Getting Started with HInput

About this Doc

This is a ten minute overview of the HInput system, covering what it does, why you might care, and how you can make the most of it.

Introduction

As an input abstraction layer, the primary purpose of HInput is to decouple game/application code from specific input implementations. For example, rather than checking whether a gamepad button is down, or a key was pressed, we just query whether a named input was satisfied without needing to know how exactly it was implemented.

This extra layer of abstraction gives us the flexibility to:

1. Change input schemes without code changes or recompilation.
2. Dynamically switch between input schemes at runtime (e.g. we might choose to load a new scheme if we detect that a gamepad was connected). We can mix and match schemes, and different player can use different schemes. All this is completely transparent to the game code.
3. Using input metadata, it is possible to allow dynamic configuration of the system at runtime (which is how the HInput Editor works).

HInput also has various layers of complexity in its interface:

1. The simplest way of interacting with the system is through the Editor application.
2. The `InputManager` singleton [`HInput::GetManager()`] provides everything you need to use an input configuration to provide input to your game.
3. Advanced users can write custom `Inputs`, `InputDeviceServices`, and plugins, as well as custom GUIs using the `InputMetaData` stored in the system.

A first input

For our first example let's use a common input found in most games: 'fire'. Our first instinct would be to write code like this:

```
if ( pGamepad->isButtonDown(GamepadBtn::A) ||
     pKeyboard->isButtonDown(Keys::Space) )
{ /* Fire a bullet */ }
```

However, with HInput this code would change to:

```
if ( HInput::GetManager().WasInputSatisfied("Fire") )
{ /* Fire a bullet */ }
```

What does an 'input' support

Through the InputManager, a client can access the following information about a named input:

1. Whether the input exists (is currently part of a player's loaded set of inputs)

2. If it exists, whether the input is 'available'. This usually means whether the service that supplies the input is available (e.g. a gamepad is connected).
3. If it exists, whether the input is 'satisfied'. The meaning of this is entirely dependent on the input (e.g. was a button pressed, did an axis value exceed a threshold, etc).
4. If it exists, what (`float`) 'value' it has. Most frequently used for analog inputs, it could however mean anything (e.g. how long a button has been held down).

You will immediately notice that there is a lot of context sensitivity in these inputs, which is a necessary side effect of giving up the tight coupling of input system to game code. However, this does mean that there is potential for ambiguity as the game code must now rely on conventions used in setting up the system.

Because of the ambiguity of these terms, it is important to be consistent when configuring the input system. For example, don't redefine an input with two different conventions. A good example is the "Gamepad_AxisInfo" input and the "Keyb_AxisEmulation" input (which emulates an analog input with two keyboard keys). You should ensure that the emulated axis maps correctly onto the gamepad axis 'value', so the game code can work with either. Likewise, it might not be appropriate to replace an input that uses the 'value' component with one that does not (e.g. a button press that mainly uses 'satisfied').

Input Sets

HInput allows you to arrange inputs together in 'sets' that can be loaded dynamically. An example would be a set of inputs for the keyboard, and another for the gamepad. Instead of:

```
// Set some flag that we will check later when getting input
m_bPadConnectedArray[0] = pGamepad->isConnected(0);

// Later…
if (m_bPadConnectedArray[0])
{
    // Check gamepad, else check keyboard
}
```

…we could write:

```
// Load an appropriate input set
HInput::GetManager().ClearAllInputsForPlayer(0);
if (pGamepad->isConnected(0))
{
    HInput::GetManager().LoadInputSetForPlayer("GamepadSet", 0);
}
else
{
    HInput::GetManager().LoadInputSetForPlayer("KeyboardSet", 0);
}

// Later…
// Just check the input as normal
```

So long as we have set up our CombinationOperators correctly, we have much more flexibility in configuring the system at runtime. We can also very easily load different sets for different layers, e.g. we might load a gamepad set for player 1 and a keyboard set for player two, while perhaps player 3 is using the wiimote. Once this is configured, the game code doesn't need to keep checking whenever it queries an input. We can also load multiple sets at once.

Integrating the System

Integrating HInput into your game is a simple process. The key points to note are that you must initialize, update and shutdown the system at appropriate times, and provide the required plugins and configuration file.

Initialization

You initialize the system by calling `HInput::System::Initialize()`. This takes optional parameters to allow you to specify a logging callback and a custom time source.

It is recommended that you specify a logging callback, so that you can see the output of the system and any error messages. The signature of the callback is: `void (const std::string&)`, and you should append a newline to each message. An example callback would be:

```
void LogCallback(const std::string& str)
{
	std::cout << str << std::endl;
}
```

If you do not specify a custom time source then the system will use a default Win32 time source, which will incur additional overhead. To specify a custom time source you need to derive a class from `HInput::TimeSource` and pass it to the system. Ensure that the instance you pass to the system remains in scope until after the `HInput::System::Shutdown()` call.

Update

It is recommended that you update the input system at the beginning of each frame, immediately after the clock/timer (if you are using a custom time source), and before any game code queries the system.

Shutdown

Ensure that the system is shutdown before program end. You should only initialize and shutdown the system once per program run. You should not attempt to 'turn it on and off'. If you wish to avoid the overhead of the system temporarily, then simply do not call Update().

