- *A terminal UI + React library with a focus on events, keymaps, and navigation. Forked from Ink.*

```
npm install tuir
```

Tuir

Manages:

- dynamically sized lists
- pages
- 1d and 2d navigation
- keymaps
- text input
- z-indexes
- modals
- in-app command lines
- click events
- styling options such as background colors and titles

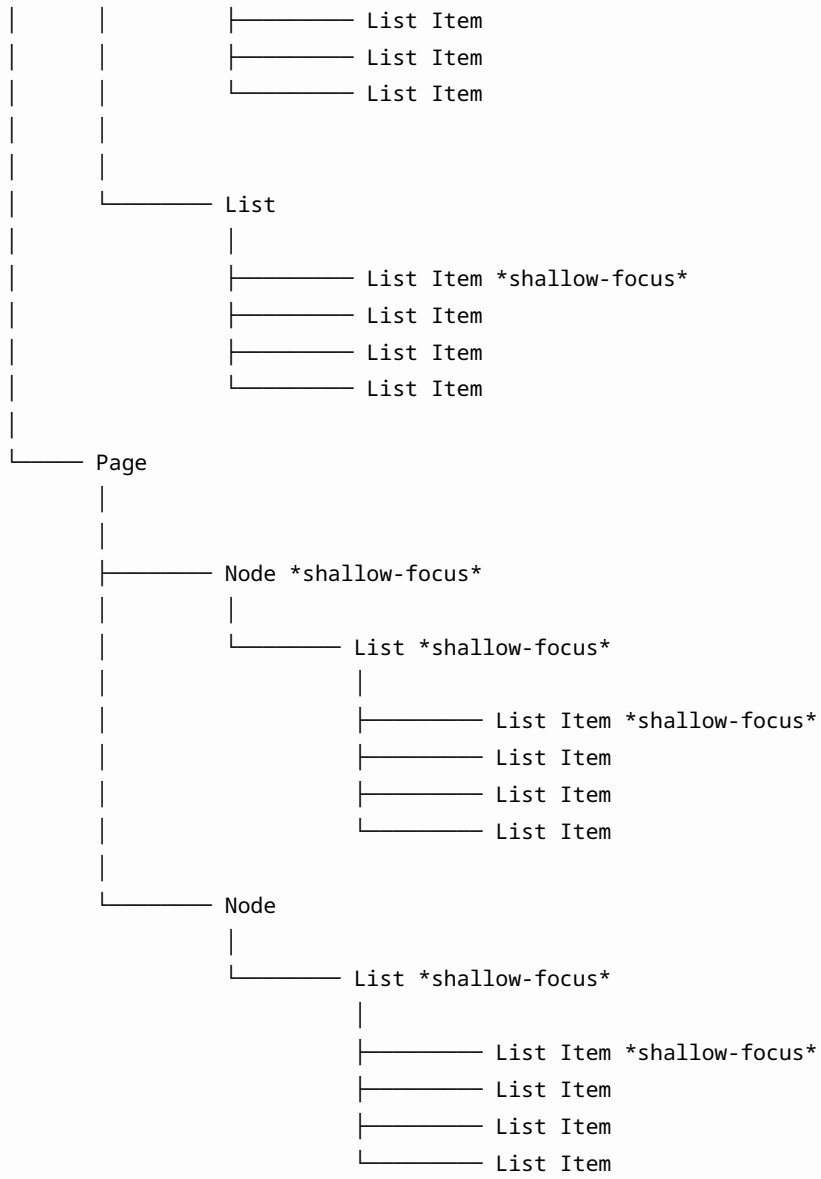
Overview

This document won't go into detail about the features available from Ink, such as `Box` and `Text` components. You can read the Ink docs to get familizarized with those features as well as basic setup. Only features in this library that differ will be detailed here.

Focus

Focus is derived from the context provided by `Page`, `List`, `Node`, and `Modal` components. A component that has no context from any of these providers is assumed to be in focus, whereas if context is available, the component is considered focused only if all of its providers are also focused. Shallow focus exists for leaf nodes that are focused relative to a parent that is not deeply focused.

```
Root *focus by default*
│
Pages *focus*
│
├──── Page *focus*
│      │
│      ├──────── List *focus*
│      │         │
│      │         ├────────── List Item *focus*
```



Events

Events can be emitted with the `useKeymap` hook and responded to anywhere in the app with the `useEvent` hook. Both of these hooks are only active if the component dispatching them is deeply focused. On every standard input event, every active useKeymap hook processes its `KeyMap` object and will emit an event if there is a match. Once an event has been emitted, all hooks are blocked from processing until the next standard input event.

```
const { useEvent } = useKeymap({
    foo: { input: "f" },
    bar: { input: "b" },
});
useEvent("foo", () => {
```

```
    /* ... */
});
useEvent("bar", () => {
    /* ... */
});
```

Navigation

`Pages`, `Lists`, `Node`, and `Modal` components display the state of focus in the app. Each of these core components has a corresponding hook that manages the focus state and returns utilities that help control navigation.

- *Pages / usePages*

```
const { pageView, control } = usePages(3);

const { useEvent } = useKeymap({
    goToPage: [
        { input: "1" },
        { input: "2" },
        { input: "3" },
    ]
});

useEvent("goToPage", (char) => {
    const pageNumber = Number(char);
    if (!Number.isNaN) {
        control.goToPage(pageNumber);
    }
});

return (
    <Viewport>
        <Pages pageView={pageView}>
        <PageOne />
        <PageTwo />
        <PageThree />
    </Viewport>
);
```

- *Lists / useList* (1d navigation)

```
const { listView, items } = useList(["foo", "bar", "baz"], {
    navigation: "vi-vertical",
});

return  (
    <Box height="50" width="50" borderStyle="round">
```

```
        <List listView={listView}>
            {items.map(item => <Text key={item}></Text>)}
        </List>
    </Box>

);
```

- *Node / useNodeMap* (2d navigation)

```
const nodeMap = [
    ["1", "2"],
    ["3", "3"],
];

const { register } = useNodeMap(nodeMap, { navigation: "vi" });

return (
    <Box height="100" width="100" flexDirection="column">
        <Box height="100" width="100" flexDirection="row">
            <Node {...register("1")}>
                <NodeOne />
            </Node>
            <Node {...register("2")}>
                <NodeTwo />
            </Node>
        </Box>
        <Box height="100" width="100">
            <Node {...register("3")}>
                <NodeThree />
            </Node>
        </Box>
    </Box>
)
```

