Getting started with making a Quaver Editor plugin

IceDynamix

2020-09-30

Contents

1 Setup

It is assumed that you are familiar with Lua syntax and basic programming knowledge.

1.1 Files

1. Create a folder in `Quaver/Plugins/yourFolderName`
   - Will be under `Quaver/bin/Debug/netcoreapp2.1/Plugins/` if building yourself
2. Create following files inside the folder
   - settings.ini

     ```
     [Settings]
     Name = Plugin Name
     Author = Your Name
     Description = Your description
     ```

   - plugin.lua

     ```
     function draw()
         imgui.Begin("Window Title")
         imgui.End()
     end
     ```

1.2 Editing

Open up the Quaver Editor. The plugin should show up under the Plugins menu, as long as you did everything correctly. Whenever you edit code and save the `plugin.lua` file, the plugin in-game will update automatically.

Use any editor you like, preferably one that has syntax highlighting for Lua.

2 Important concepts

2.1 C++ to C# to Lua?

ImGui, the base of the plugin system, was originally written in C++. Since Quaver is written in C#, a C# implementation for ImGui called ImGui.NET is used. And finally, to make things easier to write, the script language Lua in a sandbox environment with the MoonSharp interpreter is used. All ImGui or Quaver related functions and structures that work in Lua have already been implemented in C#, which is why the existing Quaver C# code is basically the documentation. All functions that derive from ImGui can be accessed with `imgui.function()` and can be found in ImGuiWrapper.cs, ImGui enums, their implementations and the Quaver functions and structures are spread over many files, which is why I have compiled them in this guide at the end in the Available Resources section, along with the C# snippets.

2.2 Semicolons

Semicolons are optional in lua. You can decide to use them, or you can decide to leave them out. I will leave them out for all code examples. You can put multiple statements into a single line if you use semicolons so separate them.

2.3 if and for

When coming from a different programming language, it's very easy to forget the "`then`" keyword at the end of the if-condition. The correct form of the if-statement is

```
if condition then
    -- do something
end
```

Same thing with the standard for-loop, which has "`do`" at the end.

```
for i=0, 10, 1 do
    -- do something
end
```

You can also use parantheses if you feel more comfortable with it!

```
for (i=0, 10, 1) do
    if (i % 2 == 0) then
        -- do something
    end
end
```

2.4 Scope

If you've programmed with different languages, then you'll have heard of this term before. It describes the persistance of a variable after the end of the containing function or structure.

Lua's variables are global by default when you declare them as is. They are accessible from any function calling the containing scope. So following code would work.

```
function draw()
    imgui.Begin("Example")
    f()
    imgui.Text(text)
    imgui.End()
```

```
end

function f()
    text = "Hello!"
end
```

Local variables are scope limited to the block they were defined in and are declared like `local a = 10`. They are prioritized over global variables, if declared with the same name (shadowing). My main recommendation would be to always use local variables, unless you know what you're doing.

You can explicitly define a block with `do ... end`.

Another thing to note is that you can define global variables to use, but only if you don't assign any script related variable to them (includes anything that isn't available in vanilla Lua by default), since the script variables are only initialized in the `draw()` function.

```
thisGlobalVarWorks = 5
thisGlobalVarDoesntWork = #map.HitObjects -- number of hitobjects

function example()
    print(thisGlobalVarWorks) -- works
    print(thisGlobalVarDoesntWork) -- doesn't work
end
```

2.5 Constants

Constants aren't a thing in Lua and the workaround is more effort for what it's worth, so I recommend just sticking to UPPER_SNAKE_CASE naming for your variables, since that's the most common way to write variable names for constants.

2.6 Arrays/Lists/Tables

There aren't any arrays/lists in Lua, only tables. A table is essentially a dictionary, and it can be used like a normal array by assigning a number as the key for the dictionary. It does this by default. You initialize a table with `myTable = {}`. Add more elements to the table with `table.insert(myTable, myVariable)`. Access table elements with `myTable[key]`, where key can be of any type. You can also access them with `myTable.key`.

Important: Take note that everything in Lua is indexed from 1, not 0!

Related: Lua-users.org Tables Tutorial

2.7 Pass by value/reference

When calling a function, it's important to know how a language handles its parameters. Is it pass by value? Is it pass by reference? In Lua it's a mix of both, it depends on the type passed. This is how each type behaves:

Type	Pass by...
`nil`	value
`boolean`	value
`number`	value
`string`	value
`function`	reference
`userdata`	reference

Type	Pass by...
`thread`	reference
`table`	reference

2.8 Immediate Mode GUI (IMGUI)

It's important that you get comfortable with the "immediate mode GUI" concept. If you've programmed GUIs in other programmin languages before, you may remember that you added everything to something via functions to a general frame, and then at the end start/render/unhide the construct. That's called Retained Mode GUI (RMGUI). This is not exactly the case with IMGUI.

A RMGUI creates the GUI once and changes the affected elements depending on the user actions. It saves the state between each frame and knows which elements it doesn't have to rerender. An IMGUI "creates" the GUI every frame and doesn't save the state (refer to the next section). Everything is redrawn every frame. (This part is very simplified.)

The advantage that IMGUI has over RMGUI is that all of the rendering, the callbacks, the data transfers and everything, is managed by the library. While in RMGUI you might have to first create a button, create a callback, assign that callback to the button, add the button to the frame and then render the frame, in IMGUI you can simply create a button and then check the value of that button with, for example, an if statement to evaluate.

You don't have to worry about any adding. Any frame management. No callbacks. And that's the system we are using with Dear ImGui.

Refer to: Retained Mode Versus Immediate Mode

2.9 State variables

Referring to IMGUI, the plugin is rerendered every frame. This also means, that all variables in the plugin are cleared every frame. The way GUI elements here work, is that they are fed a value to display and then a value is spit out whenever the user changes that element. But how does the plugin know which value to initially display? Or where does the plugin save the value?

We need a way to save the values across frames.

