Dear PyGui

Jonathan Hoffstadt and Preston Cothren

Oct 10, 2021

ABOUT DPG

CHAPTER

ONE

ABOUT DPG

Dear PyGui is an easy-to-use, dynamic, GPU-Accelerated, cross-platform graphical user interface toolkit(GUI) for Python. It is not a traditional wrapping of Dear ImGui, but instead, more of a "built with" Dear ImGui.

Features include traditional GUI elements such as buttons, radio buttons, menus and various methods to create a functional layout.

Additionally, DPG has an incredible assortment of dynamic plots, tables, drawings, logging, debugger, and multiple resource viewers.

DPG is well suited for creating simple user interfaces as well as to developing complex and demanding graphical interfaces.

DPG offers a solid framework for developing scientific, engineering, gaming, data science and other applications that require fast and interactive interfaces.

1.1 What & Why

1.1.1 What is DPG

Dear PyGui is a simple to use (but powerful) Python GUI framework. Dear PyGui provides a wrapping of Dear ImGui that simulates a traditional retained mode GUI, as opposed to Dear ImGui's immediate mode paradigm.

Under the hood, Dear PyGui uses the immediate mode paradigm allowing for extremely dynamic interfaces. Similar to PyQt, Dear PyGui does not use native widgets but instead draws the widgets using your computer's graphics card (using Directx11, Metal, and Vulkan rendering APIs).

In the same manner Dear ImGui provides a simple way to create tools for game developers, Dear PyGui provides a simple way for python developers to create quick and powerful GUIs for scripts.

1.1.2 Why use DPG

When compared with other Python GUI libraries Dear PyGui is unique with:

- GPU Rendering
- Simple built-in Asynchronous function support
- Complete theme and style control
- Built-in developer tools: logging, theme inspection, resource inspection, runtime metrics
- 70+ widgets with hundreds of widget combinations
- Detailed documentation, examples and unparalleled support

1.2 Project Information

Socials:

| | | Github Discussions | Youtube |

Funding:

Repository:

Issues:

CI Builds:

CI Static Analysis:

Platforms:

Header row, column (header rows)	Rendering API	Latest Version
Windows 10	DirectX 11	
macOs	Metal	
Linux	OpenGL 3	
Raspberry Pi 4	OpenGL ES	

CHAPTER

TWO

FIRST STEPS

If you're ready to start using Dear PyGui visit the *First Steps* in tutorials.

The *Tutorials* will provide a great overview and links to each topic in the API Reference for more detailed reading.

However, use the API reference for the most detailed documentation on any specific topic.

2.1 First Steps

Tutorials will give a broad overview and working knowledge of DPG. Tutorials can not cover every detail so refer to the Reference API on each topic to learn more.

2.1.1 Installing

Python 3.6 (64 bit) or above is required.

```
pip install dearpygui
```

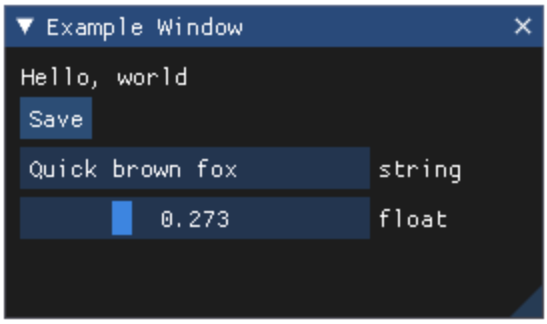
2.1.2 First Run

Lets check the pip install by creating a window, adding some Widgets. and starting the render loop with `start_dearpygui`.

Code:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Example Window"):
    dpg.add_text("Hello, world")
    dpg.add_button(label="Save")
    dpg.add_input_text(label="string", default_value="Quick brown fox")
    dpg.add_slider_float(label="float", default_value=0.273, max_value=1)

dpg.start_dearpygui()
```

Result:



Note: The main script must always conclude with `start_dearpygui`.

2.2 DPG Structure Overview

A DPG app will have an overall structure as follows:

- Viewport
- Render Loop
- Items
- Primary Window

2.2.1 Viewport

The viewport is the *window* created by the operating system.

Typically the viewport is handled automatically by DPG. The viewport needs to be explicitly created to be customized for taskbar icons, custom sizing, decorators, etc.

Lets go back and revisit the first app but create the viewport explicitly and give it a new title and size.

Code:

```
import dearpygui.dearpygui as dpg

vp = dpg.create_viewport(title='Custom Title', width=600, height=200)

with dpg.window(label="Example Window"):
    dpg.add_text("Hello, world")
    dpg.add_button(label="Save")
    dpg.add_input_text(label="string", default_value="Quick brown fox")
    dpg.add_slider_float(label="float", default_value=0.273, max_value=1)

dpg.setup_dearpygui(viewport=vp)
dpg.show_viewport(vp)

dpg.start_dearpygui()
```

See also:

For more information on the viewport *Viewport*

2.2.2 Render Loop

The render loop is responsible for displaying widgets, partially maintaining state and handling item callbacks.

The render loop is completely handled by the `start_dearpygui` command.

In some cases it's necessary to explicitly create the render loop for calling python commands that may need to run every frame.

Lets add this into the first app.

Code:

```
import dearpygui.dearpygui as dpg

vp = dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui(viewport=vp)

with dpg.window(label="Example Window"):
    dpg.add_text("Hello, world")
    dpg.add_button(label="Save")
    dpg.add_input_text(label="string", default_value="Quick brown fox")
    dpg.add_slider_float(label="float", default_value=0.273, max_value=1)

dpg.show_viewport(vp)

# below replaces, start_dearpygui()
while dpg.is_dearpygui_running():
    # insert here any code you would like to run in the render loop
    # you can manually stop by using stop_dearpygui()
    dpg.render_dearpygui_frame()

dpg.cleanup_dearpygui()
```

See also:

for more information on the render loop *Render Loop*

2.2.3 Items

DPG can be broken down into **Items**, **UI Items**, **Containers**

Items: Items are anything in the library.

UI Items: Any item in dpg that has a visual component (i.e. button, listbox, window, ect).

Containers: Items that can hold other items. A root container has no parent container.

2.2.4 Primary Window

DPG can assign one window to be the *primary window*, which will fill the viewport and always be drawn behind other windows.

Code:

```
import dearpygui.dearpygui as dpg

with dpg.window(id="Primary Window", label="Example Window"):
    dpg.add_text("Hello, world")
    dpg.add_button(label="Save")
    dpg.add_input_text(label="string", default_value="Quick brown fox")
    dpg.add_slider_float(label="float", default_value=0.273, max_value=1)

dpg.set_primary_window("Primary Window", True)
dpg.start_dearpygui()
```

See also:

for more information on the viewport *Primary Window*

2.3 Item Usage

2.3.1 Creating Items

Items are created using their *add_*** commands.

All items have a unique id which can either be specified by the keyword argument *id* or are automatically generated by DPG.

The *id* are either integers or strings and are used access the item after it has been created.

All items return their id's when they are created.

Warning: Item id's should be unique if specified using the *id* keyword. Integers 0-10 are reserved for DPG.

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):
    b0 = dpg.add_button(label="button 0")
    b1 = dpg.add_button(id=100, label="Button 1")
    dpg.add_button(id="Btn2", label="Button 2")

print(b0)
print(b1)
print(dpg.get_item_label("Btn2"))

dpg.start_dearpygui()
```

Note: Items can be created delete at runtime see *Item Creation (Runtime)*

See also:

For more information on the creating items:
Item Creation
Item ID System
Item Creation (Runtime)

2.3.2 Creating Containers

Below we will add a window, a group and a child container to the code items can either be added directly to the context manager or later by specifying the parent

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):
    dpg.add_button(label="Button 1")
    dpg.add_button(label="Button 2")
    with dpg.group():
        dpg.add_button(label="Button 3")
        dpg.add_button(label="Button 4")
        with dpg.group() as group1:
            pass
dpg.add_button(label="Button 6", parent=group1)
dpg.add_button(label="Button 5", parent=group1)

dpg.start_dearpygui()
```

See also:

For more information on containers:
Item Creation
Containers & Context Managers
Container Slots & Children
Container Stack

2.3.3 Configuration, State, Info

DPG items consist of configuration, state and info. (AND value but we will cover that separate)

Each of these can be accessed by its corresponding function

`get_item_configuration` keywords that control its appearance and behavior (label, callback, width, height)

`get_item_state` keywords that reflect its interaction (visible, hovered, clicked, ect) **State cannot be written to**

`get_item_info` keywords that reflect its information (item type, children, theme, ect)

Note: configuration, state and info have been broken into separate commands that access each individual keyword, instead of returning the entire dictionary.

Examples:
`get_item_label`
`is_item_hovered`
`get_item_children`

Below we will show the ways to configure the items and we can check their state by viewing them through the item registry tool.

Code:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):

    #configuration set when button is created
    dpg.add_button(label="Apply", width=300)

    #user data and callback set any time after button has been created
    btn = dpg.add_button(label="Apply 2")
    dpg.set_item_label(btn, "Button 57")
    dpg.set_item_width(btn, 200)

dpg.show_item_registry()

dpg.start_dearpygui()
```

See also:

For more information on the these topics:
Item Configuration
IO, Handlers, State Polling

2.3.4 Callbacks

Callbacks give UI items functionality and almost all UI Items in DPG can run callbacks.

Functions or methods are assigned as UI item callbacks when an item is created or at a later time using `set_item_callback`

Callbacks may have up to 3 standard keyword arguments:

sender: the *id* of the UI item that submitted the callback

app_data: occasionally UI items will send their own data (ex. file dialog)

user_data: any python object you want to send to the function

Code:

```
import dearpygui.dearpygui as dpg

def button_callback(sender, app_data, user_data):
```

(continues on next page)

(continued from previous page)

```
    print(f"sender is: {sender}")
    print(f"app_data is: {app_data}")
    print(f"user_data is: {user_data}")

with dpg.window(label="Tutorial"):

    #user data and callback set when button is created
    dpg.add_button(label="Apply", callback=button_callback, user_data="Some Data")

    #user data and callback set any time after button has been created
    btn = dpg.add_button(label="Apply 2", )
    dpg.set_item_callback(btn, button_callback)
    dpg.set_item_user_data(btn, "Some Extra User Data")

dpg.start_dearpygui()
```

See also:

For more information on the item callbacks *Item Callbacks*

2.3.5 Values

Most UI items have a value which can be accessed or set.

All UI items that have a value also have the *default_value* parameter which will set the items' initial starting value.

Values can be accessed using `get_value`

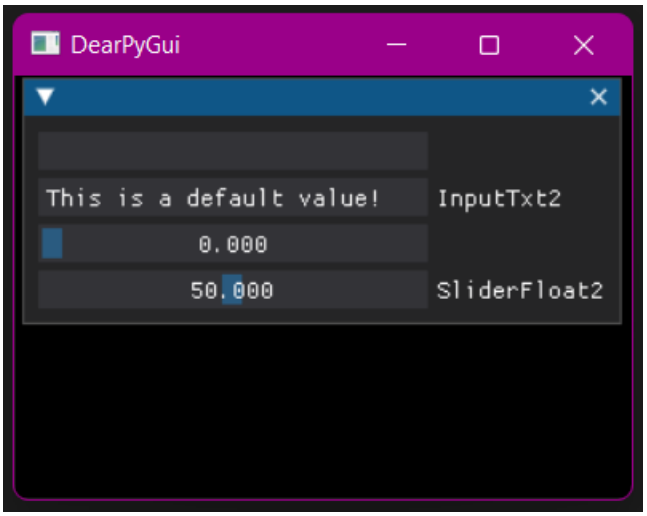
Below is an example of two setting the *default_value* for different items setting a callback to the items and printing their values.

```
import dearpygui.dearpygui as dpg

def print_value(sender):
    print(dpg.get_value(sender))

with dpg.window(width=300):
    input_txt1 = dpg.add_input_text()
    # The value for input_text2 will have a starting value
    # of "This is a default value!"
    input_txt2 = dpg.add_input_text(
        label="InputTxt2",
        default_value="This is a default value!",
        callback = print_value
    )

    slider_float1 = dpg.add_slider_float()
    # The slider for slider_float2 will have a starting value
    # of 50.0.
    slider_float2 = dpg.add_slider_float(
        label="SliderFloat2",
        default_value=50.0,
        callback=print_value
```

(continues on next page)

(continued from previous page)

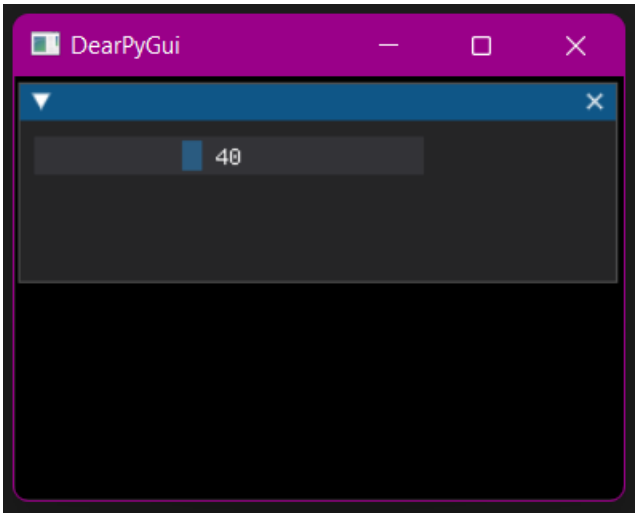
```
    )

    dpg.set_item_callback(input_txt1, print_value)
    dpg.set_item_callback(slider_float1, print_value)

    print(dpg.get_value(input_txt1))
    print(dpg.get_value(input_txt2))
    print(dpg.get_value(slider_float1))
    print(dpg.get_value(slider_float2))

dpg.start_dearpygui()
```



An input item's value is changed by interacting with it. In the above example, moving slider_float1 slider to 30.55 sets its value to 30.55.

We can set the position of the slider by changing items' value at runtime using `set_value`.

```
import dearpygui.dearpygui as dpg


with dpg.window(width=300):
    # Creating a slider_int widget and setting the
    # default value to 15.
    dpg.add_slider_int(default_value=15, id="slider_int")

# On second thought, we're gonna set the value to 40
# instead - for no reason in particular...
dpg.set_value("slider_int", 40)

dpg.start_dearpygui()
```



Note: The values' type depends on the widget. (ex.) input_int default value needs to be an integer.

See also:

For more information on item values *Item Value*

2.3.6 Item Handlers

UI item handlers listen for events (changes in state) related to a UI item then submit a callback.

```
import dearpygui.dearpygui as dpg

def change_text(sender, app_data):
    dpg.set_value("text_item", f"Mouse Button ID: {app_data}")

with dpg.window(width=500, height=300):
    dpg.add_text("Click me with any mouse button", id="text_item")
    dpg.add_clicked_handler(text_widget, callback=change_text)

dpg.start_dearpygui()
```

See also:

For more information on item handlers *IO, Handlers, State Polling*

2.4 Tips & More Resources

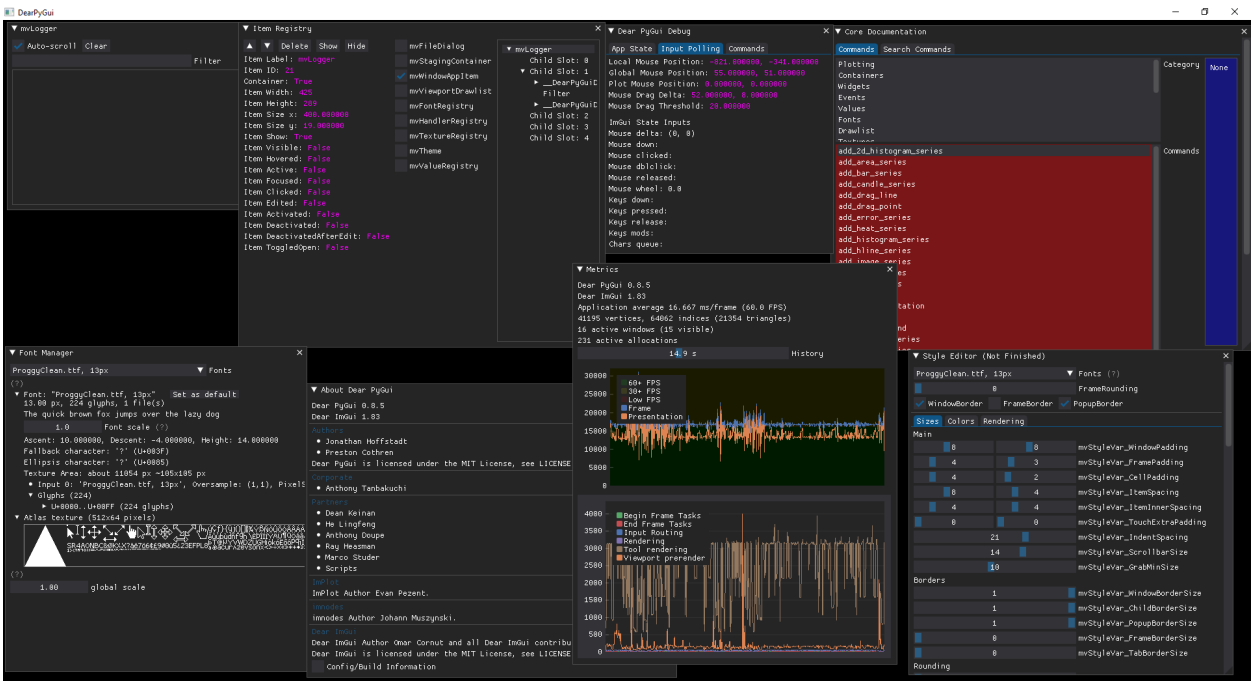
2.4.1 Developer Tools

DPG contains several tools which can help debug applications.