- *Modal / useModal*

```
const { modal, showModal } = useModal({
    show: { key: "?" }, // ? because maybe this is a help modal?
    hide: { key: "esc" },
});

return (
    <Viewport>
        <MainApp />
        <Modal
            modal={modal}
            height="50"
            width="50"
            borderStyle="round"
```

```
            justifySelf="center"
            alignSelf="center"
        >
            { /* modal content */ }
        </Modal>
    </Viewport>
)
```

Settings

setCharRegisterSize: (num: number): void

By default, input stores up to 2 non-alphanumeric chars before resetting to an empty string. Once an event is emitted, the register resets again. If this default size of 2 is not desired, you can set it with the `setCharRegisterSize` size which accepts any positive number as an argument. The `StdinState` component offers visual feedback for the char register and events that are emitted.

setMouseReporting(b: boolean): void

This function tells the app to listen for mouse input events. Set to `false` by default, this should only be set if the app is using the `Viewport` component, as mouse events are receieved relative to the **entire** terminal screen, not just the dimensions of the app.

preserveScreen(): void

Call this function before your app renders anything and when your app closes it will return your terminal screen to the original state before the app started.

```
preserveScreen();
render(<App />)
```

Throttle rendering

Throttle renders to at most once every *throttleMs*

```
render(<App />, { throttle: 8 })
```

setConsole({ enabled: boolean; path: string }): void

Call this function to configure the behavior of console.log statements in the app. This can be helpful if you are using the `Viewport` component, which will obscure any stdout coming from the nodejs console object.

```
setConsole({ enabled: true, path: "console.log" });
```

The above example sends all output from the console to a file named “console.log”. You can then watch that file with something like `tail --follow console.log`.

Logger

Log to a specified file. Allows you to have multiple log files instead of just the specified path in setConsole.

```
const logger = new Logger();
logger.file("server.log").color("green").write("[RESPONSE]: 200");
```

`file` can be also be set with `logger.setFile(file)`

Lists

```
const initGroceryList = [
    "Bananas", "Eggs", "Milk", "Bread", "Chicken breast",
    "Spinach", "Butter", "Rice", "Cheese", "Cereal"
];

const { listView, items } = useList(initGroceryList, {
    windowSize: "fit",
    unitSize: 1,
    navigation: "vi-vertical",
    centerScroll: false,
    fallthrough: false,
});

return (
    <Box
        height={5}
        width={20}
        borderStyle="round"
        titleTopCenter={{ title: " Groceries " }}
    >
        <List listView={listView}>
            {items.map((item) => (
                <Item key={item} />
            ))}
        </List>
    </Box>
);

function Item(): React.ReactNode {
```

```
    const { isFocus, item } = useListItem<string[]>();
    const color = isFocus ? "blue" : undefined;
    return (
        <Box width="100" backgroundColor={color}>
            <Text wrap="truncate-end">{item}</Text>
        </Box>
    );
}
```

https://github.com/user-attachments/assets/a0d684c2-1023-4149-84f6-fdd7b187785b

useList(itemsOrLength, Options): { listView, control, items, setItems }

itemsOrLength: any[] | number

- If an array argument is provided, useList will manage state for you. A number argument states how many items will be included in the list.

Options:

- `windowSize: "fit" | number`
  - Default: 'fit'
  - `'fit'`: Maximizes the number of list items displayed within the available cross-sectional space, based on the `unitSize` value. When windowSize is set to 'fit', unitSize defaults to 1.
  - `number`: Sets the window size explicitly, up to the lesser of the provided number and the total number of list items. This option is most effective when paired with a unitSize of `stretch`. unitSize defaults to 'stretch' when windowSize is set to a number.
- `unitSize: number | 'stretch' | 'fit-unit'`
  - Default: Defaults to 1 when windowSize is 'fit'. Defaults to 'stretch' when windowSize is a number.
  - `number`: Assumes a fixed size for each list item's cross-sectional dimension. Displays as many items as possible based on this size.
  - `'stretch'`: Dynamically adjusts the size of the list items to fit within the available space. If windowSize is 'fit', displays as many possible items as possible and adjusts the size to shrink dynamically, down to a minimum unitSize of 1.
    - `'fit-unit'`: When rendering the list item, the dimensions of the item's container are the size of the list item itself. This might be helpful in the case you wanted a list with a window

size of 1, but each item to have a different size. Otherwise, the container is stretched to fill available space.

- `navigation: 'none' | 'vi-vertical' | 'vi-horizontal' | 'arrow-vertical' | 'arrow-horizontal'`
  - Default: 'vi-vertical'
  - `'none'`: No keymaps will control the list for you. You can control navigation in the list with the `control` object returned from useList along with `useEvent`/`useKeymap`.
  - `'vi-vertical'`: 'j': down, 'k': up, 'ctrl+d': scroll down half window, 'ctrl+u': scroll up half window, 'gg': go to first index, 'G': go to last index, arrow keys also supported.
  - `'vi-horizontal'`: 'h': left, 'l': right, arrow keys also supported.
  - `'arrow-vertical'`: 'up': up, 'down': down.
  - `'arrow-horizontal'`: 'left': left, 'right': right.
  - `*NOTE:*` If a component contains more than one List and they both contain default navigation keymaps, behavior will be unpredictable.
- `centerScroll: boolean`
  - Default: 'false'
  - Keeps the focused item in the center of the visible window slice when possible. This could be useful when a scrollbar isn't desired because it provides feedback when at the start/end of lists because the focused item will shift out of center.

https://github.com/user-attachments/assets/ec816010-4d9f-46bc-a043-2f132569be03

- `fallthrough: boolean`
  - Default: 'false'
  - Controls navigation behavior at list boundaries. If `true`, when focus reaches the end of the list, wraps around to the opposite end. If `false`, focus stops at the list boundaries and cannot move further.

https://github.com/user-attachments/assets/869bae25-8cd5-43cc-bbac-0d8f9c6312e3

Return: { listView, control, items, setItems }

- `listView`: Required prop for List component.
- `control`: Utilities for controlling the list:
  - `currentIndex: number`: The current index that is focused.
  - `goToIndex(nextIdx: number, center?: boolean): void`: Shifts focus to a given index. If `center` is true, center the focus if possible.

