**Last Stop Development Journal**

|  |  |
| --- | --- |
| Wed March 11 | 4:30 PM. Development begins. I’m debating between using the .NET Framework in C# and between using Electron/TypeScript. After a bit of research, I conclude that using C# is the more prudent decision. It feels like this project would benefit from a more robust type system, and Dragon is able to use its full feature set with Windows Forms. Otherwise, I’d have to do RPC with a WinForms stub.  5:30 PM. First check in. Titled the project “Last Stop” because I hope this is the last time I’ll ever try to make an editor. Feeling discouraged, low energy. I’ve been here before and it doesn’t feel good to start from zero again. This really has to be the last time.  Decision paralysis sets in. Not sure where to start, what the project scope should be. The ultimate goal is to be able to make software by voice with no compromises in any language (provided I’ve made voice definitions for it). Many ways to achieve that goal, all different.  Preliminary structures:   * Model: a data object containing all reversible (i.e. voice controlled) information for the application. The model is only modified by deltas. Snapshots can be taken of the model (these are stored OUTSIDE the serialized model). Undo commands amount to re-applying stored deltas to a recently taken snapshot. (This prevents the need for storing inverse deltas, which is really a pain in the butt.) * External: an object containing all information which is not reversible, that is, outside of voice control. Things like changes on the file system, time/date, network access, and so on belong here. * View: The collection of windows which subscribe to the model. * Windows: each window subscribes to a certain view of the model. Changes in the model trigger events sent to the window. contains a single UserControl and subscribes to a certain view of the model. * Content: a character-matrix UserControl with background/color/in-text markup options * Input: a textbox UserControl   5:54 PM. Taking a break.  9:18 PM. Resuming!  Let’s think about how to carve this program up into separable components.  #1: View components. They know nothing about the model or where their data is coming from – they simply display what clients tell them to display and raise events back to clients when stuff happens.  #2. Model. Consists of open buffers and their text, cursors/anchors, etc.  #2: Project/file system monitor. This tracks what files and directories exist under the project root and can raise events (maybe? Or is it polling?) when files have been modified, added, or deleted. Clients can issue save and load commands, but this module need know nothing about why.  #3: Command parser. The front end for lexical analysis and parsing of spoken text takes spoken language definitions (which are guaranteed not to change between snapshots) and spits out a list of model deltas and/or irreversible commands.  #4: Syntax highlighter.  11:19 PM. Decided to use F# when I realized it’s basically a modernized, .NET compatible OCaml. Nice of them to let me know! It’s *good shit though.* Done for the night.  **Daily effort: 3h 25m**  **Total effort: 3h 25m** |
| Thu March 12 | 8:34 AM. It looks like I’m either using .NET Core with VS code (and the nice editor) or .NET Framework with Visual Studio (and QEdit). That’s going to restrict my options either way. I’m thinking it might be smart to just go with [15 minute break] Visual Studio. QEdit isn’t that bad. Break @ 9:00 AM  11:37 AM. Going to investigate mixing framework and core. Okay, it really is impossible. As far as I can tell… I’m going to stick with Framework and use QEdit. Time to start making defs. Break 28 min.  2:45 pm stopping, mostly done with defs  **Daily effort: 3h 19m**  **Total effort: 6h 44m** |
| Fri March 13 | Did about 1 hour of work… but it got killed off when I hard shutdown the computer.  **Daily effort: 1h 0m**  **Total effort: 7h 44m** |
| Sat March 14 | Ain’t do shit :<  **Daily effort: 0h 0m**  **Total effort: 7h 44m** |
| Sun March 15 | 9:40 ~ 10:00 AM. Honestly, looking like F# might \*not\* be the right choice. Gah. The documentation is broken as fuck, windows forms is janky… It’s really discouraging to see how Microsoft has let this language go to the weeds since about 2017.  Fine. I’ll use Electron/JS. Industry standards…  10:24 AM. Typescript/electron is a joy to use. Goddammit. Stopping for a bit.  12:54 PM: Starting back up. Did 13 minutes earlier. Going to create GUI first.  1:40 PM: Stopping. Feeling somewhat sick.  2:05 PM: Resuming. Building IPC.  3:40 PM: Stopping. Made solid progress. Need to get text drawing in window, subscriptions.  **Daily effort: 3h 3m**  **Total effort: 10h 47m** |
| Mon March 16 | Did about 3h 40m of work with Lux on the rendering. It was good.  Check commit history for more info.  Efficiency is good: 2 hz update on 4 windows taking up 2 monitors = 16% cpu  Next steps: Fix jank in rendering, fix vertical positioning of lines, add background color rendering, start on model or some other component!  **Daily effort: 3h 40m**  **Total effort: 14h 27m** |
| Tue March 17 | 7:25 AM start. Fixing some of the above issues – background rendering, line spacing, vertical positioning.  7:42 AM. All good. Thinking now about next steps.  Okay, big ideas incoming.   * Rather than displaying line numbers relative to the file that’s being displayed, display line numbers relative to the window. Then offset each window so that line numbers are unique across all windows on screen. * Pass the base line number to each window in the update message. * Display the token index of every fourth or fifth blob of stuff on the screen as a very small subscript of some kind. It might not need to be actual text – something I can quickly memorize over time. * Refer to tokens as decimals: line number, then token index as decimal. For instance, 141.2 is the third token on line 141. * Line numbers begin AFTER the last window ID, so window IDs are still unique. (11, for instance, is definitely a window.) * Numbers from 0 to 10 are viewed as actual numbers. Numbers starting at 11 are viewed as window or line or token references unless prefixed “actual 15”.   Stop 7:56  Start 10:05, stop 10:27. Frustrated. Unclear what solution to tokenization should be. Must be general but capable of specifics. Difficult to get it worked out. Easy to make mistake that will require deep changes later.  **Daily effort: 53m**  **Total effort: 15h 20m** |
| Wed March 18 | Ain’t do shit :3 |
| Thu March 19 | Hardcore design work (see below after daily entries).  Also worked on Model using paint.  **Daily effort: 2h 30m**  **Total effort: 17h 50m** |
| Fri March 20 | Continued design work. Getting very close to the point where I can start coding without too much worry. Still need to work out –   * Speakable text for navigation/etc * Symbol gathering (in or not in model?) * Syntax highlighting and token caching (in or not in model?)   ~1h 30m |
| Sat March 21 | ~1h 0m |
| Sun March 22 | ~3h 0m |

