Articles

A Quasi-blog

- Stateful Scriptable Objects
- On The Wonderfulness of Scriptable Objects

Stateful Scriptable Objects

Here's an example of a Scriptable Object with state. Part of TilePlus Toolkit. Note this is free on the asset store but is copyrighted.

```
// ***********************************************************************
// Assembly         : TilePlus
// Created          : 03-25-2023
//
// Last Modified On : 04-03-2023
// ***********************************************************************
// <copyright file="TpZoneManager.cs" >
//     Copyright (c) All rights reserved.
// </copyright>
// <summary></summary>
// ***********************************************************************
#nullable enable

using System;
using System.Collections.Generic;
using System.Linq;
// ReSharper disable once RedundantUsingDirective
using UnityEditor;
using UnityEngine;
using UnityEngine.Tilemaps;
using static TilePlus.TpLib;

// ReSharper disable MemberCanBePrivate.Global



namespace TilePlus
{
    /// <summary>
    /// TpZoneManager is used to manage square areas of tilemaps called Zones.
    /// </summary>
    public class TpZoneManager : ScriptableObject
    {
```

```
        #region subscriptions

        /// <summary>
        /// Notify me when a Zone Reg is added.
        /// </summary>
        /// <remarks>Be aware that when the ZoneManager instance is deleted this subscription
expires</remarks>
        public event Action<ZoneReg, TpZoneManager>? OnZoneRegAdded;
        /// <summary>
        /// Notify me when a Zone Reg is deleted.
        /// </summary>
        /// <remarks>Be aware that when the ZoneManager instance is deleted this subscription
expires</remarks>
        public event Action<ZoneReg,TpZoneManager>? OnZoneRegDeleted;

        /// <summary>
        /// Notify me that a list of TPT tiles will be deleted.
        /// </summary>
        /// <remarks>Be aware that when the ZoneManager instance is deleted this subscription
expires</remarks>
        public event Action<Tilemap,List<TilePlusBase>>? OnTptTilesWillBeDeleted;



        #endregion

        #region privateFields

        /// <summary>
        /// Mapping between RectInts (locator) and Zone Registrations.
        /// </summary>
        private Dictionary<RectInt, ZoneReg> chunkMap = new ();

        /// <summary>
        /// The default ChunkLocator from the size param to Initialize.
        /// This defines the size of each chunk - it's size.x and size.y params
        /// (both should be equal). This is available via a property.
        /// </summary>
        private RectInt defaultLocator;

        /// <summary>
```

```
        /// The starting position, the position of chunk zero.
        /// </summary>
        private  Vector2Int worldOrigin;

        /// <summary>
        /// indicates that chunking is enabled after a call to Initialize.
        /// </summary>
        private  bool       chunkingConfigured; //backup field for property



        //holds the return value for RestoreFromRegistrationJson
        private  readonly List<TilefabLoadResults> currentLoadresults = new(8);

        //temporary list but use the same one repeatedly to reduce garbage
        private    List<ZoneReg> getZoneRegForChunkInternal = new(32);

        //maps for this instance. clients need to use only these maps.
        private Dictionary<string, Tilemap> monitoredTilemaps = new();

        //the name of this instance.
        private string instanceName = string.Empty;
        #endregion

        #region properties
        /// <summary>
        /// Get the all loading results
        /// </summary>
        public IEnumerable<ZoneReg> GetAllZoneRegistrations => chunkMap.Values.OrderBy(zr =>
zr.dex);



        /// <summary>
        /// Is chunking configured?
        /// </summary>
        public  bool ChunkingConfigured => chunkingConfigured;

        /// <summary>
        /// Size of a chunk as set during initialization.
        /// </summary>
        public int ChunkSize => defaultLocator.size.x;
```

```
        /// <summary>
        /// The number of chunks in the chunkmap.
        /// </summary>
        public int NumChunksInUse => chunkMap.Count;


        /// <summary>
        /// Get a collection of the MonitoredTilemaps dictionary values: Tilemap instances. These are
        /// the only ones that clients of this instance should be using.
        /// </summary>
        public Dictionary<string, Tilemap>.ValueCollection MonitoredTilemaps => monitoredTilemaps.Values;

        /// <summary>
        /// Access to the Monitored Tilemaps for this instance. DO NOT ALTER THIS DICTIONARY. DON'T SAVE A REFERENCE.
        /// </summary>
        public Dictionary<string, Tilemap> MonitoredTilemapDict => monitoredTilemaps;


        /// <summary>
        /// Get the default locator. This is the value used when
        /// you don't specify a dimensions value when GetZoneReg
        /// or GetLocator with dimensions = null.
        /// </summary>
        // ReSharper disable once ConvertToAutoProperty
        public RectInt DefaultLocator => defaultLocator;
        /// <summary>
        /// Get the ChunkMapAnchorPosition. This is the base or startingPosition from where all Chunks begin.
        /// You can find the center of a chunk by adding multiples of ChunkMapAnchorSize to ChunkMapAnchorPosition.
