Hello ImGui

Contents

- Get started
- Application parameters
- Diagram
- RunnerParams
- Runner callbacks
- Application window params
- ImGui window params
- Fps Idling
- Dpi Aware Params
- Docking
- Backend
- API
- Run Application
- Run Application while handling the rendering loop
- Place widgets in a DPI-aware way
- Load fonts
- Applications assets
- Utility functions
- Switch between several layouts
- Ini settings
- Customize Hello ImGui Menus
- Additional Widgets
- Handling screens with high DPI
- Build instructions

Hello ImGui is a library designed to make multiplatform app development as simple as writing a "Hello World" program, built on top of Dear ImGui.

Its flexibility makes it suited for complex applications, as well as for simple prototypes; allowing you to focus on the creative aspects of your projects.

Features

Multiplatform utilities

- Seamlessly multiplatform: supports Linux, Windows, macOS, iOS, Android, and Emscripten with minimal setup (1 line of CMake code)
- Asset embedding across all platforms (no code required)
- Effortless app customization, including app icons and names for mobile platforms (no code required)
- Universal application icon customization, extending to mobile and macOS (no code required)

Dear ImGui Tweaks

- Power Save mode: optimize performance by reducing FPS when idle
- High DPI support: automatically scales UI to match screen DPI on any platform.
- Enhanced layout handling: dockable windows and multiple layouts for complex UIs
- 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

Demos & real world apps

Motto

The minimum code to start developing a GUI app should be... minimal. Here is a multiplatform Hello World in 7 lines.

6 lines of C++

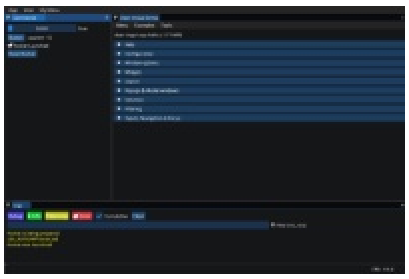
```
#include "hello_imgui/hello_imgui.h"
int main(int , char *[]) {
    HelloImGui::Run(
        []{ ImGui::Text("Hello, world!"); }, // Gui code
        "Hello!", true);                     // Window title + Window size auto
}
```

1 line of CMake

```
hello_imgui_add_app(hello_world hello_world.cpp)
```

Advanced layout

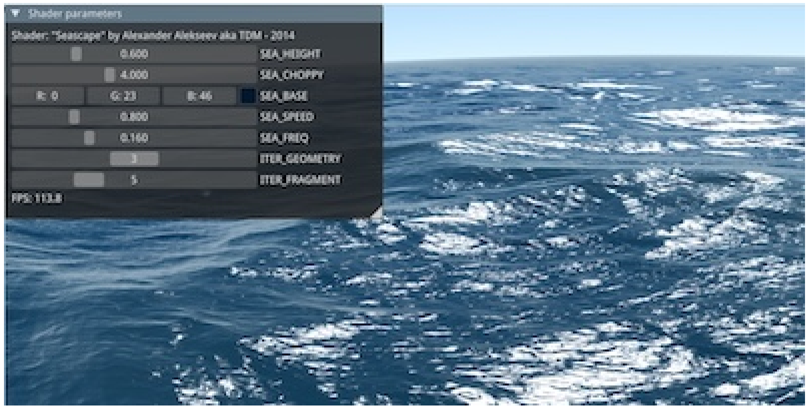
The docking demo show how to handle complex layouts, use themes, store user settings, reduce FPS and CPU usage when idling, load fonts and icons, and much more



Online demo - Source - Video tutorial: how to handle multiple complex layouts

Custom 3D background

How to use a custom 3D background in your app

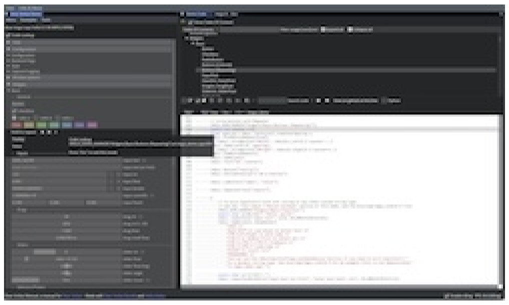


```
Shader: "Seascape" by Alexander Alekseev aka TDM - 2014
0.600                SEA_HEIGHT
4.000                SEA_CHOPPY
R: 0   G: 23   B: 46 SEA_BASE
0.800                SEA_SPEED
0.160                SEA_FREQ
3                    ITER_GEOMETRY
5                    ITER_FRAGMENT
FPS: 113.8
```

Online demo - Source

ImGui Manual

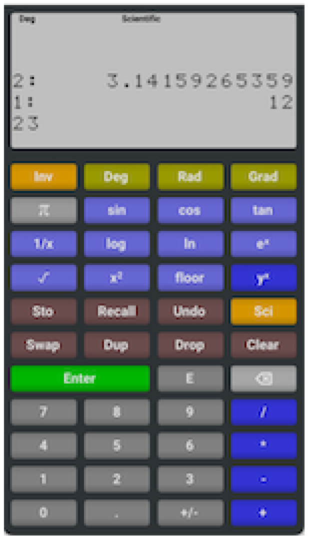
ImGui Manual is an interactive Manual for Dear ImGui, built with Hello ImGui.



Online manual - Source

RPN Calculator

RPN Calculator is a simple RPN calculator, built, built to demonstrate how easy a multiplatform app can be built with Hello ImGui.



Online demo - Source - Video tutorial

Starter template

The starter template will get you started in 5 minutes, and shows how to embed assets, customize the app icon (etc.), on all platforms.

Dear ImGui Bundle

Dear ImGui Bundle, a full-fledged library, built on top of Hello ImGui, provides lots of additional widgets (imgui, implot, imgui-node-editor, ImFileDialog, ImGuiColorTextEdit, imgui_md), as well as complete python bindings.

online interactive demo - Source for C++ demos - Source for Python demos

Theme tweaking

How to use the theme tweaking utilities provided by Hello ImGui

Online video tutorial

About

Credits

Hello ImGui is based on Dear ImGui (MIT License), by Omar Cornut. Portions use ImGui Test Engine, which is governed by a specific license

Portions of this software use the FreeType Project, plutosvg (MIT License), GLFW (zlib/libpng license), SDL (zlib license).

License

Hello ImGui is licensed under the MIT License, see LICENSE

Contribute

Quality contributions are welcome!

Support the project

Hello ImGui 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. Thank you!

Get started

Starter template

Tip

The starter template will get you started in 5 minutes, and shows how to embed assets, customize the app icon (etc.), on all platforms. It is recommended to use it as a starting point for your own project.

With this starter template, you do not need to clone HelloImGui, as it (optionally) can be downloaded and built automatically during CMake configure time.

See this extract from the CMakelists.txt file of the template:

```
# Build hello_imgui
# =================
# 1/  Option 1: if you added hello_imgui as a subfolder, you can add it to your proj
if(IS_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/external/hello_imgui)
    add_subdirectory(external/hello_imgui)
endif()

# 2/  Option 2: simply fetch hello_imgui during the build
if (NOT TARGET hello_imgui)
    message(STATUS "Fetching hello_imgui")
    include(FetchContent)
    FetchContent_Declare(hello_imgui GIT_REPOSITORY https://github.com/pthom/hello_i
    FetchContent_MakeAvailable(hello_imgui)
endif()

# 3/  Option 3: via vcpkg
# i/ You can install hello_imgui via vcpkg with:
#     vcpkg install "hello-imgui[opengl3-binding,glfw-binding]"
# ii/ Then you can use it inside CMake with:
#     find_package(hello-imgui CONFIG REQUIRED)


# Build your app
# ==============
# hello_imgui_add_app is a helper function, similar to cmake's "add_executable"
```

CMake utility: hello_imgui_add_app

```
# hello_imgui_add_app is a helper function, similar to cmake's "add_executable"
#
# Usage:
#     hello_imgui_add_app(app_name file1.cpp file2.cpp ...)
# Or:
#     hello_imgui_add_app(app_name file1.cpp file2.cpp ... ASSETS_LOCATION "path/to,
# (By default, ASSETS_LOCATION is "assets", which means that the assets will be sea
# relative to the location of the CMakeLists.txt file)
#
# Features:
#     * It will automatically link the target to the required libraries (hello_imgu
#     * It will embed the assets (for desktop, mobile, and emscripten apps)
#     * It will perform additional customization (app icon and name on mobile platf
#     * On desktop platforms, it will perform a "portable install" (i.e. assets and
#
# If you want to control the install of your app, you can set HELLOIMGUI_ADD_APP_WI
# See an example in https://github.com/pthom/hello_imgui_template/blob/main/CMakeLi
#
```

Assets and app customization

```
Anything in the assets/ folder located beside the app's CMakeLists will be bundled
together with the app (for macOS, iOS, Android, Emscripten).
The files in assets/app_settings/ will be used to generate the app icon,
and the app settings.

```
assets/
├── world.jpg   # A custom asset. Any file or folder here will be de
│   # with the app.
├── fonts/
│   ├── DroidSans.ttf   # Default fonts used by HelloImGui
│   └── fontawesome-webfont.ttf # (if not found, the default ImGui font will be use
│
├── 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 And
│   ├── apple/
│   │   ├── Info.plist   # macOS and iOS app settings
│   │   │   # (or Info.ios.plist + Info.macos.plist)
│   │   └── Resources/
│   │   └── ios/   # iOS specific settings: storyboard
│   │   └── LaunchScreen.storyboard
│   │
│   ├── android/   # Android app settings: any file placed here will b
│   │ ├── AndroidManifest.xml # (Optional manifest, HelloImGui will generate one
│   │ └── 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
```
```

Immediate GUI mode

Hello ImGui is based on Dear ImGui, which is a library for immediate mode GUI programming.

With an Immediate mode GUI you can for example display a button and handle the click event in one line of code:

