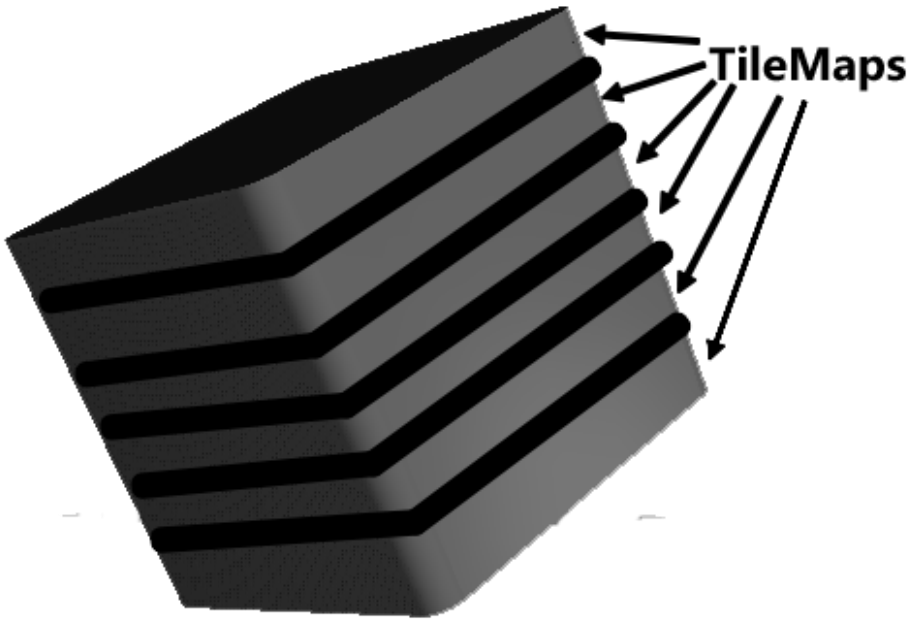
Advanced TileFab Use

Contents

Introduction

A TileFab can encompass all the tiles and Prefabs contained within a Tilemap hierarchy (Grid with child Tilemaps). The simplest use for this sort of a TileFab is to just load the whole thing at once to populate an area of a scene with tiles in one operation.

However, when you create a TileFab from a Grid Selection they're more like rectangular pieces of layer cake, as seen in this crude illustration:



Rather than repeat "TileFab from a Grid Selection" let's just call this a Chunk and reserve the name "TileFab" for one created without a Grid Selection. This document is mostly about ways to use Chunks.

A Chunk ***is*** a TileFab, but there are several important differences regarding how position data is stored.

TileFab

- The CellBounds (a BoundsInt) of each Tilemap are used to limit what is stored in each Bundle.
- The stored BoundsInt for each Bundle is the CellBounds of each Tilemap.
- The stored positions of tiles and Prefabs in the Bundles are the raw positions from the Tilemaps.
- The m_FromGridSelection field is false.

Chunk

- The Grid Selection defines the limiting area.
- The stored BoundsInt for the Chunk always has a zero-based origin. The size is that of the Grid Selection.
- The stored positions of tiles and Prefabs are adjusted to have a zero-based origin relative to the lower-left corner of the Chunk.
- The m_FromGridSelection field is true.

This means that a Chunk's size is always defined by its BoundsInt's size. All stored positions in a Chunk are relative to the position of its BoundsInt, i.e., all positions are relative to the lower-left corner (the Position property) of the Chunk. Or you can think of it as "relative addressing" since you can paint a Chunk anywhere but the tiles within the chunk retain the same relative positions.

Conversely, a TileFab's stored positions are the raw positions directly from the Tilemap. If you paint a Tilefab with the Painter, all objects' (tiles and Prefabs) positions are added to the mouse pointer position.

TileFabs are good for saving and loading entire Tilemaps. In real life, Chunks are more useful.

If you paint a Chunk or one of its Bundles using Tile+Painter, you'll notice that the mouse pointer position is always at the bottom-left of the group of tiles in the Chunk. This is because a BoundsInt's position is the lower-left corner. This is important to remember, as you'll see later.

Ways To Use Them

- **Load** a Chunk (or any TileFab) anywhere you want.
- **Paint** TileFabs, Chunks, or Bundles using Tile+Painter.
- **Load** automatically as the camera moves.

Loading:

The easiest way is to use a TpAnimZoneLoader Tile. One can paint one of those Tiles on a Tilemap, view its fields using Tile+Brush or Tile+Painter, and set up a trigger zone. Your program periodically sends the position of your Player (or something else) to this Tile using the TpMessaging library. The Tile posts a Trigger event when that position is within the trigger zone. Your code observes this and calls a method in TileFabLib to load some TileFab. This sort of use can be seen in the TopDownDemo program.

You can also call the LoadTileFab method in TileFabLib. This provides fine-grained control over the process, including optional filtering during the load process.

Painting: See the Tile+Painter documentation.

Automatically: Monitor the position of your camera and add/delete chunks as they move in and out of the camera view. There's an example of this in the Chunking demo.

Some TileFab Theory

Bundle: an archive of all the tiles and Prefabs for a single Tilemap.

TileFab: references a group of Bundles.

Let's review what type of data are in a Bundle asset:

- A List of TilePlus Tiles.
- A List of Unity Tiles.
- A List of Prefabs.
- Indexed Lists of Tile Flags, transforms, and Colors.
- The BoundsInt for the group of tiles as calculated for TileFabs or Chunks.

