Behaviors

This chapter gives an overview of what Behaviors are and shows how to use them.

- Overview
- Behaviors and Events
- Predefined Behaviors
- Behavior Definition
- Behavior Building Blocks - Behavior Blocks
- EventHandler Tab
- Implementation
- CGI-Studio Default Behaviors
- External Image Behaviour

Overview

This chapter gives an overview of Behaviors.

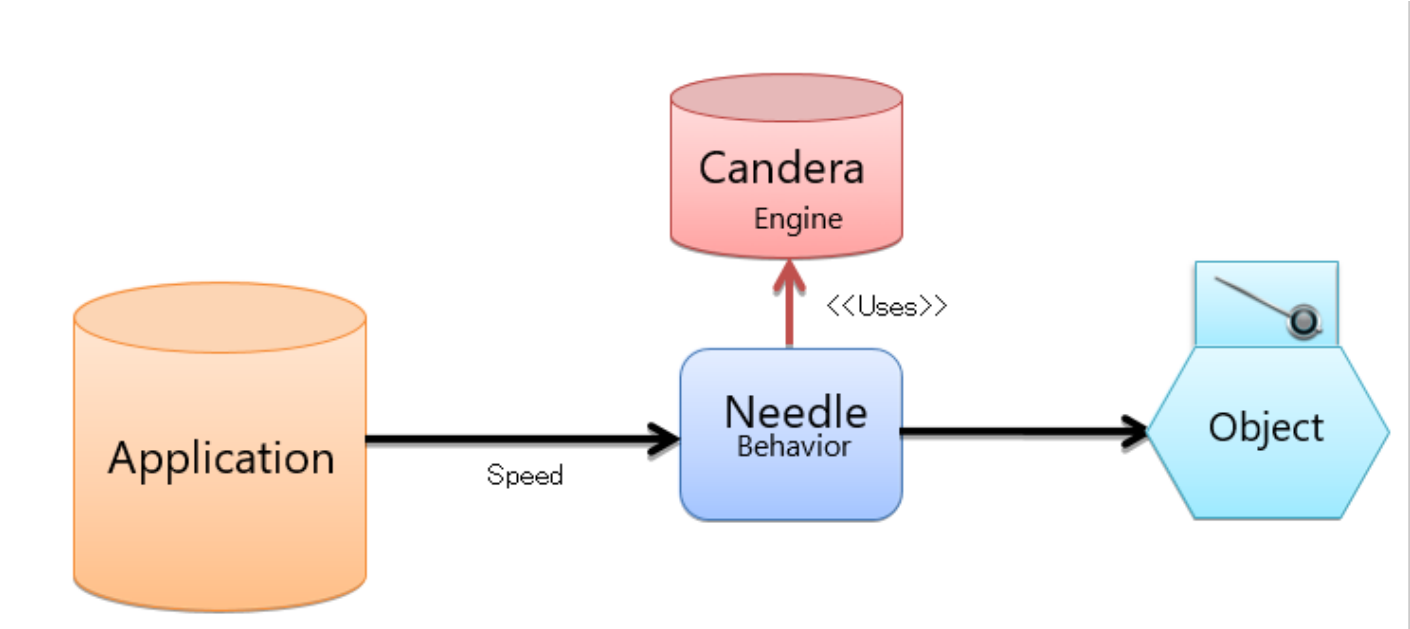
What are Behaviors?

The purpose of Behaviors is to perform modifications to the content presented to the user.

While it would be possible for the application to directly perform any required change directly on the nodes of the scene tree, this would require complex application code. The application would need to be specifically tailored to a single solution and every change in the scene tree might require a change in the application code. To avoid this strong dependency, Behaviors are used to perform the actual modification on the content.

> The following impact has to be considered: Cross-references (nodes or behaviors) to different scenes may occur. For details, see Cross-Reference Implications below.

The advantage of behaviors are that they can be easily used in Scene-Composer by people without any coding knowledge. By adding configurable properties, they can be adapted to apply to multiple different scenarios without the need to create new ones. Further, the application code can be simplified as it only needs to provide an "input" for the Behaviors. It is no longer responsible to decide how the content needs to be modified based on the input.



Since the introduction of Behaviors in CGI-Studio 3.4.2 they are a replacement for Widgets, which are now being considered deprecated. Therefore instead of Widgets, Behaviors should be used to add functionality.

Behaviors can be considered as Widgets 2.0. They can perform any task that widgets can perform but have some added features which greatly enhances their usability. While it is in most cases easy to convert an existing Widget into a Behavior, a simple conversion might not make use of the new functionality introduced, negating the actual benefit from using them.

Comparison to Widgets

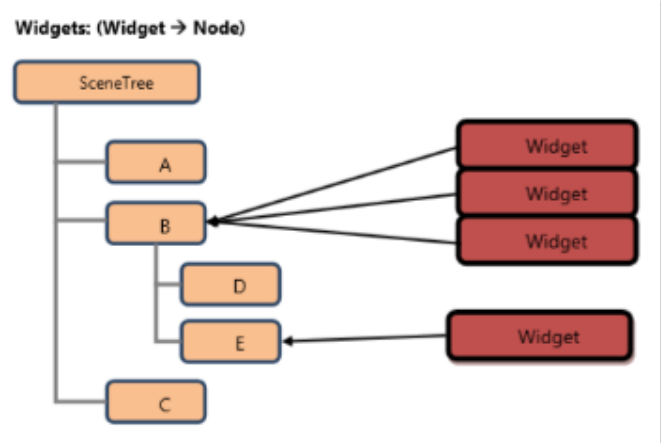
As mentioned behaviors are essentially an enhanced version of widgets. Compared to widgets they have 2 major new features to help with implementing complex functionality.

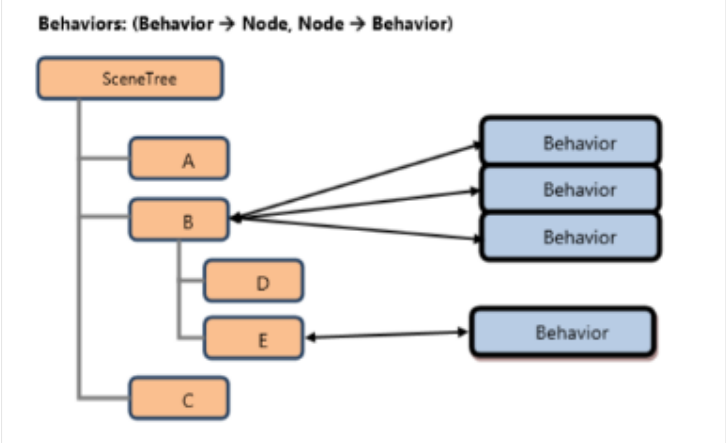
OWNERSHIP

While (most) widgets are attached to a node they are considered part of the scene itself. A widget knows about the node it is attached to, but the node does not know about the widget.

This can cause problems if multiple widgets should collaborate, as by default they can not know which other widgets they should interact with. To overcome this problem, it was required to either specify "other" widgets a property (type is Candera::WidgetBase*) directly in the widget or store custom information on the node about other widgets that are also attached.

Behaviors have simplified this by no longer considering them as part of the Scene but rather part of the node. Each node has a Behavior-Container which hold all attached behaviors for this node. Via this container it is now easily possible to communicate and collaborate between different behaviors.





EVENTS

The second major addition is the usage of events. Events allow to send "information" from one behavior to another behavior without them knowing any details about each other.

Events will be routed throughout the scene tree. This is made possible by the change in ownership mentioned before. Once an event is dispatched to a node, all attached behaviors will receive the event unless the event routing is stopped.

For more information about Events see chapter Behaviors and Events .

Cross-Reference Implications

Behaviors can have properties of type Node or Behavior. If those properties refer to parts of other scenes, those scenes will be automatically loaded by the AssetProvider.

If unloading is required, it is necessary to unload the entire cross-referencing network. Partially unloading cross-referenced scenes of such a cross-referenced network will result in dangling pointers and a will **end up in unpredictable and incorrect behavior!**

Unloading the complete cross-reference network is safe.

The same also has to be considered for Controls and State Machines.

Since the Global State Machine cannot be unloaded, it is not allowed to unload any referenced Nodes or Behaviors.

Behaviors and Events

This chapter explains how the Event-System in Behaviors work.

General

Behaviors communicate via Events with each other and with other components of the application. For each action a specific event is implemented. An Event can carry any data or none at all. For a custom Event simply derive from FeatStd::Event. Each Behavior can return an event-result after handling an Event in the "OnEvent"-method. Behaviors are using unique identifiers (FeatStd::Internal::Identifier) to identify the sender of an event.

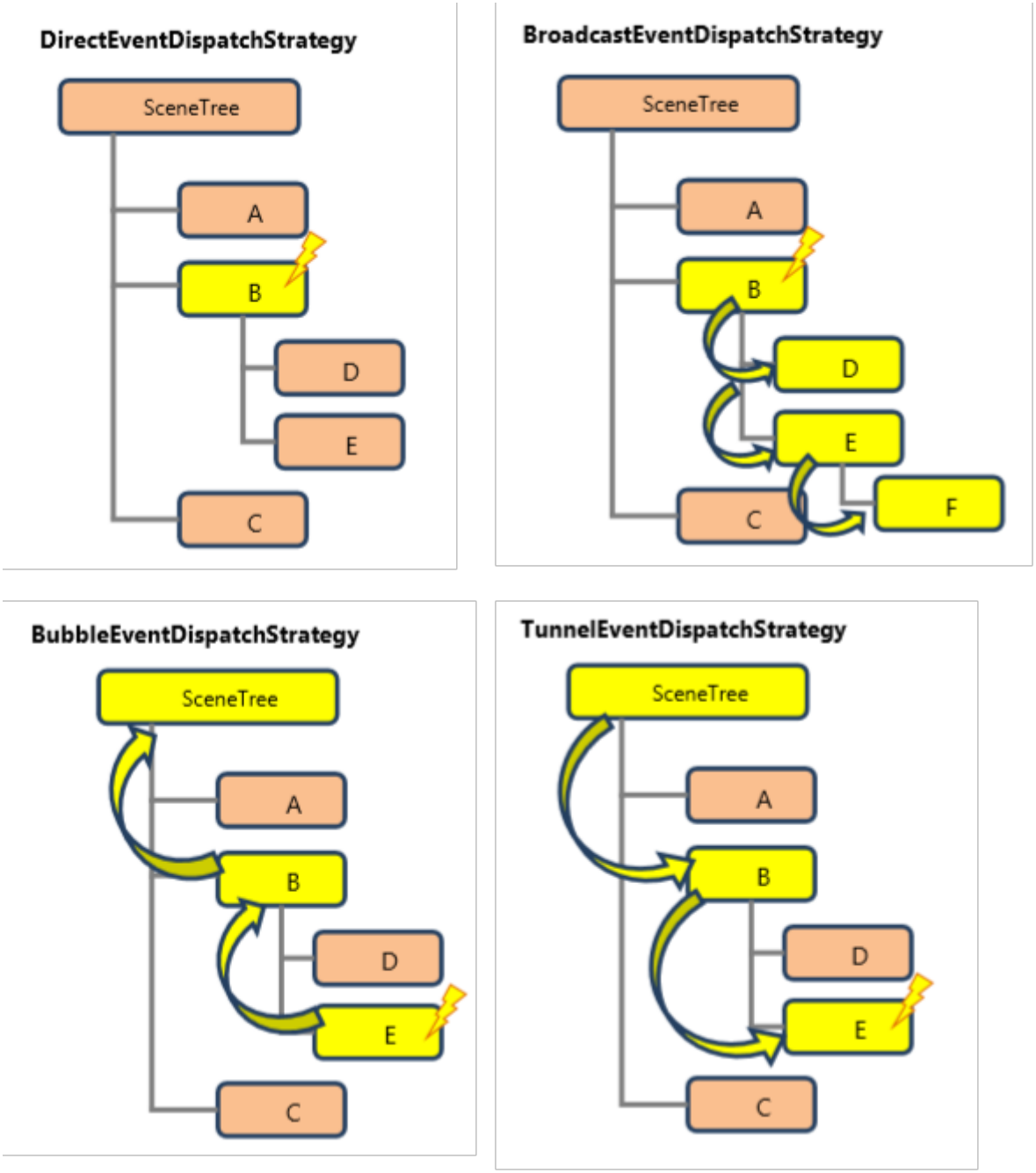
```
        void OnPressedEvent(const PressedEvent& event, Candera::EventDispatchResult& /*dispatchR
        {
            if (event.GetSender() == m_behavior.GetIdentifier()) {
                static_cast<void>(m_behavior.SetState(ControlStateEnum::Pressed
, event.GetType() == PressedEventType::Pressed));
            }
        }
```

For a list with all Events which are used by Behaviors also check chapter Control Behaviors Events . There you can also find their readable names (names in SC).

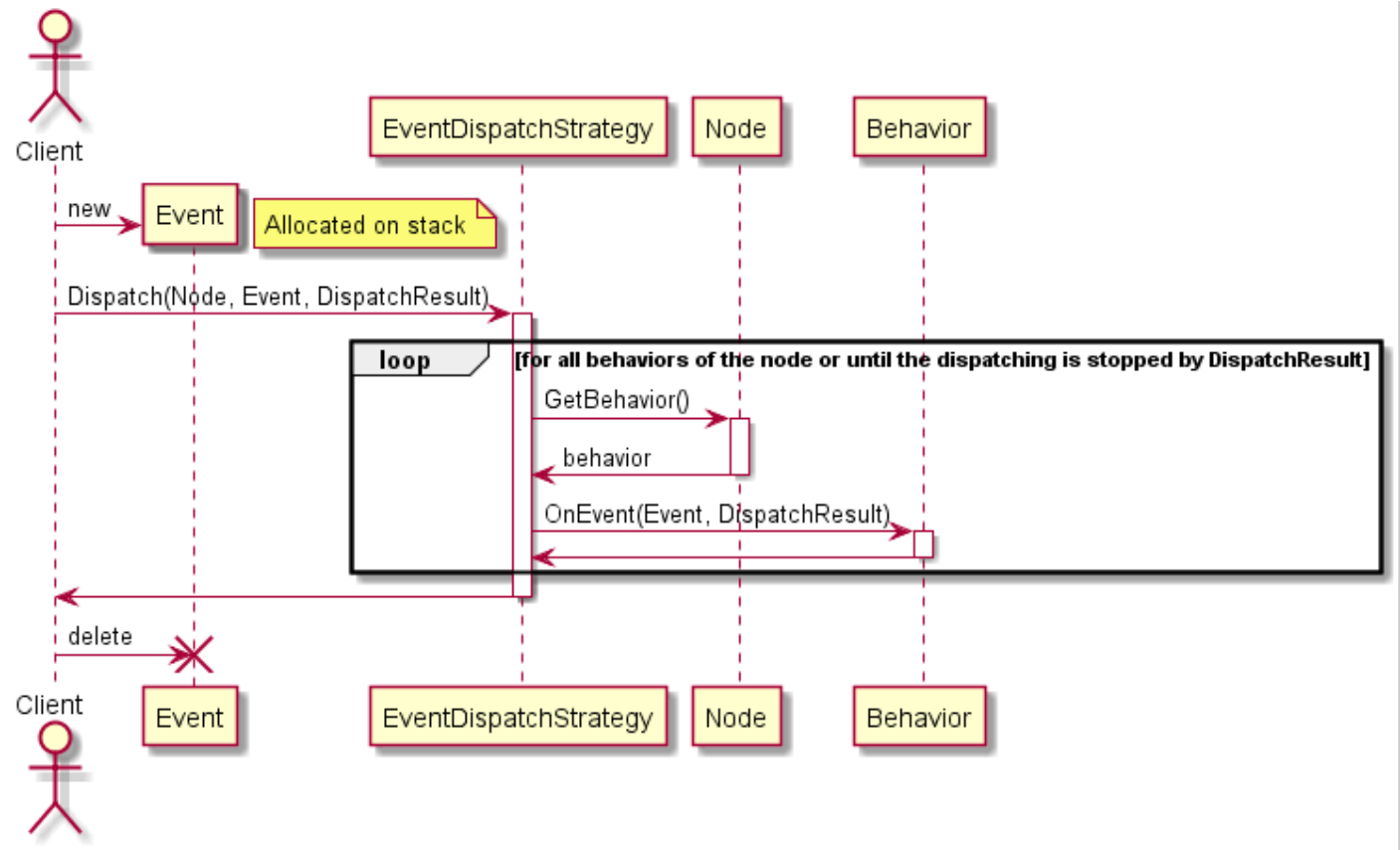
Event Dispatching Strategy

The Behavior base-class implements *DispatchEvent*-functions. Events are dispatched on a Node using one of the available *EventDispatchStrategy* objects. An Event is passed to all OnEvent function of all Behaviors. The following dispatch strategies are available:

- **DirectEventDispatchStrategy:** Only the behaviors of the node will receive the event.
- **BroadcastEventDispatchStrategy:** The event will be first dispatched to the current node then to all child nodes.
- **BubbleEventDispatchStrategy:** The event will be dispatched from the current node to the parent and so on until the root node is reached. If there is a Global State Machine, it also means the event will reach the Global State Machine.
- **TunnelEventDispatchStrategy:** The event will be dispatched from the root node to the current node.



Event Dispatching Diagram



Control Behaviors Events

This chapter gives an overview of the most important control behaviors events. Furthermore they are listed with their readable names (names in SC). For further information of how to use them in SC see chapter EventHandler Tab .

Events

Class Candera::ProcessValueEvent
ProcessValueEvent - Value Processed

Class CgiStudioControl::AnimationEvent
AnimationEvent - Animation State Changed

Class CgiStudioControl::ArTargetEvent
ArTargetEvent - ArTargetEvent event

Class CgiStudioControl::CaretPositionChangedEvent

CaretPositionChangedEvent - Caret Position Changed

Class CgiStudioControl::ChangePropertyEvent

ChangePropertyEvent - Property Changed

Class CgiStudioControl::ChangeSelectionEvent

ChangeSelectionEvent - Selection Changed

Class CgiStudioControl::ChangeStateEvent

ChangeStateEvent - State Change Triggered

Class CgiStudioControl::ChangeValueEvent

ChangeValueEvent - Value Change Triggered

Class CgiStudioControl::ClickEvent

ClickEvent - Clicked

Class CgiStudioControl::ControlTimer::Event

ControlTimer::Event - Timer Expired

Class CgiStudioControl::DragDropEvent

DragDropEvent - Drag and Drop

Class CgiStudioControl::EnterExitEvent

OnEnterExitEvent- On Enter Exit Edit

Class CgiStudioControl::HoverEvent

HoverEvent - Hovered

Member CgiStudioControl::IntervalSwitchBehavior::OnEvent (const FeatStd::Event &event, Candera::EventDispatchResult &dispatchResult)

to handle.

Class CgiStudioControl::IntervalVisualizationEvent

IntervalVisualizationEvent - Interval State Changed

Class CgiStudioControl::PoiUpdateEvent

PoiUpdateEvent - Poi update

Class CgiStudioControl::PressedEvent

PressedEvent - Pressed

Class CgiStudioControl::RegisterSelectionEvent

RegisterSelectionEvent - Register Selection

Class CgiStudioControl::ScrollableDirectionChangedEvent

ScrollableDirectionChangedEvent - Scroll Direction Changed

Class CgiStudioControl::StateEvent

StateEvent - State Changed

Class CgiStudioControl::TrackingEvent

TrackingEvent - Tracking event

Class CgiStudioControl::ValueChangedEvent

ValueChangedEvent - Value Changed

Class CgiStudioControl::VideoControlEvent

VideoControlEvent - Video Control Event

Class CgiStudioControl::VideoStateEvent

VideoStateEvent - Video changed

Predefined Behaviors

This chapter gives an explanation of the Behaviors that come with CGI Studio.

