

Services

Dynamically loadable Scriptable Object Services for the TilePlus system

- [TilePlus Services](#)
- [Tweeners Service](#)
- [Spawner Service](#)
- [Messaging Service](#)
- [TilePositionDb Service](#)

TilePlus Services

Overview

In the TilePlus system, Runtime-only Scriptable Objects are called Services or SRS, and are controlled by TpServicesManager.

Services are built on a base class called ScriptableRuntimeService, which handles automatically pre-registering the service with TpServiceManager prior to the first scene load.

It's easy to create your own, just inherit from ScriptableRuntimeService and add your own data or code. Be sure to call the base classes in OnEnable and OnDisable if you override them.

Ensure that you have something like this in your derived class:

```
[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
private static void InitOnLoad()
{
    TpServiceManager.PreRegisterSrs(    nameof(YOUR_CLASS), Create);
}
```

The `BeforeSceneLoad` version of the attribute is REQUIRED. If omitted, the service won't be available until this specific service actually registers itself.

TpServiceManager internally keeps Services in two groups:

- Available: Services which have pre-registered themselves using the method shown above.
- Running: Available Services which have been loaded.

TpServiceManager has properties and methods to deal with services:

Properties

- GetAllRunningServices: an array of ScriptableObjects that are running services.
 - This is editor-only and used for in-editor diagnostics like the ServicesInspector and System Info windows.
- GetAllRunningServiceTypes: an array of running Service Types (c# Types)
- GetAllAvailableServiceTypes: an array of available service name strings.

Methods

- `HasServiceOfTypeAvailable`: input is a string e.g., `nameof(TpSpawner)`. Returns true if the specified service is AVAILABLE (may not be running).
- `HasRunningServiceOfType`: input is a Type. Returns true if the specified Type of Service is running.
- `HasAllTheseRunningServices`: Test for a set of active services. Input is an `IEnumerable` of Types. Returns true if ALL the services are running.
- `GetServiceOfType<T>`;; One way of obtaining a service handle.
 - Example: `GetServiceOfType<TpSpawner>()`;
 - If the service is NOT running this method will start it.
 - You can specify an initialization callback to this method if the Service requires it.
- `GetRunningService<T>(out T? Service)`: Obtaining a Service handle ONLY if it's already running.
 - Example: `if(GetRunningService<TpSpawner>(out var spawner){...}`
 - Returns true if there's an already-running service of Type T (and the out param has the handle)
 - Returns false if there isn't an already-running service of Type T (and the out param is Null)
 - Unlike `GetServiceOfType<T>`, the service isn't started if it isn't already running.

`GetServiceOfType` is quite simple:

- ALWAYS fails unless the Editor is playing or about to change into Play mode.
- Does a fast dictionary lookup to see if the service is already running.
 - If it is, return the service's handle (instance).
 - If it isn't, see if it's available (pre-registered)
 - If it isn't: fail (return NULL).
 - If it is, create the Service instance, which optionally invokes the initialization callback (if needed).
 - If Service Instance was already active or newly created, return that. Otherwise return null.

`GetRunningService<T>` is also ONLY for use when in Play mode.

Finally, if you want to terminate a service, use:

```
TerminateServiceOfType<T>()
```

If the service is running this will Destroy the service. Normally, services persist throughout the lifetime of the application once they're loaded. In the Layout demo the Dialog Box seen when you first encounter a treasure chest is actually a Bundle loaded by the `DialogBoxService` and this service is terminated when the Dialog box close button is clicked.

Terminating a running service doesn't mean it's no longer available: you can just use `GetServiceOfType` to reload it.

Convenience Properties

Since the Messaging and Spawning services are commonly used, shortcut properties exist that just let you get the handle:

- `MessagingService` returns a handle to `TpMessaging`
- `SpawnerService` returns a handle to `TpSpawner`.

These use `GetServiceOfType<T>` and will auto-start the service if not already running.

Note that the `TpTileTweenService` is available via a protected static (i.e., within the class or derived classes only) property in the `TilePlusBase` class (which all `TilePlus` tiles derive from). This reflects the fact that this tweener is ONLY for use with `TilePlus` tiles.

