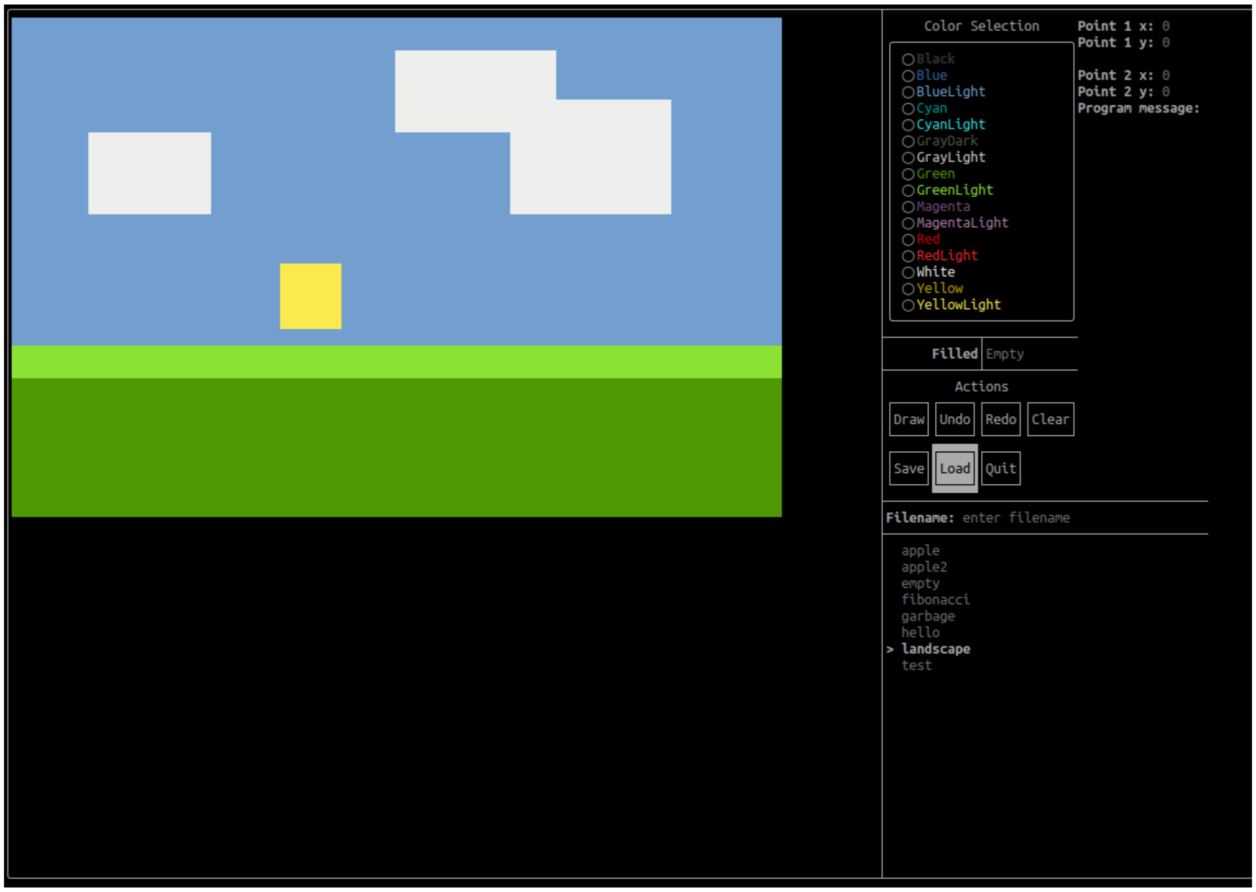
Angela Traugott
CS 20: Advanced Programming with Data Structures/C++
Prof. Carlos Moreno
19 December 2022

Final Project: Cubist Drawing Program



```
Point 1 x: 0
Point 1 y: 0

Point 2 x: 0
Point 2 y: 0
Program message:

Filename: enter filename
```

The Idea

Drawing programs like MS Paint have been around for decades. For this project, I sought to create my own drawing program written in C++, complete with an ability to save and load multiple drawings. Since my drawing program only uses quadrilaterals (squares and rectangles), it is very useful for making cubist art, like the beautiful landscape I created above.

Implementation

The program runs in a TUI. The main drawing canvas is populated by Square objects that are stored in a **LinkedStack**. Pushing to the stack draws new squares onto the canvas, and popping from the stack removes the most recent square (a.k.a. the Undo button). There is also a separate LinkedStack that acts as the redo buffer, and gets pushed to every time Undo is called on the canvas, and popped from every time Redo is called.

