Lui.el Documentation

Lens UI - React-like Component System for Emacs

Contents

Introduction

`lui.el` (Lens UI) extends `lens.el` with a React-like component system for building interactive user interfaces in Emacs.

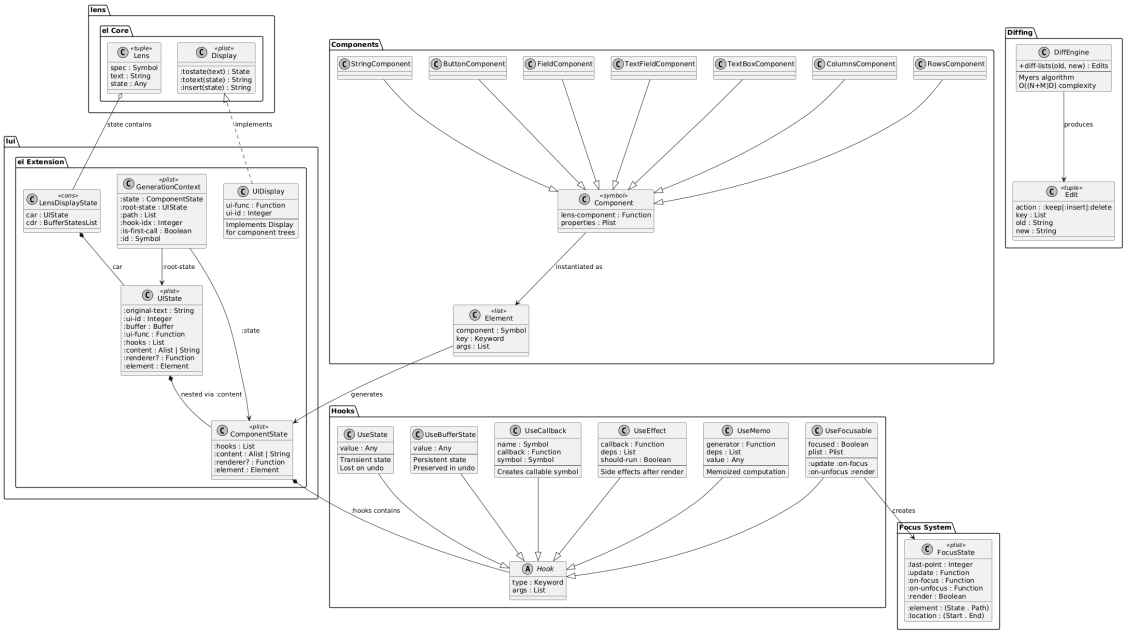
Key features:

- Declarative component-based UI
- React-style hooks (useState, useEffect, useMemo, etc.)
- Efficient diffing and partial rerendering
- Focus management for interactive elements
- Undo-aware state management