        /// </summary>
        public  Vector2Int WorldOrigin => worldOrigin;

        /// <summary>
        /// The name of this instance. Read only
        /// </summary>
```

```
public string InstanceName => instanceName;

/// <summary>
/// A ref to the ZoneLayout, if used with chunking system. Null otherwise.
/// </summary>
public TpZoneLayout? ZoneLayoutComponent { get; set; }



#endregion

#region access

/// <summary>
/// Reset all registrations, reset registrationIndex
/// </summary>
/// <param name="resetEvents">Reset event descriptions (default=true)</param>
public void ResetInstance(bool resetEvents = true )
{
    chunkMap.Clear();
    chunkingConfigured = false;
    currentLoadresults.Clear();
    getZoneRegForChunkInternal.Clear();
    monitoredTilemaps.Clear();
    if (!resetEvents)
        return;
    OnZoneRegDeleted        = null;
    OnZoneRegAdded          = null;
    OnTptTilesWillBeDeleted = null;
}




/// <summary>
/// Add a TileFab chunk to the database. In general this should ONLY be
/// called from TileFabLib. CHUNKS ONLY.
/// </summary>
/// <param name="tileFab">The TileFab to load</param>
/// <param name="createAsImmortal">mark the ZoneReg as immortal</param>
/// <param name="offset">placement offset</param>
/// <param name="rotation">rotation</param>
```

```
        /// <param name="posToGuidMaps">remapping dictionary</param>
        /// <param name="bundleAssetGuids">Asset GUIDs</param>
        /// <param name="bundleAssetNames">Asset names</param>
        /// <param name = "spawnedPrefabs" >List of prefabs spawned when the TileFab was
loaded. Note: not serialized
        /// in the ZoneReg class instance created herein.</param>
        /// <returns>Tuple of ZoneReg and RectInt (locator) or null for error. AssetReg is
null if error.</returns>
        internal (ZoneReg? reg, RectInt locator) AddZone(TpTileFab?
tileFab,
                                                         bool
createAsImmortal,
                                                         Vector3Int
offset,
                                                         TpTileBundle.TilemapRotation
rotation,
                                                         Dictionary<Vector3Int, string>[]?
posToGuidMaps,
                                                         string[]?
bundleAssetGuids,
                                                         string[]?
bundleAssetNames,
                                                         List<GameObject>?
spawnedPrefabs)
        {
            if (!chunkingConfigured)
            {
                TpLogError("Cannot add Zones to TpZoneManager before it is configured. Use
'Initialize' first!!!");
                return(null,defaultLocator);
            }

            if (!tileFab || posToGuidMaps == null || bundleAssetGuids == null ||
bundleAssetNames == null)
            {
                TpLogError("null TileFab, posToGuidMaps, bundleAssetGuids or bundleAssetName
was passed to TpZoneLayout.AddZone.");
                return(null,defaultLocator);
            }
```

```
            if (!tileFab.m_FromGridSelection)
            {
                TpLogError("Cannot use TpZoneManager.AddZone with a non-chunk TileFab!!! And
you can't just click the 'From Grid Selection checkbox on the asset: that won't work
correctly. PLease recreate the TileFabs using GridSelections!");
                return(null,defaultLocator);
            }
            // ReSharper disable once
ConditionIsAlwaysTrueOrFalseAccordingToNullableAPIContract
            if(tileFab.m_TileAssets.Count == 0  || tileFab.m_TileAssets[0] == null ||
!tileFab.m_TileAssets[0].m_Asset)
            {
                TpLogError($"Invalid TileFab [{tileFab.name}]: does not have any bundles.");
                return(null,defaultLocator);
            }

            //need a boundsInt for the chunk in order to create a 'locator' RectInt.
            //we know that the TileFab has to have at least one Chunk.
            //all chunk boundsInts are identical.
            var chunkBoundsInt = tileFab.IsChunkified ? new
BoundsInt(0,0,0,ChunkSize,ChunkSize,1) :
                                    tileFab.LargestBounds; //this method does exactly that
when a Fab is a chunk.

            //update map from BoundsInt to reg (for camera-region culling)
            //now compute the locator for this chunk: Basically it's a RectInt encompassing
the entire Chunk as placed.
            //NOTE THAT the position of a RectInt is NOT the center. It's the lower-left
corner. This is fine
            //as long as we're consistent.
            var locator = GetLocatorForGridPosition(offset);
            if (chunkMap.ContainsKey(locator))
            {
                TpLogError($"The locator [{locator}] already exists! Can't place at offset
{offset}");
                return(null,defaultLocator);
            }

            var reg = new ZoneReg(TileFabLib.RegistrationIndex,
                                  locator,
```

```
                                tileFab.AssetGuidString,
                                tileFab.name,
                                offset,
                                rotation,
                                posToGuidMaps,
                                bundleAssetGuids,
                                bundleAssetNames,
