General Purpose User Interface - GPUI

October 27, 2025

The GPUI (General Purpose UI) framework is designed to let developers build flexible, composable user interfaces with a small set of well-defined primitives. Its goals are:

- Simplicity: provide a concise API (XML + runtime) for declaring layouts and behaviour.
- Composability: make it easy to assemble complex widgets from small elements (boxes, lines, text, framed wrappers, groups).
- Reusability: support prefabs/styles so visual appearance and structure can be reused and swapped without changing layout logic.
- Predictability: explicit child/slot semantics and predictable layout rules so UIs behave consistently across different resolutions.

Who should use GPUI:

- Game developers or lightweight app authors who want a small, deterministic UI system layered on a renderer.
- People who prefer layout-by-declaration (XML-style prefabs) with runtime access to named elements.

What to expect from this manual:

- A reference of available XML elements and their attributes
- Clear guidance on what children each element accepts and how they are used
- Examples showing typical composition patterns and common use-cases

Basic example

The smallest useful GPUI layout is a ‘UI‘ that contains a single panel with a button. Below is a minimal XML example demonstrating element composition and child usage (the button wraps a ‘text‘ child to provide its label):

```
<ui>
    <box label="mainpanel" colors="(40,40,48)" inset="12">
        <group spacing="8">
            <button onclick="sayHello" inset="10">
                <text color="white">Click me</text>
            </button>
            <text color="(200,200,200)" fontsize="m">Status: idle</text>
        </group>
    </box>
</ui>
```

Notes on the example:

- The ‘button‘ contains a single ‘text‘ child that defines its normal appearance. You can add a second child to style the pressed state separately.
- The ‘group‘ arranges children vertically by default and manages spacing between them.
- The ‘ui‘ root registers named elements (like ‘mainpanel‘) for programmatic lookup at runtime.

Dynamic Layout System

One of GPUI's most powerful features is its constraint-based dynamic layout system. Instead of manually positioning elements with absolute coordinates, GPUI uses a sophisticated system of relative positioning and automatic layout management:

Key Features

- **Automatic Layout**: The UI root element intelligently manages the placement and sizing of all child elements, eliminating the need for manual positioning calculations.
- **Relative Positioning**: Elements are positioned relative to their containers or siblings using flexible constraints rather than absolute coordinates. This enables:
  - Automatic resizing when parent containers change size
  - Proper scaling across different screen resolutions
  - Dynamic rearrangement based on available space
- **Constraint-Based Layout**: Elements can be positioned using:
  - Percentage-based positioning (e.g., "center this at 50% of parent width")
  - Edge alignments (top, bottom, left, right)
  - Relative spacing between elements
  - Automatic size distribution in groups
- **Smart Grouping**: The Group element provides sophisticated layout capabilities:
  - Automatic vertical or horizontal arrangement
  - Dynamic spacing between elements
  - Proportional size distribution
  - Nested groups for complex layouts

The UI root automatically handles:

- Proper element sizing based on content
- Spacing between elements
- Consistent alignment
- Resolution independence
- Responsive behavior

Event System

GPUI includes a lightweight event system used to decouple user interaction (clicks, changes, drags) from application logic. Events are simple named signals that can carry a small payload (context) and are dispatched by elements or emitted programmatically.

Key concepts:

- **Event names:** Events are identified by string names (for example, "myEvent" or "volumeChange"). Many element attributes (like `onclick`, `onchange`) are simply event names that the parser wires to the element's interaction hooks.
- **EventManager:** A central registry that manages subscriptions. Handlers can subscribe to event names and will be invoked whenever an event with that name is emitted.
- **Event payload:** Emitted events typically include a small payload object containing at least the event source (the element that fired the event) and any relevant data (new value, state, metadata). Handlers should expect a compact object/dict and avoid heavy data copies.
- **Local vs global triggers:** Some elements can be configured to emit events only when interacted with locally (inside their bounds) or as global triggers that respond regardless of focus or pointer position. Use the element's attributes to control this behaviour.

