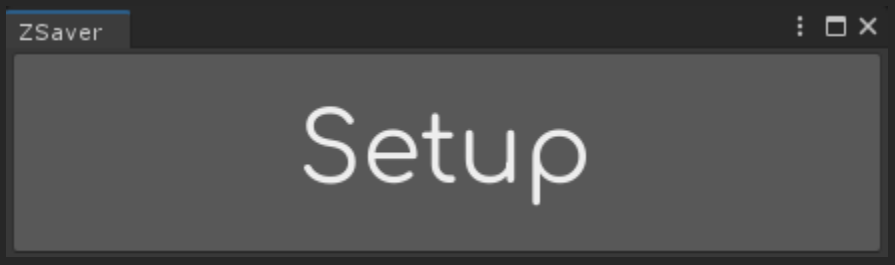
ZSerializer

Quick Start Guide

Contents

Setup

When you first install ZSerializer you will be greeted by this big Setup button. When you click it, it'll generate the necessary files you need to start using the tool.



If it's your first time updating to a version beyond 2.0, you will be prompted to Reset all the project ZUIDs. You are allowed and should do it as long as

Serializing Components

Let's say you have a component you want to serialize, for example, a Game Manager that holds information about the state of the game:

```
public class GameManager : MonoBehaviour
{
    public int highScore;
    public int currentScore;
    public string playerName;
    public Vector3 playerPosition;
}
```

If you make your class inherit from ***PersistentMonoBehaviour*** (you will need to add the ZSerializer namespace for it to be available), it will be recognized as a ***Persistent Component***.

Whenever you go back to Unity, on the ZSerializer Menu, you'll see the name of your class appear on that menu. This means your class is now recognized as Persistent.

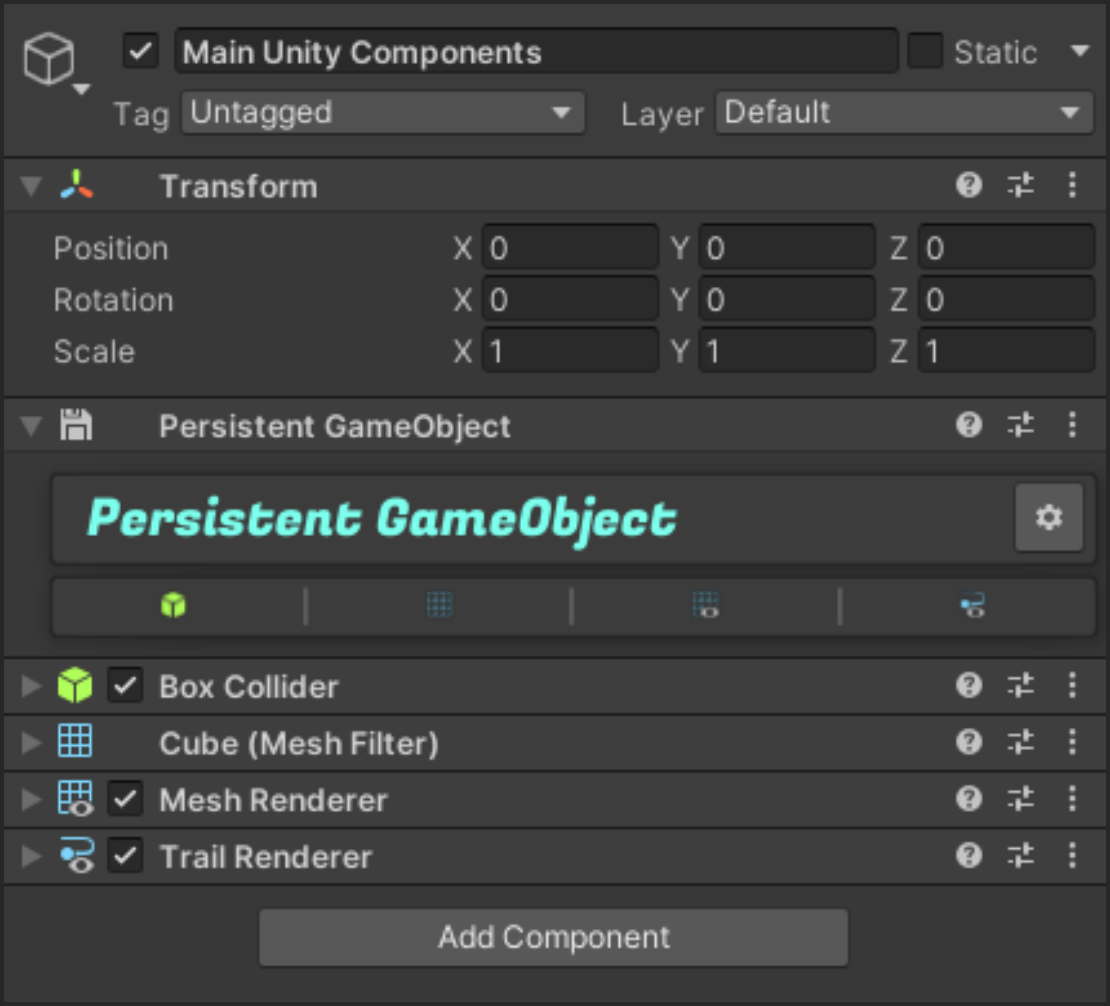
Depending on your settings, you might see either a blue, yellow, or red button to the right of your class.

