TilePlus Toolkit Programmer’s Guide

Contents

What is TilePlus Toolkit?

TilePlus Toolkit (TPT) comprises a set of new Tile classes, a new Brush, and support libraries. Using the Brush is optional, but it adds some features that are especially useful when working with these new tiles as well as normal Tiles. There's also an alternate Palette tool "Tile+Painter".

Top Features:

- You don't need to know the address of a TPT Tile to access its data.
- Each TPT Tile has its own instance data. Any Unity-Serializable type is fine.
- Access by Type, Interface, Tags, with filtering. Simple messaging and Events systems.
- All the tile instance data is stored in the Scene. No external data store required.
- Simple Async-based timed callbacks can be used in TPT Tile code.
- DOTween can be used to animate TPT Tile transforms and Colors.
- Multiple-layered Tilemaps can be preserved in total or subject to a Grid Selection, then painted on a Tilemap or loaded at runtime.

Basic Principle

A TPT Tile clones itself when it painted on a Tilemap. To put it another way, *the TPT Tile asset that's placed at a grid position on a Tilemap clones itself, and that clone replaces the placed Tile asset*.

What does "cloned" mean? Normally, when you use a brush to place a tile on a Tilemap or when you programmatically add a tile to a Tilemap, the Tilemap copies the tile's asset reference, the sprite, the color, the Transform matrix, and the TileFlags from the tile asset into the internals of the Tilemap component.

When the Scene is saved, these items from the internals of the Tilemap component are what's stored in the Scene file – nothing else from the Tile itself is saved in the Scene file. This is well-known behavior at this point.

For tiles that only need to present a sprite or perhaps a simple animation, the implementation is very sensible. You wouldn't want to store a multitude of copies of the same tile information for the same, repeated tile when all you're doing is displaying static sprites or animations.

If you want to add additional serialized fields to a Tile subclass, you can, but the additional data isn't copied to the Scene file, it's only in the asset. You'll discover that the default Brush's Selection Inspector won't even show the added fields. If you were able to edit the fields, the source asset in your project folder would change, so every instance of the tile would be affected. Not very useful.

What this means in practice is that you must place Game Objects and/or Prefabs in the scene to create certain types of functionalities. This is a hassle because "workflow".

Once you place a TPT tile, a copy or *clone* of the tile asset is created and the reference in the Tilemap is changed to the clone. Since the Tilemap maintains the reference, the clone is saved with the scene.

The cloned TPT tile becomes independent of the project asset, so editing its fields doesn't affect the project asset. The Tile+Brush or Tile+Painter tool allows you to edit the fields using attributes that you place on fields that you want editable.

If the Tilemap gets deleted or a different tile is placed in the same location, the reference count for the clone goes to zero, and it's no longer saved with the scene and eventually gets garbage collected like any other object whose lifetime has expired.

Of course, this somewhat increases the size of a Scene file and memory use in a built application. But if you're creating functionality by adding Game Objects or Prefabs, the effect is similar. In any case, you would not have a substantial number of these types of tiles compared to those that just display an image or a simple animation.

Downside? Can't place Tilemaps with TPT tiles inside Prefabs. Well, you *can*, but there are some limitations as explained further down in this document.

You can set certain supplied TPT tiles to be the default tile used for drag-and-drop tile asset creation with the Palette. In Unity, see the *Preferences/Tile Palette/Create Tile Method* setting. They can also be created from the Assets menu or from the Assets dropdown in a Project window.

Why is this better?

The big advantage is that you can directly edit these new types of tiles in a Selection Inspector and not affect the asset in the project folder. A new brush, Tile+Brush, lets you do this easily. The Tile+Painter is a combo Painter and Editor that's an alternative Palette with additional features for TilePlus tiles.

The Tile+Brush and Tile+Painter Selection Inspectors (they're the same) incorporate features that allow display and live editing of your added serialized fields using some simple Attributes that you can place on the fields; and ToolTips will be visible as well. Custom IMGUI editors can also be created right inside a TPT Tile class.

You can even save your modified tile to a new asset file. Note: these features are only available if the Tile+Brush is the palette's active brush or you're using Tile+Painter. While the Editor is in Play mode some or all fields' display is inhibited. See the Attributes section of this document for more information.

You don't need to use the Tile+Brush to use TPT tiles. The default brush works fine for tile painting, but of course the enhanced Selection Inspector won't be visible since the Selection Inspector is provided by the Brush editor code.

Check out Tile+Painter. This is a combination single-tile Painter and general-purpose Tilemap editor. You can inspect placed tiles irrespective of which brush is in use, sorted by Tilemap, with optional filtering by Type, i.e., the Tile class and/or by their tag. Most of the Selection Inspector functionality is duplicated when examining TPT tiles, and the active brush is irrelevant.

All TPT tiles have the inherent capability to simulate their runtime activity, although you need to customize it for any custom tile classes that you create. For example, when you use the Selection Inspector to examine a placed TpAnimatedTile, you'll notice a ► (Start Simulation) button in the Inspector toolbar. If you click on that button and there are multiple frames in the animation sequence, the animation will be simulated in the Scene view. It's a bit wonky at times, as the execution depends on an update callback from the Editor, which can vary in frequency somewhat. But you can preview an animation without running your game.

Note: the simulation uses EditorApplication.update for timing and the update speed can vary widely, depending on the setting: *Preferences/General/Interaction Mode*. On your author's system, the rate with the *Default* setting is approximately 250 per second but can range from 30 to over 500. For predictable simulation, set the interaction mode to *Monitor Refresh Rate* which is about 60 per second. You can see the refresh rate at the bottom of the Configuration Editor.

Simulation Timeout is how many EditorApplication.update events occur before the simulation ends. Simulation speed slows down the process by skipping update events; for example, when the slider is placed at the far right, the number of skipped events is 1. Moving it to the left increases the number of skipped update events and slows down the simulation. There's no particular reason that you need to follow this same exact method in your own tile code.

This simulation capability can be customized for different situations. For simpler cases, place a custom attribute on any simple parameterless method in your code and the Selection Inspector will show a button to invoke the method.

Also included is a “TpFlexAnimatedTile” tile that’s like TpAnimatedTile, but with the addition of multiple animations selected from a custom animation clip asset, and the ability to choose the default animation or change it at runtime.

Nomenclature Note

Note: the word “**tile**” refers to any tile. **Tile** refers to *Unity* tiles. **TilePlus or TPT** tiles are *TilePlus Toolkit* tiles, and **TileBase** tiles (those that derive from the TileBase class rather than the Tile class) are specialized tiles such as Rule tiles.

TilePlus tiles are often referred to as **TPT** tiles for brevity.

- tile = any tile (Tile or TPT tile).
- Tile = Unity tile or any tile subclassed from Tile (including all TilePlus tiles)
- TilePlus/TPT = TilePlus Tile
- TileBase = any tile directly subclassed from TileBase (aside from Tile itself); e.g., Rule tiles.

If this seems confusing, *well*, it does get complex. It’s important though because Tiles have sprites, transforms, and a Color. TileBase does not have these properties. Tile+Painter adapts to this by having quite simple plugins to support obtaining these properties, as best as possible, from custom TileBase-subclassed tiles. Plugins for 2D Tilemap Extras’ Rule and Animated tiles are provided for your convenience.

The plugins allow Tile+Painter to show previews for TileBase-subclassed tiles.

You really don’t need to think about this too much unless you want to create your own TileBase-derived tiles.

Architecture

Basics

TPT tiles have three possible states.

- **Asset**: The tile is an asset in a Project Folder. Painting it changes state to Clone.
- **Clone**: The tile is present in a Scene. You can save it to the Project as an Asset.
- **Locked**: The tile is part of a TpTileBundle (Bundle) asset in the Project Folder.

If you Inspect a TPT asset in the Project window and open the "TilePlus Basic Settings" foldout, you'll note that the *State* field is **Asset**.

When a TPT tile is added into a Palette it remains in the **Asset** state. You can see this by selecting it in the Palette Window – the Brush Inspector's *Tile Info/Name* field displays **[Asset]**.

When the tile is painted on a Tilemap, the state changes from **Asset** to **Clone**. You can see this by picking the tile using the Palette Select tool and looking at the "TilePlus Data" section's first line.

The only exception is when you use the Pick function of a Brush or Tile+Painter to copy and paste tiles. In that case, the copied tile is already a clone, and the pasted tile is the same clone, which is not what's wanted. The system recognizes when this happens (in-editor only and Play mode if you copy/paste programmatically), makes a new clone of the tile and paints it in the pasting location. This is implemented by TpLib.`CopyAndPasteTile`.

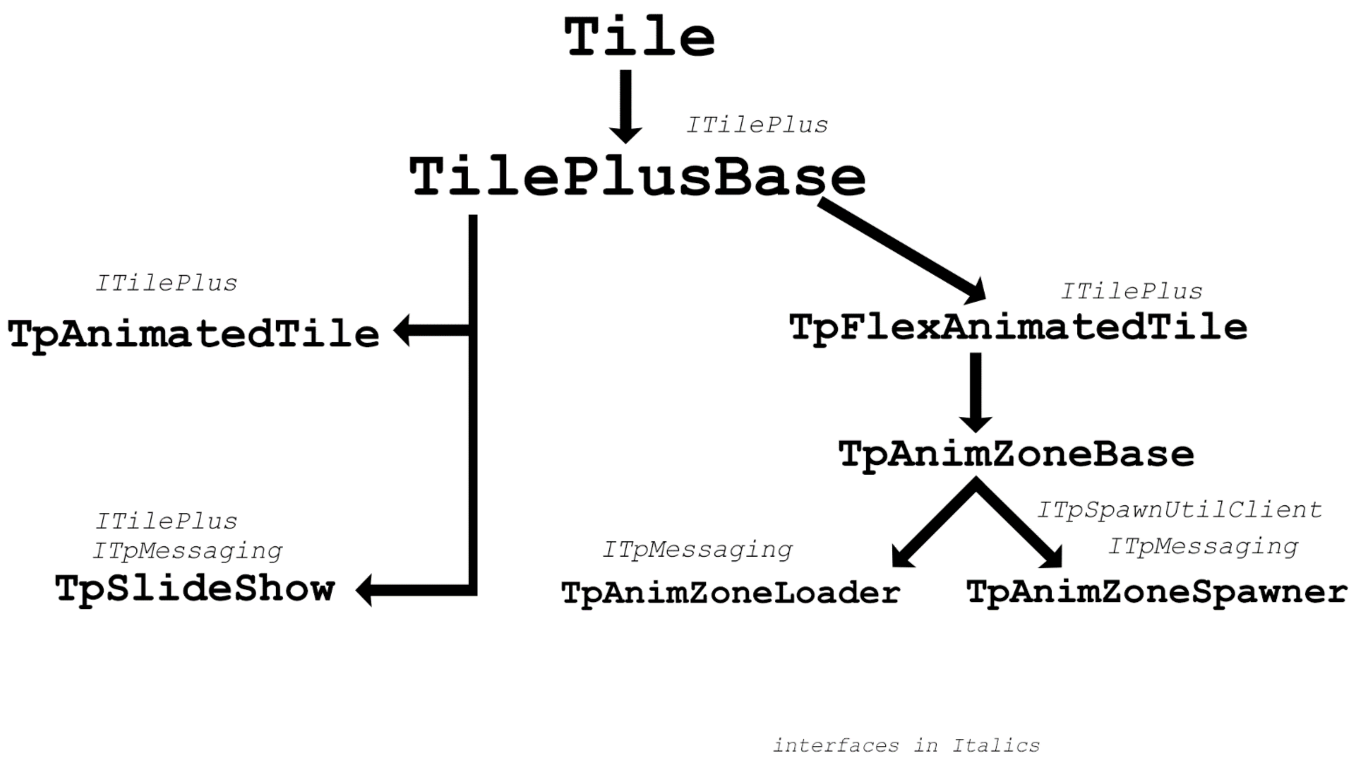
You can use a Selection Inspector toolbar button to save a TPT tile from the Scene to the Project as a Normal, cloneable TPT tile asset. The selected tile isn't affected. This is handy for prototyping: you can customize a TPT tile right in the Scene and save it either for backup or as a template for further use. A simple versioning scheme adds version numbers to the saved assets.

Locked TPT tiles are the same as ordinary tiles in the sense that any modifications to the Locked tile painted on the Tilemap affect the tile asset in the Project. They're only seen as sub-assets of an asset created by the *Make TileFab or Prefab* menu command. If a Locked tile is present in a tilemap, it converts into a Clone tile at runtime.

It's ***strongly recommended*** that nothing in TilePlusBase be tinkered with. Overridden, sure. Just be sure to call the base class in your override.

There are two Assembly Definitions for TilePlus Toolkit: one for the runtime and one for the Editor.

TilePlus Toolkit Tile Class Hierarchy



This diagram shows both inheritance and which interfaces are implemented in each subclass of Unity's Tile class.

TilePlusBase is the base class for all the TPT tiles.

