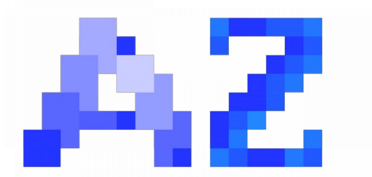


Azul

Fall 2021

Student: Anthony Schreiber

Advisor: Randy Appleton

Introduction

Hello, my name is Anthony Schreiber, I am currently a Junior here at Northern as of the Fall of 2021. I found my love for computers since a young age. Messing with computer programming when I was younger, and even toying with some 3D graphic software such as blender were some of my best memories. The choice of my computer science degree was easy due to this constant involvement with computers. I hope to further continue my education if the opportunity presents itself. However, if this doesn't come to fruition I hope to work as a computer graphics engineer working on interesting new technology based purely around making the virtual world closer to our own. This field interests me as I have always loved the visual advancement of computer graphics over the passing years and how artist are able to transform this technology into something exhilarating. Currently I have pursued two software engineer internships though none are directly related to computer graphics. This project helped to dip my toes into the world of graphics even if it is only in the 2D space.

Motivation

Every programmer uses some form of a text editor everyday, so I thought why not write my own? I felt like I should give an attempt at a tool I use so often and maybe with enough polish it could become my everyday text editor.

What is Azul?

Simply put, it's a lightweight "portable" text editor which heavily emphasizes open source and speed over anything else. The reason for the quotes around portable is due to the time frame of the project portability hasn't been tested to the extent I would like. The problem this solves is the inherent effect of bloat in software today which tries to solve every problem a user may not need a solution to, and packages it into an enormous file size. With the effect of open source it allows the user to customize the package based on their needs without the multi year process of perfecting either a Vim or Emacs configuration file to ones liking, or a user could could contribute to making Azul more rounded. One of my main incentives for developing a text editor is because I wanted to experiment with making real software from scratch without a library holding my hand.

What does Azul do?

Everything related to text editing, this includes paging, syntax highlighting, saving, deleting, specialized text editing commands (delete word, start line, end line), and even more! The features are relatively lightweight and are used mostly for making the user's life easier. This text editor was

designed with programmers in mind, and with that most of the commands are helpful in speeding up the process of modifying or writing robust code.

Technologies

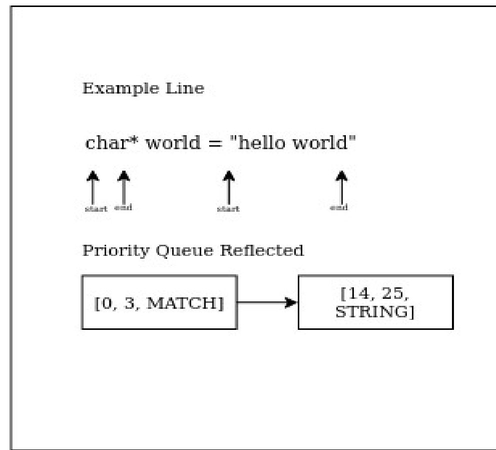
The technologies around Azul are minimal. It uses SDL to handle keyboard inputs, various window events, and all the OpenGL API handling. The program also uses KiWi to handle all the graphical interfaces which is relatively boilerplate and can be modified at will as it builds upon existing SDL technology. The last library is tinyfiledialogs which handles all the file opening and saving dialog. This library is a singular C file and header. The full program is written in C99 all throughout.

How it's built

The first step of the program was tackling character rendering, which in my eyes left me two options, ttf rendering from a font file or generate a character map from an image of my liking. I went with the second option, however I do not recommend this approach. It worked great for my needs because I wanted one specific font and for it to be reminiscent of the DOS character map. Because of this, everything is fixed and if a user hates the DOS character map they would be forced to modify the pre-compiled variables to fit with there own character map. This sucks for any project aspiring to be the world's next best text editor but for my own personal goals it aligned well. The next step after building my character map is to actually use it. This was really easy as SDL allows me to copy a texture to the current rendering target, which simply meant finding the correct character and passing the right position and passing both to the SDL_RenderCopy function. The only issue I ran into with this method came about when implementing syntax highlighting, more on that later.