Different types of Behaviors

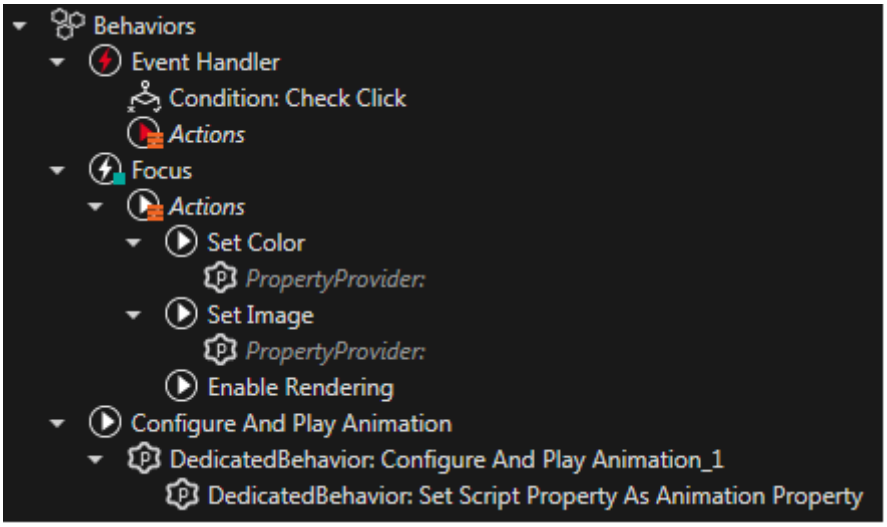
Behaviors are sorted into different categories, which are:

- **Actions:** Example Behaviors are: Play Animation, Start Timer, Set Color, Set Image, Enable Rendering, ...
- **Conditions:** Example Behaviors are: Check Control State, Check Animation State, Check Timer, ...
- **Control:** Example Behaviors are: Control State Handler, Event Handler, ...
- **Value Processing:** Example Behaviors are: Map, Filter, Interpolate, Jump to Keyframe, Set Alpha, Set Rotation, ...
- **Scripting** and **State Machine** related

Visualization of Behaviors as Tree

When many behaviors are nested, the hierarchy of the behaviours is shown thorugh the visualization of the behaviours integral map (= "tree") structure.

For example, if a conditional behavior is used together with another connected behavior, the later is displayed as a child of the conditional behavior. In this way, all behavior properties on behaviors are displayed as a tree view with item children of the owner behavior.

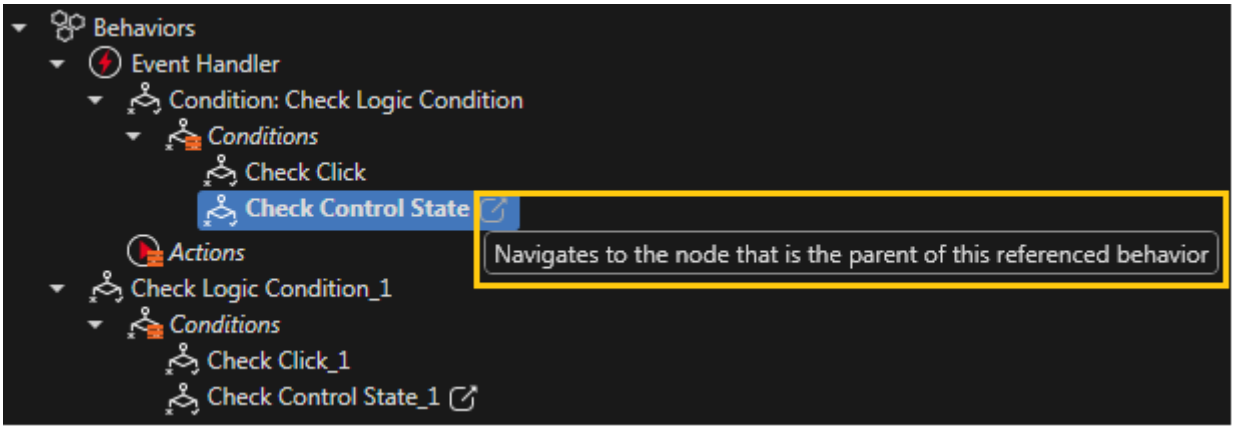


The drag and drop operations only allow behavior of the baseclass specified in the property to be assigned to the main (i.e. "parent") behavior. Any behavior that appears as a child of another behavior will not appears as a standalone behavior in the treeview. If a behavior is removed from

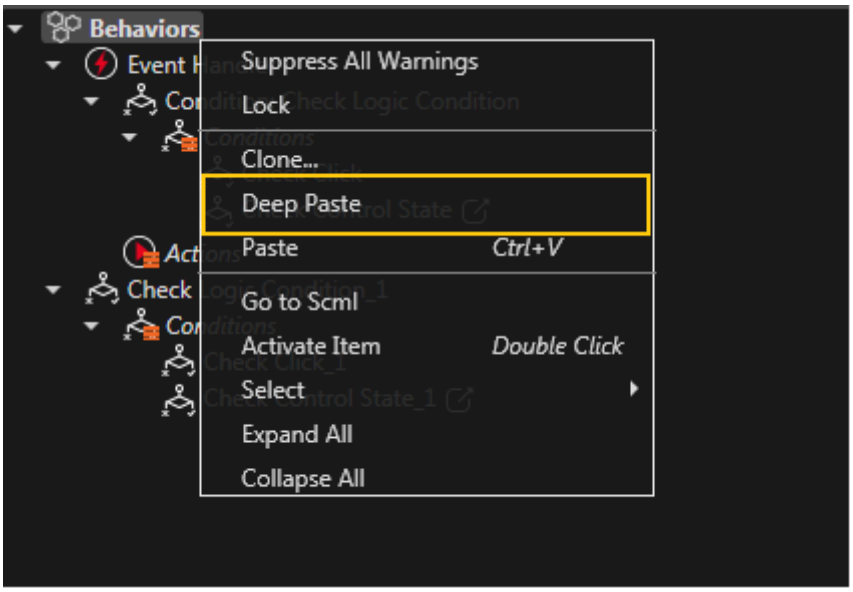
all properties, that behavior will be shown again as a stand alone behavior in the tree.

Behaviors: Deep Copy/Paste Operation

If a behavior is just the reference of another (parent) behavior, the referenced behavior will be marked by a specific sign and a tooltip will become visible on it:



If a behavior which contains referenced behavior is copied, a new option will become available in the context menu: "Deep Paste".



By using this option, referenced behaviors from other nodes will be duplicated, in the location of the original referenced behavior, and the reference will be updated to the duplicated behavior.

When copied, all behaviors will have the following behavior in the Scene extra-Tree:

A) Behavior that are referenced from the copied behaviors will now also get copied to the new location if they are from the same node as the copied behaviors. Behaviors from other nodes will remain as a reference to the same behavior.

- Group/B1
  - ref: Group/B2
  - ref: Camera/B3

When copying and pasting behavior B1 on Group, we will get the following structure:

- Group/B1_1
  - ref: Group/B2_1
  - ref: Camera/B3

When copying and pasting behavior B1 on Button, we will get the following structure:

- Button/B1
  - ref: Button/B2
  - ref: Camera/B3

B) When using the Deep Paste context menu option, referenced behaviors from other nodes will also be duplicated, in the location of the original referenced behavior, and the reference will be updated to the duplicated behavior.

- Group/B1
  - ref: Group/B2
  - ref: Camera/B3

When copying and pasting behavior B1 on Group, we will get the following structure:

- Group/B1_1
  - ref: Group/B2_1
  - ref: Camera/B3_1

When copying and pasting behavior B1 on Button, we will get the following structure:

- Button/B1
  - ref: Button/B2
  - ref: Camera/B3_1

Functional Categories of Behaviors

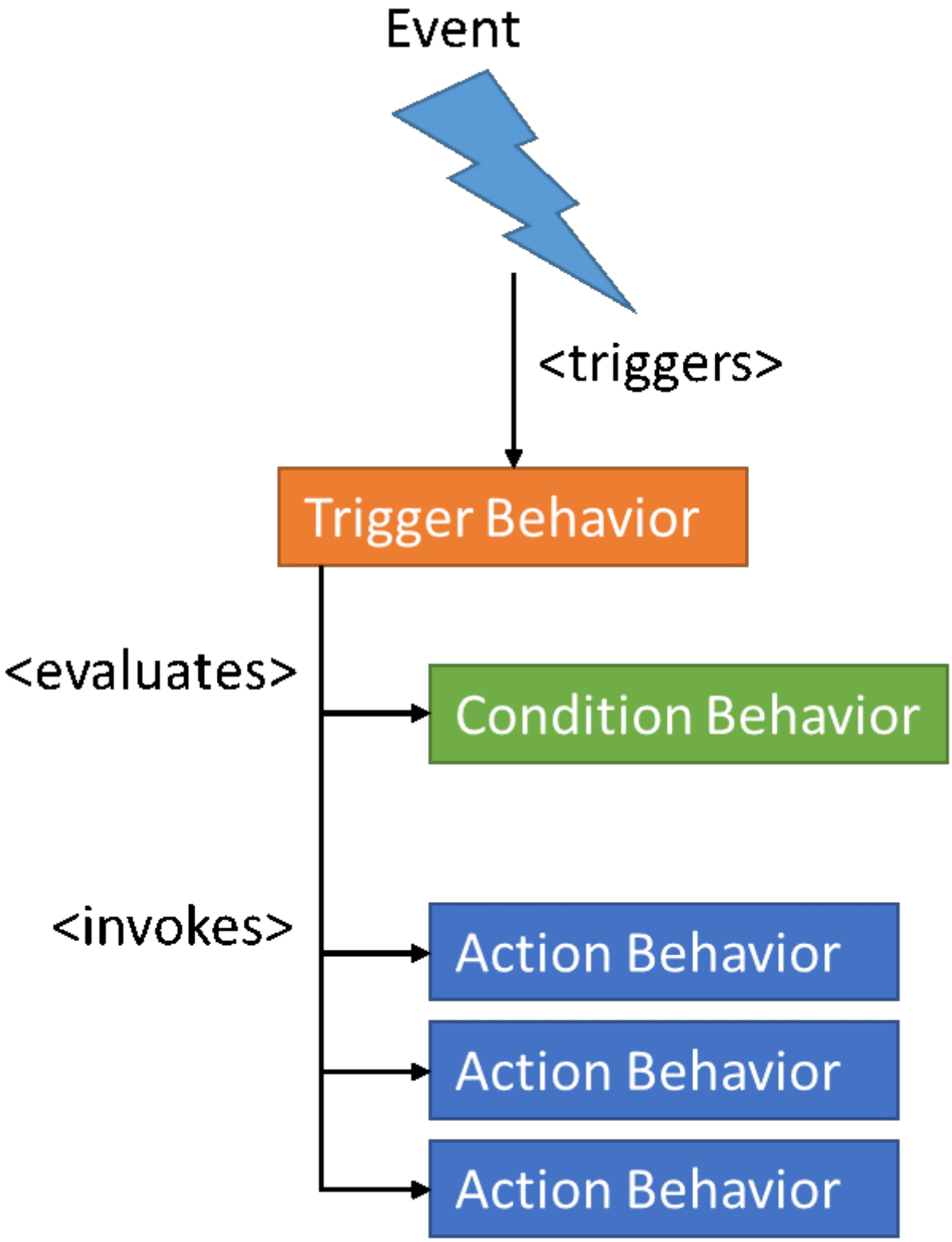
To differentiate the functionality of behaviors in the context of controls, there are two categories available: event handling and value processing.

Event Handling with Behaviors

The Behaviors for Event-Handling work like the following pattern:

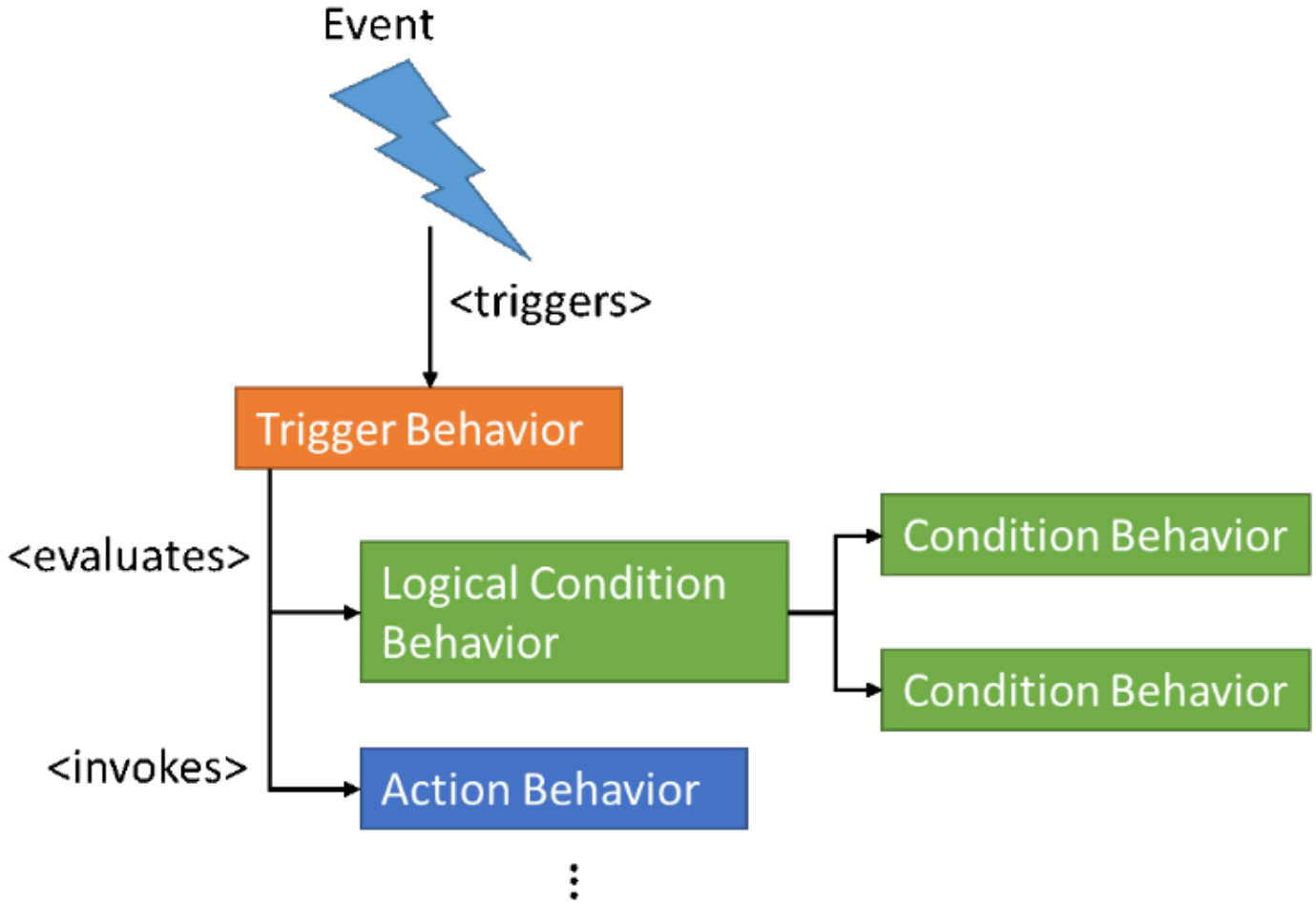
```
Upon <event> When <condition> Do <action>;
```

An input event is processed by a Trigger Behavior, which evaluates a condition (Condition Behavior). If the condition evaluates to true, then it invokes one or more Action Behaviors. The Trigger Behavior reacts on any input event. A filter on event type is achieved by specialized Condition Behaviors for these events. Event parameters are evaluated against Condition Behavior property values.



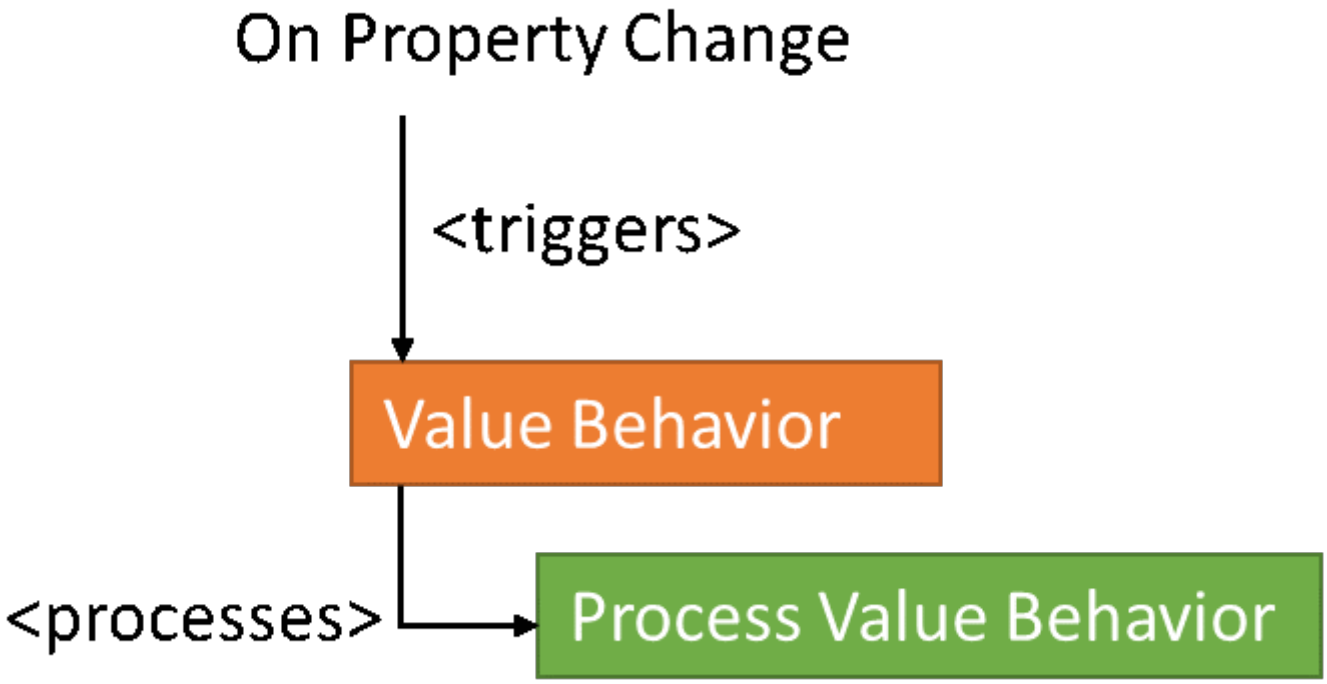
Specialized Logical Condition Behaviors enable complex conditions by combining Condition Behaviors.

Additionally there is a ConditionalActionBehavior which can be attached as an action. This can be used as a trigger with further conditions for other actions.



Value Processing with Behaviors

The second category is value processing. The Value Behavior provides a Variant data type which is bindable. A value change results in a processing of that value.

