Magic: The Gathering Card Reader with Feature Detection

Dakota Madden-Fong
Willamette University
May 2018

github.com/TrifectaIII/MTGCardReader

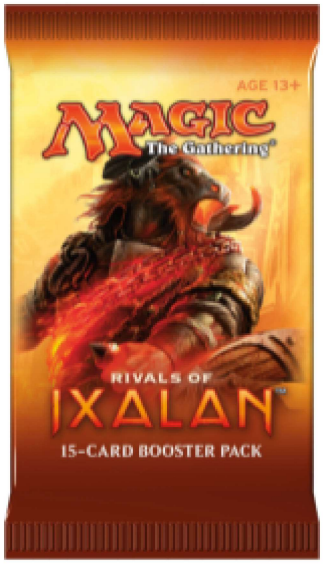
Abstract

This paper details the creation of a computer program which allows an end-user to quickly and easily generate a digital descklist of Magic: The Gathering cards using a webcam and a physical selection of cards. The program utilizes feature detection to identify the webcam image of a card from an online database of card images. The user can then automatically generate decklists without having to type the name of the card at any point. These decklists can be used for cataloging, sorting, collection management, and deck building purposes.

Motivation

An MTG Card

Magic: The Gathering (MTG) is a Trading Card Game (TCG) created by mathematician Richard Garfield, which was first published in 1993 by Wizards of the Coast. To date, more than 17,000 different cards have been printed, with more released every year. To play, a player must decide on a list of 60 cards that they want to play together in a deck, then acquire all 60 of those cards. MTG cards can be obtained through trading with other players, purchasing on the secondary market, or by opening packs (each containing fifteen randomly-selected cards).



An MTG Pack

MTG decks built by players usually follow a theme or rely on synergy between individual cards to win the game, and players often want to share the composition of their deck with others; for advice on card selection, or so others can copy it. This can be accomplished by sending decklists (plaintext lists that contain card amounts and names) to others.

Many players have collections of thousands of cards obtained over years of playing the game. Additionally, the existence of a secondary market for MTG cards means that individual cards have real market value, and some are very expensive. For instance, a copy of the card

‘Black Lotus’ printed in the set *Alpha* once sold for over $27,000 USD. Therefore, players desire tools to help manage and maintain their collections. Not only does this make playing the game easier, but it can also allows a player to keep track of expensive cards and calculate the monetary value of their collection as a whole. Often, these tools use data formats similar or identical to the decklists use for deckbuilding.

These decklists are usually very simple. If your deck had four copies of the card ‘Brainstorm’ and two copies of the card ‘Lightning Bolt’, your decklist would read:

```
4 Brainstorm

2 Lightning Bolt
```

The problem is that the creation of a decklist requires manually typing all the card names on a keyboard. This is not a huge issue if the list is only the sixty cards of a basic deck, but when you consider the thousands of cards that can exist in a player’s collection, the task of typing everything in can be daunting.

My program aims to solve this problem by serving as a quick, easy way to create decklists, requiring only access to a webcam and the physical cards themselves. Users can place a card in the frame of their camera and simply hit a button, and the program will automatically identify that card and allow the user to add that card to a text list.

Resources

This program makes use of a number of resources.

- **Python 3 -** python.org

This project is written in Python, specifically Python 3.6. Python was a natural choice for this project, as it is the programming language that I personally have the most experience with. Additionally, Python is open-source and very popular, which means there are many packages available for almost any task. My familiarity with the language meant that I was able to spend less time wrestling with syntax and instead spend time actually working on the project itself.

- **MTG-JSON -** mtgjson.com

MTG-JSON is a community created JSON library which allows easy access to information on every MTG card in existence. The card data is keyed to multiple parameters, including set. It also contains the 'Multiverse ID' of all the cards, which can be used to generate URLs for card images hosted by the official MTG reference site, Gatherer. This project would likely not be possible without this resource of one very similar to it.

- **PyQt5 -** riverbankcomputing.com/software/pyqt/

PyQt5 is a set of python bindings for the Qt5 GUI Widget framework. It allows access to essentially all the functionality of Qt by

creating python classes for all built-in Qt widget types, and allowing the user to extend those classes. It also implements a slot/signal system that allows different Qt objects to communicate with each other easily.

