5 September 2023

This is an archive of posts on a defunct Hashnode blog that may be interesting for those trying to understand what TilePlus Toolkit is about and why it exists.

Note that descriptions in this document may not exactly correspond to the current state of the project.

Tilemap Nirvana

For the last few years I've been working on a framework for casual games using the Unity game engine. Using native Unity Tilemaps is efficient, but very limiting. This is because they're designed for display efficiency but there's no apparent way to add custom data to tiles. It's just not stored anywhere.

I decided to jump down the rabbit hole and figure out why.

The answer can be found by creating a simple scene with a Tilemap and a few tiles. If you ensure that asset serialization is set to force text, then saving the scene gives you something to look at. Here's the most interesting part - the internals of the entry for the Tilemap component. Here's the serialized data for a single tile:

```
- first: {x: -4, y: 1, z: 0}
   second:
     serializedVersion: 2
     m_TileIndex: 4
     m_TileSpriteIndex: 0
     m_TileMatrixIndex: 0
     m_TileColorIndex: 0
```

What's going on here? The line `first: {x: -4, y: 1, z: 0}` tells us the grid position of the tile i.e., where it is on the Tilemap. But what's the rest of this about? What are the indexes for?

Looking elsewhere in the file one finds this:

```
m_TileAssetArray:
m_RefCount: 1 m_Data: {fileID: 206097775}
m_RefCount: 1 m_Data: {fileID: 619153061}
m_RefCount: 0 m_Data: {fileID: 0}
m_RefCount: 1 m_Data: {fileID: 1263359791}
m_RefCount: 1 m_Data: {fileID: 1690371934}
m_RefCount: 1 m_Data: {fileID: 1056322544}
m_RefCount: 1 m_Data: {fileID: 1174581608}
m_RefCount: 1 m_Data: {fileID: 1667070803}
m_RefCount: 1 m_Data: {fileID: 628196556}
```

Ah, so the indexes point into a lookup table of tile assets! The fileID's must map to actual tile assets in the Project folder somewhere. Other lookup tables for the sprites, colors, etc can also be found.

The intent here seems clear: Tilemaps are for displaying visual information and that's all. Since there's so much repetition of individual tiles it makes a lot of sense to use lookup tables to reduce the amount of data to serialize.

If you've ever worked with Unity Tilemaps you might have attempted to add some fields to a subclass of Tile, let's say some sort of boolean value to control something. It seems to work! But then you paint a second instance of your fancy new tile. You change the bool's value, and lo and behold you find that the value of this variable in the first tile that you had painted, and indeed, the value of the bool itself in the actual tile asset has also changed.

Hard stop! It doesn't work.

Actually it's almost impossible to do the steps I've just outlined. When you paint the tile you can select it with the brush and view its values in the selection inspector.

But that will only display the standard tile fields like color and so on. If you have Odin Inspector installed you can open an inspector on the tile right from the brush's selection inspector, twiddle the new fields and you'll see what I just described.

But this is not news to the Unity development community.

One way to add functionality to tiles is to add a prefab to the tile, and it will create an instance of the prefab for you. This isn't terribly useful (IMO) since it's the same prefab each time.

In the next post I'll talk about how I found a way to get around all this.

Not so fast...

In the previous (actually the first) blog entry, I suggested that there was a way around the "no instance data" issue for Unity Tilemaps and that I'd discuss that next. Actually, not this time, sorry.

Today I want to talk about how confusing some of the Tilemap nomenclature is.

There's a Tile method called RefreshTile. There's a Tilemap class method called RefreshTile. You'd think that calling Tilemap.RefreshTile would call Tile.RefreshTile. No, it doesn't.

Actually, it calls Tile.GetTileData followed by Tile.StartUp.

What about when you run your Tilemap app in the Editor? All that's called is Tile.Startup. You may well ask, why not Tile.GetTileData too? That's because the tile data is already in the Tilemap component's internal data structures.

It's important to understand that the Tile which is maintained as a reference is only really used for one thing when your app runs: Running StartUp once.

During edit-time, when you paint a tile, GetTileData and GetTileAnimationData are called to load that data (sprite, transform, sprite sequence...) into the internals of the Tilemap instance in the Tilemap component.

When you save the scene, all the data that the various tiles had loaded into the Tilemap component is saved in the scene file in an indexed fashion so that repeated references to the same Tile asset are avoided and so that repeated color, transform, and flags values are also avoided. So at runtime there's no need to call anything other than StartUp; this saves a lot of time when the scene loads when you consider the large number of repeated tiles.

About the only time that I could find Tile.RefreshTile being called is when you're editing Tilemaps and you move the brush around in the Scene or over the Palette area. This makes sense for support of Rule tiles, I suppose; and the Palette is also a Tilemap.

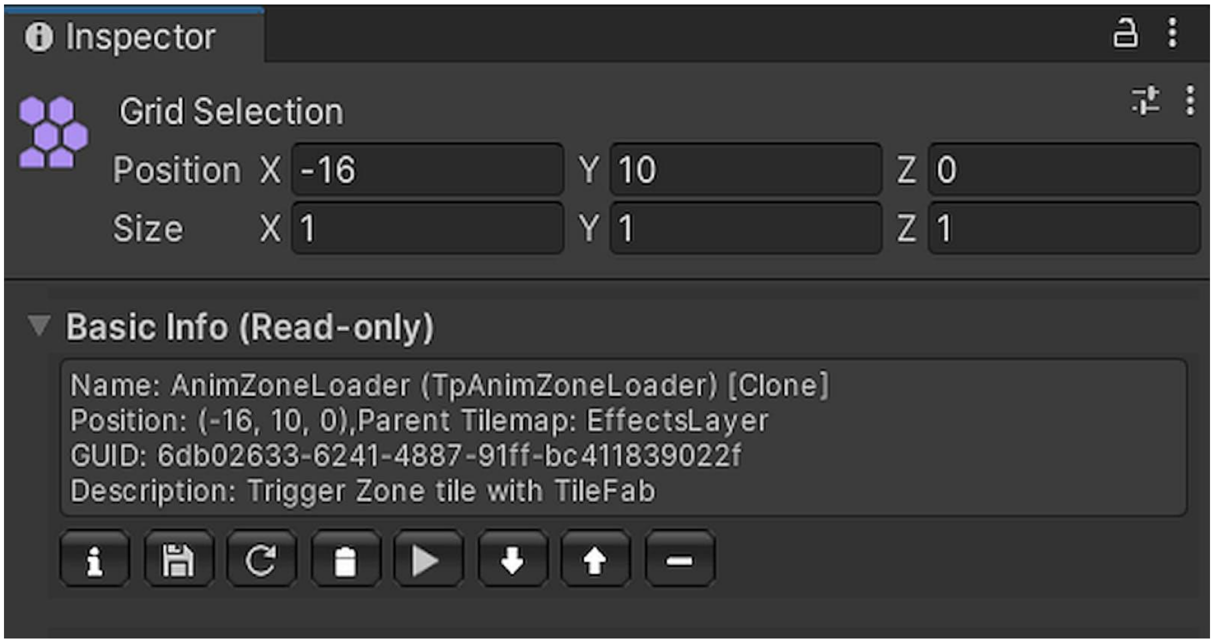
Here's a fun Tile asset that you can add to an empty Unity project. Be sure to use the name 'TestTile.cs' for the filename.