**Store.**

The data structure which enables the model to apply changes both forward and backward, and which allows changes to propagate through events to listeners such as the view.

The store is a JavaScript object consisting of values, recursively defined as numbers, strings, booleans, null, lists of values, and maps from string to value.

Positions in the store are expressed by lists. T he empty list corresponds to the notion of “here”, which out of context refers to the root of the store. Numbers and strings in a position list, in turn, correspond to positions in lists and values in maps, respectively.

For example, the position *[“symbol”, 144, “name”]* corresponds to the ordinary JavaScript notation *store.symbol[144].name*.

Deltas modify the store in such a way that they can be undone. Here is a list of all deltas. (Note that a delta with no effect – such as clearing a position that does not exist – is not stored, and therefore has no backward delta.)

|  |  |
| --- | --- |
| **Forward** | **Backward** |
| Set(p, v) | Clear(p) or Set(p, v\_old) |
| Clear(p) | Set(p, v\_old) |
| InsertList(p, v1, v2, …, v\_n) | RemoveList(p\_1, p\_n) |
| RemoveList(p\_1, p\_n) | InsertList(p, v1, v2, …, v\_n) |
| ChangeKey(p, k\_old, k\_new) | ChangeKey(p, k\_new, k\_old) |

Each delta, when applied, is added to an undo stack and given an index. The usual undo/redo semantics then apply.

**Contexts.**

Each buffer has a base context typically determined by its filename extension. During the tokenization process, certain tokens may be encountered which change the context. However, since we may need to return to the previous context, a stack is maintained. In the child context, certain tokens may pop the context and return to the previous environment. For instance, consider the following:

import **“**./index.css**”**; **//** Your typical import **\n**

blah();

The import directive and the open quote are parsed under the base context “TypeScript”. The open quote token causes the context “DoubleQuotedString” to be pushed onto the context stack. Subsequent content is tokenized in that context until the close quote token is reached, which causes a pop of the stack. The following semicolon is parsed under the original base context. Similarly, the double-slash is parsed under the base context, but the new line is parsed under the comment’s context, popping and returning to the base context.

**Tokenization**

Tokenization can occur in a variety of environments. One efficiency problem is that changes to a large file can change the context stack, requiring re-tokenization of subsequent content. To mitigate this, each line stores the context stack immediately after processing of the new line “\n”. It is assumed that tokenization is a function only of line content and context, such that if a tokenization has been recorded under a certain context stack, the previous result can be reused.

Tokenization of buffers is required for syntax highlighting, symbol searching, and navigation/selection by line and token index. It is my desire that the editor should be fast enough to work on buffers of up to 25,000 lines. On average, such a buffer might have about 500,000 tokens. Since re-tokenization is required to correctly interpret certain forms of spoken input, it is important that this process is performed as little as possible on as small a scope as possible.

One simple solution is to defer any re-tokenization beyond a certain fixed number of lines should it cascade down the buffer. In this case, re-tokenization of the entire buffer will take place when either a command is given to do so or when a new checkpoint is set after spoken text is committed. The user simply needs to be aware that manual re-tokenization is required to reinterpret symbols far away in the buffer. Re-tokenization, in general, remains O(n), but local tokenization in response to spoken commands is O(1).