- Although a `Monobehaviour` or other code can easily get a handle via `GetServiceOfType`, creating Tweens requires a `TilePlusBase` instance as part of the method call. Please refer to the [Tween documentation](#) for more information.
- This convenience property also auto-starts the Tween Service if it isn't running.

Services as Singletons

Usually a service is a front-end for something that needs to be defined by an API and/or an Interface, and in that sense, they're by nature singletons at least from an external point of view. Internally a service may handle multiple instances of something else but that's mostly hidden from code that uses the service.

That's the general model for this simple, but efficient, service scheme.

The `ScriptableRuntimeService` class can actually have multiple instances: there's no static 'Instance' at all.

It's the implementation used in `TpServiceManager` that enforces only one instance of each service for two main reasons:

- Lookup speed: simpler data structures.
- Multiple instances are harder to differentiate in your code.

Other systems in `TilePlus` need multiple `Scriptable Object` instances. Specifically, the `Layout System` which creates multiple instances of `TpZoneManager Scriptable Objects`. Here, the different instances are differentiated by their names. It's quite a bit more complex although the details are

largely hidden via the use of Monobehaviour Components for setting up parameters.

Hence, Services are forced to be singletons as an implementation detail for this particular type of dynamically loadable service.

The intent is to use Services for, well, Services and not global data storage or state.

The Tweener, Messaging, Spawning, TileFabLib, and PositionDb services are autonomous and don't depend on any external state aside from that being provided from internal Unity API interactions, and TpLib logging and 'Tilemap DB' methods. All their internal data are private and only accessible via methods or properties. The only significant dependency occurs when a Service requires an Update method invocation - that's minor and not a Service-to-Service dependency.

Creating services which have cross-dependencies is a bad idea and you will be plagued with bugs unless you are extremely careful.

Back to Singletons...

But there's nothing stopping you from using Services as (sort of) conventional singletons if you want to. See the LayoutSystem demo for an example.

- In that example one finds a 'GameState' service, which holds global state including the data which are saved when code in the example requests a game save.
- This approach was taken to keep the example simple (its complex enough as it is).
- Depending on your viewpoint about Singletons, you may or may not want to avoid this approach.

More Grisly Details

The IScriptableService interface

IScriptable is an interface that lets a SRS specify how it wants to receive Update events (or not). If a service DOES NOT implement IScriptableService it inherits the default Interface properties which proudly exclaim: No Updates!

Not all Services need an Update event; for example, TpMessaging and TpSpawner. Some Services need an Update event; for example, TpTileTweener and TpTilePositionDb.

Services which require Update events must implement the following:

```
bool IScriptableService.WantsUpdate => true;
bool IScriptableService.ReadyForUpdate => true;
void IScriptableService.Update(float deltaTime){}
```

Note that these are EXPLICIT interface implementations and MUST be done this way.

- WantsUpdate is only examined twice: when the Service registers itself when loaded and when it deregisters itself when unloaded (which may not be till program shutdown).
 - WantsUpdate should NEVER be a variable. If this value is different for registration and deregistration then an exception will occur during the deregistration code or soon after, at the next TpLib internal update.
- ReadyForUpdate is examined every Update and doesn't always have to be true. If your code doesn't always need Update or if it wants to delay enabling Update for some reason then the property can return the value of a variable or test some condition. The Tweener uses this feature to only get Update events when there is actually work to do.
 - In words: Update is called every frame unless ReadyForUpdate is false.

Update Event Timing for Services

Note that Update timing isn't the same as MonoBehaviour Update and doesn't originate from a GameObject in any scene.

- If the PlayerLoop is being used: PostLateUpdate
- Otherwise, an Awaitables-based Update 'Pump' is used: EndOfFrame.

Which variety of Update generation is controlled by a toggle in a TpLibInit asset. There's a preinstalled instance in your project at: Assets/TilePlus/Resources/TP/TpLibInit.

