Save System for UFPS

User Guide

v1.1

Pixel Crushers

Save System for UFPS

Contents

Chapter 1: Welcome to the Save System for UFPS

Thanks for using the Save System for UFPS! This asset adds single player save support to Opsive's UFPS version 1.x to save and load games and save changes across scenes.

This asset requires Unity 5.6.0+ and Opsive's UFPS 1.73+.

If you're using Opsive's newer Character Controllers, use the Save System for Opsive Character Controllers instead of this asset.

How to Get Help

We're here to help! If you get stuck, have questions, or want to request a feature, please email us:

Email: support@pixelcrushers.com

Demo

To play the demo, add these scenes to your build settings:

- Pixel Crushers ▸ Save System for UFPS ▸ Demo ▸ DemoScene1
- Pixel Crushers ▸ Save System for UFPS ▸ Demo ▸ DemoScene2

Then play DemoScene1. You can do the following:

- Press Escape to open the demo menu containing buttons to save and load your game. (If you want to add the demo menu to your own scene, add the DemoMenu script to it.)
- Use the portals to move between DemoScene1 and DemoScene2.
- Next to the portal in DemoScene1, there's a Save Checkpoint trigger. If you enter this, it will automatically save your game.

The crate in DemoScene1 will remember its position and if it's been destroyed. Similarly, a few pickups have been configured to remember in saved games when picked up. The attack turret in DemoScene2 will remember if it's been destroyed. DemoScene2 also has step switches that turn a light on and off; the light remember its state.

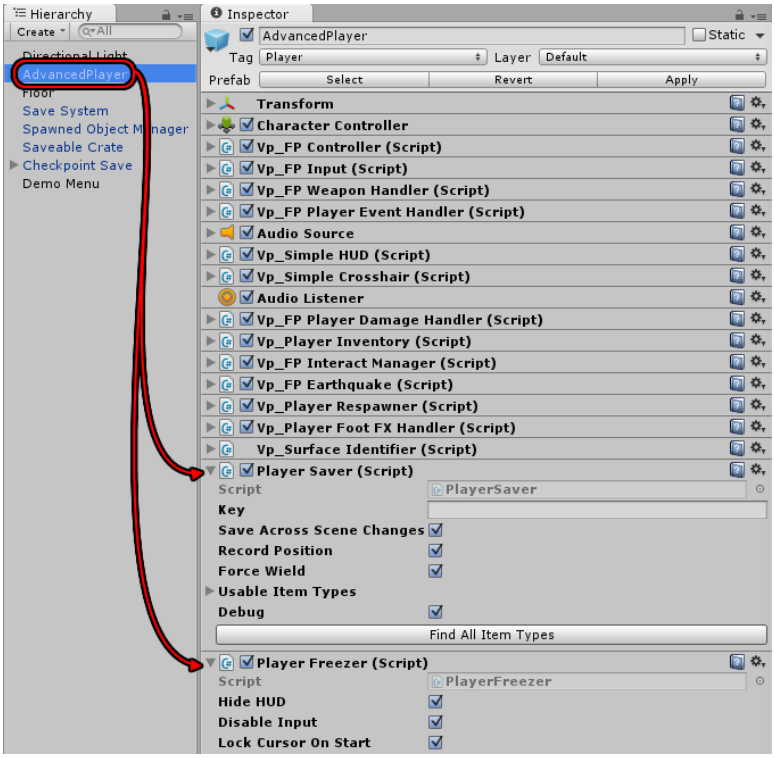
The rest of this manual describes how to set up your scenes and objects for saved games.

Chapter 2: Configuration

Follow the steps in this chapter to set up your project with the Save System for UFPS. You can apply the Save System for UFPS to prefabs as well as scene GameObjects.

Player

Add a **Player Saver** component to your player.



This will save the player's stats, inventory, and position.

Data in saved games are stored under keys. You can specify a value in the **Key** field, or just leave it blank to default to the GameObject's name (e.g., AdvancedPlayer). Make sure **Save Across Scene Changes** is ticked.

Assign any item types that the player can pick up during the game to the **Usable Item Types** list, or click **Find All Item Types** to automatically add all item types in the project. At runtime, items in the player's vp_Inventory Item Caps will automatically be added to this list, so you don't need to add them.

To log debug information to the Console, tick **Debug**.

