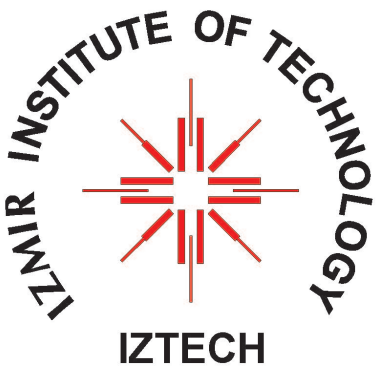


Dynamic Handwritten Digits Dataset: Generation and Analysis

Data Collection System Documentation v2.0b

Ronny Majani

Bachelor Thesis
Dynamic Handwritten Digits Dataset: Generation and Analysis

May 17, 2018

Contents

Summary

This document describes the Data Collection system of this Bachelor thesis project.
We start by describing the hardware we are currently using, and then go into detail about the software itself as it comprises most of the system. Finally we discuss our plan for actually going out and collecting the data.

The first step of this bachelor thesis is to generate the data we want to study. To generate this data we have developed a system that we refer to as our "Data Collection System."

In this case our system consists of both the mechanism by which the data is collected and the actual activities that are performed in order to collect this data (setting up a stand, advertising, giving out free cookies in exchange for people to generate data for us, etc.).

Generally the mechanism used by the system involves a drawing tablet connected to a computer running software that we developed which will read the signals that our drawing tablet is generating in response to a user's interaction with it (handwriting digits on it) and save the appropriate signals on our computer.

Before we dive in, we should first clarify a few concepts.
First, we use the word *System* to refer not only to the tools used in collecting the data, but also the way we store the data, the data format, and the activities by which we use these tools to collect the data. These activities can range from advertising, to setting up a stand and asking people to volunteer and generate data for us.

Second of all, our choice of hardware is almost independent from the software development, as the software was designed to be flexible and can be configured easily to work with different drawing tablets so long as they use the same protocol, and if they don't, we only need to rewrite the driver itself thanks to the modularity of the code.

1 Environment

1.1 Hardware

We utilize **Wacom**®[5] drawing tablets as our input devices.

1.1.1 Clarification

Sometimes we tend to use some terms interchangeably such as drawing tablet and graphics tablet, when referring to our device. For the sake of clarity we list some terms we have used that might be ambiguous:

- **Wacom:** Even though we only used a Wacom tablet, we use the terms drawing tablet more often to keep the concepts more generic and applicable to other brands and models we might use in the future
- **Drawing Pad:** The active are on the drawing/graphics tablet
- **Device:** we use this to refer to the drawing tablet itself
- **Pen:** we use this to refer to the stylus of the drawing tablet

1.1.2 Specifications

Wacom Drawing Tablet

- **Brand:** Wacom®
- **Tablet Model:** Volito2 (4×5) CTF-420/G tablet.
- **Resolution:** 5105×3713 $(w \times h)$
- **X Values Range:** $[0, 5104]$
- **Y Values Range:** $[0, 3712]$
- **Stylus Model:** FP-410-0G
- **Stylus Pressure Resolution:** 512
- **Stylus Pressure Values Range:** $[0, 511]$
- **Power Specification:** $DC\ 5V\ 0.14A$
- **Connection Type:** USB

1.2 Software

Our software was developed using the Python programming language (v2.7) in a Linux environment (Ubuntu 17.04 to be specific). The important thing to note is that although coding in Python offers great portability of our code (we can theoretically run this on any machine that implements the full Python environment), our code requires Linux to work as we are using the Linux kernel's **evdev** driver.

The GUI itself was developed using the **Qt4** framework and the canvas that shows displays a live preview of what the user is writing was made using **pygame**. Basically we do the graphic processing in pygame, and copy over the graphics buffer to a Qt canvas widget embedded in the *drawing window*.

We tried our best to keep the code as modular as possible so it shouldn't be difficult to port the code over to a non Linux system so long as the rest of the dependencies (python 2.7, pygame, qt4, and their respective python libraries) are available. You should only need to rewrite the *"driver"* class that interfaces with the drawing tablet.

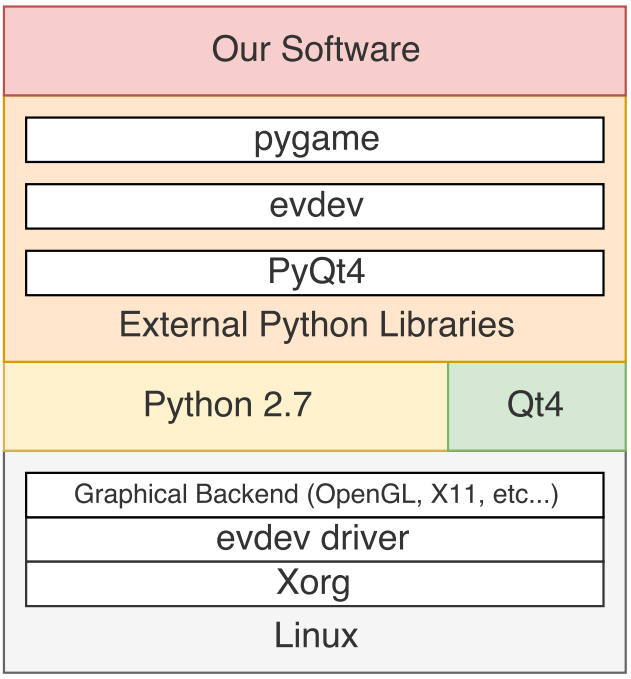


Figure 1: Software Environment Overview

1.3 Setup

The setup is simple; One Wacom graphics tablet connected to (via USB) an Ubuntu powered computer running our software.