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

int main(int , char *[])
{
    auto guiFunction = []() {
        ImGui::Text("Hello, ");                    // Display a simple label
        HelloImGui::ImageFromAsset("world.png");   // Display a static image
        if (ImGui::Button("Bye!"))                 // Display a button
            // and immediately handle its action if it is clicked!
            HelloImGui::GetRunnerParams()->appShallExit = true;
     };
    HelloImGui::Run(guiFunction, "Hello, globe", true);
    return 0;
}
```

ImGui Manual is a great resource to learn interactively about all the widgets provided by Dear ImGui.

Application parameters

HelloImGui::Run() will run an application with a single call.

Three signatures are provided:

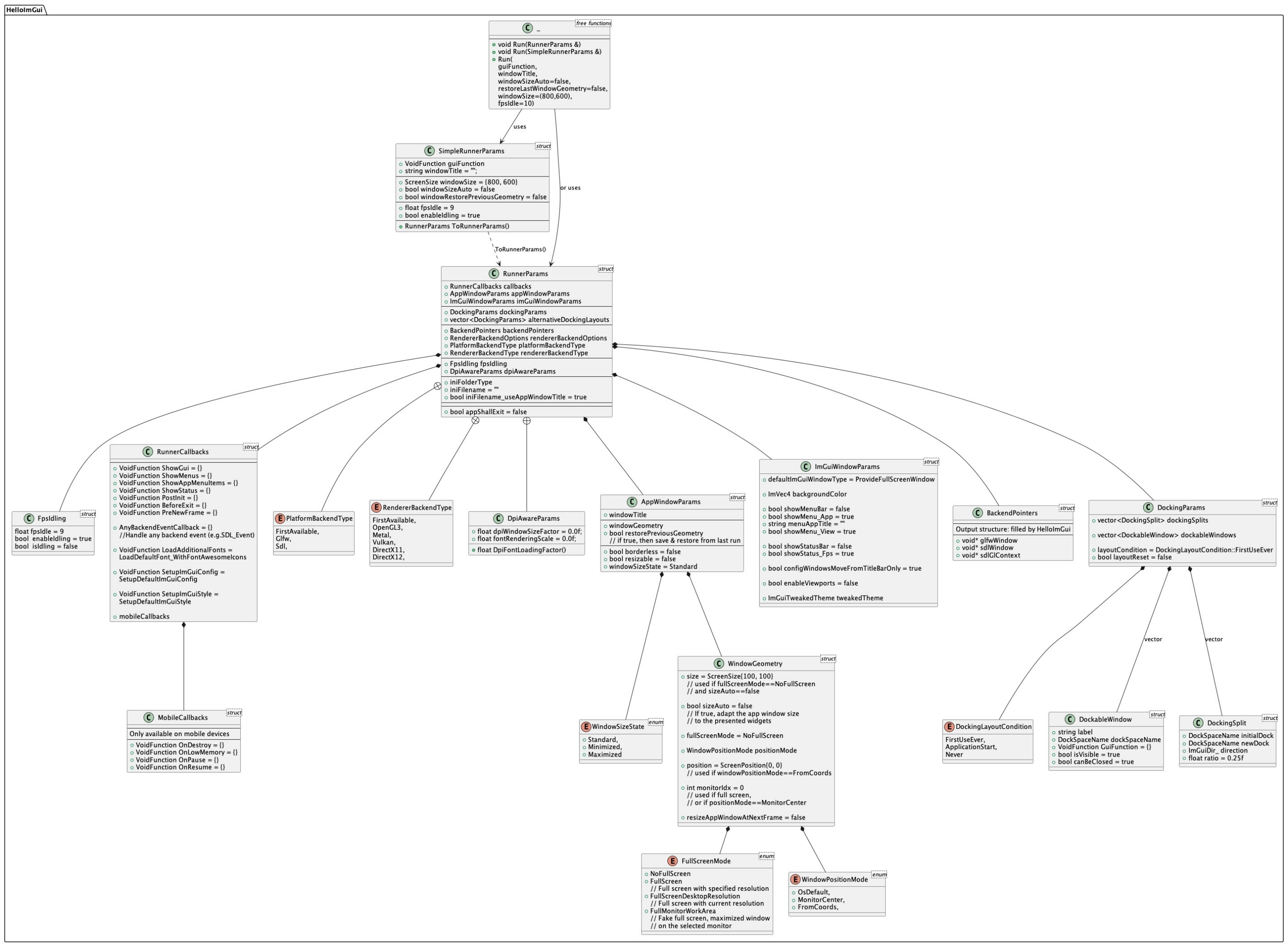
- `HelloImGui::Run(RunnerParams &)`: full signature, the most customizable version. Runs an application whose params and Gui are provided by runnerParams.
- `HelloImGui::Run(const SimpleRunnerParams&)`: Runs an application, using simpler params.
- `HelloImGui::Run(guiFunction, windowTitle, windowSize, windowSizeAuto=false, restoreLastWindowGeometry=false, fpsIdle=10)` Runs an application, by providing the Gui function, the window title, etc.

Although the API is extremely simple, it is highly customizable, and you can set many options by filling the elements in the `RunnerParams` struct, or in the simpler `SimpleRunnerParams`.

HelloImGui::GetRunnerParams() will return the runnerParams of the current application.

Diagram

The diagram below summarize all the possible settings and callbacks (which are explained in detail later in this document).


HelloImGui
_
free functions
void Run(RunnerParams &)
void Run(SimpleRunnerParams &)
Run(
guiFunction,
windowTitle,
windowSizeAuto=false,
restoreLastWindowGeometry=false,
windowSize=(800,600),
fpsIdle=10)
uses
or uses
SimpleRunnerParams
struct
VoidFunction guiFunction
string windowTitle = "";
ScreenSize windowSize = {800, 600}
bool windowSizeAuto = false
bool windowRestorePreviousGeometry = false
float fpsIdle = 9
bool enableIdling = true
RunnerParams ToRunnerParams()
ToRunnerParams()
RunnerParams
struct
RunnerCallbacks callbacks
AppWindowParams appWindowParams
ImGuiWindowParams imGuiWindowParams
DockingParams dockingParams
vector<DockingParams> alternativeDockingLayouts
BackendPointers backendPointers
RendererBackendOptions rendererBackendOptions
PlatformBackendType platformBackendType
RendererBackendType rendererBackendType
FpsIdling fpsIdling
DpiAwareParams dpiAwareParams
iniFolderType
iniFilename = ""
bool iniFilename_useAppWindowTitle = true
bool appShallExit = false
FpsIdling
struct
float fpsIdle = 9
bool enableIdling = true
bool isIdling = false
RunnerCallbacks
struct
VoidFunction ShowGui = {}
VoidFunction ShowMenus = {}
VoidFunction ShowAppMenuItems = {}
VoidFunction ShowStatus = {}
VoidFunction PostInit = {}
VoidFunction BeforeExit = {}
VoidFunction PreNewFrame = {}
AnyBackendEventCallback = {}
//Handle any backend event (e.g.SDL_Event)
VoidFunction LoadAdditionalFonts =
LoadDefaultFont_WithFontAwesomeIcons
VoidFunction SetupImGuiConfig =
SetupDefaultImGuiConfig
VoidFunction SetupImGuiStyle =
SetupDefaultImGuiStyle
mobileCallbacks
MobileCallbacks
struct
Only available on mobile devices
VoidFunction OnDestroy = {}
VoidFunction OnLowMemory = {}
VoidFunction OnPause = {}
VoidFunction OnResume = {}
PlatformBackendType
FirstAvailable,
Glfw,
Sdl,
RendererBackendType
FirstAvailable,
OpenGL3,
Metal,
Vulkan,
DirectX11,
DirectX12,
DpiAwareParams
float dpiWindowSizeFactor = 0.0f;
float fontRenderingScale = 0.0f;
float DpiFontLoadingFactor()
AppWindowParams
struct
windowTitle
windowGeometry
bool restorePreviousGeometry
// if true, then save & restore from last run
bool borderless = false
bool resizable = false
windowSizeState = Standard
ImGuiWindowParams
struct
defaultImGuiWindowType = ProvideFullScreenWindow
ImVec4 backgroundColor
bool showMenuBar = false
bool showMenu_App = true
string menuAppTitle = ""
bool showMenu_View = true
bool showStatusBar = false
bool showStatus_Fps = true
bool configWindowsMoveFromTitleBarOnly = true
bool enableViewports = false
ImGuiTweakedTheme tweakedTheme
BackendPointers
struct
Output structure: filled by HelloImGui
void* glfwWindow
void* sdlWindow
void* sdlGlContext
DockingParams
struct
vector<DockingSplit> dockingSplits
vector<DockableWindow> dockableWindows
layoutCondition = DockingLayoutCondition::FirstUseEver
bool layoutReset = false
vector
vector
DockingLayoutCondition
FirstUseEver,
ApplicationStart,
Never
DockableWindow
struct
string label
DockSpaceName dockSpaceName
VoidFunction GuiFunction = {}
bool isVisible = true
bool canBeClosed = true
DockingSplit
struct
DockSpaceName initialDock
DockSpaceName newDock
ImGuiDir_ direction
float ratio = 0.25f
WindowSizeState
enum
Standard,
Minimized,
Maximized
WindowGeometry
struct
size = ScreenSize{100, 100}
// used if fullScreenMode==NoFullScreen
// and sizeAuto==false
bool sizeAuto = false
// If true, adapt the app window size
// to the presented widgets
fullScreenMode = NoFullScreen
WindowPositionMode positionMode
position = ScreenPosition{0, 0}
// used if windowPositionMode==FromCoords
int monitorIdx = 0
// used if full screen,
// or if positionMode==MonitorCenter
resizeAppWindowAtNextFrame = false
FullScreenMode
enum
NoFullScreen
FullScreen
// Full screen with specified resolution
FullScreenDesktopResolution
// Full screen with current resolution
FullMonitorWorkArea
// Fake full screen, maximized window
// on the selected monitor
WindowPositionMode
enum
OsDefault,
MonitorCenter,
FromCoords,

RunnerParams

See runner_params.h.

Simple runner params

```cpp
// SimpleRunnerParams is a struct that contains simpler params adapted for simple us
//For example, this is sufficient to run an application:
//    ```cpp
//    void MyGui() {
//        ImGui::Text("Hello, world");
//        if (ImGui::Button("Exit"))
//            HelloImGui::GetRunnerParams()->appShallExit = true;
//    }
//
//    int main(){
//        auto params = HelloImGui::SimpleRunnerParams {
//            .guiFunction = MyGui, .windowSizeAuto = true, .windowTitle = "Example'
//        };
//        HelloImGui::Run(params);
//    }
//    ```
struct SimpleRunnerParams
{
    // `guiFunction`: _VoidFunction_.
    //  Function that renders the Gui.
    VoidFunction guiFunction = EmptyVoidFunction();
    // `windowTitle`: _string, default=""_.
    //  Title of the application window
    std::string windowTitle = "";

    // `windowSizeAuto`: _bool, default=false_.
    //  If true, the size of the window will be computed from its widgets.
    bool windowSizeAuto = false;

    // `windowRestorePreviousGeometry`: _bool, default=true_.
    //  If true, restore the size and position of the window between runs.
    bool windowRestorePreviousGeometry = false;

    // `windowSize`: _ScreenSize, default={800, 600}_.
    //  Size of the window
    // The size will be handled as if it was specified for a 96PPI screen
    // (i.e. a given size will correspond to the same physical size on different sc
    ScreenSize windowSize = DefaultWindowSize;

    // `fpsIdle`: _float, default=9_.
    //  FPS of the application when idle (set to 0 for full speed).
    float fpsIdle = 9.f;

    // `enableIdling`: _bool, default=true_.
    //  Disable idling at startup by setting this to false
    //  When running, use:
    //      HelloImGui::GetRunnerParams()->fpsIdling.enableIdling = false;
    bool  enableIdling = true;
```

```
    RunnerParams ToRunnerParams() const;
};
```

Full params

```
// RunnerParams contains the settings and callbacks needed to run an application.
//
struct RunnerParams
{
    // --------------- Callbacks and Window params -------------------

    // `callbacks`: _see runner_callbacks.h_
    // callbacks.ShowGui() will render the gui, ShowMenus() will show the menus, et
    RunnerCallbacks callbacks;

    // `appWindowParams`: _see app_window_params.h_
    // application Window Params (position, size, title)
    AppWindowParams appWindowParams;

    // `imGuiWindowParams`: _see imgui_window_params.h_
    // imgui window params (use docking, showMenuBar, ProvideFullScreenWindow, etc.
    ImGuiWindowParams imGuiWindowParams;


    // --------------- Docking -------------------

    // `dockingParams`: _see docking_params.h_
    // dockable windows content and layout
    DockingParams dockingParams;

    // `alternativeDockingLayouts`: _vector<DockingParams>, default=empty_
    // List of possible additional layout for the applications. Only used in advanc
    // cases when several layouts are available.
    std::vector<DockingParams> alternativeDockingLayouts;

    // `rememberSelectedAlternativeLayout`: _bool, default=true_
    // Shall the application remember the last selected layout. Only used in advanc
    // cases when several layouts are available.
    bool rememberSelectedAlternativeLayout = true;


    // --------------- Backends -------------------

    // `backendPointers`: _see backend_pointers.h_
    // A struct that contains optional pointers to the backend implementations.
    // These pointers will be filled when the application starts
    BackendPointers backendPointers;

    // `rendererBackendOptions`: _see renderer_backend_options.h_
    // Options for the renderer backend
    RendererBackendOptions rendererBackendOptions;

    // `platformBackendType`: _enum PlatformBackendType, default=PlatformBackendTyp
    // Select the wanted platform backend type between `Sdl`, `Glfw`.
    // if `FirstAvailable`, Glfw will be preferred over Sdl when both are available
    // Only useful when multiple backend are compiled and available.
    PlatformBackendType platformBackendType = PlatformBackendType::FirstAvailable;

    // `renderingBackendType`: _enum RenderingBackendType, default=RenderingBackend
    // Select the wanted rendering backend type between `OpenGL3`, `Metal`, `Vulkan
    // if `FirstAvailable`, it will be selected in the order of preference mentione
    // Only useful when multiple rendering backend are compiled and available.
    RendererBackendType rendererBackendType = RendererBackendType::FirstAvailable;


    // --------------- Settings -------------------

    // `iniFolderType`: _IniFolderType, default = IniFolderType::CurrentFolder_
    // Sets the folder where imgui will save its params.
    // (possible values are:
    //     CurrentFolder, AppUserConfigFolder, DocumentsFolder,
    //     HomeFolder, TempFolder, AppExecutableFolder)
    // AppUserConfigFolder is
    //     [Home]/AppData/Roaming under Windows,
    //     ~/.config under Linux,
    //     ~/Library/Application Support under macOS
    IniFolderType iniFolderType = IniFolderType::CurrentFolder;
    // `iniFilename`: _string, default = ""_
    // Sets the ini filename under which imgui will save its params.
    // Its path is relative to the path given by iniFolderType, and can include
    // a subfolder (which will be created if needed).
    // If iniFilename empty, then it will be derived from
    // appWindowParams.windowTitle
    // (if both are empty, the ini filename will be imgui.ini).
    std::string iniFilename = "";  // relative to iniFolderType
    // `iniFilename_useAppWindowTitle`: _bool, default = true_.
    // Shall the iniFilename be derived from appWindowParams.windowTitle (if not em
    bool iniFilename_useAppWindowTitle = true;


    // --------------- Exit -------------------

    // * `appShallExit`: _bool, default=false_.
    // During execution, set this to true to exit the app.
    // _Note: 'appShallExit' has no effect on Mobile Devices (iOS, Android)
    // and under emscripten, since these apps shall not exit._
    bool appShallExit = false;

    // --------------- Idling -------------------

    // `fpsIdling`: _FpsIdling_. Idling parameters
    // (set fpsIdling.enableIdling to false to disable Idling)
    FpsIdling fpsIdling;

    // --------------- DPI Handling -----------
    // Hello ImGui will try its best to automatically handle DPI scaling for you.
    // If it fails, look at DpiAwareParams (and the corresponding Ini file settings
    DpiAwareParams dpiAwareParams;

    // --------------- Misc -------------------

    // `useImGuiTestEngine`: _bool, default=false_.
    // Set this to true if you intend to use Dear ImGui Test & Automation Engine
    //     ( https://github.com/ocornut/imgui_test_engine )
    // HelloImGui must be compiled with the option -DHELLOIMGUI_WITH_TEST_ENGINE=ON
    // See demo in src/hello_imgui_demos/hello_imgui_demo_test_engine.
    // License:
    // imgui_test_engine is subject to a specific license:
    //     https://github.com/ocornut/imgui_test_engine/blob/main/imgui_test_engine/
    // (TL;DR: free for individuals, educational, open-source and small businesses u
    //  Paid for larger businesses.)
    bool useImGuiTestEngine = false;

    // `emscripten_fps`: _int, default = 0_.
    // Set the application refresh rate
    // (only used on emscripten: 0 stands for "let the app or the browser decide")
    int emscripten_fps = 0;

    #ifdef HELLOIMGUI_WITH_REMOTE_DISPLAY
    RemoteParams remoteParams; // Parameters for Remote display (experimental, unsu
    #endif
};
```

Backend selection

```
// You can select the platform backend type (SDL, GLFW) and the rendering backend ty
// via RunnerParams.platformBackendType and RunnerParams.renderingBackendType.

