V1.0

Created by Maikel 'MajinMHT' Tjin

Table of Contents

This manual will explain and describe the features and important things to know about UniSave. UniSave is easy to use, and works out of the box with most of Unity's built-in components. However, if you want to become a power user, and extend UniSave's support by adding you own custom components, some basic coding is involved. Fortunately, this is relatively easy, and only consists of creating container files. UniSave comes with templates and examples on how to do this.

All the code has been written in C#.

The UniSave package includes:

UniSave source files
Component serialization files
Data Type serialization files
Loading Screen Scene
UniSave demonstration (Build + Source)
Checkpoint prefab
Templates
Protobuf-net.dll, designed and written by Marc Gravell

Overview

What is UniSave?

UniSave is a component based save game system for games made with Unity. It allows you to save any progress or changes made in your game to a file, and load the data back in when needed. UniSave uses a .NET implementation of Google's serialization format called Protocol Buffers, designed and written by Marc Gravell.

Unity 3.5.x is required. (*Check page 14 for more info on component support*)

The supported platforms are:

- Windows
- Mac
- Android
- iOS

The source code is included. You are allowed to change any of the code. However, this is only recommended if you know what you are doing.

How does it work?

It's very easy. When you want to save specific data, you have to tell UniSave which components it has to save on a per game object basis. UniSave comes with a special component for this, called the *State* component (page 4), which you add to the game objects that should be saved. The *State* component then gives you a list of all the supported components on the game object, and you can select the ones that should be included when you initiate the saving process.

How is a component supported?

The way UniSave works, for every type of component it needs a separate file for serialization. You just have to provide UniSave with this serialization file (which is easy to create), tell it that it exists, and then UniSave supports the component. Because the source code is included, you can even edit the serialization files for the built-in components. This comes in handy when a new Unity update is released and adds new properties to a component.

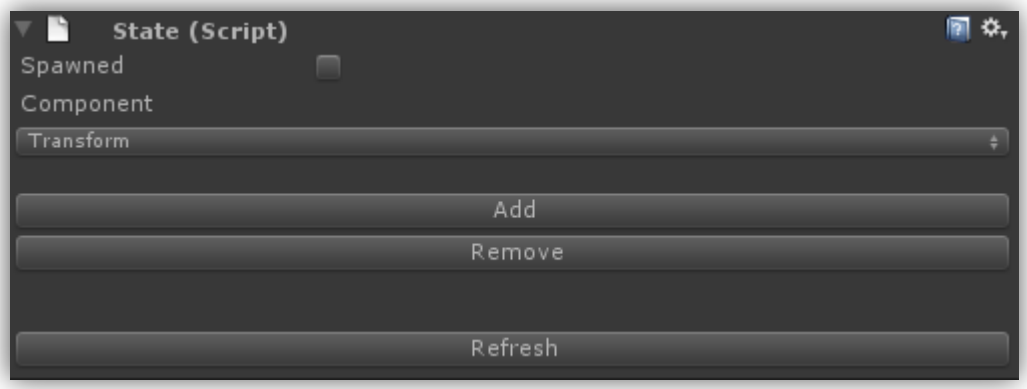
How do I save?

When you want to save to a file, all you have to do is execute one line of code, which is **UniSave.Save(saveName)**, with **saveName** being the name of the actual save file.

State Component

What is the *State* component?

The *State* component is your friend when it comes to telling UniSave which components you want it to save. You can add this component by going to *Component -> UniSave -> State*. When you add the component to a game object, it will show you a drop-down list of all the supported components on the corresponding game object. A component is supported when it has a serialization file and is added to the supported component list. (*Page 8*)



The initial state of a State component when added to a game object.

When you add the *State* component to a game object, it will automatically mark at least one component for saving, which is always the *Transform* component. You can then change this, or mark more components by using the *Add* button. The *Remove* button deletes a list from bottom to top until one list remains.

At some point you may want to add new components to your game object, and save these with UniSave. If you have added the *State* component before doing this, the new component will not appear in the drop-down list. That's where the *Refresh* button comes in, which will make the *State* component recheck the game object for supported components.