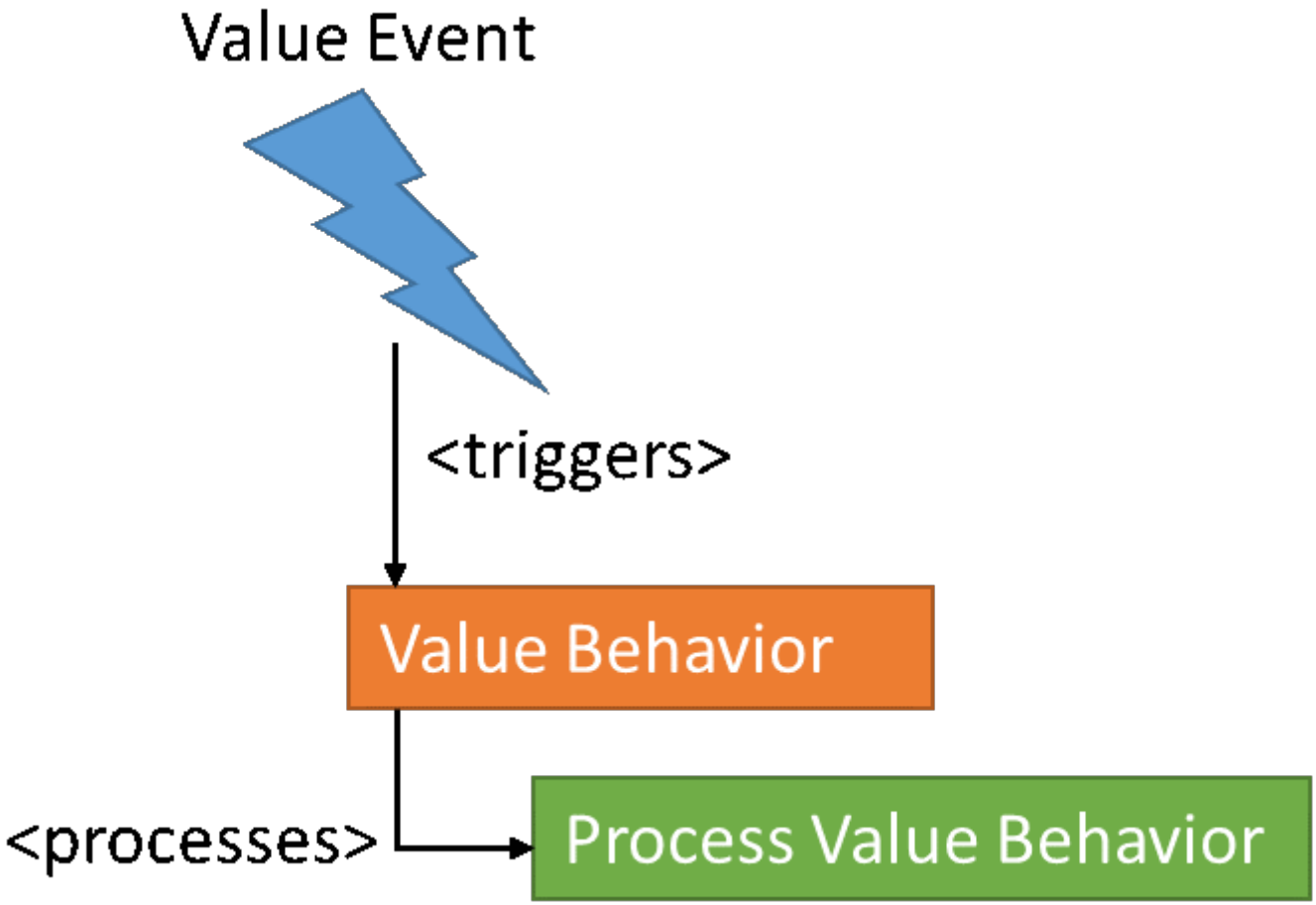


The value processing is realized in a data flow chain (including mapping to a limited value range and parallel data flow processing). As a result of this chain properties like position, scale ... will be affected.

To trigger an ActionBehavior corresponding to a value the "Action on Value"-Behavior can be used. To check the value a "Value Condition" can be attached as a ConditionBehavior.

Bridges between Event Handling and Value Processing

An option is provided to bridge the Event Handling and Value Processing flows. The first bridge is from an event to Value Processing. A dedicated Value Event carrying the value is sent to the Value Behavior. The Value Behavior receives the Value Event and proceeds the Value Processing.



The second bridge is from a value change to Event Handling. A specialized Process Value Behavior – Send Value Process Value Behavior - dispatches directly a Process Value Event. Optionally a condition can be applied (Condition Behavior) to determine whether the Value Event shall be sent. The Process Value Event is handled by a Trigger Behavior.

Behavior Definition

Behavior Definition

While SceneComposer can be used to compose the static structure and properties of a scene, with Behaviors it is possible to add dynamics to a static scene structure.

- A Behavior is defined by behavior attributes: Name, Type, Properties
- Behavior attributes are expressed in the Behavior Meta Information
- Behaviors are assembled together in a widget set
- Behaviors and the widget set can be assembled in several Behavior libraries (lib), to be linked to the application

Behavior Generation

If *cgi_studio_cit* is within the package, the base classes for Behaviors can be generated. Therefore an .xhcdl-file has to be written. The base class can include properties that can be defined in the .xhcdl-file too. The xhcdl-files can be found for each behavior in the same folder where the source-code lies. The following example shows an .xhcdl-file for a behavior.

```
<?xml version="1.0" encoding="utf-8"?>
<definition
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../../../cgi_studio_cit/schema/HmiContract.xsd">
    <include href="Behaviors/Behavior.xhcdl" addIncludeToCode="false" />
    <generator location="Behaviors/Transition/generated">
        <namespace namespace="Candera" />
        <header>
            <include file="Behaviors/ActionBehavior/ActionBehavior.h" />
            <include file="CanderaWidget/WidgetBase/StringDataTypes.h" />
        </header>
    </generator>
    <widgets setPolicy="onChange" notifierPolicy="onUpdate" viewInvalidationPolicy="wakeup">
        <group baseClassRef="ActionBehavior">
            <widget name="SimpleTransitionActionBehavior" readableName="" uid="{889EF594-E702-48
                <property name="ActivatedSceneAssetID" description="The scene to activate." type
                <property name="IsActivatedScene3D" description="If the activated scene is 3D or
                <property name="DeactivatedSceneAssetID" description="The scene to deactivate."
                <property name="IsDeactivatedScene3D" description="If the deactivated scene is 3
                <property name="Variant" description="The transition variant." type="FeatStd::St
            </widget>
        </group>
    </widgets>
</definition>
```

Some important tags and attributes:

- Generator location: Defines in which folder the base class should be generated.
- Group baseClassRef: The base class of the generated base class.
- Widget name: The name of the generated base class. The suffix "Base" will be added to the class-name.
- Property: The properties of the Behavior. For the String-type the include file *CanderaWidget/WidgetBase/StringDataTypes.h* has to be added for instance. If the property can't be generated it can still be added in the source code.

After the .xhcdl-file is written the behavior can be generated by calling *GenerateBehaviors.bat* which can be found in *cgi_studio_controls/src/Behaviors*. The files are a base class with a header- and cpp-file. It is meant to derive from that class. All the properties and its getter- and setter-methods are already in the class.

Behavior Meta Information

Behavior Name and Type

To publish a behavior to SceneComposer, it must register for the widget with its name and type:

CdaBehaviorDef	Behavior class name and its base class
CdaDescription	Behavior description for the SceneComposer tooltip
CdaReadableName	Behavior name

> Note that the example above explains how to publish a behavior for a 3D-scene. For 2D-scenes the macro *CdaBehavior2DDef* must be used instead of *CdaBehaviorDef*.

Behavior Properties

For linking a behavior with scene content in SceneComposer, behavior properties are used. Usually the properties should be defined in the .xhcdl-file of a behavior. Anyway, the properties can be defined directly in the behavior-class by using the appropriate macros in the header-file. The behavior implements a property by providing a getter and a setter method.

CdaProperty	Specifies [property_name], [property_type], [property_getter], [property_setter]
CdaDescription	Behavior property description for the SceneComposer tooltip
CdaCategory	Defines a category for the property. SceneComposer displays all properties of one category in a related Category section in the property grid.

>

> The Behavior property must be initialized correctly, e.g. in the widget constructor!
> Take care that the getter and setter methods are safe (prevent null pointer accesses)!

Usage of the introduced macro CGI_BEHAVIOR_FORWARD_AssetLoaderDataTypeDef

This macro is used to be able to set a certain behavior as property of another behavior.

Therefore following code has to be added to the xhcdl file to generate the macro in the base class of the behavior:

```
<postNamespace text="CGI_BEHAVIOR_FORWARD_AssetLoaderDataTypeDef(::CgiStudioControl::ValueBehavi
```

This is an example code to be able to use a ValueBehavior as property. The generated code in the base class of your behavior will look like this:

```
CGI_BEHAVIOR_FORWARD_AssetLoaderDataTypeDef(::CgiStudioControl::ValueBehavior)
```

Behavior Building Blocks - Behavior Blocks

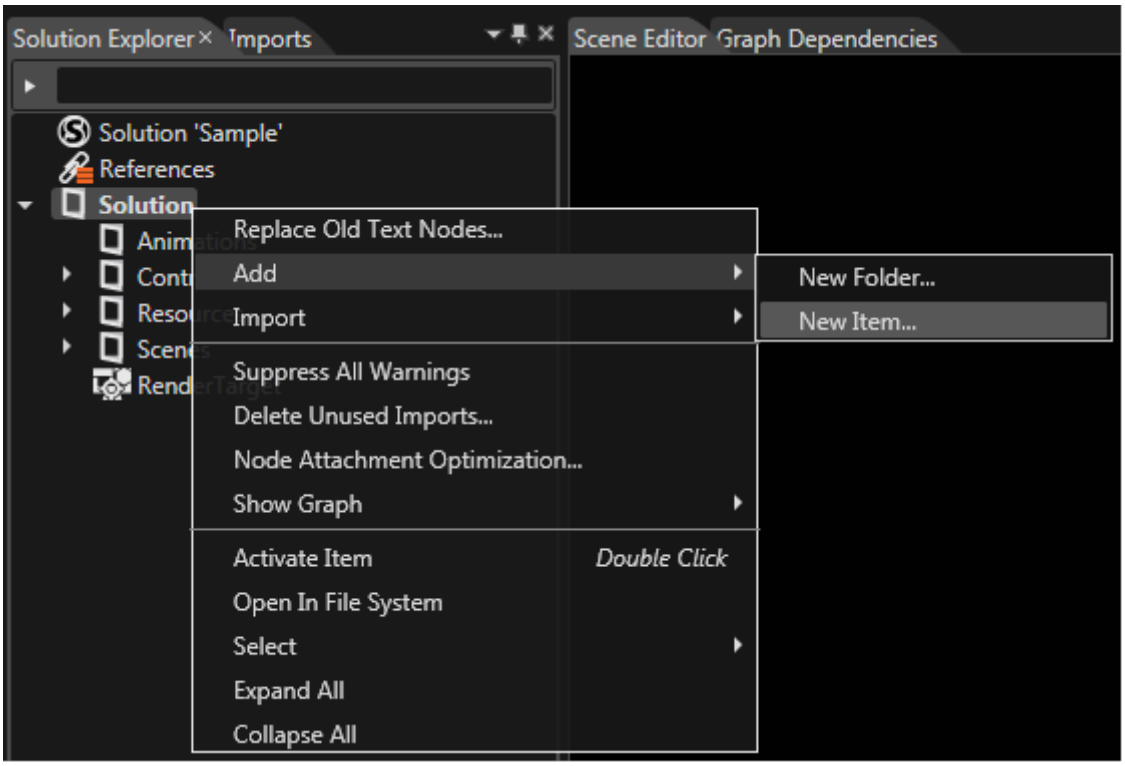
Description

This chapter describes how to work with behavior building blocks - in short: behavior blocks. Behaviors can be nested and this way combined to a reusable behavior building block. These behavior blocks offer the possibility to configure, which of the properties of this behavior block will be published and are therefore editable at the time the behavior block is instantiated and used in the SceneComposer scene.

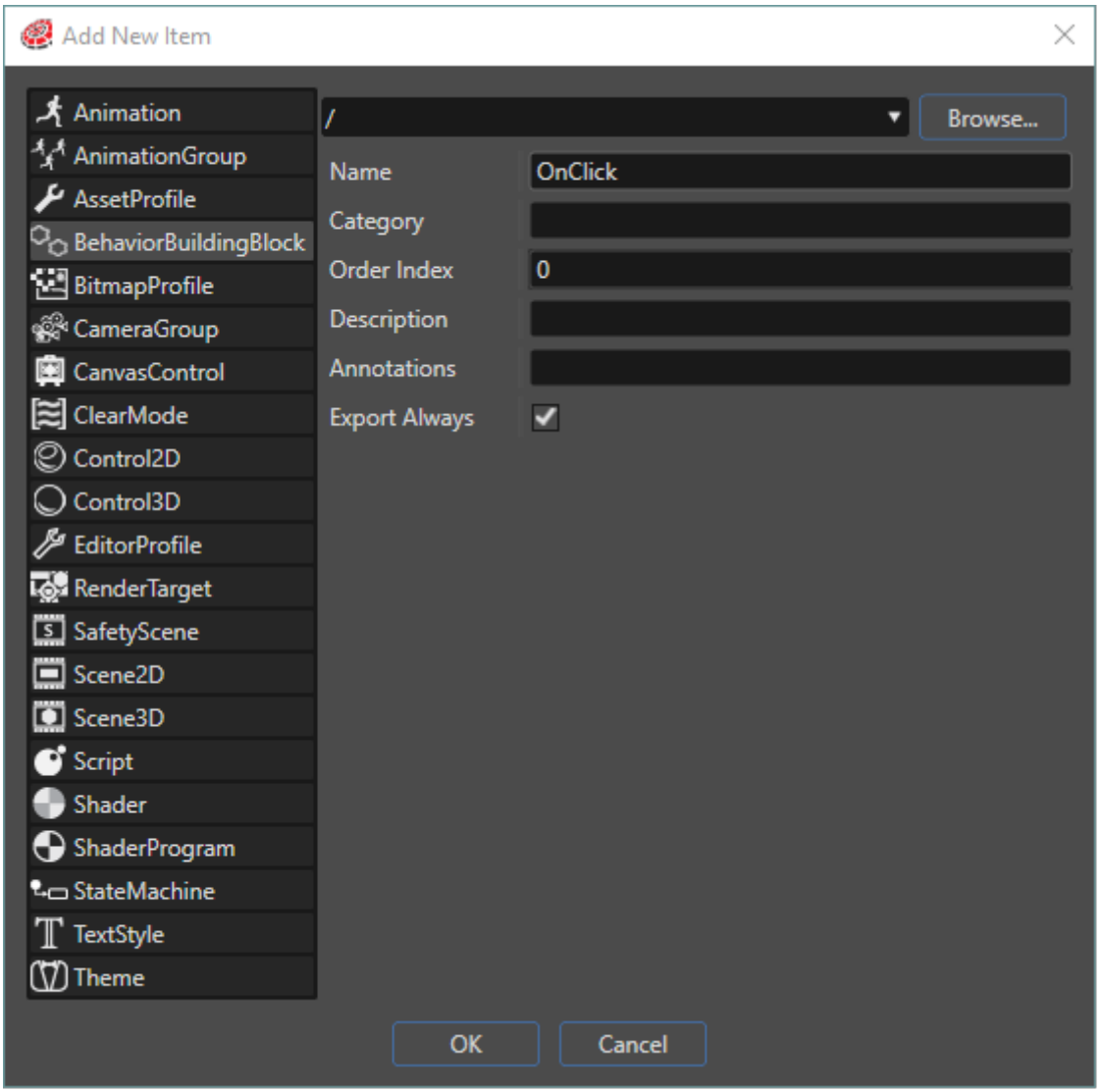
In this tutorial a 2D scene is created with a simple SolidColorNode and an OnClick Behavior Building Block. When single-clicking on the SolidColorNode, its color will be changed to red, when double-clicking on it, its color will be changed to green.
The OnClick Behavior Building Block is built out of an *EventHandler Control Behavior* and a *CheckClick Condition Behavior*. The two instances of the Behavior Building Blocks are configured with an additional *SetColor Action Behavior* to change the color.

Create a Behavior Building Block

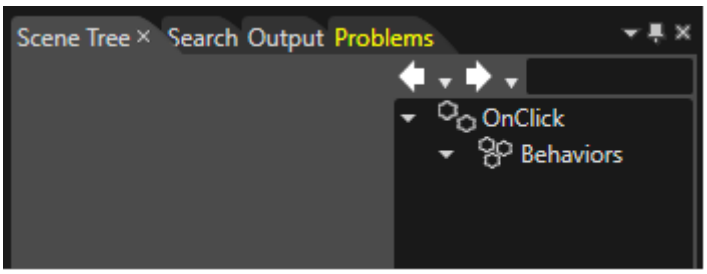
To create a behavior building block, right click on a folder in the Solution Explorer (e.g. Solution) to open the context menu and select "Add" - "New Item...".



In the opening "Add Item Dialog" select "BehaviorBlock" on the left. You can then define a location in your solution using the "Browse" button, a name (e.g. "OnClick") and a category for your behavior building block. You can also specify an order index and if you want the behavior building block to be always exported.



After Clicking "OK", the new Behavior Building Block called "OnClick" is created and visible in the Solution Explorer. Double-click on the newly created Behavior Building Block to edit it in the Scene Tree View.

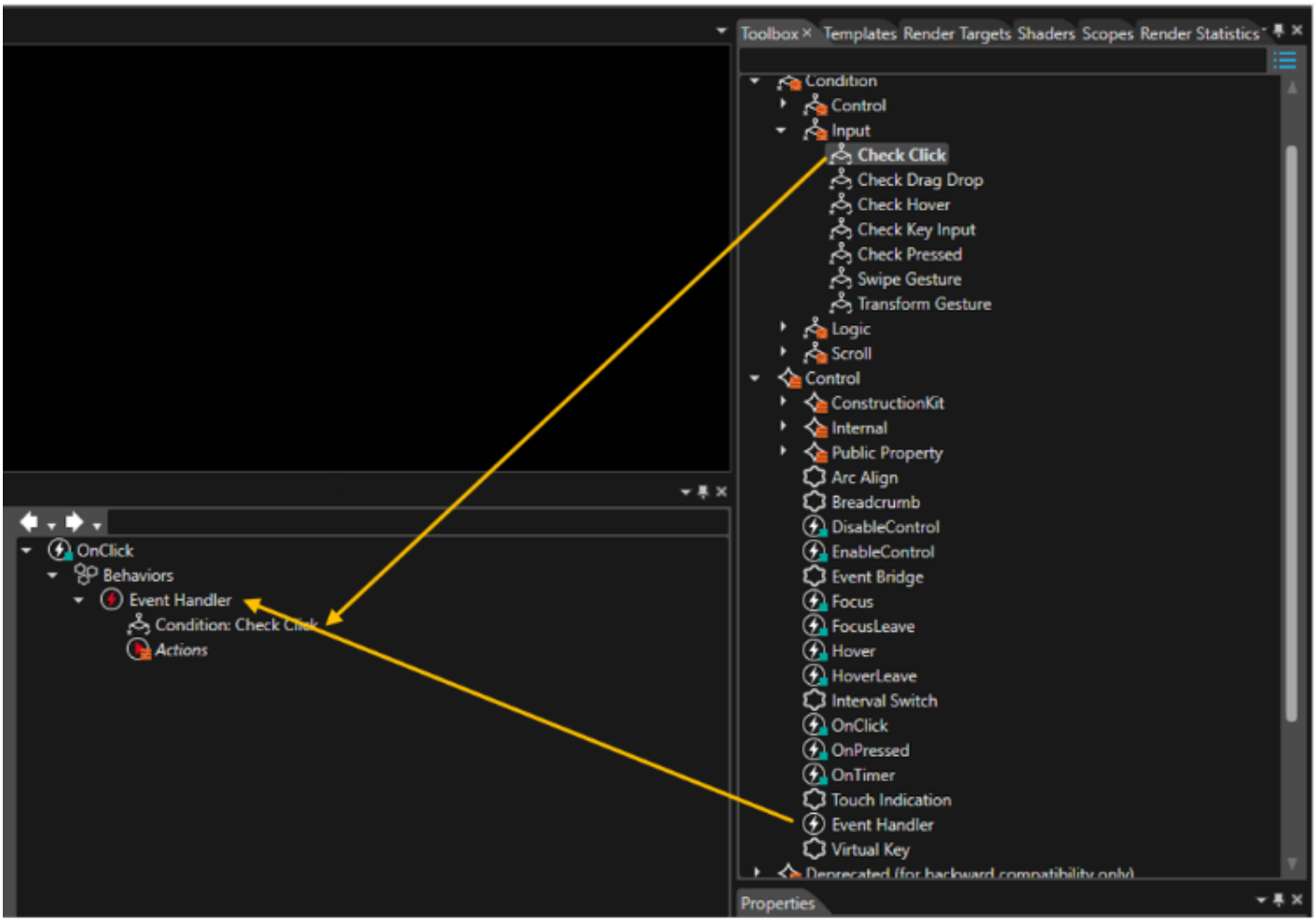


Construct a Behavior Building Block

To construct the Behavior Building Block you can drag Behaviors from the Toolbox Tab of the Libraries View and drop them onto the Behavior Building Block in the Scene Tree View.