                                chunkBoundsInt,
                                spawnedPrefabs);
            if (createAsImmortal)
                reg.imm = true;

            var hash = new AssetGuidPositionHash(tileFab.TileFabGuid, offset);
            return !AddRegistration(reg, hash)
                       ? (null,defaultLocator)
                       : (reg,locator);
        }

        /// <summary>
        /// Add a registration. Only use if you're not using AddZone and creating your own
ZoneRegs
        /// </summary>
        /// <param name="reg">The ZoneReg</param>
        /// <param name="hash">An AssetGuidPositionHash instance</param>
        /// <returns>false for failure: means that there was an entry already existing for
this locator.</returns>
        public bool AddRegistration(ZoneReg reg, AssetGuidPositionHash hash)
        {
            var locator = reg.m_MyLocator;
            if (!chunkMap.TryAdd(locator, reg))
            {
                #if UNITY_EDITOR
                TpLogError($"Fatal: duplicate key {locator} in ChunkMap for reg {reg}. ");
                #endif
                return false;
            }

            TileFabLib.S_LoadedGuids?.Add(hash);
            TileFabLib.IncrementRegistrationIndex();
            OnZoneRegAdded?.Invoke(reg,this);
```

```
            return true;
        }


        /// <summary>
        /// Is there a registration for this ZoneReg?
        /// </summary>
        /// <param name="reg">A ZoneReg</param>
        /// <returns></returns>
        public bool HasZone(ZoneReg reg)
        {
            if (chunkingConfigured)
                return chunkMap.ContainsKey(reg.m_MyLocator);
            TpLogError("Cannot use TpZoneManager before it is configured. Use 'Initialize'
first!!!");
            return false;
        }

        /// <summary>
        /// Unload a list of Zones
        /// </summary>
        /// <param name="regs">List of ZoneRegs to delete</param>
        /// <param name="destroyTiles">destroy tiles (default)</param>
        /// <param name="destroyPrefabs">destroy prefabs (default)</param>
        /// <returns>false if failed</returns>
        public bool UnloadZones(List<ZoneReg> regs, bool destroyTiles = true, bool
destroyPrefabs = true)
        {
            var error = false;
            foreach (var reg in regs)
                error |= UnloadZone(reg, destroyTiles, destroyPrefabs);
            return error;
        }

        /// <summary>
        /// Unload a list of Zones, Async
        /// Does one reg per frame.
        /// </summary>
        /// <param name="regs">List of ZoneRegs to delete</param>
```

```
        /// <param name="destroyTiles">destroy tiles (default)</param>
        /// <param name="destroyPrefabs">destroy prefabs (default)</param>
        /// <returns>false if failed</returns>
        public async Awaitable<bool> UnloadZonesAsync(List<ZoneReg> regs, bool destroyTiles =
true, bool destroyPrefabs = true)
        {
            var success = true;
            foreach (var reg in regs)
            {
                success &= UnloadZone(reg, destroyTiles, destroyPrefabs);
                if (success)
                    await Awaitable.NextFrameAsync();
            }
            return success;
        }



        /// <summary>
        /// Unload ALL zones, including all parented prefabs.
        /// </summary>
        /// <returns></returns>
        public bool UnloadAllZones()
        {
            return UnloadZones(chunkMap.Values.ToList());
        }



        /// <summary>
        /// Unload a chunk
        /// </summary>
        /// <param name="reg">corresponding ZoneReg for the chunk you want to delete.</param>
        /// <param name = "destroyTiles" >destroy tiles if true (default)</param>
        /// <param name = "destroyPrefabs" >destroy prefabs if true (default)</param>
        /// <returns>true if successful.</returns>
        /// <remarks> runtime use ONLY </remarks>
        public bool UnloadZone(ZoneReg? reg, bool destroyTiles = true, bool destroyPrefabs =
true)
        {
```

```
            if (reg == null)
            {
                TpLogError("Null ZoneReg passed to UnloadZone.");
                return false;
            }
            //reserved zones are handled simply since there aren't any tiles/prefabs to
delete.
            if (reg.m_Reserved)
            {
                if (!DeleteZoneRegistration(reg))
                    TpLogWarning($"Could not delete this zonereg: {reg}");

                OnZoneRegDeleted?.Invoke(reg,this);
                return true;
            }

            if (!chunkingConfigured)
            {
                TpLogError("Cannot delete Zones from TpZoneManager before it is configured.
Use 'Initialize' first!!!");
                return false;
            }

            if (!chunkMap.ContainsKey(reg.m_MyLocator))
            {
                TpLogError($"Unknown ZoneReg [{reg}], can't delete zone!");
                return false;
            }

            if (destroyTiles)
            {
                //area is 'largestbounds' from the asset
                var eraseBounds = reg.lb;
                //offset it
                eraseBounds.position += reg.offs;
                var sz = eraseBounds.size;
                sz.z             = 1;