- Blue means everything's fine, the class instances will be saved and loaded when prompted.
- Yellow means the files needed for serialization have been created, but they need to be rebuilt, because the class itself was changed.
- Red means there's no files and they need to be built.

Clicking the button in either Yellow or Red will fix these problems.

Serializing GameObjects

Making GameObjects serializable is even easier than components, just add the ***Persistent GameObject*** component to your gameObject, and it will be saved and loaded when prompted.



The ***Persistent GameObject*** component will save the GameObject's name, tag, layer, static state, transform, and each and every one of the Unity Components that are attached to it.

(That includes Colliders, Renderers, and everything else under the UnityEngine namespace.)

It will also display which components are to be serialized from within your gameObject.

It will however NOT serialize any user-made Components, such as a Game Manager or a Player Controller. Those will need to be converted into ***Persistent MonoBehaviours*** first

Saving and Loading.

There are several ways to Save and Load data using ZSerializer:

Scenes

Saving and Loading single scenes is great for Single Scene games like hyper-casual mobile games or game jams.

Note: These functions will save the current ACTIVE scene. This means, even if you have multiple scenes loaded at the same time, they will only save or load the active one.

For saving and loading entire scenes, there are two main functions:

```
ZSerialize.SaveScene();
ZSerialize.LoadScene();
```

These two functions can be invoked with a groupID, in order to only affect a specific set of Persistent Objects. (see: Serialization Groups)

SaveScene()

1. It will serialize every instance of a ***Persistent Component and PersistentGameObject*** you have on your scene.
2. It will serialize every instance of a ***Persistent GameObject*** along with its selected Unity Components and ***Persistent MonoBehaviours***.
3. (Optional) It will encrypt your data using the Rijndael Algorithm.
4. It will save your data based on which Save File you've selected and which scene you're currently in when performing the Save.

LoadScene()

1. It will deserialize every ***Persistent GameObject***, and instantiate new ones in case they're not already in the scene, preserving all of the gameObject's data along with their hierarchy.
2. It will deserialize every one of Unity's Components and add them into their respective Game Objects.
3. It will deserialize every one of your custom ***Persistent Components*** and add them into their respective Game Objects
4. It will restore the references and values that each component had at the time of saving.

It will however NOT:

1. Restore a custom ***Persistent MonoBehaviour***'s GameObject if said GameObject doesn't have a ***Persistent GameObject*** Component attached to it.
2. Restore the gameObject's parent in the hierarchy if said gameObject doesn't also have a PersistentGameObject component
3. Restore an instance of a Material, Mesh, or ScriptableObject and similar assets that have been modified from the original.

Levels

The word "Level" in this case stands for a series of GameObjects and components which are children of a given Transform, and are labeled with a name.

This is useful for making games like Geometry Dash™, PolyBridge™ or Besiege™, where the user can save custom made levels that other members of your community can load and explore, or for saving different solutions to a given level.