- `nextItem(): void`: Shift focus to the next list item.
- `prevItem(): void`: Shift focus to the previous list item.
- `scrollDown(n?: number): void`: Shifts focus down `n` items. If a number is not provided, shifts focus down half of the viewing slice.
- `scrollUp(n?: number): void`: Opposite of scrollDown.

- `items, setItems`:
  - If an array was provided, these are the state related variables managed internally by `useList`.
  - If an array was not provided, items defaults to a dummy array of `null`.

List

The component responsible for displaying the state of the list. The List component has a height and width of 100% of its parent container, so it needs a parent container to inherit dimensions from. If a column List has a height of 0, then no list items will be rendered.

Required Props:

- `listView`: The object received from useList

Optional Props

- `flexDirection: 'row' | 'column'`: 'column' renders the list items vertically, while 'row' renders the list items horizontally
  - Default: 'column'
- `scrollbar`: Value is a config object that styles the scrollbar
  - `hide?: boolean`: Show or hide the scrollbar
    - Default: 'false'
  - `color?: string`: Color of the scrollbar
    - Default: 'undefined'
  - `dimColor?: boolean`:
    - Default: `false`
  - `style?: 'single' | 'bold' : { char: string }`: The thickness of the scrollbar. Add your own style with the config object option
  - `align?: 'start' | 'end'` 'start' aligns on the left and top of column and row lists respectively and 'end' the right and bottom. - Default: `end`
- `justifyContent: 'flex-start' | 'center' | flex-end' | 'space-between' | 'space-around'`: Control how the list items are displayed within the viewing window.
  - Default: 'flex-start'
- `alignItems: 'flex-start' | 'center' | 'flex-end' | 'stretch'`: Control how the list items are displayed within the viewing

window. - Default: ‘flex-start’

- `gap: number`: The `gap` between list items.
  - Default: ‘0’
- `batchMap`: `batchMap?: { batchSize?: number; items: T[]; map: (item: T, index: number) => ReactNode; };`
  - Default `batchSize`: 250

batchMap

If performance becomes an issue with *excessively* large lists, render list items within the `map` callback *instead of* how you would normally list render something within the JSX. This renders only the `batchSize` of items.

```
batchMap={{
    batchSize: 100,
    items: myLongList as Foo[],
    map: (item, index) => {
        return <Item key={item} item={item} />;
    },
}}
```

Note: In the example, any state local to the `Item` component is lost once it goes out of the viewing window. This is only the case for `batchMap`. List items rendered normally always stay mounted and therefore maintain state.

---

useListItem(): { item, items, setItems, onFocus, onBlur, itemIndex, listIndex, control, isFocus, isShallowFocus }

Returns data about the list-rendered component and the list.

Generic Argument:

- `<T>`: Represents the type of the items array.
  - Default: ‘any[]’

Properties:

- `item`: If the value passed to useList was a number representing the length, this will be `null`, otherwise it will be the corresponding value in the ‘items’ array for this list item.
- `items`: The array responsible for rendering the list items. If the value passed to useList was a number representing the length, this will be an array of `null`.
- `setItems`: State action for updating the list items. If the `useList` hook is managing the state of the provided array, this is the

setState function.

- `onFocus`: A function that accepts a callback that is executed every time this component comes into deep focus.
- `onBlur`: A function that accepts a callback that is executed every time this component becomes unfocused.
- `itemIndex`: The index of this node within the 'items' array.
- `listIndex`: The currently focused index in the 'items' array.
- `isFocus`: 'true' if the index of this node matches the currently focused index in the list *and* the List itself is also focused.
- `isShallowFocus`: 'true' if the node is focused relative to the List, but the List itself is not focused.
- `control`: The `control` object containing utilities for controlling the list.

List: windowSize and unitSize examples

Fixed windowSize with unitSize of 'stretch'.

`Note`: When windowSize is a `number`, unitSize will default to `stretch`.

```
const { listView, items, control } = useList(initGroceryList, {
    windowSize: 2,
    unitSize: "stretch",
});

return (
    <Box
        height={8}
        width={20}
        borderStyle="round"
        titleTopCenter={{ title: " Groceries " }}
    >
        <List listView={listView}>
            {items.map((item) => {
                return <Item key={item} />;
            })}
        </List>
    </Box>
);

function Item(): React.ReactNode {
    const { isFocus, item } = useListItem<string[]>();
    const color = isFocus ? "blue" : undefined;
    return (
        <Box width="100" height="100" backgroundColor={color}>
            <Text wrap="truncate-end">{item}</Text>
        </Box>
```

```
    );
}
```

https://github.com/user-attachments/assets/55204fc7-914a-4f7e-a2d0-af3cd99f54b2

'fit' windowSize with fixed unitSize.

`Note`: When windowSize is set to `fit`, unitSize defaults to `1`

If unitSize is a fixed number, then all list items are assumed to have that dimension. Excess height could lead to improper fit. This is why the wrap property is set to 'truncate-end' in the Text component.

```
const { listView, items } = useList(initGroceryList, {
    windowSize: "fit",
    unitSize: 3,
});

return (
    <Box
        height={11}
        width={20}
        borderStyle="round"
        titleTopCenter={{ title: " Groceries " }}
    >
        <List listView={listView}>
            {items.map((item) => {
                return <Item key={item} />;
            })}
        </List>
    </Box>
);

function Item(): React.ReactNode {
    const { isFocus, item } = useListItem<string[]>();
    const color = isFocus ? "blue" : undefined;
    return (
        <Box
            width="100"
            borderStyle="round"
            backgroundColor={color}
            borderColor={color}
        >
            <Text wrap="truncate-end">{item}</Text>
        </Box>
    );
}
```

https://github.com/user-attachments/assets/e5eb2f04-cc96-4d26-98fc-03f5fbab4457

---

useKeymap, useEvent, useInput

This example continues on from the `List` example.

`Note`: useEvent can also be imported, but you lose autocomplete unless a generic argument is provided. The generic must satisfy the `KeyMap` type.

```
function Item(): React.ReactNode {
    const { isFocus, item, items, setItems, index } =
        useListItem<string[]>();

    const [checked, setChecked] = useState(false);

    const { useEvent } = useKeymap({
        toggleChecked: { key: "return" },
        deleteItem: { input: "dd" },
    });

    useEvent("toggleChecked", () => {
        setChecked(!checked);
    });

    useEvent("deleteItem", () => {
        const itemsCopy = items.slice();
        itemsCopy.splice(index, 1);
        setItems(itemsCopy);
    });

    const color = isFocus ? "blue" : undefined;
    return (
        <Box width="100" backgroundColor={color}>
            <Text wrap="truncate-end">{`${item} ${checked ? "✔" : "
        "}`}</Text>
        </Box>
    );
}
```

All events are derived from the same standard input stream. Once an event has been matched, it is emitted and all other useKeymap hooks are blocked until the next standard input event.