Using the Asset-Create menu, make two of these into Tile assets, add a different sprite to each, drag them into a palette and paint one of each on an empty Tilemap.

You'll see messages in the console when each of these three methods is called with info about whether the tile is in the Palette. Drag the Paint tool around on the Tilemap. Click the edit button in the Palette and drag the Paint tool over the palette.

```
using UnityEngine;
using UnityEngine.Tilemaps;

[CreateAssetMenu(menuName = "Create TestTile", fileName = "TestTile", order = 0)]
public class TestTile : Tile
{
    /// <inheritdoc />
    public override void RefreshTile(Vector3Int position, ITilemap tilemap)
    {
        Debug.Log($"RefreshTile @{position}: from palette? {IsTilemapFromPalette(tilemap)}" );
        base.RefreshTile(position, tilemap);
    }
    /// <inheritdoc />
    public override bool StartUp(Vector3Int position, ITilemap tilemap, GameObject go)
    {
        Debug.Log($"Startup @ {position}: from palette? {IsTilemapFromPalette(tilemap)}");
        return base.StartUp(position, tilemap, go);
    }



    /// <inheritdoc />
    public override void GetTileData(Vector3Int position, ITilemap tilemap, ref TileData tileData)
    {
        Debug.Log($"GetTileData @ {position}: from palette? {IsTilemapFromPalette(tilemap)}");
        base.GetTileData(position, tilemap, ref tileData);
    }



    /// <summary>
    /// Is this tilemap actually the palette?
    /// </summary>
    private bool IsTilemapFromPalette(ITilemap iMap)
    {
        var map = iMap.GetComponent<Tilemap>();
        return (map.hideFlags & HideFlags.DontSave) == HideFlags.DontSave;
    }

}
```

How to have Instanced Tile data



Name: AnimZoneLoader (TpAnimZoneLoader) [Clone]
Position: (-16, 10, 0),Parent Tilemap: EffectsLayer
GUID: 6db02633-6241-4887-91ff-bc411839022f
Description: Trigger Zone tile with TileFab

In an earlier post I mentioned that it is possible to have instanced Tile data when using Unity Tilemaps. It's been a while, but now that I've uploaded the TilePlus Toolkit asset to the Unity Asset store, it's a good time for me to STOP CODING and talk about how it's done.

Simply put, make a clone of the tile after it's painted. Sounds easy! Uh, not really.

Perhaps it might help to explain why I did this at all.

I've been working on a casual games project using Unity3D. As a Unity user since about 2009, my first sensation when Tilemaps were added to Unity in 2017.2 was "Wow! Tilemaps!!". Soon replaced by "you mean tiles can't have instance data? WTF!".

I first used the Tiled tilemap editor to create Tilemaps and import them using the Tiled TMX Importer from the Asset Store (from Gaming Garrison). But I really wanted to work within the Editor itself: Every change that I made required an import process and some manual adjustments. Besides, I ended up having instance data in prefabs, which is really annoying from a design point of view. I really wanted to have intelligent tiles.

I wanted tiles that could react to the environment and have better animation capabilities than the animated tiles from 2D Tilemap Extras.

So it became a challenge to figure out why Tiles can't have instance data. As I talked about in an earlier post, you can examine Unity's text-mode YAML scene files to infer that tiles are mostly used to set up lookup tables and such within the C++ engine code that comprises the engine's tilemap implementation. As such, all that is saved in the scene file is a set of lookup tables for transform, color, and flags, and a reference to the tile asset. At runtime (Play mode), the tile asset isn't really used for much after it's StartUp method is called.

But if one clones the tile asset using Instantiate then you have a "floating" copy of the tile, which is really just a Scriptable Object subclass.

If you use that copy when calling Tilemap.SetTile, the copy becomes bound to the Tilemap due to the reference which the Tilemap maintains. It really isn't a big deal to have a Scriptable Object reference in the scene hierarchy: it doesn't appear in the editor; but more importantly, it's saved with the scene. In effect, it's now a "transformless" GameObject.

About the only downside to this is that since these are scene objects, and since the tilemap's internal reference is to a TileBase (among other issues), if you do a simple drag-drop of a tilemap to make a prefab in a project folder, the connection is lost. There are several ways to handle this, but I chose to code a custom prefab creator.

However, there are other ways to load Tilemaps dynamically. That's the topic of an upcoming post.

It Gets Complex

The concept of cloning a tile to divorce it from the asset and enable instance data isn't that hard to understand. But say that you want to edit the data: you use your trusty selection inspector to click on one of your new-fangled tiles and .... no good!