For this tutorial we first add an EventHandler Control Behavior to the Building Block that requires a condition. We add a CheckClick Conditon Input Behavior to the EventHandler Behavior by dragging and dropping it onto the EventHandler Behavior.
Our Building Block now consists of the following Behaviors:

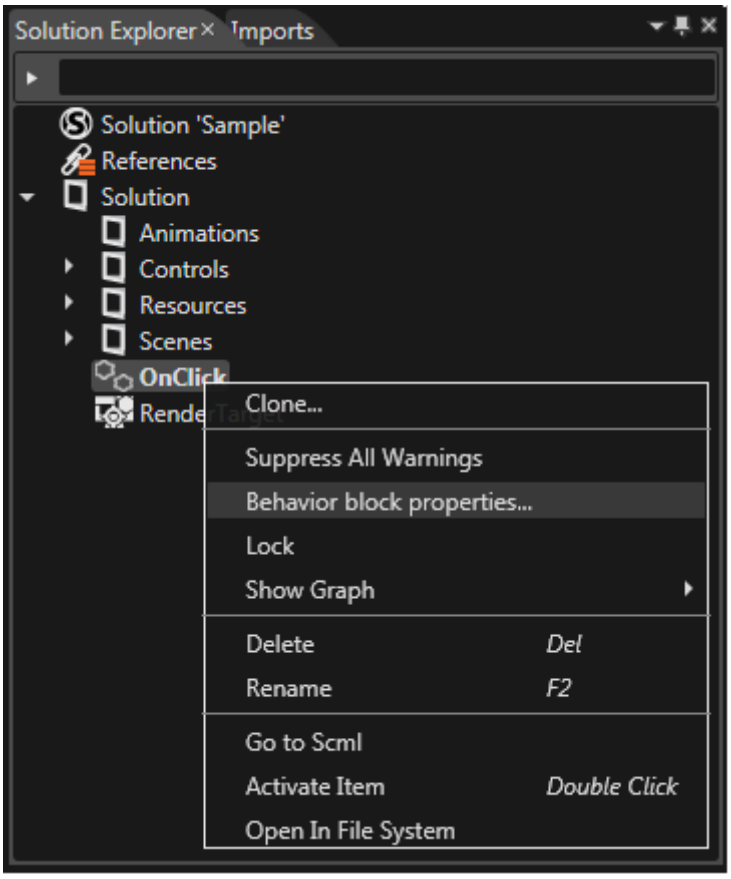
- *EventHandler Control Behavior* passes a value to a Condition Behavior and if the response is positive, invokes an Action Behavior
- *CheckClick Conditon Input Behavior* evaluates the click event condition and is to be used with a Trigger Behavior to trigger a certain action.



> You can also build up nested behaviors while working on your scene in the Scene Tree View. Whenever you decide that you want to reuse your built-up behavior construct, you can press the right mouse button on the parent behavior. The context menu of the behavior offers you the possibility to "Save as BehaviorBlock"

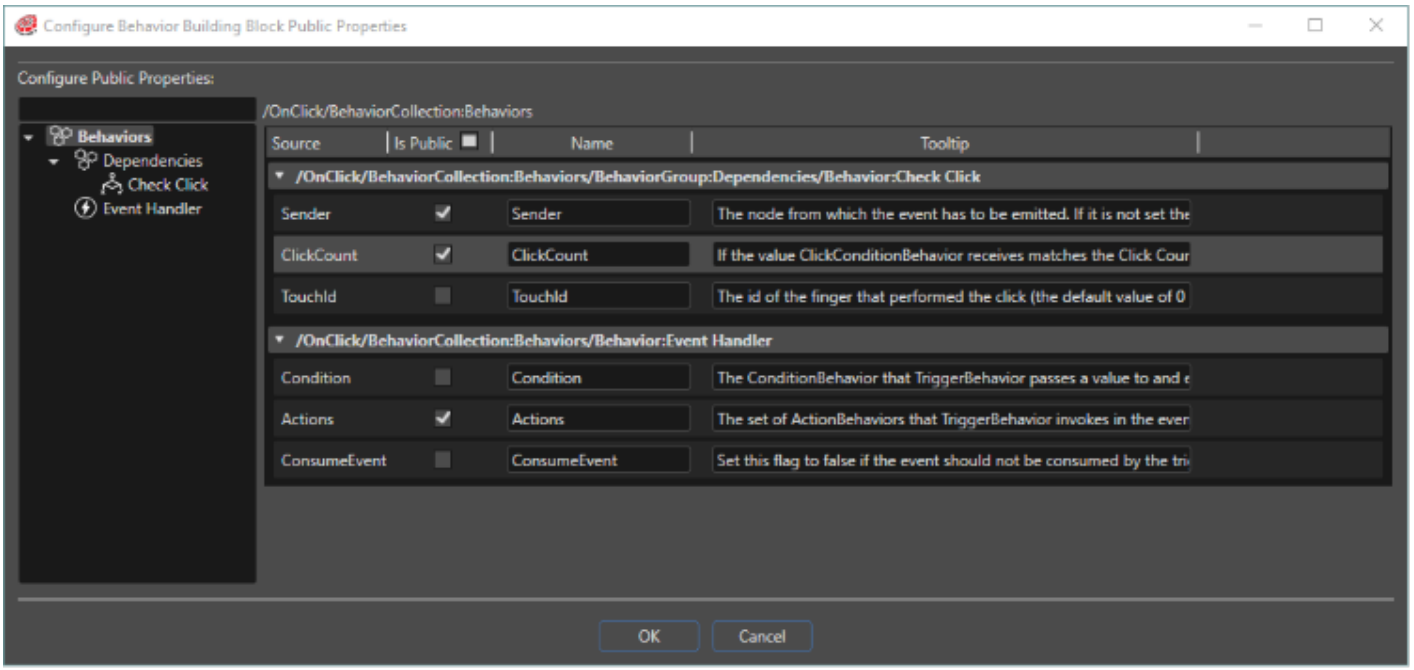
Configure Properties of the Behavior Building Block

The Properties of the Behavior Building Blocks can be edited in the Configure Properties Dialog that is opened by right-clicking on the Behavior Block in the Solution Explorer and selecting "Behavior block properties...".



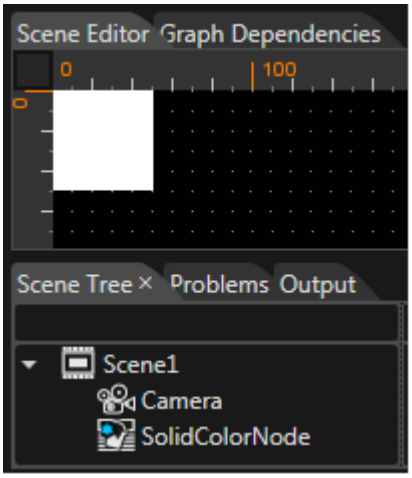
In the "Configure Properties" dialog you can define which properties you want to publish, that means, which properties you want to be able to edit when creating an instance of this Behavior Building Block.
For our tutorial we want to publish the following properties:

- the *ClickCount* of the CheckClick Behavior
- the *Emitter* of the CheckClick Behavior
- the *Actions* of the EventHandler

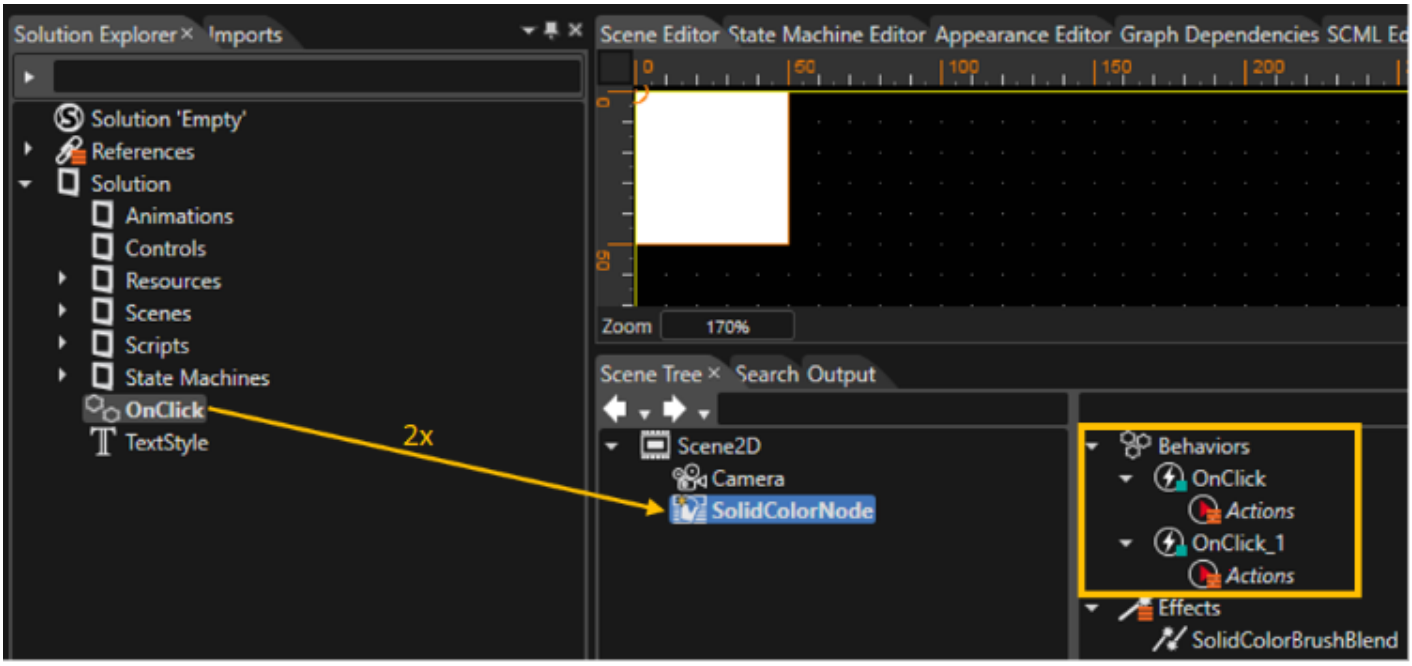


Instantiate the created Behavior Building Blocks

Now that we have created the "OnClick" Behavior Building Block, we can use it in our scene. For this, we create a simple 2D scene with one single white SolidColorNode.



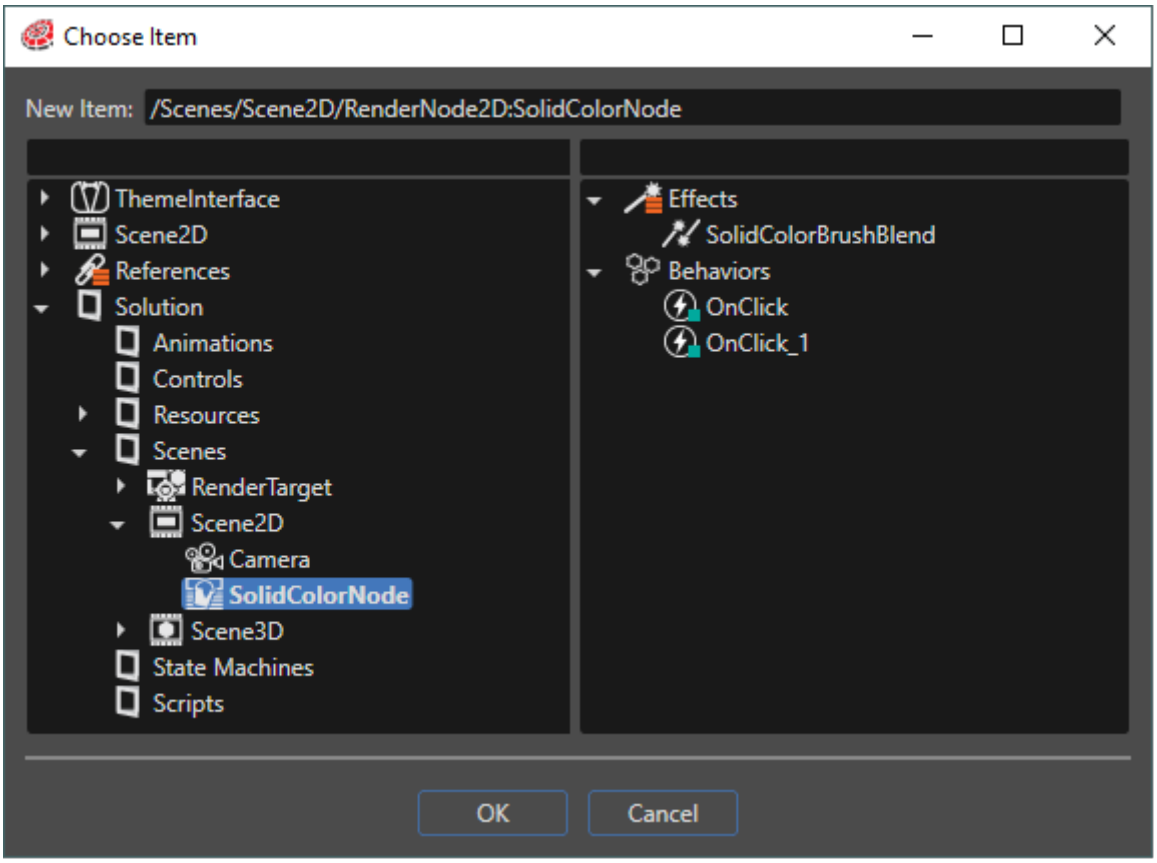
We want to have 2 OnClick Behaviors: one for a single-click and one for a double-click. To instantiate the "OnClick" Behavior Building Block, simply drag and drop it from the Solution Explorer onto the SolidColorNode in the Scene Tree View twice. You should now see the created instances of the Behavior Blocks on the right side in the Scene Tree View.



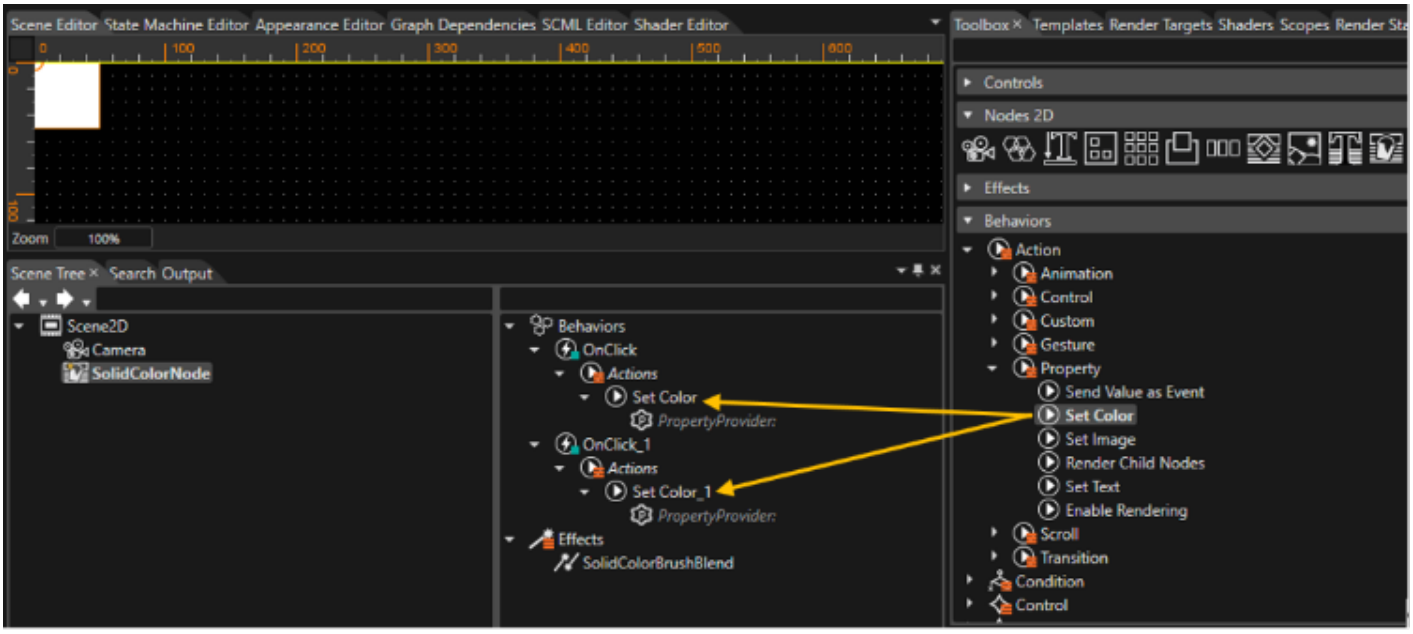
Configure Behavior Building Block Instances

These 2 instances can now be configured. One instance should be configured to change the color of the SolidColorNode to red when someone single-clicks on it and the second one should be configured to change the color of the SolidColorNode to green when someone double-clicks on it. You can configure the Click Count for one Behavior to 1 (single-click) and the Click Count for the other Behavior to 2 (double-click).

We also have to configure the Emitter, that means the node from which the click event has to be emitted. If no emitter is configured, all click events are handled. Therefore we configure the single SolidColorNode fom our 2D Scene as Emitter by clicking on the icon to the right of the Emitter property input field. In the opened "Choose Item" dialog the SolidColorNode can be selected.

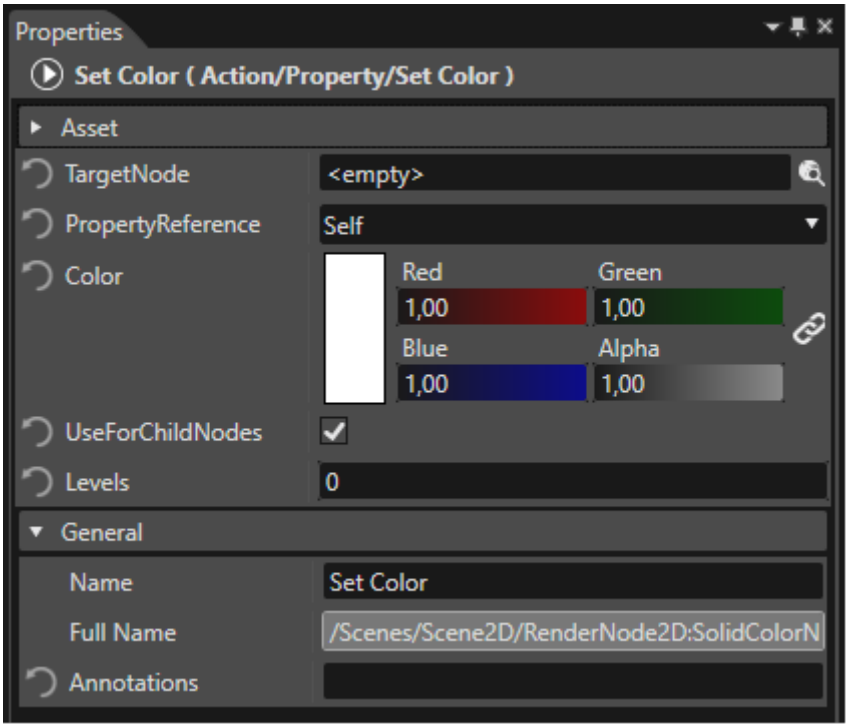


To change the color of the SolidColorNode, an action is required. This means, we need another Behavior: The action Behavior "SetColor". It can be applied by dragging it from the Behavior ToolBox and dropping it onto our "OnClick" Behavior Building Blocks in the Scene Tree View.



