Computing Lab (CS69201) Project

MyTerm –

A Custom Terminal with X11GUI

DESIGN DOC

Shreyan Naskar

25CS60R41

ESSENTIAL FEATURES:

1) Workflow for making a terminal GUI with X11

Files & locations

- run.cpp (initialization + event loop)
- draw.cpp (window creation, drawing, per-tab state)
- termgui.cpp (program entry / initial arguments)

Implementation

- **Display / window creation**: At startup termgui.cpp parses the command-line argument (the program expects a base path) and calls run(win) after opening an X display. The code opens an X display (XOpenDisplay), chooses fonts (via XLoadQueryFont) with fallback, and creates a top-level window via a makeWindow routine implemented in draw.cpp. XMapWindow() is invoked to make the window visible.
- **Graphic context**: run.cpp builds a GC (graphics context), sets the font into the GC, and stores metrics (ascent, descent, lineHeight) to drive layout.
- **Event loop**: The main run loop in run.cpp repeatedly processes X events using XPending() / XNextEvent() — the code uses the canonical pattern to service Expose, ConfigureNotify, ButtonPress, MotionNotify, and KeyPress. run.cpp drives redraws when necessary, by calling makeNavBar and makeScreen (in draw.cpp).
- **Text rendering**: The code uses X11 text routines (font + XDrawString() style rendering). draw.cpp holds the line-by-line draw routine (maps tabState.displayBuffer lines to rows on-screen). The app uses font metrics to compute pixel positions and to support cursor drawing and line wrapping (basic fixed-width handling).
- **Event capture & mapping**:
    - The UI registers for keyboard and mouse events with XSelectInput.
    - For keyboard: run.cpp attempts to use X Input Methods (XIM/XIC) when available (the code opens an input context XCreateIC); when available it calls wide-character input helpers (XwcLookupString) to receive Unicode characters correctly. If XIM is not available, the code still falls back to raw keycode mapping and maps keysyms for special keys.
- **Retained buffer model**: Each tab keeps a retained displayBuffer (vector of strings) inside struct tabState (declared in draw.cpp). The drawing routine always consults that buffer for what to paint; the UI never directly stores ephemeral screen text in the X server — it always re-renders from the retained buffer on expose/resize.
- **Tabs**:
    - Tabs are represented by a vector<tabState> tabs (global in draw.cpp) and tabActive index. draw.cpp implements makeNavBar() (draws the tab bar) and makeTabs() (calculates hit areas and returns positions), and run.cpp processes mouse clicks and keyboard shortcuts to add/close/switch tabs.

    - Tab addition/removal is handled through addTab() and vector operations; UI redraws follow immediately.

- **Input capture granularity**: run.cpp handles single keystrokes (KeyPress events) rather than relying on a blocking read of a TTY so the GUI can implement line-editing and key bindings (Ctrl combinations, arrows, Tab, Ctrl-R, etc.) at the per-keystroke level.

Important data structures

- struct tabState (in draw.cpp): stores per-tab UI buffers and state:
    - vector<string> displayBuffer — scrollback and last printed lines.
    - string input — current input line(s) typed by the user.
    - int currentCursorPosition — cursor index inside the input buffer (logical index).
    - cwd and title — per-tab working directory and display title.
    - flags: searchFlag, recommFlag (autocomplete/recommendation UI), multLineFlag, scrlOffset, etc.
- tabs[] & tabActive — global tab list and active tab index.

---

2) Run an external command

Files & locations

- exec.cpp — primary implementation for command execution (function execCommand and related helpers).
- run.cpp — calls execution flows and appends results into the tab buffer.

