Dear PyGui

Jonathan Hoffstadt and Preston Cothren

Sep 17, 2024

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 "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, debugger, and multiple resource viewers.

DPG is well suited for creating simple user interfaces as well as 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

DPG is a simple, bloat-free, and powerful Python GUI framework.

DPG is built with *Dear ImGui* and other extensions in order to create a unique retained mode API, as opposed to Dear ImGui's immediate mode paradigm.

Under the hood, DPG uses the immediate mode paradigm allowing for extremely dynamic interfaces. Similar to *PyQt*, DPG 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, DPG 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 DPG is unique with:

- GPU rendering
- Multithreaded
- Highly customizable
- Built-in developer tools: theme inspection, resource inspection, runtime metrics
- 70+ widgets with hundreds of widget combinations

- Detailed documentation, examples and 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

QUICK START

If you're ready to start using DPG 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 do not cover every detail so refer to the documentation 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

Confirm the pip install by running the code block below.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport(title='Custom Title', width=600, height=300)

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()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Result:

2.1.3 Demo

DPG has a complete built-in demo/showcase. It is a good idea to look into this demo. The code for this can be found in the repo in the `demo.py`_ file

Code:

```
import dearpygui.dearpygui as dpg
import dearpygui.demo as demo

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

demo.show_demo()

dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Result:

Note: The main script must always:

- Create the context `create_context`
- Create the viewport `create_viewport`
- Setup dearpygui `setup_dearpygui`
- Show the viewport `show_viewport`
- Start dearpygui `start_dearpygui`
- Clean up the context `destroy_context`

2.2 DPG Structure Overview

A DPG app will have an overall structure as follows:

- Setup
- Context
- Viewport
- Render Loop
- Items
- Primary Window

2.2.1 Setup

All DPG apps must do 3 things:

- Create & Destroy context
- Create & Show Viewport
- Setup & Start DearPyGui

2.2.2 The Context

To access any DPG commands the context must be created with `create_context` This should be the first DPG command and it's typical to perform this with an import.

Proper clean up of DPG can be done using `destroy_context`.

Creating and destroying the context and also setup and start dearpygui are useful when the DPG needs to be started and stopped multiple times in one python session.

> **Warning:** If `create_context` is not first DPG will not start (and will probably crash).

2.2.3 The Viewport

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

The viewport needs to be explicitly created using `create_viewport` and shown using `show_viewport`

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

See also:

For more information on the viewport *Viewport*

2.2.4 The Render Loop

The render loop is responsible for displaying items, partially maintaining state and callbacks.

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

In some cases it's necessary to explicitly create the render loop so you can call python commands that may need to run every frame. Such as per-frame ticker or counter update functions.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

# 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()
    print("this will run every frame")
    dpg.render_dearpygui_frame()

dpg.destroy_context()
```

> **Warning:** The manual render loop must be created after `setup_dearpygui`

See also:

For more information on the render loop *Render Loop*

2.2.5 Item Overview

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

Items: Items are anything in the library (i.e. button, registries, windows, etc).

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

Containers: Items that can hold other items. (i.e. window, groups, registries, etc).

2.2.6 The Primary Window

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

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(tag="Primary 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.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.set_primary_window("Primary Window", True)
dpg.start_dearpygui()
dpg.destroy_context()
```

See also:

for more information on the primary window *Primary Window*

2.3 Item Usage

2.3.1 Creating Items

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

All items must have a tag which can either be specified or are automatically generated by DPG.

Tags can be either integers or strings and are used to refer to the item after it has been created.

Items return their tag when they are created.

Warning: Item tags must be unique if specified using the *tag* keyword. Integers 0-10 are reserved for DPG internal items.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

(continues on next page)

(continued from previous page)

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

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Note: Items can be created and delete at runtime see *Item Creation*

See also:

For more information on the creating items:
Item Creation
Tag System

2.3.2 Creating Containers

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

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=600, height=400)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

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 separately)

Each of these can be accessed by their 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, etc)

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

Note: configuration, state and info are 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 demonstrate the ways to configure items and how to check their state by viewing them through the item registry tool.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

See also:

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

2.3.4 Callbacks

Callbacks give items functionality by assigning a function to run when they are activated 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 runtime using `set_item_callback`

Callbacks may have up to 3 arguments in the following order.

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

---

Note: Because they are optional positional arguments you must use the *sender* and *app_data* if you want to use *user_data* standard keyword arguments

---

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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 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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

See also:

For more information on the item callbacks *Item Callbacks*

2.3.5 Item Values

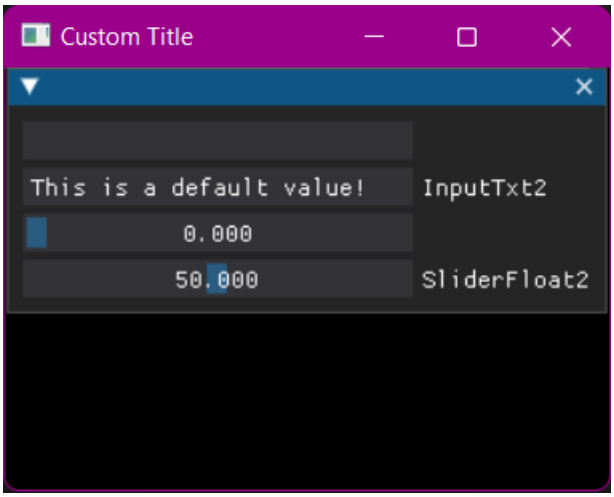
Almost all UI items have a *value* which can be accessed or set.

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 setting the *default_value* for two different items, setting a callback to the items and printing their values.

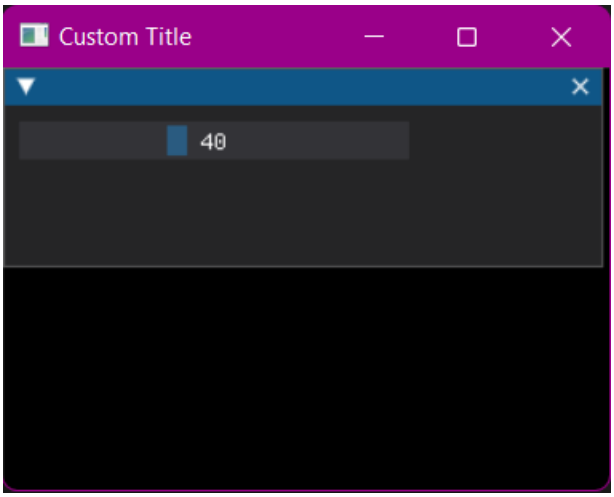
```
import dearpygui.dearpygui as dpg

dpg.create_context()

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
    )

    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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```



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

dpg.create_context()

with dpg.window(width=300):
    # Creating a slider_int widget and setting the
    # default value to 15.
    dpg.add_slider_int(default_value=15, tag="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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```



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 *Values*

2.3.6 Using 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

dpg.create_context()

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", tag="text item")
    with dpg.item_handler_registry(tag="widget handler") as handler:
        dpg.add_item_clicked_handler(callback=change_text)
    dpg.bind_item_handler_registry("text item", "widget handler")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

See also:

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

2.4 Tips & More Resources

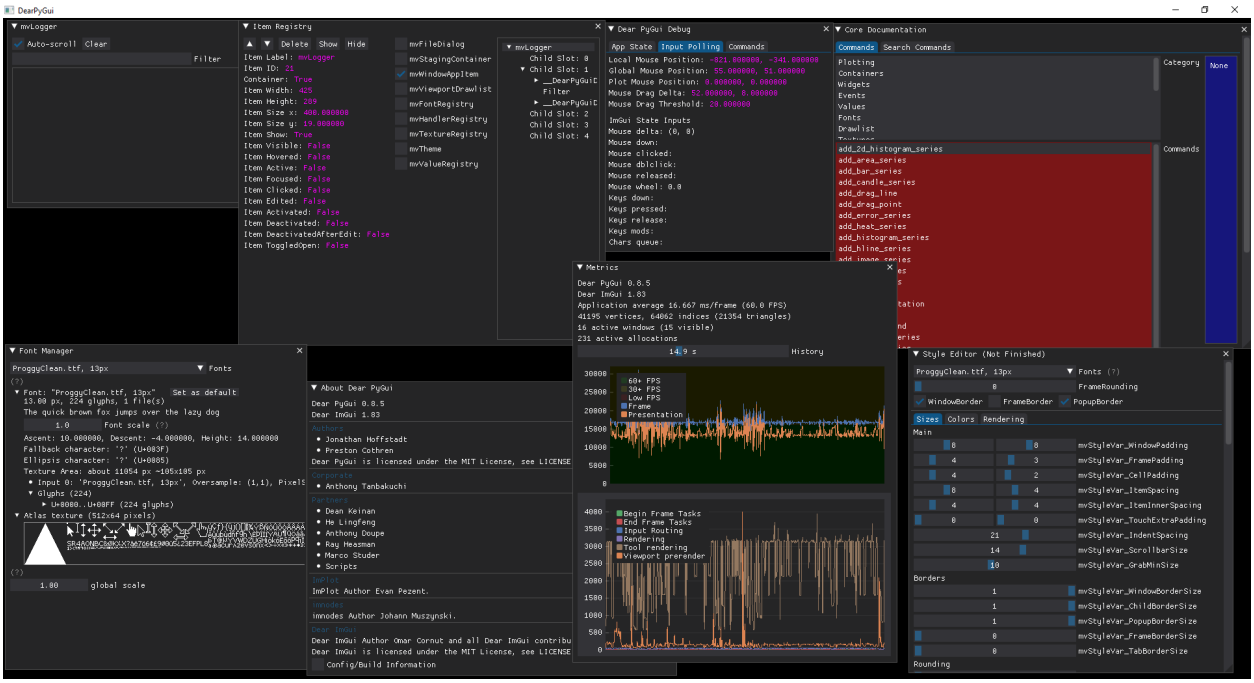
2.4.1 Developer Tools

DPG includes several tools which can help develop and debug applications.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



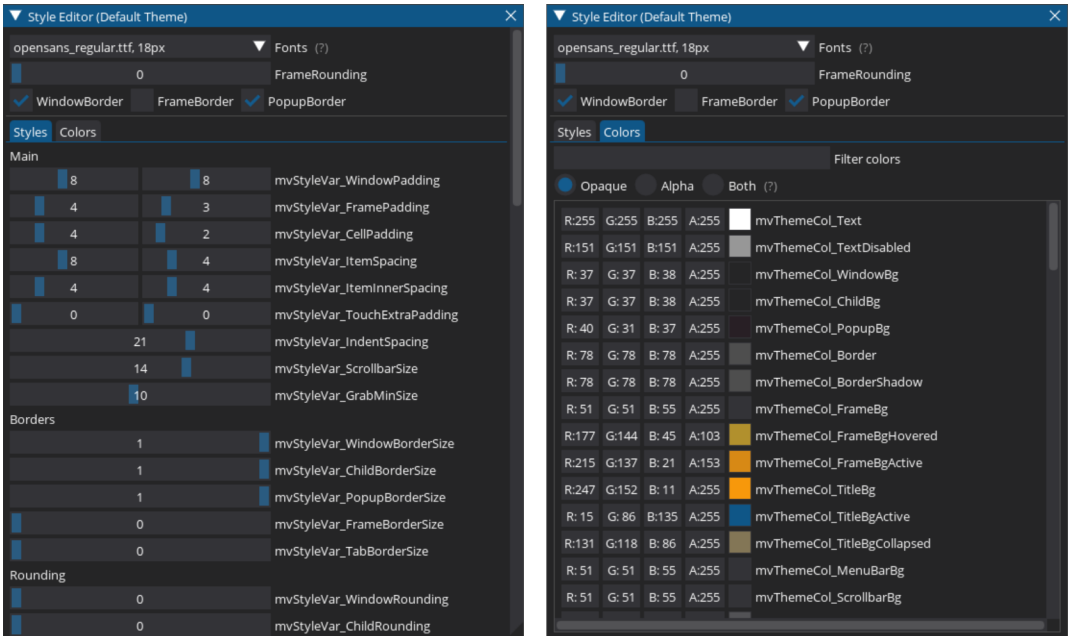
2.4.2 Style editor

The built-in style editor allows you to experiment with all style options at runtime to find the exact colors, padding, rounding and other style settings for your application. You can use the sliders to change the settings, which are applied to all items in your app, so you can immediately see what effect the changes have on your GUI.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

dpg.show_style_editor()

dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Result:



2.4.3 Item registry

The item registry shows all items of a running application in a hierarchical structure. For each item, a number of details, such as its tag ID, are shown.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport()
```

(continues on next page)

(continued from previous page)

```
dpg.setup_dearpygui()

dpg.show_item_registry()

dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Result:

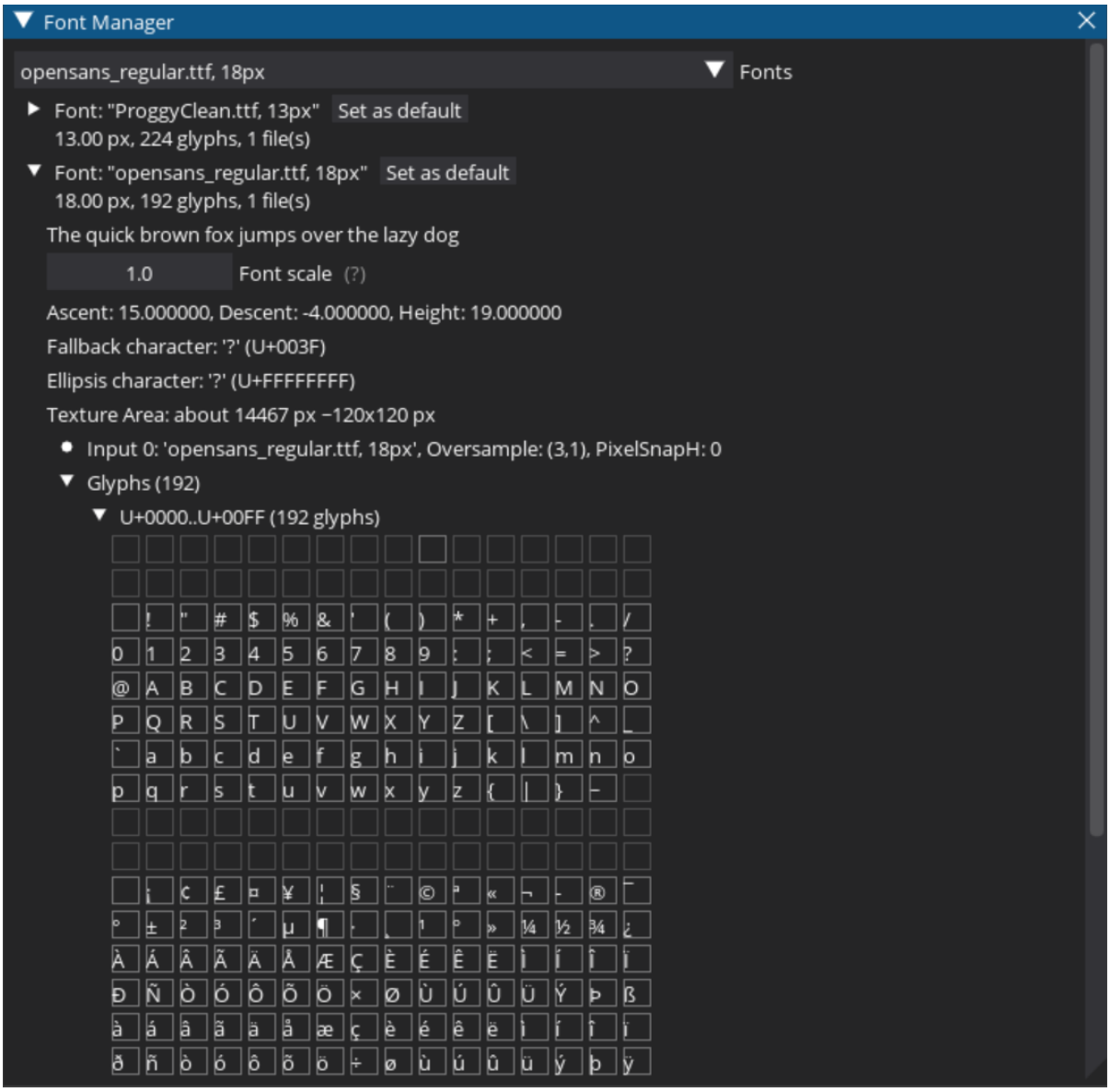
2.4.4 Font manager

The font manager shows all loaded fonts and their appropriate sizes. It allows you to inspect all characters, or glyphs, that are loaded with each font file.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

dpg.show_font_manager()

dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Result:



2.4.5 Runtime metrics

Runtime metrics show the performance of your app in real-time. Here is it shown in conjunction with the built-in style editor.

Code:

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport(title='Custom Title', width=800, height=600)

dpg.show_style_editor()
dpg.show_metrics()
```

(continues on next page)

(continued from previous page)

```
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Result:

2.4.6 More Resources

- *Showcase*
- *Video Tutorials*

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

The render loop (or event loop) runs continously and is responsible for polling user input and drawing widgets.

Drawing the items is how the DPG appears to update items. DPG does this at the rate of your monitor refresh when `set_viewport_vsync` is set **True**. If vsync is set **False** the render loop will run as fast possible.

If you try to run too many computationally expensive operations inside the render loop, you may reduce the frame rate of your application.

For most use cases the render loop does not need to be considered and is completely handled by `start_dearpygui`.

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

```
import dearpygui.dearpygui as dpg

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

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

dpg.show_viewport()

# 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()
    print("this will run every frame")
    dpg.render_dearpygui_frame()

dpg.destroy_context()
```

3.2 Viewport

The viewport is what you traditionally call a **window** in other GUI libraries.

In the case of DPG we call the operating system window the *viewport* and the DPG windows as *windows*.

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`.

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).

3.3 Primary Window

The primary window fills the viewport, resizes with the viewport and remains behind 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

dpg.create_context()

with dpg.window(tag="Primary Window"):
    dpg.add_text("Hello, world")

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.set_primary_window("Primary Window", True)
dpg.start_dearpygui()
dpg.destroy_context()
```

3.4 IO, Handlers, State Polling

3.4.1 Handlers

Handlers are items that submit a callback when the specified state of an item changes.

Handlers can be activated or deactivated by showing or hiding them.

Handlers are required to be added to a handler registry.

3.4.2 Item Handlers

Item handlers listen for states related to a specific item.

Events:

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

Item handlers are required to be added to a item handler registry.

Item handler registries can be bound to an item. They can be bound to multiple items to prevent having to duplicate handlers for every item.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

def visible_call(sender, app_data):
    print("I'm visible")

with dpg.item_handler_registry(tag="widget handler") as handler:
    dpg.add_item_clicked_handler(callback=change_text)
    dpg.add_item_visible_handler(callback=visible_call)

with dpg.window(width=500, height=300):
    dpg.add_text("Click me with any mouse button", tag="text item")
    dpg.add_text("Close window with arrow to change visible state printing to console",␣
↪tag="text item 2")

# bind item handler registry to item
dpg.bind_item_handler_registry("text item", "widget handler")
dpg.bind_item_handler_registry("text item 2", "widget handler")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.4.3 Global Handlers (IO Input)

Global handlers listen for actions not tied to a specific item:

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. They also provide the ability to turn on and off the entire registry.

For example this can allow a mouse input registry or a keyboard input registry. Registries also give the ability to deactivate all their children handlers by simply hiding the registry.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.handler_registry():
    dpg.add_mouse_down_handler(callback=change_text)

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.4.4 Polling Item State

Polling item state is accessible through `get_item_state`. These can be very powerful when combined with handlers as shown below.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.handler_registry():
    dpg.add_mouse_move_handler(callback=change_text)

with dpg.window(width=500, height=300):
    dpg.add_text("Hover Me!", tag="text item")


dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.5 Item Creation

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

3.5.1 Items

Everything created in DPG 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**.

All items have the following optional parameters: *label*, *tag*, *user_data*, and *use_internal_label*. The *tag* is generated automatically or can be specified. A *label* serves as the display name for an item. *user_data* can be any value and is frequently used for **callbacks**.

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

3.5.2 Containers

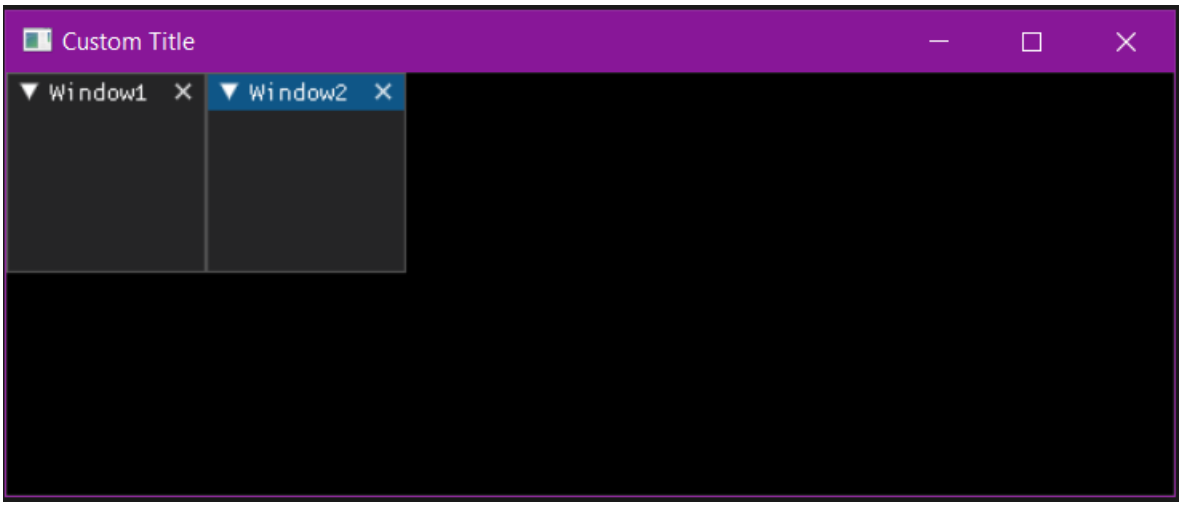
Container items that can contain other (allowable) items.

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

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 and starting the application:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

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

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```



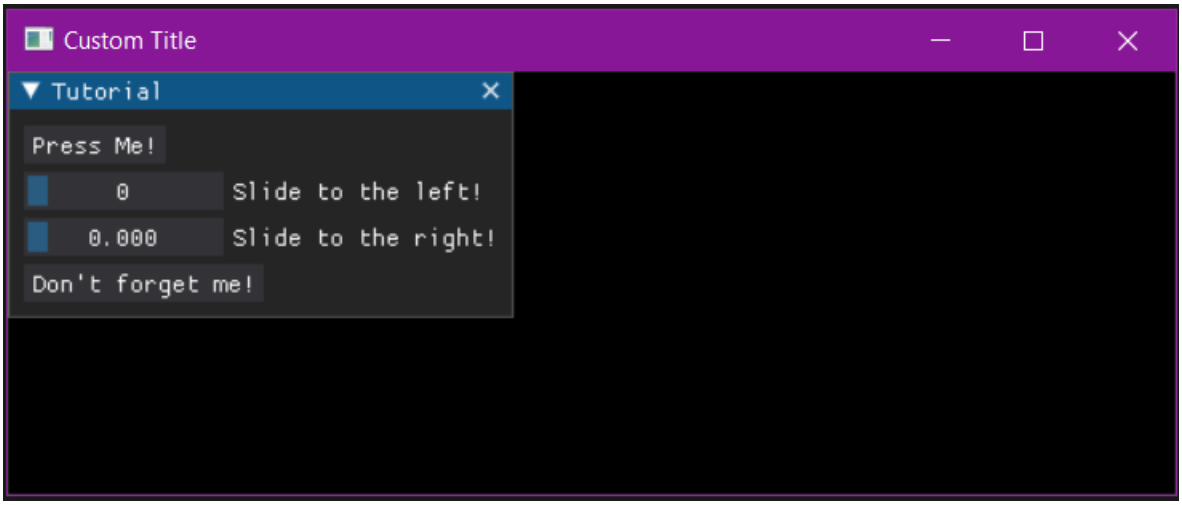
3.5.3 UI Items

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 contains a few other items:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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)
    slider_float = dpg.add_slider_float(label="Slide to the right!", width=100)

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

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

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```



3.5.4 Runtime Adding and Deleting

With DPG you can dynamically add, delete, and move 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.

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

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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


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


with dpg.window(label="Tutorial", pos=(200, 200)):
    dpg.add_button(label="Add Buttons", callback=add_buttons)
    dpg.add_button(label="Delete Buttons", callback=delete_buttons, tag="delete_button")

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

dpg.create_viewport(title='Custom Title', width=600, height=400)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Hint: 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

dpg.create_context()

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

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

(continues on next page)

(continued from previous page)

```
    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.create_viewport(title='Custom Title', width=600, height=400)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.6 Tag System

In DPG, all items must have an associated unique ID (UUID) which can either be an integer or a string.