It's not difficult at all to take all this information and populate a Tilemap with the tiles in the Bundle. Even a caveman could do it! There's a method in the Bundle asset called *TileSet* which unpacks the various Lists in the asset and returns a List of data items, each with the tile reference and Flags, transforms, and Color for each tile. Methods in TileFabLib and TpZoneManager provide higher-level APIs for both TileFab and Chunk use.

Caching

Bundle assets cache all non-TilePlus tiles the first time that the TileSet method is used. This increases performance when the same Bundle is used repeatedly and most of the tiles are 'normal' Unity tiles (basically, anything that isn't a TilePlus tile). There is a method that clears the cache if you need to.

Prefabs and TilePlus tiles are never cached. This is for several reasons:

- There won't be large numbers of these compared to normal tiles.
- Normal tiles take much longer to unpack (since there are usually many more of them).
- Prefabs need to be instantiated each time they are placed.
- TilePlus tiles need to be cloned at a bare minimum.
    - May need GUIDs changed.

Software Components

Support for TileFabs can be found in Plugins/TilePlus/Lib and comprises three main scripts: TileFabLib, TpZoneManager, and TpZoneLayout or a subclass thereof. Each of the three components present successively higher-level APIs for using TileFabs.

TileFabLib is a static library which provides the basic functions for loading TileFabs and Bundles to Tilemaps in a scene, during Editor sessions and at runtime. For example, if you'd created a TileFab from a complex scene with 10 layers of Tilemaps comprising thousands of tiles, you can paint that TileFab to a Scene using Tile+Painter, or load it to a Scene in a running app. It's also used to create TpZoneManager instances at runtime.

TpZoneManager is a Chunk Manager. Instances of TpZoneManager Scriptable Objects are created at runtime. It has an API for chunk management, and loading/unloading chunks.

TpZoneLayout is a MonoBehaviour component that queries a single TpZoneManager instance**:** it's a base-class for loading and unloading chunks as they move in and out of the Camera range. This is a basic implementation, and TpZoneLayout can be subclassed or rewritten by you to work differently if you want.

TileFabLib

TileFabLib.**LoadTileFab** loads the Bundles referenced by a TileFab asset. The TileFab asset has references to one or more Bundle assets, which include all the information required to recreate the tiles.

TileFabLib.**LoadBundle** loads the tiles from a single TpTileBundle asset. You normally don't need to use this, although you can if you just want to load tiles from a single Bundle (this is how Tile+Painter paints single Bundles). Otherwise, use LoadTileFab: a TileFab is created even if you only bundle one Tilemap and LoadTileFab is easier to use.

If you were only loading one Bundle to a Tilemap you'd intrinsically know which Tilemap to use as a target for placing the tiles extracted from a Bundle. This is why you can use the Tile+Painter to paint a Bundle anywhere you want to.

However, the normal use case, and the one supported by TpZoneManager, is to use a TileFab. That asset is a wrapper for any number of Bundles, along with the tags and/or names for the source Tilemaps that the Bundles were created from.

Why archive these two items? Because that information tells you what the Tilemap is for each Bundle; that is – where do you paint a Bundle that's part of a TileFab?

Using the original Bundle assets' names and tags, *LoadTileFab* tries to locate the target tilemap; first with its tag, then by name. *For faster performance*, a mapping between stored Tilemap names and specific Tilemap instances can be passed to LoadTileFab. This avoids searching for GameObject tags or names.

Each TpTileBundle asset is evaluated and the destination Tilemap must be found. The search is performed in the following order:

- Provided by the name-to-Tilemap instance mapping. (fastest)
- Provided by using the Tag to Find the parent GameObject of the Tilemap.
- Provided by using the Name to Find the parent GameObject of the Tilemap. (slowest)
- If a TPT tile reference is passed-in to LoadTileFab then the parent Grid of the tile is found, and the Tilemaps which are children of that Grid are examined for matching names (variable speed but faster than using Find).

If all those methods fail, then LoadTilefab silently fails, nothing happens, and no tiles are loaded from that TpTileBundle asset.

You can see this in action if you try to paint a TileFab containing multiple Bundle assets and one of the named Tilemaps isn't present. If no Tilemap is located, then there's no way to load the tiles and that Bundle asset is ignored: no preview or painting. If no Tilemaps are located for any of the Bundles, then nothing is painted at all, and no previews are possible.

If you're using the Tile+Painter to paint a single Bundle's tiles, the destination Tilemap had already been selected in the leftmost column. Hence, you can paint a Bundle on any Tilemap quite easily.

If you want to ensure that loading TileFabs or Bundles recreate what you're expecting, bear in mind that the Tilemap Component and Tilemap Renderer Component settings matter too:

- Tilemap: Frame rate, Color, Anchor, Orientation
- Tilemap Renderer: Sort Order, Sorting Layer and Order in Layer

If any of these are different than the setup when you originally archived the tiles, then the resulting visual appearance after the loading will be different.

Using LoadTileFab

Let's examine the method parameters for LoadTilefab:

tileParent	The parent of the calling object (Only needed if calling from a TPT tile, otherwise ignored and can be null)
tileFab	a TpTileFab asset reference
offset	The Offset from tiles' stored positions – you use this to set the location where you want the tiles placed. For example, if you used Vector3Int(100,200,0) then the tiles will be placed relative to that location.
rotation	optional rotation – unimplemented
fabOrBundleLoadFlags	A set of control flags for LoadTileFab. See below.
filter	a Func providing a struct with information about the item being evaluated, returning a bool. If non-null, this Func is used to filter out tiles that you don't want.
targetMap	A dictionary mapping tilemap names as found in the TileFab to actual Tilemap instances. Used to override the names from the Tilefab.
zoneManagerInstance	If using a ZoneManager, pass its instance.
Returns:	Instance of TilefabLoadResults class. If null is returned, then there was an error of some kind.

fabOrBundleLoadFlags is a value from the FabOrBundleLoadFlags enumeration. These are Flag enums so they can be ORed. The combined value 'Normal' is the common case.

