# The “Dear ImGui” User Interface Library

## What is Dear Imgui?

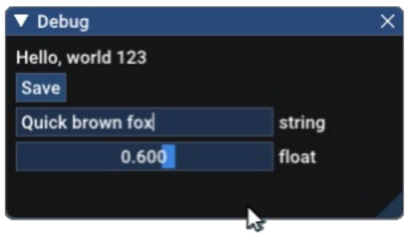
Dear ImGui, or just ImGui, is a very popular and easy to use graphical user interface library for C++ that easily can be plugged into most rendering engines. It is available at https://github.com/ocornut/imgui

The name “ImGui” stands for Immediate Mode Graphical User Interface. *Immediate Mode* means that the interface is built as the code runs, rather than being designed in an external application first; we write code that responsively creates the interface on-the-fly and reports back any input.

This library is most often used to create developer tools and debug interfaces, rather than end-user-facing UI elements, though it could technically be used for that as well.

## Dear ImGui Example

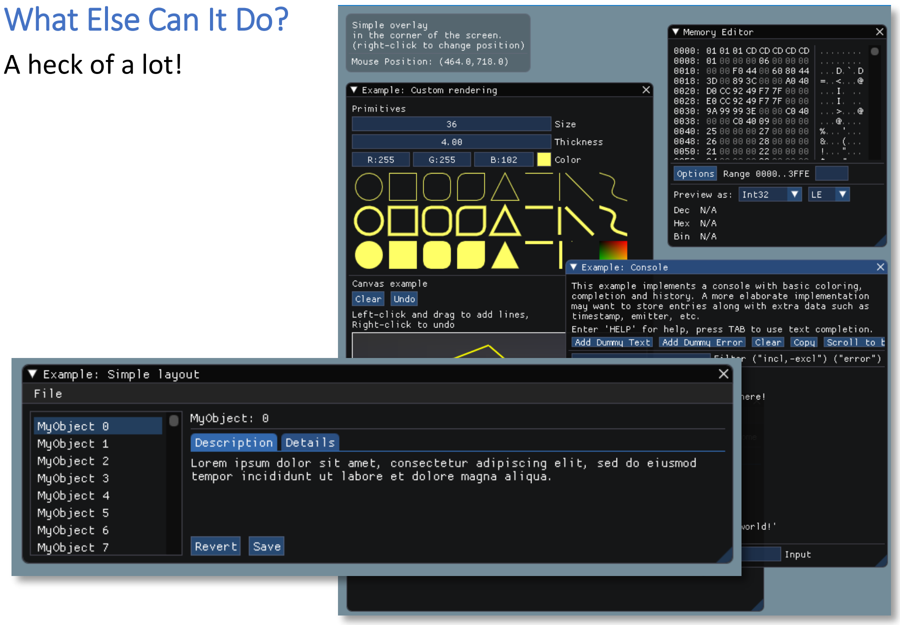
```
ImGui::Text("Hello, world %d", 123);
if (ImGui::Button("Save"))
    MySaveFunction();
ImGui::InputText("string", buf, IM_ARRAYSIZE(buf));
ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```



Pretty easy, right? Check out the GitHub repo (link above) for more details and usage examples.

## What Else Can It Do?

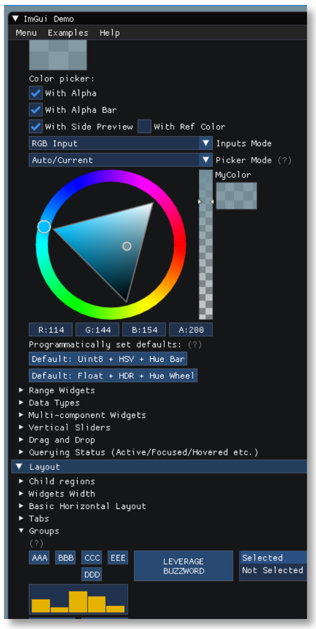
A heck of a lot!