This is done by using the provided `state` object. First you retrieve the value from the `state` object, you do your calculations and then it's saved to the state object again. Following function increments the variable `n` by 1 on every frame.

```
function draw()
    -- label is typically the same as the variable name itself, so it would be
    -- "n" in this case. it doesn't really matter, all that matters is that it's
    -- the same as in the final state.SetValue() function
    local n = state.GetValue("choose your own identifier!")

    if n == nil then
        n = 0 -- default value when variable is initialized for the first time
    end

    n = n + 1; -- yes, there is no += or -= in lua

    state.SetValue("choose your own identifier!", n)
end
```

2.10 MoonSharp Core Modules

The reason Lua was chosen as the script language was because it is easy to create a sandboxed system for it. Quaver uses such a sandboxed system to make sure that nothing malicious can be done with plugins. The list of different core modules can be found here, Quaver currently uses the HardSandbox presets, which includes:

- `GlobalConsts`, the global constants `_G`, `_VERSION` and `_MOONSHARP`
- `Basic`, includes `assert`, `collectgarbage`, `error`, `print`, `select`, `type`, `tonumber` and `tostring`
- `TableIterators`, the table iterators `next`, `ipairs` and `pairs`
- `String`, the string package
- `Table`, tables and the table package functions
- `Math`, the math package
- `Bit32`, the bit32 package

Following modules are disabled in Quaver:

- `Metatables`, the metatable methods `setmetatable`, `getmetatable`, `rawset`, `rawget`, `rawequal` and `rawlen`
- `ErrorHandling`, the error handling methods `pcall` and `xpcall`
- `Coroutine`, the coroutine package
- `OS_Time`, the time methods of the os package `clock`, `difftime`, `date` and `time`
- `LoadMethods`, the load methods `load`, `loadsafe`, `loadfile`, `loadfilesafe`, `dofile` and `require`
- `OS_System`, the methods of os package excluding those listed for OS_Time
- `IO`, io and file package

This in turn means you can't:

- Use a multi-file/module structure for your code
  - Everything has to be inside `plugin.lua`
  - Refer to section Module Management and Using an actual multi-file structure if you're ready to go the extra mile
- Import external resources
- Connect to the internet/an external API
- Work with the local file system (create, read, write)

2.11 Debugging

I'll be honest: the error messages the console returns on bad code are nearly useless. You'll have to do some manual debugging and comment out code piece by piece until you hit the part that hurts. If you're building Quaver yourself and have the console available, it's possible to print to the console using `print()` in Lua. Or you can write the content of a variable into a `imgui.TextWrapped()` in the plugin. It's up to you.

If you're testing an algorithm, I don't recommend testing that algorithm in an actual plugin. Do it in an actual Lua interpreter. Go on a website if you will.

I made a plugin that displays all state/map variables for debugging purposes.

Lastly, it's important to mention that `pcall()` (protected call, essentially a try-block) and `xpcall()` (executes a callback function if the function errors) do not work right now.

2.12 Useful links

- Quaver Github
- ImGui

- ImGui.NET
- Quaver/ImGuiWrapper.cs
- MoonSharp

3 Making a plugin

3.1 Windows

Let's take a look at the sample code provided in Setup/Files.

```
function draw() -- Quaver *always* starts out in the draw() function
    imgui.Begin("Window Title")
    imgui.End()
end
```

First of all, Quaver always looks for the `draw()` function and goes on from there. This is different to what one would expect, where a `main()` function is the entry point. Looking at the output, we can see that a window with a title has popped up in the editor. As you can already guess, you can make a window with the `imgui.Begin()` and the `imgui.End()` functions.

3.2 Draw text

It's recommended to keep each window in its own function, to keep everything organized.

```
function draw()
    window1()
    window2()
end

function window1()
    imgui.Begin("Window Title 1")
    imgui.Text("Example 1")
    imgui.End()
end

function window2()
    imgui.Begin("Window Title 2")
    -- Note: .. signifies concatenation
    local mySuperLongString = "Very long and cool text that wraps around if " ..
        "the window gets smaller and smaller " ..
        "and smaller and smaller and smaller " ..
        "and smaller and smaller and smaller."
    imgui.TextWrapped()
    imgui.End()
end
```

3.3 Integer input box

There's a lot of different elements you can implement, check out the ImGuiWrapper.cs to know what can or what can't be added.

Let's take this as an example:

```
public static bool InputInt(string label, ref int v) => ImGui.InputInt(label, ref v);
```

The only important part is `InputInt(string label, ref int v)`. Looking at the code, we can see something weird with the function; it uses a `ref` parameter (which is basically a pointer). Lua doesn't exactly work with ref parameters, so it needs a different way to express that relation in the code. If you don't care about the details, then this is what you would need to do to get an integer text box:

```lua
function draw()
    imgui.Begin("Window Title")

    local myInt = get("myInt", 0)
    uselessVariable, myNextInt = imgui.InputInt("My custom label", myInt)
    state.SetValue("myInt", myNextInt)

    imgui.End()
end

function get(identifier, defaultValue)
    return state.GetValue(identifier) or defaultValue
end
```

If you *do* care about the details of why it's handled in Lua this way, check out this.

Since the first return value of the `InputInt()` function isn't important to us, we can use any short variable name to make life easier for us. It's common to use an underscore for any irrelevant values. We can also write the new value back into the old variable, so there's no need to create a new one.

```lua
-- The code examples from now on will assume, that you're in a window environment
-- as provided by imgui.Begin() -- imgui.End(), unless specified otherwise

local myInt = get("myInt", 0)
_, myInt = imgui.InputInt("My custom label", myInt)
state.SetValue("myInt", myInt)
```

There are a few overloads for the `InputInt()` function in the ImGuiWrapper.cs, let's take a look:

```cs
// ImGuiWrapper.cs

public static bool InputInt(string label, ref int v)
    => ImGui.InputInt(label, ref v);
public static bool InputInt(string label, ref int v, int step)
    => ImGui.InputInt(label, ref v, step);
public static bool InputInt(string label, ref int v, int step, int step_fast)
    => ImGui.InputInt(label, ref v, step, step_fast);
// one more...
```

Let's filter the irrelevant information out though:

```cs
InputInt(string label, ref int v);
InputInt(string label, ref int v, int step);
InputInt(string label, ref int v, int step, int step_fast);
InputInt(string label, ref int v, int step, int step_fast, ImGuiInputTextFlags flags);
```

Feel free to play around with the different parameters! In case you want to use the input flags in the last function, take a look at ImGui Enums and check, which one you need to use. The documentation for the enums themselves can be found either in the ImGui.NET/C#_Enums or directly in the ImGui/C_Enums

3.4 Any input element

You can apply the same principles to any other datatype though. Take a string for example:

```cs
InputText(string label, ref string input, uint maxLength)
```

The only difference to the `InputInt()` function is the new parameter, which requires you to provide a maximum length of the input string. The example uses a limit of 50 characters.

```
-- Make sure to skip the `local` keyword when declaring a "constant"
MAXIMUM_INPUT_STRING_LENGTH = 50

function draw()
    imgui.Begin("Window Title")

    local myString = get("myString", "")
    _, myString = imgui.InputText("My custom label", myString, MAXIMUM_INPUT_STRING_LENGTH)
    state.SetValue("myString", myString)

    imgui.End()
end
```

It *would* be best to define a constant for it... if only constants were a thing in Lua. Refer to Constants.