For saving and loading Levels, there are two main functions:

```
ZSerialize.SaveLevel();
ZSerialize.LoadLevel();
```

There are two obligatory parameters, which are the fileName, and the parent.

FileName

This will be the name of the file containing your level, and the name you'll use to load it.

Parent

This will be the gameObject that holds, as its children, the Persistent Objects that will be saved or loaded.

OnSave and OnLoad Events

If you want to execute some code right before or after a Save or a Load, you can override any of the following methods like this:

```
public override void OnPreSave()
{
    //Your code goes here...
}

public override void OnPostSave()
{
    //Your code goes here...
}
```

```
public override void OnPreLoad()
{
    //Your code goes here...
}

public override void OnPostLoad()
{
    //Your code goes here...
}
```

Serialization Groups

Let's say there's several *kinds* of things you want to save or load at a given time. You may want to save the audio settings of your game, but not necessarily the other 300 gameObjects and components that are present in the scene.

When you click on the cogwheel icon of a Persistent GameObject or Component, you'll see a menu pop up where you can choose between a series of Save Groups.

By default, there are two Save Groups, Main and Settings, but you can add more of them in the Settings tab of the ZSerializer Menu.

The way you Save and Load specific Save Groups is by inputting the groupID into the Save and Load functions as a parameter.

You can use the function ZSerializer.NameToSaveGroup() to get the id by name:

```
void MySaveFunction()
{
    ZSerialize.SaveScene(ZSerialize.NameToSaveGroupID("Settings"));
}

void MyLoadFunction()
{
    ZSerialize.LoadScene(ZSerialize.NameToSaveGroupID("Settings"));
}
```

Whenever you call either of those functions without parameters or with a -1 as the group ID, it'll just save every group.

Scene Groups

Let's say that your game takes place inside a home, and you have a total of 3 scenes, one for the living room, one for the bedroom, and one for the kitchen.

You could Create a Scene Group with all of those scenes, and the last scene you were in will also get saved, along with the data, whenever you perform a normal scene save.

Then, on your loading screen scene, you could call:

```
ZSerialize.LoadSceneGroup("House", LoadSceneMode.Additive);
```

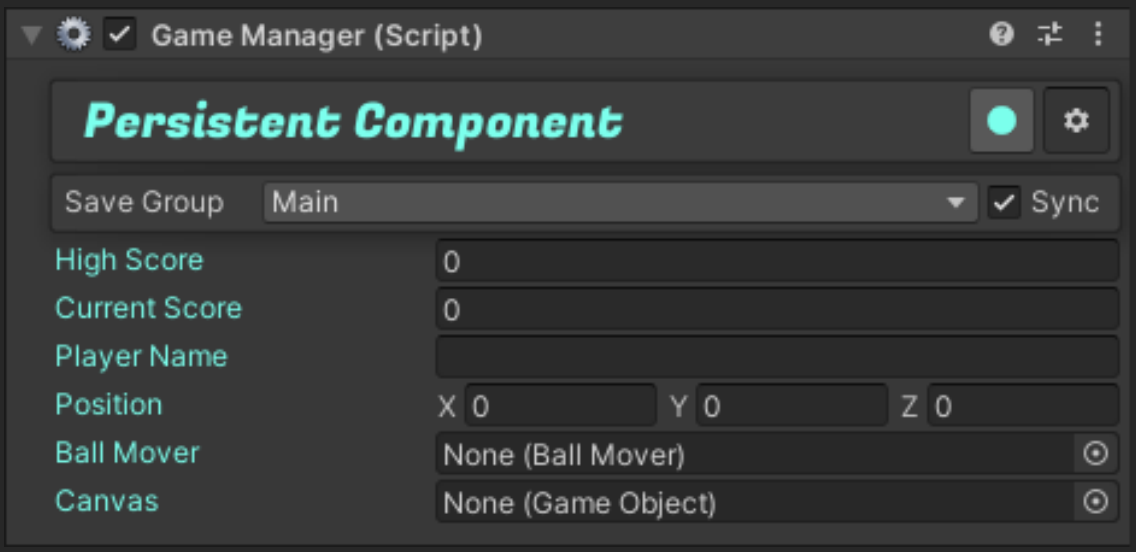
And it would load the last scene that the player performed a Scene Save in.

