUI for size and color control of the cursor and the ability to load and save the etch-a-sketch drawing.

Key Features:

- Drawing image using arrow keys
- Erasing screen
- Color picker
- Change line width
- Load / Save files
- Export to JPEG

Narrative Description:
Etch-a-sketch is a popular children’s toy used to draw, or sketch, images using two knobs. These knobs independently control the vertical and horizontal directions of a continuous line. In our implementation of this device, a user will control these directions using the arrow keys in a graphical user interface that will display the drawn image.

It will have arrow controls to move the cursor, space bar to erase, and controls to change size and color of the cursor. It will also support loading and saving of etch-a-sketch files, and exporting the loaded file to a jpeg.

We intend to use the Model-View-Controller design for our system. The model will involve the state of the drawing. The controller will be responsible for taking in data from the model, processing keyboard and mouse inputs from the user, and updating the model. The viewer implements the graphical user interface that takes the model and outputs the pixels on the screen.

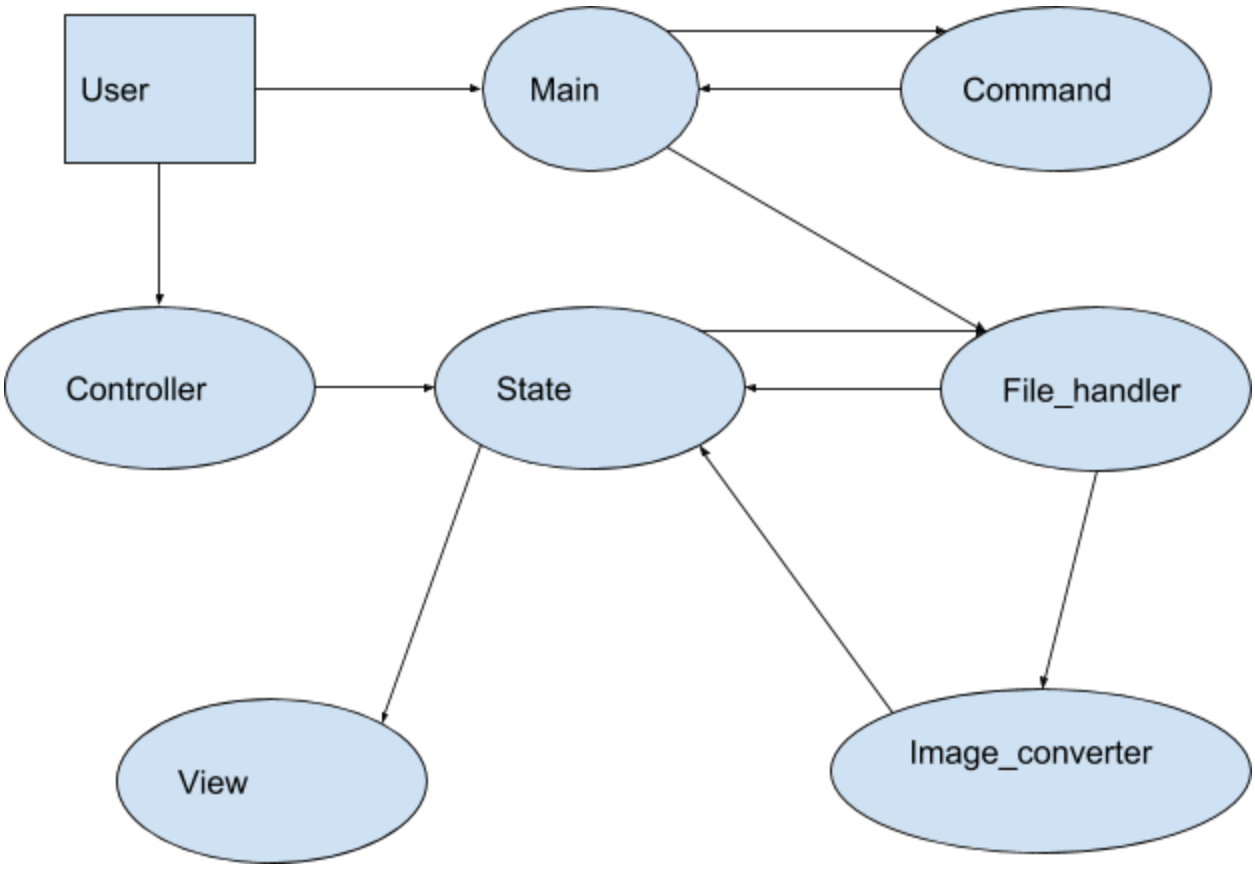
II. System Design

Necessary Modules:

- Main - Emma
    - This module will launch the GUI and create a new image state or import the image state of a previous drawing. In the main module, we will use all of the user commands from the Command module that can be inputted to run, including

open file, new file, save file, quit, and performs the action corresponding to the command. This will have an instance of the file_handler module to do these.

- Command
    - This module specifies the possible commands to be run by the command line, and provides the function that parses the string inputted by the user into the corresponding command.
- Controller
    - The controller will log and perform all the proper actions to the application from the keyboard/mouse input. This includes supporting any color picking that the user will do from the user interface. It will support the picking from a pre-set list of colors and their corresponding hex codes. This will also take in the controls for cursor movement and changes in cursor thickness / opacity.
- File_handler - Emma
    - This module will include functions for making, loading, and saving a drawing. This will handle any conversion from the image format, json, to a graphical representation in the GUI or from that graphical representation back to a json. This graphical representation data will be put into the State. It also contains a function for initializing a new state, without an input file name.
- State - Emma
    - This module will contain current settings for the pointer, such as color, thickness, length, opacity, and x and y value. It will also contain the current file name for saving later, functions for altering and accessing the data within the model, and data about each line segment in the current drawing. The data for the state data type is split into segments and settings.
- View - Jack
    - This module will handle any updates to the GUI that the user sees. It will include the canvas for the image, the color picker, and any buttons that a user can use to alter the state of the drawing.
- Test
    - This module will contain all the OUnit tests. These will include all the tests written below in the testing plan.
- Image_converter
    - This module will convert an image to a list of lists, with each element of the list being a pixel object. We will utilize the Graphics module's [dump_image] function to convert to a color array array which we can then recurse and translate those to pixel objects.
    - This list of lists will then be converted to a State object through an algorithm to detect edges that can be utilized by the View to display the image in the GUI.



