Universal Application Shell (UAS) Framework

Requirements Specification

Overview

A PySide6-based framework for building interactive visualization applications (geophysics, trading, 3D design, etc.) with flexible window management, asynchronous computation, and session persistence.

Architecture

Window System

Three-tier structure:

• **Session Level** - Global container for all main windows, saved to single JSON file per OS user

• **Main Window Level** - Independent top-level windows, each with configurable display mode

• **Subwindow Level** - Child windows containing actual visualizations/tools

Main Window Features

• **Multiple display modes** (independently configurable per main window, switchable at runtime):

- **MDI**: Floating subwindows within main window area

- **Tabbed**: Each subwindow in its own tab

- **SDI**: Single child subwindow fills main window

• Custom menu system (developer-defined) with subwindow creation options

• Standard UI components: status bar, toolbars, docking widgets

• Can clone itself entirely (saves state to JSON, creates new instance)

• Independent - don't communicate directly with other main windows

• Saves/restores geometry automatically

Subwindow Features

• Developer inherits base subwindow class to create custom types

• Can clone itself or create new instances within its main window

• Receives mouse events (hover, drag, click, right-click) with x,y coordinates

• **Single background thread** support for GPU/heavy computation

• Can trigger thread events at any time (not just on completion)

• Thread status visible (optional indicator, no mandatory progress dialogs)

• Saves/restores geometry and state automatically

Rendering Layer

The framework provides a layered approach for visualization rendering:

• **Default: QGraphicsView/QGraphicsScene** - Standard visualization container provided by framework

- Built-in zoom/pan/scroll capabilities
- Automatic coordinate transformations
- Good performance for most 2D visualization cases
- Easy layering of data, overlays, and annotations

• **Alternative: Custom QPainter** - Developers can override for specialized rendering

- Direct painting for real-time displays (seismic, trading charts)
- Full control over rendering pipeline

• **Advanced: QOpenGLWidget** - Developers can use for GPU-accelerated rendering

- For 3D visualizations or high-performance requirements
- Developer responsible for OpenGL implementation

Navigation Controls

Framework provides standard navigation for visualization subwindows:

• **Scroll bars** (horizontal/vertical) for panning large datasets

• **Mouse wheel zoom** centered on cursor position

• **Pan** with middle-mouse drag (or left-drag in pan mode)

• **Zoom to fit** and **zoom to selection** commands

• **View state serialization** (zoom level, pan position) for session persistence

• Navigation state automatically saved/restored with subwindow

Keyboard Shortcuts

- Framework provides **basic shortcut infrastructure**:
- Ctrl+W: Close active window
- Ctrl+S: Save session
- Standard application shortcuts
- Developers register **context-specific shortcuts** per subwindow type
- **Active subwindow gets priority** for shortcut handling
- Shortcut conflicts handled by framework (warn or override)

Factory System

- Developers create factory classes for each window type
- All factories registered globally at application startup
- Available to all main windows
- Identified by simple string names in serialized data
- **Session load fails if factory not found** (no graceful fallback in MVP)

Data Management

- **Singleton pattern** for data caches (one per data type, globally accessible)
- Subwindows responsible for managing their own data display
- Multiple subwindows can display same data across different main windows
- Example: Image cache singleton shared by all image-viewing subwindows

Event System

Mouse Events

- **Decentralized**: Only subwindow under cursor receives events
- Events include: hover, drag, click, right-click with x,y coordinates
- Subwindow processes events, may notify its main window to update status bar

Data Update Events

- **Listener/Observer pattern** between subwindows
- Each subwindow maintains dictionary: {key: [list_of_listeners]}
- When new subwindow opens: searches for relevant subwindows, enrolls as listener, announces its presence
- When data changes: subwindow notifies its listeners
- Listeners parse notifications independently and update themselves
- Can cascade: listener updates its data → notifies its own listeners

Lifecycle Events

Framework provides callbacks for developers to override:

- Window creation, activation, deactivation, close
- Resource allocation/deallocation (GPU buffers, file handles)
- Serialization/deserialization hooks

Threading Model

- Each subwindow supports **one background thread**
- Developer responsible for sub-threading if needed (rare case)
- Thread can send events to subwindow via main event loop
- **Serialization during active thread**:

- Default: blocked (avoid serialization while thread running)

- Optional: developer can enable (e.g., show "Stop thread?" dialog, return No to cancel save)

Serialization & Session Management

Session Structure

```
Session (single JSON file)
    ■■■ Session metadata (name, timestamp)
    ■■■ Main Window 1 state
    ■ ■■■ Display mode (MDI/Tabbed/SDI)
    ■ ■■■ Geometry
    ■ ■■■ Subwindow 1..N states
    ■■■ Main Window 2 state
    ■ ■■■ ...
```

Save/Load Behavior

- **Session selection** on startup: last recent, new, or older session
- **Auto-save** on program close: all main windows → single JSON
- **Window cloning**: serialize to dict → create new instance from dict
- **Reconstruction**: Framework uses factories + serialized data
- **Resource management**: Framework handles release/recreation of system resources
- Subwindows serialize independently (don't rely on parent)

Developer Responsibilities

- Inherit main window and subwindow base classes
- Create factories for custom window types
- Register factories at startup
- Implement serialization (to JSON/dict) for custom state
- Manage singleton data caches for their data types
- Implement listener enrollment and notification logic
- Override lifecycle callbacks for resource management
- Handle thread safety for background operations
- Define custom menus, toolbars, dialogs
- **Implement visualization rendering** using provided QGraphicsScene or custom rendering

- **Register context-specific keyboard shortcuts** for their subwindow types
- **Handle selection mechanisms** for data points/regions in visualizations
- Define context menus for right-click actions

Framework Responsibilities

- Provide base classes for main windows and subwindows
- Manage session save/load to single JSON file
- Handle window geometry persistence
- Provide display mode switching (MDI/Tabbed/SDI)
- Route mouse events to appropriate subwindow
- Provide lifecycle callback infrastructure
- Provide standard UI components (status bar, toolbars, docking)
- **Provide default QGraphicsView/QGraphicsScene rendering infrastructure**
- **Provide standard navigation controls** (zoom, pan, scroll bars)
- **Persist view state** (zoom level, pan position) in session
- **Manage keyboard shortcut infrastructure** and routing
- Handle resource cleanup during window transitions
- Reconstruct windows from factories + serialized data
- Fail session load if factory missing

Out of Scope for MVP

- Drag-and-drop between main windows
- Copy-paste across main windows
- Centralized error handling/logging
- Session file versioning
- Validation of session integrity
- Graceful handling of missing factories