Optionally add a **Player Freezer** component. This component allows you to freeze the player. The demo menu uses it to freeze the player while the menu is visible. It also has an option to lock the cursor when the scene starts so the player doesn't have to click in the window to lock the cursor for mouselook.

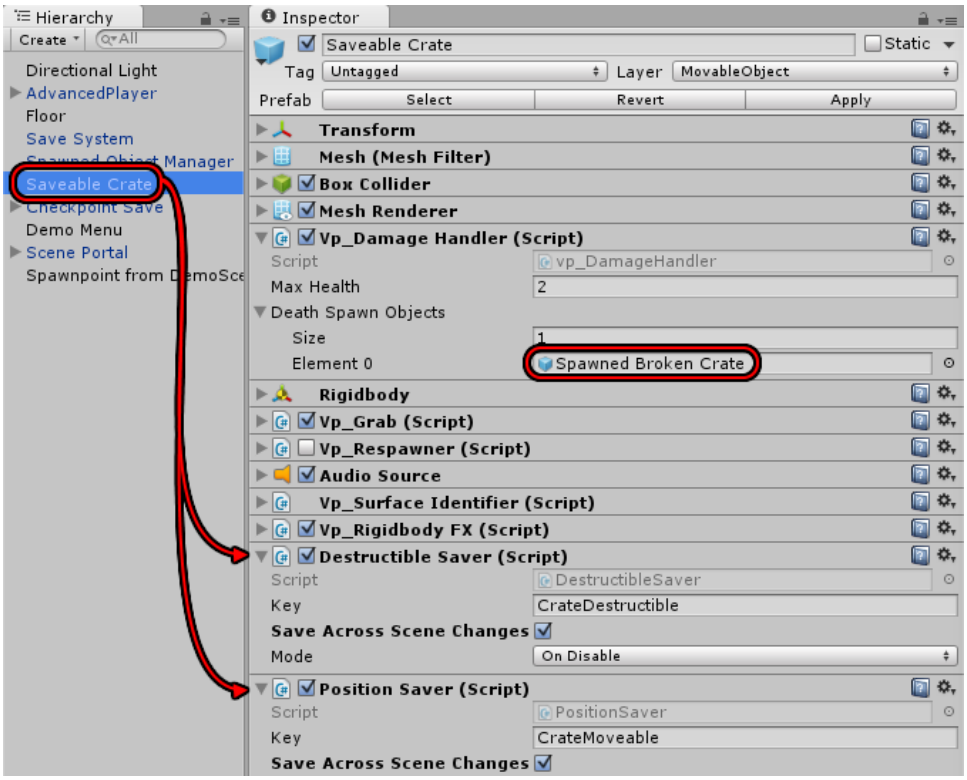
Pickups and Destructibles

To save the state of a pickup or destructible, add a **Destructible Saver**. Some objects, such as turrets, don't gracefully handle being destroyed; in this case, set **Destroy Mode** to *Deactivate* instead of *Destroy*.

If the destructible leaves behind a "corpse," assign the corpse prefab to **Destroyed Version Prefab**. Alternatively, assign a spawned object version of the prefab to the vp_DamageHandler as described in the next section, *Spawned Objects*. The demo scene's breakable crate demonstrates a spawned object, although in general you should assign Destroyed Version Prefab instead because the setup is simpler.

If you're configuring multiple GameObjects with the same name, assign a unique **Key** to each.

Tick **Save Across Scene Changes** to remember changes when leaving the scene and returning to it. This will increase the size of the saved game file.



Other Savers: Moveables, Active, Enabled, Animator

To save a GameObject's position, add a **Position Saver**. The breakable crate in DemoScene1 has a Position Saver that remembers the crate's position if the player picks it up and moves it. You don't need to add this to the player. To save a GameObject's animator state, add an **Animator Saver**.

To save a GameObject's active/inactive state add an **Active Saver**. To save a component's enabled state, add an **Enabled Saver**. In both cases, put the saver on a GameObject that's guaranteed to be active. To save the state of a pickup or destructible, use the instructions below instead.

Saver Keys

Every saver component needs a unique key under which to record its data in saved games. If the Key field is blank, it will use the GameObject name (e.g., “Orc”). If you tick **Append Saver Type To Key**, the key will also use the saver’s type (e.g., “Orc_PositionSaver”). This is useful if the GameObject has several saver components.