                eraseBounds.size = sz;
```

```
            var ri = TpTileUtils.RectIntFromBoundsInt(eraseBounds, Vector3Int.zero);
            //Debug.Log($"bounds {eraseBounds} rectint {ri}");

            //todo: could cache depending on chunk size.
            var nulls = new TileBase[sz.x * sz.y]; //these should all be null.

            //get a list of TPBs (which is cleared as required for
            using (TpLib.S_TilePlusBaseList_Pool.Get(out var pTiles))
            {
                if (pTiles != null)
                {
                    foreach (var map in monitoredTilemaps.Values)
                    {
                        var pos     = map.transform.position;
                        var gridPos = map.WorldToCell(pos);
                        var ri2     = new RectInt((Vector2Int)gridPos + ri.position,
ri.size);

                        TpLib.GetAllTilesInRegionForMap(map, pTiles, ri2);
                        OnTptTilesWillBeDeleted?.Invoke(map, pTiles); //event.
                        map.SetTilesBlock(eraseBounds, nulls);
                    }
                }
            }
        }

        if (destroyPrefabs && reg.m_Prefabs != null)
        {
            //destroy any prefabs
            if (reg.m_Prefabs.Count != 0)
            {
                foreach (var gameObj in reg.m_Prefabs)
                {
                    if (gameObj.TryGetComponent<TpSpawnLink>(out var link))
                    {
                        link.DespawnMe();
                        continue;
                    }
                    #if UNITY_EDITOR
                    UnityEngine.Object.DestroyImmediate(gameObj, false);
```

```
                    #else
                    UnityEngine.Object.Destroy(gameObj);
                    #endif
                }
            }
        }

        if (!DeleteZoneRegistration(reg))
            TpLogWarning($"Could not delete this zonereg: {reg}");

        OnZoneRegDeleted?.Invoke(reg,this);
        return true;
    }


    #endregion
    #region chunking

    /// <summary>
    /// Required if you want to use chunking. No chunking data is accumulated and
    /// chunking will not work if this isn't used. Note that you should use this again for
    /// every new scene. WIPES OUT ANY EXISTING TILEFAB REGISTRATION DATA WHEN USED!!
    /// </summary>
    /// <param name="size">Size of a chunk. Must be even, rounded up if not.
    ///  Min=4. 4x4, 6x6, 8x8 ... 16x16 chunks etc</param>
    /// <param name="origin">The origin position, the base position, such as
    /// Vector3Int.zero, where the chunk numbers should be centered. If null then
Vector3Int.zero is used. </param>
    /// <param name="initialMaxNumChunks">sets certain data structures' initial size. Base
this on the total
    /// number of chunks of 'size' that would be in your camera's FOV at one time (for
example). There's no problem
    /// if the max num chunks is exceeded, this just allocates memory early on given your
best estimate of what's
    /// required as set in this method.</param>
    public  void Initialize(int size,  Vector3Int? origin = null, int initialMaxNumChunks
= 64)
    {
        origin       ??= Vector3Int.zero; //default for origin position
```

```
            if (size < 4) //this would be a 4x4 TileFab which is ridiculously (?) small.
                size = 4;
            //test for an even number.
            if (size % 2 != 0) //remainder should be zero if this is a multiple of 2.
                size++;

            //allocate memory for arrays.
            chunkMap                     = new
Dictionary<RectInt,ZoneReg>(initialMaxNumChunks);
            getZoneRegForChunkInternal  = new List<ZoneReg>(initialMaxNumChunks / 4);

            defaultLocator   = new RectInt(Vector2Int.zero, new Vector2Int(size, size));
            worldOrigin = new Vector2Int(origin.Value.x, origin.Value.y);

            chunkingConfigured = true;
        }

        /// <summary>
        /// Get the ZoneRegs for a Zone locator RectInt
        /// </summary>
        /// <param name="locator">the RectInt chunk locator</param>
        /// <returns>ZoneReg List, list is Empty for error</returns>
        /// <remarks>return empty list if chunking not enabled or chunklocator is null.
        /// Note that the same list is cleared and re-used every time that this is called.
</remarks>
        public  List<ZoneReg> GetZoneRegsForRegion(RectInt? locator)
        {
            getZoneRegForChunkInternal.Clear();

            if (!chunkingConfigured || !locator.HasValue || locator.Value.size ==
Vector2Int.zero)
                return getZoneRegForChunkInternal;

            var loc = locator.Value;
            // ReSharper disable once ForeachCanBeConvertedToQueryUsingAnotherGetEnumerator
            foreach (var item in chunkMap.Keys)
            {
                if (item.Overlaps(loc))
                    getZoneRegForChunkInternal.Add(chunkMap[item]);
            }
```

```
            return getZoneRegForChunkInternal;
        }

        /// <summary>
        /// Obtain two datasets: one is a List of ZoneRegs that are outside of an area and
another
        /// is a HashSet of ZoneRegs that are inside the area.
        /// </summary>
        /// <param name = "locator" >the RectInt Locator describing the area. </param>
