Game-Savvy / Byterizer Lite

Documentation

Thank you very much for supporting this asset!

The purpose of Game-Savvy is to provide useful and easy to use assets to the Unity community, all Game-Savvy assets have before anything, performance in mind, we always try to use good practices in our libraries and we also focus on good architecture, making all Game-Savvy products both, reliable, easy to maintain and to extend.

Byterizer Lite

This asset pack is aimed for people who want to use a better way to encapsulate data over the network, compared to normal serialization techniques, Byterizer Lite makes the process much faster and the encapsulated final size much smaller.

Results

Over several thousands of tests, run metrics showed that on average our Encoding/Decoding times take only **7.3%** of the total time it takes to use normal serialization/deserialization. Our packet sizes overall were also **20.8%** of the total size in comparison.
This makes using the Byterizer around **14 times faster** and with a **5 times smaller footprint** than normal serialization techniques.

Usage

Please keep in mind that you need to be using **.Net 4.5 or newer**.

After importing from the asset store or importing the UnityPackage, you will find a folder structure under:

"Assets/Plugins/GameSavvy/ByterizerLite/"

- Demo/
- Editor/
- ByterizerLite.dll
- ReadMREADME

The Demo folder will contain a scene and a script, please refer to those when staring up.
The scene contains a GameObject named “ByterizerLiteDemo” with the “ByterizerLiteDemo.cs” component attached to it.

In order to use the ByterizerLite, you will have to include the using directive:

```
using GameSavvy.Byterizer;
```

After that, you will be able to start using our libraries.

Simply create an instance of the ByteStream class, which is the one that contains the logic for the Encoding/Decoding of data.

Please refer to this example:

```
using UnityEngine;
using GameSavvy.Byterizer;

public class ByterizerLiteDemo : MonoBehaviour
{

    [ContextMenu("Append Demo")]
    private void AppendDemo()
    {
        //initialize object as empty
        var byteStream = new ByteStream();

        //Append parameters in order
        byteStream.Append(true);
        byteStream.Append(1);
        byteStream.Append(2f);
        byteStream.Append("Thsi is a string");

        print($"Byte Length → {byteStream.Length}");

        //Pop and print from Byte stream in order
        print(byteStream.PopBool());
        print(byteStream.PopInt32());
        print(byteStream.PopFloat());
        print(byteStream.PopString());
    }
```

Append: this adds a single parameter to the buffer, this is the fastest way to add parameters to the buffer, faster than using encoding but longer to write.

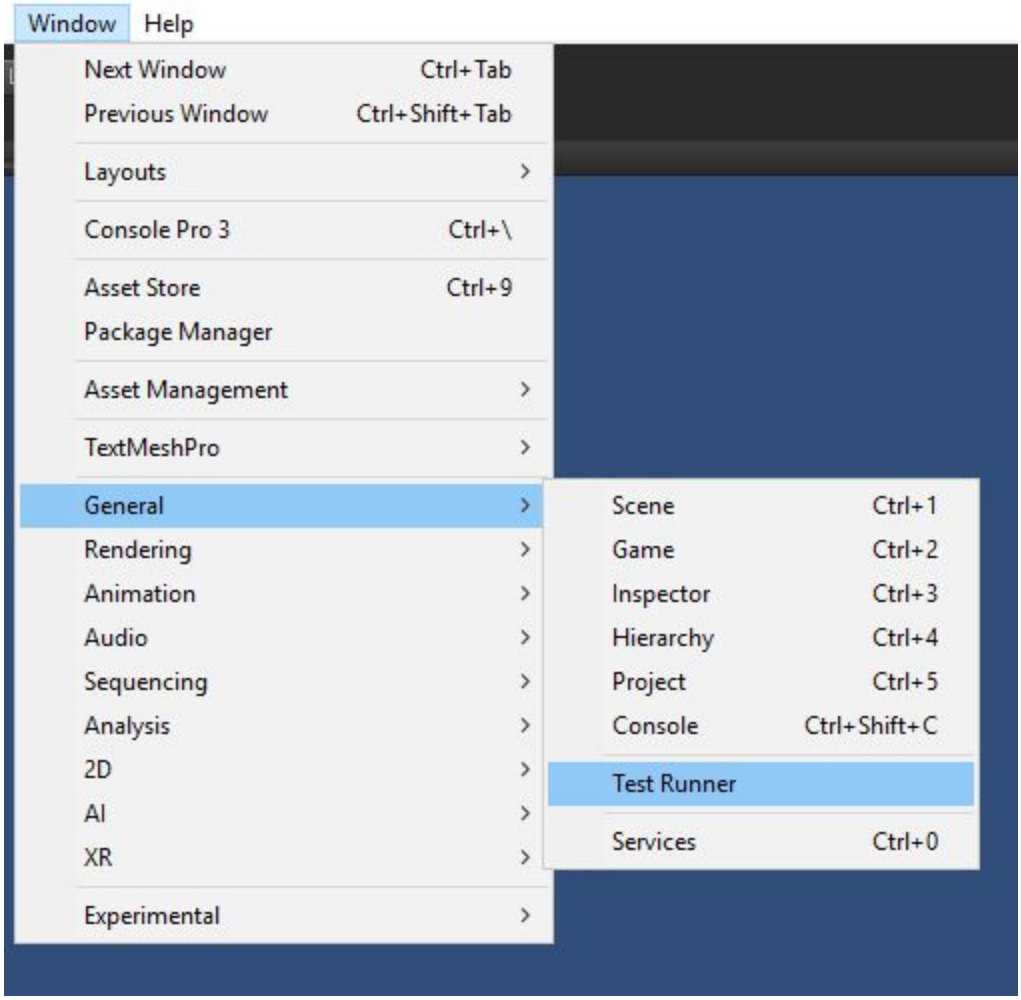
Encode: This can add multiple parameters to the buffer, keep in mind that this is slightly slower than directly appending parameters one by one (just a bit slower, not too bad).