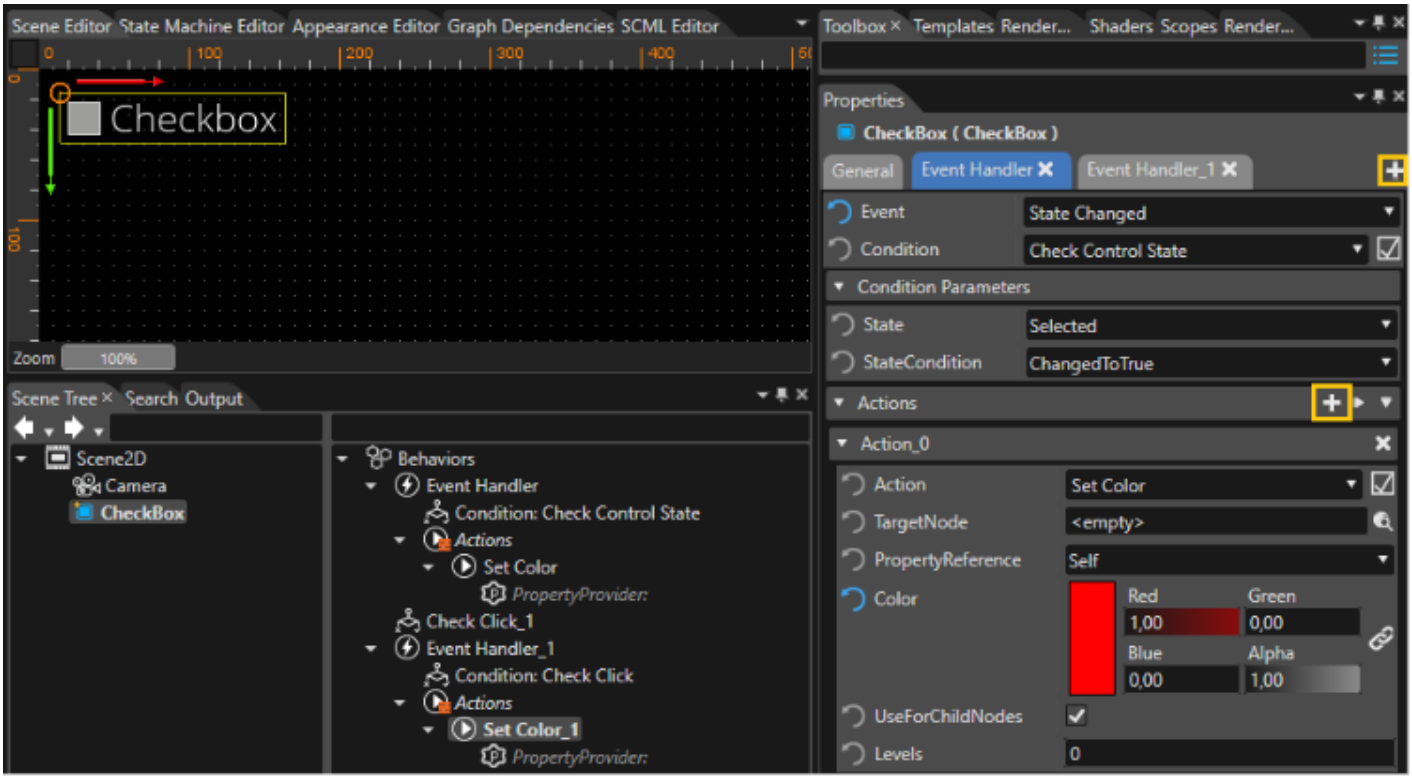
The last step of this tutorial is to configure these SetColor Action Behaviors: one SetColor Action should set the color of the SolidColorNode to red (see picture below) and the other one should set it

to green.


Properties
Set Color (Action/Property/Set Color)
Asset
TargetNode
<empty>
PropertyReference
Self
Color
Red
1,00
Green
1,00
Blue
1,00
Alpha
1,00
UseForChildNodes
Levels
0
General
Name
Set Color
Full Name
/Scenes/Scene2D/RenderNode2D:SolidColorN
Annotations

EventHandler Tab

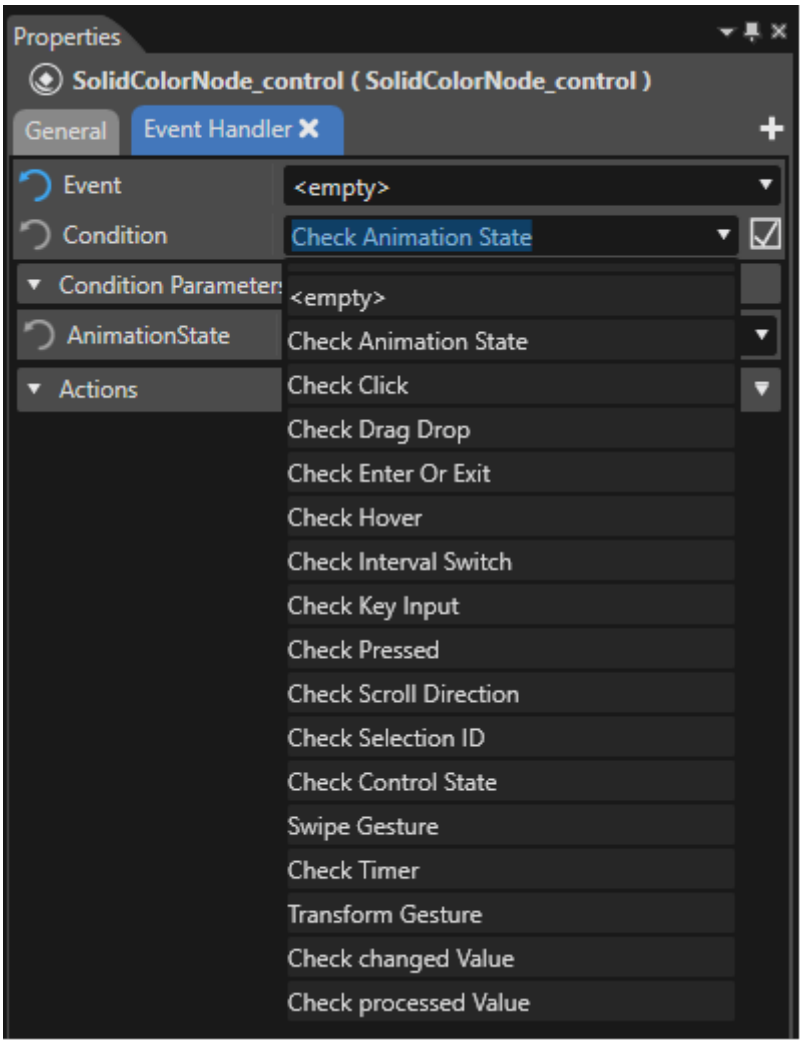
Every control needs an EventHandler in order to handle user events and take appropriate actions. Events, Conditions and Actions for a node can conveniently be edited in SceneComposer in the EventHandler Tab, which is located in the properties view. Using the EventHandler Tab the user can choose the event he wants to react on from a drop-down menu. Furthermore, conditions and actions can also be configured with drop-down menus in this tab. Once an Event is selected, only conditions that are suitable for the selected event are offered in the drop-down menu. For a list of all selectable Events see chapter Control Behaviors Events .



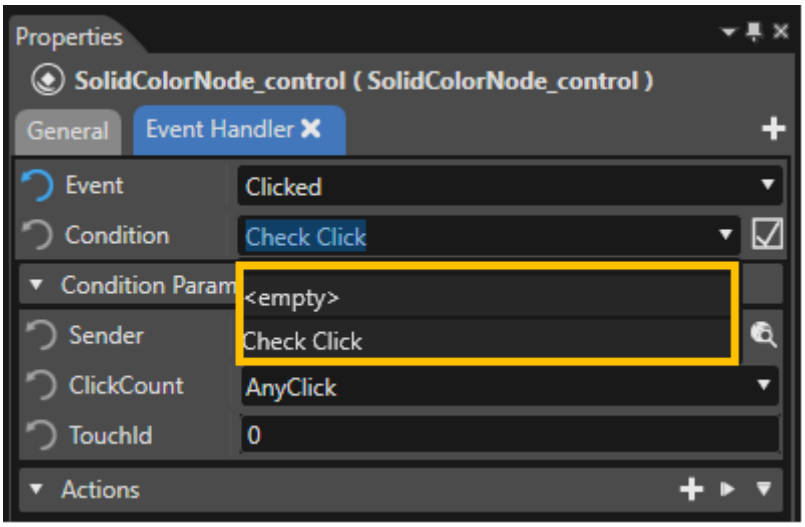
Additional EventHandlers can be added by clicking the '+' sign next to the EventHandler Tabs, more Actions can be added by clicking the '+' sign next to the Actions (marked in orange in the image above).

Filtering Conditions according to the selected Event

Before selecting an Event, the Conditions drop-down menu contains quite a lot of entries:



After selecting an Event, the Conditions drop-down menu only contains Conditions that are applicable for the selected Event:



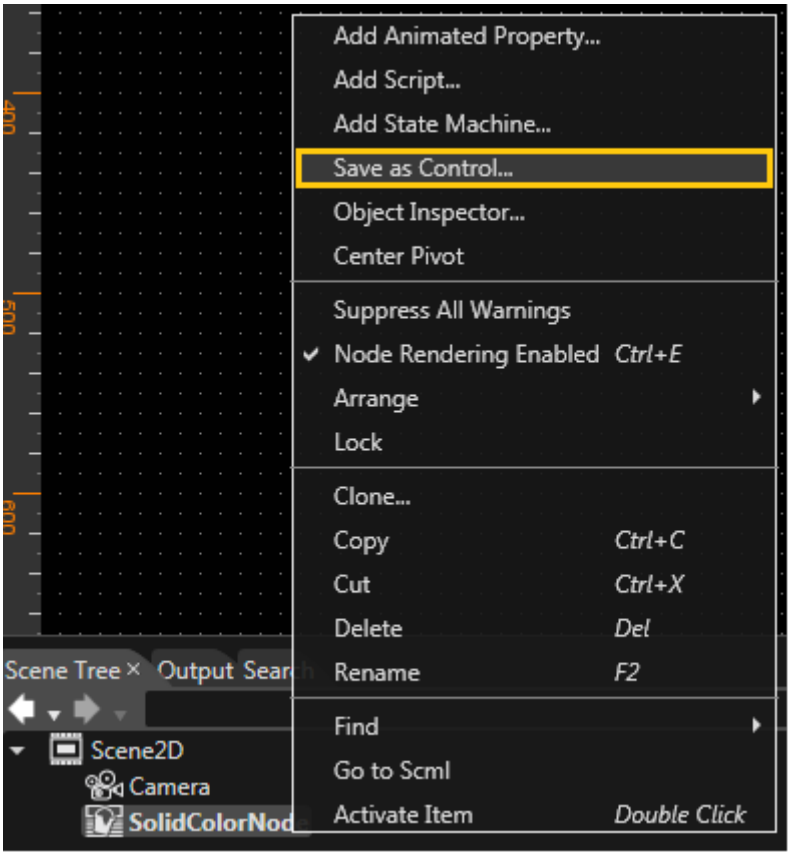
When implementing a ConditionBehavior, filtering for a specific Event can easily be configured using the CdaProperty "CdaEvents" as seen in the code example below, where the StateConditionBehavior is the configured to be visible for the StateEvent:

```
private:
    CdaBehaviorMixedDef(StateConditionBehavior, StateConditionBehaviorBase)
```

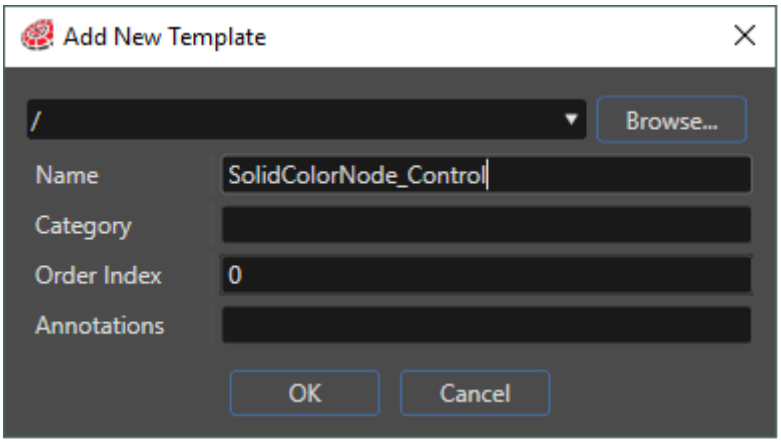
```
        CdaDescription("Expects a State Event (generated by Control State Behavior) and returns
        CdaReadableName("Check Control State")
        CdaCategory("Condition/Control")
        CdaProperties()
        CdaPropertiesEnd()
        CdaEvents()
            CdaEvent(CdaInputEvent, StateEvent, "")
        CdaEventsEnd()
    CdaBehaviorDefEnd()
```

Adding an EventHandler to a Control

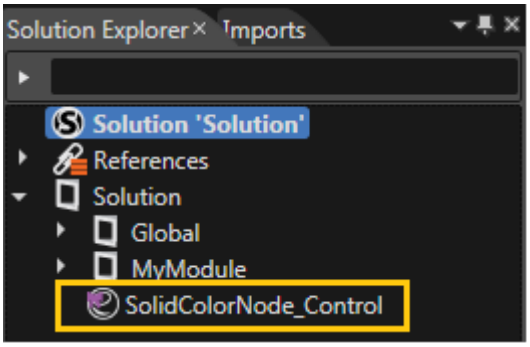
A Control can be created from a simple RenderNode using "Save as Control..." from the RenderNode's context menu in the Scene Tree View:



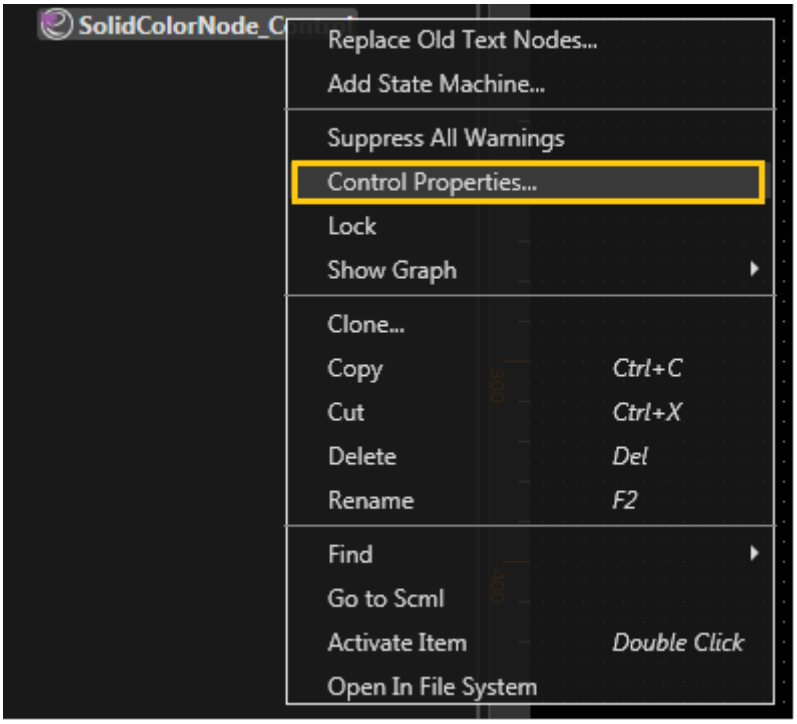
For creating a Control, a name can be specified:



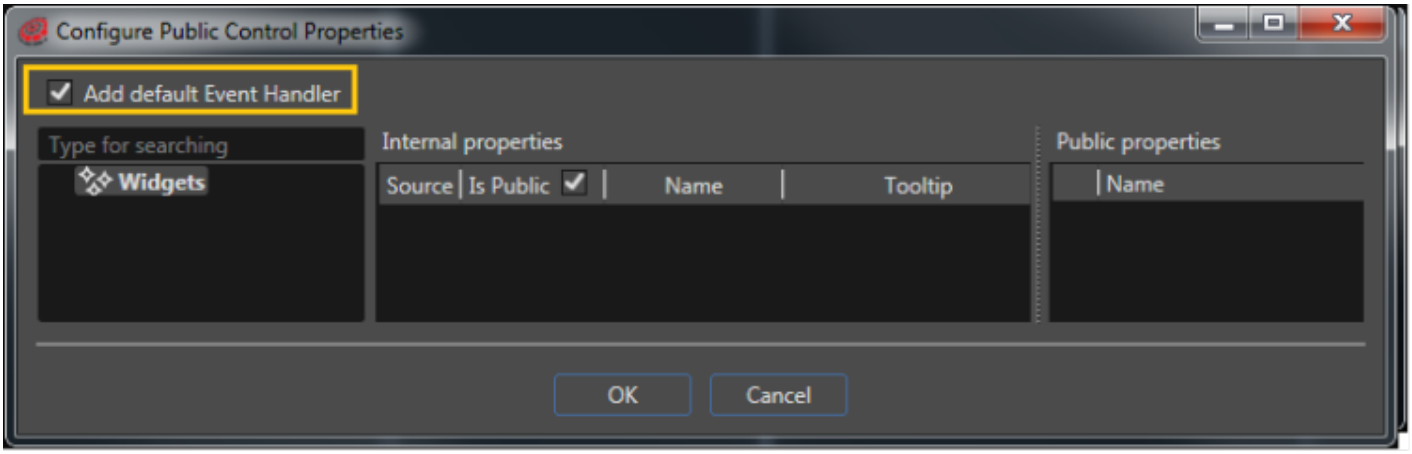
After clicking "OK", the new Control is visible in the Solution Explorer



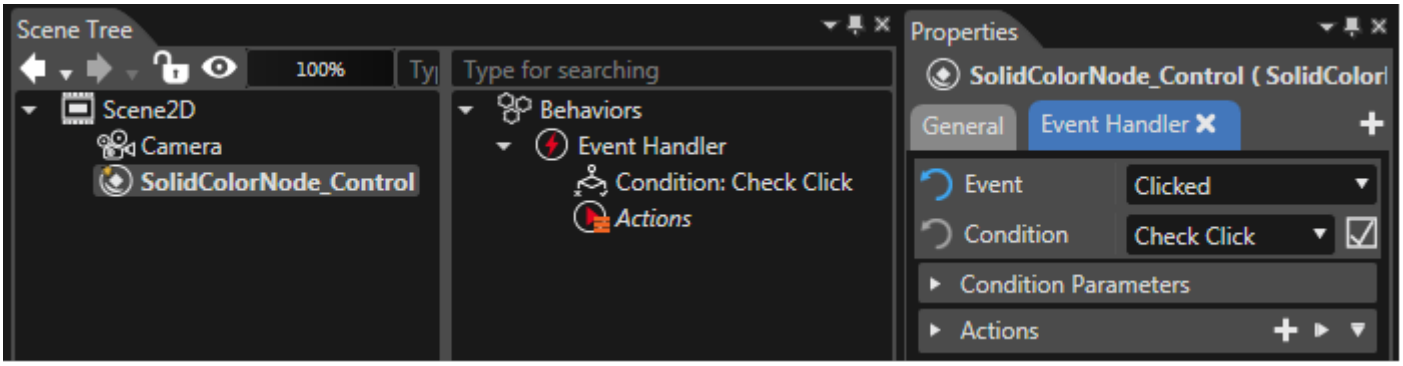
When right-clicking on the context menu of the new Control, the Control's properties can be edited choosing "Control Properties...":



The "Configure Properties" window offers a checkbox to choose if an EventHandler shall be added to every instance of this kind of Control that is created.



Now, whenever such a control is dragged-and-dropped to the SceneComposer scene, an EventHandler is added and the EventHandler Tab is visible in the Properties view.



Implementation

This page gives an overview of how Behaviors can be implemented.

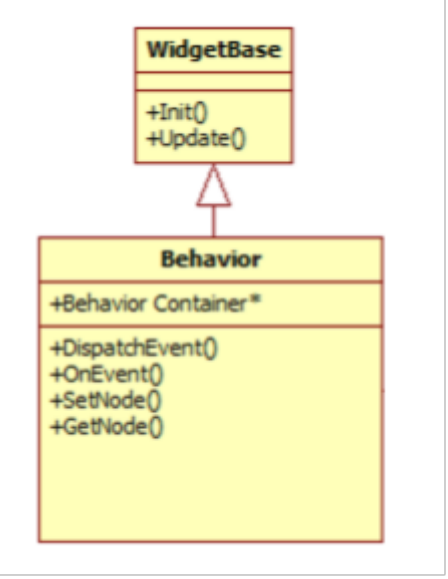
How to implement a Behavior

There is very little difference between implementing a Widget and implementing a Behavior. Though there are some difference which will be explained below.

Base Class

The major difference is that Behaviors need to be derived from the Candera::Behavior base class instead of the Candera::WidgetBase class. Incidentally the Candera::Behavior base class is internally derived from Candera::WidgetBase further demonstrating the similarities between both.

