Dear ImGui Bundle

1. Introduction

1.a. What is Dear ImGui Bundle?

Dear ImGui Bundle is a "batteries included" framework built on Dear ImGui. It bundles 20+ libraries for plotting, markdown, node editors, 3D gizmos, and more - all working seamlessly in **C++ and Python**, on **all major platforms** (Windows, Linux, macOS, iOS, Android, WebAssembly).

If you are building scientific tools, game tools, visualization applications, developer tools, or creative apps, give it a try. You'll soon see that GUI code can be clear, readable, and easy to maintain. The immediate mode paradigm makes it a joy to reason about your app logic.

Key highlights:

- **Immediate mode**: Your code reads like a book. No widget trees, no callbacks, no synchronization headaches.
- **Cross-platform**: Same code runs on desktop, mobile, and web (via Emscripten or Pyodide).
- **Python-first**: Full Python bindings with type hints and IDE autocompletion.
- **Always up-to-date**: Tracks Dear ImGui upstream closely; Python bindings are auto-generated.

1.a.i. *Code That Reads Like a Book:*

The immediate mode paradigm means your UI code is simple and direct: the app below can be coded with just 9 readable lines of Python:

```python
from imgui_bundle import imgui, hello_imgui

selected_idx = 0
items = ["Apple", "Banana", "Cherry"]

def gui():
    global selected_idx
    imgui.text("Choose a fruit:")
    _, selected_idx = imgui.list_box("##fruits", selected_idx, items)
    imgui.text(f"You selected: {items[selected_idx]}")

hello_imgui.run(gui, window_title="Fruit Picker")
```

Published Apr 26, 2026

The relation between code and behavior is direct: what you write is what runs. There are no hidden widget trees, no callback chains, and no implicit state synchronization. This makes it easier to reason about your app's logic and flow.

Easy to understand for humans

Being able to work with readable code is getting more and more important as LLMs are now widely used to generate code: *You*, the human, still need to understand, review, and maintain that code. Immediate mode makes this easier.

Easy to understand for AI

And the same clarity that helps humans also helps AI: with no implicit state to get wrong, LLMs can read and generate ImGui code reliably. The full PDF manuals give an AI assistant all the context it needs.

> **Tip**
>
> **Try it in your browser — no install needed:** Open the Online Python Playground

---

1.a.ii. *How Does It Compare?:*

Not sure if Dear ImGui Bundle is right for you? Compare the code styles with other popular GUI libraries:

ImGui Bundle

12 lines – True immediate mode: UI declaration *is* the event handler

```
from imgui_bundle import imgui, hello_imgui

selected_idx = 0
items = ["Apple", "Banana", "Cherry"]

def gui():
    global selected_idx
    imgui.text("Choose a fruit:")
    _changed, selected_idx = imgui.list_box("##fruits", selected_idx,
items)
    imgui.text(f"You selected: {items[selected_idx]}")

hello_imgui.run(gui, window_title="Fruit Picker")
```

Strengths: Simplest code, real-time capable, runs on desktop + web (Pyodide), 20+ integrated libraries, full C++ support

Best for: Tools, visualization, games, scientific apps

Qt

31 lines – Retained mode with class hierarchy and signals/slots

```
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout,
QLabel, QListWidget

items = ["Apple", "Banana", "Cherry"]

class FruitPicker(QWidget):
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Choose a fruit:")
        self.list_widget = QListWidget()
        self.list_widget.addItems(items)
        self.result_label = QLabel(f"You selected: {items[0]}")
self.list_widget.currentRowChanged.connect(self.on_selection_changed)
        layout.addWidget(self.label)
        layout.addWidget(self.list_widget)
        layout.addWidget(self.result_label)
        self.setLayout(layout)

    def on_selection_changed(self, index):
```

```
        self.result_label.setText(f"You selected: {items[index]}")

app = QApplication([])
window = FruitPicker()
window.show()
app.exec()
```

Qt strengths: More widgets, Qt Designer, larger ecosystem, rich text, accessibility, native look

ImGui Bundle strengths: Simpler code, real-time, lighter weight, scientific viz, easier cross-compilation

Qt is Best for: Traditional business apps, enterprise software

DearPyGui

29 lines – ImGui-based but with retained-mode API and callbacks

```
import dearpygui.dearpygui as dpg

items = ["Apple", "Banana", "Cherry"]
dpg.create_context()

def on_selection_changed(sender, app_data):
    dpg.set_value("result_label", f"You selected: {app_data}")

with dpg.window(tag="Primary Window", label="Fruit Picker"):
    dpg.add_text("Choose a fruit:")
    dpg.add_listbox(items=items, callback=on_selection_changed,
num_items=len(items))
    dpg.add_text("You selected: ", tag="result_label")

dpg.create_viewport(title='Fruit Picker', width=400, height=300)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.set_primary_window("Primary Window", True)
dpg.start_dearpygui()
dpg.destroy_context()
```

DearPyGui strengths: Familiar retained-mode API, large user base, good reputation

ImGui Bundle strengths: True immediate mode, more libraries (~20), C++ support, Pyodide web support

DearPyGui is Best for: Developers who prefer retained-mode patterns

NiceGUI

15 lines – Web-based with callbacks

```
from nicegui import ui

selected_idx = -1
items = ["Apple", "Banana", "Cherry"]

def on_selection_change(e):
    global selected_idx
    selected_idx = items.index(e.value)
    selection_label.text = f"You selected: {e.value}"

ui.label("Choose a fruit:")
dropdown = ui.select(options=items, value=items[0],
on_change=on_selection_change)
selection_label = ui.label(f"You selected: {items[0]}")

ui.run(title="Fruit Picker")
```

NiceGUI strengths: Web-native, modern UI, easy deployment, familiar web paradigm, reactive

ImGui Bundle strengths: Native performance, desktop-native, offline capable, advanced widgets, lower latency

NiceGUI is Best for: Web-first apps, internal dashboards, CRUD interfaces

Gradio

18 lines – Declarative blocks with event wiring

```
import gradio as gr

items = ["Apple", "Banana", "Cherry"]
selected_item = items[0]

def on_selection_change(choice):
    global selected_item
    selected_item = choice
    return f"You selected: {choice}"

with gr.Blocks() as demo:
    gr.Markdown("# Fruit Picker")
    gr.Markdown("Choose a fruit:")
    dropdown = gr.Dropdown(choices=items, value=items[0],
label="Choose a fruit")
    output = gr.Textbox(value=f"You selected: {items[0]}",
label="Selection", interactive=False)
    dropdown.change(fn=on_selection_change, inputs=dropdown,
outputs=output)
```

```
demo.launch()
```

Gradio strengths: Web-native, ML-focused, Hugging Face integration, easy sharing, pre-built media components

ImGui Bundle strengths: Native performance, desktop-native, stateful apps, professional tools, flexibility

Gradio is Best for: ML model demos, Hugging Face Spaces, sharing with non-technical users

> **Note**
>
> These examples are available here

1.a.iii. *Who is it for?:*

- **beginners and developers**: go from idea to GUI prototype in minutes, without learning a complex framework. Deploy to almost any platform.
- **ML/AI researchers**: visualize training in real time, tune hyperparameters mid-run, inside Jupyter
- **Computer vision engineers**: inspect images and tensors at the pixel level with ImmVision
- **Robotics developers**: fast, readable debug UIs in Python or C++
- **Scientific instrument builders**: cross-platform GUIs that deploy to desktop and web from the same code
- **Technical tool makers**: build node editors, gizmos, code editors without a web stack

Who is this project not for

You should prefer a more complete framework (such as Qt for example) if your intent is to build a fully fledged application, with support for accessibility, internationalization, advanced styling, etc.

1.a.iv. *Learn More:*

- **Key Features** — Library list, FAQ, and more details.
- **Immediate Mode Explained** — Understand the paradigm that makes ImGui different.
- **Interactive Manuals & Demos** — Try the demos in your browser.
- **Hello ImGui and ImmApp** — High-level runners that take care of the app loop, windowing, and assets management, so that you can start an app with a single line of code.
- **Jupyter Notebooks** — Interactive GUIs inside Jupyter. Your training loop keeps running while you tune hyperparameters.

1.b. *Key Features*

Works everywhere:

- **Cross-platform in C++ and Python:** Works on Windows, Linux, macOS, iOS, Android, and WebAssembly!
- **Web ready**: Develop full web applications, in C++ via Emscripten; or in Python thanks to ImGui Bundle's integration within Pyodide.

First class support for Python:

- **Python Bindings:** Using Dear ImGui Bundle in Python is extremely easy and productive.
- **Well documented Python bindings and stubs:** The Python bindings stubs reflect the C++ API and documentation, serving as a reference and aiding autocompletion in your IDE. See for example the stubs for imgui, and for hello_imgui.
- Use it to create **standalone apps** (on Windows, macOS, and Linux), or to add **interactive UIs to your notebooks**. Deploy your apps **on the web** with ease, using Pyodide.

Easy to use & well documented:

- The Immediate Mode GUI (IMGUI) paradigm is simple and powerful, letting you focus on the creative aspects of your projects.
- **Easy to use, yet very powerful:** Start your first app in 3 lines.
- **Interactive Demos and Documentation:** Quickly get started with the ImGui Bundle Interactive Explorer, that showcases the capabilities of the pack. Read or copy-paste the source code (Python and C++) directly from there!

Always up-to-date:

- **Auto-generated bindings:** Python bindings are automatically generated, ensuring they stay synchronized with C++ APIs.
- **Version tracking:** ImGui Bundle version numbers match Dear ImGui releases (e.g., ImGui Bundle 1.91.x includes Dear ImGui 1.91.x). Updates typically follow within days of upstream releases.

High performance:

- **Fast:** Rendering is done via OpenGL (or any other renderer you choose), through native code.

1.b.i. *Comprehensive Library Integration:*

Dear ImGui Bundle isn't just ImGui - it's a curated ecosystem with more than 20 integrated libraries, where Each library is:

- Available in Python and C++ with consistent APIs
- Always up to date, since Python bindings are autogenerated
- Documented with interactive examples

Core Libraries:

- Dear ImGui : Bloat-free Graphical User interface with minimal dependencies
- ImGui Test Engine : Dear ImGui Tests & Automation Engine
- Hello ImGui : cross-platform Gui apps with the simplicity of a "Hello World" app

Plotting & Visualization:

- ImPlot : Immediate Mode Plotting
- ImPlot3D : Immediate Mode 3D Plotting
- ImmVision : Immediate image debugger and insights
- imgui_tex_inspect : A texture inspector tool for Dear ImGui

Text Editing & Markdown:

- ImGuiColorTextEdit : Colorizing text editor for ImGui
- imgui_md : Markdown renderer for Dear ImGui using MD4C parser

Tools:

- ImGuizmo : Immediate mode 3D gizmo for scene editing
- imgui-node-editor : Node Editor built using Dear ImGui
- NanoVG : Antialiased 2D vector drawing library on top of OpenGL

Widgets:

- ImFileDialog : A file dialog library for Dear ImGui
- portable-file-dialogs : *OS native* file dialogs library (C++11, single-header)
- imgui-knobs : Knobs widgets for ImGui
- imspinner : Set of nice spinners for imgui
- imgui_toggle : A toggle switch widget for Dear ImGui
- ImCoolBar : A Cool bar for Dear ImGui
- imgui-command-palette : A Sublime Text or VSCode style command palette in ImGui

1.b.ii. *Common Questions:*

Is it up to date?:

Yes! Because: Python bindings are auto-generated, so they stay in sync with C++

"Isn't rebuilding the UI every frame slow?":

No! Because:

- Widget calls are cheap (just generate drawing commands)
- Actual rendering is GPU-accelerated
- Typical frame times: < 1ms for most UIs
- Easily achieves 60+ FPS even with complex interfaces

"How's the Python performance?":

Excellent! Because:

- Each widget call crosses to C++ once per frame
- Heavy lifting (rendering) is in C++
- Python overhead is < 0.5ms per frame typically
- Real bottleneck is usually your application logic, not the GUI

"Does it work on the web?":

Yes, impressively!

- C++ arrow.r Emscripten arrow.r WebAssembly
- Python arrow.r Pyodide arrow.r WebAssembly (!!)
- Full Python runtime in browser
- Native-speed rendering via WebGL
- Check ImGui Bundle Explorer

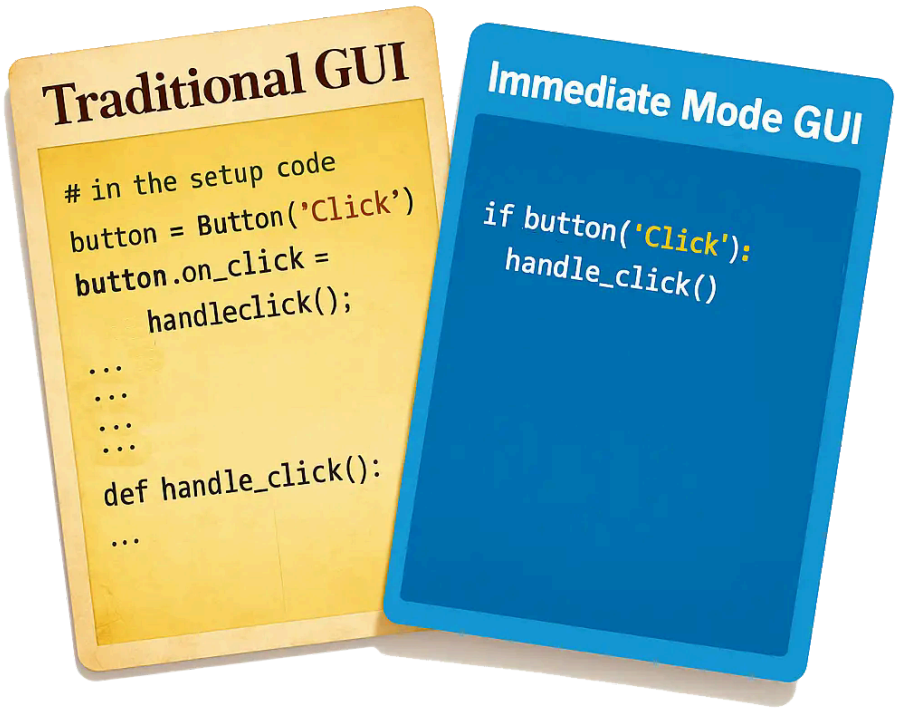
Is It Interesting for Developers?:

Absolutely yes, for several reasons:

- The immediate-mode paradigm is genuinely refreshing: your code directly expresses intent
- Rapid Development: From zero to functional UI is remarkably fast:
- The cross-platform support actually works: the same code runs on Windows, macOS, Linux. Mobile support is real (in C++), and web support via Emscripten and Pyodide also.
- Active Community: Dear ImGui itself has 70k+ stars. Dear ImGui Bundle adds comprehensive Python support, and has 1k+ stars (growing steadily).

1.c. Immediate GUI

1.c.i. *What is an Immediate GUI:*



An “Immediate Mode Graphical User Interface” lets you build user interfaces directly in code. This keeps the UI and app state in perfect sync with minimal boilerplate. This approach is especially popular for quick prototyping and tools because it’s intuitive, flexible, easy to maintain, and trivial to debug.

The example below shows a documented example to explain the Immediate Mode GUI paradigm:

Python

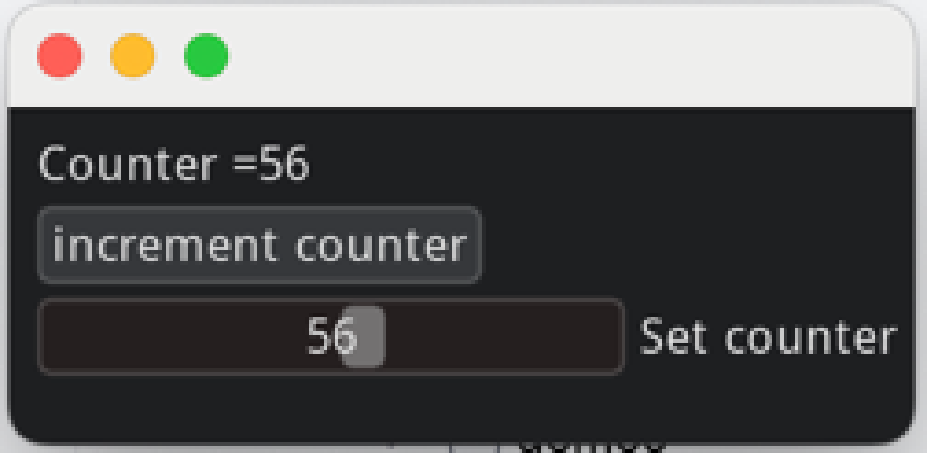
```python
from imgui_bundle import imgui, immapp

counter = 0 # our app state

# The gui() function is called every frame, so the UI updates in real
time.
def gui():
    global counter
```

```
    # The state of the UI is always in sync with the app state,
    # via standard variables: debugging UI becomes trivial!
    imgui.text(f"Counter ={counter}")

    # We can display a button, and handle its action in one line:
    if imgui.button("increment counter"):
        counter += 1
    # Below, we can also set the counter value via a slider between 0
and 100
    value_changed, counter = imgui.slider_int("Set counter", counter,
0, 100)

# Run the app (in one line!)
immapp.run(gui)
```

C++

```
#include "immapp/immapp.h"
#include "imgui.h"

int counter = 0; // our app state

// The gui() function is called every frame, so the UI updates in
real time.
void gui()
{
    // The state of the UI is always in sync with the app state,
    // via standard variables: debugging UI becomes trivial!
    ImGui::Text("Counter =%d", counter);

    // We can display a button, and handle its action in one line:
    if (ImGui::Button("increment counter"))
        counter += 1;
    // Below, we can also set the counter value via a slider between
0 and 100
    ImGui::SliderInt("Set counter", &counter, 0, 100);
}

// Run the app (in one line!)
int main(int, char **) { ImmApp::Run(gui); }
```

It produces this simple app:



Immediate Mode GUI does not mean that you cannot separate concerns!

You can still (and should) maintain a separate application state. The key difference is that your GUI can interact directly with that state in a straightforward way, without the need to maintain a separate UI state or complex event handling systems.

1.c.ii. *Dear ImGui:*

The most popular Immediate Mode GUI library is Dear ImGui, a powerful **C++** library originally created for real-time tools in game engines, now widely used in many industries, with over 60k stars on GitHub.

Dear ImGui Bundle includes Dear ImGui plus many extra libraries, making it ideal for rapid prototyping as well as building complex apps with advanced widgets, plotting, node editors; in C++ and Python.

1.c.iii. *Get started in no time with Hello ImGui and ImmApp:*

With Hello ImGui and ImmApp (both included in Dear ImGui Bundle), you can create a full-featured GUI application with just a few lines of code.

- Hello ImGui is a library based on ImGui that enables to easily create applications with ImGui. It handles window creation, backend initialization (SDL, GLFW, etc.), cross-platform assets, docking layout, and more.
- ImmApp (aka “Immediate App”, a submodule of ImGuiBundle) is a thin extension of Hello ImGui that enables to easily initialize the ImGuiBundle addons that require additional setup at startup.

Hello World in 4 lines:

4 lines are enough to start a GUI application!

Python

```python
from imgui_bundle import imgui, immapp

def gui():
    imgui.text("Hello, world!")
immapp.run(gui)
```

C++

```cpp
#include "immapp/immapp.h"
#include "imgui.h"

void gui() { ImGui::Text("Hello, world!"); }
int main() { ImmApp::Run(gui); }
```

A more complete example with plots:

The example below shows how to create a more complete application that uses an add-on (ImPlot) for plotting data.

Python

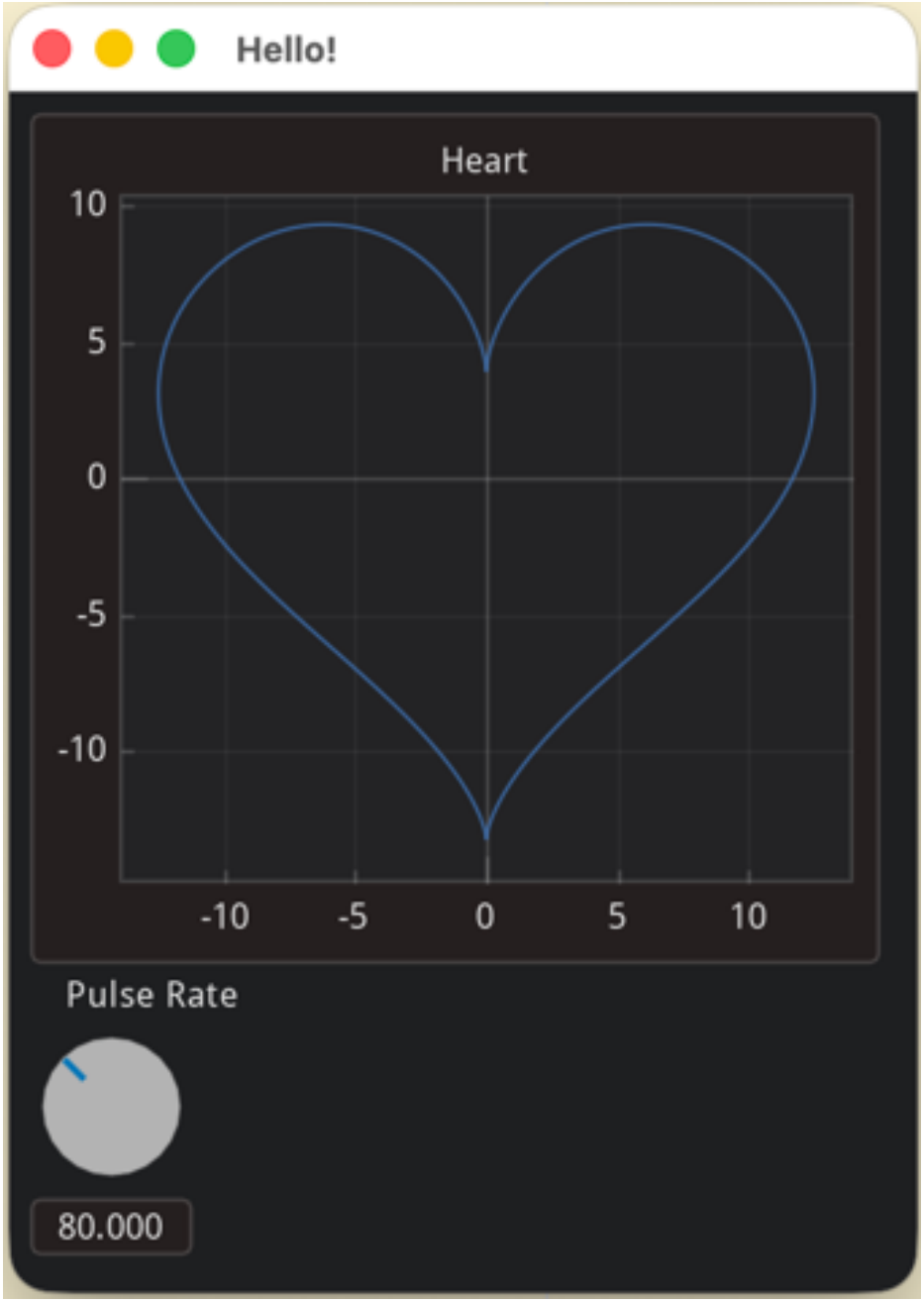
```python
import time
import numpy as np

from imgui_bundle import implot, imgui, immapp, imgui_knobs

# Fill x and y whose plot is a heart
vals = np.arange(0, np.pi * 2, 0.01)
x = np.power(np.sin(vals), 3) * 16
y = 13 * np.cos(vals) - 5 * np.cos(2 * vals) - 2 * np.cos(3 * vals) -
np.cos(4 * vals)
# Heart pulse rate and time tracking
phase = 0.0
t0 = time.time() + 0.2
heart_pulse_rate = 80


def gui():
    global heart_pulse_rate, phase, t0, x, y

    # Change heart size over time, according to the pulse rate
    t = time.time()
    phase += (t - t0) * heart_pulse_rate / (np.pi * 2)
    k = 0.8 + 0.1 * np.cos(phase)
    t0 = t

    # Plot the heart
    if implot.begin_plot("Heart", immapp.em_to_vec2(21, 21)):
        implot.plot_line("", x * k, y * k)
        implot.end_plot()
```

```
    # let the user set the pulse rate via a knob
    _, heart_pulse_rate = imgui_knobs.knob("Pulse Rate",
heart_pulse_rate, 30.0, 180.0)


if __name__ == "__main__":
    immapp.run(gui,
               window_size_auto=True,
               window_title="Hello!",
               with_implot=True,
               fps_idle=0  # Make sure that the animation is smooth
(do not limit fps when idle)
               )
```

C++

```
#include "imgui.h"
#include "implot/implot.h"
#include "imgui-knobs/imgui-knobs.h"
#include "immapp/immapp.h"
#include "hello_imgui/hello_imgui.h"

#include <cmath>

std::vector<double> VectorTimesK(const std::vector<double>& values,
double k)
{
    std::vector<double> r(values.size(), 0.);
    for (size_t i = 0; i < values.size(); ++i)
        r[i] = k * values[i];
    return r;
}

int main(int , char *[]) {
    // Fill x and y whose plot is a heart
    double pi = 3.1415926535;
    std::vector<double>  x, y; {
        for (double t = 0.; t < pi * 2.; t += 0.01) {
            x.push_back(pow(sin(t), 3.) * 16.);
            y.push_back(13. * cos(t) - 5 * cos(2. * t) - 2 * cos(3. *
t) - cos(4. * t));
        }
    }
    // Heart pulse rate and time tracking
    double phase = 0., t0 = ImmApp::ClockSeconds() + 0.2;
    float heart_pulse_rate = 80.;

    auto gui = [&]() {
        // Change heart size over time, according to the pulse rate
        double t = ImmApp::ClockSeconds();
```

```
        phase += (t - t0) * (double)heart_pulse_rate / (pi * 2.);
        double k = 0.8 + 0.1 * cos(phase);
        t0 = t;
        auto xk = VectorTimesK(x, k), yk = VectorTimesK(y, k);

        // Plot the heart
        if (ImPlot::BeginPlot("Heart", ImmApp::EmToVec2(21, 21)))
        {
            ImPlot::PlotLine("", xk.data(), yk.data(),
(int)xk.size());
            ImPlot::EndPlot();
        }

        // let the user set the pulse rate via a knob
        ImGuiKnobs::Knob("Pulse", &heart_pulse_rate, 30., 180.);
    };

    ImmApp::AddOnsParams addOnsParams{.withImplot = true};
    HelloImGui::SimpleRunnerParams runnerParams {
        .guiFunction = gui,
        .windowTitle = "Hello!",
        .windowSizeAuto = true,
        .fpsIdle = 0.f // Make sure that the animation is smooth (do
not limit fps when idle)
    };
    ImmApp::Run(runnerParams, addOnsParams);
}
```



1.c.iv. *Quickly deploy your apps on the web:*

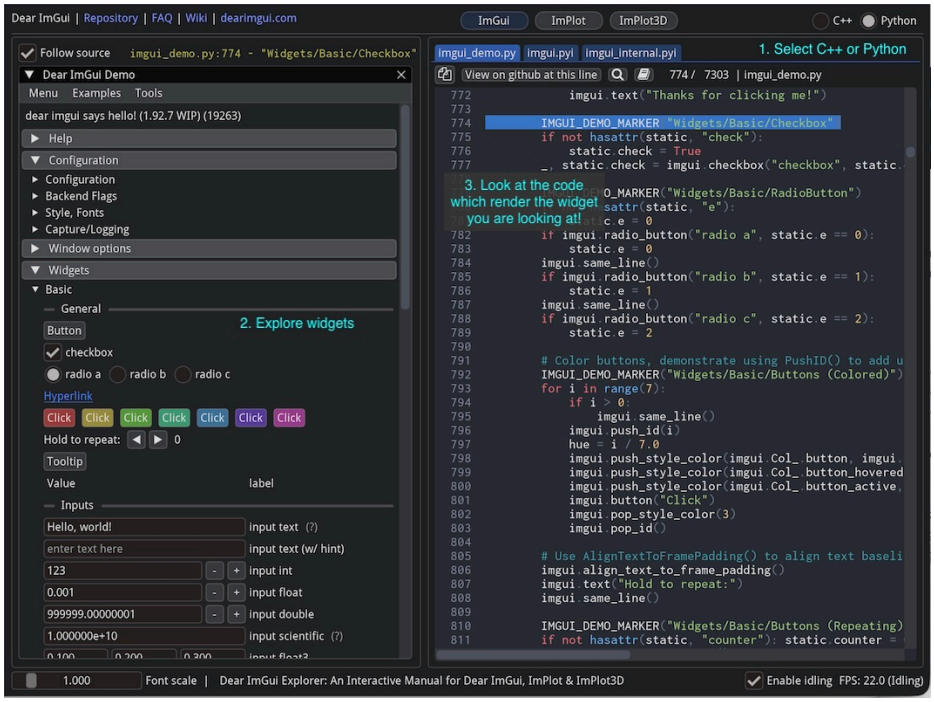
These apps can be easily deployed on the web, either in C++ via Emscripten, or in Python via Pyodide.

- Online demo (C++/Emscripten): Heart Pulse Demo
- Online demo (Python/Pyodide): Heart Pulse Demo - Pyodide, and html + python source code

1.d. Interactive Manuals

1.d.i. *Dear ImGui Explorer:*

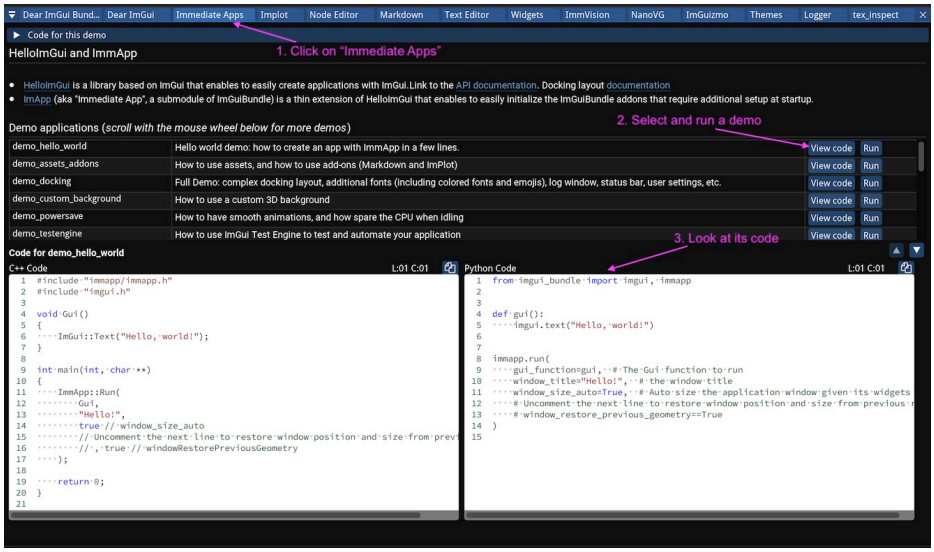
Dear ImGui Explorer lets you explore all the widgets and features of Dear ImGui, with live examples and the corresponding python or C++ code. It is built using Dear ImGui Bundle.



1.d.ii. *Dear ImGui Bundle Explorer:*

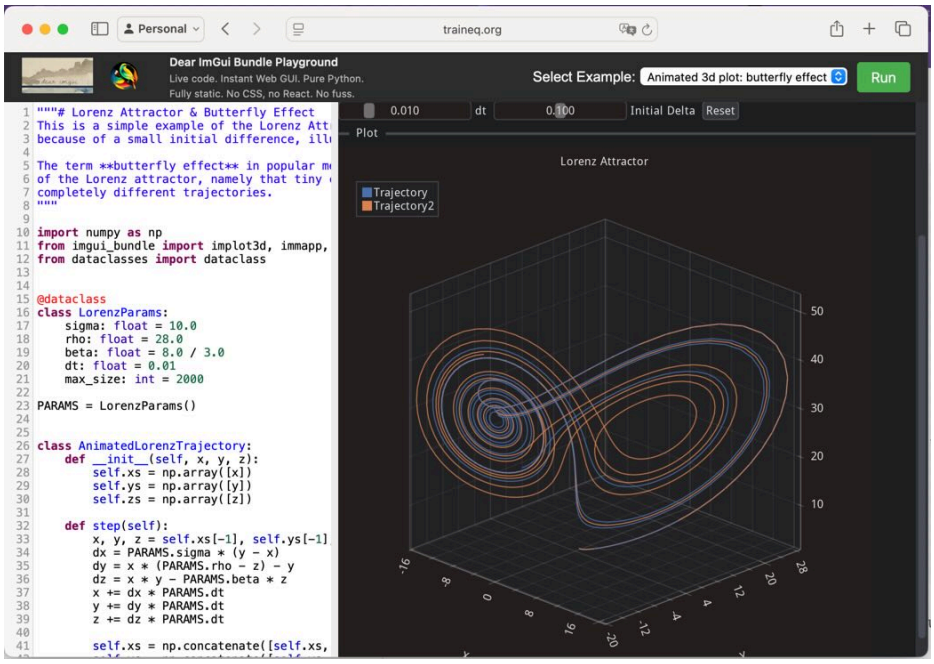
Dear ImGui Bundle Explorer: interactive reference manual - browse demos, see the code, try the widgets.

Each tab provides demos for the included libraries, along with their C++ and Python source code. The “Demo Apps” tab provides sample starter apps from which you can take inspiration.



