photon

Search

photon

Languages English , 日本語 , 한국어 , 繁体中文

FIND HELP ON DISCORD

QUANTUM | v2 switch to V1 switch to V3

Events & Callbacks

Introduction

The split between simulation (Quantum) and view (Unity) allows for great modularity during the development of the game state and the visuals. However, the view requires information from the game state to update itself. Quantum offers two ways:

- Polling the game state
- Events/Callbacks

Although both are valid approaches, their use-cases are slightly different. Generally speaking, polling Quantum information from Unity is preferable for on-going visuals while events are used for punctual occurrences where the game simulation triggers a reaction in the view. This document will focus on **Frame Events** & **Callbacks**.

Back To Top

used to modify or updates parts of the games state (`Signals` are used for that). Events have a couple of important aspects to understand that help manage them during prediction and rollbacks.

- Events do not synchronize anything between clients and they are fired by each client's own simulation.
- Since the same Frame can be simulated more than once (prediction, rollback), it is possible to have events being triggered multiple times. To avoid undesired duplicated Events Quantum identifies duplicates using a hash code function over the Event data members, the Event id and the tick. See `nothashed` keyword for further information.
- Regular, non- `synced` , Events will be either cancelled or confirmed once the predicted Frame from which they were fired has been verified. See `Canceled And Confirmed Events` for further information.
- Events are dispatched after all Frames have been simulated right after the `OnUpdateView` callback. Events are called in the same order they were invoked with the exception of non- `synced` Events which can be skipped when identified as duplicated. Due to this timing, the targeted `EntityView` may already have been destroyed.

The simplest Event and its usage looks like this:

- Define an Event using the Quantum DSL

```
  event MyEvent {
   int Foo;
}
```

- Trigger the Event from the simulation

```
f.Events.MyEvent(2022);
```

- And subscribe and consume the Event in Unity, where we generate a class for the event, with the prefix `Event`

```
QuantumEvent.Subscribe(listener: this, handler: (EventMyEvent e) => Debug.Log($"M
```

Back To Top

DSL Structure

Events and their data are defined using the Quantum DSL inside a qtn-file. Compile the project to make them become available via the `Frame.Events` API in the simulation.

```
event MyEvent {
  FPVector3 Position;
  FPVector3 Direction;
  FP Length
}
```

Class inheritance allows to share base Events classes and members.

```
event MyBaseEvent {}
event SpecializedEventFoo : MyBaseEvent {}
event SpecializedEventBar : MyBaseEvent {}
```

Use abstract classes to prevent base-Events to be triggered directly.

```
abstract event MyBaseEvent {}
event MyConcreteEvent : MyBaseEvent {}
```

Reuse DSL generated structs inside the Event.

```
struct FooEventData {
  FP Bar;
  FP Par;
  FP Rap;
}

event FooEvent {
  FooEventData EventData;
}
```

Back To Top

Keywords

Synced

To avoid rollback-induced false positive Events, they can be marked with the `synced` keyword. This will guarantee the events will only be dispatched (to Unity) when the input for the Frame has been confirmed by the server.

`Synced` Events will add a delay between the time it is issued in the simulation (during a predicted Frame) and its manifestation in the view which can be used to inform players.

```
synced event MyEvent {}
```

- `Synced` Events never create false positives or false negatives
- Non- `synced` Events are never called twice on Unity

Back To Top
Back To "Keywords"

Nothashed

To prevent an Event which has already been consumed by the view in an earlier predicted Frame to be dispatched again a hash-code is calculated for each Event instance. Before dispatching an Event, the hash-code is used to check if the event is a duplicate.