- **OpenCV** - opencv.org

OpenCV is a library of python computer vision tools. For this project, I specifically used the OpenCV contrib module (pypi.python.org/pypi/opencv-contrib-python), which contains the base package as well as additional functionality. My use of OpenCV in the final version of the project involves the SIFT algorithm and webcam access, but in early stages of the project I also used OpenCV's basic GUI tools for testing.

- **Other external python libraries used include:**
    - URLLib - docs.python.org/3/library/urllib.html
- **Text sources include:**
    - [Lowe1999] Object recognition from local scale-invariant features, <10.1023/B:VISI.0000029664.99615.94>. David Lowe, University of British Columbia, 1999.
    - [Lowe2004] Distinctive Image Features from Scale-Invariant Keypoints, <https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf>. David Lowe, University of British Columbia, 2004.

Strategy for Card Identification

---

*A note: MTG Cards are printed in **sets** of 100-300 cards that are all released at the same time. Each set has its own unique three or four letter setcode. While many cards are brand new every year, some cards are reprints of older cards. Reprints can visually differ from previous versions but are treated as identical in the rules of the game. The relative price of different printings of a card can vary drastically.*

The original plan for identifying a webcam image of a card in the library of all cards called for two basic technologies. First, an Optical Character Recognition (OCR) technology would be used to determine the name of the card in the webcam image. Secondly, a feature detection technology would be used to identify the webcam image within all printings of the named card.

Unfortunately, this approach was flawed. Even before attempting to get an OCR engine running, it became apparent that the visual differences between different printings of a single card are sometimes so small as to be difficult or impossible to detect using feature detection. When a card is reprinted, sometimes the original art, text, and border are preserved. When the

only difference between two printings of a card is the small set symbol that indicates when they were printed, even my best attempts at identification with feature detection were not very robust.

This meant that even if I were to determine what the name of the card was with OCR, I might not be able to determine what printing of the card was in front of the webcam. Instead, I moved to a system where the user manually specifies the set that their card comes from, and the webcam image is identified within all the cards in the specified set. While this approach does require a greater degree of user action, it does not requires the use of OCR technology. Additionally, as the strategy is no longer to compare different printings of a single card but instead altogether different cards that were printed at the same time, the similarity problem is a non-issue.

The trade off I had to make here by requiring more user input in order to make the base functionality of the program more robust was one I was perfectly happy to make. It is unfortunate that I could not realize my vision of a program that would not require any user input other than placing the card in frame of the webcam. However, I would much rather have a program that takes a little more work from the user to operate, than one that does not work well at the core of its functionality.

Feature Detection with SIFT

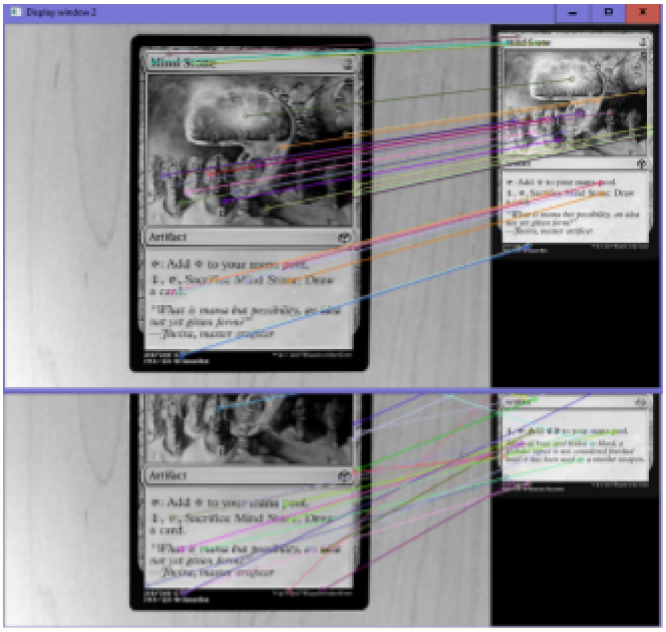
Scale-Invariant Feature Transform (SIFT) is a feature detection algorithm published by David Lowe at the University of British Columbia in 1999 [Lowe1999]. SIFT processes an image and outputs a number of keypoints that describe points of interest in the image. These keypoints are often in high-contrast areas of the image and are described in a manner that is invariant of scale, rotation, illumination, and orientation. Each keypoint consists of a description vector which contains parameters describing the point of interest in question.