The required software dependencies can be installed straight from your system's package manager (if running Ubuntu).

A configuration file is available and should be modified with the desired parameters. This file is used by our program to determine the settings it should use during execution (the configuration and settings are documented later in this document).

The software should work out of the box, simply run and it should load your specified configuration. Log messages are displayed in the console, and they are the only true indicator of any fault or error.

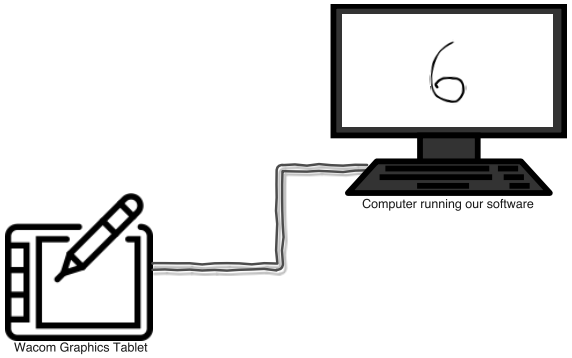


Figure 2: Visualization of our System

2 Program

2.1 Functional Overview

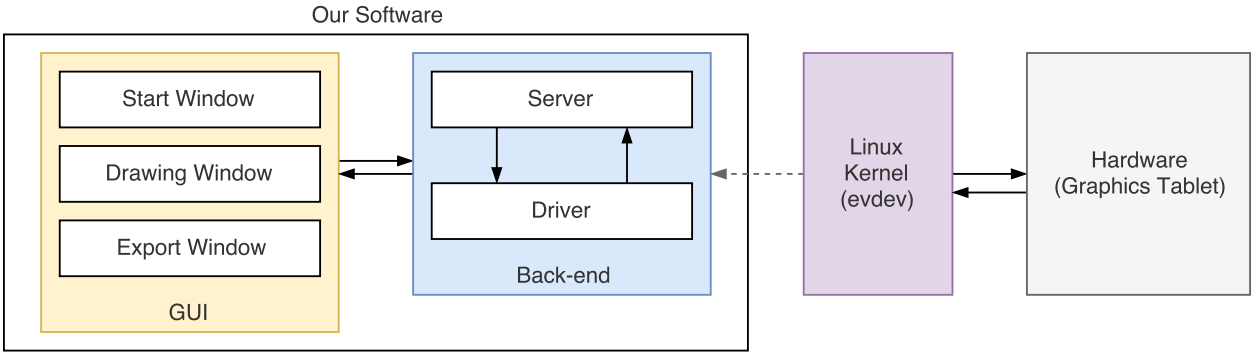


Figure 3: Data Collection Program Functional Overview

We designed our program to separate as much as we could the graphical front-end from the back-end. The GUI is responsible for providing the user interface, initiating the program's phases, and providing a live preview of the currently read data (in the case of the drawing phase).
The back-end manages our data structures and interfacing with the drawing tablet (via the evdev driver). The *Server* class of the back-end is also responsible for managing and updating the program state.

