Gui

Overview

There are currently two GUI-systems in the engine. The normal one which is called “retained” or just the “normal” GUI. This works by creating all the assets up-front and then works by having special callbacks for all of the GUI elements. For instance when a button is clicked a callback is sent to some piece of code that does the logic for that. It is easy to make an editor where you can build the element for this type of GUI (unfortunately we presently do not have one).

The other one is the immediate mode gui or just ImGui. This is a bit special and works by drawing graphics and doing logic at the same time. None of the GUI elements are saved by defined any created every update. This makes it very easy to make changes and extremely fast to iterate (especially when working in script). It does not work very well for more complex interfaces though and is mainly used to make stuff like HUD, Game menus and in-game interfaces.

The game also support to draw the GUI on rendered surfaces and how to achieve this will be brought up at the end of this document.

Retained GUI

Info coming later!

Immediate GUI

Summary

The Immediate Mode Gui (ImGui) is a bit special to how you normally do GUI and works by essentially recreating the entire interface every update. On top of this also handles logic and graphical update in the very same function. Normally this is a sort of sin, but for this system it works really well and make it very easy to set up interfaces. The ImGui have been designed around this concept and the main focus have been to have a system that make GUI creation and iteration really fast.

Anyone who has used Unity's older GUI should get the basics of this and it is actually worth looking through their manual for some extra information. This GUI has some extra tricks though and do somethings a bit differently so do not take this as a copy of the Unity one. One of the biggest changes is that it contains some state saving making it even easier to main things.

Another important feature of the ImGui is that it works seamlessly with both mouse, keyboard and gamepad input. The most important part of this is a system that calculates navigate with keys/stick on the fly so it is really easy to set up.

Names and Ids

In order to keep track of the state of the various GUI elements, the ImGui must save some states. In order to do this it creates 64-bit Ids from the names of the widgets, variables, etc. This makes it very important that the Name of the gui elements are all unique, or else the interface will not work.

Internally the actually string names are not used at all, but for speed reasons they are instead represented by 64 bit numbers (that are generated from the strings).

Widgets

Widgets are the different elements that make up an interface. Some of these are possible to interact with, while other are not. This is important to note so that you use the right kind of widget when you want a specific behavior. Worth noting is also that all widgets have one function that use the default data settings and one where a data class is needed as argument. All of the widget functions start with ImGui_Do*.

Some of the widgets have default values arguments, such as the ToggleButton, Slider, etc, which behave in a special way. The default value will only be used to set the initial value, and then as the value change through interaction, an internally saved value will be used instead. However, the default value is saved too, and if it changes then the changed default value will be used instead of the saved internal one.

Widget functions usually come in two version. One that uses the the default settings, which have been set up using ImGui_SetDefault*. The other comes with the data settings as an argument and have the suffix “Ext” (in helper functions, when directly using the ImGui class).

Finally it is only widgets that are affected by the layout, so never draw anything directly unless you know what you are doing. It is almost always much better to for instance use Label instead of DrawText to print a message.

Button
This is a normal button that returns true if it was pressed this update.

RepeatButton
Like Button, but will return true as long as it is being pressed.

ToggleButton
Another version of the button that can be turned on/off. It returns the current state.

SliderHorizontal
A slider button, which has values between two numbers. A step-size value can also be set to specify in which interval between min and max values can take. It returns the current value.

SliderVertical
Just like SliderHorizontal, but placed vertically.

Label
The label does not have any interaction logic, but only displays a text string. If you need a text string

with interaction use a button instead.

Image
This displays an image. Like Label it has no interaction.

MultiToggle
A grid of toggle buttons, where only one can be active at a time. It returns the index (0=first) of the currently selected. In order to add items use ImGui_AddItem* functions.

CheckBox
A text next to a checkbox that can be on or off. Returns the current value.

TextFrame
A scrollable text of frame. It will return, in rows, the amount of text not outside of the window. For instance, if the text contains 5 rows, and only 4 are visible, 1 is the return value. This value is very useful when making a slider that can scroll the text.

MultiSelect
Here several different items can be selected. It returns the index of the currently selected one. In order to add items use ImGui_AddItem* functions.

Modifiers

Modifiers is a bunch of states that can be set in order to change the function and appearance of the widget. Once called, any widget defined afterwards will be affected by it. In order to remove the effects, one can use ImGui_ResetModifiers to set everything back to default values. Another way of handling it is to use the modifier stack. By ImGui_PushModifiers the current state of all the modifiers will be pushed on to the stack. One can then set a bunch of new modifier values and simple use ImGui_PopModifiers when the last recently pushed state of values should be used again.