The largest initial challenge was handling memory for all these various lines and characters. The current system in place allocates both 50 lines and characters as a baseline with a scaling variable of 1.5x. This scaling allows less re-allocations per action because it eliminates the need to reallocate each character pointer to hold a new character or for each new line created. This system was desirable for my use case because users are rampantly entering new characters and lines so any speedups in this area were desirable. Some current issues with this are there are no bounds to the size of memory which is going to be implemented when I decide on a reasonable limit.

Syntax highlighting was a major goal of this project as I feel many programmers would seemingly throw away an editor without it. Due to the editor being a line by line structure, it made most of the highlighting as simple as throwing it in a priority queue with the priority being assigned to the start position of the match. As represented in the diagram below.



The priority queue allows the syntax highlighter to adjust for any changes the user makes in real time by having the ability to highlight new incoming characters varying in ranges. The current iteration of the highlighter works in a manner so that if there is a conflicting range it will be disregarded because it is programmed based on priority with comments having the highest priority. The reason for this system of storing position and type is so when rendering each character we can adjust the color of the character map when we are in range of an element in the queue. The program determines the color based on the type stored in the vector. Currently the editor is capable of the types below

```
enum syntax_flags {
      NORMAL = 0,
      NUMBER,
      COMMENT,
      STRING,
      MATCH,
      PRECOMPILED
};
```

This helps to give various color options to our function which translates each individual enumerator to a hex code color. This can be expanded in the future to house even more values. One inherent flaw with the system is multi line comments and string highlights. This issue is in the works of being fixed with a flag system to set if the last line has an incomplete line. This will allow for the highlighter to parse the line in attempt to find the closing character. If the character isn’t found the flag will remain and the queue will only contain one block from start of line to finish with either a string or comment type.

In preparation for user configuration I created a global command table which stores an array of commands. Each command consist of the desired key code to call it, the function pointer, and the type. Type is more broad but it’s used for identifying special keys within the trigger such as control. Below is a look at both the command structure and current table implemented.

```
typedef struct {
    SDL_KeyCode key;
    void (*run)();
    uint type;
    uint id;
} Command;

Command command_table[] = {
      { SDLK_RIGHT, &CURSOR_RIGHT, C_NORMAL, 0 },
      { SDLK_LEFT, &CURSOR_LEFT, C_NORMAL, 1 },
      { SDLK_UP, &CURSOR_UP, C_BOTH, 2},
      { SDLK_DOWN, &CURSOR_DOWN, C_BOTH, 3},
      { SDLK_PAGEUP, &PAGE_UP, C_NORMAL, 4},
      { SDLK_PAGEDOWN, &PAGE_DOWN, C_NORMAL, 5},
      { SDLK_RETURN, &RETURN, C_NORMAL, 6},
      { SDLK_TAB, &TAB, C_NORMAL, 7},
      { SDLK_RCTRL, &R_CTRL, C_NORMAL, 8},
      { SDLK_LCTRL, &L_CTRL, C_NORMAL, 9},
      { SDLK_BACKSPACE, &BACKSPACE, C_NORMAL, 10},
      { SDLK_a, &START_LINE, C_CONTROL_KEY, 11},
      { SDLK_e, &END_LINE, C_CONTROL_KEY, 12},
      { SDLK_DELETE, &DELETE_WORD_F, C_CONTROL_KEY, 13},
      { SDLK_BACKSPACE, &DELETE_WORD_B, C_CONTROL_KEY, 14},
      { SDLK_LEFT, &LAST_WORD, C_CONTROL_KEY, 15},
      { SDLK_RIGHT, &NEXT_WORD, C_CONTROL_KEY, 16},
      { SDLK_o, &open_file, C_CONTROL_KEY, 17},
      { SDLK_SPACE, &MARK, C_CONTROL_KEY, 18},
      { SDLK_n, &NEXT_PAGE, C_CONTROL_KEY, 19},
      { SDLK_b, &LAST_PAGE, C_CONTROL_KEY, 20},
      { SDLK_x, &close_page, C_CONTROL_KEY, 21},
      { SDLK_w, &NEW_PAGE, C_CONTROL_KEY, 22},
      { SDLK_s, &save_file, C_CONTROL_KEY, 23},
      { SDLK_DELETE, &BACKSPACE_F, C_NORMAL, 24},
};
```

