About the Layout Demo

Save Files

There are two different types of save files:

- CdemoSaveFile.txt: contains the GUID of the current TScene (the GUID in the left column of the TSceneList asset's custom editor window).
- CdemoSaveFile_guidstring.txt: contains JSON representing the data in the ChunkingDemoSavedData for a particular scene. The 'guidstring' is the GUID of the TScene that the data is for.

When the demo starts for the very first time, the first scene in the TSceneList is loaded since there aren't any save files. When a waypoint is reached the scene's data is saved, and the guidstring part of the filename is the same as the currently-loaded scene's GUID.

After that the CdemoSaveFile.txt does exist, and the GUID found there is used to load the corresponding TScene.

TdDemoGameController.Start

The `Start` method does all the setup for the demo game. It first calls VerifySetup which validates some of the fields and caches a few references.

Then Start waits for TpLib to be ready. Now let's load and set up our Services.

For simplicity this demo uses a SRS (Scriptable Runtime Service) as a global game state singleton. This demo is complex enough! A number of references are added to the GameState service. Note that this Service is basically just a Scriptable Object attached to TpServiceManager.

Some of the initialization is performed when TScenes are loaded using TSceneInitializers. Here we just set up a list of all the Tilemaps in the Unity scene, some other references and initial values such as a ChunkedSceneManager and CompositeCollider references, and a reference to the Player Prefab. The Player is added to a list of Persistent Game Objects. This list contains GameObjects that the Layout system should never remove.

The ChunkedSceneManager callbacks are set up, then the FileAccess and Layout services are loaded. The handles (references) are kept as local variables since we'll be using them repeatedly in the demo.

The last Service to load requires a bit of initialization: `TpTilePositionDb`. The initializaton just adds the maps that the PositionDb should monitor.

Next, the FileAccess Service is used to try to get the `guidKey`. That value is the GUID of the TScene to load. If this is the first time that this minigame starts then that ends up being an empty string.

The next line initializes the SceneManager component. This actually loads the appropriate TScene from the TSceneList reference of the SceneManager Monobehaviour component. If the passed-in GUID is string.Empty or if the passed-in GUID isn't found at all then the very first TScene is loaded. If the GUID matches a different TScene then that TScene is loaded.

In other words: Determining the location and GUID of the waypoint to use for placing the Player requires reading the save files. If found, the location and GUID of the current waypoint for the TScene are available. If not, the zeroth TScene from the TSceneList is used, and the single waypoint with the m_IsStartWaypoint field = true is used as the start position.

The SceneManager callbacks set up earlier:

```
    sceneManager.OnBeforeTSceneChange += OnBeforeSceneChange;
    sceneManager.OnAfterTSceneChange  += OnAfterTSceneChange;
    sceneManager.OnNewTSceneChosen    += OnNewTSceneChosen;
```

are used when the initial TScene is loaded and every time the TScene is changed.

- OnBeforeTSceneChange: called just after SceneManager.SetScene is called. This is used to clean up the current Unity scene, mask the camera during a scene change, or any other housekeeping required before a TScene change commences.
- OnNewTSceneChosen: The new scene was correctly chosen. In the demo this callback saves the GUID of the new TScene in the filesystem. That way the next time that the demo begins it will use that TScene as the starting TScene.
- OnAfterTSceneChange: The load is complete. All TSceneInitializers have run. Here one performs any final setup. For the demo this includes:

Once we know where to place the Player, we create one if necessary and/or place it in the proper position. When creating a Player we also set up the Camera follower component (placed on the Camera GO).

Finally, we update all the TpZoneLayout components, which will be discussed next.

It’s important to note that while we know where the initial waypoint location is prior to updating the TpZoneLayouts, the actual TilePlus tile used for the waypoint is NOT actually on any Tilemap until after the TpZoneLayout updating is complete and the tiles are actually loaded.

That’s the reason for this:

```
if (startWaypointGuid != string.Empty)
{
    var guid = new Guid(startWaypointGuid);
    while (!TpLib.HasGuid(guid))
        await Awaitable.NextFrameAsync();
}
```

What’s happening here?

Once the waypoint is loaded its GUID will appear in TpLib’s GUID list. This is very conservative and probably redundant but is a good practice if you want to absolutely ensure against a race condition.

Finally, if a save file was located then use the JSON data from the save files to update loaded TilePlus tiles with the current state that they should be using.

Update

The Update method in TdDemoGameController monitors the camera and if the camera orthographic size changes the view is forced to re-layout with the new camera size.

User Input

User input is done in TdDemoPlayerController. It's pretty simple:

- Set up and accept input from the New Input System
- Use the PositionDb Service to see if the Player character prefab can move in the desired direction.
- Ensure that the Player prefab isn't going to move outside of the TScene boundary.