2.2 Code Structure

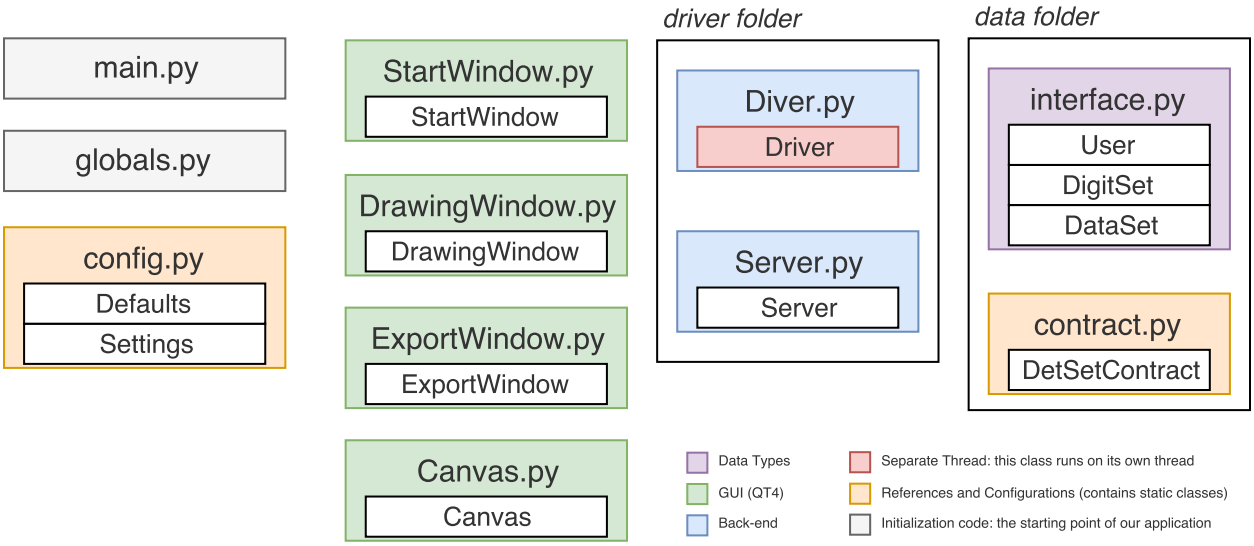


Figure 4: Structure of Project's Code

As shown in the figure above, our software is divided into a set of files and classes. The entry point to our program is **main.py** which simply sets up our environment, loads any user configurations and starts our GUI Qt4 Front-end with StartWindow.

globals.py contains the variables that are shared among all the files, such as the instance of our **Server** class *(we only have one instance of the Server class in our program)*

The **Driver** class works as an interface between our program and the Wacom driver *(via the evdev driver)* and runs in a separate thread parallel to our program.

We have some "static" classes that we use to define things like configurations, and constants.

2.3 Usage

The program on an abstract level works as follows:

1. One or more users go through a drawing process
   (a) user fills in form
   (b) user clicks start
   (c) user writes the digits 0 through 9 multiple times each *(number of repetitions per digit is specified in the configuration file)*
   (d) program creates a digitset object from the data generate by the user
   (e) program appends this digitset to a dataset object that is kept in memory throughout the lifetime of the program
   (f) program exports this digitset into a separate JSON file
   (g) program closes drawing window
2. Admin exports the dataset object kept in memory which contains all the digitsets that have been created throughout the lifetime of the program *(since the program was opened)*.

2.3.1 Configuration

There are four ways to configure the software.

- **globals.py**
  - **CONFIG_FILE:** The path where to load/save your configuration file.
  - **device_name:** The path to the event device file representing the connected Wacom Tablet. Usually on Ubuntu systems this will be /**dev**/**input**/**event*XX*** where *XX* is the number corresponding to the device you connected. You can find this path by running one of the following commands:

    ```
    $ evemu-describe
    $ evemu-record
    ```

    On the computer we are working from, the default path is */dev/input/event12*.
    An alternative is to specify your device by *ID* instead of event number. To list all the event devices by ID on your device just list all the files in the /**dev**/**input**/**by-id**/ directory.

    ```
    $ ls /dev/input/by-id
    ```

    Note: on some systems you may not be able to access an event device without the right permissions. A simple workaround is to take grant *everyone* **READ** access to the event device every time you connect it to your computer. Example:

    ```
    $ sudo chmod +r /dev/input/event12
    ```

    *(replace */dev/input/event12* with the correct device path on your system)

- **Setting the event device ID via command line arguments**

  you can specify the ID of the event device to use by passing it as an argument when running the program from the terminal. For example, to use a Wacom device connected to our computer under the handle /dev/input/device12 we just need to run our program as follows

  ```
  $ main.py 12
  ```

- **config.py**
  You can edit the **Defaults** class but I advise against it. Instead, if you don't have a configuration file yet, simply open the program then close it. One will be created for you in the directory specified by the **CONFIG_FILE** variable in **globals.py**.

- **Configuration File:** A JSON file containing the following options
  - **EXPORT_DIRECTORY_PATH:** The directory where generated data will be saved
  - **DATASET_FILENAME:** The name of the file to save any exported dataset to *(make sure that the filename ends with the **.json** extension)*
  - **SAMPLE_COUNT_PER_DIGIT:** How many samples to take for each of the 10 digits a User will write.
  - **REFRESH_INTERVAL:** The time interval in milliseconds to update the Drawing Window with. The smaller the number the smoother and more responsive the Drawing Window will feel.
  - **JSON_INDENT_LEVEL:** The level of indent to apply when saving any JSON file (digitsets and datasets) as defined by python's *json.dump()* documentation[4]
  - **JSON_SEPARATORS:** A 2 element tuple of the separators to use when saving json files (digitsets and datasets) as defined by python's *json.dump()* documentation. The first element in the tuple is the comma separator, the second element is the key/value separator. By default these are set to (",", ":")
  - **PEN_PRESSURE_MIN_THRESHOLD:** The minimum pressure threshold that should be read by our software to consider the pen *"touching"* the drawing tablet. The reason we allow this adjustment is to avoid small noise that can occur as a result of shaking the pen or a reaction force when writing or removing the pen from the drawing pad very quickly.

  If one does not exist, a new one will be created at the start preloaded with the program defaults. The configuration file is loaded at the launch of the program before the GUI is even started. If you make any changes while the program is running you must restart the program for any changes to take affect.

