Sigian User Manual

Version 1.1.5

Introduction

Thank you for downloading the Sigian Quality of life Improvements for Unity (or SQOLI)! This is the User Manual, explaining in depth every feature available, as well as tips on how to use them, and practical examples for newbies.

Please note the purpose of this "plugin" is to help game design students, if you're a professional gamedev or programmer, it has next to no use. Feel free to give it a go though.

Hopefully, you will find helpful features for faster prototyping, so it will be easier for you to test game-play ideas, or make proper games in a short period of time.
Please note, the goal of SQOLI is to help you work better and faciliate scripting, not to do the work for you. Half the fun of prototyping is solving technical problems!

If you encounter any unlikely bugs, want to suggest a feature for future updates, please hit me up on Twitter @gregoire_meyer

Cheers,

Installation

1. Create a blank unity project.

2. Unzip the Sigian.zip archive

3. Place all content of the archive in the Assets folder of your Unity project (in the file explorer, not in Unity)

4. Optional: move SigianLayout.wlt from EditorResources folder to Appdata/Roaming/Unity/Editor-5.x/Preferences/Layouts

You're good to go!

If you are updating from a previous version, just replace all files from Editor and Scripts/[Sigian] with the new ones. Done.

Chapter 1 : Utility Window, time management and other things

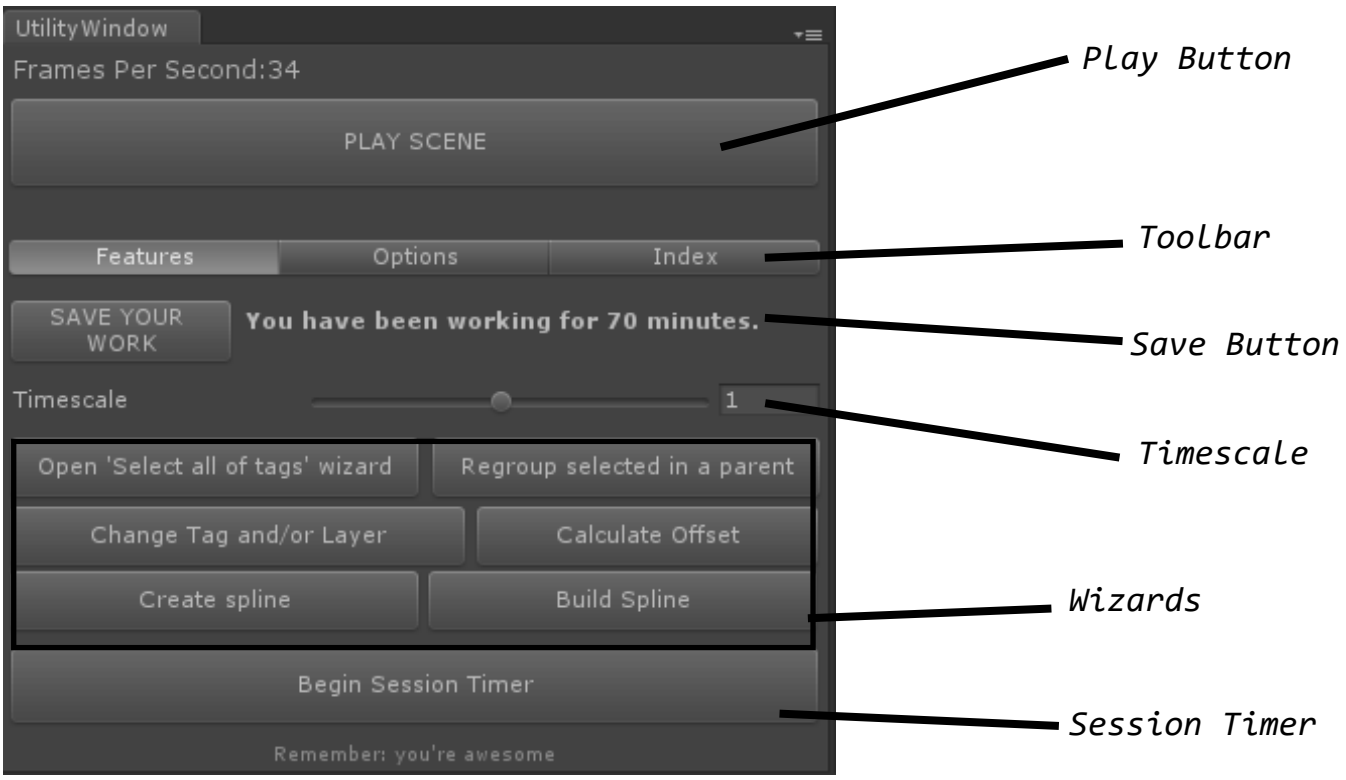
In this chapter we are going to explore the Sigian Utility Window, and what time management-related features it has. For explanations on wizards, please read Chapter 2: Wizards.