Implementation

- **Parser / Execution planner**: exec.cpp exposes execCommand() which receives a raw command string, strips whitespace and then checks for builtins (e.g., cd) and otherwise orchestrates process creation for external commands.
- **Fork & exec**:
    - For each command, the code fork()s; inside the child it uses execlp("bash", "bash", "-c", cmd, NULL) to execute the command line via bash -c. (The implementation executes the user command string through a shell invoked via execlp() rather than manually building argv[] for every command.) Several code paths that implement pipelines and multiWatch use execlp("bash", "bash", "-c", ...).
    - The result of the exec is displayed by the parent reading the child's delivered stdout/stderr (details below).
- **Capturing stdout/stderr**:
    - exec.cpp creates pipes (pipe()) for capturing child stdout and stderr (capture_out and capture_err in the code). The child dup2()s the write ends to STDOUT_FILENO / STDERR_FILENO and then execvp()s.
    - Parent closes unused pipe-ends, sets read ends possibly non-blocking, and repeatedly reads from them (via poll() or integrated read loops) to collect output and append to tabState.displayBuffer.
- **Synchronous or background**:
    - The parent waits (waitpid) for child termination for foreground commands; for background semantics the code can avoid blocking the UI or treat it specially (the code keeps track of child PIDs and reaps them).
- **Error handling**:
    - If piping or fork fails, execCommand returns an error string which is appended to the active tab buffer.
    - exec.cpp inspects child exit statuses via waitpid and records non-zero exits as an error condition which the UI prints.

---

3) Take multiline Unicode input

Files & locations

- run.cpp — keypress handling (XIM/XIC integration, XwcLookupString path).
- draw.cpp — display and cursor painting (handles multi-line input layout).
- helper_funcs.cpp — (support for locale aware operations and utilities is present for widths / printing).

Implementation

- **Unicode input**:
    - run.cpp attempts to create an X input context (XOpenIM, XCreateIC) and, if successful, uses XwcLookupString to receive wide-character input on KeyPress. This yields wide-character sequences which the code converts to UTF-8 when appending to tabState.input.
    - If XIM is not available the code falls back to a less robust path but still supports common characters.
- **Multiline support**:
    - tabState.input stores the typed input as a single string (UTF-8). The UI treats \n characters as line breaks; run.cpp and draw.cpp split and wrap text when painting.
    - The currentCursorPosition is maintained as an index in the input string; when rendering, draw.cpp computes the display coordinates by counting characters (and using font metrics) so the caret appears at the correct position across multiple displayed lines.
- **Writing to child stdin**:
    - On submit (Enter semantics), the exec.cpp machinery writes the UTF-8 bytes from the input buffer to the child's stdin (or allows the executed bash command to read them as appropriate); write() is used to send the exact bytes. The code ensures partial writes are handled via loops if needed.

---

4) Run an external command by redirecting standard input from a file (<)

Files & locations

- exec.cpp — redirection handling and pipeline construction.

Implementation

- **Parsing & redirection detection**: exec.cpp determines whether a redirection token < is present by analyzing the command string (the code leverages bash -c for basic expansions but also supports explicit redirection handling in pipeline code paths).
- **Open file and dup2()**:
    - In the child before exec, the code opens the specified input file with open(filename, O_RDONLY), and dup2()s the file descriptor into STDIN_FILENO. The code ensures original descriptor is closed afterwards.
- **Execution**:
    - After the redirection is set up, the child execs the command (execlp("bash", "bash", "-c", ...) or direct exec) and stdin reads come from the opened file.
- **Parent**:
    - Parent does not attempt to treat that stdin as interactive input; all reading is performed by the child process from the file. Parent continues to capture stdout as normal.

Error handling

- If open() fails, exec.cpp returns an error message which is appended to the tab's displayBuffer. This prevents launching the process with invalid stdin.

---

5) Run an external command by redirecting standard output to a file (>)

Files & locations

- exec.cpp

Implementation

- **Parsing & detection**: exec.cpp identifies > and handles the output redirection token; it also supports possible append semantics if provided (code currently expects typical > semantics).
- **Open file and dup2()**:
    - In the child before exec, the code opens the output target with open(filename, O_CREAT | O_TRUNC | O_WRONLY, 0644) and then dup2()s the FD to STDOUT_FILENO. STDERR may remain separate or also be redirected if 2>&1 semantics are used (the code has capture_err and capture_out logic to capture both where appropriate).
- **Parent behavior**:
    - Because child stdout writes into the file, parent does not need to capture those bytes and thus no pipe read will be appended into the tab's buffer. If stderr remains unredirected, parent may still capture it and display in the UI.