1.d.iii. *Online Python Playground:*

Online Python Playground: an online Python sandbox with ready-to-run demos - edit code, see results instantly.



1.e. Examples and Gallery

1.e.i. *Examples in the interactive explorer:*

Below are simple example applications available in the Dear ImGui Bundle Explorer, in the “Demo Apps” tab.

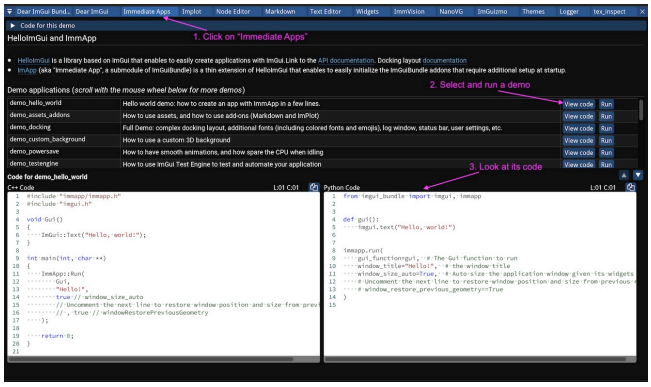


Figure 1: Inside the manual, click the “Demo Apps” tab, select a demo, run it and look at its source code.

https://imgui-bundle.pages.dev/explorer/

Complex layouts with docking windows:

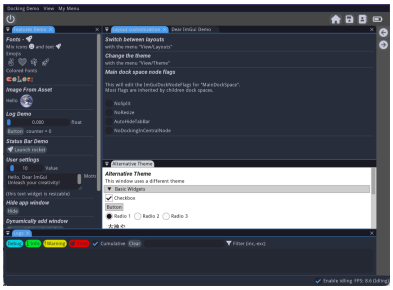


Figure 2: A complex GUI app with a docking layout, and several possible arrangements

Run this demo in your browser

This demonstration showcases how to:

- set up a complex docking layouts (with several possible layouts)
- use the status bar
- use default menus (App and view menu), and how to customize them
- display a log window
- load additional fonts
- use a specific application state (instead of using static variables)
- save some additional user settings within imgui ini file

Its source code is heavily documented and should be self-explanatory.

- C++ source code
- Python source code

Custom 3D Background:

Figure 3: A custom 3D scene rendered in the background of an ImGui application

Run this demo in your browser

This demonstration showcases how to:

- Display a 3D scene in the background via the callback `runnerParams.callbacks.CustomBackground`
- Load and compile a shader
- Adjust uniforms in the GUI

Its source code is heavily documented and should be self-explanatory.

- C++ source code
- Python source code

Display & analyze images with ImmVision:

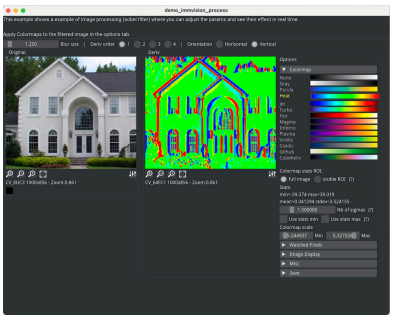


Figure 4: ImmVision in action

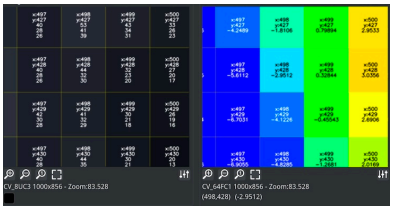


Figure 5: Zooming on the images (with the mouse wheel) to display pixel values

Run this demo in your browser

ImmVision is an immediate image debugger which can display multiple kinds of images (RGB, RGBA, float, etc.), zoom to examine precise pixel values, display float images with a versatile colormap, etc.

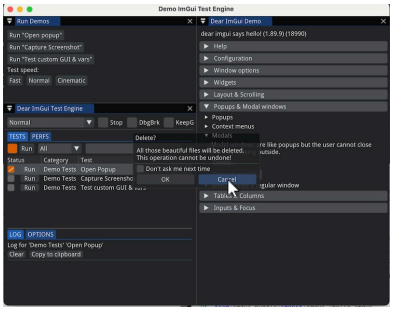
This demonstration showcases how to:

- display two versions of an image, before after an image processing pipeline
- zoom on specific ROI of those images to see pixel values
- play with the parameter of the image processing pipeline

Its source code is heavily documented and should be self-explanatory.

- C++ source code
- Python source code

Test & Automation with ImGui Test Engine:



Run this demo in your browser

ImGui Test Engine is a Tests & Automation Engine for Dear ImGui.

This demo source code is heavily documented and should be self-explanatory. It shows how to:

- enable ImGui Test Engine via RunnerParams.use_imgui_test_engine
- define a callback where the tests are registered (runner_params.callbacks.register_tests)
- create tests, and:
  - automate actions using “named references” (see Named References)
  - display an optional custom GUI for a test
- manipulate custom variables
- check that simulated actions do modify those variables

> **Note**
>
> See Dear ImGui Test Engine License. (TL;DR: free for individuals, educational, open-source and small businesses uses. Paid for larger businesses)

- C++ source code

- Python source code

1.e.ii. *Example Applications Gallery:*

More examples in the Gallery. Add yours!

4K4D:

A research project aimed for CVPR 2024, using python bindings (ImGui Bundle).

```
@inproceedings{xu20244k4d,
  title={4K4D: Real-Time 4D View Synthesis at 4K Resolution},
  author={Xu, Zhen and Peng, Sida and Lin, Haotong and He, Guangzhao and
Sun, Jiaming and Shen, Yujun and Bao, Hujun and Zhou, Xiaowei},
  booktitle={CVPR},
  year={2024}
}
```

4K4D: Real-Time 4D View Synthesis at 4K Resolution

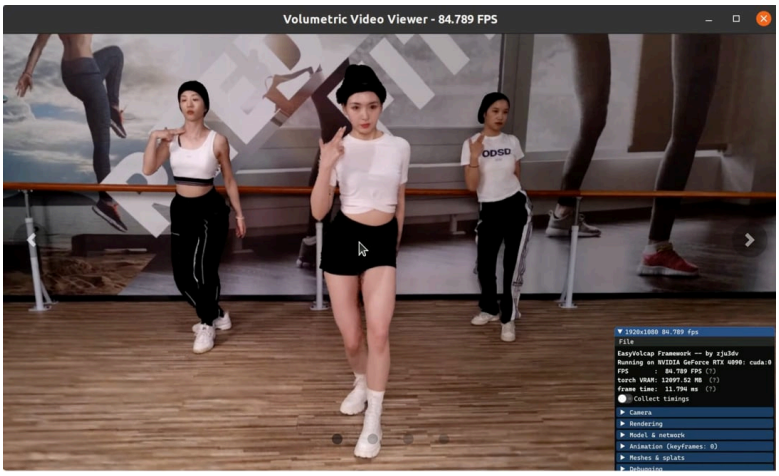


Figure 7: A volumetric video, showing an ImGui interface to control the rendering parameters.

HDRview:

HDRview is a research-oriented image viewer with an emphasis on examining and comparing high-dynamic range (HDR) images.

It is developed by Wojciech Jarosz and is built using Hello ImGui (which is included in Dear ImGui Bundle), in C++. It runs on Windows, Linux, macOS, iOS, and on the web via emscripten!

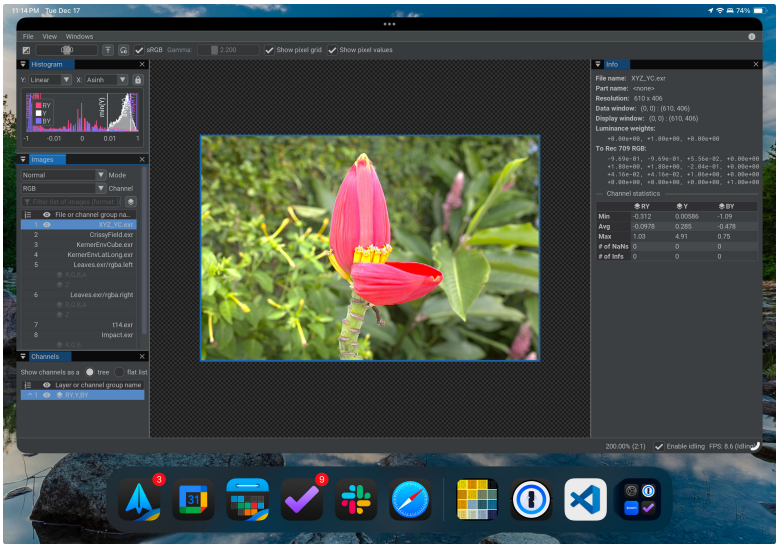


Figure 8: HDRview running on an iPad as a webapp, viewing a luminance-chroma EXR image stored using XYZ primaries with chroma subsampling.

Access HDRview online: https://wkjarosz.github.io/hdrview/

1.f. Resources

1.f.i. *Interactive demos & manuals:*

The manuals and demos below are using Dear ImGui Bundle itself!

- Dear ImGui Bundle Explorer: interactive reference manual - browse demos, see the code, try the widgets
- Online Python Playground: live Python sandbox with ready-to-run demos - edit code, see results instantly
- Dear ImGui Explorer: interactive manual for Dear ImGui, ImPlot, ImPlot3D

1.f.ii. *Community:*

- Discord: join the community for questions, showcase, and discussion
- GitHub Discussions: searchable Q&A

1.f.iii. *Documentation websites:*

- Dear ImGui Bundle documentation
- Hello ImGui documentation. Hello ImGui provides a simple framework to quickly create applications using Dear ImGui. It is included in Dear ImGui Bundle.
- Fiatlight documentation. Fiatlight provides automatic UI generation for functions and structured data (dataclasses, pydantic models), making it a powerful tool for rapid prototyping and application development. It is built on top of Dear ImGui Bundle.

1.f.iv. *YouTube Playlist:*

A series of video tutorials about Dear ImGui Bundle, Hello ImGui and Fiatlight:

- Dear ImGui Bundle - YouTube Playlist

1.f.v. *DeepWiki:*

: DeepWiki makes it easy to search and ask questions about Dear ImGui Bundle and its integrated libraries. It is generated by AI, based on the full source code and documentation. Quite useful, even if some inconsistencies may occur.

(for Dear ImGui - Dear ImGui Bundle - Hello ImGui)

1.f.vi. *Repositories:*

- Dear ImGui official repository
- Dear ImGui Bundle repository

- Hello ImGui repository
- Litgen (bindings generator) repository
- Fiatlight repository

1.f.vii. *Full PDF manuals for LLMs:*

You may feed the manuals below to a LLM, so that it can help you when using the libraries.

- Hello ImGui manual (full pdf)
- ImGui Bundle manual (full pdf)
- Fiatlight manual (full pdf)

2. For Python users

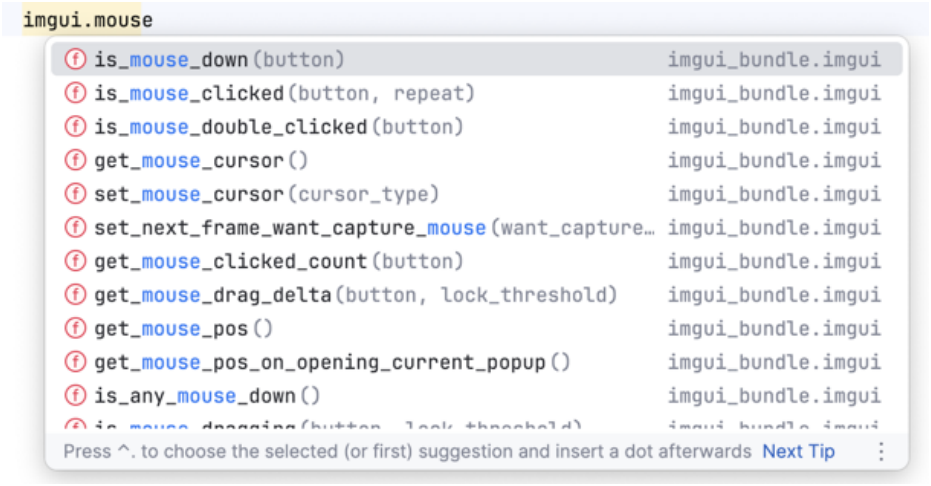
2.a. Introduction

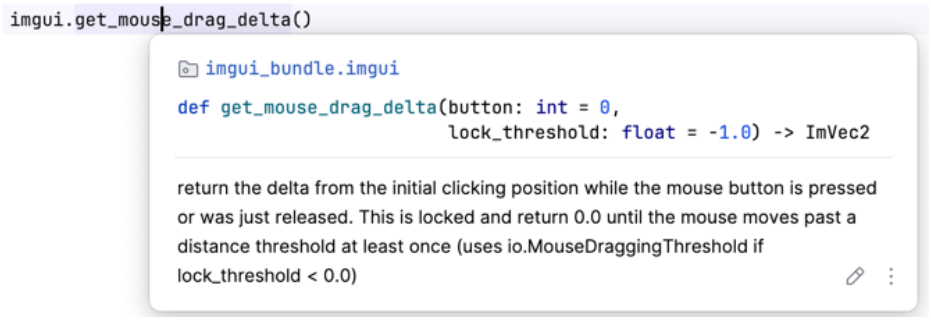
2.a.i. *Immediate GUI in Python with Dear ImGui Bundle:*

The most popular Immediate Mode GUI library is Dear ImGui, a powerful **C++** library originally created for real-time tools in game engines, now widely used in many industries, with over 60k stars on GitHub.

For Python, Dear ImGui Bundle brings full Dear ImGui support plus many extra libraries, making it ideal for rapid prototyping as well as building complex apps with advanced widgets, plotting, node editors, and more.

The python bindings are heavily documented so that they are easy to browse. They are also autogenerated, so that they are always up-to-date.





2.a.ii. *Anatomy of an application with Dear ImGui Bundle:*

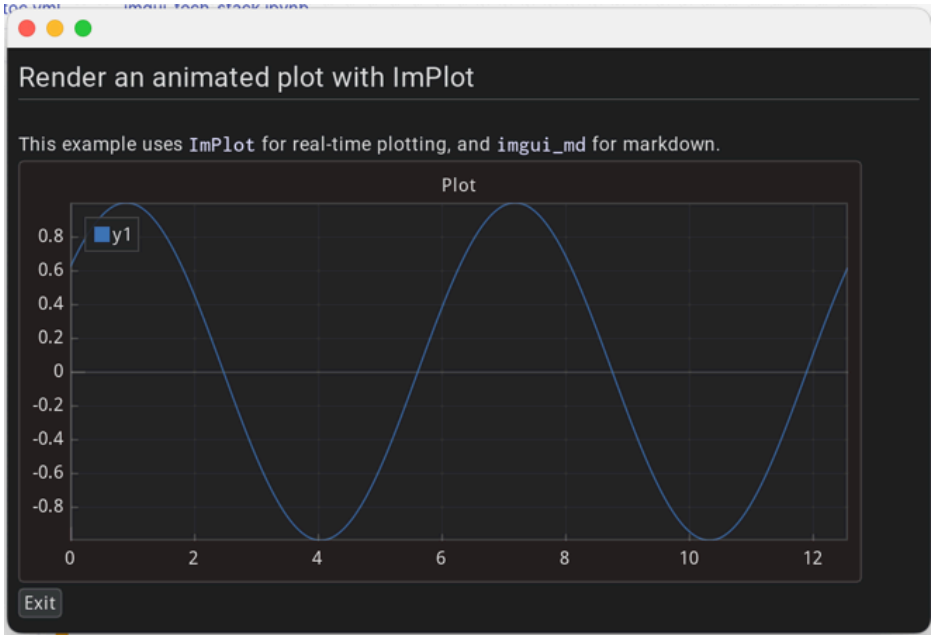
`imgui_bundle` is a Python package that unifies multiple Dear ImGui-related submodules:

- `imgui`: the core Dear ImGui library
- `implot` and `implot3d`: for advanced, real-time plotting
- `imgui_md`: markdown rendering for imgui
- `hello_imgui`: an approachable starter kit for new apps

- `immapp`: helper to activate "addons" (like implot, markdown, etc.)
- Plus about 20 other powerful tools

The example below is heavily commented and shows how to create a simple app that combines Markdown text and an animated plot using `implot`:

```
import numpy as np
from imgui_bundle import imgui, implot, imgui_md, hello_imgui, immapp

def gui():
    # Render Markdown text
    imgui_md.render_unindented("""
    # Render an animated plot with ImPlot
     This example uses `ImPlot` for real-time plotting, and `imgui_md`
for markdown.
    """)

    # Render an animated plot (updates every frame)
    if implot.begin_plot(
            title_id="Plot",
            # size in em units (1em = height of a character)
            size=hello_imgui.em_to_vec2(40, 20)):
        x = np.arange(0, np.pi * 4, 0.01)
        y = np.cos(x + imgui.get_time())
        implot.plot_line("y1", x, y)
        implot.end_plot()

    if imgui.button("Exit"):
        hello_imgui.get_runner_params().app_shall_exit = True

# Run the app with ImPlot and markdown support
immapp.run(gui,
           with_implot=True,
           with_markdown=True,
           window_size=(700, 500))
```



2.a.iii. *Deploy your applications:*

Dear ImGui Bundle apps are highly portable—they can run as standalone Python scripts, in Jupyter notebooks, or even directly in web browsers via Pyodide.

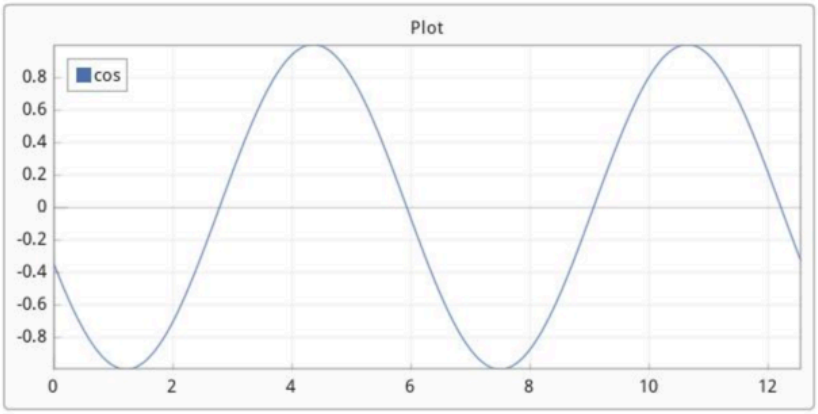
- **Standalone scripts:** Run on any PC (Windows, macOS, Linux) with minimal setup.
- **Jupyter notebooks:** The app runs in a separate window, and a screenshot is displayed in the notebook after closing (requires running Jupyter locally).
- **Web (Pyodide):** No server or installation required—just a static HTML file. Your Python app runs in the browser, with the package downloaded from a CDN.

Code | Go To

ImGui Bundle in a notebook

```python
import numpy as np
from imgui_bundle import imgui, implot, imgui_md, hello_imgui, immapp

def gui():
    if implot.begin_plot(title_id="Plot",size=hello_imgui.em_to_vec2(40, 20)):
        x = np.arange(0, np.pi * 4, 0.01)
        implot.plot_line("cos", x, np.cos(x + imgui.get_time()))
        implot.end_plot()

immapp.run(gui, with_implot=True)

```

[5] 2s 152ms



2.b. Install for Python

2.b.i. *Install from pypi:*

```
# Minimal install
pip install imgui-bundle

# or to get all optional features:
pip install "imgui-bundle[full]"
```

Binary wheels are available for Windows, macOS and Linux. If a compilation from source is needed, the build process might take up to 5 minutes, and will require an internet connection.

Platform notes

- *Windows*: Under windows, you might need to install the msvc redist
- *macOS* : under macOS, if a binary wheel is not available (e.g. for older macOS versions), pip will try to compile from source. This might fail if you do not have XCode installed. In this case, install imgui-bundle with the following command `SYSTEM_VERSION_COMPAT=0 pip install --only-binary=:all: imgui_bundle`

2.b.ii. *Install from source:*

```
# Clone the repository
git clone https://github.com/pthom/imgui_bundle.git
cd imgui_bundle

# Build and install the package (minimal install)
pip install -v .

# or build and install the package with all optional features:
#     pip install -v ".[full]"
```

The build process might take up to 5 minutes, and will clone the submodules if needed (an internet connection is required).

2.b.iii. *Run the python demo:*

Simply run `imgui_bundle_demo`.

The source for the demos can be found inside bindings/imgui_bundle/demos_python.

TIP: Consider `imgui_bundle_demo` as an always available manual for Dear ImGui Bundle with lots of examples and related code source.

2.b.iv. *Next Steps:*

- **Dear ImGui Basics** – Learn ImGui concepts (widgets, IDs, patterns)
- **Hello ImGui & ImmApp** – App runners, configuration, DPI handling
- **Python Tips** – Context managers, C++ to Python translation, debugging
- **Assets** – Managing fonts, images, and resources

2.c. Assets folder

(for python)

hello_imgui and immapp applications rely on the presence of an `assets/` folder.

This folder stores:

- Default fonts used by the markdown renderer (if the markdown addon is used).
- All the resources (images, fonts, etc.) used by the application. Feel free to add any resources there!

Assets folder location

Place the assets folder in the same folder as the script.

If needed, change the assets folder location:

Call `hello_imgui.set_assets_folder()` at startup.

Typical layout of the assets folder

```
assets/
    +-- fonts/
    |    +-- DroidSans.ttf           # Default fonts used by HelloImGui
to
    |    +-- fontawesome-webfont.ttf  # improve text rendering (esp. on
High DPI)
    |    |                           # if absent, a default LowRes font
is used.
    |    |
    |    +-- Roboto/                 # Optional: fonts for markdown
    |         +-- LICENSE.txt
    |         +-- Roboto-Bold.ttf
    |         +-- Roboto-BoldItalic.ttf
    |         +-- Roboto-Regular.ttf
    |         +-- Roboto-RegularItalic.ttf
    |         +-- Inconsolata-Medium.ttf
    +-- images/
         +-- markdown_broken_image.png  # Optional: used for markdown
         +-- world.png                  # Add anything in the assets
folder!
```

Note: in C++, the assets folder also contains an `app_settings` folder, which contains application settings and app icons for different platforms. This is not needed / not available in Python applications.

Where to find the default assets

You can download the default assets as a zip file.

Look at the folder imgui_bundle/bindings/imgui_bundle/assets to see its content.

2.c.i. *Troubleshooting:*

“Assets not found” or missing fonts/images:

1. Ensure the `assets/` folder is in the same directory as your script
2. If running from a different directory, set the path explicitly:

```
import hello_imgui
hello_imgui.set_assets_folder("/path/to/your/assets")
```

1. For packaged applications, ensure assets are included in your distribution

2.d. Pure Python Backends

Pure Python backends in ImGui Bundle let you use ImGui with Python-only windowing and rendering libraries instead of the default C++ GLFW+OpenGL backend. This gives you full control over the application loop but requires manual setup of windowing, input handling, and rendering.

Key Differences

- No Hello ImGui features: Pure Python backends don't provide DPI handling, themes, asset management, or other Hello ImGui conveniences
- Manual setup required: You must handle window creation, input events, and the render loop yourself
- Texture handling: Backends must implement support for dynamic fonts (ImGui 1.92+)

python_backends contains pure python backends for glfw, pyglet, sdl2 and sdl3. They do not offer the same DPI handling as HelloImGui, but they are a good starting point if you want to use alternative backends.

See examples where you will find:

- example_python_backend_glfw3.py
- example_python_backend_pygame.py
- example_python_backend_pyglet.py
- example_python_backend_sdl2.py
- example_python_backend_sdl3.py
- example_python_backend_wgpu.py

> **Note**
>
> Some other python libraries also provide ImGui integration, using Dear ImGui Bundle.
>
> See for example:
>
> - wgpu-py
> - moderngl-window

2.e. *Async Support*

(Since v1.92.6)

ImGui Bundle provides async/await support that enables **true parallel execution** of Python code alongside GUI rendering. This allows your Python computations to run at full speed while the GUI remains responsive.

Note: an async execution mode is also available for Jupyter notebooks; see Notebook Usage for details.

2.e.i. *Overview:*

`immapp.run_async()` and `hello_imgui.run_async()` function allows you to run ImGui applications asynchronously using Python's `asyncio` framework. This is particularly useful when:

- You need to perform computations while the GUI is running
- You're building data visualization dashboards with live updates
- You want to integrate ImGui into async Python applications
- You're working in Jupyter notebooks (see Notebook Usage)

2.e.ii. *Quick Example:*

Here's a simple example showing parallel execution:

```
import asyncio
import time
from imgui_bundle import immapp, imgui, hello_imgui, imgui_md


GUI_FINISHED = False
COMPUTATION_COUNT = 0
START_TIME = time.time()

def gui():
    params = hello_imgui.get_runner_params()
    idling_params = params.fps_idling
    idling_params.fps_idling_mode =
hello_imgui.FpsIdlingMode.early_return
    idling_params.vsync_to_monitor = False
    idling_params.fps_max = 60.0

    imgui.text(f"GUI FPS: {hello_imgui.frame_rate():.1f}")
    imgui.text(f"Computations per second: {COMPUTATION_COUNT /
(time.time() - START_TIME):.1f}")
    global GUI_FINISHED
    GUI_FINISHED = hello_imgui.get_runner_params().app_shall_exit


async def python_computation_loop():
    """Run computations while GUI is active."""
```

```
    """Python code which runs in parallel with the GUI!"""
    global COMPUTATION_COUNT
    while not GUI_FINISHED:
        _ = sum(range(1000)) # Do some work
        COMPUTATION_COUNT += 1
        await asyncio.sleep(0) # Yield to event loop (required for async
cooperation)


async def main():
    # Start GUI as an asyncio task (non-blocking)
    _gui_task = asyncio.create_task(immapp.run_async(gui,
window_size_auto=True))
    # Run computations in parallel
    await python_computation_loop()


if __name__ == "__main__":
    asyncio.run(main())
```

Also see `demos_immapp/demo_run_async.py`

2.e.iii. *Automatic FPS Optimization:*

`immapp.run_async` automatically adjusts FPS idling parameters to optimize performance, so that the Python loop can run at maximum speed.

The settings below are applied automatically by `immapp.run_async` to ensure that the GUI rendering returns early to Python instead of sleeping, allowing maximum parallelism between GUI rendering and Python code execution:

```
    runner_params.fps_idling.fps_idling_mode =
hello_imgui.FpsIdlingMode.early_return
    runner_params.fps_idling.vsync_to_monitor = False
    runner_params.fps_idling.fps_max = 60.0
```

2.e.iv. *Signature Patterns:*

`run_async()` supports two different ways to configure your application:

1. Simple GUI Function:

```
async def gui():
    imgui.text("Hello, World!")
    if imgui.button("Click me"):
        print("Button clicked!")

await immapp.run_async(
    gui,
    window_title="My App",
    window_size_auto=True,
    top_most=True,
    # Optional addons (immapp only)
```

```
    with_implot=True,
    with_markdown=True
)
```

2. Full RunnerParams (Maximum Control):

```
from imgui_bundle import hello_imgui, immapp

runner_params = hello_imgui.RunnerParams()
runner_params.callbacks.show_gui = gui
runner_params.app_window_params.window_title = "My App"
runner_params.imgui_window_params.show_menu_bar = True

# With immapp, you can use AddOnsParams
addons = immapp.AddOnsParams()
addons.with_implot = True
addons.with_node_editor = True


asyncio.run(immapp.run_async(runner_params, addons))
```

2.e.v. *Yielding to the Event Loop:*

In your async code, you **must** regularly yield control to the event loop to allow the GUI to render:

```
async def my_computation():
    while condition:
        # Do some work
        result = expensive_computation()

        # Yield to allow GUI rendering (critical!)
        await asyncio.sleep(0)
```

Without `await asyncio.sleep(0)`, the GUI will freeze because asyncio can't switch between tasks.

2.e.vi. *Troubleshooting:*

GUI Freezes:

Problem: The GUI becomes unresponsive during computations. **Solution**: Make sure to `await asyncio.sleep(0)` regularly in your computation loops.

Exceptions in the async GUI:

If your GUI raises an exception, it might be difficult to trace with the GUI is running in an async way.

In that case, it is recommended to first test your GUI in blocking mode using `immapp.run`, which will propagate exceptions normally. Once your GUI works in blocking mode, you can then switch to non-blocking mode (`immapp.run_async`).

2.f. Jupyter Notebook support

2.f.i. *Introduction:*

The notebook submodules (`immapp.nb` and `hello_imgui.nb`) provide convenient functions for the usage in a local jupyter notebook, with two main modes:

- **blocking mode**: other cells cannot be run in parallel. A screenshot is displayed after the application exits.
- **non-blocking mode**: other cells can be run in parallel. The application window updates live.

> **Note**
>
> *Note: Working on a remote notebook (or via Google Collab) may not work, since it requires a local X11 server (it might work if using X11 formwarding).*

2.f.ii. *Blocking mode:*

API:

`immapp.nb.run` and `hello_imgui.nb.run` functions will run a GUI application, wait for it to exit, and display a screenshot of the final application screen in the cell output.

During the application execution, other cells cannot be run.

Parameters

`immapp.nb.run` and `hello_imgui.nb.run` accept the same parameters as `immapp.run` and `hello_imgui.run`, respectively.

Optional additional parameters to controls the screenshot size (choose only one of the two):

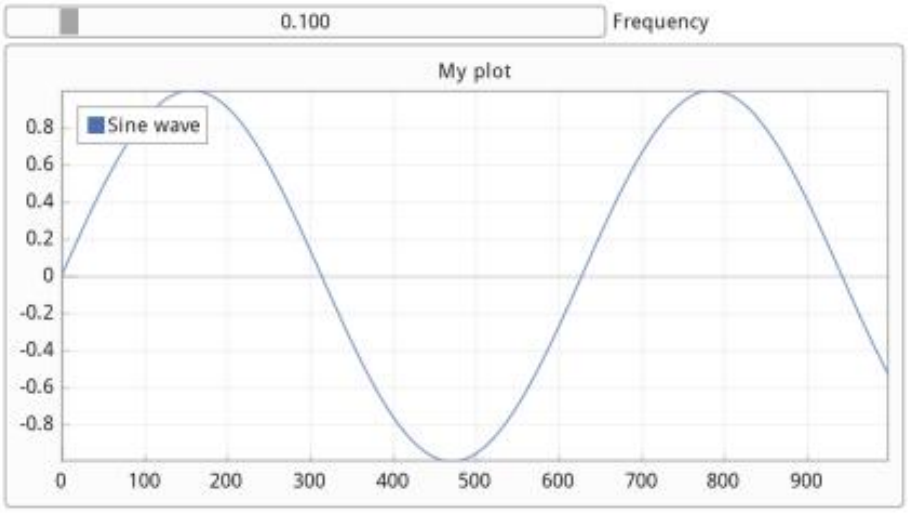
- thumbnail_ratio: (default=1.0) You can use it to change the size of the thumbnail. Passing 0.5 will create a thumbnail half the width of the window.
- thumbnail_height: (default=0) You can use it to set a fixed height for the thumbnail (in pixels). If 0, the height is computed from the app window size.

Example:

The example cell below demonstrates the blocking mode using `immapp.nb.run`. It shows a sinusoidal curve that can be adjusted with a slider. After closing the application window, a screenshot of the final state is displayed in the cell output.

```
from imgui_bundle import implot, immapp, imgui
import numpy as np

FREQ = 0.1

def gui():
```

```
    global FREQ
    _, FREQ = imgui.slider_float("Frequency", FREQ, 0.01, 1.0)
    x = np.arange(0, 100, 0.1)
    y = np.sin(FREQ * x)
    if implot.begin_plot("My plot"):
        implot.plot_line("Sine wave", y)
        implot.end_plot()


immapp.nb.run(gui, window_size=(600, 350), with_implot=True,
thumbnail_height=500)
```



2.f.iii. *Non blocking mode:*

(Since v1.92.6)

API:

start:

- `immapp.nb.start` and `hello_imgui.nb.start` will run a GUI application, display it in a top-most window on top of the browser.

Other cells can be run while the application is running. The application window will update live.

Note: these function return an `asyncio.Task`, which may be awaited or managed using asyncio.

Parameters

`immapp.nb.start` and `hello_imgui.nb.start` accept the same parameters as `immapp.start` and `hello_imgui.start`, respectively.

Optional additional parameter: `top_most` to control if the application window should stay on top of other windows.

is_running:

- `immapp.nb.is_running` and `hello_imgui.nb.is_running` return True if the application is running, False otherwise.

stop:

- `immapp.nb.stop` and `hello_imgui.nb.stop` will stop the running application.

Tip

Only one application can be run at a time from a notebook. Trying to start a new application while another one is running will exit the previous one.

Note: If other cells are running while the application is running, they should call `await asyncio.sleep(0)` periodically to allow the application to update.

Important

If your GUI raises an exception, it might be difficult to trace with the GUI is running in an async way.

In that case, it is recommended to first test your GUI in blocking mode using `immapp.nb.run`, which will propagate exceptions normally. Once your GUI works in blocking mode, you can then switch to non-blocking mode (`immapp.nb.start`).

Example:

Start the application:

The cell below demonstrates the non-blocking mode using `immapp.nb.start`. It runs the same application as before (a sinusoidal curve that can be adjusted with a slider). You can modify the frequency while the application is running by changing the value of the `FREQ` variable in another cell.

