User Interface

MuJoCo has a native UI framework. Its use is illustrated in the simulate.cc viewer. It is designed to be fast in terms of updating and rendering, easy to use both for the developer and for the user, cross-platform, and integrated with the native MuJoCo renderer. In order to achieve these design goals, we have omitted many features and customization options that are available in other UI frameworks, and instead focused on efficiency and automation.

Design overview

Native OpenGL rendering

We do not use any helper tools or libraries. Instead we provide C code for rendering all UI elements directly in OpenGL. We support multiple UIs, each of which is a virtual rectangle that can be taller than the visible window. The elements of each UI are rendered offscreen in auxiliary OpenGL buffers, via minimal updates, only when changes are necessary. At each screen refresh we then copy the pixels from these auxiliary buffers to the window framebuffer, and also implement vertical scroll bars when the window is smaller than the UI. This copy operation is done on the GPU and is very fast.

Platform abstraction

The software design has 3 layers: OpenGL rendering of UI elements working in conjunction with the MuJoCo renderer (which is fully cross-platform); abstract functions for access to windows, keyboard and mouse, defined as pure virtual functions in the `PlatformUIAdapter` class; and an implementation of those functions in the derived class `GlfwAdapter`. GLFW itself is cross-platform. Nevertheless, we have opted for this layered design in order to separate generic from platform-specific functionality. If GLFW needs to be replaced with another similar framework for some reason, only `GlfwAdapter` will need to be rewritten.

Themes and appearance

Individual UI elements do not allow customization in terms of appearance or layout. Instead, we use themes for colors and spacing, and arrang[illegible] automatically. Several built-in themes are provided and the user [illegible] custom themes, however the entire UI uses a single theme for all elements. Appearance is minimalist: mostly colored rectangles with text. Bitmaps and other

custom decorations are not supported. The UI element types are check boxes, radio button groups, selection lists, sliders, text edit boxes, static text, buttons, separators. These elements are grouped into sections which can be expanded and collapsed.

Layout and rectangles

Each UI is one virtual rectangle, whose width is determined by the theme and whose height is determined by the sections, items within each section, and also the expand/collapse state of each section. The sizes and auxiliary buffers for these virtual rectangles are handled automatically when the UI is updated. Each UI has a visible rectangle on the screen, and in addition there are other rectangles – for 3D rendering, 2D figures, and possibly custom OpenGL rendering. All these visible rectangles are saved (in mjuiState) and are used to determine where mouse events should be directed. The rectangle layout is updated by a callback provided by the user.

Static allocation and creation

Rather than allocating and deallocating a large number of objects corresponding to UI elements and linking them together, we create a single C struct (type mjUI) with static allocation supporting some maximum number of sections and elements; and then keep a record of how many are in use. UI creation is simplified by helper functions whose input is a C struct (type mjuiDef) that is essentially a table where each row describes one UI element (see below). This makes it possible to construct elaborate user interfaces with surprisingly little C code. Programmatic UI creation is also possible, for example when populating a UI with sliders corresponding to MuJoCo model joints.

Minimal state

The UI is designed to be as stateless as possible, so as to simplify development. This has two aspects. First, instead of replicating user data within the UI elements, we store pointers to user data. For example, we might create a UI slider and set its data pointer to `mjData* d->qpos+7`. This slider will visualize as well as control the 7th scalar component of the qpos vector of a MuJoCo model. Thus, when the simulation is updated, we have to remember to update the UI as well. And furthermore we have to disable UI editing when the simulation is being updated. But the advantage is that the UI becomes easier to construct, and there is no danger of discrepancies between user data and the UI. Second, the UI elements themselves are mostly stateless. Instead we keep track of a minimal set of global states, in particular mouse and keyboard state, section expand/collapse, contents of the text box being edited if any.

Automated enable and disable

While each UI item can be set in enabled or disabled state directly, we also provide automation as follows. Each UI item can be assigned an integer category. Then a mjfItemEnable callback determines whether each category should be enabled or disabled, based on some program-specific conditions. For example, sliders that can change the values of MuJoCo model joints should be disabled when the simulation state is being updated.

Main API

Click on the links below for detailed API reference of the main UI data structures and functions.

Main data structures:

- mjUI: An entire UI.
- mjuiState: Global UI state.
- mjuiDef: One entry in the definition table used for UI construction.

Main functions:

- mjui_update: Main UI update function.
- mjui_render: Renders the UI.
- mjui_event: Low-level event handler.
- mjui_add: Helper function used to construct a UI.