Error handling

- Permission or file creation failure is reported back to the user via appended error lines and the command is not executed.

---

6) Implementing support for pipe (|)

Files & locations

- exec.cpp — full pipeline setup logic (chain of pipes, child forks, dup2 wiring and wait).

Implementation

- **Pipeline decomposition**:
    - exec.cpp splits the command into multiple parts separated by | (the code builds getPipeParts / sizeOfParts).
- **Create pipes**:
    - The code precreates numPipes pipes and arranges them into a chainFds array so that for stage i the previous stage's read-end and the next stage's write-end are known.
- **Fork children for each stage**:
    - For each pipeline stage, the program fork()s. In the child:
        - If it's not the first stage it dup2()s the read-end of the previous pipe into STDIN_FILENO.
        - If it's not the last stage it dup2()s the write-end of the current pipe into STDOUT_FILENO.
        - If last stage and capture is desired, it dup2()s to the capture pipe created earlier so the parent can read the final output for UI display.
        - Child then execlp("bash", "bash", "-c", part, NULL) to execute the stage's shell command.
- **Parent orchestrates and cleans up**:
    - Parent closes unused ends of pipes, keeps track of all child PIDs, polls/captures stdout from the final stage, and waits (waitpid) for all pipeline PIDs to finish before returning control to the user prompt.
- **Signal / process group**:
    - The code maintains currChildPids and uses signal handling helpers to forward interruptions to the running pipeline (see signals below).

---

7) Implementing a new command multiWatch

Files & locations

- exec.cpp — multiWatchThreaded_using_pipes() function and related support variables (mwStopReq, mwDone, mwQueue, watchMsg).
- run.cpp — sets the sigint flag to stop multiWatch on Ctrl+C and integrates UI handling to dispatch stop requests.
- draw.cpp / tabState — where the UI will display multiWatch's output.

Implementation

- **Approach**: The implementation runs each requested command in **a child process** and captures their output via **pipes** created for each command. The code runs a set of worker threads that start child processes and read from their stdout, and the main multiWatch loop collects these outputs and posts updates into the UI.
- **Threaded worker per command**:
    - For each command string in the multiWatch list, the implementation spawns a detached thread (a workers vector). Each thread:
        - Creates a pipe() (unique per worker), fork()s and in the child dup2()s the pipe write end into STDOUT_FILENO and STDERR_FILENO and then execlp("bash", "bash", "-c", cmd, NULL) to run the command.
        - The parent side (worker thread) reads from the pipe read-end using poll() in a loop, collecting data into an outBuf string. It periodically polls with a timeout so it can be responsive to a global stop flag.
        - When data is read, it packs the data into a results container (guarded by results_mtx), or directly enqueues messages for the UI queue mwQueue (the code uses mwQueue + mwQueueMutex for message transfer).
- **Main multiWatch loop**:
    - While mwStopReq is false, multiWatchThreaded_using_pipes() iterates, processing results from the worker threads and appending timestamped output into the active tab's displayBuffer. Each event posted to the UI is prefixed with the command label and the current UNIX time (this is implemented by getTimeNow() calls and string formatting in exec.cpp).
- **Stopping**:
    - mwStopReq is an atomic flag; run.cpp sets it in the UI when Ctrl+C is detected. When set, workers kill their child with kill(pid, SIGINT) then SIGKILL if necessary, and multiWatch cleans up worker threads and any temporary resources (pipes fds) before returning control to the shell prompt. mwDone and cmdRunning flags are used to coordinate lifecycle.

---

8) Line Navigation with Ctrl-A and Ctrl-E

Files & locations

- run.cpp — key handling, mapping control key codes to actions.
- draw.cpp / tabState — stores cursor position and supports redraw.

