TilePositionDb Service

Introduction

The TilePositionDb Service monitors Tilemap callbacks to update a small internal dataset.

The dataset keeps track of which tile positions are occupied on the specific Tilemaps that you tell it to monitor.

TilePositionDb can optionally keep track of tiles with scaled sprites (x,y > 1) and position-shifted sprites.

Given this information, use query methods to see if a position is occupied by another tile even if the tile sprite is enlarged; for one or several Tilemaps at once.

This is useful, for example, as part of a custom Grid Graph for the A* Pathfinding Project (code available on request).

TilePositionDb supports time-varying size and position, such as you'd have when tweening a TPT tile's scale. BUT: ***It doesn't support accurately tracking rotated sprites.***

If a sprite is rotated and Warnings are active (Checkbox in TilePlus.Config) then a warning message is printed to the console the first time that a rotated sprite is encountered. To avoid console spamming this warning occurs only once per run-session.

Use

Please check out the TpInputActionToTile component. This shows how to use position DB via the TpActionToTile Scriptable Object.

Here's an example that can guide how to use it yourself: First, initialization.

```
 var posDb =
GetServiceOfType<TpTilePositionDb>();
 if
(!posDb)

     return;
```

```
 //Add a map to monitor
 posDb.AddMapToMonitor(m_Tilemap);  //you have a field with this reference.

//-- Optional, not needed if there are no active tilemaps when the game starts.
//-- this is generally only needed in the Editor since there can be a scene already present
//-- when you click the PLAY button.
//it may be the case that the scene has tilemaps with existing tiles
//at startup. Hence, add all existing positions 'manually'
//duplicates are rejected in the AddPosition
method.
var n =
tilemap.GetUsedTilesCount();
if (n !=
0)
{


tilemap.CompressBounds();
  foreach (var pos in
tilemap.cellBounds.allPositionsWithin)
     posDb.AddPosition(tilemap, pos);
  posDb.ForceUpdate(); //Force an Update so that the internal data structures are
refreshed.

}
```

Querying - here's the most basic query:

```
public bool PositionOccupied(int            mapId,
                             Vector3Int     position,
                             out Bounds     scaledSpriteBounds,
                             out Vector3Int tilePosition,
                             Vector3Int?    ignoreThisPosition = null,
                             bool           ignoreScaledSprites = false)
```

This method takes the

- instance ID of a Tilemap

- the position you're testing

and optionally:

- scaledSpriteBounds: if what's found is a tile with a scaled sprite then this is its bounds.
- tilePosition: if what's found is a tile with a scaled sprite then this is the tile's actual position. The tile may not be congruent with the sprite.
- ignoreThisPosition: if two tile sprites overlap, this position is ignored.
- ignoreScaledSprites: this is much faster if you're sure there aren't any scaled or position-shifted sprites on your Tilemap.

For `ignoreThisPosition`, if adjacent tiles have overlapping sprites overlap then the return value would be true. But it's indeterminate which sprite (of the two) would have its tile's position returned. If you want to exclude one of those positions use this parameter. Check out the `JumperTile` TPT tile script for an example.

There's a similar Query called `PositionOccupiedForWorldCoordinates.` This useful for certain uses. See the TpActionToTile script for an example.

If you'd like to test on multiple Tilemaps, use

`public bool PositionOccupied(int[] mapIids, Vector3Int position, Vector3Int? ignoreThisPosition = null )`

There are also shortcuts for really simple (and of limited use) "pathfinding" - these mostly exist for really simple uses.

`public bool IsPathBlocked4Way(int mapId, Vector3Int position, TpTileUtils.DirectionType4 direction, uint distance=1)`

`public bool IsPathBlocked4Way(int[] mapIids, Vector3Int position, TpTileUtils.DirectionType4 direction, uint distance = 1)`

`public bool IsPathBlocked8Way(int mapId, Vector3Int position, TpTileUtils.DirectionType8 direction, uint distance=1)`

`public bool IsPathBlocked8Way(int[] mapIids, Vector3Int position, TpTileUtils.DirectionType8 direction, uint distance = 1)` These test for any blockages in one or more Tilemaps in a given 4- or 8-way direction, and for a certain distance.

They're fairly inefficient if you have many Tilemaps.

If you want to use something like A* PathFinding Project, the PositonOccupied methods are the basis of a Grid Graph plugin.

What is this?

A mini-database of occupied positions on Tilemaps.

This was originally intended for use with the 'Chunking'/Layout mode of TilePlus Toolkit, although there's no restriction on other uses. It's also optionally used by TpActionToTile, TpInputActionToTile, and other code where one wants to locate tiles on a tilemap from a mouse position.