Common interaction mappings:

- `Button`: emits the event named by `onclick` when pressed (and optionally during hold/release depending on configuration).
- `Toggle/Checkbox`: emits `onchange` when the state cycles/changes.
- `Slider`: emits `onchange` when the value changes (continuous or on-release depending on `step` and implementation).

Minimal usage examples:
XML (declarative):

```
<button onclick="myEvent">Click me</button>
<slider onchange="volumeChanged" />
```

Python (subscribe at runtime):

```
from ui.interaction.event.eventmanager import EventManager

def on_my_event(evt):
    # evt is a small object/dict with at least: name, source, data
    print('Received', evt.name, 'from', evt.source)

EventManager.subscribe('myEvent', on_my_event)

# Emit programmatically (optional)
EventManager.emit('myEvent', {'value': 123})
```

Best practices and notes:

- Prefer simple payloads (primitives or small dicts) to avoid expensive copies across subscribers.
- Unsubscribe handlers when they are no longer needed (for example, when a screen or element is destroyed) to avoid memory leaks.
- Use unique, descriptive event names (e.g., `settings.audio.mute`) to reduce accidental collisions in large projects.

- For high-frequency events (drag/continuous slider updates) consider throttling or debouncing on the application side.

API hooks (common operations):

- `EventManager.subscribe(name, handler)` — register a handler for an event name.
- `EventManager.unsubscribe(name, handler)` — remove a previously-registered handler.
- `EventManager.emit(name, payload)` — emit a named event with an optional payload.
- Element attributes like `onclick` / `onchange` — declarative wiring from XML to event names.

This event system keeps UI code concise and makes it easy to connect declarative XML layouts to application logic without tight coupling.

Advanced use

The **GPUI** (General Purpose UI) is built around modular units called **Elements**. Each Element encapsulates its **position**, **functionality**, **normal data**, and **render data**, which collectively define its behavior and appearance on the screen.

- **Position** in GPUI is defined through *joints*—dynamic relationships between elements. A joint links a relative position within one element to a relative position in another element (or to a static rectangle). Joints also support *absolute offsets* in both x and y directions and can be used to *fix the size* of elements similarly. This system enables flexible, constraint-based layout behavior.
- **Functionality** refers to the methods available to the element, such as subscribing to button events or responding to user input.
- **Normal data** represents element-specific content that isn't exclusive to rendering—for example, the text in a text field.
- **Render data** includes appearance-related properties like font color, background color, and other visual attributes.

XML Parser

GPUI uses **XML** to define and load layouts.

Tags

The **tag system** provides a shorthand for managing complex rendering logic. A tag is a named reference initialized with element-specific data, such as a color. Elements can then refer to these tags for rendering purposes—for example, using a tag-based *color order* to alternate line segment colors in a `Line` element.

Tags must be strings that **do not start with a number** and **cannot contain** the characters: `=`, `:`, `;`, `[`, `]`, `{`, or `}`.

Available XML-Elements

Line-Element

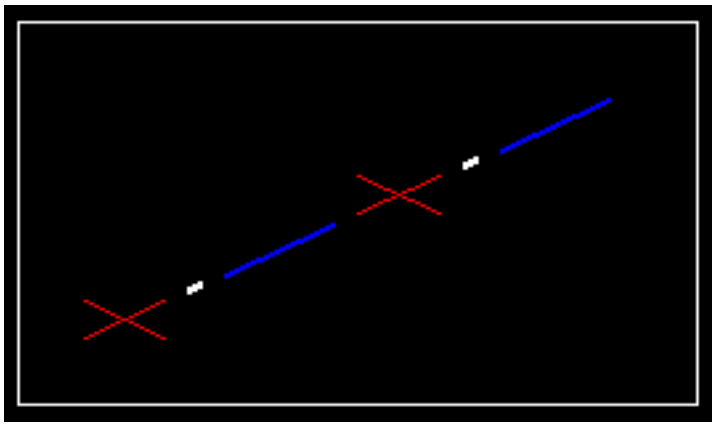
The `Line`-Element is used to specify a line to be drawn in the UI.