When you run it, the cell exits immediately, but the GUI application continues to show and to be interactive (you can then run other cells while the application is running).

Note: since, `immapp.nb.start` returns an `asyncio.Task`, you can see that the cell output shows the task information (Task pending, ...).

Important

In a non-blocking mode, the GUI will not be shown inside the notebook (not even as a screenshot). Instead, it will be displayed in a separate top-most window on top of the browser.

> Refer to the “video demonstration” below for a demo of how the cells below will render on your screen.

```
immapp.nb.start(gui, window_size=(500, 300), with_implot=True,
top_most=True)
```

```
<Task pending name='Task-35' coro=<run_async() running at /Users/pascal/
dvp/OpenSource/ImGuiWork/_Bundle/imgui_bundle/bindings/imgui_bundle/
immapp/run_async_overloads.py:63>>
```

Interact while the application is running:

The cell below shows that it is possible to modify the frequency via code while the application is running, and the curve updates live.

```
FREQ = 0.5  # Modify frequency while the app is running
```

Check if the application is running:

The cells below can be used to check if the application is running

```
immapp.nb.is_running()
```

```
True
```

Stop the application:

```
immapp.nb.stop()
```

Video demonstration:

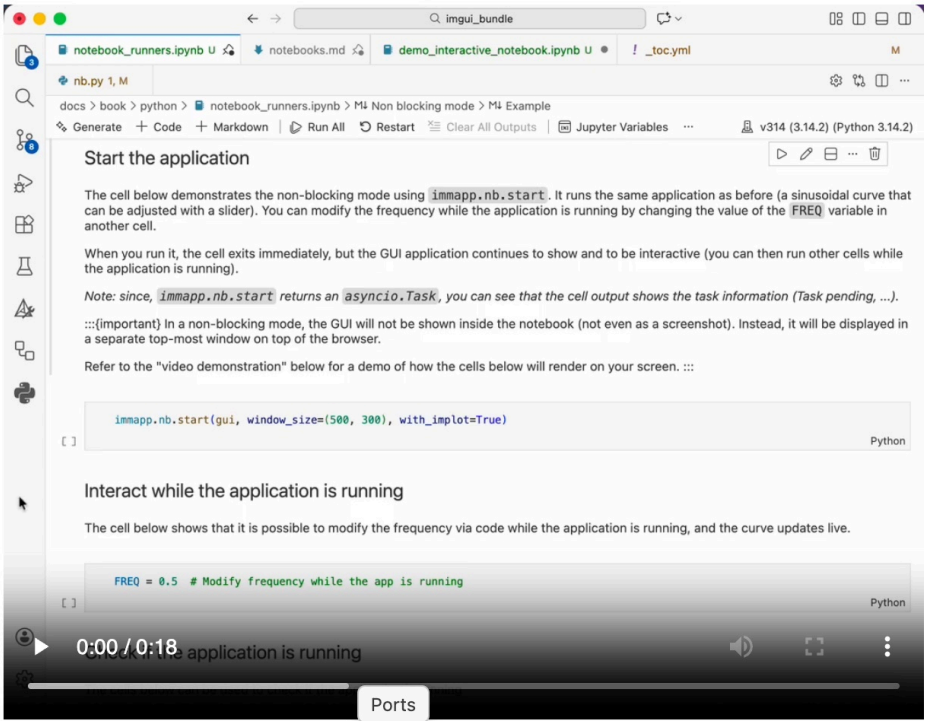


Figure 9: Demonstration of the non-blocking mode in a Jupyter notebook.

2.f.iv. *Example: Real-Time Data Stream Simulation:*

This example simulates a **live data stream** that continuously updates, like you might see in a monitoring dashboard or during ML training.

> **Tip**
>
> Refer to the "video demonstration" below for a demo of how the cells below will render on your screen.

Start the GUI:

The cell below instantiate the application data (stream_data) and starts a GUI application that displays the live data stream.

```
from imgui_bundle import immapp, imgui, hello_imgui, implot
import numpy as np
import time

# Streaming data buffer
stream_data = {
    "values": [],
    "max_points": 500,
    "paused": False
}

def streaming_gui():
    """GUI that shows a live streaming plot"""
    imgui.text("Live Data Stream")
    imgui.text(f"Points: {len(stream_data['values'])}")

    # Control buttons
    if imgui.button("Pause" if not stream_data["paused"] else "Resume"):
        stream_data["paused"] = not stream_data["paused"]

    imgui.same_line()
    if imgui.button("Clear"):
        stream_data["values"].clear()

    imgui.separator()

    # Plot the streaming data
    if len(stream_data["values"]) > 0:
        if implot.begin_plot("Data Stream", hello_imgui.em_to_vec2(40,
15)):
            x_data = np.arange(len(stream_data["values"]),
dtype=np.float32)
            y_data = np.array(stream_data["values"], dtype=np.float32)
            implot.setup_axes("x", "y", implot.AxisFlags_.auto_fit,
implot.AxisFlags_.auto_fit)
            implot.plot_line("Value", x_data, y_data)
            implot.end_plot()
```

```
    if imgui.button("Close"):
        hello_imgui.get_runner_params().app_shall_exit = True

# Start streaming GUI (note: immapp.nb.start is non-blocking
# and immediately returns an asyncio task)
immapp.nb.start(
    streaming_gui,
    window_title="Data Stream Demo",
    window_size=(800, 400),
    with_implot=True,
    top_most=True
)

print("✓ Streaming GUI started!")
print("✓ Run the next cell to start the data stream.")
```

```
✓ Streaming GUI started!
✓ Run the next cell to start the data stream.
```

Simulate Data Stream:

The cell below simulates a data stream: this will add data points while the GUI displays them in real-time.

- The GUI is already running above (in an asyncio task)
- So, we define another asyncio task to add data points (stream_data_loop below), and we run it in async way.

This cell will run for 5 seconds: while it runs, you should see the GUI updating live with new data points.

> **Important**
>
> It is important to call periodically `await asyncio.sleep(...)` in the loop, to yield control to the event loop, so that the GUI can update. You may sleep for 0 seconds if you want to yield control with the shortest possible delay. (in the example below, we sleep for 0.001 seconds to simulate a 100 Hz data stream).

```
#

import time
import random
import asyncio

async def stream_data_loop():
    print("Starting data stream... (will run for 10 seconds)")
    start_time = time.time()

    while time.time() - start_time < 5 and immapp.nb.is_running():
        if not stream_data["paused"]:
            # Add new data point
```

```
            new_value = np.sin(time.time()) + random.gauss(0, 0.1)
            stream_data["values"].append(new_value)

            # Keep buffer size limited
            if len(stream_data["values"]) > stream_data["max_points"]:
                stream_data["values"].pop(0)

        await asyncio.sleep(0.01) # Yield control to the event loop

    print(f"✓ Stream finished. Final count: {len(stream_data['values'])}
points")

# Run the streaming loop
await stream_data_loop()
```

```
Starting data stream... (will run for 10 seconds)
✓ Stream finished. Final count: 492 points
```

Video demonstration:

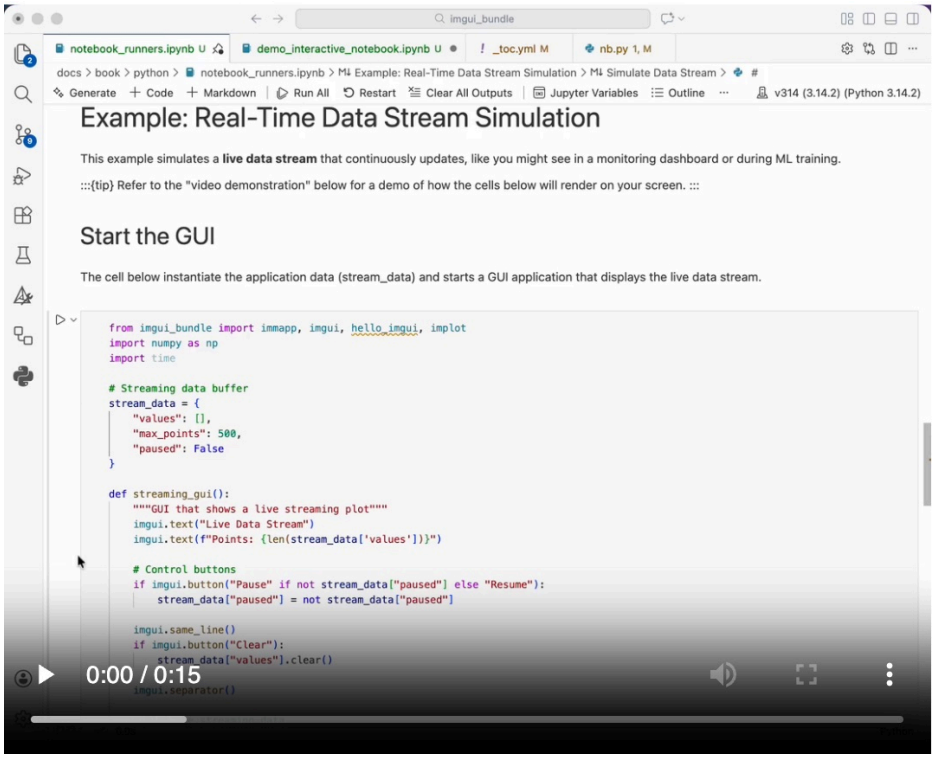


Figure 10: Demonstration of the real-time data stream simulation in a Jupyter notebook.

2.f.v. *Example: Real-Time AI Training and tuning:*

Video demonstration on youtube

Figure 11: Real-Time AI Training and tuning with Dear ImGui Bundle in Jupyter Notebooks

Links to notebooks

- notebook_ml_training_async.ipynb
- notebook_ml_training_threaded.ipynb

2.g. Deploy to the web - Pyodide

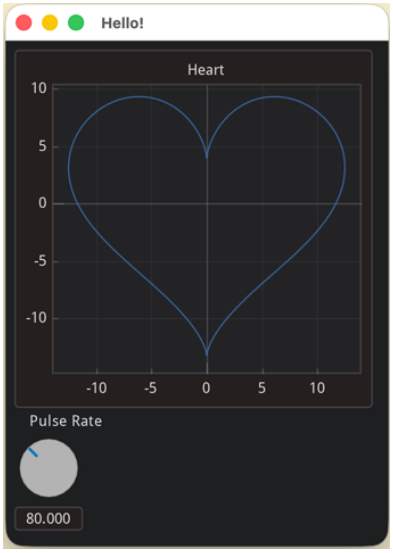
Dear ImGui Bundle applications can be effortlessly deployed to the web using Pyodide, enabling Python code to run directly in web browsers. This capability allows developers to share interactive GUI applications without requiring users to install any software.

> **Note:** Pyodide cannot use large native packages (like TensorFlow or PyTorch), and initial loading can be slow.

=== Pyodide Minimal Template

With Pyodide, web deployment is as easy as copying the HTML template below. The Python code is unchanged from what you'd use for desktop.

- HTML template source
- HTML template (run it)



Important

The HTML file must be served through a local web server (e.g. `python -m http.server`). Opening it directly via `file://` prevents package loading.

How it works:

The template is organised into four clearly marked parts.

Part 1 — Load Pyodide from a CDN

Pyodide is a Python interpreter compiled to WebAssembly; it runs directly in the browser.

```
<script src="https://cdn.jsdelivr.net/pyodide/v0.29.3/full/pyodide.
js"></script>
```

Check pyodide latest releases and update the URL accordingly.

Part 2 — Python application code

The Python code is embedded in a `<script type="text/python">` block: browsers ignore the unknown type but keep the content as `.textContent` (read from Part 4). No backtick escaping, and IDEs can still syntax-highlight the code.

```
<script type="text/python" id="pythonCode">
from imgui_bundle import imgui, immapp

def gui():
    imgui.text("Hello from Pyodide!")

if __name__ == "__main__":
    immapp.run(gui, window_title="Hello!")
</script>
```

Alternative: keep the Python in a separate file (e.g. `app.py`) and fetch it at runtime — see Part 4 below.

Part 3 — Page DOM + styles

A `<canvas>` (where the ImGui app draws) and a full-screen loader overlay with a pure-CSS spinner (shown while Pyodide and the wheel download).

Part 4 — JavaScript driver

Loads Pyodide, wires SDL to the canvas, installs `imgui_bundle` via micropip, then runs the Python code:

```
async function main() {
    const sdl2Canvas = document.getElementById("canvas");
    sdl2Canvas.addEventListener('contextmenu', e => e.preventDefault());

    // Load Pyodide
    let pyodide = await loadPyodide();

    // Setup SDL, cf https://pyodide.org/en/stable/usage/sdl.html
    pyodide.canvas.setCanvas2D(sdl2Canvas);
    pyodide._api._skip_unwind_fatal_error = true;

    // Prepare micropip
    await pyodide.loadPackage("micropip");
    const micropip = pyodide.pyimport("micropip");

    // Install imgui_bundle (two options)
    //   Option a (default): load a wheel from a local url.
    //   The wheel must match this Pyodide / Python version.
    //   By default, keep it local (same origin as this HTML). If hosted
    //   elsewhere, that server must send CORS headers.
    await micropip.install('local_wheels/imgui_bundle-1.92.601-cp313-
```

```
cp313-pyodide_2025_0_wasm32.whl');
    //   Option b: use the (older) wheel bundled with the Pyodide CDN
    // await micropip.install('imgui_bundle');

    // Load additional required packages
    await micropip.install('numpy');

    // Run the Python code
    //   Option a (default): embedded in Part 2
pyodide.runPython(document.getElementById("pythonCode").textContent);
    //   Option b: load from an external file
    // pyodide.runPython(await (await fetch('app.py')).text());
}
main();
```

Where to find Pyodide wheels for imgui_bundle:

- Release wheels — attached to each GitHub release
- Nightly builds — download a wheel directly from GitHub Actions
- Wheel used in the official demo (look at the source to find the wheel)

Wheels must match the Pyodide version **and** Python version used in the template.

> **Note**
>
> **CORS gotcha.** Passing a GitHub release URL directly to `micropip.install('https://github.com/.../wheel.whl')` looks convenient but the browser will block it: GitHub release downloads don't send `Access-Control-Allow-Origin` headers. Two workable options:
>
> - **Keep the wheel local**: download it once and serve it from the same folder as your HTML (the template's default — `local_wheels/...`).
> - **Host it somewhere CORS-friendly**: PyPI, GitHub Pages, or a CDN like jsDelivr.

2.g.i. *Pyodide API:*

In Pyodide (browser environment), `run()` behaves differently than on desktop: it starts the GUI and **returns immediately** (fire-and-forget), since browsers cannot block.

Pattern 1: Fire-and-Forget with run() (Recommended):

The simplest pattern - same code as desktop, just works:

```
from imgui_bundle import imgui, immapp

def gui():
    imgui.text("Hello from Pyodide!")
    if imgui.button("Exit"):
        from imgui_bundle import hello_imgui
```

```
        hello_imgui.get_runner_params().app_shall_exit = True

# In Pyodide: starts the GUI and returns immediately
# On desktop: blocks until GUI closes
immapp.run(gui, window_title="My App")
```

This is perfect when:

- You want the same code to work on desktop and in browser
- You don't need to do anything after the GUI closes
- You want the simplest possible code

Note: In Pyodide, `run()` returns immediately. Use `run_async()` if you need to wait for the GUI to exit.

Pattern 2: Async Control with run_async():

(since v1.92.6)

For workflows that need to wait for the GUI to exit:

```
import asyncio
from imgui_bundle import imgui, immapp

def gui():
    imgui.text("Advanced async control")

async def main():
    # Wait for GUI to exit before continuing
    await immapp.run_async(gui, window_title="My App")
    print("GUI closed")

asyncio.create_task(main())
```

Use this when:

- You need to know when the GUI exits
- You're integrating with other async code
- You want to run sequential GUI sessions

2.g.ii. *Online Python playground:*

With this online playground, you can edit and run imgui apps in the browser, without installing anything.

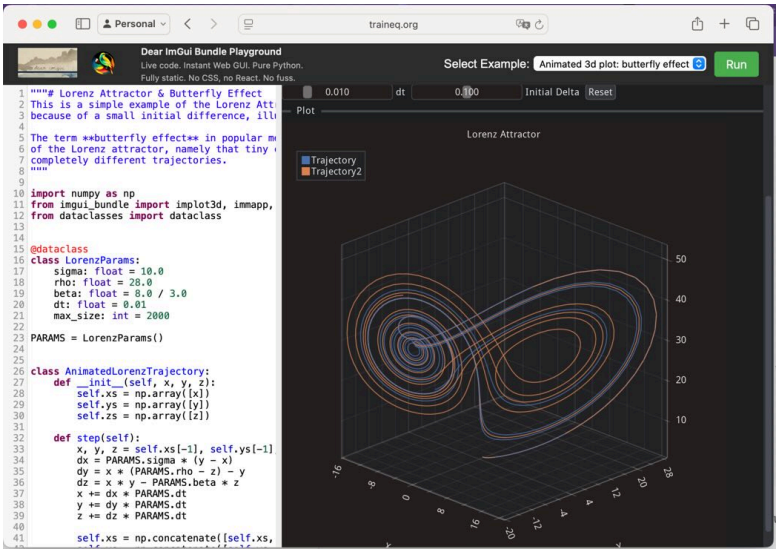


Figure 13: A browser window showing the playground: to the right an interactive demo of the butterfly effect using a 3D plot, and to the left the python code that creates it.

2.h. *Tips*

2.h.i. *Context Managers:*

In Python, the module imgui_ctx provides a lot of context managers that automatically call imgui.end(), imgui.end_child(), etc., when the context is exited, so that you can write:

```
from imgui_bundle import imgui, imgui_ctx

with imgui_ctx.begin("My Window"): # imgui.end() called automatically
    imgui.text("Hello World")
```

Of course, you can choose to use the standard API by using the module imgui:

```
imgui.begin("My Window")
imgui.text("Hello World")
imgui.end()
```

- See imgui_ctx
- See demo_python_context_manager.py

2.h.ii. *Advanced glfw callbacks:*

When using the glfw backend, you can set advanced callbacks on all glfw events.

Below is an example that triggers a callback whenever the window size is changed:

```
from imgui_bundle import glfw_utils, hello_imgui, imgui
# import glfw   # if you import glfw, do it _after_ imgui_bundle

# define a callback
def my_window_size_callback(window: glfw._GLFWwindow, w: int, h: int):
    print(f"Window size changed to {w}x{h}")


def install_glfw_callbacks():
    # Get the glfw window used by hello imgui
    glfw_win = glfw_utils.glfw_window_hello_imgui()
    glfw_utils.glfw.set_window_size_callback(glfw_win,
my_window_size_callback)


# Install the callback once everything is initialized, for example:
runner_params = hello_imgui.RunnerParams()
# ...
runner_params.callbacks.post_init = install_glfw_callbacks
```

> **Caution**
>
> It is important to import glfw after imgui_bundle, since - upon import - imgui_bundle informs glfw that it shall use its own version of the glfw dynamic library.

2.h.iii. *Display Matplotlib plots in ImGui:*

imgui_fig.py is a small utility to display Matplotlib plots in ImGui.

See demo_matplotlib.py for an example.

2.h.iv. *Read the libraries doc as a Python developer:*

General advices:

ImGui is a C++ library that was ported to Python. In order to work with it, you will often refer to its manual, which shows example code in C++.

In order to translate from C++ to Python:

1. Change the function names and parameters' names from CamelCase to snake_case
2. Change the way the output are handled.

a. in C++ ImGui::RadioButton modifies its second parameter (which is passed by address) and returns true if the user clicked the radio button.

b. In python, the (possibly modified) value is transmitted via the return: imgui.radio_button returns a Tuple[bool, str] which contains (user_clicked, new_value).

1. if porting some code that uses static variables, use the `@immapp.static` decorator. In this case, this decorator simply adds a variable value at the function scope. It is preserved between calls. Normally, this variable should be accessed via demo_radio_button.value, however the first line of the function adds a synonym named static for more clarity. Do not overuse them! Static variable suffer from almost the same shortcomings as global variables, so you should prefer to modify an application state.

Example

C++

```
void DemoRadioButton()
{
   static int value = 0;
   ImGui::RadioButton("radio a", &value, 0); ImGui::SameLine();
   ImGui::RadioButton("radio b", &value, 1); ImGui::SameLine();
   ImGui::RadioButton("radio c", &value, 2);
}
```

Python

```
@immapp.static(value=0)
def demo_radio_button():
   static = demo_radio_button
   clicked, static.value = imgui.radio_button("radio a", static.value,
0)
```

```
    imgui.same_line()
    clicked, static.value = imgui.radio_button("radio b", static.value,
1)
    imgui.same_line()
    clicked, static.value = imgui.radio_button("radio c", static.value,
2)
```

Enums and TextInput:

In the example below, two differences are important:

InputText functions:

`imgui.input_text` (Python) is equivalent to `ImGui::InputText` (C++)

- In C++, it uses two parameters for the text: the text pointer, and its length.
- In Python, you can simply pass a string, and get back its modified value in the returned tuple.

Enums handling:

- `ImGuiInputTextFlags_` (C++) corresponds to `imgui.InputTextFlags_` (python) and it is an enum (note the trailing underscore).
- `ImGuiInputTextFlags` (C++) corresponds to `imgui.InputTextFlags` (python) and it is an int (note: no trailing underscore)

You will find many similar enums.

The dichotomy between int and enums, enables you to write flags that are a combinations of values from the enum (see example below).

Example

C++

```
void DemoInputTextUpperCase()
{
    static char text[64] = "";
    ImGuiInputTextFlags flags = (
    ImGuiInputTextFlags_CharsUppercase
    | ImGuiInputTextFlags_CharsNoBlank
    );
    /*bool changed = */ ImGui::InputText("Upper case, no spaces", text,
64, flags);
}
```

Python

```
@immapp.static(text="")
def demo_input_text_decimal() -> None:
    static = demo_input_text_decimal
    flags:imgui.InputTextFlags = (
    imgui.InputTextFlags_.chars_uppercase.value
```

```
    | imgui.InputTextFlags_.chars_no_blank.value
    )
    changed, static.text = imgui.input_text("Upper case, no spaces",
static.text, flags)
```

Dear ImGui C++ vs Python API:

Dear ImGui's C++ API is thoroughly documented in its header files:

- main API
- internal API

The Dear ImGui Python API The python API closely mirrors the C++ API, and its documentation is extremely easy to access from your IDE, via thoroughly documented stub (*.pyi) files.

- main API
- internal API

2.h.v. *See Also:*

- **Dear ImGui Basics** – Widget concepts, IDs, common patterns
- **Hello ImGui & ImmApp** – App runners, DPI handling, configuration
- **Async Support** – Running GUI alongside async Python code
- **Jupyter Notebooks** – Using ImGui Bundle in notebooks

2.i. Desktop Deployment

This page covers packaging Python applications built with Dear ImGui Bundle as standalone desktop executables.

For this, you will need to use a packaging tool like **PyInstaller** to create an executable from your Python code. Other tools like cx_Freeze or Nuitka can also work.

They will allow you to distribute your application without requiring users to install Python or dependencies. They will also allow you to set an app icon, which is important for a polished user experience.

Using those packaging tools is known to work, but we do not provide official support for them.

2.i.i. *Icons: Window Icon vs App Icon:*

There are two different types of icons:

- **Window icon**: The icon shown in the window title bar and taskbar while the app is running. Works on Windows and Linux; on macOS, the window icon comes from the app bundle icon.
- **App icon**: The icon of the executable file itself (shown in file explorers, app launchers, etc.). Requires a packaging tool like PyInstaller.

Window Icon (Windows/Linux):

Place your icon at `assets/app_settings/icon.png` and make sure the assets folder is set correctly:

```
from imgui_bundle import hello_imgui, immapp
import os

def main():
    assets_path = os.path.join(os.path.dirname(__file__), "assets")
    hello_imgui.set_assets_folder(assets_path)  # or
hello_imgui.add_assets_search_path
    immapp.run(gui, window_title="My App")
```

Icon requirements:

- Square, PNG format
- At least 256x256 pixels (512x512 recommended)
- Located at `<assets_folder>/app_settings/icon.png`

This sets the **window icon** on Windows and Linux. On macOS, window icons are the same as the app icon (see below).

App Icon (packaged applications):

In Python, you will need a packaging tool like **PyInstaller** to set the app icon when creating standalone executables.

2.i.ii. *Packaging with PyInstaller:*

It is best to refer to the PyInstaller documentation for details, but here are some key points:

Example with macOS:

A complete macOS example is available in the repository: demo_packaging/macos

For windows:

For windows, you could create a spec file like this:

```
# yourapp.spec
a = Analysis(
['your_app.py'],
datas=[("assets", "assets")],  # Include assets folder
# ... other settings
)

exe = EXE(
pyz,
a.scripts,
# ... other settings
icon='path/to/your_icon.ico',  # <-- Set app icon here (Windows uses .
ico)
)
```

Then build with:

```
pip install pyinstaller
pyinstaller yourapp.spec
```

2.i.iii. *Packaging with Nuitka:*

Instructions inspired by neudinger's article

Nuitka is a Python compiler that can create standalone executables. It can be used to package Dear ImGui Bundle applications as well.

The following command compiles `your_app.py` into a standalone executable, including the assets from Dear ImGui Bundle:

```
export imgui_bundle_asset_path=`python -c "import imgui_bundle, os;
print(os.path.join(os.path.dirname(imgui_bundle.__file__), 'assets'))"`

python -m nuitka \
    --standalone \
    --include-package=imgui_bundle \
    --include-package-data=imgui_bundle \
```

```
--include-module=importlib.util \
--include-data-dir=${imgui_bundle_asset_path}=imgui_bundle/assets \
--noinclude-pytest-mode=nofollow \
--enable-plugin=no-qt \
-o your_app_compiled \
your_app.py
```

Note: Nuitka provides an option "–onefile" to create a single executable, but this will slow the startup time. It is recommended to use the default "standalone" mode, which creates a folder with the executable and dependencies, for better performance.

Do read Nuitka's documentation for more details and options: https://nuitka.net/doc/user-manual.html

3. For C++ users

3.a. C++ Installation

3.a.i. *Integrate Dear ImGui Bundle in your own project in 5 minutes:*

The easiest way to use Dear ImGui Bundle in an external project is to use the template available at https://github.com/pthom/imgui_bundle_template.

This template includes everything you need to set up your own project.

3.a.ii. *Build from source:*

If you choose to clone this repo, follow these instructions:

```
git clone https://github.com/pthom/imgui_bundle.git
cd imgui_bundle
git submodule update --init --recursive # (1)
mkdir build
cd build
cmake ..
make -j
```

(1) Since there are lots of submodules, this might take a few minutes

> **Tip**
>
> ImmVision works out of the box without OpenCV. If you need OpenCV interop (e.g. `cv::Mat` support), you can optionally pass `-DIMMVISION_FETCH_OPENCV=ON` to download and build a minimal OpenCV, or point to an existing install with `-DOpenCV_DIR=/.../path/to/OpenCVConfig.cmake`.

> **Tip**
>
> There are lots of CMake options to customize the build. See CMakeLists.txt

3.a.iii. *Run the C++ demo:*

If you built ImGuiBundle from source, Simply run build/bin/demo_imgui_bundle.

The source for the demos can be found inside bindings/imgui_bundle/demos_cpp.

> **Tip**
>
> Consider demo_imgui_bundle as a manual with lots of examples and related code source. It is always available online

3.a.iv. *Multiplatform applications:*

Hello ImGui and Dear ImGui Bundle offer excellent support for multiplatform applications (Windows, macOS, Linux, iOS, Android, and Emscripten).

See this tutorial video for Hello ImGui:

Tip

The principle with Dear ImGui Bundle is the same as described in the video, just use the dedicated Dear ImGui Bundle project template, and use `imgui_bundle_add_app` in your CMakeLists.txt.

3.b. Assets folder

(for C++)

HelloImGui and ImmApp applications rely on the presence of an `assets/` folder.

This folder stores:

- Default fonts used by the markdown renderer (if the markdown addon is used).
- All the resources (images, fonts, etc.) used by the application. Feel free to add any resources there!

Assets folder location

The assets folder should be placed in the same folder as the CMakeLists.txt for the application (the one calling imgui_bundle_add_app)

Typical layout of the assets folder

```
assets/
    +-- app_settings/                # Application settings
    |    +-- icon.png                # This will be the app icon, it
should be square
    |    |                           # and at least 256x256. It will  be
converted
    |    |                           # to the right format, for each
platform (except Android)
    |    +-- apple/
    |    |         +-- Info.plist    # macOS and iOS app settings
    |    |                           # (or Info.ios.plist +
Info.macos.plist)
    |    |
    |    +-- android/                # Android app settings: files here
will be deployed
    |    |   |-- AndroidManifest.xml # Optional manifest
    |    |   +-- res/
    |    |       +-- mipmap-xxxhdpi/ # Optional icons for different
resolutions
    |    |           +-- ...         # Use Android Studio to generate
them:
    |    |                           # right click on res/ => New >
Image Asset
    |    +-- emscripten/
    |      |-- shell.emscripten.html # Emscripten shell file
    |      |                         #   (this file will be cmake
"configured"
    |      |                         #    to add the name and favicon)
    |      +-- custom.js             # Any custom file here will be
deployed
    |                                #   in the emscripten build folder

    +-- fonts/
    |    +-- DroidSans.ttf            # Default fonts used by HelloImGui
```

```
to
    |    +-- fontawesome-webfont.ttf  # improve text rendering (esp. on
High DPI)
    |    |                            # if absent, a default LowRes font
is used.
    |    |
    |    +-- Roboto/                  # Optional: fonts for markdown
    |         +-- LICENSE.txt
    |         +-- Roboto-Bold.ttf
    |         +-- Roboto-BoldItalic.ttf
    |         +-- Roboto-Regular.ttf
    |         +-- Roboto-RegularItalic.ttf
    |         +-- Inconsolata-Medium.ttf
    +-- images/
         +-- markdown_broken_image.png  # Optional: used for markdown
         +-- world.png                  # Add anything in the assets
folder!
```

If needed, change the assets folder location:

Call `HelloImGui::SetAssetsFolder()` at startup. Or specify its location in CMake via `imgui_bundle_add_app(app_name file.cpp ASSETS_LOCATION "path/to/assets")`.

Where to find the default assets

You can download the default assets as a zip file.

Look at the folder imgui_bundle/bindings/imgui_bundle/assets to see its content.

4. Core Libraries

4.a. Dear ImGui

Dear ImGui is the foundation of ImGui Bundle. It's a bloat-free graphical user interface library for C++ that outputs optimized vertex buffers for rendering.

4.a.i. *ImGui Manual:*

The best way to learn Dear ImGui is through the interactive **ImGui Manual**:

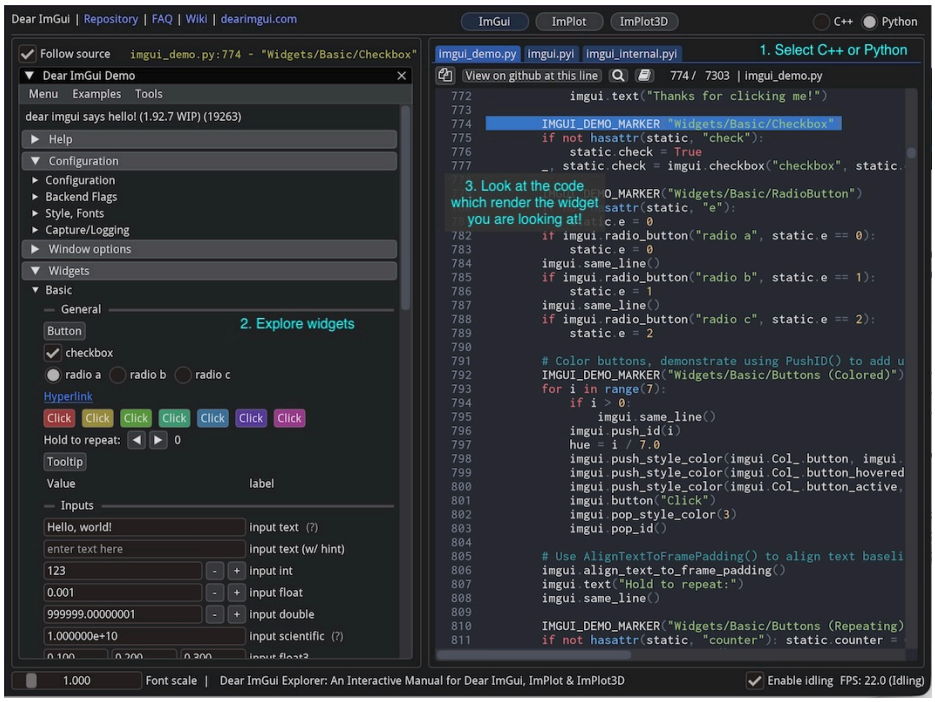


Figure 14: ImGui Manual - Interactive reference for Dear ImGui

https://pthom.github.io/imgui_explorer/

Launch the ImGui Manual

The manual lets you:

- Browse all ImGui widgets interactively
- See the corresponding C++ and Python code
- Copy code snippets directly

4.a.ii. *Basic Usage:*

Python

```python
from imgui_bundle import imgui, immapp

value = 0.5
checked = False
```

```
text = "Hello"

def gui():
    global value, checked, text

    imgui.text("Hello, world!")

    if imgui.button("Click me"):
        print("Button clicked!")

    changed, value = imgui.slider_float("Value", value, 0.0, 1.0)
    changed, checked = imgui.checkbox("Enable", checked)
    changed, text = imgui.input_text("Name", text)

immapp.run(gui, window_title="ImGui Demo")
```

C++

```
#include "immapp/immapp.h"
#include "imgui.h"

float value = 0.5f;
bool checked = false;
char text[256] = "Hello";

void gui() {
    ImGui::Text("Hello, world!");

    if (ImGui::Button("Click me")) {
        printf("Button clicked!\n");
    }

    ImGui::SliderFloat("Value", &value, 0.0f, 1.0f);
    ImGui::Checkbox("Enable", &checked);
    ImGui::InputText("Name", text, sizeof(text));
}

int main() {
    ImmApp::Run(gui, "ImGui Demo", {800, 600});
    return 0;
}
```

4.a.iii. *Key Concepts:*

Immediate Mode:

ImGui uses an **immediate mode** paradigm: you call widget functions every frame, and they return whether they were interacted with.