// Platform backend type (SDL, GLFW)
// They are listed in the order of preference when FirstAvailable is selected.
enum class PlatformBackendType
{
    FirstAvailable,
    Glfw,
    Sdl,
    Null
};

// Rendering backend type (OpenGL3, Metal, Vulkan, DirectX11, DirectX12)
// They are listed in the order of preference when FirstAvailable is selected.
enum class RendererBackendType
{
    FirstAvailable,
    OpenGL3,
    Metal,
    Vulkan,
    DirectX11,
    DirectX12,
    Null
};
```

Runner callbacks

See runner_callbacks.h.

Callbacks types

```
// VoidFunctionPointer can hold any void(void) function.
using VoidFunction = std::function<void(void)>;
inline VoidFunction EmptyVoidFunction() { return {}; }


// SequenceFunctions: returns a function that will call f1 and f2 in sequence
VoidFunction SequenceFunctions(const VoidFunction& f1, const VoidFunction& f2);


// AnyEventCallback can hold any bool(void *) function.
using AnyEventCallback = std::function<bool(void * backendEvent)>;
inline AnyEventCallback EmptyEventCallback() {return {}; }
```

RunnerCallbacks

```
// RunnerCallbacks is a struct that contains the callbacks
// that are called by the application
//
struct RunnerCallbacks
{
    // --------------- GUI Callbacks -------------------

    // `ShowGui`: Fill it with a function that will add your widgets.
    // (ShowGui will be called at each frame, before rendering the Dockable windows
    VoidFunction ShowGui = EmptyVoidFunction();

    // `ShowMenus`: Fill it with a function that will add ImGui menus by calling:
    //       ImGui::BeginMenu(...) / ImGui::MenuItem(...) / ImGui::EndMenu()
    //   Notes:
    //   * you do not need to call ImGui::BeginMenuBar and ImGui::EndMenuBar
    //   * Some default menus can be provided:
    //     see ImGuiWindowParams options:
    //         _showMenuBar, showMenu_App_QuitAbout, showMenu_View_
    VoidFunction ShowMenus = EmptyVoidFunction();

    // `ShowAppMenuItems`: A function that will render items that will be placed
    // in the App menu. They will be placed before the "Quit" MenuItem,
    // which is added automatically by HelloImGui.
    //  This will be displayed only if ImGuiWindowParams.showMenu_App is true
    VoidFunction ShowAppMenuItems = EmptyVoidFunction();

    // `ShowStatus`: A function that will add items to the status bar.
    //  Use small items (ImGui::Text for example), since the height of the status i
    //  Also, remember to call ImGui::SameLine() between items.
    VoidFunction ShowStatus = EmptyVoidFunction();

    // `EdgesToolbars`:
    // A dict that contains toolbars that can be placed on the edges of the App win
    std::map<EdgeToolbarType, EdgeToolbar> edgesToolbars;

    // `AddEdgeToolbar`: Add a toolbar that can be placed on the edges of the App w
    void AddEdgeToolbar(EdgeToolbarType edgeToolbarType,
                        VoidFunction guiFunction,
                        const EdgeToolbarOptions& options = EdgeToolbarOptions());


    // --------------- Startup sequence callbacks -------------------

    // `PostInit_AddPlatformBackendCallbacks`:
    //  You can here add a function that will be called once after OpenGL and ImGui
    //  but before the platform backend callbacks are initialized.
    //  If you, want to add your own glfw callbacks, you should use this function t
    //  (and then ImGui will call your callbacks followed by its own callbacks)
    VoidFunction PostInit_AddPlatformBackendCallbacks = EmptyVoidFunction();


    // `PostInit`: You can here add a function that will be called once after every
    //  is inited (ImGui, Platform and Renderer Backend)
    VoidFunction PostInit = EmptyVoidFunction();

    // `EnqueuePostInit`: Add a function that will be called once after OpenGL
    //  and ImGui are inited, but before the backend callback are initialized.
    //  (this will modify the `PostInit` callback by appending the new callback (us
    void EnqueuePostInit(const VoidFunction& callback);

    // `LoadAdditionalFonts`: default=_LoadDefaultFont_WithFontAwesome*.
    //  A function that is called in order to load fonts.
    // `LoadAdditionalFonts` will be called once, then *set to nullptr*.
    // If you want to load additional fonts, during the app execution, you can
    // set LoadAdditionalFonts to a function that will load the additional fonts.
    VoidFunction LoadAdditionalFonts = (VoidFunction)ImGuiDefaultSettings::LoadDefau
    // If LoadAdditionalFonts==LoadDefaultFont_WithFontAwesomeIcons, this parameter
    // which icon font will be loaded by default.
    DefaultIconFont defaultIconFont = DefaultIconFont::FontAwesome4;

    // `SetupImGuiConfig`: default=_ImGuiDefaultSettings::SetupDefaultImGuiConfig*.
    //  If needed, change ImGui config via SetupImGuiConfig
    //  (enable docking, gamepad, etc)
    VoidFunction SetupImGuiConfig = (VoidFunction)ImGuiDefaultSettings::SetupDefaul

    // `SetupImGuiStyle`: default=_ImGuiDefaultSettings::SetupDefaultImGuiConfig*.
    //  If needed, set your own style by providing your own SetupImGuiStyle callbac
    VoidFunction SetupImGuiStyle = (VoidFunction)ImGuiDefaultSettings::SetupDefault

    // `RegisterTests`: A function that is called once ImGuiTestEngine is ready
    // to be filled with tests and automations definitions.
    VoidFunction RegisterTests = EmptyVoidFunction();
    // `registerTestsCalled`: will be set to true when RegisterTests was called
    // (you can set this to false if you want to RegisterTests to be called again
    //  during the app execution)
    bool         registerTestsCalled = false;


    // --------------- Exit sequence callbacks -------------------

    // `BeforeExit`: You can here add a function that will be called once before ex
    //  (when OpenGL and ImGui are still inited)
    VoidFunction BeforeExit = EmptyVoidFunction();

    // `EnqueueBeforeExit`: Add a function that will be called once before exiting
    //  (when OpenGL and ImGui are still inited)
    // (this will modify the `BeforeExit` callback by appending the new callback (u
    void EnqueueBeforeExit(const VoidFunction& callback);

    // `BeforeExit_PostCleanup`: You can here add a function that will be called on
    // before exiting (after OpenGL and ImGui have been stopped)
    VoidFunction BeforeExit_PostCleanup = EmptyVoidFunction();
    // --------------- Callbacks in the render loop -------------------

    // `PreNewFrame`: You can here add a function that will be called at each frame
    //  and before the call to ImGui::NewFrame().
    //  It is a good place to add new dockable windows.
    VoidFunction PreNewFrame = EmptyVoidFunction();

    // `BeforeImGuiRender`: You can here add a function that will be called at each
    //  after the user Gui code, and just before the call to
    //  ImGui::Render() (which will also call ImGui::EndFrame()).
    VoidFunction BeforeImGuiRender = EmptyVoidFunction();

    // `AfterSwap`: You can here add a function that will be called at each frame,
    //  after the Gui was rendered and swapped to the screen.
    VoidFunction AfterSwap = EmptyVoidFunction();

    // `CustomBackground`:
    //  By default, the background is cleared using ImGuiWindowParams.backgroundCol
    //  If set, this function gives you full control over the background that is dr
    //  behind the Gui. An example use case is if you have a 3D application
    //  like a mesh editor, or game, and just want the Gui to be drawn
    //  on top of that content.
    VoidFunction CustomBackground = EmptyVoidFunction();

    // `PostRenderDockableWindows`: Fill it with a function that will be called
    // after the dockable windows are rendered.
    VoidFunction PostRenderDockableWindows = EmptyVoidFunction();

    // `AnyBackendEventCallback`:
    //  Callbacks for events from a specific backend. _Only implemented for SDL.
    //  where the event will be of type 'SDL_Event *'_
    //  This callback should return true if the event was handled
    //  and shall not be processed further.
    //  Note: in the case of GLFW, you should use register them in `PostInit`
    AnyEventCallback AnyBackendEventCallback = EmptyEventCallback();


    // --------------- Mobile callbacks -------------------
#ifdef HELLOIMGUI_MOBILEDEVICE
    // `mobileCallbacks`: Callbacks that are called by the application
    //  when running under "Android, iOS and WinRT".
    // Notes:
    //  * 'mobileCallbacks' is present only if the target device
    //    is a mobile device (iOS, Android).
    //    Use `#ifdef HELLOIMGUI_MOBILEDEVICE` to detect this.
    //  * These events are currently handled only with SDL backend.
    MobileCallbacks mobileCallbacks;
#endif
};
```

Edge Toolbars Callbacks

More details on `RunnerParams.edgesToolbars` (a dictionary of `EdgeToolbar`, per edge type)

```
struct RunnerCallbacks
{
    ...
    // EdgesToolbars: A map that contains the definition of toolbars
    // that can be placed on the edges of the App window
    std::map<EdgeToolbarType, EdgeToolbar> edgesToolbars;
    void AddEdgeToolbar(EdgeToolbarType edgeToolbarType,
                        VoidFunction callback,
                        const EdgeToolbarOptions& options = EdgeToolbarOptions());
    ...
};
```

Where:

```cpp
// EdgeToolbarType: location of an Edge Toolbar
enum class EdgeToolbarType
{
    Top,
    Bottom,
    Left,
    Right
};

struct EdgeToolbarOptions
{
    // height or width the top toolbar, in em units
    // (i.e. multiples of the default font size, to be Dpi aware)
    float sizeEm = 2.5f;

    // Padding inside the window, in em units
    ImVec2 WindowPaddingEm = ImVec2(0.3f, 0.3f);

    // Window background color, only used if WindowBg.w > 0
    ImVec4 WindowBg = ImVec4(0.f, 0.f, 0.f, 0.f);
};


// EdgeToolbar :a toolbar that can be placed on the edges of the App window
// It will be placed in a non-dockable window
struct EdgeToolbar
{
    VoidFunction ShowToolbar = EmptyVoidFunction();
    EdgeToolbarOptions options;
};

std::vector<EdgeToolbarType> AllEdgeToolbarTypes();
std::string EdgeToolbarTypeName(EdgeToolbarType e);
```

MobileCallbacks

```
// MobileCallbacks is a struct that contains callbacks that are called by the applic
// when running under "Android, iOS and WinRT".
// These events are specific to mobile and embedded devices that have different
// requirements from your usual desktop application.
// These events must be handled quickly, since often the OS needs an immediate respo
// and will terminate your process shortly after sending the event
// if you do not handle them appropriately.
// On mobile devices, it is not possible to "Quit" an application,
// it can only be put on Pause.
struct MobileCallbacks
{
    //`OnDestroy`: The application is being terminated by the OS.
    VoidFunction OnDestroy = EmptyVoidFunction();

    //`OnLowMemory`: _VoidFunction, default=empty_.
    // When the application is low on memory, free memory if possible.
    VoidFunction OnLowMemory = EmptyVoidFunction();

    //`OnPause`: The application is about to enter the background.
    VoidFunction OnPause = EmptyVoidFunction();

    //`OnResume`: The application came to foreground and is now interactive.
    // Note: 'OnPause' and 'OnResume' are called twice consecutively under iOS
    // (before and after entering background or foreground).
    VoidFunction OnResume = EmptyVoidFunction();
};
```

Application window params

See app_window_params.h.

AppWindowParams

```
//
// AppWindowParams is a struct that defines the application window display params.
//See https://raw.githubusercontent.com/pthom/hello_imgui/master/src/hello_imgui/doc
// for details.
struct AppWindowParams
{
    // --------------- Standard params ------------------

    // `windowTitle`: _string, default=""_. Title of the application window
    std::string windowTitle;

    // `windowGeometry`: _WindowGeometry_
    //  Enables to precisely set the window geometry (position, monitor, size,
    //  full screen, fake full screen, etc.)
    //   _Note: on a mobile device, the application will always be full screen._
    WindowGeometry windowGeometry;

    // `restorePreviousGeometry`: _bool, default=false_.
    // If true, then save & restore windowGeometry from last run (the geometry
    // will be written in imgui_app_window.ini)
    bool restorePreviousGeometry = false;

    // `resizable`: _bool, default = false_. Should the window be resizable.
    // This is taken into account at creation.
    bool resizable = true;
    // `hidden`: _bool, default = false_. Should the window be hidden.
    // This is taken into account dynamically (you can show/hide the window with th
    // Full screen windows cannot be hidden.
    bool hidden = false;


    // --------------- Borderless window params ------------------

    // `borderless`: _bool, default = false_. Should the window have borders.
    // This is taken into account at creation.
    bool   borderless = false;
    // `borderlessMovable`: if the window is borderless, should it be movable.
    //   If so, a drag zone is displayed at the top of the window when the mouse is
    bool   borderlessMovable = true;
    // `borderlessResizable`: if the window is borderless, should it be resizable.
    //  If so, a drag zone is displayed at the bottom-right of the window
    //  when the mouse is over it.
    bool   borderlessResizable = true;
    // `borderlessClosable`: if the window is borderless, should it have a close bu
    //  If so, a close button is displayed at the top-right of the window
    //  when the mouse is over it.
    bool   borderlessClosable = true;
    // `borderlessHighlightColor`:
```