Code:

```
import dearpygui.dearpygui as dpg
import dearpygui.logger as dpg_logger

logger = dpg_logger.mvLogger()

dpg.show_documentation()
dpg.show_style_editor()
dpg.show_debug()
dpg.show_about()
dpg.show_metrics()
dpg.show_font_manager()
dpg.show_item_registry()

dpg.start_dearpygui()
```

Results



2.4.2 More Resources

- *Showcase*
- *Video Tutorials*
- here

CHAPTER

THREE

DOCUMENTATION

Live Demo: A mostly complete showcase of DPG can be found by running the `show_demo` command in the `dearpygui.demo` module.

Internal Documentation: Run `show_documentation`

API Reference Guide: *Online API Reference*

3.1 Render Loop

For most use cases the render loop does not need to be considered and is completely handled by :py:func:`start_dearpygui <dearpygui.dearpygui.start_dearpygui>.

3.1.1 Manual Render Loop

For more advanced use cases full access to the render loop can be accessed like so:

```
import dearpygui.dearpygui as dpg

dpg.setup_viewport()

with dpg.window(label="Example Window", width=500, height=150):
    dpg.add_text("Hello, world")


# below replaces, start_dearpygui()

while dpg.is_dearpygui_running():
    # you can manually stop by using stop_dearpygui()
    dpg.render_dearpygui_frame()

dpg.cleanup_dearpygui()
```

3.2 Viewport

The viewport is what you traditionally call the window in other GUI libraries. In the case of Dear PyGui, we distinguish between the operating system window and the Dear PyGui window by referring to the former as _the *viewport*.

Before calling `start_dearpygui`, you must do the following: 1. Create a viewport, using `create_viewport`. 2. Assign the viewport, using `setup_dearpygui`. 3. Show the viewport, using `show_viewport`.

As a convenience, we offer the command, `setup_viewport` which handles the above setup steps.

Once the viewport has been created, you can begin configuring the viewport using `configure_viewport` or the helper commands *set_viewport_**.

Note: Large/small icon must be set before showing the viewport (i.e. you will need to setup the viewport manually).

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Example Window", width=300):
    dpg.add_text("Hello, world")
    dpg.add_input_text(label='Viewport Title', callback=lambda sender, value: dpg.set_
↪viewport_title(title=value))

dpg.setup_viewport()
dpg.set_viewport_title(title='Custom Title')
dpg.set_viewport_width(500)
dpg.set_viewport_height(200)

dpg.start_dearpygui()
```

3.2.1 Manual Viewport

A more general approach to viewport creation and modification can be seen below. This is currently required if you need to set the icons.

```
import dearpygui.dearpygui as dpg

vp = dpg.create_viewport(title='Custom Title', width=600, height=200) # create viewport␣
↪takes in config options too!

# must be called before showing viewport
dpg.set_viewport_small_icon("path/to/icon.ico")
dpg.set_viewport_large_icon("path/to/icon.ico")

with dpg.window(label="Example Window", width=500, height=150):
    dpg.add_text("Hello, world")

dpg.setup_dearpygui(viewport=vp)
dpg.show_viewport(vp)
dpg.start_dearpygui()
```

3.3 Primary Window

The primary window fills the viewport and resizes with it.

It will also always remain in the background of other windows.

A window can be set as the primary window by using the `set_primary_window` command using the required True/False allows the window to be set or unset.

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial", id="main_window"):
    dpg.add_checkbox(label="Checkbox")

dpg.set_primary_window("main_window", True)
dpg.start_dearpygui()
```

3.4 IO, Handlers, State Polling

3.4.1 Handlers

Handlers are items that listen for certain events then submit a callback when that event occurs. Handlers can be activated or deactivated by showing or hiding them.

3.4.2 UI Item Handlers

UI item handlers listen for events related to a UI item

Events:

- Activated
- Active
- Clicked
- Deactivated
- Deactivated After Edited
- Focus
- Hover
- Resize
- Toggled
- Visible

```
import dearpygui.dearpygui as dpg

def change_text(sender, app_data):
    dpg.set_value("text_item", f"Mouse Button ID: {app_data}")

with dpg.window(width=500, height=300):
```

(continues on next page)

(continued from previous page)

```
    dpg.add_text("Click me with any mouse button", id="text_item")
    dpg.add_clicked_handler("text_item", callback=change_text)

dpg.start_dearpygui()
```

3.4.3 Global Handlers (IO Input)

Global handlers listen for events related to the viewport such as:

Keys:

- Down
- Press
- Release

Mouse:

- Click
- Double Click
- Down
- Drag
- Move
- Release
- Wheel

Global handlers are required to be added to a handler registry. Registries provide a grouping aspect to handlers allowing separation by input device.

For example this can allow a mouse registry or a keyboard registry ect. Registries also give the ability to deactivate all their children handlers by simply turning off the show keyword in the registry.

```
import dearpygui.dearpygui as dpg

def change_text(sender, app_data):
    dpg.set_value("text_item", f"Mouse Button: {app_data[0]}, Down Time: {app_data[1]}␣
↪seconds")

with dpg.window(width=500, height=300):
    dpg.add_text("Press any mouse button", id="text_item")
    with dpg.handler_registry():
        dpg.add_mouse_down_handler(callback=change_text)

dpg.start_dearpygui()
```

However if you call `setup_registries` before using global handlers this will allow the creation of global handlers anywhere.

This is because setup_registries creates a default registry that handlers can fall back to.

```
import dearpygui.dearpygui as dpg

dpg.setup_registries()

def change_text(sender, app_data):
    dpg.set_value("text_item", f"Mouse Button: {app_data[0]}, Down Time: {app_data[1]}␣
↪seconds")

with dpg.window(width=500, height=300):
    dpg.add_text("Press any mouse button", id="text_item")
    dpg.add_mouse_down_handler(callback=change_text)

dpg.start_dearpygui()
```

3.4.4 Polling Item State

Polling item state is accessible through `get_item_state` or all the light wrappers provided. These can be very powerful when combined with handlers as shown below.

```
import dearpygui.dearpygui as dpg

dpg.setup_registries()

def change_text(sender, app_data):
    if dpg.is_item_hovered("text_item"):
        dpg.set_value("text_item", f"Stop Hovering Me, Go away!!")
    else:
        dpg.set_value("text_item", f"Hover Me!")

with dpg.window(width=500, height=300):
    dpg.add_text("Hover Me!", id="text_item")
    dpg.add_mouse_move_handler(callback=change_text)

dpg.start_dearpygui()
```

3.5 Item Creation

DearPyGui can be broken down into **Items**, **UI Items**, **Containers**

3.5.1 Items

Everything created in _Dear_PyGui_ is an **item**. New items can be created by calling various *add_*** or *draw_*** functions. These commands return a unique identifier that can be used to later refer to the item. **UI items** and **containers** are also **items** - but not every **item** is necessarily a **UI item** or **container**. Most items must be parented by another item.

All items have the following optional parameters: *label*, *id*, *user_data*, and *use_internal_label*. The *id* is generated automatically and normally does not need to be included. A *label* serves as the display name for an item, while *user_data* can be any value and is frequently used for **callbacks**.

3.5.2 Containers

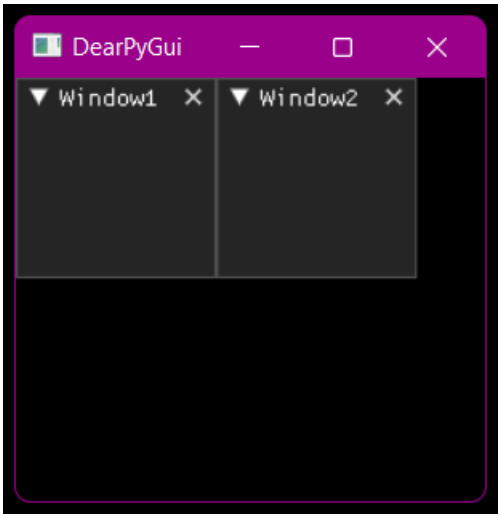
Container items are used to parent and store other items (including other containers).

In addition to creating them by calling their corresponding *add_*** function, they can also be created by calling their context manager function.

Note: Containers are more useful (and recommended) when used as context managers.

Below is an example of creating two new **window** items using their context manager function and starting the application:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Window1", pos=(0,0)) as window1:
    ...

with dpg.window(label="Window2", pos=(100,0)) as window2:
    ...

dpg.start_dearpygui()
```

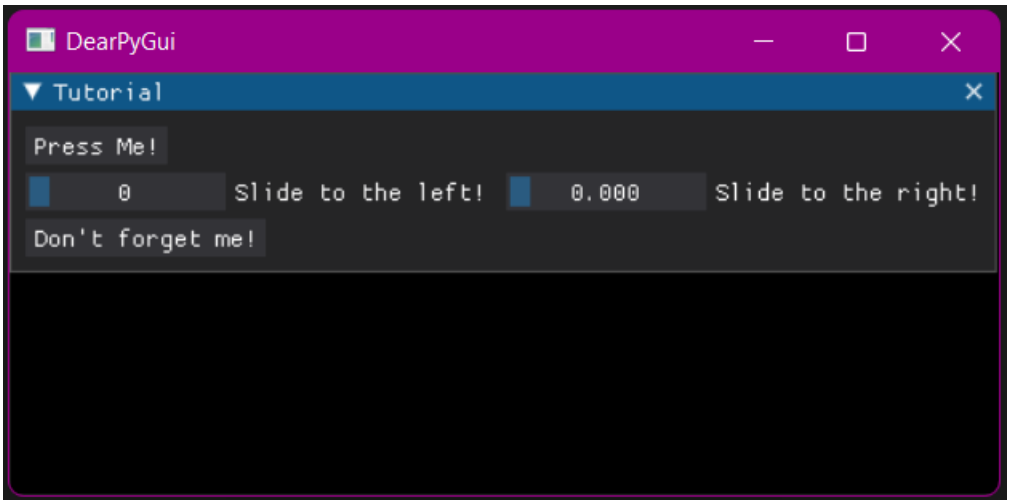


3.5.3 UI Items

UI items are items that are considered to be a visual (and usually interactable) element in your user interface. These include **buttons**, **sliders**, **inputs**, and even other containers such as **windows** and **tree nodes**.

Below is an example for creating a **window** container that parents a few other items:

```
import dearpygui.dearpygui as dpg


with dpg.window(label="Tutorial") as window:
    # When creating items within the scope of the context
    # manager, they are automatically "parented" by the
    # container created in the initial call. So, "window"
    # will be the parent for all of these items.

    button1 = dpg.add_button(label="Press Me!")

    slider_int = dpg.add_slider_int(label="Slide to the left!",width=100)
    dpg.add_same_line(spacing=10)
    slider_float = dpg.add_slider_float(label="Slide to the right!",width=100)

    # An item's unique identifier (id) is returned when
    # creating items.
    print(f"Printing item id's: {window}, {button1}, {slider_int}, {slider_float}")

# If you want to add an item to an existing container, you
# can specify which by passing the container's id as the
# "parent" parameter.
button2=dpg.add_button(label="Don't forget me!", parent=window)

dpg.start_dearpygui()
```



3.5.4 Other Items (i.e. everything else)

Event **handlers**, **registries**, **group**, **same_line**, and **themes** are also items. These are under-the-hood items for customizing the functionality, flow, and overall look of your interface.

3.6 Item Creation (Runtime)

With DPG you can dynamically add and delete any items at runtime.

This can be done by using a callback to run the desired item's *add_*** command and specifying the parent the item will belong to.

By using the **before** keyword when adding a item you can control which item in the parent the new item will come before. Default will place the new widget at the end.

3.6.1 Adding and Deleting Items

Below is an example demonstrating adding and deleting app items during runtime:

```
import dearpygui.dearpygui as dpg

def add_buttons():
    global new_button1, new_button2
    new_button1 = dpg.add_button(label="New Button", before="delete_button", id="new_
↪button1")
    new_button2 = dpg.add_button(label="New Button 2", parent="secondary_window", id=
↪"new_button2")

def delete_buttons():
    dpg.delete_item("new_button1")
    dpg.delete_item("new_button2")


with dpg.window(label="Tutorial", pos=(200, 200)):
```

(continues on next page)

(continued from previous page)

```
    dpg.add_button(label="Add Buttons", callback=add_buttons)
    dpg.add_button(label="Delete Buttons", callback=delete_buttons, id="delete_button")

with dpg.window(label="Secondary Window", id="secondary_window", pos=(100, 100)):
    pass

dpg.start_dearpygui()
```

3.6.2 Deleting Only Children

When deleting a container the container and its' children are deleted by default, unless the keyword **children_only** is set to True, i.e.:

```
import dearpygui.dearpygui as dpg

def delete_children():
    dpg.delete_item("window", children_only=True)

with dpg.window(label="Tutorial", pos=(200, 200), id="window"):
    dpg.add_button(label="Delete Children", callback=delete_children)
    dpg.add_button(label="Button_1")
    dpg.add_button(label="Button_2")
    dpg.add_button(label="Button_3")

dpg.start_dearpygui()
```

3.7 Item ID System

In DPG, all items must have an associated unique ID (UUID). Id's allow for modifications of the item at runtime.

When a widget is created, an ID is generated for you automatically. It is your responsibility to store this ID if you intend on interacting with the widget at a later time.

```
import dearpygui.dearpygui as dpg

unique_id = 0

def callback():
    print(dpg.get_value(unique_id))

with dpg.window(label="Example"):

    dpg.add_button(label="Press me", callback=callback)
    unique_id = dpg.add_input_int(label="Input")

dpg.start_dearpygui()
```

3.7.1 Generated IDs

The previous example could also be handled by generating the id beforehand like this:

```
import dearpygui.dearpygui as dpg

unique_id = dpg.generate_uuid()

def callback():
    print(dpg.get_value(unique_id))

with dpg.window(label="Example"):

    dpg.add_button(label="Press me", callback=callback)
    dpg.add_input_int(label="Input", id=unique_id)

dpg.start_dearpygui()
```

3.7.2 Aliases (added in 0.8.62)

An alias is a string that takes the place of the regular **int** ID. Aliases can be used anywhere UUID's can be used. It is the user's responsibility to make sure aliases are unique. For more details, see [Aliases](https://github.com/hoffstadt/DearPyGui/wiki/Aliases). A simple example can be seen below:

```
import dearpygui.dearpygui as dpg

def callback():
    print(dpg.get_value("unique_id"))

with dpg.window(label="Example"):

    dpg.add_button(label="Press me", callback=callback)
    dpg.add_input_int(label="Input", id="unique_id")

dpg.start_dearpygui()
```

3.7.3 Recent IDs

The most recent ID is stored for the last item, container, and root:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Example"):

    with dpg.group():
        dpg.add_button(label="Press me")
        print(dpg.last_item())
        print(dpg.last_container())
        print(dpg.last_root())

dpg.start_dearpygui()
```

3.8 Item Configuration

In DPG various configuration options and flags can be set when items are created. There are several options common to all items (i.e. **show**) but most UI items have specific options.

In order to modify an item's configuration after being created, you can use the **|configure_item|** command in conjunction with the keyword from the item's *add_*** command. You can also retrieve an item's configuration in the form of a dictionary by using the **|get_item_configuration|** command.

3.8.1 Example

Simple usage can be found below:

```
add_button(enabled=True, label="Press me", id="item")

# at a later time, change the item's configuration
configure_item("item", enabled=False, label="New Label")
```

3.9 Item Callbacks

Most UI items have a callback which is submitted to a **queue of callbacks** when the item is interacted with.

Callbacks are used to give functionality to items. Callbacks can either be assigned to the item upon creation or after creation using `set_item_callback` as shown in the code below.

Callbacks in DPG take **sender**, **app_data**, **user_data** arguments.

3.9.1 Sender, App_data

sender: argument is used by DPG to inform the callback which item triggered the callback by sending the id.

app_data: argument is used DPG to send information to the callback. This includes the current value of most basic widgets.

```
import dearpygui.dearpygui as dpg

def button_callback(sender, app_data):
    print(f"sender is: {sender}")
    print(f"app_data is: {app_data}")

with dpg.window(label="Tutorial"):

    dpg.add_button(label="Apply", callback=button_callback)

dpg.start_dearpygui()
```

3.9.2 User Data

app_data: argument is **Optionally** used to pass your own python object into the function.