2.4 Front-end

2.4.1 Start Window

This is the first window in the application. From this window you can either:

- start collecting data by first filling out the form with the new user's information and then clicking the start button which will open the drawing window and hide this window until the data collection (drawing procedure) has completed
- or export the data that has been collected so far into a dataset file by clicking the menu button in the window's frame and selecting on the "export" option

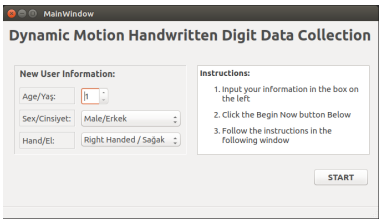


Figure 5: Start Window Screenshot

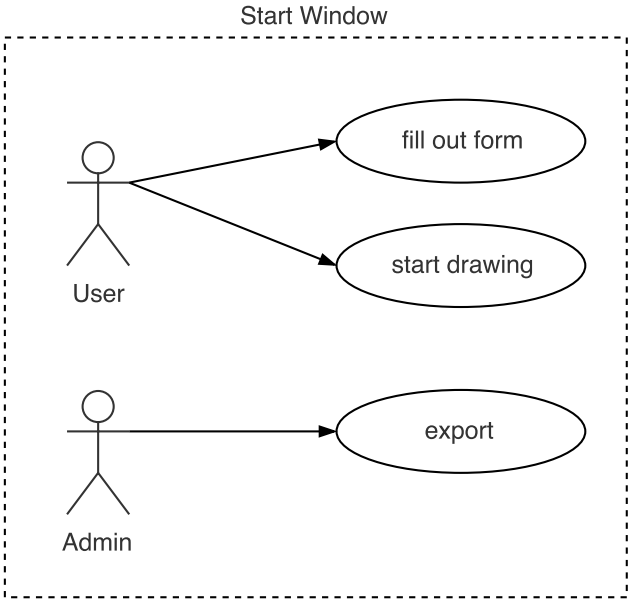


Figure 6: Start Window Use Case Diagram

2.4.2 DrawingWindow

This window is where the user performs the drawing process.
On creation, the window registers itself with the Server object (defined in **global.py**) and register's the UI buttons with the class's functions. The class also creates a Qt Timer that will repaint the UI at a fixed interval as defined in the configuration. A DrawingWindow instance also contains an instance of a Canvas object.

When this object's repaint object is called, it will first call the draw function on its Canvas object passing in the driver's temporary buffer as an argument, then the object will update its indicators with the last data our Driver thread has received from the Wacom tablet. As for the other indicators related to count and phase, the Server class handles updating them with the correct value.

Using this window, a user will be asked to write down (using the drawing tablet) the digit shown on the screen. This window also contains a canvas to show in real time what the user has written so far. The canvas basically draws the points received from the tablet and connects them with a line of varying thickness. The line thickness is proportional to the stylus pressure value of the next point that is being drawn.

The user is expected to draw each of the 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) a specified number of times (as set in the program configuration). These samples constitute a "Digit Set." After all digits have been written, the program will save the digit set to a file in the configured export directory and append this digit set to a data set structure in memory. The naming convention for these saved digit files is:
*HH.MM_dd.mm.yy*_digitset.json
*Where *HH* is the current time in hours, *MM* is the current time in Minutes, *dd* is the current date, *mm* is the current month, and *yy* is the current year
Unless explicitly told to, the program will not export its in-memory dataset to a file.

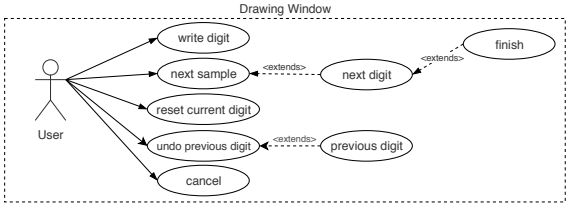


Figure 7: Drawing Window Use Case Diagram

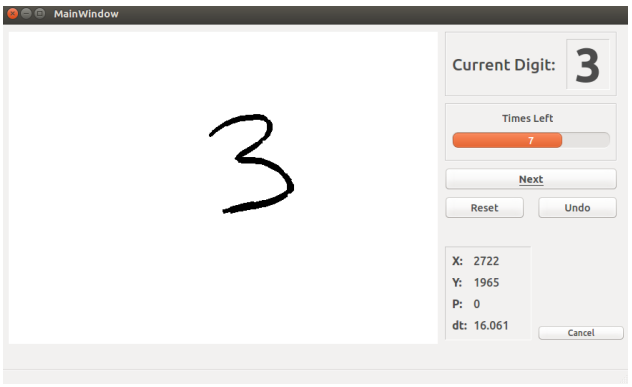


Figure 8: Screenshot of Drawing Window

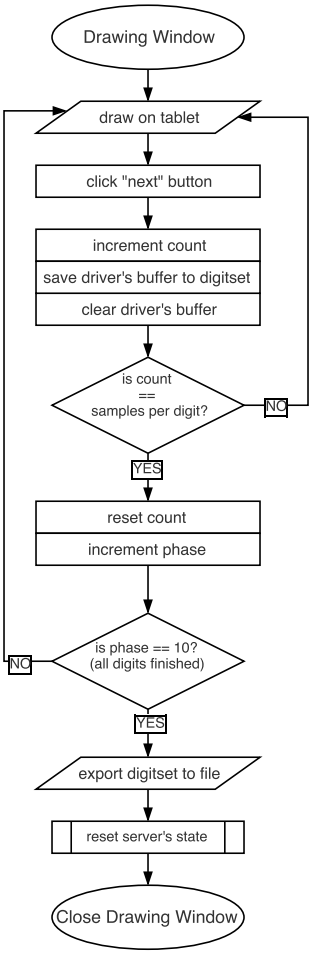


Figure 9: Flowchart of Drawing process

2.4.3 Canvas

The canvas class is a Qt widget that wraps inside it a pygame Surface object that provides us with a graphical buffer in the background for us to draw in.