TilePlus Tile Asset Varieties

- TpAnimatedTile: A simple animated tile based loosely on 2D Tilemap Extras' AnimatedTile.
- TpFlexAnimatedTile: An Animated tile that allows choice of animations from an asset.
- TpSlideShow: A tile that can change its sprite via code using a sequence from an asset.

The above tiles are all described in the User Guide.

- TpAnimatedZoneBase: Base class used to create rectangular zones on a map.
- TpAnimatedZoneLoader: Subclass of the above used for loading Tilemaps.
- TpAnimatedZoneSpawner. Subclass used to spawn prefabs or paint tiles.

These specialized tiles are more complex, and don't work "out of the box" without custom coding.

Please note: If painting TpAnimatedTile or TpFlexAnimatedTile or subclasses (TpAnimatedZoneBase, TpAnimatedZoneLoader, or TpAnimatedZoneSpawner) via script, see the User Guide FAQ entry "Animated tile not animating."

DOTween

Believe it or not, you can use DOTween within a TilePlus Tile's code. It's useful for tweening the Tile's transform or its Color.

However, DOTween is designed for GameObjects; at least in the sense that it can recover when the target of the tween is null or becomes null.

One simple step handles this problem: add a field for DOTweenAdapter like this:

```
    private readonly DOTweenAdapter dtAdapter = new();
```

Add this to `OnDisable: dtAdapter.OnDisableHandler();`

Then, anywhere you set up a sequence or tween, register it with:

`dtAdapter.AddSequence(sequence);` or `dtAdapter.AddTween(tween);`

Ok, so DOTween can be used in a TilePlus tile. Why would you want to do that? In the Tile context, DOTween is only useful for modifying the tile's transform for scale, rotation, or position; or perhaps the sprite's color/transparency.

You create your tweens or sequences as usual and use AddTween or AddSequence to add the tween or sequence to the per-tile controller instance of DOTweenAdapter as shown above.

The DOTween adapter instance uses the tween or sequence OnComplete callback (or hijacks it if you already have an OnComplete callback set in the tween or sequence) to delete the tween or sequence from the controller when it's complete. Doesn't do much, but if you delete the tile while a tween or sequence is playing then the controller cleans up (Kills) all running tweens and sequences, avoiding all sorts of unpleasantries. Note that for this to work properly, the OnComplete must execute; hence, be careful when "killing" tweens manually: use the option that allows completion, even if your tween does not itself have an OnComplete callback.

The OnDisableHandler kills all remaining tweens automatically when the TilePlus tile is deleted or destroyed.

See the "DOTween demo" for an example of how to use this feature. Ensure that you've added the **TPT_DOTWEEN** definition to the Player settings' Scripting Define Symbols list before trying to use it. Oh, and you must install DOTween too…

Note: newer versions of DOTween add DOTWEEN to the Scripting Define Symbols list. The DOTween adapter and the Collision demo program have been updated to accept either symbol.

If you don't intend to delete, copy/paste, or cut/paste (or otherwise move around tiles which internally would be using one or more tweens in a running app) **and** you won't ever be removing an entire Tilemap or a scene where tweens might be running, then you don't need to use the adapter. Rather, you can just use DOTween as usual. You may wish to examine the DOTweenDemo to see how to access a tile's transform components when using DOTween. There are no built-in DOTween extension methods for dealing with Tile transforms.

Notes:

- A tile transform isn't the same as a GameObject transform.
- Changing the transform components (position, rotation, scale) should affect the Tilemap and not the Tile. Changing the transform field of a Tile at runtime won't do anything unless the Tile is refreshed (which is a bad way to do it). The small library class TileUtil.cs has convenient functions for accessing Tile transforms to make coding simpler.

- Similarly, if you want to tween Color, affect the tilemap and not the color field of the tile.
- The transform’s position value doesn’t move the tile. Rather, it moves the effective position of the sprite relative to the tile. For example, setting the position to Vector3(1,1,0) will display the sprite at an x=1, y=1 offset from the tile position.
- Changing the sprite scale can affect how the Tilemap Renderer calculates the Chunk Culling Bounds. See the User Guide FAQ entry: **Tiles Disappear When Camera Moves.** This has nothing to do with TPT tiles.
- If results are not what you expect,
    - Check the Preferences tab of the DOTween Utility Panel, especially:
        - Recycle Tweens
        - AutoPlay
    - Examine the tile with the Selection Inspector and ensure that the initial Transform and/or Color settings in the tile are reset. You can also do this in your StartUp method as is done in the DtDemoTile.cs.

Specialized Tiles

TpAnimZoneBase

This is a base-class TPT tile that can be used for defining a trigger area. Its purpose is to define an area on the Tilemap, specifically, a BoundsInt. Since it's a subclass of TpFlexAnimatedTile, its sprite can be animated. If you don't desire animation, just don't add a SpriteAnimationClipSet.

Setting the zone size works in one of two ways, depending on the setting of the ModifySprite toggle. In either case, the ZonePositionAndSize fields let you move the position of the zone and the size of the zone.

If ModifySprite is checked (on), then as you change the area positioning and size, the tile's sprite is transformed to encompass the trigger area. Since the tile controls the transform, transform modification in the TilePlusBase section of the inspector is unavailable. The VisualizeSprite button will highlight the boundary of the zone for a few seconds when clicked.

If ModifySprite is not checked (off), then the sprite isn't modified as you change the zone size. The VisualizeArea button creates a persistent outline around the zone. The outline will be cleared if you click VisualizeOff or change the selection.

Public fields:

- TileSpriteClear: Clear the sprite when painted, when the game runs, both, or not at all. The default for this tile is ClearOnStart. Note that that the sprite will be invisible, and you won't be able to see the trigger area if this is set to ClearInSceneView or ClearInSceneViewAndOnStart.
- Position, Size: Adjust the position and size of the boundaries of the Trigger zone. Unavailable when Lock All is checked.
- ModifySprite: Controls whether the tile's sprite is affected by zone size changes.
- UseTrigger: If this is true then the tile's MessageTarget (see TpLib.SendMessage) method will trigger an event if the passed-in position is within the Trigger zone. Note that the display of this field may be inhibited in subclasses, e.g., TpAnimZoneLoader.
- Lock Position and Size: Prevent changes to X, Y, or Z position or Z size; or Lock All. (Editor-only).

Public properties:

- ZoneBoundsInt: Get or Set the BoundsInt which describes the trigger area.
- ZoneBounds: Get a Bounds based on ZoneBoundsInt.

Note that the bounds size can never be less than 1,1,1.

If you change the size of the sprite and there's another tile in the same area it might be obscured by the transformed sprite (or vice versa). If that's a problem, you can adjust the transparency in the Color field (TilePlusBase section of the Selection Inspector) or change the Tilemap Renderer's Sort Order setting.

Plugins/TilePlus/Runtime/Textures/TriggerZoneSprite can be used for the sprite for this tile, but you can use any sprite. Note that the sprite won't appear if you change the TileSpriteClear to *Clear In Scene View*. If you don't want the trigger zone to appear in Play mode, set TileClearMode to *Clear On Start*.

The other action button for this tile is Reset Zone. Reset Zone does what you'd expect and resets the boundary to one tile space. This button does not appear when the LockAll toggle is checked.

At runtime, this tile is completely passive aside from animation. But with TpLib you can, for example, get a reference to every tile of this Type and use the trigger zones to do something when a playable entity enters or leaves a trigger zone. It's very general-purpose, but how you use it is naturally bespoke to your own project.

However, one common case might be to load more tiles to open a new area. That leads us to TilePlusAnimZoneLoader, a subclass that embeds information used to load TileFabs.

Note that the Tilemap Renderer will sometimes cull enlarged sprites. There's a FAQ in the User Guide regarding this.

TpAnimZoneLoader

This class is a subclass of TpAnimatedZoneBase and inherits its fields and editor appearance. It's used to provide information to make it easy to dynamically load archived Tilemaps from TpTileFab assets, which are created by the **Tools/TilePlus/Prefabs/Bundle Tilemaps** command.

Public fields include those from TpAnimatedZoneBase and its superclasses, and these additional fields.

Public fields:

- TileFab: A TileFab archive from a Project folder.
- UseZoneManager: Optionally use a Zone Manager for detection of already loaded TileFabs.

Zone Managers and their uses are discussed in "Advanced TileFab Use".

Internally, an offset from the tile's position is used to place the TileFab, this is set interactively.

There are three action buttons for this TPT Tile: Reset Zone, Visualize, and Preview. Reset and Visualize are inherited from TpAnimatedZoneBase. Preview loads the tiles into a preview tilemap or maps, if there are multiple TpTileBundle assets referenced by the TpTileFab asset. The preview is active until you click Preview again or the Editor's Selection changes. When preview is active, you can change the loading offset and the preview area will change position.

Loading or previewing tiles depends on there being compatible Tilemaps that are named or tagged in such a way that the system can identity which Tilemap to use. This information is embedded in the TileFab asset when you create it. If you need to change it just edit the asset's **TileAssets** section in the Project window.

TpAnimZoneLoader can have animation. For example, you might want to animate the trigger zone area. Be careful of the TileClearMode in that case: set it to Ignore.

The tile *does not* automatically load Tilemaps at runtime. Rather, you send a message to it via the TpLib SendMessage methods. As configured, it expects the message to contain a Vector3Int describing a position. If the position is within the Zone bounds, the tile uses TpLib's PostTileEvent method to post a trigger event. See the TopDownDemo's TdDemoGameController MonoBehaviour component for an example. TpLib's LoadTileFab method handles all the details of the loading for you, so it's easier than it sounds!

TpAnimZoneSpawner

This can be used to spawn prefabs and TPT tiles, using assets with lists of prefabs or tiles. This tile uses a built-in static library class called **SpawningUtil** which in turn depends on an interface called **ITpSpawnUtilClient**. You can also use SpawningUtil directly with ordinary method calls.

Prefabs can be unparented or parented to a Scene Object by using the GameObject name or tag. SpawningUtil has built-in prefab pooling (using Unity built-in pool classes).

Tiles can be painted on the same tilemap or on a different tilemap which you specify using a GameObject name or tag, or a reference. This tile *can* respond to a SendMessage containing a Vector3Int describing a position. If the position is within the Zone bounds, it will spawn prefabs or paint tiles as you've configured it. Alternatively, you can spawn/paint via code using the instance methods SpawnPrefab or PaintTile if this approach doesn't suit you.

You can use either a TpTileList or a TpPrefabList asset (or both) to specify which tiles or prefabs (or both) to use with this tile. Public fields include those from TpAnimZoneBase and its superclasses, and these additional fields.

Public fields:

- TileList**:** a reference to a TpTileList asset provides a list of TPT assets along with the spawn position for each. Spawn position is one of eight positions immediately surrounding the tile position on the Tilemap, a random position in the Zone, or the painting tile's position if the painting is to a different Tilemap.
- PrefabList: a reference to a TpPrefabList asset. Note: preload amounts are set there.
- SpawnMode: spawn prefabs or paint tiles in the order that they appear in the TpPrefabList asset.
- PositioningMode: spawn or paint using the setting in the assets, or force to the Spawner's position, the contact position in the Zone, or somewhere Random in the zone.
- PaintingTilemap: Tile painting only: Optional reference to alternate Tilemap used to paint tiles.
- PaintingTilemapNameOrTag: Tile painting only, string with name or tag of alternate tilemap.
- ParentingMode: Tiles painting only, use the same tilemap to paint tiles (can't paint at the Tile's position), or use a provided reference (Painting Tilemap), or use a Tag to locate a Tilemap's GameObject using PaintingTilemapNameOrTag, or use a string to Find a Tilemap's GameObject using PaintingTilemapNameOrTag.

Note that this tile inherits the field m_UseTrigger. If that's true, then a TpLib trigger event is posted when the position passed-in by SendMessage is within the trigger zone.

When doing Tile painting you have several configuration options using PositioningMode, PaintingTilemap, PaintingTilemapNameOrTag, and Parenting mode. When painting tiles it's important that the tile which is doing the painting is not painted over. Furthermore, since the tile occupies a position on a tilemap, you probably don't want it to appear as an obstacle to a Player or NPC walking through the tile's trigger zone (TpZoneBase or TpAnimZoneBase).

All these controls exist so that you can paint the spawner on a different Tilemap from the one that you use for pathfinding and character movement.

Paint one of these tiles on Tilemap A and use it to paint tiles on Tilemap B. When you do that, the tile needs to know which Tilemap to use. That's done by setting ParentingMode appropriately and filling-in correct values for PaintingTilemap or PaintingTilemapNameOrTag.

Prefab Pooling in TpAnimZoneSpawner

SpawningUtil provides GameObject pooling based on a UnityEngine.Pool.ObjectPool<GameObject>. This is a relatively new feature in Unity 2021.1 and later.

When a TpAnimZoneSpawner tile instance's StartUp is invoked and if SpawnMode is set to RandomPrefabs or PrefabsInOrder **and** if the TpPrefabList has any prefabs where the PoolInitialSize is nonzero, the tile will ask SpawningUtil to preload prefab instances. This only occurs on the first StartUp and will not occur if you change the TpPrefabList asset during runtime.

