GUI Toolkit for Wayland

Academic project report for the Operating Systems course

Oleh Omelchuk
Ukrainian Catholic University
Computer Science
Lviv, Ukraine
oleh.omelchuk@ucu.edu.ua

Sviatoslav Lushnei
Ukrainian Catholic University
Computer Science
Lviv, Ukraine
sviatoslav.lushnei@ucu.edu.ua

***Abstract*—This document is a report on our implemented GUI Toolkit for Wayland as a part of the academic project in the Operating Sytems Fall-2023 Course at CS Program. It will describe the main features of our toolkit, the way it works, and the way it was implemented. Also, we will overview of the Wayland protocol and Cairo library, which were used in our project.**

***Index Terms*—GUI Toolkit, Wayland, Cairo, window**

I. Introduction

A. Graphical User Interface

Graphical User Interface (GUI) plays a pivotal role in enhancing user interactions with computers, providing a visual representation of applications, and facilitating user-friendly experiences. Developing a GUI for Wayland involves embracing its unique characteristics and principles, setting it apart from other systems.

B. Wayland

Wayland is a communication protocol that serves as a foundation for graphical compositors, allowing for the creation and manipulation of graphical elements on the display server.[1] It emphasizes simplicity and efficiency by providing a direct communication pathway between the application and the display server. Wayland relies on two fundamental abstractions: listeners and interfaces. Listeners serve as receptors, attentively awaiting events and updates from the server, while interfaces function as the communication medium through which applications engage with the display server.

C. Cairo

Cairo is a versatile 2D graphics library widely used for rendering, creating visually appealing graphics, and supporting vector graphics, contributing to various applications and toolkits. It brings seamless rendering and cross-platform compatibility. With robust support for vector graphics, it integrates with Wayland protocols, requiring EGL surfaces for optimal performance. Animation, extensibility, and customization contribute to a visually compelling user interface.

II. Implementation Structure

A. Overview

In the course of our implementation, the initiation phase involves the establishment of connections with display servers and the configuration of listeners to accommodate user inputs. Various surfaces, encompassing windows, buttons, and input fields, are meticulously crafted to underpin the graphical user interface. These surfaces engage in seamless communication through strategically implemented callbacks, fostering an interplay with the Wayland protocol.

The procedural sequence encompasses the systematic initialization of Wayland components, cursor management, integration of EGL for graphics rendering, and the incorporation of Cairo to refine visual aesthetics. The toolkit operates responsively, driven by events precipitated by user interactions, particularly those associated with pointer events.

In wayland everything is a surface, every object is located on sufrace, the window itself also is surface. In the Wayland paradigm, the concept prevails that everything is a surface; each object, including the window itself, is categorized as a surface. Surfaces serve as the fundamental entities upon which graphical objects reside and interact. Notably, these surfaces are not homogenous; they may assume distinct roles, thereby dictating their behavior and purpose within the graphical framework.

B. Classes and Abstractions

In our design, the Wayland GUI Toolkit features a hierarchy of abstractions: Window, Component, and SubComponent. The Window class encapsulates top-level surfaces, managing initialization, resizing, and fullscreen functionality. The Component class represents graphical elements and handles drawing, resizing, and surface management.

Every component, whether a standalone Component, a SubComponent within a parent, or a top-level Window, possesses its own surface. Subcomponents, in particular, assume the role of subsurfaces. Subsequently, the SubComponent class extends Component to enable hierarchy and other features. The code emphasizes modularity and encapsulation, facilitating a flexible and scalable GUI architecture within the Wayland framework.

1) Component Abstraction: Every graphical element is encapsulated within the Component abstraction, each having its own surface. This class manages surface creation, resizing, and drawing operations. It is designed to be versatile, representing a broad category of graphical entities within the GUI toolkit.

2) SubComponent Abstraction: Building upon the Component abstraction, SubComponent introduces the concept of hierarchy. Subcomponents are embedded within parent components, offering features like anchored positioning, pivoting, and resize propagation. This hierarchy allows for a structured arrangement of graphical elements, facilitating more complex and organized GUI layouts.