Tweener Service

Intro

TpTileTweener is a TPT service that can be used for tweening TilePlus tile sprites. The Tweener also has support for sequences.

Generally it's for use within TilePlus tile code, ZoneActions, or EventActions. The main restriction is that a TPT tile reference must be provided when creating Tweens or Sequences.

You use it by first obtaining a service handle from `protected static TpTileTweener TweenerService` property within any TPT tile code.

For example:

```
TweenerService.CreateTween(this,
    Vector3.zero,
    Color.red,
    Matrix4x4.identity,
    TpEasingFunction.Ease.Linear,
    TpTileTweener.EaseTarget.Color,
    1,
    0,
    TpTileTweener.LoopType.PingPong,
    -1,
    OnFinished);
```

This creates and starts a Color tween (EaseTarget.Color) with the final color being Color.red.

Although the Linear easing function is specified here, it's ALWAYS linear (Lerp) for Color tweens (the value is ignored for Color tweens).

The duration is one second. The -1 means that this tween repeats a PingPong tween until killed. When the tween is killed then the OnFinished callback is invoked.

The first four parameters are a reference to the tile itself (this), a Vector3 value, a Color value and a Matrix value. For the Vector3, Color, and Matrix parameters you just use the one you need for a particular tween.

Here, the Vector3 field is set to Vector3.zero and the Matrix field is set to Matrix4x4.identity because this is a Color tween.

The Vector3 field is used for tweens that use Vector3 values like Position, Rotation, or Scale.

The Matrix parameter is used for Matrix tweens which are a special tween variety.

Why Another Tweener?

But why create another Tweener in the first place? The Unity Asset store is full of free Tweeners on the Unity Asset Store and DemiGiant's DOTween is really great.

Here's why: none of them support natively tweening Tile sprites.

- You can do it in DOTween with Getters and Setters but DOTween can't handle tiles suddenly becoming null (if they're deleted) in the same way as it does for GameObjects.
- TilePlus Toolkit has a 'DOTween Adapter' which handles some of that, but it's inconvenient (and deprecated).
- DOTween is great but is way more complex than needed just for tiles.

TpTileTweener is optimized for TilePlus tiles. It isn't for normal Unity tiles. But it does some things DOTween doesn't do. It's non-generic, tight, hard-coded and single-mindedly Tilemaps only.

Demos

TilePlus Extras has a TpTweener folder with a few example tiles and several scenes.

- TweenerFlexTile has fields for all possible types of tweens (except DELAY) and the fields which are shown change with what's being affected
 - I.E., if the tween target is Color then a color picker is available, but for, say, position, a Vector3 field is shown.
- TweenSpecTile uses a TpTweenSpec asset to run a tween from that asset.
- TweenSpecSequenceTile uses all the entries from a TpTweenSpec asset to create and run a sequence. You can also check a toggle for interactive use. This allows you to tweak the TweenSpec asset while the Editor is playing: each time the sequence ends it re-loads from the asset rather than internally repeating the cached sequence.

These three tiles are part of the normal distribution in the Plugins folder but for many uses, in a real app, you'd run a tween as the result of a Message being sent to a tile or because of a Zone entry or exit.

However, the stock 'Tweening' tiles are great for your experimentation with this feature; especially TweenSpecSequenceTile.

To use tweens in the code for Event or Zone actions, check out the `TpTweenSubObject`. `SubObjects` are scriptable objects which are attached as references to `TpTileZoneActions` or `TpTileEventActions`.

`TpTweenSubObject` can be used by a `TileAction` to easily tween the tile's sprite when an event is handled (`EventAction`) or when a Zone is entered or exited (`ZoneAction`.)

This approach is used in the `TileFabDemos/LayoutSystem` demo.

Tweens and Sequences

Tweens

`TpTileTween` supports tweening Position, Rotation, Scale, and Color of Tile sprites.

For color tweens you can also specify 'Constant A', which means that the alpha value doesn't change from the value found when the tween begins.