None	(usually not used)
Load Prefabs	Normally true
Clear Prefabs.	Normally false
Clear Tilemap	Normally false
Force Refresh	Normally false.
New GUIDs for TPT tiles	Normally true
FilterOnlyTilePlusTiles	Normally true
NormalWithFilter	LoadPrefabs | NewGuids | FilterOnlyTilePlusTiles
Normal	LoadPrefabs | NewGuids

The meanings:

forceRefresh	Executes Tilemap.RefreshAllTiles after the tiles are loaded.
loadPrefabs	Load any prefabs found in the TileFab
clearPrefabs	Delete all prefabs attached to the target Tilemap's GameObject.
clearTilemap	Clear all tiles on a target Tilemap prior to loading new tiles.
newGuids	Provide new GUIDs for all TilePlus tiles. Use when placing these assets at runtime to avoid duplicate GUIDs. There's a discussion about this a bit farther down.
filterOnlyTilePlusTiles	If the filter is provided then the filter is only applied to TilePlus tiles, which is often sufficient and saves much time.

Most of these are easy to understand and the `Normal` value for fabOrBundleLoadFlags is usually a good choice. You use the **offset** to set the origin for placement. As mentioned earlier, position information for Chunks and TileFabs are handled differently, but this is mostly handled internally. For example, in Tile+Painter the tiles in a TileFab or Bundle assets are placed using LoadTileFab. The offset value is the mouse position converted into Tilemap (Grid) positions. You can see how that's done by examining the code in TpPainterSceneView.

TileParent is null unless this is being called from within a TPT tile, such as the TpAnimZoneLoader. It's used when trying to find the painting Tilemap, as mentioned in the preceding section.

ZoneManagerInstance can be left null (the default) if you're not using it. If not null, the results of the loading operation are archived in the TpZoneManager instance that you provided.

NewGuids: A new GUID is created for each TilePlus Tile in the Bundle. When the Bundle is created, the GUIDs of the TilePlus tiles are saved in the Bundle, and when the Bundle is painted by Tile+Painter or via code, the GUID of the painted TilePlus tiles are the same as when the tile was archived. But this isn't always desirable. For example, if you wanted to use a TileFab repeatedly in a game you'd end up with the same GUID for multiple tiles.

That's an issue because the GUIDs are supposed to be unique: the TilePlus system rejects TilePlus tiles with duplicate GUIDs and an error message is issued. If **NewGuids** is false, GUIDs are unchanged.

Filtering

What's the filter for? There are several uses for this feature: limiting the number of tiles loaded to those within a certain area, extracting information from the loaded tiles, eliminating certain tiles, changing the Color or transform for a tile, or for replacing a tile with different one (like for seasonal events). You can also adjust the position of a Prefab.

As input, the filter is provided with:

- An object containing information about the Unity tile, TilePlus Tile, or the prefab asset with position information.
- A value from the FabOrBundleFilterType enum. This tells you what the object is.
- The BoundInt for the Bundle.

Note that for Prefab assets or Unity tiles: these are the actual project assets so don't modify the asset. Changing to a different tile (not TPT) asset can be OK if it contextually makes sense.

When the filter receives a TPT tile it's a clone so you can change its fields if you want to.

Each of these objects has different types of information:

- Unity tiles: TileSetItem: a tile reference (as TileBase), position, color, transform, and tile flags Again, note that the tiles are **assets**.
- TilePlus tile: the TPT Tile reference and the position.
- Prefab: the prefab **asset** reference and the placement position.

If you want to change values for the actual objects such as the Unity Tile or the TPT tile, please be aware that:

- Don't change from a TilePlus tile to a non-TilePlus tile or vice versa.
- Prefabs and Unity tiles are project assets and should not be altered.

- TilePlus tiles should be clones. If they are not, they auto-clone when painted on the Tilemap, which depends on the settings of TpLib.MaxNumClonesPerUpdate. To clone a TilePlus tile, call its Cloner method with the newGuid parameter = true.
- Since there are so many possibilities, this sort of use is unsupported aside from (our) bugs.

Associated Demos

The best way to see how filtering works is to examine the various examples in the TileFabDemos folder.

TwoLayerMapDemo

The simplest demo in this set is the **TwoLayerMapDemo** scene from the Demos/TileFabDemos/Basic/DemoScene folder. The scene in the DevelopmentScene folder was used to create the TileFabs for this example. That scene can run but isn't the demonstration you're looking for.

The simple script attached to the GameController lets you play with different options, including multiple iterations of loading with a variable offset. The results from LoadTilefab are printed to the console. You'll note that the different options affect load times.

When trying this in the editor with multiple iterations you might notice that the first iteration is much slower than the rest. This is an Editor artifact caused by Domain Reloading when you click Play. Go to Project Settings and check the Enter Play Mode Options toggle, then ensure that the Reload Domain toggle is UN-checked. Re-running the demo will show a very different result. In other words, in a built application this won't be an issue.

If you check the "Filter Out Blue Thing Tiles" checkbox, then a filter is used to eliminate all the Blue tiles. You can be much more specific than this, for example, you could filter out all TilePlus tiles with the tag "ZOT" or those of certain colors or within a certain Bounds – it's useful if you wanted to replace one variety of tile with another at runtime since the filter receives position and tile information which you can use to place new tiles where you filtered out others.