To open the Utility Window, click on Sigian -> Utility Window on top of the screen. You can also use the Sigian UI Layout provided.
This layout is optimized for maximum readability with SQOLI on a 1920*1080 screen.

Then, simply select the Sigian Layout with the Layout menu on top-right of the screen.

General View

The Utility Window will be your main contact with the plugin features, especially if you are a game or level designer on the project. Here is a global view of the window:



Play Scene Button: Use this button to play, pause or stop scene. It has exactly the same effect as the 3 buttons on top of the screen, but are easier to reach. When the scene is playing, this button splits into 2 for pausing and stopping.

— — — — — — — — — — — — — — — —

Save your work Button: Use this button to save the scene. The Utility Window also has an autosave feature, that is detailed later.

— — — — — — — — — — — — — — — —

Timescale slider: Use this slider to adjust timescale in the scene. Very useful if you want to track glitches and / or visual bugs, such as AnimationState-related issues. 1 is normal timescale, 0.5 is half speed, etc…

— — — — — — — — — — — — — — — —

Regroup selected in a parent Button: This button regroups all your selected GameObjects in a common parent. Bonus feature, if all your selected GameObjects are duplicated from a same object, SQOLI will detect it and rename the parent accordingly. It even detects architecture-related naming and ignores it. For example, if all your selection is SM_Wall duplicates, and regroup them, the parent will automatically be named Walls.
This action can also be achieved by pressing Ctrl + G.

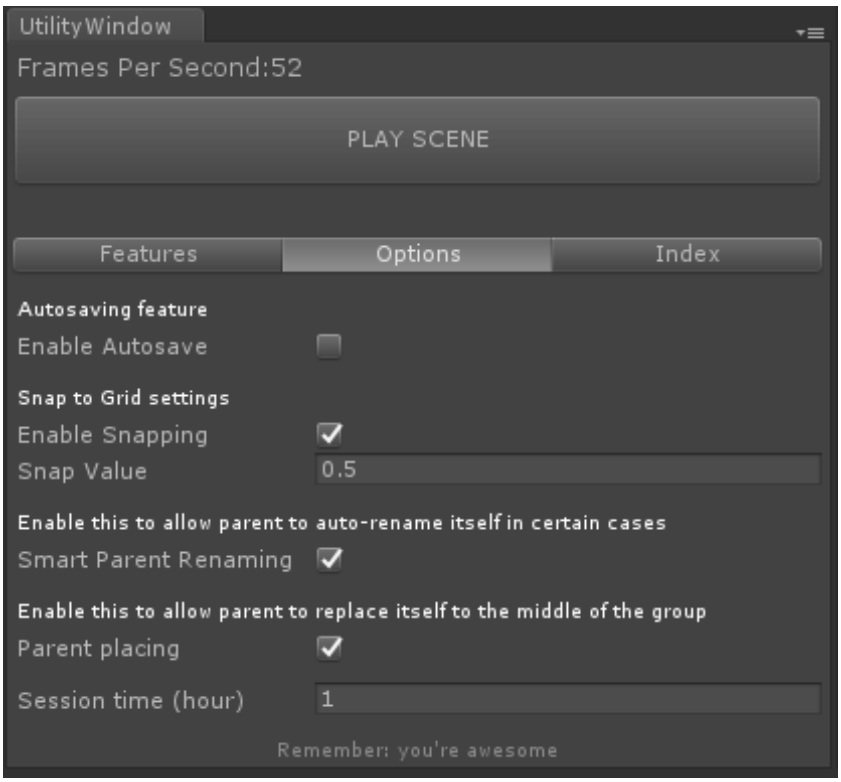
— — — — — — — — — — — — — — — —

The 5 other buttons (not counting Begin Session Timer Button) open **wizards**, that's why they will be detailed in Chapter 2.

— — — — — — — — — — — — — — — —

Begin Session Timer: This button is a time management feature. If you are easily distracted or you have issues with working, you can use this button. Once pressed, a gauge will appear, progressively filling (default is 1 hour to fill). When full, a popup will appear congratulating you for working a whole session. You can modify session length in the options.

Options



Autosaving: enables or disables autosaving. When enabled, you can modify the autosave frequency (here, every 10 minutes)

Enable Snapping and Snap Value: enables or disables automatic grid snapping for gameObjects, and sets the grid snapping value (1 is 1 unit, 0.5 is half, etc...)

Smart Parent Renaming: enables or disables parent renaming on regrouping items. If disabled, SQOLI will not try to detect a pattern in the selected GameObjects and set the default name for the parent, which is "PLEASE RENAME".

