Dear ImGui Bundle

1. Introduction

1.a. Bundled Libraries

1.a.i. *Full list of included libraries:*

Dear ImGui Bundle includes the following libraries, which are available in C++ *and* in Python:

- 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
- ImPlot : Immediate Mode Plotting
- ImPlot3D : Immediate Mode 3D Plotting
- ImGuizmo : Immediate mode 3D gizmo for scene editing
- ImGuiColorTextEdit : Colorizing text editor for ImGui
- imgui-node-editor : Node Editor built using Dear ImGui
- imgui_md : Markdown renderer for Dear ImGui using MD4C parser
- ImmVision : Immediate image debugger and insights
- NanoVG : Antialiased 2D vector drawing library on top of OpenGL
- imgui_tex_inspect : A texture inspector tool for Dear ImGui
- 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

A big thank you to their authors for their awesome work!

1.a.ii. *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:

Published Jan 21, 2026

- **Python Bindings:** Using Dear ImGui Bundle in Python is extremely easy and productive.

- **Beautifully 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:

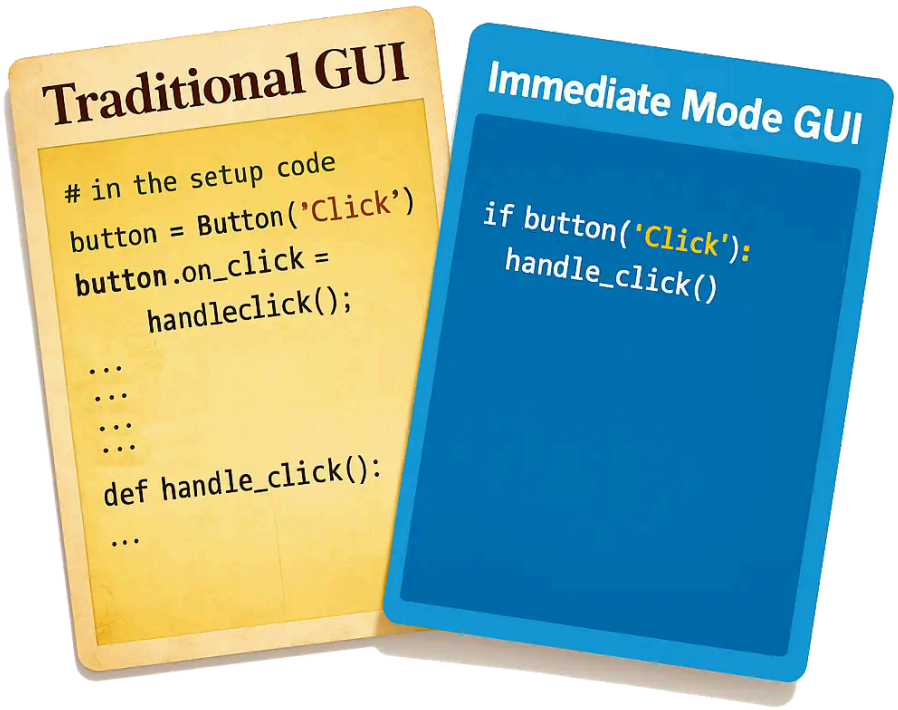
- The Immediate Mode GUI (IMGUI) paradigm is simple and powerful, letting you focus on the creative aspects of your projects.
- **Easy to use, yet very powerful:** Start your first app in 3 lines.
- **Interactive Demos and Documentation:** Quickly get started with our interactive manual and demos that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

Always up-to-date:

- **Always up-to-date:** The libraries are always very close to the latest version of Dear ImGui. This is also true for Python developers, since the bindings are automatically generated.
- **Fast:** Rendering is done via OpenGL (or any other renderer you choose), through native code.

1.b. Immediate GUI

1.b.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

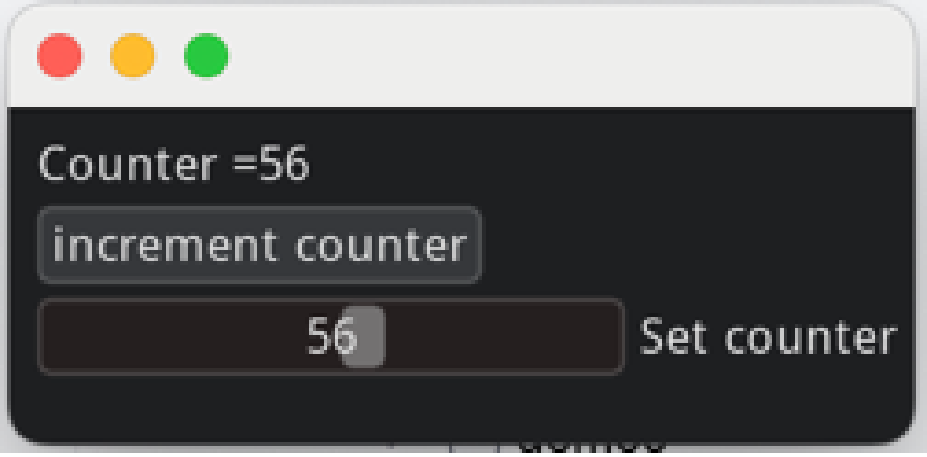
```python
from imgui_bundle import imgui, immapp

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.b.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.b.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.
- ImApp (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

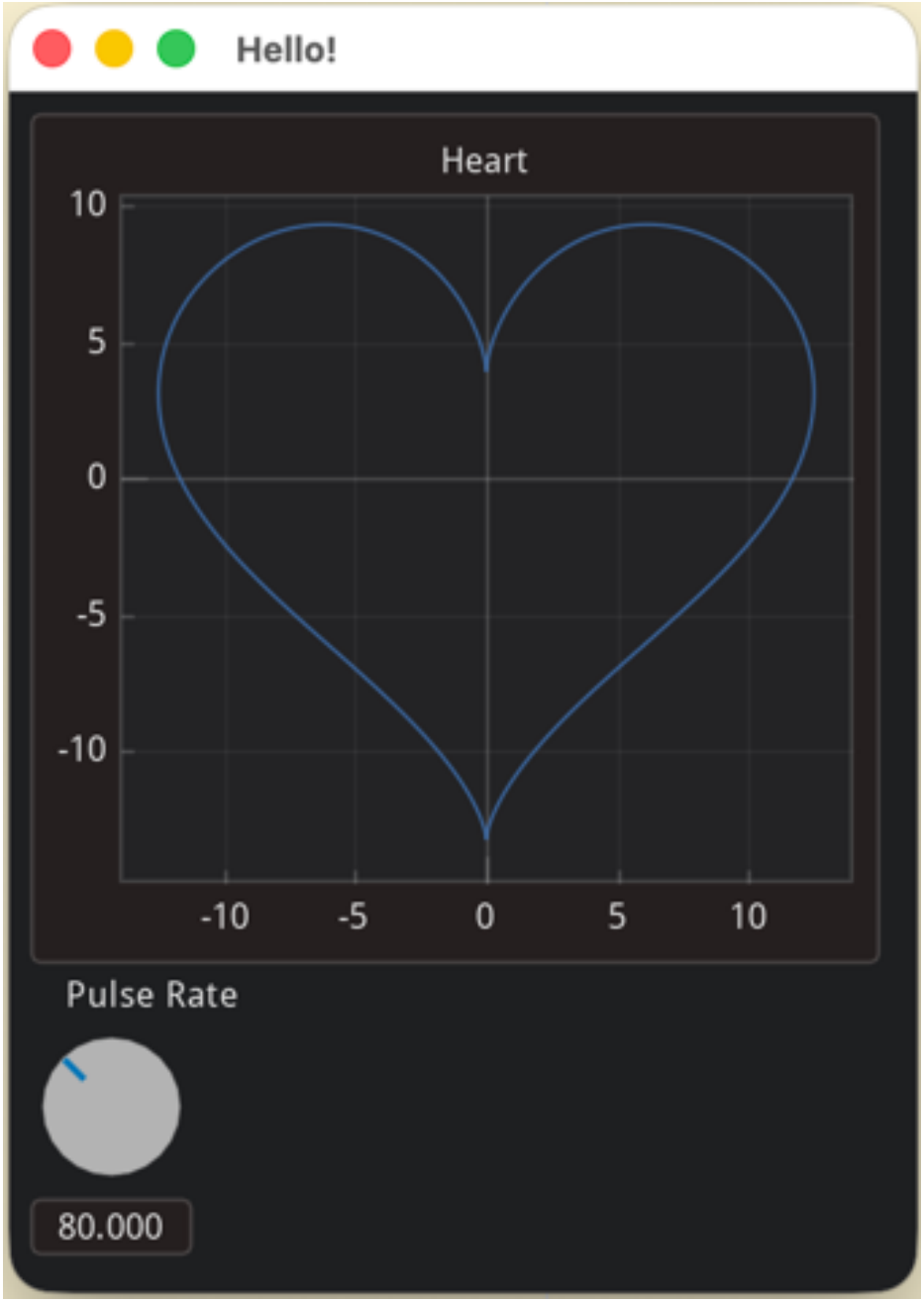
```python
import time
import numpy as np

