Events

Events System

Superficially, the Events system implemented by TpEvents seems very simple. A tile calls TpEvents.PostTileEvent with `this` as the method parameter. That passed instance is added to a HashSet. A HashSet is used to ensure that repeated calls to this method from the same TPT tile are ignored. The OnTileEvent subscribed callback is triggered.

The real power of Events can be seen when ProcessEvents is used. When your controlling program gets the OnTileEvent callback it can either handle it immediately or just set a flag, then process the events later; e.g.; during Update.

What sounds like a simple approach would be to obtain the list of TPT tile references by copying them from the HashSet.

And you can do that if you want. Create a List<TilePlusBase> and pass it to TpEvents.ProcessEvents like this:

```
var theList = new List<TilePlusBase>();
ProcessEvents(theList,false);
```

The list will be filled with however many TPT tile references are in the HashSet.

But what does one do with this list of TPT tile references? It's simple to do things like:

(Pseudocode)

```
foreach(var tile in theList)
{
    if(tile is typeof(XYZ))
    {
        Examine the tile's fields and properties
            Do something
    }
    else if(tile is typeof(ABC))
    {
        Examine the tile's fields and properties
```

```
            Do something else.
    }
}
```

Which works but can get very hard to extend or maintain.

It would be nice if the tiles themselves had a way to do whatever function would be done in the control program.

Augh. That just changes the problem - you'd need different TPT tile Types for each "Do Something" variation. For example: Two animated tiles with different "Do Something" would require different subclasses:

```
Class DoSomethingA :TpAnimatedTile
{
    public void DoSomethingA();
}

Class DoSomethingB :TpAnimatedTile
{
    public void DoSomethingB();
}
```

This is probably worse!

Scriptable Objects to the rescue: Event Actions

If you look at a TPT tile asset in the project folder you'll see a field called: Event Action. This field is available via the Painter or Tile+Brush Selection Inspector: check the "Event Support" toggle to view the field.

Event Actions (along with Zone Actions) are a powerful feature which allows adding customizable behaviours to a TPT tile Type via the normal Unity Inspector viewing the TPT tile asset in the Project folder, or on any individual TPT tile instance in a scene via the Painter or Tile+Brush Selection Inspector.

Please be careful not to have any state variables in any Event Action. These are project-folder Scriptable Objects that can be re-used by different TPT tiles and variables:

- In Editor-play mode the actual project asset will be affected.
- The value of the variable will change for each invocation of the Event Action code.

Event Actions have a built-in base class: TpTileEventAction and an interface: IActionPlugin.

IActionPlugin is simple:

```
public class TpTileEventAction : ScriptableObject, IActionPlugin
{
     /// <summary>
     /// A subasset: optional.
     /// </summary>
     /// <remarks>if it exists, the object field in the SelectionInspector will
     /// have an additonal button to open this in a popup inspector.</remarks>
     public ScriptableObject? m_Subasset;

     /// <summary>
     /// A subobject, if any.
     /// </summary>
     ScriptableObject? IActionPlugin.InspectableObject => m_Subasset;

     /// <summary>
     /// DEFAULT if not overriden is TRUE.
     /// If true, this EventAction doesn't do everything
     /// needed, and TpEvents.ProcessEvents will not remove the tile reference
     /// from the output list.
     /// </summary>
     public virtual bool Incomplete => true;

     /// <summary>
     /// Execute event handler.
     /// </summary>
     /// <returns>T/F</returns>
     /// <remarks>Overrides should use base class to ensure tile isn't null</remarks>
     public virtual bool Exec(TilePlusBase tile, object? obj = null)
     {
         return tile;
     }
}
```

If all TPT tiles which emit Events have EventActions then one can do this:

`ProcessEvents()`

In this case, all TPT tiles which have posted events are examined for EventAction scriptable objects. Any EventActions S.O. found have their Exec() method invoked. Those which don't have one are ignored.

Or:

`ProcessEvents(list)`

That does the same thing but with two differences:

Any TPT tiles with EventActions have the Exec() methods run. If the EventAction property `Incomplete` is true, the tile instance is added to the list. This means that you can have a hybrid setup with both the EventAction and some custom code based on evaluating the Types and/or TPT tile instance fields/properties one by one.

Or:

`ProcessEvents(list, false)`

Here, no EventActions are used, all the TPT tiles which had posted events are returned in the list.

But don't do this:

`ProcessEvents(null,false)`

It does nothing at all, and a warning is printed to the console if TpLib warnings are enabled.

Details

EventActionObject

EventActionObject is a virtual property in TilePlusBase which returns an arbitrary c# object. If not null, this object is passed to the EventAction's Exec method(as the second 'object' parameter) when it's called by ProcessEvents(). The base class implementation does this:

```
get => eventActionObject ?? new StandardEventData(string.Empty, m_ZoneBoundsInt, this);
set => eventActionObject = value;
```

Hence, the return value defaults to StandardEventData if the setter is never used.

But one can use any C# object as long as your EventAction understands how to use it.

```
public override object? EventActionObject
{
    get => eventActionObject ?? "12345";
    set => eventActionObject = value;
}
```

You can see how this is used in TpUiButtonEventAction.cs, but the general idea is that you can customize the data sent during ProcessEvents' calls to the EventActions' Exec() method, eliminated much of the need to actually examine the tiles' fields and properties from the EventAction.

Sub-Assets

These are less frequently used, but allow you to add plugins (Scriptable Objects) to EventActions. This uses two fields in the EventAction asset:

```
public ScriptableObject? m_Subasset;
ScriptableObject? IActionPlugin.InspectableObject => m_Subasset;
```

The InspectableObject property provides a Type-invariant way to access the subasset. This is mostly used as a way to display a button for opening an Inspector for the subasset when using Painter or the Tile+Brush. Note that since EventActions are Project assets, changing the subobject in one EventAction changes all uses of that EventAction.

A subasset can be used when you have common functionality that you want to add to an EventAction. For example, you use a specific type of TpTween repeatedly and want to use it in several different EventActions. Rather than code it directly in several different EventAction assets, you can create one asset with the TpTween and add it via the subasset field.

For an example see Runtime/AssetScripts/Actions/TpTweenerSubObject. You add a TweenSpec asset reference to the subobject's asset and play one tween or a sequence using the TweenSpec asset's tween setup. You run the tween from the EventAction's code using the subasset method PlaySequence:

```
public long PlaySequence(TilePlusBase                      tile,
                                 bool relative = true,
                                 int []?                              indices     = null,
                                 int                                  numLoops    = -1,
                                 Action<TpTweenSequence, bool>?       onFinished  = null,
                                 Action<TpTweenSequence, TpTween>? onNextTween = null,
                                 bool                                 evenIfAlreadyRunning =
false)
```

The 'indices' parameter allows you to choose which tweens from the TweenSpec asset to add to the sequence.

Revision #8
Created 21 June 2025 12:33:58 by Vonchor
Updated 6 July 2025 11:49:52 by Vonchor