You can also tween the Tile's Matrix, or the Matrix with Color, or the Matrix with Color-Constant-A. These let you create a single tween that can change Position, Rotation, Scale and/or Color/Color-Constant-A, with optional fine-grained control over which parts of the transform change. For example, one could specify a Matrix tween where only position and scale are changed.

- Matrix tweens also allow specifying different Ease functions for Position, Rotation, and Scale.
- `MatrixColor` and `ColorConstantA` use Lerp for the Color tween.
- Color Tweens always Lerp

It's incredibly flexible, albeit a bit more work to set up.

Don't get confused: if you use one of the Matrix varieties, including those with Color, it's one tween. When the tween updates it can affect Position, and/or Rotation, and/or Scale, and/or Color in one operation. It's not three or four tweens in parallel.

To that end, the `TweenSpecSequenceTile` has an interactive mode. In this mode, the sequence is forced to not loop. When the sequence completes, it restarts with a fresh set of value from the Tween Spec asset. Hence, if you change values of this asset while the editor is Playing, such changes will be seen when the sequence restarts. As is true for project assets, the changes will remain after you exit play mode. It's a great way to play with sequences and Matrix-style tweens.

The tweener supports one-shot, looping, and ping-pong tweens. For looping tweens there's a loop count and as usual if the loop count is -1 the loop continues forever until cancelled or the tile is deleted.

Delay tweens can also be created, but they're only for sequences.

Sequences

A sequence can be created and tweens can be created with automatic adding to a sequence. Sequences do what you'd expect: run all the tweens in a list of tweens. Delay tweens can be added to a sequence. They don't tween anything, just time out. Since a callback can be issued each time the sequence changes to the next tween, a sequence of just Delay tweens can be used to create a simple sequencer.

Callbacks

Individual tweens have callbacks for each Update, on completion, and when a looping tween loops. Tweens are auto-deleted when their parent tile becomes null. Since a callback is issued to a tile and the tile is null in this case, the tween-ending callback is not issued.

Note that callbacks for each update can get extremely processor-intensive depending on how many tweens are running and what is done during execution of the callback. For example, if you have 100 tweens running then you'd get 100 callbacks every frame. Hence, these are recommended ONLY during development but there's no explicit restriction on this.

Note that while each callback provides a reference to the specific tween that caused the callback, the values in the instance are read-only properties so

1. Do not hold a reference outside the local scope. These are pooled at the end of the tween: it is returned to the pool and reset.
2. You can't change anything except a "Diagnostics" property.
3. You can't copy a tween and inject it back into the system.

Sequences have callbacks when complete and when the next tween in the sequence begins.

Sequences intercept the individual tween callbacks for the sequence's internal use but don't relay them to tiles.

Create a tween

There are four ways to create and run a tween.

- Use the Create methods
 - CreateTween: create any sort of tween.
 - CreateDelayTween: Don't tween anything, just wait. Sequences only.

- `CreateTweenFromSpec`: Use the `TpTweenSpec` asset to create tweens.
 - `TpTweenSpec.Tween` are the individual specs in a list of specs in the asset.

Create methods return a long integer which is the ID of the tween.

Note that tweens are relative: for example, if you tween position, the value provided as an endpoint is the `CHANGE` that you want and **NOT** the absolute position.

For example: Creating a tween that changes Position with an EndValue of `Vector3(2.2,3.5,0)` doesn't move the Tile's sprite to `(2.2,3.5,0)`. It *offsets* the position by 2.2 units in the X direction and 3.5 units in the Y direction from the tile's position in the tilemap: it's relative to the tile.

So if the tile is at `(0,0,0)` for this example, the tile's sprite would move to `2.2,3.5,0`. If the tile were at `(10,15,0)` then the tile's sprite moves to `(10+2.2, 15+3.5, 0+0) = (12.2,18.5,0)`.

And that makes a lot of sense in this context: the tile itself doesn't move while you're tweening its color or transform. You're actually affecting the *Tilemap* and how it displays the tile's sprite. Nothing in the tile's data changes at all.

Similarly, if affecting rotation, the change is relative to the existing rotation of the tile's sprite, and if affecting scale, the change is relative to the existing scale of the tile's sprite.