Parent Placing: if enabled, the parent will be placed on the average position of every selected GameObjects on regrouping.

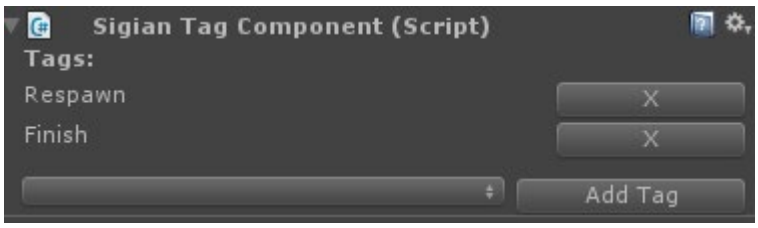
Session Time: Use this to set the session time length (example: 0.5 is half an hour).

Index

The index of the utility window is just a list of useful keyboard shortcuts for Unity.
Sorry, no revolutionnary feature here, just a plain, good old tip list.

Multiple tags per GameObject

This feature was very demanded when I began developing Sigian. It allows a gameObject to have multiple tags, and to look for them directly in code, or in a wizard.
Previously, this was a standalone tag system, but in Sigian 1.1.5 I merged it with the Unity tag system, so it would be easier to use and more robust.

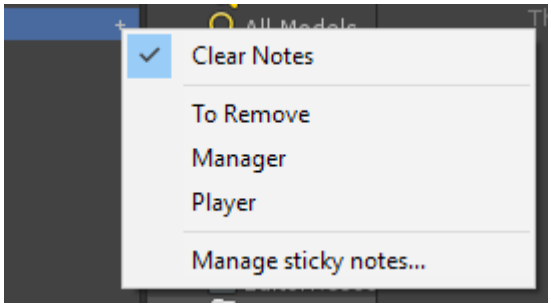


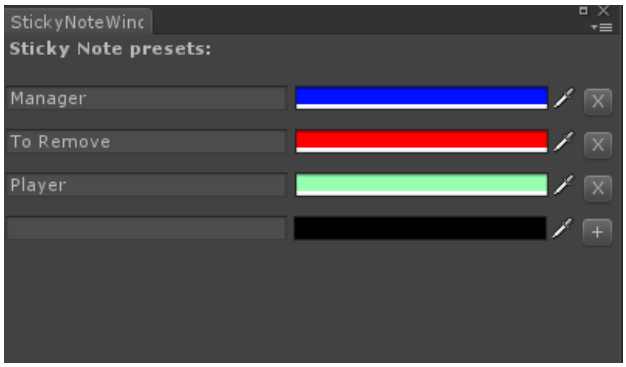
To use this tag system, you must add the "Sigian Tag Component" component on a gameObject.
Once this is done, you can directly add or remove tags via the tag field and the "Add Tag" button.
Eveything is saved in a tag manager, and you will be notified when it is created (usually at the beginning of a project).

Sticky Notes on Hierarchy

This feature is very useful for teamwork, and for personal reminders. It simply allows you to highligh gameObjects in the hierarchy, and attach messages on them, choosing from your own presets (via a separate tool). The sticky notes are saved, there is one save file per scene (untitled scenes don't get saved), so it will also work with source control solutions.

To put a sticky note, just press the "+" button next to a selected gameobject on the hierarchy. A dropdown menu will appear. "Clear Notes" means no note, "Manage sticky notes..." opens the preset tool (see below).





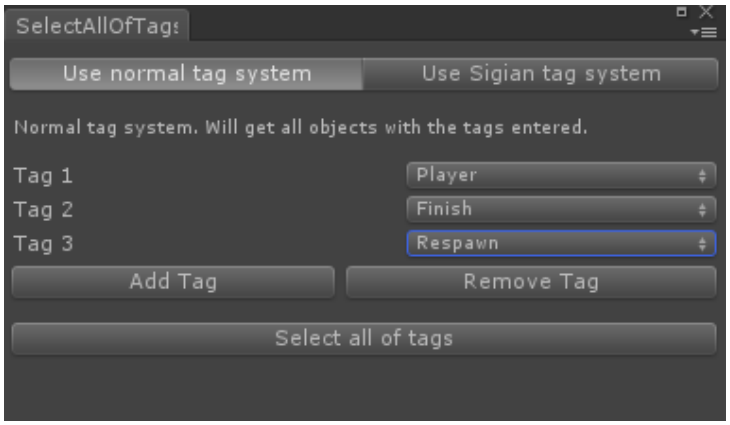
Chapter 2 : Wizards

[Insert Harry Potter joke here]

The wizards are a key feature of SQOLI, especially for level designers. There are 4 of them, each designed for time saving. In this chapter they will be explained in depth.