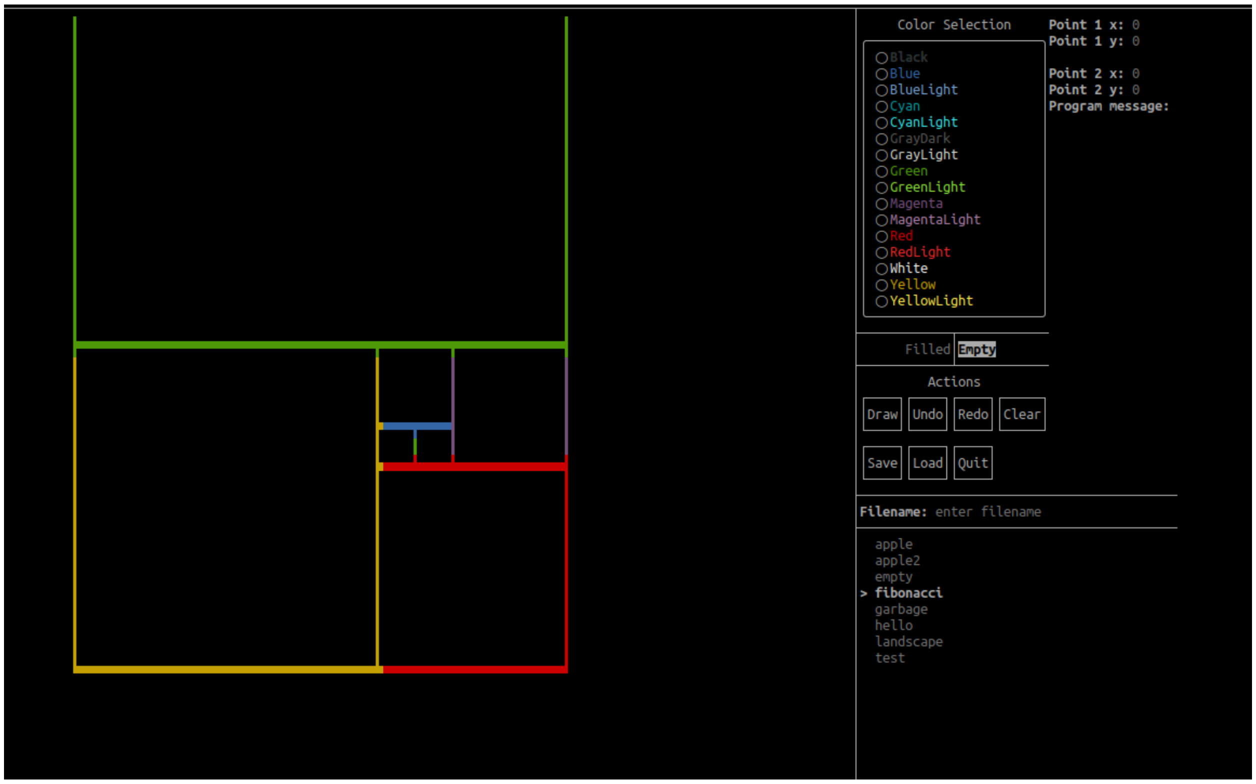
The LinkedStack also has the ability to save() its contents to a file by storing information about each square line-by-line. Another function, load(), can create a LinkedStack by reading from a file. Every time the user provides a name for a drawing and then clicks the Save button, the drawing is saved to a file and the filename is inserted into the SortedList.

Another ADT is the **SortedList**, which is implemented recursively and acts as the linked list that stores the string names of each drawing file that has been saved. Inserting into the SortedList puts the string in the right place according to ascending alphabetical order. Running the **binary search** algorithm in the SortedList produces a boolean depending on whether or not the target string was found in the list. Also, a populate() function is used to convert an unsorted vector of strings into a SortedList (and sort the vector in the process) via the **quicksort** algorithm. This is useful for the TUI, since the file selection menu in the TUI is populated with a vector.

I also wrote some additional code for the TUI itself, after copying and making adjustments to the homework03 code:

- A filename menu to show names of drawings that have already been saved.

- An incremental search for the sorted vector of filenames, so that when a user starts typing in the filename field, the filename menu is narrowed down to only show filenames that contain the string the user is typing.
- A “Filled”/”Empty” toggle for the user to display all squares on the canvas as either filled or empty.