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

Tags allow for modification of the associated item at runtime.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

unique_id = 0 # to be filled out later

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

with dpg.window(label="Example"):
    dpg.add_button(label="Press me (print to output)", callback=callback)
    unique_id = dpg.add_input_int(label="Input")

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.6.1 Generated Tags

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

```
import dearpygui.dearpygui as dpg

dpg.create_context()

unique_tag = dpg.generate_uuid()

def callback():
```

(continues on next page)

(continued from previous page)

```
    print(dpg.get_value(unique_tag))

with dpg.window(label="Example"):
    dpg.add_button(label="Press me (print to output)", callback=callback)
    dpg.add_input_int(label="Input", tag=unique_tag)

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.6.2 Aliases

An alias is a string that takes the place of the regular **int** tag. Aliases can be used anywhere UUID's can be used. It is the user's responsibility to make sure aliases are unique.

A simple example can be seen below:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

with dpg.window(label="Example"):
    dpg.add_button(label="Press me (print to output)", callback=callback)
    dpg.add_input_int(default_value=5, label="Input", tag="unique_tag")

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.6.3 Recent Tags

The most recent tag is stored for the last item, container, and root.

This is useful when the last item created may be done at run time, is anonymous or sometimes just for convenience.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Example"):
    with dpg.group():
        dpg.add_button(label="View the Terminal for item tags")
        print(dpg.last_item())
        print(dpg.last_container())
```

(continues on next page)

(continued from previous page)

```
        print(dpg.last_root())

dpg.create_viewport(title='Custom Title', width=600, height=200)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.7 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 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.

3.7.1 Example

Simple usage can be found below:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(width=500, height=300):
    dpg.add_button(enabled=True, label="Press me", tag="item")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.8 Item Callbacks

Most 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 can have up to 3 arguments. The first is usually the sender or item triggering the callback. The second is data sent by DPG for different reasons. The third is reserved for user specified data. We refer to this in general terms as: **sender**, **app_data**, and **user_data**.

Note: Because they are optional positional arguments you must use the *sender* and *app_data* if you want to use *user_data* keyword arguments:

3.8.1 Sender, App_data

sender: argument is used by DPG to inform the callback which item triggered the callback by sending the tag or 0 if trigger by the application.

app_data: argument is used by DPG to send information to the callback i.e. the current value of most basic widgets.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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="Print to Terminal", callback=button_callback)

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.8.2 User Data

user_data: argument is **Optionally** used to pass your own python data into the function.

The python data 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

dpg.create_context()

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="Print to Terminal", callback=button_callback, user_data="Some␣
↪Data")

    # user data and callback set any time after button has been created
```

(continues on next page)

(continued from previous page)

```
    dpg.add_button(label="Print to Terminal 2", tag="btn")
    dpg.set_item_callback("btn", button_callback)
    dpg.set_item_user_data("btn", "Some Extra User Data")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.8.3 Debugging Callbacks (new in 1.2)

Because callbacks are not run on the main thread, debugging can be a hassle. In 1.2 we added a few utilities to help with this.

By default, Dear PyGui handles the callbacks internally on a worker thread. This allows for optimizations and steady framerates. However, to help with debugging, you can set the new **manual_callback_management** key to **True** with `configure_app`. This will prevent Dear PyGui from handling the callbacks. Instead the callbacks and arguments will be stored. You can then retrieve (and clear) them by calling `get_callback_queue` within your main event loop. This will return a list of "Jobs". A "Job" is just list with the first item being the callable and the remaining items (up to 3) being the typical arguments. We have also provided `run_callbacks` to properly handle the jobs for simple usage.

Below is a simple example

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.configure_app(manual_callback_management=True)
dpg.create_viewport()
dpg.setup_dearpygui()

def callback(sender, app_data, user_data):
    print("Called on the main thread!")

with dpg.window(label="Tutorial"):
    dpg.add_button(label="Press me", callback=callback)


# main loop
dpg.show_viewport()
while dpg.is_dearpygui_running():
    jobs = dpg.get_callback_queue() # retrieves and clears queue
    dpg.run_callbacks(jobs)
    dpg.render_dearpygui_frame()

dpg.destroy_context()
```

3.9 Values

When an item is created, it creates an associated value by default. Values can be shared between items with the same underlying value type. This is accomplished by using the *source* keyword. One of the benefits of this is to have multiple items control the same value.

Values are retrieved from the value `get_value`.

Values can be changed manually using `set_value`.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Tutorial"):
    dpg.add_checkbox(label="Radio Button1", tag="R1")
    dpg.add_checkbox(label="Radio Button2", source="R1")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.9.1 Value Items

There are several "Value" items that can be used. These are items 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

dpg.create_context()
```

(continues on next page)

(continued from previous page)

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

with dpg.window(label="Tutorial"):
    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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.10 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(...):**
add_viewport_drawlist	**with viewport_drawlist(...):**

continues on next page

Table 1 – continued from previous page

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(...):**

3.10.1 Benefits:

1. Automatically push containers to container stack.
2. Automatically pop containers off container stack.
3. More structured, readable code.

3.10.2 Context Managers:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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", default_value=(0.3, 0.9, 2.5, 8.9),␣
↪height=80)

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
```

(continues on next page)

(continued from previous page)

```
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.10.3 Explicit Parental Assignment (using UUIDs):

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

mb = dpg.add_menu_bar(parent=w)

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

other_themes = dpg.add_menu(label="Other Themes", parent=themes)
dpg.add_menu_item(label="Purple", parent=other_themes)
dpg.add_menu_item(label="Gold", parent=other_themes)
dpg.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", parent=mb)
dpg.add_button(label="A Button", parent=oddities)
dpg.add_simple_plot(label="A menu plot", default_value=(0.3, 0.9, 2.5, 8.9), height=80,␣
↪parent=oddities)

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.10.4 Explicit Parental Assignment (using aliases):

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

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

dpg.add_menu(label="Themes", parent="mb", tag="themes")
```

(continues on next page)

(continued from previous page)

```
dpg.add_menu_item(label="Dark", parent="themes")
dpg.add_menu_item(label="Light", parent="themes")

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

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

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.10.5 Container Stack Operations:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=800, height=600)
```

(continues on next page)

(continued from previous page)

```
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.11 Container Slots & Children

Most items can have child items. Items can only be children to valid **container** 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 slot

`get_item_parent` returns the item's parent UUID

`get_item_children` returns an item's children

`reorder_items` reorders children in a single call

`move_item_up` moves an item up within its slot

`move_item_down` moves an item down within its slot

`move_item` moves an item between containers

`set_item_children` unstaging a stage into an item's children slot

3.11.1 Slots

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

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

Slot 1: Most items

Slot 2: Draw items

Slot 3: **mvDragPayload**

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

3.11.2 Basic Example

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

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="about", width=400, height=400):
    dpg.add_button(label="Press me")
    dpg.draw_line((0, 10), (100, 100), color=(255, 0, 0, 255), thickness=1)
```

(continues on next page)

(continued from previous page)

```
# 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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

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

Note: Use the *slot* and *children_only* keywords with `delete_item` to delete a specific slot of children from a parent.

3.12 Container Stack

Unless an item is a root item, all items need to belong to a valid container. 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 5 using parent of "before" item.
3. Check *parent* keyword, if used skip to 5.
4. Check container stack, if used skip to 5.
5. Check if parent is compatible.
6. Check if parent accepts.
7. If runtime, add item using runtime methods; finished.
8. 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.create_context()

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

(continues on next page)

(continued from previous page)

```
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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.12.1 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.create_context()

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

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

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.12.2 Context Managers

Context managers can be used to simplify the above example. All the context managers can be found in the *Containers & Context Managers* but a simple example can be found below:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Benefits 1. Automatically push container to container stack. 2. Automatically pop container off container stack. 3. More structured, readable code.

3.13 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 added to containers like drawlist, viewport_drawlist, or a window.

A drawlist item 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.

The coordinate system is right-handed with the x axis point left, y axis point down, and z axis pointing into the screen.

Code

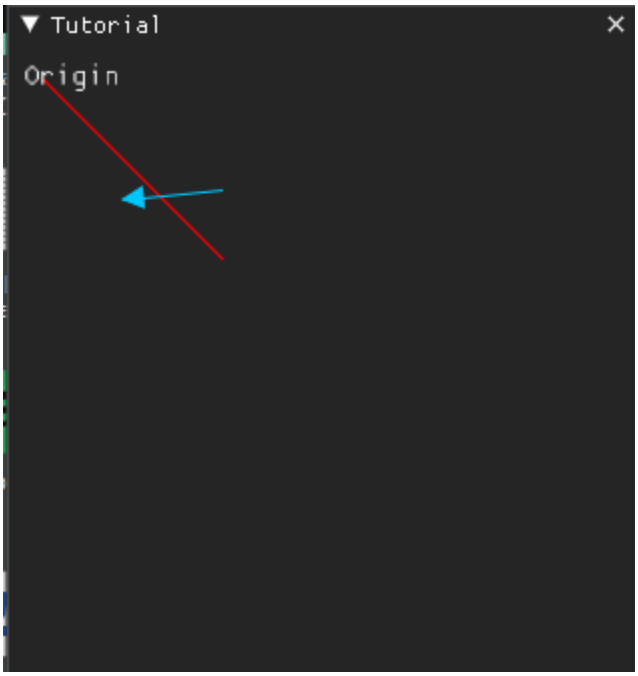
```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
```

(continues on next page)

(continued from previous page)

```
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



3.13.1 Layers

Drawlists can also contain layers. Layers are an effective way to group drawing items for better control of hiding, Z ordering, etc.

New in 1.1. Layers can be used to assist with some 3D operations. See "3D Operations" section below.

Code

```
import dearpygui.dearpygui as dpg

dpg.create_context()

def toggle_layer2(sender):
    show_value = dpg.get_value(sender)
    dpg.configure_item("layer2", show=show_value)

with dpg.window(label="Tutorial"):
    dpg.add_checkbox(label="show layer", callback=toggle_layer2, default_value=True)

    with dpg.drawlist(width=300, height=300):

        with dpg.draw_layer():
            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)
```

(continues on next page)

(continued from previous page)

```
        with dpg.draw_layer(tag="layer2"):
            dpg.draw_line((10, 60), (100, 160), color=(255, 0, 0, 255), thickness=1)
            dpg.draw_arrow((50, 120), (100, 115), color=(0, 200, 255), thickness=1,␣
↪size=10)

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.13.2 Images

Drawlists can display images of types PNG, JPEG, or BMP (See *Textures & Images* for more detail). Images are added 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 drawn onto the canvas. The image will scale to fit the specified area.