Select all of Tags

New in version 1.1.5, this wizard has been entirely remade and now supports Sigian tag system. You can select all gameObjects of at most 9 tags for Unity tag system, 3 tags for Sigian tag system. You can add/remove tags from your selection with the "Add tag" and "Remove Tag" buttons



It is a very simple wizard, meant to be quickly opened and quickly closed.
To open it, simply press "Open Select All of Tags Wizard" button on the Utility Window.

Change Tag or Layer

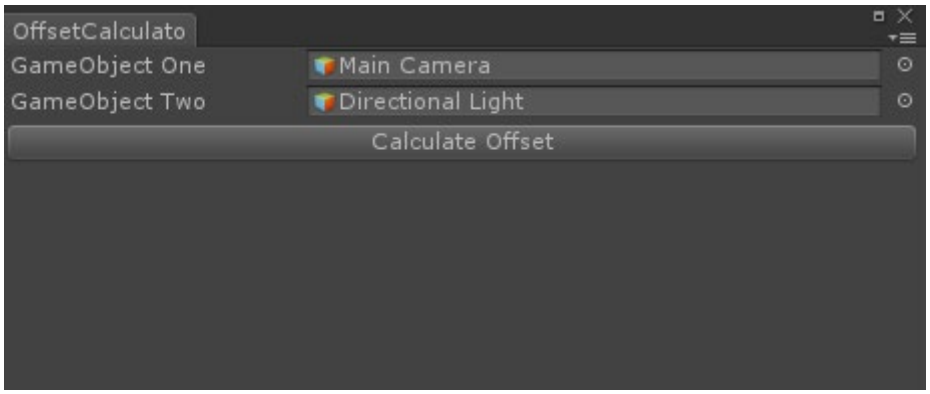
Use this wizard to quickly change the tag and/or layer of all the GameObjects of your selection. Useful for modifying quickly multiple gameObjects.

Again, this is a very simple wizard, yet it is time-saving.

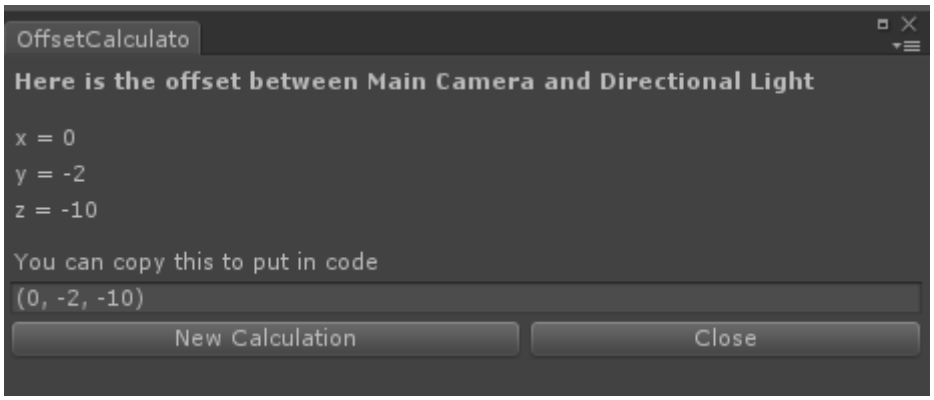
Offset Calculator

This wizard is very helpful to get the offset between 2 GameObjects of your choice. For example, if you are creating a modular generation system (with same-sized chunks), you can vertex-snap them, get the offset, paste it in your code and get a seamless transition between chunks.

To open it, just press the "Calculate Offset" button in the Utility Window.



Click "Calculate Offset" and you will get this screen:

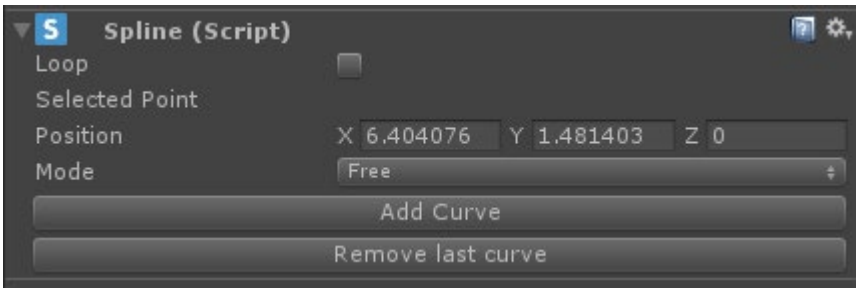


You can then copy/paste the vector3 in your code. You can also make a new offset calculation, or close the wizard.

Spline Creator and Builder

By far the most complex wizard, and perhaps feature, of SQOLI.
Splines are a type of curve you can edit in 3D. They use the mathematical concept of Bezier curves, and derivatives.

You can create a Spline by pressing the "Create Spline" button in the Utility Window. When selected, the inspector will display several buttons.