`<line>` | `<l>`

Arguments

Name	Values	Description
colors/ color/ col	tag:color;...	Defines tag-based colors for lines. Values can be RGB (r,g,b) or color names (e.g., "red"). "inv"/"none" makes them invisible. If no tag is set (e.g., "color=red"), default tag ” is used. Default: inv
flip		Draws line from bottom-left to top-right instead of default top-left to bottom-right. Default: noflip
inset	float|float,float| int|int,int	Adds padding to the line. One value applies to both dimensions; two values set width and height separately. Ints are absolute; floats are percentages. Default: 0
order/ sectionorder	tag1,tag2,...	Specifies a sequence of tags for color/size/thickness, repeated cyclically along the line. Default: ” (default tag)
sizes/ size	tag:float|int;...	Sets segment lengths (absolute or relative) per tag. Uses default tag ” if unspecified. Default: 1.0
thickness/ width	tag:int;...	Sets absolute line thickness per tag. Uses default tag ” if unspecified. Default: 1
	EXPERIMENTAL	
altmode	tag:mode;...	Alternate drawing mode. Supported: cross. Default: default

Example (256×144):
<line
color="inv;r:red;b:blue;w:white"
thickness="1;b:2;w:3"
flip=""
sizes="10;r:0.15;b:0.20;w:5"
inset="25"
sectionorder="r,,w,,b,"
altmode="r:cross"> </line>



Box-Element

The `Box`-Element is used to specify a box to be drawn in the UI.

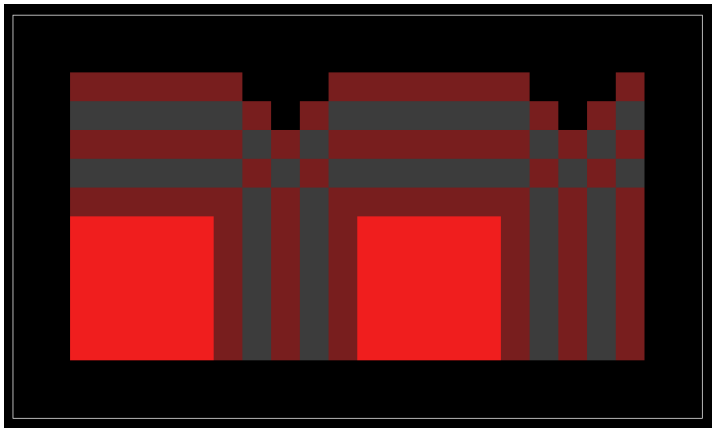
`<box>` | `<b>`

Arguments

Name	Values	Description
colors/ color/ col	tag:color;...	Defines tag-based colors for rendering. Colors can be RGB tuples (r,g,b) or names (e.g., "red"). "inv" or "none" make elements invisible. If no tag is specified (e.g., "color=red"), the default tag ” is used. Default: inv
fillmode(s)/ fill(s)/ altmode(s)/ mode(s)	tag:mode;...	Sets the fill pattern for sub-boxes: striped vertically/horizontally or checkerboard. Values: striped_vert (strv), striped_hor (strh), checkerboard (cb). Uses default tag ” if none given. Default: solid
fillsize(s)/ innersizing(s)/ size(s)	tag:int|float;...	Sets size of the alternating pattern, either absolute or relative. Uses default tag ” if none given. Default: 10
inset	float|float,float| int|int,int	Sets padding inside the box. One value applies to both dimensions; two values specify width and height separately. Ints are absolute; floats are percentages. Default: 0
orders/ sectionorders/ ord	tag:tag1,tag2,...;...	Defines tag sequence used to render fill patterns cyclically in sub-boxes. Uses default tag ” if none given. Default: ”
partitioning/ part	intxint;[c1,c2,...]| [1=c1,3=c3,...]| 4=[...]	Splits the box into columns × rows of sub-boxes. Each sub-box can have its own fillmode, size, and filter defined by tags. Labels are set per row or column using square brackets. Sub-boxes have ”-tag per default. Default: 1x1
	EXPERIMENTAL	
filter(s)/ filt	tag:mode= float,float,float+float;...	Applies shape filters to sub-boxes. Modes: triangle/linear/quadratic/circle, with optional inversion ("i" prefix). Filters define a point and max distance for visible area. Default: nofilter