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.b.iv. *Quickly deploy your apps on the web:*

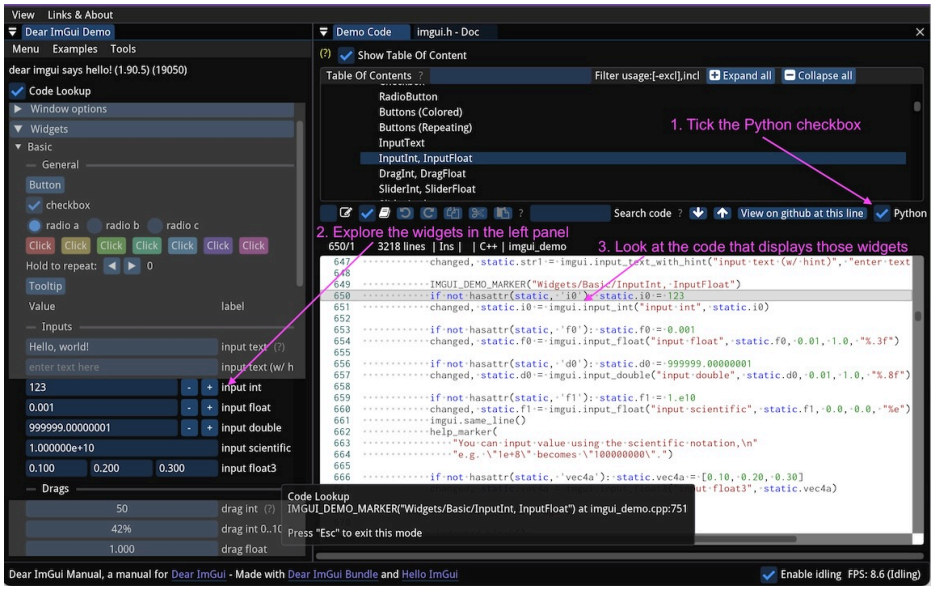
These apps can be easily deployed on the web, either in C++ via Emscripten, or in Python via Pyodide.

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

1.c. Interactive Manuals

1.c.i. *Dear ImGui Manual:*

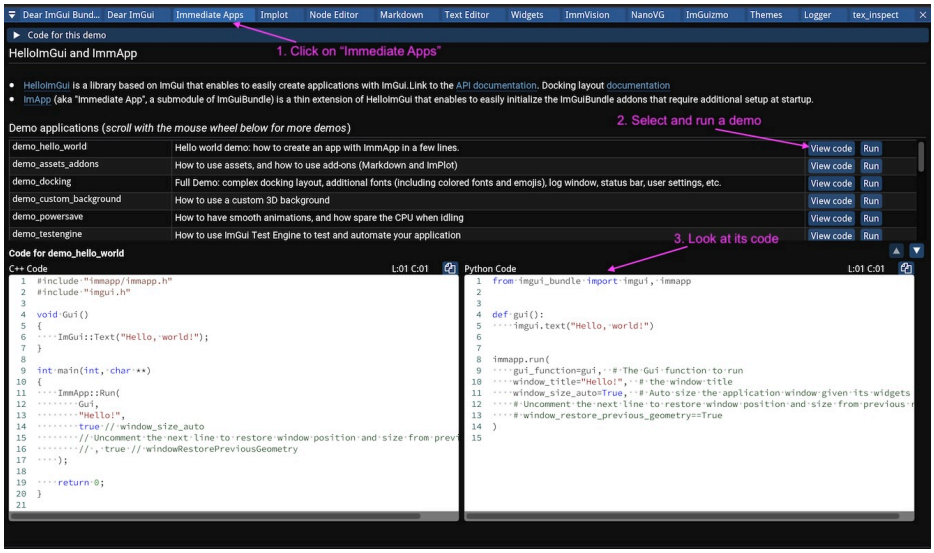
Dear ImGui Manual lets you explore all the widgets and features of Dear ImGui, with live examples and the corresponding python or C++ code. It is built using Dear ImGui Bundle.



1.c.ii. *Dear ImGui Bundle Interactive Manual:*

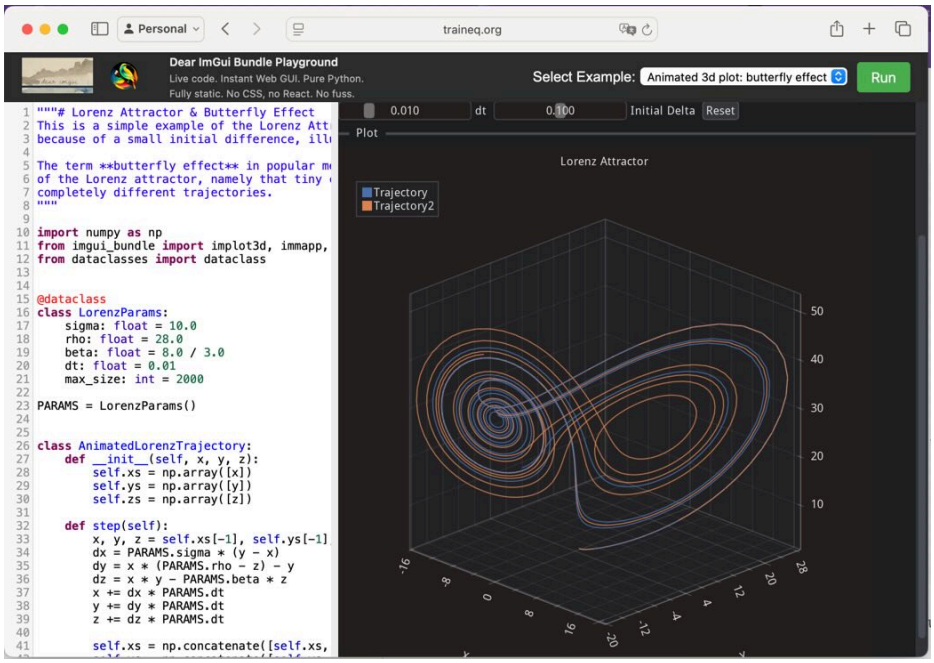
Dear ImGui Bundle interactive manual lets you explore the features of Dear ImGui Bundle in your web browser.