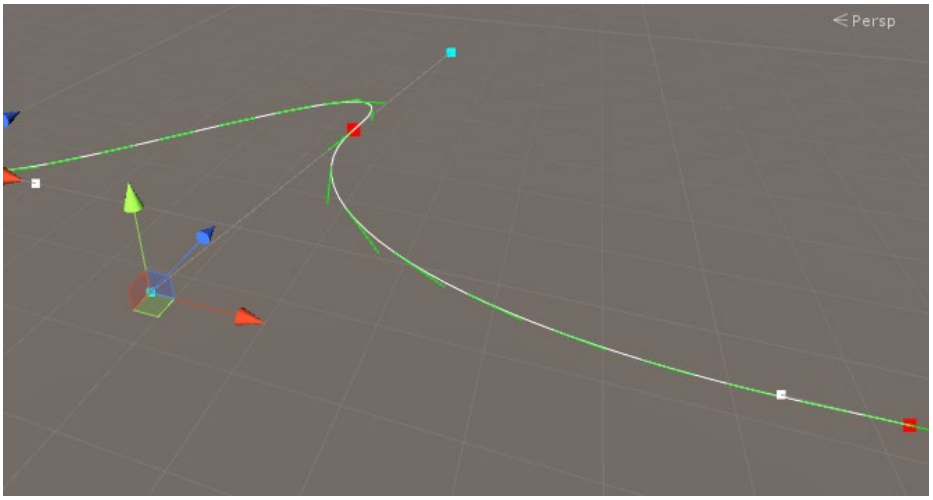


The **Loop** toggle decides if this Spline is a loop.
There are 2 types of points in the Spline: major points (in red, bigger), which determine the position of the curve, and the minor (yellow, smaller), which determine the orientation the curve will take at the associated major point.
Each major point is linked to 2 minor points.

There are 3 **Modes** for each point of the curve: Free, where there is no limitation, Aligned, where the 2 minor points are aligned with each other (but not distance, which allows for more asymmetrical curves), and Mirrored, where the 2 minor points are aligned and at the same distance from the major point, allowing for more harmonious curves.

The **Add Curve** button adds 1 major point and 2 minor points at the end of the curve, allowing to make more complex curves.

The **Remove last Curve** button does exactly the opposite: it deletes the last 3 points of the curve.



With enough training, you can easily make harmonious curves.