The object's draw method take in a buffer (list) of data points (X, Y, P) and draws each point (scaled to size) on a canvas of fixed dimensions, and connects these dots with lines. The thickness of the dots and lines is proportional to the P value of each point.

When the Canvas object's draw repaint event is called, it will copy its raw graphics buffer and paint its corresponding QtCanvas widget in the UI with this buffer. The reason we do this is because the pygame buffer it contains runs in the background, so all the drawing functionality is done via pygame in the background and then the final result is copied and drawn in the Qt UI manually. This allows more easier control with drawing since we get to use pygame's graphics capabilities.

2.4.4 Exporting Window

The export window's functionality is quite simple, when you open the window, the first thing that the program does is check whether a datset file already exists *(remember that the dataset filename is specified in the configuration)*.

If a file already exists, then the pre-existing dataset is loaded into memory and merged with the program's currently loaded dataset. Then the export window's input fields are filled with the loaded datasets metadata.

If no file was found then the form fields will all default to being empty.

Next the admin edits/fills in the desired metadata and then clicks save, which instructs the program to either create a new dataset file or overwrite a previously existing file with the dataset object in memory *(which consists of the data collected during the lifetime of the program merged with the data loaded from a pre-existing dataset)*. After the dataset was successfully exported, the dataset object in memory is deleted and a new empty one is created.

The program checks whether a dataset file already exists, and if it does it loads its metadata into the form on screen. The system admin is expected to fill out the metadata appropriately and then click on the save button, which will either create a new dataset JSON file and save our in-memory dataset in it, or if a datset file already exists, it will load that dataset into memory, merge the newly loaded data-set with the in-memory dataset and then write back the final dataset into the file overwriting the old version with the newly merged one.