Unit Tests

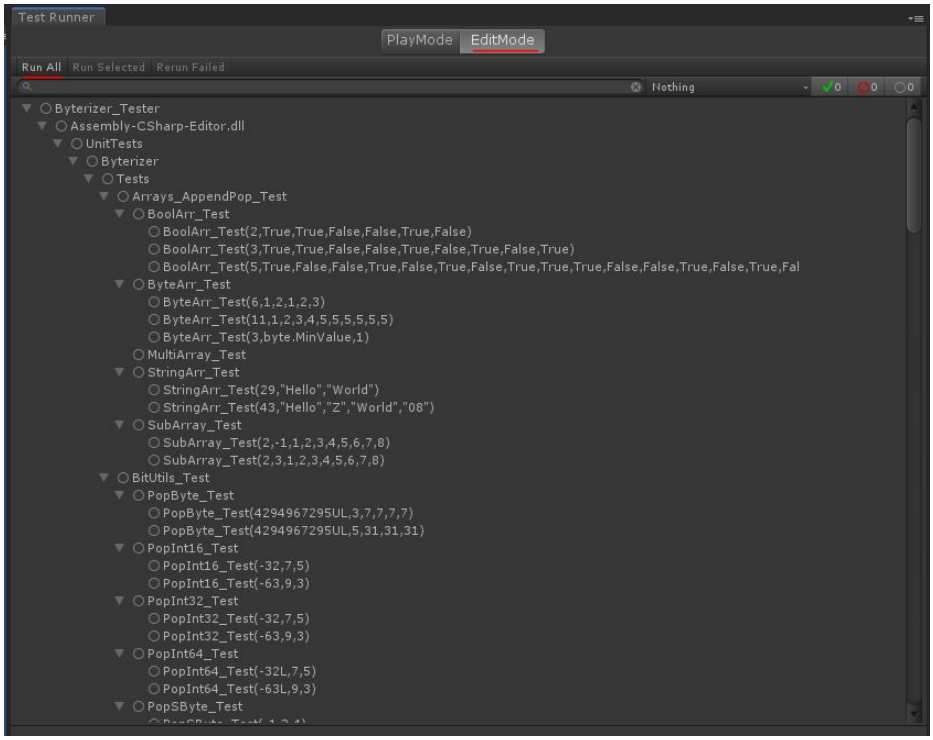
We have included over 170 Unit tests for you to check both functionality and usage of the entire API of the Byterizer Libraries.