Python

```python
# The button returns True when clicked
if imgui.button("Save"):
    save_file()

# Sliders return (changed, new_value)
changed, value = imgui.slider_float("Speed", value, 0.0, 100.0)
if changed:
    update_speed(value)
```

C++

```cpp
// The button returns true when clicked
if (ImGui::Button("Save")) {
    save_file();
}
// Sliders modify the value in place
if (ImGui::SliderFloat("Speed", &value, 0.0f, 100.0f)) {
    update_speed(value);
}
```

Widget IDs:

ImGui identifies widgets by their label. You shall not have two widgets with the same label in the same scope.

Either use `##` to add a hidden ID suffix:

```
imgui.button("OK")
imgui.button("OK##dialog2")   # Will be displayed as "Ok", but is
different from "OK##dialog2"
```

Or add a scope using `push_id()`/`pop_id()`:

```
for i in range(3):
    imgui.push_id(i)
    imgui.button("Button")  # IDs are "0/Button", "1/Button", "2/Button"
    imgui.pop_id()
```

Begin/End Pairs:

Many ImGui functions come in pairs:

Python

```python
if imgui.begin_menu("File"):
    if imgui.menu_item("Open"):
        open_file()
    # you should call `end_*` after `begin_*`, if the `begin_*`
returned `True`.
    imgui.end_menu()
```

```
# Note: begin() is an exception, always call end(), even if begin()
returned False
if imgui.begin("My Window"):
    imgui.text("Content here")
imgui.end()  # Always call end!
```

C++

```
if (ImGui::BeginMenu("File")) {
    if (ImGui::MenuItem("Open")) {
        open_file();
    }
    ImGui::EndMenu();
}
// Note: Begin() is an exception, always call End(), even if Begin()
returned false
if (ImGui::Begin("My Window")) {
    ImGui::Text("Content here");
}
ImGui::End();  // Always call End!
```

Tip

Python users can use context managers for cleaner code:

```
from imgui_bundle import imgui, imgui_ctx

with imgui_ctx.begin("My Window") as window_opened:
    if window_opened:
        imgui.text("Content here")
# end() called automatically
```

4.a.iv. *Common Patterns:*

App State Management:

Keep your application state outside the GUI function:

```
# Good: State in a class or module-level variables
class AppState:
    counter = 0
    name = ""

state = AppState()

def gui():
    if imgui.button("Increment"):
        state.counter += 1
    _, state.name = imgui.input_text("Name", state.name)
```

Conditional Widgets:

Remember that ImGui widgets only exist when rendered:

```
# Widget only exists when show_advanced is True
if show_advanced:
    _, advanced_value = imgui.slider_float("Advanced", advanced_value,
0, 1)
```

Layout with same_line:

Use `same_line()` to place widgets horizontally:

```
imgui.button("One")
imgui.same_line()
imgui.button("Two")
imgui.same_line()
imgui.button("Three")
```

DPI-Aware Sizing (Basic):

Avoid hardcoded pixel sizes for portable UIs. Use sizes relative to the font:

```
font_size = imgui.get_font_size()
imgui.button("Click", imgui.ImVec2(font_size * 8, font_size * 2))
```

> **Tip**
>
> Prefer the more convenient `em_to_vec2()` function, available directly from `imgui_bundle`. See DPI-Aware Sizing.

4.a.v. *Documentation:*

- **ImGui Manual** - Interactive widget reference
- **Dear ImGui Repository** - Official repository with extensive documentation
- **Python API Reference:** imgui/__init__.pyi, imgui/internal.pyi
- **C++ API Reference:** imgui.h, imgui_internal.h
- Dear ImGui Support & FAQ

4.a.vi. *See Also:*

- **Hello ImGui & ImmApp** – App runners, window management, DPI handling
- **Python Tips** – Context managers, C++ to Python translation
- **Add-on Libraries** – ImPlot, ImmVision, node editors, and more

4.b. Hello ImGui - ImmApp

Hello ImGui and **ImmApp** are the two main ways to create applications with ImGui Bundle.

- **Hello ImGui** is a cross-platform framework that handles window creation, backend initialization, assets, theming, and more.
- **ImmApp** (Immediate App) is a thin wrapper around Hello ImGui that simplifies the initialization of add-ons (ImPlot, Markdown, Node Editor, etc.).

These runners enable you to create powerful ImGui applications with minimal boilerplate code.

Tip

- Use `hello_imgui.run()` for simple apps without add-ons. Use `immapp.run()` when you need add-ons like ImPlot or Markdown.
- In Python, you may also choose to use pure Python backends for full control over windowing and rendering.

Note

You may also choose to bypass these runners: use pure python backends, or integrate Dear ImGui directly in your C++ code.

4.b.i. *Hello ImGui:*

Quick Start:

Python

```python
from imgui_bundle import hello_imgui, imgui

def gui():
    imgui.text("Hello, world!")

hello_imgui.run(gui, window_title="My App", window_size=(800, 600))
```

C++

```cpp
#include "hello_imgui/hello_imgui.h"
#include "imgui.h"

void gui() {
    ImGui::Text("Hello, world!");
}

int main() {
    HelloImGui::Run(gui, "My App", {800, 600});
```

```
    return 0;
}
```

Documentation:

- **Hello ImGui Documentation** - Full documentation
- **RunnerParams Reference** - All configuration options
- **API Reference** - Full API documentation

Configuration with RunnerParams:

For full control, configure your application via `RunnerParams`:

Python

```python
from imgui_bundle import hello_imgui, imgui

def gui():
    imgui.text("Hello!")

# Create and configure runner params
params = hello_imgui.RunnerParams()
params.app_window_params.window_title = "My Application"
params.app_window_params.window_geometry.size = (1200, 800)
params.app_window_params.restore_previous_geometry = True

# ImGui window settings
params.imgui_window_params.show_menu_bar = True
params.imgui_window_params.show_status_bar = True

# Set the GUI callback
params.callbacks.show_gui = gui

# Run
hello_imgui.run(params)
```

C++

```cpp
#include "hello_imgui/hello_imgui.h"
#include "imgui.h"

void gui() {
    ImGui::Text("Hello!");
}

int main() {
    HelloImGui::RunnerParams params;
    params.appWindowParams.windowTitle = "My Application";
    params.appWindowParams.windowGeometry.size = {1200, 800};
    params.appWindowParams.restorePreviousGeometry = true;
```

```
    params.imGuiWindowParams.showMenuBar = true;
    params.imGuiWindowParams.showStatusBar = true;

    params.callbacks.ShowGui = gui;

    HelloImGui::Run(params);
    return 0;
}
```

See RunnerParams Reference for all configuration options. For Python, see RunnerParams Type Hints

Callbacks:

Hello ImGui provides several callback hooks:

Callback	When Called
`show_gui`	Every frame (main GUI)
`show_menus`	Every frame (menu bar content)
`show_status`	Every frame (status bar)
`post_init`	Once, after OpenGL initialization
`before_exit`	Once, before shutdown

See Full Callback Reference for details. For Python, see Callbacks Type Hints.

Application Settings:

ImGui applications store settings (window positions, etc.) in an INI file. By default, it's named after your window title. For production apps, use a proper config location:

```
params.ini_folder_type =
hello_imgui.IniFolderType.app_user_config_folder  # ~/.config or AppData
params.ini_filename = "my_app/settings.ini"
```

You can also store custom settings: `hello_imgui.save_user_pref("key", "value")` / `load_user_pref("key")`

DPI-Aware Sizing:

Never use fixed pixel sizes. This leads to portability issues on high-DPI screens.

Instead, use sizes relative to the font size using "em" units. Hello ImGui provides helper functions:

> **Tip**
>
> 1 em = the height of the current font. See em (typography).

Python

```python
from imgui_bundle import imgui, em_to_vec2, em_size

def gui():
    # Button sized as 10em x 2em (scales with DPI)
    imgui.button("A button", em_to_vec2(10, 2))

    # For single values, use em_size
    width = em_size(10)
```

C++

```cpp
#include "imgui.h"
#include "hello_imgui/hello_imgui.h"

void gui() {
    // Button sized as 10em x 2em (scales with DPI)
    ImGui::Button("A button", HelloImGui::EmToVec2(10, 2));

    // For single values, use EmSize
    float width = HelloImGui::EmSize(10);
}
```

Note

`em_to_vec2` and `em_size` are available:

- Directly from `imgui_bundle` (Python, since v1.92.6: `from imgui_bundle import em_to_vec2`)
- In the `hello_imgui` and `immapp` modules (Python)
- In the `HelloImGui` and `ImmApp` namespaces (C++, as `EmToVec2` and `EmSize`)

4.b.ii. *ImmApp:*

ImmApp handles add-on initialization automatically via simple boolean flags.

Quick Start:

Python

```python
from imgui_bundle import immapp, imgui, implot, imgui_md

def gui():
    imgui_md.render("# Hello with Markdown!")

    if implot.begin_plot("My Plot"):
        implot.plot_line("data", [1, 2, 3, 4], [1, 4, 2, 3])
        implot.end_plot()

# Enable add-ons with simple flags
```

```
immapp.run(
    gui,
    window_title="My App",
    window_size=(800, 600),
    with_implot=True,
    with_markdown=True
)
```

C++

```
#include "immapp/immapp.h"
#include "imgui_md_wrapper/imgui_md_wrapper.h"
#include "implot/implot.h"

void gui() {
    ImGuiMd::Render("# Hello with Markdown!");

    if (ImPlot::BeginPlot("My Plot")) {
        double x[] = {1, 2, 3, 4};
        double y[] = {1, 4, 2, 3};
        ImPlot::PlotLine("data", x, y, 4);
        ImPlot::EndPlot();
    }
}

int main() {
    HelloImGui::SimpleRunnerParams runnerParams;
    runnerParams.guiFunction = gui;
    runnerParams.windowSize = {800, 600};

    ImmApp::AddOnsParams addons;
    addons.withImplot = true;
    addons.withMarkdown = true;

    ImmApp::Run(runnerParams, addons);
    return 0;
}
```

Available Add-ons:

Flag	Add-on	Description
`with_implot`	ImPlot	2D plotting
`with_implot3d`	ImPlot3D	3D plotting
`with_markdown`	imgui_md	Markdown rendering
`with_node_editor`	imgui-node-editor	Node graphs
`with_tex_inspect`	imgui_tex_inspect	Texture inspector

Full Configuration:

For advanced configuration, use `RunnerParams` (same as Hello ImGui) combined with `AddOnsParams`:

```
immapp.run(runner_params, addons)  # Python
ImmApp::Run(runnerParams, addons);  // C++
```

4.b.iii. *Usage examples and demonstrations:*

Below are demonstrations from the ImGui Bundle Interactive Manual, showcasing various features of Hello ImGui and ImmApp.

Docking Demo:

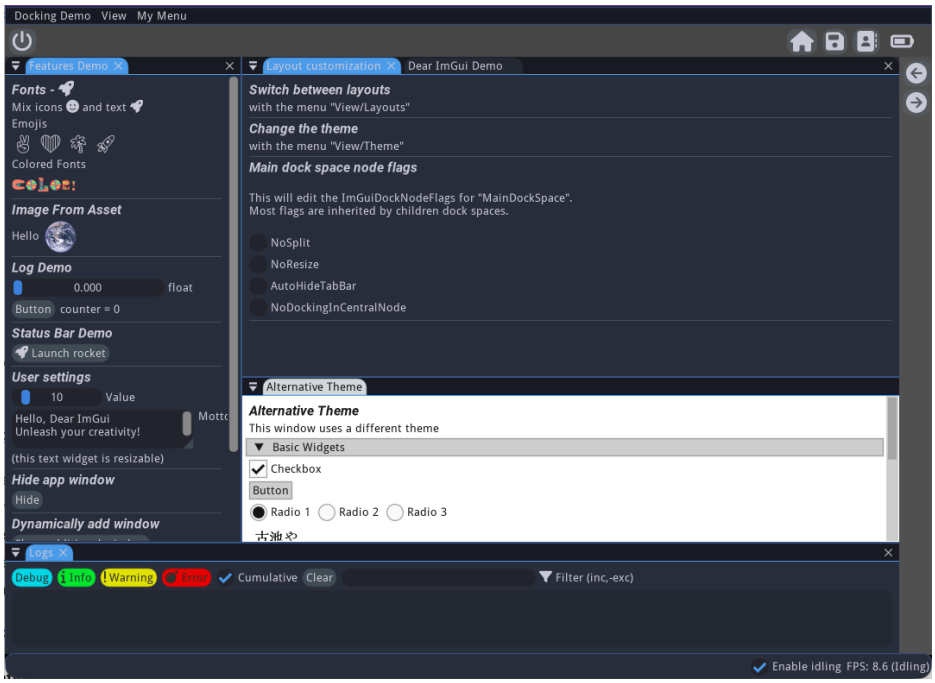


Figure 15: Docking Demo - Full-featured ImGui application with Hello ImGui

https://imgui-bundle.pages.dev/explorer/demo_docking.html

Docking Demo shows how to create a full-featured application:

- Complex app layout (with several possible layouts)
- Load additional fonts, possibly colored, and with emojis
- Display a status bar and log window
- Customize the theme
- User settings persistence

The source code is heavily documented and can be used as a template for your own applications.

Source code: Python | C++

ImmApp - Launch an app with addons:

Figure 16: ImmApp with add-ons: assets, markdown, and ImPlot

https://imgui-bundle.pages.dev/explorer/demo_assets_addons.html

Demonstrates how to use ImmApp with multiple add-ons:

- Load and display assets (images, icons)
- Render markdown content with imgui_md
- Display interactive plots with ImPlot

Source code: Python | C++

Custom 3D Background:

Figure 17: Custom 3D background with OpenGL shaders

https://imgui-bundle.pages.dev/explorer/demo_custom_background.html

Demonstrates how to render a custom 3D background using OpenGL:

- Use `runner_params.callbacks.custom_background` callback
- Load and compile shaders
- Adjust uniforms via the GUI

Source code: Python | C++

Power Save Mode:

Demonstrates FPS idling to reduce CPU usage when the app is idle.

https://imgui-bundle.pages.dev/explorer/demo_powersave.html

Hello ImGui automatically reduces FPS when no user interaction is detected. Configure this with:

```
immapp.run(gui, fps_idle=10.0)  # 10 FPS when idle

# Or dynamically:
runner_params = hello_imgui.get_runner_params()
runner_params.fps_idling.fps_idle = 10.0
runner_params.fps_idling.enable_idling = True
```

- Demo: Try online |
- Source code: Python | C++

4.b.iv. *Advanced - Manual Rendering:*

For complete control over the render loop (useful for game engines or custom frameworks), use manual rendering instead of `run()`.

Python

```python
from imgui_bundle import imgui, hello_imgui, immapp

def gui():
    imgui.text("Hello, ImGui Bundle!")

# Setup
runner_params = hello_imgui.RunnerParams()
runner_params.callbacks.show_gui = gui
addons = immapp.AddOnsParams()
addons.with_implot = True
immapp.manual_render.setup_from_runner_params(runner_params, addons)

# Custom render loop
while not hello_imgui.get_runner_params().app_shall_exit:
    immapp.manual_render.render()
    # Do other work here (physics, networking, etc.)

# Cleanup
immapp.manual_render.tear_down()
```

C++

```cpp
#include "imgui.h"
#include "hello_imgui/hello_imgui.h"
#include "immapp/immapp.h"

int main()
{
    // Setup
    HelloImGui::RunnerParams runnerParams;
    runnerParams.callbacks.ShowGui = []() {
        ImGui::Text("Hello, ImGui Bundle!");
    };
    ImmApp::AddOnsParams addons;
    addons.withImplot = true;
    ImmApp::ManualRender::SetupFromRunnerParams(runnerParams,
addons);

    // Custom render loop
    while (!HelloImGui::GetRunnerParams()->appShallExit) {
        ImmApp::ManualRender::Render();
        // Do other work here (physics, networking, etc.)
    }

    // Cleanup
    ImmApp::ManualRender::TearDown();
    return 0;
}
```

Use cases: game engine integration, heavy computation between frames, synchronizing with external systems, precise frame timing control.

Demo: Try online | Python | C++

4.b.v. *See Also:*

- **Dear ImGui Basics** – Widget concepts, IDs, common patterns
- **ImGui Test Engine** – Automated testing for ImGui apps
- **Add-on Libraries** – ImPlot, ImmVision, markdown, node editors

4.c. *ImGui Test Engine*

ImGui Test Engine is the official testing and automation framework for Dear ImGui. It enables automated UI testing, screenshot capture, and regression testing.

> **Note**
>
> **License:** Free for individuals, educational, open-source, and small business uses. Larger businesses require a paid license. See the full license.

> **Important**
>
> **Python:** Requires Hello ImGui or ImmApp – standalone usage isn't supported due to internal threading requirements.

4.c.i. *Quick Start:*

Python

```python
from imgui_bundle import imgui, hello_imgui

my_test: imgui.test_engine.Test

def my_register_tests():
    global my_test
    engine = hello_imgui.get_imgui_test_engine()
    my_test = imgui.test_engine.register_test(engine, "My Tests",
"Click Button")

    def test_func(ctx: imgui.test_engine.TestContext):
        ctx.item_click("**/My Button")

    my_test.test_func = test_func

def gui():
    test_engine = hello_imgui.get_imgui_test_engine()
    if imgui.button("My Button"):
        print("Clicked!")
    if imgui.button("Run Tests"):
        imgui.test_engine.queue_test(test_engine, my_test)

params = hello_imgui.RunnerParams()
params.use_imgui_test_engine = True
params.callbacks.register_tests = my_register_tests
params.callbacks.show_gui = gui
hello_imgui.run(params)
```

C++

```cpp
#include "hello_imgui/hello_imgui.h"
#include "imgui_test_engine/imgui_te_engine.h"
```

```cpp
#include "imgui_test_engine/imgui_te_context.h"

ImGuiTest * myTest = nullptr;

void RegisterTests() {
    ImGuiTestEngine* engine = HelloImGui::GetImGuiTestEngine();
    myTest = ImGuiTestEngine_RegisterTest(engine, "My Tests", "Click
Button");

    myTest->TestFunc = [](ImGuiTestContext* ctx) {
        ctx->ItemClick("**/My Button");
    };
}

void Gui() {
    auto testEngine = HelloImGui::GetImGuiTestEngine();
    if (ImGui::Button("My Button"))
        printf("Clicked!\n");
    if (ImGui::Button("Run Tests"))
        ImGuiTestEngine_QueueTest(testEngine, myTest);
}

int main() {
    HelloImGui::RunnerParams params;
    params.useImGuiTestEngine = true;
    params.callbacks.RegisterTests = RegisterTests;
    params.callbacks.ShowGui = Gui;
    HelloImGui::Run(params);
    return 0;
}
```

4.c.ii. *Full demo:*

For more complete examples, see the test engine demo from the interactive explorer:

- Try online
- Python
- C++

4.c.iii. *Named References:*

Test Engine uses “named references” to find widgets. The syntax uses path separators:

Pattern	Meaning
`"Button"`	Widget named “Button” in current reference
`"**/Button"`	Search for “Button” anywhere in children
`"//Window/Button"`	Absolute path from root
`"$FOCUSED"`	Currently focused window

```
ctx.set_ref("My Window")           # Set current reference
ctx.item_click("**/Save")          # Find "Save" anywhere in window
ctx.item_click("//Dialog/OK")      # Absolute path to OK button
```

4.c.iv. *Common Actions:*

```
# Click items
ctx.item_click("**/Button")
ctx.item_double_click("**/Item")

# Input text
ctx.item_input("**/Name", "John Doe")

# Check/uncheck
ctx.item_check("**/Enable")
ctx.item_uncheck("**/Disable")

# Open/close tree nodes
ctx.item_open("**/Settings")
ctx.item_close("**/Settings")
ctx.item_open_all("**/Tree")

# Screenshots
ctx.capture_screenshot_window("Window Name")

# Keyboard input
ctx.key_press(imgui.Key.enter)
ctx.key_chars("Hello")
```

4.c.v. *Assertions:*

Use the `CHECK` function to verify conditions:

```
from imgui_bundle.imgui.test_engine_checks import CHECK

def test_func(ctx: imgui.test_engine.TestContext):
    ctx.item_click("**/Increment")
    CHECK(counter == 1)  # Verify counter was incremented
```

4.c.vi. *Custom Test GUI:*

Tests can display their own GUI:

```
my_test = imgui.test_engine.register_test(engine, "Category", "Test
Name")


def test_gui_func(ctx: imgui.test_engine.TestContext):
    imgui.text("Custom test controls here")
    if imgui.button("Do Something"):
        # Custom action
        pass
```

```
my_test.gui_func = test_gui_func
```

4.c.vii. *Driving a GUI and capturing screenshots — `immapp.testing`:*

ImGui Bundle ships a small testing module that combines the test engine with screenshot capture. Use it when you want to script an interaction (click, type, expand a header) and grab a PNG at chosen moments.

- `immapp.testing.run(gui, test_fn, ...)` — Python: runs the GUI and drives it with `test_fn(ctx)`; exits once the test finishes (override with `exit_after_test=False`).
- `immapp.testing.capture(ctx, path, window=None, flags=0)` — Python: write a PNG from inside a test. Captures the full framebuffer, or a single window if `window="My Window"`.
- `immapp.testing.capture_final_frame(gui, path, ...)` — Python one-shot: run the GUI for a few frames, save the final frame. No test engine.
- `ImmApp::Testing::Capture(ctx, path, {window, flags})` — C++ equivalent of `capture`, from `immapp/testing.h`.
- `ImmApp::Testing::CaptureFinalFrame(guiFn, path, opts)` — C++ one-shot.

Example:

```
from imgui_bundle import imgui
from imgui_bundle.immapp import testing

def gui():
    imgui.button("Click me")

def my_test(ctx: imgui.test_engine.TestContext):
    testing.capture(ctx, "/tmp/00_initial.png")
    ctx.item_click("//**/Click me")
    testing.capture(ctx, "/tmp/01_after_click.png")

testing.run(gui, my_test, window_size=(600, 400))
```

See the full demo in `demo_testapp.py` and `demo_testapp.cpp`.

4.c.viii. *Running Tests:*

When test engine is enabled:

1. A "Test Engine" window appears
2. Click tests to run them
3. Watch as actions are automated
4. Check results (green = pass, red = fail)

4.c.ix. *Documentation & Resources:*

- **Test Engine Wiki** - Official documentation
- **Named References** - Path syntax reference

5. Add-on Libraries

5.a. Plotting Libraries

Dear ImGui Bundle includes two powerful plotting libraries for 2D and 3D visualization.

5.a.i. *ImPlot - 2D Plotting:*

Introduction:

ImPlot adds interactive 2D plots to your GUI: line charts, bar charts, scatter plots, heatmaps, and more. Plots support zooming, panning, and hover inspection.

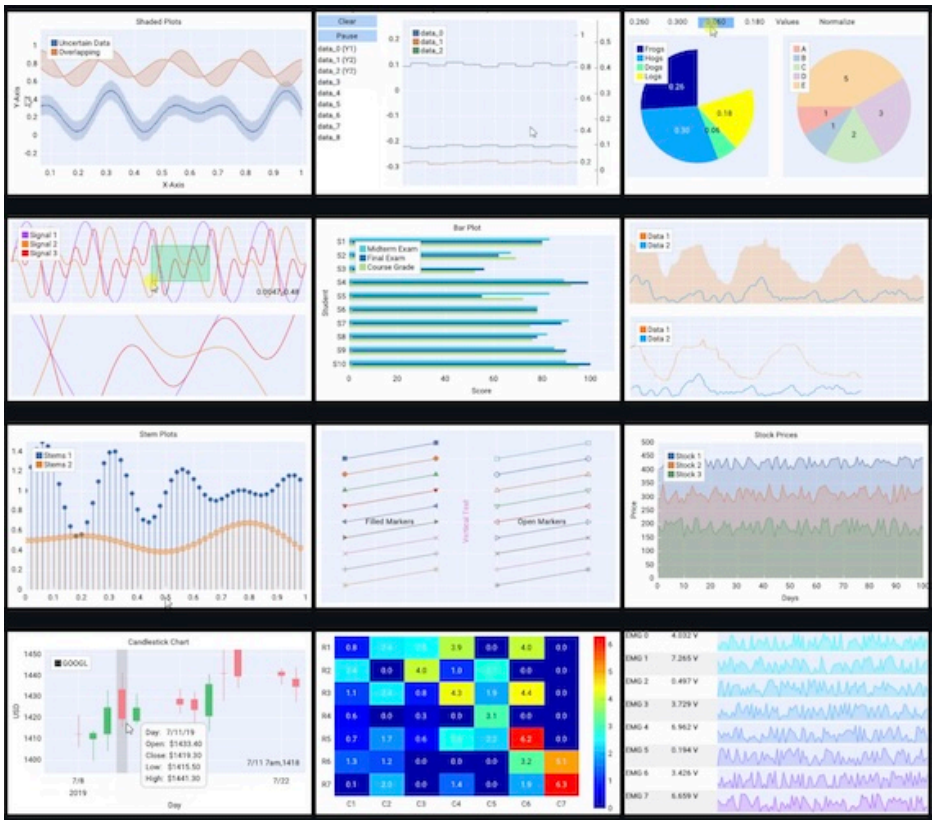


Figure 18: Some of the many plot types supported by ImPlot.

https://github.com/epezent/implot

Quick example:

Python

```python
from imgui_bundle import implot, immapp
import numpy as np

x = np.arange(0, 10, 0.1)
y = np.sin(x)
```

```python
def gui():
    if implot.begin_plot("My Plot"):
        implot.plot_line("sin(x)", x, y)
        implot.end_plot()

immapp.run(gui, with_implot=True)
```

> **Tip**
>
> Enable ImPlot by passing `with_implot=True` to `immapp.run()`.

C++

```cpp
#include "immapp/immapp.h"
#include "implot/implot.h"
#include <cmath>
#include <vector>

std::vector<double> x, y;

void gui() {
    if (ImPlot::BeginPlot("My Plot")) {
        ImPlot::PlotLine("sin(x)", x.data(), y.data(), x.size());
        ImPlot::EndPlot();
    }
}

int main() {
    for (double i = 0; i < 10; i += 0.1) {
        x.push_back(i);
        y.push_back(std::sin(i));
    }
    HelloImGui::RunnerParams runner_params;
    runner_params.callbacks.ShowGui = gui;
    ImmApp::AddOnsParams addons;
    addons.withImplot = true;
    ImmApp::Run(runner_params, addons);
    return 0;
}
```

> **Tip**
>
> In C++, enable ImPlot by setting `withImplot = true` in `ImmApp::AddOnsParams`.

Full Demo:

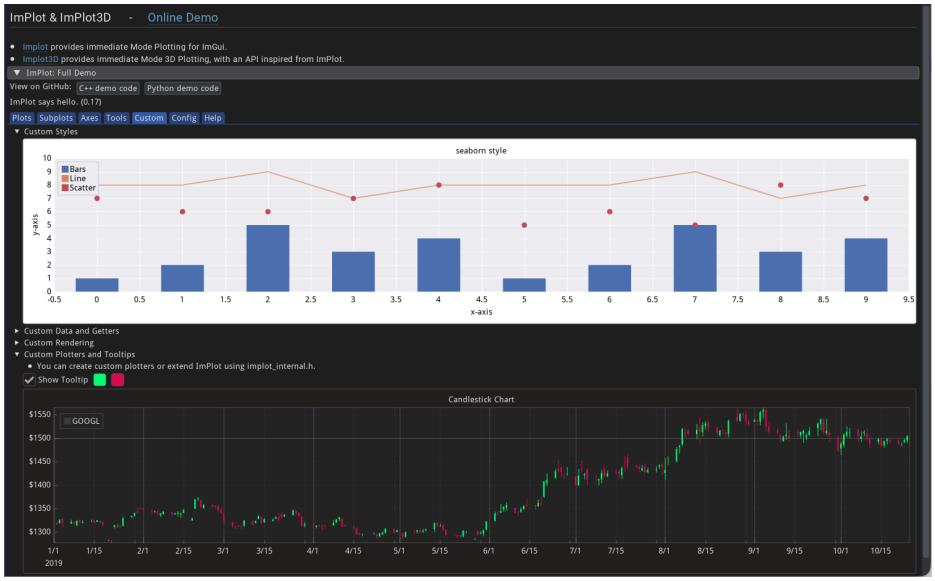


Figure 19: ImPlot demo showcasing various plot types and features.

https://imgui-bundle.pages.dev/explorer/demo_implot.html

- Try online
- Python demo code: implot_demo.py
- C++ demo code: implot_demo.cpp

Documented APIs:

- Python API reference: implot/__init__.pyi
- C++ API reference: implot.h

5.a.ii. *ImPlot3D - 3D Plotting:*

Introduction:

ImPlot3D extends ImPlot with 3D visualization capabilities. Create interactive 3D line plots, scatter plots, surface plots, and more with rotation, zoom, and pan controls.

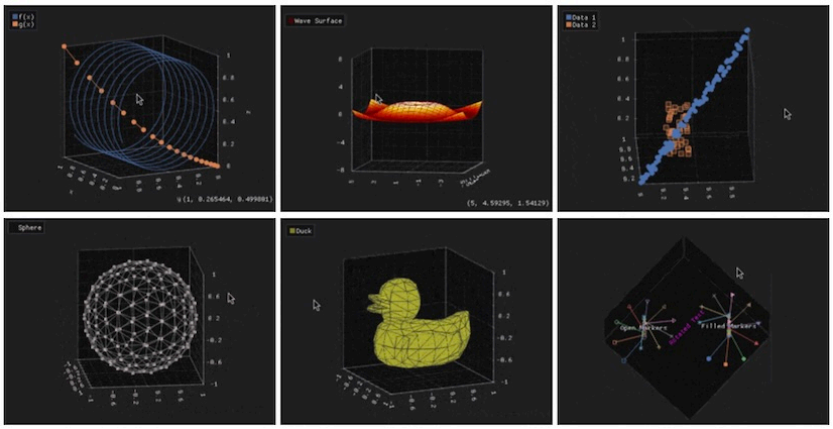


Figure 20: Some of the plot types supported by ImPlot3D.

https://github.com/brenocq/implot3d

Quick example:

Python

```python
from imgui_bundle import implot3d, immapp, hello_imgui
import numpy as np

t = np.linspace(0, 10, 100)
x, y, z = np.cos(t), np.sin(t), t

def gui():
    if implot3d.begin_plot("3D Helix", hello_imgui.em_to_vec2(30,
30)):
        implot3d.setup_axes("X", "Y", "Z")
        implot3d.plot_line("helix", x, y, z)
        implot3d.end_plot()

immapp.run(gui, with_implot3d=True)
```

> **Tip**
>
> Enable ImPlot3D by passing `with_implot3d=True` to `immapp.run()`.

C++

```cpp
#include "immapp/immapp.h"
#include "implot3d/implot3d.h"
#include <vector>
#include <cmath>

std::vector<double> x, y, z;

void gui() {
    if (ImPlot3D::BeginPlot("3D Helix")) {
        ImPlot3D::SetupAxes("X", "Y", "Z");
        ImPlot3D::PlotLine("helix", x.data(), y.data(), z.data(),
x.size());
        ImPlot3D::EndPlot();
    }
}

int main() {
    for (double t = 0; t < 10; t += 0.1) {
        x.push_back(std::cos(t));
        y.push_back(std::sin(t));
        z.push_back(t);
    }

    HelloImGui::RunnerParams runner_params;
```

```
    runner_params.callbacks.ShowGui = gui;
    ImmApp::AddOnsParams addons;
    addons.withImplot3d = true;
    ImmApp::Run(runner_params, addons);
    return 0;
}
```

Tip

In C++, enable ImPlot3D by setting `withImplot3d = true` in `ImmApp::AddOnsParams`.

Full Demo:

The full demo for ImPlot3D is available online together with ImPlot's full demo.

Try online

- Python demo code: implot3d_demo.py
- C++ demo code: implot3d_demo.cpp

Documented APIs:

- Python API reference: implot3d/__init__.pyi
- C++ API reference: implot3d.h

5.b. Image Visualization

Dear ImGui Bundle includes specialized tools for image inspection and debugging.

5.b.i. *ImmVision - Image Viewer Widget:*

Introduction:

ImmVision provides an interactive image viewer widget for ImGui applications, with zoom, pan, pixel inspection, and colormap support. Useful for building image processing tools and debugging computer vision pipelines from within your application.

