Interfaces

ITilePlus

This is the main interface for TilePlus tiles. Almost all of the interface is implemented in TilePlusBase. When creating new tiles you'd normally be inheriting from TilePlusBase, TpFlexAnimatedTile or TpSlideShow which complete the remaining methods and/or properties.

ITilePlus exists mostly to provide a Type-independent means of accessing TilePlusBase fields, properties, and methods.

Why so many properties? Fewer serialized fields for things that are just basically boolean switches used by various parts of the system. For subclasses that don’t implement a particular functionality, specific fields aren’t needed, just a constant value.

For those that do, serialized and non-serialized fields allow data to be provided via the properties, or a return value is computed for the property.

The TilePlusBase class implements all the items in the ITilePlus interface except those having to do with simulation: that has default values provided by the properties in the interface (C# 8 feature).

Most of what’s in ITilePlus are used internally and it’s unlikely that you’ll use them at all. But there are three which are especially useful: ParentTilemap, TileGridPosition, and TileWorldPosition.

- ParentTilemap always has a reference to the tilemap for the tile. This is useful for a lot of things, but beware: if a tile tries to erase itself by using the ParentTilemap reference to place a null tile at the TileGridPosition, Unity will crash.
  - Or, it did the last time I tried it. Don't try it. Use TpLib's DelayedCallback to change the timing of the null-tile placement.
- TileGridPosition always has the location of the tile in Grid coordinates
- TileWorldPosition always has the location of the tile in World coordinates.

ITpMessaging

ITpMessaging is used to implement targets for the Messaging Service.

It's pretty simple:

```
/// <summary>
/// Interface for using TpLib SendMessage methods.
/// </summary>
/// <typeparam name="T">Type for sending a message</typeparam>
public interface ITpMessaging <in T> where T:MessagePacket<T>
{
    /// <summary>
    /// Send a message of type T
    /// </summary>
    /// <param name="sentPacket">The sent packet.</param>
    void MessageTarget(T sentPacket);

    /// <summary>
    /// Optional "are you ready?" method that can be used in filtering
    /// prior to sending a message. Useful in some edge cases. Override
    /// in implementation if necc. NOTE this is NOT checked internally
    /// somehow. You have to use a filter and test this.
    /// </summary>
    /// <returns>True if the tile is prepared to get the message.</returns>
    bool CanAcceptMessage() { return true; }

}
```

Message Packets are discussed here.

In your custom tile code you use explicit implementations for these members, for example:

```
void ITpMessaging<ActionToTilePacket>.MessageTarget(ActionToTilePacket sentPacket)
{
    ActivateAnimation(!AnimationIsRunning);
}
```

`T` is ActionToTilePacket. In this example, the packet information is ignored.

Here's a more complex example where the packet contents are used to control what happens. In this case, `T` is PositionPacket, which just contains a position (like the Player position). This is from TpAnimZoneSpawner.

```
void ITpMessaging<PositionPacket>.MessageTarget(PositionPacket sentPacket)


{
    var pos = sentPacket.m_Position;
    lastContactPosition = pos;


    pos -= m_TileGridPosition; //remove offset
    if (m_ZoneBoundsInt.Contains(pos))
        SpawnTileOrPrefab();
}
```

This tile has a 'Zone' (aka, BoundsInt) that describes an area. It doesn't even have to cover the tile itself. But the BoundsInt position is the offset of the zone from the tile's position and not an absolute position: think of it as relative addressing.

That's why the tile's grid position is subtracted from the position information in the packet before seeing if the packet's position information is a point within the BoundsInt.

If the position is within the BoundsInt then we spawn something, based on how the tile is set up.

ITpPersistence

This interface specifies endpoints for save and restore data.

It's something optional and a bit low-level. But it's an easy way to save and restore TPT tile's instance data of your choosing.

```
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.
        /// New in 3.1
        /// </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;
        /// New in 3.1
        /// </summary>
```

```
        bool AllowsSave => true;


    }
```

There's a chapter on Persistence here.

Layout-Related

ITSceneInitializer

Scene Initializers are used as a way to move scene initialization code into Scriptable Object assets, and are discussed here.

IChunkSelector

Selectors are used by the Layout system to find out what to put where, and are discussed here.

Others

These are mostly for internal use.

IActionPlugin

Used with ZoneActions and EventActions to provide a way to have a second asset (typ, a Scriptable Obj but can be any UnityEngine.Object) be inspectable thru the IMGUI tile editor (selection inspector).

Note that the asset ought to be a PROJECT asset and NOT a SCENE object, although this is not enforced or checked.

IHoverableControl

This is used to mark a tile as a 'Hover Zone', that is, a tile which accepts zone-based hover events. This interface inherits from `ITpMessaging<BoolPacket>` and the MessageTarget for a BoolPacket must be implemented. See the `UiHoverZone` script for an example.

When a tile implements this interface (and it is on the SINGLE tilemap used for Hoverable controls) it gets messaged with a BoolPacket (a small class with a boolean value in it) when the zone as described by tile's ZoneBoundsInt (part of every TilePlus tile) when the Zone is entered and exited.

Basically, for Hover Zones the zone is set (as usual) within the HoverZone tile to encompass whatever area you want. When that area is entered, a BoolPacket is sent with the value = true and when the area is exited the packet is sent with the value = true. The HoverZone tile uses this to do whatever it wants; typically the tile will invoke a ZoneAction S.O. which can do whatever arbitrary task you want. As set up in the UI demo they're used for tooltips.

- See this.

IScriptableService

This is used by TpLib's Update dispatcher and by the ServiceManager. See Services.

ITpSpawnUtilClient

This is used by TPT tiles that wish to use the spawner in a particular way. You'll never need it but you can see how it's used in TpAnimZoneSpawner and TpAnimatedSpawner.

ItpUiControl

This provides a Type-independent way of communicating with the TPT UI-variety tiles.

IZoneActionTarget

This interface is applied to tiles that can be targets of Zone Actions. Zone Actions may re-transmit TpMessaging messages to other tiles, but always should test this interface to see if the tile wants to accept such messages. See the Zone Actions chapter.