Pay attention to the “Demo Apps” tab, which contains many examples built with Dear ImGui Bundle. You can read the documentation, run the demos, and even view the source code (in C++ and Python) directly from the manual!



1.c.iii. *Online Python playground:*

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



1.d. Examples and Gallery

1.d.i. *Examples in the interactive manual:*

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

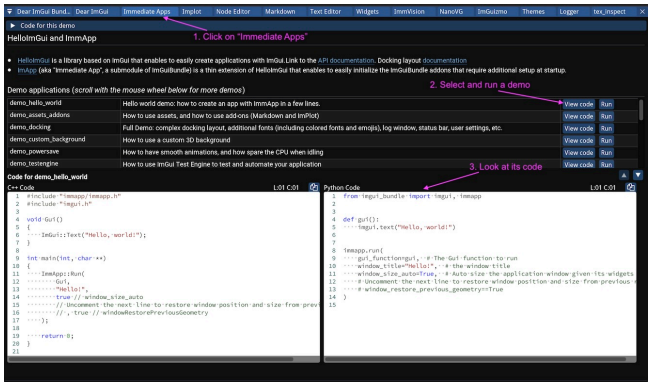


Figure 1: Inside the manual, click the “Demo Apps” tab, select a demo, run it and look at its source code.

https://traineq.org/ImGuiBundle/emscripten/bin/demo_imgui_bundle.html

Complex layouts with docking windows:

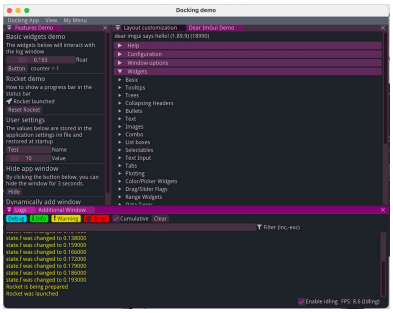


Figure 2: A complex GUI app with a docking layout, and several possible arrangements

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:

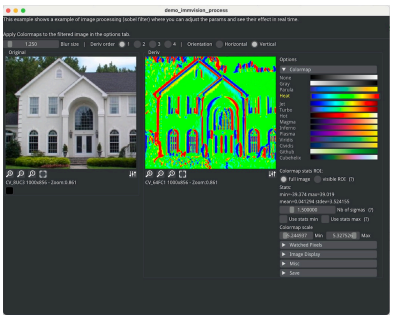


Figure 4: ImmVision in action

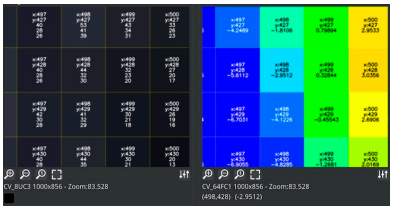


Figure 5: Zooming on the images (with the mouse wheel) to display pixel values

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.

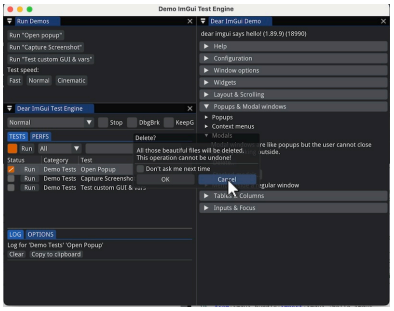
This demonstration showcases how to:

- 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.d.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

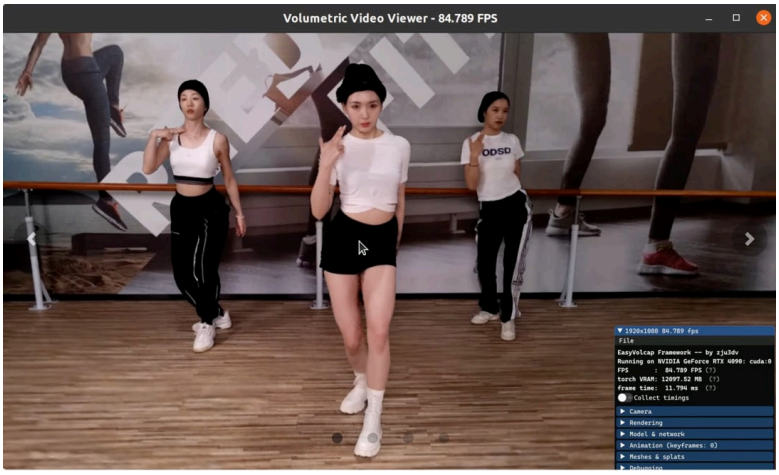


Figure 7: A volumetric video, showing an ImGui interface to control the rendering parameters.

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!

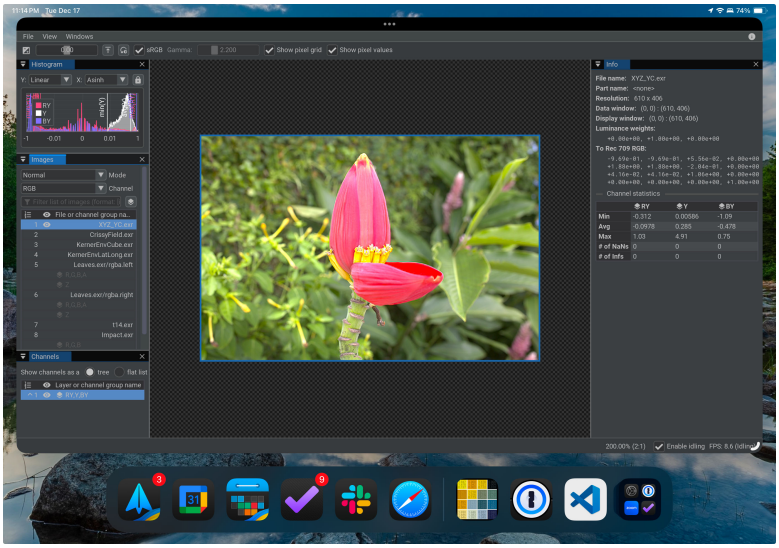


Figure 8: HDRview running on an iPad as a webapp, viewing a luminance-chroma EXR image stored using XYZ primaries with chroma subsampling.

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

1.e. Resources

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

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

- ImGui Bundle interactive manual: lots of example apps which you can run and inspect the source code
- ImGui Manual (widget reference & code): explore all the widgets and features of Dear ImGui, with live examples and the corresponding python or C++ code
- Online Pyodide playground: try imgui apps using Python, directly in your browser

1.e.ii. *Documentation websites:*

- Hello ImGui documentation. Hello ImGui provides a simple framework to quickly create applications using Dear ImGui. It is included in Dear ImGui Bundle.
- Dear ImGui Bundle documentation.
- 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 build on top of Dear ImGui Bundle.

1.e.iii. *YouTube Playlist:*

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

- Dear ImGui Bundle - YouTube Playlist

1.e.iv. *DeepWiki:*

DeepWiki is an AI based website where you can ask questions about the usage of Dear ImGui Bundle and get answers. It is trained on the full documentation and the source code of the Dear ImGui Bundle. Expect some inconsistencies, but it is still helpful.