Again, check out the Quaver/ImGuiWrapper.cs for all possible input values. You should be able to figure everything out by now. Try thinking about how you would go on about making a checkbox or an integer slider!

```
/*...*/ Checkbox(string label, ref bool v) /*...*/
/*...*/ SliderInt(string label, ref int v, int v_min, int v_max) /*...*/
```

3.5 Buttons

A button in Lua is as simple as `imgui.Button(label)`. Checking the state of the button (pressed/not pressed) is done by checking the value (a boolean) of the button itself. This example prints out "Hello World!" to the console when you press the button.

```
local myButton = imgui.Button("Hello?")

if myButton then
    print("Hello World!")
end
```

Or in short:

```
if imgui.Button("Hello?") then
    print("Hello World!")
end
```

3.6 Plots

There's something off about the original ImGui.NET wrapper function for plots. Can you find it?

```
void PlotLines(string label, ref float values, int values_count)
```

Right. Why is `values` a float reference when it should actually be a float[] reference? Refer to issue #105 on the ImGui.NET repository. But now is the question... How do you actually make a plot in Lua now? Lua doesn't really do references.

That's where I took matters into my own hands. I rewrote the Quaver ImGuiWrapper to make it take an array. This is how it is currently implemented, after merging my pull request:

```
void PlotLines(string label, ref float[] values, int values_count)
    => ImGui.PlotLines(label, ref values[0], values_count)
```

Now you can pass the entire table like this!

```
local values = { 1.0, -1.0, 3.0, 6.6 }
PlotLines("My Plot!", values, #values)
```

3.6.1 Pushing and popping

Since Quaver plugins rely on immediate mode GUI, there are methods for "toggling" certain states on and off. Take this for example:

```
imgui.Columns(2) -- Everything from here on is split into two columns
imgui.Text("I'm in column 1")
imgui.NextColumn()
imgui.Text("I'm in column 2")
imgui.NextColumn()
imgui.Columns(1) -- Back to one column!
```

Or this:

```
if imgui.TreeNode("My foldable section") then -- starts the tree section
    imgui.Text("You can see me") -- Only visible if the tree node is active
    imgui.TreePop() -- ends the tree section
--  ^^^^^^^^^^^^^^^ If this weren't here, everything after this might also be
--                  affected by a tree section.
end
```

This kind of principle works with various other elements. Take for example:

Push Function	Pop Function
`imgui.Begin()`	`imgui.End()`
`imgui.Begin___()`	`imgui.End___()`
`imgui.Columns(n)`	`imgui.Columns(1)`
`imgui.Indent()`	`imgui.Unindent()`
`imgui.PushItemWidth()`	`imgui.PopItemWidth()`
`imgui.PushStyleVar()`	`imgui.PopStyleVar()`
`imgui.PushStyleColor()`	`imgui.PopStyleColor()`

3.7 More UI elements

There are many more GUI elements to be discovered! There's a demo window by ImGui, showcasing many of the available elements, which can be accessed by calling the `imgui.ShowDemoWindow()` function in a plugin.

```
function draw()
    imgui.ShowDemoWindow()
end
```

Everything seen in the demo window can (probably) be realized in a Quaver plugin as well. The source code for the demo window can be found at ImGui/imgui_demo.cpp. Just apply different syntax (`imgui.Function()` instead of `ImGui::Function()`) and apply pointers/addresses/ref parameters as seen in Creating an integer input box.

3.8 Keypresses

You can detect keypresses with plugins, which can allow you to make assign all kinds of custom keybinds to your favorite actions! This is achieved by using these utility functions:

```
// Only returns true on the exact frame the key was pressed/releaseed
bool IsKeyPressed(Keys k);
bool IsKeyReleased(Keys k);
// Always returns true as long as the key is (not) held
bool IsKeyDown(Keys k);
bool IsKeyUp(Keys k);
```

The keys are specified with the MonoGame.Framework.Input.Keys enum, which can be accessed in the script with `keys.Tab` as an example.

```
if utils.IsKeyDown(keys.Space) then
    imgui.Text("Space is pressed!")
else
    imgui.Text("Space is not pressed!")
end
```

For key combinations involving modifier keys to work as traditionally (hold modifier and press next key), you need to use `utils.IsKeyDown()` for the modifier and `utils.IsKeyPressed()` for the key. Otherwise you'd have to hit both keys at the exact same frame. Here is an example to make Ctrl+K/L move the current time forward/back by 1 second:

```
if utils.IsKeyDown(keys.LeftControl) and utils.IsKeyPressed(keys.L) then
    actions.GoToObjects(state.SongTime + 1000)
elseif utils.IsKeyDown(keys.LeftControl) and utils.IsKeyPressed(keys.K) then
    actions.GoToObjects(state.SongTime - 1000)
end
```

Keep in mind that keybinds don't require any interface! You could just as well put the above code into the draw() without a imgui.Begin()/End() environment and it would still work as intended.

3.9 Drawing

Plugin drawing was added in Quaver update `v0.25.0`.

The basic principle of drawing in a plugin is getting the drawlist and calling functions from it.

```
-- Draws a white circle near the top left corner
function draw()
    local drawlist = imgui.GetOverlayDrawList()
    local position = {100, 100} -- Absolute coordinates, with top left being (0,0)
    local radius = 10
    local whiteColor = 16 ^ 8 - 1 -- Explained later
    drawlist.AddCircleFilled(position, radius, whiteColor)
end
```

There are two different drawlists, one is the overlay drawlist which allows you to draw anywhere on the screen, the other is the window drawlist which only renders objects drawn in the absolute coordinates of / clipped by the current window. They are called with `imgui.GetOverlayDrawList()` and `imgui.GetWindowDrawList()` respectively. The overlay drawlist can be used without `imgui.Begin()` and `End()`.