The PlayerController's Update method's *local* method `TestForMove` is used to interact with the PositionDb Service and do these and other tests.

- Convert from the Vector2 obtained from the New Input System callback into an enum value which describes what direction to move in.
- Do some conversions and obtain the Grid and World position of the Player prefab.
- Use the PositionDb Service to ensure that the Player stays on the road tiles.
- Use the PositionDb Service to see if there are any blocking tiles.
- Test the set of spawned GameObjects to see if there are any collider intersections.
- Determine what direction the Player should rotate to and perform the rotation.

This shows the hybrid approach used by the Layout system: PositionDb for tile collisions and colliders for GameObjects.

One advantage using the PositionDb for tiles is that the PositionDb responds to changes within one frame with low compute overhead. This is important when using the tweener since you want to be able to collide with the sprite and not the tile. A Tilemap collider can certainly adapt to sprite position and scale changes. However, when tweening these changes are happening every frame and that's extra work for the Tilemap collider.

What about Layout?

In this demo, that's performed in LayoutDemoLayoutService.

When the Player moves, TdDemoPlayerController invokes its OnPlayerHasMoved callback to TdDemoGameController. Let's take a look at that code in TdDemoGameController:

```
private async void OnPlayerHasMoved(Vector3Int newPlayerGridPosition, Vector3
currentPosition)
{

    if (!layout ||
!gameState)

return;

    if (layout.LayoutIsRunning) //if layout is still in progress we just
return.

return;



    //do a layout
```

```
pass.
    var layoutSuccess = await
layout.UpdateLayout(currentPosition,

newPlayerGridPosition,

m_Grid!,

m_Camera!,

gameState.SceneManager!,

m_LayoutMessages);
    if
(!layoutSuccess)

        Debug.LogWarning("UpdateLayout had error
return...");



}
```

`await layout.UpdateLayout` is the only substantial action taken by this callback. Note that this whole callback chain is Async, hence, the Layout pass doesn't block anything else. Of course, the Layout code runs on the main thread so it *can* slow down your app if you set it up incorrectly.

UpdateLayout

- UpdateLayout checks to see if the Player has moved to a new Grid position. If not, there’s no need to re-layout the Tilemap chunks.
- Any null spawned GameObjects are deleted from the spawned GameObjects list.
- Then for each TpZoneLayout which is currently in use an Awaitable is created which invokes the TpZoneLayout’s UpdateTickAsync, which we’ll talk about next.
- WhenAll is used to await the completion of all the Awaitable tasks.
- Tiles are messaged.

Why Message Tiles Now?

Since the Player has moved to a new Grid position, all tiles that take action based on the Player's position need to know about the change.

At this stage, TilePlus tiles that are messaged may post an event to TpEvents.

- Waypoints: Waypoints can just save game data and the current waypoint position, or it can do that AND change to a new TScene.
  - The m_IsLevelChange field in the tile controls whether the waypoint changes level or not.
    - false: Save the TScene's data (ChunkingDemoSavedData class is JSONized).
    - true: Save the TScene's data and load a new TScene based on the waypoint's m_NextLevelGuid field.
- Treasure Chests: if the Player is within the Zone set by each chest then the chest animation is run, etc.

Waypoint Saves:

This updates the save data: if the Event was from a waypoint then we record the waypoint's position and GUID in the ChunkingDemoSavedData instance, then disable all other waypoints in the TScene. Then the GUID of the scene is saved and the overall game data are saved.

TpZoneLayout.UpdateTickAsync

UpdateTickAsync uses information from the method's parameter list and from the TpZoneLayout's serialized fields in order to determine what to unload and load. Unloading means removing Chunks of tiles and Loading means adding Chunks of tiles. Chunks can be any size from 4x4 and higher. Note that the Chunk Size is specified as a single integer and not a Vector2Int: this means that chunks are always square. Chunk Size can't be less than 4 and must be an even number.

Basically, the area in the Camera view area (plus padding) is examined for any empty Chunks using data about already-loaded chunks which are maintained in the TpZoneLayout's spawned TpZoneManager. As the Camera moves around, new empty chunks will be encountered, and they're loaded. Additionally, already-loaded chunks will pass out of the Camera view area (plus padding) and such chunks are deleted.

It's deceptively simple; most of the actual loading and unloading is performed in TileFabLib and the "Chunkifying" of TileBundles is performed in the TileBundle assets themselves.

Revision #15
Created 2025-07-11 11:50:25 UTC by Vonchor
Updated 2025-09-21 17:21:36 UTC by Vonchor