The OpenCV-contrib computer vision library has implemented SIFT, and this is the implementation that I used for the project. OpenCV also has implementations of other feature detection algorithms, notably SURF (Speeded-Up Robust Features) and ORB (Oriented FAST and Rotated BRIEF), but for this project SIFT proved to be the easiest to use.

Once an instance of SIFT has been instantiated, I can use it to calculate keypoints for all the relevant images in a given situation: in this project, that means all the card images from the user-selected set, along with the image of a physical card from the webcam. After SIFT computes keypoints for all of those images, I had to determine the best way to use these keypoints in identifying the webcam image as one of the card images.

My basic plan for this identification was relatively simple. When comparing two images, the idea is to match individual keypoints in the first image with keypoints in the second. As each

keypoint is described as a vector containing many parameters, it is possible to calculate the nearest neighbor of a keypoint in terms of euclidean distance. By choosing a keypoint in the webcam image and finding its nearest neighbor from the keypoints of the card image, we can find a candidate for keypoint matching. To prevent very different keypoints from being matched with each other, some test must be implemented to ensure that a match is of sufficient quality.



An early attempt at visualizing keypoint matches

Once keypoints are matched, I can count the total number of matches between every individual card image and the webcam image, and select the library image with the highest number of matches as my primary candidate for the identity of the card in the webcam image.

The difficulty arises when considering the test of keypoint-match quality required by this plan. My first attempt at this test was a mutual-nearest-neighbor approach. In this strategy, I take the candidate keypoint that was found in the card image, and calculate its nearest neighbor in the webcam image. If the original keypoint and the candidate's nearest neighbor are one and the same, then they are mutual nearest neighbors and under mutual-nearest-neighbor, they would be a successful match.

This strategy was serious flawed. First, there was no way to tune it; aside from changing the euclidean distance formula, there was no way to adjust it to try and increase accuracy. Aside from that problem, two keypoints being mutual nearest neighbors was not an effective measure of match quality. It was common for a successful match to register between very dissimilar keypoints because those keypoints happened to be slightly more similar to each other than they were to other poorly-matched keypoints. Essentially, mutual-nearest-neighbor would always skim some matches off the top of a sea of dissimilar keypoints.

This problem was exacerbated because this strategy was biased in favor of images for which SIFT had generated more keypoints. Card images with more detail were regularly selected over the correct images under mutual-nearest-neighbor, because the increased number of keypoints would create more of the low-similarity successful matches. These issues plagued a good portion of the time I was working on this project, and I was never able to solve them in a satisfactory manner. Instead, I had to swap to a new type of test.

My second attempt at a keypoint-match quality test was the K-nearest-neighbors (KNN) ratio test, originally proposed by David Lowe, the creator of SIFT [Lowe2004]. In this strategy, both the first and second nearest neighbors of the webcam image keypoint in the card image are calculated, and the first nearest neighbor must be significantly closer (in terms of euclidean distance) to the original keypoint than the second nearest neighbor is. This ensures that a successful match is only registered if the match is the clear best by a significant margin. Therefore, the quality of successful matches under this strategy is usually much higher.

This approach solves the tuning problem, as the threshold can be adjusted to allow for more or less distance between the first nearest neighbor and the second.

This approach also solves the bias issue. As matches must be significantly better than even the next-best option, we don't run into the 'skim from a sea of bad matches' problem, and increasing the amount of keypoints in an image does nothing to bias the program towards that image.