1.e.v. *Repositories:*

- Dear ImGui official repository
- Dear ImGui Bundle repository
- Hello ImGui repository
- Litgen (bindings generator) repository
- Fiatlight repository

1.e.vi. *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

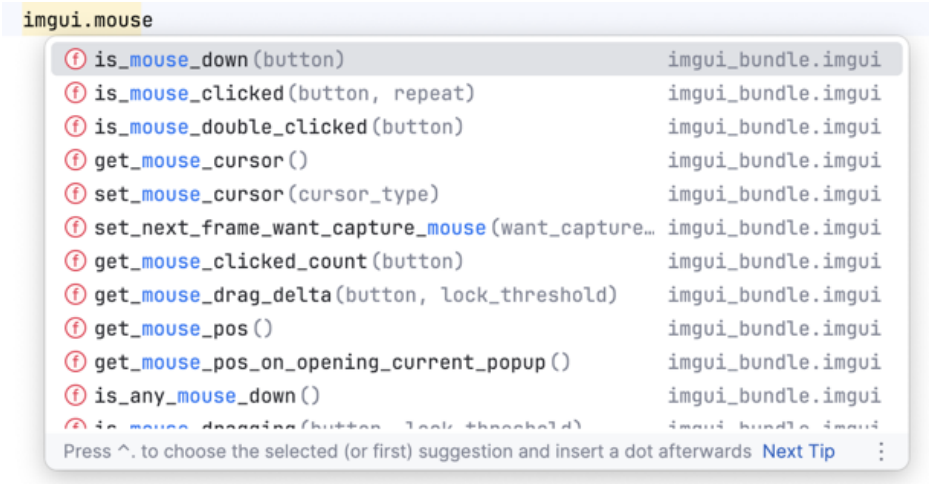
2.a. Introduction

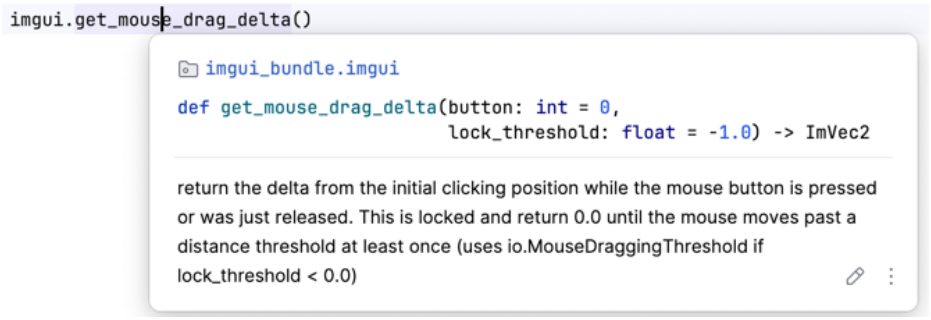
2.a.i. *Immediate GUI in Python with Dear ImGui Bundle:*

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:*

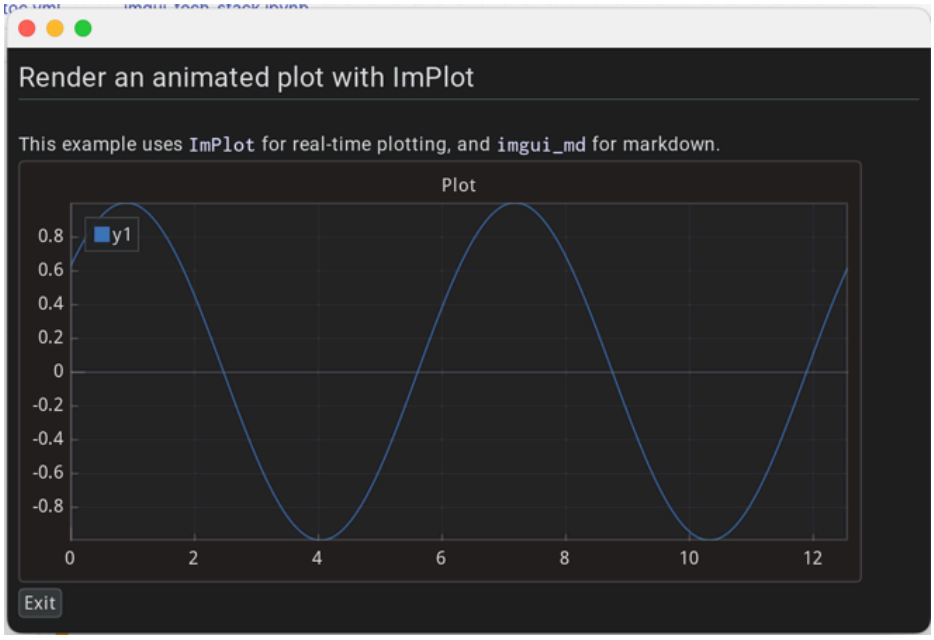
`imgui_bundle` is a Python package that unifies multiple Dear ImGui-related submodules:

- `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.

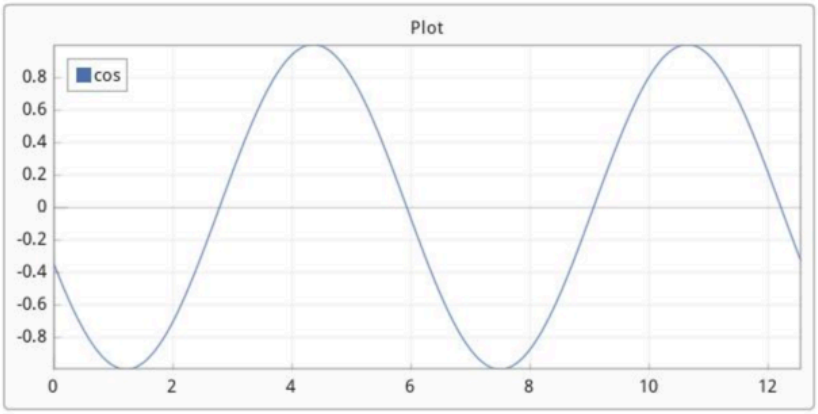
- **Standalone scripts:** Run on any PC (Windows, macOS, Linux) with minimal setup.
- **Jupyter notebooks:** The app runs in a separate window, and a screenshot is displayed in the notebook after closing (requires running Jupyter locally).
- **Web (Pyodide):** No server or installation required—just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

2.c.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.c.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.c.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.c.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.d. 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.e. *Pure Python Backends*

HelloImGui and ImmApp use glfw as a default backend. If you wish to use a different backend, it is possible to use sdl2 or pyglet, via pure python backends.

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 for more information.

2.f. Async Support

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.f.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.f.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.f.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.f.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.f.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.f.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.g. *Jupyter Notebook support*

2.g.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.g.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):

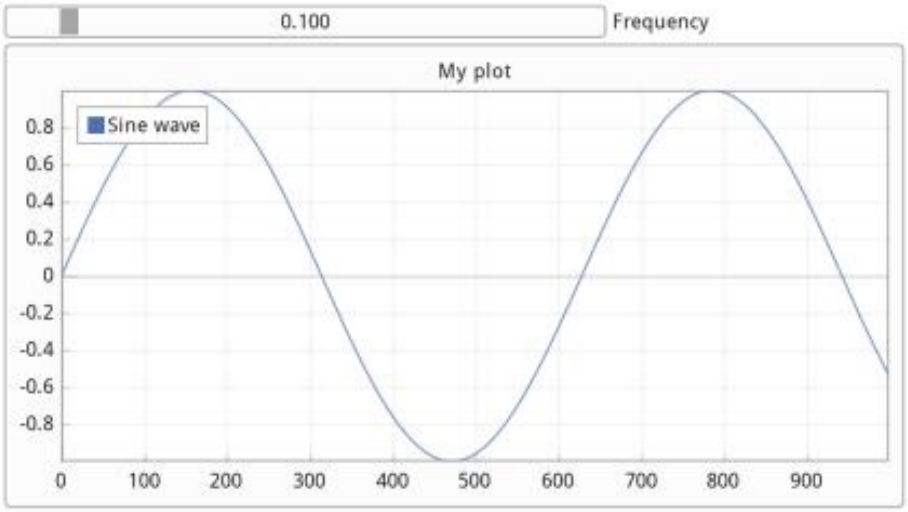
- thumbnail_ratio: (default=1.0) You can use it to change the size of the thumbnail. Passing 0.5 will create a thumbnail half the width of the window.
- thumbnail_height: (default=0) You can use it to set a fixed height for the thumbnail (in pixels). If 0, the height is computed from the app window size.