The python object can be assigned or updated to the keyword *user_data* when the item is created or after the item is created using `set_item_user_data`

User data can be any python object.

```
import dearpygui.dearpygui as dpg

def button_callback(sender, app_data, user_data):
    print(f"sender is: {sender}")
    print(f"app_data is: {app_data}")
    print(f"user_data is: {user_data}")

with dpg.window(label="Tutorial"):

    #user data set when button is created
    dpg.add_button(label="Apply", callback=button_callback, user_data="Some Data")

    #user data and callback set any time after button has been created
    btn = dpg.add_button(label="Apply 2", )
    dpg.set_item_callback(btn, button_callback)
    dpg.set_item_user_data(btn, "Some Extra User Data")

dpg.start_dearpygui()
```

3.10 Item Value

When a widget is added, it creates an associated value by default. Values can be shared between widgets with the same underlying value type. This is accomplished by using the *source* keyword. One of the benefits of this is to have multiple widgets control the same value. Values are retrieved from the value `get_value`.

Values can be changed manually using `set_value`.

3.10.1 Value App Items

There are several "Value" items that can be used. These are widgets that have no visual component. These include:

- **mvBoolValue**
- **mvColorValue**
- **mvDoubleValue**
- **mvDouble4Value**
- **mvFloatValue**
- **mvFloat4Value**
- **mvFloatVectValue**
- **mvIntValue**
- **mvInt4Value**

- **mvSeriesValue**
- **mvStringValue**

Basic usage can be found below:

```
import dearpygui.dearpygui as dpg

with dpg.value_registry():
    bool_value = dpg.add_bool_value(default_value=True)
    string_value = dpg.add_string_value(default_value="Default string")

with dpg.window(label="Tutorial"):
    dpg.get_value("string_value")
    dpg.add_checkbox(label="Radio Button1", source=bool_value)
    dpg.add_checkbox(label="Radio Button2", source=bool_value)

    dpg.add_input_text(label="Text Input 1", source=string_value)
    dpg.add_input_text(label="Text Input 2", source=string_value, password=True)

dpg.start_dearpygui()
```

3.11 Containers & Context Managers

We have added context managers as helpers for most container items.

See also:

For more detail *Container Stack*

Core Command	Context Manager
add_table	**with table(...):**
add_table_row	**with table_row(...):**
add_window	**with window(...):**
add_menu_bar	**with menu_bar(...):**
add_child	**with child(...):**
add_clipper	**with clipper(...):**
add_collapsing_header	**with collapsing_header(...):**
add_colormap_registry	**with colormap_registry(...):**
add_group	**with group(...):**
add_node	**with node(...):**
add_node_attribute	**with node_attribute(...):**
add_node_editor	**with node_editor(...):**
add_staging_container	**with staging_container(...):**
add_tab_bar	**with tab_bar(...):**
add_tab	**with tab(...):**
add_tree_node	**with tree_node(...):**
add_tooltip	**with tooltip(...):**
add_popup	**with popup(...):**
add_drag_payload	**with payload(...):**
add_drawlist	**with drawlist(...):**
add_draw_layer	**with draw_layer(...):**

continues on next page

Table 1 – continued from previous page

add_viewport_drawlist	**with viewport_drawlist(...):**
add_file_dialog	**with file_dialog(...):**
add_filter_set	**with filter_set(...):**
add_font	**with font(...):**
add_font_registry	**with font_registry(...):**
add_handler_registry	**with handler_registry(...):**
add_plot	**with plot(...):**
add_subplots	**with subplots(...):**
add_texture_registry	**with texture_registry(...):**
add_value_registry	**with value_registry(...):**
add_theme	**with theme(...):**
add_item_pool	**with item_pool(...):**
add_template_registry	**with template_registry(...):**

3.11.1 Benefits:

1. Automatically push the container to the container stack.
2. Automatically pop the container off the container stack.
3. They make the code more readable and structured.

3.11.2 Context Managers:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Main"):

    with dpg.menu_bar():

        with dpg.menu(label="Themes"):
            dpg.add_menu_item(label="Dark")
            dpg.add_menu_item(label="Light")
            dpg.add_menu_item(label="Classic")

            with dpg.menu(label="Other Themes"):
                dpg.add_menu_item(label="Purple")
                dpg.add_menu_item(label="Gold")
                dpg.add_menu_item(label="Red")

        with dpg.menu(label="Tools"):
            dpg.add_menu_item(label="Show Logger")
            dpg.add_menu_item(label="Show About")

        with dpg.menu(label="Oddities"):
            dpg.add_button(label="A Button")
            dpg.add_simple_plot(label="Menu plot", (0.3, 0.9, 2.5, 8.9), height = 80)

dpg.start_dearpygui()
```

3.11.3 Explicit Parental Assignment (using UUIDs):

```
import dearpygui.dearpygui as dpg

w = dpg.add_window(label="Main")

mb = dpg.add_menu_bar(parent=w)

themes = dpg.add_menu("Themes", parent=mb)
dpg.add_menu_item(label="Dark", parent=themes)
dpg.add_menu_item(label="Light", parent=themes)

other_themes = dpg.add_menu("Other Themes", parent=themes)
dpg.add_menu_item(label="Purple", parent=other_themes)
dpg.add_menu_item(label="Gold", parent=other_themes)
add_menu_item(label="Red", parent=other_themes)

tools = dpg.add_menu(label="Tools", parent=mb)
dpg.add_menu_item(label="Show Logger", parent=tools)
dpg.add_menu_item(label="Show About", parent=tools)

oddities = dpg.add_menu(label="Oddities")
dpg.add_button(label="A Button", parent=oddities)
dpg.add_simple_plot(label="A menu plot", (0.3, 0.9, 2.5, 8.9), height=80,␣
↪parent=oddities)

dpg.start_dearpygui()
```

3.11.4 Explicit Parental Assignment (using aliases):

```
import dearpygui.dearpygui as dpg

dpg.add_window(label="Main", id="w")

dpg.add_menu_bar(parent=w, id="mb")

dpg.add_menu("Themes", parent="mb", id="themes")
dpg.add_menu_item(label="Dark", parent="themes")
dpg.add_menu_item(label="Light", parent="themes")

dpg.add_menu("Other Themes", parent="themes", id="other_themes")
dpg.add_menu_item(label="Purple", parent="other_themes")
dpg.add_menu_item(label="Gold", parent="other_themes")
add_menu_item(label="Red", parent="other_themes")

dpg.add_menu(label="Tools", parent="mb", id="tools")
dpg.add_menu_item(label="Show Logger", parent="tools")
dpg.add_menu_item(label="Show About", parent="tools")

dpg.add_menu(label="Oddities", id="Oddities")
dpg.add_button(label="A Button", parent="Oddities")
```

(continues on next page)

(continued from previous page)

```
dpg.add_simple_plot(label="A menu plot", (0.3, 0.9, 2.5, 8.9), height=80, parent=
↪"Oddities")

dpg.start_dearpygui()
```

3.11.5 Container Stack Operations:

```
import dearpygui.dearpygui as dpg

dpg.push_container_stack(dpg.add_window(label="Main"))

dpg.push_container_stack(dpg.add_menu_bar())

dpg.push_container_stack(dpg.add_menu(label="Themes"))
dpg.add_menu_item(label="Dark")
dpg.add_menu_item(label="Light")
dpg.pop_container_stack()

dpg.push_container_stack(dpg.add_menu(label="Tools"))
dpg.add_menu_item(label="Show Logger")
dpg.add_menu_item(label="Show About")
dpg.pop_container_stack()

# remove menu_bar from container stack
dpg.pop_container_stack()

# remove window from container stack
dpg.pop_container_stack()

dpg.start_dearpygui()
```

3.12 Container Slots & Children

Most app items can have child app items. App items can only be children to valid **container** items with the exception of item event handlers, which can belong to non-container app items. Some related commands can be found below:

is_item_container checks if an item is a container type

get_item_slot returns the item's target slot

get_item_parent returns the item's parent's UUID

get_item_children returns an item's children

reorder_items reorders children in a single call

move_item_up moves an item within its slot

move_item_down moves an item within its slot

move_item moves an item anywhere

set_item_children unstaging a staging container

3.12.1 Slots

App items are stored in target slots within their parent container. Below is the breakdown of slots:

Slot 0: **mvFileExtension**, **mvFontRangeHint**, **mvNodeLink**, **mvAnnotation**, **mvDragLine**, **mvDragPoint**, **mvLegend**, **mvTableColumn**

Slot 1: All other app items

Slot 2: Draw items

Slot 3: item event handlers

Slot 4: **mvDragPayload**

To query what slot an item belongs to, use `get_item_slot`.

3.12.2 Basic Example

Below is a simple example that demonstrates some of the above:

```
import dearpygui.dearpygui as dpg

dpg.setup_registries()

with dpg.window(label="about"):
    dpg.add_button(label="Press me")
    dpg.draw_line((10, 10), (100, 100), color=(255, 0, 0, 255), thickness=1)

# print children
print(dpg.get_item_children(dpg.last_root()))

# print children in slot 1
print(dpg.get_item_children(dpg.last_root(), 1))

# check draw_line's slot
print(dpg.get_item_slot(dpg.last_item()))


dpg.start_dearpygui()
```

Note: Use the *slot* keyword with `get_item_children` to return just a specific slot.

3.13 Container Stack

Unless an item is a root type or staging mode is active, all app items need to belong to a valid container item. An item's parent is deduced through the following process:

1. If item is a root, no parent needed; finished.
2. Check *before* keyword, if used skip to 6 using parent of "before" item.
3. Check *parent* keyword, if used skip to 6.
4. Check container stack, if used skip to 6.

5. If parent is not deduced and staging is active, add item to staging; finished.
6. Check if parent is compatible.
7. Check if parent accepts.
8. If runtime, add item using runtime methods; finished.
9. If startup, add item using startup methods; finished.

Container items can be manually pushed onto the container stack using `push_container_stack` and popped off using `pop_container_stack`.

This process is automated when using *Containers & Context Managers*. Below is a simple example demonstrating manual stack operations:

```
import dearpygui.dearpygui as dpg

dpg.push_container_stack(dpg.add_window(label="Tutorial"))

dpg.push_container_stack(dpg.add_menu_bar())

dpg.push_container_stack(dpg.add_menu(label="Themes"))
dpg.add_menu_item(label="Dark")
dpg.add_menu_item(label="Light")
dpg.pop_container_stack()

# remove menu_bar from container stack
dpg.pop_container_stack()

# remove window from container stack
dpg.pop_container_stack()

dpg.start_dearpygui()
```

3.13.1 Explicit Parental Assignment

Parents can be explicitly assigned using the *parent* keyword. This is most often used for adding new items at runtime. The above example can be shown again below using explicit parent assignment:

```
import dearpygui.dearpygui as dpg

dpg.add_window(label="Tutorial", id="window")

dpg.add_menu_bar(parent="window", id="menu_bar")

dpg.add_menu("Themes", parent="menu_bar", id="themes")
dpg.add_menu_item(label="Dark", parent="themes")
dpg.add_menu_item(label="Light", parent="themes")

dpg.start_dearpygui()
```

3.13.2 Context Managers

Context managers can be used to simplify the above example. All the context managers can be found [here](https://github.com/hoffstadt/DearPyGui/wiki/Context-Managers) but a simple example can be found below:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):

    with dpg.menu_bar():

        with dpg.menu(label="Themes"):
            dpg.add_menu_item(label="Dark")
            dpg.add_menu_item(label="Light")
            dpg.add_menu_item(label="Classic")

dpg.start_dearpygui()
```

Benefits 1. Automatically push the container to the container stack. 2. Automatically pop the container off the container stack. 3. They make the code more readable and structured.

3.14 Drawing-API

DPG has a low level drawing API that is well suited for primitive drawing, custom widgets or even dynamic drawings.

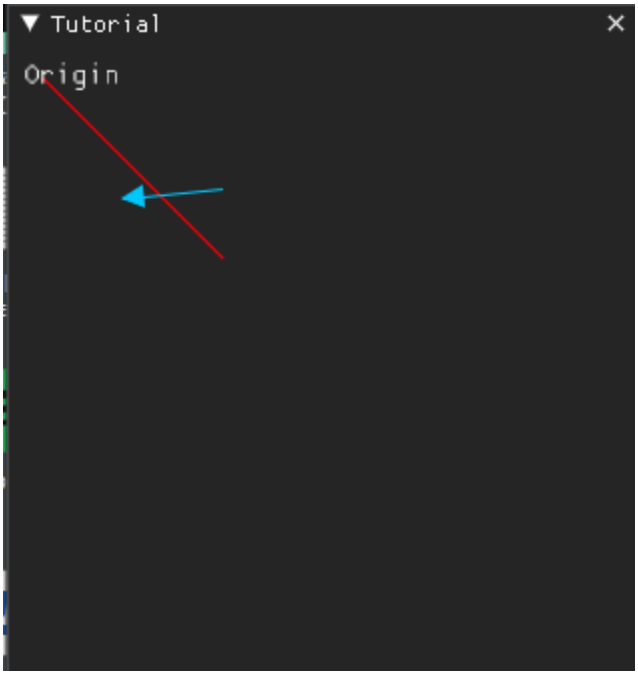
Drawing commands can be used in containers like drawlist, viewport_drawlist, or a window.

A drawlist widget is created by calling `add_drawlist` then items can be added by calling their respective draw commands. The origin for the drawing is in the top left and the y axis points down.

Code

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):

    with dpg.drawlist(width=300, height=300): # or you could use dpg.add_drawlist and␣
↪set parents manually

        dpg.draw_line((10, 10), (100, 100), color=(255, 0, 0, 255), thickness=1)
        dpg.draw_text((0, 0), "Origin", color=(250, 250, 250, 255), size=15)
        dpg.draw_arrow((50, 70), (100, 65), color=(0, 200, 255), thickness=1, size=10)

dpg.start_dearpygui()
```

Results



3.14.1 Images

Drawlists can display any textures including images of types PNG, JPEG, or BMP (See *Textures* for more detail). Images are drawn using draw_image.

Using the keywords **pmin** and **pmax** we can define the upper left and lower right area of the rectangle that the image will be draw onto the canvas. The image will scale to fit the specified area.

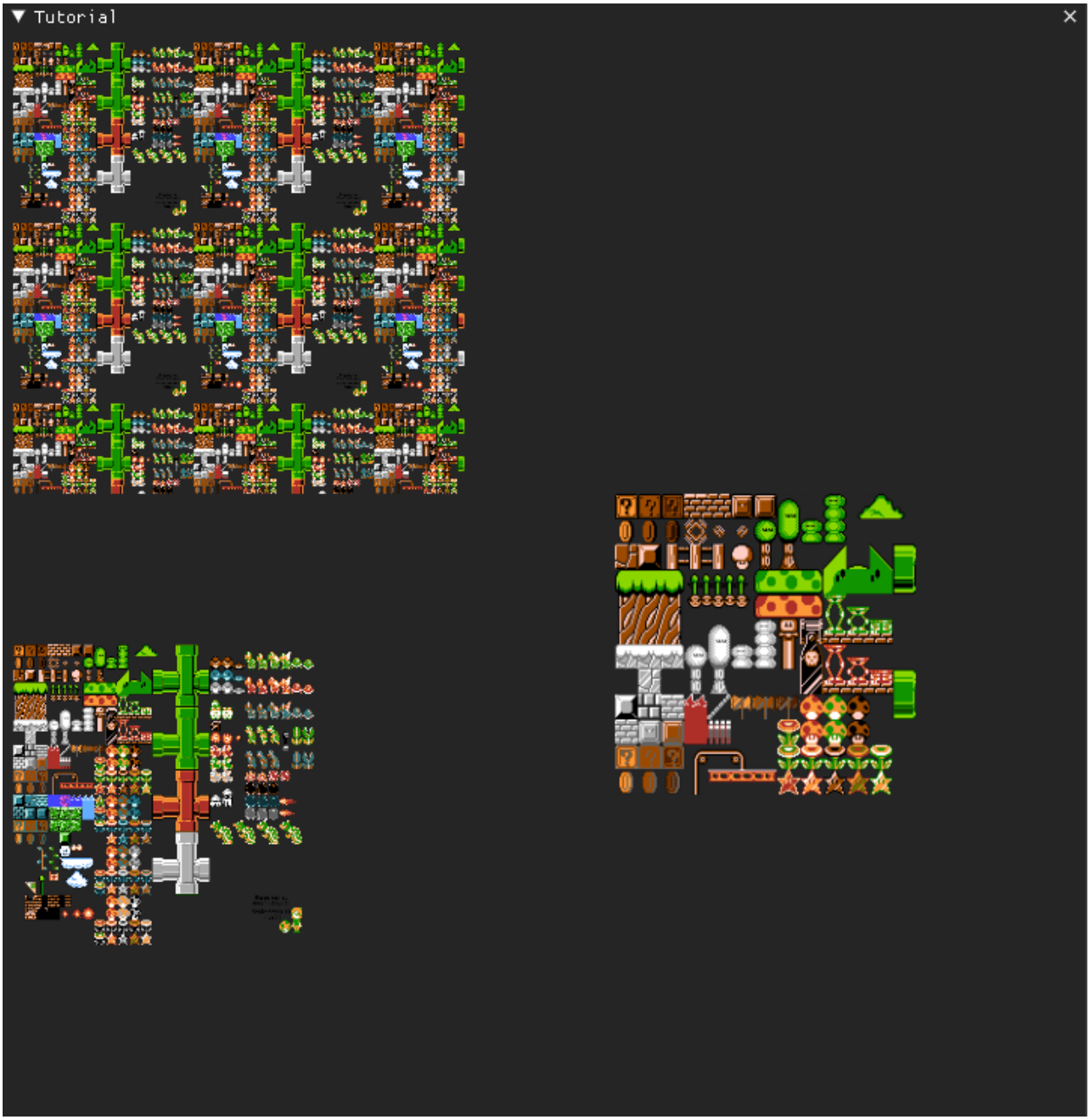
With keywords **uv_min** and **uv_max** we can define a scalar number of what area on the image should be drawn to the canvas. The default of uv_min = [0,0] and uv_max = [1,1] will display the entire image while uv_min = [0,0] uv_max = [0.5,0.5] will only show the first quarter of the drawing.

To be able to demonstrate these features you must update the directory to that of an image on your computer, such as SpriteMapExample.png.

Code