Example (960×560):

```
<box
partitioning="4x2;[a,b,a,b][2=c,4=c]"
filter="b:it=0.5,0.0,0.5+0.0"
inset="80"
colors=
"(240,30,30);c1:(120,30,30);c2:(60,60,60)"
fillmode="a:strh;b:cb;c:strv"
fillsize="40"
sectionorders=
"c3;a:c1,c2;b:c1,c2;c:c1,c2"></box>
```



Text-Element

The `Text`-Element is used to specify a text to be drawn in the UI.

`<text>` | `<t>`

Arguments

Name	Values	Description
align	float|l|r,float|t|b	Sets the align of the text inside the text-box. Default: 0.5,0.5
colors/ color/ col	color	Defines fontcolor for rendering. Color can be RGB tuple (r,g,b) or name (e.g., "red"). "inv" or "none" makes text invisible. Default: inv
fontname/ sysfont/ font	fontname;...	Sets font for rendering . Default: Arial
fontsize/ size	d|int|xxs|xs|...|xl|xxl;...	Sets fontsize for rendering. 'd' sets the fontsize to be dynamicly calculated. Default: 24
inset	float|float,float| int|int,int	Sets padding inside the box. One value applies to both dimensions; two values specify width and height separately. Ints are absolute; floats are percentages. Default: 0

Example (640×360):

```
<text
inset="100"
color="red"
fontsize="l"
align="0.3,0.9">
Hello World
</text>
```

Framed-Element

The `Framed`-Element provides a flexible container that enhances elements with customizable borders and backgrounds. It supports individual border styling for each edge and optional background elements with advanced visual effects.

`<framed>` | `<fr>` | `<f>`

Arguments

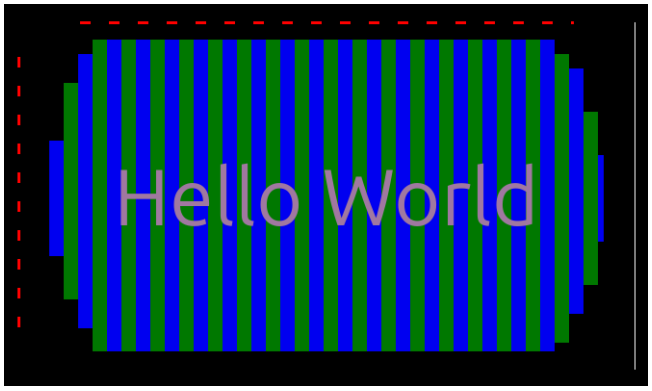
Name	Values	Description
inset/ offset/ padding	int	Sets padding inside the box. Default: 0

Children

Name	Amount	Description
`Line`	0 - 4	Sets the borders of the framed. If one is provided it is applied to all sides. If two are provided, the first is applied to left and right and the second to top and bottom. Otherwise borders are applied like 1-left, 2-right, 3-top, 4-bottom. Default: noborders
`Box`	0 - 1	Sets the background of the framed. Default: noback-ground
`Any Element`	1	Wrapped element.

Example (640×360):

```
<framed offset="50">
<line color="inv;r:red" thickness="3" sizes="10;n:20" inset="0.1" sectionorder="r,n"></line>
<line color="white"></line>
<line color="inv;r:red" thickness="3" sizes="10;n:20" inset="0.1" sectionorder="r,n"></line>
<box colors="(0,0,240);a:(0,120,0)" inset="0.05" fillmode="strv" sectionorders=",a" filter="c=0.5,0.5,0.5+0.5"
fillsize="15"></box>
<text inset="50" color="(160,120,160)" fontsize="d">Hello World</text>
</framed>
```



Group-Element

The `Group`-Element provides a powerful layout organization system that arranges multiple elements in vertical or horizontal configurations. Groups handle automatic sizing, spacing, and alignment while supporting nested grouping for complex layouts.