Note that TwoLayerMapDemo's Tilemaps are set up the same as those in the Development scene. At the most basic level, this ensure that the tiles are loaded to the proper Tilemaps. But the Tilemap Renderers have different 'order in layer' settings. If this were different in the TwoLayerMapDemo scene the maps would load properly but the visual appearance might be different.

If you're trying to benchmark the timings, please ensure that there's no selection shown in the Inspector, and that the Tile+Painter and the Unity Palette windows are both closed. Open the TilePlus Configuration editor, uncheck 'Informational' and 'Warnings', then click the RELOAD button.

Retargeting

The targetMap parameter takes a string-to-Tilemap Dictionary. If provided, the Dictionary is used to look-up the Tilemaps specified in the TileFab rather than searching for named GameObjects or GameObject tags. This is much faster (duh).

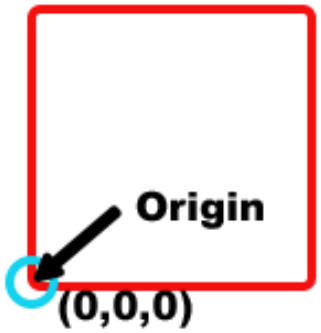
The **TwoLayerRetargetingDemo** scene is essentially the same as that in **TwoLayerMapDemo**. However, the Tilemap names are different, so a mapping is created in code.

Why Care About Grid Selection?

One might recall from the User Guide that you can create TileFabs and Bundles using a **Grid Selection**. A Grid Selection is when you use the Palette to make an area selection in a Tilemap. If the Bundler tool sees an active Grid Selection, it will ask if you want to use it. If you agree, the selection is used to limit what is archived. Normally, the Bundler tool grabs every tile on a Tilemap, and every Prefab parented to the Tilemap's GameObject.

One of the things that's stored in a Bundle is the BoundsInt for what was archived. When archiving a GridSelection, that selection supplies the BoundsInt.

Say that you want to paint (load via TileFabLib) a Chunk at Vector3Int.Zero. All the tiles in the bundle will be painted relative to Vector3Int.Zero as shown below. In other words, the stored locations of the tiles in the bundle are pure Tilemap "grid" coordinates regardless of where the Tilemap's origin was placed.



Note that all the tiles and Prefabs are placed relative to the position of the Chunk at the lower left corner, i.e., to the upper-right of the placement position for the Chunk.

Having a good understanding of this concept is important if you don't want to get confused by what's next.

TpZoneManager

TpZoneManager is a chunk management subsystem. Your code interacts with instances of the TpZoneManager class, which are Scriptable Object instances created at runtime by using TileFabLib. The ChunkSystemDemo scene illustrates how to use the system.

TileFabLib implements some simple management features for creating, locating, and destroying TpZoneManager instances. Let's call them ZMs to avoid me having to type TpZoneManager over and over.

To save memory, the data structures used for management are not initialized until you enable the subsystem by setting the TileFabLib.EnableZoneManagers property to true. This enables the subsystem and allocates memory to manage the ZMs, which you create with TileFabLib. CreateZoneManagerInstance.

For efficiency, the GUID remapping data tables discussed earlier are also maintained within TileFabLib.

```
public static bool CreateZoneManagerInstance(
      out TpZoneManager? instance,
      string iName,
      Dictionary<string, Tilemap> targetMap);
```

TileFabLib.CreateZoneManagerInstance requires that you provide a string as a unique identifier, and a mapping between Tilemap names and Tilemap instances. The ZM instance is placed in the out-parameter *instance*.

Don't use the same Tilemaps with different ZMs.

Once created you can always get this specific ZM instance by using TileFabLib.GetNamedInstance.

Delete a named ZM with TileFabLib.DeleteNamedInstance. If you delete a named instance and there are no more, TileFabLib disables the subsystem and releases memory subject to GC.

Once you have a ZM, you initialize it with ZM.Initialize, providing the chunk size, the world origin coordinates (in the Tilemap address space), and an initial number of chunks expected.

For a **single** ZM instance, constrictions apply:

1. Chunks (TileFabs) all need to be all the same size: square, with even-number dimensions such as 64 x 64.
2. The smallest chunk size is 4x4.
3. Chunks are always aligned to the ***super-grid*** defined by the chunk size.
4. A Tilemap should not be used with more than one ZM. A ZM is unaware of Tilemap positions that may have had Zones filled or deleted by another ZM, and conflicting chunk sizes between different ZMs would naturally be problematic.

Each ZM can be set up completely differently. Just don't cross the streams…. uh, share Tilemaps between ZMs.

