

# Top-Down Layout

## Important:

**This demo requires the New Input System and won't work unless it's enabled.**

[Layout Demo.png](#)

## Purpose

**This demo uses almost every main feature of the TilePlus system.**

- Tile Scenes and Initializers
- TileFabs and Bundles
- TpMessaging, TpSpawner, TpTilePositionDb, and TpTileTweeners Scriptable Runtime Services (SRS)
- Tile Ui: UiAnimButtonTile, UiAsciiCharTile, UiButtonTile, etc.
- TilePlus Events, EventActions, ZoneActions
- TpFlexAnimatedTile, TpSlideShow, TpAnimatedSpawner, TpImmortalizer, TpAnimZoneSpawner, TpZoneAnimator

If you aren't yet familiar with what a SRS is, please see [this](#).

If you look in the Demo code you'll see a SRS folder. There you'll find several SRS scripts:

- ChunkingDemoFileAccess: a simple data save/restore back-end
- ChunkingDemoGameState: all of the global state for this minigame.
- ChunkingDemoLayout: the customized code for using the Layout system with this minigame.
- DialogBox: pops up a TileUi-based dialog box.

Next, look at the Demo code's Components folder. Here you'll find:

- ChunkingDemoPlayerController: interacts with the New Input system to move the player character.
  - If the player moves from one Grid position to the next an event is issued.
  - ChunkingGameController is the subscriber to that event.
- ChunkingGameController: custom code for this demo.

- When it gets the OnPlayerHasMoved event from PlayerController it invokes UpdateLayout on the ChunkingDemoLayout Service.

ChunkingGameController's Start doesn't do much:

- Verifies that the fields in the inspector are set up properly and load the player prefab.
- Loads and initializes the demo's services
- Loads and initializes the PositionDatabase Service. This requires initialization: others used by this demo like TpMessaging and TpSpawner don't require any initialization.
- Tries to get the 'GuidKey' from the filesystem
  - the GuidKey is the last-used TScene GUID.
  - TScenes are discussed elsewhere in documentation about the layout system.
  - They're used by the SceneManager
- Initializes the SceneManager.
  - The SceneManager loads the first TScene.
    - A TScene is a specification that tells the layout system what to load on which Tilemap.
  - The SceneManager issues events during this process:
    - OnBeforeSceneChange is issued just prior to anything actually occurring but after the SceneManager evaluates all its inputs for correctness. Here, the event handler just clears 'Collidables'.
      - Collidables are just prefabs which have been instantiated and placed during the scene loading.
      - It makes sense that when changing to a new TScene you'd want to auto-delete these.
    - OnNewTSceneChosen is issued when a new TScene is chosen.
      - This handler Creates a new blank game dataset for this scene.
        - If there IS a save file existing from a previous playthrough this will get updated with the saved values.
        - We also save the GUID key for the new scene: that way the next time the minigame starts up it'll use the save data/
    - OnAfterTSceneChange is issued when the SceneManager has completed its work.
      - This is the most complex part of the TScene loading process: it's where you customize.
    - The code comments should help but basically:
      - Restore Data
        - If not possible, use a new Game Dataset.
      - Check the waypoint data to see if we can find an initial waypoint from the save-game data.
      - If not, we check all the waypoints to find one that has the 'start waypoint' field = true.
      - Then we move the camera to that spot.
      - And move the player character to that same spot
      - UpdateLayout is called: note that throttling is inhibited so it loads immediately in this case.

- Normally throttling is used to smear the work across multiple frames.
- Wait for the start waypoint tile to appear in TpLib's data: we know it's been loaded.
- ClearAllCollidables again to be sure.
- Update all loaded tiles: this uses the save data to restore said data to tiles that implement the ITpPersistence interface.

At this point the code mostly runs on its own: events from PlayerController cause re-layout as the player character moves around.

As the player character moves around, its position is messaged to all tiles implementing ITpMessaging<PositionPacket>.MessageTarget. This packet is used to send the current Tilemap position of the player character to each of these tiles. If the position sent matches the position of a tile, it posts an Event to TpEvents.