```
    //   Color of the highlight displayed on resize/move zones.
    //   If borderlessHighlightColor.w==0, then the highlightColor will be automati
    //   set to ImGui::GetColorU32(ImGuiCol_TitleBgActive, 0.6f)
    ImVec4 borderlessHighlightColor = ImVec4(0.2f, 0.4f, 1.f, 0.3f);


    // --------------- iOS Notch ------------------

    // `edgeInsets`: _EdgeInsets_. iOS only, out values filled by HelloImGui.
    // If there is a notch on the iPhone, you should not display inside these insets
    // HelloImGui handles this automatically, if handleEdgeInsets is true and
    // if runnerParams.imGuiWindowParams.defaultImGuiWindowType is not NoDefaultWin
    // (warning, these values are updated only after a few frames,
    //  they are typically 0 for the first 4 frames)
    EdgeInsets edgeInsets;
    // `handleEdgeInsets`: _bool, default = true_. iOS only.
    // If true, HelloImGui will handle the edgeInsets on iOS.
    bool       handleEdgeInsets = true;


    // --------------- Emscripten ------------------
    // `emscriptenKeyboardElement`: _EmscriptenKeyboardElement, default=Default_. HT
    // (For Emscripten only)
    // Choose between: Window, Document, Screen, Canvas, Default.
    // If Default:
    // - the default SDL behavior is used, which is to capture the keyboard events f
    //   if no previous hint was set in the javascript code.
    // - under Pyodide, the default behavior is to capture the keyboard events for t
    EmscriptenKeyboardElement emscriptenKeyboardElement = EmscriptenKeyboardElement


    // ----------------- repaint the window during resize -----------------
    // Very advanced and reserved for advanced C++ users.
    // If you set this to true, the window will be repainted during resize.
    // Do read https://github.com/pthom/hello_imgui/issues/112 for info about the po
    // (This API is not stable, as the name suggests, and this is not supported)
    bool repaintDuringResize_GotchaReentrantRepaint = false;
};
```

WindowGeometry

```
//
// WindowGeometry is a struct that defines the window geometry.
struct WindowGeometry
{
    // --------------- Window Size ------------------

    // Size of the application window
    // used if fullScreenMode==NoFullScreen and sizeAuto==false. Default=(800, 600)
    // The size will be handled as if it was specified for a 96PPI screen
    // (i.e. a given size will correspond to the same physical size on different sc
    ScreenSize size = DefaultWindowSize;

    // If sizeAuto=true, adapt the app window size to the presented widgets.
    // After the first frame was displayed, HelloImGui will measure its size, and tl
    // application window will be resized.
    // As a consequence, the application window may change between the 1st and 2nd
    // If true, adapt the app window size to the presented widgets. This is done at
    bool sizeAuto = false;

    // `windowSizeState`: _WindowSizeState, default=Standard_
    //  You can choose between several window size states:
    //      Standard,
    //      Minimized,
    //      Maximized
    WindowSizeState windowSizeState = WindowSizeState::Standard;

    // `windowSizeMeasureMode`: _WindowSizeMeasureMode_, default=RelativeTo96Ppi
    // Define how the window size is specified:
    //      * RelativeTo96Ppi enables to give a screen size whose physical result
    //      (in millimeters) is independent of the screen density.
    //         For example, a window size expressed as 800x600 will correspond to a
    //            - 800x600 (in screen coords) if the monitor dpi is 96
    //            - 1600x120 (in screen coords) if the monitor dpi is 192
    //          (this works with Glfw. With SDL, it only works under windows)
    //      * ScreenCoords: measure window size in screen coords
    //        (Note: screen coordinates might differ from real pixels on high dpi s
    WindowSizeMeasureMode windowSizeMeasureMode = WindowSizeMeasureMode::RelativeTo


    // --------------- Position ------------------

    // `positionMode`: you can choose between several window position modes:
    //      OsDefault,
    //      MonitorCenter,
    //      FromCoords,
    WindowPositionMode positionMode = WindowPositionMode::OsDefault;

    // `position`: used if windowPositionMode==FromCoords, default=(40, 40)
```

```cpp
    ScreenPosition position = DefaultScreenPosition;

    // `monitorIdx`: index of the monitor to use, default=0
    //  used if positionMode==MonitorCenter or if fullScreenMode!=NoFullScreen
    int monitorIdx = 0;


    // --------------- Full screen ------------------

    // `fullScreenMode`: you can choose between several full screen modes:
    //      NoFullScreen,
    //      FullScreen,                  // Full screen with specified resolution
    //      FullScreenDesktopResolution, // Full screen with current desktop mode &
    //      FullMonitorWorkArea          // Fake full screen (maximized window) on 
    FullScreenMode fullScreenMode = FullScreenMode::NoFullScreen;


    // --------------- Auto Resize ------------------

    // `resizeAppWindowAtNextFrame`: _bool_, default=false;
    //  If you set this to flag to true at any point during the execution, the appl
    //  window will then try to resize based on its content on the next displayed f
    //  and this flag will subsequently be set to false.
    //  Example:
    //   ```cpp
    //   // Will resize the app window at next displayed frame
    //   HelloImGui::GetRunnerParams()->appWindowParams.windowGeometry.resizeAppWin
    //   ```
    //  Note: this flag is intended to be used during execution, not at startup
    //  (use sizeAuto at startup).
    bool resizeAppWindowAtNextFrame = false;
};
```

ImGui window params

See imgui_window_params.h.

ImGuiWindowParams

```
// `ImGuiWindowParams` is a struct that defines the ImGui inner windows params
// These settings affect the imgui inner windows inside the application window.
// In order to change the application window settings, change the `AppWindowsParams`
struct ImGuiWindowParams
{
    // ------------ Main Options  -------------------------------------------------------

    // defaultImGuiWindowType: (enum DefaultImGuiWindowType)
    // Choose between:
    //    - ProvideFullScreenWindow (default)
    //      a full window is provided in the background
    //      You can still add windows on top of it, since the Z-order
    //      of this background window is always behind
    //    - ProvideFullScreenDockSpace:
    //      a full screen dockspace is provided in the background
    //      (use this if you intend to use docking)
    //    - NoDefaultWindow:
    //      no default window is provided
    DefaultImGuiWindowType defaultImGuiWindowType =
        DefaultImGuiWindowType::ProvideFullScreenWindow;

    // enableViewports: Enable multiple viewports (i.e. multiple native windows)
    // If true, you can drag windows outside the main window,
    // in order to put their content into new native windows.
    bool enableViewports = false;

    // Make windows only movable from the title bar
    bool configWindowsMoveFromTitleBarOnly = true;


    // ------------ Menus & Status bar --------------------------------------------------

    // Set the title of the App menu. If empty, the menu name will use
    // the "windowTitle" from AppWindowParams//
    std::string menuAppTitle = "";

    // Show Menu bar on top of imgui main window.
    // In order to fully customize the menu, set showMenuBar to true, and set showMe
    // and showMenu_View params to false. Then, implement the callback
    // `RunnerParams.callbacks.ShowMenus`
    // which can optionally call `HelloImGui::ShowViewMenu` and `HelloImGui::ShowApp
    bool showMenuBar = false;

    //  If menu bar is shown, include or not the default app menu
    bool showMenu_App = true;

    // Include or not a "Quit" item in the default app menu.
    // Set this to false if you intend to provide your own quit callback
    // with possible user confirmation
    // (and implement it inside RunnerCallbacks.ShowAppMenuItems)
    bool showMenu_App_Quit = true;

    // If menu bar is shown, include or not the default _View_ menu, that enables
    // to change the layout and set the docked windows and status bar visibility)
    bool showMenu_View = true;

    // Show theme selection in view menu
    bool showMenu_View_Themes = true;
    // `rememberTheme`: _bool, default=true_. Remember selected theme
    bool rememberTheme = true;

    // Flag that enable to show a Status bar at the bottom. You can customize
    // the status bar via RunnerCallbacks.ShowStatus()
    bool showStatusBar = false;

    // If set, display the FPS in the status bar.
    bool showStatus_Fps = true;
    // If set, showStatusBar and showStatus_Fps are stored in the application setti
    bool rememberStatusBarSettings = true;


    // ------------ Change the dockspace or background window size ----------------

    // If defaultImGuiWindowType = ProvideFullScreenWindow or ProvideFullScreenDockS
    // you can set the position and size of the background window:
    //    - fullScreenWindow_MarginTopLeft is the window position
    //    - fullScreenWindow_MarginBottomRight is the margin between
    //      the "application window" bottom right corner
    //      and the "imgui background window" bottom right corner
    // Important note:
    //     In order to be Dpi aware, those sizes are in *em units*, not in pixels,
    //     i.e. in multiples of the font size! (See HelloImGui::EmToVec2)
    ImVec2 fullScreenWindow_MarginTopLeft     = ImVec2(0.f, 0.f);
    ImVec2 fullScreenWindow_MarginBottomRight = ImVec2(0.f, 0.f);


    // ------------ Theme -----------------------------------------------------------

    // tweakedTheme: (enum ImGuiTheme::ImGuiTweakedTheme)
    // Changes the ImGui theme. Several themes are available, you can query the lis
    // by calling HelloImGui::AvailableThemes()
    ImGuiTheme::ImGuiTweakedTheme tweakedTheme;

    // backgroundColor:
    // This is the "clearColor", i.e. the app window background color, is visible *
    //    runnerParams.imGuiWindowParams.defaultImGuiWindowType = NoDefaultWindow
    // Alternatively, you can set your own RunnerCallbacks.CustomBackground to have
    // control over what is drawn behind the Gui.
    ImVec4 backgroundColor = ImVec4(0.f, 0.f, 0.f, 0.f);

};
```

Default window types

```
// `DefaultImGuiWindowType` is an enum class that defines whether a full screen bac
// window is provided or not
enum class DefaultImGuiWindowType
{
    // `ProvideFullScreenWindow`: a full window is provided in the background
    ProvideFullScreenWindow,
    // `ProvideFullScreenDockSpace`: a full screen dockspace is provided in the bac
    ProvideFullScreenDockSpace,
    // `NoDefaultWindow`: No default window is provided
    // (except for ImGui's default "debug" window)
    NoDefaultWindow
};
```

Fps Idling

See runner_params.h.

```
// FpsIdlingMode is an enum that describes the different modes of idling when rende
// - Sleep: the application will sleep when idling to reduce CPU usage.
// - EarlyReturn: rendering will return immediately when idling.
//   This is specifically designed for event-driven, and real-time applications.
//   Avoid using it in a tight loop without pauses, as it may cause excessive CPU c
// - Auto: use platform-specific default behavior.
//    On most platforms, it will sleep. On Emscripten, `Render()` will return immed
//    to avoid blocking the main thread.
// Note: you can override the default behavior by explicitly setting Sleep or Early
enum class FpsIdlingMode
{
    Sleep,
    EarlyReturn,
    Auto,
};

// FpsIdling is a struct that contains Fps Idling parameters
struct FpsIdling
{
    // `fpsIdle`: _float, default=9_.
    //  ImGui applications can consume a lot of CPU, since they update the screen
    //  very frequently. In order to reduce the CPU usage, the FPS is reduced when
    //  no user interaction is detected.
    //  This is ok most of the time but if you are displaying animated widgets
    //  (for example a live video), you may want to ask for a faster refresh:
    //  either increase fpsIdle, or set it to 0 for maximum refresh speed
    //  (you can change this value during the execution depending on your applicati
    //  refresh needs)
    float fpsIdle = 9.f;

    // `timeActiveAfterLastEvent`: _float, default=3.f_.
    //  Time in seconds after the last event before the application is considered i
    float timeActiveAfterLastEvent = 3.f;

    // `enableIdling`: _bool, default=true_.
    //  Disable idling by setting this to false.
    //  (this can be changed dynamically during execution)
    bool  enableIdling = true;

    // `isIdling`: bool (dynamically updated during execution)
    //  This bool will be updated during the application execution,
    //  and will be set to true when it is idling.
    bool  isIdling = false;

    // `rememberEnableIdling`: _bool, default=true_.
    //  If true, the last value of enableIdling is restored from the settings at st
    bool  rememberEnableIdling = false;

    // `fpsIdlingMode`: _FpsIdlingMode, default=FpsIdlingMode::Automatic_.
    // Sets the mode of idling when rendering the GUI (Sleep, EarlyReturn, Automati
    FpsIdlingMode fpsIdlingMode = FpsIdlingMode::Auto;
};
```

Dpi Aware Params

Optionally, DPI parameters can be fine-tuned. For detailed info, see handling screens with high dpi

Source: dpi_aware.h

```
//
// Hello ImGui will try its best to automatically handle DPI scaling for you.
//
// Parameter to change the scaling behavior:
// ------------------------------------------
// - `dpiWindowSizeFactor`:
//        factor by which window size should be multiplied
//    By default, Hello ImGui will compute it automatically, when it is set to 0.
//
//
// How to set manually:
// ---------------------------------
// If it fails (i.e. your window and/or fonts are too big or too small),
// you may set them manually:
//    (1) Either by setting them programmatically in your application
//        (set their values in `runnerParams.dpiAwareParams`)
//    (2) Either by setting them in a `hello_imgui.ini` file. See hello_imgui/hello_
// Note: if several methods are used, the order of priority is (1) > (2)
//
// For more information, see the documentation on DPI handling, here: https://pthom
//
struct DpiAwareParams
{
    // `dpiWindowSizeFactor`
    //     factor by which window size should be multiplied to get a similar
    //     physical size on different OSes (as if they were all displayed on a 96 PI
    //     This affects the size of native app windows,
    //     but *not* imgui Windows, and *not* the size of widgets and text.
    //  In a standard environment (i.e. outside of Hello ImGui), an application wit
    //  may have a physical size (in mm or inches) that varies depending on the scr
    //
    //  Inside Hello ImGui, the window size is always treated as targeting a 96 PPI
    //  look similar whatever the OS and the screen DPI.
    //  In our example, our 960x480 pixels window will try to correspond to a 10x5
    //
    //  Hello ImGui does its best to compute it on all OSes.
    //  However, if it fails you may set its value manually.
    //  If it is set to 0, Hello ImGui will compute it automatically,
    //  and the resulting value will be stored in `dpiWindowSizeFactor`.
    float dpiWindowSizeFactor = 0.0f;