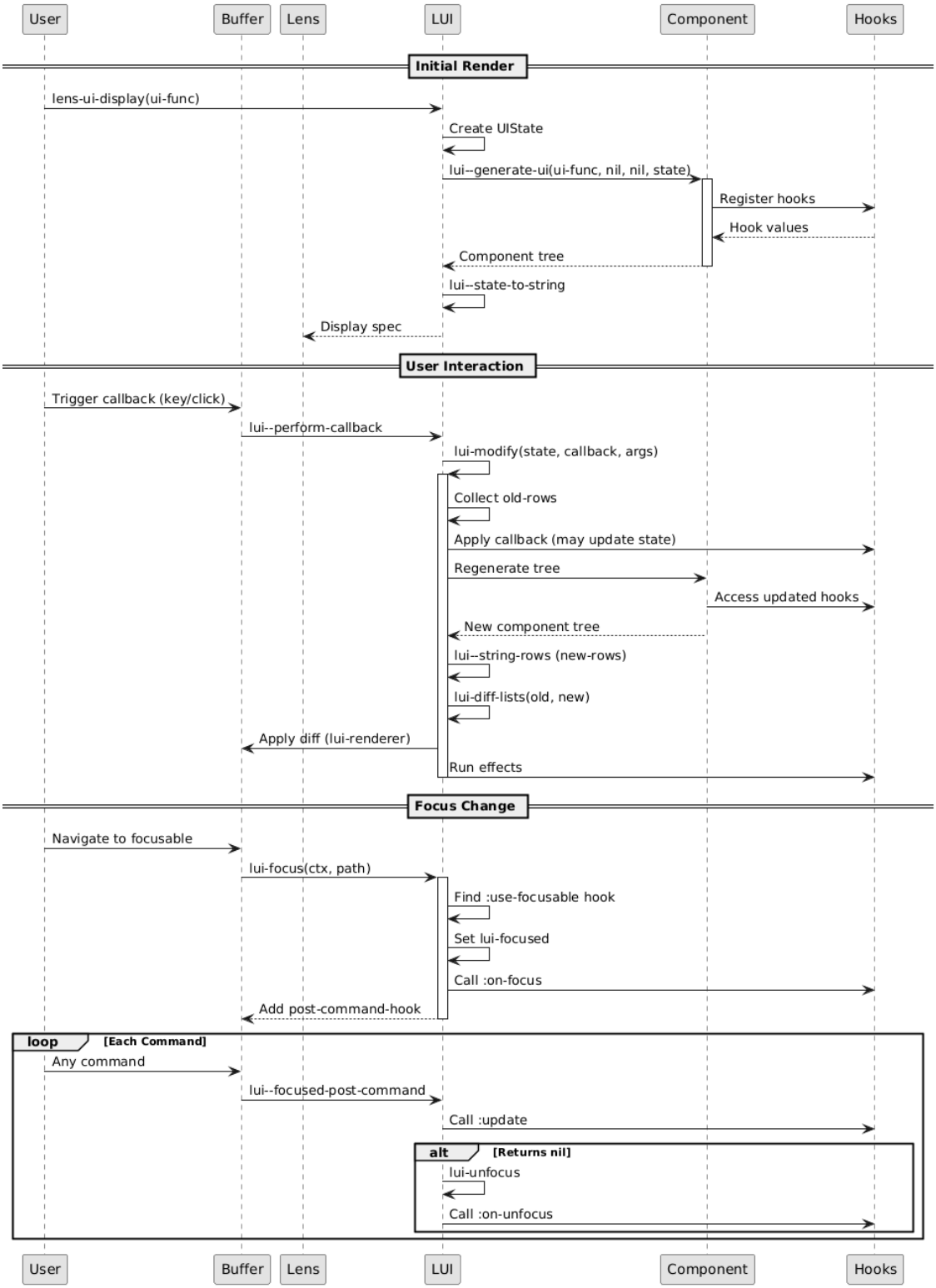
System Architecture

Extended UML Diagram

This diagram shows how lui.el extends the lens.el architecture:



Render and Update Flow



Core Concepts

Components

Components are functions that generate UI content. They receive a context and return either:

- A string (leaf content)
- A list of child components

Defining Components

```
(lens-defcomponent my-component (ctx arg1 arg2)
  :can-be-column t  ; optional property

  (:use-state value set-value 0)

  (format "Value: %d" value))
```

Defining UI Functions

```
(lens-defui my-ui ()
  (list (my-component :first "arg1" "arg2")
        (my-component :second "arg3" "arg4")))
```

Elements

An element is a component invocation: `(COMPONENT KEY ARGS...)`

Part	Description
COMPONENT	Symbol with `lens-component` property
KEY	Keyword for identification (`:my-key`)
ARGS	Arguments passed to component

Keys must be unique among siblings and are used for:

- State tree navigation
- Efficient diffing
- Focus targeting

State Structure

Component State

```
(:hooks ((HOOK-TYPE ARG1 ...) ...)
 :content "string" | ((KEY . CHILD-STATE) ...)
 :renderer FUNCTION  ; optional
 :element ELEMENT)
```

UI State (Root)

```
(:original-text TEXT
 :ui-id INTEGER
 :buffer BUFFER
 :ui-func FUNCTION
 ;; plus component state properties
 :hooks ... :content ...)
```

Lens Display State

```
(UI-STATE . BUFFER-STATES)
```

`BUFFER-STATES` is a list of `(PATH HOOK-IDX VALUE)` for undo support.