---

useKeymap(KeyMap, Opts): { useEvent }

```
const keymap = {
```

```
    quit: { input: "q" },
    foobar: { input: "fb" },
} satisfies KeyMap;

useKeymap(keymap);
```

If the component is focused, useKeymap processes its KeyMap and emits events with `node:events`. Any `useEvent` hook that is focused within the app can respond to these events.

Beyond handling localized keymaps and events, useKeymap can be placed at the top level of the app and provide events that are relevant at different scopes.

KeyMap: { [eventName: string]: KeyInput | KeyInput[] }

The keys in the KeyMap object are the names of the events that will be emitted when standard input matches the KeyInput.

- `KeyInput`: { key?: Key; input?: string; notKey?: Key[]; notInput?: string[] }
  - `Key`: Non alphanumeric keys.
  - `input`: Any combination of alphanumeric keys.
  - `notKey`: Any input *except* the Keys in this array will trigger this event.
  - `notInput`: Any input *except* the strings in this array will trigger this event.

Opts

- `priority`: `never` | `always` | `default` | `override` | `textinput`
  - Default: 'default'
  - Sets priority levels for useKeymap hooks so that control can be passed between different hook instances. This is necessary for operations such as text input.
  - `always` and `never` do not interfere with other priority levels
  - `override` overrides anything set to default
  - `textinput` overrides everything including `always`

---

useEvent(eventName: keyof T, handler: (char: char) => unknown, additionalFocusCheck?: boolean = true): void

```
useEvent<typeof keymap>("foo", () => {
    /* do something */
});
```

- `eventName`: The event derived from the keys of the `KeyMap` object
- `handler`: The last keypress that triggered the event.

- `additionalFocusCheck`: When ‘false’, the specific hook will not respond to any events.
  - Default: ‘true’

---

useTypedEvent(): { useEvent }

useEvent can be returned from the useKeymap hook, or it can be imported. If imported, you need to provide the generic argument to receive autocomplete. useTypedEvent returns a type-safe version of useEvent for better autocomplete.

```
const keymap = { foo: { input: "f" } } satisfies KeyMap;
/* ... */
const { useEvent } = useTypedEvent<typeof keymap>();
```

---

useInput(handler: (input: string, key: Key) => unknown, opts?: Opts): void

```
useInput(
    (input, key) => {
        if (key.return) {
            console.log("'Return' pressed");
        }
        if (input === "f") {
            console.log("'f' pressed");
        }
    },
    { isActive: true, inputType: "char" },
);
```

Disregards focus and responds to standard input anywhere and anytime this hook is dispatched. Functions the same as the `useInput` hook in Ink. This is useful if you want to escape the focus dependent response to standard input. Its also exists as a less boilerplate option when focus isn’t as much of a concern.

- **`isActive: boolean`**: Control whether or not the hooks read from standard input
- **`inputType: 'char' | 'register'`**: Controls what kind of `input` is recieved
  - Default: ‘char’
  - `char`: Check input against single chars
  - `register`: Check input against however many chars are in the `char register`.

Pages

Pages are just Lists with a windowSize of 1 and unitSize of ‘stretch’. There are no default keymaps to control navigation of pages. If you want your app to have different pages, you might also want to use the `Viewport` component which is just a `Box` that uses the dimensions of the entire terminal screen.

```
const pagesKeymap = {
    goToPage: [
        { input: "1" },
        { input: "2" }
    ],
    quit: { input: "q" },
} satisfies KeyMap;

export default function Docs(): React.ReactNode {
    const { pageView, control } = usePages(2);

    const { useEvent } = useKeymap(pagesKeymap);

    useEvent("goToPage", (char: string) => {
        const pageIndex = Number(char);

        if (!Number.isNaN(pageIndex)) {
            control.goToPage(pageIndex - 1);
        }
    });

    return (
        <Viewport>
            <Pages pageView={pageView}>
                <ListDemo />
                <PageTwo />
            </Pages>
        </Viewport>
    );
}
```

https://github.com/user-attachments/assets/68700c6f-6a7f-47fc-945e-d76c887c42f8

usePages(amountOfPages, Opts?): { pageView, control }

`amountOfPages: number`: The number of pages must match how many pages are rendered.

`Opts`: { fallthrough: boolean }

- Default: `false`

- The same as in the `useList` hook. If at the last page and `control.lastPage()` is executed, focus is shifted to the first page if is this option is true.

`pageView`: Required prop for `Pages` component.

`control`: Utilities for controlling the Pages component.

- `currentPage: number`: the current index that is focused.
- `goToPage(num: number): void`: Go to a page with specified index.
- `nextPage(): void`: Shift focus to the next page.
- `prevPage(): void`: Shift focus to the previous page.

Pages (component)

Props:

`pageView`: The object received from `usePages`.

usePage(): { control, isFocus, isShallowFocus, onPageFocus, onPageBlur }

- `control`: Utilities for controlling the `Pages` component. The same object received from usePages (plural).
- `isFocus: boolean / isShallowFocus: boolean`: Is this page focused/shallow focused?
- `onPageFocus`: A function that accepts a callback that is executed every time this page *gains* focus.
- `onPageBlur`: A function that accepts a callback that is executed every time this page *loses* focus.