If GameObjects have the same name, you will need to assign unique keys. Otherwise they will all try to record their data under the same key, overwriting each other. To automatically assign unique keys to every saver in a scene, select menu item **Tools > Pixel Crushers > Common > Save System > Assign Unique Keys…**.

You can also write your own Saver components. The starter template script in Plugins / Pixel Crushers / Common / Templates contains comments that explain how to write your own Savers.

Spawned Objects

This section describes how to configure spawned objects to be saved in saved games.

Spawned objects are managed by a component called **Spawned Object Manager**. Although it has the word “Spawned” in the name, it has nothing to do with UFPS's vp_Respawner component. “Spawned” in this case means UFPS has instantiated an object into the scene, such as the clutter left behind after destroying a breakable crate. The Spawned Object Manager keeps track of these spawned objects. When you load a game, it re-instantiates the objects.

This is an overview of the configuration process:

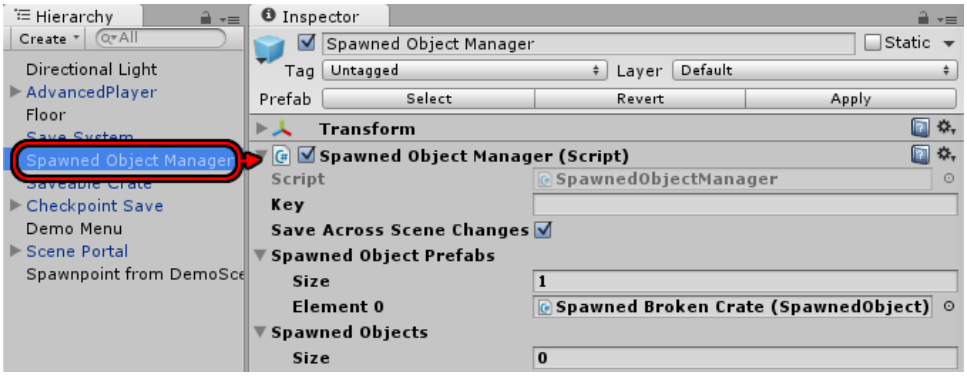
1. Create a copy of a drop/death spawn prefab, and add a **Spawned Object** component.
2. Add a **Spawned Object Manager** to the scene, and assign the prefab to it.
3. Configure objects to drop instances of this prefab instead of the regular UFPS prefab.

Create Spawned Object Prefab

Find the prefab (such as BreakingCrate), and make a copy, for example named Spawned Broken Crate. Add a **Spawned Object** component to it. Repeat for all items that the player can drop.

Create Spawned Object Manager

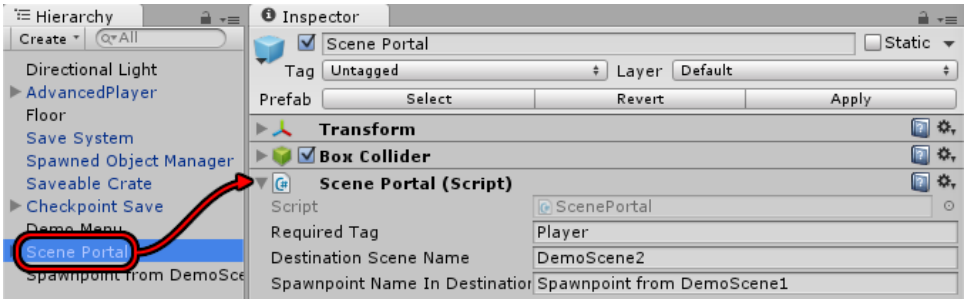
Drag the **Spawned Object Manager** prefab from the Prefabs folder into the scene, or create an empty GameObject and add a **Spawned Object Manager** component as shown below. Each scene should have its own Spawned Object Manager.



Add *all* Spawned Object prefabs that the player can drop to the **Spawned Object Prefabs** list. If a prefab is missing from the list, the Spawned Object Manager will not be able to respawn it when loading games or returning to the scene.

Scene Portals

To set up a transition to another scene, drag the **Scene Portal** prefab from the Prefabs folder into your scene, or add a GameObject with a trigger collider and add a **Scene Portal** component.