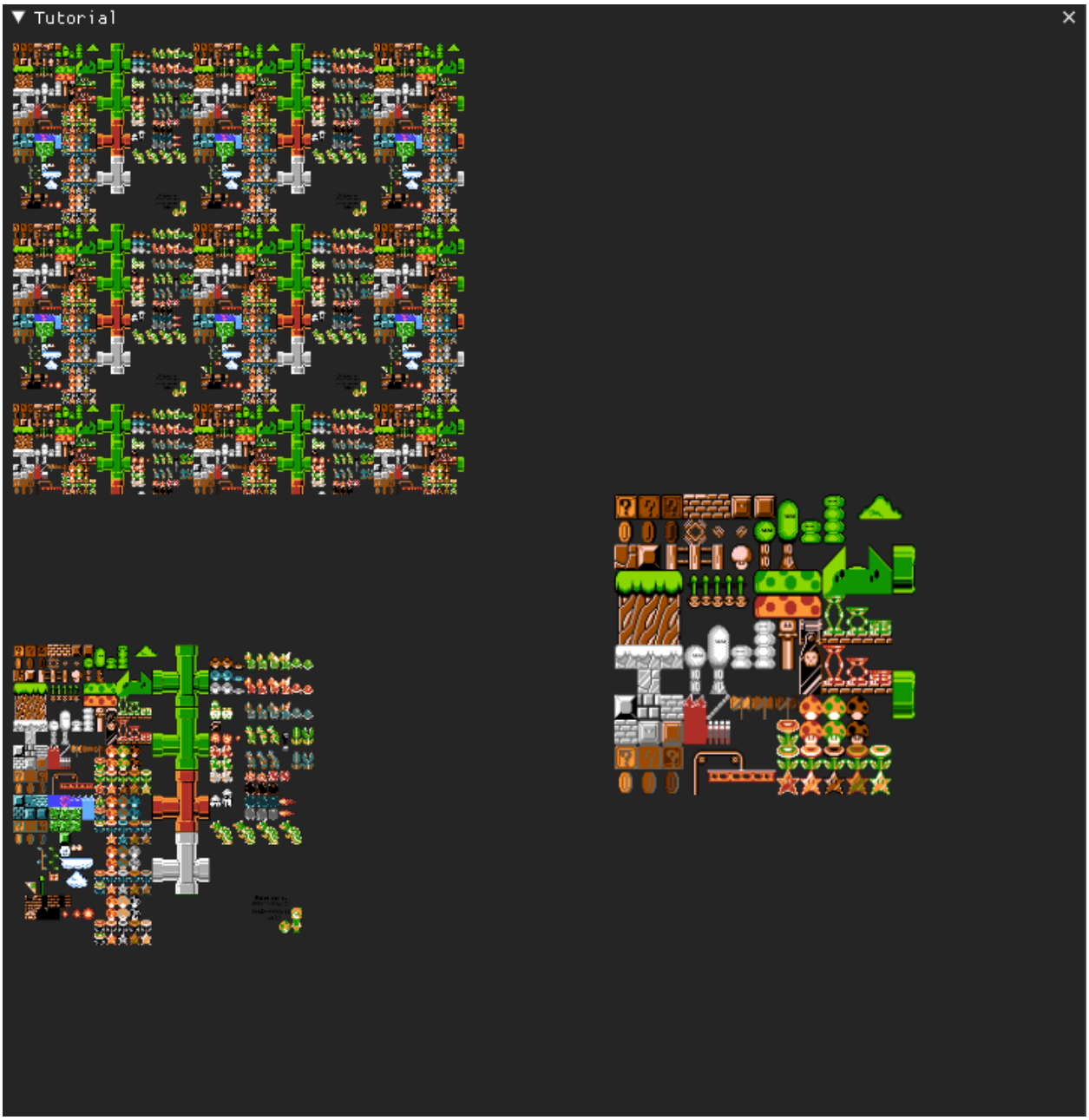
With keywords **uv_min** and **uv_max** we can specify normalized texture coordinates to use just a portion of the area on the image. The default of uv_min = [0,0] and uv_max = [1,1] will display the entire image while uv_min = [0,0] uv_max = [0.5,0.5] will only show the first quarter of the drawing.

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

dpg.create_context()

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, tag="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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



3.13.3 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

dpg.create_context()
```

(continues on next page)

(continued from previous page)

```
# creating font and back viewport drawlists
with dpg.viewport_drawlist():
    dpg.draw_circle((100, 100), 25, color=(255, 255, 255, 255))

dpg.add_viewport_drawlist(front=False, tag="viewport_back")

dpg.draw_circle((200, 200), 25, color=(255, 255, 255, 255), parent="viewport_back")

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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results

3.13.4 Scene Graph

New in 1.1. For more complex drawing, you can utilize a **draw_node** item. A draw node is used to associate a transformation matrix with a group of draw items. You can use **apply_transform** to apply a transformation matrix to a node. This matrix will be used to multiply each child draw item's points. If a child is another draw node, the matricies will concatenate.

Code

```
import dearpygui.dearpygui as dpg
import math

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

with dpg.window(label="tutorial", width=550, height=550):

    with dpg.drawlist(width=500, height=500):

        with dpg.draw_node(tag="root node"):
            dpg.draw_circle([0, 0], 150, color=[0, 255, 0])                    # inner␣
↪planet orbit
            dpg.draw_circle([0, 0], 200, color=[0, 255, 255])                  # outer␣
↪planet orbit
            dpg.draw_circle([0, 0], 15, color=[255, 255, 0], fill=[255, 255, 0]) # sun

            with dpg.draw_node(tag="planet node 1"):
                dpg.draw_circle([0, 0], 10, color=[0, 255, 0], fill=[0, 255, 0]) # inner␣
↪planet
```

(continues on next page)

(continued from previous page)

```
                dpg.draw_circle([0, 0], 25, color=[255, 0, 255])                    # moon␣
↪orbit path

                with dpg.draw_node(tag="planet 1, moon node"):
                    dpg.draw_circle([0, 0], 5, color=[255, 0, 255], fill=[255, 0, 255])
↪# moon

            with dpg.draw_node(tag="planet node 2"):
                dpg.draw_circle([0, 0], 10, color=[0, 255, 255], fill=[0, 255, 255]) #␣
↪outer planet
                dpg.draw_circle([0, 0], 25, color=[255, 0, 255])                    #␣
↪moon 1 orbit path
                dpg.draw_circle([0, 0], 45, color=[255, 255, 255])                  #␣
↪moon 2 orbit path

                with dpg.draw_node(tag="planet 2, moon 1 node"):
                    dpg.draw_circle([0, 0], 5, color=[255, 0, 255], fill=[255, 0, 255])
↪# moon 1

                with dpg.draw_node(tag="planet 2, moon 2 node"):
                    dpg.draw_circle([0, 0], 5, color=[255, 255, 255], fill=[255, 255,␣
↪255]) # moon 2

planet1_distance = 150
planet1_angle = 45.0
planet1_moondistance = 25
planet1_moonangle = 45

planet2_distance = 200
planet2_angle = 0.0
planet2_moon1distance = 25
planet2_moon1angle = 45
planet2_moon2distance = 45
planet2_moon2angle = 120

dpg.apply_transform("root node", dpg.create_translation_matrix([250, 250]))
dpg.apply_transform("planet node 1", dpg.create_rotation_matrix(math.pi*planet1_angle/
↪180.0 , [0, 0, -1])*dpg.create_translation_matrix([planet1_distance, 0]))
dpg.apply_transform("planet 1, moon node", dpg.create_rotation_matrix(math.pi*planet1_
↪moonangle/180.0 , [0, 0, -1])*dpg.create_translation_matrix([planet1_moondistance, 0]))
dpg.apply_transform("planet node 2", dpg.create_rotation_matrix(math.pi*planet2_angle/
↪180.0 , [0, 0, -1])*dpg.create_translation_matrix([planet2_distance, 0]))
dpg.apply_transform("planet 2, moon 1 node", dpg.create_rotation_matrix(math.pi*planet2_
↪moon1distance/180.0 , [0, 0, -1])*dpg.create_translation_matrix([planet2_moon1distance,
↪ 0]))
dpg.apply_transform("planet 2, moon 2 node", dpg.create_rotation_matrix(math.pi*planet2_
↪moon2angle/180.0 , [0, 0, -1])*dpg.create_translation_matrix([planet2_moon2distance,␣
↪0]))

dpg.show_viewport()
while dpg.is_dearpygui_running():
    dpg.render_dearpygui_frame()
```

(continues on next page)

(continued from previous page)

```
dpg.destroy_context()
```

3.13.5 3D Operations

New in 1.1. Version *1.1* added 3 new options to layers, **perspective_divide**, **depth_clipping**, and **cull_mode**.

When perspective divide is set to **True**, the x, y, and z components of each point are divided by the w component after transformation.

When depth clipping is set to **True**, points will be clipped when they are outside the clip space set using **set_clip_space**. Setting the clip space will scale and transform points. Scaling is based on normalized coordinates (use perspective or othographic matrices).

Cull mode is used to activate front/back face culling.

Matrices are column major. Post-multiplication is used, for example to scale, then rotate, then transform you use: Transform = Translate * Rotate * Scale.

The following matrix helper functions are provided: - **create_rotation_matrix** - **create_translation_matrix** - **create_scale_matrix** - **create_lookat_matrix** - **create_perspective_matrix** - **create_orthographic_matrix** - **create_fps_matrix**

Code

```
import dearpygui.dearpygui as dpg
import math

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

size = 5
verticies = [
        [-size, -size, -size],  # 0 near side
        [ size, -size, -size],  # 1
        [-size,  size, -size],  # 2
        [ size,  size, -size],  # 3
        [-size, -size,  size],  # 4 far side
        [ size, -size,  size],  # 5
        [-size,  size,  size],  # 6
        [ size,  size,  size],  # 7
        [-size, -size, -size],  # 8 left side
        [-size,  size, -size],  # 9
        [-size, -size,  size],  # 10
        [-size,  size,  size],  # 11
        [ size, -size, -size],  # 12 right side
        [ size,  size, -size],  # 13
        [ size, -size,  size],  # 14
        [ size,  size,  size],  # 15
        [-size, -size, -size],  # 16 bottom side
        [ size, -size, -size],  # 17
        [-size, -size,  size],  # 18
        [ size, -size,  size],  # 19
```

(continues on next page)

(continued from previous page)

```
        [-size,  size, -size],  # 20 top side
        [ size,  size, -size],  # 21
        [-size,  size,  size],  # 22
        [ size,  size,  size],  # 23
    ]

colors = [
        [255,   0,   0, 150],
        [255, 255,   0, 150],
        [255, 255, 255, 150],
        [255,   0, 255, 150],
        [  0, 255,   0, 150],
        [  0, 255, 255, 150],
        [  0,   0, 255, 150],
        [  0, 125,   0, 150],
        [128,   0,   0, 150],
        [128,  70,   0, 150],
        [128, 255, 255, 150],
        [128,   0, 128, 150]
    ]

with dpg.window(label="tutorial", width=550, height=550):

    with dpg.drawlist(width=500, height=500):

        with dpg.draw_layer(tag="main pass", depth_clipping=True, perspective_
↪divide=True, cull_mode=dpg.mvCullMode_Back):

            with dpg.draw_node(tag="cube"):

                dpg.draw_triangle(vertices[1],  vertices[2],  vertices[0], color=[0,0,
↪0.0],  fill=colors[0])
                dpg.draw_triangle(vertices[1],  vertices[3],  vertices[2], color=[0,0,
↪0.0],  fill=colors[1])
                dpg.draw_triangle(vertices[7],  vertices[5],  vertices[4], color=[0,0,
↪0.0],  fill=colors[2])
                dpg.draw_triangle(vertices[6],  vertices[7],  vertices[4], color=[0,0,
↪0.0],  fill=colors[3])
                dpg.draw_triangle(vertices[9],  vertices[10], vertices[8], color=[0,0,
↪0.0],  fill=colors[4])
                dpg.draw_triangle(vertices[9],  vertices[11], vertices[10], color=[0,
↪0,0.0], fill=colors[5])
                dpg.draw_triangle(vertices[15], vertices[13], vertices[12], color=[0,
↪0,0.0], fill=colors[6])
                dpg.draw_triangle(vertices[14], vertices[15], vertices[12], color=[0,
↪0,0.0], fill=colors[7])
                dpg.draw_triangle(vertices[18], vertices[17], vertices[16], color=[0,
↪0,0.0], fill=colors[8])
                dpg.draw_triangle(vertices[19], vertices[17], vertices[18], color=[0,
↪0,0.0], fill=colors[9])
                dpg.draw_triangle(vertices[21], vertices[23], vertices[20], color=[0,
↪0,0.0], fill=colors[10])
```