Below is the code for the KNN ratio test:

Line	Code	What it does
1	**rawmatches** = self.bf.knnMatch(desr,des, k=2)	Find closest 2 neighbors for every keypoint in the webcam image
2	**matches** = []	Sets up list to store successful matches
3	for **m**,**n** in rawmatches:	Loop through each pair of closest 2 neighbors
4	if **m**.distance < 0.75***n**.distance:	Test to ensure that first nearest neighbor is sufficiently closer compared to the second nearest neighbor (in this case, 25% closer)

5	matches.**append**([m])	Add the match to the successful matches list
6	return (**len(matches)**)	Output the total number of matches for the image comparison

While this is only a handful of lines of code, it took a long time to arrive at. The primary problem is that I did not understand how exactly the KNN algorithm works, and therefore could not understand what the KNN ratio test was trying to accomplish. A combination of reading papers on the subject and learning about KNN in our data science class eventually allowed me to understand and implement this test.

Overall, using SIFT features with the KNN ratio test has proven to be an extremely robust method of object identification and I am somewhat blown away by its stellar performance in terms of accuracy.

GUI

Fig. 1 *The Mock GUI*

In figure 1, you can see a mock-up of what I originally envisioned the program might look like. It contains three primary components. First, a live feed from the user's webcam that allows them to make sure the card is in frame and in focus. Second, a text area that contains the decklist the program is generating. Third, an information panel to tell the user which card the webcam image was identified as, along with the ability to add or remove that card from the text area. This mock GUI is simple but contains all the basic operations that the program requires to perform its intended function.

The original plan for the program's Graphical User Interface (GUI) framework was to use python's included version of the tcl/tk framework called tkinter. Unfortunately, tkinter had one major problem. It had no innate threading capability. This meant that a live webcam feed actively harms the performance of the rest of the program, as significant processing time in the main thread is used to update an image many times a second. The result is an unresponsive, slow interface that is difficult to use.

Fig. 2 *An early Qt-based GUI*

Instead, I swapped to using PyQt5. PyQt accommodates easy threading using the built-in, extensible QThread class. Figure 2 shows an early version of the Qt-based GUI. When compared to the mock up, I have added a drop-down menu for the user to select a set, in line with the change in identification strategy. Additionally, the information panel has moved and has changed to display the image of the identified card instead of just text information. Hopefully, this will allow the user to more quickly and easily recognize if the identified card is indeed the card they put in front of the webcam.

Fig. 3 *The Final GUI*

In figure 3 you can see the final version of the GUI. Elements have been resized to better fit their spaces, and additional features have been added. A status indicator has been added to give the user information on the current state of the program. Many more options have been added to the text area, including the ability to load from and save to a text file. Though it has changed somewhat, this final version is remarkably similar to the original mock GUI in figure 1, and retains the three fundamental sections that the mock version had.

This is my first real foray into GUI design, as I had never built one from scrath before. However, the learning curve for PyQt was pretty forgiving, and I had a pretty easy time getting my feet wet.

Unfortunately, I got ahead of myself early in the design of the PyQt GUI and had to re-do large sections of code because of it. PyQt implements a slot/signal system to allow different GUI elements (such as buttons, text fields, etc) to communicate with each other. Essentially, PyQt

allows me to create a signal that a GUI element (let's call it *Element A*) can emit. Data and other information can piggyback on that signal, if need be. I can then plug this signal into a slot of another GUI element (*Element B*) which is essentially a method that is executed when the signal is emitted by *Element A*. Whatever data was piggybacked onto the signal can be passed to the slot in Element B as arguments. This is a relatively elegant and highly object-oriented system for handling communication between GUI elements. Importantly, a proper slot/signal setup does not require me to pass individual GUI elements as argument for other elements, and instead all GUI elements can communicate on equal terms.

Where I went wrong early in my PyQt GUI design was that I did not understand the benefits of this slot/signal design. Instead, I was passing elements to newly-created elements as arguments in order to facilitate communication, and this was a huge mistake. This meant that many of my elements were dependant on others not only to function properly, but even to properly initialize. It also meant that altering one element would have a cascade effect on other elements, which would require editing those elements as well. Once I figured out that I should have been using the signal/slot system from the beginning, I had already completed most of the GUI design and had to retool significant amounts of code to operate under the signal/slot system. Indeed, certain elements of the code have still not been ported to the signal/slot system. In the future, when working with a new tool like PyQt, I will try to give more credit to the 'intended' systems, even if I don't understand the benefits right away.