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.g.iii. *Non blocking mode:*

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:

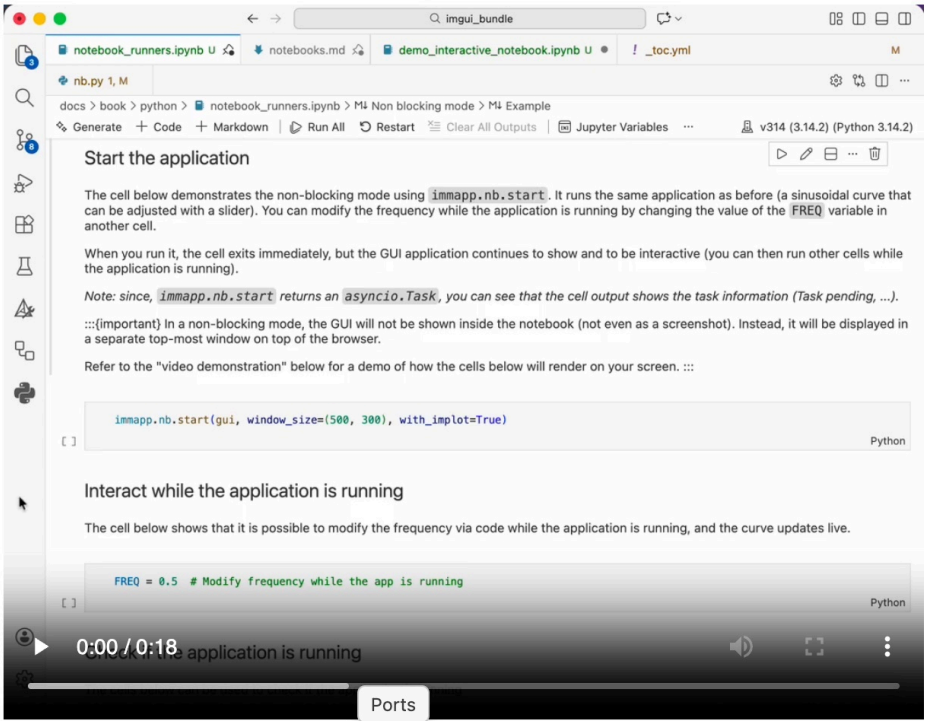


Figure 9: Demonstration of the non-blocking mode in a Jupyter notebook.

2.g.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:

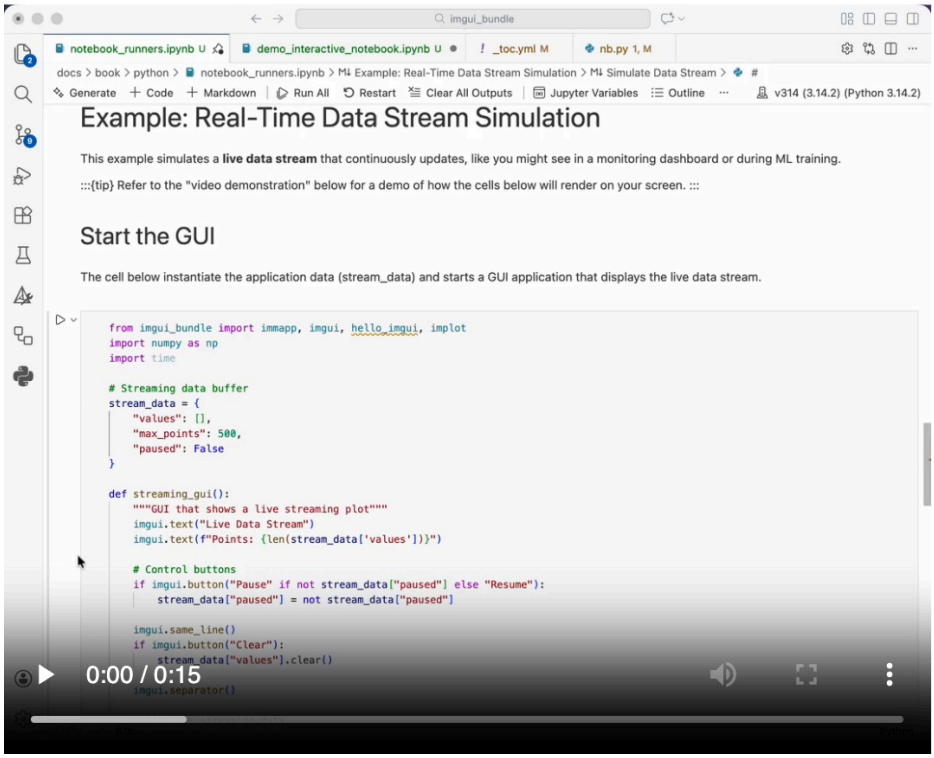


Figure 10: Demonstration of the real-time data stream simulation in a Jupyter notebook.

2.g.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.h. Deploy to the web with 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 Example

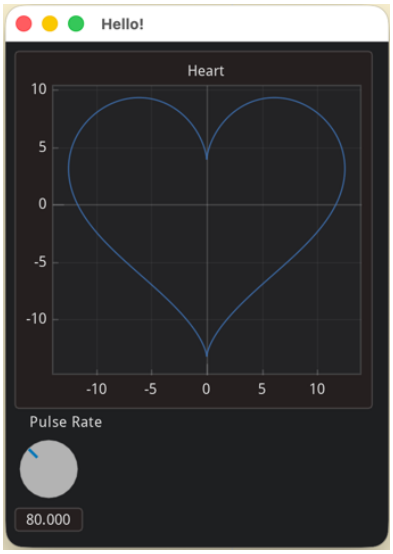
With Pyodide, web deployment is as easy as copying this HTML template. The Python code is unchanged from what you'd use for desktop.

```
<!doctype html>
<html>
<head>
    <style>
        html, body { width: 100%; height: 100%; margin: 0; }
        #canvas { display: block; width: 100%; height: 100%;}
    </style>
    <script src="https://cdn.jsdelivr.net/pyodide/v0.28.2/full/pyodide.
js"></script>
</head>
<body>
<canvas id="canvas" tabindex="0"></canvas>
<script type="text/javascript">
    // ====================== Start of Python code
============================
    // Write your python code here
    pythonCode = `
from imgui_bundle import imgui, immapp

def gui():
    imgui.text(f"hello, world")

immapp.run(gui)
`
    // ====================== End of Python code
==============================
    async function main(){
        // This enables to use right click in the canvas
        document.addEventListener('contextmenu', event =>
event.preventDefault());
        // Load Pyodide
        let pyodide = await loadPyodide();
        // Setup SDL, cf https://pyodide.org/en/stable/usage/sdl.html
        let sdl2Canvas = document.getElementById("canvas");
        pyodide.canvas.setCanvas2D(sdl2Canvas);
        pyodide._api._skip_unwind_fatal_error = true; // SDL requires to
```

```
enable an opt-in flag :
        // Load imgui_bundle
        await pyodide.loadPackage("imgui_bundle");
        // Run the Python code
        pyodide.runPython(pythonCode);
    }
    main();
</script>
</body>
</html>
```