Previous State

The previous state is a set of properties that are set after a widget function have been called. All come in the form of: ImGui_Prev*.

They can be used to get all sort of information on the state of the most recently defined widget. Mostly, these are useful when doing custom Widgets, (see below) and can then be used to see if the widget just became into focus, or more importantly what as the absolute size and position used to draw it.

Groups and Layout

The functions ImGui_GroupBegin/ImGui_LayoutBegin and ImGui_GroupEnd/ImGui_LayoutEnd are very important functions as they can make it a lot easier to create a nicely aligned Gui. Important to note is that one always must start with *Begin and then *End for any Layout or Group. They must be nicely nested so the last defined *Begin must be the first *End to be called. It is possible to go quite

advanced with the nesting and create nice looking gui using this.

Groups just define a position and area any widgets or draw-calls between its Begin and End. It is possible to turn on clipping so that the the objects inside the group will not be have any parts drawn outside of the area.

Layouts automatically places widgets according to a certain scheme. They come in three different shapes:
X: This will align all widgets inside it horizontally.
Y: This will align all widgets inside it vertically.
XY: Aligns the widgets along the X-axis until the widget will not fit. It will then jump down a row (based on highest widget).
Except for the XY version, none of the layouts use the size parameter.

State Variables

State variables is an easy way to keep track of information without setting any class member variables. They are saved by the GUI and is meant to be used to keep track of simpler states and effect properties. These are also used internally by some of the widgets to keep track of stuff like the current slider value. Currently only three different types of values are supported: int float, 3d vectors and color.

To make variables even more usable, they can be very easily faded to different values. This is done by calling the ImGui_FadeState* functions. An importnat argument for these functions is the ReplaceIfExist one. If true (the default setting), it will be checked if if the variable is already exist and then be replaced. If false new fade will happen until the current one is done. Setting to false can be very useful in case the ImGui_FadeState might be called many times in a row. A

Another important fade setting is the Type one. This will determine in what way the value will be faded between start and end. The available ones are:
Linear: Constant change. **Sigmoid**: Starts and ends smoothly. **Cosine**: Also smooth start and end, but slightly different. **Cube**: Starts slow and accelerates. **InvCube**: Starts fast and decelerates.

Another way to use fading it so let it oscillate between two values. This is useful to a blinking button or some moving graphic. To do this use the ImGui_FadeOscillate functions. This are meant to be called every update when defining the value of some properties. Example:

```
cColor = cColor(1,1)*ImGui_FadeOscillateFloat("pulse",1.0f, 0.25f,1);
```

Timers

Timers are used in order for events to happen after a delay or at certain intervals. The first way to use timers is to start one and the check and see if it has ended. To do this use the function ImGui_AddTimer to start a timer then then check ImGui_TimerOver to see when it is over. Another way to use timers is to check for events periodically. This is done with the ImGui_RepeatTimer functon and should be called every update to see if it is time for a certain event.

Assets

Assets like graphics and fonts are all taken care of internally and loaded when needed. So to use a graphic or font, you only need to have the file path for it. Most of the needed properties are available in the constructor for the data class. So using a flower graphics with additive blending mode is done like this:

```
cImGuiGfx flowerGfx("flower.tga", eGuiMaterial_Additive);
```

Then flowerGfx can be used where ever it is needed. No need to neither load or destroy the asset.

Custom Widgets

Doing custom widgets is really simple and is done by simply writing a function. This means that you can build a library of widgets by having global functions inside an include file. There is no need to handle or properties of any kind, and any properties needed can be saved using state variables. Easiest way is then to name each variable NameOfWidget+NameOfValue.

Also note that it is very much okay to use groups and layouts inside the custom widgets.

In-Scene GUI Rendering

In order for the a game entity to have a GUI surfaces the following criteria must be met:

- The surface that is to contain the GUI must be flat, consist of 4 vertices and have UV coords going from 0→1 where (0,0) is the upper left corner.
- The material of this surface must be translucent. It can be any blend mode though, making it possible to have transperant in-game GUI.
- The material must have a diffuse texture set, although this texture will never be used in-game (it will only show up in editors and such).
- Any other textures may also be set, so it is possible to give the GUI screen refractive patterns, etc.

Currently, the only entity type that support GUI is “Prop_Terminal”, so set the entity to that type and then in the ent-file class settings, set up the properties required. Most important is to set the ConnectedSubmesh variables which is the name of the flat surface where the GUI will be displayed. Also remember to set up a ScreenSize (in pixels) that has the same ratio has the ConnectedSubmesh.