Despite this, I am quite happy with how the GUI turned out. In my opinion it is transparent, easy to use, and (most importantly) true to the core functionality of the program.

Structure of Card Identification

The implementation for card identification part of the program follows a waterfall design, where classes/functions feed into each other in a linear manner.

The class **card_set_json** (defined in mtg_json_get.py) accepts the setcode as input. It will read in the MTG-JSON files (using python's built-in json package) and create lists that detail all the relevant information about the cards in that set. For an instance of card_set_json called cardset, cardset.names contains the list of all names for cards in that set, cardset.uids contains all of the unique card IDs that MTG-JSON uses. Importantly, one of the lists is cardset.imgurls, which contains individual URLs pointing to card images hosted on the official MTG reference site. These lists are then used by the fetchSetImages function.

The function **fetchSetImages** (defined in fetchSetImages.py) accepts the setcode as input and checks to see if there are local <setcode>.images and <setcode>.names files for the given set. If these files do not exist, fetchSetImages will call card_set_json. Using the list of card image URLs provided by card_set_json, the function will fetch the individual card images and place them in a dictionary, keyed to the unique IDs used by the MTG-JSON files. It will then save this dictionary to the local disk as <setcode>.images using python's included object serialization package, pickle. The same will happen for the card names, which will be saved as <setcode>.names. These dictionaries, keyed to the MTG-JSON unique IDs, are then used by the processSetImages function.

The function **processSetImages** (defined in processSetImages.py) takes the dictionary of card images from fetchSetImages and creates a SIFT detector object using the OpenCV-contrib toolkit. processSetImage then runs each card image though the SIFT detector and places the output (a position and description vector for each image) into new position and description dictionaries keyed to the same unique IDs used in fetchSetImages. Those dictionaries, as well as the SIFT detector object are then used by the compare2set class.

The **compare2set** class (defined in comare2set.py) is a python class that handles the actual identification of cards. It also is the code that calls the previous functions and classes, making it the root for the overall card identification structure. It accepts a single initialization argument: the unique setcode for the set the user specifies. Upon creation, the new compare2set object will call the previously mentioned functions and accept all their output. This means that after the initialization function has run, the compare2set object has access to dictionaries containing the names, images, and keypoints of all cards in the specified set. Finally, compare2set creates an OpenCV keypoint matcher object that allows for the calculation of nearest neighbor keypoints.

The compare2set class has one method: the **compareimg** method. This method accepts as input an image (this is the webcam image of a physical card). compareimg then calculates keypoints of the webcam image with the SIFT detection object, and uses the keypoint matcher and the KNN ratio test to calculate the number of matches that the webcam image has with each individual card image from the fetchSetImages card image dictionary. The image with the

highest number of keypoint matches is selected, and compareimg returns the name and image of the now-identified card.

Overall Program Layout

Fig. 3 *The three sections of the GUI*

The program begins with MTGCardReader.py, which is the main file that the user would run and the root of the entire program. MTGCardReader.py is an extension of the PyQt QWidget class, the base class for Qt Widgets/Windows. This main file instantiates all GUI elements and hooks them into each other under the slot/signal paradigm.

The GUI elements are arranged using a system of layouts, a method of element placement that PyQt supports. Specifically, a base grid layout that contains multiple nested horizontal or vertical linear layouts in at different points. For instance, the the entire right side of the GUI is a vertical layout that contains the text box and three horizontal layouts, one for the load and save buttons, one for the sideboard and divider buttons, and a final one for the other text

options. When using these layouts, I can set the individual elements to scale in size with the layout and with other elements which cannot be resized. This means that with a correct layout structure, I do not have to manually specify the sizes of objects, or their specific location in the window.

The final version of the GUI contains three main sections (input section, information section , text section) that represent different functions of the program. As the main file, MTGCardReader.py sets these sections up in the layout structure, but we will consider each of them individually.