Methods



From the Candera::WidgetBase the already known Init() and Update() methods are inherited.

- **Init:** Called once when the Widget/Behavior is created, after all Widget/Behavior properties have been set. This is used to initialize the Widget/Behavior.
- **Update:** Called once per iteration (frame). Here normally the Widget/Behavior performs its actual functionality.

Newly added are the methods from Candera::Behavior which are related to event handling and the attached node:

- **DispatchEvent:** Starts dispatching an event with a specified strategy
- **OnEvent:** Called when an event is received by a Behavior
- **SetNode/GetNode:** Allows access to the node this behavior is attached to. Note that this uses a new data type called Candera::AbstractNodePointer which abstracts between 2D and 3D nodes. This allows creating behaviors that work und 2D and 3D without duplicating code.

META INFO

The Meta-Info is essential to provide generic access to all properties of the widget/behavior as well as special information for Scene-Composer about the available properties, their types and functionality. The available macros to define the Meta-Info are very similar to those used by widgets.

The first set of Macros defines a Behavior either for 3D, 2D or both (mixed):

```
...
CdaBehaviorDef(<<ClassName>>, <<BaseClassName>>)
    [Information][Properties][Events]
CdaBehaviorDefEnd()
...
```

```
...
CdaBehavior2DDef(<<ClassName>>, <<BaseClassName>>)
    [Information][Properties][Events]
CdaBehaviorDefEnd()
...
```

```
...
CdaBehaviorMixedDef(<<ClassName>>, <<BaseClassName>>)
    [Information][Properties][Events]
CdaBehaviorDefEnd()
...
```

All other macros are the same as where used by Widgets. This includes the macros to add additional information:

```
...
CdaReadableName("Name")
CdaCategory("Category")
CdaDescription("Description")
...
```

As well as the macros to define properties:

```
...
CdaProperties()
...     [Property List]
CdaPropertiesEnd()
...
```

```
...
CdaProperty(<<Name>>, <<Type>>, <<Setter>>, <<Getter>>)
CdaDescription("Column used by this Item inside the Area")
CdaCategory("Area")
CdaPropertyEnd()
...
```

The new macros for events will be described in the Event Handling chapter.

For further information about the Meta Info please also see chapter Behavior Definition.

RUNTIME TYPE INFORMATION

CGI-Studio uses its own RTTI system for major parts instead of relying on the RTTI of the compiler. For this a set of macros is added to each class that uses this system. In case of Behaviors only a single macro is required to be added to the code:

```
...
CGI_BEHAVIOR_RTTI_DEFINITION(<<ClassName>>)
...
```

Event Handling

Events are one of the major new features introduced via Behaviors. This chapter shows how they can be utilized from within the Behavior code.

Meta Info

The first requirement is to add all events that should are either received or sent to the Meta-Info via the following macros:

```
...
CdaEvents()
   CdaEvent(CdaOutputEvent, <<EventType>>, "Description")
   CdaEvent(CdaInputEvent, <<EventType>>, "Description")
CdaEventsEnd()
```

```
...
```

Sending Events

An Event can be sent by multiple different approaches.

The first is achieved by calling the DispatchEvent() method of the Behavior or any of the other available alternatives. It requires providing the dispatch strategy and event itself. When using this method the dispatching will start from the current node of the Behavior.

```
...
    CgiStudioControl::Event event;
    EventDispatchResult dispatchResult;
    EventDispatchStrategy dispatchStrategy;

    DispatchEvent(dispatchStrategy, event, dispatchResult);
...
```

> Note that the Event and EventDispatchStrategy need to be replaced with the concrete implementations.

The second options is to directly call the DispatchEvent() method of the Event-Dispatch-Strategy. This allows to define the node from which the dispatching should start:

```
...
    Candera::EventDispatchResult eventDispatchResult;
    Candera::BroadcastEventDispatchStrategy dispatchStrategy;
    CgiStudioControl::ChangeValueEvent valueChangedEvent(FeatStd::Variant
(listDataEvent->GetData()),
                                                         CgiStudioControl::ChangeValue::Absolute
    dispatchStrategy.DispatchEvent(GetNode(), valueChangedEvent, eventDispatchResult);
...
```

Receiving Events

Receiving of events is done via the OnEvent-Method. Here normally the incoming events are type-cast to check the Event-Type and processed if applicable. Further it is possible to stop further dispatching of the event via the Dispatch-Result. In case a custom Dispatch-Result is used (again type-casting can be used) then additional information can be returned to the sender.

```
...
void ListItemEventBehavior::OnEvent(const FeatStd::Event
& event, Candera::EventDispatchResult& dispatchResult)
{
    const ListDataEvent* listDataEvent = Candera::Dynamic_Cast<const ListDataEvent*>(&event);
```

```
    if (0 != listDataEvent) {
        if (m_index == listDataEvent->GetIndex()) {
            dispatchResult.StopDispatchingImmediately();

            // Do something with the data of the event
        }
    }
}
...
```

If derived from Behaviors that also do event processing also the Base-Class OnEvent method should be called.

```
...
Base::OnEvent(event, dispatchResult);
...
```

Custom Events

It is possible to define custom events by simply deriving from FeatStd::Event class. An Event can contain any number custom data.

```
...
class ListDataEvent : public FeatStd::Event
{
    public:
        FEATSTD_RTTI_DECLARATION();

        ListDataEvent(FeatStd::Int32 index, FeatStd::Int32 value) :
            m_index(index),
            m_value(value)
        {
        }

        FeatStd::Int32 GetIndex() const
        {
            return m_index;
        }

        FeatStd::Int32 GetData() const
        {
            return m_value;
        }

    private:
        FeatStd::Int32 m_index;
        FeatStd::Int32 m_value;
};
...
```

```
...
FEATSTD_RTTI_DEFINITION(ListDataEvent, FeatStd::Event)
...
```

FINDING BEHAVIORS

The second major advantage compared to widgets is that it is now possible to determine if any and which Behaviors are attached to a node. This can be done by iterating over all behaviors of a node via the code snipped shown below.

```
...
Candera::Behavior* behavior = Candera::Behavior::GetFirstBehavior(node);

while (0 != behavior) {
    // do something

    behavior = behavior->GetNextBehavior();
}
...
```

For a better understanding of how events work, please see chapter Predefined Behaviors.

Creating a custom Behavior

First to create a custom Behavior for most of them a Base Behavior has to be implemented. (If the Base Behavior is not needed the next few steps can be skipped)

To do this achieve that a <CUSTOMBEHAVIOR>Base.xhcdl file has to be created.
This file contains things like: includeName, behavior name, readableName, category, properties and the UUID.
These need to be filled out for the specifications of the Behavior.

Samples for this file can be found under: "...\cgi_studio_controls\src\Behaviors". Just select a BehaviorBase.xhcdl which has similarities with the custom behaviour copy it and modify it to the special needs.

If the file is finished add it to the "...\cgi_studio_controls\src\Behaviors\Behaviors.xhcdl" file in this way:

Behaviors.xhcdl

```
<include href="<BEHAVIORFOLDER>/<CUSTOMBEHAVIOR>Base.xhcdl" />
```

Next start the "...\cgi_studio_controls\src\Behaviors\GenerateBehaviors.bat". This file then generates the <CUSTOMBEHAVIOR>Base.h and <CUSTOMBEHAVIOR>Base.cpp files.

After this step is completed the actual Behavior logic has to be written. The <CUSTOMBEHAVIOR>.h/.cpp files will be made and the files of other already implemented Behaviors can serve as the Template for the custom Behavior.

Adding the Behavior to the entire CGI Studio:

If the Intention is to use the Behavior in every CGI Studio project it can be added to the whole Studio.

For this the <CUSTOMBEHAVIOR>Base.h/.cpp have to be added to "...\cgi_studio_controls\src\Behaviors\<BEHAVIORFOLDER>\generated\FileList.cmake" in this manner:

FileList.cmake

```
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.h"
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.cpp"
```

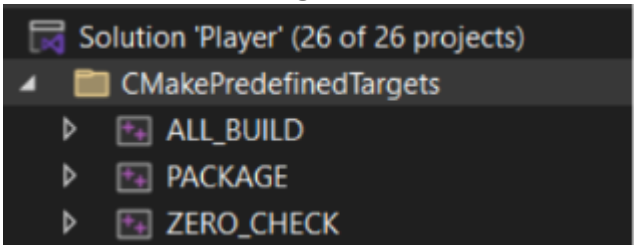
Next the <CUSTOMBEHAVIOR>.h/.cpp as well as <CUSTOMBEHAVIOR>Base.xhcdl have to be added to "...\cgi_studio_controls\src\Behaviors\<BEHAVIORFOLDER>\FileList.cmake" like this:

FileList.cmake

```
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.xhcdl"
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.h"
"${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.cpp"
```

After these two steps the Behavior can be build with CMake and then Visual Studio. (For a Guide how to do that have a look at the Wiki Page: Building-Player-and-Widgets)

After opening Visual Studio ALL_Build -> Build can be started which is found under CMakePredefinedTargets.



Now CGI Studio can be opened and the custom Behavior can be used.

Adding the Behavior to a specific Project:

To add a Behavior to a specific Project the "...\cmake\Apps\Courier\CourierSampleApp\CMakeLists.txt" of the Project has to be modified with this line. (CUSTOMBEHAVIOR_FOLDER has to be substituted with the actual folder name)

CMakeLists.txt

```
add_subdirectory("${CANDERA_ROOT_PATH}/<CUSTOMBEHAVIOR_FOLDER>"
"${CMAKE_BINARY_DIR}/<CUSTOMBEHAVIOR_FOLDER>")
```

> **Important:** This include has to be after all other includes, because some of them clear the Behavior list.

Next the <CUSTOMBEHAVIOR_FOLDER> has to be created in the Candera root Directory. This folder then needs to include CMakeLists.txt as well as a FileList.cmake.

CMakeLists.txt

```
# Creation of new dedicated behavior project
project(MyCustomBehaviors)
add_library(${PROJECT_NAME})

#------------------------------------------------------------------
# collecting all files
include("${CMAKE_CURRENT_LIST_DIR}/FileList.cmake")

# load the paths as sources into 'all_sources'
get_target_property(all_sources ${PROJECT_NAME} SOURCES)

# create a structured visualization inside the solution
source_group(TREE ${CMAKE_CURRENT_LIST_DIR} FILES ${all_sources})

#------------------------------------------------------------------
# this is the include path to the custom behaviors which shall be used.
target_include_directories(${PROJECT_NAME}
  PUBLIC
  ${CMAKE_CURRENT_LIST_DIR}
)

#------------------------------------------------------------------
# definition of which project must be linked
# if application parts like application lib is required, please link the project
accordingly.
target_link_libraries(${PROJECT_NAME}
  PUBLIC
  Candera
)

#------------------------------------------------------------------
# These two calls attach a Behavior project to the internal Behavior set.
# IMPORTANT: the 0 at the end tells the system that already existing behaviors should not be
cleared.
CgiBehaviorSet_AddTarget(${PROJECT_NAME} 0)
CgiCreateBehaviorPartSet("${PROJECT_NAME}" "BehaviorSet_${PROJECT_NAME}")
```

FileList.cmake

```
target_sources(${PROJECT_NAME}
  PRIVATE
  "${CMAKE_CURRENT_LIST_DIR}/FileList.cmake"
)
include("${CMAKE_CURRENT_LIST_DIR}/Behaviors/FileList.cmake")
```

After that we need to create another folder in our initial <CUSTOMBEHAVIOR_FOLDER>.

> **Important:** This folder has to be named "Behaviors".

Into this folder we paste our custom Behavior files. (<CUSTOMBEHAVIOR>Base.h/.cpp, <CUSTOMBEHAVIOR>.h/.cpp, <CUSTOMBEHAVIOR>Base.xhcdl)

Then we create another FileList.cmake where we add lines for all our custom Behavior files as seen below.

FileList.cmake

```
target_sources(${PROJECT_NAME}
  PRIVATE
  "${CMAKE_CURRENT_LIST_DIR}/FileList.cmake"

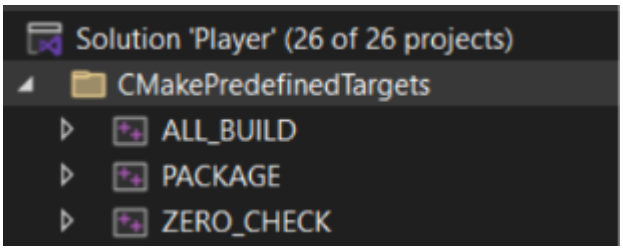
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.h"
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.cpp"

  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>Base.xhcdl"
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.h"
  "${CMAKE_CURRENT_LIST_DIR}/<CUSTOMBEHAVIOR>.cpp"

)
```

After these steps the Behavior can again be build with CMake and then Visual Studio. (For a Guide how to do that have a look at the Wiki Page: Building-Player-and-Widgets)

After opening Visual Studio ALL_Build -> Build can be started which is found under CMakePredefinedTargets.



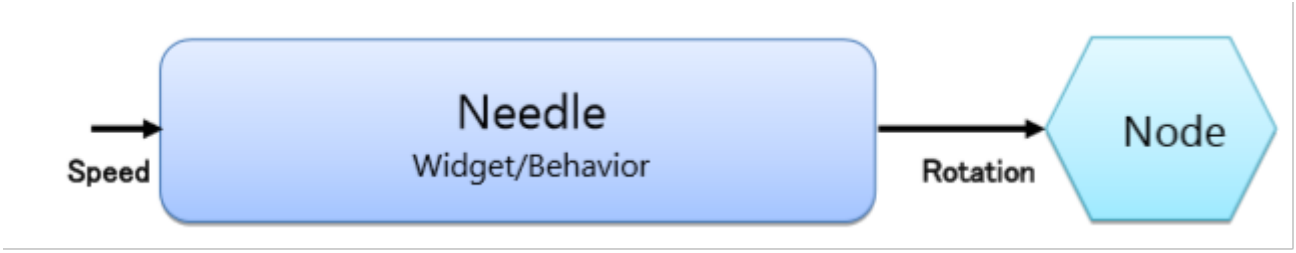
Now the specific Project can be opened and the custom Behavior can be used.

CGI-Studio Default Behaviors

CGI-Studio already provides multiple Behaviors by default which can already be used in projects. While they are also just Behaviors the follow a new concept for development which has been made possible by the new features of Behaviors.

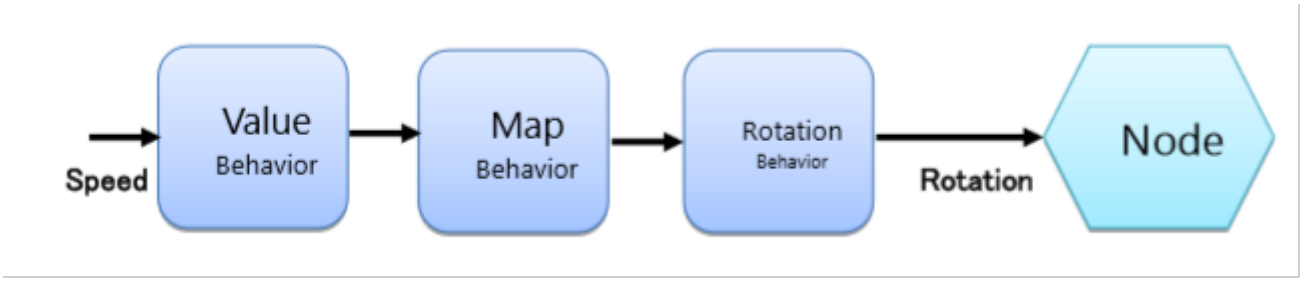
Basic Concept

Previously Widgets where rather self-contained blocks of functionality. For example a needle widget would receive an input value (e.g. Speed) map it to a rotation angle and apply this rotation to a node.

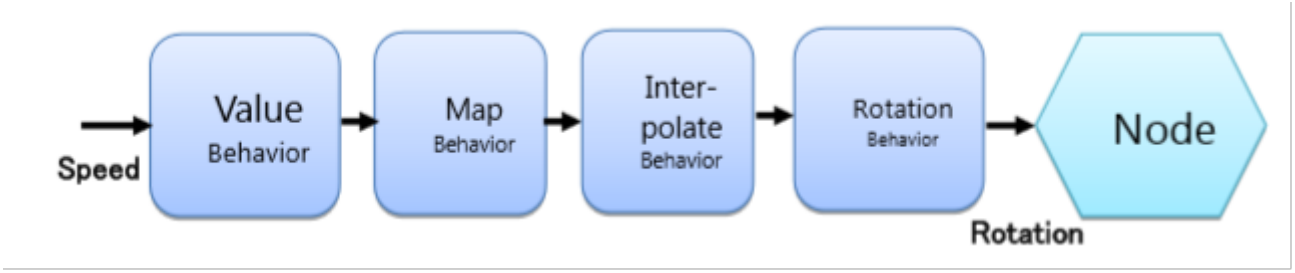


It would be possible to recreate this exact functionality as a behavior though the decision at Candera GmbH was to have multiple smaller behaviors which each implement only a small bit of functionality. Then combine these to achieve the desired results.

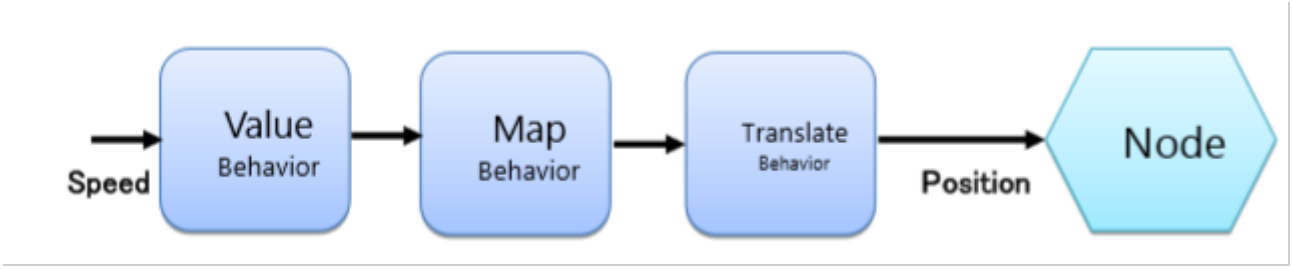
In the above sample this would result in a Value-Behavior which receives the value, a Map-Behavior which maps it to an angle and a Rotation-Behavior which applies the angle as a rotation on the node.



This approach has the advantage that it drastically improves the flexibility due to the modular design. For example if the rotation shouldn't happen instantly but slowly over time an interpolation behavior can be added.



If instead of rotating a needle, a pointer is just moved forward or back to indicate the value the Rotation-Behavior can be replaced with a Translation-Behavior.



All this can be achieved by simply recombining existing behaviors without the need to code new ones.

Event Handler Behavior

When an Event-Handler Behavior receives an event it will pass this event to a number of Condition-Behaviors. The Condition-Behaviors can then decide if this event is "accepted" or not. In case the event is accepted all attached Action-Behaviors will be executed.

How to create a new Condition-Behavior

To create a custom Condition-Behavior a new Behavior derived from Candera::ConditionBehavior must be created.

In the OnEvent() method an EvaluateConditionEvent will be received which allows to get the original event via the GetTriggerEvent() method. Further the dispatchResult will be of the type ConditionEvaluationResult. To accept the event the Match() method of the result must be called.

```
...
void ClickConditionBehavior::OnEvent(const FeatStd::Event
& event, Candera::EventDispatchResult& dispatchResult)
{
    const Candera::EvaluateConditionEvent
* conditionBehaviorEvaluationEvent = Candera::Dynamic_Cast<const Candera::EvaluateConditionEvent

    if (0 != conditionBehaviorEvaluationEvent) {
        const ClickEvent* clickEvent =
              Candera::Dynamic_Cast<const ClickEvent*>(&(conditionBehaviorEvaluationEvent->
GetTriggerEvent()));

        if (0 != clickEvent) {
            if (clickEvent->GetClickCount() == GetClickCount()) {
                Candera::ConditionEvaluationResult* conditionEvaluationResult = Candera::Dynamic

                if (0 != conditionEvaluationResult) {
                    conditionEvaluationResult->Match();
                }
            }
```

```
        }
    }
}
...
```

How to create a new Action Behavior

To create a custom Action-Behavior a new Behavior derived from Candera::ActionBehavior must be created.