Every available drawlist function can be found here and starts with "Add". There is a "Add...Filled" for many of the objects available. You can't use the image functions and `ImDrawCornerFlags` isn't registered to use right now.

Colors have to be provided as a number and the format is RGBA reversed, which looks like following if put as Lua code.

```
function colorToUint(r, g, b, a)
    return a*16^6 + b*16^4 + g*16^2 + r
end
```

Trying to draw with absolute coordinates can be hard sometimes, since the screen size is variable and depends on the user. I added the current window size as `state.WindowSize` to be used along with it, it returns a table of two elements with the width and height of the current screen in that order. For example, if you want to draw an object in the middle of the screen, you can divide the screen sizes by 2 and use those as absolute coordinates.

```
-- Draws a white circle in the center
function draw()
    local drawlist = imgui.GetOverlayDrawList()
    local position = {state.WindowSize[1]*0.5, state.WindowSize[2]*0.5}
    drawlist.AddCircleFilled(position, 10, 16^8 - 1)
end
```

Having a drawing move relative to the window (for example having a drawing *in* a window) isn't done by using the window drawlist, since the coordinates used there are still absolute from the top left corner of the screen. You can use `imgui.GetWindowPos()` with `imgui.GetWindowSize()` or use some of the other size functions to work with.

3.10 Styling

Following code will bring up the style editor, where you can experiment with all of the possibilities with changing the looks of your plugin:

```
function draw()
    imgui.ShowStyleEditor()
end
```

Anything you change in the style editor will apply to all windows in the plugin. Feel free to bring up the demo window with `imgui.ShowDemoWindow()` and check out how everything looks.

There are two importants sections, sizes and colors. You won't be able to change the font, so don't bother. Now, to apply the styles to your plugin you have to do following.

3.10.1 Plugin Sizes

You can apply new sizes with this:

```
imgui.PushStyleVar(imgui_style_var.property, value)
```

If there are multiple values associated with a value, then pass a table with n elements. If I wanted to change the WindowPadding to be 20, 20 then I would do

```
imgui.PushStyleVar(imgui_style_var.WindowPadding, { 20, 20 })
```