`<group>` | `<group>` | `<gr>` | `<g>`

Arguments

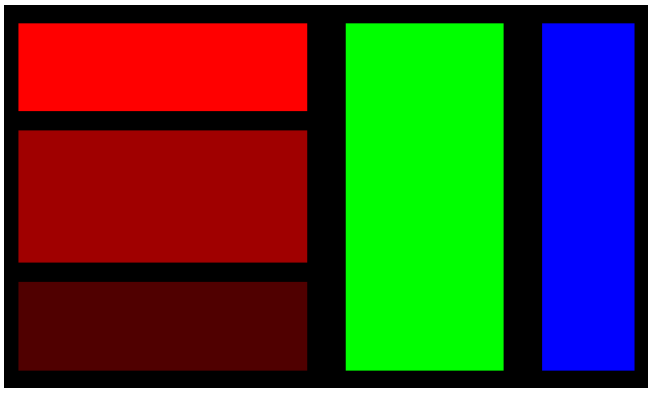
Name	Values	Description
horizontal/ hor		Sets the group to align the elements horizontal. Default: vertical
offset/ spacing	int	Sets the spacing between elements in the group. Default: 0
size(s)/ sizing(s)	float|int=float,...	Sets the relative sizing of the elements inside the group. Default: 1.0

Children

Name	Amount	Description
`Any Element`	1+	Grouped elements.

Example (640×360):

```
<group spacing="40" hor="" sizings="0.5,0.3,0.2">
<group spacing="20" sizing="2=1.4">
<box color="red"></box>
<box color="(160,0,0)"></box>
<box color="(80,0,0)"></box>
</group>
<box color="green"></box>
<box color="blue"></box>
</group>
```



Dropdown-Element

The `Dropdown`-Element is used to create a dropdown.

`<dropdown>` | `<dpd>`

Arguments

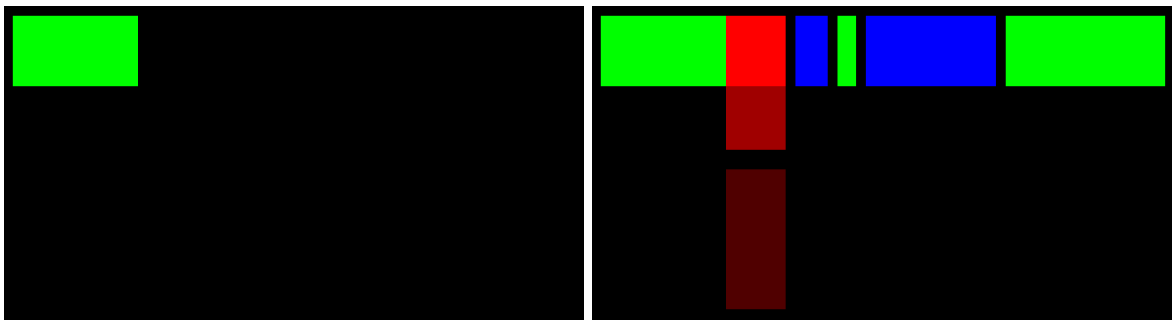
Name	Values	Description
horizontal/ hor		Sets the dropdown to be horizontal. Default: vertical
offset/ spacing	int	Sets the spacing between elements in the dropdown. Default: 0
size(s)/ sizing(s)	float|int=float,...	Sets the relative sizing of the elements inside the dropdown. Default: 1.0

Children

Name	Amount	Description
`Any Element`	1+	First element is the one to click to reveal the other elements as dropdown.