In the OnEvent() method an InvokeActionEvent will be received which indicates that the action should be executed.

```
...
void ChangeValueActionBehavior::OnEvent(const FeatStd::Event
& event, Candera::EventDispatchResult& dispatchResult)
{
    FEATSTD_UNUSED(dispatchResult);
    const Candera::InvokeActionEvent
* invokeActionEvent = Candera::Dynamic_Cast<const Candera::InvokeActionEvent*>(&event);

    if (0 != invokeActionEvent) {
        // Do something
    }
}
...
```

How to handle Touch-Events

To create a Behavior which reacts on Touch interaction a new Behavior derived from CgiStudioControl:: TouchableBehavior must be created. This class registers to the TouchSession and processes the incoming TouchSession-Events to perform a hit detection. If a hit is detected it is possible to react to Touch-Event in the behavior.

From the event the Touch-Info can be retrieved which contains important information about the Touch-State (Down, Move, Up), Pointer-Id (Multi-Touch), Source-Id (Multi-Display) and Touch-Coordinates.

```
...
    virtual void OnEvent(const FeatStd::Event
& event, Candera::EventDispatchResult& dispatchResult)
    {
        const Candera::TouchEvent* touchEvent = Candera::Dynamic_Cast<const Candera::TouchEvent*

        if (touchEvent != 0) {
            switch (touchEvent->GetTouchInfo().m_state) {
```

```
                case Candera::TouchInfo::Down:
                    //OnBeginDrag(event.GetTouchInfo().m_x, event.GetTouchInfo().m_y);
                    break;
                case Candera::TouchInfo::Move:
                    //OnDrag(event.GetTouchInfo().m_x, event.GetTouchInfo().m_y);
                    break;
                case Candera::TouchInfo::Up:
                    //OnEndDrag(event.GetTouchInfo().m_x, event.GetTouchInfo().m_y);
                    break;
                default: break;
            }
        }
        else {
            Base::OnEvent(event, dispatchResult);
        }
    }
...
```

Manual Touch handling

If a more customized processing of Touch-Events is required, instead of deriving from CgiStudioControl:: TouchableBehavior it is also possible to manually implement Touch-Handling. First the behavior must be marked as Touchable by implementing the "virtual bool IsTouchable() const" method or simply adding the following macro.

```
...
CGI_BEHAVIOR_IS_TOUCHABLE()
...
```

Next the Behavior must be registered/unregistered to the Touch-Session. This allows receiving TouchSession-Events.

```
...
void TouchableBehavior::Register()
{
    Deregister();
    Candera::TouchSession::GetInstance().Register(GetTouchable());
    m_touchSession = &Candera::TouchSession::GetInstance();
}

void TouchableBehavior::Deregister()
{
    if (0 != m_touchSession) {
        m_touchSession->Deregister(GetTouchable());
        m_touchSession = 0;
    }
}
...
```

The basic touch handling (object hit detection) must be manually implemented. This can be a bit more complicated but related code can be found in the TouchBehavior::OnEvent which can be used as basis for a custom implementation.

```
...
    void TouchableBehavior::OnEvent(const FeatStd::Event
& event, Candera::EventDispatchResult& dispatchResult)
    {
        if (!TouchSessionEventHandler::Handle(*this, event, dispatchResult)) {
...
            Base::OnEvent(event, dispatchResult);
        }
    }
...
```

And the TouchSessionEventHandlerImpl::OnTouchSessionEvent.

```
...
    void OnTouchSessionEvent(const Candera::TouchSessionEvent& event, Candera::TouchSessionEvent
    {
        if (m_intersectionTest) {
            if (m_behavior.GetNode().IsEffectiveRenderingEnabled()) {
                Courier::View* view = m_behavior.GetParentView();
                if (0 != view) {
                    const Candera::AbstractNodePointer& abstractNode = m_alternativeNode.
IsValid() ? m_alternativeNode : m_behavior.GetNode();
                    const Candera::TouchInfo& touchInfo = event.GetTouchInfo();
#ifdef CANDERA_2D_ENABLED
                    Courier::ViewScene2D
* viewScene2D = view->ToViewScene2D();
                    if (0 != viewScene2D) {
                        Candera::Node2D* node2D = abstractNode.ToNode2D
();
                        if (0 != node2D) {
                            Courier::ViewScene2D::CameraPtrVector& cameras = viewScene2D->GetCam
                            for (FeatStd::SizeType i = 0; i < cameras.Size(); ++i) {
                                Candera::Vector2
 touchPosition(FeatStd::Float(touchInfo.m_x), FeatStd::Float(touchInfo.m_y));
                                if ((0 != cameras[i]) && node2D->
IsPickIntersectingBoundingRectangle(*cameras[i], touchPosition)) {
                                    dispatchResult.SetTouched(true);
                                    dispatchResult.SetTouchUsage(m_touchUsage);
                                    return;
                                }
                            }
                        }
                }
#endif
#ifdef CANDERA_3D_ENABLED
```

```
                    Courier::ViewScene3D
* viewScene3D = view->ToViewScene3D();
                    if (0 != viewScene3D) {
                        Candera::Node* node = abstractNode.ToNode
();
                        if (0 != node) {
                            Courier::ViewScene3D::CameraPtrVector& cameras = viewScene3D->GetC
                            for (FeatStd::SizeType i = 0; i < cameras.Size(); ++i) {
                                Candera::Camera
* camera = cameras[i];
                                FeatStd::Float distance = Candera::Math::MaxFloat();
                                if ((0 != camera) && camera->IsRenderingEnabled() && node->
IsPickIntersectingGeometry(*camera, static_cast<FeatStd::Int>(touchInfo.m_x),
                                                                                    static_c
                                    dispatchResult.SetTouched(true);
                                    dispatchResult.SetTouchUsage(m_touchUsage);
                                    return;
                                }
                            }
                        }
                    }
#endif
                }
            }
        }
        else {
            dispatchResult.SetTouched(true);
            dispatchResult.SetTouchUsage(m_touchUsage);
        }
    }
...
```

How to handle View-Activation-Events

View-Activation-Events are events send by the View to indicate if it has been activated or deactivated. To react to these events the Behavior must register at the View as an Event-Listener.

```
...
EventListenerAdapter m_eventListenerAdapter;
...
```

```
...
    CustomBehavior::CustomBehavior()
    {
        m_eventListenerAdapter.SetBehavior(this);
        Courier::View::GetEventSource().AddEventListener(&m_eventListenerAdapter);
    }

    CustomBehavior::~CustomBehavior ()
```

```
    {
        Courier::View::GetEventSource().RemoveEventListener(&m_eventListenerAdapter);
        m_eventListenerAdapter.SetBehavior(0);
    }
...
```

It is now possible to receive as Courier::ViewActivationEvent via the OnEvent() method.

External Image Behaviour

Behaviour Overview

Feature Description

The external resource behaviours **SetExternalBitmap** and **ExternalBitmapProvider** support a file resource manager called **FileSystemResourceManager** (identified as **FileSystem**).

These behaviours support two resource managers: the **Android Resource Manager** and the **FileSystem Resource Manager**. When the **Android Resource Manager** is used, a unique resource ID (e.g., a bitmap name) is specified, and the target platform is Android. When the **FileSystem Resource Manager** is used, a file path from the project file system (outside the binary) can be specified, and **ExternalResourceManagerId** and **ExternalResourceId** become configurable properties of the External Resource behaviour.

When the **ExternalResourceManagerId** is set to **FileSystem**, the behaviour enables dynamic image loading from the file system at runtime to VRAM. Users can specify a relative image path in the **ExternalResourceId** property to load image files such as PNG and JPEG without embedding them directly into the binary file. These behaviours internally use the lodepng and IJG (libjpeg) libraries to decode, encode, and compress image data as needed.

A **relative path** refers to a path that is defined relative to the application's working directory or a configured asset root.
For example:

- Logos\400x150\CGI-Studio-Translator-final.png
- Logos\400x150\CGI-Studio-Translator-final.jpg

The **Filesystem resource manager** enables the management of image resources without the need to import them into the Scene Composer. Images can be accessed and loaded directly from the application's file system at runtime.

Supported Platforms

It supports all platforms.

Limitations:

1. These behaviours are used to set images on 2D bitmap nodes and cannot be used to set textures for **3D objects**.

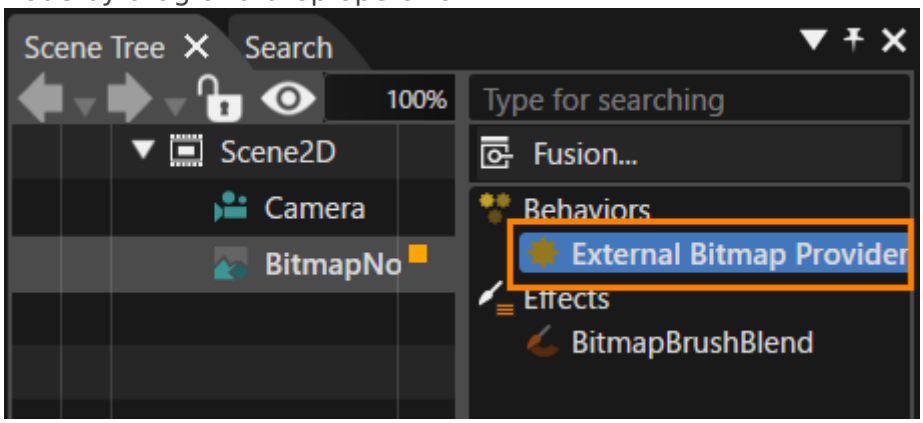
2. There is no dropdown list available in the **ExternalResourceManagerID** property to select between **FileSystem** and **Android system** — the user must set it manually.
3. The FileSystemResourceManager supports loading and displaying only PNG and JPG images. It does not support file paths of other formats.

Using External Resource Behaviours

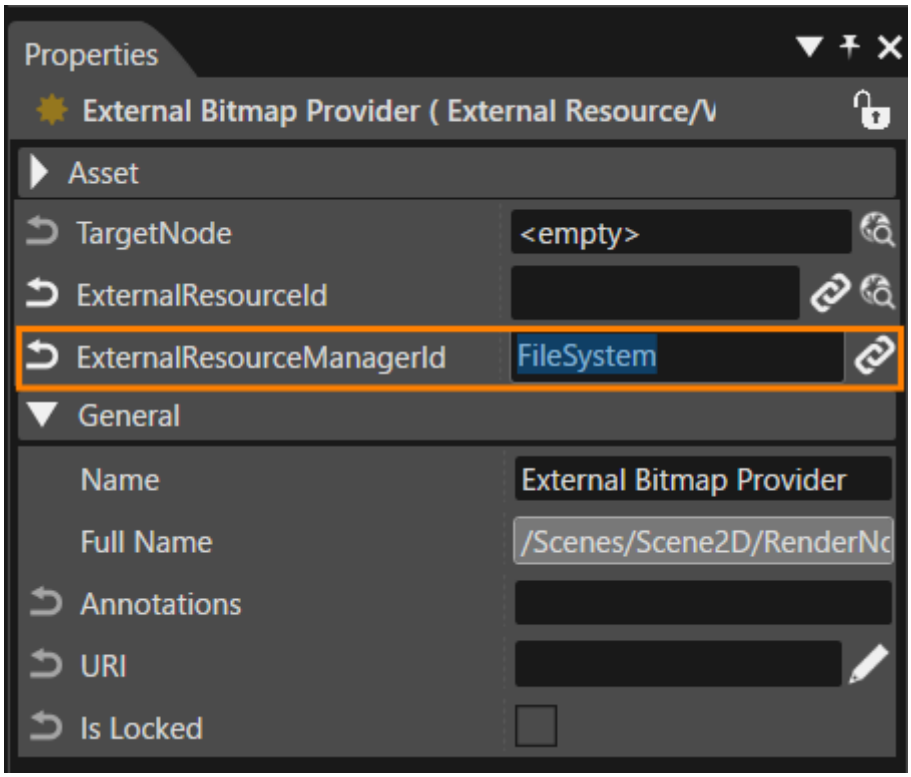
External Bitmap Provider

1. Drag and drop the **Node 2D > Bitmap Node** from the toolbox onto the scene editor, or the scene tree.

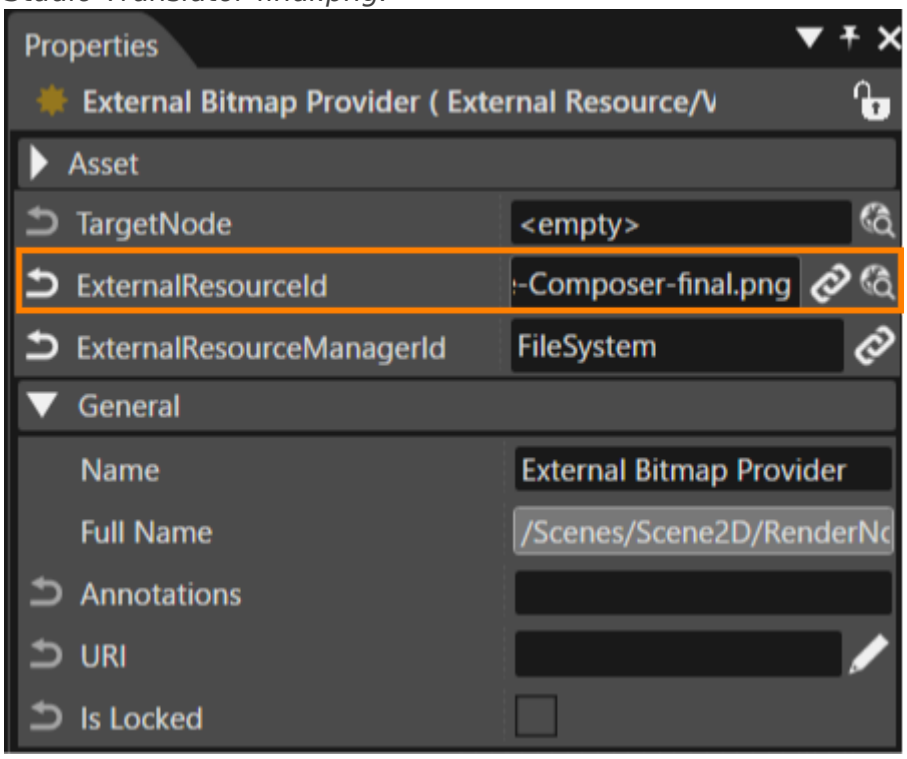
2. Attach **Behavior > External Resource > Value Processing > External Bitmap Provider** in the Toolbox to the Behavior in the Extra Scene Tree of the placed bitmap node by drag-and-drop operation.



3. Click **External Bitmap Provider**, and in the properties panel that opens, change **ExternalResourceManagerId** from **Android** to **FileSystem** (default: **Android**).



4. Next, set the relative path to the external image in the **ExternalResourceId** property. For example: *<trunk>\cgi_studio_content\Resources\cgi_studio\Logos\400x150\CGI-Studio-Translator-final.png.*



5. The image will be loaded into the solution, and the image from the given relative path will be displayed on the screen.

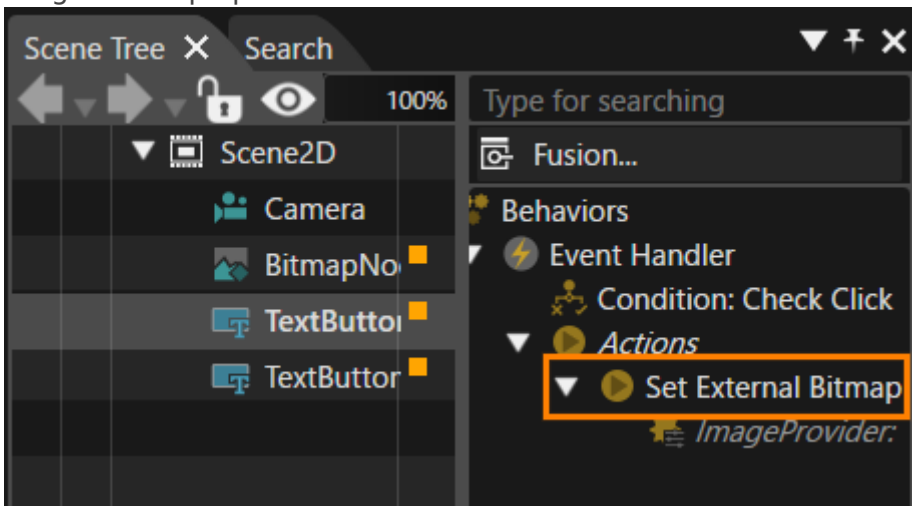


Set External Bitmap

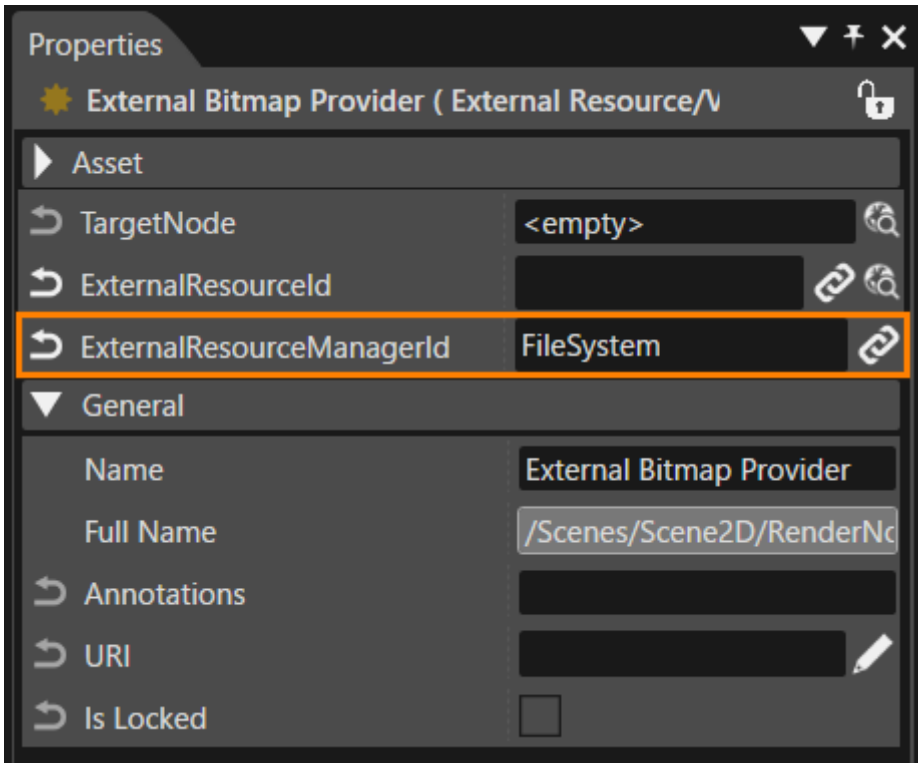
1. Drag and drop the **Node 2D > Bitmap Node** from the toolbox onto the scene editor, or the scene tree.

2. Drag and drop the **Controls > Touchable > TextButton** from the toolbox onto the scene editor, or the scene tree. Take place this operation twice (add two button control nodes).