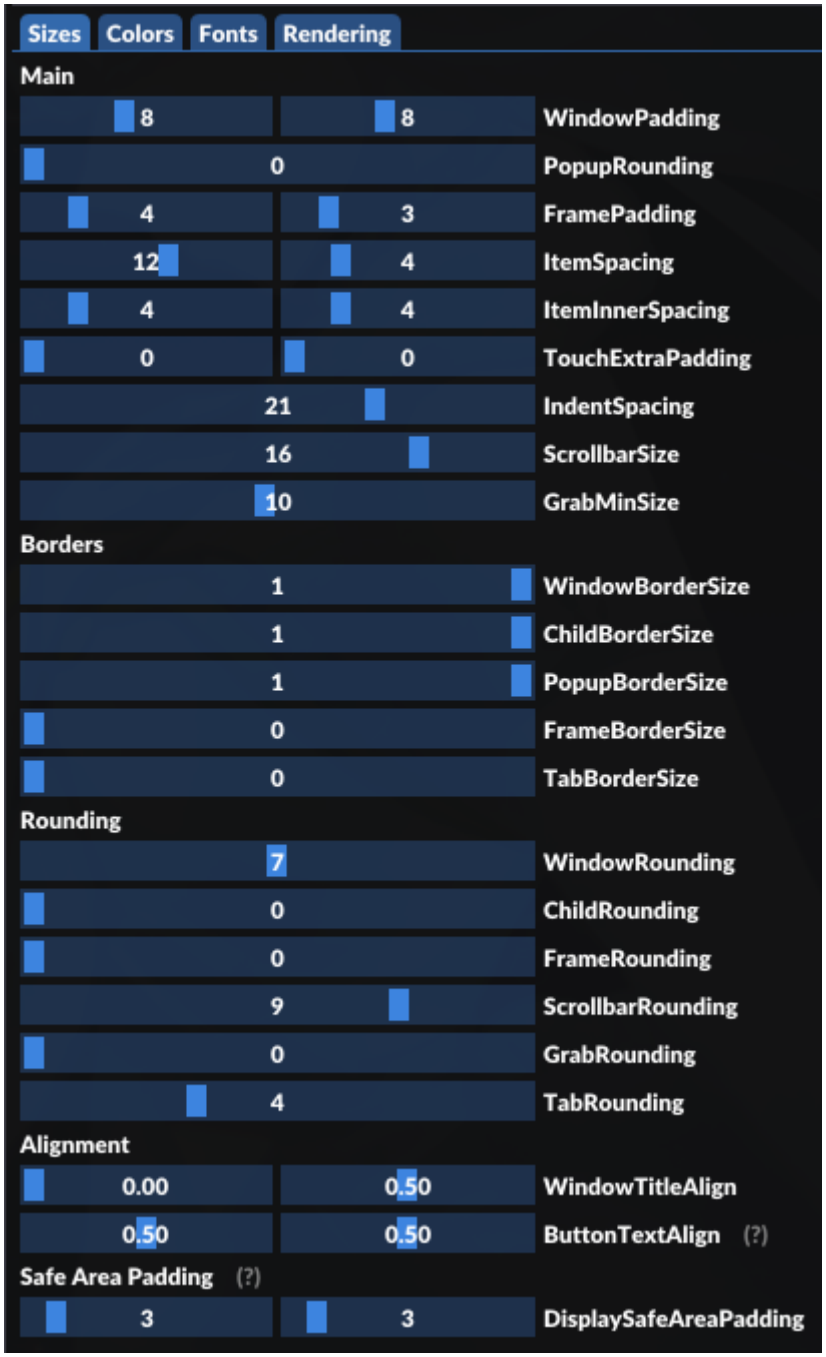

Figure 1: Style editor sizes panel

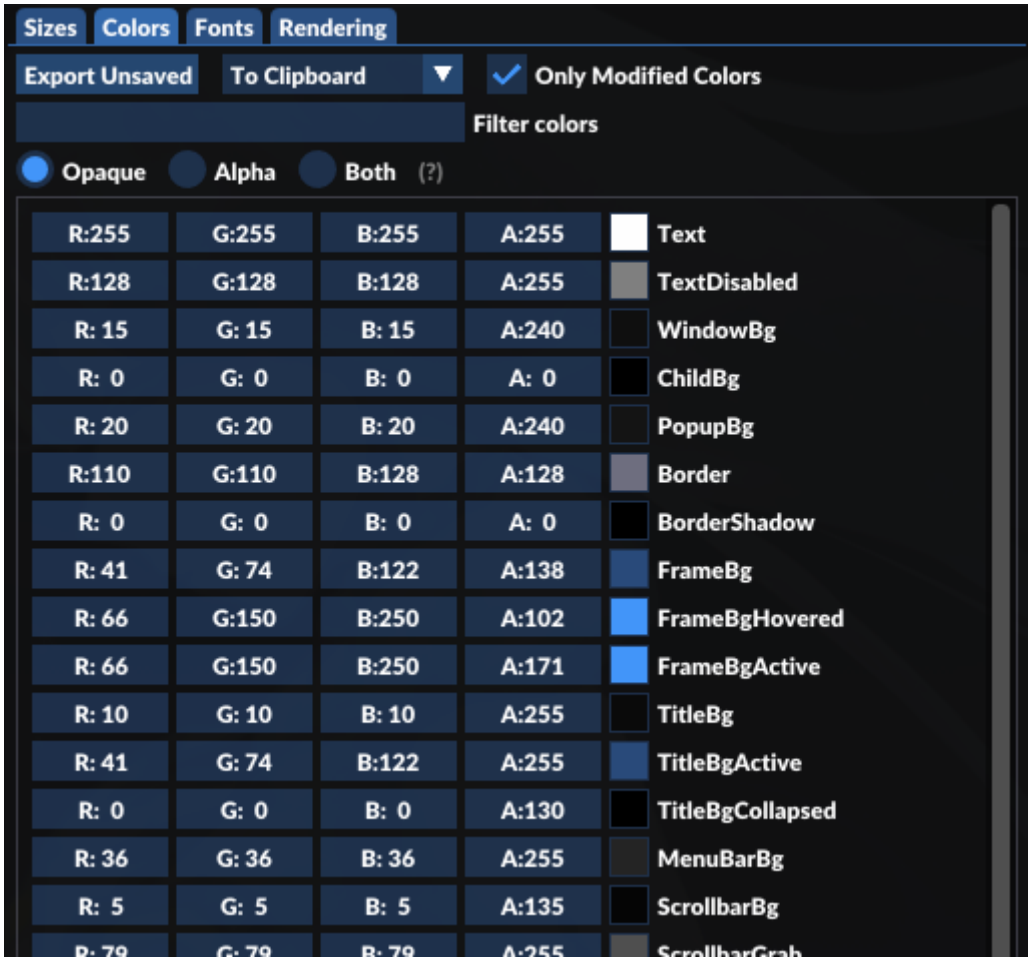

Figure 2: Style editor colors panel

3.10.2 Plugin Colors

You can change the color with a color picker upon clicking on the colored square next to the color element. After you're done adjusting, you can export the colors to your clipboard with the button near the top called "Export Unsaved" (refer to image). The checkbox is optional, unselecting it will color all colors, including the default ones. The copied colors will look like this:

```
ImVec4* colors = ImGui::GetStyle().Colors;
colors[ImGuiCol_Text]                   = ImVec4(1.00f, 1.00f, 1.00f, 1.00f);
colors[ImGuiCol_FrameBg]                = ImVec4(0.72f, 0.44f, 0.71f, 0.54f);
colors[ImGuiCol_FrameBgHovered]         = ImVec4(0.55f, 0.20f, 0.51f, 0.40f);
colors[ImGuiCol_TitleBgActive]          = ImVec4(0.16f, 0.19f, 0.23f, 1.00f);
colors[ImGuiCol_Tab]                    = ImVec4(0.63f, 0.67f, 0.72f, 0.86f);
```

Now of course, this is C++ code, and we can't just copy that into our Lua script. We also can't directly edit the style colors like shown, we have to rely on pushing/popping colors instead. So each line you get in the style of:

```
colors[ImGuiCol_Element] = ImVec4(0.10f, 0.20f, 0.30f, 0.40f);
```

You need to convert that into:

```
imgui.PushStyleColor(imgui_col.Element, {0.10, 0.20, 0.30, 0.40});
```

I made a very lazy plugin, that converts the C++ code into Lua code, so you don't have to do it yourself. Make sure to rename the plugin file to plugin.lua and put everything into a directory in your plugins folder.

4 Interact with the editor and maps

Make sure to check out the available resources to see which values you can access! The common workflow for placing any kind of object is going to be **converting values into an object with an utility function and then placing/removing them with the according action function**.

Remember:

1. Create Object with `utils.Create___()`
2. Place Object with `action.Place___()`

- Object
  - Utility function
  - Action function
  - ActionBatch function
- HitObject
  - `utils.CreateHitObject()`
  - `action.PlaceHitObject(obj)`
  - `action.PlaceHitObjectBatch(table)`
- ScrollVelocity
  - `utils.CreateScrollVelocity()`
  - `action.PlaceScrollVelocity(obj)`
  - `action.PlaceScrollVelocityBatch(table)`
- TimingPoint
  - `utils.CreateTimingPoint()`
  - `action.PlaceTimingPoint(obj)`
  - `action.PlaceTimingPointBatch(table)`

4.1 Place a single object

```
-- This is an example for a single hitobject.

-- Place a note at the current editor position in lane 1
-- You would place a long note by specifiying an end time, refer to
-- section Utilities
obj = utils.CreateHitObject(1, state.SongTime)
actions.PlaceHitObject(obj)

-- This is an example for a single SV point.

sv = utils.CreateScrollVelocity(1000, 1.5) -- offset, sv
actions.PlaceScrollVelocity(sv)
```

The same applies to timing points.