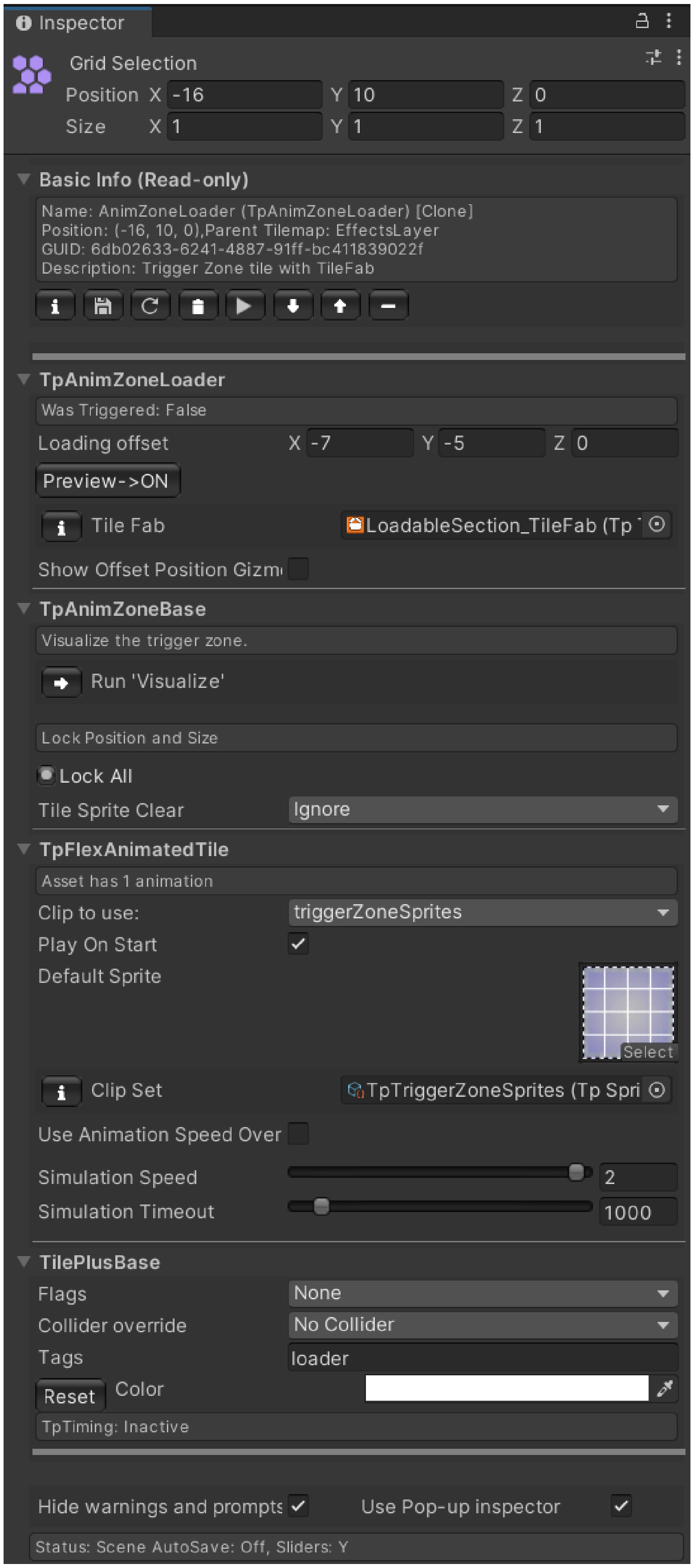
It turns out that the Selection Inspector in the default brush will only display the fields in a standard Unity tile. Unlike other inspectors which are fairly general purpose, this one is specific to the Tile class and hard-wired to only display those fields.

Similarly, the inspector seen when using the palette needs updating.

So a custom brush needed to be created with new inspectors.

One thing I really dislike about Unity inspectors for normal objects (as opposed to one you might create for a specific monobehaviour) is that there's no real way of ordering the information unless you use something like Odin Inspector (which is a terrific product).

So I had to roll my own. Without getting too deep into it, you merely decorate your fields and/or properties with specific attributes and a complex chunk of reflection code orders these items in the same class hierarchy as your code. Here, a pic is worth a thousand... well, you know what I mean.


Inspector

Grid Selection

Position X -16 Y 10 Z 0

Size X 1 Y 1 Z 1

Basic Info (Read-only)

Name: AnimZoneLoader (TpAnimZoneLoader) [Clone]
Position: (-16, 10, 0),Parent Tilemap: EffectsLayer
GUID: 6db02633-6241-4887-91ff-bc411839022f
Description: Trigger Zone tile with TileFab

TpAnimZoneLoader

Was Triggered: False

Loading offset X -7 Y -5 Z 0

Preview->ON

Tile Fab: LoadableSection_TileFab (Tp

Show Offset Position Gizm

TpAnimZoneBase

Visualize the trigger zone.

Run 'Visualize'

Lock Position and Size

Lock All

Tile Sprite Clear: Ignore

TpFlexAnimatedTile

Asset has 1 animation

Clip to use: triggerZoneSprites

Play On Start ✓

Default Sprite

Clip Set: TpTriggerZoneSprites (Tp Spri

Use Animation Speed Over

Simulation Speed 2

Simulation Timeout 1000

TilePlusBase

Flags: None

Collider override: No Collider

Tags: loader

Reset Color

TpTiming: Inactive

Hide warnings and prompts ✓ Use Pop-up inspector ✓

Status: Scene AutoSave: Off, Sliders: Y

Note: the inspector looks different in Toolkit 2.x and later.

Here's a small block of TilePlus Tile code with attributes:

```
        /// <summary>
        /// Play animation on Startup
        /// </summary>
        [Tooltip("Check this to have the selected animation begin at Startup")]
        [TptShowField(0,0,SpaceMode.None,ShowMode.NotInPlay)]
        public bool m_PlayOnStart = true;

        /// <summary>
        /// Default sprite when selected animation sequence is invalid.
        /// </summary>
        [Tooltip("The default sprite when clip is null or of length 0")]
        [TptShowObjectField(typeof(Sprite),false,false,SpaceMode.None,ShowMode.NotInPlay)] //false means no
scene objects
        public Sprite m_DefaultSprite;

        /// <summary>
        /// The asset with the animation clips
        /// </summary>

[TptShowObjectField(typeof(TpSpriteAnimationClipSet),false,true,SpaceMode.None,ShowMode.NotInPlay,"",true)]
        [Tooltip("A SpriteAnimationClipSet asset.")]
        public TpSpriteAnimationClipSet m_ClipSet;

        /// <summary>
        /// Override clip animation speed if true
        /// </summary>
        [Tooltip("Check to override the animation speed in clip.")]
        [TptShowField()]
        public bool m_UseAnimationSpeedOverride;
```

There are attributes for normal fields like bool, int, float, string, and the various Vector/VectorInt types as well as special ones for object fields (because scene references are possible now), enums, custom GUI for specific tile types, and showing invoke buttons for methods, and more.