3) Window Abstraction: The Window abstraction encapsulates top-level surfaces and incorporates a specialized XDG top surface. This class handles surface initialization, resizing, and fullscreen functionality. It orchestrates the creation of additional components, forming the basis for the graphical user interface.

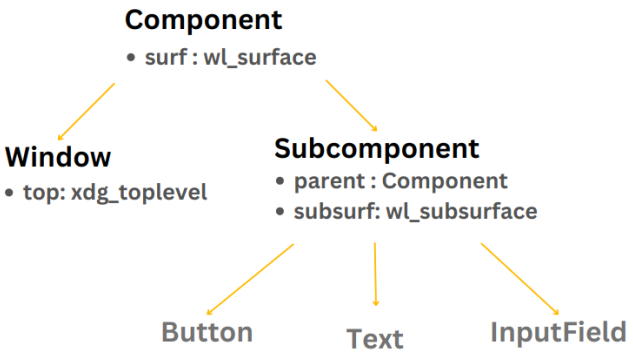

Fig. 1. Class diagram.

C. Implemented Subcomponents

1) Button Class: The Button class encapsulates interactive elements with a text label. It extends the SubComponent and handles pointer events, invoking the OnClick action. Users can create responsive buttons with a specified size and text. The implementation includes text rendering and color customization. This class is integral for constructing user-friendly interfaces with interactive elements, facilitating user engagement and interaction.

2) Text Class: The Text class facilitates the rendering and display of textual content within the GUI. Extending SubComponent, it integrates seamlessly into the hierarchical structure. Users can instantiate text elements with specific content, size, and color, contributing to diverse visual presentations. The class offers a dynamic setText method for modifying text content, font size, and color. It is a crucial component for conveying information and messages within the graphical user interface.

3) InputField Class: The InputField class represents a text input field within the GUI. Extending SubComponent, it features a dynamic text display and responds to keyboard input. Users can customize its appearance and access the input text programmatically. Implementation includes handling focus events and keyboard inputs and contributing to the creation of forms and user input areas in the GUI.

D. Actions

The Action class serves as a versatile event-handling mechanism, allowing users to define and manage callbacks. With support for multiple functions, it facilitates modular and extensible code. Users can subscribe functions to events, enhancing the decoupling of components in the GUI architecture. This class is foundational for managing various user interactions and events within the GUI toolkit.

```
template <typename... Ts>
class Action
{
    std::vector<std::function<void(Ts...)>> callbacks {};

public:
    Action() = default;
    Action(std::function<void(Ts...)> func)
    {
        callbacks.emplace_back(func);
    }

    void operator+=(const std::function<void(Ts...)>& func)
    {
        callbacks.emplace_back(func);
    }
    void operator-=(const std::function<void(Ts...)>& func)
    {
        std::erase(callbacks, func);
    }

    void operator+=(const Action& action)
    {
        callbacks.emplace_back([&action] { action(); });
    }

    void operator()(Ts... args) const
    {
        for (auto& func : callbacks)
            func(args...);
    }
};
```

III. Features

A. Structural Features

1) Hierarchical Structure: Components have the surfaces, and windows have XDG top surfaces. Subcomponents utilize subsurfaces with parent-child relations, enabling a hierarchical structure.

2) Dynamic Resizing: Components, including windows, support dynamic resizing. Text and images dynamically adjust to new sizes.

3) Image Support: Background Images: Components, including buttons, support image backgrounds. Dynamic Scaling: Images can dynamically scale based on component size, preserving aspect ratios.

4) Input Handling: Handles input events such as mouse clicks and keyboard input. Supports focus management for input fields.

5) Callback System: Utilizes the Action template class for managing and triggering callbacks. Supports flexible callback registration and execution.