UniSave makes a distinction between game objects that are placed in the scene at design time (*Default Game Objects*) and game objects that are instantiated during runtime (*Runtime Game Objects*). It does this because these two types of objects are being treated differently behind the scenes.

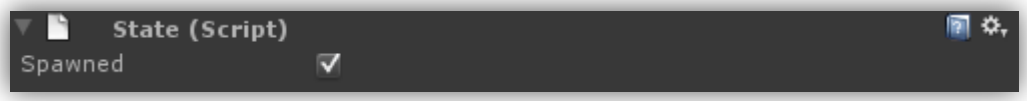
Default Game Objects

When creating a game object at design time, it's a default object that's part of the scene. This makes it possible for the user to only save the components that you want UniSave to save. (As shown in the previous image) This reduces the save file size.

For example, consider you have a *Light* in your scene that's there from the start. This *Light* is basically a game object that has *Light* component attached to it. It never moves, but it should change color during the game. When you save the game, you want it to remember the current color, so that when you load the data back in, the *Light* should have the appropriate color. Since this information is stored in the *Light* component, it's the only component that you have to save with the *State* component. Saving the *Transform* component would not make much sense if the *Light* doesn't change any of its *Transform* properties, or if they aren't supposed to be saved.

Runtime Game Objects

Saving game objects that were instantiated at runtime works differently. When you want to save an instantiated prefab, this prefab should, of course, have a *State* component on it. The only difference is that when it is supposed to be instantiated at runtime, it should have the *Spawned* checkbox ticked. As you can see in the image below, when this checkbox is ticked, you cannot select individual components to save. This is because when saving game objects that were not in the scene at the start, UniSave is not able to update an existing game object, because it doesn't exist. This means it has to save the complete game object so it can create it when loading a save file. This increases the save file size. Also, when a runtime game object is in a parent/child relationship, every child in the hierarchy requires a *State* Component as well, or else that child will not be saved.



Destroyed Game Objects

Besides saving component data so that a specific state can be loaded back in, UniSave is also able to remember if a game object was destroyed in-game. If, for example, you have a boss in your scene that the player is able to kill, you may want to destroy that boss game object when it has been slain. If the boss has a *State* component attached to it, it will detect if the game object has been destroyed, and will save this information to the save file as well. When loading the save file, it will automatically destroy the game object.

Saving & Loading

Saving

As mentioned earlier, when you want to save to a file, you only have to execute one line of code, which is **UniSave.Save(SaveName)**, with **saveName** being the actual name of the save file. When you call this method, UniSave will look for every *State* component in your scene, and execute a method on each of them. If, for some reason, you want to check if UniSave is currently saving, you can check the bool property **UniSave.IsSaving**. This might be useful when you want to display something on the screen or do something else while the game is being saved.

Note: these are save file locations for Windows and Mac OS X.

Windows: *<Game Folder>/Saves*
Mac OS X: */Users/MacAdmin/Library/Caches/<Product Name>/Saves*

The name of the folder (*Saves*) can be modified in the Unisave.cs file.

When building a game for Mac OS X, make sure that the API Compatibility Level is set to NET 2.0 in the Player Settings. If you use NET 2.0 Subset, the file path of the save file location is incorrectly identified as the Documents folder.

Save File Info

When you save game data with UniSave, it creates two files. One is the actual game data, which by default is a .dat file (This can be changed in the *UniSave.cs* file), and the other one is called a Save File Info file, and has a .info extension by default . The info file holds information about the game data file. Information like the name of the level, and on what date and time the save took place. This comes in handy when you are creating something like a load/save menu in your game, and you want to retrieve this kind of information when the user select a specific file. You can get this information by calling **UniSave.GetSaves()**, which returns an array of *SaveFileInfo*. Each *SaveFileInfo* represents a save file, and has properties containing information about this save file.