2.h.i. *A more advanced example:*

- animated heart,
- source code



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

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

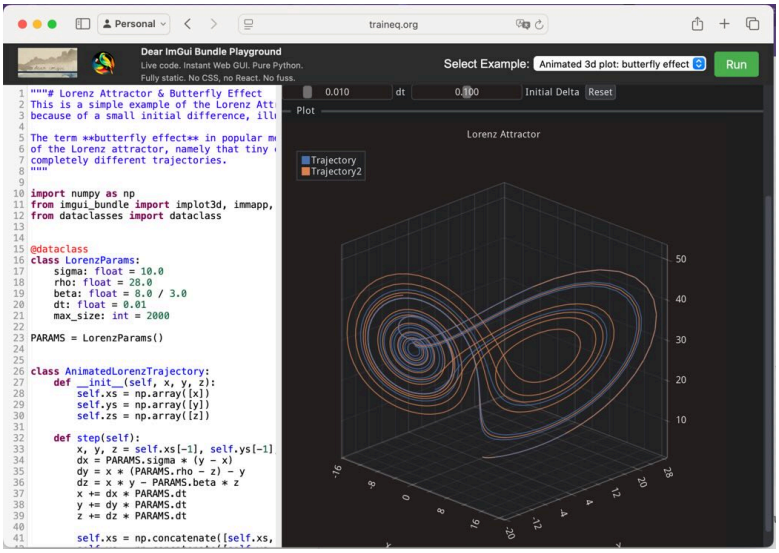


Figure 13: A browser window showing the playground: to the right an interactive demo of the butterfly effect using a 3D plot, and to the left the python code that creates it.

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 .. -DIMMVISION_FETCH_OPENCV=ON # (2)
make -j
```

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

(2) The flag -DIMMVISION_FETCH_OPENCV=ON is optional. If set, a minimal version of OpenCV will be downloaded a compiled at this stage (this might require a few minutes)

The immvision module will only be built if OpenCV can be found. Otherwise, it will be ignored, and no error will be emitted.

If you have an existing OpenCV install, set its path via:

```
cmake .. -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.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.

3.c. Multiplatform C++ applications

When developing C++ applications, Hello ImGui and Dear ImGui Bundle offer an excellent support for multiplatform applications.

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`

3.d. 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.

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).

4. Runners (Hello ImGui & ImmApp)

4.a. Intro to Runners

ImGui Bundle uses two main libraries to manage the application lifecycle: **Hello ImGui** and **ImmApp**.

4.a.i. *Hello ImGui vs ImmApp:*

- **Hello ImGui**: A "starter pack" for Dear ImGui. It handles window creation, backend initialization (SDL, GLFW, etc.), cross-platform assets, docking, and more.
- **ImmApp (Immediate App)**: A thin wrapper around Hello ImGui specifically designed for ImGui Bundle. Its main purpose is to simplify the initialization of **add-ons** (like ImPlot or Markdown) that require specific setup.

4.a.ii. *Starting an Application:*

The simplest way to start an application is to use `immapp.run()` (Python) or `ImmApp::Run()` (C++).

Python

In Python, `immapp.run` accepts a `gui_function` and several optional parameters to quickly configure the window and add-ons.

```python
from imgui_bundle import immapp, imgui

def gui():
    imgui.text("My App")

immapp.run(
    gui,
    window_title="Hello",
    window_size=(800, 600)
)
```

C++

In C++, you typically use a lambda or a function pointer for the GUI, and pass configuration via `SimpleRunnerParams`.

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

int main() {
    auto gui = []() { ImGui::Text("My App"); };
    ImmApp::Run(gui, "Hello", {800, 600});
    return 0;
}
```

> **Note**
>
> You may also call `hello_imgui.run()` (Python) or `HelloImGui::Run()` (C++), but in that case you cannot use addons, such as ImPlot; unless you initialize them manually.

4.a.iii. *Activating Add-ons with ImmApp:*

Many libraries in the bundle (like **ImPlot** or **imgui_md**) require initialization at startup (e.g., creating contexts or loading specific fonts). `ImmApp` manages this via `AddOnsParams`.

Python

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

def gui():
    imgui_md.render("# Title")
    if implot.begin_plot("My Plot"):
        # ...
        implot.end_plot()

immapp.run(
    gui,
    with_implot=True,   # Activates ImPlot context
    with_markdown=True  # Loads Markdown fonts
)
```

C++

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

int main() {
    auto gui = []() { /* ... */ };

    HelloImGui::SimpleRunnerParams runnerParams;
    runnerParams.guiFunction = gui;

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

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

4.a.iv. *Advanced: Manual Rendering:*

If you need complete control over the render loop, you can use the functions inside `hello_imgui.manual_render`, or `immapp.manual_render`, instead of the standard `run()` functions.

Python

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

# Setup
runner_params = hello_imgui.RunnerParams()
runner_params.callbacks.show_gui = lambda: imgui.text("Hello, ImGui
Bundle!")
addons = immapp.AddOnsParams()
addons.with_implot = True
immapp.manual_render.setup_from_runner_params(runner_params, addons)

# Render loop
while not hello_imgui.get_runner_params().app_shall_exit:
    hello_imgui.manual_render.render()
    # Do other work here if needed

# Cleanup
hello_imgui.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);

    // Render loop
    while (!HelloImGui::GetRunnerParams().app_shall_exit) {
        ImmApp::ManualRender::Render();
        // Do other work here if needed
    }

    // Cleanup
    ImmApp::ManualRender::TearDown();
```

```
    return 0;
}
```

This approach is useful for:

- Custom event loops
- Integration with other frameworks
- Fine-grained control over frame timing

(For python users, also see the page on async usage for more info and performance tips.)

4.b. Hello ImGui

Dear ImGui Bundle includes Hello ImGui, which is itself based on ImGui. "Hello ImGui" can be compared to a starter pack that enables to easily write cross-platform Gui apps for Windows, macOS, Linux, iOS, and emscripten.

4.b.i. *API & Usage:*

RunnerParams

Applications can be fully configured via `RunnerParams` (this incudes window size, app icon, fps settings, etc.). `hello_imgui.get_runner_params()` will return the runnerParams of the current application.

See the Application parameters doc.

API

See the "Hello ImGui" API doc.

4.b.ii. *Features:*

Multiplatform utilities

- Truly multiplatform: Linux, Windows, macOS, iOS, Android, emscripten (with 4 lines of CMake code)
- Easily embed assets on all platforms (no code required)
- Customize app settings (icon and app name for mobile platforms, etc.- no code required)
- Customize application icon on all platforms (including mobile and macOS - no code required)

Dear ImGui Tweaks

- Power Save mode: reduce FPS when idling
- High DPI support: scale UI according to DPI, whatever the platform
- Advanced layout handling: dockable windows, multiple layouts
- Window geometry utilities: autosize application window, restore app window position
- Theme tweaking: extensive list of additional themes
- Support for movable and resizable borderless windows
- Advanced font support: icons, emojis and colored fonts
- Integration with ImGui Test Engine: automate and test your apps
- Save user settings: window position, layout, opened windows, theme, user defined custom settings
- Easily add a custom 3D background to your app

Backends

- Available platform backends: SDL2, Glfw3

- Available rendering backends: OpenGL3, Metal, Vulkan, DirectX

> **Note**
>
> The usage of Hello ImGui is optional. You can also build an imgui application from scratch, in C++ or in python (see python example)

> **Tip**
>
> HelloImGui is fully configurable by POD (plain old data) structures. See their description

4.b.iii. *Advanced layout and theming with Hello ImGui::*

See the demo named “demo_docking”, which demonstrates:

- How to handle complex layouts: you can define several layouts and switch between them: each layout which will remember the user modifications and the list of opened windows
- How to use theming
- How to store you own user settings in the app ini file
- How to add a status bar and a log window
- How to reduce the FPS when idling (to reduce CPU usage)

Links:

- See demo_docking.py
- See demo_docking.cpp
- Run this demo online
- See a short video explanation about layouts on YouTube

4.c. *ImmApp - Immediate App*

ImGui Bundle includes a library named ImmApp (which stands for Immediate App). ImmApp is a thin extension of HelloImGui that enables to easily initialize the ImGuiBundle addons that require additional setup at startup

4.c.i. *API:*

- C++ API-
- Python API

4.c.ii. *How to start an application with addons:*

Some libraries included by ImGui Bundle require an initialization at startup. ImmApp makes this easy via AddOnParams.

The example program below demonstrates how to run an application which will use implot (which requires a context to be created at startup), and imgui_md (which requires additional fonts to be loaded at startup).

Python