Example (128×72):

```
<dpd spacing="10" hor="" sizings="0.5,0.3,0.2,c5=1.2">
<box color="green"></box>
<dpd spacing="20" sizing="c2=1.9">
<box color="red"></box>
<box color="(160,0,0)"></box>
<box color="(80,0,0)"></box>
</dpd>
<box color="blue"></box>
<box color="green"></box>
<box color="blue"></box>
<box color="green"></box>
</dpd>
```



Button-Element

The `Button`-Element provides an interactive clickable component with press, hold, and release states. It supports both local and global event triggering, and can be visually customized for different interaction states.

`<button>` | `<btn>`

Buttons support several interaction modes:

- Click events with optional hold-state tracking
- Global trigger events that work anywhere on screen
- Local trigger events that only work within button bounds
- Press and release state visualization
- Multiple callback subscription methods

Arguments

Name	Values	Description
text/ content	string	Label text shown on the button. Default: ”
colors/ color	tag:color;...	Render colors for background/border/text using tag-based scheme (see Box/Text). Default: inv
onclick/ on	eventname	Event name to be fired when button is pressed (subscribe via EventManager). Default: none
inset	float|int	Padding inside the button. Default: 0
size/ sizing	float|int	Size hints. Default: dynamic

Children

Name	Amount	Description
`Any Element`	0 - 2	First element defines the unclicked state appearance, second (optional) defines the clicked state appearance. If only one element is provided, it’s used for both states. If no elements are provided, default styling is used.

Example:

```
<button onclick="myEvent" inset="20">
<text color="white">Click me</text>
</button>
```

Checkbox / Toggle-Element

The `Checkbox` or generic `Toggle`-Element provides boolean and multi-state toggles with customizable state transitions and visual feedback. Each state can have its own visual representation and trigger specific callbacks.

`<checkbox>` | `<cycle>` | `<elementcycle>`

The Toggle system provides:

- Multiple visual states with smooth transitions
- State-specific callback subscriptions
- Flexible state cycling patterns
- Global and local trigger events
- Optional hold-state behavior

Arguments

Name	Values	Description
typ	checkbox|cycle	Type of toggle (checkbox = boolean, cycle = cycles through values). Default: checkbox
values	v1,v2,...	For cycle toggles, the sequence of states. Default: on,off
onchange	eventname	Event fired on state change.
colors	tag:color;...	Colors for each state; follows tag-based color scheme.

Children

Name	Amount	Description
`Any Element`	1 per state	For checkbox: two children define unchecked/checked states. For cycle toggles: provide one child per value in 'values' attribute. Children are shown/hidden based on current state.

Example:

```
<checkbox onchange="toggleAudio">
<box colors="(200,200,200)"><text>Off</text></box>
<box colors="(100,255,100)"><text>On</text></box>
</checkbox>
```

Slider-Element

The `Slider`-Element exposes a continuous (or discrete) range control.

`<slider>` | `<sld>`

Arguments

Name	Values	Description
min	float|int	Minimum value. Default: 0
max	float|int	Maximum value. Default: 1
step	float|int	Step size (optional, makes slider discrete). Default: continuous
value	float|int	Initial value. Default: min
onchange	eventname	Event fired when the slider value changes.

Children

Name	Amount	Description
`Track`	0 - 1	First child defines the slider track appearance. Default: gray bar
`Handle`	0 - 1	Second child defines the movable handle appearance. Default: white box

Example:

```
<slider min="0" max="100" value="25" onchange="volChange">
<box colors="(80,80,80)"></box>
<box colors="(200,200,200)"></box>
</slider>
```

Multiselect-Element

The `Multiselect`-Element allows selecting multiple options from a list.

`<multiselect>` | `<multi>`

Arguments

Name	Values	Description
options	comma separated strings	If provided as attribute, creates radio/option entries.
selected	idx1,idx2,...	Indices of initially selected options.
onchange	eventname	Event fired when selection changes.

Children

Name	Amount	Description
`Option`	0+	Each child defines one option's appearance. Child order determines option index.
`Selected`	0 - 1	Optional last child defines selected state appearance. Applied to all options unless they provide their own.