The way this is intended to be used is in a loading screen, If you wanted to, say, have a loading screen fade in, load the scene, and then fade out, you could do something like this when the loading scene gets loaded:

```
Event function
async void Start()
{
    await CanvasFadeIn();

    await SceneManager.UnloadSceneAsync(0).WaitUntilDone();

    await ZSerialize.LoadSceneGroup("Scene Group 1", out var lastSavedScenePath:string,LoadSceneMode.Additive).WaitUntilDone();

    SceneManager.SetActiveScene(SceneManager.GetSceneByPath(lastSavedScenePath));

    await ZSerialize.LoadScene();

    await CanvasFadeOut();

    SceneManager.UnloadSceneAsync(1);
}
```

Component Settings

Whenever you add a PersistentMonoBehaviour or a PersistentGameObject component to a GameObject, you'll see they have a cogwheel icon to the right. If you click it, you'll see some settings being displayed for them.



Save Group
This lets you choose in which Save Group this component will be saved.

Sync
When this option is activated, when you change a Save Group on one component, it'll also change in the rest of them in that scene, and on the ones you create afterwards.

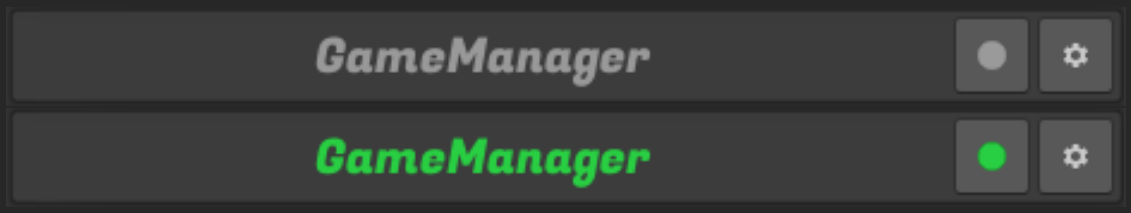
Serialized Fields
Whenever the settings button is active, you'll see your fields turn a certain color. This defines their Serialization State.

When the variable is blue, it'll be saved and loaded when prompted.

If they're gray, however, they'll be ignored. You can learn more about field serialization in Advanced Serialization

On/Off
Whenever you've built your ZSerializers through the Red or Yellow buttons, the Blue button works a bit differently: It toggles the serialization on and off. This means that, even if the component is present in the scene, it'll not be saved nor loaded.

A component being off is signified by the label and button becoming gray.



Global Switches
Much like how you can change the configuration of Persistent Components one by one, you can also do so globally in the ZSerializer Menu.

You can just change a setting for any given Persistent Type, and it'll update it on every instance of said component in the scene, provided they have their Sync enabled.

IsLoading / IsSaving

These two variables will be true while there's an active process that involves the PersistentMonoBehaviour it's in. For instance, if you're currently saving a persistent GameManager, IsSaving will be "true" while this process is happening.

Advanced Serialization.

On Persistent Components

Let's say that you have a script like this:

```
public class WaveManager : PersistentMonoBehaviour
{
    public int wave;
    public float waveTimer;
    public BigBulkyDataObject[] bigObjects;
}
```

Where BigBulkyDataObject is a gigantic class that holds enormous amount of information that gets processed when the game starts, or depends on the state of the game.

Whatever the case, you may not want to serialize this, as the save files could get really bulky really quick.

If you wish not to serialize a specific field inside a PersistentMonoBehaviour, you can do so by writing [NonZSerialized] just before the declaration, and that field will no longer be serialized. And it will now appear gray in your Settings Menu.

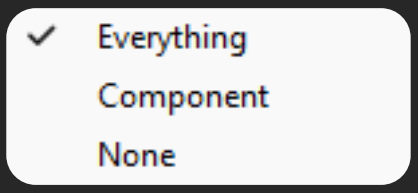
```
public class WaveManager : PersistentMonoBehaviour
{
    public int wave;
    public float waveTimer;
    [NonZSerialized] public BigBulkyDataObject[] bigObjects;
}
```

On Persistent GameObjects.