Note that preloading takes time, as each prefab must be instantiated, so choose the preload sizes carefully. If there’s no preload, the pool expands automatically as prefabs are instantiated and spawned.

Normally the pooler does not attach the pooled and/or preloaded prefabs to a parent GameObject. 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.

The pooler will automatically add a component of type TpSpawnLink to spawned prefabs if it doesn’t already exist. TpSpawnLink can also despawn the prefab after a timeout.

Other TilePlus Toolkit Assets

Aside from TPT tile assets, there are several other assets you can use:

- TpTileBundle: The asset created when you create a prefab from Tilemaps.
- TpTileFab: Another asset created when you create a prefab from Tilemaps.
- TpPrefabList: A group of prefab references for TpAnimZoneSpawner tiles.
- TpTileList: A group of TPT tile asset references for TpAnimZoneSpawner tiles.
- TpSlideShowSpriteSet: A group of sprite references for the TpSlideShow tile.
- TpSpriteAnimationClipSet: A group of sprite references for the TpFlexAnimatedTile tile.

All except the first two can be created from the Assets/Create/TilePlus menu.

TpTileBundle and **TpTileFab** are assets created when you use the **Tools/TilePlus/Prefabs/ Bundle Tilemaps** menu command. TpTileBundle is referenced by a or TpPrefabHelper component placed on the Tilemap's GameObject, and by the TpTileFab asset. TpTileFab assets are used with TpAnimZoneLoader tiles.

There's a longer discussion about TpTileBundle and TpTileFab in the next section.

TpPrefabList is a list of TpPrefabSpawnerItems. Each item has the following fields:

- Prefab: The prefab to spawn
- Parent: Name or Tag of a GameObject to parent the spawned Prefab to. Optional.
- UseParentNameAsTag: If a Parent is specified, interpret as a Tag if this is checked.
- Position: Position of the prefab. Can be left at Vector3.zero.
- PositionIsRelative: If checked, the Position value is relative to the tile grid position.
- KeepWorldPosition: If checked, keeps the prefab's world position relative to tile grid position.
- PoolInitialSize: The pool preload size.

TpTileList is like TpPrefabList, but for TPT tiles.

- Tile: The tile to paint
- PaintPosition: Where to paint it. An Enum selects where to paint.
  - Around the position of the tile doing the painting.
  - At the position of the tile doing the painting if the target Tilemap is not the same.
  - A random position within the painting tile's Zone (see TpZoneSpawner/TpAnimatedZoneSpawner)

TpSlideShowSpriteSet has a list of TpSlideClips. Each clip has the following fields:

- Name: Name of the slide show
- WrapAround: Stop at the last sprite or wrap around to the first.
- StartIndex: The starting sprite for this slide show
- Sprites: A list of sprites to display.

TpSpriteAnimationClipSet has a list of TpAniClips. Each clip has the following fields:

- Name: Name of the Clip Set
- DefaultTileIndex: When animation isn't running, which tile to use for the static sprite.
- AnimationSpeed: Speed of animation relative to that set on the Tilemap component.
- OneShot: Stop the animation at the end of the sequence or repeat.
- RewindAfterOneShot: Rewind to the first frame after a one-shot animation ends.

- Note: currently, this is done with a timer and is not 100% accurate.

Please note that when adding a new clip to the asset: AnimationSpeed will be 0, will cause a runtime warning if OneShot is true. Internally, a value of 1 is used if AnimationSpeed is zero, which may produce unintended results, including incorrect one-shot timeouts.

If you're using Odin Inspector, AnimationSpeed is initialized to 1.

TpLib

Behind the scenes there are several static classes, the most important are TpLib and TpLibEditor. TpLib keeps track of TilePlus tiles. TpLibEditor uses hooks into the editor to invisibly aid workflow. When the term '***TpLib'*** is used in this document it includes all the following static library classes.

TpLibEditor includes code and assets in these folders, none of which are included in a build:

- TpLibEditor
    - Scene and Prefab management, Marquee, and line drawing.
- TpEditorUtilities
    - Deferred Editor-mode tasks, and the entry points for most of the Tools/TilePlus menu items.
- TpPrefabUtilities
    - This is used when creating TileFabs or Tilemap Prefabs.
- TpConditionalTasks
    - Simple task manager for delayed callbacks after a condition is met.
- TpIconLib
    - Used to fetch editor icons. Create Sprites from Texture2Ds. Caching applies to both.
- TpImageLib
    - Image processing utilities
- TpPreviewUtility
    - Different varieties of Previews for Tilemaps.
- ImGuiTileEditor, BasicTileInfoGui, InspectorToolbar, and SelectionInspectorGui, TileCustomGui
    - These format and display tile information when the Selection Inspector or Tile+Painter are used.
- Painter Folder
    - The Tile+Painter window code, libraries, and UI Toolkit implementation.
- SysInfo
    - A simple debugging/system info window.
- Template Tool
    - See Advanced TileFab use
- Config
    - The various Scriptable Singletons used for configuration and for Painter's Favorites list.
- Brush
    - The code for the Tile+Brush.

TpLib must be included in a build. Runtime TpLib includes these static classes:

- TpLib: Tile 'database' and support.
- TileUtil: Utility methods for BoundsInt, Matrix4x4, and Tile transforms.
- TpEvents: TilePlus Event system for tiles.
- TpMessaging: TilePlus messaging system for tiles.
- TpEditorBridge: Bridge to access certain editor-assembly Unity and/or TilePlus methods (not available in builds).
- SpawningUtil: Handles spawning tiles and Prefabs.
- The Lib subdirectory, which includes optional features:
    - TileFabLib: Supports TpTileBundle and TileFab loading and reloading.
    - TpZoneManager: Builds on TileFabLib to support chunked TileFab loading/unloading.
    - TpZoneLayout: Basic client for TpZoneManager.

When your game starts in an editor session or in a built application and a scene with Tilemaps containing TPT tiles is loaded, or if you add a TPT tile programmatically, the TilePlusBase basic functionality registers the tile in TpLib.

TpLib has runtime functions that can be used to delete, replace, or move tiles rather than dealing directly with the Tilemap. You don't have to use them unless you want to use TpLib or are painting tiles at runtime (which includes any operation where tiles are added or deleted to or from a tilemap).

Most of the demo examples use TpLib. All the functionality related to Editor session workflow is eliminated in builds by inclusion in TpLibEditor or some conditional compilation and isn't discussed in this document.

Pro tip: Calling methods in TpLib or adding callbacks to TpLib events during a Monobehaviour's Start or Awake methods is generally a bad idea. This is because the TPT tiles may not have all had their StartUp methods invoked yet. *GetParentGrid* is the only significant exception. If you want to do these sorts of things in Start, you can make it an IEnumerator or async and wait a few frames.

Introduction

It’s a database of sorts, although that’s perhaps a bit grandiose. The most important data structure of the class is a nested dictionary.

```
private static Dictionary<int, Dictionary<Vector3Int, TilePlusBase>> s_TilemapsDict;
```

The outer dictionary’s key is an int and represents the Instance Id of a Tilemap. The inner dictionary is a mapping from Tilemap positions to TilePlusBase instances. Hence, it’s a dictionary of Tilemaps to tile positions and their tile instances.

Other significant data structures:

- A dictionary that maps TilePlus tile tags to tile instances.
- A dictionary that maps TilePlus tile Types to tile instances.
- A dictionary that maps Interfaces to TilePlus tile instances.
- A dictionary that maps TilePlus tile GUIDs to tile instances.

Pro Tip: For runtime use, you can optimize the size of these dictionaries with TpLib.Resize. This should be done as soon as possible after startup and before any Scenes containing Tilemaps are loaded. If that’s not possible, use the ‘rescan’ parameter of this method to rescan all Tilemaps.

One of the key features of TilePlusBase tiles (and their subclasses) is that the Tile instance always knows what Tilemap it’s part of and always knows its position on the Tilemap. The position information isn’t static: if you move a tile using the palette tools or methods in TpLib, the position is updated. To be clear, this is performed entirely within the tile and has nothing to do with which brush is used. The only exception is when you use a Brush’s Pick function to copy/paste a TPT tile: this is discussed in the *Limitations and Notes* section.

When a tile is placed or moved, it calls a method in TpLib to register itself in the various data structures. This also happens when you start your app or load a new scene.

Again, to be clear, TpLib does not keep track of tiles which do not subclass TilePlusBase. That class implements the ITilePlus interface and should be the base class for your own tile classes. Almost everything in TilePlusBase can be overridden if needed.

There’s a zipped API reference in HTML format in the Documentation folder. The following section discusses some selected concepts.

TpLib Main Functionality

For a complete list, see the HTML help file included in the Documentation folder.

`IsTilemapRegistered`	Returns true if the specified Tilemap has TPT tiles.
`GetTile`	Returns a TilePlusBase instance for a particular map and position
`HasTile`	Returns whether a given Tilemap has a TilePlus tile at a particular position.
`GetTilePlusBaseFromGuid`	Returns the TPT instance for a specified GUID.
`GetAllTiles<T>`	Get a list of all tiles of type T
`GetAllTilesOfType`	Creates a list of TPT instances of a specific Type in a specific Tilemap or all maps.
`GetAllTilesWithInterface`	Creates a list of TPT instances with a specific interface. Overloads allow optional filtering by Type and specific tilemap.
`NumTilesWithInterface`	Returns the number of tiles with a specific interface.
`GetTilesWithTag`	Creates a list of all TPT tiles with a specific tag string in a specific Tilemap or all maps.
`GetFirstTileWithTag`	Returns the first TPT from GetTilesWithTag
`GetAllTagsUsedByTileType`	Returns the number of tags in use for a particular Type of tile.
`GetAllTypesInDb`	Returns a list of all Types of TPT tiles.
`IsTilemapLocked`	Tests if a Tilemap contains any Locked tiles.
`GetAllPositionsForMap`	Returns a HashSet<Vector3Int> of all occupied positions in a map. Optionally include non-TPT tiles
`GetParentGrid`	Searches up the hierarchy from a given Transform for a Grid component.
`DelayedCallback`	Call-back a method after a delay in milliseconds (see “Deferred Callbacks” appendix)
`SceneScan`	Scans all scenes in all Tilemaps. Use this after loading a new scene.

The GetAllTilesWithInterface methods are particularly useful for selecting tiles. There are two overloads, one fills the output List with TPT tiles as TilePlusBase instances. Here’s a trivial example:

```
var tiles = new List<TilePlusBase>();

TpLib.GetAllTilesWithInterface<SomeInterface>(ref tiles);
```

The other override is often more useful: here, the output list is filled with instances of the tiles which match the interface type. This saves much casting.

```
var tiles = new List<SomeInterface>();

TpLib.GetAllTilesWithInterface<SomeInterface>(ref tiles);
```

Each of these has an optional filter delegate. For this second one, the Func has input parameters of an TilePlusBase instance as T (i.e., the interface you specified) and a TilePlusBase instance.

Similarly, the NumTilesWithInterface method is handy to determine if there are any tiles which implement a particular interface. Here's an example. The ITpInteractive interface is not included in TPT.

```
var numTilesHere = TpLib.NumTilesWithInterface<ITpInteractive>(tpb =>
                          tpb.TileGridPosition == gridPos
                          ? NumTileWithInterfaceFilterResult.PassAndQuit
                          : NumTileWithInterfaceFilterResult.Fail);
```

Here, we're trying to see if there are ANY tiles that match a grid (i.e., tilemap) position. The filter allows the NumTilesWithInterface method to quickly exit after it finds at least one match. Hence, the result in numTilesHere will be either 0 or 1.

Useful properties include:

Note: 'Use with Caution' means that you should never change the contents. Reading from them is perfectly OK. However, do not hold any tile references outside of a local method scope or you will create a memory leak.

`TpLibIsInitialized`	Returns true when the static TpLib class is initialized (this happens automatically, you don't need to do it yourself).
`TilemapsDictionary`	Returns a reference to the entire TilemapsDict. Use with caution!
`TaggedTilesDict`	Returns a reference to the TaggedTiles Dictionary. Use with caution!
`TileTypes`	Returns a reference to the TileTypes Dictionary. Use with caution!
`TileInterfaces`	Returns a reference to the TileInterfaces Dictionary. Use with caution!
`GuidToTile`	Returns a reference to the GuidToTile Dictionary. Use with caution!
`GetAllTagsInDb`	Returns an array of all tile tag strings.
`TilemapsCount`	Returns an integer representing how many Tilemaps are being monitored.

If you add a tile by writing it to a Tilemap, the tile will automatically be added to TpLib right after placement as soon as the tile's Startup method is invoked.

If you want to use TpLib in your code, and you're removing or moving tiles at runtime, it needs some help from you!