Set **Destination Scene Name** to the scene that this portal leads to. Make sure the destination scene is in your project's build settings.

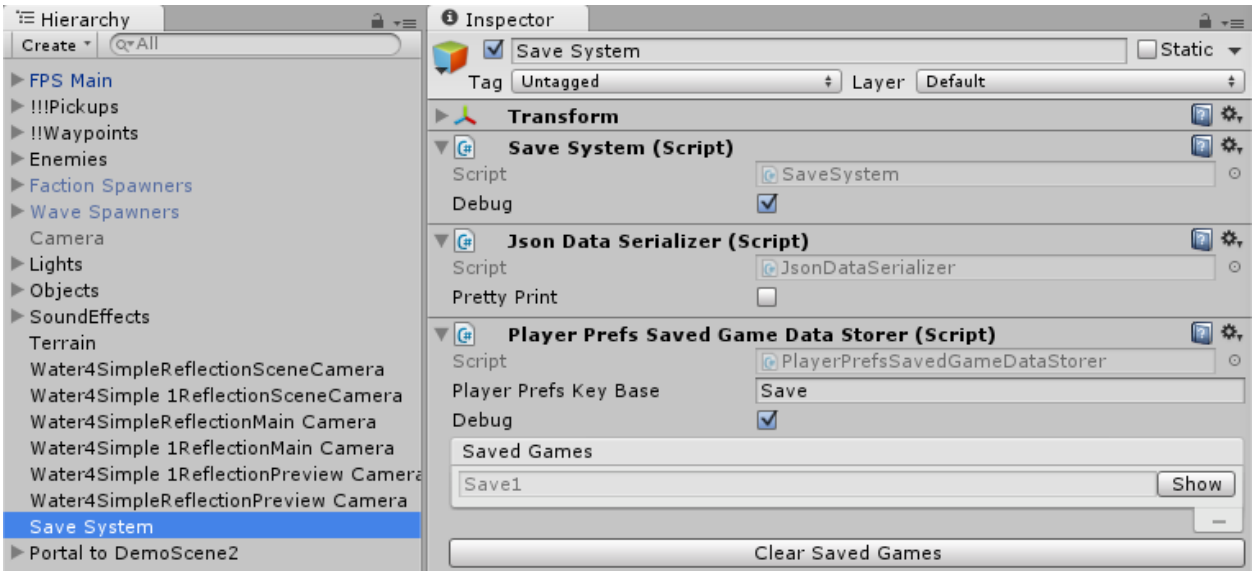
In the destination scene, create an empty GameObject where the player should appear. This is called a *spawnpoint*. In the original scene's Scene Portal component,set the **Spawnpoint Name In Destination Scene** field to the name of that spawnpoint.

Make sure your spawnpoint is far enough away from any scene portals in the destination scene so it won't immediately trigger another scene change.

If you don't want to use trigger colliders, you can manually call the **ScenePortal.UsePortal** method.

Save System

This step is optional. The Save System will automatically create a Save System GameObject at runtime. However, you can manually add one if you want to customize it. To add it, drag the **Save System** prefab from the Prefabs folder into your scene, or create a GameObject and add a **Save System** component. This GameObject acts as a *persistent singleton*, meaning it survives scene changes and typically there is only one instance.



The Save System relies on two types of components:

- **Data Serializer**: Converts binary saved game data into a saveable format.
- **Saved Game Data Storer**: Writes serialized data to persistent storage such as PlayerPrefs or local disk files.

By default, the Save System uses **Json Data Serializer**. If you want to use a different serializer, you can add your own implementation of the DataSerializer class to the Save System GameObject.