By enabling Advanced Serialization on your ZSerializer Menu's Settings, you'll get some additional settings when displaying the settings for PersistentGameObjects:

There will now be a whole new menu enumerating every Unity Component attached to the GameObject, along with their ZUID (provided you're in Debug Mode).

To the right, you'll be able to select the type of Serialization that specific component will have:



Everything
This will make it so the entire component is saved, along with its fields.

Component
This will load a default version of a component when loading a destroyed gameObject.

None
This will make it so the component is neither saved nor loaded when prompted. This is useful for temporary components.

Adding Components through code.
Whenever you add a component through code, it'll by default not be added to the serialization list of the Persistent GameObject component and it will thus, not be saved nor loaded when prompted.

To add a component with a specific Serialization type, you'll need to use the special AddComponent methods that are present within the Persistent GameObject Component:

```
public void MyAddComponentFunction()
{
    persistentGameObject.AddComponent<BoxCollider>(PersistentType.Component);
}
```

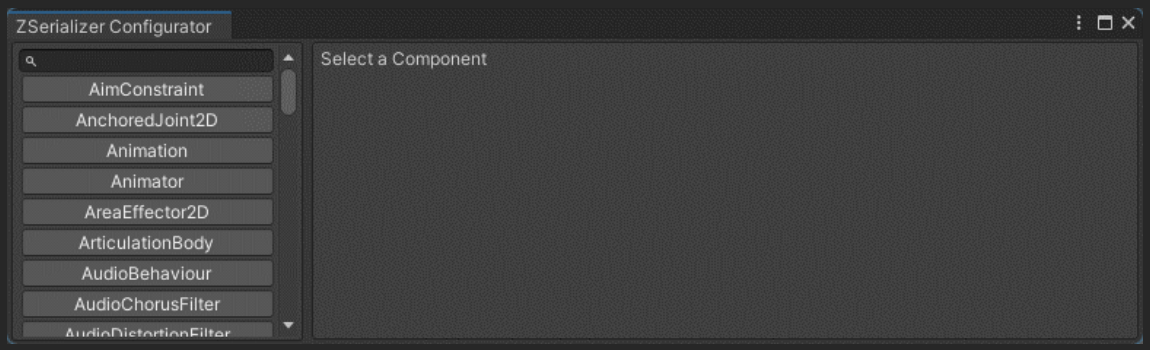
On Unity Components

There are some unity components like the Mesh or TrailRenderer which have a lot of properties that many people usually leave untouched.

For the most part, the only thing we care about serializing in a MeshRenderer, if at all, is the materials it has.

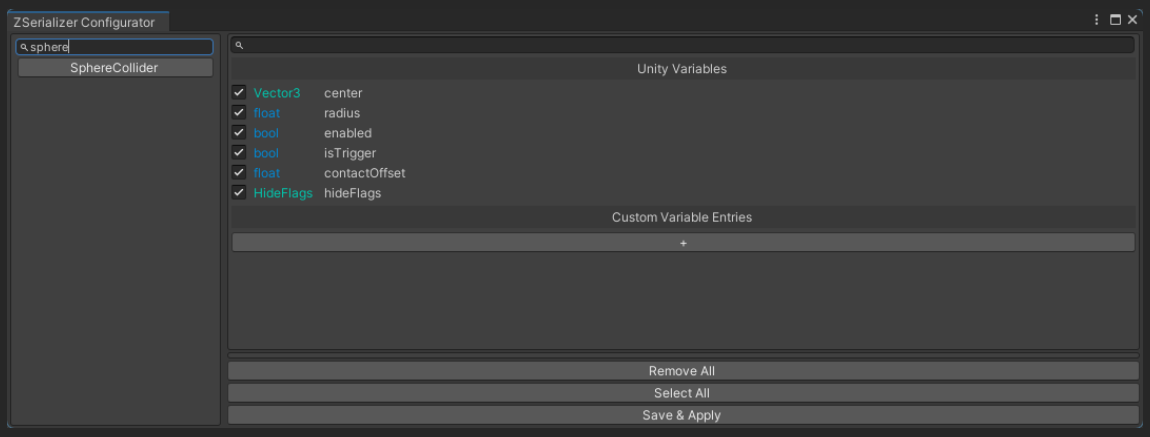
This is when the ZSerializer Configurator comes in handy.

