# Event system

The event system is at the core of NRP scripting. Events work closely in conjunction with the element tree. Events are triggered when something happens - a player enters a marker, an element is clicked on etc. Each event has a source element, this is the element that performed the action.

## Event handlers

To use the event system, you attach event handlers to elements in the element tree using addEventHandler. When you do this, your function will get triggered for all the events triggered on that element, it's parents (and their parents, etc.) and it's children (and their children). As such, an event handler attached to the *root* element will be triggered when an event occurs for any element. As a consequence you should generally use as specific a handler as you can. If you wish to just see when the player enters a specific marker, just attach the event handler to that marker.

Each event handler has three 'hidden' variables:

- **source**: This is the element that the event originated from.
- **this**: This is the element that the handler is being triggered on (i.e. the one you attached it to with addEventHandler).
- **eventName**: This is the string of the name of the event that was called upon (i.e. the event name that was added with addEventHandler).

Additionally, the server-side event system also has one more 'hidden' variable:

- **client**: This is the client that triggered the event using triggerServerEvent. This is not set if the event was not triggered from a client.

The *source* variable is the most important one for most handlers. You almost always will want to reference this variable to tell what element triggered the event. The *this* variable has some uses for ensuring that an event was emitted by the element that you attached the handler to.

It is *important* to note that events follow the element hierachy. All events are initially triggered on the *source* element, followed by all the parent and children elements. This has few important implications:

- An event triggered on the root element will be triggered on every element in the element tree. This should be avoided where possible.
- All events anywhere in the element tree will be triggered on the root element. This means you can easily catch every event of a type by attaching a handler to the root element. Only do this if you genuinely want every event of that type, otherwise attach it somewhere more specific in the element tree.
- You can attach an event handler to your resource's root element to get all the events triggered by elements your resource contains.
- You can create 'dummy' elements to catch events from a group of child elements
- You can use dummy elements specified in a .map file (e.g. <flag>) and create 'real' representations for them (e.g. objects) and make these real elements children of the dummy element. Event handlers can then be attached to the dummy element and it will receive all the events of the real elements. This is useful for when one resource manages the representation of the element (creating the objects, for example), while another wants to handle special events. This could be a map resource that wants to handle a flag being captured in a specific way - the map resource would (generally) not be aware of the way the flag is represented. This doesn't matter as it can just attach handlers to it's dummy flag element while the other gamemode resource can handle the representation.

The function you attached to an event gets called and passed a bunch of arguments. These arguments are event-specific. Each event has specific parameters, for instance onClientGUIClick has 4 parameters, which are:

```
string button, string state, int absoluteX, int absoluteY
```

The function you attached to this event will be passed these parameters as arguments. You must remember that each event has different parameters.

## Built in events

NRP has a number of built in events. These are listed on the pages Client Scripting Events and Server Scripting Events.

## Custom events

You can create your own events that can be triggered across all resources. This is an important way to communicate with other resources and allow them to hook into your code. To add your own custom event, just call the addEvent function. You can then use the triggerEvent function to trigger that event any time you want - either using a timer, or based on a more general event.

For example, you could be making a Capture the Flag game mode and want to trigger an event when a player captures the flag. You could do this by attaching a event handler to the standard NRP onMarkerHit event and checking that the player entering the marker has the flag. if they do, you can then trigger your more specific *onFlagCaptured* event and other resources could handle this as they please.

## Canceling

Events can be canceled with cancelEvent. This can have a variety of effects, but in general this means that the server will not perform whatever action it would usually do. For example, canceling onPickupUse would prevent a player being given what they tried to pick up, canceling onVehicleStartEnter would prevent the player entering the vehicle. You can check if the currently active event has been canceled using wasEventCanceled. It's important to note that canceling event *does not* prevent other event handlers being triggered.

# Event system

The event system is at the core of NRP scripting. Events work closely in conjunction with the element tree. Events are triggered when something happens - a player enters a marker, an element is clicked on etc. Each event has a source element, this is the element that performed the action.

## Event handlers

To use the event system, you attach event handlers to elements in the element tree using addEventHandler. When you do this, your function will get triggered for all the events triggered on that element, it's parents (and their parents, etc.) and it's children (and their children). As such, an event handler attached to the *root* element will be triggered when an event occurs for any element. As a consequence you should generally use as specific a handler as you can. If you wish to just see when the player enters a specific marker, just attach the event handler to that marker.

Each event handler has three 'hidden' variables:

- **source**: This is the element that the event originated from.
- **this**: This is the element that the handler is being triggered on (i.e. the one you attached it to with addEventHandler).
- **eventName**: This is the string of the name of the event that was called upon (i.e. the event name that was added with addEventHandler).

Additionally, the server-side event system also has one more 'hidden' variable:

- **client**: This is the client that triggered the event using triggerServerEvent. This is not set if the event was not triggered from a client.

The *source* variable is the most important one for most handlers. You almost always will want to reference this variable to tell what element triggered the event. The *this* variable has some uses for ensuring that an event was emitted by the element that you attached the handler to.

It is *important* to note that events follow the element hierachy. All events are initially triggered on the *source* element, followed by all the parent and children elements. This has few important implications:

- An event triggered on the root element will be triggered on every element in the element tree. This should be avoided where possible.
- All events anywhere in the element tree will be triggered on the root element. This means you can easily catch every event of a type by attaching a handler to the root element. Only do this if you genuinely want every event of that type, otherwise attach it somewhere more specific in the element tree.
- You can attach an event handler to your resource's root element to get all the events triggered by elements your resource contains.
- You can create 'dummy' elements to catch events from a group of child elements
- You can use dummy elements specified in a .map file (e.g. <flag>) and create 'real' representations for them (e.g. objects) and make these real elements children of the dummy element. Event handlers can then be attached to the dummy element and it will receive all the events of the real elements. This is useful for when one resource manages the representation of the element (creating the objects, for example), while another wants to handle special events. This could be a map resource that wants to handle a flag being captured in a specific way - the map resource would (generally) not be aware of the way the flag is represented. This doesn't matter as it can just attach handlers to it's dummy flag element while the other gamemode resource can handle the representation.

The function you attached to an event gets called and passed a bunch of arguments. These arguments are event-specific. Each event has specific parameters, for instance onClientGUIClick has 4 parameters, which are:

```
string button, string state, int absoluteX, int absoluteY
```

The function you attached to this event will be passed these parameters as arguments. You must remember that each event has different parameters.

## Built in events

NRP has a number of built in events. These are listed on the pages Client Scripting Events and Server Scripting Events.

## Custom events

You can create your own events that can be triggered across all resources. This is an important way to communicate with other resources and allow them to hook into your code. To add your own custom event, just call the addEvent function. You can then use the triggerEvent function to trigger that event any time you want - either using a timer, or based on a more general event.

For example, you could be making a Capture the Flag game mode and want to trigger an event when a player captures the flag. You could do this by attaching a event handler to the standard NRP onMarkerHit event and checking that the player entering the marker has the flag. if they do, you can then trigger your more specific *onFlagCaptured* event and other resources could handle this as they please.

## Canceling

Events can be canceled with cancelEvent. This can have a variety of effects, but in general this means that the server will not perform whatever action it would usually do. For example, canceling onPickupUse would prevent a player being given what they tried to pick up, canceling onVehicleStartEnter would prevent the player entering the vehicle. You can check if the currently active event has been canceled using wasEventCanceled. It's important to note that canceling event *does not* prevent other event handlers being triggered.