Loading

Loading a file is just as easy as saving. Instead of the Save method, you have to call **UniSave.Load(saveName)**. Unisave comes with a scene called *Loading*. This scene has a camera and a game object with a custom loading script and a *GUITexture* attached to it (which displays the loading screen texture). It functions as a loading screen that UniSave will first load before loading the actual level and its save file.

Extending Component Support

Adding support for new a component involves some typing, but is relatively easy to accomplish. For every component, UniSave needs a separate file that will act as a container and hold the properties that are supposed to be saved. For example, the Light component has a LightSerializer class that holds this information when saving to a file. This is the **serialization file**. UniSave has support for 53 components out of the box, which means that the serialization file for each of these components is included, so you don't have to create one yourself.

Serialization is the process of turning runtime objects into text or binary data that can be transferred and stored, and then read back into a runtime object at a later time.

The reason a serialization file is needed, is because these components themself are not serializable. If the source code of these components was available to the end-user, you would be able to make them serializable just by placing an attribute at the top of the class. Unfortunately, this is not possible, and that's where UniSave steps in.

Components can have fields with all sorts of data types, like *string* or *vector3*, but the custom data types need a serialization file as well. Types like *vector3*, *quaternion*, and *color* are custom data types that aren't serializable. UniSave has serialization files for these common Unity data types included. Built-in types like *int*, string, and *bool* are serializable automatically.

Every component and data type's serialization file has the same layout. UniSave comes with a template you can use that saves you some typing, and makes it easier to only fill in the blanks.

Adding New Component

Adding support for a new component is easy. The first thing you have to do is make sure that the data types in your component are serializable. Let's go through an example.

This is a custom component called *ZooCage*, which you can add to a game object.

```
using UnityEngine;

public class ZooCage : MonoBehaviour
{
    public string CageName;
    public Animal CagedAnimal;

    public void Awake()
    {
        CageName = "Default Cage Name";
    }

    public void OpenCage()
    {
        // Code here
    }
}
```

As you can see, *ZooCage* has two fields. One is a string, and one is an *Animal*, which is a custom class.

```
using UnityEngine;

public class Animal
{
    public string Name;
    public int Age;
    public Color NoseColor;

    public Animal(string name, int age, Color noseColor)
    {
        Name = name;
        Age = age;
        NoseColor = noseColor;
    }

    public void Run()
    {
        // Code here
    }
}
```

To add support for the *ZooCage* component, you have to look at the two variables at the top of the *ZooCage* class (*CageName* and *CageAnimal*). You want to save the information that they hold, and in order to do so, they should be serializable. *String* is a built-in C# data type, which makes it serializable automatically. *Animal*, however, is not, because it is a custom class. To make the *Animal* class serializable, you have to write a serialization file for it, which will hold the data for saving purposes.

```
using UnityEngine;
using ProtoBuf;

[ProtoContract]
public class AnimalSerializer
{
    [ProtoMember(1)] public string Name;
    [ProtoMember(2)] public int Age;
    [ProtoMember(3)] public ColorSerializer NoseColor;

    // This gets called when casting Animal to AnimalSerializer
    public AnimalSerializer(Animal data)
    {
        Name = data.Name;
        Age = data.Age;
        // Explicitly cast Color to ColorSerializer
        NoseColor = (ColorSerializer) data.NoseColor;
    }

    // Converts AnimalSerializer to Animal
    public static explicit operator Animal(AnimalSerializer data)
    {
        // Notice how data.noseColor, which is a ColorSerializer, gets cast back to Color
        return new Animal(data.Name, data.Age, (Color) data.NoseColor);
    }

    // Converts Animal to AnimalSerializer
    public static explicit operator AnimalSerializer(Animal data)
    {
        return new AnimalSerializer(data);
    }

    // Empty constructor required for ProtoBuf.
    public AnimalSerializer()
    {
    }
}
```

The first thing you might notice is the **[ProtoContract]** attribute just above the class declaration, and the **[ProtoMember]** attributes for the fields. These attributes are necessary, because UniSave uses the ProtoBuf format for serialization. The number you attach to each **[ProtoMember]** attribute doesn't matter, as long as it is unique in the file. Remember to add the **using ProtoBuf;** directive at the top of the file, or else you won't have access to these attributes.