If you go to Tools/ZSerializer/ZSerializer Configurator, a new window will open up containing two separate boxes:



To the left, you'll be able to select any Unity Component you want to configure.

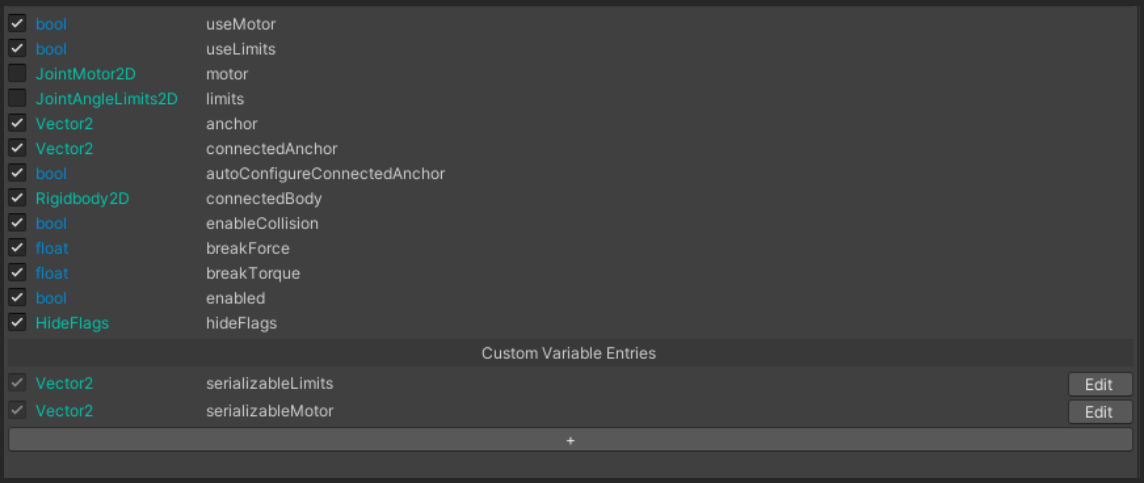
When you select your desired one, a list of fields will appear on your right:



In this case, this is the SphereCollider ZSerializer. You can toggle off any fields you don't want to serialize. If I wanted to save only the center and radius, I'd click "Remove All" and toggle those ones on.

Custom Variable Entries

You can also create Custom Variable Entries, which are special variables that will be added to the ZSerializer to serialize extra data.

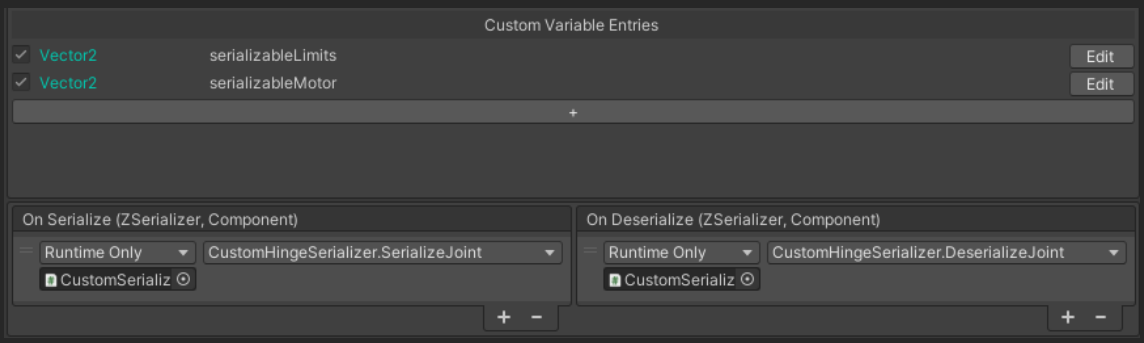


The HingeJoint2D component has two parameters which are not serializable by default with JsonUtility, so I created my own two variables that store the same data in Vector2s, while I disabled the problematic variables.

OnSerialize and OnDeserialize

When you modify the configuration of any given component, two events will appear in your screen: OnSerialize and OnDeserialize.