    // `DpiFontLoadingFactor`
    //     factor by which font size should be multiplied at loading time to get a
    //     This is equal to dpiWindowSizeFactor
    //  The size will be equivalent to a size given for a 96 PPI screen
    float DpiFontLoadingFactor() const {
        return dpiWindowSizeFactor;
    }

};

// ---------------------------------------------------------------------------
```

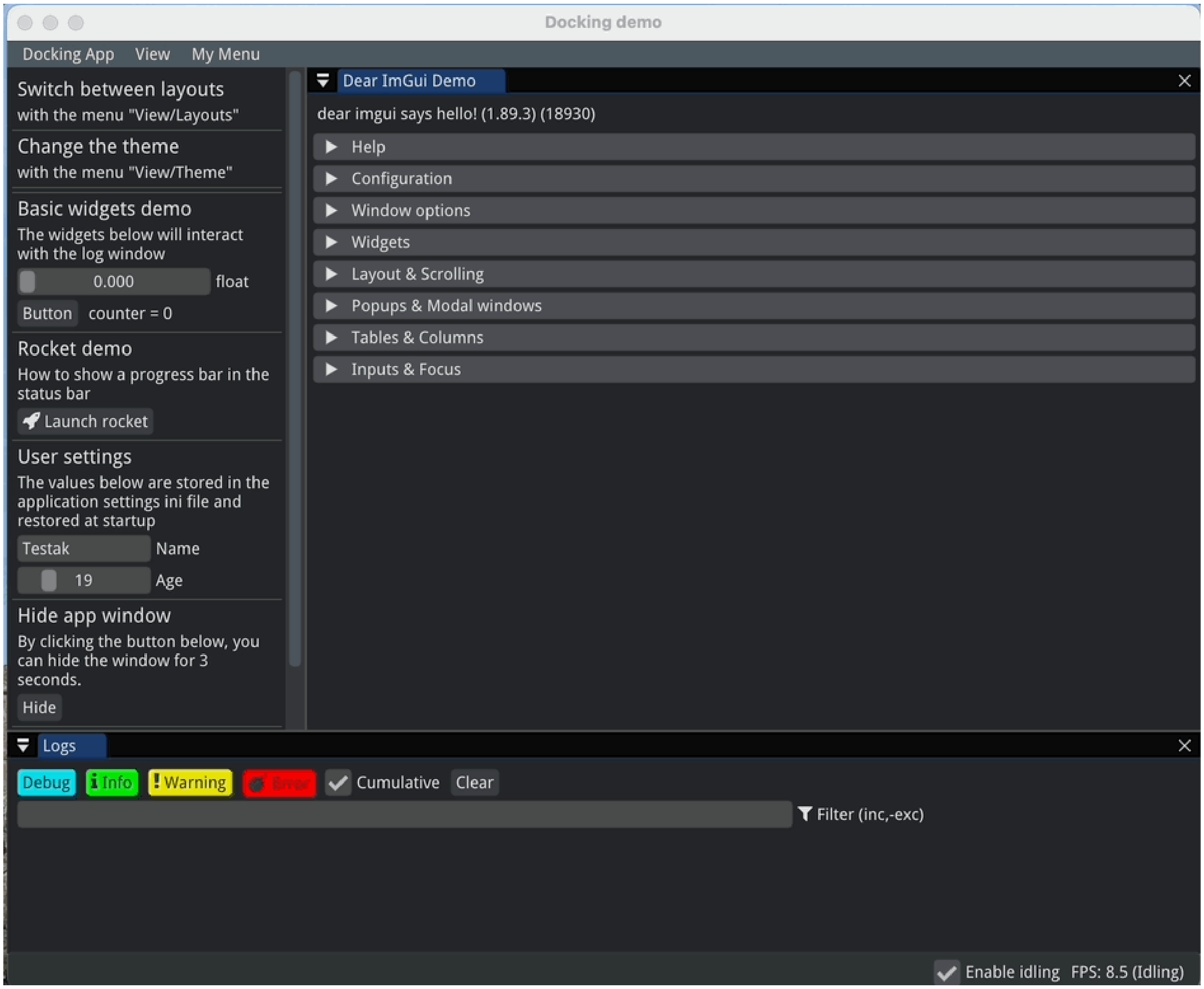
Docking

See docking_params.h.

HelloImGui makes it easy to use dockable windows (based on ImGui docking branch).

You can define several layouts and switch between them: each layout which will remember the user modifications and the list of opened windows

HelloImGui will then provide a “View” menu with options to show/hide the dockable windows, restore the default layout, switch between layouts, etc.



Switch between layouts
with the menu "View/Layouts"

Change the theme
with the menu "View/Theme"

Basic widgets demo
The widgets below will interact with the log window

0.000 float

counter = 0

Rocket demo
How to show a progress bar in the status bar

User settings
The values below are stored in the application settings ini file and restored at startup

Testak Name

19 Age

Hide app window
By clicking the button below, you can hide the window for 3 seconds.

dear imgui says hello! (1.89.3) (18930)

FPS: 8.5 (Idling)

- Source for this example: pthom/hello_imgui
- Video explanation on YouTube (5 minutes)

The different available layouts are provided inside RunnerParams via the two members below:

```
struct RunnerParams
{
    ...
    // default layout of the application
    DockingParams dockingParams;

    // optional alternative layouts
    std::vector<DockingParams> alternativeDockingLayouts;

    ...
};
```

And `DockingParams` contains members that define a layout:

```
struct DockingParams
{
    // displayed name of the layout
    std::string layoutName = "Default";

    // list of splits
    // (which define spaces where the windows will be placed)
    std::vector<DockingSplit> dockingSplits;

    // list of windows
    // (with their gui code, and specifying in which space they will be placed)
    std::vector<DockableWindow> dockableWindows;

    ...
};
```

Inside DockingParams, the member `dockingSplits` specifies the layout, and the member `dockableWindows` specifies the list of dockable windows, along with their default location, and their code (given by lambdas).

Docking Params: Example usage

Below is an example that shows how to instantiate a layout:

1. First, define the docking splits:

```
std::vector<HelloImGui::DockingSplit> CreateDefaultDockingSplits()
{
    //   Here, we want to split "MainDockSpace" (which is provided automatically)
    //   into three zones, like this:
    //    ___________________________________________
    //    |        |                                |
    //    | Command|                                |
    //    | Space  |    MainDockSpace               |
    //    |        |                                |
    //    |        |                                |
    //    |        |                                |
    //    -------------------------------------------
    //    |     MiscSpace                           |
    //    -------------------------------------------
    //

    // add a space named "MiscSpace" whose height is 25% of the app height.
    // This will split the preexisting default dockspace "MainDockSpace" in two par
    HelloImGui::DockingSplit splitMainMisc;
    splitMainMisc.initialDock = "MainDockSpace";
    splitMainMisc.newDock = "MiscSpace";
    splitMainMisc.direction = ImGuiDir_Down;
    splitMainMisc.ratio = 0.25f;

    // Then, add a space to the left which occupies a column
    // whose width is 25% of the app width
    HelloImGui::DockingSplit splitMainCommand;
    splitMainCommand.initialDock = "MainDockSpace";
    splitMainCommand.newDock = "CommandSpace";
    splitMainCommand.direction = ImGuiDir_Left;
    splitMainCommand.ratio = 0.25f;

    std::vector<HelloImGui::DockingSplit> splits {splitMainMisc, splitMainCommand};
    return splits;
}
```

2. Then, define the dockable windows:

```
std::vector<HelloImGui::DockableWindow> CreateDockableWindows(AppState& appState)
{
    // A Command panel named "Commands" will be placed in "CommandSpace".
    // Its Gui is provided calls "CommandGui"
    HelloImGui::DockableWindow commandsWindow;
    commandsWindow.label = "Commands";
    commandsWindow.dockSpaceName = "CommandSpace";
    commandsWindow.GuiFunction = [&] { CommandGui(appState); };

    // A Log window named "Logs" will be placed in "MiscSpace".
    // It uses the HelloImGui logger gui
    HelloImGui::DockableWindow logsWindow;
    logsWindow.label = "Logs";
    logsWindow.dockSpaceName = "MiscSpace";
    logsWindow.GuiFunction = [] { HelloImGui::LogGui(); };

    ...
}
```

3. Finally, fill the RunnerParams

```
HelloImGui::RunnerParams runnerParams;
runnerParams.imGuiWindowParams.defaultImGuiWindowType =
    HelloImGui::DefaultImGuiWindowType::ProvideFullScreenDockSpace;

runnerParams.dockingParams.dockingSplits = CreateDefaultDockingSplits();
runnerParams.dockingParams.dockableWindows = CreateDockableWindows();


HelloImGui::Run(runnerParams);
```

Docking Splits

```
// DockingSplit is a struct that defines the way the docking splits should
// be applied on the screen in order to create new Dock Spaces.
// DockingParams contains a
//     vector<DockingSplit>
// in order to partition the screen at your will.
struct DockingSplit
{
    // `initialDock`: _DockSpaceName (aka string)_
    //  id of the space that should be split.
    //  At the start, there is only one Dock Space named "MainDockSpace".
    //  You should start by partitioning this space, in order to create a new dock s
    DockSpaceName initialDock;

    // `newDock`: _DockSpaceName (aka string)_.
    //  id of the new dock space that will be created.
    DockSpaceName newDock;

    // `direction`: *ImGuiDir_*
    //  (enum with ImGuiDir_Down, ImGuiDir_Down, ImGuiDir_Left, ImGuiDir_Right)*
    //  Direction where this dock space should be created.
    ImGuiDir direction;

    // `ratio`: _float, default=0.25f_.
    //  Ratio of the initialDock size that should be used by the new dock space.
    float ratio = 0.25f;

    // `nodeFlags`: *ImGuiDockNodeFlags_ (enum)*.
    //  Flags to apply to the new dock space
    //  (enable/disable resizing, splitting, tab bar, etc.)
    ImGuiDockNodeFlags nodeFlags = ImGuiDockNodeFlags_None;

    // Constructor
    DockingSplit(const DockSpaceName& initialDock_ = "", const DockSpaceName& newDoc
                 ImGuiDir direction_ = ImGuiDir_Down, float ratio_ = 0.25f,
                 ImGuiDockNodeFlags nodeFlags_ = ImGuiDockNodeFlags_None)
        : initialDock(initialDock_), newDock(newDock_), direction(direction_), ratio
};
```

Dockable window

```
// DockableWindow is a struct that represents a window that can be docked.
struct DockableWindow
{
    // --------------- Main params -------------------

    // `label`: _string_. Title of the window. It should be unique! Use "##" to add
    std::string label;

    // `dockSpaceName`: _DockSpaceName (aka string)_.
    //  Id of the dock space where this window should initially be placed
    DockSpaceName dockSpaceName;

    // `GuiFunction`: _VoidFunction_.
    // Any function that will render this window's Gui
    VoidFunction GuiFunction = EmptyVoidFunction();


    // --------------- Options --------------------------

    // `isVisible`: _bool, default=true_.
    //  Flag that indicates whether this window is visible or not.
    bool isVisible = true;

    // `rememberIsVisible`: _bool, default=true_.
    //  Flag that indicates whether the window visibility should be saved in settin
    bool rememberIsVisible = true;

    // `canBeClosed`: _bool, default=true_.
    //  Flag that indicates whether the user can close this window.
    bool canBeClosed = true;

    // `callBeginEnd`: _bool, default=true_.
    //  Flag that indicates whether ImGui::Begin and ImGui::End
    //  calls should be added automatically (with the given "label").
    //  Set to false if you want to call ImGui::Begin/End yourself
    bool callBeginEnd = true;

    // `includeInViewMenu`: _bool, default=true_.
    //  Flag that indicates whether this window should be mentioned in the view men
    bool includeInViewMenu = true;

    // `imGuiWindowFlags`: _ImGuiWindowFlags, default=0_.
    //  Window flags, see enum ImGuiWindowFlags_
    ImGuiWindowFlags imGuiWindowFlags = 0;


    // --------------- Focus window -----------------------------
    // `focusWindowAtNextFrame`: _bool, default = false_.
    //  If set to true this window will be focused at the next frame.
    bool focusWindowAtNextFrame = false;


    // --------------- Size & Position --------------------------
    //              (only if not docked)

    // `windowSize`: _ImVec2, default=(0.f, 0.f) (i.e let the app decide)_.
    //  Window size (unused if docked)
    ImVec2 windowSize = ImVec2(0.f, 0.f);

    // `windowSizeCondition`: _ImGuiCond, default=ImGuiCond_FirstUseEver_.
    //  When to apply the window size.
    ImGuiCond  windowSizeCondition = ImGuiCond_FirstUseEver;