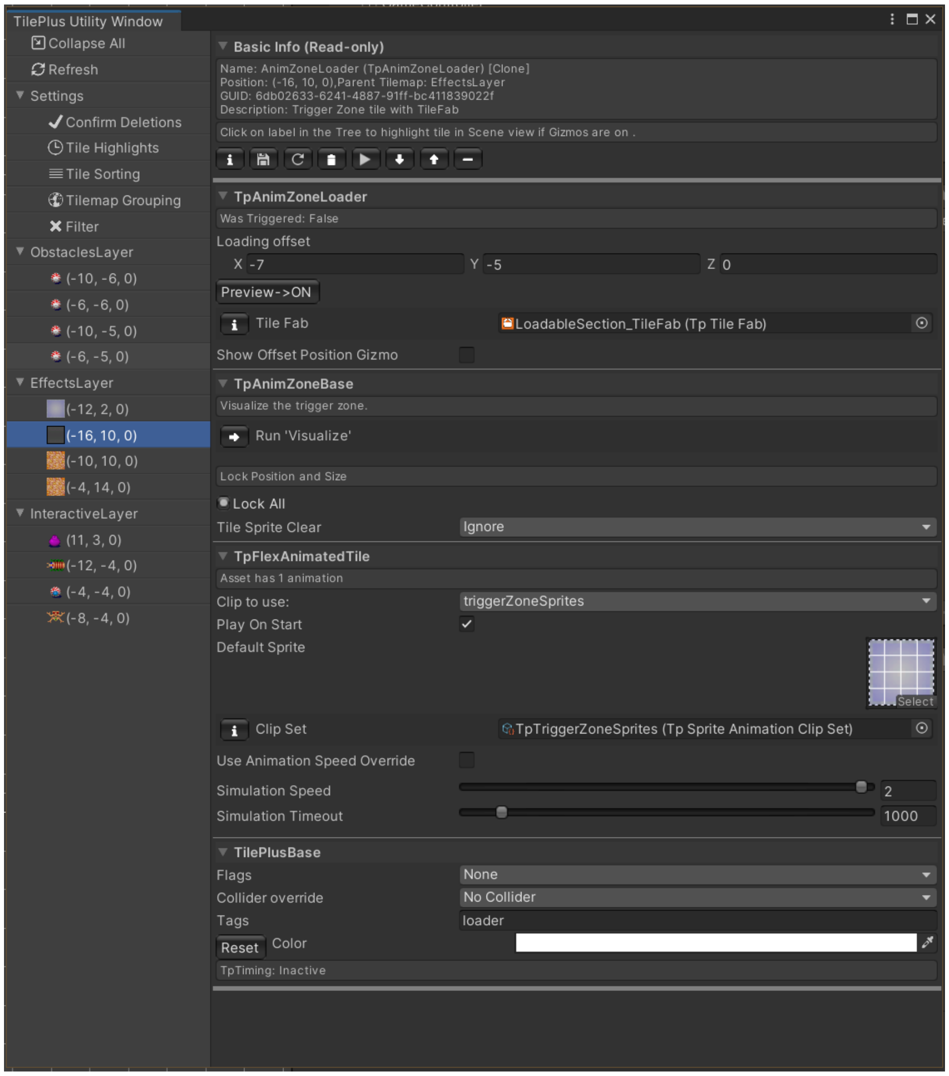
It Gets Complex, Part 2

There are several problems with relying on the Brush's Selection inspector to examine tiles: often it's just not that easy to locate which tile you wish to examine. This is especially an issue for TilePlus tiles because they can be invisible (if you wish). You have to select the correct tilemap as well, and that isn't always obvious or you might just forget.

Another problem is that you're relying on the brush. What if you want to use a different brush: you end up switching brushes repeatedly. It would be really nice to be able to view all the TilePlus tiles in one window, sorted by which tilemap they're on.

If you have Odin inspector installed, the Tools/TilePlus menu has an option called "Utility Window."

(note: in Toolkit 2.X and newer the 'Utility Window' is replaced by TilePlus Painter)


TilePlus Utility Window

Collapse All
Refresh
Settings
- ✓ Confirm Deletions
- Tile Highlights
- Tile Sorting
- Tilemap Grouping
- Filter

ObstaclesLayer
- (-10, -6, 0)
- (-6, -6, 0)
- (-10, -5, 0)
- (-6, -5, 0)

EffectsLayer
- (-12, 2, 0)
- (-16, 10, 0)
- (-10, 10, 0)
- (-4, 14, 0)

InteractiveLayer
- (11, 3, 0)
- (-12, -4, 0)
- (-4, -4, 0)
- (-8, -4, 0)

Basic Info (Read-only)

Name: AnimZoneLoader (TpAnimZoneLoader) [Clone]
Position: (-16, 10, 0),Parent Tilemap: EffectsLayer
GUID: 6db02633-6241-4887-91ff-bc411839022f
Description: Trigger Zone tile with TileFab

Click on label in the Tree to highlight tile in Scene view if Gizmos are on .

TpAnimZoneLoader

Was Triggered: False

Loading offset X -7 Y -5 Z 0

Preview->ON

Tile Fab: LoadableSection_TileFab (Tp Tile Fab)

Show Offset Position Gizmo: ☐

TpAnimZoneBase

Visualize the trigger zone.