        /// <param name="inside">ref HashSet for inside</param>
        /// <param name="outside">ref List for outside</param>
        /// <returns>false if any error occurs</returns>
        public bool FindRegionalZoneRegs(RectInt? locator, ref HashSet<RectInt> inside, ref
List<ZoneReg> outside)
        {
           if (!chunkingConfigured || !locator.HasValue)
               return false;

           var loc = locator.Value;
           inside.Clear();
           outside.Clear();
            //    return other.xMin < this.xMax && other.xMax > this.xMin && other.yMin <
this.yMax && other.yMax > this.yMin;

            // ReSharper disable once ForeachCanBeConvertedToQueryUsingAnotherGetEnumerator
            foreach ((var zone, var reg) in chunkMap)
            {
                if (zone.Overlaps(loc)) //if ANY part of the locator/zone overlaps the
'locator' RectInt
                    inside.Add(zone); //inside the locator's region
                else
                    outside.Add(reg); //outside the locator's region
            }
            return true;
        }



        /// <summary>
        /// Get the ZoneRegs for a chunk located at a Grid position.
        /// </summary>
```

```
        /// <param name="gridPosition">The position to use when searching for Zone
registrations</param>
        /// <param name = "dimensions" >[Nullable] if not null, this is the search area. If
null, ChunkMapAnchorPosition as set by Initialize</param>
        /// <param name = "align" >Align to grid. Default=true</param>
        /// <returns>a list of ZoneRegs. Empty list is valid and means error or nothing
found.</returns>
        public  List<ZoneReg> GetZoneRegsForGridPosition(Vector3Int gridPosition, Vector2Int?
dimensions = null, bool align = true)
        {
            return
GetZoneRegsForRegion(GetLocatorForGridPosition(gridPosition,dimensions,align));
        }

        /// <summary>
        /// Get the Zone registration for a chunk located at a World position.
        /// </summary>
        /// <param name="worldPosition">The position to use when searching for Zone
registrations</param>
        /// <param name = "map" >Tilemap to use for translating world to grid positions. If
null, an empty list is returned.</param>
        /// <param name = "dimensions" >[Nullable] if not null, this is the search area. If
null, ChunkMapAnchorPosition as set by Initialize</param>
        /// <param name = "align" >Align to grid. Default=true</param>
        /// <returns>a list of ZoneRegs. Empty list is valid and means error or nothing
found.</returns>
        public List<ZoneReg> GetZoneRegsForWorldPosition(Vector3 worldPosition, Tilemap? map,
Vector2Int? dimensions = null, bool align = true)
        {
            if (map)
                return GetZoneRegsForRegion(GetLocatorForWorldPosition(worldPosition, map,
dimensions, align));

            getZoneRegForChunkInternal.Clear();
            return getZoneRegForChunkInternal;
        }


        /// <summary>
        /// Create an Zone registration locator from a grid position
```

```
        /// </summary>
        /// <param name="gridPosition">position on a Tilemap</param>
        /// <param name="dimensions">[Nullable] optional size of locator. If null,
ChunkMapAnchorPosition as set by Initialize </param>
        /// <param name = "align" >Align to grid. Default=true</param>
        /// <returns>RectInt Zone registration locator</returns>
        public  RectInt GetLocatorForGridPosition(Vector3Int gridPosition, Vector2Int
?dimensions = null, bool align = true)
        {
            if (align)
                gridPosition = AlignToGrid(gridPosition);
            return new RectInt((Vector2Int) gridPosition + worldOrigin, dimensions ??
defaultLocator.size);
        }

        /// <summary>
        /// Create an Zone registration locator from a world position
        /// </summary>
        /// <param name="position">world position</param>
        /// <param name = "map" >Tilemap to use for translating world to grid positions. If
null, new RectInt() is returned.</param>
        /// <param name="dimensions">[Nullable] optional size of locator. If null,
ChunkMapAnchorPosition as set by Initialize </param>
        /// <param name = "align" >Align to grid. Default=true</param>
        /// <returns>RectInt Zone registration locator</returns>
        public RectInt GetLocatorForWorldPosition(Vector3 position, Tilemap? map, Vector2Int?
dimensions = null, bool align = true)
        {
            return !map ? new RectInt()
                        : GetLocatorForGridPosition(map.WorldToCell(position),
dimensions,align);
        }




        /// <summary>
        /// Is there a Zone registration associated with a RectInt locator?
        /// </summary>
        /// <param name="locator">the RectInt to check</param>
```

```
        /// <returns>true if there's already a locator there.</returns>
        public bool HasZoneRegForLocator(RectInt locator)
        {
            return chunkMap.ContainsKey(locator);
        }

        /// <summary>
        /// Get a Zone Reg for a locator RectInt
        /// </summary>
        /// <param name="locator">a locator</param>
        /// <param name="reg">a registration</param>
        /// <returns>true if reg was found. Note: if false the reg is default</returns>
