

Technical Notes

Miscellaneous Information for delvers

- [Why Use New GUIDs?](#)
- [Preparing for Builds](#)
- [Limitations](#)
- [TpLibInit and TpLib Memory allocation](#)
- [Lifetime of a TilePlus tile](#)
- [Inhibiting Callbacks](#)

Why Use New GUIDs?

If you want to load the same TileFab multiple times programmatically you need to set the `newGuids` option true when using `LoadTilefab`. This is because a TilePlus Tile's GUID is used in `TpLib`, and unsurprisingly, GUIDs are expected to be unique.

Recall: if the same TileFab (or Bundle) is used repeatedly the TilePlus tile's GUID will also be reused, and that tile is ignored by the TilePlus system.

Why does this matter? It might not. If you're not using TilePlus tiles in a particular TileFab then it probably doesn't. But if you want to use TilePlus features like sending messages to TPT tiles or dealing with TilePlus events, or finding a tile by tag, Type, GUID, or interface, you'll find that these unregistered tiles can't be located.

If `newGuids` is set true when using `LoadTilefab`, all the TilePlus tiles have their GUIDs replaced with new ones. This means that they will register properly. Note that true is the default value for this parameter. Ordinary tiles are not affected by this option at all.

Note that the `NewGuids` feature isn't used by the Layout system since it does not repeat the use of any TileFabs. However, if you're not using the Layout system and are repeatedly loading the same TileFab to many places on one or more Tilemaps, it does matter only if these TileFabs include TilePlus Tiles.

TANSTAFL Dep't

(There Ain't No Such Thing as A Free Lunch, originally coined by noted SF author Larry Niven)

There's another issue, and it's not strictly a TilePlus issue, but more of a design issue regarding data persistence in saved game information.

Let's say that your game has some TileFabs containing Waypoints, like what's seen in `TopDownDemo`. This isn't about the chunking system where chunks are added and deleted frequently. Your game loads chunks of gameplay tiles, and some of the tiles are TPT tiles with data that you might want to save, e.g., the most recent Waypoint.

As the game progresses and the Player moves around, new sections are loaded as needed. When the Player moves over a Waypoint then that Waypoint is enabled (perhaps it changes appearance) and a game save is created: it's reasonable to persist the GUID of the Waypoint TPT tile.

The next time you run the game, it looks in its save data for the GUID of the most recent Waypoint and tries to enable it. But what would happen if the Waypoint was in one of the dynamically loaded TileFab sections? And how do you know what TileFabs to load and where to place them?

The next time the game is played, how do you restore the sections already loaded up till the most recently used waypoint and then place the Player at that proper Waypoint? You can't preserve the GUID of the waypoint in a loaded section since that GUID changes each time that the TileFab is loaded.

This leads us to a great use for the ZoneRegistrations accumulated in a ZM: they contain all the information that you need to save to reconstruct the game world's TileFabs and remap GUIDs correctly.

When you load a TileFab using LoadTileFab and provide the ZM instance as one of the parameters, the ZM is sent the results of the load. It adds this information to a "breadcrumbs" list in the form of ZoneReg instances. The breadcrumbs are therefore a list of information about which TileFab assets were loaded and where they were placed.

This list has instances of serializable type ZoneReg, and each instance contains an index, the asset GUID, and the offset and rotation parameters which were used when loading. All the ZoneReg instances, in load order, can be retrieved using the GetAllZoneRegistrations property. The last N Registrations can be obtained with GetLastRegistrations.

However, there's no one way for these to be used since each game's requirements are different. However, each ZoneManager instance has other two useful methods that you can build on or use as an example:

- GetZoneRegJson creates a JSON string with all the serialized ZoneReg instances that you can save.
- RestoreFromZoneRegJson takes that exact JSON string and recreates all the TileFabs specified within.

The serialized ZoneReg instances are in ascending order of creation.

Get/Restore works for the simple (but common) use case of wanting to restore everything. It's possible to only save some of the ZoneRegs, but that's not handled in any built-in way.

A good starting point to develop a different approach is to use GetAllZoneRegistrations and process it yourself.

The Remapping

RestoreFromZoneRegJson also handles remapping GUIDs, using TpZoneManagerUtils.UpdateGuidLookup. That code scans the asset registrations and creates a lookup table mapping the GUIDs from the TilePlus tiles in the loaded ZoneReg entries with those in the tiles placed from the Bundles. In other words, create a mapping from the GUIDs you may have saved as part of the game state to the new GUIDs that were created when the TileFab was loaded during the execution of RestoreFromZoneRegJson.