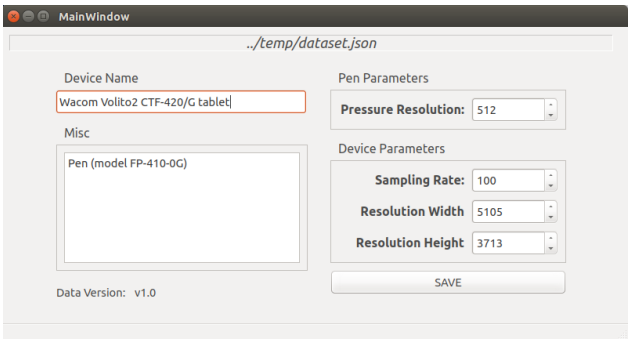


Figure 10: Screenshot of Export Window

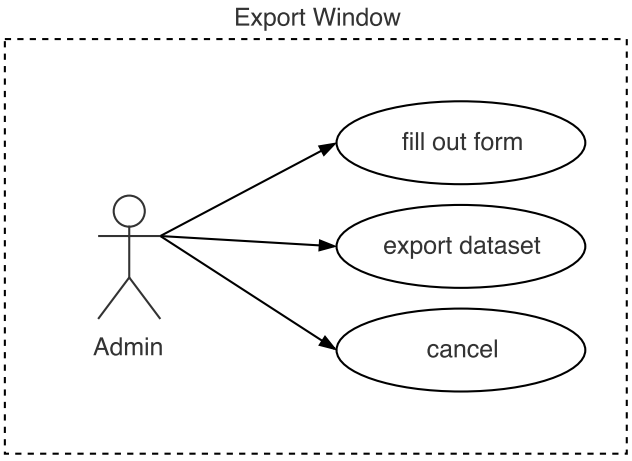
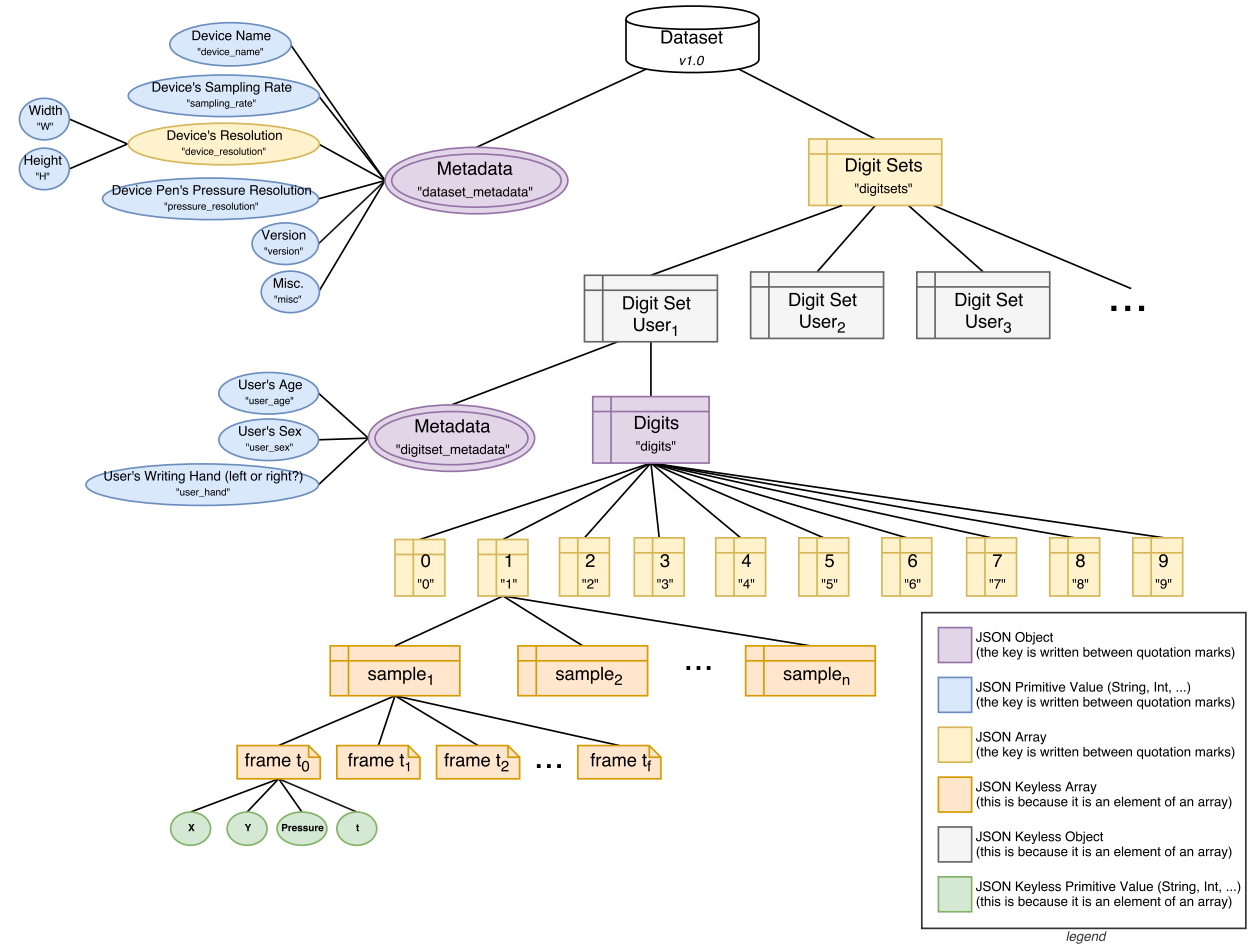


Figure 11: Export Window Use Case Diagram

2.5 Data

Version 1.0



Device Name ***(String)***	Name of the device used
Device's Sampling Rate ***(Int)***	The sampling rate of the device measured in "Points per Second" *Valid values: Natural Numbers > 0*
Device Pen's Pressure Resolution ***(Int)***	How many distinct and discrete pressure levels the pen can measure *Valid values: Natural Numbers > 0*
Version. ***(String)***	The data version of this dataset *Format v**X.Y***
Misc. ***(String)***	Any other optional note (Not to be used by the software)

W ***(Int)***	The virtual width of the tablet's resolution. *Valid values: integers > 0*
H ***(Int)***	The virtual height of the tablet's resolution. *Valid values: integers > 0*

User's Age ***(Int)***	The age of the person who generated this set of samples *Valid values: integers > 0*
User's Sex ***(String)***	The sex of the person who generated this set of samples *Valid values: "male", "female"*
User's Hand ***(String)***	The hand that was used to generate this set of samples *Valid values: "left", "right"*

X ***(Int)***	The absolute X coordinate where the pen's tip is touching the tablet *Valid values: integers in the range [0, W)*
Y ***(Int)***	The absolute Y coordinate where the pen's tip is touching the tablet *Valid values: integers in the range [0, H)*
Pressure ***(Int)***	The pressure with which the pen is being pressed onto the tablet *Valid values: integers in the range [0, "pressure_resolution")*
t ***(Float32)***	The elapsed time in ms since the first sample of this digit *Valid values: 32 bit floating point numbers >0*

Description of the values we are storing in our dataset

Figure 12: Data v1.0 Hierarchy

2.5.1 File Format

Our data is stored as JSON[3] files.
Every Digitset is saved as a separate file.

And when requested, the separate digitset files are compiled into one Dataset JSON file.
The layout and heirarchy of the data and digit sets is shown in the previous section

2.5.2 Defined Classes

We defined a few classes to make interfacing and managing our data easier and more abstract, allowing for easier changes to the dataset format in the future.