How to run the test?
First, we need to have unity open the Test Runner

To Open the TestRunner in Unity:

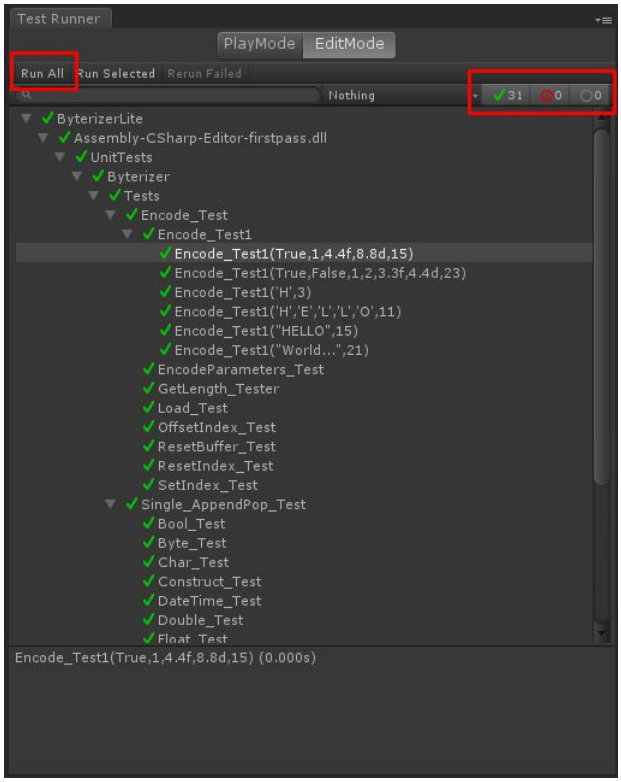


Pin the test runner somewhere and go to the **Edit Mode,** you will see something like this:



After this simply click on **Run All**, to run all *Unit Tests* and check that everything works.

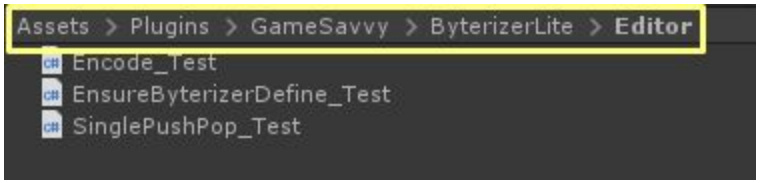
The results should show no error messages and all green flags.



Check the Unit Tests

The Unit Test contained in the project has the usage example for every single API function.

You can find the unit tests under our Editor folder, open any of the files with “_Test” as part of their name.



Each file pertains to a specific class in our library.

We hope you get the most out of the Byterizer not only for its Byte Encoding capacities but also for all of its Bit-Manipulation Utility libraries that make it so much easier to deal with parameters over the network and to mask/unmask bytes.

Byterizer vs Byterizer Lite

The Lite version simply contains functionality to encapsulate parameters one by one with the Append (add at the end of the Array) and Prepend (Add at the beginning of the Array) plus the usage of the Encode Function which allows you to append multiple parameters dynamically.

The Full version of the Byterizer allows you to do a lot of bitwise operations to any type, starting from packing multiple booleans into a single byte, to stripping down not used bits from other types.

Example using a Vector3:
Instead of sending over the network 4 bytes per axis which might be unnecessary, you can select how many bits per axis you want the resulting array to contain.
In the case of a Hockey rink, for example, you might only need so many units for the width, some other number for the Depth and a much smaller number for the height.
Having the result be potentially a fraction of the original 12 bytes.

We will be also adding functionality to automatically encapsulate full information for Transforms and RigidBodies, which the user will be able to select what to sync in each case:

- Position
- Position + Rotation
- Position + Linear Velocity
- Position + Rotation + Linear Velocity
- Position + Rotation + Linear Velocity + Angular Velocity

On top of this, we will add the functionality for automatically keeping track of delta compression and syncing only needed information out of a pre-populated array of Transforms or RegidBodies.

We will also be adding functionality to sync a quaternion via the ‘Smallest 3’ method,
The ‘Smallest 3’ method allows you to sync over the network with a user-selected amount of bits (recommended: 29 bits ~> 4 bytes) a much smaller representation of a quaternion (16 Bytes), while keeping almost all of the resolution for rotation.