`DeleteTile`	Removes a tile from TpLib for a specified Tilemap and position and deletes the tile from the Tilemap. An overload allows using a TilePlusBase instance instead of the Tilemap and position (since those are usually obtained from the instance anyway).
`DeleteTileBlock`	Removes all tiles (TPT tiles and Unity tiles) from an area (BoundsInt). An overload allows deleting blocks of TilePlus tiles via a list or array.
`ReplaceTile`	Does the same thing as DeleteTile and then adds a specified TPT tile in the same position.
`CutAndPasteTile`	Cut/Paste: Moves a tile to a new position, deletes it from the original position.
`CopyAndPasteTile`	Copy/Paste: Copies a tile and pastes it in a new location on any Tilemap. Clones the tile so a new instance is created.
`RemoveMap`	Deletes the specified Tilemap from the internal data structures. Does not actually delete the actual Tilemap or any of its tiles. Use before deleting a Tilemap.

Events and Communication

TpLib has general-purpose messaging and Event functionality. **Messages** get sent to tiles, and tiles can post **events**.

Messages functionality uses the TpMessaging static class and events functionality uses the TpEvents static class.

Messages can be sent to tiles; for example, to notify a tile of the player's position. Tiles can send events to subscribers of a .net `event`; for example, after receiving a message about the player's position the tile can post an event that the tile's position intersects that of the player.

`SendMessage`	Sends a message to a tile. Several overloads.
`OnTileEvent`	An event that Monobehaviours or other scripts can subscribe to be notified of events that a Tile initiates via a call to `PostTileEvent`.
`PostTileTriggerEvent`	Allows a tile to notify subscribers to OnTileEvent that a tile wants it or them to do something.
`PostTileSaveDataEvent`	Allows a tile to notify subscribers (usually only one) to OnTileEvent that a tile wants it to save data from the tile.
`AnyMatchingFilteredEvents`	Is there at least one event that matches an EventType and satisfies a filter Func?
`ClearQueuedTileEvents`	Clears the queue
`RemoveEvent`	Removes an event
`NumSaveEvents`	Property: Returns the number of Save events
`NumTriggerEvents`	Property: Returns the number of Trigger events
`AnySaveEvents`	Property: Returns true if there are any queued SaveData events
`AnyTriggerEvents`	Property: Returns true if there are any queued Trigger Events.
`HasDeferredTileEvents`	Property: Returns true if there are deferred Tile events.
`DeferredTileEvents`	Property: The deferred tile events list. Use with caution!

Using the built-in messaging and event functions requires implementation of one or two interfaces, depending on what you're trying to do.

- `ITpMessaging` specifies methods required for communication and events.
- `ITpPersistence` specifies methods used for saving and restoring data.

Both interfaces are generic and have type specifications for data sent to a tile and for data read from a tile. This is covered a bit later.

SendMessage

TpMessaging.SendMessage is used to send messages to tiles. Using method overloads, the message can be sent to all tiles of a particular Type or Interface, or to all tiles with a particular tag. Tiles which need to be recipients of SendMessage must implement **ITpMessaging**. All these overloads have filtering delegates. These can be useful, for example, to only select tiles within a certain boundary area.

As can be seen from the **ITpMessaging** interface, the message is sent to a method called **MessageTarget**. Depending on what you're trying to accomplish, the method **GetData** can be used to retrieve data that the tile sets up in response to receiving a message. In the interface specification, GetData is a default interface method and only needs to be implemented if you want to override the default behavior (returning null).

Tile Events

TpEvents.PostTileTriggerEvent and **TpEvents.PostTileSaveDataEvent** are used by tiles to send messages to subscribers of OnTileEvent, who receive an instance of the event type when the event is invoked. TpEvents saves the tile instance that sent the event, and the event subscriber can retrieve it, and other cached events, with **TpEvents.GetFilteredEvents**. The cache can be cleared with **TpEvents.ClearQueuedTileEvents**.

The TpEvents.PostTileTriggerEvent and TpEvents.PostTileSaveDataEvent methods don't require implementation of any interface methods. Also, multiple cached events from the same tile instance are not cached since only the instance reference is cached and that would not have changed.

PostTileXXXEvent is meant for MonoBehaviour components as subscribers to OnTileEvent, although this is not enforced. As usual, ensure that OnDisable or OnDestroy unsubscribe from the events.

Save and Restore

A simple but powerful save/restore subsystem is built into TpLib. Tiles implementing **ITpPersistence** will have methods that can be used to get and restore data from and to individual tiles.

One viable way to do that is for a tile to use TpEvents.PostTileSaveDataEvent to indicate when it has data to save. For example, this action can be in response to a SendMessage from a scene component.

This is only a framework since what you save and restore is extremely specific to your project. The demo programs TopDownDemo and SaveRestore are examples of how to use this feature.

Types for these ***Interfaces***

ITpMessaging and ITpPersistence provide a general framework for messaging and game data persistence. The implementation is up to you, or you can ignore this optional feature completely. Both interfaces require you to implement the data structures that you want to use for messages and for saving/restoring data.

The important thing to note is that you can set up and use different data structures for both directions of data movement. Notation: We use T for the data type sent to a tile's MessageTarget<T> method, and TR for the data type returned from a tile's optional <TR>GetData method. Alternatively, you can use the same data structure for both directions, i.e., T and TR can be the same class.

The data structures must all derive from an abstract class called MessagePacket (in ITpMessaging.cs). MessagePacket is a totally empty abstract class. This can be useful if, for example, you want to use ITpMessaging to send a message to a tile but don't need to ever extract data from that tile when an event subscriber reacts. For that use case, a concrete empty class is provided for the GetData method: EmptyPacket.

If you have different data structures that you wish to use for different purposes, you can have multiple MessageTarget<T> and/or multiple <TR>GetData methods in your tile class, one for each type of data structure. This requires multiple interface specifications on the Tile class declaration. For example, the TpAnimZoneBase class is declared like this:

```
public class TpAnimZoneBase : TpFlexAnimatedTile, ITpMessaging<PositionPacketOut,PositionPacketIn>
```

with implementations:

```
        public void MessageTarget(PositionPacketIn sentPacket){…}
```

In this case, T is of type PositionPacketIn.

Please note that although not shown here, 'explicit' interface use should be used in the tiles. See the provided tile sources for examples of this. If not done, the compiler may complain since it won't be able to determine which MessageTarget to call.

The declaration below adds two more data types where T = Foo and TR = Bar:

```
public class TpAnimZoneBase : TpFlexAnimatedTile, ITpMessaging<PositionPacketOut,PositionPacketIn>, ITpMessaging<Bar,Foo>
```

and one would have to add these implementations:

```
public void MessageTarget(Foo inputParameter){…}
public Bar GetData(){…}
```

Each MessageTarget handles whatever actions are needed for the differing inputs, and the different GetData implementations provide any return data required. Again, please note that GetData is implemented by default in the interface specification, so you only need to override it if you're using it.

If this seems overly complex, the alternative for a general-purpose framework would be to use **object**, which introduces boxing/unboxing. The approach used here enforces static type safety, that is, the compiler will help prevent errors.

In Tilemap-based programs it is common to pass around Vector3Ints to denote a position on a tilemap. For this use case, a pair of identical simple classes is also provided: PositionPacketIn and PositionPacketOut. These are identical classes with just an internal position field, and both are provided to show that two different subclasses of the abstract MessagePacket class can be used with the same interface (ITpMessaging in this case).

For TilePlus tiles using ITpPersistence, strings are often sent to a tile to restore data (e.g., a string of JSON data). For this purpose, the StringPacketIn class is available.

There are two demo scenes that show how to use ITpPersistence: TopDownDemo and SaveRestoreDemo. TopDownDemo also shows how to use ITpMessaging. These demos are discussed later in this manual where use of these interfaces is explained more completely.

SendMessage overrides

There are several overrides of SendMessage, and those that operate on groups of tiles can fill a provided List with tile instance references that the message was sent to.

This override sends a packet of Type T to all tiles with a specified tag on a specified map. If the map is null, then all Tilemaps with TPT tiles are searched. The List `output` is filled with references to all the tiles that were messaged.

```
SendMessage<TR,T>(Tilemap map, string tag, T packet, List<TilePlusBase> output,
Func<TilePlusBase,bool filter = null)
```

If you don't want the output list to be used, just set it to null in the method call.

The List output can be used by the caller to read packets of type TR from the tiles' GetData method if that makes sense for your application. Example of this use case can be found in the Update method of TdDemoGameController, which is part of the TopDownDemo minigame project. The filter Func can be used to further define which tiles are messaged.

It should be noted that the search for tagged tiles is not done by reading every single tile from one or more Tilemaps and examining their tag fields. When TPT tiles' StartUp methods execute, the tiles register themselves with TpLib. One of the things that this process does is to create a mapping between tag strings and tile instances. This means that the search is very efficient.

This override sends a packet of Type T to all tiles of a specified Type on a specified map. If map is null, then all Tilemaps with TPT tiles are searched. If `typeSpec` is null but `map` is not null, then all tiles in the map are messaged. If `typespec` is null and map is null, then all TPT tiles are messaged. The List `output` is filled with references to all the tiles that were messaged unless it's set to null. Again, this process is faster than you might think since it uses an internal mapping from tile Types to tile instances.

```
SendMessage<TR,T>(Tilemap map, Type typeSpec, T packet, List<TilePlusBase> output,
Func<TilePlusBase,bool filter = null)
```

This override sends a packet of Type T to all tiles of a specified Type which also implements a specified interface `TI`, on a specified map. If `typeSpec` is null but `map` is not null, then all tiles in the map which implement interface TI are messaged. If `typespec` is null and map is null, then all TPT tiles which implement interface `TI` are sent the message. The List `output` is filled with references to all the tiles that were messaged unless it's set to null.

```
SendMessage<TR,T,TI>(Tilemap map, Type typeSpec, T packet, List<TilePlusBase> output,
Func<TI, TilePlusBase, bool> filter = null)
```

For this last override, the filter has two inputs: the tile instance as available through the interface and the base tile instance. This avoids casting in the filter delegate in most cases.

Here's a (fictional) example for this last, most complex override:

```
private List<TilePlusBase> output = new List<TilePlusBase>();
private PositionPacketIn packetIn = new PositionPacketIn(somePosition);

TpLib.SendMessage<BoolPacket,PositionPacketIn,SomeInterface>(null,null,packetIn,output);
foreach(var tile in output)
  Debug.Log($"Tile at {tile.TileGridPosition.ToString()} returned {tile.GetData.ToString()}");
```

What's being done here? We're sending a PositionPacketIn to all tiles that implement SomeInterface. TpLib.SendMessage sends the PositionPacketIn to all the found tiles and returns a list of these found tiles. We then iterate over these tiles and print out the BoolPackets returned from the GetData method of each tile.

Let's say we had an interface that had some boolean property XYZ that the tile had to implement and that you were interested in limiting the messages sent to only those tiles where XYZ is true, and the tile's position is within some BoundsInt. The filter Func could be written like this:

```
TpLib.SendMessage<BoolPacket,PositionPacketIn, SomeInterface >(null,null,packetIn, output, (ti,
tile) => {ti.XYZ && boundsInt.Contains(tile.TileGridPosition)}) ;
```

In General

The scheme of examining the tiles in `output` works well if the tile can compute a result (in this simple case, a true/false value) and make it available to GetData during execution of the tile method called by SendMessage. That way, valid data is returned from GetData. An example of this can be seen in TdDemoGameController where messages sent to tiles with the Spawner or RawZone tags have responses immediately available.

If the result isn't immediately available or if a Trigger/SaveData event is more appropriate, the tile can use TpEvents to post a Trigger or SaveData event. That will cause the C# event OnTileEvent to notify its subscribers. An example of this can be seen in TdDemoGameController where messages sent to tiles with the LoaderTag cause those tiles to post events.

If you're not using the GetData method for anything, just omit it. The default implementation returns null.

Inspectors

Selection Inspector (also seen in the Tile+Painter)

- Show internal data such as position and other descriptive fields.
- Show selected custom fields and properties in your tile classes using attributes.
  - Supported editable fields.
    - bool, int, float, string; int and float can optionally use slider controls.
    - Vector2, Vector3, Vector2Int, and Vector3Int
    - Enums
    - Object Fields like GameObject
    - Color fields
    - Asset files (open in an Inspector with a button-click).
    - Custom-coded IMGUI fields.
  - Properties are display-only.
  - Tooltips can be provided for Fields and Properties.
  - Buttons can invoke selected Methods.
- TilePlusBase class inspector section supports common customizations.
  - Color (affects tile and Tilemap)
  - Transform (affects tile and Tilemap)
  - Flags (affects tile and Tilemap)
  - Comma-delimited tile tags
  - Override tile's collider mode
  - Lockable tile name field
  - Tile's GameObject and the two associated flags.
  - When the Editor is in Play mode, the Tile animation flags (new in 2022.2) are displayed.
- Toolbar functions
  - Zoom scene view Camera to the selected tile.
  - Save a tile as a new Asset (very handy for prototyping!).
  - Open a normal inspector for the tile.
  - Open a normal inspector for the prefab if one exists (must be TpPrefab or subclass).
  - Refresh the tile.
  - Delete the tile.
  - Copy the tile GUID to the clipboard.
  - Simulate arbitrary tile functions; for example, previewing an animated tile's animation.
  - Hide/Show class names.
  - Collapse/Expand foldouts.