- **User:** *defined in interface.py*
  This class represents a single user. It contains the meta-data about the user that is needed by each digitset, and the class defines a method for converting this metadata into a python dictionary in JSON format.
- **DigitSet:** *defined in interface.py*
  Every user who uses our system will produce one **Digitset.**
  The structure of a digitset is shown in the figure above. Our program defines a DigitSet class to contain this data and perform operations on it.
  Each digitset contains one **User** and a list of data that the user has generated so far.
  The **Server** class appends new digit data to this class's data list via it's add_digit() method.
  The class also defines a method for converting itself into a JSON compatible python dictionary.
- **DataSet:** *defined in interface.py*
  This class contains all the digitsets that have been created during the lifetime of the program (since the current instance of the program has been created). This class also defines a few methods to interface with the contained data, such as converting itself into a JSON compatible python dictionary, merging other datasets with this one, importing a dataset from a file, etc.
- **DataSetContract:** *defined in contract.py*
  This class defines the contract for our dataset.
  A contract class is a concept I learned from Android development. In short, it is a class that defines the constants that help applications interface with a database and it's column names, URIs, and other relevant fields that are needed by the rest of the API when performing operations on the relevant database.
  In our case we defined a contract class that defines all the constants relevant to our dataset's structure and format. This abstraction allows us to easily modify, update, and extend our dataset format in the future without having to search through the entire program for hard-coded strings and dataset operations since the entire program uses these constant variables.

2.6 Program Flow

2.7 Back-end

Here we designed two abstract roles: Server and Driver.

2.7.1 Server

The Server is the backbone of the entire program. It is also where our data is stored, and where operations on that data are initiated.

2.7.2 Driver

The Driver class, as the name suggests, works as the interface between the Server and evdev driver. It runs on a separate thread (using python's threading class) and continuously polls for new events from the Wacom tablet (via the evdev driver) and stores the relevant events (X, Y, Pressure) in a temporary local buffer. The Server class is responsible for saving the data in that buffer and clearing it as needed. The Driver's role is minimal, as the Server class handles the program logic.

Wacom Tablet evdev Events

- **ABS_X:** *New* Stylus X coordinates
- **ABS_Y:** *New* Stylus Y coordinates
- **ABS_P:** *New* Stylus Pressure value
- **SYN_REPORT:** A synchronization event that comes at a (almost) fixed interval.

Note our emphasis on the word ***"New"*** as the Wacom tablet will not inform us if the values have not changed and will only send events when a new value has been read. Hence we always keep track of the last value read and use the Synchronization event as a signal to package the last read X, Y, and Pressure values into a tuple and append them to our buffer.

Note:

I personally didn't find a standard documentation on which events the Wacom tablet produces and it's button mapping with evdev, but one quick and easy way to gain a good intuition for which events correspond to what actions is to use a tool called **evemu-record** that comes as part of the **evemu-tools**[1] package which can be installed via the *apt* package manager in Ubuntu

2.7.3 Timing:

Since **Data v1.0**, the dataset also includes a timing feature for every sample in addition to the X, Y, P features. The evdev driver sends a timestamp with every event, and this program simply extracts thaese values, and measures the elapsed time between *SYN_REPORT* events in milliseconds rounded to 3 decimal points and saves it.