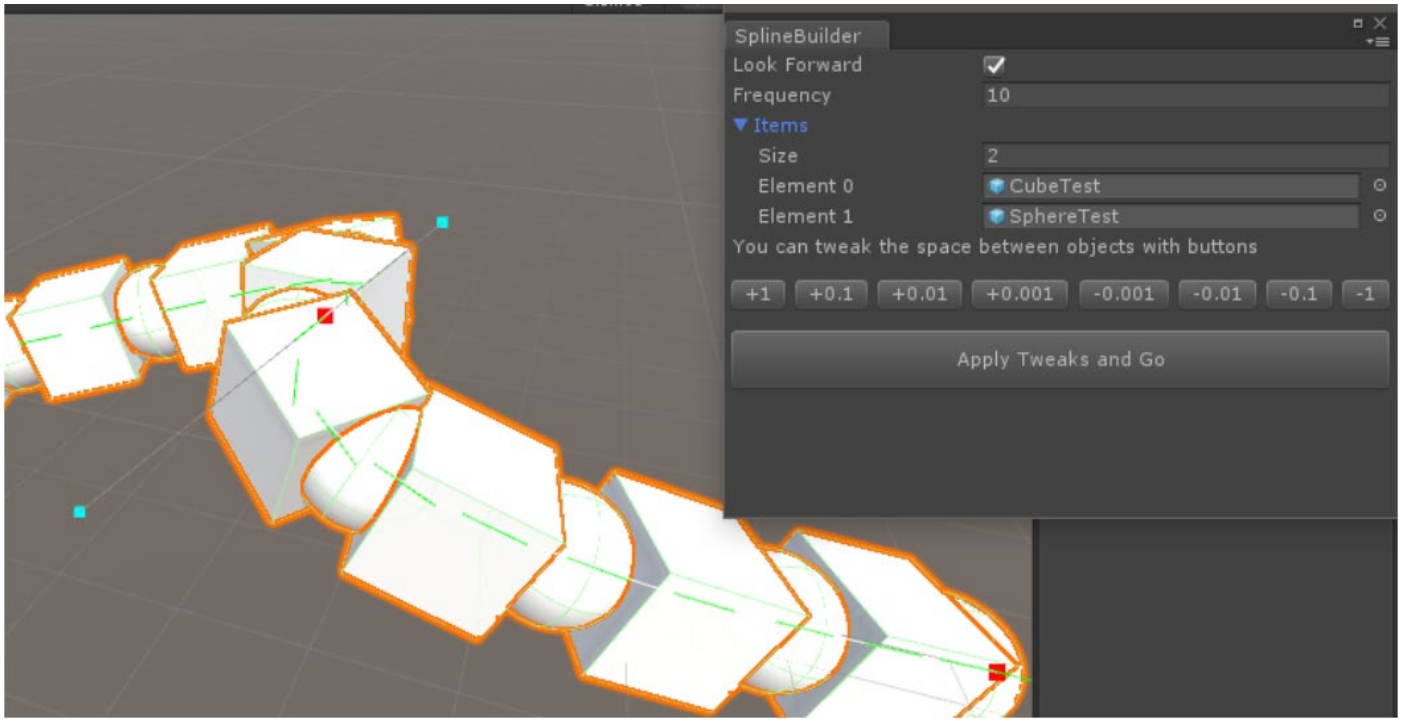
Spline Builder

Disclaimer: originally, it was planned to make a Spline Follower Wizard as well, but Cinemachine arrived on Unity, so it became more or less deprecated.

The Spline Builder wizard allows you to automatically place prefabs on a Spline, aligned with the curve direction or not. You can also easily tweak the spacing between objects.

To build a Spline, simply press the "Spline Builder" button in the Utility Window.

If you have only a Spline selected, you will be able to move on. However, if you don't you will have to specify which Spline you want to build in the wizard.



The **Look Forward** toggle decides if the instanciated objects are aligned with curve direction.
The **Frequency** is the number of times the objects will be instanciated.
The **Items list** is where you put the prefabs you want. They will be instanciated in order.

Finally, you will have a **Build Spline** button, which will build the Spline. You will then have a tweaking menu for you to modify the spacing between objects, change the frequency, etc…
Click "Apply Tweaks and Go" to close the wizard.

If you want to tweak the Spline after, simply open again the Spline Builder wizard, builder data is saved for each Spline in the SplineData component.

Chapter 3 : Coding

This is the chapter where the coding features that SQOLI brings are detailed. Nearly every feature is explained with a practical example, except for the really simple ones.

In order to use SQOLI coding features, you must include the Sigian namespace at the beginning of every script you plan to use them. You can do this just by typing this at the top of your script:

```
using Sigian;
```

Structs

Structs are a type of data you can declare as a variable, like a GameObject or a Sprite. They don't need to be constructed, though they can be.

```
Animator myAnim = new Animator();
```
A class being constructed

SQOLI features 3 brand new structs, thought for either improving work efficiency, or simplifying complex process.

MinMax

`MinMax` is a struct composed of 3 floats: a minimum, a maximum, a value, and a boolean, breakIfError. The value is clamped between the min and the max, and breakIfError determines if the script is stopped when the value tries to go beyond limits.

This struct is useful for making a health system for a player, for example, or an upgradable stat.

The `MinMax` struct can be declared and/or constructed in 4 ways:

```
MinMax test1;
MinMax test2 = new MinMax(minimum, maximum);
MinMax test3 = new MinMax(minimum, maximum, value);
MinMax test4 = new MinMax(minimum, maximum, value, breakIfError);
```

After declaring a MinMax, you can edit its properties with the .`value`, the .`min`, the .`max` and the .`breakIfError` commands.

```
test1.min = -2.3f;
float testfloat = test1.max;
test1.value = 5;
bool isValid = false;
test1.breakIfError = isValid;
```
A few examples on how you can edit MinMax properties

There are also a few functions (voids) for quick settings. The `.SetToMin()` and the `.SetToMax()` commands set the value respectively to the minimum and the maximum of the MinMax.

The `MinMaxVector` command returns the minimum and the maximum as a Vector2, which may be useful.

```
//Will set at 10%
test1.SetToRatio(0.1f);

Vector2 testVector = test1.MinMaxVector;
```

Finally the last function is the `SetToRatio(float)` void. It sets the value between min and max at the ratio specified. For example, `.SetToRatio(0.5f)` will set the value halfway between the min and the max.

The `MinMax` struct features a custom PropertyDrawer, so basically when you declare it public, you will be able to modify it directly in the Inspector.

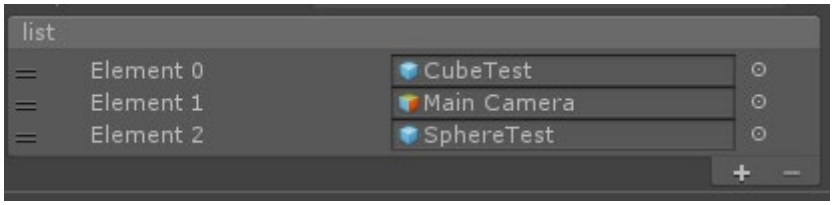
You will get this interface, featuring a slider and 3 float fields for manual input.

