Services

Dynamically loadable Scriptable Object Services for the TilePlus system

TilePlus Services

Overview

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

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

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).
- `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.
- `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.
    - 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.

Initialization

It's important to avoid race conditions for Services which require initialization. This can occur if more than one source (such as a Component) use the Service.

For example, in the TopDownLayout demo, both the GameController and the TpInputActionToTile components both use the TilePositionDb service. The PositionDb is designed to reject duplicate 'monitored' Tilemaps (i.e., those maps where the PositionDb keeps track of tile positions). Other control parameters should only be modified by one entity. In this demo, that's TpInputActionToTile.

Note that this warning ONLY applies to Services which require initialization. For TilePlus, that's only the TilePositionDb

Convenience Properties

Since the Messaging, Tween, 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.
- TweenerService returns a handle to TpTweener.

These use `GetServiceOfType<T>` and will auto-start the service if not already running. These services do not require any initialization.

Note that the TpTweener Service is also 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 property is an alias for TpLib.TweenerService.

- A Monobehaviour or other code can use TweenerService (via TpLib.TweenerService) to create GameObject tweens.
- Creating Tile Tweens requires a TilePlusBase instance as part of the method call.

Please refer to the Tweener documentation for more information.

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, 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 (it's complex enough as it is).
- Depending on your viewpoint about Singletons, you may or may not want to avoid this approach.

Profiling

Services that require updating can set this up via the IScriptableService interface, discussed below. The update invocation for each service is automatically wrapped with Profiler.BeginSample and Profiler.EndSample regions with the Service's name. When using a modified PlayerLoop for updates, look for the TpLib section.

Messaging

Use `ServiceMessage(ObjectPacket packet)` to send messages to running Services. This is implemented with two IScriptableService members (see below).

Use a pooled (wrapped with `using`) or an an unpooled instance of `ObjectPacket` and set the Command property to a particular command string. Other properties of can be used to send whatever information the target understands (this is up to you).

Using the instance's properties, you can add one or more of:

- a UnityEngine.Object
- any instance of any c# class
- a string
- an int
- a boolean value
- all of the above.

Services indicate which command strings are supported.

When ServiceMessage is used, the packet's Command is examined and is used to determine which Services get the message. This is very flexible and open-ended. It's not used by the TilePlus system at all, but is used in the game that this was developed for, which is largely based on services.

More Grisly Details

The IScriptableService interface

IScriptable is an interface that lets a SRS specify how it wants to receive Update events (or not) and if it should persist across Unity scene changes.

If a service DOES NOT implement IScriptableService it inherits the default Interface properties or; no Updates and auto-kill on scene changes.

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.

Persist Across Scene Changes

If the `PeristThruReload` interface property is EXPLICITLY implemented and returns true then this Service will not be destroyed when a Unity scene changes.

IScriptable messaging

As mentioned above, messages can be sent to Services via ObjectPackets. The packet's Command property controls what Services are sent a particular message.

To do this, implement `AcceptableMessages` and `MessageTarget.` AcceptableMessages is a property that returns an array of strings representing the Commands that the MessageTarget can accept.

AcceptableMessages is examined only when the Service starts.

- Returning new string[1]{"XYZZY"} would mean messages with XYZZY as the command are accepted.
- Returning new string[2]{"XYZZY", "ABCDEF"} would mean messages with XYZZY or ABCDEF as commands are accepted.
- Returning Empty array (the default value of the property) == NO messages accepted.
- Returning single-item array with first item == `"__ALL__"` => ALL messages.
- AVOID: Returning new string[2]{`"__ALL__"`, "ABCDEF"} would mean ALL messages + all ABCDEF messages
  - ABCDEF messages would be sent to the service twice! (so don't do this).

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.

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).

Collidables

A `Collidable` is a GameObject with a TpSpawnlink component on the main or root GO. The TpSpawnLink's IgnoreCollider flag must be false, the root GO must have a Collider or Collider2D component.

The GO is checked for this condition when its spawned from a prefab or when it is despawned. If this condition is true then the reference is added to or removed from an internal HashSet of these 'Collidable' Game Objects.

This is a way to keep track of spawned GameObjects which are not part of an Archived Tilemap in a TileFab. See how this is used in the Layout demo.

Events

- OnCollidableObjectSpawned: when a collidable GameObject is spawned.
- OnCollidableObjectDespawned: when a collidable GameObject is despawned.

Service Messages

The Spawner Service doesn't accept any Service Messages. However, it sends the following messages right after invoking the CollidableObject events.

- When Spawned: Command = "SPAWNER-ADD-PREFAB" and the ObjectPacket's UnityObject property has a reference to the spawned GameObject.
- When Despawned: Command = "SPAWNER-DEL-PREFAB" and the ObjectPacket's UnityObject property has a reference to the GameObject that will be despawned.
  - As usual: it makes no sense to cache this reference as it will become inactive.