Run 'Visualize'

Lock Position and Size

Lock All

Tile Sprite Clear: Ignore

TpFlexAnimatedTile

Asset has 1 animation

Clip to use: triggerZoneSprites

Play On Start: ✓

Default Sprite: Select

Clip Set: TpTriggerZoneSprites (Tp Sprite Animation Clip Set)

Use Animation Speed Override: ☐

Simulation Speed: 2

Simulation Timeout: 1000

TilePlusBase

Flags: None

Collider override: No Collider

Tags: loader

Reset Color

TpTiming: Inactive

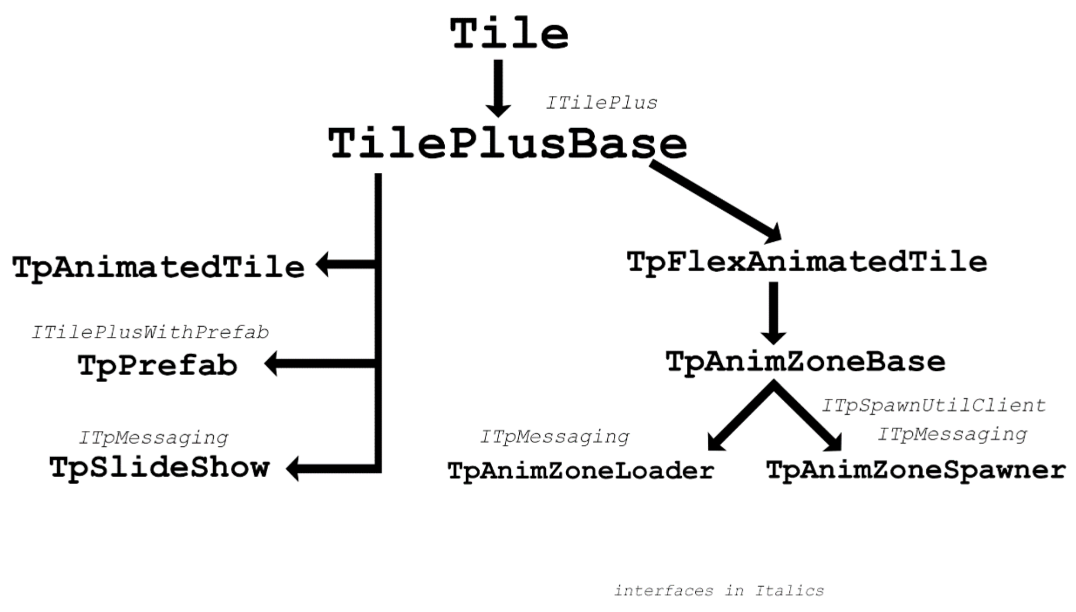
As you can see, it's almost exactly the same as the Tile+Brush Selection Inspector display, except that tiles are sorted by which tilemap they are on. Clicking on a tile on the left side both displays the tile information on the right, but also draws a box around the tile in the scene view if Gizmos are on. If the tile is a TpPrefab (a tile with a linked prefab) and the tile and the prefab are at the same place then a box is drawn around the prefab as well.

You'd be surprised how handy this is!

What's It Good For

Are tiles with instance data worth the trouble? This post discusses the tile class architecture, then we can talk about specific tiles.

Here's the class diagram:



(note: Toolkit 2.X and newer does not have TpPrefab, also, the fields described below may be different).

TilePlusBase

TilePlusBase subclasses Tile and implements the ITilePlus interface. The interface specifies a set of properties and a few methods required by other parts of the system. Tiles provided with the Asset Store version of the asset all derive from TilePlusBase. The CreateAssetMenu attribute of this class is commented out because there's no reason to ever create a tile based on this class.

Seven different tiles subclass TilePlusBase. Those on the left side of the diagram are not subclassed any further while those on the right all subclass TpFlexAnimatedTile.

TpAnimatedTile

This is a simple animated tile but adds one-shot animation and other features.

Public fields:

PlayOnStart: Begin animation when the game starts.

AnimationSpeed: Set animation speed relative to that set in the Tilemap

OneShot: Play the animation once, then stop.

ManualTimeout: A manual timeout for One-shot mode.

UnchangingSprites: Click this toggle ON if you don't intend to change sprites at runtime.

AnimatedSprites: A list of sprites to animate.

TpFlexAnimatedTile

TpFlexAnimatedTile is an upgraded TpAnimatedTile that adds the ability to have multiple animation sequences contained in an asset file. Once placed, you can select on a per-tile basis which animation sequence is used initially, and several other settings, including whether to play automatically when the Scene is loaded, and which sequence is in use. When using many animated tiles, the use of an asset is more memory efficient than TpAnimatedTile.

The asset file for this tile is TpSpriteAnimationClipSet, and its fields are mostly the same as those for TpAnimatedTile. You can create it from the Assets/Create menu.

TpPrefab

(deleted)

TpPrefab tiles have a linked Prefab. When the tile is painted on a Tilemap, the Prefab is automatically added as a child GameObject to the Tilemap at the same position that the tile is placed; it's immediately visible. Linked means that during Editor sessions, if you delete one of the pair then the other one is deleted. If you select, then move the TpPrefab tile then the linked Prefab is moved in a relative fashion, even if you've changed the Prefab's position. This works in-editor or at runtime; including optional easing built-in (refer to Programmer's Guide, Third-Party Libraries).

Why do this? You can paint prefabs right from the Palette. The tile's sprite appears in the Palette. You can have that sprite or icon as a visual cue in the Palette.

For scripters, it means that one can locate the prefab's GameObject by examining fields in the TPT tile. It's as easy as reading any other tile from a Tilemap component; and you can build actions that affect the prefab right into the Tile. No more searching for named prefabs or tags, just use the position. You can also find the paired tile from the Prefab with a TpLib method.

Public fields:

AutoInstantiatePrefab: Automatically place the prefab above the tile when it's painted.

PrefabToInstance: The prefab to use.

ForceLayer: Make the prefab's layer the same as it's parent Tilemap.

TileSpriteClear: Clear the sprite when painted, when the game runs, both, or not at all.

EasePrefab: If the tile is moved, the prefab position is Eased. If false, the prefab moves instantly.

EasingFunction: If Easing is used, which Easing function to use (Linear, EaseInQuad, etc.).

EaseSpeed: if Easing is used, how fast should the tile move.

TpSlideShow

TpSlideShow lets you display one Sprite at a time from a list of Sprites contained in an asset file. The initial Sprite to display can be changed, and you move from one Sprite to the next programmatically, with automatic wrapping or limiting. Wrapping means that incrementing from the last slide returns to the first slide (or when decrementing, from the last slide to the first slide) and limiting means that incrementing from the last slide or decrementing from the first slide has no effect. This tile is used for the background in the BasicTiles demo.