SigianReorderableGameObjectList

Behind this barbaric name, hides a very useful feature.

Normally, you cannot reorder arrays and Lists in the editor when you declare them public. Now, you can with the `SigianReorderableGameObjectsList` struct, and more precisely its custom PropertyDrawer. It behaves just like a `List<GameObject>`, but you can reorder the elements in the editor when you declare it public. You can also add and delete elements directly in the inspector.

This struct is meant to be public. If you want a private list or array, just use `List<>` or `array[]`.



Why only GameObjects when a List can do any type of data? Complex, yet short answer: Unity doesn't allow generic type serialization. And GameObject is the most flexible type, since you can GetComponent<>() after.

You can get elements of the list at a given index (if you couldn't, what would be the point of a List after all?).

Yes, there is only one code instruction, but this struct is meant to be used mostly in the Inspector.

```
public SigianReorderableGameObjectList test1;

// Use this for initialization
void Start () {

    GameObject testGO = test1.GetAtIndex(2);
}
```

SigianEvent

Delegate events are a relatively complex feature for a newbie, and is not very intuitive (but then again, programming is not very intuitive in itself).
With the `SigianEvent` struct, you can create events more easily, at a cost of flexibility.

SigianEvents must always be constructed with an argument (a dummy boolean), and you can only add parameterless voids in it.

You can "subscribe" voids to the `SigianEvent` with the `Add()` command, "unsubscribe" them with the `Remove` command, clear the SigianEvent from all voids with the `Clear()` command, and finally you can activate all the voids at once with the `Fire()` command.

```
SigianEvent test = new SigianEvent(true);

// Use this for initialization
void Start () {

    test.Add(Test1);
    test.Add(Test2);
    test.Fire();
}

void Test1()
{
    Debug.Log("test1");
}

void Test2()
{
    Debug.Log("test2");
}
```

Output: test1, test2.

Unlike the 2 other structs, SigianEvent has no property drawer, meaning every editing is made by code.
Sorry not sorry.

Extension methods

Basically, an extension method is a free DLC for data types. I don't know how to explain it better. Sorry.

OK let's take an example. GetRandom is an extension method for arrays and lists (of any type), that returns a random element from this array or list.

```
GameObject[] array;
GameObject randomElement;

// Use this for initialization
void Start () {

    randomElement = array.GetRandom();

}
```

You get the concept? Good.

So now is the part where I explain every extension method available in SQOLI. For clarity, I'm going to regroup them by the type of data they extend (or the "base game", to keep going with the DLC metaphor).

Please note: Each of these have a brief description in the code editor itself, thanks to some XML stuff. Also, please use Visual Studio.

Array (any type) and/or Lists (any type)

GetRandom() : returns a random element from this array or List. (explained previously)

Contains(element) : a boolean that returns true if the element of type T is contained within this array. Lists have this feature natively.

```
GameObject[] array;
GameObject randomElement;

List<string> bestMoviesEver = new List<string>();

// Use this for initialization
void Start () {

    bool isRandomInArray = array.Contains(randomElement);

    if (bestMoviesEver.Contains("Forrest Gump"))
    {
        //Do something
    }
}
```

```
string[] testString = { "test", "test", "test" };

// Use this for initialization
void Start () {

    if (testString.IsConsistent())
    {
        //Do something
    }
}
```

IsConsistent() : a boolean that returns true if every element from this Array or List is the same.

IsNullOrEmpty() : a boolean that returns true if this array or List is null or empty.

```
string[] testString = { "test", "test", "test" };

// Use this for initialization
void Start () {

    if (testString.IsNullOrEmpty())
    {
        //Do something
    }
}
```

```
string[] testString = { "test1", "test2", "test3" };

// Use this for initialization
void Start () {

    testString.Shuffle();

}
```

Shuffle() : shuffles this array or List. Useful for… stuff, I guess.

Push(T[] array or List<T>) : Pushes, or merges 2 arrays or Lists. It will result in a bigger array or list, elements will be kept in the same order, and the array in parameter will be put at the end of this array.

```
string[] testString = { "test1", "test2"};
string[] testString2 = { "test3", "test4" };

// Use this for initialization
void Start () {

    testString.Push(testString2);
    //output: test1, test2, test3, test4.
}
```

Strings

IsNullOrEmpty() : basically the same as the one for arrays and Lists. Works exactly in the same way, no picture needed.

WordCount() : returns the number of words in this string.

```
string testString = "Boy this manual is tough to write";

// Use this for initialization
void Start () {

    int testWordCount = testString.WordCount();
    //output: 7
}
```

```
string testString = "123456789";

// Use this for initialization
void Start () {

    if (testString.IsNumeric())
    {
        float testFloat;
        float.TryParse(testString, out testFloat);
        //converts the string into float
        //testFloat = 123456789
    }
}
```

IsNumeric() : returns true if this string is composed exclusively of numbers.
Bonus, how to convert a string into a float (natively supported).