Option handling:

- If no children provided, options are automatically created from 'options' attribute
- Each option can have unique visuals by providing explicit children
- Selected state appearance is shared unless options define their own
- Options are arranged vertically unless 'horizontal' attribute is set

Example:

```
<multiselect selected="0,2">
<box colors="(220,220,220)"><text>Apple</text></box>
<box colors="(220,220,220)"><text>Banana</text></box>
<box colors="(220,220,220)"><text>Cherry</text></box>
<box colors="(100,200,100)"></box>
</multiselect>
```

Dropdownselect-Element

The `Dropdownselect` is a hybrid between a dropdown and a select-list, providing a compact chooser.

`<dropdownselect>` | `<dpds>`

Children

Name	Amount	Description
`Trigger`	1	First child defines the dropdown trigger button appearance
`Option`	0+	Following children define option appearances, shown when expanded
`Selected`	0 - 1	Optional last child defines selected state appearance for options

Behavior notes:

- Trigger child is always visible and toggles dropdown expansion
- Option children are only visible when dropdown is expanded
- Without explicit children, options are created from 'options' attribute
- Selected option's appearance is copied to trigger unless trigger provides states

Example:

```
<dropdownselect selected="1">
<box colors="(200,200,200)"><text>Select difficulty</text></box>
<box colors="(220,220,220)"><text>Easy</text></box>
<box colors="(220,220,220)"><text>Normal</text></box>
<box colors="(220,220,220)"><text>Hard</text></box>
<box colors="(100,200,100)"></box>
</dropdownselect>
```

Section-Element

The `Section`-Element provides a sophisticated content organization system with optional headers, footers, and automatic content pagination. It handles content that exceeds the display area by creating navigable sections.

`<section>` | `<sec>`

Section features:

- Optional header and footer elements
- Automatic content pagination based on height limits
- Navigation controls for multi-section content
- Keyboard navigation support (arrow keys)
- Configurable section separators
- Dynamic content sizing and positioning

Child Elements

Name	Position	Description
`Header`	First	When 'header' attribute is true, the first child becomes the section header. Fixed at top.
`Main Content`	Middle	All middle children form the scrollable content area. These are automatically paginated if they exceed the section height.
`Footer`	Last	When 'footer' attribute is true, the last child becomes the section footer. Fixed at bottom.

Special behaviors:

- Header and footer remain fixed while content scrolls
- Content elements are automatically arranged in pages when they exceed the height limit
- Navigation arrows appear when content spans multiple pages
- Each content element can specify its own relative size using 'size' attribute

Example:
<section title="Controls">
<group> ... </group>
</section>

UI-Element (Root)

The `UI`-Element serves as the root container that manages the complete layout structure and provides sophisticated organization features including header/footer bars and side panels.

```
<ui>
```

The UI element provides:

- Named element lookup system for easy access to any UI component
- Optional header and footer areas with automatic sizing
- Left and right side panel support with automatic layout
- Dynamic element sizing based on container dimensions
- Efficient element activation state management
- Automatic z-index handling for proper layering

Child Layout

Position	Type	Description
Top	Header Bar	When 'header' attribute is set, first child becomes the header. Spans full width, fixed height.
Center	Side Panels	Middle children are arranged in side panels. Distribution alternates: first to left panel, second to right, etc.
Bottom	Footer Bar	When 'footer' attribute is set, last child becomes the footer. Spans full width, fixed height.

Important child requirements:

- Every child must have a unique `label`, `id` or `name` attribute for lookup
- Side panel elements should specify preferred widths using 'size' attribute
- Header/footer heights are determined by their content unless explicitly sized
- Side panels automatically balance elements between left and right sides
- Z-index is managed automatically for overlapping elements

The UI maintains a registry of all named elements, making them accessible through both parser and runtime APIs. This enables programmatic access to any element using its registered identifier.

Prefab and Style System

The GPUI prefab & style system separates structure (prefabs) and visual defaults (styles) so you can reuse element blueprints and switch appearances globally.