        public bool GetZoneRegForLocator(RectInt locator, out ZoneReg? reg)
        {
            return chunkMap.TryGetValue(locator, out reg);
        }



        /// <summary>
        /// Convert a super-grid position to a locator
        /// </summary>
        /// <param name="sGridPosition">s sGrid postion</param>
        /// <param name="dimensions">dimensions of locator or null for default</param>
        /// <returns>a Locator.</returns>
        public RectInt GetLocatorForSgridPosition(Vector2Int sGridPosition, Vector2Int?
dimensions = null)
        {
            var chunkSize = defaultLocator.size.x;
            var gridPos   = new Vector3Int(sGridPosition.x * chunkSize, sGridPosition.y *
chunkSize);
            return GetLocatorForGridPosition(gridPos, dimensions);
        }

        /// <summary>
        /// Get a Tilemap grid position from a sGrid position.
        /// </summary>
        /// <param name="sGridPosition">a sGrid position</param>
        /// <returns>a Tilemap grid position</returns>
        public Vector3Int GetGridPositionForSgridPosition(Vector2Int sGridPosition)
        {
```

```
            var chunkSize = defaultLocator.size.x;
            return new Vector3Int(sGridPosition.x * chunkSize, sGridPosition.y * chunkSize);
        }


        #endregion


        #region utils


        /// <summary>
        /// Get a JSON version of the Zone registrations
        /// </summary>
        /// <param name="prettyPrint"></param>
        /// <returns></returns>
        public  string GetZoneRegJson(bool prettyPrint = true)
        {
            var registrations= chunkMap.Values.OrderBy(zr => zr.dex).ToArray();
            var loadWrapper = new LoadWrapper(registrations);
            return JsonUtility.ToJson(loadWrapper, prettyPrint);
        }


        /// <summary>
        /// Restore all Tilefabs or Bundles based on a json-archived dataset.
        /// </summary>
        /// <param name="jsonString">data string to decode</param>
        /// <param name = "targetMap" >optional mapping from tilemap name to Tilemap instance.
Speeds tilemap lookups greatly</param>
        /// <param name = "filter" >a Func of [(enum)FilterDataSource, object] returning a
bool. See also LoadTileFab. </param>
        /// <param name = "filterOnlyTilePlusTiles"> if true (default) only applies filters to
TilePLus tiles, which is usually sufficient and saves much time.  </param>
        /// <returns>A list of the TilefabLoadResults from however many Bundles are in the
TileFab. Null is returned for errors</returns>
        /// <remarks>The list of TilefabLoadResults is cleared the next time that this method
is used.</remarks>
        public List<TilefabLoadResults>?
RestoreFromZoneRegJson(string                                                    jsonString,
                                                                Dictionary<string,
```

```
Tilemap>?                          targetMap = null,
                                                                Func<FabOrBundleFilterType,
BoundsInt, object, bool>? filter                  = null,
                                                                bool
filterOnlyTilePlusTiles = true
            )
        {
            var data = JsonUtility.FromJson<LoadWrapper>(jsonString);
            if(data == null)
                return null;

            var loadResultsArray = data.m_Res.ToList();
            loadResultsArray.Sort(Comparison); //ensure sorted in ascending index order.
            var numLoadsToMake   = loadResultsArray.Count;
            var numPrevLoads      = currentLoadresults.Count;
            currentLoadresults.Clear(); //absolutely required to avoid exception from next
line if count < capacity (no this was thought of in advance)
            if(numLoadsToMake > numPrevLoads) //don't play with Capacity unless enlarging.
                currentLoadresults.Capacity = loadResultsArray.Count;

            var loadFlags = filterOnlyTilePlusTiles
                                ? FabOrBundleLoadFlags.NormalWithFilter
                                : FabOrBundleLoadFlags.Normal;



            foreach (var r in loadResultsArray)
            {
                if(!TileFabLib.GetTileFabFromGuid(r.g, out var fab) || !fab)
                    continue;

                var result = TileFabLib.LoadTileFab(null,
                            fab,
                            r.offs,
                            r.rot,
                            loadFlags,
                            filter,
                            targetMap,
                            this);
                if(result == null)
                    continue;
```

```
                currentLoadresults.Add(result);

                TileFabLib.UpdateGuidLookup(r, result.ZoneReg!);

            }

            return currentLoadresults;
        }

        private  int Comparison(ZoneReg x, ZoneReg y)
        {
            if (x.dex < y.dex)
                return -1;
            return x.dex == y.dex ? 0 : 1;
        }



        /// <summary>
        /// Sets the name and managed Tilemaps for this instance.
        /// Note you can only do this once.
        /// </summary>
        /// <param name="iName">instance name</param>
        /// <param name="stringToTilemap">Dictionary of tilemap names to tilemap