(continues on next page)

(continued from previous page)

```
                dpg.draw_triangle(verticies[23], verticies[22], verticies[20], color=[0,
↪0,0.0], fill=colors[11])

x_rot = 10
y_rot = 45
z_rot = 0

view = dpg.create_fps_matrix([0, 0, 50], 0.0, 0.0)
proj = dpg.create_perspective_matrix(math.pi*45.0/180.0, 1.0, 0.1, 100)
model = dpg.create_rotation_matrix(math.pi*x_rot/180.0 , [1, 0, 0])*\
                        dpg.create_rotation_matrix(math.pi*y_rot/180.0 , [0, 1, 0])*\
                        dpg.create_rotation_matrix(math.pi*z_rot/180.0 , [0, 0, 1])

dpg.set_clip_space("main pass", 0, 0, 500, 500, -1.0, 1.0)
dpg.apply_transform("cube", proj*view*model)

dpg.show_viewport()
while dpg.is_dearpygui_running():
    dpg.render_dearpygui_frame()

dpg.destroy_context()
```

3.13.6 3D Operations Limitations

New in 1.1. The drawing API 3D operations are not hardware accelerated (this will be introduced with DearPy3D). This API is for 'light' 3D operations. There are a few issues you may come across while performing 3D operation with the drawing API.

Issue 1: Z ordering

With the current API, users are responsible for correct Z ordering. The recommended way to address this is to use "painter's algorithm". Basically, just order the items in proper depth order.

This is not the best solution overall. We would prefer to use proper depth buffering but this requires pixel level control which is not practical with this API.

Issue 2: Perspective Texture Correction

When using draw_image_quad, viewing the image at sharp angles will deform the image. This is due to texture coordinates being linearly interpolated in normalized device coordinate space.

There is currently no practical solution but an attempt could be made to split the quad into several smaller quads. The actual solution requires pixel level control which is not practical with this API.

Issue 3: Culling

Currently, culling is only setup for triangles.

Dear Py3D and Software Renderer

All of the above issues will resolved in **Dear Py3D**. Before **Dear Py3D**, we will also be introducing a software renderer for **Dear PyGui** that will resolve the above issues.

3.14 File & Directory Selector

The file dialog item 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 run. An optional second callback, to be run when the cancel button is clicked, can be provided as a keyword argument.

When OK is clicked, information is passed through the app_data argument such as: * file path * file name * current path * current filter (the file type filter)

The simplest case is as a director picker. Below is the example

Code

```
import dearpygui.dearpygui as dpg

dpg.create_context()

def callback(sender, app_data):
    print('OK was clicked.')
    print("Sender: ", sender)
    print("App Data: ", app_data)

def cancel_callback(sender, app_data):
    print('Cancel was clicked.')
    print("Sender: ", sender)
    print("App Data: ", app_data)

dpg.add_file_dialog(
    directory_selector=True, show=False, callback=callback, tag="file_dialog_id",
    cancel_callback=cancel_callback, width=700 ,height=400)

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

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

3.14.1 File Extensions

File extensions are 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

dpg.create_context()

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

with dpg.file_dialog(directory_selector=False, show=False, callback=callback, id="file_
↪dialog_id", width=700 ,height=400):
    dpg.add_file_extension(".*")
    dpg.add_file_extension("", color=(150, 255, 150, 255))
    dpg.add_file_extension("Source files (*.cpp *.h *.hpp){.cpp,.h,.hpp}", color=(0, 255,
↪ 255, 255))
    dpg.add_file_extension(".h", color=(255, 0, 255, 255), custom_text="[header]")
    dpg.add_file_extension(".py", color=(0, 255, 0, 255), custom_text="[Python]")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.14.2 Customizing

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

This can allow the creation of a pinned menu, favorites, directory tree, and much more.

Below is an example:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

with dpg.file_dialog(directory_selector=False, show=False, callback=callback, tag="file_
↪dialog_tag", width=700 ,height=400):
    dpg.add_file_extension(".*")
    dpg.add_file_extension("", color=(150, 255, 150, 255))
    dpg.add_file_extension(".cpp", color=(255, 255, 0, 255))
```

(continues on next page)

(continued from previous page)

```
    dpg.add_file_extension(".h", color=(255, 0, 255, 255))
    dpg.add_file_extension(".py", color=(0, 255, 0, 255))

    with dpg.group(horizontal=True):
        dpg.add_button(label="fancy file dialog")
        dpg.add_button(label="file")
        dpg.add_button(label="dialog")
    dpg.add_date_picker()
    with dpg.child_window(height=100):
        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_tag
↪"))

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.14.3 Selecting Multiple Files

You can select multiple files by setting the *file_count* keyword **Must use Ctrl + click to select multiple files Must use Shift + click to select multiple files**

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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, tag="file_dialog_tag", width=700 ,height=400):
    dpg.add_file_extension("", color=(255, 150, 150, 255))
    dpg.add_file_extension(".*")
    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_window(width=100):
        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):
```

(continues on next page)

(continued from previous page)

```
    dpg.add_button(label="File Selector", callback=lambda: dpg.show_item("file_dialog_tag
↪"))

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.15 Filter Set

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

Most items have a *filter_key* keyword that can be set when creating the item. Filtering is based on the set value of the filter set.

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

Code

```
import dearpygui.dearpygui as dpg

dpg.create_context()

def callback(sender, filter_string):
    dpg.set_value("filter_id", filter_string)

with dpg.window(label="about", width =500, height=300):
    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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Note:

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

- Hide lines containing xxx with *"-xxx"*

3.16 Fonts

DPG 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 DPG in an application aiming to look nice and wanting to support multiple resolutions.

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

3.16.1 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_font_manager` to browse your fonts and understand what's going on if you have an issue.

3.16.2 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

dpg.create_context()

# add a font registry
with dpg.font_registry():
    # first argument ids the path to the .ttf or .otf file
    default_font = dpg.add_font("NotoSerifCJKjp-Medium.otf", 20)
    second_font = dpg.add_font("NotoSerifCJKjp-Medium.otf", 10)

with dpg.window(label="Font Example", height=200, width=200):
    dpg.add_button(label="Default font")
    b2 = dpg.add_button(label="Secondary font")
    dpg.add_button(label="default")

    # set font of specific widget
    dpg.bind_font(default_font)
    dpg.bind_item_font(b2, second_font)

dpg.show_font_manager()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.16.3 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

dpg.create_context()

with dpg.font_registry():
    with dpg.font("NotoSerifCJKjp-Medium.otf", 20) as font1:

        # 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.show_font_manager()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.16.4 Where to find unicode character codes?

Unicode Characters

3.17 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 tag of the window. Make sure the tag does not change between sessions by generating the tag beforehand or specifying it as a string.

3.17.1 Creating init files

Use `save_init_file` while your application is running.

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

3.17.2 Loading init files

Use `configure_app` before creating the viewport.

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

- Position the windows
- Press the save button and the init file will be saved in the current working directory
- Restart the app and see your windows in the previous positions.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

def save_init():
    dpg.save_init_file("dpg.ini")

dpg.configure_app(init_file="dpg.ini")  # default file is 'dpg.ini'
```

(continues on next page)

(continued from previous page)

```
with dpg.window(label="about", tag="main window"):
    dpg.add_button(label="Save Window pos", callback=lambda: save_init)

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.18 Menu Bar

DPG contains two types of Menu Bars

Viewport Menu Bar Attached to the viewport

Menu Bar: Attached to a specified window

Other than being attached to different parents they both act the same.

A typical menu bar consists of the following components:

Menu Bar: The main menu ribbon. Used to contain menus.

Menu: Popup windows that are used to contain items in a collapsable fashion.

Menu Item: Items that can run callbacks and display a checkmark.

3.18.1 Usage

Items added to the Menu Bar are displayed from left to right. Items added to the menus are displayed from top to bottom.

Menus can be nested inside other menus.

Any widget can be added to a menu.

Viewport Menu Bar Example

```
import dearpygui.dearpygui as dpg

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

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

with dpg.viewport_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)
```

(continues on next page)

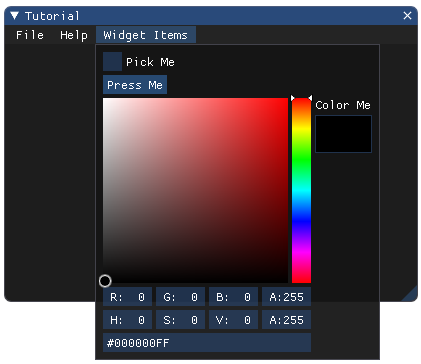
(continued from previous page)

```
        with dpg.menu(label="Settings"):
            dpg.add_menu_item(label="Setting 1", callback=print_me, check=True)
            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.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Menu Bar Example

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

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, check=True)
                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.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



3.19 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

Attributes can contain any UI Items. When a user clicks and drags a node's attribute the node editor's callback is ran. DPG sends the attributes' tags through the _app_data_ argument of the callback.

It's 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 the link and dragging it.

Code

```
import dearpygui.dearpygui as dpg

dpg.create_context()

# 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):
```

(continues on next page)

(continued from previous page)

```
    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"):
            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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.19.1 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)
```

3.19.2 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

3.19.3 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

3.19.4 Associated Items

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

3.20 Plots

Plots are composed of multiple components.

Y-axis: This is a container and is the parent of all the data series that are added to the plot. The plot can have multiple Y-axis at one time (up to 3).

X-axis: This is the x data scale (only 1 x axis is allowed).

Series: These are the containers for the data you wish to display. Data series need to be added as a child of a Y-axis to be displayed on the plot. There are many different types of data series avaliable. Series also can contain UI Items that will be displayed when right-clicking the series label in the legend as a context menu.

Legend (optional): This is a normal legend and alos allows the user to toggle which data series are visible.

Plots have some functionality built in:

Toggle Data Series: Click on the legend name of the desired data series to toggle

Settings: Double Right Click

Pan Plot: Click & Drag on plot

Pan Axis: Click & Drag on Axis

Zoom: Scroll Mouse Wheel

Zoom Axis: Hover Axis & Scroll Mouse Wheel

Zoom Region: Right Click & Drag

Zoom Extents: Double Click

Zoom Axis Area: Shift + Right Click & Drag

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

```

(continues on next page)

(continued from previous page)

```
dpg.create_context()

# creating data
sindatax = []
sindatay = []
for i in range(0, 500):
    sindatax.append(i / 1000)
    sindatay.append(0.5 + 0.5 * sin(50 * i / 1000))

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", tag="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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.1 Updating Series Data

You can change a series on a plot by

- setting the series value
- deleting that specific series item from they y-axis and adding it again
- deleting all the series items from they y-axis and adding that specific series again

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

dpg.create_context()

sindatax = []
sindatay = []
for i in range(0, 500):
    sindatax.append(i / 1000)
    sindatay.append(0.5 + 0.5 * sin(50 * i / 1000))

def update_series():
```

(continues on next page)

(continued from previous page)

```
    cosdatax = []
    cosdatay = []
    for i in range(0, 500):
        cosdatax.append(i / 1000)
        cosdatay.append(0.5 + 0.5 * cos(50 * i / 1000))
    dpg.set_value('series_tag', [cosdatax, cosdatay])
    dpg.set_item_label('series_tag', "0.5 + 0.5 * cos(x)")

with dpg.window(label="Tutorial", tag="win"):
    dpg.add_button(label="Update Series", callback=update_series)
    # 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", tag="y_axis")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.2 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(...)*
- *set_axis_limits_constraints(...)*
- *reset_axis_limits_constraints(...)*
- *set_axis_zoom_constraints(...)*
- *reset_axis_zoom_constraints(...)*

An example demonstrating some of this can be found below:

```
import dearpygui.dearpygui as dpg

dpg.create_context()
```

(continues on next page)

(continued from previous page)

```
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("y_axis"))
        dpg.add_button(label="unlock x limits", callback=lambda: dpg.set_axis_limits_
↪auto("x_axis"))
        dpg.add_button(label="unlock y limits", callback=lambda: dpg.set_axis_limits_
↪auto("y_axis"))
        dpg.add_button(label="print limits x", callback=lambda: print(dpg.get_axis_
↪limits("x_axis")))
        dpg.add_button(label="print limits y", callback=lambda: print(dpg.get_axis_
↪limits("y_axis")))

    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, tag="x_axis")
        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", tag="y_axis")
        dpg.set_axis_limits("y_axis", 0, 110)

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.3 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

dpg.create_context()

with dpg.window(label="Tutorial", width=400, height=400):
    with dpg.plot(label="Bar Series", height=-1, width=-1):
        dpg.add_plot_legend()
```

(continues on next page)

(continued from previous page)

```
        # create x axis
        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", tag="yaxis_tag")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.4 Multiple Y Axes

Plots can contain up to Three Y-axis for different data that needs a different scale.