Implementation

- **Key detection**: run.cpp detects control keys using the KeyPress event and interprets keysyms with XIM/XIC (wide char). The code checks keysym values and event.xkey.state for Control/Shift modifiers.
- **Specific commands**:
    - Ctrl+A — mapped to move to the start of the current input: code sets tabState.currentCursorPosition = 0.
    - Ctrl+E — mapped to move to the end of the input: code sets currentCursorPosition to input size (visual end).
- **Redraw**: After moving the cursor, makeScreen() is called to re-render the input area so the caret moves visually.

---

9) Interrupting commands running in your shell (signal handling / job control)

****COULD NOT DO THIS****

---

10) Implementing a searchable shell history

Files & locations

- helper_funcs.cpp — history persistence, getHistory(), and searching helpers.
- run.cpp — loads history at startup and uses it for up/down navigation and Ctrl+R search flow.

Implementation

- **Persistence**:
    - helper_funcs.cpp maintains historyPath (default ./input_log.txt) and implements getHistory() which reads the file at startup into a vector<string> inputs. The code expects each entry to be prefixed with numbers (the code strips numeric prefixes on getline()).
    - On each command execution the code appends the issued command to the same path (the code uses ofstream at points in the execution path — search points exist in the code that append to the log). (If code does periodic writes, the write code is centralized in helper_funcs or in exec command success paths.)
- **history builtin behavior**:
    - execCommand recognizes builtin history and prints the most recent entries — this is implemented by scanning the in-memory inputs vector and pushing up to a configured number of last entries into tabState.displayBuffer.
- **Ctrl+R search**:
    - When user hits Ctrl+R, run.cpp sets tabState.searchFlag and the UI switches to a mini prompt (rendered by makeScreen) "Enter search term:".
    - On Enter, the code calls a helper that first searches for **exact match** in the saved history (scanning backward from the most recent); if found it returns that command.
    - If an exact match is not found, the fallback is a longest-substring matching strategy: the code scans entries and computes a longest matching substring length between the search term and each history entry (the helper contains the algorithm to compute candidate matches); it picks the entries with the maximum match length and returns them as suggestions if the length is > 2.
    - If no candidate meets the >2 threshold the helper returns the message "No match for search term in history", which is appended into the active tab buffer.
- **Search complexity**:
    - The code does a backward scan for exact match (fast), and otherwise a per-entry substring/overlap comparison. Given the configured max history size, this approach is straightforward and performed inline.

---

11) Implementing auto-complete feature for file names (Tab completion)

Files & locations

- run.cpp — Tab detection and logic that drives completion.
- helper_funcs.cpp — getRecommendations() used to filter candidate filenames.
- exec.cpp — execInDir("ls", T.cwd) helper is used by the code to list files (parent spawns ls and reads results) to avoid reimplementing directory scanning (this is the shipped approach — calling ls gives a list that is then filtered).

Implementation

- **Trigger**: When KeyPress yields XK_Tab in run.cpp, the code activates completion if the tabState.input is non-empty.
- **Token extraction**:
    - The code computes a query token by calling extractQuery(T.input) which isolates the filename fragment at cursor position. If the token starts with ./ it normalizes it (removes ./) for matching ease.
- **Candidate collection**:
    - The code calls execInDir("ls", T.cwd) — this invokes a tiny helper in exec.cpp/execCommand family that runs ls in the tab's current directory and returns the resulting filenames as a vector (the implementation runs ls and parses output lines).
    - The result set is a vector of filenames (allCandidates) which is then filtered with getRecommendations(query, allCandidates) (in helper_funcs.cpp) — this is a prefix filter that returns the filenames with the given string as a prefix.
- **Resolution**:
    - **No matches** → nothing happens: recommFlag is cleared and UI unchanged.
    - **Exactly one match** → the code appends the rest of the file name into T.input (i.e., completes inline), and updates the last prompt line in T.displayBuffer so the printed prompt shows the completed text. currentCursorPosition is moved to the end of the inserted text.
    - **Multiple matches** → the UI constructs a string that enumerates the options (e.g., "1. abc.txt 2. abcd.txt") and pushes into T.displayBuffer with T.recommFlag = true. The program then waits for the user to pick a number; the UI supports numeric selection: the user types the selection number and the code uses getRecommendationChoice() helper to parse numeric selection and substitute the chosen candidate into the input buffer. (If the user cancels or times out, the selection is aborted.)
- **Longest common prefix**:
    - Prior to asking for numeric selection the code computes the longest common prefix among candidates and if that prefix extends the typed token it inserts those extra characters automatically.
- **Escaping/quoting**:

    - The code tries to preserve the surrounding command if it must insert quoted or escaped filenames — for names with spaces it relies on either user quotes or the ls output to indicate the filename; the code's insertion logic avoids breaking the rest of the typed command.

- **Hidden files & . behavior**:
    - If token starts with . the matching includes dotfiles; otherwise ls output will naturally exclude hidden files unless ls -a is used. Current implementation chooses the simple ls invocation so hidden-file behavior depends on user typing leading ..
- **UI presentation**:
    - The candidate list is presented inline inside the tab's scrollback displayBuffer so that the user sees choices and can type a single digit to confirm.

EXTRA FEATURES:

1. Left / Right Arrow — Move Across Input

Files & Locations

- run.cpp — X11 key event handling and cursor motion logic.
- draw.cpp — cursor drawing and text layout rendering.
- struct tabState (declared in draw.cpp) — stores cursor position and current input buffer.

Functional Goal
Allow users to move the cursor horizontally within the current input line to edit text inline, accurately handling multibyte UTF-8 characters.

Implementation Details

- **Event Capture**
    - In run.cpp, during X11 event loop processing (XNextEvent()), keypress events are decoded via XLookupString() / XwcLookupString().
    - When the keysym corresponds to XK_Left or XK_Right, the handler adjusts tabState.currentCursorPosition.
- **Cursor Position Tracking**
    - currentCursorPosition tracks the logical index in the UTF-8 input string, not the byte offset.
    - Movement decrements or increments the index by one "character unit," computed using helper functions that account for multibyte sequences.
- **Edge Conditions**
    - The handler guards against underflow (cursor < 0) or overflow (cursor > input length).
    - For multi-line inputs, horizontal moves stay within the logical line; if the cursor is at the end of one line and Right is pressed, it moves to the next line's start only when the newline character is part of the buffer.
- **Rendering**
    - draw.cpp computes pixel coordinates of the cursor by multiplying character count by the font's fixed width (retrieved during initialization).
    - After each cursor move, the screen is redrawn via makeScreen() to reflect the new caret position.

---

2. Up / Down Arrows — Command History Navigation

Files & Locations

- run.cpp — keypress event handling for XK_Up and XK_Down.
- helper_funcs.cpp — history management (getHistory(), inputs vector).
- tabState — maintains current input string and cursor.

Functional Goal
Provide Bash-like command recall using the up/down keys to scroll through stored history entries and edit previously executed commands.

Implementation Details

- **History Storage**
  - Command history is loaded from input_log.txt into a global vector<string> inputs at startup via getHistory().
  - Each tab accesses this shared list to recall commands.
- **Event Handling**
  - When XK_Up or XK_Down is detected in run.cpp, the active tab's historyIndex (a per-tab field) is incremented or decremented.
  - If historyIndex exceeds bounds, it is clamped (topmost or most recent command).
- **Input Buffer Replacement**
  - The retrieved history entry replaces tabState.input.
  - The cursor position currentCursorPosition is reset to the end of the string.
- **Redrawing**
  - The prompt area is redrawn using makeScreen() to reflect the loaded command.
- **Performance & Persistence**
  - History lookup is instantaneous (vector access); updates are persisted to disk upon command execution in exec.cpp.
- **Edge Handling**
  - Repeated Down beyond the latest command clears the input buffer (empty input line).
  - Multiline history entries are rendered correctly as multi-line prompts in draw.cpp.