The asset file for this tile is TpSlideShowSpriteSet. You can create it from the Assets/Create menu.

Public fields:

SlidesClipSet: The TpSlideShowSpriteSet asset from your Project folder.

SlideShowAtStart: The name of the slide show to show when your game starts.

WrappingOverride: Override the 'wrap' setting from the TpSlideShowSpriteSet asset.

When inspecting one of these tiles using the Tile+Brush Selection Inspector you’ll see a dropdown where you can select which slideshow set from the TpSlideShowSpriteSet to be used at start. This can also be changed via code.

In the next post I'll talk about some of the more complex tiles.

What’s It Good For Part 2

The previous post discussed a few of the simpler tiles contained in the TilePlus Toolkit asset. There are several special tiles that provide much more complex functionality but require integration into whatever control code your project is using.

But before getting into that some background on the support library is necessary.

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 a TilePlus tile's StartUp method runs, the tile "registers" itself with TpLib. What this means is that information about the tile is cached in several data structures - here are a few:

- A dictionary of Tilemaps to tile instances and their positions.
- A dictionary that maps TilePlus tile tags to tile instances.
- A dictionary that maps TilePlus tile Types to tile instances.

TpLib has nothing at all to do with maintaining instance data in tiles. There are no hidden data files, no modification of scene files; nothing like that. As a matter of fact, with a couple of minor mods to TilePlusBase, TpLib can be completely omitted - although operating in the Unity Editor environment would be much more confusing.

Some of data structures' names sound sort of strange. Tiles with tags? Yep. One great thing about tags (and this is plural, any number of tags is possible) is that you can search for a tile without knowing which tilemap it is on. Tags are inherently built into the tiles themselves and are saved with the scene like any other instance data.

Likewise, being able to look for tiles based on Types is remarkably useful in practice. The TpLib implementation also allows you to filter on interfaces, so lookup is very flexible.

TpLib also includes support for sending messages to tiles and for tiles posting events. Why would you want to do this? Here's a hypothetical example:

Let's say that you have a top-down game where the player moves one tilemap square at a time. After the user moves the player you need to see if anything needs to react to that position change.

The way that's done in the TilePlus system is to message tiles with the position information. This can be done with tags, that is, send position information to all tiles with a certain tag. It can also be done with Types, that is, send position information to all tiles with a certain Type. If you want to message a specific tile and you know its map and position you can do that too.

If the tile examines the position information and wants to create an action it can post an event using TpLib. For example, the position might match that of the tile itself. This is easy to detect as TilePlusBase implements a way to internally store its position and parent Tilemap.

Tiles can post two types of events: Triggers and SaveData. Triggers denote any arbitrary action you want to implement and SaveData events indicate that the tile has some data that you want to save; for example, the tile is a waypoint and it's time to create a save.

Events are cached so that your control code (e.g., some sort of GameController monobehaviour) can react. The control code can subscribe to an event so that it becomes notified when an event is posted. At an appropriate time the control code can dequeue the cache and take whatever action you want.

The provided demo TopDownDemo shows how to use messages and events for NPC control and saving the position of a Player tile in the filesystem.

With that background, let's get on to the more specialized tiles.

Tiles Gone Wild

This branch of the class architecture starts with TpAnimZoneBase, which subclasses TpFlexAnimatedTile. The other tiles in this branch inherit the ability to have multiple animations selectable via the inspector or from code. Since all of these tiles implement a trigger zone, let's call them zone-based tiles.

First though, what problem are these tiles trying to solve? Certainly, the use of animated tiles is pretty easy to understand, but what are trigger zones?

The idea of a trigger zone is easy enough to understand too: when your Player or some NPC enters an area you want something to happen. TpAnimZoneBase provides the basic framework for establishing a BoundsInt which your code can use to determine zone entry, exit, etc.

By using the messaging and events systems in TpLib you can message a position to a zone-based tile and if it finds a match within its zone it can post an event. That's the basic functionality of TpAnimZoneBase.

TpAnimZoneBase

This is a base-class TPT tile that can be used to 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.

As you change the area positioning and size in editor (with the Selection Inspector), the tile's sprite is transformed to encompass the trigger area; useful for visualizing the size of the trigger area. Since the tile controls the transform, transform modification in the TilePlusBase section of the inspector is unavailable.

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 create an action when a playable entity enters or leaves a trigger zone. It's very general-purpose, so 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 tileset information.

TpAnimZoneLoader

This class is a subclass of TpAnimatedZoneBase, and inherits its fields and editor appearance. It's used to to provide information to make it easy to dynamically load archived Tilemaps from TpTileFab assets, which are created by the Tools/TilePlus/Prefabs/Make Tilefab or Prefab command. Public fields allow you to offset the loaded files from the tile position. You can also preview the loaded tiles.

The tile does not automatically load Tilemaps at runtime. Rather, you send a message to it via the 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 LoadImportedTileFab method handles all the details of the loading for you, so its 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.

Prefabs can be unparented or parented to a Scene Object by using the GameObject name or tag. SpawningUtil has built-in but optional 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 some additional fields.

As mentioned in an earlier post, the provided demo TopDownDemo uses all of the tiles discussed in this post.

TpLib Messaging, Events, and Persistence

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

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. Four overloads.
- OnTileEvent: An event that Monobehaviours or other scripts can subscribe to in order 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.
- AnySaveEvents: Property, Returns true if there are any queued SaveData events
- AnyTriggerEvents: Property, Returns true if there are any queued Trigger Events.
- GetFilteredEvents: Retrieves queued deferred events.
- ClearQueuedTileEvents: Clears the queue

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

SendMessage is used to send messages to tiles. Using method overloads, the message can be sent to a particular tile on a particular Tilemap, or to all tiles of a particular Type, or to all tiles with a particular tag. Tiles which need to be recepients of SendMessage must implement ITpMessaging.

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.

PostEvent

TpLib.PostTileTriggerEvent and TpLib.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. TpLib saves the tile instance that sent the event, and the event subscriber can retrieve it, and other cached events, with TpLib.GetFilteredEvents. The cache can be cleared with Tplib.ClearQueuedTileEvents.