4.2 Place a batch of objects

Placing objects in a batch will result in a grouped undo/redo in the editor. Keep in mind that there aren't any "arrays" in the traditional sense, only tables (refer to Arrays/Lists/Tables). The following example shows how to work with a batch of SVs. The same concepts apply to hit objects and timing points! You can find an example in my SV plugin: `sv_linear()`, `editor_placeSVs()`.

```
svObject1 = utils.CreateScrollVelocity(1000, 1.5)
svObject2 = utils.CreateScrollVelocity(2000, 2.0)
```

```lua
-- There aren't any "arrays" or "lists" in lua, only tables
svList = {}
table.insert(svList, svObject1) -- Iterable with a for-loop
table.insert(svList, svObject2)

-- Alternatively use actions.PlaceScrollVelocityBatch({svObject1, svObject2})
actions.PlaceScrollVelocityBatch(svList)
```

5 Advanced concepts

5.1 State variable management

There are a few ways to go about this. One would be to use wrapper functions like this, every time we want to set up a new variable to use.

```
function draw()
    state.SetValue("n", get("n", 0) + 1)
end

function get(identifier, defaultValue)
    return state.GetValue(identifier) or defaultValue -- return default if nil
end
```

This gets really really confusing as the number of variables increase. You can't use the same identifier, or otherwise you're going to run into issues with retrieving the wrong values. Additionally, you have to use `get()` and `setValues()` every time you declare a new variable. Say you have 20 persistent values to keep across states? Gotta use 40 lines just to manage them.

This is why I have set up following system in my personal project:

```
function draw()
    imgui.Begin("Window")
    exampleMenu()
    imgui.End()
end

function exampleMenu()
    local menuID = "example"
    local vars = {
        myString = "default",
        myNumber = -1
    }

    retrieveStateVariables(menuID, vars)

    vars.myNumber = vars.myNumber + 1
    -- alternatively: vars["myNumber"] = vars["myNumber"] + 1

    saveStateVariables(menuID, vars)
end

function retrieveStateVariables(menuID, variables)
    for key in pairs(variables) do
        variables[key] = state.GetValue(menuID..key) or variables[key]
    end
end

function saveStateVariables(menuID, variables)
    for key in pairs(variables) do
        state.SetValue(menuID..key, variables[key])
    end
end
```

Every time you want to set up a new scope you want your variables to reside in (usually separate menus), you need to define a menu ID (to prevent other menus with identical variable names) and a variables (vars) table. The variables table is filled with the variables you want to define, along with their default values.

You call the `retrieveStateVariables()` function once after defining your vars table. You do your calculations and call the `saveStateVariables()` function once at the end of your scope. Now you only need to worry about your menu IDs being different.

If you want to track create a new state variable, all you need to do is create another element in the vars table. Your intelligent editor with IntelliSense will also provide autocompletion as soon as you type `vars.`.

5.2 Module management

Remember in A few things you can't do with plugins, when I told you you can't use a multifile structure? Next best thing you can do is simulate module behavior. Let's take the code example from the section above:

```
function draw() --[[ code ]] end
function exampleMenu() --[[ code ]] end
function retrieveStateVariables(menuID, variables) --[[ code ]] end
function saveStateVariables(menuID, variables) --[[ code ]] end
```

The most logical behavior I would put the functions into different files/modules would be

- menu
  - `exampleMenu()`
- util/management
  - `retrieveStateVariables()`
  - `saveStateVariables()`

When you put functions or variables into different files/modules, all you're really doing is defining a table with value and function assignments in another file, so in the end you can call your functions like `menu.example()` or `utilities.retrieveStateVariables()`. But if a module is just defining functions and values for a table, then that can be done in a single file as well!

```
menu = {}
util = {}

function draw()
    imgui.Begin("Window")
    menu.example()
    imgui.End()
end

function menu.example()
    local menuID = "example"
    local vars = { myNumber = -1 }

    utilities.retrieveStateVariables(menuID, vars)
    vars.myNumber = vars.myNumber + 1
    utilities.saveStateVariables(menuID, vars)
end
```

```lua
function utilities.retrieveStateVariables(menuID, variables)
    for key in pairs(variables) do
        variables[key] = state.GetValue(menuID..key) or variables[key]
    end
end

function utilities.saveStateVariables(menuID, variables)
    for key in pairs(variables) do
        state.SetValue(menuID..key, variables[key])
    end
end
```

It might not be as clean as using different files, but it's just as useful with IntelliSense and ordering your functions. You know what the next best thing would be though?

5.3 Using an actual multi-file structure

Now, this can't be done without external tools, because the sandboxed Lua environment doesn't allow for file reading or writing. This is how it would work though:

- Create a directory in your plugin folder
- Put all of your separate Lua files in
- Define functions and variables in each file like they were part of a table with the same name as the file
  - Example: `menu.lua` contains the functions `menu.information()` and `style.lua` contains `style.variable`
- Iterate over each Lua file in the directory and generate `fileName = {}` for each and write that to a new file `plugin.lua` in the parent directory
- Append all files in the directory to the created file

I'm using this system in my personal project iceSV and use Python for it. Feel free to look at `compile.py` and the output `plugin.lua`. There are a few pros and cons to this.

Pros:

- Less clutter
- Smaller files
- More structure
  - Makes you think more about how you want to structure your code

Cons:

- Not worth the trouble setting up for a small project
- The file does not automatically recompile on save (not that it worked anyway for me)
  - I made a AutoHotkey script that runs the `compile.py` when I press F6
- User has to download a lot more files when pulling with Git

5.4 Intellisense

The most frustrating part of making Editor plugins is having no autocomplete for any of the global constants (like state, map, actions etc.) and thus having to rely on the guide or the source code to find what you need. I remedied that issue by automatically generating a file that contains all of a constants available values and functions and keeping that file in my workspace. I'm using Visual Studio Code with the `sumneko.lua` extension, which picks up the intellisense file in the workspace to use as available functions and values in other Lua files. You can find the repository for that intellisense file here.

6 Noteworthy findings

6.1 Instant plugin update

While Swan has implemented the feature that the plugin updates automatically on file change, it actually doesn't work for me. It only (partially) works, when I write to the file via a Python script, instead of using my usual editor (VSCode). It complains about a file access error, which I don't seem to understand.

The "partially" part of the script save is that only basic text changes work. If anything regarding functions, button behaviors or similar is changed, the plugin loads to the point to where everything stayed the same and then simply stop loading (needs further confirmation).

So whether I save the plugin file via my editor or a script, I still have to leave and reenter the editor to view an updated version of my plugins.

6.2 Slider/Drag Int/Float 4 types

It seems that the order of return values is not like you'd expect. The order is actually 4,1,2,3. Here's a small example:

```
local n1,n2,n3,n4
local vars = {n1,n2,n3,n4}
_, vars = imgui.DragFloat4("label", vars)
n4,n1,n2,n3 = vars
```

I know, it's weird. But there isn't really anything you can do about it. I assume it's got something to do with how Lua table indexing starts at 1.