---

3. Ctrl + V — Paste Clipboard Text

Files & Locations

- run.cpp — Ctrl+V detection and clipboard integration.
- draw.cpp — updates visible input after paste.

Functional Goal
Allow users to paste text from the system clipboard directly into the terminal input field.

Implementation Details

- **Event Detection**
  - run.cpp checks for ControlMask + V (keysym == XK_V) in keypress events.
- **Clipboard Access**
  - Uses X11 clipboard mechanism:
    - Sends a XConvertSelection() request for the CLIPBOARD or PRIMARY selection.
    - Waits for a SelectionNotify event.
    - Retrieves text data with XGetWindowProperty().
  - The result is stored as a UTF-8 string.
- **Insertion**
  - The pasted text is inserted into tabState.input at the current cursor position.
  - currentCursorPosition is advanced by the number of inserted characters.
- **Rendering**
  - After insertion, makeScreen() is called to redraw the updated input line.
- **Unicode Support**
  - The pasted data is assumed to be UTF-8 and directly merged into the input buffer.
- **Edge Handling**
  - Large paste operations are truncated to prevent buffer overflow (implementation defines a safe upper limit).
  - If clipboard access fails, a short “Clipboard empty or unavailable” message is displayed in the tab’s buffer.

---

4. Ctrl + Tab / Ctrl + Shift + Tab — Switch Between Tabs

Files & Locations

- run.cpp — captures key combinations for tab navigation.
- draw.cpp — maintains the tabs vector and tabActive index.
- makeNavBar() — redraws tab labels to show active tab.

Functional Goal
Enable keyboard-based tab switching:

- Ctrl+Tab → next tab
- Ctrl+Shift+Tab → previous tab

Implementation Details

- **Event Detection**
    - run.cpp checks for ControlMask with Tab key and uses the Shift modifier to differentiate direction.
- **Active Tab Update**
    - When triggered:
        - Increment tabActive for Ctrl+Tab.
        - Decrement for Ctrl+Shift+Tab.
    - Indices wrap around ((index + count) % numTabs).
- **State Updates**
    - The newly active tab's context becomes the current working tab; draw.cpp updates global T reference.
    - Input focus remains consistent (all keystrokes now act on the new tab's buffers).
- **Rendering**
    - makeNavBar() is called to visually highlight the new active tab.
    - The main terminal area is redrawn with that tab's displayBuffer and input.
- **Thread Safety**
    - Tab switching occurs in the UI thread; no locks required on tabs.

---

5. Mouse Click — Add, Close, or Switch Tabs

Files & Locations

- run.cpp — handles ButtonPress events.
- draw.cpp — defines clickable tab regions and implements tab addition/removal (makeTabs() and addTab()).
- tabState — per-tab data management.

Functional Goal
Allow tab management using the mouse — click to switch tabs, add new tabs, or close existing ones.

Implementation Details

- **Event Capture**
  - XButtonEvent structures from X11 are processed in the event loop.
  - The x and y coordinates are compared with tab region boundaries computed in makeTabs().
- **Tab Switching**
  - If a click falls inside a tab's bounding box, tabActive is updated to that tab index.
  - The display is redrawn to bring that tab to the foreground.
- **Adding Tabs**
  - Clicking the "+" icon area (a region at the end of the navbar) triggers addTab(), which:
    - Instantiates a new tabState with default parameters.
    - Pushes it into the global tabs vector.
    - Sets it as the active tab.
  - Redraw is triggered via makeNavBar() and makeScreen().
- **Closing Tabs**
  - Clicking the "x" region on a tab calls closeTab(index) — implemented by erasing that entry from tabs.
  - If the closed tab was active, focus shifts to the previous or next available tab.
- **Rendering**
  - makeNavBar() draws clickable hitboxes (rectangles) for each tab, "+", and "x".