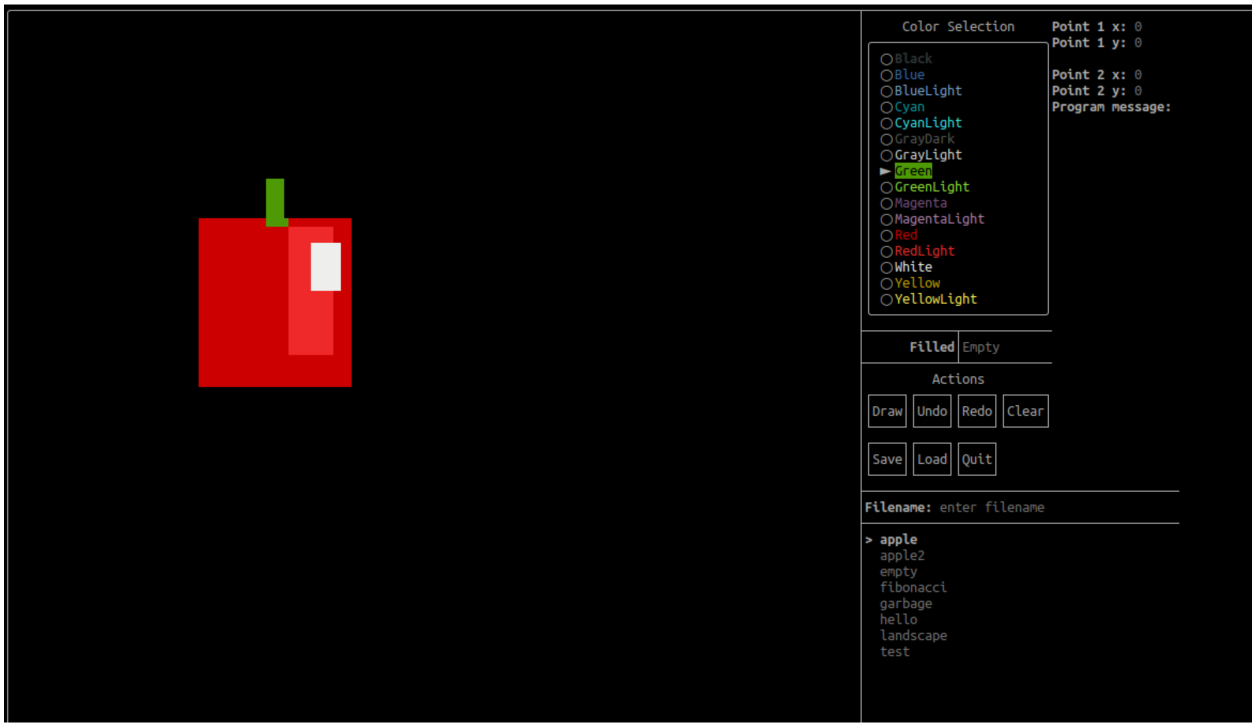


```
Point 1 x: 0
Point 1 y: 0

Point 2 x: 0
Point 2 y: 0
Program message:
```

Filled [✓ Empty]

```
Filename: enter filename
```



```
Point 1 x: 0
Point 1 y: 0

Point 2 x: 0
Point 2 y: 0
Program message:

Filename: enter filename
```

Above: A fibonacci spiral looks best when rendered with “Empty” toggled, but an apple looks best with “Filled” toggled.

Setbacks & Future Work

In general, I am pleased with how this project turned out. I implemented most of what I had planned to do, and the existing program runs smoothly according to the testing I’ve done. I also learned a lot about how the ftxui library works. However, there are some significant problems.

The most obvious problem is that I did not meet the minimum line requirement for this project. The project needs to have been 1000 lines, and instead it is only about 920 lines. The primary reason this happened is that I had accidentally copied an unrelated file into the src directory that had about 150 lines of code, and only noticed the file the night before the project was due. Also, as I kept cleaning up the code while documenting it, I found the code kept getting shorter. If I had more time, I would meet the line number requirement by including more shapes (e.g. triangle, line, etc) that can be drawn on the canvas and stored in their own classes.

Speaking of other shapes, I spent a good deal of time trying to figure out how to implement the idea of drawing multiple kinds of shapes. I kept running into difficulties with the class inheritance and passing the canvas into draw() methods for the shapes (nothing was showing up on the canvas, even if I passed it by reference). I have no doubt that I would have been able to sort this out if I had more time, but I just didn't have enough time for this project.

There are some functions in SortedList and LinkedStack, like copy(), that are not used in the TUI. These functions are still very useful for debugging purposes, but implementing them in the TUI somehow would also probably help meet the required line count.

Another useful feature would be to let users draw shapes with the mouse. This is also something I had originally thought of including, but I would need to learn more about how the TUI works with mouse coordinates.