The `nothashed` keyword can be used to control what key-candidate data is used to test the Event uniqueness by ignoring parts of the Event data.

```
abstract event MyEvent {
  nothashed FPVector2 Position;
  Int32 Foo;
}
```

Back To Top
Back To "Keywords"

Local, Remote

If an event has a `player_ref` member special keywords are available : `remote` and `local`

Before the Event is dispatched in Unity on a client the keywords will cause the `player_ref` to be checked if assigned to a `local` or `remote` player respectively. If all conditions match, the event is dispatched on this client.

```
event LocalPlayerOnly {
  local player_ref player;
}


event RemotePlayerOnly {
  remote player_ref player;
}
```

To recap: the simulation itself is agnostic to the concept of `remote` and `local` . The keywords only alter if a particular event is raised in the view of an individual client.

Should an Event have multiple `player_ref` parameters, `local` and `remote` can be combined. This event will only trigger on the client who controls the `LocalPlayer` and when the `RemotePlayer` is assigned to different player.

```
event MyEvent {
  local player_ref LocalPlayer;
  remote player_ref RemotePlayer;
  player_ref AnyPlayer;
}
```

If a client controls several players (e.g. split-screen), all their `player_ref` will be considered local.

Back To Top
Back To "Keywords"

Client, Server

Since Quantum 2.1

Events can be qualified using `client` and `server` keywords to scope where they will executed. By default all Events will be dispatched on the client and server.

```
server synced event MyServerEvent {}


client event MyClientEvent {}
```

Back To Top

Using Events

Trigger Events

Events types and signatures are code-generated into the `Frame.FrameEvents` struct which is accessible over `Frame.Events` .

```
public override void Update(Frame f) {
  f.Events.MyEvent(2022);
}
```

Back To Top
Back To "Using Events"

Choosing Event Data

Ideally, the Event data should be self-contained and carry all information the subscriber will need to handle it on the view.

The Frame at which the Event was raised in the simulation might no longer be available when the Event is actually called on the view. Meaning that information to be retrieved from the Frame needed to handle the Event could be lost.

A `QCollection` or `QList` on an Event is actually only passed as a `Ptr` to memory on the Frame heap. Resolving the pointer may fail because the buffer is no longer available. The same can be true for `EntityRefs` , when accessing Components from the most current Frame at the time the Event is dispatched the data may not be the same as when the Event was originally invoked.

Ways to enrich the Event data with an `array` or a `List` :

- If the collection data payload is of a known and reasonable max size a `fixed array` can be wrapped inside a struct and added to the Event. Unlike `QCollections` the arrays do not store the data on the Frame heap but carry it on the value itself.

```
  }
  event FooEvent {
    FooEventData EventData;
  }
```

- The DSL currently does not allow to declare an Event with a regular C# `List<T>` type in it. But the Event can be extended using partial classes. See the `Extend Event Implementation` section for more details.

Back To Top
Back To "Using Events"

Event Subscriptions In Unity

Quantum supports a flexible Event subscription API in Unity via `QuantumEvent`.

```
QuantumEvent.Subscribe(listener: this, handler: (EventPlayerHit e) => Debug.Log($"Play
```

In the example above, the listener is simply the current `MonoBehaviour` and the handler an anonymous function. Alternatively, a delegate function can be passed in.

```
QuantumEvent.Subscribe<EventPlayerHit>(listener: this, handler: OnEventPlayerHit);

private void OnEventPlayerHit(EventPlayerHit e){
  Debug.Log($"Player hit in Frame {e.Tick}");
}
```

`QuantumEvent.Subscribe` offers a few optional QoL arguments that allow to qualify the subscription in various ways.

```
// only invoked once, then removed
QuantumEvent.Subscribe(this, (EventPlayerHit e) => {}, once: true);

// not invoked if the listener is not active
// and enabled (Behaviour.isActiveAndEnabled or GameObject.activeInHierarchy is checke
QuantumEvent.Subscribe(this, (EventPlayerHit e) => {}, onlyIfActiveAndEnabled: true);

// only called for runner with specified id
QuantumEvent.Subscribe(this, (EventPlayerHit e) => {}, runnerId: "SomeRunnerId");

// only called for a specific
QuantumEvent.Subscribe(this, (EventPlayerHit e) => {}, runner: runnerReference);

// custom filter, invoked only if player 4 is local
QuantumEvent.Subscribe(this, (EventPlayerHit e) => {}, filter: (QuantumGame game) => g

// only for replays
QuantumEvent.Subscribe(this, (EventPlayerHit e) => {}, gameMode: DeterministicGameMode

// not for replays (Quantum SDK v2.0)
QuantumEvent.Subscribe(this, (EventPlayerHit e) => {}, excludeGameMode: DeterministicG
//=> The gameMode parameter accepts and array of DeterministicGameMode
```