These two TpLib 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 save/restore system 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 possible way to do that is for a tile to use TpLib.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 very 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 of these 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 method, and TR for the data type returned from a tile's 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 actually 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 and/or multiple 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){…}

public PositionPacketOut GetData(){…}
```

In this case, T and TR are both of type PositionPacketIn.

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.

If this seems overly-complex, the alternative for a general-purpose framework would be to use object, which introduces a lot of boxing and 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.

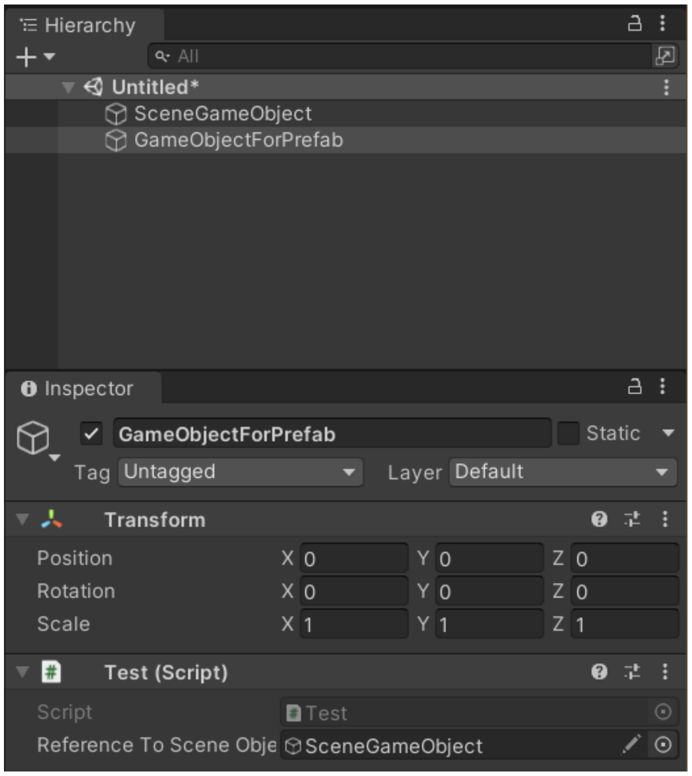
There are two demo scenes that show how to use ITpPersistence: TopDownDemo and SaveRestoreDemo. TopDownDemo also shows how to use ITpMessaging.

TilePlus and Prefabs

If you're a Unity3D dev you're familiar with prefabs. What is a prefab exactly?

A prefab is not unlike a scene file with the major difference being that scene references are lost. Here's a simple example:

This is the scene:



Hierarchy
Untitled*
SceneGameObject
GameObjectForPrefab

Inspector
GameObjectForPrefab   Static
Tag Untagged   Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Test (Script)
Script Test
Reference To Scene Obje SceneGameObject

Note that the Test script has a field called ReferenceToSceneObject. That's setup to reference the GameObject called SceneGameObject.

Here's what the Test script component looks like in the scene file:

```
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 8134853141899817304}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: f03a015a208c36a48901e47762c633c5, type: 3}
  m_Name:
  m_EditorClassIdentifier:
  m_ReferenceToSceneObject: {fileID: 132779707}
```

As you can see, the very last line holds the reference. After creating a prefab of GameObjectForPrefab we can look at the prefab file.

```
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 6779053189875886357}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: f03a015a208c36a48901e47762c633c5, type: 3}
  m_Name:
  m_EditorClassIdentifier:
  m_ReferenceToSceneObject: {fileID: 0}
```

Whoops! No more reference to the scene object.

This is exactly how it's supposed to work! Imagine if the prefab had maintained a reference to the scene object but you loaded this prefab into a scene without the scene object. Null-reference exception? Crash? Who knows. But it would be wrong.

This creates a problem for TilePlus tiles since they're scene objects. If you create a prefab out of a Grid+Tilemap with these new tiles, instantiating the prefab creates the famous "pink tiles" display.

But prefabs are pretty useful. Briefly, the solution involves creating a prefab programmatically and not allowing the user to create a prefab which has a TilePlus tile in it.

How do you stop a user from creating a prefab? It's actually pretty easy: just hook into the editor callback PrefabUtility.prefabInstanceUpdated and (simplified) if any TilePlus tiles exist on that map revert the prefab instance. This won't affect any other type of prefab creation.

How do you create a custom prefab? It's not unlike creating any other asset, although the code is a bit involved. Interested readers can peruse Plugins/TilePlus/Editor/StaticLib/TpPrefabUtilities.cs.

The interesting part of the process involves properly archiving the TilePlus tiles. Simply, a tilemap archive asset is created for each tilemap with the TilePlus tiles as sub-assets.

Similar to what you'd see in the scene file for a tilemap, lookup tables are used to save space for the oft-repeated values such as transform, color, and so on. In essence, it's a custom scene format for Tilemaps that avoids the limitations of Tilemap data storage in a standard Unity scene file.

One extra new asset is also created: a TileFab. This is a master asset which refers to all the tilemap archives. Hence, if you archive a Grid with, say, three child Tilemaps then the Tilefab has references to all three archives. What's this for?

TileFabs can be easily loaded into position in an existing tilemap. For the example of a grid with three child Tilemaps, the tiles from all three Tilemaps can be loaded into an existing map at the same time with any Vector3Int offset that you like.

This allows dynamic loading during gameplay. You can see this in operation in the TopDownDemo, and it uses a TpLib method called LoadImportedTileFab.

Also provided is TpLoader, a Monobehaviour component which can have a reference to a tilemap archive asset (as opposed to a TileFab) in the Project folder. Attached to the same GameObject as a tilemap component, this script can be used to load the archive to the tilemap when the scene is loaded or at any other time.

Unity Pools in 2021.1 and newer

Pools?

Pooling GameObjects has always been a thing you do in Unity. Perhaps you rolled your own pooler or obtained one from the AssetStore or Github.

But if not:

Pools use some extra memory to hold references to reusable objects that might ordinarily become garbage collected.

Constant re-constructing new versions of an object can end up fragmenting memory and causing more frequent garbage collection.

Performance can be improved.

Well, Unity has added object pooling as of 2021.1. How do you use it?

First of all, this isn't an easy way to get GameObject pooling for free. These pools seem to be intended as a way to pool common objects that are created over and over and would otherwise have to be GC'd. For example, you might be using a List<int> in a Monobehaviour Update method. Pooling that would be neat, eh? It's easy-peasy now.

And actually, with a little work, it's not too hard to create a prefab pooler too.

All Sorta Pools

What gives with all the different versions? There's ListPool, DictionaryPool, HashSetPool, ObjectPool, and a few others. All of these derive from the base class ObjectPool.

Even though ObjectPool is the most complex to use, it's also the most flexible, and the only one that's easy to observe for debugging.

Let's talk about how to declare a pool of List<int>