Brush Inspector

- Show information about the tile including a preview of a prefab if one exists.
- Show selected custom properties of the tile using attributes.

- Toolbar functions
    - Open Project folder asset in an inspector
    - Focus Project folder on asset
    - Open TPT (only) asset script in IDE (VS, Rider, etc.)
- Toggle controls
    - Overwriting control
    - Show/Hide toolbar

TpTileBundle and Prefabs

As mentioned in the User Guide, TpTileBundle (**Bundle**) assets are used to archive all or a section of a Tilemap. TpTileFab (**TileFab**) assets combine references to several Bundles, creating an archive of one or more Tilemaps; a multiple-Tilemap prefab of a sort.

In Bundles, cloned tiles in the scene are changed to Locked tiles and are stored as sub-objects of the **Bundle** asset. In other words, the cloned tiles are changed to Project folder assets and stored as part of the Bundle asset.

When normal Unity tiles are archived, just their transform, color, and flags settings are preserved, along with a reference to the tile asset in the project folder.

Prefabs which are children of the selected Tilemaps are archived by reference only: no asset copying occurs.

Note that Prefabs can be bundled from a Scene hierarchy or from a Project folder. This implies differences in how the transform information is archived:

- When bundling Prefabs from a scene, the stored rotation and scale are the rotation and localScale of the root GameObject of the Prefab in the scene hierarchy.
- When bundling Prefabs from a project folder, the stored rotation and scale are the rotation and lossyScale of the root GameObject of the Prefab in the Project folder.

Just to be clear about naming: **Tilemap Prefab** (with upper-case P) refers to a created prefab encompassing one grid with one or more child Tilemaps.

It’s encouraged to use a different folder each time you create **Bundles, TileFabs,** or **Tilemap Prefabs**, although this isn’t enforced.

You can use the TpBundleLoader component to load a single **Bundle** to a Tilemap. Place the component on any compatible (same layout, etc.) Tilemap’s GameObject and drag in the Bundle asset reference. Switch to Play mode and loading will happen automatically. Or you can click the Load button if you wish to test or perhaps restore a Tilemap. If you maintain references to several Archive assets in your Scene (to ensure availability in a build) then you can change the asset reference in TilePlusLoader and call the Load method of the component.

Similarly, you can use the TpFabLoader component to load a TileFab. This is discussed in the User Guide.

Here are two other ways to use **TileFabs**, plus there’s a lot more about them as you read on in this section.

- Paint a TileFab (or a single Bundle) using Tile+Painter. This is discussed in the Painter documentation.
- Load all or some of the Bundles in a TileFab on to different Tilemaps in your scene at runtime.

The second use listed above is particularly useful. Coders can use methods in **TileFabLib** to load one or more Tilemap sections dynamically from Bundles and TileFabs. For example, the TpAnimZoneLoader tile can be used to specify TileFabs to load when an entity passes into or out of a trigger zone. The TileFabLib static library has comprehensive methods to load TileFabs and Bundles, also see the **Advanced TileFab Use** document.

The Bundling Process

When you use the **Bundle Tilemaps** menu command, the bundling process asks you some questions, as outlined in the User Guide. Using this information, the bundler creates the TpTileBundle (Archive) assets: one for each Tilemap. Then creates copies of all the TPT tiles and adds them to the asset, saves any asset references to Unity tiles, and saves all the information required to rebuild a Tilemap, including position, transform, color, and flags.

Since the values for transform, color, and flags are often the same for large numbers of tiles (especially so for Unity tiles), these are stored in indexed look-up tables.

After all the Bundle assets are created, a TileFab is created in the same folder. References to the Bundle assets are stored in this new asset.

If you're making a **Tilemap Prefab** of a Grid and its child Tilemaps, the bundler creates a new prefab with the Grid and the child Tilemaps. These child Tilemaps are empty. The parent Grid of the Tilemaps has the TpPrefabMarker component added with a reference to the created TileFab. The TpPrefabMarker component loads up the empty Tilemaps when the Prefab is dragged into the Scene or otherwise loaded.

If there are any prefabs as children of the Grid or Tilemap GameObjects then the Project folder references to these prefabs are added to the TpTileBundle.

- When bundling Prefabs from a scene, the stored rotation and scale are the rotation and localScale of the root GameObject of the Prefab in the scene hierarchy.
- When bundling Prefabs from a project folder, the stored rotation and scale are the rotation and lossyScale of the root GameObject of the Prefab in the Project folder.

The completed Tilemap Prefab is placed in the folder that you selected earlier.

If you're archiving a single Tilemap, it's mostly the same process except that a Tilemap Prefab isn't created.

As mentioned in the User Guide, the Tilemap Prefab can be used just like any other prefab with one exception: it's "Locked," and you can't edit the Tilemaps. In Editor sessions, the system will try to stop you opening or making any changes to a Tilemap Prefab.

Why? A Tilemap Prefab contains copies of the Tilemaps, with no tiles. When the Prefab is placed in the Scene, TpPrefabMarker loads the tiles and prefabs from the TileFabs and Bundles. If you modify the Prefab in the Unity Editor by painting a TilePlus tile, then save prefab overrides, the new tile (being a scene asset) can't be saved in the Tilemap Prefab for reasons described earlier.

One could use the Allow Prefab Editing configuration option and paint normal Unity tiles or add GameObjects. That will preserve these changes in the Tilemap Prefab.

However, it's better to unpack the prefab, modify it, and generate a new Tilemap Prefab for maximum compatibility. You still have the original Grid and/or Tilemaps; unlike normal Prefab creation via Drag and Drop, the source isn't linked to the created prefab. If you ever need to recreate the original for editing, drag the Prefab into a scene, use the Unity menu command "Prefab/Unpack Completely."

Why does the **Bundle Tilemaps** command insist on a parent Grid to make a Prefab? If you make a prefab out of a Tilemap without a parent Grid, then instantiating it won't work properly unless you parent the instance to a Grid (this has nothing to do with TPT tiles). It will look fine in the mock scene created when you edit a prefab, because Unity adds the parent Grid for you in the editing Stage scene.

The created Archive asset contains all the converted, Locked tile assets. It's named using the Scene path; that is, the path from the tilemap up to its root GameObject; for example, MyTilemapName_MyGridName or MyTilemapName.MyGridName, depending on where it appears.

This is just for convenience so that you can have an idea of where the asset was created from.

If you inspect the asset, you'll notice that it has a few fields:

- Time Stamp: The creation time, in UTC time.
- Scene Path: The Scene path with dot notation.
- Original Scene: The name of the scene that the asset was created from.
- A flag that informs whether this Bundle was created from a Grid Selection.
- Various asset lists.

The Time Stamp, Scene Path, and Original Scene aren't currently used.

If you know that you've edited a locked Tilemap, then to ensure that there are no issues, use the **Tools/TilePlus/Prefabs/Unlocked Tiles test** after selecting a single Tilemap.   If there are clone tiles in a Prefab, you'll have issues.

What's the difference?

- A Tilemap Prefab instantiates Tilemaps and loads all the tiles from Tilefabs.
- A TileFab's tiles are loaded on to an existing Tilemap and never creates new Tilemaps.

It's a big difference! When a Tilemap Prefab is instantiated, a new set of Tilemaps is created and parented to a Grid. Therefore, if you instantiate 10 of these you have 10 independent sets of Grids and Tilemaps.

TileFabs require a set of compatible Tilemaps to exist in advance. So, if you have a scene with some Tilemaps and load a TileFab, the tiles load onto the existing Tilemaps. If you load the same TileFab 10 times the same tile would be written 10 times to the same locations.

The power of TileFabs comes when you consider that they can be painted anywhere on the Tilemaps. If you offset the placement of each of the 10 TileFab loads, they can be used to fill-in an area of a Tilemap. That can't be done with Prefabs. A bonus is that Bundles and TileFabs can be painted with Tile+Painter.

Tilemap Prefabs can be useful as a quick way to load part of a scene. But you can also do that with TileFabs, and they're way more flexible as they can be edited whilst being loaded.

To learn more about TileFabs, see the "**Advanced TileFab Use**" document.

Other Useful Classes

TpLoader is a component that can be added Tilemap's GameObject to load tilemap archives from a TpTileBundle asset. It has the following fields:

- TpTileBundle: the primary asset which holds all the tile assets and their locations.
- LoadOnRun: automatically load when app runs. Uses LoadPrefabs and ClearMap settings.
- Offset: offset every item in the archive when loading. Normally 0,0,0.
- DelayTime: delay before refreshing the Tilemap in Play mode or in a built application.
- When the Asset field is populated, a Load button will appear.
- LoadPrefabs: if this is checked then runtime-loading or the Load button also loads prefabs in the asset.
- ClearMap: if this is checked then runtime-loading or the Load button clear the Tilemap before loading.
- ClearPrefabs: if checked runtime-loading or the Load button clears ALL GameObjects parented to the Tilemap.

If the TpTileBundle was built with prefabs included, then clicking "Load" with the LoadPrefabs toggle checked instantiates the saved prefabs (or variant prefabs, if that's what you chose to save) if LoadPrefabs is true.

TpNoPaint is a component that can be added to a Tilemap's GameObject if you wish to force the Tile+Brush to disallow painting on a particular Tilemap.

Attributes

If you've done any Unity coding then you're probably familiar with Attributes by now, such as those having to do with serialization, or those affecting inspectors. The normal Unity Tile Selection inspector only displays the fields of the Tile class and no others. But we'd like to be able to view and modify fields and properties from TilePlusBase-derived tiles.

The Tile+Brush Inspectors and underlying library functions control what you see when using the Selection and Brush inspectors. It's a fair amount of IMGUI code. But normal humans should not have to struggle with that. So here in TilePlus land, we have several new Attributes which can be added to your code to display fields, properties, and even invoke methods or provide custom IMGUI code for functionality that the inspectors can't handle.

The Selection Inspector, the Brush Inspector, and the Utility Window use these attributes to display simple fields and properties. Fields can be modified, and the changes are saved in the Scene. Like any other change made to a scene, you need to save the scene to persist the changes. Normally, the Configuration Editor sets "Autosave" on and the save is done automatically for every change you make.

Your original TPT tile in your Project folder will never be altered. Note that field, property, and method declarations need to be `public` or `protected` to work with these attributes.

Attribute	**Affected**	**Description**
`[TptShowField]`	Selection Inspector and Utility window.	The types of fields that you can use this on are bool, int, float, string, Color, Vector2, Vector3, Vector2Int, and Vector3Int. Ints and floats can optionally use range sliders.
`[TptShowEnum]`	Selection Inspector and Utility window	Show Enums in a pop-up.
`[TptShowObjectField]`	Selection Inspector and Utility window	Objects such as GameObjects can be referenced with this attribute.
`[TptShowAsLabelSelectionInspector]`	Selection Inspector and Utility window	For a field or property, the value returned is displayed with ToString, so whatever you mark with this attribute must have a ToString() method or return a string.
`[TptShowAsLabelBrushInspector]`	Brush Inspector	Same as above.
`[TptShowMethodAsButton]`	Selection Inspector	Invoke a method, see below.
`[TptShowCustomGui]`	Selection Inspector and Utility Window	Create your own IMGUI function
`[Tooltip]`	Selection Inspector, Utility Window, Brush Inspector	This is a normal Unity attribute that you can find in the scripting reference. Note: Fields only.
`[Note]`	Selection Inspector, Utility Window	Add a note to a field or method. Note can be a static string or be provided by a property.

If a tooltip is provided as part of any attribute, then any normal [Tooltip] attribute will be ignored.

Display Order

The display formatter organizes the various attributes in the same order as the class hierarchy for the tile. For example, a TpAnimZoneLoader tile shows information from TpAnimZoneLoader, then TpAnimZoneBase, then TpFlexAnimatedTile, and then finally TilePlusBase.

In each section, Attributes are processed in the following order:

- Properties with TptShowAsLabelSelectionInspector or TptShowAsLabelBrushInspector.
- Methods with TptShowCustomGui
- Methods with TptShowMethodAsButton
- Simple fields with TptShowField
- Enum fields with TptShowEnum
- Object fields with TptShowObject field

Common Parameters

These parameters apply to all Attributes except TptShowAsLabelBrushInspector.

spaceMode	enum SpaceMode	Adds space or a line before or after the item, or both before and after.
showMode	enum ShowMode	Controls visibility: Always, only when playing, only when not playing, or use a property.
visibilityProperty	string	When showMode == Property then this is the name (in quotes) of the property is used to control visibility. If the property returns true, then the item is shown. Prepending a ! character to the name of the property inverts the property's value.

About "Show as Label" Attributes

- You can create a property just to show it as a label, based on other values in your class. See "Tips," below.
- The Brush Inspector is only affected by [TptShowAsLabelBrushInspector], hence, it can only show strings for both fields and properties. This is because what's displayed is information about the actual asset, and that should only be changed from the Project window and a normal Unity inspector.
- These attributes have a few extra parameters:
  - useHelpBox: show as an IMGUI HelpBox rather than a label.
  - splitCamelCaseNames: controls how field or property names are displayed.
  - toolTip: a Tooltip for the field or property.