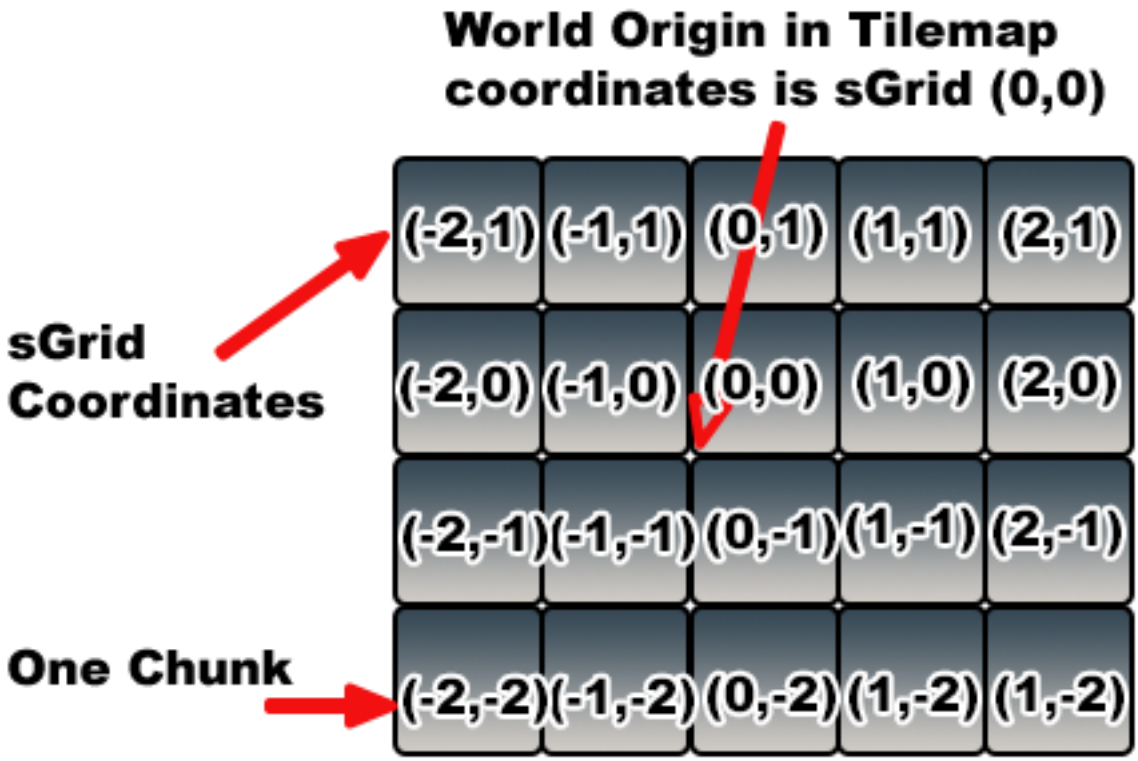
When you provide a ZM instance to TileFabLib.LoadTileFab it sends the results of the load to the ZM, which creates a ZoneRegistration.

Why are Chunks Square?

Chunks must be even dimensioned due to integer math. If a Chunk had an odd size, division could induce a positioning error. They don't *have* to be square (they could be rectangles), but for this implementation they must be square with even dimensions because it makes the logic and math easier and much faster. It's also more intuitive.

The Super-Grid

When you initialize a ZM with a chunk size and world origin you're defining a higher-dimension or super-grid (sGrid) virtually layered on the normal Tilemap Grid. In the illustration below, one Chunk is any number of Tilemap locations, from 4x4 to 128x128 or any reasonable value. If the chunk size is 4x4 then this 5x4 super-grid below actually comprises 20x16 tile positions. The position of a chunk is its origin, the lower-left corner.



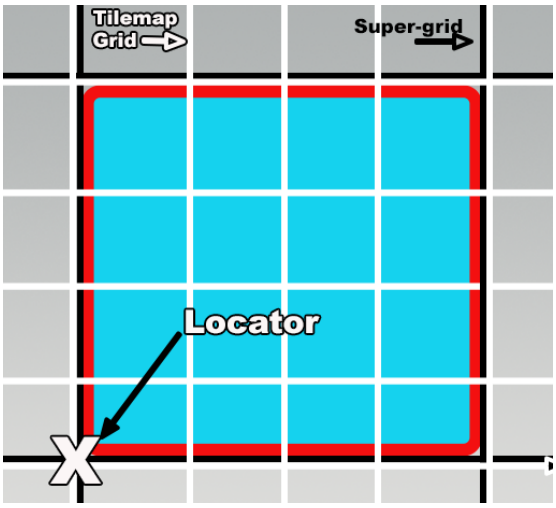
The sGrid coordinate system's origin is the same as the world origin provided when the ZM instance is created. It's also important to understand that each sGrid only applies to one ZM, even those created with the same chunk size and world origin. ZMs are completely unaware of each other, by design.

The array of Chunks is a sGrid virtually layered on top of the Tilemap's grid. Array elements are called **Zones** and are addressable using **Locators**. Each Zone comprises one Chunk.

Locators are **RectInts**, which are a 2D version of a BoundsInt: a RectInt is also a struct and its position is also in the lower-left corner. They're used throughout ZMs to define Zones, and to access and/or query data from a ZM instance.

Each Locator must be aligned to the sGrid of the ZM. This means that its position must be divisible by the chunk size, e.g., a chunk size of 4 means the sGrid addresses must be divisible by 4. The size of each locator is usually the size of a chunk, but doesn't have to be, as we'll see later.

Here's a detail of one Zone, outlined in red. Each Zone is aligned with the sGrid. The size of each Zone is the size of a Chunk. In this example, the size of a Chunk and of Zones is 4 x 4 Grid locations. Hence, the sGrid is 4x the Tilemap Grid.



The Locator of a Chunk can be used to **locate** a particular chunk on the sGrid just like a Tilemap position addresses a specific location on a Tilemap.

There are several ZM instance methods to convert between Tilemap Grid address space and sGrid address space.

- GetLocatorForGridPosition takes a Tilemap Grid position and returns a Locator.
- GetLocatorForWorldPosition takes a world coordinate position and returns a Locator.

There are also instance methods to check for and convert to sGrid alignment.

- IsAlignedToGrid takes a Tilemap Grid position and returns true if it's aligned to the sGrid.
- AlignToGrid takes a Tilemap Grid position and returns one that's aligned to the sGrid.
- GetLocatorForSgridPosition takes a sGrid position and returns a Locator.
- GetGridPositionForSgridPosition takes a sGrid position and returns a Tilemap Grid Position.

You don't need to use the sGrid, just be aware of it. But if you do need the conversions offered by the last two methods above, they're available for your use. Note that the conversions will produce different results on ZMs with different chunk sizes and/or world origins. In a way, a chunk is like a giant tile and the sGrid is like the Tilemap grid with bigger element sizes.