```
0000: 01 01 01 CD CD CD CD CD
0008: 01 00 00 00 06 00 00 00
0010: 00 00 F0 44 00 60 80 44
0018: 3D 00 89 3C 00 00 A0 40
0020: D0 CC 92 49 F7 7F 00 00
0028: E0 CC 92 49 F7 7F 00 00
0030: 9A 99 99 3E 00 00 C0 40
0038: 00 00 C0 40 09 00 00 00
0040: 25 00 00 00 27 00 00 00
0048: 26 00 00 00 28 00 00 00
0050: 21 00 00 00 22 00 00 00
```

This example implements a console with basic coloring, completion and history. A more elaborate implementation may want to store entries along with extra data such as timestamp, emitter, etc.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.



## Integrating Dear ImGui

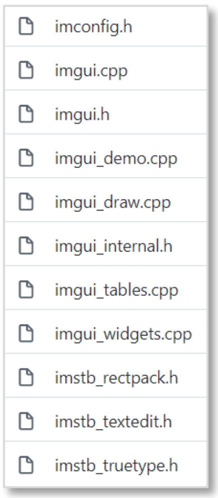
Basic integration requires getting the files, initializing the library, feeding it the correct information at run-time and telling it when to draw to the screen. After that, you can build a custom interface with very little code.

### Step 1: Getting the Files

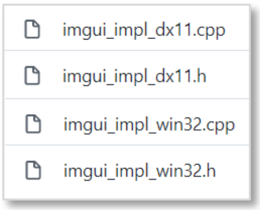
The first step to integrating the library is to get all of the necessary files. For that, you'll need to head to GitHub: [https://github.com/ocornut/imgui](https://github.com/ocornut/imgui)

Create a subfolder named "ImGui" in your project's directory. Copy the following files into that new subfolder:

- All .h & .cpp files from repo's root folder
- DX11 and Win32 files in /backends folder



```
imconfig.h
imgui.cpp
imgui.h
imgui_demo.cpp
imgui_draw.cpp
imgui_internal.h
imgui_tables.cpp
imgui_widgets.cpp
imstb_rectpack.h
imstb_textedit.h
imstb_truetype.h
```



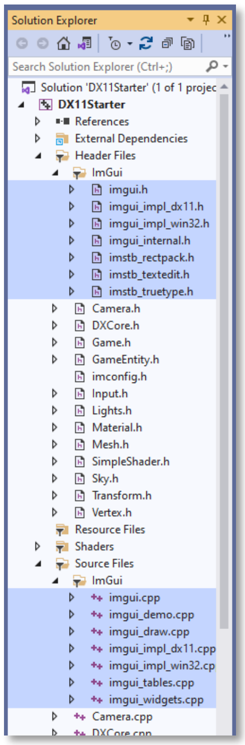
```
imgui_impl_dx11.cpp
imgui_impl_dx11.h
imgui_impl_win32.cpp
imgui_impl_win32.h
```

### Step 2: Including the Files

Next, include ALL of these files in your Visual Studio project. You may want to use *filters* here to organize the files so you don't have to sort through them or scroll past them in the future.

> *Filters in a Visual Studio project are like folders for your project files. However, they're simply a visual organization tool and do <u>not</u> correspond to actual folders on your hard drive.*

Since we are adding the code files directly to our project, there are no .lib or .dll files we need to worry about. We'll literally be building the code along with our project.



```
Solution 'DX11Starter' (1 of 1 projec
  DX11Starter
    References
    External Dependencies
    Header Files
      ImGui
        imgui.h
        imgui_impl_dx11.h
        imgui_impl_win32.h
        imgui_internal.h
        imstb_rectpack.h
        imstb_textedit.h
        imstb_truetype.h
      Camera.h
      DXCore.h
      Game.h
      GameEntity.h
      imconfig.h
      Input.h
      Lights.h
      Material.h
      Mesh.h
      SimpleShader.h
      Sky.h
      Transform.h
      Vertex.h
    Resource Files
    Shaders
    Source Files
      ImGui
        imgui.cpp
        imgui_demo.cpp
        imgui_draw.cpp
        imgui_impl_dx11.cpp
        imgui_impl_win32.cp
        imgui_tables.cpp
        imgui_widgets.cpp
      Camera.cpp
      DXCore.cpp
```

## Step 3: Integrating the Library

There are several places you'll need to "hook up" Dear ImGui to your engine:

- Game – Initializing, updating, drawing, etc.
- DXCore – Feeding operating system messages to the library

### Header File Includes

The main header for the library is "imgui.h". If you created a subfolder named "ImGui" to organize the project files, then you'll need to reference that folder when including the headers. In addition, you'll need a few other headers to actually draw the UI. Here is the whole list:

```
// This code assumes files are in "ImGui" subfolder!
// Adjust as necessary for your own folder structure
#include "ImGui/imgui.h"
#include "ImGui/imgui_impl_dx11.h"
#include "ImGui/imgui_impl_win32.h"
```

You'll need to include these in any file that references the library, including Game.cpp and DXCore.cpp. If you decide to eventually create a custom class for the UI, do the same there.

### `Game::~Game()`

The Game class destructor needs to clean up the library and free its memory. This is extremely simple:

```
// ImGui clean up
ImGui_ImplDX11_Shutdown();
ImGui_ImplWin32_Shutdown();
ImGui::DestroyContext();
```

### `Game::Init()`

Initializing the library is similarly easy. The platform and rendering backend functions need some information about the window and our graphics API. You can also choose the color scheme here:

```
// Initialize ImGui itself & platform/renderer backends
IMGUI_CHECKVERSION();
ImGui::CreateContext();
ImGui_ImplWin32_Init(hWnd);
ImGui_ImplDX11_Init(device.Get(), context.Get());

// Pick a style (uncomment one of these 3)
ImGui::StyleColorsDark();
//ImGui::StyleColorsLight();
//ImGui::StyleColorsClassic();
```

## Game::Update()

The following must be done at the *very top of Update()*, so that the UI has fresh input data and it knows a new frame has started. Even better, *create a helper method* for all of this and call that in Update().

```
// Feed fresh input data to ImGui
ImGuiIO& io = ImGui::GetIO();
io.DeltaTime = deltaTime;
io.DisplaySize.x = (float)this->windowWidth;
io.DisplaySize.y = (float)this->windowHeight;

// Reset the frame
ImGui_ImplDX11_NewFrame();
ImGui_ImplWin32_NewFrame();
ImGui::NewFrame();

// Determine new input capture
Input& input = Input::GetInstance();
input.SetKeyboardCapture(io.WantCaptureKeyboard);
input.SetMouseCapture(io.WantCaptureMouse);

// Show the demo window
ImGui::ShowDemoWindow();
```

## Game::Draw()

Another easy one. We tell ImGui to prepare its buffers and then feed that draw data to another built-in function. This should be the last thing drawn, so do this *right before* the call to swapChain->Present();

```
ImGui::Render();
ImGui_ImplDX11_RenderDrawData(ImGui::GetDrawData());
```

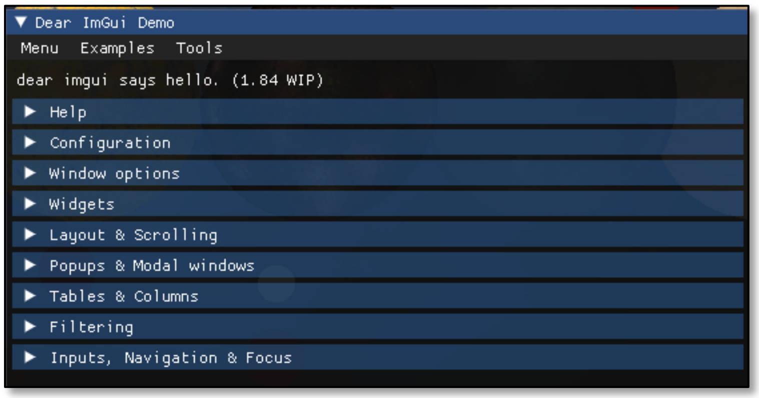
## DXCore::ProcessMessage()

We need to feed any and all input to ImGui, which can be done quite simply by passing any operating system messages along to the library – so we must edit DXCore slightly. Open DXCore.cpp and *include imgui_impl_win32.h*. Then, add the following to the *very top* of the ProcessMessage() function, before the switch statement:

```
// Forward declare ImGui's message handler (this is required!)
extern IMGUI_IMPL_API LRESULT ImGui_ImplWin32_WndProcHandler(
	HWND hWnd,
	UINT msg,
	WPARAM wParam,
	LPARAM lParam);

// Call ImGui’s message handler and exit early if necessary
if (ImGui_ImplWin32_WndProcHandler(hWnd, uMsg, wParam, lParam))
	return true;
```

### Testing

If you've added all of the necessary code, you should see the official demo window when you run your program. You should be able to move it around, minimize it and explore its features at this point, too. While it doesn't tell you exactly how to *code* each widget, it does showcase many possibilities. Once you find something you want to mimic, you can open imgui_demo.cpp and see the code that built it.



```
dear imgui says hello. (1.84 WIP)
▶ Help
▶ Configuration
▶ Window options
▶ Widgets
▶ Layout & Scrolling
▶ Popups & Modal windows
▶ Tables & Columns
▶ Filtering
▶ Inputs, Navigation & Focus
```

## Creating a Custom UI

Now that Dear ImGui is up and running, you can actually create something with it.

### UI Code Placement

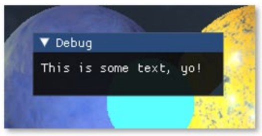
In general, since the UI will not only be using data from your engine, but potentially also changing it, you should place your UI-creation code within Game::Update (or a helper method called from there). It's rare for your drawing code to build its own UI.

Note that this must happen AFTER the "new frame initialization" code you've already added to Update().

### General ImGui Usage

All functions are within the ImGui namespace. Most of the functions create a corresponding UI element, though some can retrieve information about the current layout of the UI. For example:

```
ImGui::Text("This is some text, yo!");
```



```
This is some text, yo!
```

In this example, a basic window with the title "Debug" was created automatically since the call to ImGui::Text() was the only user interface call made. See later in this document for examples of creating your own window.

Any function that creates a UI element that the user can interact with, like a button or a slider, returns a Boolean indicating whether or not that element took input *this frame*. This allows you to write code that only executes when an element is actually clicked, like so:

```
if (ImGui::Button("Click me"))
{
	// This will execute only when the button is clicked
}
```

Any function that is meant to edit data, like a slider or check box, will take a pointer to a variable holding a current value. If the UI element is altered, the new value will be written to that pointer's variable.

```
// value is an integer variable, so &value is its address (a pointer)
// Create a slider from 0-100 which reads and updates value
ImGui::SliderInt("Choose a number", &value, 0, 100);
```

Functions exist to display and edit more complex data like vectors. To accomplish this, you'll need to provide the address of the *first element*.

```
XMFLOAT3 vec(10.0f, -2.0f, 99.0f);

// Provide the address of the first element to create a
// 3-component, draggable editor for a vector
ImGui::DragFloat3("Edit a vector", &vec.x);

XMFLOAT4 color(1.0f, 0.0f, 0.5f, 1.0f);

// You can create a 3 or 4-component color editor, too!
// Notice the two different function names below (ending with 3 and 4)
ImGui.ColorEdit3("3-component (RGB) color editor",  &color.x);
ImGui.ColorEdit4("4-component (RGBA) color editor", &color.x);
```

Editing a value retrieved from an object, like the position of a Transform object, is slightly more complex since you probably don't have the address of the underlying data. You can still create an editor for this kind of data by putting it in a variable first and overwriting the object's data if the UI was used.

```
// First, copy the data locally
XMFLOAT3 pos = myEntity->GetTransform()->GetPosition();

// Create the editor and check the resulting Boolean to know
// exactly when the data has changed
if (ImGui::DragFloat3("Entity Position", &pos.x))
{
	// Something changed, so overwrite the transform's data
	myEntity->GetTransform()->SetPosition(pos);
}
```

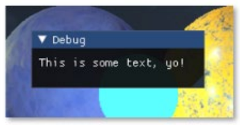
Note: The string you provide as a label for each element is also used as an internal ID for that element. This means that having two elements with the exact same label is problematic (and will result in both elements referring to the same data). To remedy this, you can add special characters in the string that aren't displayed (see below) or use the functions PushID() and PopID() to change the current ID.

```
// Each float3 editor below wants the same name ("Direction").  To prevent conflicts
// conflicts, a double pound sign "##" and unique characters are added.
// The "##" and anything after are not displayed in the UI.
ImGui::Text("Light 1");
ImGui::DragFloat3("Direction##1", &light1.x);

ImGui::Text("Light 2");
ImGui::DragFloat3("Direction##2", &light2.x);
```

## Creating a Custom Window

Reminder: If you don't explicitly create a window, it places all elements in a window named "Debug":

```
ImGui::Text("This is some text, yo!");
```

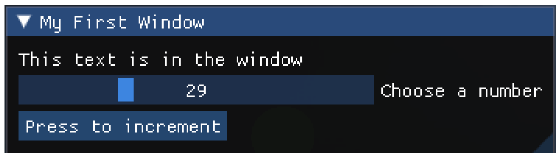


This is some text, yo!

While this is fine, you may want to create multiple windows and/or customize them. To do so, you'll call ImGui::Begin() to signify that you're starting to build the window and ImGui::End() to signify that you're done building it. The call to Begin() can take several parameters to customize the window, though at a minimum it needs a name to both display and to use as the internal window ID.

```
ImGui::Begin("My First Window"); // Everything after is part of the window

ImGui::Text("This text is in the window");

// value is an integer variable
// Create a slider from 0-100 which reads and updates value
ImGui::SliderInt("Choose a number", &value, 0, 100);

// Create a button and test for a click
if (ImGui::Button("Press to increment"))
{
        value++; // Adds to value when clicked
}

ImGui::End(); // Ends the current window
```

The code above creates the following window:



This text is in the window

## More Examples?

There are far too many functions to list here. I'm hoping that you explore the demo window to see what's available and dig into imgui_demo.cpp to see some of that code in action. The readme file on Dear ImGui's GitHub repo has a few more examples, too.

There's also a third-party online "interactive" manual for ImGui that you might find useful. It displays the demo window in your browser and shows the code of any element you hover over, allowing you to immediately see the code that created it. It does not, however, provide any extra documentation.