Messages and Events

TilePlus has three internal messaging systems:

Events and Messages: for Tiles

ServiceMessages: send messages to Services. See the Services chapter.

Overview

TilePlus Messaging and Events

TpLib has general-purpose messaging and Event functionality. Messages get sent to tiles, and tiles can post events. Messages can also be sent between any classes: Monobehaviours or Static classes are supported.

TpServiceManager has limited messaging to any running Services.

Messages functionality uses the TpMessages Static class and Events functionality uses the TpEvents static class.

The use of Messages and Events facilitates event-driven design with TilePlus tiles. As your game proceeds you send Messages to tiles; usually from a Monobehaviour or other code which is responding to user input. Tiles post Events in response.

Messages are sent immediately. Events are stored and evaluated later.

A simple example: a game where there's a player character moving around. Your code messages all tiles interested in knowing where the player is located. If one of the messaged tiles decides that the player is at that tile's position or nearby it "posts" an Event. Your controlling program gets a callback announcing that an Event had been posted. At some point, perhaps during Update, you call a method in TpEvents to process the events.

The `TpInputActionToTile` Component is a convenient way to use the Unity's *New Input System*. Attached to a GameObject and properly configured, it can be used to process the New Input System events and:

- Message a tile using the Messaging System
  - Or get callbacks that a Click or other New Input System event had occurred.
- Automatically handle Events sent from tiles using EventAction Scriptable Objects.
  - And/Or get information about which tiles posted Events and handle them yourself.

It's highly configurable and really useful: check it out in the Components chapter.

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.

Messages

So that's Events. Messages are an entirely different animal.

- Events is a Static class.
- Events is normally used as a store-and-forward events system: tiles post events and something else evaluates them later.
- Messages are sent immediately.

Messages has two main ways of sending messages from one entity to another.

- Messages can be sent to TilePlus (TPT) tiles; for example, to notify a tile of the player’s position. Messages can also be sent from one TPT tile (or from Event or Zone Actions) to other TPT tiles.
- Messages can also be sent from any Object to any other using a Typed subscription system. This includes sending from Monobehaviours to static class instances and vice-versa.

Use with TPT tiles

When a TPT tile receives a message it can optionally espond by posting an event to TpEvents. For example, after receiving a message about the player’s position and finding that it matches the player's position.

Rather than do something like "I want to control something on a tile on Tilemap ABC at position (3,2,0), what was that method I created last week?", use TpMessages.

What's the advantage? Although you can use this Service to send messages to a tile on a Tilemap at a certain position, you usually use it to message several tiles on any tilemap, perhaps filtered for tile type, data packet contents, or a rectangular region.

You can send to all tiles, all with a particular tag, or all of a particular type or interface.

TpMessages uses generic interfaces and generic message packet classes. What does this mean?

`Messages.SendMessage`

This has six different overloads, including one simple one which is rarely needed:

`SendMessage<T>(Tilemap map, Vector3Int position, T packet)`

Clearly this just corresponds to "I want to send a message...".

But let's talk about what this does and what T is in this context.

T is a TypeSpec for a concrete class inherited from an abstract class named `MessagePacket<T>`.

MessagePacket has two basic data items:

- SourceInstance: what sent the message
- Id: a uLong identifier for the message.

A useful simple packet is PositionPacket, which builds on MessagePacket to add a Vector3Int for position. This can be used to send a position of the player or an NPC to one or more TPT tiles.

Another simple packet is BoolPacket, which has (you guessed it) a boolean variable in it.

Similarly, StringPacket has a string.

For more complex use: ObjectPacket. This sort of like a 'Union' of several different types of fields, and is used extensively within the TilePlus system for events and Zone/Event Actions (discussed in other documentation).

AnimatorControlPacket is used by the TpZoneAnimator tile and an animator StateMachineBehaviour to allow a tile to control an Animator and get callbacks from the Animator.

ActionToTilePacket is used by the TpActionToTile Scriptable Object to send New Input System actions to tiles. See the section about 'ActionToTile'.

ObjectPacket, ActionToTilePacket, PositionPacket, and StringPacket all have pools accessible from the TpMessaging service. See various demos for how to use these. Please use the 'using' statement version of the pool access to ensure that the message packet instance is returned to the pool. It's up to you to ensure proper pooling use: if you can't, use unpooled message packet instances.

The Messaging Service maintains a context stack. The message packet and its Type are pushed on the top of the stack before messages are sent to tiles and popped off the stack after the messages are sent. The top of the stack can be 'peeked' so that message recipients can examine the original source packet. You can see how this is used in the Patterns demo program.