About TptShowCustomGui

This attribute is placed above a method in a tile's code. The method should use IMGUI to display whatever information desired. Note that the method name itself is irrelevant.

The method should have a signature like this: `public CustomGuiReturn CustomGui(GuiSkin,Vector2)`

The GuiSkin may be useful in the method, and the Vector2 has button sizes. You may or may not need these.

If the access is not public or protected, then the method will be ignored. The return value from the custom GUI method indicates if any fields were modified, whether the tile should be refreshed, and whether to force a scene save. It's not advised to make custom GUI methods virtual, but if you do, please use '*new'* rather than '*override'* in subclasses as using *override* will make the generated GUI appear in the same foldout section as the overridden method.

About TptShowField for int and float fields

TptShowField's min and max parameters control whether ints and floats display editable fields or slider controls. If these are both zero (the default) then normal editable fields are displayed. If not, then a slider using those two values will appear. When the slider is visible and the slider values change, the configuration setting "AutoSave" is ignored, as that results in way too many scene saving invocations as the sliders are moved. The configuration setting "Allow sliders" can be unchecked to revert to editable fields. Using these parameters with other types of fields has no effect.

How to Trigger Methods

The TptShowMethodAsButton attribute is very handy for debugging. If you place this attribute right above a public or protected method, a button will appear which you can use to Invoke the method from within the Selection Inspector. The buttons won't appear if the tile is simulating. Note that if the method access is private then the method is ignored.

This comes with one restriction: the method must have no parameters, and you'll see an error message in the Selection Inspector or Utility window if the method has parameters. You can find numerous examples of this technique in the supplied TPT tiles.

About Object Field Attributes

The TptShowObjectField attribute requires a specification for Type. For example:

```
[TptShowObjectField(typeof(GameObject))]
public GameObject m_AGameObject;
```

This field can be used to drag Scene or Project references into fields for GameObjects, Components, Transforms, Prefabs, etc., when the Selection Inspector is focused on a TPT tile. It's not necessary to create a reference for the parent Tilemap of the TPT tile, it's already available as a protected field and a public property, just like the grid position of the tile.

Another important parameter for this attribute is allowSceneObjects. If this is true, the Object field can include scene objects. Set this parameter to false if the tile will be end up in a Prefab or archived in a TpTileBundle.

Using TppShowField with Unsupported Types

If the TppShowField attribute is used on an unsupported Type or custom Type, an Object picker is shown, or in some cases just the ToString() representation of the object. If the field represents a UnityEngine.Object (for example, a Scriptable Object) then an Open Inspector button will appear.

For example, TpFlexAnimatedTile uses this feature to display the name of the AnimationClipSet. Since that's a Scriptable Object asset, the button opens an Inspector where you can edit the list of sprites. Note that the list of sprites is a normal asset, so if you change anything it affects any TPT tile that uses the asset.

Field, Enum, Object special feature

The Field, Enum, and Object attributes have another parameter: updateTpLib. The default value is false. If set true and the field, enum, or Object is changed in the Selection Inspector or TilePlus Utility then

TpLib.UpdateInstance is called. This feature was created to accommodate multiple tags for a tile, and TpLib.UpdateInstance handles that automatically.

To support custom use of this feature, during Editor sessions, **TpLib.UpdateInstance** writes an array of field names that were modified to the overridable property **UpdateInstance**, although it's unlikely that there'd be more than one field name. You can use .Contains on the received array to see if there's a field you're interested in if you ever need to use this. Examples can be found in TpFlexAnimatedTile.cs and TpSlideShow.cs, where changes in the sprite and slide assets require some internal updates to the tile instance.

In Play Mode

The ShowMode value for attributes can be used to control what is displayed when in Play mode. However, some features automatically change in Play mode or when a TilePlus tile is Locked.

- Unsupported field editor buttons become unavailable.
- All fields display as IMGUI Helpboxes, except if you set ForceFieldInPlay = true
  - Note that ForceFieldInPlay is intended for development only and depending on the field Type may have unintended results including crashes.

If you have any concern that the display might affect your playing app, either ensure that the Tile+Painter window is closed, and that the Selection Inspector isn't focused on a tile ***or*** use the Configuration Editor (Tools/TilePlus/Special/Configuration Editor) to toggle the "Safe Play Mode."

Limitations and Notes

Unity3D 2022.3 LTS or later is required.

Changing code in subclasses may result in missing references in already-placed clone tiles. This is like what happens in the hierarchy when components' internal structure is altered. Adding serialized fields or altering the names of serialized fields will result in uninitialized fields in the placed clone tile. Note that the [FormerlySerializedAs] attribute can be used as a workaround if you use it while editing your code. Again, this is not an issue exclusive to TPT. Obviously, one can edit the placed clone tile to update the references if needed, but this can be time consuming.

When using the Palette's "Move" function, it's not possible to prevent overwrites. At least, I haven't figured out how to do that yet for every possible edge case.

This extension has mostly been tested with square tiles using a top-down XY view.

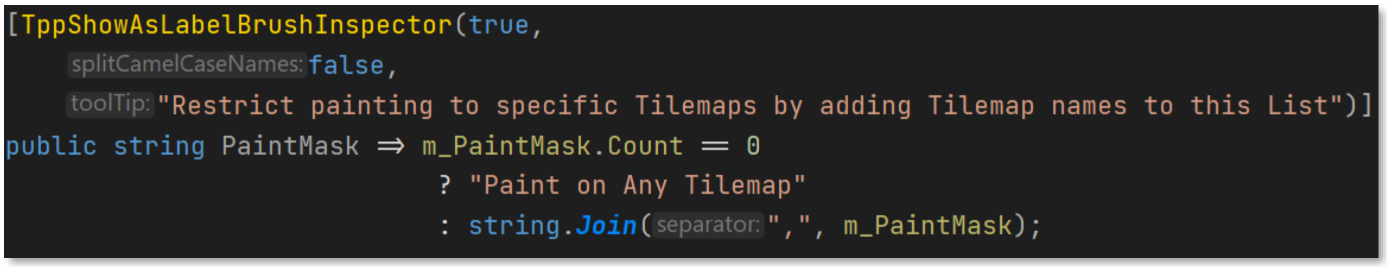
Picking/Copying TPT tiles

When you Pick one or more TPT tiles with a brush or with Tile+Painter and subsequently paint those tiles, what you're really doing is painting a copy of a cloned tile. TpLib catches this and re-clones the tile. Two identical tile instances would re-create the problem of asset modification since both tiles share the same instance data.

If you want to copy/paste a TPT tile in a running app, please use TpLib.CopyAndPasteTile. To replace a TPT tile use TpLib.ReplaceTile, and to move a TPT tile use TpLb.CutAndPasteTile.

Tips

The [TppShowAsLabel…] attributes allow you to display a property. This can be useful in many ways. Here's an example from TilePlusBase.cs that shows how the PaintMask property as shown in the Brush Inspector works. The PaintMask is a list, which would take up a lot of room to display. Below, the list is turned into a string for display.



```
[TppShowAsLabelBrushInspector(true,
    splitCamelCaseNames: false,
    toolTip: "Restrict painting to specific Tilemaps by adding Tilemap names to this List")]
public string PaintMask ⇒ m_PaintMask.Count == 0
                           ? "Paint on Any Tilemap"
                           : string.Join(separator: ",", m_PaintMask);
```

It's not a great idea to do anything too complex since there's repeated access during an Editor session when the Selection Inspector or TilePlus Utility window are in use.

How to Create New TilePlus Tiles

In most cases, the class that you want to subclass is TilePlusBase. But you might want to extend from one of the supplied tiles like TpFlexAnimatedTile, or TpPrefab. Many TilePlus Tiles' fields, properties, and methods are marked as 'virtual' so they can easily be overridden.

When creating subclasses of TilePlus tiles you should specify a namespace. Use *Tools/TilePlus/Configuration Editor* and add your namespace to the *Namespaces* field. Then click the *Reload* button. Namespaces are required for derived TilePlus classes. If the namespace isn't added to the system via the Configuration Editor, then the Selection and Brush inspectors will not display any fields or properties that you've decorated with TilePlus attributes.

Many of the properties in the sections demarked by #if UNITY_EDITOR are things that you can ignore. If you want fields or properties to display in the Tile+Brush Inspector, you can use attributes to display them. The TilePlusBase property "Description" can be used to show some text information about your tile in the Tile+Brush inspector.

It's helpful to examine the ITilePlus interface as it's not cluttered up with code, tooltips, attributes, conditional compilation directives, and so on.

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.

TileGridPosition always has the location of the tile in Grid coordinates, and TileWorldPosition always has the location of the tile in World coordinates.

Methods

Simulate can be implemented to use the Editor update event to do something. In TpAnimatedTile and TpFlexAnimatedTile it's used to show an animation preview.

TilePlusBase has two other methods that you probably want to override: StartUp, GetTileData, and ResetState.

Overriding **StartUp** must be done a specific way: a simple example can be found in TpAnimatedTile. Basically, you must call the base method as usual, but pay attention to the return value: if it's false your override ***must*** return false without doing anything else:

```
//this has to be the first thing to do.
if (!base.StartUp(position, tilemap, gameObject))
      return false;
```

It would be an extremely bad idea to change any code in the Startup method in TilePlusBase. Worse than being eaten by the Sarlacc on Tatooine or looking into the atomic furnaces of the Forbidden Planet, Altair IV.

GetTileData is probably the most complicated override, aside from being sure to call the base class or duplicating its code, examine some of the examples for guidance. But it's obviously extremely specific to what you're trying to do. Be sure to use the tilemapIsPalette field to exit the method after the base call if tilemapIsPalette is true. Otherwise, your tile might animate in the Palette window. **GetTileAnimationData** code should also test tilemapIsPalette. See TpFlexAnimatedTile for examples.

ResetState is used in-editor when using the **Bundle Tilemaps** menu command or the **TilePlus Bundler** command in the hierarchy window's context menu. It's also used when you pick and re-paint a clone TPP tile or use TpLib.CopyAndPasteTile. The implementation must reset fields so that stale data isn't persisted. Be sure to call the base method. This is a misleadingly simple method that you need to think about carefully. You don't want to reset all fields, or you'll be undoing changes made to your TPT tile. See the various implementations for examples.

Namespaces

The GUI formatter for the Brush and Selection inspectors displays information in class-hierarchical order. But it needs to know what not to display, otherwise it will breeze through the class hierarchy all the way to UnityEngine.Object. Therefore, by default it ignores anything outside of specific namespaces. The TilePlus namespace is hard-coded in. The configuration editor has a **Namespaces** text field where you can provide a comma-delimited list of namespaces to use. The default for that text field includes TilePlusDemo. When creating your own tile classes, place the namespace that you're using in this list. Don't forget commas! Note that if you add a namespace, attributes are still required to display information. For example, if you were to add the *UnityEngine.Tilemaps.Tile* namespace then the TilePlusBase' base class of Tile would not appear in a foldout. Please click the Reload button in the configuration editor when you change this. ***Also, be aware that if you click "Reset To Defaults" that you'll need to re-add the namespaces!***

Interfaces

ITilePlus specifies several properties and a few methods that are common to all tiles subclassed from TilePlusBase since that class implements everything in the interface. Please note that any subclasses of TilePlusBase using ITilePlus properties with default members need to specify the interface to 'override' the defaults. This can be seen in the tiles which support simulation (TpSlideShow, TpAnimatedTile, and TpFlexAnimatedTile).

ITpPersistence specifies properties required for tiles using TpLib's save/restore framework.

ITpMessaging specifies properties required for tiles using TpLib's two-way messaging framework.

ITpSpawnUtilClient specifies properties required for tiles which spawn prefabs or paint tiles when using the SpawningUtil library methods.

ITpMessaging and **ITpPersistence** are the interfaces you'll most likely implement if creating your own tiles and you want to use TpLib's messaging and save/restore frameworks. If you don't want to use those then you can ignore those interfaces.

If you're subclassing tiles, you'll probably be overriding implementations of ITilePlus from TilePlusBase or one of its subclasses.

Demo projects

TilePlus includes several demonstration projects that you can examine to help you understand the many ways to use TPT tiles. Some more complex demos are discussed in the AdvancedTileFab Use document.

Let’s start with a simple one.

BasicTiles

This comprises a small scene with three instances each of the **same** TpAnimatedTile asset showing an animated eyeball. Each cloned tile has different animation speeds. There are multiple instances of the **same** TpFlexAnimatedTile asset showing an animated bug, also with differing animation speeds. Finally, there’s a TpSlideShow tile with the transform and position adjusted so that it acts as a background.

When run, the simple BasicDemo game controller enters a loop where it moves each tile a little in each pass using TpLib.*CutAndPasteTile*. The first time through the loop, the tiles are cloned with TpLib.*CopyAndPasteTile*. These are all independent tile instances with their own instance data.

The TpSlideShow tile is advanced to the next slide every 16 loop iterations, causing the background to change.

You can try using the Tile+Brush’s selection inspector to modify animation speeds, easing settings and so on, and you can look at the original assets in the Project folder to convince yourself that the original assets are unmodified.