AnimalSerializer does three things:

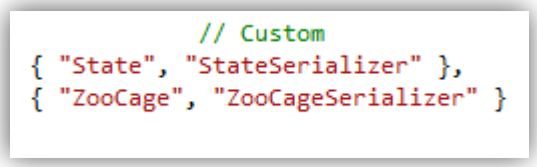
1. Hold the data for the *Animal* class.
2. Handles converting *AnimalSerializer* to *Animal*
3. Handles converting *Animal* to *AnimalSerializer*

These conversions take place when you're explicitly casting *Animal* to *AnimalSerializer* and vice versa.

After you have made sure that every field is serializable, it's time to create the serialization file for the component itself.

```
using UnityEngine;
using ProtoBuf;

[ProtoContract]
public sealed class ZooCageSerializer
{
    [ProtoMember(1)] public string CageName;
    // Notice how the AnimalSerializer class is being used instead of Animal.
    [ProtoMember(2)] public AnimalSerializer CagedAnimal;

    // This will be called when UniSave is saving data to a file.
    public ZooCageSerializer(GameObject gameObject)
    {
        // Retrieve the component from the game object.
        var zooCage = gameObject.GetComponent<ZooCage>();

        // Add the component's data to this class.
        CageName = zooCage.CageName;
        // Explicitly convert Animal to AnimalSerializer.
        CagedAnimal = (AnimalSerializer) zooCage.CagedAnimal;
    }

    // This will be called when UniSave is loading data back in.
    public ZooCageSerializer(GameObject gameObject, ZooCageSerializer component)
    {
        // Retrieve the component from the game object or add it.
        var zooCage = gameObject.GetComponent<ZooCage>();

        if (zooCage == null)
            zooCage = gameObject.AddComponent<ZooCage>();

        // Add data back to the component.
        zooCage.CageName = component.CageName;
        // Explicitly convert AnimalSerializer to Animal.
        zooCage.CagedAnimal = (Animal) component.CagedAnimal;
    }

    // Empty constructor required for ProtoBuf.
    public ZooCageSerializer()
    {
    }
}
```

With both the *AnimalSerializer* and *ZooCageSerializer* created, the *ZooCage* component is now serializable. There's one thing left to do in order to tell UniSave that this component is ready to be supported, and this is adding the name of the component to the *SupportedComponents* dictionary in the *SupportedComponents.cs* file, which is located in the *UniSave* folder.



The *ZooCage* component is now supported by UniSave.

One interesting thing to note there is that you actually don't always have to create a serialization file for a custom data type. In our *Animal* class example, the only reason we made a serialization file for this class was because of the *NoseColor* field. Like previously mentioned, built-in data types (like *string* and *int*) are serializable automatically, but custom data types like *Color* are not. Color is a data type made by Unity, and you don't have access to its source file, which means you cannot make it serializable. UniSave includes the serialization file for *Color*, among other data types. Because of this, the *Animal* class itself needs a serialization file so it can cast *Color* to *ColorSerializer*, and vice versa. Despite not needing a serialization file for your custom data type in some cases, it might be handy to do so anyway, in case you might need one in the future.

You might be wondering why you can't just use *ColorSerializer* directly in your class. In fact, you can, but the problem is that it lacks any functionality whatsoever. These serialization files for built-in Unity data types are only meant for storage, and lack the methods you can call on them. The underlying implementation of these methods are hidden, thus they are unknown.

Because the source code is included with Unity, you can also change the serialization files of components that are supported out of the box. This comes in handy when Unity gets updates and adds new features to existing components. You won't have to rely on official updates to UniSave, and will be able to update support yourself.

Saving Asset References

When a component has a reference to one or more assets (like *material*, *font*, *animation*, *prefab*, *texture*, etc.), you have to save the actual name of the asset as opposed to serializating the asset itself. If you know the name of an asset, you can load it at runtime. This does mean that you have to place the assets in a folder called *Resources*, in your *Assets* folder.