Messaging Methods

`SendMessage<T>(Tilemap map, Vector3Int position, T packet)`

is the simplest but least useful method.

If the TPT tile instance is available, you can use

`SendMessage<T>(TilePlusBase tile, T packet)`

Which is a convenience method for exactly the same thing since the tile has built-in properties to retrieve its parent Tilemap and its Grid position.

If you have the GUID of a TPT tile you can also use this:

`bool SendMessage<T>(Guid guid, T packet) where T:MessagePacket<T>`

- This is the only one of these methods which returns a value: false if the tile with a given GUID isn't found.
- C#'s GUID class has conversions between the string, byte, and Guid forms of a Guid. `Guid.TryParse()`
- This method requires the Guid form of a GUID.

If you know the map and have a list of positions:

```
SendMessage<T>(Tilemap map, IEnumerable<Vector3Int>   positions,
               T packet,
               Func<TilePlusBase, bool>? tileFilter = null)
```

Here we see the addition of a filter. This can be used to filter out tiles from those found. For example, you might want to examine some property of a tile and potentially exclude it from messaging by returning false from the filter callback.

TPT tiles can have tags. Here's how to message a group of TPT tiles with tags:

```
SendMessage<T>(Tilemap map, string tag,
             T                           packet,
             Func<TilePlusBase, bool>?   tileFilter       = null,
             Func<T, TilePlusBase, bool>? packetFilter = null,
             RectInt?                    region       = null)`
```

If `map` is null then every Tilemap with TPT tiles is examined to find tiles with tags.

Here we see the addition of a packetFilter. This filter is similar to the tileFilter with the addition of the packet itself as a filter parameter.

Notable is the addition of a region. If you supply a RectInt describing a region then that's used as an additional filter.

```
SendMessage<T>(Tilemap? map,
              Type                        typeSpec,
              T                           packet,
              Func<TilePlusBase, bool>?   tileFilter       = null,
              Func<T, TilePlusBase,bool>? packetFilter = null,
```

```
            RectInt?                region = null)
```

Similarly, this overload sends the packet to all tiles of a particular Type, with both filtering varieties and the region filtering also available. Again, if `map` is null then every Tilemap with TPT tiles is examined to find tiles of matching Type.

```
SendMessage<T, TI>(T   packet,
                   Func<TI, TilePlusBase, bool>? interfaceAndTileFilter       = null,
                   Func<T, TilePlusBase,bool>?   packetFilter = null,
                   Func<TilePlusBase,bool>?   tileFilter = null,
                   RectInt?                      region       = null)
```

This is the most complex SendMessage overload. Here, there's no `map` specification at all, so all Tilemaps with TPT tiles are examined.

What's going on here? This can be used to send messages with packets of Type T to all tiles with a particular interface TI. There's an additional filter, the `interfaceAndTileFilter`. This filter gets the interface type, and the tile instance.

How Does a TPT Tile Handle Messages?

TPT tiles can handle multiple message packet Types by merely adding explicit interface declarations for whatever packets the tile wants to recieve. Here's a simple one from the Waypoint tile used in the Layout demo.

```
void ITpMessaging<PositionPacket>.MessageTarget(PositionPacket sentPacket)
{
    if (m_IsLevelChange)
    {
        //This tile won't respond until the game's Goal has been achieved. In this simple
game, get all of the chests.
        //It blinks to warn you and posts a message to the signboard.
        if (!gameState.SceneExitGoalAchieved)
            StartColorTween();
        else if (tweenerId != 0 && TweenerService)
            TweenerService.KillTween(tweenerId);
    }
```

```
    if (sentPacket.m_Position != m_TileGridPosition)
        return;
    ChangeSlide(true);
    isEnabled      = true;
    WasEncountered = true;
    TpEvents.PostTileEvent(this); //the WaypointEventAction plugin handles disabling other
WPs, saving data, level change etc.

}
```

and this is how the message is sent from within the LayoutDemoLayout Service (slightly edited for clarity):

```
using (messaging.PositionPacketPool.Get(out var pkt))
{
    pkt.m_Position     = newPlayerGridPosition;
    pkt.SourceInstance = null;

    messaging.SendMessage(null!, typeof(CdemoWaypointTile), pkt);

    //update all tiles that implement ITpMessaging<EmptyPacket,PositionPacket> with a new
position BUT filter out
    //the waypoints since they've been messaged already.
    //in this example, it's the NPC spawners, the ZoneSpawners etc
    if (!gameState.HadTileEvent) //don't send out this general message if an event had
occurred.
    {
        bool Filter(TilePlusBase tpb) => tpb.GetType() != typeof(CdemoWaypointTile);

        //This uses the simplest filter: the 'tileFilter'. Here we toss out all Waypoint
tiles.
        messaging.SendMessage<PositionPacket, ITpMessaging<PositionPacket>>(pkt, null, null,
Filter);
    }
}
```