2d Navigation: Node / useNodeMap

`useNodeMap` is a utility for managing 2-dimensional navigation. It maps nodes to specific coordinates which enables intuitive navigation. Like Pages and Lists, Nodes also provide focus context.

```
const nodeMap = [
    ["A", "B"],
    ["C", "D"],
    ["", "E"],
] as const satisfies NodeMap;

function NodeMapDemo(): React.ReactNode {
    const { register, control } = useNodeMap(nodeMap, { navigation:
        "vi" });

    const { useEvent } = useKeymap({
```

```
    goToNode: [
        { input: "A" },
        { input: "B" },
        { input: "C" },
        { input: "D" },
        { input: "E" },
    ],
    next: { key: "tab" },
});

useEvent("goToNode", (char: string) => {
    control.goToNode(char);
});

useEvent("next", () => {
    control.next();
});

const styles: Styles["Box"] = {
    height: "100",
    width: "100",
    flexDirection: "row",
};

return (
    <Viewport flexDirection="column">
        <Box styles={styles}>
            <Node {...register("A")}>
                <Square />
            </Node>
            <Node {...register("B")}>
                <Square />
            </Node>
        </Box>
        <Box styles={styles}>
            <Node {...register("C")}>
                <Square />
            </Node>
            <Node {...register("D")}>
                <Square />
            </Node>
        </Box>
        <Box styles={styles}>
            <Box height="100" width="100"></Box>
            <Node {...register("E")}>
                <Square />
            </Node>
        </Box>
```

```
        </Viewport>
    );
}

function Square(): React.ReactNode {
    const { isFocus, name } = useNode();

    const color = isFocus ? "blue" : undefined;

    return (
        <Box
            height="100"
            width="100"
            borderStyle="round"
            justifyContent="center"
            alignItems="center"
            borderColor={color}
            backgroundColor={color}
        >
            <Text>{name}</Text>
        </Box>
    );
}
```

https://github.com/user-attachments/assets/fb00a49e-fe0f-429e-8811-119e9315c952

useNode(nodeMap: NodeMap, opts? Opts): { register, control, nodesView, node }

NodeMap: This is the 2d string array that navigation is derived from. An empty string is considered a `null` space in the `NodeMap`.

Maps do not need to follow a strict width. This is valid.

```
const nodemap1 = [["1"], ["2", "3", "4"], ["5"]];
```

Nodes can extend cells. This is also valid.

```
const nodemap2 = [
    ["1", "2"],
    ["1", "3"],
    ["1", "4"],
];
```

Opts: { initialFocus?: string; navigation?: 'vi' | 'arrow' | 'none' }

- `initialFocus`: The name of the node that you want focused initially. If not provided, the first non-empty string is used.
- `navigation`: The default navigation keymaps provided.

- Default: `vi`
- `vi`: h,j,k,l and also arrow keys
- `arrow`: arrow keys
- `none`: Handle yourself with the `control` object

Return: { register, control, nodesView, node }

- `register(nodeName: string)`: The function used to register a `Node` component to the `NodeMap`. Not strictly necessary to use this function, but convenient. It returns `{ name: string; nodesView: NodeView }` which are required props for the `Node` component
- `nodesView`: Required prop for the `Node` component.
- `node`: The name of the current node that is focused.
- `control`: Utilities for controlling the state of the map.
  - `next(): string`: Go to the next available node. This is like pressing tab in the browser when filling out a form. Returns the name of the focused node after the operation.
  - `prev(): string`: Opposite of `next`
  - `goToNode(node: string | number): string`: Shift focus directly to the provided node. If argument is a string, this should correlate to one of the nodes in the NodeMap. If argument is a number, this is the nth node in the map, if you were to count starting at the top left corner and ending at the bottom right corner.
  - `left(): string`: Shift focus left. Returns the name of the focused node after the operation after the operation.
  - `right(): string`: Shift focus right. Returns the name of the focused node after the operation after the operation.
  - `up(): string`: Shift focus up. Returns the name of the focused node after the operation after the operation.
  - `down(): string`: Shift focus down. Returns the name of the focused node after the operation after the operation.
  - `getSize(): number`: Returns the count of all the nodes.
  - `getLocation(): string`: The name of the currently focused node.
  - `getIteration(): number`: The *index* of the currently focused node.

Node

Passes the state of the `NodeMap` to its child components, which includes the focus status of the node with the given name.

```
<Node nodesView={nodesView} name={"foo"}>
    <FooComponent />
</Node>
```

or

```
<Node {...register("foo")}>
    <FooComponent />
```

```
</Node>
```

Props

`Required:`

- `nodesView`: The object returned from useNodeMap
- `name`: The name of the node.

`Node.Box`

Because it might be helpful or might lead to less jsx nesting in certain situations, the `Node` component can be extended to include all of the props as a `Box` component.

```
<Node.Box nodesView={nodesView} name={"foo"} >
    <FooComponent />
</Node.Box>
```

useNode(): { name, isFocus, isShallowFocus, control}

Returns context about the `Node` we are rendering.

- `name`: The name of the Node we are rendering.
- `isFocus`: If the current Node is focused *and* the component containing the `useNodeMap` hook is also focused.
- `isShallowFocus`: If the current Node is focused but the component containing the `useNodeMap` hook is *not* focused.
- `control`: The same control object return from `useNodeMap`.

Modals

```
function ModalDemo(): React.ReactNode {
    const { modal } = useModal({
        show: { input: "m" },
        hide: { key: "esc" },
    });

    return (
        <>
            <Modal
                modal={modal}
                closeOnOutsideClick={true}
                justifySelf="center"
                alignSelf="center"

                // Normal Box props
                height="75"
```

```
                width="75"
                borderStyle="round"
                flexDirection="column"
                titleTopCenter={{ title: " Modal Demo " }}
            >
                <ModalContents />
            </Modal>
        </>
    );
}

function ModalContents(): React.ReactNode {
    const { hide } = useHideModal();

    return (
        <>
            <Text>Modal contents...</Text>
            <Text>To close:</Text>
            <Text>- Press Esc</Text>
            <Text>- Click outside this modal</Text>
            <Text>- Click the close button</Text>
            <Box onClick={hide} borderStyle="round" width="10">
                <Text>close</Text>
            </Box>
        </>
    );
}
```

https://github.com/user-attachments/assets/507e3bc7-b036-45cc-9186-aebae9ae1f07

Modal components unfocus all components that are not themselves or their children. Modals have a default `zIndex` of 1, but zIndex has no effect on focus handling. A `Box` with a zIndex of 5 would still be unfocused when modal component pops up if that Box is not a child of the modal.

---

useModal(ToggleKeymaps): { modal, showModal, hideModal }

`ToggleKeymaps: { show: KeyInput | KeyInput[] | null; hide: KeyInput | KeyInput[] | null }`: The keymaps assigned to show and hide the modal. - If show or hide is assigned to `null`, then you'll need to show or hide the modal with `showModal` or `hideModal`.

- `modal`: Required prop in the Modal component
- `showModal(): void`: Method other than assigning keymap to show the modal.
- `hideModal(): void`: Method other than assigning keymap to hide the

modal.

useHideModal(): { hideModal }

Can be called within a Modal component to receive the `hideModal` function.