Key concepts

- **Style:** A named collection of element templates (XML element nodes) that serve as prefabs and visual defaults. Styles are loaded via `Parser.loadStyleFromXML(path)` and registered with the `StyleManager`. Use the style-loading XML to declare many templates under one style name.
- **Prefab (styled element node):** An XML element stored inside a style file and identified by a `label/id/name` attribute. Prefabs are used by the parser whenever an unknown tag is encountered: the parser will look up a same-named prefab inside the selected style and instantiate that node instead.
- **Style tags:** In element XML you can specify which style to use via any of the tags defined in `Element.styleTags` (by default: `style`, `styled`, `styleid`, `styledid`). If none is provided the `StyleManager.defaultStyle` is used.
- **fixstyle attribute:** During parsing the parser sets an internal `fixstyle` attribute on the element attributes dictionary to the chosen style name (either from the element or the default). This value is used when looking up prefabs.

How to author a style file

Create an XML file whose root contains a style definition and a set of named child elements. Each child must include an ID (one of `label`, `id`, `name` etc.) — that string becomes the lookup key for prefabs.

Minimal example (style.xml):

```
<styles>
    <style id="default">
        <button label="fancybutton">
            <framed offset="8">
                <box colors="(240,240,240)" inset="6"></box>
                <text color="black" fontsize="l">Press</text>
            </framed>
        </button>
        <box label="card">
            <box colors="(255,255,255)" inset="12"></box>
        </box>
    </style>
</styles>
```

When this file is loaded with `Parser.loadStyleFromXML('style.xml')` the `StyleManager` will register a style named `default` and store the raw XML nodes for keys `fancybutton` and `card`.

How the parser uses prefabs

When the parser encounters a tag it doesn't recognize as a built-in element (e.g., `<fancybutton>`) it does the following:

1. Determine the style name to use: either an explicit style attribute on the node (one of the style tags) or the global default style in `StyleManager.defaultStyle`.

2. Ask `StyleManager.getStyledElementNode(node.tag, styleName)` for a raw XML node stored under that key in the style.
3. If a prefab node is found, the parser recursively parses that prefab node as if it was present in place of the unknown tag. The prefab's node tree (including its attributes and children) is used to create the element instance.

Notes and implications:

- The parser instantiates the prefab node as-is; attributes on the original unknown tag are not automatically merged into the prefab. If you need parameterized prefabs, either store several prefab variants or design prefabs to reference named children which you can override by name in the UI XML.
- Prefabs are stored as raw XML nodes in the `StyleManager.styles` mapping. This allows styles to contain full element trees (boxes, framed wrappers, text, etc.) that become the instantiated element when referenced by tag.
- Because styles may contain any element nodes, they can be used both for visual defaults (colors, fonts, paddings) and for fully structured prefabs (composite elements built from primitives).

Precedence and best practices

- Explicit element tags that the parser knows (`<button>`, `<box>` etc.) are parsed directly and may accept attributes that override defaults defined in code.
- Use style-prefabs when you want a short tag to expand into a complex widget tree. Keep prefab labels stable and document them in your style files.
- To change look-and-feel globally, create multiple style files (e.g., `light.xml`, `dark.xml`) and use `Parser.setDefaultStyle('dark')` at startup. Prefabs with the same label can be provided in both styles to produce different visuals while keeping layout XML unchanged.
- If you want to supply small overrides to a prefab at instantiate-time, consider using a named child inside the prefab and referencing it by `label` from the layout, which the parser will wire into the prefab instance.

API hooks:

- `Parser.loadStyleFromXML(path)` — load and register styles from an XML file.
- `StyleManager.getAllStyles()` — returns registered style names.
- `StyleManager.setDefaultStyle(name)` — make a style the global default.
- `Element.getStyledElement(elementName, styleName?)` — programmatically request an element instance from the style system at runtime.

This concludes the reference for the primary interactable elements and the prefab/style system. Use the examples above as starting points — styles are intentionally flexible so you can reuse and re-skin components quickly.