```
import dearpygui.dearpygui as dpg

width, height, channels, data = dpg.load_image('SpriteMapExample.png') # 0: width, 1: height, 2: channels, 3: data

with dpg.texture_registry():  dpg.add_static_texture(width, height, data, id="image_id")

with dpg.window(label="Tutorial"):

    with dpg.drawlist(width=700, height=700):

        dpg.draw_image("image_id", (0, 400), (200, 600), uv_min=(0, 0), uv_max=(1, 1))
        dpg.draw_image("image_id", (400, 300), (600, 500), uv_min=(0, 0), uv_max=(0.5, 0.5))
        dpg.draw_image("image_id", (0, 0), (300, 300), uv_min=(0, 0), uv_max=(2.5, 2.5))

dpg.start_dearpygui()
```

Results



3.14.2 Viewport and Window

You can also use all the same draw_* drawings commands with a window as the parent. Similarly you can draw to the viewport foreground or background by using a viewport_drawlist.

Code

```
import dearpygui.dearpygui as dpg

# creating font and back viewport drawlists
dpg.add_viewport_drawlist(id="viewport_front")
dpg.add_viewport_drawlist(front=False, id="viewport_back")
```

(continues on next page)

(continued from previous page)

```
with dpg.window(label="Tutorial", width=300, height=300):
    dpg.add_text("Move the window over the drawings to see the effects.", wrap=300)
    dpg.draw_circle((100, 100), 25, color=(255, 255, 255, 255))
    dpg.draw_circle((100, 100), 25, color=(255, 255, 255, 255), parent="viewport_front")
    dpg.draw_circle((200, 200), 25, color=(255, 255, 255, 255), parent="viewport_back")

dpg.start_dearpygui()
```

Results

3.15 File & Directory Selector

The file dialog widget can be used to select a single file, multiple files, or a directory. When the user clicks the **Ok** button, the dialog's callback is ran. Information is passed through the app_data argument. The simplest case is as a director picker. Below is the example

Code

```
import dearpygui.dearpygui as dpg

def callback(sender, app_data):
    print("Sender: ", sender)
    print("App Data: ", app_data)

dpg.add_file_dialog(directory_selector=True, show=False, callback=callback, id="file_
↪dialog_id")

with dpg.window(label="Tutorial", width=800, height=300):
    dpg.add_button(label="Directory Selector", callback=lambda: dpg.show_item("file_
↪dialog_id"))

dpg.start_dearpygui()
```

Note: If no file extensions have been added, the selector defaults to directories.

3.15.1 File Extensions

File extensions are app items that are added to the file dialog. You can even set the color of the file extensions. Below is a simple example

```
import dearpygui.dearpygui as dpg

def callback(sender, app_data, user_data):
    print("Sender: ", sender)
    print("App Data: ", app_data)
```

(continues on next page)

(continued from previous page)

```
with dpg.file_dialog(directory_selector=False, show=False, callback=callback, id="file_
↪dialog_id"):
    dpg.add_file_extension(".*", color=(255, 255, 255, 255))
    dpg.add_file_extension("Source files (*.cpp *.h *.hpp){.cpp,.h,.hpp}", color=(0, 255,
↪ 255, 255))
    dpg.add_file_extension(".cpp", color=(255, 255, 0, 255))
    dpg.add_file_extension(".h", color=(255, 0, 255, 255), custom_text="header")
    dpg.add_file_extension("Python(.py){.py}", color=(0, 255, 0, 255))

with dpg.window(label="Tutorial", width=800, height=300):
    dpg.add_button(label="File Selector", callback=lambda: dpg.show_item("file_dialog_id
↪"))

dpg.start_dearpygui()
```

3.15.2 Customizing

File dialogs can be customized with a panel by just adding app items to the file dialog as if it were a regular container. Below is an example

```
import dearpygui.dearpygui as dpg

def callback(sender, app_data):
    print("Sender: ", sender)
    print("App Data: ", app_data)

with dpg.file_dialog(directory_selector=False, show=False, callback=callback, id="file_
↪dialog_id"):
    dpg.add_file_extension(".*", color=(255, 255, 255, 255))
    dpg.add_file_extension(".cpp", color=(255, 255, 0, 255))
    dpg.add_file_extension(".h", color=(255, 0, 255, 255))
    dpg.add_file_extension(".py", color=(0, 255, 0, 255))

    dpg.add_button(label="fancy file dialog")
    with dpg.child():
        dpg.add_selectable(label="bookmark 1")
        dpg.add_selectable(label="bookmark 2")
        dpg.add_selectable(label="bookmark 3")

with dpg.window(label="Tutorial", width=800, height=300):
    dpg.add_button(label="File Selector", callback=lambda: dpg.show_item("file_dialog_id
↪"))

dpg.start_dearpygui()
```

3.15.3 Selecting Multiple Files

You can select multiple files by setting the *file_count* keyword

```
import dearpygui.dearpygui as dpg

def callback(sender, app_data):
    print("Sender: ", sender)
    print("App Data: ", app_data)

with dpg.file_dialog(directory_selector=False, show=False, callback=callback, file_
↪count=3, id="file_dialog_id"):
    dpg.add_file_extension(".*", color=(255, 255, 255, 255))
    dpg.add_file_extension(".cpp", color=(255, 255, 0, 255))
    dpg.add_file_extension(".h", color=(255, 0, 255, 255))
    dpg.add_file_extension(".py", color=(0, 255, 0, 255))

    dpg.add_button(label="fancy file dialog")
    with dpg.child():
        dpg.add_selectable(label="bookmark 1")
        dpg.add_selectable(label="bookmark 2")
        dpg.add_selectable(label="bookmark 3")

with dpg.window(label="Tutorial", width=800, height=300):
    dpg.add_button(label="File Selector", callback=lambda: dpg.show_item("file_dialog_id
↪"))

dpg.start_dearpygui()
```

3.15.4 Bookmarks

Not ready yet.

3.16 Filter Set

The filter set app item is a container that can be used to filter its children based on their *filter_key*.

Most app items have a *filter_key* keyword that can be set when creating the item. This works by setting the value of the filter set.

The easiest way to understand this is by considering the example below

Code

```
import dearpygui.dearpygui as dpg

def callback(sender, filter_string):

    # set value of filter set
    dpg.set_value("filter_id", filter_string)

with dpg.window(label="about"):
```

(continues on next page)

(continued from previous page)

```
    dpg.add_input_text(label="Filter (inc, -exc)", callback=callback)
    with dpg.filter_set(id="filter_id"):
        dpg.add_text("aaa1.c", filter_key="aaa1.c", bullet=True)
        dpg.add_text("bbb1.c", filter_key="bbb1.c", bullet=True)
        dpg.add_text("ccc1.c", filter_key="ccc1.c", bullet=True)
        dpg.add_text("aaa2.cpp", filter_key="aaa2.cpp", bullet=True)
        dpg.add_text("bbb2.cpp", filter_key="bbb2.cpp", bullet=True)
        dpg.add_text("ccc2.cpp", filter_key="ccc2.cpp", bullet=True)
        dpg.add_text("abc.h", filter_key="abc.h", bullet=True)
        dpg.add_text("hello, world", filter_key="hello, world", bullet=True)

dpg.start_dearpygui()
```

3.16.1 Tips

- Display everything with *""*
- Display lines containing xxx with *"xxx"*
- Display lines containing xxx or yyy with *"xxx,yyy"*
- Hide lines containing xxx with *"-xxx"*

3.17 Fonts

Dear PyGui embeds a copy of 'ProggyClean.ttf' (by Tristan Grimmer), a 13 pixels high, pixel-perfect font used by default. ProggyClean does not scale smoothly, therefore it is recommended that you load your own file when using Dear PyGui in an application aiming to look nice and wanting to support multiple resolutions.

You do this by loading external .TTF/.OTF files. In the Resources folder you can find an example of a otf font.

3.17.1 Readme First

All loaded fonts glyphs are rendered into a single texture atlas ahead of time. Adding/Removing/Modifying fonts will cause the font atlas to be rebuilt.

You can use the style editor `show_documentation` from the *simple* module to browse your fonts and understand what's going on if you have an issue.

3.17.2 Font Loading Instructions

To add your own fonts, you must first create a font registry to add fonts to. Next, add fonts to the registry. By default only basic latin and latin supplement glyphs are added (0x0020 - 0x00FF).

```
import dearpygui.dearpygui as dpg

# add a font registry
with dpg.font_registry():
```

(continues on next page)

(continued from previous page)

```
    # add font (set as default for entire app)
    dpg.add_font("path/to/font/file/CoolFont.otf", 20, default_font=True)

    # add second font
    dpg.add_font("path/to/font/file/AnotherCoolFont.ttf", 13, id="secondary_font")

with dpg.window(label="Font Example"):
    dpg.add_button(label="Default font")
    dpg.add_button(label="Secondary font")

    # set font of specific widget
    dpg.set_item_font(dpg.last_item(), "secondary_font")

dpg.start_dearpygui()
```

3.17.3 Loading Specific Unicode Characters

There are several ways to add specific characters from a font file. You can use range hints, ranges, and specific characters. You can also remap characters.

```
import dearpygui.dearpygui as dpg

with dpg.font_registry():

    with dpg.font("path/to/font/file/CoolFont.otf", 20, default_font=True):

        # add the default font range
        dpg.add_font_range_hint(dpg.mvFontRangeHint_Default)

        # helper to add range of characters
        #    Options:
        #        mvFontRangeHint_Japanese
        #        mvFontRangeHint_Korean
        #        mvFontRangeHint_Chinese_Full
        #        mvFontRangeHint_Chinese_Simplified_Common
        #        mvFontRangeHint_Cyrillic
        #        mvFontRangeHint_Thai
        #        mvFontRangeHint_Vietnamese
        dpg.add_font_range_hint(dpg.mvFontRangeHint_Japanese)

        # add specific range of glyphs
        dpg.add_font_range(0x3100, 0x3ff0)

        # add specific glyphs
        dpg.add_font_chars([0x3105, 0x3107, 0x3108])

        # remap  to %
        dpg.add_char_remap(0x3084, 0x0025)

dpg.start_dearpygui()
```

3.17.4 Where to find unicode character codes?

Unicode Characters

3.18 Init Files

Init files are used to preserve the following data between application sessions

- window positions
- window sizes
- window collapse state
- window docking
- table column widths
- table column ordering
- table column visible state
- table column sorting state

Note: Init files use the ID of the window. Make sure the ID does not change between sessions by generating the ID beforehand.

3.18.1 Creating init files

There are two procedures for creating **init** files: 1. Use `save_init_file` while

your application is running.

2. **Temporarily add `set_init_file`** to your application before starting DPG.

Note: windows and tables can individually opt out of having their settings saved with the *no_saved_settings* keyword.

3.18.2 Loading init files

There are two procedures for loading **init** files: 1. Use `load_init_file` before starting DPG.

Note: Procedure 2 will overwrite the **init** file.

3.18.3 Basic Usage

Below is an example of using **init** files to preserve settings between sessions

```
import dearpygui.dearpygui as dpg

dpg.set_init_file()  # default file is 'dpg.ini'

with dpg.window(label="about"):
    dpg.add_button(label="Press me")

dpg.start_dearpygui()
```

3.19 Built-In Logger

Dear PyGui provides a built-in logger for easy logging. This widget resides in its own module which will need to be imported *dearpygui.logger*.

This logger has basic functionality

Log levels can be set by setting the member log level to a desired level. Any new log lower than the set level will not show. Avaliable levels are

- Trace = 0
- Debug = 1
- Info = 2
- Warning = 3
- Error = 4
- Critical = 5

Log can be cleared by setting the message count to zero *logger.count = 0*

Automatic log clearing will occur at message count set with the flush_count member this is important to keep your message list from using too much memory.

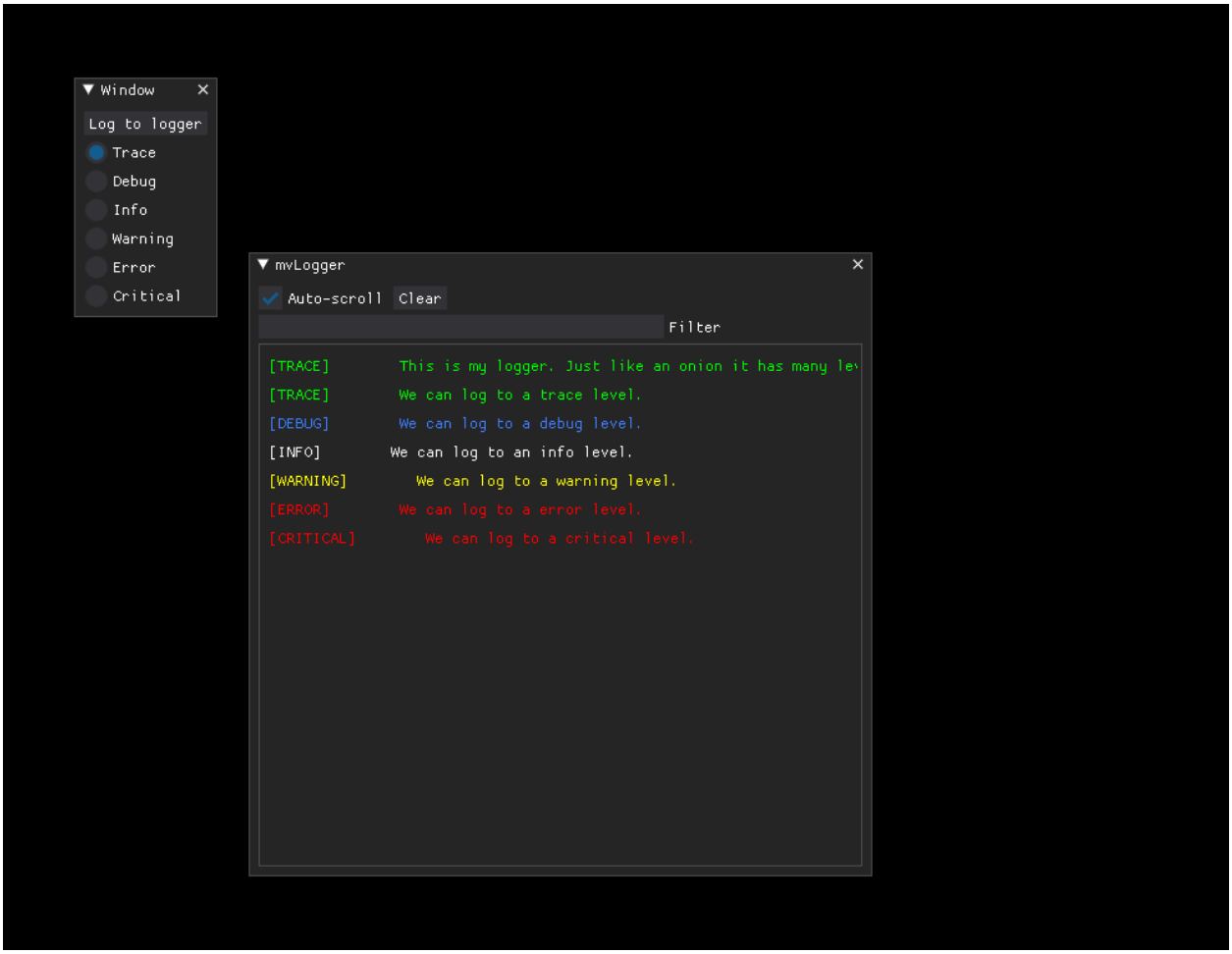
Programmatic filtering can be done setting the filter_id to a desired string.