SaveRestore

This is another simple example that shows how to use ITpPersistence to save and restore data without using any ITpMessaging features.

There are two painted instances of the same tile asset, A and B. The tile class is called SaveRestoreDemo. One instance was modified in the scene to change its sprite. Each was modified in the scene to change the example enum, bool, string, float, and int fields.

When you run the scene in the editor you can use the Save button to save the instance data from each tile to the filesystem. Then use the left-side buttons to clear one or both tiles’ instance data. Clicking Restore will restore the saved data. Status->Console just prints the values of the tiles’ instance data.

This is intentionally a simple example to illustrate one way to use ITpPersistence. Let’s see how it works. First, open the tile asset script SaveRestoreDemo.cs. Most of it is quite simple. The implementation of ITpPersistence has three elements:

- Class DemoData
- GetSaveData
- RestoreSaveData

DemoData is the class definition for the data packet for this tile. It merely encapsulates the tile’s fields and adds an additional field for the tile’s GUID. Note that it inherits from MessagePacket as required by the ITpPersistence interface specification.

GetSaveData implements the ‘fetch’ part of the interface: it returns a DemoData instance with the tile’s fields’ values copied into the instance along with the tiles’ GUID. You might notice that the input parameter to GetSaveData in the SaveRestoreDemoClass is ‘_’. That’s called a discard parameter. But it’s actually an ‘object’ parameter, so you can use it for anything you want. In the TopDownDemo program, this parameter is used to pass a (boxed) boolean value denoting whether to pretty-print the save files (make human-readable).

RestoreSaveData implements the 'put' part of the interface: it accepts a DemoData instance and copies the DemoData instance fields into the tile's fields.

Now look at the MonoBehaviour script SaveRestoreDemoController.cs. This component was added to the MainCamera GameObject for simplicity.

In the script's Start method the demo tile instances are located and saved for use when clicking the Clear Tile A/B buttons.

When you click the Save button the SaveData method is invoked. SaveData uses TpLib to obtain references to any tiles using the ITpPersistence interface.

```
TpLib.GetAllTilesWithInterface<ITpPersistenceBase>(ref demoTiles);
```

> Note that the Base class for the interface is used. This will return all tiles that use ITpPersistence no matter what class is used for the data structure. Alternatively, if this statement was:
>
> ```
> TpLib.GetAllTilesWithInterface<ITpPersistence<DemoData,DemoData>>(ref demoTiles);
> ```

Then only the specific type of tile using DemoData classes for moving data would be returned.

The code then loops thru the list of tiles, using the ITpPersistence method GetSaveData on each tile instance and bundles it all up in a JSON string which is saved to the filesystem.

The Restore button invokes RestoreData, which loads the JSON string and splits it into sections; one for each tile. For each section, the JSON is unpacked into DemoData class instances. The destination tile is located by using the Guid member and TpLib.GetTilePlusBaseFromGuidString(data.m_Guid). Finally, the DemoData class instance is sent to the tile ITpPersistence method RestoreSaveData.

One issue when adapting this sort of 'distributed' save/restore method to your own projects is how to determine where to send restored data. Using the GUID means that multiple Tilemaps aren't an issue. If the position were used instead, then if multiple Tilemaps were used the recipient of restored data could be ambiguous. One could use the position and the name (or tag) of a Tilemap's GameObject. However, if the name or tilemap tag were to change then the save file would be broken. The GUIDs used in TilePlus tiles are persistent.

Furthermore, if you use TileFabLib to populate your Tilemap, the GUID system is extended to include saving and correctly restoring the state of such loaded Tilemap sections. See *Dynamic TileFab Loading*.

DOTweenDemo

This demo provides a simple example of how to use DOTween and the DOTween plug-in. Ensure that you've both installed DOTween and have added the **TPT_DOTWEEN** definition to the Player settings Scripting Define Symbols list. The example scripts are #ifdef'd out unless this definition is found. Note: newer versions of DOTween add DOTWEEN to that list and that's acceptable as well: you don't need both.

The small scene for this demo has four instances of a DtDemoTile. Each is set differently: from left to right, for scaling, position translation, Color, and Rotation. These values can be changed via the Selection Inspector.

Each tile can be inspected using the Tile+Brush or Tile+Painter if you'd like to change certain exposed tweening parameters like duration and easing. Note that when the type of tween is set to Scale you can select between using a pair of tweens operating sequentially or the same pair of tweens added to a sequence.

You might notice that there is no "Game Controller" or indeed any custom MonoBehaviours in this scene. That's not 100% true: there's a GameObject spawned and parented to the "DontDestroyOnLoad" GameObject that appears when the scene is run in Play mode. That's added by TpLib when a TPT tile uses a delayed callback. There's an appendix on this subject later in this document.

AnimatedTiles

This is another simple example: How to control TPT animated tiles from a MonoBehaviour. There are two scenes: Animation-MouseClicks and Animation-UnityUI.

Animation-MouseClicks

Mouse-button left-clicks are checked for a match for one of the tiles. The left tile (a TpAnimatedTile) will trigger a one-shot animation. The right tile (a TpFlexAnimatedTile) will animate until you click the button again.

A small Monobehaviour script called TpPickTile converts the mouse position is converted to Tilemap grid coordinates and TpLib is used to locate a tile.

Animation-UnityUI

You can toggle an animation on or off via a UI button on instances of two animated tile versions: TpAnimatedTile (left) and TpFlexAnimatedTile (right). The latter's script is the base class for other tiles such as TpAnimZone.

A small MonoBehaviour script called TpTileProxyAnimOnOff (quite a mouthful) uses the GUID of a tile (which can be seen in the OnClick field of each button) to find the tile and control the animation. If you want to change the GUID in the OnClick, select a tile with the Tile+Brush or view it in the Tile+Painter and use the Copy GUID (eyedropper) button to copy the GUID into the clipboard. You can use CTRL+V (Windows) to paste it into the OnClick field as there's no context menu for this field.

TopDownDemo

This somewhat complex demo project uses most of the provided TPT tile classes, adds a few new ones, and demonstrates a simple implementation of the messaging and save/restore functionality as well as showing how to load TileFabs.

New Tiles

TopDownWaypointTile: This is a subclass of TpSlideShow, and additionally implements the ITpPersistence and ITpMessaging interfaces.

The TdDemoGameController sends this tile a position packet message every move. The packet passes the location of the movable Player character. If the location matches that of the Waypoint, then the slideshow advances to the 'enabled' image and a SaveData event is created. This event is handled by the TdDemoGameController as detailed below.

AgentTile: A subclass of TpFlexAnimatedTile. These are the bad guys who follow the PlayerTile around the Tilemap and try to attack it. The tile has a randomized lifetime (based on min/max values which you can customize in-scene with the Selection Inspector) that is initialized on StartUp. A method is added to allow the tile to "rotate" (modify its transform matrix) so that it appears to point towards the Player tile. There are a few of these scattered around the Tilemap when the minigame begins, and more are spawned as the Player tile moves around and passes over visible and invisible TpAnimatedZoneSpawner tiles pre-placed on the Tilemap.

Mono-Components

PathfinderInfo: A tiny component which is placed on the same GameObject as a Tilemap. The PathFinder component looks for these to determine what type of Tilemap this is: one with static (non-moveable) obstacles, mobile obstacles, or Floor (overall boundary area).

PathFinder: Not so deluxe as A* pathfinding, this simple component is placed on the parent Grid of the Tilemaps that need to be monitored. It caches the positions of mobile and static obstacles as indicated by the Tilemaps' Pathfinder components. IsWalkablePosition is used to check if the Player or one of the Agents can move in the direction that it wishes to move. It subscribes to TpLib.OnTpLibChanged so that it can update the positions of mobile obstacles (Player, Agents) since those moves will trigger that Event.

TdDemoPlayerPrefabLink: This is a subclass of TpSpawnLink which adds some simple functionality to make the Player tile's linked prefab Flash a few times when an Agent tile 'hits' it.

TdDemoGameController: A simple controller which accepts WASD input to move the Player tile.

Start is a Coroutine which waits a few frames and then:

- Restores all the dynamically-loaded sections, restores data to all the tiles and sets the initial position of the spawned Player GameObject.
- Rotates any pre-placed Agent tiles to point to the Player tile.
- Tells the PathFinder component to scan the Tilemaps.
- Forces TpLib to rescan all the Tilemaps in the scene.
- Shows the initial score (number of lives left for the Player tile)
- Subscribes to the TpLibOnTileEvent, allowing it to be notified when a tile sends a message.

Update debounces keyboard input. If there's a valid keystroke, it checks to see if the Player prefab is still moving. This is possible since its position is eased, so it takes some time to reach its final position. If the prefab isn't moving, the keystroke is examined to determine which direction to move. The use of a lookup table allows certain combinations of keys to move diagonally (e.g., W and D move diagonally-up). The Player prefab is rotated to match the new direction. But can we walk there? The call to the Pathfinder component controls that: if any obstacle is in the new location, then no movement is possible.

If movement is possible, the prefab is eased to the new position.

Following that, **UpdateAgents** is called. This method uses TpLib.GetTilesWithTag to get all the Agent tiles. Then it deletes all Agent tiles whose lifetimes have expired. The remaining Agent tiles' positions are examined to see if they are touching or next to the Player tile.

After rotating the Agent tile to point at the Player tile, agents which didn't touch the Player are moved using TpLib.CutAndPasteTile, using a lambda method to test for a walkable position.

If the Agent tile did touch the Player tile, then it's hit position is recorded for action on the next Update and the agent tile is deleted.

Then messages are sent to various Tiles on different Tilemaps. This tells the tiles the Player's position. Some or none of the tiles will create TileEvents in response, others will spawn enemies and/or effect prefabs. The ZoneLoader tiles send a trigger event.

Then events are examined. If there were any SaveData events data is saved via SaveData. Trigger events are examined using GetFilteredEvents to select events only from ZoneLoader tiles. If such an event is found, a TileFab is loaded using LoadZone.

The **LoadZone** method handles loading after a ZoneLoader creates a trigger event. All the work is done by the static library TileFabLib. Note that the newGuids and registerThisTileFab parameters are true. NewGuids tells TileFabLib to create new GUIDs for any TilePlus tiles. This is important since the same TileFabs are loaded repeatedly and TilePlus tiles require unique GUIDs. See "*Advanced Uses For TileFabs*."

After ProcessTileEvents returns and if the Player had been touched by an agent tile, its lifetime is decremented, and the game is over when there are no more lives.

It's important to note that neither the Player nor any Agent tile has its position changed via direct calls to Tilemap methods; rather, TpLib methods are used. There's no problem using the native Tilemap methods to affect things like Color, Flags, or Transform (although the TPT tiles themselves may undo your changes when refreshed). However, anything that affects the position of a TPT tile needs to be done through TpLib.

This demo is by no means a game template. There are numerous other ways to perform messaging, data saving, etc.

CollisionDemo

CollisionDemo illustrates one way to detect collisions of GameObjects and tiles. The collision detection is decently accurate and may or may not be an approach that you can or want to use, but it was decided to leave this example in the distribution to illustrate some of the unusual things that you can do with TilePlus tiles; and if you have DOTween installed and TPT_DOTWEEN defined in your Player settings, illustrates using DOTween in a similar way to that seen in DOTweenDemo. Note: newer versions of DOTween add DOTWEEN to that list and that's acceptable as well: you don't need both.

The ICollidableTile interface specifies what properties and methods are required to work with the Monobehaviour component TilemapCollisionDirector. TilePlusCollidableBase provides a base class.

Two new tiles:

- AnimatedTileWithCollision subclasses TpAnimatedTile and implements ICollidableTile. It animates when hit, freezes the rotation of and adds force to the GameObject which hit it.
- TileCollision subclasses TilePlusCollidableBase. It spawns a prefab when hit, unfreezes the rotation of and adds force to the GameObject which hit it. If DOTween is installed, the sprite of each TileCollision animates after being collided-with once.

There are several instances of AnimatedTileWithCollision and TileCollision in the scene. Each placed tile instance was set up for different amounts of force by modifying fields in the Selection Inspector.

The TilemapCollisionDirector component is placed on the Tilemap and has event handlers for the six types of collisions and triggers; however, it's only been tested with "OnCollisionEnter."

When one of those events occurs, it compares the bounds of the collision with the bounds of sprites with this one TpLib method call:

```
TpLib.GetAllTilesWithInterface<ICollidableTile>(ref collidedTiles,
                           (ict, tpb) => bounds.Intersects(ict.TileColliderBounds));
```

Here, the collidedTiles list is filled with tiles which implement ICollidableTile and whose bounds intersect with those of the source collider. This is sufficient for this small number of tiles, but a slightly different approach would be needed if there were a much larger number of tiles.

The effects prefabs are pooled: you can see this in action while the CollisionDemo is running by opening the TP System Info panel. You'll see the pools expanding as more prefabs are created.

TileFab Demos

See the Advanced TileFab Use document for more information.