These both happen at the end of Saving and Loading said component respectively, and they're meant to be public static method that take in a ZSerializer and a Component as a parameter.



In this example, I created two of those methods in a MonoBehaviour called CustomHingeSerializer, and stored it in a prefab to be able to access the function globally, since UnityEvents don't let you call Static functions without a GameObject instance.

When you hit "Save & Apply", your UnityComponentZSerializers.cs file will be rebuilt to reflect these changes, and any unchecked fields will no longer be serialized.

Asynchronous Serialization

There may be some instances, mainly for very large scenes, where saving and loading will take multiple frames, and there will be visible lagspikes happening throughout your gameplay. This is fine if saving is manual, but if it's automatic, it'll most likely end up hindering the player's experience.

This is when Asynchronous Serialization comes into place. You can make the Serialization run "parallel" to the gameplay, so your player will still be able to move around the scene without any hiccups, and your loading screen will not freeze when trying to load a gigantic scene.

The way to activate this feature is through the ZSerializer Menu. When Serialization Type is set to "Async", any Serialization process (Saving or Loading) will be done parallelly.

This, however, comes with some caveats. Because the process is now split over multiple frames, there may be some Unity functions, like Start() or Update() that will start running without you wanting to.

This is where the PersistentMonobehaviour's event functions and properties come into play.

```
public override void OnPostLoad()
{
    ballMover = FindObjectOfType<BallMover>();
}

private void Start()
{
    ballMover = FindObjectOfType<BallMover>();
}
```

Any functionality that was done during Start() should probably also be placed inside OnPostLoad(), since this is the method that will synchronously when the loading is finished.

Similarly, we may not want out update function to run while the component is being processed, so we'll use the "isLoading" and "is Saving" properties like:

```
private void Update()
{
    if(!isLoading) // Do stuff...
}
```

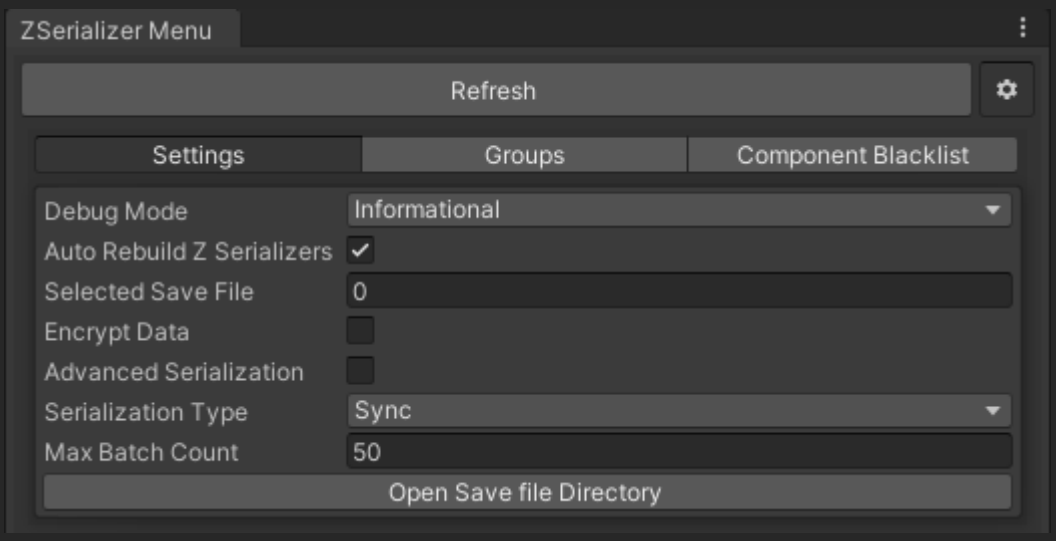
The ZSerializer Menu

This menu contains information about your current scene's ***Persistent Components*** as well as some settings.

The first panel will show you which classes have been marked as ***Persistent*** in your current Scene.

When you press on the cogwheel to the right, you'll see the Settings menu:

Settings menu



This tab will show you some settings to do with how the tool works and how serialization is managed in your project.

