Persistence

- TilePlus Persistence

TilePlus Persistence

Introduction

Look at any Unity or Reddit forum and you'll see endless posting about how to go about saving and restoring data.

TilePlus Toolkit has a built-in, efficient, and simple to use save/restore scheme designed specifically for TilePlus tiles: TpPersistence.

TpPersistence lets you obtain save data directly from tiles and restore it directly to tiles using the ITpPersistence interface for Type independence.

It's not a TilePlus service like the Spawner or Tweener. Rather, it's a flexible interface specification that lets you easily save and restore arbitrary save-data from TPT tile instances without regard to what Type they are.

The tile class itself decides what it wants to save and restore and supplies that information as a string (e.g., JSON although there's no requirement to do so) and the TPT tile's GUID.

The GUID can be used to look up the tile instance when moving data from the filesystem to a tile.

ITpPersistence interface

The ITpPersistence interface is pretty simple:

```
public interface ITpPersistenceBase { }



public interface ITpPersistence<out TR, in T> : ITpPersistenceBase
               where T:MessagePacket<T> where TR:MessagePacket<TR>
{
   /// <summary>
   /// Implement to provide data to save
   /// </summary>
   /// <returns>TR.</returns>
  TR GetSaveData(object options = null);
```

```
    /// <summary>
    /// Implement to be sent data to restore
    /// </summary>
    /// <param name="dataToRestore">The data to restore.</param>
   void RestoreSaveData(T dataToRestore);

    /// <summary>
    /// Implementations may set this false if either
    /// data has already been restored OR if it doesn't
    /// want data restoration at all.
    /// </summary>
   bool AllowsRestore { get; }

    /// <summary>
    /// Implementations may set this false if
    /// nothing has changed in the tile.
    /// That avoids saving data from this tile if nothing
    /// has changed since instantiaton.
    /// Default is true;
    /// </summary>
   bool AllowsSave => true;
 }
```

If you've already read the Messaging chapter this may look familiar: it uses the same MessagePacket abstract class.

But there are two different 'packets': one for getting save data (TR) and one for restoring data (T).

They're both derived from the abstract MessagePacket<T> class and can be different.

Here's an example from the LayoutSystem demo:

```
SaveDataWrapper ITpPersistence<SaveDataWrapper, StringPacket>.GetSaveData(object? options)
{
    var obj = new TreasureChestSaveData(this, wasEncountered);
    return new SaveDataWrapper() {m_Json = JsonUtility.ToJson(obj, false), m_Guid =
TileGuidString};
}

/// <inheritdoc />
void ITpPersistence<SaveDataWrapper, StringPacket>.RestoreSaveData(StringPacket dataToRestore)
```

```
{
    if(dataToRestore == null)
        return;
    var data = JsonUtility.FromJson<TreasureChestSaveData>(dataToRestore.m_String);
    if (data != null)
        wasEncountered = data.m_Encountered;
    if (!wasEncountered)
        return;

    //clear the sprite and schedule removing this tile.
    ClearSprite(); //avoids a visual artifact.
    if(ParentTilemap)
        DelayedCallback(this, () => { ParentTilemap.SetTile(TileGridPosition,null); },
"Tchest-deltile");
}
```

PLEASE NOTE that these implementations are 'Explicit' and should always be written this way.

and this is SaveDataWrapper (part of the standard distribution)

```
/// <summary>
/// A simple wrapper class for saving data
/// </summary>
[Serializable]
public class SaveDataWrapper : MessagePacket<SaveDataWrapper>
{
    /// <summary>
    /// The JSON string for this object
    /// </summary>
    [SerializeField]
    public string m_Json = string.Empty;
    /// <summary>
    /// The GUID to be used when restoring data
    /// </summary>
    [SerializeField]
    public string m_Guid = string.Empty;

    /// <inheritdoc />
    public SaveDataWrapper() : base(null)
    {
    }
}
```

So What Does It Do?

Start from the bottom-up: SaveDataWrapper. Two strings. One is whatever JSON-formatted data that you want to save. The other is the GUID of the tile.

The basic idea is that TPT tiles won't usually have a lot of data to save. Create a small class specific to what you want to save. JSON-encode the class into a string, and place that JSON string into an instance of SaveDataWrapper along with the GUID of the tile.