(*More on the resources folder can be read here*: http://docs.unity3d.com/Documentation/ScriptReference/Resources.html)

For example, the built-in *TextMesh* component has a field named font that is of data type *Font*, and as the name implies, holds a reference to the actual font asset for that component. In the serialization file for this component, instead of saving the actual font, the name of the font gets saved.

Here is the property that holds his name.

```
[ProtoMember(10)] public string FontName { get; set; }
```

In the constructor that gets called when saving, the name of the *Font* asset is being saved to the serialization file like this.

```
if (textMesh.font != null)
    FontName = textMesh.font.name;
```

In in the constructor that gets called when loading, the *Font* is being loaded back in by using its name that was saved.

```
if (!String.IsNullOrEmpty(component.FontName))
    textMesh.font = (Font) UniSave.TryLoadResource(component.FontName);
```

UniSave.TryLoadResource() is basically just a wrapper around Unity's built-in **Resources.Load()**, which must be used to avoid erroneous behaviour. It adds some extra functionality, like loading default built-in assets when no asset name is saved, and giving you an error message if an asset cannot be found in the *Resources* folder.

Supported Components List

Here is the complete list of the current built-in Unity components that work out of the box. Currently 53 are supported, 10 are partially supported, and 6 are not supported. When a component is partially supported, it means that one or more of its properties cannot be saved. The reason some of these components are not or partially supported is due to accessibility limitations of Unity.

Note: The properties that are not supported on some components are displayed in red.

- ✓ Transform
- ✓ Terrain
  - *Draw*
  - *Wind Settings: Speed/Size/Bending/Grass Tint*
- ✓ SkinnedMeshRenderer
  - *Bones*

Mesh

- ✓ MeshFilter
- ✓ TextMesh
- ✓ MeshRenderer

Effects

- ✓ ParticleSystem
  - *Modules*
  - *Missing Properties*
- ✓ TrailRenderer
- ✓ LineRenderer
  - *Positions*
  - *Parameters*
- ✓ LensFlare
  - *Ignore Layers*
  - *Directional*
- x Halo
- ✓ Projector

Legacy

- x EllipsoidParticleEmitter
- x MeshParticleEmitter
- ✓ ParticleAnimator
- x WorldParticleCollider
- ✓ ParticleRenderer

Physics

- ✓ Rigidbody
- ✓ CharacterController
  - *Skin Width*
  - *Min Move Distance*
- ✓ BoxCollider
- ✓ SphereCollider
- ✓ CapsuleCollider
- ✓ MeshCollider
- ✓ WheelCollider
- ✓ TerrainCollider
- ✓ InteractiveCloth
- ✓ SkinnedCloth

- ✓ ClothRenderer
- ✓ HingeJoint
- ✓ FixedJoint
- ✓ SpringJoint
- ✓ CharacterJoint
- ✓ ConfigurableJoint
- ✓ ConstantForce

Navigation

- ✓ NavMeshAgent
- ✓ OffMeshLink
  - *Start*
  - *End*
  - *Bi Directional*

Audio

- ✓ AudioListener
- ✓ AudioSource
- ✓ AudioReverbZone
- ✓ AudioLowPassFilter
- ✓ AudioHighPassFilter
- ✓ AudioEchoFilter
- ✓ AudioDistortionFilter
- ✓ AudioReverbFilter
- ✓ AudioChorusFilter

Rendering

- ✓ Camera
- ✓ Skybox
- ✓ FlareLayer
- ✓ GUILayer
- ✓ Light
  - *Cookie Size (Directional Light)*
- ✓ LightProbeGroup
- ✓ OcclusionArea
  - *Is View Volume*
  - *Is Target Volume*
  - *Target Resolution*
- ✓ OcclusionPortal
  - *Center*
  - *Size*
- ✓ LODGroup
- ✓ GUITexture
- ✓ GUIText
  - *Left Border*
  - *Right Border*
  - *Top Border*
  - *Bottom Border*