```python
import numpy as np
# imgui_bundle is a package that provides several imgui-related
submodules
from imgui_bundle import (imgui,       # first we import ImGui
                          implot,      # ImPlot provides advanced
real-time plotting
                          imgui_md,    # imgui_md: markdown rendering
for imgui
                          hello_imgui, # hello_imgui: starter pack
for imgui apps
                          immapp,      # helper to activate addons
(like implot, markdown, etc.)
                          )

def gui():
    # Render some markdown text
    imgui_md.render_unindented("""
    # Render an animated plot with ImPlot
    This example shows how to use `ImPlot` to render an animated
plot,
    and how to use `imgui_md` to render markdown text (*this text!*).
    """)

    # Render an animated plot
    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


def main():
    # This call is specific to the ImGui Bundle interactive manual.
    from imgui_bundle.demos_python import demo_utils
    demo_utils.set_hello_imgui_demo_assets_folder()

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


if __name__ == "__main__":
    main()
```

C++

```
#include "immapp/immapp.h"
#include "imgui_md_wrapper/imgui_md_wrapper.h"
#include "implot/implot.h"
#include "demo_utils/api_demos.h"
#include <vector>
#include <cmath>


int main(int, char**)
{
    constexpr double pi = 3.14159265358979323846264338;
    std::vector<double> x, y1, y2;
    for (double _x = 0; _x < 4 * pi; _x += 0.01)
    {
        x.push_back(_x);
        y1.push_back(std::cos(_x));
        y2.push_back(std::sin(_x));
    }

    auto gui = [x,y1,y2]()
    {
        ImGuiMd::Render("# This is the plot of _cosinus_ and
*sinus*");  // Markdown
        if (ImPlot::BeginPlot("Plot"))
        {
            ImPlot::PlotLine("y1", x.data(), y1.data(), x.size());
            ImPlot::PlotLine("y2", x.data(), y2.data(), x.size());
```

```
            ImPlot::EndPlot();
        }
    };

    HelloImGui::SimpleRunnerParams runnerParams { .guiFunction =
gui, .windowSize = {600, 400} };
    ImmApp::AddOnsParams addons { .withImplot = true, .withMarkdown =
true };
    ImmApp::Run(runnerParams, addons);

    return 0;
}
```

4.d. Application Settings

ImGui applications usually store settings such as window positions, opened windows (etc.), in a file "imgui.ini". HelloImGui and ImmApp extend this functionality by storing additional settings such as application layouts, status bar settings, and user-defined custom settings.

4.d.i. *Settings location:*

By default, the settings are stored in a ini file whose named is derived from the window title (i.e. `runnerParams.appWindowParams.windowTitle`). This is convenient when developing, but not so much when deploying the app.

You can finely define where they are stored by filling `runnerParams.iniFolderType` and `runnerParams.iniFilename`.

runnerParams.iniFolderType

Choose between: `CurrentFolder`, `AppUserConfigFolder`, `AppExecutableFolder`, `HomeFolder`, `TempFolder` and `DocumentsFolder`.

> **Note**
>
> AppUserConfigFolder corresponds to `...\[Username]\AppData\Roaming` under Windows, `~/.config` under Linux, `~/Library/Application Support` under macOS or iOS

runnerParams.iniFilename

This will be the name of the ini file in which the settings will be stored. It can include a subfolder, in which case it will be created under the folder defined by runnerParams.iniFolderType.

> **Note**
>
> if left empty, the name of the ini file will be derived from appWindowParams.windowTitle.

4.d.ii. *Settings content:*

The settings file contains, standard ImGui settings (window position, size, etc.), as well as additional settings defined by HelloImGui:

- Application status: app window location, opened windows, status bar settings, etc. See members named remember_xxx in the parameters doc for a complete list.
- Settings for each application layout (see video for an example)

4.d.iii. *Store custom settings:*

You may store additional user settings in the application settings. This is provided as a convenience only, and it is not intended to store large quantities of text data. See related doc for more details.

4.e. Tips

4.e.i. *Correctly size and position the widgets:*

It is almost always a bad idea to use fixed sizes. This will lead to portability issues, especially on high-DPI screens.

Instead of using fixed pixel sizes, it is recommended to use sizes relative to the font size, aka "em" units.

> **Tip**
>
> See the definition of the em CSS Unit.

To achieve this, you should multiply your positions and sizes by `ImGui::GetFontSize()` (C++), or `imgui.get_font_size()` (Python).

In order to make this simpler, the `HelloImGui::EmToVec2` (C++) or `hello_imgui::em_to_vec2` (Python) function below can greatly reduce the friction: it transforms a size in "em" units to a size in pixels.

Example with Python:

```
from imgui_bundle import imgui, hello_imgui

def gui():
    imgui.button("A button", hello_imgui.em_to_vec2(10, 2))  # 10em x
2em button
```

Example with C++:

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

void gui() {
    ImGui::Button("A button", HelloImGui::EmToVec2(10, 2)); // 10em x
2em button
}
```

> **Note**
>
> - `EmSize(x)` functions are also available to get only one dimension in pixels. (e.g., `hello_imgui.em_size(2)` or `HelloImGui::EmSize(2)`).
> - `EmToVec2` and `EmSize` are also available in the `immapp` module in Python, and in the `ImmApp` namespace in C++.

5. Support

5.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. 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. License

Dear ImGui Bundle is licensed under the MIT License

5.b. Closing words

5.b.i. *Who is this project for:*

Dear ImGui Bundle aims to make applications prototyping fast and easy, in a multiplatform / multi-tooling context. The intent is to reduce the time between an idea and a first GUI prototype down to almost zero.

It is well adapted for

- developers and researchers who want to switch easily between and research and development environment by facilitating the port of research artifacts
- beginners and developers who want to quickly develop an application without learning a GUI framework

5.b.ii. *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 internationalization, advanced styling, etc.

Also, the library makes no guarantee of ABI stability, and its API is opened to slight adaptations and breaking changes if they are found to make the overall usage better and/or safer.

5.b.iii. *License:*

5.b.iv. *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.

5.b.v. *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!

5.b.vi. *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.

6. Developper docs

6.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.