instances.</param>
        /// <returns>false if this has been called already.</returns>
        /// <remarks>This is only called by TileFabLib when creating a ZoneManager
instance.</remarks>
        internal bool SetNameAndMap(string iName, Dictionary<string, Tilemap> stringToTilemap)
        {
            if (!string.IsNullOrEmpty(instanceName) || string.IsNullOrEmpty(iName))
                return false;
            instanceName  = iName;
            monitoredTilemaps = stringToTilemap;
            return true;
        }

        /// <summary>
        /// Set or change the monitored Tilemaps dict.
        /// </summary>
```

```
        /// <param name="stringToTilemap">Dict of stringTilemapName -> TilemapInstance</param>
        internal void SetMaps(Dictionary<string, Tilemap> stringToTilemap)
        {
            monitoredTilemaps = stringToTilemap;
        }

        /// <summary>
        /// Remove an Zone registration given an instance of one
        /// </summary>
        /// <param name="reg">ZoneReg instance</param>
        /// <returns>true if found</returns>
        public  bool DeleteZoneRegistration(ZoneReg reg)
        {
            var locator = reg.m_MyLocator;
            if (!chunkMap.Remove(locator))
            {
                TpLogError($"Could not delete ZoneReg {reg}");
                return false;
            }

            if (reg.m_Reserved)
                return true;

            //need to delete all entries from the s_LoadedGuidLookup that were originally
added.
            //this is done by getting the new GUIDs from the registration, then doing a
            //reverse lookup. That gets us the OLD guid which is the key for the
LoadedGuidLookup dictionary.
            //of course also need to remove the corresponding item in the reverse-lookup
dictionary
            // ReSharper disable once LoopCanBePartlyConvertedToQuery
            foreach (var bundleGuidMap in reg.ptgm) //these are the GUIDs assigned when loaded
via LoadTileFab
                TileFabLib.RemoveGuidLookup(bundleGuidMap);
            TileFabLib.S_LoadedGuids!.Remove(new AssetGuidPositionHash(new Guid(reg.g),
reg.offs));
            return true;

        }
```

```
        /// <summary>
        /// Get the last N Zone registrations.
        /// </summary>
        /// <param name="numResults"># of results desired. For a tilefab that should be
1</param>
        /// <returns>Enumerable of registrations, which could be empty.</returns>
        public  IEnumerable<ZoneReg> GetLastRegistrations(int numResults = 1)
        {
            return chunkMap.Values.OrderBy(zr => zr.dex).TakeLast(numResults);
        }

        /// <summary>
        /// Get the very last Zone Reg. Handy when you know there is only one.
        /// </summary>
        /// <returns>the last zone reg used or a new one (index will be 0) if there aren't any
regs in the ChunkMap for this ZM.</returns>
        public ZoneReg? GetLastRegistration()
        {
            return chunkMap.Count == 0 ? new ZoneReg() : chunkMap.Values.OrderBy(zr =>
zr.dex).Last();
        }

        /// <summary>
        /// Get all zone registrations with optional filtering and  ordering
        /// </summary>
        /// <param name="orderByIndex">order by ZoneReg index if true</param>
        /// <param name="filter">Func of ZoneReg returning bool. If ret val true then zm
returned else it is skipped.</param>
        /// <returns>IEnumerable of ZoneReg instances. DON'T HOLD REFERENCES TO THESE!!! will
make a memory leak.</returns>
        public IEnumerable<ZoneReg> GetAllZoneRegistrationsFiltered(bool orderByIndex=false,
Func<ZoneReg, bool>? filter = null)
        {
            if (filter != null)
                return orderByIndex
                           ? chunkMap.Values.OrderBy(zr => zr.dex).Where(filter)
                           : chunkMap.Values.Where(filter);

            if(orderByIndex)
```

```
                return chunkMap.Values.OrderBy(zr => zr.dex);
            return chunkMap.Values;
        }

        /// <summary>
        /// Is this grid position aligned to the super-grid?
        /// </summary>
        /// <param name="position">position to test</param>
        /// <returns>true if aligned</returns>
        public bool IsAlignedToGrid(Vector3Int position)
        {
            var relativePosition = position - (Vector3Int)worldOrigin;
            var size             = defaultLocator.size.x;
            return relativePosition.x % size == 0 && relativePosition.y % size == 0;
        }



        /// <summary>
        /// Align a position to the super-grid. Note: positions are aligned to the lower-left
corner of a rectint.
        /// </summary>
        /// <param name="position">position to adjust</param>
        /// <returns>Adjusted position. Won't change if already aligned.</returns>
        public Vector3Int AlignToGrid(Vector3Int position)
        {

            if (IsAlignedToGrid(position))
                return position;

            var relPos = position - (Vector3Int)worldOrigin;
            var size   = defaultLocator.size.x;
            var diffX  = relPos.x % size;
            var diffY  = relPos.y % size;

            return new Vector3Int(relPos.x - diffX, relPos.y - diffY, position.z);

        }

        #endregion
```

```
    }
}
```