Back To Top
Back To "Using Events"

Unsubscribing From Events

Unity manages the lifetime of `MonoBehaviours` , so there is no need to be unregistered as listeners get cleaned up automatically.

If tighter control is required unsubscribing can be handled manually.

```
var subscription = QuantumEvent.Subscribe();

// cancels this specific subscription
QuantumEvent.Unsubscribe(subscription);

// cancels all subscriptions for this listener
QuantumEvent.UnsubscribeListener(this);

// cancels all listeners to EventPlayerHit for this listener
QuantumEvent.UnsubscribeListener<EventPlayerHit>(this);
```

Back To Top
Back To "Using Events"

Event Subscriptions In CSharp

If an Event is subscribed outside of a `MonoBehaviour` the subscription has to be handled manually.

```
var disposable = QuantumEvent.SubscribeManual((EventPlayerHit e) => {}); // subscribes
// ...
disposable.Dispose(); // disposes the event subscription
```

Back To Top
Back To "Using Events"

Canceled And Confirmed Events

Non- `synced` Events are either cancelled or confirmed once the verified Frame has been simulation. Quantum offers the callbacks `CallbackEventCanceled` and `CallbackEventConfirmed` to react to them.

```
QuantumCallback.Subscribe(this, (Quantum.CallbackEventCanceled c) => Debug.Log($"Cance
QuantumCallback.Subscribe(this, (Quantum.CallbackEventConfirmed c) => Debug.Log($"Conf
```

Event instances are identified by the `EventKey` struct. The previously received Event can be added into a dictionary for example by creating the `EventKey` like this.

```
  // ...
}
```

Back To Top

Extend Event Implementation

Although Events support using a `QList`. When resolving the list the corresponding Frame might not be available anymore. Additional data types can be added using partial class declarations.

```
event ListEvent {
  Int32 Foo;
}


public partial class EventListEvent {
  public List<Int32> ListOfFoo;
}
```

To be able to raise the customized Event via the `Frame.Event` API extend the `FrameEvents` struct.

```
f.Events.ListEvent(f, 1, new List<FP>() {2, 3, 4});.


namespace Quantum {
  public partial class Frame {
    public partial struct FrameEvents {
      public EventListEvent ListEvent(Frame f, Int32 foo, List<Int32> listOfFoo) {
        var ev = f.Events.ListEvent(foo);
        ev.ListOfFoo = listOfFoo;
        return ev;
      }
    }
  }
}
```

Back To Top

Callbacks

Callbacks are a special type of event triggered internally by the Quantum Core. The ones made available to the user are:

Callback	Description
CallbackPollInput	Is called when the simulation queries local input.

CallbackInputConfirmed	Is called when local input was confirmed.
CallbackGameStarted	Is called when the game has been started.
CallbackGameResynced	Is called when the game has been re-synchronized from a snapshot.
CallbackGameDestroyed	Is called when the game was destroyed.
CallbackUpdateView	Is guaranteed to be called every rendered frame.
CallbackSimulateFinished	Is called when frame simulation has completed.
CallbackEventCanceled	Is called when an event raised in a predicted frame was cancelled in a verified frame due to a roll-back / missed prediction. Synchronized events are only raised on verified frames and thus will never be cancelled; this is useful to graciously discard non-synced events in the view.
CallbackEventConfirmed	Is called when an event was confirmed by a verified frame.
CallbackChecksumError	Is called on a checksum error.
CallbackChecksumErrorFrameDump	Is called when due to a checksum error a frame is dumped.
CallbackChecksumComputed	Is called when a checksum has been computed.
CallbackPluginDisconnect	Is called when the plugin disconnects the client with an error. The reason parameter is filled with an error discription (e.g. "Error #15: Snapshot request timed out"). The client state is unrecoverable after that and needs to reconnect and restart the simulation. The current QuantumRunner should be shutdown immediately.