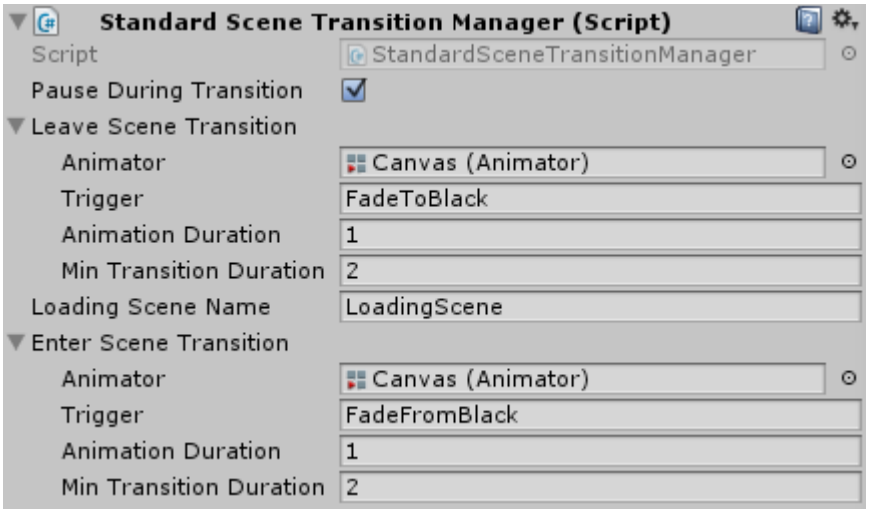
By default, the Save System uses **PlayerPrefs Saved Game Data Storer** to save games to PlayerPrefs. The Save System also ships with a **Disk Saved Game Data Storer** that saves games to encrypted local disk files; to use it, remove the PlayerPrefs Saved Game Data Storer (if present) and add Disk Saved Game Data Storer. If you want to store games a different way, you can add your own implementation of the SavedGameDataStorer class.

If you have a menu system, to load and save games you can assign the methods **SaveSystem.SaveGameToSlot** and **SaveSystem.LoadGameFromSlot** to your UI buttons' **OnClick()** events. Or in scripts you can use the equivalent static methods **SaveSystem.SaveToSlot** and **SaveSystem.LoadFromSlot** described in the next chapter.

Similarly, to change scenes without using the Scene Portal component, you can call **SaveSystem.LoadScene** in scripts or assign **SaveSystem.LoadScene** to your UI buttons' **OnClick()** events. Since the Save System GameObject might not be in all of your scenes, you can add a **Save System Methods** component and direct OnClick() to its LoadScene method instead.

Scene Transition Manager

To play outro and intro animations, and/or show a loading scene while loading the next actual scene, add a **Standard Scene Transition Manager** to your Save System:



If **Pause During Transition** is ticked, make sure your Animator(s) are set to Unscaled Time.

If a Scene Transition Manager is present, the Save System will:

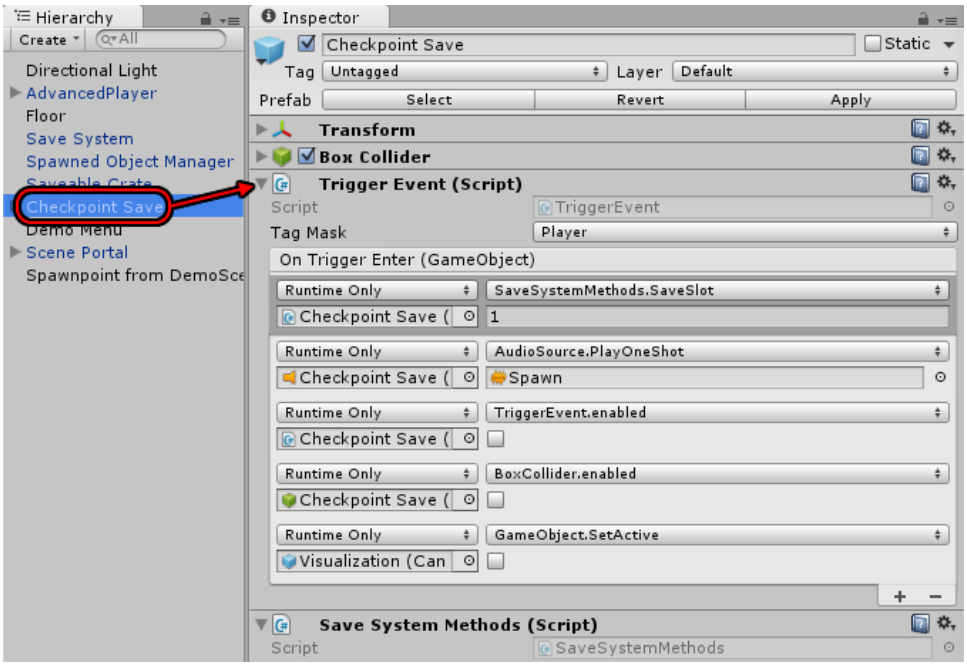
1. Set the Leave Scene Transition's animator trigger (if specified).
2. Load the loading scene (if specified).
3. Asynchronously load the next actual scene.
4. After the actual scene has been loaded, set the Enter Scene Transition's trigger (if specified).