`Modal:`

- `Required Props:`
  - `modal`: Object returned from `useModal`
- `Props:`
  - `all of the props available to *Box* components are also available`
  - `justifySelf: 'flex-start | 'center' | flex-end`: Position the modal left/right within the parent element.
  - `alignSelf: 'flex-start' | 'center' | 'flex-end'`: Position the modal up/down within the parent element.
  - `xOffset: number / yOffset: number`: Fine tune the positioning relative to the parent element.

TextInput / useTextInput

```
function TextInputDemo(): React.ReactNode {
    const { onChange, setValue } = useTextInput("");

    const onExit = (value: string) => {
        /* do something with the value */
    };

    // Reset the value to '' every time we enter insert mode
    const onEnter = () => {
        setValue("");
    };

    return (
        <Box
            height="100"
            width="100"
            justifyContent="center"
            alignItems="center"
        >
            <Box borderStyle="round" width={10} height={3}>
                <TextInput
                    onChange={onChange}
                    enterKeymap={{ key: "return" }}
```

```
                    exitKeymap={{ key: "return" }}
                    onExit={onExit}
                    onEnter={onEnter}
                    textStyle={{
                        color: "magenta",
                    }}
                    cursorColor="green"
                    autoEnter
                />
            </Box>
        </Box>
    );
}
```

https://github.com/user-attachments/assets/2b88fefe-4436-42f4-b338-1357e86d8cbe

useTextInput(initialValue?: string): { onChange, value, setValue, insert, enterInsert }

- `onChange`: Function that is a required prop to the `TextInput` component.
- `value`: The current string value of the text.
- `setValue(nextValue: string, insert?: boolean)`: Utility function that updates the value to a specificed value
  - The optional `insert` defaults to not change whatever insert value it has currently, but when set explicitly, controls the insert value in the next state.
- `enterInsert()`: Utility function that puts you into insert mode.

TextInput

Required Props

- `onChange`: Function returned form `useTextInput`.

Optional Props

- `enterKeymap: KeyInput`:
  - Default: [{ key: “return” }, { input: “i” }]
- `exitKeymap: KeyInput`:
  - Default [{ key: “return” }, { key: “esc” }]
- `onExit(value: string, char: string)`: Allows you to do something with the value when exiting insert mode. - `char` is the key that triggered the TextInput to exit insert.
- `onEnter(value: string, char: string)`: Allows you to do something with the value when entering insert mode. - `char` is the key that triggered the TextInput to enter insert.

- `onKeypress(char: string)`: Called on every keypress event when inserting text.
- `onUpArrow() / onDownArrow()`: Control what happens when these keys are pressed while inserting text. Could be used to shift focus.
- `textStyles`: Style the displayed text
- `cursorColor`: Color the cursor block
- `autoEnter`: When focus is gained, does TextInput automatically enter insert mode?
  - Default: 'false'

Box Styling

styles? Styles["Box"]

- Default: `undefined`
- This is an object that contains all of the styles available to `Box` components *except* the `styles` prop. Useful in the case you have multiple boxes that need similar styles.

```
const styles: Styles["Box"] = {
    height: 10,
    width: 10,
    justifyContent: "center",
    alignItems: "center",
    borderStyle: "round",
    borderColor: "blue",
};

return <Box styles={styles} borderColor="green"></Box>;
```

The above Box has a green border. Normal props overwrite the props in the styles prop.

`Note`: Styles["Text"] exists for `Text` components as well.

---

backgroundColor?: string | 'inherit'

Colors the background color of a Box. When set to `inherit`, if the immediate parent has a backgroundColor prop set, it will inherit from the parent.

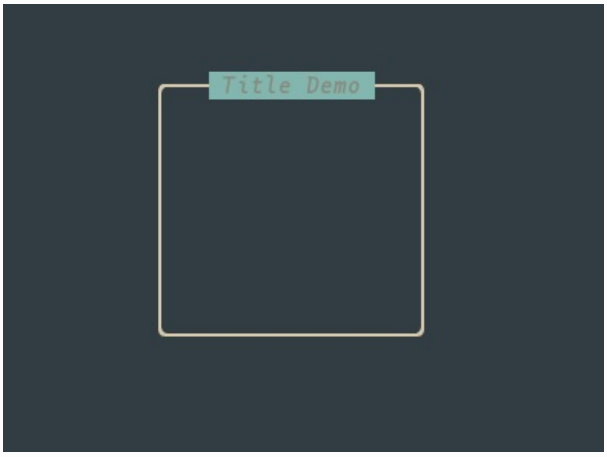
- Default: `undefined`

---

titleTopLeft, titleTopCenter, titleTopRight, titleBottomLeft, titleBottomCenter, titleBottomRight

Applies a title in the specified location. The only required prop is `title`. Nothing is styled unless explicitly set.

```
<Box
    height={10}
    width={20}
    borderStyle="round"
    titleTopCenter={{
        title: " Title Demo ",
        color: "white",
        backgroundColor: "blue",
        dimColor: false,
        inverse: false,
        bold: true,
        italic: true,
    }}
></Box>
```



title-demo

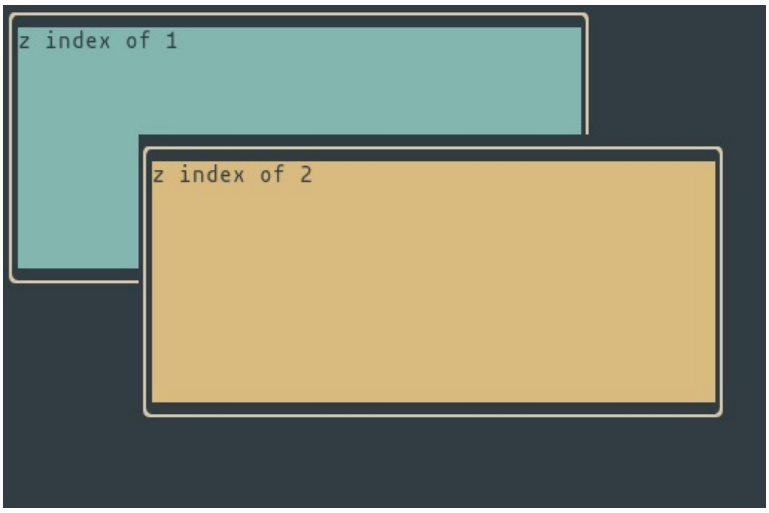
---

zIndex?: 'auto' | number

- Default: `'auto' (equivalent to 0)`
- Sets the render order of a Box component. `zIndex` is only relative to the parent node, so a zIndex of 1 when the parent Box has a zIndex of 1 is really a zIndex of 2.

```
<Box zIndex={1} styles={styles} backgroundColor="blue">
    <Text>z index of 1</Text>
</Box>
<Box
    zIndex={2}
    styles={styles}
    backgroundColor="yellow"
    marginLeft={10}
    marginTop={5}
```

```
>
    <Text>z index of 2</Text>
</Box>
```



zindex-demo

---

wipeBackground?: boolean

- Default: `false` Clears the background of a Box component. When a zIndex is set, this happens by default. Can be a drain on performance if this happens too often.