Back To Top

Unity-side Callbacks

By tweaking the value of `Auto Load Scene From Map` in the `SimulationConfig` asset, it is possible to determine if the game scene will be loaded automatically or not and it is also possible to determine whether the preview scene unloading will happen before or after the game scene is loaded.

CallbackUnitySceneLoadBegin , CallbackUnitySceneLoadDone .

Back To Top

MonoBehaviour

Callbacks are subscribed to and unsubscribe from in the same way one as Frame Events presented earlier, albeit through `QuantumCallback` instead of `QuantumEvent` .

```
var subscription = QuantumCallback.Subscribe(...);
QuantumCallback.Unsubscribe(subscription); // cancels this specific subscription
QuantumCallback.UnsubscribeListener(this); // cancels all subscriptions for this liste
QuantumCallback.UnsubscribeListener<CallbackPollInput>(this); // cancels all listeners
```

Unity manages the lifetime of its objects. Therefore, Quantum can detect whether the listener is alive or not. "Dead" listeners are removed with each `LateUpdate` and with each event invocation for specific event type.

For example, to subscribe to the `PollInput` method and set up the player input, the following steps are necessary:

```
public class LocalInput : MonoBehaviour {
  private DispatcherSubscription _pollInputDispatcher;
  private void OnEnable() {
    _pollInputDispatcher = QuantumCallback.Subscribe(this, (CallbackPollInput callback
  }

  public void PollInput(CallbackPollInput callback) {
    Quantum.Input i = new Quantum.Input();
    callback.SetInput(i, DeterministicInputFlags.Repeatable);
  }

  private void OnDisable(){
    QuantumCallback.Unsubscribe(_pollInputDispatcher);
  }
}
```

Back To Top

Pure CSharp

If a callback is subscribed outside of a `MonoBehaviour` the subscription has to be handled manually.

```
var disposable = QuantumCallback.SubscribeManual((CallbackPollInput pollInput) => {});
// ...
disposable.Dispose(); // disposes the callback subscription
```

Back To Top

executed in order:

1. `OnUpdateView` , the view for the newly created entities are instantiated.
2. `Monobehaviour.Awake`
3. `Monobehaviour.OnEnabled`
4. `EntityView.OnEntityInstantiated`
5. `Frame.Events` are called.

Event and Callback subscription can be done in either `Monobehaviour.OnEnabled` or `EntityView.OnEntityInstantiated` .

- `MonoBehaviour.OnEnabled` , it is possible to subscribe to events in code here; however, the `EntityView` 's EntityRef and Asset GUID will not have been set yet.
- `EntityView.OnEntityInstantiated` is a UnityEvent part of the EntityView component. It can be subscribed to via the in-editor menu. When `OnEntityInstantiated` is called, the EntityRef and Asset GUID of the EntityView are guaranteed to be set. If the event subscription or custom logic requires either of those parameters, this is where it should be executed.

OnEntityInstantiated subscription menu as seen in the Editor.

To unsubscribe from an event or callback, simply use the complementary functions:

- Unsubscribe in `OnDisabled` for any subscription made in `OnEnabled` .
- Unsubscribe in `OnEntityDestroyed` for any subscription made in `OnEntityInstantiated` .

To Document Top

Gaming Circle
Industries Circle

Support

Gaming Circle
Industries Circle
Circle Discord
Circle Stack Overflow

Resources

Dashboard
Samples
SDK Downloads
Cloud Status

Connect

Public Discord
YouTube
Facebook
Twitter
Blog
Contact Us

Languages

English
日本語
한국어
简体中文
繁体中文

Terms Regulatory Privacy Policy Privacy Code of Conduct Cookie Settings