[TpEvents](#) is outwardly pretty simple: it just stores references to any tile which 'posts' an event. The ChunkingDemoGameState Service subscribes to TpEvents, and gets notified when an event is posted; setting a flag variable.

Note that ChunkingDemoGameState is less of a 'service' and more like a global variables class; that is, a singleton. In a non-demo app you probably don't want to use this approach but it's easier to understand.

The Update method of ChunkingDemoGameState AND the UpdateTiles method of ChunkingDemoLayout both use TpEvents.ProcessEvents.

Why both?

In ChunkingDemoLayout, TpEvents.ProcessEvents is called after we have sent position data to tiles. So it makes sense to process events here. However, if the app is using the "TileUi" feature of TPT, these tiles can and will send events if a tile is clicked on (or otherwise messaged via the 'New Input System'). Hence, ChunkingDemoGameState's UpdateMethod handles any of these type of UI events.

If you look at the code for ProcessEvents it doesn't look like it does much. However it's a really important part of the system.

Basically, what it does is look at all the tiles in the HashSet of tiles which had posted events. Unless you have a lot of overlapping tiles receiving position messages, there will usually be only one instance to evaluate.

A really simple method would be to look at each tile, determine its type, and have custom code for each situation. Obviously, that's really bad. Hard to add features, easy to create bugs, hard to maintain.

In the TPT system, each tile instance has a field for a scriptable object called an EventAction. There's also one for a ZoneAction, which is a different thing I'll discuss a bit later.

Event Actions have one method: Exec, and one property: Incomplete.

As ProcessEvents looks at each tile in its HashSet it sees if there is an EventAction reference. If there is, it calls Exec and passes in the tile reference along with an optional object called the EventActionObject, which is an object (c# object) with whatever sort of data that you want to pass to EventAction.Exec.

You can see the use of EventActionObjects in UIButtonTile. Here, we pass along a tag string, the zone bounds, and the tile instance (which is redundant). The TileUi demo has UIButtonToAsciiEventAction which uses this mechanism to pass along the target control.

Being able to pass-in arbitrary data from the tile to the EventAction means that the EventAction itself does not have any state information. That's important because it's a ScriptableObject and changing anything in it during Editor-Play will alter the fields. Not good. Avoiding state info in the EventAction means that we can use this S.O. for several tiles of the same type or of any other type.

If the Incomplete property is true, the event isn't discarded and is returned ProcessEvents' caller for further custom processing.

As implemented in this demo, everything is done via Event and Zone actions. Hence, no list for returned tile instances is even provided to ProcessEvents.

## Let's examine an EventAction used for TreasureChests.

The minigame's goal is to get all of the chests. Can't change to another level (TScene) until that's met.

Open the Scripts/Actions/TreasureChestEventAction.cs and look at Exec().

After some simple validity checks, the treasure chest count is incremented. This is saved on a per-level basis in the game data which is periodically saved to the filesystem.

It checks to see if it's 'parent' tile actually implements the ITpPersistence interface. If it does, the tile's save data is obtained and 'Poked' into the save data using the ChunkingDemoGameState Service.

Next it does something sort of fun: opens a tile-based dialog box. The dialog box is a pre-made TileBundle which is just loaded to a separate tilemap for UI tiles.

After setting a flag in the GameState to temporarily block other UI (ie - a modal dialog) it just waits for a callback to unblock ui, which occurs when the user closes the box.

# Here's an EventAction used for Waypoints

You can find WaypointEventAction in the same folder.

This action checks for the proper type of tile, disables all other waypoints, and checks to see if the waypoint is marked as a level change waypoint.

If not, it saves the game, directly from this S.O. `` If this IS a level change waypoint, is the SceneExitGoalAchieved?

If it is, Save the game and change levels (TScene).

As you can see, the EventActions are away to move object-specific code from a core part of your app back to the object itself. The relatively simple mechanism used by ProcessEvents can handle it all.

---

Revision #8

Created 13 July 2025 14:30:04 by Vonchor

Updated 22 July 2025 12:01:11 by Vonchor