---

6. Blinking Cursor — Realistic Typing Experience

Files & Locations

- draw.cpp — cursor rendering within makeScreen().
- run.cpp — manages periodic blinking via timers or animation intervals.

Functional Goal
Provide a realistic, blinking caret at the current cursor position for improved typing feedback.

Implementation Details

- **Timer Mechanism**
    - The main event loop in run.cpp uses a timed interval (e.g., via select() timeout or periodic usleep) to toggle a boolean cursorVisible flag.
- **Rendering Logic**
    - In makeScreen(), the cursor's pixel position is calculated using:
        - cursor_x = currentCursorPosition * charWidth
        - cursor_y = baseY + currentLine * lineHeight
    - When cursorVisible == true, a filled rectangle (using XFillRectangle) is drawn at the cursor position.
    - When cursorVisible == false, the same region is redrawn with background colour to "hide" the cursor.
- **Thread Safety**
    - Cursor blinking occurs in the main UI thread (Xlib not thread-safe); all drawing remains synchronous.
- **Synchronization**
    - Input keystrokes reset the blink timer — ensures the cursor becomes visible immediately after typing, mimicking natural terminal behavior.

---

7. Backspace / Delete — Edit Text Inline

Files & Locations

- run.cpp — key event detection for XK_BackSpace and XK_Delete.
- draw.cpp — reflects input change on screen.
- tabState — stores editable input buffer.

Functional Goal
Allow inline editing by deleting characters before or after the cursor, with full Unicode support.

Implementation Details

- **Event Detection**
  - XK_BackSpace removes the character *before* the cursor.
  - XK_Delete removes the character *at* the cursor.
- **Buffer Update**
  - The handler modifies tabState.input:
    - Calculates UTF-8 character boundaries using mbrtowc() to safely remove multi-byte characters.
    - Updates currentCursorPosition accordingly.
  - Uses erase() operations on the string, ensuring no partial multibyte characters are removed.
- **Rendering**
  - After modification, the entire prompt line is re-rendered using makeScreen().
  - Caret position is updated to reflect the new buffer state.
- **Edge Handling**
  - If cursor is at start (currentCursorPosition == 0), backspace is ignored.
  - Delete at end of line has no effect.
- **Performance**
  - Since the input length is bounded (typically short commands), operations are O(n) and negligible.

---

8. Coloured Output for Errors, Search, and Autocomplete

Files & Locations

- draw.cpp — manages coloured text rendering.
- exec.cpp — error reporting (injects tagged output strings).
- helper_funcs.cpp — search and autocomplete results formatting.
- headers.cpp — defines ANSI colour macros or RGB values for Xlib GC (Graphics Context).

Functional Goal
Visually distinguish different types of output using colour coding:

- Red for errors and failed executions
- Cyan/Yellow for search matches and suggestions
- Green for autocomplete results

Implementation Details

- **Colour Configuration**
    - headers.cpp defines colour constants (RGB or X11 pixel values) for standard colour roles.
    - Each colour corresponds to a dedicated GC instance configured via XAllocColour() and stored globally.
- **Output Classification**
    - exec.cpp appends output lines into displayBuffer with type tags (ERR, SEARCH, RECOMMEND, or default).
    - The tag determines which GC colour is used during rendering.
- **Rendering Pipeline**
    - In makeScreen() and makeNavBar(), before calling XDrawString(), the code sets the GC's foreground colour based on the line type.
    - Example:
        - ERR → Red
        - SEARCH → Yellow.
        - RECOMMEND → Yellow.
    - The colour is reverted to default after drawing that line.
- **Search Result Highlighting**
    - For Ctrl+R matches, substring matches are coloured differently by calculating match start/end offsets and drawing them separately.
- **Autocomplete Colouring**
    - When multiple suggestions are listed, their numbers are drawn in one colour (e.g., yellow), and filenames in another (e.g., white).

- **Error Display**
    - execCommand() and related functions append red-coloured lines when a fork, open, or exec failure occurs. These are drawn using the error GC.