The GetLocator for GridPosition and WorldPosition methods have an *align* parameter which if true (the default value) will automatically use the AlignToGrid method on the Tilemap Grid position.

Referring to the above illustration, any Tilemap grid coordinate within the Zone will return the Locator for the Zone when AlignToGrid is used. IsAlignedToGrid will only return true when the Tilemap grid coordinate matches the exact position of the Locator, shown here as X.

The GetLocator methods allow you to also pass in a Vector2Int *dimensions* parameter. This lets you create arbitrary-sized Locators. Normally, Locators are the size of a chunk. If you do not pass-in dimensions, that's the size that's used. Using a different dimensions value allows you to size a Locator however you want.

Why do that? Let's say you want to find out which Zones are inside a Camera Viewport, and which are outside. The Viewport is (usually) bigger than a single chunk. Passing in the dimensions of the Viewport creates a Viewport-sized locator. That's handy for use with GetZoneRegsForRegion and FindRegionalZoneRegs.

GetZoneRegsForRegion takes a Locator for input and returns a list of all the ZoneRegs within the Locator.

FindRegionalZoneRegs takes a Locator for input and populates a HashSet with all the individual Locators within the input Locator and populates a List with all the individual ZoneRegistration instances found outside the input Locator. This is a specialized method that's used by another component called a ZoneLayout, we'll discuss that in a later section.

To get individual ZoneRegistrations, use these instance methods:

- GetZoneRegsForGridPosition
- GetZoneRegsForWorldPosition
- HasZoneRegForLocator
- GetZoneRegForLocator

Camera Projection

A ZM is agnostic to the camera's projection mode. However, there's a big difference in how one calculates the viewport bounds for the two types of cameras. This matters for ZM clients such as TpZoneLayout, which only supports Orthographic cameras.

Multiple ZoneManagers

You can have as many as you want. Note the warning about sharing the same Tilemap with different ZMs. Don't do it. Besides that, ZMs are independent, and can have different chunk sizes, world origins, and so on.

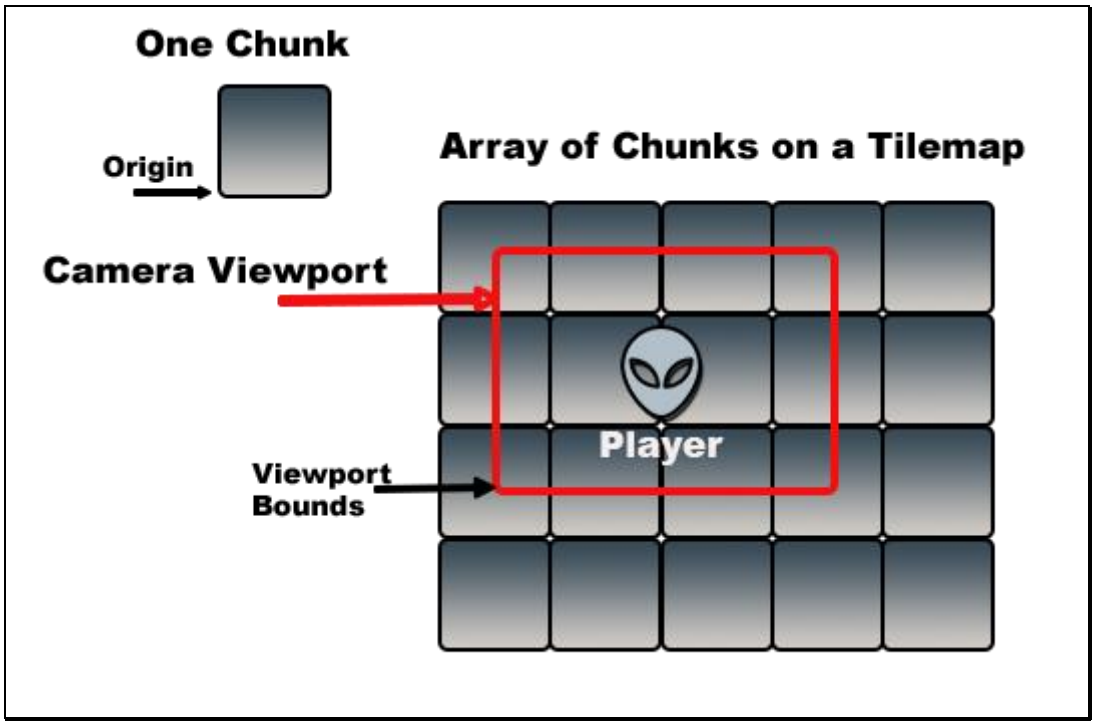
Scene Loading and Unloading

In general, you are responsible for deleting ZMs when a scene that they are in is destroyed. An example of how that's done can be seen in the TpZoneLayout component's OnDisable event handler, which handles this for you.

TpZoneLayout

At an even higher level, the **TpZoneLayout** MonoBehaviour component leverages a ZM to implement a basic camera Viewport chunking system. This component can be used as is, or as a base class for something more complex. Examine the "**Chunking**" demo to see how to use it.

The following crude illustration shows an array of chunks on a Tilemap, some type of Player character, and a camera tethered to the Player, that is, a simple Top-Down view of a Tilemap.



ZoneLayout places TileFabs in the visible area of a camera viewport and deletes them from outside the viewport. The provided implementation only supports orthographic cameras. The location for each of the Chunks shown in the above illustration corresponds to a Locator maintained by a ZoneManager instance.