On The Wonderfulness of Scriptable Objects

There are already a vast number of online articles about Unity's Scriptable Objects. Most of them tell you the same things:

- They're assets and can't be attached to GameObjects.
- Affect the asset in the Project and the S.O. is also affected.
- You can't really have stateful runtime code in them.

[Note: some of the following is simplified for readability]

But that's misleading. While it's true that the S.O. class definition resides in the Project folder, so does every other piece of code, be it a component, a static class, whatever.

One big difference is that a component or any sort of class is just a text file. A S.O. asset is inspectable and changes to it are saved in the Asset Database (ADB).

So it is true that changes to a S.O. at runtime affect the asset in the ADB.

Change your view

Think of a S.O. as a GameObject without a transform and missing a few events. And treat it like a prefab.

I first 'discovered' this scheme when trying to figure out how to have editable persistent Tilemap tiles. I wanted to be able to have editable fields in Tiles right in a scene rather than using Tiled (etc) and importing.

In Editor-edit mode, it turns out that if you merely clone (Instantiate) a Tile S.O. (a tile is a S.O.) and then add it to a Tilemap via SetTile (or any other method) then that clone is:

- Independent of the asset in the ADB.
- Saved with the scene and loaded when the scene is loaded by Unity.

And these tiles can have any sort of code and data that you want; have state, respond to events and so on.

There are a host of other issues to solve for using tiles this way, such as how to inspect tiles containing fields not in the Tile class, but that's talked about in the TilePlus Toolkit docs elsewhere on this site.

This approach can be extended to other uses:

- As DLLs, that is, as dynamically loaded services and/or singletons etc.
- As loadable code blocks created on-demand at runtime.

The 'DLL' approach is used in the TilePlus system for Services (Runtime Scriptable Object Singleton).

The 'loadable blocks' approach is used in the Tilemap Layout subsystem where blocks of tiles on multiple maps are added and deleted as the camera moves. This uses SOs called "Zone Managers" which are created and destroyed on demand depending on how many different Layouts are running.

Although upcoming Unity versions are going to 'fix' the reload time in-editor, currently these two approaches save reload time by avoiding static class initialization and so on. At runtime, lazily cloning SOs as needed improves startup time and can save memory/reduce GC issues.

But, Update?

And you can have Update, too. Using a modified GameLoop allows you to inject your own Update or other events wherever you want in the GameLoop.

Since the state of data in a S.O. that's part of a scene, such as a cloned Tile instance, is frozen at the time the scene is saved then the same issues arise as you have with any other runtime state changes. Changes in data can preserved with your other game data and restored the next time that a level is loaded. Just Json-ize it.

If you're curious about how this works and want to see some examples, check out the other documentation on this website. The Toolkit asset itself is free.