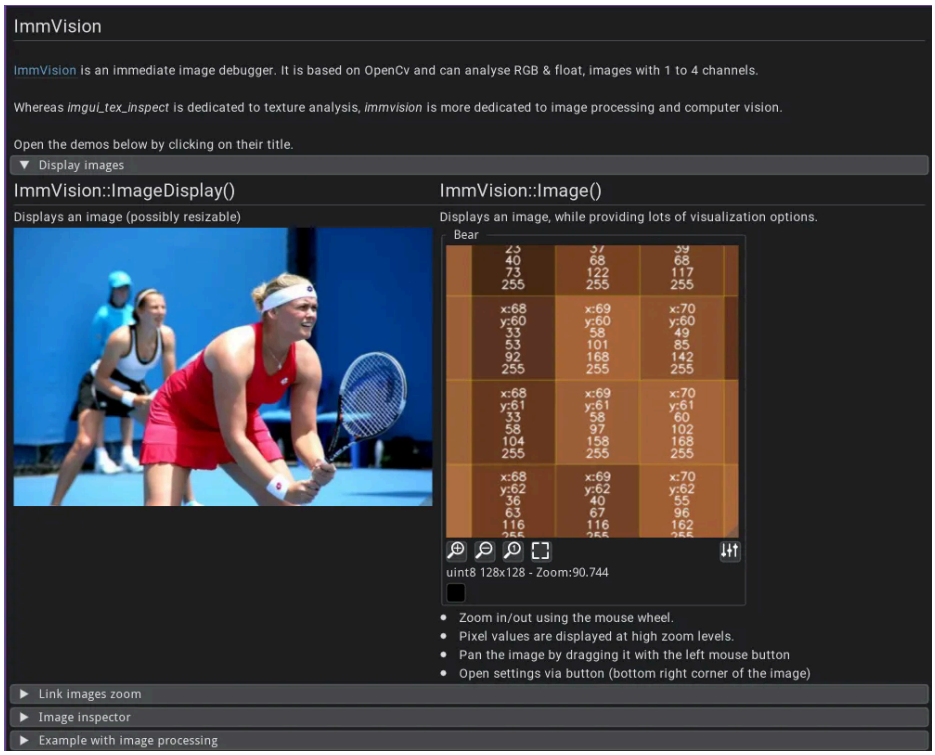


Figure 21: ImmVision: interactive image display with zoom, pan, and pixel inspection.

https://github.com/pthom/immvision

Quick example:

Python

```python
from imgui_bundle import immvision, immapp
import numpy as np

immvision.use_rgb_color_order()

image = np.zeros((100, 100, 3), dtype=np.uint8)
params = immvision.ImageParams()
```

```python
def gui():
    # Simple display
    immvision.image_display("Simple", image)

    # Full interactivity (zoom, pan, pixel inspection)
    immvision.image("Interactive", image, params)

immapp.run(gui)
```

C++

```cpp
#include "immapp/immapp.h"
#include "immvision/immvision.h"
#include <opencv2/core.hpp>

cv::Mat image;
ImmVision::ImageParams params;

void gui() {
    // Simple display
    ImmVision::ImageDisplay("Simple", image);

    // Full interactivity
    ImmVision::Image("Interactive", image, &params);
}

int main() {
    ImmVision::UseBgrColorOrder();
    image = cv::Mat::zeros(100, 100, CV_8UC3);
    ImmApp::Run(gui);
    return 0;
}
```

> **Note**
>
> This example uses `cv::Mat` from OpenCV, but **OpenCV is optional**. ImmVision works standalone with its own `ImmVision::ImageBuffer` type. If you don't need OpenCV, replace `cv::Mat` with `ImageBuffer` and use `UseRgbColorOrder()` instead.

Features:

- Zoom in/out using the mouse wheel
- Pixel values displayed at high zoom levels
- Pan by dragging with left mouse button
- Settings panel for colormap, channels, etc.

> **Tip**

> Call `immvision.use_rgb_color_order()` once at startup for RGB images. Call `use_bgr_color_order()` for OpenCV BGR images.

Full Demo:

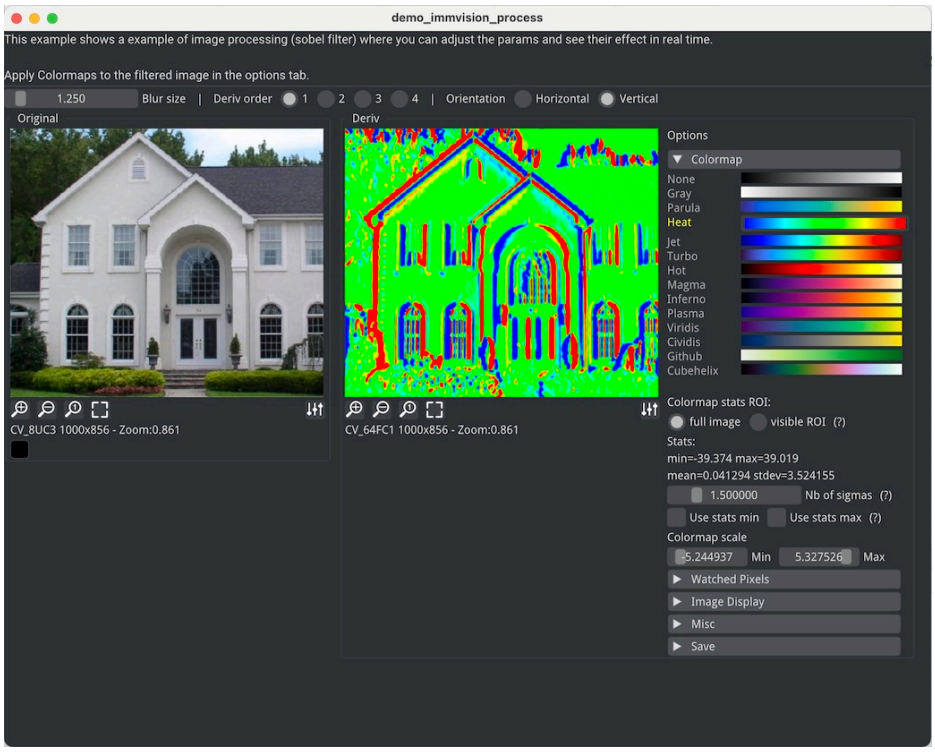


Figure 22: Click the image to run a launcher that includes several examples.

https://imgui-bundle.pages.dev/explorer/demo_immvision_launcher.html

Demo	Python	C++
Display Image	demo_immvision_display.py	demo_immvision_display.cpp
Link Images Zoom/Pan	demo_immvision_link.py	demo_immvision_link.cpp
Image Inspector	demo_immvision_inspector.py	demo_immvision_inspector.cpp
Image Processing	demo_immvision_process.py	demo_immvision_process.cpp

Documented APIs:

- **Python:** immvision.pyi
- **C++:** image.h | inspector.h

5.b.ii. *ImmDebug — Standalone Image Debugger:*

Video tutorial on Youtube

ImmDebug a tool from ImmVision lets you visually inspect images from any running program — during execution or even after it finishes (post-mortem). Add one-line

calls to send images to a standalone viewer with zoom, pan, pixel inspection, and colormaps.

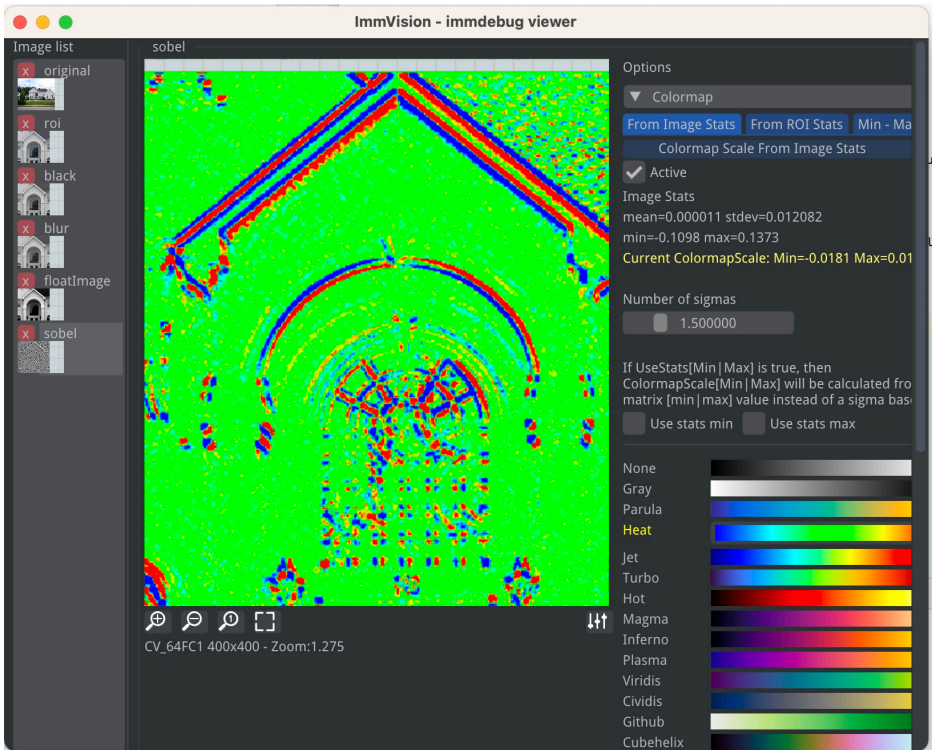


Figure 23: The immdebug viewer displaying images sent from a running program.

Python:

Install from PyPI:

```
pip install immdebug
```

Start the viewer in a terminal, then send images from your code:

```
# In a terminal: immdebug-viewer

import numpy as np
import cv2
from immdebug import immdebug

image = cv2.imread("photo.jpg")
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 100, 200)

immdebug(image, "image")  # inspect the original image
immdebug(gray, "gray")    # inspect different steps of your processing
pipeline
immdebug(edges, "edges")
```

Works with OpenCV, PIL, matplotlib, or any numpy array. See the immdebug PyPI package for full API documentation.

C++:

Drop 4 files from src/immdebug into your project (only OpenCV required):

```
#include "immdebug/immdebug.h"

cv::Mat image = cv::imread("photo.jpg");
ImmVision::ImmDebug(image, "original");
```

Build and run the C++ viewer separately — see the immvision README for instructions.

Features:

- **Non-blocking** — just writes a file and returns immediately
- **Cross-language** — C++ and Python clients use the same protocol, both work with either viewer
- **Post-mortem** — images persist in the temp directory for 1 hour; start the viewer after your script finishes
- **Single instance** — re-launching the viewer brings the existing instance to the top

5.b.iii. *imgui_tex_inspect - Texture Inspector:*

Introduction:

imgui_tex_inspect is a texture inspector tool for debugging GPU textures. It displays textures with zoom, pan, and detailed pixel information.

Figure 24: imgui_tex_inspect: GPU texture inspection with zoom and pixel details.

https://github.com/andyborrell/imgui_tex_inspect

> **Tip**
>
> Enable imgui_tex_inspect by passing `with_tex_inspect=True` (Python) or `addons.withTexInspect = true` (C++).

Full Demo:

- Demo Window | Python | C++
- Simple Example | Python | C++

Documented APIs:

- **Python:** imgui_tex_inspect.pyi
- **C++:** imgui_tex_inspect.h

5.c. *Text Editing & Markdown*

Dear ImGui Bundle includes libraries for syntax-highlighted text editing and markdown rendering.

5.c.i. *imgui_md - Markdown Rendering:*

Introduction:

imgui_md renders markdown content directly in your ImGui interface. Supports headers, bold, italic, links, code blocks, lists, and more.

Quick example:

Python

```python
from imgui_bundle import imgui_md, immapp

def gui():
    imgui_md.render("""
# Hello Markdown

This is **bold** and this is *italic*.

- List item 1
- List item 2
    """)

immapp.run(gui, with_markdown=True)
```

You may also use `imgui_md.render_unindented(s)` – it removes the leading indentation of the markdown string before rendering, which is useful when the string is defined inside a function with indentation.

C++

```cpp
#include "immapp/immapp.h"
#include "imgui_md_wrapper/imgui_md_wrapper.h"

void gui() {
    ImGuiMd::Render(R"(
# Hello Markdown

This is **bold** and this is *italic*.

- List item 1
- List item 2
    )");
}

int main() {
    ImmApp::RunWithMarkdown(gui);
```

```
    return 0;
}
```

> **Tip**
>
> Enable markdown by passing `with_markdown=True` to `immapp.run()` (Python) or use `ImmApp::RunWithMarkdown()` (C++).

Images:

imgui_md supports images from local assets and from URLs.

Standard markdown images:

```
![local image](images/world.png)
![online image](https://example.com/photo.jpg)
```

HTML img tags with explicit size:

```
<img src="https://example.com/photo.jpg" width="200">
<img src="images/logo.png" width="100" height="50">
```

> **Note**
>
> **Python:** URL images are downloaded asynchronously (a loading spinner is shown while downloading). This works automatically when using `immapp.run(with_markdown=True)`. The download callback can be customized via `MarkdownCallbacks.on_download_data`.
>
> **C++ (Emscripten):** URL images are downloaded automatically using `emscripten_fetch` (non-blocking, async).
>
> **C++ (desktop):** URL images are not downloaded by default. To enable them, set `MarkdownCallbacks::OnDownloadData` to a function that downloads data from a URL (e.g. using libcurl). The callback should return a `MarkdownDownloadResult` with `Ready`/`Downloading`/`Failed` status.

LaTeX math:

imgui_md can render inline and display LaTeX math via the bundled imgui_microtex library — a thin wrapper around MicroTeX.

Quick example:

Python

```
from imgui_bundle import imgui_md, immapp

def gui():
    imgui_md.render_unindented(r"""
```

```
# Math in markdown

Inline math sits on the text baseline: $E = mc^2$, $\sqrt{a^2 +
b^2}$,
$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Display math is centered on its own line:

$$
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
$$

Sums, integrals, matrices all work:

$$
\int_{-\infty}^{\infty} e^{-x^2}\, dx = \sqrt{\pi}
\qquad
A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}
\qquad
e^{i\pi} + 1 = 0
$$
    """)

immapp.run(gui, with_latex=True)   # implies with_markdown=True
```

Note the **raw string** (`r"""..."""`): without it, Python would interpret sequences like `\theta` as escape characters and silently corrupt the LaTeX.

C++

```
#include "immapp/immapp.h"
#include "imgui_md_wrapper/imgui_md_wrapper.h"

void gui() {
    ImGuiMd::Render(R"(
# Math in markdown

Inline math: $E = mc^2$, $\sqrt{a^2 + b^2}$.

$$
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
$$
    )");
}

int main() {
    ImmApp::AddOnsParams addons;
    addons.withLatex = true;        // implies withMarkdown=true
    HelloImGui::SimpleRunnerParams simple;
    simple.guiFunction = gui;
    ImmApp::Run(simple, addons);
```

```
    return 0;
}
```

Tip

Enable LaTeX math by passing `with_latex=True` to `immapp.run()` (Python) or setting `addOnsParams.withLatex = true` (C++). Both imply markdown support — you do not need to also set `with_markdown=True`.

- When LaTeX is **disabled** (the default), `$` is displayed as a normaly
- When LaTeX is **enabled**, `$` is treated as math delimiters. To include a literal `$`, escape it like this: `\$`.

Pyodide note

Pyodide wheels do not include the math fonts required for rendering LaTeX (to save space, since they occupy about 500KB). They will be downloaded when needed: the download is triggered by a call to `immapp.run(with_latex=True)`.

Full Demo:

Try online | Python | C++

Documented APIs:

- **Python:** imgui_md.pyi
- **C++:** imgui_md_wrapper.h

5.c.ii. *ImGuiColorTextEdit - Syntax Highlighting Editor & Diff Viewer:*

Introduction:

ImGuiColorTextEdit is a syntax highlighting text editor and diff viewer for ImGui (originally by BalazsJako, rewritten from scratch by Johan A. Goossens).

Dear ImGui Bundle uses a fork with a few additions for Python bindings.

Features:

- Syntax highlighting for C, C++, Python, GLSL, HLSL, Lua, SQL, AngelScript, C#, JSON, Markdown
- Multiple color palettes (dark, light)
- Find/replace UI with keyboard shortcuts
- Text markers (colored line highlights with tooltips)
- Bracket matching with visual indicators
- Line decorators (custom gutter content per line, e.g. breakpoints)
- Context menu callbacks (separate for line numbers and text area)
- Change and transaction callbacks
- Filter selections/lines (transform text via callbacks)
- Autocomplete framework

- Undo/redo, copy/paste, multi-cursor support
- **TextDiff widget**: combined and side-by-side diff view for comparing two texts

> **Tip**
>
> The text editor requires a fixed-width font. If you are using ImmApp with Markdown enabled, you may use its code font:
>
> ```
> code_font = imgui_md.get_code_font()
> imgui.push_font(code_font.font, code_font.size)
> editor.render("Code")
> imgui.pop_font()
> ```

Full Demo:

Try online | Python | C++

Documented APIs:

- **Python:** imgui_color_text_edit.pyi
- **C++:** TextEditor.h | TextDiff.h

5.d. Tools

Dear ImGui Bundle includes specialized tools for 3D editing and visual programming.

5.d.i. *ImGuizmo - 3D Gizmos:*

Introduction:

ImGuizmo provides immediate mode 3D gizmos for scene editing: translate, rotate, and scale manipulators similar to those found in 3D modeling software.

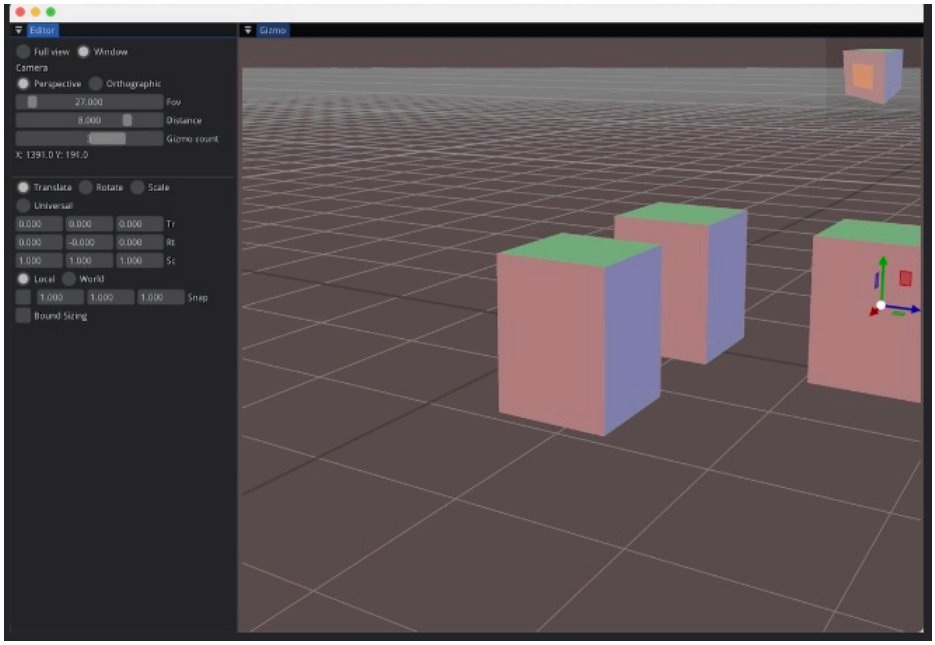


Figure 25: ImGuizmo: 3D gizmos for translation, rotation, and scale.

https://github.com/CedricGuillemet/ImGuizmo

Features:

- Translation, rotation, scale gizmos
- Local and world coordinate modes
- View cube for camera orientation
- Snap to grid support

> **Note**
>
> **Python:** ImGuizmo requires PyGLM for matrix operations: `pip install PyGLM`

Full Demo:

Try online | Python | C++

Documented APIs:

- **Python:** imguizmo.pyi
- **C++:** ImGuizmo.h

5.d.ii. *imgui-node-editor - Visual Node Graphs:*

Introduction:

imgui-node-editor is a node editor built using Dear ImGui. Create visual programming interfaces, data flow graphs, etc.

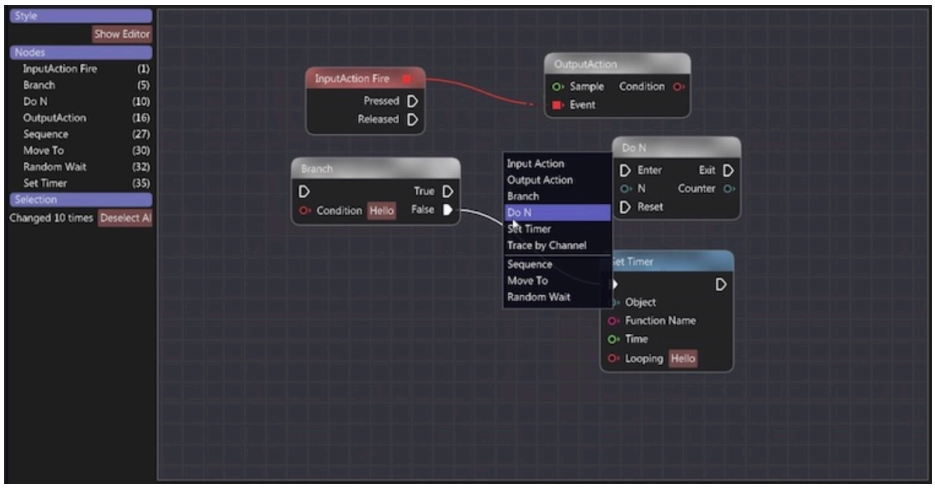


Figure 26: imgui-node-editor: visual node graphs for data flow and shader editing.

https://github.com/thedmd/imgui-node-editor

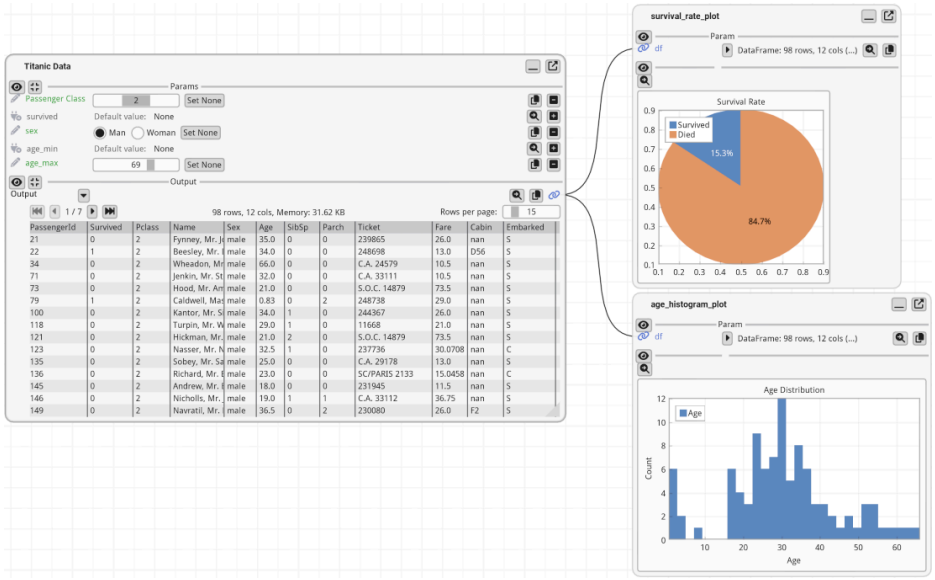


Figure 27: Note: Fiatlight, a library built on top of Dear ImGui Bundle uses imgui-node-editor intensively. It provides an automatic UI generation for functions and structured data.

https://pthom.github.io/fiatlight

Tip

> Enable the node editor by passing `with_node_editor=True` (Python) or `addons.withNodeEditor = true` (C++).

Full Demo:

Try online - A launcher with several demos:

Demo	Python	C++
Basic Demo	demo_node_editor_basic.py	demo_node_editor_basic.cpp
Romeo and Juliet	demo_romeo_and_juliet.py	demo_romeo_and_juliet.cpp

Documented APIs:

- **Python:** imgui_node_editor.pyi
- **C++:** imgui_node_editor.h

5.d.iii. *NanoVG - 2D Vector Graphics:*

Introduction:

NanoVG is an antialiased 2D vector drawing library on top of OpenGL. Use it for custom drawing, charts, diagrams, or any vector graphics needs.

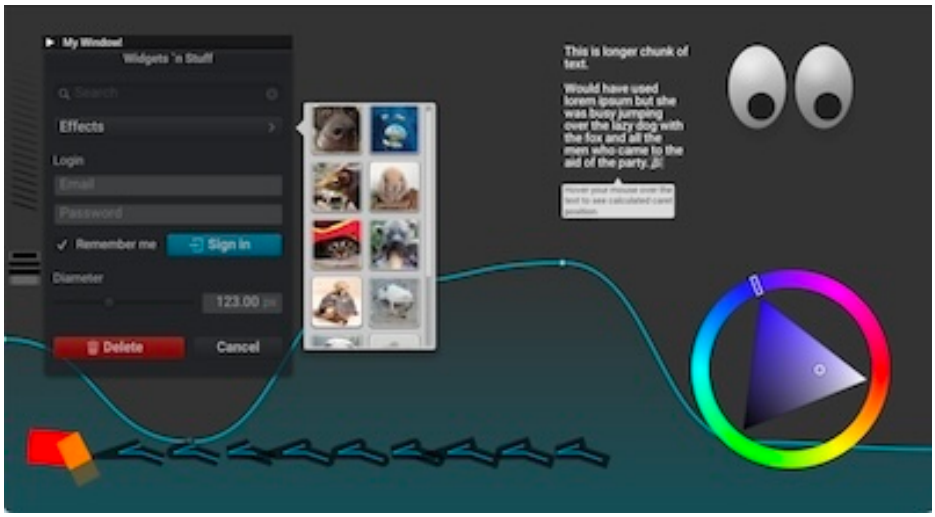


Figure 28: NanoVG: antialiased 2D vector graphics with paths, shapes, and text.

https://github.com/memononen/nanovg

Features:

- Antialiased rendering
- Paths, shapes, gradients
- Text rendering with font support
- Scissoring and clipping

Full Demo:

Try online - A launcher with two demos:

Demo	Python	C++
Full Demo	demo_nanovg_full.py	demo_nanovg_full.cpp
Simple Demo	demo_nanovg_heart.py	demo_nanovg_heart.cpp

Documented APIs:

- **Python:** nanovg.pyi
- **C++:** nanovg.h

5.e. *Widgets*

Dear ImGui Bundle includes a variety of additional widgets beyond the standard ImGui set.

5.e.i. *Full Demo:*

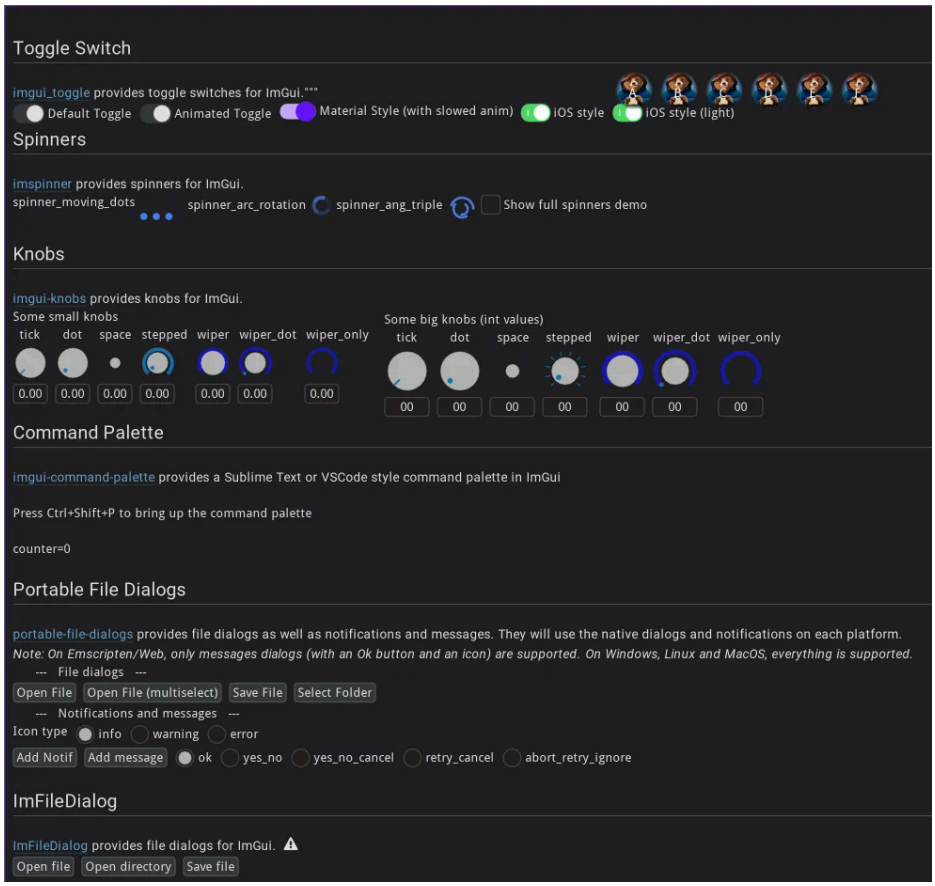


Figure 29: Widgets demo showcasing knobs, toggles, spinners, coolbar, and more.

https://imgui-bundle.pages.dev/explorer/demo_widgets.html

Try online | Python | C++

5.e.ii. *imgui-knobs - Rotary Knobs:*

Introduction:

imgui-knobs adds rotary knob widgets similar to those found in audio software.

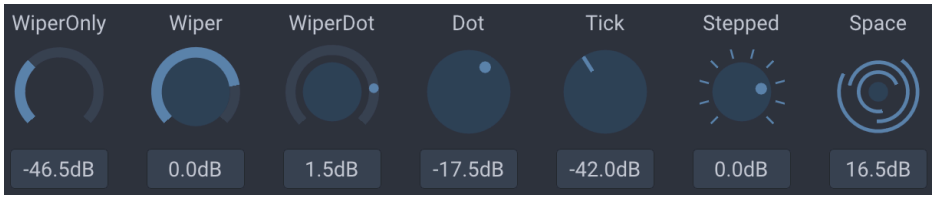


Figure 30: imgui-knobs: rotary knobs with multiple visual styles.

https://github.com/altschuler/imgui-knobs

Knob variants: `tick`, `dot`, `wiper`, `wiper_only`, `stepped`

Quick Example:

Python

```python
from imgui_bundle import imgui_knobs, immapp

value = 50.0

def gui():
    global value
    changed, value = imgui_knobs.knob("Volume", value, 0.0, 100.0)

immapp.run(gui)
```

C++

```cpp
#include "immapp/immapp.h"
#include "imgui-knobs/imgui-knobs.h"

float value = 50.0f;

void gui() {
    ImGuiKnobs::Knob("Volume", &value, 0.0f, 100.0f);
}

int main() {
    ImmApp::Run(gui, "Knobs Demo", {400, 300});
    return 0;
}
```

Documented APIs:

- **Python:** imgui_knobs.pyi
- **C++:** imgui-knobs.h

5.e.iii. *imgui_toggle - Toggle Switches:*

Introduction:

imgui_toggle provides iOS-style toggle switches.

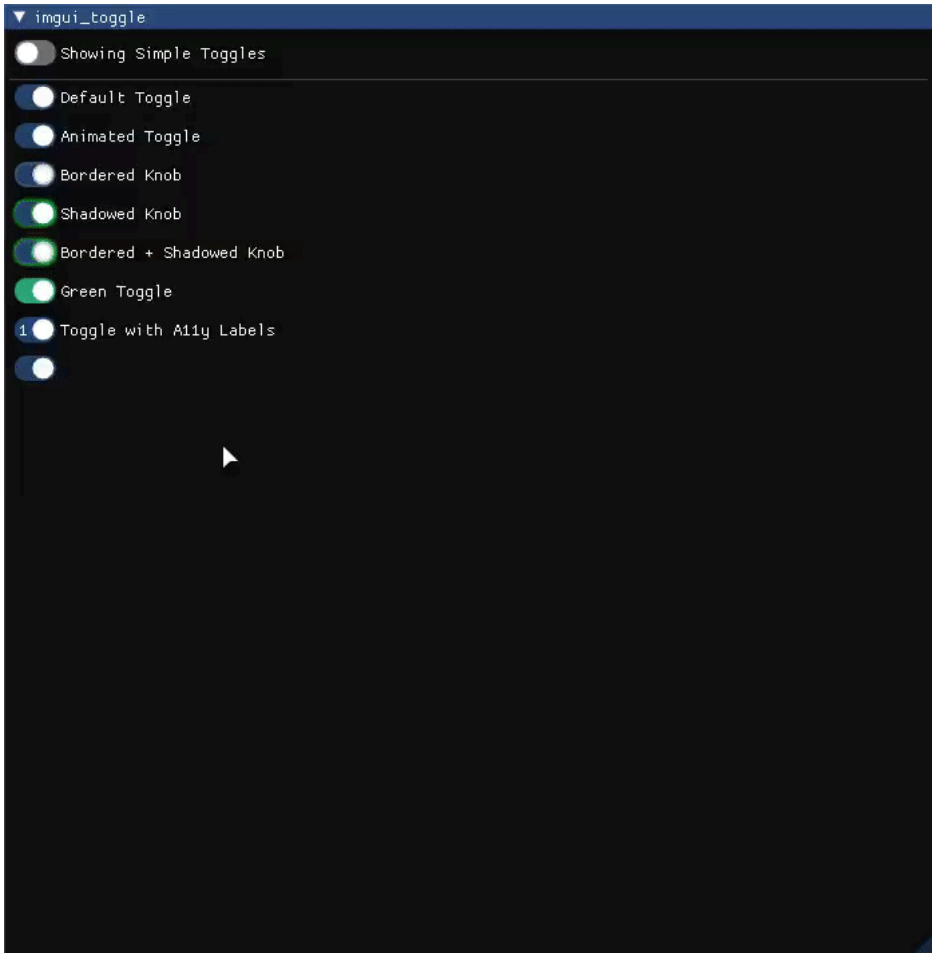


Figure 31: imgui_toggle: iOS-style toggle switches with animations.

https://github.com/cmdwtf/imgui_toggle

Quick Example:

Python

```python
from imgui_bundle import imgui_toggle, immapp

enabled = False

def gui():
    global enabled
    changed, enabled = imgui_toggle.toggle("Enable feature", enabled)

immapp.run(gui)
```

C++

```cpp
#include "immapp/immapp.h"
#include "imgui_toggle/imgui_toggle.h"
```

```cpp
bool enabled = false;

void gui() {
    ImGui::Toggle("Enable feature", &enabled);
}

int main() {
    ImmApp::Run(gui, "Toggle Demo", {400, 300});
    return 0;
}
```

Documented APIs:

- **Python:** imgui_toggle.pyi
- **C++:** imgui_toggle.h

5.e.iv. *imspinner - Loading Spinners:*

Introduction:

imspinner provides a large collection of animated loading spinners.