During the writeup for the commands I tried to keep the names as concise as possible to avoid confusion, though some of the naming needs work I believe it’s in a rather readable state. The command table is checked every time a key combination is pressed which works incredibly well. This system makes adding a command as easy as throwing it into the array and assigning it the desired values.

Some commands required more robust solutions. Most notable of the bunch is backspace, this required handling if the row is empty and a row below it exist we would shift the row below up. Next, we had to handle if our position in the row was at the zero position we would append the current row to the one above it. These specialized events make the editor feel more complete and provides a better use experience. Two commands which required more attention are copying and pasting. This required special checks because to copy it requires use of the marking system. Most other commands in the table

are straight forward and are typically below 10 lines. Next on the TODO list for commands is to add I-search to help make searching for keywords or lines possible.

Structure

The editor splits the problem intro fragmented structures that all fit into a centralized editor structure. All structures are used entirely to keep the code readable and robust. The editor houses general information such as window size, current font, if it is alive, and all tiles. The key note of this is the Tiles which is the precursor to the soon to be implemented tile manager. Each tile holds an array of pages which has a capacity currently of 16 but can be adjusted at will. Within the tile structure array, information is stored such as the real size of elements, position, and the tile size. Pages store multiple line structures and general information about the file and syntax highlighting desired. This structure allows complete control over the program in an understandable manner.

What would I change?

The one component of the software I pushed off was the GUI. I felt the functionality of the editor itself was more crucial. Which I still feel is true but the interface should still have not been as neglected as it was here. Towards the end of the project I rewrote most of the GUI to give it somewhat of a presentable nature but even then it is not near what I was hoping for. Another aspect I was hoping to touch more on was both refactoring and more robust testing. During the course of the semester I was so focused on creating something that functioned and felt great which is a major milestone. However, I do feel I should have dedicated time towards the end for implementing some form of automated testing and refactoring specific functions such as the main syntax highlighter. One other issue in the code is the zooming feature, currently the scaling of the text is done pretty poorly and it results in some scale factors having unreadable text. This should be a pretty easy fix and mostly would involve adjusting some math.

What’s next?

I hope to continue this project well into the future. The open ended nature of a text editor allows me to scale it to my desire, perfect as a hobby. Below is a rounded feature list for the next few months.

- Portability, target all Linux distributions while providing an easy install.
- Undo & Redo
- Refactored GUI
- Intuitive Configuration

- Full fledged tile manager
- Robust & Automated testing

What I learned

Time-management is tough, especially with such an open ended project. There were a lot more features I wanted to deliver on before my presentation, especially in regards to fixing the current GUI but certain features took precedence . Memory management gets a bad reputation and is more enjoyable than I expected. Though, debugging memory can be troublesome I have become more aware of the memory I am using and what leads to death related to memory. SDL is great, but it got me intrigued with the underlying graphics API even more. I hope to dive deeper into OpenGL or even better Vulkan following this project. In the future I plan to handle all rendering myself within Azul and let SDL solely handle user inputs. My programming style has evolved during this project as I am more wary of variable types and I would like to believe I have a more data-orientated mindset. The project expanded my knowledge of data structures and using them creatively. I really enjoyed employing different data structures for different problems such as the focus of the syntax highlighter being a priority queue and with the future tiling manger being a simple tree.

Closing thoughts

Overall, this project was great. It widened my software knowledge and helped develop new skills. I hope to continue this project and get it to a point where I can completely switch to using it as my daily editor. Thank you to both my committee and advisor for contributing to the knowledge I was able to use to foster the development of this project.