Color tweens are a little different: you're tweening between the color when the tween is launched and an absolute end color specified in the `CreateTween` method call.

In a practical application, `CreateTweenFromSpec` is the most useful. Most likely you'll have several tweens that you'll use repeatedly for many tiles.

Rather than have individual tiles have all the fields for specifying tweens (as in `TweenTileExample`) you use a Tween specification from a `TpTweenSpec` asset; this can be seen in `TweenTileExample2`.

You can also use specs from this asset when creating sequences.

Ignoring `CreateDelayTween`, the Create methods launch the tween immediately. If you need a delay, create a two-element sequence with a `DELAY` as the first tween in the sequence.

Operations on Tweens

- A running tween can be cancelled with `KillTween`.
- Get a reference to a running tween with `GetTween`. However, since these are pooled items, do not hold this reference outside a local scope or you **WILL** get memory leaks.
- Get an unpooled copy of a running tween with `CopyTweener`. Note that this is **NOT** pooled but is a **COPY** of the running tween at that point in time.
- Get the `ToString` of a running tween without affecting the instance with `GetTweenInfo`.
- Tweens are auto-deleted if their parent tile or the parent tile's `Tilemap` become null.

Create a sequence.

- Use `CreateSequence`, add tweens using a `Create` method, and use `PlaySequence` to run the sequence.
- Use `CreateSequenceFromSpec` and use `PlaySequence` to run the sequence.

`CreateSequenceFromSpec` uses the tween specifications in a `TpTweenSpec` asset and makes a sequence out of the tweens.

- An array of indices into the List of Tweens in a `TpTweenSpec` asset can be used if you want only specific tweens from the list to be used.
 - Don't do this unless really needed: if you change the `TpTweenSpec` asset later you have to update the array.
- If the array is null (the default) then the entire list of Tweens is used.

`CreateSequence` and `CreateSequenceFromSpec` return a long integer which is the ID of the sequence.

If you're not using `CreateSequenceFromSpec`:

- Create a sequence using `CreateSequence`.
- Use `CreateTween` or `CreateDelayTween` and supply the ID of the sequence in the method call.
 - When the sequence ID is supplied, the created tween is NOT immediately run, but its data structure (a `TpTween` instance) is added to the sequence.
- Use `PlaySequence` to start the sequence.

Operations on Sequences

- Delete an unused sequence with `DeletePendingSequence`.
- Kill a running sequence with (wait for it...) `KillRunningSequence`.
- Sequences are auto-deleted if their parent tile or the parent tile's `Tilemap` become null.

Reset

Reset the tweener with `ResetTweener` and the Sequencer with `ResetSequencer`. Both release all active tweens and those which are included in the sequences' lists of tweens.

Observe

- Use the Tween Monitor (menu item). This opens an Editor window which displays all of the information about running tweens and sequences.
 - If you have many tweens running this can slow down your game.
- Use the Services Inspector (menu item). This shows all running services (including this one).

Spawner Service

Introduction

The TilePlus system uses pooling as much as possible and reasonable. Most of this activity is behind the scenes, but there is one pooler available for general use as a Service: SpawnerService.

SpawnerService is based on the TpSpawner : ScriptableRuntimeService<TpSpawner> class.

Use

SpawnerService can be used for painting tiles, which is a specialized use primarily for TpAnimZoneSpawner. In that case there's no object pooling, which applies only to Prefabs. That use isn't further discussed here.

A tile can use SpawnerService to spawn pooled prefabs, with preloading. That feature is used by TpAnimZoneSpawner and TpAnimatedSpawner. You can examine those tiles' code to see how it works.

You can use SpawnerService from any code, not just tiles. Just use the SpawnPrefab method shown below:

```
public GameObject? SpawnPrefab(GameObject? prefab,
                                Vector3      position,
                                Transform?   parentTransform = null,
                                string        parentNameOrTag = "",
                                bool          searchForTag = false,
                                bool          keepWorldPosition = true)
```

The first two parameters are obvious.

parentTransform: If provided then the spawned Prefab is parented to that transform. If the value is null a search is done for the parent using the parentNameOrTag string. If that string has a value, then one of two things occurs:

- searchForTag = true: Use GameObject.FindWithTag to locate a GameObject to use for a parent.
- searchForTag = false; Use GameObject.Find to locate a GameObject to use for a parent (slower).