The default Save System prefab includes a one second fade-to-black when leaving the old scene and a one second fade-from-black when entering the new scene. This allows the Save System to reposition GameObjects in the new scene while it's covered in black.

If you want to use a different effect between scenes, you can write your own subclass of SceneTransitionManager and add it to your Save System instead.

Checkpoint Saves

To set up a checkpoint save trigger, drag the **Checkpoint Save** prefab from the Prefabs folder into the scene, or add a GameObject with a trigger collider and add **Trigger Event** and **Save System Methods** components. The prefab is preconfigured to save in slot 1, play an audio clip, and hide the checkpoint.

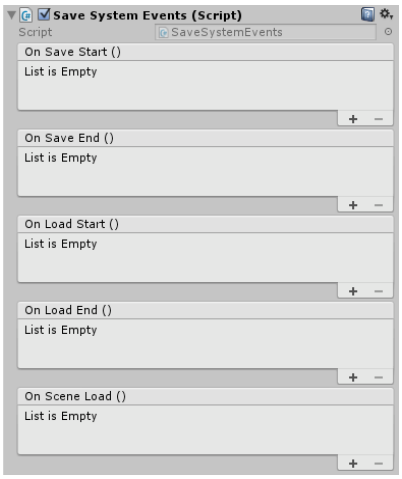


If you add the components manually, configure the Trigger Event's **On Trigger Enter** event to call **SaveSystemMethods.SaveSlot**, and specify a slot number. Optionally, disable the trigger after the checkpoint save by adding events as shown above.

If you want the player to automatically respawn at the last checkpoint save when killed, replace the player's vp_PlayerRespawner component with vp_PlayerRespawnAtCheckpoint.

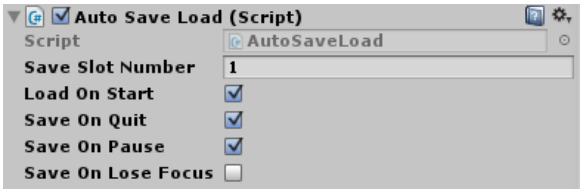
Save System Events

The Save System Events components allows you to hook up events in the inspector.



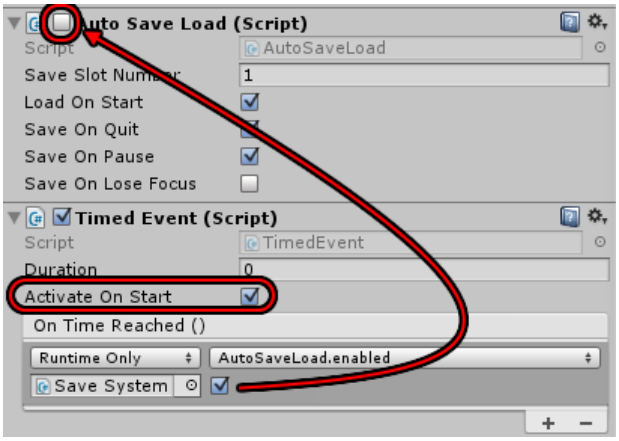
Auto Save Load

Mobile games typically auto-save when the player closes the game and auto-load when the player resumes the game. To add this behavior to your game, add an **Auto Save Load** component to the Save System:



Tick **Load On Start** to load the saved game (if it exists) on start, and **Save On Quit** to save the game when quitting the app. Tick **Save On Pause** to also save the game when the player pauses/minimizes it. This way the game will be saved correctly if the player pauses the app and then kills it, instead of exiting it normally in the app itself. Tick **Save On Lose Focus** to also save the game when the app loses focus.

If your game uses a splash screen, you may not want to enable Auto Save Load until the first scene actually loads. Otherwise, if the player minimizes the app during the splash screen, the Auto Save Load component could save the game in its initial state, before it has loaded its previously-saved data. To set this up, disable Auto Save Load, and use a **Timed Event** component to enable it on Start:


Auto Save Load (Script)
Script
AutoSaveLoad
Save Slot Number
1
Load On Start
Save On Quit
Save On Pause
Save On Lose Focus
Timed Event (Script)
Script
TimedEvent
Duration
0
Activate On Start
On Time Reached ()
Runtime Only
AutoSaveLoad.enabled
Save System

Chapter 3: Scripting

The Save System has a very simple scripting interface.