6) EGL Surfaces: Utilizes EGL surfaces for rendering graphics. Drawing code example:

```
void Component::draw()
{
        if (!isActive) return;
        if (needSurfaceResize)
                forceUpdateSurfaces();
```

B. Window Featrues

1) Window Resizing and Moving: Resizable Windows: Users can dynamically resize windows by dragging their edges and adjusting both width and height. Window Movement: Windows can be moved by clicking and dragging the title bar, providing flexibility in window arrangement. Header with Buttons:

2) Default Header: Each window includes a default header containing buttons for minimizing, maximizing, and closing the window. Button-Headed Windows: Specialized windows, such as buttons, feature headers with additional buttons for specific actions.

3) Button Click Events: Buttons respond to pointer events, particularly the OnPointerDown event, allowing dynamic interactions. Color Changes: Buttons can dynamically change color, providing visual feedback to user actions. Window State Control:

4) Minimize and Maximize: Windows support minimizing and maximizing actions, enhancing user control over window visibility. Close Operation: Users can close windows using the provided close button, ensuring proper termination of application components. Input Field Focus Handling:

5) Focused Input Fields: Input fields manage focus, with visual indications when an input field is actively selected. Keyboard Interaction: Users can input text into focused input fields, with support for keyboard events such as key presses and backspace.

6) Anchors: Our GUI toolkit for Wayland comes with a handy resizing feature that makes adjusting window sizes a breeze. We've added anchor points to the corners so you can resize a window. It's a simple and intuitive way to customize window dimensions—horizontally, vertically, or both. This feature offers flexibility and interactivity.

```
void SubComponent::updateSurfacePosition() const
{
    if (!isActive) return;
    auto pivotedPos = localPos - pivot * size;
    wl_subsurface_set_position(subsurf, (int)pivotedPos.x,
                                        (int)pivotedPos.y);
}
```

```
        auto cr = cairo_create(cSurf);

        if (scaleImage)
                cairo_scale(cr, size.x / imageSize.x,
                        size.y / imageSize.y);
        cairo_set_source_surface(cr, rSurf, 0, 0);
        cairo_pattern_set_filter(cairo_get_source(cr),
                                        CAIRO_FILTER_NEAREST);
        cairo_paint(cr);

        cairo_gl_surface_swapbuffers(cSurf);
        cairo_destroy(cr);
        wl_surface_commit(surf);
}
void SubComponent::resizeRec(glm::vec2 prevContainerSize,
                             glm::vec2 newContainerSize)
{
    glm::vec2 factor = (newContainerSize - prevContainerSize)/
            prevContainerSize * (anchorsMax - anchorsMin);
    glm::vec2 newSize = targetSize + targetSize * factor;
    resize(newSize);
    this->localPos = anchoredPos + getAnchorCenter();
        updateSurfacePosition();
}
```

7) Interactive cursor: Enhancing the user experience, our implementation seamlessly integrates interactive cursor support when the cursor dynamically adapts to different elements. Whether it's resizing a window or interacting with buttons, the cursor intuitively provides visual cues. This interactive cursor support contributes to a fluid and immersive interaction with the graphical elements.

IV. Conclusion

We have implemented a revolutionary GUI Toolkit for Wayland, synthesizing the strengths of Wayland's surface-centric approach and the versatility of the Cairo library. The source files can be found on our GitHub repository [2]. Inspired by Wayland's abstractions, we introduced our new ones, like Window, Component, and SubComponent hierarchies. Our toolkit provides a dynamic resizing feature with anchor-based precision, as well as other features mentioned above.

Our toolkit provides a user-friendly and straightforward API inspired by the Qt library. Detailed documentation is available on the WikiPage [3], offering comprehensive descriptions of the primary methods. For a practical demonstration, refer to our presentation slides [4].

References

[1] Wayland Guide Book: *https://wayland-book.com/*

[2] Project's GitHub Page: *https://github.com/qLate/GUIToolkit*

[3] GUIToolkit Documentation page: *https://github.com/qLate/GUIToolkit/wiki*

[4] Project Presentation slides: *https://www.canva.com/design/DAFywPpVq7s/TgLbIp470kEJa-_T0Wrd2g/view?utm_content=DAFywPpVq7s&utm_campaign=designshare&utm_medium=link&utm_source=editor*