    // `windowPos`: _ImVec2, default=(0.f, 0.f) (i.e let the app decide)_.
    //  Window position (unused if docked)
    ImVec2 windowPosition = ImVec2(0.f, 0.f);

    // `windowPosCondition`: _ImGuiCond, default=ImGuiCond_FirstUseEver_.
    //  When to apply the window position.
    ImGuiCond  windowPositionCondition = ImGuiCond_FirstUseEver;


    // --------------- Constructor ------------------------------
    // Constructor
    DockableWindow(
        const std::string & label_ = "",
        const DockSpaceName & dockSpaceName_ = "",
        const VoidFunction guiFunction_ = EmptyVoidFunction(),
        bool isVisible_ = true,
        bool canBeClosed_ = true)
        : label(label_), dockSpaceName(dockSpaceName_),
          GuiFunction(guiFunction_),
          isVisible(isVisible_),
          canBeClosed(canBeClosed_) {}

};
```

Docking Params

```
// DockingParams contains all the settings concerning the docking:
//     - list of splits
//     - list of dockable windows
struct DockingParams
{
    // --------------- Main params -----------------------------

    // `dockingSplits`: _vector[DockingSplit]_.
    //  Defines the way docking splits should be applied on the screen
    //  in order to create new Dock Spaces
    std::vector<DockingSplit>   dockingSplits;

    // `dockableWindows`: _vector[DockableWindow]_.
    //  List of the dockable windows, together with their Gui code
    std::vector<DockableWindow> dockableWindows;

    // `layoutName`: _string, default="default"_.
    //  Displayed name of the layout.
    //  Only used in advanced cases, when several layouts are available.
    std::string layoutName = "Default";


    // --------------- Options -----------------------------

    // `mainDockSpaceNodeFlags`: _ImGuiDockNodeFlags (enum),
    //      default=ImGuiDockNodeFlags_PassthruCentralNode_
    //  Flags to apply to the main dock space
    //  (enable/disable resizing, splitting, tab bar, etc.).
    //  Most flags are inherited by children dock spaces.
    //  You can also set flags for specific dock spaces via `DockingSplit.nodeFlags
    ImGuiDockNodeFlags mainDockSpaceNodeFlags = ImGuiDockNodeFlags_PassthruCentralN


    // --------------- Layout handling -----------------------------

    // `layoutCondition`: _enum DockingLayoutCondition, default=FirstUseEver_.
    //  When to apply the docking layout. Choose between
    //      FirstUseEver (apply once, then keep user preference),
    //      ApplicationStart (always reapply at application start)
    //      Never
    DockingLayoutCondition layoutCondition = DockingLayoutCondition::FirstUseEver;

    // `layoutReset`: _bool, default=false_.
    //  Reset layout on next frame, i.e. drop the layout customizations which were
    //  applied manually by the user. layoutReset will be reset to false after this
    bool layoutReset = false;
    // --------------- Helper Methods ------------------------------

    // `DockableWindow * dockableWindowOfName(const std::string & name)`:
    // returns a pointer to a dockable window
    DockableWindow * dockableWindowOfName(const std::string& name);

    // `bool focusDockableWindow(const std::string& name)`:
    // will focus a dockable window (and make its tab visible if needed)
    bool focusDockableWindow(const std::string& windowName);

    // `optional<ImGuiID> dockSpaceIdFromName(const std::string& dockSpaceName)`:
    // returns the ImGuiID corresponding to the dockspace with this name
    std::optional<ImGuiID> dockSpaceIdFromName(const std::string& dockSpaceName);
};
```

Backend

Backend Pointers

```
//
// BackendPointers is a struct that contains optional pointers to the
// backend implementations (for SDL and GLFW).
//
// These pointers will be filled when the application starts, and you can use them
// to customize your application behavior using the selected backend.
//
// Note: If using the Metal, Vulkan or DirectX rendering backend, you can find
// some interesting pointers inside
//     `src/hello_imgui/internal/backend_impls/rendering_metal.h`
//     `src/hello_imgui/internal/backend_impls/rendering_vulkan.h`
//     `src/hello_imgui/internal/backend_impls/rendering_dx11.h`
//     `src/hello_imgui/internal/backend_impls/rendering_dx12.h`
struct BackendPointers
{
    //* `glfwWindow`: Pointer to the main GLFW window (of type `GLFWwindow*`).
    //  Only filled if the backend is GLFW.
    void* glfwWindow     = nullptr; /* GLFWwindow*    */

    //* `sdlWindow`: Pointer to the main SDL window (of type `SDL_Window*`).
    //  Only filled if the backend is SDL (or emscripten + sdl)
    void* sdlWindow      = nullptr; /* SDL_Window*    */

    //* `sdlGlContext`: Pointer to SDL's GlContext (of type `SDL_GLContext`).
    //  Only filled if the backend is SDL (or emscripten + sdl)
    void* sdlGlContext   = nullptr; /* SDL_GLContext  */
};
```

Renderer Backend Options

```
// `bool hasEdrSupport()`:
// Check whether extended dynamic range (EDR), i.e. the ability to reproduce
// intensities exceeding the standard dynamic range from 0.0-1.0, is supported.
//
// To leverage EDR support, you need to set `floatBuffer=true` in `RendererBackendO
// Only the macOS Metal backend currently supports this.
//
// This currently returns false on all backends except Metal, where it checks whethe
// this is supported on the current displays.
bool hasEdrSupport();


// RendererBackendOptions is a struct that contains options for the renderer backen
// (Metal, Vulkan, DirectX, OpenGL)
struct RendererBackendOptions
{
    // `requestFloatBuffer`:
    // Set to true to request a floating-point framebuffer.
    // Only available on Metal, if your display supports it.
    // Before setting this to true, first check `hasEdrSupport()`
    bool requestFloatBuffer = false;

    // `openGlOptions`:
    // Advanced options for OpenGL. Use at your own risk.
    OpenGlOptions openGlOptions;
};


// Note:
// If using Metal, Vulkan or DirectX, you can find interesting pointers inside:
//     src/hello_imgui/internal/backend_impls/rendering_metal.h
//     src/hello_imgui/internal/backend_impls/rendering_vulkan.h
//     src/hello_imgui/internal/backend_impls/rendering_dx11.h
//     src/hello_imgui/internal/backend_impls/rendering_dx12.h
```

API

Run Application

HelloImGui is extremely easy to use: there is **one** main function in the API, with three overloads.

HelloImGui::Run() will run an application with a single call.

Three signatures are provided:

- `HelloImGui::Run(RunnerParams &)`: full signature, the most customizable version. Runs an application whose params and Gui are provided by runnerParams.
- `HelloImGui::Run(const SimpleRunnerParams&)`: Runs an application, using simpler params.
- `HelloImGui::Run(guiFunction, windowTitle, windowSize, windowSizeAuto=false, restoreLastWindowGeometry=false, fpsIdle=10)` Runs an application, by providing the Gui function, the window title, etc.

Although the API is extremely simple, it is highly customizable, and you can set many options by filling the elements in the `RunnerParams` struct, or in the simpler `SimpleRunnerParams`.

HelloImGui::GetRunnerParams() will return the runnerParams of the current application.

Run Application while handling the rendering loop

If you want to be in control of the rendering loop, you may use the namespace `HelloImGui::ManualRender` (available since September 2024)

```cpp
namespace ManualRender
{
    // HelloImGui::ManualRender is a namespace that groups functions, allowing fine-
    // - It is customizable like HelloImGui::Run: initialize it with `RunnerParams`
    // - `ManualRender::Render()` will render the application for one frame:
    // - Ensure that `ManualRender::Render()` is triggered regularly (e.g., through
    //   to maintain responsiveness. This method must be called on the main thread.
    //
    // A typical use case is:
    // C++
    //        ```cpp
    //        HelloImGui::RunnerParams runnerParams;
    //        runnerParams.callbacks.ShowGui = ...; // your GUI function
    //        // Optionally, choose between Sleep, EarlyReturn, or Auto for fps idli
    //        // runnerParams.fpsIdling.fpsIdlingMode = HelloImGui::FpsIdlingMode::S
    //        HelloImGui::ManualRender::SetupFromRunnerParams(runnerParams);
    //        while (!HelloImGui::GetRunnerParams()->appShallExit)
    //        {
    //            HelloImGui::ManualRender::Render();
    //        }
    //        HelloImGui::ManualRender::TearDown();
    //        ```
    // Python:
    //        ```python
    //        runnerParams = HelloImGui.RunnerParams()
    //        runnerParams.callbacks.show_gui = ... # your GUI function
    //        while not hello_imgui.get_runner_params().app_shall_exit:
    //            hello_imgui.manual_render.render()
    //        hello_imgui.manual_render.tear_down()
    //        ```
    //
    // **Notes:**
    //  1. Depending on the configuration (`runnerParams.fpsIdling.fpsIdlingMode`),
    //     an idle state to reduce CPU usage, if no events are received (e.g., no i
    //     In this case, `Render()` will either sleep or return immediately.
    //     By default,
    //       - On Emscripten, `ManualRender::Render()` will return immediately to a
    //       - On other platforms, it will sleep
    //  2. If initialized with `RunnerParams`, a copy of the `RunnerParams` will be
    //     (which can be accessed with `HelloImGui::GetRunnerParams()`).

    // Initializes the rendering with the full customizable `RunnerParams`.
    // This will initialize the platform backend (SDL, Glfw, etc.) and the renderin
    // A distinct copy of `RunnerParams` is stored internally.
    void SetupFromRunnerParams(const RunnerParams& runnerParams);

    // Initializes the rendering with `SimpleRunnerParams`.
    // This will initialize the platform backend (SDL, Glfw, etc.) and the renderin
    void SetupFromSimpleRunnerParams(const SimpleRunnerParams& simpleParams);

    // Initializes the renderer with a simple GUI function and additional parameter
    // This will initialize the platform backend (SDL, Glfw, etc.) and the renderin
    void SetupFromGuiFunction(
```

```
        const VoidFunction& guiFunction,
        const std::string& windowTitle = "",
        bool windowSizeAuto = false,
        bool windowRestorePreviousGeometry = false,
        const ScreenSize& windowSize = DefaultWindowSize,
        float fpsIdle = 10.f
    );

    // Renders the current frame. Should be called regularly to maintain the applica
    void Render();

    // Tears down the renderer and releases all associated resources.
    // This will release the platform backend (SDL, Glfw, etc.) and the rendering ba
    // After calling `TearDown()`, the InitFromXXX can be called with new parameters
    void TearDown();
} // namespace ManualRender
```

Place widgets in a DPI-aware way

Special care must be taken in order to correctly handle screen with high DPI (for example, almost all recent laptops screens).

Using ImVec2 with fixed values is *almost always a bad idea* if you intend your application to be used on high DPI screens! Otherwise, widgets might be misplaced or too small on different screens and/or OS.

Instead, you should use scale your widgets and windows relatively to the font size, as is done with the em CSS Unit.

```
//  __HelloImGui::EmToVec2()__ returns an ImVec2 that you can use to size
//  or place your widgets in a DPI independent way.
//  Values are in multiples of the font size (i.e. as in the em CSS unit).
ImVec2 EmToVec2(float x, float y);
ImVec2 EmToVec2(ImVec2 v);

// __HelloImGui::EmSize()__ returns the visible font size on the screen.
float EmSize();
// __HelloImGui::EmSize(nbLines)__ returns a size corresponding to nbLines text line
float EmSize(float nbLines);

// __HelloImGui::PixelToEm()__ converts a Vec2 in pixels coord to a Vec2 in em unit
ImVec2 PixelsToEm(ImVec2 pixels);

// __HelloImGui::PixelSizeToEm()__ converts a size in pixels coord to a size in em
float  PixelSizeToEm(float pixelSize);
```

Load fonts

See hello_imgui_font.h.

```
// When loading fonts, use
//          HelloImGui::LoadFont(..)
//      or
//          HelloImGui::LoadDpiResponsiveFont()
//
// Use these functions instead of ImGui::GetIO().Fonts->AddFontFromFileTTF(),
// because they will automatically adjust the font size to account for HighDPI,
// and will help you to get consistent font size across different OSes.