Hooks

Hooks provide state and lifecycle management within components. They must be called in the same order on every render.

:use-state

Transient local state (lost on undo).

```
(:use-state count set-count 0)

;; count: current value
;; set-count: setter function
;; 0: initial value

(set-count (1+ count))
```

:use-buffer-state

Persistent state (preserved in undo history).

```
(:use-buffer-state text set-text "initial")

;; Works like :use-state but survives undo/redo
```

:use-callback

Create a callable function for keymaps/actions.

```
(:use-callback on-click ()
               (message "Clicked!"))

;; on-click is a symbol you can use in keymaps
(:use-text-properties 'keymap `(keymap (return . ,on-click)))
```

With arguments:

```
(:use-callback handle-input (value)
               (set-text value))
```

:use-effect

Run side effects after rendering.

```
(:use-effect [count]  ; dependencies
             (message "Count changed to %d" count))

;; Runs when count changes
;; Empty [] means run only on first render
```

:use-memo

Memoize computed values.

```
(:use-memo expensive-result [input]
           (compute-expensive input))

;; Only recomputes when input changes
```

:use-focusable

Make the element focusable.

```
(:use-focusable focused?
                :update (lambda (plist) t)  ; return nil to unfocus
                :on-focus (lambda (plist) ...)
                :on-unfocus (lambda (plist) ...)
                :render t)  ; rerender on focus change
```

:use-text-properties

Apply text properties to the element.

```
(:use-text-properties 'face 'bold
                      'keymap my-keymap)
```

:use-renderer

Set a custom renderer for child components.

```
(:use-renderer ctx
               (lambda (children)
                 (string-join children " | ")))
```

Built-in Components

string

Simple read-only text.

```
(string :key "Hello, World!")
(string :key (list "line 1" "line 2"))  ; joined with newlines
```

button

Clickable button.

```
(button :ok "OK" on-click)
(button :cancel "Cancel" on-cancel :face 'warning)
```

field

Basic editable text field.

```
(field :input text on-change
       :header "[start]\n"
       :footer "\n[end]")
```

text-field

Enhanced field with focus support.

```
(text-field :editor text set-text
            :header ">>> "
            :footer " <<<"
            :props '(face font-lock-string-face))
```

text-box

Bordered text input with word wrapping.

```
(text-box :input text set-text
          :width 40
          :onenter submit-callback)
```

Features:

- Unicode/ASCII borders
- Word wrapping
- Cursor tracking
- ESC to exit, RET for onenter

columns

Horizontal layout.

```
(columns :layout
         (string :col1 "Left")
         (string :col2 "Middle")
         (string :col3 "Right"))
```

Children must have `:can-be-column` property.

rows

Vertical layout.

```
(rows :stack
      (string :row1 "First")
      (string :row2 "Second")
      (string :row3 "Third"))
```

API Reference

Defining UI

lens-defui

```
(lens-defui NAME ARGLIST BODY...)
```

Define a UI function for use with `lens-ui-display`.

lens-defcomponent

```
(lens-defcomponent NAME ARGLIST
  :property value ...
  BODY...)
```

Define a reusable component.

lens-ui-display

```
(lens-ui-display UI-FUNC) -> display-plist
```

Create a display specification for a UI function.

State Management

lui-modify

`(lui-modify STATE CALLBACK ARGS)`

Execute `CALLBACK` with `ARGS` and rerender the UI.

Focus Management

lui-focus

`(lui-focus CTX &rest RELATIVE-PATH)`

Focus the element at `RELATIVE-PATH` from `CTX`.

lui-unfocus

`(lui-unfocus)`

Unfocus the currently focused element.