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

dpg.create_context()

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.mvYAxis2, label="y2")
        dpg.add_stem_series(sindatax, sindatay, label="y2 stem", parent=dpg.last_item())
```

(continues on next page)

(continued from previous page)

```
        # create y axis 3
        dpg.add_plot_axis(dpg.mvYAxis3, label="y3 scatter")
        dpg.add_scatter_series(sindatax, sindatay, label="y3", parent=dpg.last_item())

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.5 Annotations

Annotations can be used to mark locations on a plot.

Annotations are owned by the plot and their coordinates correspond to the 1st y axis.

They are clamped by default.

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

dpg.create_context()


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.create_viewport(title='Custom Title', width=800, height=600)
```

(continues on next page)

(continued from previous page)

```
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.6 Drag Points and Lines

Drag lines/points are owned by the plot and their coordinates 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!

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

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")
        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,␣
↪callback=print_val)
        dpg.add_drag_line(label="dline2", color=[255, 255, 0, 255], vertical=False,␣
↪default_value=-2, callback=print_val)
        dpg.add_drag_point(label="dpoint1", color=[255, 0, 255, 255], default_value=(1.0,
↪ 1.0), callback=print_val)
        dpg.add_drag_point(label="dpoint2", color=[255, 0, 255, 255], default_value=(-1.
↪0, 1.0), callback=print_val)

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.7 Querying

Querying allows the user to select a region of the plot by holding with the right mouse button and clicking with the left one.

Querying requires setting *query* to **True** when creating the plot.

The callback of the plot will run when the plot is being queried.

All the query areas are sent through the *app_data* argument as *[(x_min, x_max, y_min, y_max), (x_min, x_max, y_min, y_max), ...]*.

It is also possible to poll the plot for the query areas by calling: `get_plot_query_rects` and

Below is an example using the callback

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

dpg.create_context()

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=600):
    dpg.add_text("Click and drag the middle mouse button over the top plot!")


    def query(sender, app_data, user_data):
        dpg.set_axis_limits("xaxis_tag2", app_data[0], app_data[1])
        dpg.set_axis_limits("yaxis_tag2", app_data[2], app_data[3])


    # plot 1
    with dpg.plot(no_title=True, height=200, 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=200, no_menus=True, width=-1):
        dpg.add_plot_axis(dpg.mvXAxis, label="x1", tag="xaxis_tag2")
        dpg.add_plot_axis(dpg.mvYAxis, label="y1", tag="yaxis_tag2")
        dpg.add_line_series(sindatax, sindatay, parent="yaxis_tag2")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.8 Custom Context Menus

Plot series are actually containers!

Adding UI Items to a plot series, they will show up when right-clicking the series in the legend.

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

dpg.create_context()

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", tag="yaxis")

        # series 1
        dpg.add_line_series(sindatax, sindatay, label="series 1", parent="yaxis", tag=
↪"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", tag=
↪"series_2")
        dpg.add_button(label="Delete Series 2", parent=dpg.last_item(), callback=lambda:␣
↪dpg.delete_item("series_2"))

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.9 Custom Series

New in 1.6. Custom series allow you to control the way a series is rendered.

A custom series can currently have between 2 and 5 channels. A channel is an array/list of data. Each channel must be the same length. The first 2 channels and channel count are required arguments. Additional channels can be provided with the y1, y2, and y3 keywords. You must also set the "callback" keyword. The second argument will be provided by DPG as a list. The first item being useful information. The following items are the original data sent in but transformed into pixel space. The combination of all this information can be used to create a custom series. See simple example below:

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

x_data = [0.0, 1.0, 2.0, 4.0, 5.0]
y_data = [0.0, 10.0, 20.0, 40.0, 50.0]

def callback(sender, app_data):

    _helper_data = app_data[0]
    transformed_x = app_data[1]
    transformed_y = app_data[2]
    #transformed_y1 = app_data[3] # for channel = 3
    #transformed_y2 = app_data[4] # for channel = 4
    #transformed_y3 = app_data[5] # for channel = 5
    mouse_x_plot_space = _helper_data["MouseX_PlotSpace"]   # not used in this example
    mouse_y_plot_space = _helper_data["MouseY_PlotSpace"]   # not used in this example
    mouse_x_pixel_space = _helper_data["MouseX_PixelSpace"]
    mouse_y_pixel_space = _helper_data["MouseY_PixelSpace"]
    dpg.delete_item(sender, children_only=True, slot=2)
    dpg.push_container_stack(sender)
    dpg.configure_item("demo_custom_series", tooltip=False)
    for i in range(0, len(transformed_x)):
        dpg.draw_text((transformed_x[i]+15, transformed_y[i]-15), str(i), size=20)
        dpg.draw_circle((transformed_x[i], transformed_y[i]), 15, fill=(50+i*5, 50+i*50,␣
↪0, 255))
        if mouse_x_pixel_space < transformed_x[i]+15 and mouse_x_pixel_space >␣
↪transformed_x[i]-15 and mouse_y_pixel_space > transformed_y[i]-15 and mouse_y_pixel_
↪space < transformed_y[i]+15:
            dpg.draw_circle((transformed_x[i], transformed_y[i]), 30)
            dpg.configure_item("demo_custom_series", tooltip=True)
            dpg.set_value("custom_series_text", "Current Point: " + str(i))
    dpg.pop_container_stack()

with dpg.window(label="Tutorial") as win:
    dpg.add_text("Hover an item for a custom tooltip!")
    with dpg.plot(label="Custom Series", height=400, width=-1):
        dpg.add_plot_legend()
        xaxis = dpg.add_plot_axis(dpg.mvXAxis)
        with dpg.plot_axis(dpg.mvYAxis):
            with dpg.custom_series(x_data, y_data, 2, label="Custom Series",␣
↪callback=callback, tag="demo_custom_series"):
```

(continues on next page)

(continued from previous page)

```
                dpg.add_text("Current Point: ", tag="custom_series_text")
            dpg.fit_axis_data(dpg.top_container_stack())

dpg.set_primary_window(win, True)
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.10 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*

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

dpg.create_context()

sindatax = []
sindatay = []
for i in range(0, 100):
    sindatax.append(i / 100)
    sindatay.append(0.5 + 0.5 * sin(50 * i / 100))
sindatay2 = []
for i in range(0, 100):
    sindatay2.append(2 + 0.5 * sin(50 * i / 100))

with dpg.window(label="Tutorial", width=500, height=400):
    # create a theme for the plot
    with dpg.theme(tag="plot_theme"):
        with dpg.theme_component(dpg.mvStemSeries):
            dpg.add_theme_color(dpg.mvPlotCol_Line, (150, 255, 0), category=dpg.
↪mvThemeCat_Plots)
            dpg.add_theme_style(dpg.mvPlotStyleVar_Marker, dpg.mvPlotMarker_Diamond,␣
↪category=dpg.mvThemeCat_Plots)
            dpg.add_theme_style(dpg.mvPlotStyleVar_MarkerSize, 7, category=dpg.
↪mvThemeCat_Plots)

        with dpg.theme_component(dpg.mvScatterSeries):
            dpg.add_theme_color(dpg.mvPlotCol_Line, (60, 150, 200), category=dpg.
↪mvThemeCat_Plots)
            dpg.add_theme_style(dpg.mvPlotStyleVar_Marker, dpg.mvPlotMarker_Square,␣
↪category=dpg.mvThemeCat_Plots)
            dpg.add_theme_style(dpg.mvPlotStyleVar_MarkerSize, 4, category=dpg.
↪mvThemeCat_Plots)

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

(continues on next page)

(continued from previous page)

```
        # 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", tag="yaxis")

        # series belong to a y axis
        dpg.add_stem_series(sindatax, sindatay, label="0.5 + 0.5 * sin(x)", parent="yaxis
↪", tag="series_data")
        dpg.add_scatter_series(sindatax, sindatay2, label="2 + 0.5 * sin(x)", parent=
↪"yaxis", tag="series_data2")

        # apply theme to series
        dpg.bind_item_theme("series_data", "plot_theme")
        dpg.bind_item_theme("series_data2", "plot_theme")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.20.11 Colormaps

Under construction

3.20.12 Gallery

3.21 Popups

Popups are windows that disappear when clicked outside of the popup's border.

They are typically used as context menus when right-clicking an item or as dialogs.

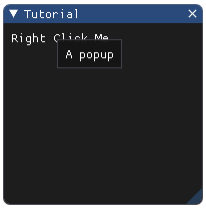
In DPG, popups are just windows with *popup* set to **True**, *show* set to **False**, and a *clicked_handler* attached to a widget that shows the window when clicked.

Normally when used, a popup will be shown until you click away from it. By default, a right click activates the popup.

Code

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Tutorial"):
    dpg.add_text("Right Click Me")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



3.21.1 Modal Usage

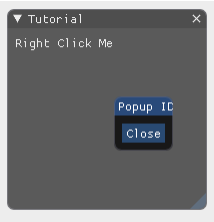
When the modal keyword is set to **True**, the popup will be modal.

This prevents the user from interacting with other windows until the popup is closed. You must hide or delete the popup to remove it.

Code

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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, tag=
↪"modal_id"):
        dpg.add_button(label="Close", callback=lambda: dpg.configure_item("modal_id",␣
↪show=False))

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



Mouse Button options include:

- _mvMouseButton_Right_
- _mvMouseButton_Left_
- _mvMouseButton_Middle_
- _mvMouseButton_X1_
- _mvMouseButton_X2_

3.21.2 Window as Dialog Popup

This is an example of a window made into a typical dialog.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Delete Files", modal=True, show=False, tag="modal_id", no_title_
↪bar=True):
    dpg.add_text("All those beautiful files will be deleted.\nThis operation cannot be␣
↪undone!")
    dpg.add_separator()
    dpg.add_checkbox(label="Don't ask me next time")
    with dpg.group(horizontal=True):
        dpg.add_button(label="OK", width=75, callback=lambda: dpg.configure_item("modal_
↪id", show=False))
        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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.22 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

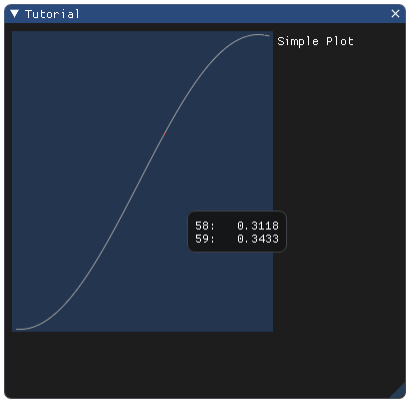
```
import dearpygui.dearpygui as dpg

dpg.create_context()

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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

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

dpg.create_context()


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,␣
↪tag="plot")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



3.23 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.