//
// Font loading parameters: several options are available (color, merging, range
struct FontLoadingParams
{
    // if true, the font size will be adjusted automatically to account for Hig
    //
    bool adjustSizeToDpi = true;

    // if true, the font will be merged to the last font
    bool mergeToLastFont = false;

    // if true, the font will be loaded using colors
    // (requires freetype, enabled by IMGUI_ENABLE_FREETYPE)
    bool loadColor = false;

    // if true, the font will be loaded using HelloImGui asset system.
    // Otherwise, it will be loaded from the filesystem
    bool insideAssets = true;

    // ImGui native font config to use
    ImFontConfig fontConfig = ImFontConfig();
};


// Loads a font with the specified parameters
ImFont* LoadFont(
    const std::string & fontFilename, float fontSize,
    const FontLoadingParams & params = {});

ImFont* LoadFontTTF(
    const std::string & fontFilename,
    float fontSize,
    ImFontConfig config = ImFontConfig()
);

ImFont* LoadFontTTF_WithFontAwesomeIcons(
    const std::string & fontFilename,
    float fontSize,
    ImFontConfig configFont = ImFontConfig()
);
```

Applications assets

See hello_imgui_assets.h.

Load Assets as data buffer

```
struct AssetFileData
{
    void * data = nullptr;
    size_t dataSize = 0;
};

// LoadAssetFileData(const char *assetPath)`
// Will load an entire asset file into memory. This works on all platforms,
// including android.
// You *have* to call FreeAssetFileData to free the memory, except if you use
// ImGui::GetIO().Fonts->AddFontFromMemoryTTF, which will take ownership of the
// data and free it for you.
// This function can be redirected with setLoadAssetFileDataFunction. If not redire
// it calls DefaultLoadAssetFileData.
AssetFileData LoadAssetFileData(const char *assetPath);

// FreeAssetFileData(AssetFileData *)
// Will free the memory.
// Note: "ImGui::GetIO().Fonts->AddFontFromMemoryTTF" takes ownership of the data
// and will free the memory for you.
void FreeAssetFileData(AssetFileData * assetFileData);
```

Get assets path

```
//`std::string AssetFileFullPath(const std::string& assetRelativeFilename)`
// will return the path to assets.
//
// This works under all platforms *except Android*
// For compatibility with Android and other platforms, prefer to use `LoadAssetFileI
// whenever possible.
//    * Under iOS it will give a path in the app bundle (/private/XXX/....)
//    * Under emscripten, it will be stored in the virtual filesystem at "/"
//    * Under Android, assetFileFullPath is *not* implemented, and will throw an er
//      assets can be compressed under android, and you can't use standard file ope
//      Use LoadAssetFileData instead
std::string AssetFileFullPath(const std::string& assetRelativeFilename,
                              bool assertIfNotFound = true);

// Returns true if this asset file exists
bool AssetExists(const std::string& assetRelativeFilename);

// Sets the assets folder location
// (when using this, automatic assets installation on mobile platforms may not work
void SetAssetsFolder(const std::string& folder);
```

Display images from assets

See image_from_asset.h.

```
//
//Images are loaded when first displayed, and then cached
// (they will be freed just before the application exits).
//
//For example, given this files structure:
//```
//├── CMakeLists.txt
//├── assets/
//│         └── my_image.jpg
//└── my_app.main.cpp
//```
//
//then, you can display "my_image.jpg", using:
//
//    ```cpp
//    HelloImGui::ImageFromAsset("my_image.jpg");
//    ```


// `HelloImGui::ImageFromAsset(const char *assetPath, size, ...)`:
// will display a static image from the assets.
void ImageFromAsset(const char *assetPath, const ImVec2& size = ImVec2(0, 0),
                    const ImVec2& uv0 = ImVec2(0, 0), const ImVec2& uv1 = ImVec2(1,

// `HelloImGui::ImageFromAsset(const char *assetPath, size, ...)`:
// will display a static image from the assets, with a colored background and a bor
void ImageFromAssetWithBg(const char *assetPath, const ImVec2& size = ImVec2(0, 0),
            const ImVec2& uv0 = ImVec2(0, 0), const ImVec2& uv1 = ImVec2(1,1),
            const ImVec4& tint_col = ImVec4(1,1,1,1),
            const ImVec4& border_col = ImVec4(0,0,0,0));


// `bool HelloImGui::ImageButtonFromAsset(const char *assetPath, size, ...)`:
// will display a button using an image from the assets.
bool ImageButtonFromAsset(const char *assetPath, const ImVec2& size = ImVec2(0, 0),
                          const ImVec2& uv0 = ImVec2(0, 0),  const ImVec2& uv1 = Im
                          int frame_padding = -1,
                          const ImVec4& bg_col = ImVec4(0,0,0,0),
                          const ImVec4& tint_col = ImVec4(1,1,1,1));

// `ImTextureID HelloImGui::ImTextureIdFromAsset(assetPath)`:
// will return a texture ID for an image loaded from the assets.
ImTextureID ImTextureIdFromAsset(const char *assetPath);

// `ImVec2 HelloImGui::ImageSizeFromAsset(assetPath)`:
// will return the size of an image loaded from the assets.
ImVec2 ImageSizeFromAsset(const char *assetPath);


// `HelloImGui::ImageAndSize HelloImGui::ImageAndSizeFromAsset(assetPath)`:
// will return the texture ID and the size of an image loaded from the assets.
struct ImageAndSize
{
```

```
    ImTextureID textureId = ImTextureID(0);
    ImVec2 size = ImVec2(0.f, 0.f);
};
ImageAndSize ImageAndSizeFromAsset(const char *assetPath);


// `ImVec2 HelloImGui::ImageProportionalSize(askedSize, imageSize)`:
//  will return the displayed size of an image.
//     - if askedSize.x or askedSize.y is 0, then the corresponding dimension
//       will be computed from the image size, keeping the aspect ratio.
//     - if askedSize.x>0 and askedSize.y> 0, then the image will be scaled to fit
//       exactly the askedSize, thus potentially changing the aspect ratio.
//  Note: this function is used internally by ImageFromAsset and ImageButtonFromAss
//        so you don't need to call it directly.
ImVec2 ImageProportionalSize(const ImVec2& askedSize, const ImVec2& imageSize);
```

Utility functions

```
// `GetRunnerParams()`:  a convenience function that will return the runnerParams
// of the current application
    RunnerParams* GetRunnerParams();

// `IsUsingHelloImGui()`: returns true if the application is using HelloImGui
    bool IsUsingHelloImGui();

// `FrameRate(durationForMean = 0.5)`: Returns the current FrameRate.
//  May differ from ImGui::GetIO().FrameRate, since one can choose the duration
//  for the calculation of the mean value of the fps
//  Returns the current FrameRate. May differ from ImGui::GetIO().FrameRate,
//  since one can choose the duration for the calculation of the mean value of the 
//  (Will only lead to accurate values if you call it at each frame)
float FrameRate(float durationForMean = 0.5f);

// `ImGuiTestEngine* GetImGuiTestEngine()`: returns a pointer to the global instanc
//  of ImGuiTestEngine that was initialized by HelloImGui
//  (iif ImGui Test Engine is active).
ImGuiTestEngine* GetImGuiTestEngine();

// `GetBackendDescription()`: returns a string with the backend info
// Could be for example:
//     "Glfw - OpenGL3"
//     "Glfw - Metal"
//     "Sdl - Vulkan"
std::string GetBackendDescription();

// `ChangeWindowSize(const ScreenSize &windowSize)`: sets the window size
// (useful if you want to change the window size during execution)
void ChangeWindowSize(const ScreenSize &windowSize);


// `UseWindowFullMonitorWorkArea()`: sets the window size to the monitor work area
// (useful if you want to change the window size during execution)
void UseWindowFullMonitorWorkArea();
```

Switch between several layouts

See hello_imgui.h.

```
// In advanced cases when several layouts are available, you can switch between lay
// See demo inside
//     https://github.com/pthom/hello_imgui/tree/master/src/hello_imgui_demos/hello_

// `SwitchLayout(layoutName)`
//  Changes the application current layout. Only used in advanced cases
//  when several layouts are available, i.e. if you filled
//      runnerParams.alternativeDockingLayouts.
void           SwitchLayout(const std::string& layoutName);

// `CurrentLayoutName()`: returns the name of the current layout
std::string    CurrentLayoutName();

// `AddDockableWindow()`: will add a dockable window to the current layout.
// Will dock the window to the dockspace it belongs to if forceDockspace is true,
// otherwise will dock it to the last space it was docked to (using saved settings)
void AddDockableWindow(const DockableWindow& dockableWindow, bool forceDockspace = 

// `RemoveDockableWindow()`: will remove a dockable window from the current layout.
// (dockableWindowName is the label of the window, as provided in the DockableWindo
void RemoveDockableWindow(const std::string& dockableWindowName);
```

Ini settings

Ini settings location

```
// IniFolderType is an enum which describes where is the base path to store
// the ini file for the application settings.
//
// You can use IniFolderLocation(iniFolderType) to get the corresponding path.
//
// RunnerParams contains the following members, which are used to compute
// the ini file location:
//     iniFolderType           (IniFolderType::CurrentFolder by default)
//     iniFilename             (empty string by default)
//     iniFilename_useAppWindowTitle
//         (true by default: iniFilename is derived from
//          appWindowParams.windowTitle)
//
// iniFilename may contain a subfolder
// (which will be created inside the iniFolderType folder if needed)
//
enum class IniFolderType
{
    // CurrentFolder: the folder where the application is executed
    // (convenient for development, but not recommended for production)
    CurrentFolder,

    // AbsolutePath: an absolute path
    // (convenient, but not recommended if targeting multiple platforms)
    AbsolutePath,

    // AppUserConfigFolder:
    //      AppData under Windows (Example: C:\Users\[Username]\AppData\Roaming und
    //      ~/.config under Linux
    //      "~/Library/Application Support" under macOS
    // (recommended for production, if settings do not need to be easily accessible
    AppUserConfigFolder,

    // AppExecutableFolder: the folder where the application executable is located
    // (this may be different from CurrentFolder if the application is launched fro
    // (convenient for development, but not recommended for production)
    AppExecutableFolder,

    // HomeFolder: the user home folder
    // (recommended for production, if settings need to be easily accessible by the
    HomeFolder,

    // DocumentsFolder: the user documents folder
    DocumentsFolder,

    // TempFolder: the system temp folder
    TempFolder
};

// Returns the path corresponding to the given IniFolderType
std::string IniFolderLocation(IniFolderType iniFolderType);
```

```
// IniSettingsLocation returns the path to the ini file for the application setting
std::string IniSettingsLocation(const RunnerParams& runnerParams);

// HasIniSettings returns true if the ini file for the application settings exists.
bool HasIniSettings(const RunnerParams& runnerParams);

// DeleteIniSettings deletes the ini file for the application settings.
void DeleteIniSettings(const RunnerParams& runnerParams);
```

Store user settings in the ini file

See hello_imgui.h.

```
// 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. Use sparingly.

// `SaveUserPref(string userPrefName, string userPrefContent)`:
//  Shall be called in the callback runnerParams.callbacks.BeforeExit
void        SaveUserPref(const std::string& userPrefName, const std::string& userPr

// `string LoadUserPref(string& userPrefName)`
//  Shall be called in the callback runnerParams.callbacks.PostInit
std::string LoadUserPref(const std::string& userPrefName);
```

Customize Hello ImGui Menus

Hello ImGui provides a default menu and status bar, which you can customize by using the params: `RunnerParams.imGuiWindowParams.` `showMenuBar` / `showMenu_App` / `showMenu_View`

If you want to fully customize the menu:

- set `showMenuBar` to true, then set `showMenu_App` and `showMenu_View` params to false
- implement the callback `RunnerParams.callbacks.ShowMenus`: it can optionally call `ShowViewMenu` and `ShowAppMenu` (see below).

```
// `ShowViewMenu(RunnerParams & runnerParams)`:
// shows the View menu (where you can select the layout and docked windows visibili
void ShowViewMenu(RunnerParams & runnerParams);

// `ShowAppMenu(RunnerParams & runnerParams)`:
// shows the default App menu (including the Quit item)
void ShowAppMenu(RunnerParams & runnerParams);
```

Additional Widgets

InputTextResizable

```
// `InputTextResizable`: displays a resizable text input widget
//
// The `InputTextResizable` widget allows you to create a text input field that
// It supports both single-line and multi-line text input.
// Note: the size of the widget is expressed in em units.
// **Usage example:**
//    C++:
//       ```cpp
//       // Somewhere in the application state
//       (static) InputTextData textInput("My text", true, ImVec2(10, 3));
//       // In the GUI function
//       bool changed = InputTextResizable("Label", &textInput);
//       ```
//    Python:
//       ```python
//       # Somewhere in the application state
//       text_input = hello_imgui.InputTextData("My text", multiline=True, size_
//       # In the GUI function
//       changed = hello_imgui.input_text_resizable("Label", text_input)
//       ```
struct InputTextData
{
    // The text edited in the input field
    std::string Text;

    // An optional hint displayed when the input field is empty
    // (only works for single-line text input)
    std::string Hint;

    // If true, the input field is multi-line
    bool Multiline = false;

    // If true, the input field is resizable
    bool Resizable = true;

    // The size of the input field in em units
    ImVec2 SizeEm = ImVec2(0, 0);

    InputTextData(const std::string& text = "", bool multiline = false, ImVec2 s
};
bool InputTextResizable(const char* label, InputTextData* textInput);

// Serialization for InputTextData
```

```
    // --------------------------------
    // to/from dict
    using DictTypeInputTextData = std::map<std::string, std::variant<std::string, b
    DictTypeInputTextData InputTextDataToDict(const InputTextData& data);
    InputTextData InputTextDataFromDict(const DictTypeInputTextData& dict);
    // to/from string
    std::string InputTextDataToString(const InputTextData& data);
    InputTextData InputTextDataFromString(const std::string& str);
```

WidgetWithResizeHandle