> **Tip**
>
> Call `imspinner.demo_spinners()` (Python) or `ImSpinner::demoSpinners()` (C++) to see all available spinner types.

Quick Example:

Python

```python
from imgui_bundle import imspinner, imgui, immapp

def gui():
    imspinner.spinner_ang_triple(
        "spinner",
        radius1=16, radius2=13, radius3=10,
        thickness=3,
        c1=imgui.ImColor(255, 255, 255),
        c2=imgui.ImColor(255, 100, 100),
        c3=imgui.ImColor(100, 100, 255)
    )

immapp.run(gui)
```

C++

```cpp
#include "immapp/immapp.h"
#include "imspinner/imspinner.h"
```

```cpp
void gui() {
    ImSpinner::SpinnerAngTriple(
        "spinner",
        16, 13, 10,  // radii
        3,           // thickness
        ImColor(255, 255, 255),
        ImColor(255, 100, 100),
        ImColor(100, 100, 255)
    );
}

int main() {
    ImmApp::Run(gui, "Spinner Demo", {400, 300});
    return 0;
}
```

Documented APIs:

- **Python:** imspinner.pyi
- **C++:** imspinner.h

5.e.v. *ImCoolBar - macOS-style Dock Bar:*

Introduction:

ImCoolBar creates a macOS-style dock bar with magnification effect on hover.

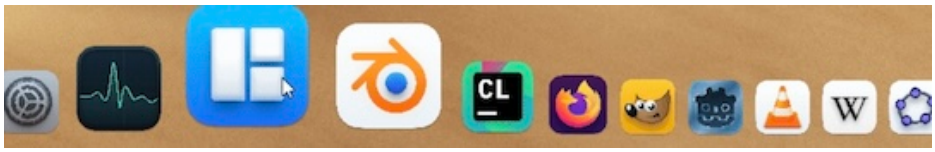


Figure 32: ImCoolBar: macOS-style dock bar with magnification effect.

https://github.com/aiekick/ImCoolBar

Documented APIs:

- **Python:** im_cool_bar.pyi
- **C++:** ImCoolbar.h

5.e.vi. *imgui-command-palette - VSCode-style Command Palette:*

Introduction:

imgui-command-palette adds a Sublime Text / VSCode style command palette.

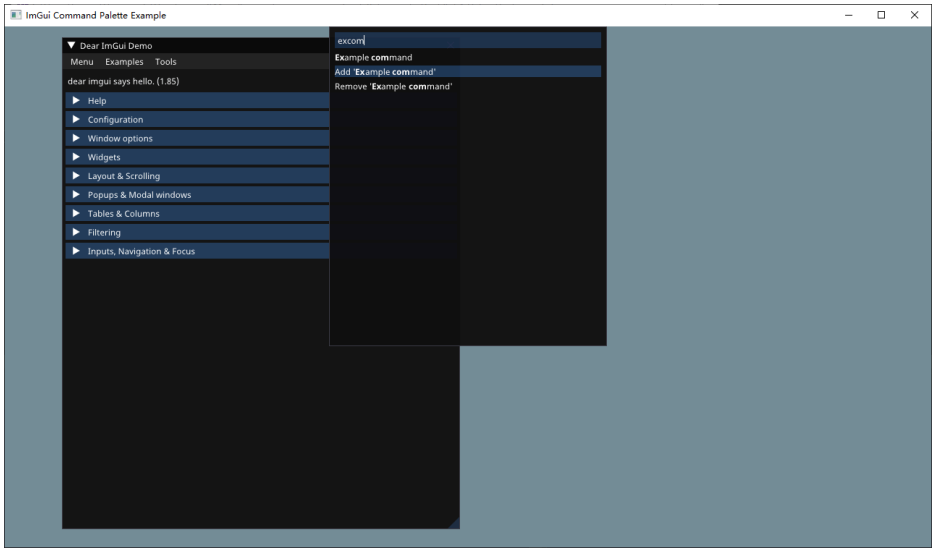


Figure 33: imgui-command-palette: VSCode-style command palette.

https://github.com/hnOsmium0001/imgui-command-palette

Full Demo:

Try online | Python | C++

Documented APIs:

- **Python:** imgui_command_palette.pyi
- **C++:** imcmd_command_palette.h

5.e.vii. *File Dialogs:*

Dear ImGui Bundle provides two file dialog libraries.

portable-file-dialogs:

portable-file-dialogs uses native OS dialogs for file selection, notifications, and messages.

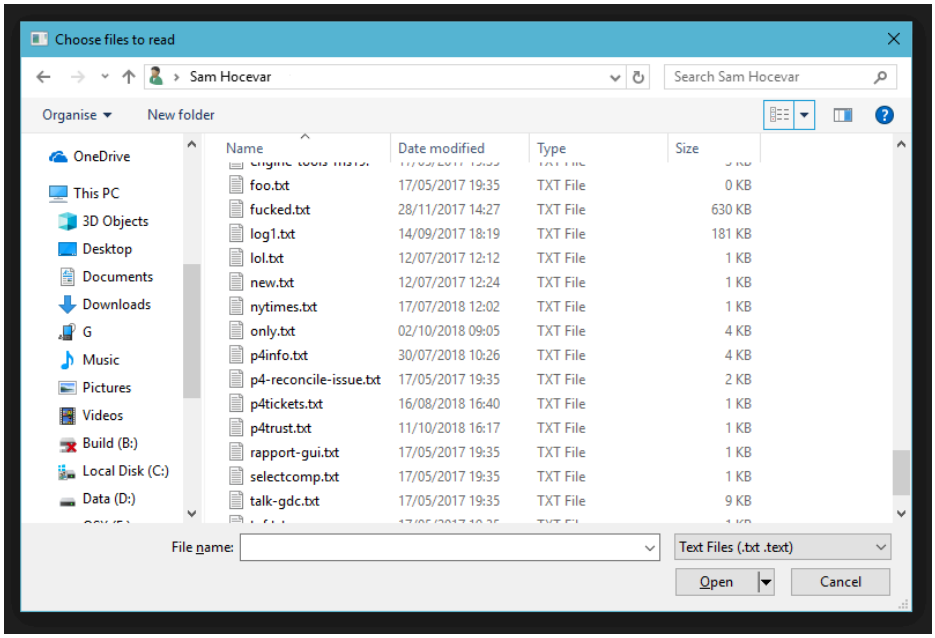


Figure 34: portable-file-dialogs: native OS dialogs on all platforms.

https://github.com/samhocevar/portable-file-dialogs

> **Note**
>
> On **Emscripten/Web**, only message dialogs (with an OK button and icon) are supported since native file dialogs are unavailable. On **Windows, Linux, and macOS**, all features work fully.

Quick Example:

Python

```python
from imgui_bundle import portable_file_dialogs as pfd

# Open file dialog (native OS)
selection = pfd.open_file("Select a file", ".", ["Image Files",
"*.png *.jpg"])

# Message box
pfd.message("Title", "Message content", pfd.choice.ok, pfd.icon.info)
```

C++

```cpp
#include "portable-file-dialogs/portable-file-dialogs.h"

// Open file dialog (native OS)
auto selection = pfd::open_file("Select a file", ".",
    {"Image Files", "*.png *.jpg"}).result();
```

```
// Message box
pfd::message("Title", "Message content", pfd::choice::ok,
pfd::icon::info);
```

Documented APIs:

- **Python:** portable_file_dialogs.pyi
- **C++:** portable-file-dialogs.h

ImFileDialog:

ImFileDialog is a file dialog library with a modern look.

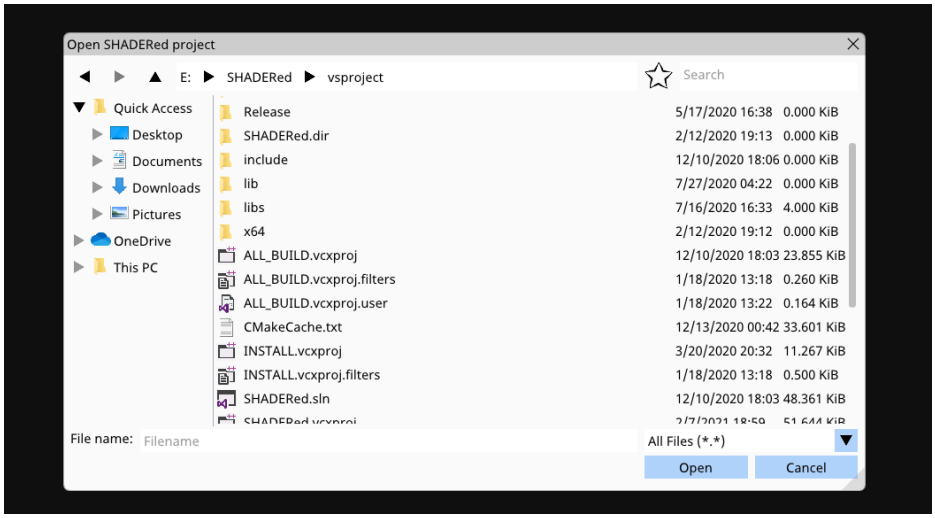


Figure 35: ImFileDialog: modern file dialog with preview support.

https://github.com/dfranx/ImFileDialog

> **Warning**
>
> Consider using **portable-file-dialogs** instead, which provides native OS dialogs on each platform plus notifications and messages.
>
> **Known limitations of ImFileDialog:**
>
> - Not adapted for High DPI resolution under Windows
> - No support for multi-selection
> - Will not work under Python with a pure Python backend (requires `immapp.run()`)

Quick Example:

Python

```
from imgui_bundle import im_file_dialog as ifd, imgui, immapp
```

```python
def gui():
    if imgui.button("Open File"):
        ifd.FileDialog.instance().open(
            "OpenFile", "Open a file", "Image files (*.png;*.jpg)
{.png,.jpg},.*"
        )

    if ifd.FileDialog.instance().is_done("OpenFile"):
        if ifd.FileDialog.instance().has_result():
            result = ifd.FileDialog.instance().get_result()
            print(f"Selected: {result}")
        ifd.FileDialog.instance().close()

immapp.run(gui)
```

C++

```cpp
#include "immapp/immapp.h"
#include "ImFileDialog/ImFileDialog.h"

void gui() {
    if (ImGui::Button("Open File")) {
        ifd::FileDialog::Instance().Open(
            "OpenFile", "Open a file", "Image files (*.png;*.jpg)
{.png,.jpg},.*"
        );
    }

    if (ifd::FileDialog::Instance().IsDone("OpenFile")) {
        if (ifd::FileDialog::Instance().HasResult()) {
            auto result = ifd::FileDialog::Instance().GetResult();
            printf("Selected: %s\n", result.string().c_str());
        }
        ifd::FileDialog::Instance().Close();
    }
}

int main() {
    ImmApp::Run(gui, "File Dialog", {800, 600});
    return 0;
}
```

Documented APIs:

- **Python:** im_file_dialog.pyi
- **C++:** ImFileDialog.h

6. Support

6.a. Support the project

Dear ImGui Bundle is a free and open-source project, and its development and maintenance require considerable efforts.

If you find it valuable for your work – especially in a commercial enterprise or a research setting – please consider supporting its development by making a donation. Your contributions are greatly appreciated!

For commercial users seeking tailored support or specific enhancements, please contact the author by email.

6.a.i. *Contribute:*

Quality contributions are always welcome! If you're interested in contributing to the project, whether through code, ideas, or feedback, please refer to the development documentation.

6.a.ii. *License:*

Dear ImGui Bundle is licensed under the MIT License

6.b. Closing words

6.b.i. *License:*

The MIT License (MIT)

Copyright (c) 2021-2026 Pascal Thomet

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

6.b.ii. *About the author:*

Dear ImGui Bundle is developed by Pascal Thomet. I am reachable on my Github page. I sometimes blog. There is a playlist related to ImGui Bundle on YouTube.

I have a past in computer vision, and a lot of experience in the trenches between development and research teams; and I found ImGui to be a nice way to reduce the delay between a research prototype and its use in production code.

I also have an inclination for self documenting code, and the doc you are reading was a way to explore new ways to document projects.

6.b.iii. *How is Dear ImGui Bundle developed:*

The development of the initial version of Dear ImGui Bundle took about one year at full time.

The bindings are auto-generated thanks to an advanced parser, so that they are easy to keep up to date.

Please be tolerant if you find issues! Dear ImGui Bundle is developed for free, under a very permissive license, by one main author (and most of its API comes from external libraries).

If you need consulting about this library or about the bindings generator in the context of a commercial project, please contact me by email.

Contributions are welcome!

6.b.iv. *Thanks:*

Dear ImGui Bundle would not be possible without the work of the authors of "Dear ImGui", and especially Omar Cornut.

It also includes a lot of other projects, and I'd like to thank their authors for their awesome work!

A particular mention for Evan Pezent (author of ImPlot), Cédric Guillemet (author of ImGuizmo), Balázs Jákó (author of ImGuiColorTextEdit), and Michał Cichoń (author of imgui-node-editor), and Dmitry Mekhontsev (author of imgui-md), Andy Borrel (author of imgui-tex-inspect, another image debugging tool, which I discovered long after having developed immvision).

Immvision was inspired by The Image Debugger, by Bill Baxter.

7. Developer docs

7.a. Intro - Developer docs

This section is for developers willing to build and modify the imgui_bundle library. It covers topics such as building the library, updating dependencies, and adding new features or bindings.

> **Tip**
>
> **New here?** Start with Getting Started (Developer) — it takes you from clone to working build in 5 minutes.

7.a.i. *Architecture Overview:*

ImGui Bundle implements a **four-layer architecture**:

1. **Rendering Backends** – GLFW/SDL + OpenGL/Metal/DirectX/Vulkan
2. **Dear ImGui Core** – immediate-mode widget system
3. **Hello ImGui Framework** – window lifecycle, docking, DPI handling, asset management
4. **ImmApp Layer** – simplified runner with automatic add-on initialization

Users can work at their preferred abstraction level, from raw ImGui calls to the high-level `immapp.run()` API.

7.a.ii. *Python Bindings:*

The bindings are auto-generated using litgen, a code generator that transforms C++ headers into:

- **nanobind C++ code** – the actual binding implementation
- **`.pyi` type stubs** – for IDE autocompletion and type checking

All 23+ C++ libraries compile into a single `_imgui_bundle` native extension, with submodules conditionally available based on build configuration.

7.a.iii. *Key Concepts:*

- **Add-On System**: ImmApp manages automatic context creation for optional libraries via `AddOnsParams` flags (`with_implot`, `with_markdown`, `with_latex`, `with_node_editor`, etc.)
- **Cross-Platform**: Same codebase deploys to desktop (Windows/macOS/Linux), mobile (iOS/Android), and web (Emscripten/Pyodide)
- **DPI-Aware Sizing**: Helper functions like `EmToVec2()` enable responsive layouts across high-DPI displays

7.a.iv. *Glossary:*

Terms used throughout the developer docs:

Term	Meaning
Dear ImGui	The core C++ immediate-mode GUI library by Omar Cornut
Hello ImGui	Framework layer on top of Dear ImGui: window lifecycle, docking, DPI, asset management, multi-platform deployment
ImmApp	High-level runner that wraps Hello ImGui and auto-initializes add-ons (ImPlot, Markdown, etc.) via simple flags
litgen	Literate Generator — the tool that reads C++ headers and generates Python bindings (nanobind C++ code + `.pyi` stubs)
nanobind	C++ library used to create Python extension modules. Successor to pybind11; used by litgen's generated code
srcML	XML representation of C++ source code; used internally by litgen to parse headers
`.pyi` stubs	Python type stub files providing IDE autocompletion and type checking for the native extension
pybind	Legacy naming seen in generated filenames (`pybind_*.cpp`). These files actually use nanobind, not pybind11 — the name is historical
ibex	Abbreviation fo Dear ImGui Bundle Explorer — the full demo app showcasing all libraries, used in just recipes (`just ibex_...`)
imex	Abbreviation for Dear ImGui Explorer - source code in external/imgui_explorer. Used just recipes `just imex_...`
`_imgui_bundle`	The single compiled native extension (`.so`/`.pyd`) containing all 23+ C++ libraries
justfile	Task runner config at repo root; provides shortcuts like `just libs_bindings`, `just test_pytest`, etc.

7.a.v. *More Resources:*

- **DeepWiki - ImGui Bundle** – AI-powered documentation explorer for in-depth architecture questions
- **Litgen Documentation** – the bindings generator used by this project

7.a.vi. *In This Section:*

- Getting Started (Developer) – clone, build, run, test
- Repository Structure – folder organization
- Build Guide – CMake options, presets, justfile commands
- Testing – running tests, CI workflows
- Bindings Introduction – how the generator works
- Update Bindings – updating existing library bindings
- Add New Library – adding a new library to the bundle
- Debug Native C++ – debugging C++ code from Python
- PyPI Deployment – releasing to PyPI
- Pyodide Build Guide – building for Python-in-the-browser

- Cloudflare Pages Deploy – publishing the playground + explorer to imgui-bundle.pages.dev
- OpenCV Build Guide – building with OpenCV / ImmVision

7.b. Getting Started (Developer)

This guide gets you from a fresh clone to a working build with demos and tests. It targets contributors and developers who want to build imgui_bundle from source.

> **Tip**
>
> If you just want to **use** imgui_bundle as a Python package, run `pip install imgui-bundle` and see the Python install guide. This page is for people who want to **build and modify** the library itself.

7.b.i. *Prerequisites:*

Tool	Version	Notes
git	any recent	With submodule support
CMake	>= 3.18	
C++ compiler	C++17	GCC 9+, Clang 10+, MSVC 2019+, or Apple Clang 12+
Python	>= 3.8	Only needed if building Python bindings
just	any	Optional but recommended — task runner for common commands (`brew install just` / `cargo install just`)

Linux only: install `libglfw3-dev` (or pass `-DHELLOIMGUI_DOWNLOAD_GLFW_IF_NEEDED=ON`).

7.b.ii. *Clone & init submodules:*

```
git clone --recursive https://github.com/pthom/imgui_bundle.git
cd imgui_bundle
```

If you already cloned without `--recursive`:

```
git submodule update --init --recursive
```

Submodules are auto-cloned during CMake configure by default (via `IMGUI_BUNDLE_AUTO_CLONE_SUBMODULES=ON`), so this step is often optional.

7.b.iii. *Quick build: C++ only (desktop):*

```
mkdir -p builds/my_build && cd builds/my_build
cmake ../.. --preset default
cmake --build . -j
```

This builds with GLFW + OpenGL3 (the default). Run the demos:

```
./demo_imgui_bundle          # Main demo showcasing all libraries
```

7.b.iv. *Quick build: Python bindings:*

```
# Create a venv
python -m venv venv && source venv/bin/activate
pip install -r requirements-dev.txt   # numpy, pytest, mypy, etc.

# Build
mkdir -p builds/my_python && cd builds/my_python
cmake ../.. --preset python_bindings -DPython_EXECUTABLE=$(which python)
cmake --build . -j
```

Then set your `PYTHONPATH` to include the build output, and run:

```
cd ../..
python bindings/imgui_bundle/demos_python/demo_imgui_bundle.py
```

> **Note**
>
> Python bindings force GLFW + OpenGL3 as the backend. For other backends, build C++ only.

Alternative: editable pip install:

For a development workflow where changes are picked up without rebuilding manually:

```
pip install -v -e .
```

This uses scikit-build-core under the hood and takes longer the first time.

7.b.v. *Build with ImmVision (OpenCV):*

ImmVision works without OpenCV (using `ImageBuffer` / numpy arrays). To also enable `cv::Mat` interop:

```
cmake ../.. --preset python_bindings \
    -DPython_EXECUTABLE=$(which python) \
    -DIMMVISION_FETCH_OPENCV=ON
```

This fetches and builds a minimal OpenCV in the build directory. See OpenCV builds for immvision for platform-specific details.

7.b.vi. *Run the tests:*

```
# From the repo root
just test_pytest    # Run pytest
just test_mypy      # Run mypy type checking on bindings
```

Or without `just`:

```
pytest
cd bindings && ./mypy_bindings.sh
```

See Testing for details on GUI tests and CI.

7.b.vii. *Build the docs:*

```
just doc_serve                # Live-reload dev server (recommended for
editing)
# or, for the full Cloudflare-deploy build (HTML anchored under /doc/,
plus PDF):
just doc_build_cf
# Then preview under the right /doc/ prefix:
just cf_stage && just cf_serve_local   # http://localhost:8765/doc/
```

7.b.viii. *Useful justfile commands:*

Run `just` (no arguments) to see all available commands. Key groups:

Command	What it does
`just libs_info`	Show all external libraries with their remotes and branches
`just libs_bindings <name>`	Regenerate Python bindings for one library
`just libs_bindings_all`	Regenerate all Python bindings
`just libs_check_upstream`	Check which fork libraries have new upstream changes
`just test_pytest`	Run pytest
`just test_mypy`	Run mypy on bindings
`just doc_serve_interactive`	Build & serve docs with live reload

See Build Guide for the full command reference.

7.b.ix. *What next?:*

I want to…	See
Understand the folder layout	Repository structure
Learn about the build system & CMake options	Build guide
Understand how bindings are generated	Bindings introduction
Update an existing library's bindings	Update bindings
Add a new C++ library to the bundle	Add new library
Debug native C++ code from Python	Debug bindings
Run and write tests	Testing
Deploy to PyPI	PyPI deployment
Build with OpenCV / ImmVision	OpenCV build guide

7.c. Repository folders structure

Below is the folder structure of Dear ImGui Bundle repository.

Key areas:

- `bindings/` – Python package, stubs (`.pyi`), and demos
- `external/` – All C++ libraries (submodules) and their binding generators
- `imgui_bundle_cmake/` – CMake build infrastructure
- `docs/` – This documentation (Jupyter Book)

7.c.i. *Top-level overview:*

```
./
├── CMakeLists.txt                          Main CMake build file
├── CMakePresets.json                       Build presets
(python_bindings, etc.)
├── pyproject.toml                          Python package config
(scikit-build-core + nanobind)
├── justfile                                Task runner for common build
commands
├── litgen_imconfig.h                       ImGui config header for the
bindings generator
├── pytest.ini                              Pytest configuration
├── requirements-dev.txt                    Development dependencies
├── CHANGELOG.md
├── LICENSE                                 MIT
├── Readme.md
│
│   ── Python package & demos ──────────────────────────────────────
│
├── bindings/                               Root of the Python package
(see below)
│
│   ── C++ libraries ───────────────────────────────────────────────
│
├── external/                               All C++ libraries + binding
generators (see below)
│
│   ── CMake build infrastructure ──────────────────────────────────
│
├── imgui_bundle_cmake/                     CMake helpers & build logic
(see below)
│
│   ── Documentation ───────────────────────────────────────────────
│
├── docs/
│     └── book/                             Jupyter Book documentation
(see below)
│
│   ── Other ───────────────────────────────────────────────────────
│
```

```
├── _example_integration/                  Template project for using
imgui_bundle in C++ apps
│     ├── CMakeLists.txt                   (also available as https://
github.com/pthom/imgui_bundle_template)
│     ├── hello_world.cpp
│     └── assets/
│
├── imgui_bundle_assets/                   Shared asset files for C++
apps (fonts, etc.)
│
├── pybind_native_debug/                   Debug utility for stepping
into native C++ from Python
│     ├── pybind_native_debug.cpp
│     └── pybind_native_debug.py
│
├── tests/                                 Test suite
│     ├── lg_imgui_bundle_test.py
│     └── tests_python_gui/                GUI tests (require display)
│
├── ci_scripts/                            CI/CD scripts (Docker,
Pyodide local builds)
├── _plans/                                Specs, plans, and todos for
features
├── logo/                                  Project logo files
└── builds/                                Build output directories
(default, claude_*, etc.)
```

7.c.ii. *`bindings/` – Python package:*

This is the root of the `imgui_bundle` Python package. It contains the Python API, type stubs, demos, and the compiled native extension.

```
bindings/imgui_bundle/
│
│   ── Package core ──────────────────────────────────────────────
│
├── __init__.py                            Package initialization,
module exports, version
├── __init__.pyi                           Top-level type stubs
├── _imgui_bundle.cpython-*-.so            Compiled native extension
(all C++ libs in one .so)
│
│   ── Type stubs (.pyi) ── one per C++ library ──────────────────
│   (auto-generated by litgen; provide IDE autocompletion & type
checking)
│
├── hello_imgui.pyi                        Hello ImGui API
├── im_anim.pyi                            Animation library
├── im_cool_bar.pyi                        Cool toolbar widget
├── im_file_dialog.pyi                     File dialog
├── imgui_bundle.pyi                       Core bundle types
```

```
├── imgui_color_text_edit.pyi              Code editor with syntax
highlighting
├── imgui_command_palette.pyi              VSCode-style command palette
├── imgui_explorer.pyi                     Interactive widget explorer
├── imgui_knobs.pyi                        Knob/dial widgets
├── imgui_md.pyi                           Markdown rendering
├── imgui_microtex.pyi                     Native LaTeX math rendering
(used by imgui_md)
├── imgui_node_editor.pyi                  Node graph editor
├── imgui_tex_inspect.pyi                  Texture inspector
├── imgui_toggle.pyi                       Toggle switches
├── imguizmo.pyi                           3D gizmo/transform
├── immvision.pyi                          Image debugger (zoom,
colormaps, pixel inspection)
├── imspinner.pyi                          Spinner/loading widgets
├── nanovg.pyi                             Vector graphics
├── portable_file_dialogs.pyi              OS-native file dialogs
│
│   ── Subpackages with their own stubs ──────────────────────────
│
├── imgui/                                 Dear ImGui bindings
│     ├── __init__.pyi                     Main ImGui API stubs (very
large)
│     ├── internal.pyi                     ImGui internal API stubs
│     ├── backends.pyi                     Backend stubs
│     └── test_engine.pyi                  Test engine stubs
│
├── implot/                                ImPlot bindings
│     ├── __init__.pyi                     ImPlot API stubs
│     └── internal.pyi                     ImPlot internals
│
├── implot3d/                              ImPlot3D bindings
│     ├── __init__.pyi                     ImPlot3D API stubs
│     └── internal.pyi                     ImPlot3D internals
│
│   ── Python utilities (.py) ────────────────────────────────────
│
├── imgui_ctx.py                           Context managers for ImGui
begin/end pairs
├── imgui_node_editor_ctx.py               Context managers for node
editor
├── imgui_pydantic.py                      Pydantic integration for
ImGui types
├── imgui_fig.py                           Figure/plotting utilities
├── im_col32.py                            Color utilities
├── glfw_utils.py                          GLFW utility functions
├── hello_imgui_run_async.py               Async runner support
├── hello_imgui_nb.py                      Jupyter notebook support
├── hello_imgui_nb.pyi                     Type stubs for notebook
support
├── notebook_patch_runners.py              Patches for notebook
environments
```

```
├── pyodide_patch_runners.py                Patches for Pyodide/browser
├── _glfw_set_search_path.py                GLFW library path setup
├── _patch_runners_add_save_screenshot_param.py
│
│   ── ImmApp module ──────────────────────────────────────────────
│
├── immapp/                                 High-level app runner
utilities
│     ├── __init__.py                       Main ImmApp module
│     ├── __init__.pyi                      Type stubs
│     ├── immapp_cpp.pyi                    C++ ImmApp API stubs
│     ├── immapp_utils.py                   General utilities
│     ├── immapp_code_utils.py              Code utility functions
│     ├── immapp_notebook.py                Notebook-specific utilities
│     ├── nb.py                             Notebook integration
(nb.start / nb.stop)
│     ├── nb.pyi
│     ├── run_async_overloads.py            Async runner overloads
│     ├── runnable_code_cell.py             Runnable code cell support
│     ├── icons_fontawesome_4.py            FontAwesome 4 icon constants
│     └── icons_fontawesome_6.py            FontAwesome 6 icon constants
│
│   ── Pure Python backends ──────────────────────────────────────
│
├── python_backends/                        ImGui backends implemented
in pure Python
│     ├── opengl_base_backend.py            Base OpenGL backend class
│     ├── opengl_backend_programmable.py    Programmable pipeline
backend
│     ├── glfw_backend.py                   GLFW backend
│     ├── sdl2_backend.py                   SDL2 backend
│     ├── sdl3_backend.py                   SDL3 backend
│     ├── pygame_backend.py                 Pygame backend
│     └── pyglet_backend.py                 Pyglet backend
│
│   ── Demos ─────────────────────────────────────────────────────
│
├── demos_python/                           Python demos (15 categories)
│     ├── demos_immapp/                     ImmApp framework demos
(hello world, docking, etc.)
│     ├── demos_implot/                     2D plotting demos
│     ├── demos_implot3d/                   3D plotting demos
│     ├── demos_immvision/                  Image inspection demos
│     ├── demos_node_editor/                Node editor demos
│     ├── demos_imguizmo/                   3D gizmo demos
│     ├── demos_tex_inspect/                Texture inspector demos
│     ├── demos_nanovg/                     Vector graphics demos
│     ├── demos_imgui_explorer/             Widget explorer demos
│     ├── demos_immdebug/                   Image debug viewer demos
│     ├── haikus/                           Small "haiku" example
programs
│     ├── notebooks/                        Jupyter notebook examples
```

```
│     ├── sandbox/                          Experimental/sandbox
examples
│     ├── demo_packaging/                   Packaging/distribution
examples
│     └── demo_utils/                       Shared demo utilities
│
├── demos_cpp/                              C++ demos (9 categories)
│     ├── demos_immapp/                     ImmApp demos (hello world,
docking, etc.)
│     ├── demos_imanim/                     Animation demos
│     ├── demos_imguizmo/                   Gizmo demos
│     ├── demos_immvision/                  Image inspection demos
│     ├── demos_nanovg/                     Vector graphics demos
│     ├── demos_node_editor/                Node editor demos
│     ├── demos_tex_inspect/                Texture inspector demos
│     ├── sandbox/                          Experimental demos
│     ├── demo_utils/                       Shared C++ demo utilities
│     └── _auto_main/                       Auto-generated main() for
demos
│
├── assets/                                 Assets folder (fonts for
markdown, etc.)
└── demos_assets/                           Images and resources used by
demos
```

7.c.iii. *`external/` – C++ libraries and bindings:*

Each library typically contains a git submodule with the C++ source and a `bindings/` folder with the litgen generation script and generated nanobind C++ code.

```
external/
│
│   ── Core libraries (with Python bindings) ──────────────────────────
│
├── imgui/                                  Dear ImGui — core widget
library (imgui_bundle fork)
│     ├── imgui/                            Submodule
│     └── bindings/                         litgen options + generated
pybind code
│
├── hello_imgui/                            Hello ImGui — window
lifecycle, docking, DPI, assets
│     ├── hello_imgui/                      Submodule
│     └── bindings/
│
├── immapp/                                 ImmApp — high-level runner
with add-on initialization
│     └── bindings/
│
│   ── Plotting & visualization (with bindings) ──────────────────────
│
```

```
├── implot/                                 ImPlot — 2D plotting
├── implot3d/                               ImPlot3D — 3D plotting
├── immvision/                              ImmVision — image debugger
(zoom, colormaps, pixel inspection)
├── imgui_tex_inspect/                      imgui_tex_inspect — texture
inspector
│
│   ── Text editing & markdown (with bindings) ──────────────────────
│
├── ImGuiColorTextEdit/                     Code editor with syntax
highlighting
├── imgui_md/                               Markdown rendering (MD4C-
based)
├── imgui_microtex/                         Native LaTeX math rendering
(MicroTeX + FreeType backend)
│     ├── MicroTeX/                         Submodule (forked, branch
imgui_bundle)
│     └── imgui_microtex/                   Our wrapper: public API +
FreeType graphics backend
│
│   ── Tools (with bindings) ──────────────────────────────────────────
│
├── ImGuizmo/                               3D gizmo for scene editing
│     ├── ImGuizmo/                         Submodule
│     └── ImGuizmoPure/                     Manual wrappers to help
binding generation
├── imgui-node-editor/                      Node graph editor
├── nanovg/                                 Antialiased 2D vector
drawing (OpenGL)
├── imgui_explorer/                         Interactive widget explorer/
demo app
│
│   ── Widgets (with bindings) ─────────────────────────────────────────
│
├── ImFileDialog/                           File dialog
├── portable_file_dialogs/                  OS-native file dialogs
(single-header)
├── imgui-knobs/                            Knob/dial widgets
├── imspinner/                              Spinner/loading widgets
├── imgui_toggle/                           Toggle switches
├── ImCoolBar/                              macOS-style cool toolbar
├── imgui-command-palette/                  VSCode-style command palette
├── ImAnim/                                 Animation library
│
│   ── Testing (with bindings) ─────────────────────────────────────────
│
├── imgui_test_engine/                      Dear ImGui test & automation
engine
│
│   ── Support libraries (no Python bindings) ──────────────────────────
│
├── glfw/                                   GLFW window/input backend
```

```
├── fplus/                                  Functional programming C++
library
│
│   ── Binding generation tooling ──────────────────────────────────
│
├── bindings_generation/
│     ├── autogenerate_all.py               Master script: regenerates
all bindings
│     ├── all_external_libraries.py         Registry of all libraries
│     ├── bundle_libs_tooling/              Helpers for external library
management
│     └── bindings_generator_template/      Template for adding new
library bindings
│
└── _doc/                                   Documentation about bindings
(Howto)
```