Mouse Events

Mouse event handlers can be attached to `Box` components and to any `titleTopLeft, titleTopCenter, etc...` prop object on a Box.

Requires manually telling the app to listen for mouse input events with the `setMouseReporting(b: boolean)` function.

Important:

- Mouse events are only accurate if the app is using the entire terminal screen, so if you aren't using the Viewport component, clicks will not be accurate.

All handlers accept a callback in the form of `(e: MouseEvent) => unknown`.

- `onClick`
- `onDoubleClick`
- `onMouseDown`
- `onMouseUp`

- `onRightClick`
- `onRightMouseDown`
- `onRightMouseUp`
- `onRightDoubleClick`
- `onScrollUp`
- `onScrollDown`
- `onScrollClick`

`MouseEvent: { clientX, clientY, target, targetPosition}`: Object that contains information about the click.

- `clientX`: The x coordinate relative to the entire screen where the mouse event occured.
- `clientY`: The y coordiante relative to the entire screen where the mouse event occured.
- `target`: The reference to the component's YogaNode.
- `targetPosition`: The corner positions of the component.
  - `{ topleft: [number, number]; topRight: [number, number]; bottomLeft: [number, number]; bottomRight: [number, number] }`

Box leftActive and rightActive

These are props for the `Box` component which allow you to apply different styles for when the left or right mouse is pressed down over the component.

```
<Box
    width={10}
    borderStyle="round"
    leftActive={{
        borderColor: "blue",
    }}
    onClick={() => setCount(count + 1)}
>
    <Text>{count}</Text>
</Box>
```

https://github.com/user-attachments/assets/eea00621-319b-463f-a361-4b47edf05e9c

---

Cli

Manage emitting events throughout the app with the `Cli` component.

```
const commands = {
    greet: (args) => {
```

```
        if (!args[0]) {
            return Promise.reject("Provide a name!");
        } else {
            return `Hello, ${args[0]}!`;
        }
    },
    shell: (args) => {
        return new Promise((res, rej) => {
            const [cmd, ...cmdArgs] = args;
            execFile(cmd, cmdArgs, (err, stdout, stderr) => {
                err && rej(err);
                res(stdout || stderr);
            });
        });
    },
    DEFAULT: (args) => {
        return Promise.reject(`Unknown Command: ${args[0] ?? ""}`);
    },
} satisfies Commands;

useCommand("greet", (args) => {
    /*
     * You can respond to commands with useCommand as well, but they
       won't
     * have any effect on inputStyles, resolveStyles, or
       rejectStyles like
     * the handlers in the Commands object does.
     */
})

return (
    <Viewport flexDirection="column">
        <Box height="100" width="100"></Box>
        <Cli
            commands={commands}
            prompt={"~ > "}
            inputStyles={{
                italic: true,
                color: "magenta",
            }}
            rejectStyles={{
                color: "red",
                italic: true,
                dimColor: true,
            }}
            resolveStyles={{
                color: "green",
                italic: true,
                dimColor: true,
```

```
            }}
        />
    </Viewport>
);
```

https://github.com/user-attachments/assets/9d9d9f4c-3577-4f3c-afad-9bd54a0af64f

`Commands: { [command: string | 'DEFAULT']: CommandHandler }`
`CommandHandler: (args: string[], rawinput: string): string | Promise<unknown>`

- The `args` array is an array of all the words written to the cli `*after*` the command. The first word, the command itself, is omitted from the args array.
- The `rawinput` string is the raw input of everything *after* the command was written to input.
- `CommandHandler`: When this returns a string or a Promise, this sets the value of the Cli component. If a rejected Promise is returned, the rejectedStyles will be applied, and a resolved Promise, the resolvedStyles.
- The `DEFAULT` handler is a special handler that executes its callback when input does not match any command. Unlike the other handlers, `args` is an array of *all* the words written to the cli.
- When a handler returns a string or number value (either directly or as the result of a Promise), that value is displayed as the CLI output after the command finishes.

`Controls`

- `:` opens the `Cli`
- `return`: executes the command
- `esc` closes the `Cli`
- `up arrow / down arrow`: Cycle through the history of commands entered within the current session.

If the handler returns nothing, the Cli is closed automatically.

`Required Props:`

- `commands:`: The `Commands` object.

`Optional Props:`

- `prompt`: The prompt visible when entering a command
  - Default: ":"
- `inputStyles, resolveStyles, rejectStyles`: Style the text differently based on the different scenarios.
- `persistPrompt: boolean`: When the Cli in inactive should the prompt be visible?
  - Default: `false`

- `actionPrompt`: See section on setting prompts and messages
- `message`: See section on setting prompts and messages

useCommand(commandName | 'DEFAULT', CommandHandler): void

The equivalent of `useEvent` but for events emitted from the Cli. Just like useEvent, useCommand becomes inactive when the component dispatching it is not focused.

Setting Prompts and Messages

The Cli component can accept an `actionPrompt` and `message` props which are both tuples. The `actionPrompt` prop is there to prompt users for input, and respond to that input, while the `message` prop sets the value of the Cli. Both of these take effect when their references change, so the values for these props should be state variables and should be initialized to undefined.

`actionPrompt: [string, (args: string[], rawinput: string) => Promise<unknown>]`

- The first index is the prompt that is shown when asking for user input.
- The second index is the callback that is executed when the input is received. Similar to the callbacks in the `Commands` object, the returned values set the value of the Cli component until the next stdin event, and style according to whether or not the callback rejected or resolved.