2.7.4 UI Updating

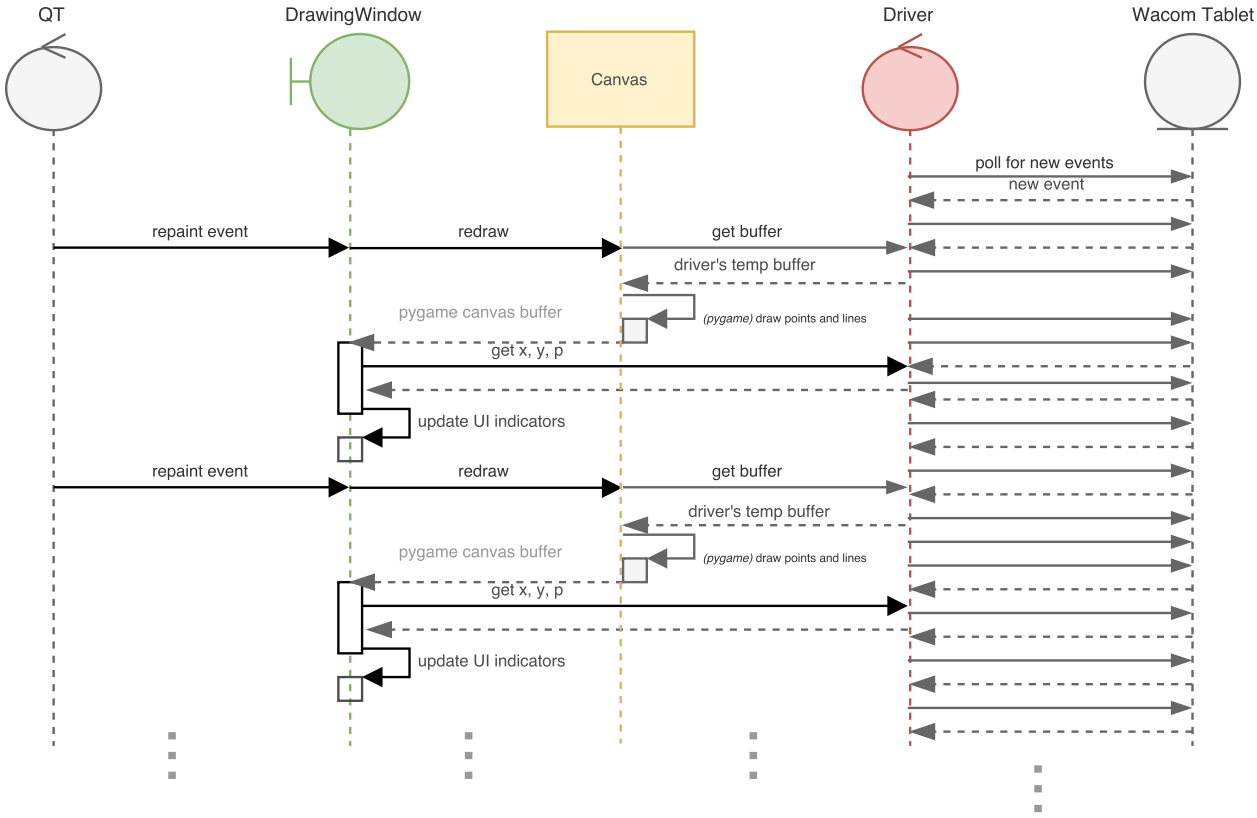


Figure 13: Sequence Diagram showing how the GUI updates itself while the *Driver* class continuously deals with the Wacom tablet. The *Driver* class continuously polls for events from the Wacom tablet and saves relevant data to a temporary buffer. The DrawingWindow sets up a Qt timer that guarantees that the UI will be updated (repaint event) at a fixed interval. Whenever a repaint event comes, the Drawing window will first call it's Canvas instance to update itself, then it will update the X, Y, P indicators with the last received event the Driver thread has received. The Canvas object performs its draw method in a hidden pygame graphics buffer, which it will then copy, convert it into a QtImage (bitmap image) object and paint it's QtCanvas widget with the created QtImage

3 Field Plan

The device and software are only half the work.
Once we have a fully functioning setup, we can finally begin focusing on actual data collection.
Collecting data involves:

- Setting up our system in an easy accessible place where many people pass by and will see our setup.
- Holding some sort of advertising campaign to spread awareness of our project.
- Some sort of incentive to motivate people to volunteer their time and use our system to generate data.

Since this is a first version of our data collection system, we do not intend to collect a huge amount of data yet. Our intention now is to collect a small sample as through a test session, to test the functionality of the system. Then we will use this data in some naive algorithms. From this we will discover any flaws and modifications that are needed in our system and update it accordingly.

3.1 Initial Plan

Set up a stand in the library of my university (Izmir Institute of Technology[2]), and ask people to volunteer. As an incentive, we will give out free home-baked cookies to people who volunteer.

4 Extra Notes

- Your operating system might automatically use the wacom tablet as an alternative mouse, which means when you try to draw on the tablet you also end up moving the mouse cursor around which is an undesired affect. You can easily disable this via the control panel or via the terminal. To disable your wacom tablet from being used as a mouse on your system, type in the following command:

```
$ xinput list
```

You'll see a list of input devices on your device such as:

```
$ xinput list

>> Virtual core pointer                    id=2    [master pointer (3)]
>>   Virtual core XTEST pointer                id=4    [slave  pointer (2)]
>>   ETPS/2 Elantech Touchpad                  id=15   [slave  pointer (2)]
>>   Wacom Volito2 4x5 Pen stylus              id=11   [slave  pointer (2)]
>>   Wacom Volito2 4x5 Pen eraser              id=16   [slave  pointer (2)]
>>   Wacom Volito2 4x5 Pen cursor              id=17   [slave  pointer (2)]
>>   Virtual core keyboard                     id=3    [master keyboard (2)]
>> Virtual core XTEST keyboard             id=5    [slave  keyboard (3)]
>> Asus Wireless Radio Control             id=6    [slave  keyboard (3)]
>> Video Bus                               id=7    [slave  keyboard (3)]
>> Video Bus                               id=8    [slave  keyboard (3)]
>> Power Button                            id=9    [slave  keyboard (3)]
>> Sleep Button                            id=10   [slave  keyboard (3)]
>> Asus WMI hotkeys                        id=13   [slave  keyboard (3)]
>> AT Translated Set 2 keyboard            id=14   [slave  keyboard (3)]
```

Then just look for the ID of the connected Wacom device *cursor*. In this case the id is *17*. Then type the following command to disable it:

```
$ xinput disable 17
```

References

[1] Evemu tools. URL https://freedesktop.org/wiki/Evemu/.

[2] Izmir institute of technology. URL http://www.iztech.edu.tr/.

[3] Javascript object notation. URL http://json.org/.

[4] Python 2.7 json library json.dump() documentation. URL https://docs.python.org/2/library/json.html#json.dump.

[5] Wacom. URL http://www.wacom.com.