Events

This page describes the events sytem included in the TopDown Engine.

Introduction

The TopDown Engine includes its own **event manager**, and relies on it for communication between classes. Events are messages broadcasted by a class in your application that can be caught by any **listening** class so it can take action. For example, let's consider an enemy in your level. When your character kills it, it'll probably increase the score, but also count as progress towards an achievement, and you'll need to update the GUI somehow. You could of course have your Kill method call all these directly, but it creates **coupling** and **dependencies**. Plus sometimes you don't know how many other classes you need to inform of that enemy death. That's where events come in.

Events are a way for you to **broadcast** the fact that something just happened. **Any class can listen to it**, and take appropriate action. They can be very useful to extend the engine and implement your own features without modifying the base code.

MMEventManager

The MMEventManager class is a static class (meaning you don't need to add this as a component in your scenes) that is responsible for event broadcasting, and letting listeners know an event has been triggered. It's the only class you need to use to trigger or listen to an event.

Triggering an event, from any class, is extremely simple, you just need to call YOUR_EVENT.Trigger(YOUR_EVENT_PARAMETERS); For example, here we broadcast an

MMGameEvent named GameStart to all our listeners :

```
MMGameEvent.Trigger("GameStart");
```

Listening to events can be a bit trickier, as there are a few things you need to add to a class so it can listen properly. Make sure you don't forget one of these steps!

Step 1 : specify that your class implements the MMEventListener interface for that kind of event. This is done at the top of your class. For example, this is the MMAchievementDisplayer's declaration :

```
public class MMAchievementDisplayer : MonoBehaviour,
MMEventListener<MMAchievementUnlockedEvent>
{
  // ... the content of the class goes here
}
```

As you can see, here we say that this class will listen to MMAchievementUnlockedEvents.

Step 2 : we need to start listening to events on OnEnable, and stop listening to them on OnDisable. This actually lets the MMEventManager know that this instance will want to be kept informed of any events of that type that get triggered. Make sure you also stop listening to events OnDisable, otherwise the MMEventManager may try to "contact" this instance even if it's been disabled or destroyed, which would cause errors. This is quite simple to do :

```
void OnEnable()
{
  this.MMEventStartListening<MMAchievementUnlockedEvent>();
}
void OnDisable()
{
  this.MMEventStopListening<MMAchievementUnlockedEvent>();
}
```

Of course you'll want to do that for each type of event you want to listen to.

Step 3 :

All that's left to do is implement the MMEventListener interface for that event :

```
public virtual void OnMMEvent(MMAchievementUnlockedEvent
achievementUnlockedEvent)
{
  // here we start a coroutine that will display our achievement
  StartCoroutine(DisplayAchievement
(achievementUnlockedEvent.Achievement));
}
```

The engine contains lots of examples of event uses, check them out, and harness the events potential to improve your own game!

Event Types

The asset comes with a number of predefined event types, but it's built to work with any type of event, as long as they're structs, so feel free to create yours. You can use the following as examples. Bundled with the asset are the following event types :

MMGameEvent :

MMGameEvents are simple, multi-purpose events, made only of a string name. You can use these to trigger events that don't require more information than that name (the game starts, stuff like that).

```
public struct MMGameEvent
{
  public string EventName;
  public MMGameEvent(string newName)
  {
    EventName = newName;
  }
}

// trigger one like this (here we're broadcasting a save event):
MMGameEvent.Trigger("Save");
```

TopDownEngineEvent :

TopDownEngineEvents are an alternative to MMGameEvents. They're defined by a TopDownEngineEventTypes instead of a string. TopDownEngineEventTypes are defined in an enum. This prevents typo errors you could get with strings.

```
public enum TopDownEngineEventTypes
{
  LevelStart,
  LevelEnd,
  PlayerDeath
}

// trigger one like this
TopDownEngineEvent.Trigger(TopDownEngineEventTypes.LevelStart);
```

MMCharacterEvent :

Usually triggered from within an ability, you can use these to let the rest of your game know that a character has performed a specific action. Said actions are defined in an enum. Most of the time you should be fine with the state machine events, but if you need more, there's also this option.

```
public enum MMCharacterEventTypes
{
  ButtonActivation,
  Jump
}

// trigger one like this:
MMCharacterEvent.Trigger(_character, MMCharacterEventTypes.Jump);
```

PickableItemEvent :

Triggered by pickable items when they get picked.

MMDamageTakenEvent :

Triggered every time something takes damage.

CheckpointEvent :

Triggered every time a checkpoint is reached.

MMAchievementUnlockedEvent :

MMAchievementUnlockedEvents are typically triggered by an achievement when it's been unlocked. It can then be caught and used by the GUI to display the achievement, but could also be used to increase score, etc.

MMSfxEvent :

MMSfxEvent are events that allow you to request the SoundManager to play a specific sound. Admittedly, you could also directly do SoundManager.Instance.PlaySound(clipToPlay,transform.position); Both will work. Events allow you to do that without risking an error in the case of a missing sound manager. The event will just be broacasted, nothing will intercept it (and the sound won't play, but at least you won't get an error).

```
// trigger one like this (here we're asking for the playing of a sound
set in the achievement displayer's inspector):
MMSfxEvent.Trigger(achievement.UnlockedSound);
```

Again, events can be any kind of struct, and quite complex ones too, so feel free to create your own ones and use them with this system.

State Machine

StateMachines (used in the Character system for example) can also trigger events automatically **every time they change state**. This is particularly useful as it saves you the trouble of triggering them manually, you can just focus on your listeners. These particular events are MMStateChangeEvent, generic events that can work with any kind of state machine. Grabbing the event will allow you to know the state machine's target, what new state it's in, and in what state it was previously. If you don't plan on using events, you can turn them off for a Character via the Character component's inspector.

```
public struct MMStateChangeEvent<T> where T: struct, IComparable,
IConvertible, IFormattable
{
  public GameObject Target;
  public MMStateMachine<T> TargetStateMachine;
  public T NewState;
  public T PreviousState;

  public MMStateChangeEvent(MMStateMachine<T> stateMachine)
  {
```

```
        Target = stateMachine.Target;
        TargetStateMachine = stateMachine;
        NewState = stateMachine.CurrentState;
        PreviousState = stateMachine.PreviousState;
      }
    }
```