`message: ["INPUT" | "RESOLVE" | "REJECT", string]`

- The first index says how to style this message.
- The second index is the message itself

The message will be shown until the next stdin event.

CliModal

The `Cli` component but as a pop up modal. Accepts the same props as the `Modal` component and `Cli` component, with the exception of the `actionPrompt` and `message` props from the Cli component.

```
<CliModal
    // Cli props
    commands={commands}
    prompt={"~ > "}
    persistPrompt
    inputStyles={{
        italic: true,
```

```
        color: "magenta",
    }}
    rejectStyles={{
        color: "red",
        italic: true,
        dimColor: true,
    }}
    resolveStyles={{
        color: "green",
        italic: true,
        dimColor: true,
    }}

    // Modal props
    borderStyle="round"
    width="75"
    titleTopCenter={{ title: " Command Line ", color: "green" }}
    borderColor="green"
/>
```

https://github.com/user-attachments/assets/9756e56f-0106-489a-a3b9-cf959b4dd178

Viewport Component

A `Box` component that is set to the same dimensions as the terminal screen.

Props

All of the `Box` properties are available except for `height` | `width` | `minHeight` | `minWidth` | `alignSelf`.

```
return (
    <Viewport justifyContent="center" alignItems="center">
        <Text>Hello World!</Text>
    </Viewport>
);
```

StdinState Component

```
useKeymap({
    foo: { input: "F" },
    bar: { input: "B" },
    foobar: { input: "fb" },
});
```

```
/* ... */

return <StdinState
        showEvents={true}
        showRegister={true}
        eventStyles={{
            color: "green",
        }}
        registerStyles={{
            color: "blue",
        }}
        width={25}
    />
```

https://github.com/user-attachments/assets/b6f39a00-85a5-495b-b473-c3bbe532a283

The `StdinState` component keeps track of stdin and emitted events and displays them. This is useful because of the visual feedback it gives. For example, if pressing 'a' doesn't trigger an event, it could be because the char register was 'ja' at the time.

Optional Props:

- `showEvents: boolean`: Display event names when they are emitted?
  - Default: 'true'
- `showRegister: boolean`: Show keypresses in the char register?
- `eventStyles`: Value is an object that, if provided, styles the text of the displayed events.
- `registerStyles`: Value is an object that, if provided, styles the text of the displayed keypresses.
- `width: number`: How much space the component takes up. When an event is too long to fit inside this number, it is truncated.
  - Default: 20

---

useShellCommand(): { exec }

- `Should be used along with the *preserveScreen* function` This hook returns a function called `exec` that runs a shell command and temporarily leaves the app to output the results.

- `exec(command: string, reattachedMessage?: string)`

  - The exec function returns a Promise with the exit status which is either a `number` or `null` based on whether or not the spawned command provides an exit status.
  - `reattachedMessage` is the text that is displayed in the terminal when the command is finished executing, prompting you to re-

attach to the app.
    - Default: `press any key to continue`

```
const { exec } = useShellCommand();

const commands = {
    exec: async (args) => {
        const command = args.join(" ");
        const exitStatus = await exec(command);
        if (exitStatus !== 0) {
            return Promise.reject(`Running shell command: exit
        status ${exitStatus}`);
        }
    },
    DEFAULT: (args) => {
        return Promise.reject(`Unknown Command: ${args[0] ?? ""}`);
    },
} satisfies Commands;

/* implement Cli component */
```

https://github.com/user-attachments/assets/51b244c6-1a16-4231-a488-2fbdb59a2c22

VerticalLine / HorizontalLine Components

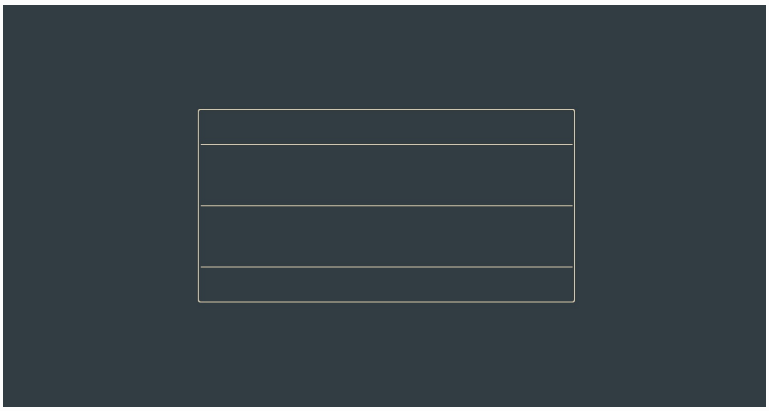
`Optional Props:`

- `length: string | number`: The height or width or the VerticalLine or Horizontal line respectively.
    - Default: `'100%'`
- `color`: The color of the line.
- `bold`: Increase thickness of the line.
- `dimColor`: Dim the color of the line.
- `char`: Build the line with a character of your own choosing.

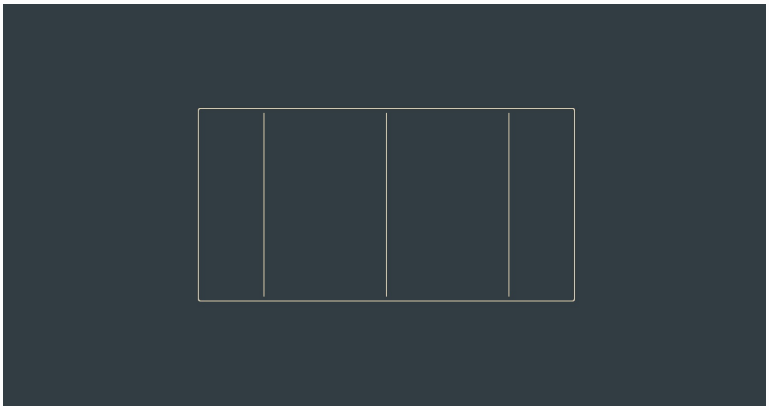
```
return <Box
        width="50"
        height="50"
        borderStyle="round"
        flexDirection="column"
        justifyContent="space-around"
    >
        <HorizontalLine />
        <HorizontalLine />
        <HorizontalLine />
    </Box>

return <Box
```

```
    width="50"
    height="50"
    borderStyle="round"
    flexDirection="row"
    justifyContent="space-around"
>
    <VerticalLine />
    <VerticalLine />
    <VerticalLine />
</Box>
```



horizontal-lines



vertical-lines