State to json

III. Data

Overview:
We will use a json file to save and load the path of the etch-a-sketch. The data will be in the form of a list. Starting from an arbitrary location on the etch-a-sketch board, one can save the entire drawing in terms of the directions because the lines in an etch-a-sketch are continuous. This heuristic will keep the stored data file very compact because a given etch-a-sketch drawing can be drawn by simply the directions and metadata of the paths.

Example:
Suppose the starting point is the bottom left hand corner at coordinate (0,0). A list of directions [Left, Right, Right, Up, Up, Right, Right, Down, Down] will correspond to a square with corners at (1,0),(1,2),(3,2),(3,0), ignoring the extra data from other line attributes.

Metadata:
In addition to the directions, the program will have capabilities to vary line thickness, length, color, and opacity. These can be stored as metadata in addition to each direction, so one entry in

the list would be for example “data: { Direction: Left, Thickness: 3, Color: “#14f234”, Length: 4, Opacity: .7}”. This will be stored as a record. The list of these directions plus their metadata will constitute sufficient data for redrawing a graphic when a saved file is loaded.

Settings:
In addition, the model module will save information about the current settings of the drawing, so that once a user sets the settings for line thickness, color, and opacity, these will be saved for future lines drawn. The last used settings are also saved in the json file.

File name:
The current file name will also be part of the json file because when we want to save the file, we will need to write to the same file name.

IV. External Dependencies

YoJson:
We will use **YoJson** for saving json etch-a-sketch files as a list of line segments. We will also use it for loading existing json etch-a-sketch files.

Graphics:
We will use the **Graphics** module for drawing line segments with differing line thickness, color, and possibly opacity. This will also be responsible for drawing other UI elements, like a color picker and the frame of the etch-a-sketch. **Graphics** will also be responsible for picking up mouse and keyboard events, such as moving the cursor or changing line thickness with the **read_key** function.

OUnit:
We will use **OUnit** to create our test suite.

V. Testing Plan

We plan on creating regression, black-box, and glass-box tests in order to have a comprehensive test suite. We’ll also use play testing to make sure everything looks right on the surface. We will write tests during implementation, so we can know immediately after writing a function, if it is working properly.

Glass-box:
For each main functionality of the program, we’ll create glass-box tests to see how they interact. This will involve checking every path a program could take, and making sure it acts correctly each time. Every non-helper function in the program should have at least 1 glass-box test to make sure it is working according to specifications.

Black-box:

Black box tests will serve to make sure the functions work correctly when integrated with each other as well as independently. We'll have a few drawing files to use for tests to make sure it works properly. These tests will include loading and saving files several times and seeing if the json changes.

Regression:
Everytime we solve a bug, we'll make a test screening for that same bug.

Play testing:
To play test, we'll try loading our own written files and see if they look as expected. We will also save files and see if they load looking the same again. We plan on using the cursors the way a user would when we add new feature and seeing if they act according to specifications.

VI. Changes

We did not deviate from our plan and specifications during the implementation of the prototype. The only small change that needs to be made is how the save/load commands will be called. The REPL is an infinite loop in the terminal command line, but when the GUI starts, there is another infinite loop within the graphics window, so commands in the REPL cannot be responded to after the GUI starts. Thus we will let the GUI handle the save command instead.

VII. Status

The current status of the project is that the basic etch-a-sketch part of the project itself is almost complete. We have the state, view and controller modules synced up appropriately so that typing in "new" in the command line will open up a graphics window, in which Etch-A-Sketch can be played. The keys WASD can be used for navigation, -/= for changing width, and numbers 1-5 are hardcoded to change color.

VIII. Roadmap

The remaining functionality to be completed largely revolves around the image border detection piece of the program. The saving and loading from json in the file-handler is complete but needs to be linked to the gui.

There are two main algorithms that we will try. The first is a more greedy, naiive one that starts at a seed and calculates which direction would maximize the contrast between the two neighboring corners, and to recursively go as such such that the borders of highest contrast are highlighted. This has the problem that if there are two objects far away, the algorithm will keep tracing over one object and never reach the other. This can be dealt with by either keeping track of points already visited, or having a random deviation from the correct choice.

The second algorithm assumes that the we have the graph network of all the nodes(pixels) and the edges between them represent the contrast between neighboring pixels by color. We first have a function that finds the objects, then have another function that traces the objects and connects the traces.

Ultimately, both of these strategies would allow a user to import an image (JPEG, PNG, etc.) and our system would "sketch" a line drawing in the GUI while also creating a new state object for the image and the sketched line segments. Whether our system will work on any photograph or only black-and-white line drawings is yet to be concluded.

Because implementing the basic functionality of the system required less effort than expected, we are currently exploring further extensions such as a game of snake, different brushes (i.e. shapes or patterns that will follow the segment), and an undo stack.

IX. Testing

The testing for the prototype was mostly done in the controller.ml and state.ml files. The main.ml and view.ml were tested by looking at the result of opening the GUI, as there is no easy way of unit testing the view is displayed correctly other than viewing it directly.

X. Screenshot