Navigation

lui-find-lens

`(lui-find-lens STATE) -> region`

Navigate to and return the lens for `STATE`.

lui-find-element

`(lui-find-element STATE KEY-PATH) -> (START . END)`

Find element at `KEY-PATH` within `STATE`'s lens.

lui-find-ctx

`(lui-find-ctx CTX) -> (START . END)`

Find element for the given context.

Utilities

lui-text-to-box

`(lui-text-to-box TEXT &rest PROPS) -> string`

Wrap `TEXT` in a border box. Props: `:width`, `:title`, `:charset`, `:shrink`, `:pad`.

lui-diff-lists

`(lui-diff-lists OLD-LIST NEW-LIST) -> edits`

Diff two lists using Myers algorithm.

Customization

Variables

Variable	Default	Description
`lui-debug`	nil	Show rerender highlights

Faces

Face	Description
`lui-rerender-highlight`	Highlight for debug rerendering
`lens-button`	Default button face

Hooks

Hook	Description
`lens-box-enter-hook`	Run when entering text-box
`lens-box-exit-hook`	Run when exiting text-box

Examples

Counter

```
(lens-defui counter-ui ()
  (:use-state count set-count 0)
  (:use-callback increment () (set-count (1+ count)))
  (:use-callback decrement () (set-count (1- count)))

  (list
   (string :display (format "Count: %d" count))
   (columns :buttons
            (button :dec "-" decrement)
            (button :inc "+" increment))))

;; Usage:
(lens-create (lens-replace-source "")
             (lens-ui-display #'counter-ui))
```

Text Editor

```
(lens-defui editor-ui ()
  (:use-buffer-state text set-text "")  ; survives undo
  (:use-state char-count set-char-count 0)
```

```
  (:use-effect [text]
               (set-char-count (length text)))

  (list
   (text-box :editor text set-text :width 50)
   (string :status (format "Characters: %d" char-count))))
```

Dynamic List

```
(lens-defui list-ui ()
  (:use-state items set-items '("Item 1" "Item 2"))

  (:use-callback add-item ()
                 (set-items (append items
                                    (list (format "Item %d"
                                                  (1+ (length items)))))))

  (cons (button :add "Add Item" add-item)
        (cl-loop for item in items
                 for i from 0
                 collect (string (intern (format ":item-%d" i))
                                 item))))
```

State Management Deep Dive

Transient vs Persistent State

Type	Hook	Undo Behavior	Use Case
Transient	`:use-state`	Lost on undo	UI state, selections
Persistent	`:use-buffer-state`	Preserved	User data, form values

How Buffer State Works

1. On each render, buffer states are collected
2. Stored in lens display state as `(UI-STATE . BUFFER-STATES)`
3. `UI-STATE` is shared/mutated across all undo states
4. `BUFFER-STATES` is copied per undo version
5. Before callbacks, buffer states are propagated back to UI-STATE

Hook Order Constraint

Hooks must be called in the same order every render:

```
;; BAD - conditional hook
(when condition
  (:use-state x set-x 0))  ; will break
```

```
;; GOOD - unconditional hook
(:use-state x set-x 0)
(when condition
  (set-x 1))
```

Debugging

Enable Debug Mode

```
(setq lui-debug t)
```

This highlights rerendered regions briefly after each update.

Inspecting State

```
;; Get the lens at point
(lens-at-point)

;; The third element of the lens is (UI-STATE . BUFFER-STATES)
(caddr (car (lens-at-point)))
```

Integration with lens.el

lui.el integrates with lens.el through the display system:

1. `lens-ui-display` creates a display plist
2. `:tostate` generates the component tree
3. `:insert` converts the tree to insertable text
4. Modifications use `lens-modify` with custom renderer
5. The renderer applies diffs for efficient updates

Key integration points:

- `lens-silent-edit` for modification batching
- `lens-save-position-in-ui` for cursor preservation
- Field system from lens.el for editable content