MTGCardReader.py also contains code that triggers new windows on certain actions. When the user attempts to quit the program, MTGCardReader.py will instead create a new window prompting the user to confirm that they actually intend to close the program. This step is crucial as the program does not auto-save the contents of the text box, and if the user accidentally closes the program without saving, all their work will have been lost. Similarly, if the program does not register that a webcam is connected or the webcam fails while the program is running, MTGCardReader.py will display a new window containing an error message that will ask the user to save their work, then restart the program.

The Input Section

Unless otherwise stated, all functions and operations described here are defined in MTGCardReader.py

The Input Section

The first of three sections of the GUI is the input section, containing four primary elements: The set selection dropdown menu, the status indicator, the live webcam feed, and the Read button.

- Set Selection Dropdown Menu

This dropdown menu is created with PyQt's QComboBox class, and is populated by the getSets function (defined in mtg_json_get.py) which derives the setcode of all sets from the MTG-JSON files. On program startup, the default selection is 'None', but this option is removed from the dropdown menu when any actual set is selected. When the dropdown selection is changed, the switchset function is executed, which creates a compare2set (defined in compare2set.py) object with the newly selected setcode as its argument, allowing cards from that set to be identified when the user pushes the Read button.

- Status Indicator

The status indicator lets the user know what state the program is currently in, so that they know what they need to do next. When the program is first run, the status indicator will prompt the user to select a set with the dropdown menu. When a set is selected, the indicator will tell the user that that set is currently being loaded, and when the compare2set object has been fully initialized, the status indicator display let the user know that the program is ready for card identification. When the Read button is pushed, the status indicator will tell the user that the program is busy reading the card, and when that has finished the indicator once again indicates that the program is ready to identify another card.

- Webcam Feed

The webcam feed is a PyQt image label that is updated every iteration of a loop that that uses OpenCV to read a frame from the webcam, and display it on the label. If this loop were to exist in the main thread of MTGCardReader.py, it would slow down the responsiveness of the program as a whole. Instead, this loop runs in my QWebcamThread class (defined in QWebcamThread.py), which is an extension of PyQt's QThread class. By running in a thread, the webcam feed can be updated many times a second without impact on the performance of the rest of the program.

- Read Button

The Read button is an instance of PyQt's QButton class (so are all the other buttons in the GUI). The Read button is grayed out until a compare2set object has been created by selecting a set in the set selection dropdown menu. Once such an object has been created, the button becomes pushable and pushing it calls the read_match function, which in turn calls the compare2set object's compareimg method, passing in the current image from the QWebcamThread as the argument. This identifies the card in the webcam image, and that information is passed to the information section.

Information Section

The Information Section

Unless otherwise stated, all functions and operations described here are defined in MTGCardReader.py

The second section of the GUI is the information section containing four primary elements: add buttons, remove buttons, the card name label and the card image label.

- Card Name and Image Labels

These labels are where the information about the identified card is placed. As discussed in the previous section, when the user pushes the Read button, the card name and image are passed to these labels and displayed for the user to see. Before any card is identified, or after a new set has been selected by the user, these labels are blank.

- Add Buttons

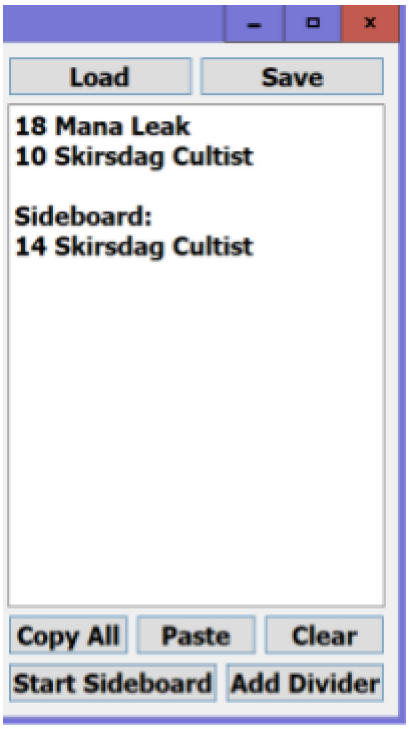
These buttons add the appropriate number of the identified card to the decklist in the text box from the next section. If a line of the textbox already contains some number of that card, these buttons will add to the quantity on that line. If no such line exists, a new line will be

created. These buttons only become active when a card has been identified, and are grayed out before that point.