Staged items will show up in the item registry.

Items can be moved out of staging by using `move_item`.

The most basic example can be found below:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

def stage_items():
    with dpg.stage(tag="stage1"):
        dpg.add_text("hello, i was added from a stage", tag="text_tag")

def present_stage_items():
    dpg.move_item("text_tag", parent="main_win")

with dpg.window(label="Tutorial", tag="main_win"):
    dpg.add_button(label="stage items", callback=stage_items)
    dpg.add_button(label="present stages items", callback=present_stage_items)

dpg.show_item_registry()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Prefered way to "unstage" items is using `unstage`. This will place the items as if they were newly created items according to the standard rules of *Container Stack*.

Also using the unstage command will automatically clean up the stage container.

Using push_container_stack and pop_container_stack is recomended here as it provides better performance when unstaging.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

def stage_items():
    with dpg.stage(tag="stage1"):
        dpg.add_text("hello, i was added from a stage")
        dpg.add_text("hello, i was added from a stage")
        dpg.add_text("hello, i was added from a stage")
        dpg.add_text("hello, i was added from a stage")
        dpg.add_text("hello, i was added from a stage")

def present_stage_items():
    dpg.push_container_stack("main_win")
    dpg.unstage("stage1")
    dpg.pop_container_stack()

with dpg.window(label="Tutorial", tag="main_win", height=400, width=400):
    dpg.add_button(label="stage items", callback=stage_items)
    dpg.add_button(label="present stages items", callback=present_stage_items)

dpg.show_item_registry()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.23.1 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 are 2 examples:

Example 1

```
import dearpygui.dearpygui as dpg

dpg.create_context()


class Button:

    def __init__(self, label):
        with dpg.stage() as self._staging_container_id:
            self._id = dpg.add_button(label=label)
```

(continues on next page)

(continued from previous page)

```
    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, parent):
        dpg.push_container_stack(parent)
        dpg.unstage(self._staging_container_id)
        dpg.pop_container_stack()

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", tag="main_win"):
    dpg.add_text("hello world")

my_button.submit("main_win")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Example 2

```
import dearpygui.dearpygui as dpg

dpg.create_context()


class Window:

    def __init__(self, label):
        self._children = []
        with dpg.stage() as stage:
            self.id = dpg.add_window(label=label)
        self.stage = stage

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

    def submit(self):
        dpg.unstage(self.stage)


class Button:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, label):
        with dpg.stage():
            self.id = dpg.add_button(label=label)

    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.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24 Tables

The table API is a low level API that can be used to create tables.

It can also be used as a layout mechanism. Tables are composed of multiple components which include columns, rows, cells, and the actual 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

dpg.create_context()

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
```

(continues on next page)

(continued from previous page)

```
        for i in range(0, 4):
            with dpg.table_row():
                for j in range(0, 3):
                    dpg.add_text(f"Row{i} Column{j}")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Note: The maximum number of columns is 64.

multiple items can go into a single cell by creating a cell as shown

Code

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

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

        # once it reaches the end of the columns
        for i in range(0, 4):
            with dpg.table_row():
                for j in range(0, 3):
                    with dpg.table_cell():
                        dpg.add_button(label=f"Row{i} Column{j} a")
                        dpg.add_button(label=f"Row{i} Column{j} b")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24.1 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

dpg.create_context()

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

    with dpg.table(header_row=False, row_background=True,
                   borders_innerH=True, borders_outerH=True, borders_innerV=True,
                   borders_outerV=True):

        # 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):
            with dpg.table_row():
                for j in range(0, 3):
                    dpg.add_text(f"Row{i} Column{j}")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24.2 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

dpg.create_context()

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

    with dpg.table(header_row=True):

        # use add_table_column to add columns to the table,
        # table columns use slot 0
```

(continues on next page)

(continued from previous page)

```
        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):
            with dpg.table_row():
                for j in range(0, 3):
                    dpg.add_text(f"Row{i} Column{j}")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24.3 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

3.24.4 Stretch

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

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

    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")

        # add_table_next_column will jump to the next row
        # once it reaches the end of the columns
```

(continues on next page)

(continued from previous page)

```
        # table next column use slot 1
        for i in range(0, 4):
            with dpg.table_row():
                for j in range(0, 3):
                    dpg.add_text(f"Row{i} Column{j}")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Fixed

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

```
import dearpygui.dearpygui as dpg

dpg.create_context()

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

    # 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")
        dpg.add_table_column(label="Header 3")
        for i in range(0, 4):
            with dpg.table_row():
                for j in range(0, 3):
                    dpg.add_button(label=f"Row{i} Column{j} a")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

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

dpg.create_context()

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

    with dpg.table(header_row=True, policy=dpg.mvTable_SizingFixedFit, row_
↪background=True, reorderable=True,
```

(continues on next page)

(continued from previous page)

```
                    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):
            with dpg.table_row():
                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}")

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24.5 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	Default 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.

3.24.6 Sorting

DPG does not actually do any sorting for you. The table API is a more general purpose API and it is up to the library user to handle sorting. To sort a table based on user interaction, you must assign a callback to the table. This callback will be ran when a user tries to sort a table by clicking on the table's column headers.

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

def sort_callback(sender, sort_specs):

    # sort_specs scenarios:
    #   1. no sorting -> sort_specs == None
    #   2. single sorting -> sort_specs == [[column_id, direction]]
    #   3. multi sorting -> sort_specs == [[column_id, direction], [column_id,␣
↪direction], ...]
    #
    # notes:
    #   1. direction is ascending if == 1
    #   2. direction is ascending if == -1
```

(continues on next page)

(continued from previous page)

```
    # no sorting case
    if sort_specs is None: return

    rows = dpg.get_item_children(sender, 1)

    # create a list that can be sorted based on first cell
    # value, keeping track of row and value used to sort
    sortable_list = []
    for row in rows:
        first_cell = dpg.get_item_children(row, 1)[0]
        sortable_list.append([row, dpg.get_value(first_cell)])

    def _sorter(e):
        return e[1]

    sortable_list.sort(key=_sorter, reverse=sort_specs[0][1] < 0)

    # create list of just sorted row ids
    new_order = []
    for pair in sortable_list:
        new_order.append(pair[0])

    dpg.reorder_items(sender, 1, new_order)

with dpg.window(label="Tutorial", width=500):

    with dpg.table(header_row=True, borders_innerH=True, borders_outerH=True,
                   borders_innerV=True, borders_outerV=True, row_background=True,
                   sortable=True, callback=sort_callback):

        dpg.add_table_column(label="One")
        dpg.add_table_column(label="Two", no_sort=True)

        for i in range(25):
            with dpg.table_row():
                dpg.add_input_int(label=" ", step=0, default_value=i)
                dpg.add_text(f"Cell {i}, 1")

# main loop
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24.7 Scrolling

Under construction

3.24.8 Selecting

You can make rows and/or cells selectable by adding a *selectable* to the table and assigning a callback to it. Use a theme to control the hover style. The *span_columns* option of the *selectable* is used to control whether the row or the cell is selectable.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.theme() as table_theme:
    with dpg.theme_component(dpg.mvTable):
        # dpg.add_theme_color(dpg.mvThemeCol_HeaderHovered, (255, 0, 0, 100),␣
↪category=dpg.mvThemeCat_Core)
        dpg.add_theme_color(dpg.mvThemeCol_HeaderActive, (0, 0, 0, 0), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_color(dpg.mvThemeCol_Header, (0, 0, 0, 0), category=dpg.mvThemeCat_
↪Core)

def clb_selectable(sender, app_data, user_data):
    print(f"Row {user_data}")

with dpg.window(tag="Selectable Tables"):
    with dpg.table(tag="SelectRows", header_row=True) as selectablerows:
        dpg.add_table_column(label="First")
        dpg.add_table_column(label="Second")
        dpg.add_table_column(label="Third")

        for i in range(15):
            with dpg.table_row():
                for j in range(3):
                    dpg.add_selectable(label=f"Row{i} Column{j}", span_columns=True,␣
↪callback=clb_selectable, user_data=i)
    dpg.bind_item_theme(selectablerows, table_theme)

    with dpg.table(tag="SelectCells", header_row=True) as selectablecells:
        dpg.add_table_column(label="First")
        dpg.add_table_column(label="Second")
        dpg.add_table_column(label="Third")

        for i in range(15):
            with dpg.table_row():
                for j in range(3):
                    dpg.add_selectable(label=f"Row{i} Column{j}", callback=clb_
↪selectable, user_data=(i,j))
    dpg.bind_item_theme(selectablecells, table_theme)

dpg.create_viewport(width=800, height=600)
```

(continues on next page)

(continued from previous page)

```
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24.9 Clipping

Using a clipper can help performance with large tables.

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

dpg.create_context()


def clipper_toggle(sender):
    dpg.configure_item("table_clip", clipper=dpg.get_value(sender))


with dpg.window(label="Tutorial"):
    dpg.add_checkbox(label="clipper", callback=clipper_toggle, default_value=True)

    with dpg.table(header_row=False, tag="table_clip", clipper=True):

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

        for i in range(30000):
            with dpg.table_row():
                for j in range(5):
                    dpg.add_text(f"Row{i} Column{j}")

dpg.show_metrics()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.24.10 Filtering

Under construction

3.24.11 Padding

Under construction

3.24.12 Outer Size

Under construction

3.24.13 Column Widths

Under construction

3.24.14 Row Height

Under construction

3.24.15 Search Delay

Under construction

3.25 Textures & Images

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.

Using the keyword Show on the texture registry will open the texture registry.

3.25.1 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

dpg.create_context()

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(show=True):
    dpg.add_static_texture(width=100, height=100, default_value=texture_data, tag=
↪"texture_tag")

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


dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

The texture can be deleted with *dpg.delete_item("texture_tag")*. However, for the tag/alias to be released items that use *"texture_tag"* (such as a plot series) must also be deleted.

3.25.2 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

dpg.create_context()

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(show=True):
    dpg.add_dynamic_texture(width=100, height=100, default_value=texture_data, tag=
↪"texture_tag")
```

(continues on next page)

(continued from previous page)

```
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_tag", new_texture_data)


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


dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.25.3 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

dpg.create_context()


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

(continues on next page)

(continued from previous page)

```
    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(show=True):
    dpg.add_raw_texture(width=100, height=100, default_value=raw_data, format=dpg.
↪mvFormat_Float_rgba, tag="texture_tag")


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_tag")
    dpg.add_color_picker((255, 0, 255, 255), label="Texture",
                         no_side_preview=True, alpha_bar=True, width=200,
                         callback=update_dynamic_texture)


dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.25.4 Formats

The following formats are currently supported

Format	Static Texture	Dynamic Texture	Raw Texture
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.

3.25.5 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

dpg.create_context()

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

with dpg.texture_registry(show=True):
    dpg.add_static_texture(width=width, height=height, default_value=data, tag="texture_
↪tag")

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


dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.25.6 Saving Images

New in 1.4. DPG provides the function `save_image` for saving image data to a file.

The image is a rectangle of pixels stored from left-to-right, top-to-bottom. Each pixel contains up to 4 components of data interleaved with 8-bits per channel, in the following order: 1=Y, 2=YA, 3=RGB, 4=RGBA. (Y is monochrome color.)

PNG creates output files with the same number of components as the input. The BMP format expands Y to RGB in the file format and does not output alpha.

Additional options will will be released with v1.4.1.

The accepted file types include:

- PNG
- JPG (new in v1.4.1)
- BMP (new in v1.4.1)
- TGA (new in v1.4.1)
- HDR (new in v1.4.1)

File type is determined by extension. Must be lowercase (png, jpg, bmp, tga, hdr).