If neither search provides a parent, or if `parentNameOrTag` is null or white space then the spawned prefab is unparented (which may be what you want).

TLDR; want parent? Provide one or use the `parentNameOrTag` search string.

The `keepWorldPosition` parameter is passed to `Transform.SetParent` if a parent is provided or located. If true, the parent-relative position, scale and rotation are modified such that the object keeps the same world space position, rotation and scale as before (from the Unity documentation).

Preloading

Preloading a pool can be done with `Preload()`.

You can check to see if a Prefab is preloaded with `IsPreloaded()`.

Reset

Resetting the Pooler (normally not needed) is done with (wait for it) `ResetPools()`.

Notes

For maximum efficiency, it's suggested that you add the `TpSpawnLink` component to the root `GameObject` of your Prefab. If it isn't present, then the call to `SpawnPrefab` will add that component automatically so that the Prefab can be tracked.

`TpSpawnLink` provides optional timed auto-destroy. The `OnTpSpawned` and `OnTpDespawned` methods can be overridden in derived classes to provide customized activity when the Prefab is spawned or despawned. The `DespawnMe` method can be used to despawn a Prefab from some other entity.

If you do override anything that's `virtual` please be sure to properly call the base class methods from any overridden methods even if they appear to do nothing.

It's a simple but functional system that can maintain the complete lifetime of a Prefab's placement from a pool and a despawn with automatic return to the pool.

The System Info and Services Inspector windows show information about the pool status. Try out the Collision demo to see the pooling in action.

PoolHost

Normally the pooler does not attach the pooled and/or preloaded prefabs to a parent GameObject, which can make the hierarchy look messy.

If this bothers you, head over to the Project folder Plugins/TilePlus/Runtime/Assets and drag the Tpp_PoolHost prefab into your scene. This prefab has an attached component which sets DontDestroyOnLoad so that the prefab persists between scene loads. The pooler looks for a GameObject with this specific name and will parent non-spawned and/or preloaded Prefabs to this GameObject.

GameObjects are set inactive when held in the Pooler after preloading or despawning but are set active when fetched from the pool.

Messaging Service

Please see [Messages](#)

If you're going to use the Messaging Service you need to know about Events. The two are related, see: [Events](#)

TilePositionDb Service

Introduction

The TilePositionDb Service monitors Tilemap callbacks to update a small internal dataset.

The dataset keeps track of which tile positions are occupied on the specific Tilemaps that you tell it to monitor.

TilePositionDb can optionally keep track of tiles with scaled sprites ($x,y > 1$).

Given this information, use query methods to see if a position is occupied by another tile even if the tile sprite is enlarged; for one or several Tilemaps at once.

This is useful, for example, as part of a custom Grid Graph for the A* Pathfinding Project (code available on request).

TilePositionDb supports time-varying size and position, such as you'd have when tweening a TPT tile's scale. BUT: ***It doesn't support accurately tracking rotated sprites.***

If a sprite is rotated and Warnings are active (Checkbox in TilePlus.Config) then a warning message is printed to the console the first time that a rotated sprite is encountered. To avoid console spamming this warning occurs only once per run-session.

Use

Please check out the [TpInputActionToTile](#) component. This shows how to use position DB via the TpActionToTile Scriptable Object.