```
public ObjectPool<List<int>> m_IntListPool = new ObjectPool<List<int>>(
            () => new List<int>(),
            null, l => l.Clear());
```

This declaration specifies that the action on create (instantiating a new list) is new List<int>, the null for the second parameter means that there's no action on Get (asking for something from the pool) from the pool, and that the action on Release (returning to the pool) is to clear the list.

There are other parameters but this is a common pattern for this generic class.

If you replaced the 'null' in the declaration with l => l.Add(255) then a Get of a List<int> instance would would have one single element, set to 255. Not too useful in this case, and indeed, for most collections clearing the collection when Releasing the instance makes sense.

As a matter of fact, if the list holds references to other objects instead of ints, it really needs to be cleared, or the instance returned to the pool would still have the object references intact. That's a potential problem!

The bespoke collection pooling classes like ListPool, DictionaryPool, and HashSetPool all clear the collection on release.

Using the Pooler

To get a pooled list you can do one of two things:

```
var listPool = m_IntListPool.Get();
```

```
or if you need the pooled list for a short time, you can do this:
```

```
using (m_IntListPool.Get(out var list))
{
     //Do something with the list
}
```

When the list goes out of scope it's disposed. Dispose in this case just returns the list to the pool.

If you are not using 'using' then here's how to return the list to the pool:

```
m_IntListPool.Release(that_list);
```

Here's a property that you can use to get the status of the pool:

```
public string ListPoolStat => $"All:{m_IntListPool.CountAll.ToString()},
Active:{m_IntListPool.CountActive.ToString()},
Inactive:{m_IntListPool.CountInactive.ToString()}";
```

This property will show all pooled instances, all active instances, and all inactive instances. Why is this useful? When using the pools you need some way to be able to check if you are returning the pooled object to the pool. If not, you'll have a memory leak.

This becomes important if you try to use the more specific generic versions like ListPool. These have a static reference to the underlying pool, but it's declared internal so you don't have access to CountAll, CountActive, or CountInactive. You can't look at these in a debugger either, at least not easily.

Practical use

Although the simple examples shown here are instance fields, it's better to have these in a static class so that the pool is persistent throughout your app's lifetime.

In a monobehaviour, you can do things like this:

```
using (ListPool<int>.Get(out var list))
{
     //do something with the list
}
```

The pool itself is static, so every time you use the pool, lists are obtained from and restored to the same pool, even if you use this construct in different scripts: there's one pool for each Type. Clearly, since the scope of the list is known, you know it's been released to the pool when it's no longer being used.

But if you're using Get to obtain a pooled list and Release to return the list to the pool, you have be careful to pair these operations carefully. Two types of errors can occur:

- InvalidOperationException if you release an object more than once.
- Memory leaks if you don't release the object at all.

If you have the pool in a static class you can access it from anywhere and it can be monitored with CountAll, CountActive, and/or CountInactive. The using forms are generally only useful when you have a limited scope.

More Examples

For some good examples, download my free TilePlus Toolkit asset from the Unity Asset Store at https://u3d.as/2EJR

The file SpawningUtil.cs shows how to pool and spawn prefabs, including preloading.

The next release has more implementations of pooling:

TpLib.cs uses pooled Lists and Dictionaries.

The file TdDemoGameController.cs method UpdateAgents shows how to use the using statement with a list of tiles.

Using these pools carefully is a great way to reduce the load on the garbage collector.

Unity 2021.2 and New Tilemap features for Animated Tiles

Unity2021.2 introduces a few new methods related to animation. You can now get the current frame, the frame count, and the current running animation time for an animated tile. There are also added methods for setting the animation frame and time. I'm unsure why they don't implement a one-shot animation internally, but it's much easier to do now in your own code. (note: one-shot animation was added in newer Unity versions and Toolkit supports that too).

One of the things that's a bit tricky to do is to have a one-shot animation for an animated tile. I really wanted to have this feature for my TilePlus Toolkit (TPT) asset (free on the Asset Store).

Prior to 2021.2 one needed to compute a timeout based on the number of frames and animation speed. TPT tiles can have events, so it's possible to get a callback after this computed time, then refresh the tile: your GetTileAnimationData method can just return false, and animation will stop (that's oversimplified but basically that's all you have to do).

This actually works pretty well, but the new Tilemap method GetAnimationFrame makes it much easier. At the present time the current TPT version on the asset store (1.1.0) doesn't implement this but my internal version does.

Here's a simplified example. Note that this is code in a tile class.

```
private void Check()
 {
        var frame = m_ParentTilemap.GetAnimationFrame(m_TileGridPosition);
        if (frame < cachedCurrentClipNSprites)
            return;
        tpTiming.RemoveEvent(TpTimingEventType.Update,Check);
        animationOn = false;
        ParentTilemap.RefreshTile(TileGridPosition);
 }
```

What's not shown here is that the tpTiming Tile Plugin has been set to call *Check * every time that a Unity Update event occurs.

The integer cachedCurrentClipNSprites is set up outside what's shown here - it's the number of sprites in the clip, which there's no need to retrieve every single time that Check is invoked.

When the frame count reaches the number of sprites in the clip, tpTiming is told to remove the event so the Check method won't be called anymore.

Then the internal variable animationOn (part of the TpFlexAnimatedTile class) is set false and the tile is refreshed.

With animationOn = false, GetTileAnimationData also returns false when it is called from the Tilemap as part of the refresh, so there's no animation.

The TpFlexAnimatedTile code for GetTileData provides the static sprite to be displayed when there's no animation.

It's almost too easy, really.