Related: Action Maps

For information on controller mapping, head over to the documentation here: Setting Up Controls and Action Maps

Input System Overview

CryInput's main purpose is to provide an abstraction over obtaining input/status from different input device such as a keyboard, mouse, joysticks, 360/PS3 gamepads, etc.

It also has support for sending feedback events back to the input devices, for example in the form of force feedback events.

The common interfaces for the input system can be found in IInput.h, in the CryCommon project.

IInput

The main interface to the input system is IInput. An instance implementing this interface is created during system initialisation automatically in the InitInput function (InitSystem.cpp in CrySystem, see also CryInput.cpp in CryInput).

Only one instance of this interface will be created. The updating and shutdown of the input system will also be managed by CrySystem.

This IInput instance will be stored in the SSystemGlobalEnvironment structure gEnv, and can be accessed through gEnv->pInput or alternatively through the system interface by GetISystem()->GetIInput(). Access through the gEnv variable is the most commonly used method.

IInputListener

One of the most common use cases in relation with the input system is to create listener classes in other modules (e.g. CryGame) by inheriting from IInputEventListener and registering/unregistering the listener class with the input system to receive notifications on input events.

For example, the Action Map System registers itself as an input listener and forwards game events only for the keys defined in the profile configuration files to further abstract the player input from device->game.

SInputEvent

This is the structure that encapsulates information created by any input device and are received by all input event listeners.

InputDevice

InputDevices normally relate directly to the physical input devices such as a joypads, mouse, keyboard, etc. To create a new input device all that's needed is to implement all functions in the IInputDevice interface and registering an instance of it with the Input System via the AddInputDevice function.

The Init function will be called when registering the IInputDevice with the Input System, so there's no need to call it manually when creating the input devices.

The Update function will be called at every update of the Input System, and this is generally where the state of the device should be checked/updated and the Input Events generated and forwarded to the InputSystem.

It's common for Input Devices to create and store list of SInputSymbol for each symbol the input device is able to generate in the Init function, and then in the update function lookup the symbol for the button/axis that have changed and use those (via their AssignTo function) to fill in most of information needed for the events and forward them to the input system.

Example:

```
// function from CInputDevice ( accessible only within CryInput )
MapSymbol( ... )
{
     SInputSymbol\* pSymbol = new SInputSymbol( deviceSpecificId, keyId, name, type );
     pSymbol->user = user;
     pSymbol->deviceId = m_deviceId;
     m_idToInfo\[ keyId \] = pSymbol;
     m_devSpecIdToSymbol\[ deviceSpecificId \] = pSymbol;
     m_nameToId\[ name \] = deviceSpecificId;
     m_nameToInfo\[ name \] = pSymbol;

     return pSymbol;
}
bool CMyKeyboardInputDevice::Init()
{
      ...
     //CreateDeviceEtc();
      ...
     m_symbols\[ DIK_1 \] = MapSymbol( DIK_1, eKI_1, "1" );
     m_symbols\[ DIK_2 \] = MapSymbol( DIK_2, eKI_2, "2" );
      ...
}
void CMyKeyboardInputDevice::Update( ... )
{
     // Acquire device if necessary
      ...
     // Will want to check for all keys probably, so the following section might be part of a loop
     SInputSymbol\* pSymbol = m_symbols\[ deviceKeyId \];
      ...
     // check if state changed
      ...
     // This is an example for when pressed, see ChangeEvent function for axis type symbols
     pSymbol->PressEvent( true );

     SInputEvent event;
     pSymbol->AssignTo( event, modifiers );

     gEnv->pInput->PostInputEvent( event );
}
```

To forward events to the input system so that event listeners can receive them, use the PostInputEvent function from IInput.

If adding your input device to CryInput, it may be useful to inherit directly from CInputDevice as it already provides a generic implementation for most functions in IInputDevice.

This file is included with the full source of CryEngine and is not available in the FreeSDK or GameCodeOnly solutions. For these licenses please derive from IInputDevice directly.