A simple example can be found below

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

width, height = 255, 255

data = []
for i in range(width*height):
    data.append(255)
    data.append(255)
    data.append(0)

with dpg.window(label="Tutorial"):
    dpg.add_button(label="Save Image", callback=lambda:dpg.save_image(file="newImage.png
↪", width=width, height=height, data=data, components=3))

dpg.show_viewport()
while dpg.is_dearpygui_running():
    dpg.render_dearpygui_frame()

dpg.destroy_context()
```

3.26 Themes

Themes are containers which are composed of:

Theme Components: containers within a theme that can specify an item type the theme colors/styles target

Theme Colors: items that are added to a theme component and set colors

Theme Styles: items that are added to a theme component and set styles

The theme can be:

- bound as the default theme. This will have a global effect across all windows and propagate.
- bound to a container. This will propagate to its children if applicable theme components are in the theme.
- bound to an item type if applicable theme components are in the theme.

Theme components must have a specified item type. This can either be *mvAll* for all items or a specific item type.

Style and color items have a named constant and will apply that constant to their components named item type. Style and color items must have a named category. Constants contain their category in the name.

Theme colors and styles fall into the following categories:

mvThemeCat_Plots: Items that are associated with plots. Style/color constants identified by *mvPlotCol_*** or *mvPlotStyle_*****

mvThemeCat_Nodes: Items that are associated with Nodes. Style/color constants identified by *mvNodeCol_*** or *mvNodeStyle_*****

mvThemeCat_Core: All other items within dearpygui. Style/color constants identified by *mvThemeCol_*** or *mvThemeStyle_*****

3.26.1 Default Theme (global)

Default themes will apply the theme globally across all windows and propagate to children.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Tutorial", pos=(20, 50), width=275, height=225) as win1:
    t1 = dpg.add_input_text(default_value="some text")
    t2 = dpg.add_input_text(default_value="some text")
    with dpg.child_window(height=100):
        t3 = dpg.add_input_text(default_value="some text")
        dpg.add_input_int()
    dpg.add_input_text(default_value="some text")

with dpg.window(label="Tutorial", pos=(320, 50), width=275, height=225) as win2:
    dpg.add_input_text(default_value="some text")
    dpg.add_input_int()

with dpg.theme() as global_theme:

    with dpg.theme_component(dpg.mvAll):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (255, 140, 23), category=dpg.
↪mvThemeCat_Core)
```

(continues on next page)

(continued from previous page)

```
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)

    with dpg.theme_component(dpg.mvInputInt):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (140, 255, 23), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)

dpg.bind_theme(global_theme)

dpg.show_style_editor()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.26.2 Container Propagation

Applying a theme to a container will propagate the theme to its children:

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Tutorial", pos=(20, 50), width=275, height=225) as win1:
    t1 = dpg.add_input_text(default_value="some text")
    t2 = dpg.add_input_text(default_value="some text")
    with dpg.child_window(height=100):
        t3 = dpg.add_input_text(default_value="some text")
        dpg.add_input_int()
    dpg.add_input_text(default_value="some text")

with dpg.window(label="Tutorial", pos=(320, 50), width=275, height=225) as win2:
    dpg.add_input_text(default_value="some text")
    dpg.add_input_int()

with dpg.theme() as container_theme:

    with dpg.theme_component(dpg.mvAll):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (150, 100, 100), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)

    with dpg.theme_component(dpg.mvInputInt):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (100, 150, 100), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)
```

(continues on next page)

(continued from previous page)

```
dpg.bind_item_theme(win1, container_theme)

dpg.show_style_editor()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.26.3 Item Specific

Applying a theme to an item will overide any previous themes on the specified item if the theme contains an applicable component.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Tutorial", pos=(20, 50), width=275, height=225) as win1:
    t1 = dpg.add_input_text(default_value="some text")
    t2 = dpg.add_input_text(default_value="some text")
    with dpg.child_window(height=100):
        t3 = dpg.add_input_text(default_value="some text")
        dpg.add_input_int()
    dpg.add_input_text(default_value="some text")

with dpg.window(label="Tutorial", pos=(320, 50), width=275, height=225) as win2:
    dpg.add_input_text(default_value="some text")
    dpg.add_input_int()

with dpg.theme() as item_theme:
    with dpg.theme_component(dpg.mvAll):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (200, 200, 100), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 0, category=dpg.mvThemeCat_
↪Core)

dpg.bind_item_theme(t2, item_theme)

dpg.show_style_editor()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.26.4 Priority of Themes

The theme prioritizes the latest applied theme in the order of

1. specific item
2. container inherited
3. global

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Tutorial", pos=(20, 50), width=275, height=225) as win1:
    t1 = dpg.add_input_text(default_value="some text")
    t2 = dpg.add_input_text(default_value="some text")
    with dpg.child_window(height=100):
        t3 = dpg.add_input_text(default_value="some text")
        dpg.add_input_int()
    dpg.add_input_text(default_value="some text")

with dpg.window(label="Tutorial", pos=(320, 50), width=275, height=225) as win2:
    dpg.add_input_text(default_value="some text")
    dpg.add_input_int()

with dpg.theme() as global_theme:
    with dpg.theme_component(dpg.mvAll):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (255, 140, 23), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)

    with dpg.theme_component(dpg.mvInputInt):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (140, 255, 23), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)

with dpg.theme() as container_theme:
    with dpg.theme_component(dpg.mvAll):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (150, 100, 100), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)

    with dpg.theme_component(dpg.mvInputInt):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (100, 150, 100), category=dpg.
↪mvThemeCat_Core)
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 5, category=dpg.mvThemeCat_
↪Core)

with dpg.theme() as item_theme:
    with dpg.theme_component(dpg.mvAll):
        dpg.add_theme_color(dpg.mvThemeCol_FrameBg, (200, 200, 100), category=dpg.
↪mvThemeCat_Core)
```

(continues on next page)

(continued from previous page)

```
        dpg.add_theme_style(dpg.mvStyleVar_FrameRounding, 0, category=dpg.mvThemeCat_
↪Core)

dpg.bind_theme(global_theme)
dpg.bind_item_theme(win1, container_theme)
dpg.bind_item_theme(t2, item_theme)

dpg.show_style_editor()

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.26.5 Theme for Disabled Items

Each item has a separate disabled theme that is used when the item is disabled. The disabled theme follows the same propagation rules as the enabled theme. When an item's parameter *enabled* is set to *False* the item will use its disabled theme. If no disabled theme has been set the default theme will be used.

```
import dearpygui.dearpygui as dpg

dpg.create_context()
dpg.create_viewport()
dpg.setup_dearpygui()

with dpg.theme() as disabled_theme:
    with dpg.theme_component(dpg.mvInputFloat, enabled_state=False):
        dpg.add_theme_color(dpg.mvThemeCol_Text, [255, 0, 0])
        dpg.add_theme_color(dpg.mvThemeCol_Button, [255, 0, 0])

    with dpg.theme_component(dpg.mvInputInt, enabled_state=False):
        dpg.add_theme_color(dpg.mvThemeCol_Text, [255, 0, 0])
        dpg.add_theme_color(dpg.mvThemeCol_Button, [255, 0, 0])

dpg.bind_theme(disabled_theme)

with dpg.window(label="tutorial"):
    dpg.add_input_float(label="Input float", enabled=False)
    dpg.add_input_int(label="Input int", enabled=False)

dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

3.26.6 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	

3.26.7 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	

3.26.8 Plot Colors

Plot Colors		
mvPlotCol_Line	mvPlotCol_LegendBg	mvPlotCol_AxisBgHovered
mvPlotCol_Fill	mvPlotCol_LegendBorder	mvPlotCol_AxisGrid
mvPlotCol_MarkerOutline	mvPlotCol_LegendText	mvPlotCol_AxisText
mvPlotCol_MarkerFill	mvPlotCol_TitleText	mvPlotCol_Selection
mvPlotCol_ErrorBar	mvPlotCol_InlayText	
mvPlotCol_FrameBg	mvPlotCol_AxisBg	
mvPlotCol_PlotBg	mvPlotCol_AxisBgActive	
mvPlotCol_PlotBorder	mvPlotCol_Crosshairs	

3.26.9 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		

3.26.10 Core Styles

Constant	Components
mvStyleVar_Alpha	1
mvStyleVar_DisabledAlpha	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_TabBorderSize	1
mvStyleVar_TabBarBorderSize	1
mvStyleVar_TableAngledHeadersAngle	1
mvStyleVar_TableAngledHeadersTextAlign	2
mvStyleVar_ButtonTextAlign	2
mvStyleVar_SelectableTextAlign	2
mvStyleVar_SeparatorTextBorderSize	1
mvStyleVar_SeparatorTextAlign	2
mvStyleVar_SeparatorTextPadding	2

3.26.11 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

3.26.12 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

3.27 Tooltips

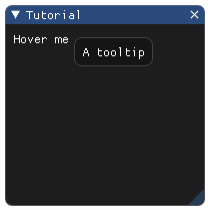
Tooltips are windows that appear when an item is hovered.

Tooltips are containers that can hold any other UI Item.

Tooltips must have the parent argument. This is the *tag* of the parent what will show the tooltip.

```
import dearpygui.dearpygui as dpg

dpg.create_context()

with dpg.window(label="Tutorial"):
    dpg.add_text("Hover me", tag="tooltip_parent")

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

dpg.create_viewport(title='Custom Title', width=800, height=600)
dpg.setup_dearpygui()
dpg.show_viewport()
dpg.start_dearpygui()
dpg.destroy_context()
```

Results



3.28 dearpygui.dearpygui

CHAPTER

FOUR

MORE

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.

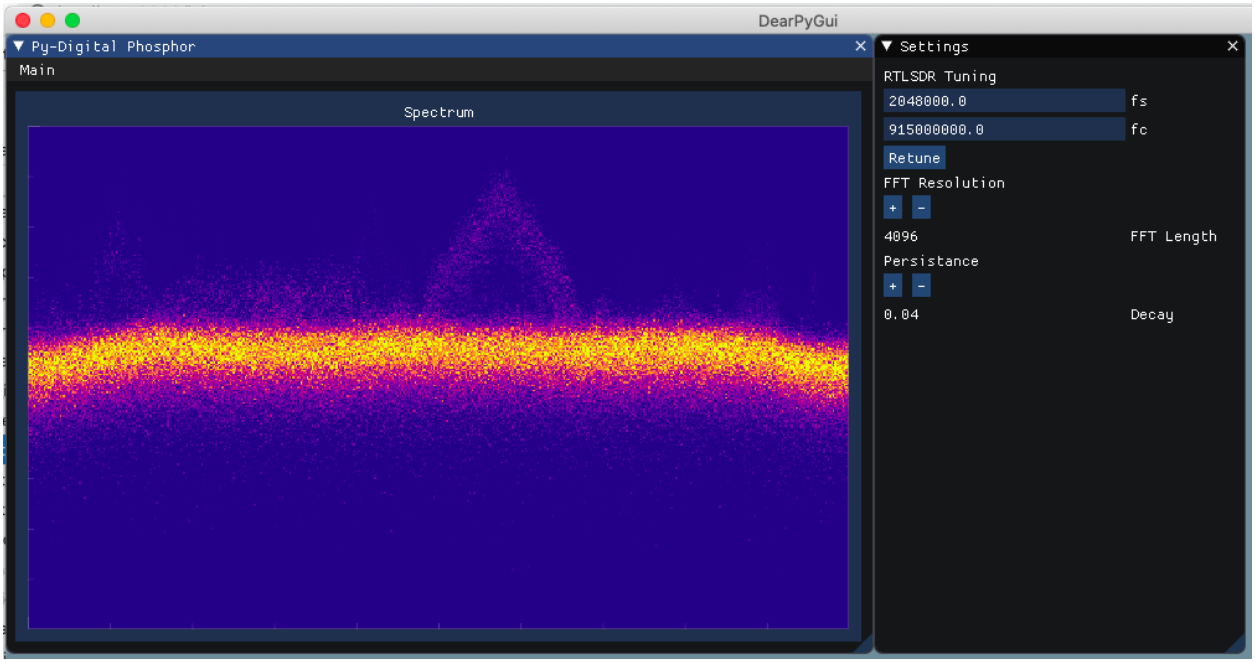
The source code is available in the Snake Github repository.

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.