Services that want to get these messages should add these two strings to their `AcceptableMessages` property return value.

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) and position-shifted sprites.

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: First, initialization.

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

     return;
```

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

//-- 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);
  posDb.ForceUpdate(); //Force an Update so that the internal data structures are
refreshed.

}
```

Querying - here's the most basic query:

```
public bool PositionOccupied(int            mapIid,
                             Vector3Int     position,
                             out Bounds     scaledSpriteBounds,
                             out Vector3Int tilePosition,
                             Vector3Int?    ignoreThisPosition = null,
                             bool           ignoreScaledSprites = false)
```

This method takes the

- instance ID of a Tilemap
- the position you're testing

and optionally:

- scaledSpriteBounds: if what's found is a tile with a scaled sprite then this is its bounds.
- tilePosition: if what's found is a tile with a scaled sprite then this is the tile's actual position. The tile may not be congruent with the sprite.
- ignoreThisPosition: if two tile sprites overlap, this position is ignored.
- ignoreScaledSprites: this is much faster if you're sure there aren't any scaled or position-shifted sprites on your Tilemap.

For `ignoreThisPosition`, if adjacent tiles have overlapping sprites overlap then the return value would be true. But it's indeterminate which sprite (of the two) would have its tile's position returned. If you want to exclude one of those positions use this parameter. Check out the `JumperTile` TPT tile script for an example.

There's a similar Query called `PositionOccupiedForWorldCoordinates.` This useful for certain uses. See the TpActionToTile script for an example.

If you'd like to test on multiple Tilemaps, use

`public bool PositionOccupied(int[] mapIids, Vector3Int position, Vector3Int? ignoreThisPosition = null )`

There are also shortcuts for really simple (and of limited use) "pathfinding" - these mostly exist for really simple uses.

`public bool IsPathBlocked4Way(int mapId, Vector3Int position, TpTileUtils.DirectionType4 direction, uint distance=1)`

`public bool IsPathBlocked4Way(int[] mapIids, Vector3Int position, TpTileUtils.DirectionType4 direction, uint distance = 1)`

`public bool IsPathBlocked8Way(int mapId, Vector3Int position, TpTileUtils.DirectionType8 direction, uint distance=1)`

`public bool IsPathBlocked8Way(int[] mapIids, Vector3Int position, TpTileUtils.DirectionType8 direction, uint distance = 1)` These test for any blockages in one or more Tilemaps in a given 4- or 8-way direction, and for a certain distance.

They're fairly inefficient if you have many Tilemaps.

If you want to use something like A* PathFinding Project, the PositonOccupied methods are the basis of a Grid Graph plugin.

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 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.

Since evaluating scaled sprites involves some computation and Tilemap access (to get the actual sprite transform) these are cached and evalutated 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.

Properties

`PosDBRefreshPerUpdate`

The Tilemap callback can dump an enormous amount of data 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.

The side effect is that updates to the PositionDb's internal data may take more frames to be completely updated.

`MinScaleOffset` and `MinPositionOffset`

MinPositionOffset: Determines the minimum position shift for a tile sprite to be added to the set of position-shifted or scaled sprites.

MinScaleOffset: Determines the minimum transform size change > 1,1 for a tile sprite to be added to the set of position-shifted or scaled sprites.

WARNING

Take care that only one entity affects these properties. It's best to set these once and not have multiple scripts changing these values.

Multiple scripts can add Tilemaps to be monitored. However, it's best to avoid having multiple scripts removing such Tilemaps unless there's some coordination.

Merging

The `Merge` method can be used to combine the 'occupied position' data from multiple tilemaps into one HashSet. This can be handy to have a lookup table that can be used repeatedly rather than using `PositionOccupied` repeatedly.

However, it can be slow - don't update it until you need to. Updating it every Monobehaviour Update is a bad idea. If you need updating that frequently you should use Tilemap colliders instead or stick with `PositionOccupied` instead.

But for a turn-based game where your user does some input and you have a number of on-screen entities that need to move, Merge can be used once and the resultant HashSet can be used to determine if the entities can move to particular locations.

Note that **even if** none of your tiles ever move or have their sprites tweened *but* you are using the Layout system you should be aware that the PositionDb is updated whenever new tiles are loaded or unloaded via the layout engine. Here, you can call `Merge` right after a Layout pass.

HOWEVER: be aware of delays caused by PosDbRefreshPerUpdate, which may delay complete updating for several TpLib Update events. The `PendingUpdates` property can be used to see how many updates remain in the queue.

In the Top-down layout demo, see the `OnPlayerHasMoved` event handler to see where that demo does a layout pass.

TpTweener Service

See Tweener Service