- Remove Buttons

Similarly, these buttons remove quantity from a line of the text box that already contains the identified card. If the quantity would then be equal to or less than 0, the line is simply deleted. These buttons have no effect if no relevant line exists in the first place. These buttons also only become active when a card has been identified, and are grayed out before that point.

Text Section



The Text Section

Unless otherwise stated, all functions and operations described here are defined in MTGCardReader.py

The third and final section of the GUI is the text section containing eight primary elements: the textbox itself, the load button, the save button, the copy all button, the paste button, the clear button, the sideboard button and the divider button.

- Text Box

The text box is an extension of PyQt's QPlainTextEdit class called QMtgPlainTextEdit. It adds multiple methods to the underlying class that are all executed by buttons, including all the buttons from this section and the add/remove buttons from the previous section. (The QMtgPlainTextEdit class and its methods are defined in QMtgPlainTextEdit.py)

- Save and Load Buttons

These buttons use PyQt's QFileDialog to provide basic saving/loading functionality for the text box contents. As the textbox only handles plaintext, these really only work if you are handling plaintext files, but as plaintext is much simpler and easier to handle, I think this tradeoff is adequate.

- Copy All/Paste/Clear Buttons

These buttons implement common text options you might expect to find in any text editor. In particular, the paste and clear buttons execute pre-existing methods in PyQt's QPlainTextEdit class and did not need to be implemented or altered by me in my QMtgPlainTextEdit class.

- Sideboard and Divider Buttons

MTG decks have sideboards, which are 15 cards that serve as a deck's substitutes. Mainboard cards (cards not in the sideboard) can be swapped out for sideboard cards after the first game in a match. Some number of an individual card can be in the sideboard, even if that card also occurs in the mainboard. Therefore, a decklist needs to have a way to separate mainboard and sideboard cards. These buttons add dividers that accomplish this by preventing the add/remove buttons from changing anything above them. For example, if you tried to add one copy of the card Brainstorm to this text box:

3 Brainstorm

you would simply change the 3 to a 4. However, if you were to first add a sideboard divider and then add one copy of Brainstorm, you would be left with this text box:

3 Brainstorm Sideboard: 1 Brainstorm

Conclusion

A couple of way to quantify the project:

- 86 commits to the GitHub repository
- 767 lines in the final version of the program

Overall, I am quite happy with how this project turned out. It functions beyond what I thought I would be able to accomplish, and I will use it myself many times in the future. The project itself stayed very true to its original concept and I feel like the compromises and adjustments that I did have to make do not represent a significant departure from the initial idea.

Over the course of this project, I learned a lot about GUI design and feature detection, as well as version control with git. At times, I would run into brick walls that seemed impossible to pass, but with time and dedication, I was able to overcome or avoid these issues. I am very satisfied that I was able to complete this project to a degree that I find satisfactory, and will remember this experience at times in the future when a task seems insurmountable.

That said, there are still changes and improvements that could be made to my program. I would like to reduce the run times for both set loading and card identification, so that the program is faster to use. I would like to set up both those functions in a thread as well, so that they do not cause the main program to hang while they are running. As a more long term goal, I

would like to find some way to eliminate the set-selection element and identify a card based on image alone, per the original concept of the program.

In terms of distribution, I had originally considered trying to sell the program as a piece of consumer software. Unfortunately, the use of multiple licensed tools during the development of the project means that this is not really possible any more. While tools like SIFT and PyQt allowed me to make progress I never would have been able to make if I had to write everything from scratch, they come with the caveat that any commercial use must be licensed, and these licenses can be very expensive.

Instead, the project will remain on its public GitHub repository (at github.com/TrifectaIII/MTGCardReader) for anyone to download and use in the future. In fact, I did have one individual (who as near as I can tell is from Brazil and is not associated with Willamette or myself in any way) fork my repository in February. To aid distribution, I would also like to eventually build my program as a standalone executable, rather than require a user to install python and the requisite packages manually.