Miscellaneous

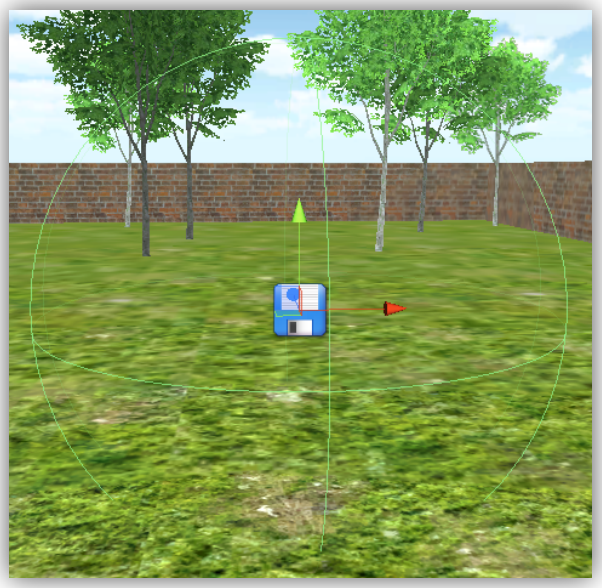
- ✓ Animation
- x NetworkView
- x WindZone

Appendix: Extra Features

This appendix shows a couple of minor extras that are nice to know about, and can be used.

Checkpoint

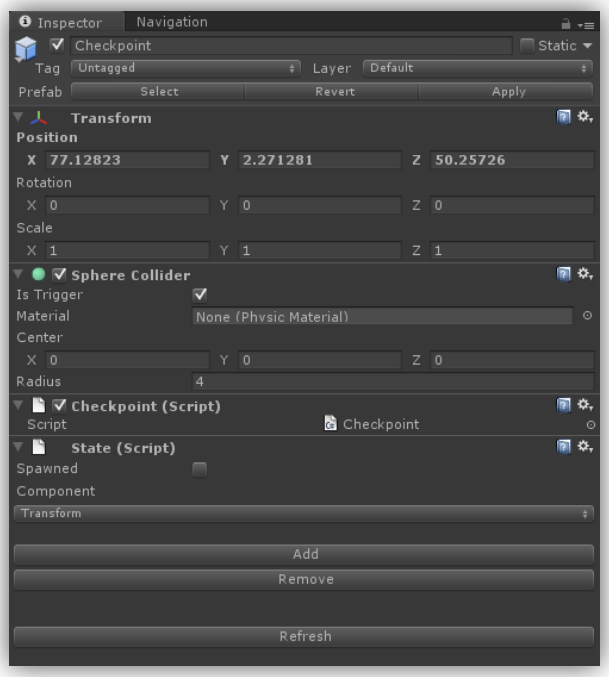
UniSave includes a special *Checkpoint* prefab. This is a basic prefab that includes a script that will save the game automatically when the player passes the checkpoint. It works like most checkpoint/autosave features that you can find in games these days.



The *Checkpoint* prefab uses a custom icon in the scene view, which represents a floppy disk.

When the player enters the radius of the *Checkpoint*, it will disable the game object and save game data to a file called Autosave. This name can be changed in the *Checkpoint.cs* file.

To change the radius of the *Checkpoint*, you have to change the radius on the *SphereCollider* component.



Because the game object itself gets destroyed when triggered, and has to stay destroyed, it would of course be nice to save this information. The State component takes care of this.

Callbacks

When you want to be notified and perform a certain action when UniSave has failed or succeeded at loading/saving a file, there are a couple of events you can subscribe to.

UniSave.OnSavingFailed
UniSave.OnSavingCompleted

UniSave.OnLoadingFailed
UniSave.OnLoadingCompleted

Logging

UniSave has a public field called **EnableLogging** which can be set to true, to turn on UniSave's logging. This will print information to the console relevant to loading and saving when performing these actions. This includes the amount of time it took to load or save.