Debug Mode

DebugMode has three modes, Off, Informational, and Developer, each of which will show you incrementally more information about your project's state, in both logging, and the UI. This is useful when encountering potencial errors, to help you debug what the cause of them could be.

Auto Rebuild ZSerializers

This will rebuild the ZSerializers of your Persistent Components, so that you don't have to click that filthy yellow or red button ever again.

Selected Save File

The selected Save File in which the saving and loading will be done. There can technically be any amount of save files you want, although it may take a toll on disk space on very large games if it's not used correctly, so limiting it is a good idea.

Encrypt Data

Determines whether or not your data will be encrypted using the Rijndael Algorithm.

Advanced Serialization

This will toggle on or off the use of Advanced Serialization.

Serialization Type

Determines whether the Serialization will be done Synchronously or Asynchronously.

- Sync is more stable, but may produce lagspikes and other artifacts when loading and saving heavy scenes
- Async makes it so the serialization is chunked and runs parallel to Unity's Main thread. This means it may take longer, but there won't be any lagspikes, provided the next two options are set up correctly.

Max Batch Count

This will dictate the maximum number of components that will be serialized per component type per frame. If you set the number to 1, it'll take a frame to process one component, and so on. The larger this number gets, the more likely it is to cause lagspikes, but it will in turn be quicker, and vice versa.

Open Save File Directory

When you click this button, the folder in which save files will be created will open, and you will be able to explore it.

The file structure of this folder will be the following:

<SelectedSaveFile>/<CurrentSceneBuild>/<SaveGroup>/...

Groups Menu

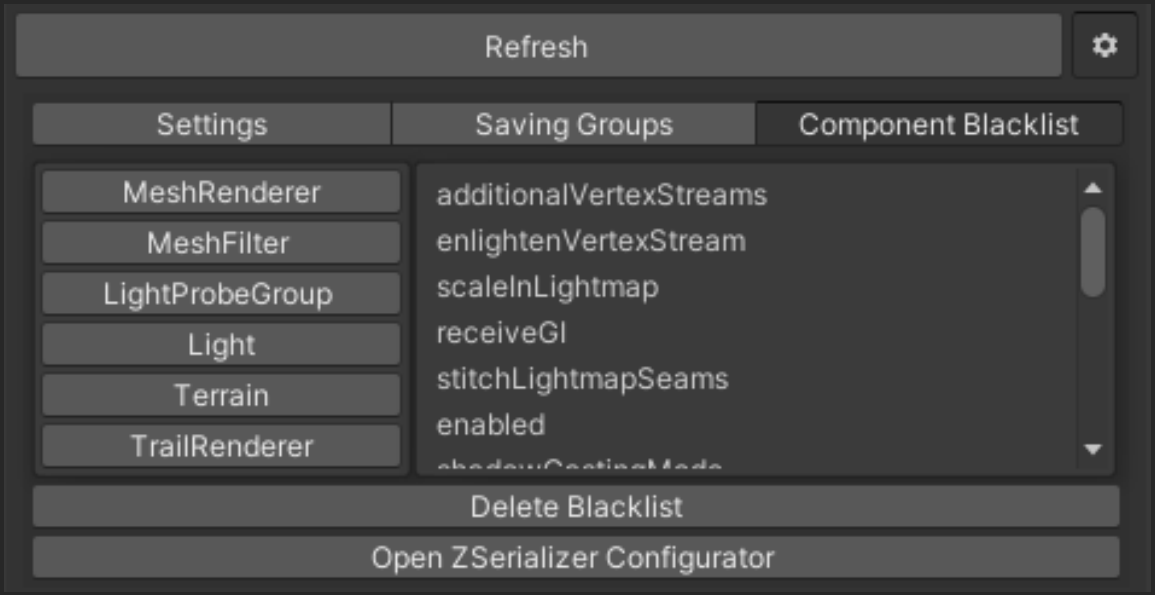
Serialization Groups

This menu will show you your current project's Serialization Groups. You can have up to 16 of these, and Main and Settings come predefined.

Scene Groups

This menu will show you your current project's Scene Groups.

Component Blacklist



This menu will show you your current project's Component Blacklist. This is a List defined for every Unity Component about which fields should be serialized or not. You can learn more about it here.