Note that it's irrelevant what Tilemap the CDemoWaypointTiles are placed on. Since Waypoint tiles can change game state (such as changing to a new level) they're messaged first. The second SendMessage executes only if the first SendMessage call didn't result in any messaged tile posting an Event.

The second SendMessage call uses the more complex SendMessage call that allows you to only message tiles implementing ITpMessaging<PositionPacket>. The filter is the local method just above the second SendMessage call.

This must be terribly inefficient!!

You might think that one would need to tediously examine all these different Tilemaps to locate tiles to message, and it would be slow and inefficient. That would be true if SendMessage worked that way!

This isn't ye olde Unity GameObject `SendMessage`!!

Packets can only be sent to TilePlus tiles. These tiles 'register' themselves in TpLib data structures and 'deregister' themselves when they're deleted from a Tilemap.

The registration process stores Types, tags, interfaces, and other information that make locating tiles both fast and tilemap-independent. Tiles are found with fast Dictionary and HashSet lookups.

Here's how the overload with tags works:

```
public int[] SendMessage<T>(Tilemap map,
          string                       tag,
          T                            packet,
          Func<TilePlusBase, bool>?    tileFilter       = null,
          Func<T, TilePlusBase, bool>? packetFilter = null,
          RectInt?                     region       = null)
            where T : MessagePacket<T>
{
    SaveContext(packet);

    using (S_TilePlusBaseList_Pool.Get(out var list))
    {
        GetTilesWithTag(map, tag, list, tileFilter, region);

        var num = list.Count;

        if (packetFilter != null)
        {
            for (var i = 0; i < num; i++)
            {
                var tile = list[i];
                if (!messaged.Add(tile.Id))
```

```
                    continue;
                if (!packetFilter(packet, tile))
                    continue;
                var tgt = tile as ITpMessaging<T>;
                tgt?.MessageTarget(packet);
            }
        }
        else
        {

            for (var i = 0; i < num; i++)
            {
                var tile = list[i];
                if (!messaged.Add(tile.Id))
                    continue;
                var tgt = tile as ITpMessaging<T>;
                tgt?.MessageTarget(packet);
            }
        }
    }

    var ary = messaged.ToArray();
    PopContextAndDiscard();
    return ary;

}
```

The line `GetTilesWithTag(map, tag, list, tileFilter, region);` illustrates this. Without getting too deeply into the weeds here, that method examines an internal dictionary `s_TaggedTiles` to locate tiles to message.

It's very fast and (generally) scales linearly to larger number of tiles and/or tilemaps.

Messages Between Objects

TpMessages has a secondary, separate typed-subscription system using the same MessagePacket scheme.

Recall that messages to TPT tiles requires implementation of explicit interfaces. Here, you need to subscribe to a Type of message packet.

`public static ulong RegisterMessageTarget<T>(Object? parent, Action<object> target) where T : MessagePacket<T>`

You register your target to a particular variety of MessagePacket (which can be any arbitrary blob of data as long as the class is derived from MessagePacket). If the parent is UnityEngine.Object such as a Monobehaviour then null-checks ensure that if the Object becomes null then it is automatically deregistered. Leave parent null when using a static class instance (which can't become null);

The target is an Action which receives a c# object which you have to cast back to the Type of the MessagePacket.

For example (not complete and just for illustration):

```
public class Example :  Monobehaviour
{

        public void Start()
        {
            Messages.RegisterMessageTarget<PositionPacket>(this, PlayerMoveCompleteTarget));
            return;



            void PlayerMoveCompleteTarget(object obj)
            {
                var packet = (PositionPacket) obj;
                Debug.Log($"Player moved to {packet.m_Position});
             }


        }
```

If this Monobehaviour's parent GameObject becomes null then the 'registration' is auto-deleted.

Calls to RegisterMessageTarget return a ulong ID which can be used to unregister.

There are two Unregister methods:

```
public static bool UnregisterMessageTarget(Type parentType, ulong id)
public static bool UnregisterMessageTarget(ulong id)
```

The first method is faster. Both return false if the operation failed.