```
import dearpygui.dearpygui as dpg
import dearpygui.logger as dpg_logger

logger = dpg_logger.mvLogger()
logger.log("This is my logger. Just like an onion it has many levels.")


def log_things(sender, app_data, user_data):
    user_data.log("We can log to a trace level.")
    user_data.log_debug("We can log to a debug level.")
    user_data.log_info("We can log to an info level.")
    user_data.log_warning("We can log to a warning level.")
    user_data.log_error("We can log to a error level.")
    user_data.log_critical("We can log to a critical level.")
```

(continues on next page)

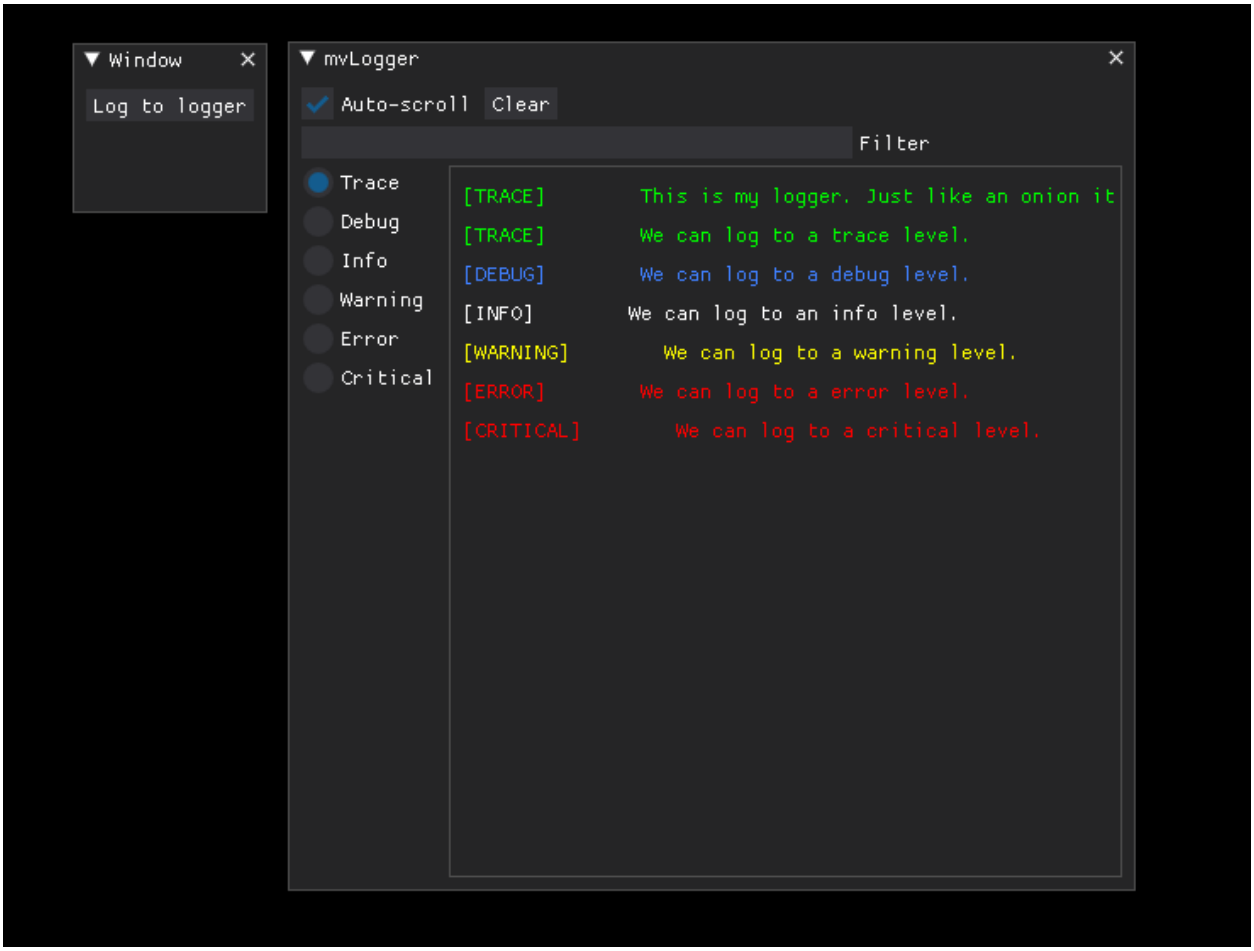
(continued from previous page)

```
def set_level(sender, app_data, user_data):
    # changing the logger level will ignore any log messages below the set level
    logger = user_data[0]
    level_options = user_data[1]
    logger.log_level = (level_options[dpg.get_value(sender)])

    # we do this so we can see the set level effect
    log_things(sender, app_data, logger)

with dpg.window():
    dpg.add_button(label="Log to logger", callback=log_things, user_data=logger)
    level_options = {"Trace": 0, "Debug": 1, "Info": 2,  "Warning": 3, "Error": 4,
↪"Critical": 5}
    dpg.add_radio_button(list(level_options.keys()), callback=set_level, user_
↪data=[logger, level_options])

dpg.start_dearpygui()
```



3.20 Custom Logger

The built-in logger is completely made from DPG which means you can make your own or even recreate the built in one by looking at the logger.py module when you pip install or you can see it on the in the logger.py



3.21 Menus

The menu bar consists of the following components

- Menu Bar - The main menu ribbon
- Menu - Drop down menus “sub-menus”
- Menu Item - Items that can run callbacks (basically selectables)
- Items are added to the Menu Bar from right to left. Items are added to the menus from top to bottom.

Menus can be nested as necessary.

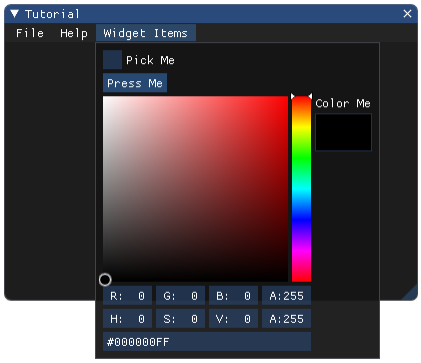
Any widget can be added to a menu.

3.22 Basic Usage

Code

```
import dearpygui.dearpygui as dpg

def print_me(sender):
    print(f"Menu Item: {sender}")

with dpg.window(label="Tutorial"):

    with dpg.menu_bar():

        with dpg.menu(label="File"):

            dpg.add_menu_item(label="Save", callback=print_me)
            dpg.add_menu_item(label="Save As", callback=print_me)

            with dpg.menu(label="Settings"):

                dpg.add_menu_item(label="Setting 1", callback=print_me)
                dpg.add_menu_item(label="Setting 2", callback=print_me)

        dpg.add_menu_item(label="Help", callback=print_me)

        with dpg.menu(label="Widget Items"):

            dpg.add_checkbox(label="Pick Me", callback=print_me)
            dpg.add_button(label="Press Me", callback=print_me)
            dpg.add_color_picker(label="Color Me", callback=print_me)

dpg.start_dearpygui()
```

Results



3.23 Node Editor

A Node Editor presents an editable schematic or graph, displaying nodes and the connections between their attributes. It allows you to view, modify, and create new node connections.

You can see an example below

There are 4 main components

1. **Node Editor** - the area in which the nodes are located
2. **Nodes** - the free floating "windows" which contains attributes
3. **Attributes - the collections of widgets with pins to create links to/from.** Can be input, output, or static.
4. **Links** - the connections between attributes

3.23.1 Regular usage

You must first create a node editor, followed by nodes which contains attributes.

Attributes can contain any UI Items. When a user attempts to link attributes, the node editor's callback is ran. DPG sends the attribute ID's through the _app_data_ argument of the callback. It is the developer's responsibility to create the link.

Below is a basic example. You can grab an output pin and connect it to an input pin. You can detach a link by **ctrl** clicking and dragging the link away.

Code

```
import dearpygui.dearpygui as dpg

# callback runs when user attempts to connect attributes
def link_callback(sender, app_data):
    # app_data -> (link_id1, link_id2)
    dpg.add_node_link(app_data[0], app_data[1], parent=sender)


# callback runs when user attempts to disconnect attributes
def delink_callback(sender, app_data):
    # app_data -> link_id
    dpg.delete_item(app_data)

with dpg.window(label="Tutorial", width=400, height=400):

    with dpg.node_editor(callback=link_callback, delink_callback=delink_callback):
        with dpg.node(label="Node 1"):
            with dpg.node_attribute(label="Node A1"):
                dpg.add_input_float(label="F1", width=150)

            with dpg.node_attribute(label="Node A2", attribute_type=dpg.mvNode_Attr_
↪Output):
                dpg.add_input_float(label="F2", width=150)

        with dpg.node(label="Node 2"):
```

(continues on next page)

(continued from previous page)

```
            with dpg.node_attribute(label="Node A3"):
                dpg.add_input_float(label="F3", width=200)

            with dpg.node_attribute(label="Node A4", attribute_type=dpg.mvNode_Attr_
↪Output):
                dpg.add_input_float(label="F4", width=200)

dpg.start_dearpygui()
```

3.23.2 Selection Querying

You can retrieve selected nodes and links (and clear this selections with the following commands)

```
dpg.get_selected_nodes(editor_id)
dpg.get_selected_links(editor_id)
dpg.clear_selected_nodes(editor_id)
dpg.clear_selected_links(editor_id)
```

3.23.3 Node Attribute Types

The following constants can be used in the *attribute_type* argument for node attributes

Attribute |
—- |
mvNode_Attr_Input (default) |
mvNode_Attr_Output |
mvNode_Attr_Static |

3.23.4 Node Attribute Pin Shapes

The following constants can be used in the *shape* argument for node attributes

Shape |
—- |
mvNode_PinShape_Circle |
mvNode_PinShape_CircleFilled (default) |
mvNode_PinShape_Triangle |
mvNode_PinShape_TriangleFilled |
mvNode_PinShape_Quad |
mvNode_PinShape_QuadFilled |

3.23.5 Associated App Items

- **mvNode**
- **mvNodeAttribute**
- **mvNodeLink**

3.24 Plots

Plots are composed of multiple components. These include plot axes, data series, and an optional legend. Below is the minimal example for creating a plot

```
import dearpygui.dearpygui as dpg
from math import sin

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i/100)
    sindatay.append(0.5 + 0.5*sin(50*i/100))

with dpg.window(label="Tutorial"):

    # create plot
    with dpg.plot(label="Line Series", height=400, width=400):

        # optionally create legend
        dpg.add_plot_legend()

        # REQUIRED: create x and y axes
        dpg.add_plot_axis(dpg.mvXAxis, label="x")
        dpg.add_plot_axis(dpg.mvYAxis, label="y", id="y_axis")

        # series belong to a y axis
        dpg.add_line_series(sindatax, sindatay, label="0.5 + 0.5 * sin(x)", parent="y_
↪axis")

dpg.start_dearpygui()
```

3.24.1 Tips

- Click & Drag: to pan the plot
- Click & Drag on Axis: to pan the plot in one direction
- Double Click: scales plot to data
- Right Click & Drag: to zoom to an area
- Double Right Click: opens settings
- Shift + Right Click & Drag: to zoom to an area that fills a current axis
- Scroll Mouse Wheel: zooms

- Scroll Mouse Wheel on Axis: zooms only that axis
- Toggle data sets on the legend to hide them

3.24.2 Updating Series

You can update the series on a plot by either adding a series using the same name or by clearing the plot. This is shown below:

```
import dearpygui.dearpygui as dpg
from math import sin, cos

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i/100)
    sindatay.append(0.5 + 0.5*sin(50*i/100))

def update_series():

    cosdatax = []
    cosdatay = []
    for i in range(0, 100):
        cosdatax.append(i/100)
        cosdatay.append(0.5 + 0.5*cos(50*i/100))
    dpg.set_value("series_id", [cosdatax, cosdatay])

with dpg.window(label="Tutorial"):

    dpg.add_button(label="Update Series", callback=update_series)

    with dpg.plot(label="Line Series", height=400, width=400):
        dpg.add_plot_axis(dpg.mvXAxis, label="x")
        dpg.add_plot_axis(dpg.mvYAxis, label="y")
        dpg.add_line_series(sindatax, sindatay, label="0.5 + 0.5 * sin(x)", parent=dpg.
↪last_item(), id="series_id")

dpg.start_dearpygui()
```

3.24.3 Axis Limits

The following commands can be used to control the plot axes limits

- *set_axis_limits(...)*
- *get_axis_limits(...)*
- *set_axis_limits_auto(...)*
- *fit_axis_data(...)*

An example demonstrating some of this can be found below:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial", width=400, height=400):

    with dpg.group(horizontal=True):
        dpg.add_button(label="fit y", callback=lambda: dpg.fit_axis_data("yaxis"))
        dpg.add_button(label="unlock x limits", callback=lambda: dpg.set_axis_limits_
↪auto("xaxis"))
        dpg.add_button(label="unlock y limits", callback=lambda: dpg.set_axis_limits_
↪auto("yaxis"))
        dpg.add_button(label="print limits x", callback=lambda: print(dpg.get_axis_
↪limits("xaxis")))
        dpg.add_button(label="print limits y", callback=lambda: print(dpg.get_axis_
↪limits("yaxis")))

    with dpg.plot(label="Bar Series", height=-1, width=-1):
        dpg.add_plot_legend()

        # create x axis
        dpg.add_plot_axis(dpg.mvXAxis, label="Student", no_gridlines=True, id="xaxis")
        dpg.set_axis_limits(dpg.last_item(), 9, 33)
        dpg.set_axis_ticks(dpg.last_item(), (("S1", 11), ("S2", 21), ("S3", 31)))

        # create y axis
        dpg.add_plot_axis(dpg.mvYAxis, label="Score", id="yaxis")
        dpg.set_axis_limits("yaxis", 0, 110)

        # add series to y axis
        dpg.add_bar_series([10, 20, 30], [100, 75, 90], label="Final Exam", weight=1,␣
↪parent="yaxis")
        dpg.add_bar_series([11, 21, 31], [83, 75, 72], label="Midterm Exam", weight=1,␣
↪parent="yaxis")
        dpg.add_bar_series([12, 22, 32], [42, 68, 23], label="Course Grade", weight=1,␣
↪parent="yaxis")

dpg.start_dearpygui()
```

3.24.4 Custom Axis Labels

Custom labels can be set per axis using `set_axis_ticks`. They can be reset with `reset_axis_ticks`. An example can be found below

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial", width=400, height=400):

    with dpg.plot(label="Bar Series", height=-1, width=-1):

        dpg.add_plot_legend()

        # create x axis
```

(continues on next page)

(continued from previous page)

```
        dpg.add_plot_axis(dpg.mvXAxis, label="Student", no_gridlines=True)
        dpg.set_axis_ticks(dpg.last_item(), (("S1", 11), ("S2", 21), ("S3", 31)))

        # create y axis
        dpg.add_plot_axis(dpg.mvYAxis, label="Score", id="yaxis_id")

        # add series to y axis
        dpg.add_bar_series([10, 20, 30], [100, 75, 90], label="Final Exam", weight=1,␣
↪parent="yaxis_id")
        dpg.add_bar_series([11, 21, 31], [83, 75, 72], label="Midterm Exam", weight=1,␣
↪parent="yaxis_id")
        dpg.add_bar_series([12, 22, 32], [42, 68, 23], label="Course Grade", weight=1,␣
↪parent="yaxis_id")

dpg.start_dearpygui()
```

3.24.5 Multiple Y Axes

In DPG you can have up to 3 Y axes. Below is an example

```
import dearpygui.dearpygui as dpg
from math import sin

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i / 100)
    sindatay.append(0.5 + 0.5 * sin(50 * i / 100))

with dpg.window(label="Tutorial", width=400, height=400):

    with dpg.plot(label="Multi Axes Plot", height=400, width=-1):
        dpg.add_plot_legend()

        # create x axis
        dpg.add_plot_axis(dpg.mvXAxis, label="x")

        # create y axis 1
        dpg.add_plot_axis(dpg.mvYAxis, label="y1")
        dpg.add_line_series(sindatax, sindatay, label="y1 lines", parent=dpg.last_item())

        # create y axis 2
        dpg.add_plot_axis(dpg.mvYAxis, label="y2")
        dpg.add_stem_series(sindatax, sindatay, label="y2 stem", parent=dpg.last_item())

        # create y axis 3
        dpg.add_plot_axis(dpg.mvYAxis, label="y3 scatter")
        dpg.add_scatter_series(sindatax, sindatay, label="y3", parent=dpg.last_item())

dpg.start_dearpygui()
```

3.24.6 Annotations

Annotations can be used to mark locations on a plot. They do NOT belong to an axis in the same manner that series do. They are owned by the plot. The coordinates correspond to the 1st y axis. They are clamped by default. Below is an example:

```
import dearpygui.dearpygui as dpg
from math import sin

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i/100)
    sindatay.append(0.5 + 0.5*sin(50*i/100))

with dpg.window(label="Tutorial", width=400, height=400):

    with dpg.plot(label="Annotations", height=-1, width=-1):

        dpg.add_plot_legend()
        dpg.add_plot_axis(dpg.mvXAxis, label="x")
        dpg.add_plot_axis(dpg.mvYAxis, label="y")
        dpg.add_line_series(sindatax, sindatay, label="0.5 + 0.5 * sin(x)", parent=dpg.
↪last_item())

        # annotations belong to the plot NOT axis
        dpg.add_plot_annotation(label="BL", default_value=(0.25, 0.25), offset=(-15, 15),
↪ color=[255, 255, 0, 255])
        dpg.add_plot_annotation(label="BR", default_value=(0.75, 0.25), offset=(15, 15),␣
↪color=[255, 255, 0, 255])
        dpg.add_plot_annotation(label="TR not clampled", default_value=(0.75, 0.75),␣
↪offset=(-15, -15), color=[255, 255, 0, 255], clamped=False)
        dpg.add_plot_annotation(label="TL", default_value=(0.25, 0.75), offset=(-15, -
↪15), color=[255, 255, 0, 255])
        dpg.add_plot_annotation(label="Center", default_value=(0.5, 0.5), color=[255,␣
↪255, 0, 255])

dpg.start_dearpygui()
```

3.24.7 Drag Points and Lines

Similar to annotations, drag lines/points belong to the plot and the values correspond to the 1st y axis. These items can be moved by clicking and dragging. You can also set a callback to be ran when they are interacted with! Below is a simple example

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial", width=400, height=400):

    with dpg.plot(label="Drag Lines/Points", height=-1, width=-1):
        dpg.add_plot_legend()
        dpg.add_plot_axis(dpg.mvXAxis, label="x")
```

(continues on next page)

(continued from previous page)

```
        dpg.set_axis_limits(dpg.last_item(), -5, 5)
        dpg.add_plot_axis(dpg.mvYAxis, label="y")
        dpg.set_axis_limits(dpg.last_item(), -5, 5)

        # drag lines/points belong to the plot NOT axis
        dpg.add_drag_line(label="dline1", color=[255, 0, 0, 255], default_value=2.0)
        dpg.add_drag_line(label="dline2", color=[255, 255, 0, 255], vertical=False,␣
↪default_value=-2)
        dpg.add_drag_point(label="dpoint1", color=[255, 0, 255, 255], default_value=(1.0,
↪ 1.0))
        dpg.add_drag_point(label="dpoint2", color=[255, 0, 255, 255], default_value=(-1.
↪0, 1.0))

dpg.start_dearpygui()
```

3.24.8 Querying

Querying allows the user to select a region of the plot by clicking and dragging the middle mouse button.

Querying requires setting *query* to **True** when creating the plot. If you would like to be notified when the user is querying, you just set the callback of the plot. DPG will send the query area through the *app_data* argument as *(x_min, x_max, y_min, y_max)*.

Alternatively, you can poll the plot for the query area by calling: `get_plot_query_area` and `is_plot_queried`.

Below is an example using the callback

```
import dearpygui.dearpygui as dpg
from math import sin

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i/100)
    sindatay.append(0.5 + 0.5*sin(50*i/100))

with dpg.window(label="Tutorial", width=400, height=400):

    dpg.add_text("Click and drag the middle mouse button!")
    def query(sender, app_data, user_data):
        dpg.set_axis_limits("xaxis_id2", app_data[0], app_data[1])
        dpg.set_axis_limits("yaxis_id2", app_data[2], app_data[3])

    # plot 1
    with dpg.plot(no_title=True, height=400, callback=query, query=True, no_menus=True,␣
↪width=-1):
        dpg.add_plot_axis(dpg.mvXAxis, label="x")
        dpg.add_plot_axis(dpg.mvYAxis, label="y")
        dpg.add_line_series(sindatax, sindatay, parent=dpg.last_item())

    # plot 2
    with dpg.plot(no_title=True, height=400, no_menus=True, width=-1):
```

(continues on next page)

(continued from previous page)

```
        dpg.add_plot_axis(dpg.mvXAxis, label="x1", id="xaxis_id2")
        dpg.add_plot_axis(dpg.mvYAxis, label="y1", id="yaxis_id2")
        dpg.add_line_series(sindatax, sindatay, parent="yaxis_id2")

dpg.start_dearpygui()
```

3.24.9 Custom Context Menus

Plot series are actually containers! If you add widgets to a plot series, they will show up when you right-click the series in the legend.

Below is an example

```
import dearpygui.dearpygui as dpg
from math import sin

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i/100)
    sindatay.append(0.5 + 0.5*sin(50*i/100))

with dpg.window(label="Tutorial", width=400, height=400):

    # create plot
    dpg.add_text("Right click a series in the legend!")
    with dpg.plot(label="Line Series", height=-1, width=-1):

        dpg.add_plot_legend()

        dpg.add_plot_axis(dpg.mvXAxis, label="x")
        dpg.add_plot_axis(dpg.mvYAxis, label="y", id="yaxis")

        # series 1
        dpg.add_line_series(sindatax, sindatay, label="series 1", parent="yaxis", id=
↪"series_1")
        dpg.add_button(label="Delete Series 1", parent=dpg.last_item(), callback=lambda:␣
↪dpg.delete_item("series_1"))

        # series 2
        dpg.add_line_series(sindatax, sindatay, label="series 2", parent="yaxis", id=
↪"series_2")
        dpg.add_button(label="Delete Series 2", parent=dpg.last_item(), callback=lambda:␣
↪dpg.delete_item("series_2"))

dpg.start_dearpygui()
```

3.24.10 Colors and Styles

The color and styles of a plot and series can be changed using theme app item

See also:

For more information on item values *Themes*

Below is a simple example demonstrating this

```
import dearpygui.dearpygui as dpg
from math import sin

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i/100)
    sindatay.append(0.5 + 0.5*sin(50*i/100))

with dpg.window(label="Tutorial"):

    # create a theme for the plot
    with dpg.theme(id="plot_theme"):
        dpg.add_theme_color(dpg.mvPlotCol_XAxisGrid, (0, 255, 0), category=dpg.
↪mvThemeCat_Plots)
        dpg.add_theme_style(dpg.mvPlotStyleVar_MarkerSize, 5, category=dpg.mvThemeCat_
↪Plots)

    # create plot
    with dpg.plot(label="Line Series", height=-1, width=-1):

        # apply theme to plot
        dpg.set_item_theme(dpg.last_item(), "plot_theme")

        # optionally create legend
        dpg.add_plot_legend()

        # REQUIRED: create x and y axes
        dpg.add_plot_axis(dpg.mvXAxis, label="x")
        dpg.add_plot_axis(dpg.mvYAxis, label="y", id="yaxis")

        # create a theme for the series
        with dpg.theme(id="series_theme"):
            dpg.add_theme_color(dpg.mvPlotCol_Line, (0, 255, 0), category=dpg.mvThemeCat_
↪Plots)
            dpg.add_theme_style(dpg.mvPlotStyleVar_Marker, dpg.mvPlotMarker_Diamond,␣
↪category=dpg.mvThemeCat_Plots)

        # series belong to a y axis
        dpg.add_stem_series(sindatax, sindatay, label="0.5 + 0.5 * sin(x)", parent="yaxis
↪")

        # apply theme to series
        dpg.set_item_theme(dpg.last_item(), "series_theme")
```

(continues on next page)

(continued from previous page)

```
dpg.start_dearpygui()
```

3.24.11 Colormaps

Under construction

3.24.12 Gallery

3.25 Popups

Popups are windows that disappear when clicked off of. They are typically used as context menus when right-clicking a widget or as dialogs. In DPG popups are just windows with *popup* set to **True**, *show* set to **False**, and a *clicked_handler* attached to a widget that shows the window when clicked.

3.25.1 Regular Usage

Normally when used, a popup exist until you click away from it. By default, a right click activates the popup. An example is found below

Code .. code-block:: python

import dearpygui.dearpygui as dpg

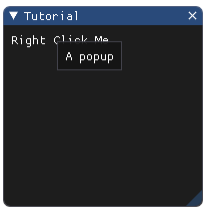
with dpg.window(label="Tutorial"):

dpg.add_text("Right Click Me")

with dpg.popup(dpg.last_item()): dpg.add_text("A popup")

dpg.start_dearpygui()

Results



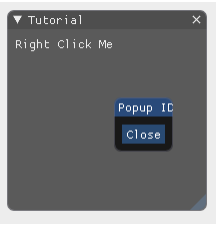
3.25.2 Modal Usage

When the modal keyword is set to **True**, the popup will be modal. This prevents the user from interacting with other windows until the popup is closed. To close the popup, you must hide it. Below is an example

Code

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):

    dpg.add_text("Left Click Me")

    # check out simple module for details
    with dpg.popup(dpg.last_item(), mousebutton=dpg.mvMouseButton_Left, modal=True, id=
↪"modal_id"):
        dpg.add_button(label="Close", callback=lambda: dpg.configure_item("modal_id",␣
↪show=False))

dpg.start_dearpygui()
```

Results



Mouse Button options include: * _mvMouseButton_Right_ * _mvMouseButton_Left_ * _mvMouseButton_Middle_ * _mvMouseButton_X1_ * _mvMouseButton_X2_

3.25.3 Dialog Usage

Simple dialog usage:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Delete Files", modal=True, show=False, id="modal_id"):
    dpg.add_text("All those beautiful files will be deleted.\nThis operation cannot be␣
↪undone!")
    dpg.add_separator()
```

(continues on next page)

(continued from previous page)

```
    dpg.add_checkbox(label="Don't ask me next time")
    dpg.add_button(label="OK", width=75, callback=lambda: dpg.configure_item("modal_id",␣
↪show=False))
    dpg.add_same_line()
    dpg.add_button(label="Cancel", width=75, callback=lambda: dpg.configure_item("modal_
↪id", show=False))

with dpg.window(label="Tutorial"):

    dpg.add_button(label="Open Dialog", callback=lambda:dpg.configure_item("modal_id",␣
↪show=True))

dpg.start_dearpygui()
```

3.26 Simple Plots

Simple plots take in a list and plot y-axis data against the number of items in the list. These can be line graphs or histograms and are demonstrated below

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial", width=500, height=500):
    dpg.add_simple_plot(label="Simpleplot1", default_value=(0.3, 0.9, 0.5, 0.3),␣
↪height=300)
    dpg.add_simple_plot(label="Simpleplot2", default_value=(0.3, 0.9, 2.5, 8.9), overlay=
↪"Overlaying", height=180, histogram=True)

dpg.start_dearpygui()
```

You can change the simple plot's data using *set_value*.

Here we are using a mouse move handler and each callback that runs will set the plot data to make it animated!

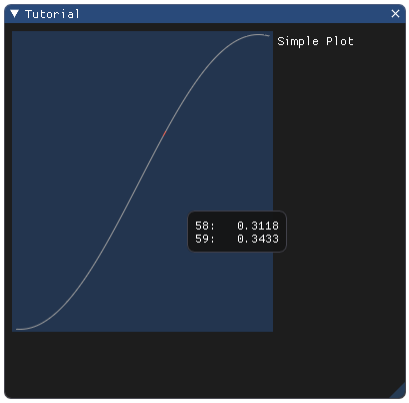
```
import dearpygui.dearpygui as dpg
from math import sin

def update_plot_data(sender, app_data, plot_data):
    mouse_y = app_data[1]
    if len(plot_data) > 100:
        plot_data.pop(0)
    plot_data.append(sin(mouse_y/30))
    dpg.set_value("plot", plot_data)

data=[]
with dpg.window(label="Tutorial", width=500, height=500):
    dpg.add_simple_plot(label="Simple Plot", min_scale=-1.0, max_scale=1.0, height=300,␣
↪id="plot")

with dpg.handler_registry():
    dpg.add_mouse_move_handler(callback=update_plot_data, user_data=data)
```

(continues on next page)

(continued from previous page)

```
dpg.start_dearpygui()
```

Results



3.27 Staging

The staging system is used to create items or item hierarchies where the parent/root is to be decided at a later time. Staged items are not submitted for rendering.

They can later be "unstaged", at which time a parent is known.

The most basic example can be found below:

```
import dearpygui.dearpygui as dpg

dpg.set_staging_mode(True)

dpg.add_button(label="Press me", id="button_id")

# proof the item has been created
print(dpg.get_item_configuration("button_id"))

dpg.set_staging_mode(False)

with dpg.window(label="Tutorial"):
    dpg.unstage_items(["button_id"])

dpg.start_dearpygui()
```

When staging mode is enabled, items will still attempt to *Container Stack* according to the regular procedure however if a parent can not be deduced, it will stage the item.

Using `unstage_items` will attempt to place the item as if you created it there, following the standard rules of *Container Stack*. You can also unstage an item by using `move_item`

3.27.1 Staging Container

DPG provides a special container called a **Staging Container**. This container can only be created when staging mode is enabled and has the special property that when "unstaged" it unpacks its children then deletes itself.

Below is a basic example

```
import dearpygui.dearpygui as dpg

dpg.set_staging_mode(True)

with dpg.staging_container(id="staging_container_id"):
    dpg.add_button(label="Button 1")
    dpg.add_button(label="Button 2")
    dpg.add_button(label="Button 3")
    dpg.add_button(label="Button 4")
    dpg.add_button(label="Button 5")

dpg.set_staging_mode(False)

with dpg.window(label="Tutorial"):
    dpg.unstage_items(["staging_container_id"])

dpg.start_dearpygui()
```

3.27.2 Wrapping Items with Classes

The most obvious benefit of this system is for advanced users who are wrapping DPG items into classes. Rather than having to duplicate the various configurable options as members of the class (to avoid making calls to `configure_item` or `get_item_configuration` before the item is actually created), you can create and stage the item in the constructor of the wrapping class!

Below is are 2 simple examples:

Example 1

```
import dearpygui.dearpygui as dpg

class Button:

    def __init__(self, label):
        dpg.set_staging_mode(True)
        with dpg.staging_container() as self._staging_container_id:
            self._id = dpg.add_button(label=label)
        dpg.set_staging_mode(False)

    def set_callback(self, callback):
        dpg.set_item_callback(self._id, callback)

    def get_label(self):
        return dpg.get_item_label(self._id)

    def submit(self):
        dpg.unstage_items([self._staging_container_id])
```

(continues on next page)

(continued from previous page)

```
my_button = Button("Press me")
my_button.set_callback(lambda: print("I've been pressed!"))

print(my_button.get_label())

with dpg.window(label="Tutorial"):
    my_button.submit()

dpg.start_dearpygui()
```

Example 2

```
import dearpygui.dearpygui as dpg

class Window:

    def __init__(self, label):
        self._children = []
        dpg.set_staging_mode(True)
        self._id = dpg.add_window(label=label)
        dpg.set_staging_mode(False)

    def add_child(self, child):
        dpg.move_item(child._id, parent=self._id)

    def submit(self):
        dpg.unstage_items([self._id])

class Button:

    def __init__(self, label):
        dpg.set_staging_mode(True)
        self._id = dpg.add_button(label=label)
        dpg.set_staging_mode(False)

    def set_callback(self, callback):
        dpg.set_item_callback(self._id, callback)


my_button = Button("Press me")
my_button.set_callback(lambda: print("I've been pressed!"))

my_window = Window("Tutorial")

my_window.add_child(my_button)

my_window.submit()

dpg.start_dearpygui()
```

3.28 Table API (0.8.0)

The table API is a low level API that can be used to create a table. It can also be used as a layout mechanism. Tables are composed of multiple components which include columns, rows, next_columns, and the actual app items to be displayed. The best place to learn about the various configuration options for the table is by running the demo!

Below is the minimum example for creating a table

Code

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):

    with dpg.table(header_row=False):

        # use add_table_column to add columns to the table,
        # table columns use child slot 0
        dpg.add_table_column()
        dpg.add_table_column()
        dpg.add_table_column()

        # add_table_next_column will jump to the next row
        # once it reaches the end of the columns
        # table next column use slot 1
        for i in range(0, 4):
            for j in range(0, 3):
                dpg.add_text(f"Row{i} Column{j}")

                # call if not last cell
                if not (i == 3 and j == 2):
                    dpg.add_table_next_column()

dpg.start_dearpygui()
```

Note: The maximum number of columns is 64.

3.28.1 Borders, Background

You can control the borders of the table using the *borders_innerH*, *borders_innerV*, *borders_outerH*, and *borders_outerV* keywords. You can also turn on alternate row coloring using the *row_background* keyword.