To do this you need to create a class for whatever data you want to save. In the case of the TreasureChest tile it just needs to save a boolean value that indicates whether or not the Treasure Chest has been encountered, i.e., triggered and the opening treasure chest animation has played.

```
public class TreasureChestSaveData : MessagePacket<TreasureChestSaveData>
{
    /// <summary>
    /// true if the treasure chest was encountered.
    /// </summary>
    [SerializeField]
    public bool m_Encountered;

    /// <inheritdoc />
    /// <param name="sourceInstance">Source of message or null</param>
    /// <param name="encountered">true if this chest was already activated.</param>
    public TreasureChestSaveData(Object? sourceInstance, bool encountered) :
base(sourceInstance)
    {
        m_Encountered = encountered;
    }
}
```

Process

Saving

When a tile handles the GetSaveData implementation it just has to return a string in whatever format it understands. It doesn't have to be JSON although that's really convenient.

```
SaveDataWrapper ITpPersistence<SaveDataWrapper, StringPacket>.GetSaveData(object? options)
{
    var obj = new TreasureChestSaveData(this, wasEncountered);
    return new SaveDataWrapper() {m_Json = JsonUtility.ToJson(obj, false), m_Guid =
TileGuidString};
}
```

So we create an instance of TreasureChestData, JSON-ize it, and 'wrap' it in a SaveDataWrapper.

Restoring

When a tile handles the RestoreSaveData implementaton it's sent a packet containing a string that can be JSON-decoded as shown below.

```
void ITpPersistence<SaveDataWrapper, StringPacket>.RestoreSaveData(StringPacket dataToRestore)
{
    if(dataToRestore == null)
        return;
    var data = JsonUtility.FromJson<TreasureChestSaveData>(dataToRestore.m_String);
    if (data != null)
        wasEncountered = data.m_Encountered;
    if (!wasEncountered)
        return;

    //clear the sprite and schedule removing this tile.
    ClearSprite(); //avoids a visual artifact.
    if(ParentTilemap)
        DelayedCallback(this, () => { ParentTilemap.SetTile(TileGridPosition,null); },
"Tchest-deltile");
}
```

Here, the JSON is unpacked into an instance of TreasureChestSaveData, and the 'wasEncountered' value is restored.

If the tile HAD been encountered before it should be deleted. We clear the sprite and set up a delayed callback that will delete the tile near the end of the current frame.

Why delay it? Unity may/might/can crash if a tile itself does SetTile(position, null).

It's much safer to use DelayedCallback. With no 'delay' time specified, the callback is invoked at the next TpLib Update (which is always near the end of a frame).

In the same Layout demo you can look at the Waypoint tile. There you'll see that the data restore process is used to change the visual appearance of the Waypoint: if it had been encountered before then the Waypoint displays as enabled (a different sprite).

There's Something Missing

What's missing is what you have to do to actually save and restore data. That's of course up to you.

A sample implementation can be found in the Layout demo. In LayoutDemoSaveData.cs you'll see that the JSON and GUIDs from the tiles are saved in a dictionary which is serialized along with all the other save data.

Since that's a Layout system demo, chunks of tiles are added and deleted as the player character moves around. There's an in-depth chapter on this demo which explains how save data is moved beteen tiles and save-files as tiles are loaded and unloaded by the layout system.

A simpler implementation when not using the Layout system could just save the tile JSON and GUIDs as a separate file.

That Interface Specification...

```
public interface ITpPersistence<out TR, in T> :
ITpPersistenceBase where T:MessagePacket<T> where TR:MessagePacket<TR>
```

This notation means that the T parameter is covariant and the TR parameter is contravariant.

In practice:

- TR is a value to be returned. In these examples, that's SaveDataWrapper.
  - Or any concrete subclass of the MessagePacket abstract class.
- T is a value to be sent. In these examples, that's StringPacket.
  - Or any concrete subclass of the MessagePacket abstract class.

(this is a somewhat simplistic explanation for an advanced C# topic).

So you do not have to use SaveDataWrapper, although it is usually a good class to use.

Neither do you have to use StringPacket.

However, these are very flexible and are able to handle most uses case where you'd want to store and restore tile data.

As long as you implement the interface EXPLICITLY you should have no issues.

Notes

You don't have to use JSON. For a simple use like shown above, you could have an empty or non-empty string represent the two states of a boolean value.

```
wasEncountered = string.IsNullOrEmpty(dataToRestore.m_String);
if (!wasEncountered)
   return;
```

However, if you look at how the data are handled in the Layout demo, using JSON has a few advantages:

- All tiles use exactly the same process to encode save data.
- It's easy to modify the save-data as Tiles are added and deleted during layout without having to know that tile A uses JSON and tile B does something different.

References

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation

https://learn.microsoft.com/en-us/dotnet/standard/generics/covariance-and-contravariance