As the Player moves, so does the Camera. As the Camera moves, its Viewport moves, and overlaps different chunks after each movement, akin to a window sliding over the sGrid and the Tilemap. When a chunk moves out of view it should get deleted, and when there's an empty chunk in the view then a Chunk should be placed there.

The TpZoneLayout component does most of the initialization work for you. In *your* initialization you need to:

Initialize the ZoneManagement system in TileFabLib. Read the notes about how to build a GUID->TileFab map. This is optional but recommended if possible.

Assuming that the ZoneLayout components are on the same GO as the GameController (or whatever you like to call it) just do:

- Get all the ZoneLayout components.
- Initialize all the ZoneLayout components that you found.
- Optionally call UpdateTick on all layouts to fill-in the initial area of the viewport
- In your input handling code, every time the player moves call UpdateTick on all layouts.

If you examine the ChunkSystemDemo scene and the ChunkingDemoController component, you can see that this is basically what's done in the ChunkingDemoController (CDC). As an optimization, you could restrict calling UpdateTick to times when the camera has moved say, ½ unit. CDC does not do that so that the viewport and subdivided chunk Locators (yeah, a lot of jargon) are visible in the Scene window.

Let's look at some of the ZoneLayout fields:

- Active: Handy for debugging, if unchecked then this layout is ignored.
- Chunk Size: Size of chunks 4,6,8… If this doesn't match the size of the Chunks returned by the Chunk Selector, then the system will not work correctly.
- Reference Camera: the Camera you want to use for its viewport. In many situations this value will be the same for multiple layouts. But it doesn't have to be.
- World Origin: the origin of the sGrid. Can be different on each layout.
- Grid: the parent Grid of the Tilemaps that this layout will be using. When using multiple layouts, it is important to ensure that this is the correct Grid for the Chunks that you'll use.
- Camera View Padding: adds some extra space around the Camera viewport.
- Zone Manager Name: Must be unique for each Layout. If not, one of the duplicates will be ignored.
- Chunk Selector: A Scriptable Object asset that selects which Chunk to use at a particular Locator.
- Show Camera Rect: displays a marquee around the Viewport and chunk-sized squares showing how it is subdivided. Useful for debugging.
- Do Not Load: No Chunks are ever loaded. You can watch the marquees move about without any distractions. Can get hypnotic.

TpZoneLayout uses a method in TpZoneManagerUtils to calculate a RectInt which describes the viewport. Since the viewport size is floating-point and a RectInt is not, the calculations always round up, so in general, the RectInt computed as the viewport size is bigger than the actual viewport. This is completely OK and is desirable as the effect helps to reduce visual artifacts.

Even with the rounding-up of the viewport size, you always need to add some padding so that there are no visual artifacts. This is very app dependent.

Then ZoneLayout uses the ZM to detect which chunks are within the viewport and which are not. Chunks outside the viewport are deleted from the Tilemap. Any empty Zones within the viewport are filled in with a Chunk provided by either a **ChunkSelector** asset or via a callback provided during initialization of the Layout (that's not done in this demo). The TileFabs can have filtering if you provide a filter callback.

Each TileFab has **two user fields**: a boolean and a string. These can be used by the filter. The use of this can be seen in ChunkingDemoController.LoadingFilter.

Selectors and ChunkSelectors

By now you're wondering what this new bit of jargon is all about. A Selector is a bit of code that ZoneLayout invokes to obtain the Chunk to place at a certain location. At a low-level, a callback for this can be provided when initializing the ZoneLayout from code. A ChunkSelector is the same thing, except it is embedded inside a Scriptable Object so that it's easier to use in the Unity Editor environment.

Since they're essentially the same thing, we'll just call them **Selectors** from now on.

ChunkSelectors use the IChunkSelector interface, which contains methods for initialization and Selection.

TpSingleFabChunkSelector couldn't be simpler: it just returns the same Chunk every time. This is useful when you want to fill a solid background layer.

TpTemplateChunkSelector can use a pattern Template as a Selector. Now we can dig into the demo code.

Selectors and Templates

Let's examine the ChunkingDemoController (CDC) component found in the "ChunkSystemDemo" scene.

It uses the same input controller as that in TopDownDemo and performs the type of setup that the ZoneLayout component requires. Layout.UpdateTick is called at the end of each player move; the object being to delete Chunks outside the Camera viewport and add ones that are about to become within the viewport.

Look at the GameController GO. Note that it has two TpZoneLayout components. As supplied, they're both active when you run the demo. You can deactivate one or the other to see the difference and watch the computation visually if you toggle DoNotLoad to ON.

They're completely independent since they use two different Grids. Not only that, but they also have different Chunk sizes, ZoneManager names, and Chunk Selectors.

The layout named ChunkingDemoBg uses only one 4x4 chunk to fill-in the "Background" Tilemap using the TpSingleFabChunkSelector asset. That asset just returns the same Chunk every time. It's a one-Bundle TileFab Chunk.

The layout named ChunkingDemo uses TpTemplateChunkSelector. That returns a checkerboard pattern of some 16x16 Chunks. These chunks comprise four Tilemaps full of data which are placed on the four child Tilemaps of "Main Grid". These Tilemaps are set up the same as the source Tilemaps for the Chunks; in this case just the Tilemap Renderer sort orders are the same. It's important to get that right for multilayer Chunks.

A Template is a simple solution to having a Selector which returns different Chunks depending on position. It's an array of Chunks (TileFabs) set up in the same XY orientation as the pattern you'd like to create. If the Template has empty spots, then no Chunk is placed at that spot.

What makes a good Template?

- The Template array of TileFabs needs to be square.
- Each element's size is a multiple (not 1) of the Chunk size.
- The Chunks in the Template need to be square and of even dimensions (by definition)

The Selector callbacks get several parameters: the Locator for the area that needs to be filled-in with a Chunk, the ZoneLayout instance, and the Tilemap name-to-instance Dictionary.