```
    // WidgetWithResizeHandle: adds a resize handle to a widget
    // Example usage with ImPlot:
    //        void gui()
    //        {
    //            static ImVec2 widget_size(200, 200);
    //            auto myWidgetFunction = []()
    //            {
    //                if (ImPlot::BeginPlot("My Plot", widget_size)) {
    //                    ImPlot::PlotLine("My Line", x.data(), y.data(), 1000);
    //                    ImPlot::EndPlot();
    //                }
    //            };
    //            widget_size = widget_with_resize_handle("plot", myWidgetFunction)
    //        }
    ImVec2 WidgetWithResizeHandle(
        const char* id,
        VoidFunction widgetGuiFunction,
        float handleSizeEm = 1.0f,
        std::optional<VoidFunction> onItemResized = std::nullopt,
        std::optional<VoidFunction> onItemHovered = std::nullopt
        );
```

Handling screens with high DPI

Note: This part is relevant only for more advanced usages. If you use `HelloImGui::LoadFont()`, and always use `HelloImGui::EmToVec2()` to place widgets, you do not need to worry about DPI handling

OS specificities

There are several important things to know about high-DPI handling within Hello ImGui and Dear ImGui:

1. (virtual) screen coordinates vs (physical) pixels
2. DisplayFramebufferScale: Frame buffer size vs window size
3. FontGlobalScale: display-time font scaling factor
4. How to load fonts with the correct size
5. How to get similar window sizes on different OSes/DPI

Screen coordinates

Screen coordinates are the coordinates you use to place and size windows on the screen.

Screen coordinates do not always correspond to physical pixels

- On macOS/iOS retina screens, a screen coordinate corresponds typically to 2x2 physical pixels (but this may vary if you change the display scaling)
- On most Linux distributions, whenever there is a high DPI screen you can set the display scale. For example if you set the scale to 300%, then a screen coordinate will correspond to 3x3 physical pixels
- On Windows, there are two possible situations:
    - If the application is DPI aware, a screen coordinate corresponds to 1x1 physical pixel, and you can use the full extent of your screen resolution.
    - If the application is not DPI aware, a screen coordinate may correspond to 2x2 physical pixels (if the display scaling is set to 200% for example). However, the rendering of your application will be blurry and will not use the full extent of your screen resolution.
    - Notes:
        - Applications created with HelloImGui are DPI aware by default (when using glfw and sdl backends).
        - SDL applications are normally not DPI aware. However, HelloImGui makes them DPI aware.

DisplayFramebufferScale

`DisplayFramebufferScale` is the ratio between the frame buffer size and the window size.

The frame buffer size is the size of the internal buffer used by the rendering backend. It might be bigger than the actual window size. `ImVec2 ImGui::GetIO().DisplayFramebufferScale` is a factor by which the frame buffer size is bigger than the window size. It is set by the platform backend after it was initialized, and typically reflects the scaling ratio between physical pixels and screen coordinates.

Under windows, it will always be (1,1). Under macOS / linux, it will reflect the current display scaling. It will typically be (2,2) on a macOS retina screen.

Notes:

- You cannot change DisplayFramebufferScale manually, it will be reset at each new frame, by asking the platform backend.

How to load fonts with the correct size

Using HelloImGui (recommended)

`HelloImGui::LoadFont()` will load fonts with the correct size, taking into account the DPI scaling.

Using Dear ImGui

`ImGui::GetIO().Fonts->AddFontFromFileTTF()` loads a font with a given size, in *physical pixels*. KKDYNFONT: TBC...

Reproducible physical window sizes (in mm or

inches)

Using HelloImGui

Simply specify a window size that corresponds to theoretical 96 PPI screen (inside `RunnerParams.appWindowParams.windowGeometry.size`)

Using your own code to create the backend window

If you prefer to create the window by yourself, its physical size in millimeters may vary widely, depending on the OS and the current screen DPI setting. Typically under Windows, your window may appear to be very small if your screen is high DPI.

To get a similar window size on different OSes/DPI, you should multiply the window size by `HelloImGui::DpiWindowSizeFactor()`.

Note: DpiWindowSizeFactor() is equal to `CurrentScreenPixelPerInch / 96` under windows and linux, and always 1 under macOS.

Fine tune DPI Handling

See **`HelloImGui::DpiAwareParams`** for more information on how to fine tune DPI handling when using Hello ImGui.

Build instructions

Build Hello ImGui and its demos

On almost all platforms, HelloImGui can be compiled with simple commands:

~~~
```bash
git clone https://github.com/pthom/hello_imgui.git
cd hello_imgui
mkdir build && cd build
cmake ..
make -j
~~~

This will compile the HelloImGui library, and the demos (which will be located in the build/bin/ folder).

Build your application using Hello ImGui

To build an application that uses HelloImGui, you can either place HelloImGui inside your project (for example as a submodule), or it can be downloaded and built automatically by cmake.

In any case, follow the build instructions given in the HelloImGui Starter Template

Available backends

See documentation below (extract from CMakeLists.txt):

```
#
# You need to select at least two backends:
#
#     - At least one (or more) rendering backend (OpenGL3, Metal, Vulkan, DirectX11
#       Make your choice according to your needs and your target platforms, between
#          -DHELLOIMGUI_HAS_OPENGL3=ON    # This is the recommended choice, especia
#          -DHELLOIMGUI_HAS_METAL=ON      # Apple only, advanced users only
#          -DHELLOIMGUI_HAS_VULKAN=ON     # Advanced users only
#          -DHELLOIMGUI_HAS_DIRECTX11=ON  # Windows only, still experimental
#          -DHELLOIMGUI_HAS_DIRECTX12=ON  # Windows only, advanced users only, stil
#
#     - At least one (or more) platform backend (SDL2, Glfw3):
#      Make your choice according to your needs and your target platforms, between:
#          -DHELLOIMGUI_USE_SDL2=ON
#          -DHELLOIMGUI_USE_GLFW3=ON
#
# If you make no choice, the default will be selected:
#     HELLOIMGUI_USE_GLFW3 + HELLOIMGUI_HAS_OPENGL3
#
# Note about rendering backends:
#   OpenGL3 is the recommended choice as a starting point, especially for beginners
#   Vulkan, Metal, and DirectX11, DirectX12 do work, but you may need to customize
#   see src/hello_imgui/internal/backend_impls/rendering_xxxx.[h,cpp]
#   (using those backends probably implies that you want to heavily customize the r
#
################################################################################
# Platform backends:
option(HELLOIMGUI_USE_GLFW3 "Use Glfw3 as a platform backend" OFF)
option(HELLOIMGUI_USE_SDL2 "Use Sdl2 as a platform backend" OFF)
# Rendering backends
option(HELLOIMGUI_HAS_OPENGL3 "Use OpenGL3 as a rendering backend" OFF)
option(HELLOIMGUI_HAS_METAL "Use Metal as a rendering backend" OFF)
option(HELLOIMGUI_HAS_VULKAN "Use Vulkan as a rendering backend" OFF)
option(HELLOIMGUI_HAS_DIRECTX11 "Use DirectX11 as a rendering backend" OFF)
option(HELLOIMGUI_HAS_DIRECTX12 "Use DirectX12 as a rendering backend" OFF)

# Headless mode: by default, HelloImGui's cmake tooling will always check that there
# rendering backend and one platform backend that is not a "Null" backend.
# If you set HELLOIMGUI_HEADLESS, you can disable this check, and compile HelloImGu
# using only the Null rendering/platform backends.
option(HELLOIMGUI_HEADLESS "Allow headless mode (will use Null rendering/platform ba
```

In order to select your own backend, use one of the afore mentioned backend combinations, for example:

```
cmake .. -DHELLOIMGUI_USE_SDL2=ON -DHELLOIMGUI_HAS_OPENGL3=ON
```

Hello ImGui dependencies

HelloImGui depends on the following libraries:

- **Glfw3 or SDL2**, depending on the platform backend you selected
- **Freetype**, for font rendering, if HELLOIMGUI_USE_FREETYPE is ON, which is the default

Those dependencies may be downloaded and built automatically by cmake, or you can provide your own version of those libraries.

See documentation below (extract from CMakeLists.txt):

```
# Automatic download of Glfw3, SDL2, and Freetype (provided as a convenience)
#
# (this is disabled by default on Linux, which prefers to use the system libraries,
# enabled by default on other platforms)
#
# Note:
#
# SDL and Glfw3 will be downloaded only if:
# -----------------------------------------
#   - HELLOIMGUI_DOWNLOAD_GLFW_IF_NEEDED or HELLOIMGUI_DOWNLOAD_SDL_IF_NEEDED is ON
#   - HELLOIMGUI_USE_SDL_XXXX or HELLOIMGUI_USE_GLFW_XXXX is ON
#   - SDL and/or Glfw3 were not added as CMake target
#     (add_subdirectory(external/SDL) or add_subdirectory(external/glfw) for exampl
#   - find_package(glfw3 or SDL2) fails. If a system library is found,
#     or a user library provided in a path added to CMAKE_PREFIX_PATH,
#     it will be used instead
#
# - Freetype will be downloaded only if:
# --------------------------------------
#   - HELLOIMGUI_DOWNLOAD_FREETYPE is ON, HELLOIMGUI_USE_FREETYPE is ON
#   - Freetype was not added as a CMake target
#         (add_subdirectory(external/freetype) for example)
#   - find_package(freetype) fails
#   (it will also be forcibly downloaded if HELLOIMGUI_FREETYPE_STATIC is ON)
#
#
# Automatic download of SDL2, Glfw3, and Freetype is disabled by default on Linux,
# because it is recommended to use the system libraries instead:
# On ubuntu, you can install them with:
#    sudo apt install libglfw3-dev libsdl2-dev libfreetype-dev
#
#------------------------------------------------------------------------------
if(CMAKE_SYSTEM_NAME MATCHES "Linux")
    set(autodownload_default OFF)
else()
    set(autodownload_default ON)
endif()
option(HELLOIMGUI_DOWNLOAD_GLFW_IF_NEEDED "Download and build GLFW if needed" ${aut
option(HELLOIMGUI_DOWNLOAD_SDL_IF_NEEDED "Download and build SDL2 if needed" ${auto

# HELLOIMGUI_DOWNLOAD_FREETYPE_IF_NEEDED:
# - if ON: freetype will be downloaded and built if needed.
#   plutosvg will also be downloaded and built if needed (if HELLOIMGUI_USE_FREETYP
# - if OFF: freetype shall be provided by the system, or by the user (via CMAKE_PRE
#   Same for plutosvg (if HELLOIMGUI_USE_FREETYPE_PLUTOSVG is ON)
option(HELLOIMGUI_DOWNLOAD_FREETYPE_IF_NEEDED "Download and build Freetype if neede
option(HELLOIMGUI_FREETYPE_STATIC "Force static linking of freetype (only used for
```

Get dependencies via vcpkg

You can install almost all required dependencies with vcpkg.

Note: this will not support ImGui Test Engine, as it is not available in vcpkg yet.

Manually

The file vcpkg.json defines the dependencies required by HelloImGui.

For example:

```
# Clone hello_imgui
git clone https://github.com/pthom/hello_imgui.git
cd hello_imgui
# Clone vcpkg -& bootstrap
git clone https://github.com/microsoft/vcpkg.git
./vcpkg/bootstrap-vcpkg.sh
export VCPKG_ROOT=$(pwd)/vcpkg     # You *need* to set this environment variable
                                   # to tell cmake where to find vcpkg

# Install dependencies required by hello_imgui
# (they will be read from the vcpkg.json file)
./vcpkg/vcpkg install

# Build hello_imgui
mkdir build && cd build
cmake .. -DCMAKE_TOOLCHAIN_FILE=../vcpkg/scripts/buildsystems/vcpkg.cmake
cmake --build . -j 4
```

Using CMake Presets:

The file CMakePresets.json defines several presets which will use vcpkg to grab the required dependencies.

Thus, you can build HelloImGui with vcpkg dependencies, using a CMake preset, like this:

```
# Clone hello_imgui
git clone https://github.com/pthom/hello_imgui.git
cd hello_imgui
# Clone vcpkg -& bootstrap
git clone https://github.com/microsoft/vcpkg.git
./vcpkg/bootstrap-vcpkg.sh
export VCPKG_ROOT=$(pwd)/vcpkg     # You *need* to set this environment variable
                                   # to tell cmake where to find vcpkg

# Use the CMake preset "build_vcpkg_default"
# This will grab all dependencies from vcpkg,
# and use vcpkg toolchain file
cmake --preset build_vcpkg_default
cmake --build . -j 4
```

Hello ImGui CMake options

The CMakelists.txt file is heavily documented.

Below are the most important options:

Freetype (default: ON, except for Android and Mingw)

```
option(HELLOIMGUI_USE_FREETYPE "Use freetype for text rendering" ${freetype_default}
```

ImGui Test Engine (default: OFF)

```
option(HELLOIMGUI_WITH_TEST_ENGINE "Provide ImGui Test engine" OFF)
```

OS specific instructions

Windows instructions

Under windows, Hello ImGui will automatically provide a `WinMain()` function that will call main, and expects its signature to be `int main(int, char**)`. You may get a linker error if your main function signature is for example `int main()`.

You can disable this via cmake by passing `-DHELLOIMGUI_WIN32_AUTO_WINMAIN=OFF` as a command line cmake option. In this case, write your own `WinMain` under windows.

Warning: if using SDL, you will need to `#define SDL_MAIN_HANDLED` before any inclusion of SDL.h (to refrain SDL from #defining `#define main SDL_main`)

iOS instructions

"SDL + OpenGL ES3" is currently the preferred backend for iOS.

This project uses the ios-cmake toolchain which is a submodule in the folder hello_imgui_cmake/.

See compilation instructions in the HelloImGui Starter Template

Emscripten instructions

> emscripten is a toolchain for compiling to asm.js and WebAssembly, built using LLVM, that lets you run C and C++ on the web at near-native speed without plugins.

See compilation instruction in the HelloImGui Starter Template

To test your emscripten application, you will need a web server. Python provides a basic web server that is easy to usen which you can launch like this:

```
cd build_emscripten
python3 -m http.server
```

Open a browser, and navigate to http://localhost:8000.

macOS instructions

If you prefer to build regular terminal executables (not app bundles), use the cmake option `-DHELLOIMGUI_MACOS_NO_BUNDLE=ON`.

Android instructions

The Android version uses SDL + OpenGLES3.

See compilation instructions in the HelloImGui Starter Template