Binding generation pattern (e.g. for ImCoolBar):

```
external/ImCoolBar/
├── ImCoolBar/                              Git submodule (C++ source)
│     ├── ImCoolbar.h                       C++ header (input to litgen)
│     └── ImCoolbar.cpp
└── bindings/
      ├── generate_imcoolbar.py             litgen script: reads .h →
generates pybind + stubs
      ├── pybind_imcoolbar.cpp              Generated nanobind C++ code
      └── im_cool_bar.pyi                   Symlink → bindings/
imgui_bundle/im_cool_bar.pyi
```

7.c.iv. *`imgui_bundle_cmake/` – Build infrastructure:*

```
imgui_bundle_cmake/
├── imgui_bundle_build_lib.cmake            Main build logic (the bulk
of the build system)
├── imgui_bundle_add_app.cmake              imgui_bundle_add_app():
create an app in one line
├── imgui_bundle_add_demo.cmake             Helper for adding demo
targets
├── imgui_bundle_config.h                   C++ config header (#defines
for enabled features)
├── imgui_bundle_pyodide.cmake              Pyodide/Emscripten web build
configuration
├── imgui_bundle_conda.cmake                Conda dependency
configuration
├── imgui-bundleConfig.cmake                CMake package config (for
find_package)
├── ios-cmake/                              iOS CMake toolchain (symlink
→ hello_imgui)
└── internal/
      ├── add_imgui.cmake                   Integrates Dear ImGui into
the build
```

```
      ├── add_hello_imgui.cmake             Integrates Hello ImGui
      ├── add_imgui_bundle_bindings.cmake   Integrates Python bindings
      ├── add_simple_library.cmake          Generic helper: add a
library with bindings
      ├── add_glfw_submodule.cmake          GLFW backend setup
      ├── litgen_setup_module.cmake         litgen binding generation
setup
      └── dump_cmake_variables.cmake        Debug: print all CMake
variables
```

7.c.v. `docs/book/` – Documentation (Jupyter Book):

The documentation is structured as a Jupyter Book and published as a website + PDF.

```
docs/book/
├── _toc.yml                              Table of contents
├── myst.yml                              MyST Markdown configuration
│
├── intro/                                Introduction
│     ├── intro.md                        Landing page
│     ├── what_is_imgui_bundle.md         Project overview &
comparisons
│     ├── key_features.md                 Feature list, library
catalog, FAQ
│     ├── imm_gui.md                      The immediate mode paradigm
explained
│     ├── interactive_manuals.md          Online demos & explorer
│     ├── examples_gallery.md             Curated examples
│     └── resources.md                    Links & resources
│
├── python/                               For Python users
│     ├── python_imgui_intro.md           ImGui introduction for
Python developers
│     ├── python_install.md               pip install, setup
│     ├── python_assets.md                Asset management
│     ├── pure_python_backend.md          Pure Python backends (no
compilation)
│     ├── python_async.md                 Async/await patterns
│     ├── notebook_runners.ipynb          Jupyter notebook integration
│     ├── python_pyodide.md               Web deployment via Pyodide
│     ├── python_tips.md                  Python-specific tips
│     └── desktop_deploy.md               Desktop deployment
│
├── cpp/                                  For C++ users
│     ├── cpp_install.md                  C++ installation & CMake
setup
│     └── cpp_assets.md                   C++ asset management
│
├── core_libs/                            Core library documentation
│     ├── imgui.md                        Dear ImGui
│     ├── hello_imgui_immapp.md           Hello ImGui & ImmApp
```

```
│     └── test_engine.md                    Test engine
│
├── addons/                                 Add-on library documentation
│     ├── plotting.md                       ImPlot & ImPlot3D
│     ├── visualization.md                  ImmVision
│     ├── text_markdown.md                  Markdown & code editor
│     ├── tools.md                          Gizmo, node editor, NanoVG
│     └── widgets.md                        Knobs, toggles, spinners,
etc.
│
├── support/                                Support & closing
│     ├── support_project.md
│     └── closing_words.md
│
├── devel_docs/                             Developer documentation
(this section)
│     ├── intro.md                          Architecture overview
│     ├── structure.md                      Repository structure (this
file)
│     ├── bindings_intro.md                 How litgen binding
generation works
│     ├── bindings_update.md                Updating existing library
bindings
│     ├── bindings_newlib.md                Adding a new library to the
bundle
│     └── bindings_debug.md                 Debugging native C++ from
Python
│
└── images/                                 Documentation images
```

7.c.vi. *Architecture layers:*

ImGui Bundle implements a four-layer architecture:

```
┌──────────────────────────────────────────────────────┐
│  4. ImmApp Layer                                     │  immapp.run() —
simplified runner
│     Auto-initializes add-ons (ImPlot, Markdown...)   │
with_implot=True, with_markdown=True
├──────────────────────────────────────────────────────┤
│  3. Hello ImGui Framework                            │  Window
lifecycle, docking, DPI,
│     Asset management, themes, callbacks              │  multi-platform
deployment
├──────────────────────────────────────────────────────┤
│  2. Dear ImGui + Add-on Libraries                    │  Immediate-mode
widgets, plotting,
│     ImPlot, ImGuizmo, node editor, ImmVision...      │  gizmos, image
inspection, etc.
├──────────────────────────────────────────────────────┤
│  1. Rendering Backends                               │  GLFW/SDL +
```

```
OpenGL/Metal/DirectX/
│     Platform abstraction                                  │  Vulkan/WebGPU
└─────────────────────────────────────────────────────────┘
```

Users work at their preferred level: from raw ImGui calls to `immapp.run()`.

7.c.vii. *Binding generation flow:*

All 23+ C++ libraries compile into a single `_imgui_bundle` native extension:

```
C++ header (.h)
      │
      ▼
litgen (generate_*.py)         Reads C++ headers via srcML
      │
      ├──► pybind_*.cpp        nanobind C++ code (compiled into
_imgui_bundle.so)
      └──► *.pyi               Python type stubs (IDE autocompletion +
type checking)
```

Regenerate all bindings: `python external/bindings_generation/autogenerate_all.py`

7.d. Build Guide

This page documents CMake options, presets, the justfile task runner, and common build scenarios.

7.d.i. *CMake presets:*

The repo ships a `CMakePresets.json` with these presets:

Preset	Description
`default`	Desktop C++ with GLFW + OpenGL3
`default_sdl`	Desktop C++ with SDL2 + OpenGL3
`python_bindings`	Python bindings (forces GLFW + OpenGL3)
`all_cpp_options`	Everything enabled: all backends, demos, tests, ImmVision with OpenCV

Usage:

```
mkdir -p builds/my_build && cd builds/my_build
cmake ../.. --preset python_bindings -DPython_EXECUTABLE=$(which python)
cmake --build . -j
```

7.d.ii. *Module selection (`IMGUI_BUNDLE_WITH_*`):*

Each library can be individually enabled or disabled. All default to ON.

Option	Library	Notes
`IMGUI_BUNDLE_WITH_HELLO_IMGUI`	Hello ImGui	Disabling also disables immapp, imgui_md, test_engine
`IMGUI_BUNDLE_WITH_IMMAPP`	ImmApp	Depends on hello_imgui + imgui_node_editor
`IMGUI_BUNDLE_WITH_IMPLOT`	ImPlot	2D plotting
`IMGUI_BUNDLE_WITH_IMPLOT3D`	ImPlot3D	3D plotting
`IMGUI_BUNDLE_WITH_IMGUI_NODE_EDITOR`	Node editor	
`IMGUI_BUNDLE_WITH_IMGUI_MD`	Markdown	Depends on immapp
`IMGUI_BUNDLE_WITH_IMGUIZMO`	ImGuizmo	3D gizmos
`IMGUI_BUNDLE_WITH_NANOVG`	NanoVG	Needs OpenGL3 or Metal
`IMGUI_BUNDLE_WITH_IMFILEDIALOG`	ImFileDialog	Needs OpenGL3
`IMGUI_BUNDLE_WITH_IMGUI_TEX_INSPECT`	Texture inspector	Needs OpenGL3
`IMGUI_BUNDLE_WITH_IMMVISION`	ImmVision	Needs OpenGL3
`IMGUI_BUNDLE_WITH_IMANIM`	ImAnim	Animation library
`IMGUI_BUNDLE_WITH_IMGUI_EXPLORER`	ImGui Explorer	Interactive widget explorer

Dependency chain: CMake automatically disables dependent modules. For example, setting `IMGUI_BUNDLE_WITH_HELLO_IMGUI=OFF` also disables immapp, imgui_md, test_engine, and the GLFW backend.

Convention: CMake option names and C++ `#define` names are identical. For example, `IMGUI_BUNDLE_WITH_IMANIM` is both the `option()` and the `#ifdef` guard in C++.

Legacy options: `IMGUI_BUNDLE_DISABLE_*` options are deprecated (removal ~ January 2028). Use `IMGUI_BUNDLE_WITH_*=OFF` instead.

7.d.iii. *Backend selection:*

You need at least one platform backend and one rendering backend:

Platform backends:

Option	Backend
`HELLOIMGUI_USE_GLFW3`	GLFW3 (recommended default)
`HELLOIMGUI_USE_SDL2`	SDL2

Rendering backends:

Option	Backend	Notes
`HELLOIMGUI_HAS_OPENGL3`	OpenGL3	Recommended, especially for beginners
`HELLOIMGUI_HAS_METAL`	Metal	macOS/iOS only
`HELLOIMGUI_HAS_VULKAN`	Vulkan	Advanced
`HELLOIMGUI_HAS_DIRECTX11`	DirectX 11	Windows only, experimental
`HELLOIMGUI_HAS_DIRECTX12`	DirectX 12	Windows only, experimental

If you make no choice, the default is **GLFW3 + OpenGL3**.

> **Note**
>
> Python bindings always force GLFW3 + OpenGL3. Pyodide forces SDL2 + OpenGL3.

7.d.iv. *Other build options:*

Option	Default	Description
`IMGUI_BUNDLE_BUILD_PYTHON`	OFF (ON via pip)	Build Python bindings
`IMGUI_BUNDLE_BUILD_DEMOS`	ON (top-level)	Build C++ demo executables
`IMGUI_BUNDLE_BUILD_IMGUI_EXPLORER_APP`	OFF	Build ImGui Explorer standalone app

Option	Default	Description
IMGUI_BUNDLE_INSTALL_CPP	ON (top-level, non-Python)	Install C++ targets
IMGUI_BUNDLE_AUTO_CLONE_SUBMODULES	ON	Auto-clone submodules during configure
HELLOIMGUI_WITH_TEST_ENGINE	ON (desktop)	Include ImGui Test Engine
HELLOIMGUI_USE_FREETYPE	auto	Use FreeType for font rendering
IMMVISION_FETCH_OPENCV	OFF	Fetch & build OpenCV for ImmVision
IMGUI_BUNDLE_BUILD_CI_AUTOMATION_TESTS	OFF	Build CI automation tests
IMGUI_BUNDLE_WITH_IMANIM_FULL_DEMOS	ON	Include all ImAnim author demos in manual

For pip builds, pass CMake options via the `CMAKE_ARGS` environment variable:

```
CMAKE_ARGS="-DIMGUI_BUNDLE_WITH_IMMVISION=OFF" pip install -v -e .
```

7.d.v. *Common build scenarios:*

Desktop C++ (minimal):

```
mkdir -p builds/desktop && cd builds/desktop
cmake ../.. --preset default
cmake --build . -j
```

Python bindings with ImmVision:

```
mkdir -p builds/python && cd builds/python
cmake ../.. --preset python_bindings \
    -DPython_EXECUTABLE=$(which python) \
    -DIMMVISION_FETCH_OPENCV=ON
cmake --build . -j
```

Emscripten (ImGui Bundle Explorer):

```
just ibex_build       # builds into build_ibex_ems/
just ibex_serve       # serve on port 8642
```

Or manually:

```
mkdir -p builds/ems && cd builds/ems
source ~/emsdk/emsdk_env.sh
emcmake cmake ../.. -DCMAKE_BUILD_TYPE=Release
make -j
```

Emscripten (ImGui Explorer, lightweight):

```
just imex_ems_build   # builds into build_imex_ems/
just imex_ems_serve   # serve on port 7006, add ?lib=imgui etc. to URL
```

Pyodide (Python in the browser):

```
just pyodide_setup_local_build    # one-time setup
just pyodide_build                # build wheel
just pyodide_test_serve           # serve test page
```

See also `ci_scripts/pyodide_local_build/Readme.md`.

Offline build (vcpkg):

```
git clone https://github.com/microsoft/vcpkg && ./vcpkg/bootstrap-
vcpkg.sh
./vcpkg/vcpkg install opencv freetype glfw3 lunasvg
export CMAKE_TOOLCHAIN_FILE=vcpkg/scripts/buildsystems/vcpkg.cmake
mkdir build && cd build && cmake ..
```

7.d.vi. *justfile reference:*

The `justfile` at the repo root provides shortcuts for common tasks. Run `just` to list all commands.

Library management (`just libs_*`):

Command	Description
`just libs_info`	Show all libraries with their remotes and branches
`just libs_reattach`	Reattach all submodules to branches (fork + official)
`just libs_fetch`	Fetch all remotes for all submodules
`just libs_pull`	Pull all submodules
`just libs_check_upstream`	Check which forks have new upstream changes
`just libs_log <name>`	Show new upstream commits for a fork library
`just libs_rebase <name>`	Tag and rebase a fork on its upstream
`just libs_tag <name>`	Push a date tag to a fork library
`just libs_bindings <name>`	Regenerate bindings for one library
`just libs_bindings_all`	Regenerate all bindings

Build & deploy:

Command	Description
`just ibex_build`	Build ImGui Bundle Explorer (Emscripten + OpenCV)
`just ibex_serve`	Serve explorer on port 8642
`just imex_ems_build`	Build ImGui Explorer (Emscripten, lightweight)
`just imex_ems_serve`	Serve ImGui Explorer on port 7006
`just imex_ems_deploy`	Deploy ImGui Explorer to GitHub Pages

Documentation:

Command	Description
`just doc_serve_interactive`	Build docs with live reload
`just doc_build_static`	Build static HTML
`just doc_serve_static`	Serve static HTML on port 7005
`just doc_build_pdf`	Build PDF

Testing:

Command	Description
`just test_pytest`	Run pytest
`just test_mypy`	Run mypy on bindings

Pyodide:

Command	Description
`just pyodide_setup_local_build`	Install tools for local Pyodide builds
`just pyodide_build`	Build Pyodide wheel
`just pyodide_test_serve`	Serve browser test page
`just pyodide_clean`	Clean Pyodide build artifacts

CI / Docker:

Command	Description
`just cibuild_docker_musllinux`	Run a musllinux Docker container with repo mounted
`just cibuild_docker_manylinux`	Run a manylinux Docker container with repo mounted

7.e. Testing

7.e.i. Quick start:

```
just test_pytest    # Run the test suite
just test_mypy      # Type-check the Python bindings
```

Or without `just`:

```
pytest                              # from repo root
cd bindings && ./mypy_bindings.sh # mypy
```

7.e.ii. What the test suite covers:

- **`tests/lg_imgui_bundle_test.py`** — Main test file: exercises the litgen binding generation for imgui_bundle.
- **`bindings/mypy_bindings.sh`** — Runs mypy on all `.pyi` stubs to verify type consistency.

7.e.iii. GUI tests:

GUI tests live in `tests/tests_python_gui/` and require a display (they open windows). They are not run by default in CI.

To run them locally:

```
pytest tests/tests_python_gui/
```

These tests verify:

- Widget rendering (e.g. `test_color_edit3_tuple.py`)
- RunnerParams identity across Python/C++ boundary (`test_runner_params_identity.py`)
- Async integration (`async/`)

7.e.iv. Manual smoke testing:

The most thorough manual test is to run the main demo application, which exercises most libraries:

C++:

```
cd builds/my_build
./demo_imgui_bundle
```

Python:

```
python bindings/imgui_bundle/demos_python/demos_immapp/
demo_hello_world.py
```

7.e.v. CI workflows:

The project has extensive CI via GitHub Actions (`.github/workflows/`):

Workflow	What it tests
`cpp_lib.yml`	C++ library build (multiple platforms)
`cpp_lib_with_bindings.yml`	C++ build with Python bindings
`pip.yml`	`pip install` from source
`wheels.yml`	Build distributable wheels (cibuildwheel)
`emscripten.yml`	Emscripten / WebAssembly build
`pyodide.yml`	Pyodide / browser build
`Metal.yml`	macOS Metal renderer
`Vulkan.yml`	Vulkan renderer
`DirectX.yml`	Windows DirectX
`android.yml`	Android build
`ios.yml`	iOS build
`ci_automation_test.yml`	Automated GUI test (runs demo, takes screenshots)

Most workflows trigger on push to main and on pull requests.

> **Tip**
>
> If you haven't built the project yet, start with Getting Started (Developer).

The bindings are generated automatically thanks to a sophisticated generator, which is based on srcML.

The generator is provided by litgen (Literate Generator), an automatic Python bindings generator developed by the same author as Dear ImGui Bundle. See also the litgen PDF manual for in-depth documentation.

7.f.i. *Installing the generator:*

See the installation instructions (do a local installation).

7.f.ii. *Quick information about the generator:*

`litgen` (aka "Literate Generator") is the package that will generate the python bindings.

Its source code is available here.

It is heavily configurable by a wide range of options.

See for examples the specific options for imgui bindings generation.

7.f.iii. *Folders structure:*

In order to work on bindings, it is essential to understand the folders structure inside Dear ImGui Bundle. Please study the dedicated doc.

7.f.iv. *Study of a bound library generation:*

Let's take the example of the library ImCoolBar.

> **Tip**
>
> The processing of adding a new library from scratch is documented in Adding a new library. It uses ImCoolBar as an example

Here is how the generation works for the library. The library principal files are located in external/ImCoolBar:

```
external/ImCoolBar/                    # Root folder for ImCoolBar
├── ImCoolBar/                         # ImCoolBar submodule
│         ├── CMakeLists.txt           # ImCoolBar code
│         ├── ImCoolbar.cpp
│         ├── ImCoolbar.h
│         ├── LICENSE
│         └── README.md
```

```
└── bindings/                                  # Scripts for the bindings
generations & bindings
    ├── generate_imcoolbar.py                  # This script reads
ImCoolbar.h and generates:
    |                                          #     - binding C++ code
in ./pybind_imcoolbar.cpp
    |                                          #     - stubs in
    |                                          #          bindings/
imgui_bundle/im_cool_bar_pyi
    ├── im_cool_bar.pyi -> ../../../bindings/imgui_bundle/
im_cool_bar.pyi   # this is a symlink!
    └── pybind_imcoolbar.cpp
```

The actual stubs are located here:

```
imgui_bundle/bindings/imgui_bundle/
├── im_cool_bar.pyi              # Location of the stubs
├── __init__.pyi                 # Main imgui_bundle stub file, which
loads im_cool_bar.pyi
├── __init__.py                  # Main imgui_bundle python module which
loads
|                                # the actual im_cool_bar module
├── ...
```

And the library is referenced in a global generation script:

```
imgui_bundle/external/bindings_generation/
├── autogenerate_all.py          # This script will call
generate_imcoolbar.py (among many others)
├── all_external_libraries.py    # ImCoolBar is referenced here
├── ...
```

7.f.v. *Regenerating bindings via justfile:*

The `justfile` provides convenient shortcuts for binding generation and library management:

```
just libs_bindings im_cool_bar    # Regenerate bindings for one library
just libs_bindings_all            # Regenerate all bindings
just libs_info                    # Show all libraries with their
remotes
just libs_check_upstream          # Check which forks have new upstream
changes
```

See Update Bindings for the full update workflow, and Managing external libraries and forks for how forks, remotes, and bundle-specific changes are managed.

> **Tip**
>
> This page explains the fork model, conventions, and tooling for managing ImGui Bundle's external libraries. For the step-by-step update workflow, see Update existing bindings. For adding a new library, see Adding a new library.

7.g.i. *Direct vs. forked libraries:*

ImGui Bundle includes ~24 external C++ libraries. Each lives under `external/<LibName>/` and is typically a git submodule.

Some libraries can be used as-is from their upstream repository. Others need modifications to work with Python bindings — for example, replacing `pointer + count` C APIs with `std::vector`, or swapping C function pointers for `std::function`. In those cases, a **fork** is maintained on GitHub under github.com/pthom.

Run `just libs_info` to see the full picture:

```
NAME                      FORK
OFFICIAL                                                PATH
--------------------------------------------------------------------------------
imgui                     https://github.com/pthom/imgui.git
https://github.com/ocornut/imgui.git                    external/imgui/
imgui
imgui_test_engine         https://github.com/pthom/imgui_test_engine.git
https://github.com/ocornut/imgui_test_engine.git        external/
imgui_test_engine/imgui_test_engine
glfw
https://github.com/glfw/glfw.git                        external/glfw/glfw
hello_imgui
https://github.com/pthom/hello_imgui.git                external/
hello_imgui/hello_imgui
im_cool_bar               https://github.com/pthom/ImCoolBar.git
https://github.com/aiekick/ImCoolBar.git                external/
ImCoolBar/ImCoolBar
im_file_dialog            https://github.com/pthom/ImFileDialog.git
https://github.com/dfranx/ImFileDialog.git              external/
ImFileDialog/ImFileDialog
imgui_command_palette     https://github.com/pthom/imgui-command-
palette.git    https://github.com/hnOsmium0001/imgui-command-palette.git
external/imgui-command-palette/imgui-command-palette
imgui_knobs               https://github.com/pthom/imgui-knobs.git
https://github.com/altschuler/imgui-knobs               external/imgui-
knobs/imgui-knobs
imgui_node_editor         https://github.com/pthom/imgui-node-editor.git
https://github.com/thedmd/imgui-node-editor.git         external/imgui-
node-editor/imgui-node-editor
imgui_md                  https://github.com/pthom/imgui_md.git
https://github.com/mekhontsev/imgui_md                  external/imgui_md/
imgui_md
```

```
md4c
https://github.com/mity/md4c                          external/imgui_md/
md4c
imgui_microtex            https://github.com/pthom/MicroTeX.git
https://github.com/NanoMichael/MicroTeX.git           external/
imgui_microtex/MicroTeX
imgui_tex_inspect         https://github.com/pthom/imgui_tex_inspect.git
https://github.com/andyborrell/imgui_tex_inspect.git  external/
imgui_tex_inspect/imgui_tex_inspect
imgui_toggle              https://github.com/pthom/imgui_toggle.git
https://github.com/cmdwtf/imgui_toggle.git            external/
imgui_toggle/imgui_toggle
imgui_color_text_edit     https://github.com/pthom/ImGuiColorTextEdit.
git       https://github.com/goossens/ImGuiColorTextEdit.git
external/ImGuiColorTextEdit/ImGuiColorTextEdit
imguizmo                  https://github.com/pthom/ImGuizmo.git
https://github.com/CedricGuillemet/ImGuizmo.git       external/ImGuizmo/
ImGuizmo
immvision
https://github.com/pthom/immvision.git                external/
immvision/immvision
implot                    https://github.com/pthom/implot.git
https://github.com/epezent/implot.git                 external/implot/
implot
implot3d                  https://github.com/pthom/implot3d.git
https://github.com/brenocq/implot3d.git               external/implot3d/
implot3d
nanovg                    https://github.com/pthom/nanovg.git
https://github.com/memononen/nanovg.git               external/nanovg/
nanovg
im_anim                   https://github.com/pthom/ImAnim.git
https://github.com/soufianekhiat/ImAnim.git           external/ImAnim/
ImAnim
```

- Libraries with a **FORK** column are maintained as forks with bundle-specific patches.
- Libraries with only an **OFFICIAL** column are used directly from upstream.
- A few internal libraries (e.g. `immapp`, `imgui_explorer`) have no git URL — they live directly in the repo.

When is a fork created?:

A fork is created when a library needs C++ source modifications that cannot be achieved through the binding generator alone. Typical reasons:

- **Python API adaptations**: replacing `pointer + size` patterns with `std::vector`, C function pointers with `std::function`, `const char*` with `std::optional<std::string>`, etc.
- **Bug fixes** or behavior changes specific to ImGui Bundle's usage.
- **Build adaptations** for cross-platform support.

If a library can be bound without source modifications, it is used directly from upstream (no fork needed).

7.g.ii. *The fork branch model:*

Branches and remotes:

Each forked library has **two git remotes**:

Remote	Points to	Branch tracked
`official`	Upstream author's repo (e.g. `ocornut/imgui`)	The upstream branch (e.g. `docking`, `master`, `main`)
`fork`	pthom's fork (e.g. `pthom/imgui`)	`imgui_bundle`

The fork branch is always named **`imgui_bundle`**. All bundle-specific changes live on this branch.

Non-forked libraries have a single `official` remote pointing to their upstream repo.

> **Note**
>
> Dear ImGui tracks the **`docking`** branch of upstream (not `master`), since ImGui Bundle uses the docking-enabled version. The fork branch is still `imgui_bundle`.

Rebase, not merge:

When updating a fork to incorporate new upstream changes, we **rebase** the `imgui_bundle` branch on top of the latest upstream — we do not merge. This keeps the bundle-specific diff small, self-contained, and easy to review. It also makes future rebases easier since there are no merge commits to carry forward.

The typical update sequence is:

1. **Tag** the current fork state (safety snapshot): `just libs_tag <name>`
2. **Rebase** the fork branch on upstream: `just libs_rebase <name>`
3. **Force-push** the rebased branch to the fork remote (if the rebase changed anything)

See Update existing bindings for the full step-by-step workflow.

7.g.iii. *Marking bundle-specific changes:*

In code: `[ADAPT_IMGUI_BUNDLE]` markers:

All bundle-specific modifications in forked library source code are bracketed with comment markers:

```
// [ADAPT_IMGUI_BUNDLE]
// ... bundle-specific code ...
// [/ADAPT_IMGUI_BUNDLE]
```

This makes it easy to find all bundle adaptations by searching for `ADAPT_IMGUI_BUNDLE`, and to review the diff against upstream.

Example from `imgui.h` — adding a Python-friendly overload:

```
//[ADAPT_IMGUI_BUNDLE]
IMGUI_API void ShowDemoWindow_MaybeDocked(
    bool create_window, bool* p_open = NULL,
    ImGuiWindowFlags initial_extra_flags = 0,
    ImVec2 window_pos = ImVec2(0, 0), ImVec2 window_size = ImVec2(0,
0));
//[/ADAPT_IMGUI_BUNDLE]
```

In commits: `[Bundle]` prefix:

Bundle-specific commits use a **`[Bundle]`** prefix in the commit message. This clearly distinguishes them from upstream commits when reading the git log:

```
$ git log --oneline (imgui fork)
8f3f4c6 [Bundle] Add python API for SetWindowFocus(optional<string>)
4a7d77e [Bundle] imgui_demo.cpp: imgui_manual -> imgui explorer
245cc3e [Bundle] Python API for SliderFloat2/4, InputFloat2/4,
ColorEdit3/4, ColorPicker3/4
f6ba577 [Bundle] imgui_demo.cpp: move demo marker / Layout/Stack Layout
into TreeNode
8e8f3ed [Bundle]: add ShowDemoWindow_MaybeDocked (with flags & position)
...
```

Some older commits may use `[ADAPT_IMGUI_BUNDLE]` as the prefix — this is equivalent.

Preprocessor macros:

Two preprocessor macros control conditional compilation in forked code:

- **`IMGUI_BUNDLE_PYTHON_API`**: Code inside `#ifdef IMGUI_BUNDLE_PYTHON_API` is compiled **only** when building Python bindings. Use this to provide Python-friendly alternatives.
- **`IMGUI_BUNDLE_PYTHON_UNSUPPORTED_API`**: Code inside this guard is **excluded** from Python bindings (it is always defined, but the binding generator skips it). Use this to hide C++ APIs that cannot be wrapped.

These are often used together to swap an unwrappable API for a Python-friendly one:

```
// From imgui-node-editor — replacing pointer+size with std::vector
#ifdef IMGUI_BUNDLE_PYTHON_UNSUPPORTED_API
IMGUI_NODE_EDITOR_API int GetSelectedNodes(NodeId* nodes, int size);
IMGUI_NODE_EDITOR_API int GetSelectedLinks(LinkId* links, int size);
#endif
#ifdef IMGUI_BUNDLE_PYTHON_API
IMGUI_NODE_EDITOR_API std::vector<NodeId> GetSelectedNodes();
```

```
IMGUI_NODE_EDITOR_API std::vector<LinkId> GetSelectedLinks();
#endif

// From imgui.h — replacing function pointer with std::function
#ifdef IMGUI_BUNDLE_PYTHON_API
using ImGuiInputTextCallback =
std::function<int(ImGuiInputTextCallbackData*)>;
#else
#ifdef IMGUI_BUNDLE_PYTHON_UNSUPPORTED_API
typedef int (*ImGuiInputTextCallback)(ImGuiInputTextCallbackData* data);
#endif
#endif
```

See Adding a new library (Step 3) for more adaptation patterns.

Contributing fixes upstream:

If a fix or improvement is **not** specific to ImGui Bundle (e.g. a genuine bug fix, a useful feature), it should be contributed upstream:

- **Do not** use `[Bundle]` prefix or `[ADAPT_IMGUI_BUNDLE]` markers.
- **Do not** use `IMGUI_BUNDLE_PYTHON_API` / `IMGUI_BUNDLE_PYTHON_UNSUPPORTED_API` guards.
- Submit a PR to the upstream repository.
- Once merged upstream, the fix will be picked up during the next rebase.

7.g.iv. *Tooling:*

Python module: `bundle_libs_tooling`:

The library management logic lives in `external/bindings_generation/bundle_libs_tooling/`:

File	Purpose
`all_external_libraries.py`	Central registry — defines all ~24 libraries as `ExternalLibrary` instances in the `ALL_LIBS` list. Contains functions for bulk operations (reattach, fetch, pull, check upstream).
`external_library.py`	`ExternalLibrary` dataclass — holds name, git URLs, branches, remote names, and provides methods for git operations (add remotes, fetch, rebase, tag, etc.).
`shell_commands.py`	`ShellCommands` helper — runs multiline shell commands with echo and error handling.

The `ExternalLibrary` dataclass fields:

```
@dataclass
class ExternalLibrary:
    name: str                          # Library name (e.g. "ImCoolBar")
```

```
    official_git_url: Optional[str]    # Upstream repo URL
    official_branch: str = "master"    # Upstream branch to track
    fork_git_url: Optional[str]        # Fork repo URL (None if not
forked)
    fork_branch: str = "imgui_bundle"  # Fork branch (always
"imgui_bundle")
    is_published_in_python: bool       # Whether this lib gets Python
bindings
    is_sub_library: bool               # Sub-library (e.g. md4c inside
imgui_md)
```

Justfile recipes: `just libs_*`:

The `justfile` at the repo root wraps the Python tooling into convenient commands:

Command	Description
`just libs_info`	Show all libraries with their remotes and paths
`just libs_reattach`	Reattach all submodules to their correct branches and remotes
`just libs_fetch`	Fetch all remotes for all submodules
`just libs_pull`	Pull all submodules
`just libs_check_upstream`	Check which forked libraries have new upstream changes
`just libs_log <name>`	Show new upstream commits not yet in a fork
`just libs_tag <name>`	Push a dated tag (`bundle_YYYYMMDD`) to a fork
`just libs_rebase <name>`	Tag current state, then rebase fork branch on upstream
`just libs_bindings <name>`	Regenerate Python bindings for one library
`just libs_bindings_all`	Regenerate all Python bindings

Library names can be passed in snake_case or original form (e.g. `just libs_log imgui` or `just libs_log ImGuiColorTextEdit`).

7.g.v. *Common operations:*

After a fresh clone: reattach submodules:

By default, git submodules are in "detached HEAD" mode. To set up the correct remotes (`official` / `fork`) and attach each submodule to its tracking branch:

```
just libs_reattach
```

This is typically the **first thing to do** after cloning the repository when you plan to work on library updates.