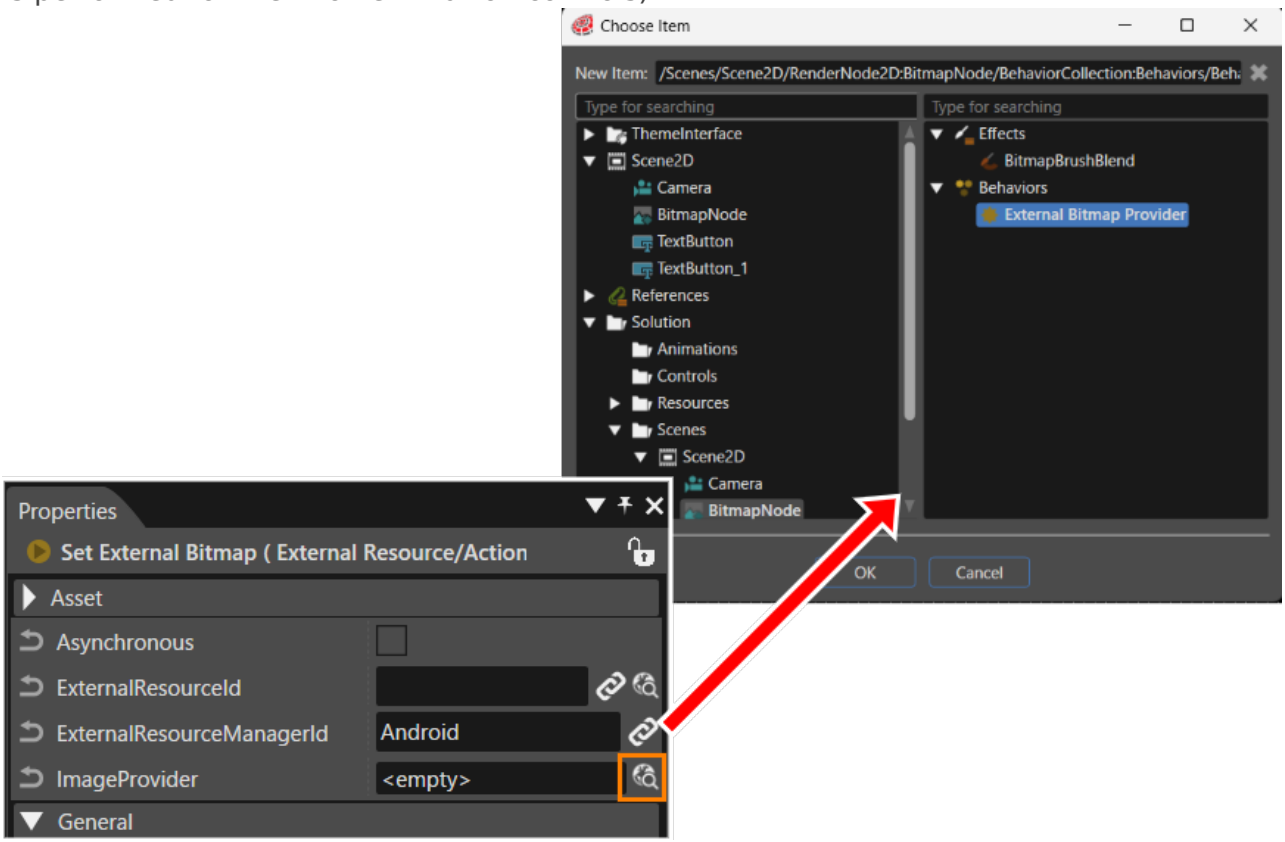
3. Attach **Behavior > External Resource > Action > Set External Bitmap** in the Toolbox to the Actions in the Extra Scene Tree of the placed Text Button control nodes by drag-and-drop operation.



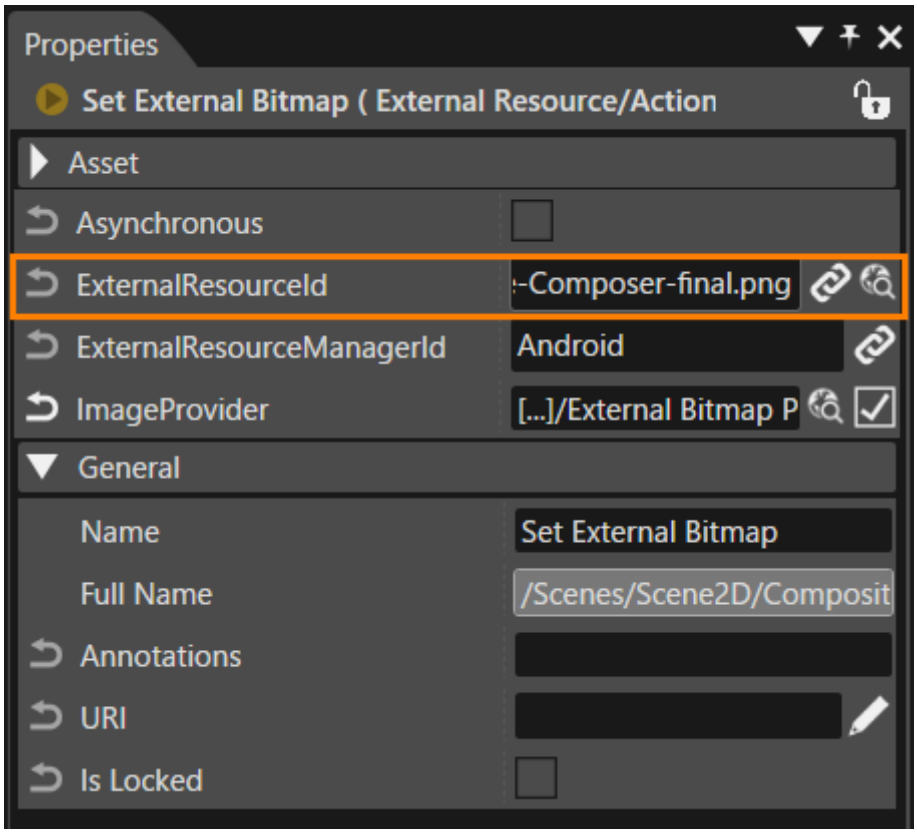
4. Change the *ExternalResourceManager ID* to **FileSystem** to make the behavior support file system resources. Click **Set External Bitmap** for each control, and in the properties panel that opens, change **ExternalResourceManagerId** from **Android** to **FileSystem** (default: **Android**).



5. Click the select item button on the Image Provider in the properties panel that appears and select the **External Bitmap Provider** connected to the BitmapNode (this operation is performed for the two Text Button controls).



6. Set the relative image path to the *ExternalResourceID* property of the **Set External Bitmap** behavior in the button's actions.(This operation is performed for both button controls that use the **Set External Bitmap** behavior.)



7. Run the player and press the text buttons. When each button is pressed, the image at the specified path will be loaded by the **Set External Bitmap** behaviors and displayed on the screen.



External Resource Behaviour

Set External Bitmap Behavior

Sets **ExternalResourceId** and **ExternalResourceManagerId** properties of an associated **ExternalBitmapProvider** when its action is triggered. For details on the property items, please refer to the Action External Resource Resource.

External Resource Provider Behavior

Sets an image on a render node based on the **ExternalResourceId**. For details on the property items, please refer to the Value Processing External Resource Behavior.

- If the ID is *Android*, it retrieves an external bitmap using the **ExternalResourceId** from the external resource provider specified by the **ExternalResourceManagerId**, and provides the bitmap to the associated bitmap node.
- If the ID is *FileSystem*, it retrieves the image from the specified file, converts it to a bitmap using the **FileSystemResourceManager**, and provides it to the associated bitmap node.

Implementation Structure

1. **Included the IJG Library** in the project to decode JPEG images and convert them into pixel data.
   - The library source files are located in the cgi_studio_3psw\ src\ijg
   - Both **LodePNG** (for PNG) and **IJG** (for JPEG) are compiled as static libraries.
2. The **SetExternalBitmap** and **External Behaviour** functionalities already supported the **Android Resource Manager**.
   - Bitmaps are loaded at runtime based on the configured ExternalResourceManagerId, which determines whether to use the **File System** or **Android Resource System**.
3. **Added two new files:** FileSystemResourceManager.cpp and FileSystemResourceManager.h in the trunk.
   - Definedaclassnamed**FileSystemResourceManager**, derived from ExternalResourceManager.
4. The ExternalResourceManagerId determines which resource manager handles the request (e.g., FileResourceManager or FileSystemResourceManager).
5. The **FileSystemResourceManager** class allows loading PNG and JPEG images directly from file paths and interacts with the **IJG** and **lodepng** libraries.
   - An instance of this class is created when the user sets the **ExternalResourceManager Property ID** to **Filesystem**.
6. The **External Resource Manager** handles resources (such as images and bitmaps) that exist **outside** the project's asset binary file.
7. Resource managers are registered by calling the **DefaultSetup**() function in . in the applications.
   - The definition of this function is located in **src\CanderaPlatform\Device\Common\Base\DefaultSetupExternalResourceM**

anagers.cpp

8. When Default Setup() is executed, two static objects are created —
   - One in FileResourceManager for Android resources
   - Another in FileSystemResourceManager for file system resourcesBoth classes are derived from the common base class **ExternalResourceManager**.
9. The objects of these classes are registered using the Register() function of ExternalResourceManager, and are inserted into a **global map** for access during runtime.
10. A new class named **ImageLoader** has been defined in the newly introduced file **ImageLoader**, located in ***featstd\src\FeatStd\Platform***.The **ImageLoader** class acts as a platform-independent interface for loading, decoding, and storing image files in various formats. It ensures consistent usage across different systems and simplifies image handling for the resource manager across multiple operating systems.
11. Each platform directory contains a file named **PlatformImageLoader**, which defines the platform-dependent implementation corresponding to the **ImageLoader** interface. The **PlatformImageLoader** implementation ensures that image handling is optimized and compatible with the target operating system (e.g., Windows, Linux, Android, RTOS, etc.).

Other Changes

1. Modification in **IdentifierHelper** Structure
   - **File:** IdentifierHelper.cpp
   - **Why:** The resource ID (e.g., ExternalResourceId) is not a plain string but an object that can be converted to a string using IdentifierHelper::GetStringByIdentifier.
   - **Purpose:** This change allows the resource manager to handle hierarchical and structured resource paths efficiently and safely.

2. Replaced Name with Identifier Data Type
   - The data type **Name** has been replaced with **Identifier** in all external resource management classes.
   - **Reason for Change:**
     - **Name** represents a single, non-hierarchical string label (e.g., "Logo" or "Scene1") and is suitable only for flat resources such as bitmap names or unique IDs obtained from ExternalResourceId.
     - **Identifier** supports hierarchical resource paths, improving the flexibility of resource management — mainly for the FileSystemResourceManager.
   - **Files Modified:**
     - ExternalBitmapProviderBehavior.cpp
     - ExternalBitmapProviderBehavior.h
     - ExternalResourceChangeConditionBehavior.cpp
     - ExternalResourceChangeConditionBehavior.h
     - SetExternalBitmapActionBehavior.cpp
     - SetExternalBitmapActionBehavior.h
     - ExternalBitmapProviderTest.cpp
     - ExternalResourceChangeConditionTest.cpp

    - SetExternalBitmapActionTest.cpp

3. Update in GetBitmapResource Member Function (ExternalResourceManager Class)
    - The **GetBitmapResource** member function definition in **ExternalResourceManager** has been updated.
    - **Reason for Change:**
      To retrieve a bitmap resource from a registered external resource manager using a **Manager ID** and a **Resource ID** (both as Name::SharedPointer).
    - The function converts the resource name to an **Identifier**, then delegates the actual bitmap retrieval to the selected resource manager.This design enables **flexible and modular access** to bitmap resources from different managers.

Display Bitmap from File Path

1. **The Candera framework uses the LodePNG and IJG libraries to decode image files:**
   LodePNG handles PNG images.
   IJG handles JPEG images.
   During decoding, the compressed data is converted into raw bitmap data (e.g., RGBA for PNG or RGB for JPEG).

2. When the **ExternalBitmapProvider behavior** properties are configured with an External Resource Manager ID and a full file path,
   The OnChanged() function of the behavior is triggered,
   Which then calls updateImage() inside the function

3. ExternalResourceManager::GetBitmapResource (const FeatStd::Internal::Identifier::SharedPointer& resourceId)
4. The GetBitmapResource() function retrieves the corresponding Resource Manager object based on the Resource Manager ID and reads the user-configured properties from ExternalBitmapProvider.
5. Depending on the ID, it invokes the LoadBitmapResource() function:
6. If the Resource ID is set to FileSystem,
7. the LoadBitmapResource() function defined inside the FileSystemResourceManager class is called.
8. LoadBitmapResource() takes the resource ID (image file path) as input and processes it based on the file extension (e.g., .png or .jpg).
9. It then calls the LoadImage() function with the file path and detected format.
10. This function is defined in ImageLoader.h and implemented in GenericImageLoader.cpp.
11. The GenericImageLoader class uses two specialized loaders:
    GenericJpegLoader (GenericJpegLoader.cpp/h)
    GenericPngLoader (GenericPngLoader.cpp/h)
12. When the Load() function of either loader is called:
    The lodepng library loads PNG files.
    The IJG (libjpeg) library loads JPEG files.

13. The image data is decoded and converted into raw pixel data, stored in a structure containing image properties (height, width, pixel info).
14. The decoded bitmap is then returned from LoadBitmapResource() to the updateImage() function of ExternalBitmapProviderBehavior, where:
15. The bitmap effect is applied.
16. The image properties are updated for display on the screen.
17. When the **SetExternalBitmapActionBehavior** is triggered (for example, by an event or property change), it sets the Resource Manager ID and Resource ID on the ExternalBitmapProviderBehavior,
18. after which the process continues as described above.

For example:
Call Stack (Reference): Demonstrates how a relative image file path is converted into pixel data using the **lodepng** library during bitmap creation.

```
Courier::ViewScene::Update(renderHint)
CgiStudioControl::ExternalBitmapProviderBehavior::UpdateImage()
ExternalResourceManager::GetBitmapResource(resourceManagerId, resourceId)
FileSystemResourceManager::LoadBitmapResource(resourceId)   // when Manager ID = FileSystem
FeatStd::Internal::ImageLoader::LoadImage(filename, format, output)
FeatStd::Internal::Generic::GenericImageLoader::LoadImage(filename, format, output)
FeatStd::Internal::Generic::GenericPngLoader::Load(filename, output)
lodepng_decode32_file(out, w, h, filename)
```

FileSystemResourceManager Overview:

The **FileSystemResourceManager** class, defined in *FileSystemResourceManager.h*/*cpp*, provides functionality to load bitmap images from external files. It handles image format detection, error reporting, and seamless integration with the existing resource management system.

The **FileSystemResourceManager** class is derived from the common base class **ExternalResourceManager**. **ExternalResourceManager** is the common parent of both the **AndroidResourceManager** and **FileSystemResourceManager** classes and defines a shared interface and common functionality for managing and loading external resources, enabling consistent handling of bitmap images across different platforms and resource types.

Class Overview

The FileSystemResourceManager class includes the following member functions

*FileSystemResourceManager(**const FeatStd ::Internal::Name::SharedPointer**&resourceManagerId*	Initializes the resource manager with a unique identifier(FileSystem). Stores the provided resourceManagerId in the member variable m_resourceManagerId.
*~**FileSystemResourceManager()***	
*Bitmap::**SharedPointer LoadBitmapResource**(**const FeatStd**::Internal::Identifier::**SharedPointer**& **resourceId**)*	Loads a bitmap image from the file system using the provided resource identifier. This function is called from ExternalBitmapProvider when the ExternalResourceManagerID is set to FileSystem. **Path Extraction:** Converts the identifier to a file path string using IdentifierHelper::GetStringByIdentifier. **Format Detection:** Determines the image format (PNG, JPEG) by checking the file extension or reading the file header. **File Access:** Opens the file using FileStream. If the format is unknown, reads the header to detect PNG/JPEG signatures. **Image Loading:** Uses ImageLoader::LoadImage to read the image data into a RawImageData structure. **Bitmap Creation:** Calls Bitmap::Create to construct a bitmap object from the raw image data, pixel format, and other properties. **Return:** Returns the created bitmap as a shared pointer
*bool **RegisterFileSystemResourceManager**(const FeatStd::Internal::Name::SharedPointer& name)*	Registers a new instance of FileSystemResourceManager with the global ExternalResourceManager registry. Creates a new manager and calls ExternalResourceManager::Register with the provided name and manager instance.
bool UnregisterFileSystemResourceManager(const FeatStd::Internal::Name::SharedPointer& name	: Unregisters an existing FileSystemResourceManager from the global registry. : Unregisters an existing FileSystemResourceManager from the global registry.
GetPixelFormat()	Converts an internal image format (e.g., RGBA, RGB) to the corresponding Bitmap::PixelFormat used by the Candera engine.
GetFileExtension()	Extracts the file extension from a file path (e.g. image path) string. Iterates through the string to find the last dot (**.**) and returns the substring after it, which is the extension.

Dependent Classes of FileResourceManager for Converting File Path to Bitmap.

New files have been introduced to serve as a common, platform-independent interface. The files listed below have been added to the project and act as a common interface for handling different image formats. Based on the file format (PNG or JPEG), the lodepng and IJG (libjpeg) libraries are used to convert the image from the file system into pixel data

The following C++ files have been introduced:

GenericImageLoader.cpp/h GenericJpegLoader.cpp/h GenericPngLoader.cpp/h ImageLoader.h

Details about the classes defined above these files.

ImageLoader	1. The ImageLoader class provides a platform-independent interface for loading image files (such as PNG, JPEG, BMP) into raw image data structures that can be used by the application. 2. **Identifier** supports hierarchical resource paths, improving the flexibility of resource management — mainly for the FileSystemResourceManager.
GenericImageLoader	1. The GenericImageLoader class provides a unified, platform-independent interface for loading and storing image files in various formats (such as PNG and JPEG). 2. **Supported Formats:** PNG (calls GenericPngLoader::Store), JPEG (calls GenericJpegLoader::Store). 3. This class has one main member function, *LoadImage*(**const FeatStd**::Charfilename, ImageInputFormat), which loads and decodes an image file (e.g., PNG or JPEG) from disk using the file pathprovidedby **FileSystemResourceManager**, and stores it into a raw image structure.

GenericJpegLoader	1. The GenericJpegLoader class is designed to load and decode JPEG image files into a raw image data structure that can be used FileSystemResourceManager. 2. This class has one main member function, *GenericJpegLoader*::Load(**const FeatStd**::*Char* filename, RawImageData& output),which is called form the **GenericImageLoader** class when the file format is JPEG. It loads and decodes the JPEG image file into a **RawImageData** structure that can be used by the application.This function uses the **IJG** (Independent JPEG Group) library, a widely used open-source JPEG decoder, to read the JPEG file specified by filename(e.g. .png/jpeg). It decodes the compressed JPEG data into uncompressed pixel data and fills the output structure with the image's width, height, pixel format, and pixel buffer.
GenericPngLoader	1. The GenericPngLoader class is responsible for loading and decoding PNG image files into a raw image. 2. Uses a standard PNG decoding library (such as lodepng) to parse and decompress PNG images. 3. GenericPngLoader::Load(const FeatStd::Char filename, RawImageData& output) is to load and decode a PNG image file into a raw image data structure (RawImageData) for use in ExteranalResourceBehaviors. 4. This function uses a PNG decoding library (Lodepng) to read the PNG file specified by filename. It parses the PNG format, decompresses the image data, and fills the output structure with the image's width, height, pixel format, and pixel buffer.

Filesystem vs. Android Resource Manager ID

Resource managers in CGI Studio abstract the way external resources (such as images) are loaded and managed. Two commonly used managers are the FileSystem Resource Manager and the Android System Resource Manager. Each is designed for different platforms and use cases.

FileSystem Resource Manager

Purpose:

The FileSystem Resource Manager (FileSystemResourceManager) is designed to load resources directly from the device’s filesystem. It is platform-agnostic and works on desktop, embedded, and other systems where resources are stored as files. Uses file paths or hierarchical identifiers to locate resources (e.g., images) on disk. Supports common image formats like PNG and JPEG, with automatic format detection. Reads files using standard file I/O operations and Convert to bitmap with help of Loadepng and ijg Opensource libraries.

Android System Resource Manager

Purpose:

The Android System Resource Manager is tailored for Android platforms, leveraging the Android resource system to access images and other assets packaged within the APK or available via Android’s resource APIs.

How It Works:

Uses Android resource IDs (not file paths) to locate resources. Resources are typically bundled with the application and managed by the Android OS. The resources read from the

- May have limitations on dynamic resource updates, as resources are often static once packaged.

Benefits of FileSystem Resource Manager :

1. Platform Flexibility:
   Works across multiple platforms, not limited to Android.

2. Dynamic Resource Updates:
   Resources can be updated, replaced, or added at runtime without rebuilding the application.