Code

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about"):

    with dpg.table(header_row=False, row_background=True,
                           borders_innerH=True, borders_outerH=True, borders_
↪innerV=True,
                           borders_outerV=True):
```

(continues on next page)

(continued from previous page)

```
        # use add_table_column to add columns to the table,
        # table columns use slot 0
        dpg.add_table_column()
        dpg.add_table_column()
        dpg.add_table_column()

        # add_table_next_column will jump to the next row
        # once it reaches the end of the columns
        # table next column use slot 1
        for i in range(0, 4):
            for j in range(0, 3):
                dpg.add_text(f"Row{i} Column{j}")
                if not (i == 3 and j == 2):
                    dpg.add_table_next_column()

dpg.start_dearpygui()
```

3.28.2 Column Headers

Column headers are simply shown by setting *header_row* to **True** and setting the label of the columns.

Code

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about"):

    with dpg.table(header_row=True):

        # use add_table_column to add columns to the table,
        # table columns use slot 0
        dpg.add_table_column(label="Header 1")
        dpg.add_table_column(label="Header 2")
        dpg.add_table_column(label="Header 3")

        # add_table_next_column will jump to the next row
        # once it reaches the end of the columns
        # table next column use slot 1
        for i in range(0, 4):
            for j in range(0, 3):
                dpg.add_text(f"Row{i} Column{j}")
                if not (i == 3 and j == 2):
                    dpg.add_table_next_column()

dpg.start_dearpygui()
```

3.28.3 Resizing

In order for a table to have resizable columns, the *resizable* and *borders_innerV* keywords must be set to **True**.

You can also set the sizing policy keyword, *policy*, using the following options

Policy |
—- |
mvTable_SizingFixedFit |
mvTable_SizingFixedSame |
mvTable_SizingStretchProp |
mvTable_SizingStretchSame |

3.28.4 Stretch

Below is an example of setting the stretch policy for the entire table

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about"):

    with dpg.table(header_row=False, resizable=True, policy=dpg.mvTable_
↪SizingStretchProp,
                borders_outerH=True, borders_innerV=True, borders_outerV=True):

        dpg.add_table_column(label="Header 1")
        dpg.add_table_column(label="Header 2")
        dpg.add_table_column(label="Header 3")

        for i in range(0, 5):
            for j in range(0, 3):
                dpg.add_text(f"Row{i} Column{j}")
                if not (i == 4 and j == 2):
                    dpg.add_table_next_column()

dpg.start_dearpygui()
```

Fixed

Below is an example of setting the fixed fit policy for the entire table

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about"):

    # Only available if scrollX/scrollY are disabled and stretch columns are not used
    with dpg.table(header_row=False, policy=dpg.mvTable_SizingFixedFit, resizable=True,␣
↪no_host_extendX=True,
                borders_innerV=True, borders_outerV=True, borders_outerH=True):

        dpg.add_table_column(label="Header 1")
        dpg.add_table_column(label="Header 2")
```

(continues on next page)

(continued from previous page)

```
        dpg.add_table_column(label="Header 3")

        for i in range(0, 5):
            for j in range(0, 3):
                dpg.add_text(f"Row{i} Column{j}")
                if not (i == 4 and j == 2):
                    dpg.add_table_next_column()

dpg.start_dearpygui()
```

Mixed

You can also set columns individually by using the *width_fixed* or *width_stretch* keyword along with the *init_width_or_weight* keyword.

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about"):

    with dpg.table(header_row=True, policy=dpg.mvTable_SizingFixedFit, row_
↪background=True, reorderable=True,
                resizable=True, no_host_extendX=False, hideable=True,
                borders_innerV=True, delay_search=True, borders_outerV=True, borders_
↪innerH=True, borders_outerH=True):

        dpg.add_table_column(label="AAA", width_fixed=True)
        dpg.add_table_column(label="BBB", width_fixed=True)
        dpg.add_table_column(label="CCC", width_stretch=True, init_width_or_weight=0.0)
        dpg.add_table_column(label="DDD", width_stretch=True, init_width_or_weight=0.0)

        for i in range(0, 5):
            for j in range(0, 4):
                if j == 2 or j == 3:
                    dpg.add_text(f"Stretch {i},{j}")
                else:
                    dpg.add_text(f"Fixed {i}, {j}")
                if not (i == 4 and j == 3):
                    dpg.add_table_next_column()

dpg.start_dearpygui()
```

3.28.5 Column Options

There are a large number of options available for table columns which are best learned through running the demo, these include

keyword | default value | description |

|---------|-------------|-------------| | init_width_or_weight | 0.0 | sets the starting width (fixed policy) or proportion (stretch) of the column. | | default_hide | False | Default as a hidden/disabled column. | | default_sort | False | De-

fault as a sorting column. | | width_stretch | False | Column will stretch. Preferable with horizontal scrolling disabled (default if table sizing policy is _SizingStretchSame or _SizingStretchProp). | | width_fixed | False | Column will not stretch. Preferable with horizontal scrolling enabled (default if table sizing policy is _SizingFixedFit and table is resizable). | | no_resize | False | Disable manual resizing. | | no_reorder | False | Disable manual reordering this column, this will also prevent other columns from crossing over this column. | | no_hide | False | Disable ability to hide/disable this column. | | no_clip | False | Disable clipping for this column. | | no_sort | False | Disable sorting for this column. | | no_sort_ascending | False | Disable ability to sort in the ascending direction. | | no_sort_descending | False | Disable ability to sort in the descending direction. | | no_header_width | False | Disable header text width contribution to automatic column width. | | prefer_sort_ascending | True| Make the initial sort direction Ascending when first sorting on this column (default). | | prefer_sort_descending | False | Make the initial sort direction Descending when first sorting on this column. | | indent_enabled | False | Use current Indent value when entering cell (default for column 0). | | indent_disable | False | Ignore current Indent value when entering cell (default for columns > 0). Indentation changes _within_ the cell will still be honored. |

3.28.6 Sorting

Under construction.

3.28.7 Scrolling

Under construction

3.28.8 Clipping

Using a clipper can help performance with large tables.

Because the clipper works on single items, you must group your table rows with `add_table_row` or the corresponding context manager. For the clipper to work properly, the rows must have uniform height.

Try using the example below with and with out clipping and see the effect on the framerate listed in metrics.

```
import dearpygui.dearpygui as dpg

def clipper_toggle(sender, value):

    if value:
        dpg.show_item("clipper")
        dpg.hide_item("no_clipper")
    else:
        dpg.show_item("no_clipper")
        dpg.hide_item("clipper")

with dpg.window(label="Tutorial"):

    dpg.add_checkbox(label="clipper", default_value=True, callback=clipper_toggle)

    with dpg.table(header_row=False, id="clipper"):

        for i in range(5):
            dpg.add_table_column()

        with dpg.clipper():
```

(continues on next page)

(continued from previous page)

```
            for i in range(20000):
                with dpg.table_row(): # clipper must use table_row item
                    for j in range(5):
                        dpg.add_text(f"Row{i} Column{j}")

    with dpg.table(header_row=False, id="no_clipper", show=False):

        for i in range(5):
            dpg.add_table_column()

        for i in range(20000):
            with dpg.table_row(): # clipper must use table_row item
                for j in range(5):
                    dpg.add_text(f"Row{i} Column{j}")

dpg.show_metrics()
dpg.start_dearpygui()
```

3.28.9 Filtering

Under construction

3.28.10 Padding

Under construction

3.28.11 Outer Size

Under construction

3.28.12 Column Widths

Under construction

3.28.13 Rows

Under construction

3.28.14 Row Height

Under construction

3.28.15 Search Delay

Under construction

3.29 Textures

DPG uses the Graphics Processing Unit (GPU) to create the graphical user interface(GUI) you see. To display an image, you must first create a texture with the image data that can then be uploaded to the GPU. These textures belong to a texture registry.

We offer 3 types of textures

- Static
- Dynamic
- Raw

This textures are then used in the following App Items

- **mvDrawImage**
- **mvImage**
- **mvImageButton**
- **mvImageSeries**

They are always 1D lists or arrays.

3.30 Static Textures

Static textures are used for images that do not change often. They are typically loaded at startup. If they need to be updated, you would delete and recreate them. These accept python lists, tuples, numpy arrays, and any type that supports python's buffer protocol with contiguous data. Below is a simple example

```
import dearpygui.dearpygui as dpg

texture_data = []
for i in range(0, 100*100):
    texture_data.append(255/255)
    texture_data.append(0)
    texture_data.append(255/255)
    texture_data.append(255/255)

with dpg.texture_registry():
    dpg.add_static_texture(100, 100, texture_data, id="texture_id")

with dpg.window(label="Tutorial"):
    dpg.add_image("texture_id")
```

(continues on next page)

(continued from previous page)

```
dpg.start_dearpygui()
```

3.31 Dynamic Textures

Dynamic textures are used for small to medium sized textures that can change per frame. These can be updated with `set_value` but the width and height must be the same as when the texture was first created. These are similar to raw textures except these perform safety checks and conversion. Below is a simple example

```
import dearpygui.dearpygui as dpg

texture_data = []
for i in range(0, 100*100):
    texture_data.append(255/255)
    texture_data.append(0)
    texture_data.append(255/255)
    texture_data.append(255/255)

with dpg.texture_registry():
    dpg.add_dynamic_texture(100, 100, texture_data, id="texture_id")

def _update_dynamic_textures(sender, app_data, user_data):

    new_color = dpg.get_value(sender)
    new_color[0] = new_color[0]/255
    new_color[1] = new_color[1]/255
    new_color[2] = new_color[2]/255
    new_color[3] = new_color[3]/255

    new_texture_data = []
    for i in range(0, 100*100):
        new_texture_data.append(new_color[0])
        new_texture_data.append(new_color[1])
        new_texture_data.append(new_color[2])
        new_texture_data.append(new_color[3])

    dpg.set_value("texture_id", new_texture_data)

with dpg.window(label="Tutorial"):
    dpg.add_image("texture_id")
    dpg.add_color_picker((255, 0, 255, 255), label="Texture",
        no_side_preview=True, alpha_bar=True, width=200,
        callback=_update_dynamic_textures)

dpg.start_dearpygui()
```

3.32 Raw Textures

Raw textures are used in the same way as dynamic textures. The main differences

- Only accepts arrays (numpy, python, etc.)
- No safety checks are performed.

These textures are used for high performance applications that require updating large textures every frame. Below is a simple example

```
import dearpygui.dearpygui as dpg
import array

texture_data = []
for i in range(0, 100*100):
    texture_data.append(255/255)
    texture_data.append(0)
    texture_data.append(255/255)
    texture_data.append(255/255)

raw_data = array.array('f', texture_data)

with dpg.texture_registry():
    dpg.add_raw_texture(100, 100, raw_data, format=dpg.mvFormat_Float_rgba, id="texture_
↪id")

def update_dynamic_texture(sender, app_data, user_data):

    new_color = dpg.get_value(sender)
    new_color[0] = new_color[0]/255
    new_color[1] = new_color[1]/255
    new_color[2] = new_color[2]/255
    new_color[3] = new_color[3]/255

    for i in range(0, 100*100*4):
        raw_data[i] = new_color[i % 4]

with dpg.window(label="Tutorial"):
    dpg.add_image("texture_id")
    dpg.add_color_picker((255, 0, 255, 255), label="Texture",
        no_side_preview=True, alpha_bar=True, width=200,
        callback=update_dynamic_texture)

dpg.start_dearpygui()
```

3.33 Formats

The following formats are currently supported

mvFormat_Float_rgba **mvFormat_Float_rgb** - - * **mvFormat_Int_rgba** - - - **mvFormat_Int_rgb** - - *

Note:

mvFormat_Float_rgb not currently supported on MacOS
More formats will be added in the future.

3.34 Loading Images

DPG provides the function `load_image` for loading image data from a file.

This function returns a tuple where

- 0 -> width
- 1 -> height
- 2 -> channels
- 3 -> data (1D array, mvBuffer)

On failure, returns **None**.

The accepted file types include

- JPEG (no 12-bit-per-channel JPEG OR JPEG with arithmetic coding)
- PNG
- BMP
- PSD
- GIF
- HDR
- PIC
- PPM
- PGM

A simple example can be found below

```
import dearpygui.dearpygui as dpg

width, height, channels, data = dpg.load_image("Somefile.png")

with dpg.texture_registry():
    dpg.add_static_texture(width, height, data, id="texture_id")

with dpg.window(label="Tutorial"):
    dpg.add_image("texture_id")
```

(continues on next page)

(continued from previous page)

```
dpg.start_dearpygui()
```

3.35 Themes

In DPG, there is an app item container called a "theme". A theme is composed of theme colors and styles which are themselves app items. The theme can either be set as the default theme, attached to an app item type, a item container, or a specific item.

3.35.1 Categories

Theme colors and styles fall into the following categories:

- **mvThemeCat_Core**
- **mvThemeCat_Plots**
- **mvthemeCat_Nodes**

3.35.2 How does an app item decide its color/style?

Every app item requires certain styles/colors to be set. When an app item is drawn, it performs several checks to locate the colors/styles its needs. The search order is:

1. Locally attached theme item.
2. Globally attached theme item.
3. Ancestor tree attached theme.
4. User set default theme.
5. DPG default theme.

3.35.3 Apply theme to specific item

Below is an example of attaching a theme to a specific widget:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about"):
    dpg.add_button(label="Button 1", id="button1")
    dpg.add_button(label="Button 2", id="button2")

# create a theme
with dpg.theme(id="theme_id"):
    dpg.add_theme_color(dpg.mvThemeCol_Button, (255, 140, 23), category=dpg.mvThemeCat_
↪Core)
    dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_Core)

dpg.set_item_theme("button1", "theme_id")
```

(continues on next page)

(continued from previous page)

```
dpg.start_dearpygui()
```

3.35.4 Apply theme to a type

By applying a theme to a type, the theme only effects a specific app item type:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about"):
    dpg.add_button(label="Button 1")
    dpg.add_button(label="Button 2")

# create a theme
with dpg.theme(id="theme_id"):
    dpg.add_theme_color(dpg.mvThemeCol_Button, (255, 140, 23), category=dpg.mvThemeCat_
↪Core)
    dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_Core)

dpg.set_item_type_theme(dpg.mvButton, "theme_id")

dpg.start_dearpygui()
```

3.35.5 Apply theme to a container

By applying a theme to a container, the theme is propagated to its children:

```
import dearpygui.dearpygui as dpg

with dpg.window(label="about", id="window_id"):
    dpg.add_button(label="Button 1")
    dpg.add_button(label="Button 2")

# create a theme
with dpg.theme(id="theme_id"):
    dpg.add_theme_color(dpg.mvThemeCol_Button, (255, 140, 23), category=dpg.mvThemeCat_
↪Core)
    dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_Core)

dpg.set_item_theme("window_id", "theme_id")

dpg.start_dearpygui()
```

3.35.6 Apply default theme

Default themes will replace the default theme for every new item created. Below is an example of applying a default theme:

```
import dearpygui.dearpygui as dpg

# create a theme
with dpg.theme(default_theme=True):
    dpg.add_theme_color(dpg.mvThemeCol_Button, (255, 140, 23), category=dpg.mvThemeCat_
↪Core)
    dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_Core)

with dpg.window(label="about"):
    dpg.add_button(label="Button 1")
    dpg.add_button(label="Button 2")