The method `TpLib.GetTilePlusBaseFromGuidString` will try to use the lookup table if it can't find a match in the primary lookup contained in the `TpLib` static class. That solves the changed GUID issue. So that saved Waypoint GUID 'points' to the proper Waypoint if it was in a dynamically loaded section of the Tilemap.

If you're 'rolling your own' there's a property in `TpLib` which allows you to provide your own Guid-to-TilePlusBase mapping. `TpZoneMangagerUtils.UpdateGuidLookup` can be used to create the mapping. It also creates a reverse mapping, which is needed when Zones are deleted.

It should be noted that the `TileFab` and `Bundle` assets must be part of the build. For example, when a `TpAnimZoneLoader` tile is part of a scene, the asset reference in that tile causes the `TileFab`, referenced `Bundles`, and other associated assets such as textures for the tile sprites, to be included in the build.

However, if you're loading everything dynamically you need to ensure that the assets you need are available in the build. If you're reading this far you probably know how to do that, but you could place them in a `Resources` folder or reference them all somehow in a `Monobehaviour` component attached to some `GameObject` in at least one scene.

Finally, if you are using the TPT persistence scheme, do not try to Restore to TPT tiles prior to remapping GUIDs. This is important since the restore process depends on the GUIDs being mapped correctly.

Preparing for Builds

When using `ZoneManager` and `ZoneLayout`, you need to ensure that the referenced `TileFabs` are correctly included in a build. This is because the system must use `TileFab` GUIDs to locate each `TileFab` when using `RestoreFromZoneRegJson`.

When calling `EnableZoneManagers` one of the optional parameters is a map from `TileFab` GUID to `TileFab` asset instances. If you're using `ZoneLayout` the `TileFabs` can be easily located at runtime by examining the layout instances. This can be seen in the `Chunking` demo program.

If that map isn't provided, then the `Resources` folder is examined, and a mapping is created automatically. This occurs only once.

If you have references to the `TileFabs` somehow otherwise included in a build and not in `Resource` folders, then you can create this mapping yourself.

If you don't provide the mapping and you don't have the `TileFabs` in a `Resources` folder, then they won't be located and the loading of such `TileFabs` will fail: this can confusingly work just fine in the Editor and fail in a build.

Limitations

Code Changes

Changing code in subclasses may result in missing references in already-placed clone tiles. This is like what happens in the `GameObject` 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.

Move Function and Overwrites

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.

Mostly Tested in Top-Down Applications

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`, and to move a TPT tile use `TpLib.CutAndPasteTile`.

TpLibInit and TpLib Memory allocation

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.

TpLibInit

This is a scriptable object in a resources folder that you can use to set up memory allocations and certain optional features. It's in TilePlus/Runtime/Resources/Tp. TpLib startup code examines this asset and uses the following fields:

- Active: the asset is ignored if this is false.
- RefreshRequestsPerUpdate: Maximum number of Tile Refresh Requests per-internalUpdate
- TargetFrameRate: useful for diagnostics. Shown in System Info editor window.
 - If zero, adaptive features are disabled (not discussed herein)

- UsePlayerLoop: if true, TpLib modifies the PlayerLoop, if false, uses an Awaitables-based updater.
- ResizeMemory: if true, use the following info to resize memory.
 - MemAlloc: various fields for memory allocations. Read up before changing these.
- InhibitTilemapCallbacks. Normally false. Useful for debugging. Bad for production.

PlayerLoop

Using the PlayerLoop timing is the best choice unless it inteferes with your project code somehow. When TpLib shuts down (app closes) or (Unity state change Play->Edit) it terminates the Awaitables-based updater OR restores the default Player loop.

If you are using PlayerLoop in your app and this doesn't work for you, subscribe to the `OnResetPlayerLoop` callback (it's not an Event, only one subscriber allowed.). Before restoring the default Player Loop, this callback is invoked.

If the callback exists and returns false then the default Player Loop is restored. If the callback returns true then it's assumed that you handled this situation yourself and no further action is taken.

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`.

The cloning only happens once: when the tile is placed by an editor tool like the `UTE` or `Tile+Painter` or by code.

- Move the tile from one place to another (Cut/Paste): no cloning
- Copy/Paste: the new tile is cloned in `TpLib`.

Since the clone is referenced in the `Tilemap` now, it's saved with the scene.

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.

If you message the TPT tile it may perform other actions such as messaging other tiles, tweening, or even deleting itself. But aside from your code causing such actions the tile can't do anything since it doesn't get any events such as `Update`.

Events

- `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. See [this](#).

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.

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`.

Inhibiting 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.

To have this property set true at runtime, use [TpLibInit](#).