SaveSystem class

These methods are available though SaveSystem:

```
public static void LoadFromSlot(int slotNumber)
```
Loads a previously-saved game from a slot.

```
public static void SaveToSlot(int slotNumber)
```
Saves the current game to a slot.

```
public static void DeleteSavedGameInSlot(int slotNumber)
```
Deletes the data in a saved game slot.

```
public static void LoadScene(string sceneNameAndSpawnpoint)
```
Loads a scene, optionally positioning the player at a specified spawnpoint. The parameter is a string containing the name of the scene to load, optionally followed by "*@spawnpoint*" where "*spawnpoint*" is the name of a GameObject in that scene. The player will be spawned at that GameObject's position. This method implicitly calls RecordSavedGameData() before leaving the current scene and calls ApplySavedGameData() after loading the new scene.

```
public static SavedGameData RecordSavedGameData()
```
Returns a SavedGameData object containing the saved data from the current game.

```
public static void ApplySavedGameData(SavedGameData savedGameData)
```
Applies a SavedGameData object to the current game.

```
public static void LoadGame(SavedGameData savedGameData)
```
Loads a game previously saved in a SavedGameData object.

```
public static void LoadScene(string sceneName, string spawnpointName = null)
```
Loads a scene, optionally positioning the player at a spawnpoint in the new scene.

```
public static void RestartGame(string startingSceneName)
```
Clears the current save game data and restart the game at the specified scene.

Saver class

Saver is the base class for any components that record data for saved games. You can create subclasses to extend the data that the Save System saves. A starter template with detailed comments is provided in Plugins ▶ Pixel Crushers ▶ Common ▶ Templates ▶ SaverTemplate.cs .

DataSerializer class

DataSerializer is the base class for data serializers that the Save System can use to serialize and deserialize saved game data. The default subclass is JsonDataSerializer.

SavedGameDataStorer class

SavedGameDataStorer is the base class for data storers. A data storer writes and reads a SavedGameData object to and from some storage location, such as PlayerPrefs or a disk file. The default subclass is PlayerPrefsSavedGameDataStorer, but the Save System also includes DiskSavedGameDataStorer which saves to encrypted disk files on supported platforms (e.g., desktop).

Events

You can register listeners for these C# events:

```
public static event SceneLoadedDelegate sceneLoaded = delegate { };
public static event System.Action saveStarted = delegate { };
public static event System.Action saveEnded = delegate { };
public static event System.Action loadStarted = delegate { };
public static event System.Action loadEnded = delegate { };
```

Save System Methods

To access Save System methods without scripting, such as in a UI Button's OnClick() event, add a **Save System Methods** component to the scene. This component exposes the methods of the SaveSystem class to the inspector.

Chapter 4: Serializers

The default serializer is JSON Data Serializer. However, you can remove this and add a Binary Data Serializer if you want to serialize to binary format. If your Savers use types that are not serializable, you will need to make a subclass of Binary Data Serializer and add serialization surrogates. For examples, see the scripts BinaryDataSerializer.cs, Vector3SerializationSurrogate.cs, and QuaternionSerializationSurrogate.cs.

Chapter 5: Third Party Support

Integration support for third party assets is included the unitypackages listed below. Documentation is included in each third party support package.

Compass Navigator Pro Integration

Compass Navigator Pro © Kronnect

Plugins ▶ Pixel Crushers ▶ Common ▶ Third Party Support ▶ Compass Navigator Pro Support

Emerald AI 2 Integration

Emerald AI © Black Horizon Studios

Plugins ▶ Pixel Crushers ▶ Common ▶ Third Party Support ▶ Emerald AI Support

PlayMaker Integration

PlayMaker © Hutong Games

Plugins ▶ Pixel Crushers ▶ Common ▶ Third Party Support ▶ PlayMaker Support

Tactical Shooter AI Integration

Tactical Shooter AI © Squared 55

Plugins ▶ Pixel Crushers ▶ Common ▶ Third Party Support ▶ Tactical Shooter AI Support