6.b. Repository folders structure

Below is the folders structure of Dear ImGui Bundle repository:

```
./
+-- Readme.md -> bindings/imgui_bundle/Readme.md           # doc
+-- Readme_devel.md
|
+-- _example_integration/                                 # Demonstrate how to
easily use
|         +-- CMakeLists.txt                              # imgui_bundle in a C+
+ app
|         +-- assets/                                     # (this is a github
template available a
|         +-- hello_world.main.cpp                        # https://github.com/
pthom/imgui_bundle_template
|
+-- imgui_bundle_cmake/                                   #
imgui_bundle_add_app() :
|         |                                               # a cmake function you
can use
|         +-- imgui_bundle_add_app.cmake                  # to create an app in
one line
|
+-- bindings/                                             # root for the python
bindings
|         +-- imgui_bundle/
|                   +-- assets/                           # assets/ folder: you
need to
|                   |                                     # copy this folder
|                   |                                     # into your app folder
if you
|                   |                                     # intend to use
markdown
|                   |
|                   +-- demos_assets/                     # assets used by demos
|                   +-- demos_cpp/                        # lots of C++ demos
|                   +-- demos_python/                     # lots of python demos
|                   +-- imgui/                            # imgui stubs
|                   |     +-- __init__.pyi
|                   |     +-- backends.pyi
|                   |     +-- internal.pyi
|                   |     +-- py.typed
|                   +-- implot.pyi                                 # implot
stubs
|                   +-- __init__.py
|                   +-- __init__.pyi
|                   +-- hello_imgui.pyi
|                   +-- ...                                        # lots of
other libs stubs
|                   +-- ...
|                   +-- ...
|                   +-- immapp/                                    # immapp:
```

```
immediate app
|                   |        |                                  # utilities
|                   |        +-- __init__.py
|                   |        +-- __init__.pyi
|                   |        +-- icons_fontawesome.py
|                   |        +-- immapp_cpp.pyi
|                   |        +-- immapp_utils.py
|                   |        +-- py.typed
|                   +-- _imgui_bundle.cpython-38-darwin.so  #
imGui_bundle python
|                   |                                       # dynamic
library
|                   +-- glfw_utils.py
|                   +-- python_backends/                   # Backends
implemented in pure python
|                   +-- py.typed
|
|
+-- cmake/                                                  # Private
cmake utilities
|         +-- add_imgui.cmake
|         +-- ...
|
+-- external/                                               # Root of all
bound libraries
|         +-- CMakeLists.txt
|         +-- imgui/                                        # ImGui root
|         |         +-- bindings/                           # ImGui
bindings
|         |         +-- imgui/                              # ImGui
submodule
|         +-- ImGuizmo/
|         |         +-- bindings/                           # ImGuizmo
bindings
|         |         +-- ImGuizmo/                          # ImGuizmo
submodule
|         |         +-- ImGuizmoPure/                      # Manual
wrappers to help
|         |                                                 # bindings
generation
|         |
|         +-- ... lots of other bound libraries/          # Lots of
other bound libraries
|         |         +-- {lib_name}/
|         |         +-- bindings/
|         |
|         +-- _doc/
|         |
|         +-- bindings_generation/                        # Script to
generate bindings
|         |         |                                     # and to
facilitate external
```

```
|         |         +-- __init__.py                        # libraries
update
|         |         +-- all_external_libraries.py
|         |         +-- autogenerate_all.py
|         |         +-- ...
|         |
|         +-- SDL/SDL/                                     # Linked
library (without
|         |                                                # python
bindings)
|         +-- fplus/fplus/                                 # Library
without bindings
|         +-- glfw/glfw                                    # Library
without bindings
|
+-- lg_cmake_utils/                                        # Cmake utils
for bindings
|         |                                                # generation
|         +-- lg_cmake_utils.cmake
|         +-- ...
|
+-- pybind_native_debug/
|         +-- CMakeLists.txt
|         +-- Readme.md
|         +-- pybind_native_debug.cpp
|         +-- pybind_native_debug.py
|
+-- src/
|         +-- imgui_bundle/                              # main cpp
library: almost empty,
                                                         # but linked
to all external libraries
```

6.c. *Automated bindings: introduction*

The bindings are generated automatically thanks to a sophisticated generator, which is based on srcML.

The generator in provided by in litgen an automatic python bindings generator, developed by the same author as Dear ImGui Bundle.

6.c.i. *Installing the generator:*

See the installation instructions (do a local installation).

6.c.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.

6.c.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.

6.c.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
├── ...
```

6.d. Update existing bindings

6.d.i. *Introduction:*

Run `generate_LIBNAME.py`:

The process for updating bindings for a given library is straightforward:

1. Update the library submodule in external/LIBNAME/LIBNAME
2. Run the generation script in external/LIBNAME/generate_LIBNAME.py
3. Compile and test python bindings (carefully study that nothing was broken)
4. Commit and push

For example with ImCoolBar, in order to update the bindings for ImCoolBar, one needs to run:

```
python external/ImCoolBar/bindings/generate_imcoolbar.py
```

Submodules maintenance:

external/bindings_generation contains some scripts for the submodules maintenance.

See this extract of external/bindings_generation/all_external_libraries.py, which shows that imgui and imgui_test_engine are using forks.

These forks include small modifications added for compatibility with imgui_bundle (most modifications are small changes to accommodate with python bindings).

```
def lib_imgui() -> ExternalLibrary:
    return ExternalLibrary(
        name="imgui",
        official_git_url="https://github.com/ocornut/imgui.git",
        official_branch="docking",
        fork_git_url="https://github.com/pthom/imgui.git"
    )


def lib_imgui_test_engine() -> ExternalLibrary:
    return ExternalLibrary(
        name="imgui_test_engine",
        official_git_url="https://github.com/ocornut/imgui_test_engine.
git",
        official_branch="main",
        fork_git_url="https://github.com/pthom/imgui_test_engine.git",
    )
```

When using forked libraries, the git remote name for the fork is “fork”, and the remote name for the official repository is “official”.

Reattach all submodules to their upstream branch

By default, all submodules, are in mode “detached head”. We need to attach them to the correct remote/branch.

We can use the utilities from external/bindings_generation:

For example, external/bindings_generation/sandbox.py contains this:

```
from bindings_generation import all_external_libraries

all_external_libraries.reattach_all_submodules()
```

It will reattach all submodules to the correct remote/branch.

6.d.ii. *Example: update imgui & bindings:*

> **Tip**
>
> This video demonstrates from starts to finish the process of updating imgui and its bindings (17 minutes).

Update imgui and imgui_test_engine:

First, add a tag to our forks

Since we will be updating our imgui and imgui_test_engine forks via a rebase, we should push a tag, so that old versions remain accessible on GitHub.

In this example, the current version of imgui_bundle is v1.0.0-beta1. So we push a "bundle_1.0.0-beta1" tag to the forks.

```
cd external/imgui/imgui
git tag "bundle_1.0.0-beta1"
git push fork --tags
cd -

cd external/imgui_test_engine/imgui_test_engine
git tag "bundle_1.0.0-beta1"
git push fork --tags
cd -
```

Then rebase our forks on the official branch changes

```
cd external/imgui/imgui
git rebase official/docking
cd -

cd external/imgui_test_engine/imgui_test_engine
git rebase official/main
cd -
```

Run generate_imgui.py:

Run generate_imgui

We will run external/imgui/bindings/generate_imgui.py.

It will generate the python bindings for imgui, imgui_internal and imgui_test_engine.

See main() function of generate_imgui.py:

```
def main():
    autogenerate_imgui()
    autogenerate_imgui_internal()
    autogenerate_imgui_test_engine()
```

Examine the changes Look at the changes, and check if they look ok

Compile & Test:

Correct possible compilation errors due to breaking changes in imgui's API

Test in C++

Run `demo_imgui_bundle`

(demo_imgui_bundle is a global demonstration program, that uses most of the feature of all libraries)

Test in Python

Run `demo_imgui_bundle.py`

Update forked submodules::

if some forked submodules required to be changed:

- tag them, push the tag
- rebase the fork branch on the official branch
- push the changes

6.e. *Adding a new library to the bindings*

This example is based on the addition of ImCoolBar, which was added in Oct 2023.

6.e.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