Sending a Message

`public static bool SendToTargets<T>(MessagePacket<T> packet) where T : MessagePacket<T>`

This generic method sends the message packet to all targets for that particular message packet Type. In the context of the above example:

```

var packet = new PositionPacket();
packet.m_Position = new Vector3Int(1,2,3);
Messages.SendToTargets(packet);
```

This is a simple but powerful way to have event-driven messages controlling your gameplay systems without lots of dependencies.

Zone Actions

Zone Actions are similar to Event Actions but they're not automatically invoked in the same way as EventActions.

These are normally invoked by MessageTarget methods: the targets for SendMessage.

While it's certainly possible to invoke a ZoneAction's Exec method from anyplace in your code where you have the tile reference, it's a great way to introduce bugs. It's best to invoke these from TPT tile MessageTarget methods rather than have invocations scattered around your codebase.

Here's an example from TpSlideShow (edited for brevity):

```
void ITpMessaging<ActionToTilePacket>.MessageTarget(ActionToTilePacket sentPacket)
{

    if (sentPacket.SourceInstance ==
this)

        return; //avoid
recursion.

    if
(!m_AcceptsClicks)



return;

    if (sentPacket.Delay !=
0)



{



        TpLib.DelayedCallback(this,()=>ChangeSlide(SlideIndex == 0,false,false),
                              "DelayATTP_Slideshow",
sentPacket.Delay);
```

```
if(m_ZoneAction)

            m_ZoneAction.Exec(this, this.GetType(), m_ZoneTargetTag,

ObjectPacket.EventsPropagationMode.ZoneEvents,"",SlideIndex);

}



else

        ChangeSlide(slideIndex == 0,

true,sentPacket.PostEvent);

}
```

So this MessageTarget is a recipient of Messages from TpActionToTile which is the S.O. that converts clicks to tile positions. Having found one, it sends this message and (if there's a delay) invokes the ZoneActions's Exec method.

The Exec method declaration:

```
public virtual bool Exec(TilePlusBase
sourceInstance,
                         Type?
targetType,
                         string
tagString,
                         ObjectPacket.EventsPropagationMode eventPropagationMode =
ObjectPacket.EventsPropagationMode.None,
                         string                             optionalString       =
"",
                         int                                optionalInt          =
0,
                         bool                               optionalBool         =
false,
                         object?                            optionalObject       =
null)
```

As you can see, the parameters have some specific information like:

- targetType (here passed-in as the Type of the invoking TPT tile)
- tagString (here passed-in as a specific value from the invoking TPT tile)

and some general parameters:

- optionalString
- optionalInt
- optionalBool
- optionalObject

These can be whatever you want. In this example, optionalInt is used to pass an index for which slide (sprite) should be displayed.

It's probably obvious that the invoking tile has to know what a ZoneAction needs *and* the ZoneAction has to be designed for a specific purpose: there's an unavoidable dependency.

To be clear, a TPT tile could do whatever the ZoneAction does within its own code. The idea of Zone Actions is that often there will be actions that different varieties (Types) of TPT tiles perform. Using a Scriptable Object plugin like this means you don't have to have similar code peppered throughout your project, improving maintainability and making debugging easier.

Similar to Event Actions, avoid having any state variables in the Event Action Scriptable Object since it may be re-used among several different tiles.

Zone Actions may relay messages to other tiles. This is commonly used by the `UI` tiles but there's no restriction. If you examine the base-class Zone Action (TpTileZoneAction.cs) you'll see that what it does is relay a message to other tiles within the BoundsInt region of whatever tile invoked the Zone Action, with filtering.

The filter has this code:

```
bool FilterFunc(ITpMessaging<ObjectPacket> pkt, TilePlusBase tpb)
{
    if (!tpb)
        return false;

    // ReSharper disable once Unity.NoNullPatternMatching
    if(tpb is not IZoneActionTarget receiver )
        return false;
    if(!receiver.AcceptsZoneAction)
        return false;

    if (!checkTag)
        return true;
```

```
    //this can be slow if you have many tags.
    (var count, var tags) = tpb.TrimmedTags;
    return count != 0 && ((IList)tags).Contains(tagString);
```

The filter rejects TPT tiles that don't implement IZoneActionTarget. That interface has one property: `AcceptsZoneAction` which is simply a way for a TPT tile to say "leave me alone."

The the filter checks for a tag match and discards TPT tiles without matching tags.