Append() : adds a string at the end of this string, with a space.

```
string testString = "Hey";
string testString2 = "you.";

// Use this for initialization
void Start () {

    testString.Append(testString2);
    //output: Hey you.
}
```

```
string testString = "Hey";
string testString2 = "you.";

// Use this for initialization
void Start () {

    bool myBool = testString.ContainsChar('H');

    if (testString2.ContainsChar('.'))
    {
        //do something
    }
}
```

ContainsChar(char) : a boolean that returns true if this string contains a given character.

Ints

IsEven() : a boolean that returns true if this int is even.

```
int evenInt = 4;
int oddInt = 15;

// Use this for initialization
void Start () {

    bool testBool = evenInt.IsEven();

    if (oddInt.IsEven())
    {
        //do something
    }
}
```

```
int evenInt = 20;
int oddInt = 15;

// Use this for initialization
void Start () {

    bool testBool = evenInt.IsMultipleOf(10);

    int multiple = 5;
    if (oddInt.IsMultipleOf(multiple))
    {
        //do something
    }
}
```

IsMultipleOf(int) : a boolean that returns true if this int is a multiple of the given number.

GenerateRandomSeed(int) : generates a random seed of numbers, of a given length.

```
int seed;

// Use this for initialization
void Start () {

    seed.GenerateRandomSeed(5);
    //output 72695 for example
}
```

Bools

DiceRoll(float) : a boolean that returns true or false at a given probability. For example, 0.5 will grant a 50/50 chance to return true, 0.1 a 10% chance, etc...

```
bool testBool;

// Use this for initialization
void Start () {

    testBool.DiceRoll(0.5f);
    //50% chance for testBool to be true.
}
```

GameObjects

```
GameObject GO;

// Use this for initialization
void Start () {

    if (GO.HasComponent(typeof(Animator)))
    {
        //GO has an animtor, do stuff
    }
}
```

HasComponent(typeof(Type)) : a boolean that returns true if this GameObject has at least one component of the given type. Remember to put typeof() in the parameter.

Transforms

LongForward() : just a longer version of transform.forward, no picture needed really.

LongUp() : just a longer version of transform.up, no picture needed.

LongRight() : just a longer version of transform.right, no picture needed.

And that's all the extension methods available in SQOLI!
There are also a few experimental extension methods, that are in the Sigian.Experimental namespace. I won't detail them here, but you are free to use them at your peril, since they are largely untested and/or resource hungry.

Just to tease you a bit, there is a method for converting numbers into letters, one for purging a GameObject from all its components and one for converting numbers into hours/minutes/seconds.

Okay, that's it for the coding chapter. These features were designed primarily for my needs as a game designer, so they might not help you if you are making neural networks or an MMO backend. But if they help you prototype shitty games faster, it's good for me!

Chapter 4 : Architecture and Asset Post Processor

SQOLI features a pre-made project architecture, so it is best to use it at the very beginning of a project. It features folders for everything needed in a 3D project.

Nomenclature is clear and hierarchy is designed to be easily understood. The only touchy folders are the Editor, EditorResources, and the [Sigian] folder. They all contain the scripts used by SQOLI to work its magic. Please refrain from deleting files from these folders.

Don't worry, the plugin won't impact build size and process, as only the needed scripts will be kept in the build (such as the structs and extension methods). UtilityWindow and Wizards will automatically be ejected from the build, as they are in the Editor folder.

Asset Post Processor

With the pre-made architecture comes another cool feature: automatic asset process on import. If a certain type of asset is imported in a certain folder, it will be automatically be treated to be optimized.

For example, images imported in the UI folder will be optimized for 2D/UI (by changing Texture Type), whereas images imported in the Normal Map folder will be optimized for normal maps, saving you a few clicks for nothing.

The same goes for sounds, depending on if you put them in the SFX folder or the Music folder, they won't be compressed the same, resulting in increased performance and better quality. However, sound recompression is quite long in Unity, so sound importing is significantly longer than usual (around 20 seconds).

Conclusion

Again, thank you for installing Sigian Quality of Life Improvements. If you know anyone that might need it, or find any use of it, please share it to them, that means a lot to me.
SQOLI is bound to be updated, as I use is on personal projects and add features as I feel the need for them.

If you publish a project using SQOLI, no need to credit me, simply put "Thanks to the Sigian plugin", that would make me really happy. Well, you can credit me if you want, I won't mind ;)

Thank you again, and remember: you're awesome.

Cheers,
Grégoire Meyer