Since it also keeps track of tiles whose sprites are larger than one Tilemap unit, you can locate tiles EVEN if what you click on is the sprite outside of the tile's single position. It also properly handles sprites where the position has changed and the sprite's bounds no longer overlap the tile's native sprite bounds. But for efficiency, a sprite's rotation is ignored.

You can see this in the Oddities/Jumper demo where some of the tiles have sprites bigger than one unit.

When using the Chunking system: even if you have a huge Tilemap only the parts of it within the camera view (plus optional padding) are loaded into the Tilemap. Hence, the HashSets in this Service never get that large.

HOWEVER, this assumes that you do not use this subsystem to monitor dense maps such as a 'floor' or 'ground' Tilemap where almost all positions are filled. One should use colliders or some other approach for dense maps.

Sparse tilemaps make sense to use with this Service, so roads, obstacles etc.

Tiles are added and deleted from the data structures in this Service by the OntilemapTileChanged callback (from Tilemap).

Any tiles which are on the specified Tilemaps are added or deleted from a HashSet.

But that only covers one single position. If you have tiles with sprites that are offset by the transform-position of the tile sprite then that doesn't work.

Scaled or position-shifted sprites are handled automatically.

Since evaluating scaled sprites involves some computation and Tilemap access (to get the actual sprite transform) these are cached and evalutated when this SRS is updated. That occurs at EndOfFrame (using the Awaitable pump) or at PostLateUpdate when using the PlayerLoop pump.

Note that TileBase tiles like Rule Tile and AnimatedTile cannot have scaled or position-shifted sprites since these tiles don't have a transform. They appear internally only at their actual Tilemap position. However, this isn't much of a limitation: use TpAnimatedTile or TpFlexAnimatedTile; and Rule Tiles (this one assumes) never have enlarged sprites.

When using the layout system you handle initialization via a callback (see Chunking Demo).

Otherwise you use AddMapsToMonitor (several overloads). After initialization the only way to completely change the setup is to use the .ResetPositionDb method or terminate/restart the service.

Once initialized, when a tile is added or deleted the internal HashSets are updated.

Use PositionOccupied(Tilemap, Vector3Int) to test if there's a tile at a position.

Important

Tiles are deleted from the internal dataset during the Tilemap callback. However, as mentioned earlier, additions are cached and evaluated near the end of the frame.

This means that there's always at least a one-frame delay for additions to the internal data.

Properties

`PosDBRefreshPerUpdate`

The Tilemap callback can dump an enormous amount of data into the update queue. This property can be used to control how many updates are processed each time that the PositionDb updates. This defaults to int.MaxValue.

For example, if you move a 10,000 tiles there will be 10,000 entries in the update queue. If PosDbRefreshPerUpdate is left at it's default value execution will block until all the updates are processed.

The side effect is that updates to the PositionDb's internal data may take more frames to be completely updated.

`MinScaleOffset` and `MinPositionOffset`

MinPositionOffset: Determines the minimum position shift for a tile sprite to be added to the set of position-shifted or scaled sprites.

MinScaleOffset: Determines the minimum transform size change > 1,1 for a tile sprite to be added to the set of position-shifted or scaled sprites.

WARNING

Take care that only one entity affects these properties. It's best to set these once and not have multiple scripts changing these values.

Multiple scripts can add Tilemaps to be monitored. However, it's best to avoid having multiple scripts removing such Tilemaps unless there's some coordination.

Merging

The `Merge` method can be used to combine the 'occupied position' data from multiple tilemaps into one HashSet. This can be handy to have a lookup table that can be used repeatedly rather than using `PositionOccupied` repeatedly.

However, it can be slow - don't update it until you need to. Updating it every Monobehaviour Update is a bad idea. If you need updating that frequently you should use Tilemap colliders instead or stick with `PositionOccupied` instead.

But for a turn-based game where your user does some input and you have a number of on-screen entities that need to move, Merge can be used once and the resultant HashSet can be used to determine if the entities can move to particular locations.

Note that **even if** none of your tiles ever move or have their sprites tweened *but* you are using the Layout system you should be aware that the PositionDb is updated whenever new tiles are loaded or unloaded via the layout engine. Here, you can call `Merge` right after a Layout pass.

HOWEVER: be aware of delays caused by PosDbRefreshPerUpdate, which may delay complete updating for several TpLib Update events. The `PendingUpdates` property can be used to see how many updates remain in the queue.

In the Top-down layout demo, see the `OnPlayerHasMoved` event handler to see where that demo does a layout pass.

Revision #28
Created 2025-06-22 20:11:28 UTC by Vonchor
Updated 2025-08-19 13:21:22 UTC by Vonchor