dpg.start_dearpygui()
```

3.35.7 Plot Markers

Plot Markers	
mvPlotMarker_None	mvPlotMarker_Circle
mvPlotMarker_Square	mvPlotMarker_Diamond
mvPlotMarker_Up	mvPlotMarker_Down
mvPlotMarker_Left	mvPlotMarker_Right
mvPlotMarker_Cross	mvPlotMarker_Plus
mvPlotMarker_Asterisk	

3.35.8 Core Colors

Core Colors		
mvThemeCol_Text	mvThemeCol_TabActive	mvThemeCol_SliderGrabActive
mvThemeCol_TextDisabled	mvThemeCol_TabUnfocused	mvThemeCol_Button
mvThemeCol_WindowBg	mvThemeCol_TabUnfocusedActive	mvThemeCol_ButtonHovered
mvThemeCol_ChildBg	mvThemeCol_DockingPreview	mvThemeCol_ButtonActive
mvThemeCol_Border	mvThemeCol_DockingEmptyBg	mvThemeCol_Header
mvThemeCol_PopupBg	mvThemeCol_PlotLines	mvThemeCol_HeaderHovered
mvThemeCol_BorderShadow	mvThemeCol_PlotLinesHovered	mvThemeCol_HeaderActive
mvThemeCol_FrameBg	mvThemeCol_PlotHistogram	mvThemeCol_Separator
mvThemeCol_FrameBgHovered	mvThemeCol_PlotHistogramHovered	mvThemeCol_SeparatorHovered
mvThemeCol_FrameBgActive	mvThemeCol_TableHeaderBg	mvThemeCol_SeparatorActive
mvThemeCol_TitleBg	mvThemeCol_TableBorderStrong	mvThemeCol_ResizeGrip
mvThemeCol_TitleBgActive	mvThemeCol_TableBorderLight	mvThemeCol_ResizeGripHovered
mvThemeCol_TitleBgCollapsed	mvThemeCol_TableRowBg	mvThemeCol_ResizeGripActive
mvThemeCol_MenuBarBg	mvThemeCol_TableRowBgAlt	mvThemeCol_Tab
mvThemeCol_ScrollbarBg	mvThemeCol_TextSelectedBg	mvThemeCol_TabHovered
mvThemeCol_ScrollbarGrab	mvThemeCol_DragDropTarget	
mvThemeCol_ScrollbarGrabHovered	mvThemeCol_NavHighlight	
mvThemeCol_ScrollbarGrabActive	mvThemeCol_NavWindowingHighlight	
mvThemeCol_CheckMark	mvThemeCol_NavWindowingDimBg	
mvThemeCol_SliderGrab	mvThemeCol_ModalWindowDimBg	

3.35.9 Plot Colors

Plot Colors		
mvPlotCol_Line	mvPlotCol_LegendBg	mvPlotCol_YAxisGrid
mvPlotCol_Fill	mvPlotCol_LegendBorder	mvPlotCol_YAxis2
mvPlotCol_MarkerOutline	mvPlotCol_LegendText	mvPlotCol_YAxisGrid2
mvPlotCol_MarkerFill	mvPlotCol_TitleText	mvPlotCol_YAxis3
mvPlotCol_ErrorBar	mvPlotCol_InlayText	mvPlotCol_YAxisGrid3
mvPlotCol_FrameBg	mvPlotCol_XAxis	mvPlotCol_Selection
mvPlotCol_PlotBg	mvPlotCol_XAxisGrid	mvPlotCol_Query
mvPlotCol_PlotBorder	mvPlotCol_YAxis	mvPlotCol_Crosshairs

3.35.10 Node Colors

Node Colors		
mvNodeCol_NodeBackground	mvNodeCol_TitleBarSelected	mvNodeCol_BoxSelector
mvNodeCol_NodeBackgroundHovered	mvNodeCol_Link	mvNodeCol_BoxSelectorOutline
mvNodeCol_NodeBackgroundSelected	mvNodeCol_LinkHovered	mvNodeCol_GridBackground
mvNodeCol_NodeOutline	mvNodeCol_LinkSelected	mvNodeCol_GridLine
mvNodeCol_TitleBar	mvNodeCol_Pin	mvNodeCol_PinHovered
mvNodeCol_TitleBarHovered		

3.35.11 Core Styles

Constant	Components
mvStyleVar_Alpha	1
mvStyleVar_WindowPadding	2
mvStyleVar_WindowRounding	1
mvStyleVar_WindowBorderSize	1
mvStyleVar_WindowMinSize	2
mvStyleVar_WindowTitleAlign	2
mvStyleVar_ChildRounding	1
mvStyleVar_ChildBorderSize	1
mvStyleVar_PopupRounding	1
mvStyleVar_PopupBorderSize	1
mvStyleVar_FramePadding	2
mvStyleVar_FrameRounding	1
mvStyleVar_FrameBorderSize	1
mvStyleVar_ItemSpacing	2
mvStyleVar_ItemInnerSpacing	2
mvStyleVar_IndentSpacing	1
mvStyleVar_CellPadding	2
mvStyleVar_ScrollbarSize	1
mvStyleVar_ScrollbarRounding	1
mvStyleVar_GrabMinSize	1
mvStyleVar_GrabRounding	1
mvStyleVar_TabRounding	1
mvStyleVar_ButtonTextAlign	2
mvStyleVar_SelectableTextAlign	2

3.35.12 Plot Styles

Constant	Components
mvPlotStyleVar_LineWeight	1
mvPlotStyleVar_Marker	1
mvPlotStyleVar_MarkerSize	1
mvPlotStyleVar_MarkerWeight	1
mvPlotStyleVar_FillAlpha	1
mvPlotStyleVar_ErrorBarSize	1
mvPlotStyleVar_ErrorBarWeight	1
mvPlotStyleVar_DigitalBitHeight	1
mvPlotStyleVar_DigitalBitGap	1
mvPlotStyleVar_PlotBorderSize	1
mvPlotStyleVar_MinorAlpha	1
mvPlotStyleVar_MajorTickLen	2
mvPlotStyleVar_MinorTickLen	2
mvPlotStyleVar_MajorTickSize	2
mvPlotStyleVar_MinorTickSize	2
mvPlotStyleVar_MajorGridSize	2
mvPlotStyleVar_MinorGridSize	2
mvPlotStyleVar_PlotPadding	2
mvPlotStyleVar_LabelPadding	2
mvPlotStyleVar_LegendPadding	2
mvPlotStyleVar_LegendInnerPadding	2
mvPlotStyleVar_LegendSpacing	2
mvPlotStyleVar_MousePosPadding	2
mvPlotStyleVar_AnnotationPadding	2
mvPlotStyleVar_FitPadding	2
mvPlotStyleVar_PlotDefaultSize	2
mvPlotStyleVar_PlotMinSize	2

3.35.13 Node Styles

Constant	Components
mvNodeStyleVar_GridSpacing	1
mvNodeStyleVar_NodeCornerRounding	1
mvNodeStyleVar_NodePaddingHorizontal	1
mvNodeStyleVar_NodePaddingVertical	1
mvNodeStyleVar_NodeBorderThickness	1
mvNodeStyleVar_LinkThickness	1
mvNodeStyleVar_LinkLineSegmentsPerLength	1
mvNodeStyleVar_LinkHoverDistance	1
mvNodeStyleVar_PinCircleRadius	1
mvNodeStyleVar_PinQuadSideLength	1
mvNodeStyleVar_PinTriangleSideLength	1
mvNodeStyleVar_PinLineThickness	1
mvNodeStyleVar_PinHoverRadius	1
mvNodeStyleVar_PinOffset	1

3.36 Tooltips

Tooltips are windows that appear when a widget is hovered. In DPG tooltips are containers that can contain any other widget. Tooltips are 1 of 2 widgets in which the first argument is not the name of the tooltip, but widget that the tooltip is associated with.

```
import dearpygui.dearpygui as dpg

with dpg.window(label="Tutorial"):

    dpg.add_text("Hover me", id="tooltip_parent")

    with dpg.tooltip("tooltip_parent"):
        dpg.add_text("A tooltip")

dpg.start_dearpygui()
```

Results

CHAPTER

FOUR

MORE

Showcase:

doc *extra/showcase*

Video Tutorials:

doc *extra/video-tutorials*

4.1 Showcase

The following apps have been developed with Dear PyGui by various developers.

4.1.1 Tetris

Tetris is a remake of the original Tetris tile-matching game as adopted by IBM PC. Even though Dear PyGui is not a game engine, it can easily handle graphical animations such as these.

The source code is available in the Tetris Github repository.

4.1.2 Snake

Snake is a simple game with customisable settings for changing the speed and colours and fixing the snake length. Entirely made with Dear PyGui.

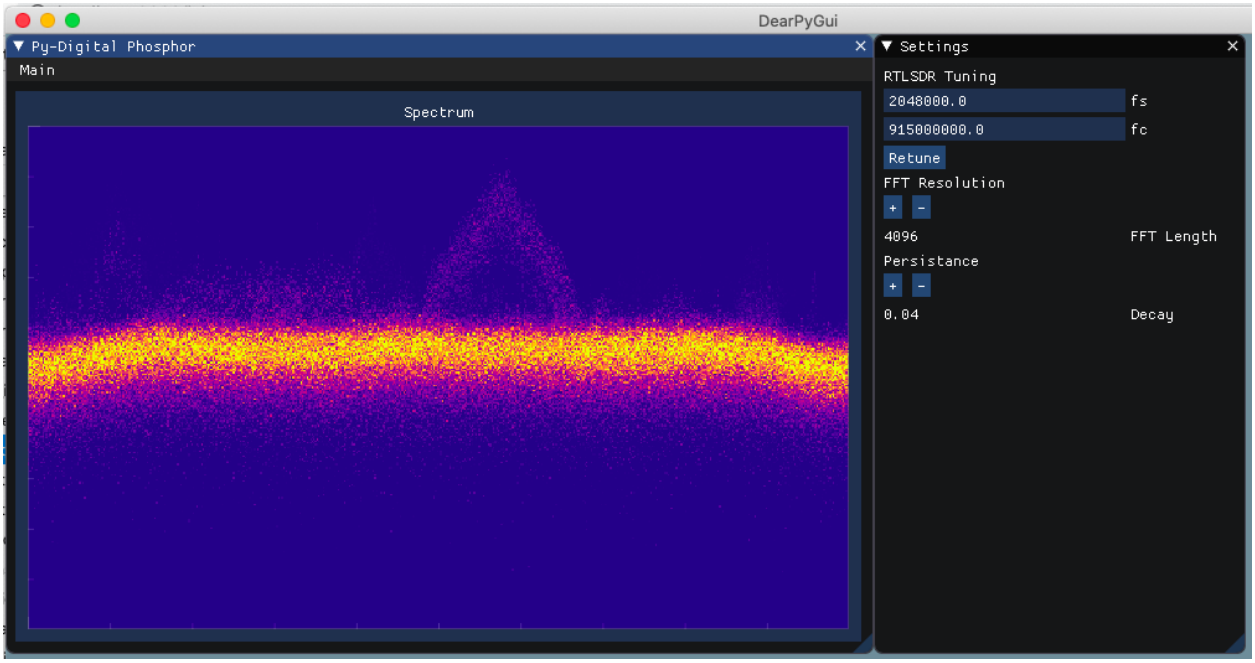
The source code is available in the Snake Github repository.

4.1.3 Digital Phosphor Display with RTLSDR

Digital Phosphor Display Video

This video demonstrates an Intensity graded FFT or Python Digital Phosphor Display by Thomas Schucker. The accompanying blog post shows how to create a this dynamic graph.

The source code is available in the Digital Phosphor Github repository.



4.2 Video Tutorials

The following video tutorials explain various aspects of Dear PyGui, which are outlined per video below. The first section contains all videos by the core developers. The second section lists a great video by the community.

View all video tutorials on Dear PyGui by the core developers on YouTube.

4.2.1 Introduction to Dear PyGui

Dear PyGui is an extended Python wrapper around Dear ImGUI. Dear ImGUI is an Immediate Mode GUI (IMGUI) for real-time applications

While Dear ImGui uses the fast Immediate Mode in the background, it simulates a traditional retained mode GUI to the Python developer, which makes it easy to work with.

Requirements for installation: Python 3.6 and up (64-bit only)

IDE: Setting up PyCharm, only pip install DPG is required

How to run the built-in demo.

Changing the default font type (OTF or TTF font)

Creating a basic window

Using a primary window so that the window matches the area of the viewport

4.2.2 Basics of callbacks

Callbacks are functions that are run when any action is taken on a certain widget.

If a widget supports it, there will be a keyword called callback and that can be any callable.

When using a callback, the keyword should not end with round brackets, e.g. (). The correct way to use a callback is: add_button("Press me", callback=func_name). The function func_name is called without ().

The callback always transmits two *variables* to the callback function, e.g. sender and data.

Sender is the name of the widget. Data is None by default, but additional data can be sent using the following: callback_data='extra data'.

The tutorial shows the use of a callback for a float slider.

Throughout the tutorial, the use of the built-in logger and documentation is demonstrated.

The callback can be changed during runtime using set_item_callback.

4.2.3 ID system and debug tool

The first argument of a widget is the ID.

The label defaults to the ID if no label is provided.

Widgets need unique IDs, but can share the same label. There are two ways to accomplish this.

Method 1: add_input_float('Float1', label='float1') and add_input_float('Float2', label='float1')

Method 2: add_input_float('Float1##1') and add_input_float('Float1##2')

ID's are used to retrieve data from widgets.

Start the debug tool by typing show_debug()

The debug tool lists all available commands under tab Commands

You can execute code at runtime using the debug tool.

Track the float values by log_debug(get_value('Float1##1'))

4.2.4 Parent stack system

The parent stack is a collection of all containers in a GUI.

A window is a root item, meaning that it can't have a parent and doesn't need to look at the parent stack. A window is also a container. Because it is a container, a window gets added to the parent stack.

When an item is not a root item, it requires a parent. Every tab bar is added to the parent stack and to a container. A tab bar is a container itself as well. A tab is a child of tab bar, but it is also a container.

When adding a second item of a parent, it is necessary to remove the first item from the parent stack, e.g. pop it, so that the second item becomes part of the containing parent and not its sibling.

The end() command in the core module pops an item off of the parent stack.

A checkbox or (radio) button is part of a container, but not a container itself.

The simple module adds the context managers (e.g. 'with window'). The *with* statements of the context managers automate the application of the end() statement, making the code easier to read.

4.2.5 Value Storage System

In many GUI's the widget's value is stored inside the widget.

In Dear PyGui, a key-value pair for each widget is stored in the value storage system. A key-value pair tracks the type of the value and the value itself. A widget's value can be retrieved and changed through the widget (by the user) and by the program.

Every widget has a keyword source, which by default is equal to the widget's name. If you specify the source, the widget will use that key instead to look up and change values in the value storage system. This allows several widgets to manipulate a single value.

If multiple widgets refer to the same keyword, the type and size have to be the same.

Pre-add a value with add_value if you are using multiple widgets of different types or sizes on a single key-value pair.

A code example is given to demonstrate the value storage system and its types and sizes.

4.2.6 Widget basics

This tutorial shows how to use a number of widget types. Widget types include button, checkbox, label_text, input_int, drag_int, radio_button, combo, listbox and progress_bar widgets.

The use of the callback keyword of a widget is shown. For example, add_button('Press me', callback=callback_function).

The callback_function is called whenever that button is pressed. The callback always sends two arguments to the callback_function: sender and data. Sender is the name of the widget. The 'data' argument is often empty unless the widget has data to send or it is specified in the code. Nonetheless, the argument 'data' is always included.

The use of a number of widget specific keywords are discussed.

It is demonstrated how a progress bar widget can be controlled via a drag_int slider using set_value(...) and configure_item(...)

Many widgets have multi-component versions as well.

More complex use of widgets and multi-component widgets will be shown in future videos.

4.2.7 Tab bar, tabs, and tab button basics

Create a tab bar with the context manager from the simple module, e.g. with tab_bar('tb1') –> with tab('t1') –> add_button('b1').

You can add a callback to a tab_bar using with tab_bar('tb1', callback=callback).

You can add a button to a tab_bar using add_tab_button('+').

Tabs in a tab bar can be made reorderable by using the keyword reorderable=True on the tab_bar.

4.2.8 Simple Plot & Tooltip

Simple Plots is for plotting simple data. This is not to be confused with the more powerful and complex *Plots*.

Create a basic histogram using add_simple_plot("Plot 1", value=[1, 4.3, 8, 9, 3], histogram=True). There are several keywords to customise the plot.

add_text("Hover me", tip="A simple tooltip"). This simple tooltip is only for text. The *Tooltips* is more powerful.

The tooltip widget is a container, i.e. context manager, just like 'with window' and 'with group'. The widget basically acts as another window, so that it can contain any other widget, such as a graph. The example in the video shows how to embed a simple plot in a tooltip in two lines of code.

Note that the user cannot interact with the tooltip widget.

4.2.9 Popups

Popups require a parent. That may change in future versions of Dear PyGui.

A popup is a container, so it has a context manager (with popup:).

Popup is the only widget where the name is not the first argument.

By default, popups are set on the right-click. To change to left-click, add the keyword mousebutton=mvMouseButtonLeft.

Popups are a container and can contain any other widget, i.e. plots.

The modal keyword greys everything else out to draw attention to the popup.

To close the modal popup, it is necessary to add a button with a callback close_popup("popup1").

4.2.10 Experimental Windows Docking

The docking feature enables the user to dock windows to each other and the viewport.

The docking feature is not documented yet (as of January 2021).

enable_docking() will enable experimental docking.

By default, the user needs to hold the shift key to enable docking.

The keyword shift_only = False enables docking without holding the shift key.

The keyword dock_space = True enables docking windows to the viewport.

The docking feature is experimental because you cannot programmatically set up the docking positions.

When the feature comes out of experimental, it can also function as a layout tool, but it still requires lots of work to be released as non-experimental.

4.2.11 Smart tables

This is an elaborate tutorial on creating a smart, interactive table.

The table is created using managed_columns.

The widgets used in the table are add_text, add_input_text and add_input_float

After creating a working example, the code is refactored into a SmartTable class with header, row and get_cell_data methods.

A widget's label can be hidden by using ## at the beginning of a label's name, e.g. add_input_text('##input_text_1') where input_text_1 is not shown in the GUI.

Using add_separator() to change the horizontal spacing of the widgets.

Using the built-in Dear PyGui debugger and logger for solving an coding issue.

4.2.12 Community Videos

Creating a complete Python app with Dear PyGui

Learn how to create a fully-functional Python app step by step! In this project, we will build a graphic user interface with the brand new Dear PyGui library! We will connect this interface to a Simple SMS Spam Filter, which we've built together in a previous project. We will learn how to display images, text, user input, buttons, and separators, as well as hiding widgets and "click" event callback functions.

4.3 Glossary

- alias - A string that takes the place of the regular **int** ID. Aliases can be used anywhere UUID's can be used.
- item - Everything in **Dear PyGui** created with a context manager or a *add_* command.
- root - An item which has no parent (i.e. window, registries, etc.)
- window - A **Dear ImGui** window created with *add_window(...)*.
- viewport - The operating system window.