Check what's new upstream:

```
just libs_check_upstream
```

Example output:

```
Unchanged libraries: glfw, hello_imgui, ImCoolBar, ImFileDialog, imgui-
command-palette, ...
Libraries with new changes in official repo: imgui, imgui_test_engine,
ImGuiColorTextEdit
```

Then inspect the new commits for a specific library:

```
just libs_log imgui
```

Tag a fork (safety snapshot):

Before rebasing, create a dated tag so you can recover the previous state:

```
just libs_tag imgui
```

This creates a tag named `bundle_YYYYMMDD` and pushes it to the fork remote.

Rebase a fork on upstream:

```
just libs_rebase imgui
```

This command:

1. Creates a dated tag (safety snapshot)
2. Fetches both `official` and `fork` remotes
3. Checks out the `imgui_bundle` branch
4. Rebases it on top of `official/<upstream_branch>`

If the rebase introduces changes, you need to **force-push** manually:

```
cd external/imgui/imgui
git push fork --force-with-lease
cd -
```

Update a non-forked library:

For libraries without a fork, simply pull:

```
cd external/glfw/glfw
git pull
cd -
```

Or pull all submodules at once:

```
just libs_pull
```

7.h. Update existing bindings

> **Tip**
>
> Before following this workflow, read Managing external libraries and forks to understand the fork model, conventions, and available tooling.

7.h.i. *Quick reference: justfile commands:*

The `justfile` provides shortcuts for the most common library management tasks:

```
just libs_info              # Show all libraries with their remotes
(fork / official)
just libs_check_upstream    # Check which forks have new upstream
changes
just libs_log <name>        # Show new upstream commits for a library
just libs_rebase <name>     # Tag current state, then rebase fork on
upstream
just libs_tag <name>        # Push a date tag to a fork
just libs_bindings <name>   # Regenerate bindings for one library
just libs_bindings_all      # Regenerate all bindings
just libs_reattach          # Reattach all submodules to their branches
just libs_fetch             # Fetch all remotes
just libs_pull              # Pull all submodules
```

See Managing external libraries and forks for detailed explanations of each command.

7.h.ii. *Typical workflow:*

1. Check what's new upstream:

```
just libs_check_upstream
```

Example output:

```
Unchanged libraries: imgui, glfw, hello_imgui, ImCoolBar, ...
Libraries with new changes in official repo: imgui_test_engine
```

Inspect the new commits:

```
just libs_log imgui_test_engine
```

2. Update the library:

For a non-forked library (e.g. `immvision`, `glfw`):

```
cd external/immvision/immvision
git pull
cd -
```

For a forked library (e.g. `imgui_test_engine`): tag the current state, then rebase on upstream:

```
just libs_rebase imgui_test_engine
```

This is equivalent to:

```
cd external/imgui_test_engine/imgui_test_engine
git tag "bundle_$(date +%Y%m%d)"
git push fork --tags
git rebase official/main
cd -
```

3. Regenerate bindings:

```
just libs_bindings imgui_test_engine
```

Or call the generation script directly:

```
python external/imgui_test_engine/bindings/generate_imgui_test_engine.py
```

Examine the changes in the generated `.cpp` and `.pyi` files with `git diff`.

4. Compile & test:

If you don't have a build directory yet, see Getting Started or Build Guide.

Build:

```
cd builds/my_build
cmake --build . -j
```

Fix any compilation errors due to breaking changes in the upstream API.

Test in C++:

```
./demo_imgui_bundle    # Global demo exercising most libraries
```

Test in Python:

```
python bindings/imgui_bundle/demos_python/demos_immapp/
demo_hello_world.py
```

Run the test suite:

```
just test_pytest   # or: pytest
just test_mypy     # or: cd bindings && ./mypy_bindings.sh
```

See Testing for more details.

5. Push fork changes (if applicable):

If the fork submodule was modified during rebase or to fix binding compatibility:

```
cd external/imgui_test_engine/imgui_test_engine
git push fork
cd -
```

6. Commit:

```
git add -A
git commit -m "Update imgui_test_engine and regenerate bindings"
```

7.h.iii. *Example: update imgui & bindings (detailed):*

> **Tip**
>
> This video demonstrates from start to finish the process of updating imgui and its bindings (17 minutes).

imgui and imgui_test_engine use forks. The full update process:

1. Tag current fork state

```
just libs_tag imgui
just libs_tag imgui_test_engine
```

2. Rebase forks on upstream

```
just libs_rebase imgui
just libs_rebase imgui_test_engine
```

Or manually:

```
cd external/imgui/imgui
git rebase official/docking
cd -

cd external/imgui_test_engine/imgui_test_engine
git rebase official/main
cd -
```

3. Regenerate bindings

```
just libs_bindings imgui
```

This runs external/imgui/bindings/generate_imgui.py, which generates bindings for imgui, imgui_internal, and imgui_test_engine.

4. Examine, build, and test (see steps 3-4 above)

5. Push updated forks

```
cd external/imgui/imgui && git push fork && cd -
cd external/imgui_test_engine/imgui_test_engine && git push fork && cd -
```

7.i. *Adding a new library to the bindings*

This example is based on the addition of ImCoolBar, which was added in Oct 2023.

7.i.i. *Step 1: Reference the new library:*

> **Tip**
>
> All the modifications done in step 1 can be seen in this commit.

Step 1-a: Add needed folders, files and submodules inside external/:

Add the library as a submodule in external/lib_name/lib_name:

If the library can be included without adaptations for inclusion inside ImGui Bundle, you can add it directly as a submodule.

```
mkdir external/ImCoolBar
git submodule add https://github.com/aiekick/ImCoolBar.git external/
ImCoolBar/ImCoolBar
```

However, if it requires adaptations, you need to create a fork (it was the case for ImCoolBar): So, the following actions were done separately:

- ImCoolBar was cloned into github.com/pthom/ImCoolBar.git
- a branch imgui_bundle was created and pushed to github. It will contain the adaptations and bug corrections for imgui_bundle.

Then, we add this fork as a submodule.

```
git submodule add https://github.com/pthom/ImCoolBar.git external/
ImCoolBar/ImCoolBar
cd external/ImCoolBar/ImCoolBar
git checkout imgui_bundle
cd -
```

Create the folder external/lib_name/bindings/:

Copy the folder external/bindings_generation/bindings_generator_template into external/lib_name/bindings/

```
cp -r external/bindings_generation/bindings_generator_template external/
ImCoolBar/bindings
```

Rename files in external/lib_name/bindings:

After having copied the template files, we need to rename them. In the example of ImCoolbar, we will rename them as follows:

```
 mv external/ImCoolBar/bindings/generate_LIBNAME.py external/ImCoolBar/
bindings/generate_imcoolbar.py
mv external/ImCoolBar/bindings/pybind_LIBNAME.cpp external/ImCoolBar/
bindings/pybind_imcoolbar.cpp
# im_cool_bar will be the final name of the python module:
```

```
imgui_bundle.im_cool_bar
mv external/ImCoolBar/bindings/LIBNAME.pyi external/ImCoolBar/bindings/
im_cool_bar.pyi
```

Move external/ImCoolBar/bindings/im_cool_bar.pyi to bindings/imgui_bundle/:

The stub file (*.pyi) *must* be inside bindings/imgui_bundle. In order to facilitate development, we will create a symlink to it inside external/ImCoolBar/bindings/

```
mv external/ImCoolBar/bindings/im_cool_bar.pyi bindings/imgui_bundle/
im_cool_bar.pyi
cd external/ImCoolBar/bindings/
ln -s ../../../bindings/imgui_bundle/im_cool_bar.pyi .
cd -
```

Final folder structure:

We end up with the following structure:

```
external/ImCoolBar/
├── ImCoolBar/                    # Note that the submodule is inside
│         ├── CMakeLists.txt      # external/ImCoolBar/ImCoolBar/ !!!
│         ├── ImCoolbar.cpp
│         ├── ImCoolbar.h
│         ├── LICENSE
│         └── README.md
└── bindings/
    ├── im_cool_bar.pyi               # We will edit and rename those
files later
    ├── generate_imcoolbar.py -> symlink to ../../../bindings/
imgui_bundle/im_cool_bar.pyi
    └── pybind_imcoolbar.cpp
```

Step 1-b: Update python generator manager:

Update external/bindings_generation/all_external_libraries.py

Add a function that returns info about this new library:

```
def lib_imcoolbar() -> ExternalLibrary:
    return ExternalLibrary(
        name="ImCoolBar",
        official_git_url="https://github.com/aiekick/ImCoolBar.git",
        official_branch="master",
        fork_git_url="https://github.com/pthom/ImCoolBar.git",
        fork_branch="imgui_bundle"
    )

ALL_LIBS = [
    lib_imgui(),  # must be first as it declare bindings used by the
next ones
    # ...
```

```
    lib_imcoolbar(),  # Add the lib here
    # ...
```

Step 1-c: Update the C++ sources to include the new lib binding generation:

In external/CMakeLists.txt: Add a cmake directive to compile the new library.

```
# If the library is "simple" to compile you can use
`add_simple_external_library_with_sources`
add_simple_external_library_with_sources(imcoolbar ImCoolBar)
```

In external/bindings_generation/cpp/all_pybind_files.cmake:

add `external/ImCoolBar/bindings/pybind_imcoolbar.cpp`

> **Note**
>
> the script external/bindings_generation/autogenerate_all.py will also regenerate this file from scratch.

In external/bindings_generation/cpp/pybind_imgui_bundle.cpp:

Add the bindings

```
// ... Near the start of the file, add a new function declaration
void py_init_module_imgui_command_palette(py::module& m);
void py_init_module_implot_internal(py::module& m);
void py_init_module_imcoolbar(py::module& m);         // added this line
//  ...


void py_init_module_imgui_bundle(py::module& m)
{
    // ...

    // At the end of py_init_module_imgui_bundle, register your new
python module
    auto module_imcooolbar = m.def_submodule("im_cool_bar"); // the
python module will be known as imgui_bundle.im_cool_bar
    py_init_module_imcoolbar(module_imcooolbar);
```

Now, run cmake.

Step 1-d: Edit and adapt the generation scripts:

Edit the 3 files inside `external/ImCoolBar/bindings` and replace occurrences of LIBNAME with appropriate values.

*Step 1-e: Edit and adapt the imgui_bundle **init** scripts:*

In bindings/imgui_bundle

7.j. Debug native C++ in Python bindings

ImGui Bundle provides tooling to help you debug the C++ side when you encounter a bug that is difficult to diagnose from Python. For this, you will need to clone the repository and build the C++ project `pybind_native_debug` in debug mode. It can be used in two steps:

1. Edit the file `pybind_native_debug/pybind_native_debug.py`. Change its content so that it runs the Python code you would like to debug. Make sure it works when you run it as a Python script.
2. Now, debug the C++ project `pybind_native_debug` which is defined in the directory `pybind_native_debug/`. This will run your Python code from C++, and you can debug the C++ side (place breakpoints, watch variables, etc).

7.k. PyPI package distribution

7.k.i. *Release process:*

1. Update the version number in `pyproject.toml` and `CMakeLists.txt` (they must match). Version scheme: `ImGui patch × 100 + bundle release` (e.g. `1.92.601` = ImGui 1.92.6, bundle release 1).
2. Create a GitHub release with a new tag (e.g. `v1.92.601`). The `wheels.yml` CI workflow builds and uploads wheels to PyPI automatically.
3. Manually build and upload the macOS arm64 wheel (see below).

7.k.ii. *About cibuildwheel:*

cibuildwheel is a tool that builds Python wheels for multiple platforms and Python versions in a consistent, reproducible way. It:

- Runs your build inside isolated environments (Docker on Linux, native on macOS/Windows)
- Builds wheels for all supported Python versions (3.8–3.13+)
- Handles platform-specific quirks (manylinux, musllinux, macOS universal2, etc.)
- Is used both in CI (`wheels.yml`) and for manual local builds

The project's cibuildwheel configuration is in `pyproject.toml` under `[tool.cibuildwheel]`.

7.k.iii. *Manual build using cibuildwheel:*

To target specific Python versions:

```
CIBW_ARCHS_MACOS="arm64" CIBW_BUILD="cp311-* cp312-*" uv tool run
cibuildwheel --platform=macos
```

To build for macOS 11 (disables FreeType): in `pyproject.toml`, change `MACOSX_DEPLOYMENT_TARGET="14.0"` to `"11.0"`.

Upload wheels to PyPI:

```
uv tool run twine upload wheelhouse/*
```

7.k.iv. *Pyodide release:*

Pyodide (Python-in-the-browser via WebAssembly) has its own build and release process. See the Pyodide build guide for:

- Local build environment setup (`just pyodide_setup_local_build`)
- Building the Pyodide wheel (`just pyodide_build`)
- Browser testing (`just pyodide_test_serve`)
- Publishing to the pyodide-recipes repository

7.l. Pyodide Local Build Environment

Goal:

The folder `ci_scripts/pyodide_local_build/` contains tools for building imgui-bundle as a Pyodide package locally (out-of-tree).

Recommended usage: Use the `justfile` targets from the repository root for a streamlined workflow.

```
> just pyodide_
pyodide_build               -- Args: # Build pyodide wheel (excludes
demos to reduce size)
pyodide_clean               -- Args: # Clean pyodide build artifacts
pyodide_deep_clean          -- Args: # Pyodide deep clean (removes also
the local build setup)
pyodide_setup_local_build   -- Args: # Install the tools to build
pyodide wheels locally (pyodide-build, emsdk, etc.)
pyodide_setup_recipe_clone  -- Args: # Clone pyodide-recipes repo and
add fork remote
pyodide_test_serve          -- Args: # Start browser test server (serves
test HTML pages)
```

What's included:

- Python virtual environment with `pyodide-build` installed
- Emscripten SDK (emsdk) with the correct version for Pyodide
- Setup and build scripts
- Browser testing infrastructure

Directory Structure:

```
imgui_bundle/
    ci_scripts/pyodide_local_build/
        ├── Readme.md                       # This file
        ├── config_versions_pyodide.sh      # Version configuration (edit
this to change versions)
        ├── setup_pyodide_local_build.sh    # Automated setup script
        ├── .gitignore                      # Ignores venv_pyo/ and
emsdk/
        ├── venv_pyo/                       # Python virtual environment
(created during setup)
        └── emsdk/                          # Emscripten SDK (created
during setup)
    pyodide_projects/
        ├── pyodide_test_bundle/            # Browser testing tools and
HTML test pages
```

Both `venv_pyo/` and `emsdk/` are gitignored and must be set up locally.

Configuration:

Before setup, you can customize versions by editing `config_versions_pyodide.sh`:

```
# Pyodide version to use (determines ABI compatibility)
PYODIDE_VERSION="0.29.3"

# Python version (major.minor, e.g., "3.13", "3.12", "3.11")
PYTHON_VERSION="3.13"
```

This central configuration file is used by all build scripts in this directory.

Setup Instructions:

Quick Setup (Recommended):

From the repository root, use the justfile target:

```
just pyodide_setup_local_build
```

This will automatically set up the complete build environment.

Manual Setup:

Alternatively, run the setup script directly:

```
cd ci_scripts/pyodide_local_build
./setup_pyodide_local_build.sh
```

The setup script will:

1. Load version configuration from `config_versions_pyodide.sh`
2. Create the Python virtual environment (`venv_pyo/`)
3. Install `pyodide-build`
4. Install the Pyodide cross-compilation toolchain (xbuildenv)
5. Clone and configure Emscripten SDK with the correct version
6. Download the Pyodide distribution for browser testing
7. Verify the installation

Reference: https://pyodide.org/en/stable/development/building-packages.html

Building imgui-bundle:

Quick Build (Recommended):

From the repository root, use the justfile target:

```
just pyodide_build
```

This automatically:

1. Activates the virtual environment
2. Sources the Emscripten environment
3. Builds the wheel with `pyodide build`
4. Fixes the wheel name on macOS (workaround for scikit-build-core issue #920)
5. Copies the wheel to `imgui_bundle/pyodide_projects/_pyodide_resources/local_wheels/` for testing

Manual Build:

If you prefer to run steps manually:

```
# 1. Activate environments
source ci_scripts/pyodide_local_build/venv_pyo/bin/activate
source ci_scripts/pyodide_local_build/emsdk/emsdk_env.sh

# 2. Build from repository root
cd ../..  # Go to imgui_bundle root
pyodide build
```

Output:

After building, you'll find the wheel in the `dist/` directory:

```
dist/imgui_bundle-X.Y.Z-cp313-cp313-pyodide_2025_0_wasm32.whl
```

Browser Testing:

Test your locally built wheel in a browser with Pyodide:

```
just pyodide_test_serve
```

This starts a CORS-enabled web server at http://localhost:8123/ with three test pages:

- `test_local_pyodide.html` - Local Pyodide + local wheel
- `test_cdn_pyodide_local_wheel.html` - CDN Pyodide + local wheel
- `test_cdn_all.html` - Full CDN (official Pyodide package)

See `imgui_bundle/pyodide_projects/pyodide_test_bundle/` directory for more details.

Cleanup:

Remove build artifacts:

```
just pyodide_clean
```

Deep clean (removes build artifacts, downloaded Pyodide distribution, and build environment):

```
just pyodide_deep_clean
```

References:

- **Pyodide Build Docs**: https://pyodide.org/en/stable/development/building-packages.html
- **Pyodide ABI**: https://pyodide.org/en/stable/development/abi.html
- **Emscripten SDK**: https://emscripten.org/docs/getting_started/downloads.html
- **scikit-build-core issue #920**: scikit-build/scikit-build-core#920

7.l.i. *Release a new version of pyodide:*

- After creating a new release for imgui bundle
- Build & test the new version locally with pyodide (see above)
- Upload the wheel to the release assets on github
- Clone https://github.com/pyodide/pyodide-recipes
- Create a branch, e.g. `imgui_bundle_v1.92.600`
- Edit packages/imgui-bundle/meta.yaml, add the new version, and update the URL to point to the new wheel in the github release assets. Update the sha256 hash of the wheel.
- Make a PR to the pyodide-recipes repository, and ask for review and merge.

7.m. *Cloudflare Pages deploy*

The project publishes several static subparts to a single Cloudflare Pages project named `imgui-bundle`, reachable at https://imgui-bundle.pages.dev/:

Path	Contents
`doc/`	Jupyter-book documentation (this book);
`/playground/`	Pyodide playground (Python examples in the browser)
`/min_pyodide_app/`	Minimal Pyodide template (single-file demo)
`/explorer/`	Dear ImGui Bundle Explorer (Emscripten build with OpenCV)
`/local_wheels/`	Shared `imgui_bundle-*.whl`, loaded by the pyodide subparts
`doc/assets/imgui_bundle_book.pdf`	PDF export of the documentation

A `_headers` file (`pyodide_projects/cf_headers`) scopes `Cross-Origin-Opener-Policy` / `Cross-Origin-Embedder-Policy` to `/explorer/*` only; site-wide COI would break the playground (which pulls Pyodide and CodeMirror from CDNs that don't set `Cross-Origin-Resource-Policy`). The same file also sets `Content-Encoding: gzip` on `/explorer/*.data` because those emscripten asset bundles (`application/octet-stream`) aren't auto-compressed by the CF edge; they are pre-gzipped by `cf_stage`.

7.m.i. *How the deploy works:*

Cloudflare Pages replaces the whole site on every upload, so we maintain a local composed staging directory (`pyodide_projects/_cf_staging/`, gitignored) that holds the union of all subparts. `just cf_stage` refreshes the staging tree; `just cf_deploy` uploads it via `wrangler`.

The trade-off: on every deploy the other subparts must already be present (built) on disk. `cf_stage` does not build them — it only copies:

- Pyodide wheel: `just pyodide_build` must have run (output in `pyodide_projects/projects/local_wheels/`).
- Ibex binaries: `just ibex_build` must have run (output in `build_ibex_ems/bin/`).
- Jupyter-book docs: `just doc_build_cf` must have run (output in `docs/book/_build/html/`).

7.m.ii. *One-time setup:*

1. **Install wrangler** (needs Node.js 18+):

```
npm install -g wrangler
```

1. **Create an API token** at https://dash.cloudflare.com/profile/api-tokens using the **Edit Cloudflare Workers** template (it covers Pages too).
2. **Export credentials** in your shell (or your shell rc):

```
export CLOUDFLARE_API_TOKEN=...
export CLOUDFLARE_ACCOUNT_ID=...
```

1. **Verify**: `wrangler whoami` should print the account.

7.m.iii. *Local workflow:*

```
# Build every subpart (slow: ibex with OpenCV takes 30-60 min).
# `cf_stage_prepare` runs: pyodide_setup_local_build + pyodide_build
#                        + ibex_build + doc_build_cf
just cf_stage_prepare

# Stage everything into _cf_staging, then deploy
just cf_stage
just cf_deploy

# Test the composed site locally before deploying
just cf_serve_local        # http://localhost:8765/
# → COOP/COEP headers are applied only to /explorer/* (mirrors prod)
# → Content-Encoding: gzip is applied to /explorer/*.data

# Inspect the staging tree (gitignored)
ls pyodide_projects/_cf_staging/

# Nuke staging (next deploy re-uses on-disk build outputs)
just cf_clean
```

7.m.iv. *CI workflow:*

`.github/workflows/cf_pages_deploy.yml` mirrors the local flow and is triggered manually from the Actions tab (`workflow_dispatch`). It runs:

1. `just pyodide_setup_local_build` + `just pyodide_build` (builds the wheel)
2. `just ibex_build` (builds Emscripten binaries; slowest step, typically 30–60 min because `IMMVISION_FETCH_OPENCV=ON` rebuilds OpenCV for wasm)
3. `just doc_build_cf` (jupyter-book HTML + PDF, needs Typst)
4. `just cf_stage` (composes the staging tree)
5. `npm install -g wrangler` + `just cf_deploy` (uploads)

Required GitHub Actions secrets: `CLOUDFLARE_API_TOKEN`, `CLOUDFLARE_ACCOUNT_ID` (same values used locally).

7.m.v. *Related:*

- Local traineq deploy (kept as fallback): `just pyodide_deploy_imgui_bundle_online`, `just ibex_deploy`

7.m.vi. *Links on imgui-bundle pages at claoudflare:*

- Doc & Root
- Python Playground
- Minimal Pyodide Template
- Minimal Pyodide Template - Source
- ImGui Bundle Explorer

7.n. OpenCV builds for immvision

ImmVision no longer requires OpenCV — it works standalone with its own `ImageBuffer` type. OpenCV is optional and only needed for `cv::Mat` interop in C++. When enabled, a minimalist static OpenCV (only `core`, `imgcodecs`, `imgproc`) is used. How OpenCV is obtained depends on the platform. This document is the central reference for all OpenCV build/download strategies.

When updating the OpenCV version, check all the locations listed below.

7.n.i. *Overview by platform:*

Platform	Strategy	Where
Linux CI wheels	Pre-built once per container via `before-all`	`build_opencv.sh`
Windows CI wheels	Pre-built once per runner via `before-all` (Git Bash)	`build_opencv.sh`
Linux/macOS local `pip install`	Built from source at configure time	`find_opencv.cmake` arrow.r `immvision_fetch_opencv_from_source()`
Windows x64 local `pip install`	Precompiled `opencv_world.dll` (fallback: source build)	`find_opencv.cmake` arrow.r `immvision_download_opencv_official_pa`
Windows ARM64 local `pip install`	Built from source (precompiled x64 package skipped)	`find_opencv.cmake` arrow.r `immvision_fetch_opencv_from_source()`
Emscripten	Precompiled package downloaded	`find_opencv.cmake` arrow.r `immvision_download_emscripten_precomp`
Emscripten (rebuild)	Manual build, upload as release asset	See Rebuilding emscripten package below

7.n.ii. *Key files:*

- **`external/immvision/immvision/cmake/build_opencv.sh`** — Single bash script that downloads, builds, and installs minimalist static OpenCV. Used by both CI (`before-all`) and CMake (called at configure time via `execute_process`). This is the **single source of truth** for the minimalist build flags and the OpenCV version used in source builds. Supports `--full` flag for non-minimalist builds.
- **`external/immvision/immvision/cmake/find_opencv.cmake`** — CMake entry point. `immvision_find_opencv()` tries `find_package(OpenCV)` first (succeeds when CI has pre-built it), then falls back to platform-specific strategies. On Linux/macOS, the fallback calls `build_opencv.sh` via bash. On Windows x64, it tries a precompiled `opencv_world.dll` first, then falls back to source build. On

Windows ARM64, it skips the precompiled x64 package and goes straight to source build.

- **`pyproject.toml`** — cibuildwheel config:
  - `[tool.cibuildwheel.linux]`: `before-all` runs the script, `environment` sets `CMAKE_PREFIX_PATH`
  - `[tool.cibuildwheel.windows]`: same script via Git Bash, `environment` sets `CMAKE_PREFIX_PATH` + `OpenCV_STATIC`
  - `CMAKE_PREFIX_PATH` is used instead of `OpenCV_DIR` because the install layout varies by MSVC version; `find_package` searches standard subdirectories under the prefix.
- **`docs/book/devel_docs/oldies/emscripten_build.md`** — Redirects here for OpenCV.

7.n.iii. *Version and URL locations:*

When bumping the OpenCV version, update these locations:

What	File	What to change
Source build version (all platforms)	`external/immvision/immvision/cmake/build_opencv.sh`	`OPENCV_VERSION` variable
Windows precompiled URL (fallback)	`find_opencv.cmake`	URL + MD5 in `immvision_download_opencv_official_pa`
Emscripten precompiled URL	`find_opencv.cmake`	URL + MD5 in `immvision_download_emscripten_precomp`

Note: the Windows fallback package (`opencv_world.dll`) and emscripten precompiled packages have **their own version lifecycle**. They don't need to match the source build version — they are updated separately when new precompiled packages are built and uploaded as GitHub release assets. The Windows fallback is only used for local `pip install` on Windows (CI wheels use the static pre-build instead).

7.n.iv. *How the CI pre-build works (Linux and Windows):*

Without pre-building, OpenCV is compiled from source for **each** Python version (~4 times per runner), which is the main build time bottleneck.

The fix uses cibuildwheel's `before-all` to build OpenCV once per runner/container:

1. `before-all` in `pyproject.toml` runs `external/immvision/immvision/cmake/build_opencv.sh <install_dir>`
2. The script downloads the OpenCV tarball, builds minimalist static OpenCV, and installs it
3. `environment` sets `OpenCV_DIR` (and `OpenCV_STATIC` on Windows)

4. Each Python wheel build's `find_package(OpenCV)` finds the pre-built install and skips building from source entirely

On Linux, this applies to both manylinux and musllinux containers (each has its own `before-all`). On Windows, Git Bash (from Git for Windows, pre-installed on GitHub Actions runners) is used to run the same bash script.

7.n.v. *Windows: static vs DLL:*

Previously, Windows wheels shipped a precompiled `opencv_world.dll` (~50MB), making wheels ~25MB vs ~11MB on macOS/Linux. The CI pre-build strategy builds OpenCV 4.13.0 statically instead, linking it into `_imgui_bundle.pyd` with no DLL needed. OpenCV 4.13.0 is required because it recognizes MSVC 1950+ (Visual Studio 18/2025, `vc18` runtime).

The DLL handling code in `find_opencv.cmake` (glob for `opencv_world*.dll`, install to wheel, `IMMVISION_OPENCV_WORLD_DLL` cache variable) degrades gracefully: when OpenCV is static, no DLLs are found and the entire chain is a no-op.

For C++ app deployment (`imgui_bundle_add_app.cmake`), the `IMMVISION_OPENCV_WORLD_DLL` copy-to-output logic is similarly a no-op when no DLL exists.

7.n.vi. *Windows ARM64 local builds:*

On ARM64 Windows, the precompiled x64 `opencv_world.dll` package is skipped automatically (detected via `CMAKE_SYSTEM_PROCESSOR`). The build goes straight to `build_opencv.sh`, which compiles OpenCV from source with the correct VS generator and architecture.

SIMD is disabled globally (`-DWITH_SIMD=OFF` in `build_opencv.sh`) to avoid cross-compilation mismatches (ARM NEON headers vs x64 compiler). The performance impact is negligible for the minimalist OpenCV (image loading only).

7.n.vii. *Rebuilding the emscripten precompiled package:*

The emscripten builds use precompiled OpenCV packages downloaded from GitHub release assets. To rebuild these packages:

References:

- https://docs.opencv.org/3.4/d4/da1/tutorial_js_setup.html
- https://www.ubble.ai/how-to-make-opencv-js-work/
- https://answers.opencv.org/question/212376/how-to-decode-an-image-using-emscripten/

Steps:

1. Clone and checkout the desired OpenCV version:

```
git clone https://github.com/opencv/opencv.git
cd opencv
git checkout 4.11.0
cd ..
```

2. Manually edit `opencv/CMakeLists.txt` and add:

```
add_compile_options(-pthread)
add_link_options(-pthread)
```

For pyodide, also add:

```
add_compile_options(
  "-fwasm-exceptions"
  "-sSUPPORT_LONGJMP"
)
add_link_options(
  "-fwasm-exceptions"
  "-sSUPPORT_LONGJMP"
)
```

Note: to compile without pthread support, remove `-pthread` below and replace *twice* `-s USE_PTHREADS=1` by `-s USE_PTHREADS=0`.

3. Use the following makefile (or justfile) to compile:

```
default:
    echo "Nothing for default"

clean:
    rm -rf build opencv_emscripten_install

call_cmake:
    mkdir -p build && \
    cd build && \
    export EMSDK=~/emsdk && \
    source ~/emsdk/emsdk_env.sh && \
    export OPENCV_SRC=$(pwd)/../opencv && \
    export OPENCV_INSTALL=$(pwd)/../opencv_emscripten_install && \
    export TOOLCHAIN=$EMSDK/upstream/emscripten/cmake/Modules/Platform/
Emscripten.cmake && \
    export PYTHON_EXE=$(which python3) && \
    \
    emcmake cmake \
    -S ../opencv -B . \
    -DPYTHON_DEFAULT_EXECUTABLE=$PYTHON_EXE \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=$OPENCV_INSTALL \
    -DCMAKE_TOOLCHAIN_FILE=$TOOLCHAIN \
    \
    \"-DCMAKE_C_FLAGS='-pthread -s WASM=1 -s USE_PTHREADS=1 ' \"  \
    \
    \"-DCMAKE_CXX_FLAGS='-pthread -s WASM=1 -s USE_PTHREADS=1 ' \" \
```

```
    -DCPU_BASELINE='' -DCPU_DISPATCH='' -DENABLE_PIC=ON -DCV_TRACE=OFF -
DBUILD_SHARED_LIBS=OFF -DWITH_1394=OFF -DWITH_ADE=OFF -DWITH_VTK=OFF -
DWITH_EIGEN=OFF -DWITH_FFMPEG=OFF -DWITH_GSTREAMER=OFF -DWITH_GTK=OFF -
DWITH_GTK_2_X=OFF -DWITH_IPP=OFF -DWITH_JASPER=OFF -DWITH_JPEG=ON -
DWITH_WEBP=OFF -DWITH_OPENEXR=OFF -DWITH_OPENGL=OFF -DWITH_OPENVX=OFF -
DWITH_OPENNI=OFF -DWITH_OPENNI2=OFF -DWITH_PNG=ON -DWITH_TBB=OFF -
DWITH_TIFF=OFF -DWITH_V4L=OFF -DWITH_OPENCL=OFF -DWITH_OPENCL_SVM=OFF -
DWITH_OPENCLAMDFFT=OFF -DWITH_OPENCLAMDBLAS=OFF -DWITH_GPHOTO2=OFF -
DWITH_LAPACK=OFF -DWITH_ITT=OFF -DWITH_QUIRC=OFF -DWITH_PROTOBUF=OFF -
DBUILD_ZLIB=ON -DBUILD_opencv_apps=OFF -DBUILD_opencv_calib3d=OFF -
DBUILD_opencv_dnn=OFF -DBUILD_opencv_features2d=OFF -
DBUILD_opencv_flann=OFF -DBUILD_opencv_gapi=OFF -DBUILD_opencv_ml=OFF -
DBUILD_opencv_photo=OFF -DBUILD_opencv_imgcodecs=ON -
DBUILD_opencv_shape=OFF -DBUILD_opencv_videoio=OFF -
DBUILD_opencv_videostab=OFF -DBUILD_opencv_highgui=OFF -
DBUILD_opencv_superres=OFF -DBUILD_opencv_stitching=OFF -
DBUILD_opencv_java=OFF -DBUILD_opencv_js=OFF -DBUILD_opencv_python2=OFF
-DBUILD_opencv_python3=OFF -DBUILD_EXAMPLES=OFF -DBUILD_PACKAGE=OFF -
DBUILD_TESTS=OFF -DBUILD_PERF_TESTS=OFF -DBUILD_DOCS=OFF -
DWITH_PTHREADS_PF=OFF -DCV_ENABLE_INTRINSICS=OFF -
DBUILD_WASM_INTRIN_TESTS=OFF

build: call_cmake
    cd build && \
    make && \
    make install

tgz: build
    tar -czvf
opencv_4.11_wasmexcept_pthread_fpic_emscripten_minimalist_install.tgz
opencv_emscripten_install

all: tgz
    md5
opencv_4.11_wasmexcept_pthread_fpic_emscripten_minimalist_install.tgz
```

4. Upload the `.tgz` to a GitHub release, then update the URL and MD5 hash in `find_opencv.cmake` arrow.r `immvision_download_emscripten_precompiled_opencv_4_9_0()`.