Here's an example that can guide how to use it yourself:

```
var posDb =  
GetServiceOfType<TpTilePositionDb>();  
if  
(!posDb)  
  
    return;
```

```

//Add a map to monitor
posDb.AddMapToMonitor(m_Tilemap); //you have a field with this reference.

posDb.HandleScaledSprites = true; //recognize and keep track of scaled sprites (this is the
default).

//-- Optional, not needed if there are no active tilemaps when the game starts.
//-- this is generally only needed in the Editor since there can be a scene already present
//-- when you click the PLAY button.
//it may be the case that the scene has tilemaps with existing tiles
//at startup. Hence, add all existing positions 'manually'
//duplicates are rejected in the AddPosition
method.
var n =
tilemap.GetUsedTilesCount();
if (n !=
0)
{

tilemap.CompressBounds();
    foreach (var pos in
tilemap.cellBounds.allPositionsWithin)
        posDb.AddPosition(tilemap, pos); //ignores positions without TPT tiles.
        posDb.ForceUpdate(); //Force an Update so that the internal data structures are
refreshed.

}

```

What is this?

A mini-database of occupied positions on Tilemaps.

This was originally intended for use with the 'Chunking'/Layout mode of TilePlus Toolkit, although there's no restriction on other uses. It's also optionally used by TpActionToTile, TpInputActionToTile, and other code where one wants to locate tiles on a tilemap from a mouse position.

Since it also (optionally) keeps track of tiles whose sprites are larger than one Tilemap unit, you can locate tiles EVEN if what you click on is the sprite outside of the tile's single position. It also properly handles sprites where the position has changed and the sprite's bounds no longer overlap the tile's native sprite bounds. But for efficiency, a sprite's rotation is ignored.

You can see this in the Oddities/Jumper demo where some of the tiles have sprites bigger than one unit.

When using the Chunking system: even if you have a huge Tilemap only the parts of it within the camera view (plus optional padding) are loaded into the Tilemap. Hence, the HashSets in this Service never get that large.

HOWEVER, this assumes that you do not use this subsystem to monitor dense maps such as a 'floor' or 'ground' Tilemap where almost all positions are filled. One should use colliders or some other approach for dense maps.

Sparse tilemaps make sense to use with this Service, so roads, obstacles etc.

Tiles are added and deleted from the data structures in this Service by the `OnTilemapTileChanged` callback (from Tilemap).

Any tiles which are on the specified Tilemaps are added or deleted from a HashSet.

But that only covers one single position. If you have tiles with sprites that are offset by the transform-position of the tile sprite then that doesn't work.

Scaled or position-shifted sprites are handled automatically IF the `HandleScaledSprites` property is set true. That's the default value. If you don't want the extra overhead involved in keeping track of these type of sprites then set the property `false` *immediately* after the first request for this service.

After the first `OnTilemapTileChanged` event from the Tilemap the property value is locked.

Since evaluating scaled sprites involves some computation and Tilemap access (to get the actual sprite transform) these are cached and evaluated when this SRS is updated. That occurs at `EndOfFrame` (using the Awaitable pump) or at `PostLateUpdate` when using the `PlayerLoop` pump.

Note that `TileBase` tiles like `Rule Tile` and `AnimatedTile` cannot have scaled or position-shifted sprites since these tiles don't have a transform. They appear internally only at their actual Tilemap position. However, this isn't much of a limitation: use `TpAnimatedTile` or `TpFlexAnimatedTile`; and `Rule Tiles` (this one assumes) never have enlarged sprites.

When using the layout system you handle initialization via a callback (see `Chunking Demo`).

Otherwise you use `AddMapsToMonitor` (several overloads). After initialization the only way to completely change the setup is to use the `.ResetPositionDb` method or terminate/restart the service.

Once initialized, when a tile is added or deleted the internal HashSets are updated.

Use `PositionOccupied(Tilemap, Vector3Int)` to test if there's a tile at a position.

Important

Tiles are deleted from the internal dataset during the Tilemap callback. However, as mentioned earlier, additions are cached and evaluated near the end of the frame.

This means that there's always at least a one-frame delay for additions to the internal data.

The `PosDBRefreshPerUpdate` property

The Tilemap callback can dump an enormous amount tiles into the update queue. This property can be used to control how many updates are processed each time that the PositionDb updates. This defaults to `int.MaxValue`.

For example, if you move a 10,000 tiles there will be 10,000 entries in the update queue. If `PosDbRefreshPerUpdate` is left at it's default value execution will block until all the updates are processed.