6.3 Vector2/3/4 Datatypes

ImGui uses a custom array datatype that contains 2/3/4 elements, called Vector2 (or Vector3/Vector4), depending on the amount of elements. If a function asks for it, you can use the function `imgui.CreateVector2(int x, int y)` (or the 3/4 element equivalents) to create a vector of that type that you can pass to the function. You can (almost always) use a regular table of length 2/3/4 instead though, since it has been observed to work as well. The only instance I have personally found it to not work is for the size parameter in the plot functions, where you're forced to create a vector using said function as it won't work otherwise.

7 Available resources

All of the code blocks in this file are automatically generated from Quaver's source code.

7.1 Quaver Enums

7.1.1 GameMode

Accessible in Lua with `game_mode.Keys4`.

```
// Quaver/Quaver.API/Quaver.API/Enums/GameMode.cs

Keys4 = 1,
Keys7 = 2
```

7.1.2 Hitsounds

Accessible in Lua with `hitsounds.Normal`.

```
// Quaver/Quaver.API/Quaver.API/Enums/Hitsounds.cs

Normal = 1 << 0, // This is 1, but Normal should be played regardless if it's 0 or 1.
Whistle = 1 << 1, // 2
Finish = 1 << 2, // 4
Clap = 1 << 3 // 8
```

7.1.3 TimeSignature

Accessible in Lua with `time_signature.Quadruple`.

```
// Quaver/Quaver.API/Quaver.API/Enums/TimeSignature.cs

Quadruple = 4,
Triple = 3,
```

7.2 ImGui Enums

The PascalCase name is simply converted into a snake_case variant.

Enum	Accessible in Lua with
ImGuiInputTextFlags	imgui_input_text_flags
ImGuiDataType	imgui_data_type
ImGuiTreeNodeFlags	imgui_tree_node_flags
ImGuiSelectableFlags	imgui_selectable_flags
ImGuiMouseCursor	imgui_mouse_cursor
ImGuiCond	imgui_cond
ImGuiWindowFlags	imgui_window_flags
ImGuiDir	imgui_dir
ImGuiDragDropFlags	imgui_drag_drop_flags
ImGuiTabBarFlags	imgui_tab_bar_flags
ImGuiTabItemFlags	imgui_tab_item_flags
ImGuiColorEditFlags	imgui_color_edit_flags
ImGuiCol	imgui_col

In-depth structure of enums can be found in ImGui.NET/C#_Enums and in ImGui/C_Enums.

7.3 Quaver Structures

7.3.1 HitObjectInfo

```
// Quaver/Quaver.API/Quaver.API/Maps/Structures/HitObjectInfo.cs

// The time in milliseconds when the HitObject is supposed to be hit.
int StartTime { get; [MoonSharpVisible(false)] set; }

// The lane the HitObject falls in
int Lane { get; [MoonSharpVisible(false)] set; }

// The endtime of the HitObject (if greater than 0, it's considered a hold note.)
int EndTime { get; [MoonSharpVisible(false)] set; }

// Bitwise combination of hit sounds for this object
HitSounds HitSound { get; [MoonSharpVisible(false)] set; }

// The layer in the editor that the object belongs to.
int EditorLayer { get; [MoonSharpVisible(false)] set; }

bool IsEditableInLuaScript { get; [MoonSharpVisible(false)] set; }

IEqualityComparer<HitObjectInfo> ByValueComparer { get; }
```

7.3.2 SliderVelocityInfo

```
// Quaver/Quaver.API/Quaver.API/Maps/Structures/SliderVelocityInfo.cs

// The time in milliseconds when the new SliderVelocity section begins
float StartTime { get; [MoonSharpVisible(false)] set; }

// The velocity multiplier relative to the current timing section's BPM
float Multiplier { get; [MoonSharpVisible(false)] set; }

bool IsEditableInLuaScript { get; [MoonSharpVisible(false)] set; }

IEqualityComparer<SliderVelocityInfo> ByValueComparer { get; }
```

7.3.3 TimingPointInfo

```
// Quaver/Quaver.API/Quaver.API/Maps/Structures/TimingPointInfo.cs

// The time in milliseconds for when this timing point begins
float StartTime { get; [MoonSharpVisible(false)] set; }

// The BPM during this timing point
float Bpm { get; [MoonSharpVisible(false)] set; }

// The signature during this timing point
TimeSignature Signature { get; [MoonSharpVisible(false)] set; }
```

```
bool IsEditableInLuaScript { get; [MoonSharpVisible(false)] set; }

IEqualityComparer<TimingPointInfo> ByValueComparer { get; }
```

7.4 State

Accessible via `state.attribute`.

```
// Quaver/Quaver.Shared/Screens/Edit/Plugins/EditorPluginState.cs

// The current time in the song
double SongTime { get; [MoonSharpVisible(false)] set; }

// The objects that are currently selected by the user
List<HitObjectInfo> SelectedHitObjects { get; [MoonSharpVisible(false)] set; }

// The current timing point in the map
TimingPointInfo CurrentTimingPoint { get; [MoonSharpVisible(false)] set; }

// The currently selected editor layer
EditorLayerInfo CurrentLayer { get; [MoonSharpVisible(false)] set; }

// The currently selected beat snap
int CurrentSnap { get; [MoonSharpVisible(false)] set; }
```

```
// Quaver/Quaver.Shared/Scripting/LuaPluginState.cs

// The time elapsed between the previous and current frame
double DeltaTime { get; set; }

// Unix timestmap of the current time
long UnixTime { get; set; }

bool IsWindowHovered { get; set; }

// Width and height of the current Quaver window
Vector2 WindowSize { get; set; }
```

```
// Quaver/Quaver.Shared/Scripting/LuaPluginState.cs

// Gets a value at a particular key
object GetValue(string key);

// Sets a value at a particular key
void SetValue(string key, object value);
```