When a Template is used, the Selector callback indexes into a Template to get a Chunk.

It's an example of how to place a pattern using the ZoneManager, but not the only way to do it.

How to Create Pattern Templates

There's an app for that! A simple wizard called Template Tool that you can find in the Tools/TilePlus menu.

Create a design scene with a Parent single grid and Child Tilemaps which exactly match the environment that the pattern will be painted to. That means that the Tilemaps need to have the same origin, sort order, Color, and so on.

Design your pattern. You can do this by painting TileFabs, tiles, prefabs, whatever you like. Bear in mind that the design will be cut up into TileFabs based on the Chunk size set in Tile+Painter. GridSelection overlays are handy during this stage since you can create a GridSelection which defines the area that you're working in.

Open the Template Tool. Notice that it will use the same chunk size and world origin as is set in the Tile+Painter.

Drag the Parent Grid of the pattern into the field then click Next. Use the Unity Palette or Tile+Painter to draw a Grid Selection around the pattern, then click Make Template.

Alternatively, activate a stored Grid Selection from the Painter's Grid Selection panel when you're in the Make Template stage.

You'll notice that the mouse cursor shows the location and size of the Grid Selection. Once you've created the selection, the normal Unity inspector should show the Grid Selection in the Inspector pane. Both the X and Y size of the selection should be even numbers.

If you paint TileFabs to create your pattern, turn on Chunk Snapping mode in Tile+Painter. The visual display of Snap zones (based on the World Origin and Chunk Size that you have set) will be the same as the areas captured.

If the pattern has any empty areas on any Tilemap then that location on the target Tilemap won't be affected.

If the GridSelection has any empty area corresponding to an entire Chunk, then no TileFab will be created for that area. If there is even one tile or Prefab, then the area is recognized.

This affects how Chunks are removed as the Camera moves. If there was no TileFab at all then internally there will be no ZoneReg for that location and when the Camera moves such that that area is outside of the Camera view + padding the entire area is erased.

If there are no tiles or prefabs at all then no TileFab is generated, and the area is never touched.

Once you create the Selection, the Make Template button will enable; click that and follow the prompts. If the "Create Template Selector Asset" checkbox is ON then a ChunkTemplateSelector asset is also created, and pre-populated with the Template asset reference.

The toggle "Hide the created assets from Tile+Painter" is normally checked because it is possible to create many assets using this tool and you normally don't need or want to paint these. This setting avoids cluttered lists in Tile+Painter.

If it seems like the Make Template button never enables, that's because internally the Template Tool is watching the Grid Selection. The button won't enable if there's no Grid Selection or the size of the Grid Selection isn't a multiple of the chunk size or isn't a square.

Please use a different folder each time. Depending on the pattern and the chunk size, many Bundles and TileFabs can be generated, and it can be confusing to keep track of the assets since the asset names are created programmatically.

If you created a Template comprising 4x4 Chunks, then that pattern would be placed in the area around the Camera viewport rather than just using the same tile repeatedly. The Selector then does some simple indexing into the Palette's Tilemap to get the Chunk to place.

In this manner, complex layouts of tiles can be automatically swapped in and out of view as the Camera moves. Depending on the Chunk size and Camera settings, tweaking the padding values in the ZoneLayout component is always needed. Too large is safe but loads more than needed. Too small may be acceptable if you want the Chunk loading and unloading to be visible as an effect.

This can be extended as you like to enable terrain-like chunking for sections of Tilemaps and all sorts of other applications.

Tile+Painter Chunk Snapping

The Painter Settings panel has three settings that enable Chunk Snapping.

Chunk snapping allows you to easily paint and erase equal-sized Chunks on a virtual grid, with snapping.

The *Chunk Snapping* checkbox enables the mode and locks the fields just below.

- Set the *Chunk Size for Snapping* field to the fixed size of the TileFabs that you want to paint (4,6,8…).
- Set the *Chunk Snapping World Origin* field to the Origin position for TileFab painting. Usually 0,0,0 is what you want.

As mentioned above, the Size and Origin values are also used by the Template Tool.

The center column will show only Chunks (TileFabs) whose bounds are the correct size.

The only active Tools are Paint and Erase.

Two marquees appear while using those tools when Snapping is on:

- At the nearest Chunk location on the virtual grid.
- At the Mouse Position

Both are the same size as the Chunk dimensions. If this doesn't match the Chunk dimensions then Chunk Size for Snapping doesn't really match the size of the Chunk (which must be square and of even dimensions e.g., 4x4, 6x6 etc.). If the positioning seems off, check that the World Origin is set properly.

When the mouse position is on the virtual grid defined by the Chunk Size and the World Origin:

- The two Marquees merge.
- The color of the Marquee will change.
- <G> appears at the beginning of the informational text string *and the font becomes italicized*.

When the mouse cursor isn't on the virtual grid and you click, the Painter paints the Chunk in that nearest virtual grid location. The mouse cursor won't move.

If you paint the same chunk at the same virtual grid location, then the area is erased first.

When using the Erase tool when Snapping is on, a TileFab that uses the same Tilemaps must be selected in the center column so that Erase knows which maps to erase. If that's not the case, nothing happens, and a reminder dialog box will appear.

Erase deletes all tiles in the highlighted region. Any children of the Tilemaps' GameObjects in that area are also deleted.

Erase requires that the mouse position be aligned with the virtual grid; watch the marquees and the <G>.

Why Do New GUIDs matter?

As mentioned earlier in this section, 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.

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.

Notes

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.