Appendix: The Lifetime of a TilePlus tile

TilePlus lets you treat a Tile script much like a script attached to a GameObject: but Tiles are not GameObjects. It's easy to forget that a Tile is based on the ScriptableObject class. Here's part of what the Unity manual says about Scriptable Objects:

Just like MonoBehaviours, ScriptableObjects derive from the base Unity object but, unlike MonoBehaviours, you can not attach a ScriptableObject to a GameObject. Instead, you need to save them as Assets in your Project.

What's left out of that statement is that *you cannot attach a ScriptableObject to a GameObject **as a Component***. But you can attach it as a reference via a field in a script. When you do that, the reference is to the asset in the project folder. Clearly, you can create an actual instance of the Scriptable Object in memory, and it can be placed in the reference 'slot'. That's essentially how TilePlus tiles work:

- You paint the tile (or place it programmatically). It's in the ASSET state at that time.
- The tile's StartUp method sees that the state is ASSET and queues a cloning request in TpLib. The tile is cloned at the next Update and the clone is placed at the same location, replacing the tile asset reference in the Tilemap.

Since the clone is referenced in the Tilemap now, it's saved with the scene. Note that when painting a tile using the Unity Tile Editor or TilePlus Painter the cloning occurs only when the tile is placed on a Tilemap.

But it's still not a GameObject: most of the events are missing. The only really useful events are OnEnable and OnDisable. Fortunately, Tiles have a StartUp method where it is passed a reference to the parent Tilemap and the position. Follow along by examining the StartUp method of TilePlusBase.cs (not every line is discussed):

- A reference to the parent Tilemap for the tile is cached.
- The tile's position is cached.
- A flag is set if the position has changed (for example, if you'd moved it using the Cut/Paste function of TpLib).

From that point there are two code branches depending on whether the tile is already a clone.

- Clone: check for a proper GUID and register the tile with TpLib.
- Asset: queue for cloning in TpLib. The clone replaces the asset reference in the Tilemap via Tilemap.SetTile. This causes StartUp to be executed again.

From this point in time the tile is essentially passive. When the Tilemap calls GetTileData and GetTileAnimationData the information returned from those methods are copied into the Tilemap data structures.

OnEnable, which generally will execute before StartUp, lets you set up initial conditions. Examples of this can be seen in the animated tile classes.

OnDisable, can be used for cleanup.

OnDestroy, while theoretically available, is **not** useful since it's not called at any predictable time unless one were to Destroy tile instances programmatically. You'll not see it used in TPT tiles at all.

Note that the TpLib.MaxNumClonesPerUpdate property controls how many cloning operations are executed on each Update (default is 16). This allows performance optimization.

In a build or when Playing in the Editor: The first time that a TPT tile requests cloning, a root-level prefab is added to the scene with DontDestroyOnLoad set. This prefab acts as a proxy for TpLib to provide an Update tick.

TPT Tiles are always cloned when painted in the Editor or when added programmatically during the application's execution.

When a Tilemap is made into a prefab using BundleTilemaps, archived TPT tiles are 'Locked' assets.

Cloning also occurs when loading TileFabs or TpTileBundle contents to your scene. For efficiency, this is done inline, within the loading process; and all at once, so this approach doesn't use this special TpLib cloning feature.

The process is the same for TPT Prefabs which are essentially "wrappers" for TileFabs. When the prefab is instantiated, or if a scene is loaded with TPT tiles in Tilemap prefabs then all these Locked tiles are cloned, inline.

In other words, TPT tiles will only request cloning when painted in-editor or at runtime, in code. If you want to paint numerous TPT tiles at runtime, consider using TileFabs, Bundles, or TPT Prefabs. If you won't want to use those, examine the loading code for the Bundle asset to see how to clone Locked tiles inline.

This should factor into your setting value for TpLib.MaxNumClonesPerUpdate.

For example, with the default value of 16 for TpLib.MaxNumClonesPerUpdate, painting 1024 TPT tiles will cause 1024 cloning operations. If only 16 are added during each Update, then assuming a 60 Hz Update frequency this would take about 1 second.

Knowing this, you might want to dynamically change this property's value when you load a scene or instantiate a prefab.

If you're not painting huge numbers of TPT tiles at runtime, then you don't need to worry about the value of TpLib.MaxNumClonesPerUpdate.

Appendix: Deferred Callbacks in TpLib

Sometimes you just need to wait.

Certain types of actions must be deferred. Here are some simple examples:

- A tile's Startup method overwrites its own position with a null tile. Probable Crash.
- A UIElements or IMGUI method wants to open a dialog box. Endless GUIclip and other similar warnings.
- A tile wants to rewind a one-shot animation after it's complete: this requires a time delay.

In a Monobehaviour one can use Update or a Coroutine to create a delay. But neither of those can work in a tile instance. TpLib has a facility to provide delayed callbacks using the DelayedCallback method like so:

```
public static async void DelayedCallback(UnityEngine.Object parent,
                              Action            callback,
                              string            info,
                              int               delayInMsec = 1,
                              bool              silent = false,
                              bool              forceTaskDelay = false)
```

This can be used by any code, not just tiles.

The *parent* reference can be null if you're calling this method from a static class. However, for a tile or other UnityEngine.Object, the parent will be examined for non-null prior to invoking the *callback*.

The *info* string is used for diagnostic information such as in the case of an exception when the callback is executed. If *silent* is true, then no diagnostic information is printed unless there's an error or exception.

DelayedCallback examines the *delayInMsec* parameter. If it is <= 10 the callback is placed in a Queue and is handled on the next update. Otherwise, the delay is created by using Task.Delay. The optional parameter *forceTaskDelay* can be used to push this callback into a Task regardless of the delayInMsec parameter value.

When Queued (rather than using a Task) you should note that the TpLib.MaxNumDeferredCallbacksPerUpdate property controls how many Queued callbacks are executed during each Update. This value defaults to 16. Changing the value of the property allows performance optimization.

You can observe the Queue and other aspects of TilePlus internals with the System Information editor window.

Wait, Update?

TpLib is a static class. Yet it has an Update method. In Editor (but not Playing) this is handled by subscribing to the EditorApplication.update callback.

In a build or when Playing in the Editor: The first time that DelayedCallback is used with a delayInMsec value > 10 OR the first time that a TPT Tile is cloned, a root-level GameObject is added to the scene with DontDestroyOnLoad set. This prefab redirects its Update event into TpLib to provide an Update tick. If you want to avoid any additional delay for that first use, call **TpLib.Preload** in your program's initialization code.

A Warning

Using many Delayed Callbacks with delays > 10 milliseconds will create a large number of Tasks and slow down your program***: in extreme cases using 100% of your CPU***! Similarly, a large number of Delayed Callbacks with delays < 10 milliseconds will cause some number of them to execute per-Update (the number is determined by MaxNumDeferredCallbacksPerUpdate). This feature should be used sparingly, and NOT as a coroutine substitute where coroutines could be used.

Appendix: One-Shot Animation with TpAnimatedTile and TpFlexAnimatedTile

Some new Tilemap animation control features were added in Unity 2022.2, including the ability to have a one-shot animation. This has one caveat: the animation ends on the last frame. There's no facility to rewind the animation to show the first sprite in the sequence or the Tile's Sprite, nor any way to know when the one-shot animation has completed. Rewinding or changing to a static sprite upon completion of a one-shot animation is often desirable.

These two types of Tiles have a unique feature to allow rewinds.

For TpAnimatedTIle there's a field called "Rewind After One-Shot" field.

For TpFlexAnimatedTile the SpriteAnimationClipSet's Customization field for each animation clip has a "Rewind After One Shot" field. The Tile itself has a "Rewind After Forced One Shot" field that only applies if "Force One Shot" is true.

With that said, if either of these three rewind features is used, a time delay is computed and that's used to set up a Delayed Callback in TpLib. That callback does the rewind. It's fairly accurate for slower animation speeds but not perfect for faster ones. Unfortunately, there's no easy alternative until Unity Inc adds a callback when the one-shot animation is completed (supposedly this will appear in a future version of the Unity API).

Appendix: TpLib Memory Allocations

The static library TpLib.cs has several Dictionaries that keep track of all TPT tiles. The initial size of these dictionaries is set by constants in the TpLib.cs file. Similarly, pooled Dictionaries and Lists have a constant size when new pooled instances are created.

These constant size values are small. It is possible to change these allocations during App startup with the TpLib.Resize method. An instance of the TpLibMemAlloc class is passed to Resize.

One could change the constants themselves, however, updating the Plugin will naturally revert these values. Hence, Resize is a better choice.

There's no reason to use this feature in an editor session. At runtime, use Resize immediately/soon after startup.

Internally, Resize releases all pooled items and resets MaxNumClonesPerUpdate and MaxNumDeferredCallbacksPerUpdate to their initial values.

The TpLibMemalloc values for pooled Dictionaries and Lists affect the size of new pooled instances of

- `Dictionary<Vector3Int,TilePlusBase>`
- `List<TilePlusBase>`

These are created and pooled frequently and optimizing this size can have a significant effect on performance.

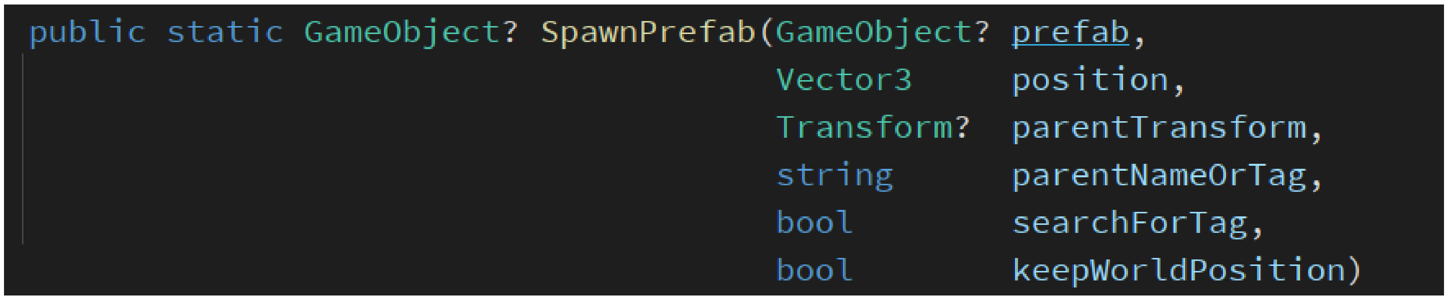
Appendix: TilePlus' Pooler

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. This is implemented in `SpawningUtil.cs`.

SpawningUtil 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.

A tile can use SpawningUtil to spawn pooled prefabs, with preloading. That feature is used by `TpAnimZoneSpawner`. You can examine that tile class code to see how it works.

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



```
public static GameObject? SpawnPrefab(GameObject? prefab,
                                      Vector3     position,
                                      Transform?  parentTransform,
                                      string      parentNameOrTag,
                                      bool        searchForTag,
                                      bool        keepWorldPosition)
```

The first two parameters are obvious.

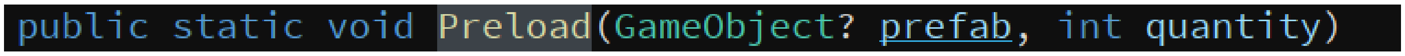
`parentTransform`: If provided then the spawned Prefab is parented to that transform. If the value is null a search is done for the parent using the `parentNameOrTag` string. If that string has a value, then one of two things occurs:

- `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).

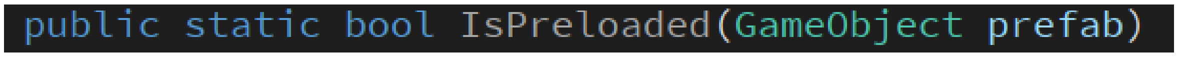
TLDR; want parent? Provide one or use the `parentNameOrTag` search string.

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).



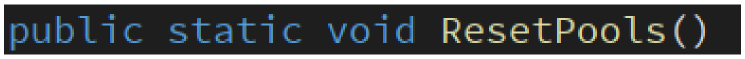
```
public static void Preload(GameObject? prefab, int quantity)
```

Preloading a pool can be done with Preload().



```
public static bool IsPreloaded(GameObject prefab)
```

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



```
public static void ResetPools()
```

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

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.

It's important to properly call the base class methods from any overridden methods. For an example see `ParticleSystemColorChanger.cs` in `CommonAssets/Prefabs/PrefabScripts` or `TdDemoPlayerPrefabLink` in the TopDownDemo scripts folder.

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 window shows 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.

Appendix: Inhibiting TpLib Callbacks

For performance reasons you might want to temporarily inhibit TpLib from responding to certain Tilemap callbacks, specifically:

- tilemapPositionsChanged
- tilemapTileChanged

For example, if you fill a large area of tiles these callbacks will trigger repeatedly.

Use the TpLib property **InhibitTilemapCallbacks** to force TpLib to ignore these callbacks. Note that if any TilePlus tiles are added or deleted by whatever you're doing then TpLib will be out of sync. You can use SceneScan to rescan.

When using the Unity Editor, this property is reset after a scripting reload or when the Editor switches to Play mode.