7.5 Map

Accessible via `map.attribute`.

```
// Quaver/Quaver.Shared/Screens/Edit/Plugins/EditorPluginMap.cs

// The game mode of the map
```

```
GameMode Mode { get; [MoonSharpVisible(false)] set; }

// If the scroll velocities are in normalized format (BPM does not affect scroll velocity).
bool Normalized { get; [MoonSharpVisible(false)] set; }

// The slider velocities present in the map
List<SliderVelocityInfo> ScrollVelocities { get; [MoonSharpVisible(false)] set; }

// The HitObjects that are currently in the map
List<HitObjectInfo> HitObjects { get; [MoonSharpVisible(false)] set; }

// The timing points that are currently in the map
List<TimingPointInfo> TimingPoints { get; [MoonSharpVisible(false)] set; }

// The non-default editor layers that are currently in the map
List<EditorLayerInfo> EditorLayers { get; [MoonSharpVisible(false)] set; }

// The default editor layer
EditorLayerInfo DefaultLayer { get; [MoonSharpVisible(false)] set; }

// Total mp3 length
double TrackLength { get; [MoonSharpVisible(false)] set; }
```

```
// Quaver/Quaver.Shared/Screens/Edit/Plugins/EditorPluginMap.cs

string ToString();

int GetKeyCount(bool includeScratch = true);

// Finds the most common BPM in the current map
float GetCommonBpm();

// Gets the timing point at a particular time in the current map.
TimingPointInfo GetTimingPointAt(double time);

// Gets the scroll velocity at a particular time in the current map
SliderVelocityInfo GetScrollVelocityAt(double time);

// Finds the length of a timing point.
double GetTimingPointLength(TimingPointInfo point);

// Gets the nearest snap time at a time to a given direction.
double GetNearestSnapTimeFromTime(bool forwards, int snap, float time);
```

7.6 Editor Actions

Accessible via `actions.function()`. Reminder: Any place/remove function needs to be called in Lua with an object created by the appropriate utility function! Refer to: Interacting with the editor and maps

```
// Quaver/Quaver.Shared/Screens/Edit/Actions/EditorActionManager.cs

// Detects if the user has made changes to the map before saving.
```

```
bool HasUnsavedChanges => UndoStack.Count != 0 && UndoStack.Peek() != LastSaveAction || UndoStack.Cou

// Performs a given action for the editor to take.
void Perform(IEditorAction action);

// Undos the first action in the stack
void Undo();

// Redos the first action in the stack
void Redo();

void PlaceHitObject(HitObjectInfo h);

HitObjectInfo PlaceHitObject(int lane, int startTime, int endTime = 0, int layer = 0, HitSounds hitso

void PlaceHitObjectBatch(List<HitObjectInfo> hitObjects);

// Removes a HitObject from the map
void RemoveHitObject(HitObjectInfo h);

// Removes a list of objects from the map
void RemoveHitObjectBatch(List<HitObjectInfo> objects);

// Resizes a hitobject/long note to a given time
void ResizeLongNote(HitObjectInfo h, int originalTime, int time);

// Places an sv down in the map
void PlaceScrollVelocity(SliderVelocityInfo sv);

// Places a batch of scroll velocities into the map
void PlaceScrollVelocityBatch(List<SliderVelocityInfo> svs);

// Removes a batch of scroll velocities from the map
void RemoveScrollVelocityBatch(List<SliderVelocityInfo> svs);

// Changes the offset of a batch of scroll velocities
void ChangeScrollVelocityOffsetBatch(List<SliderVelocityInfo> svs, float offset);

// Changes the multiplier of a batch of scroll velocities
void ChangeScrollVelocityMultiplierBatch(List<SliderVelocityInfo> svs, float multiplier);

// Adds a timing point to the map
void PlaceTimingPoint(TimingPointInfo tp);

// Removes a timing point from the map
void RemoveTimingPoint(TimingPointInfo tp);

// Places a batch of timing points to the map
void PlaceTimingPointBatch(List<TimingPointInfo> tps);

// Removes a batch of timing points from the map
void RemoveTimingPointBatch(List<TimingPointInfo> tps);
```

```
// Changes the offset of a timing point
void ChangeTimingPointOffset(TimingPointInfo tp, float offset);

// Changes the BPM of an existing timing point
void ChangeTimingPointBpm(TimingPointInfo tp, float bpm);

// Changes a batch of timing points to a new BPM
void ChangeTimingPointBpmBatch(List<TimingPointInfo> tps, float bpm);

// Moves a batch of timing points' offsets by a given value
void ChangeTimingPointOffsetBatch(List<TimingPointInfo> tps, float offset);

// Resets a timing point back to zero
void ResetTimingPoint(TimingPointInfo tp);

// Adds an editor layer to the map
void CreateLayer(EditorLayerInfo layer);

// Removes a non-default editor layer from the map
void RemoveLayer(EditorLayerInfo layer);

// Changes the name of a non-default editor layer
void RenameLayer(EditorLayerInfo layer, string name);

// Changes the editor layer of existing hitobjects
void MoveHitObjectsToLayer(EditorLayerInfo layer, List<HitObjectInfo> hitObjects);

// Changes the color of a non-default editor layer
void ChangeLayerColor(EditorLayerInfo layer, Color color);

// Toggles the visibility of an existing editor layer
void ToggleLayerVisibility(EditorLayerInfo layer);

void GoToObjects(string input);

void SetHitObjectSelection(List<HitObjectInfo> hitObjects);

// Resnaps all notes in a given map to the closest of the specified snaps in the list.
void ResnapAllNotes(List<int> snaps, List<HitObjectInfo> hitObjectsToResnap);

// Detects the BPM of the map and returns the object instance
EditorBpmDetector DetectBpm();

void SetPreviewTime(int time);

// Triggers an event of a specific action type
void TriggerEvent(EditorActionType type, EventArgs args);

void Dispose();
```

7.7 Utilities

Accessible via `utils.function()`, refer to Structures for more information on the returned objects.

```
// Quaver/Quaver.Shared/Screens/Edit/Plugins/EditorPluginUtils.cs

SliderVelocityInfo CreateScrollVelocity(float time, float multiplier);

HitObjectInfo CreateHitObject(int startTime, int lane, int endTime = 0, HitSounds hitsounds = 0, int

TimingPointInfo CreateTimingPoint(float startTime, float bpm, TimeSignature signature = TimeSignature

EditorLayerInfo CreateEditorLayer(string name, bool hidden = false, string colorRgb = null);

// Converts milliseconds to the appropriate mm:ss:ms time
string MillisecondsToTime(float time);

bool IsKeyPressed(Keys k);

bool IsKeyReleased(Keys k);

bool IsKeyDown(Keys k);

bool IsKeyUp(Keys k);
```