Loading and Choosing Input Sets

Once the system is initialized, it won't do anything until you:

1. Activate the player(s) you care about.
2. Load an input set for each player you care about.

HInput supports up to 4 players, but by default only player 1 is active after the system is initialized. If you need to activate more players you can call `HInput::GetManager().SetPlayerActive(player, true)`, where 'player' is a value from the `HInput::PlayerID` enumeration. You can also use this function to deactivate players, which means that their loaded inputs will not be updated.

An active player has a set of loaded inputs. You can load input sets with `HInput::GetManager().LoadInputSetForPlayer("SetName", player)`, where "SetName" is the name you assigned to the set in the editor. This works as follows:

- For every record in the input configuration, in order of first to last:
  - If it matches the set name then:
    - If an input with the same ID (e.g. "Fire") already exists then:
      - Combine the new input with the existing one using the method specified. 'None' means the input will not be loaded, 'And'/'Or' mean the input will be added to the appropriate decision chain. *Do not mix and match 'And' and 'Or' for the same input.*
    - Create it and add it to the player's active inputs

Loading an input set does **not** clear the existing inputs (to allow you to load multiple sets at once). To clear inputs use `HInput::GetManager().ClearAllInputsForPlayer(player).`

Other setup and required files

To do useful work the system requires input services and inputs. These can be registered manually from your game code, if you have written your own (see other tutorials for details on how), or from plug-in components.

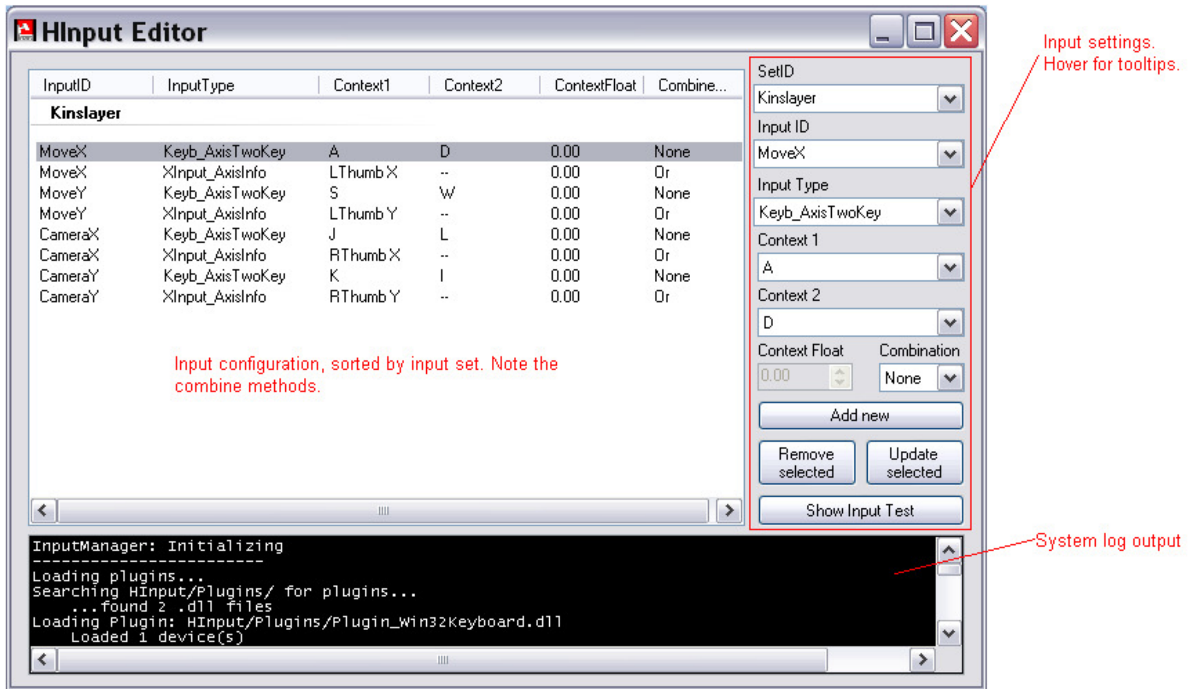
The system will look for plug-in .dll files in `".\HInput\Plugins"` for release builds, or in `".\HInput\Plugins\Debug"` for debug builds.

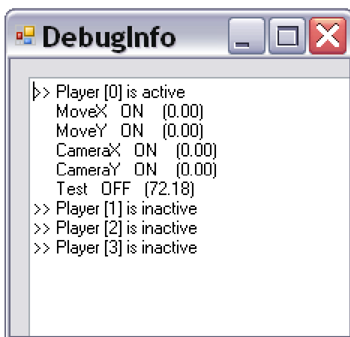
The system will look for its input configuration at `".\HInput\HInputConfig.bin"`. If it is not found, it will load a default set of inputs and save them to a new file with that name. The file uses a custom binary format and is not human-readable. The HInput Editor allows the file to be edited.

Using the Editor

When you run the HInput Editor it will save an input configuration at `".\HInput\HInputConfig.bin"`. You can copy this file to `"[yourgamedir]\HInput\"` and the system will automatically load it. The inputs that are available to select from will be limited by the plugins found (since the plugins contain the metadata the editor uses). Changes are automatically saved.

Most of the fields on the form have tooltips. In the case of context sensitive ones, these will show the descriptions loaded from the input metadata. You can click "Show Input Test" to open a form that will display the result of the InputManager::DumpDebugData() method, i.e. the current state of all loaded inputs.





Writing a Plugin

If you want to write your own custom inputs you can do so easily. It is recommended that you write a new plugin, but you could also register inputs from your client application at runtime without using a plugin. However, they would not be available in the editor.

The best place to start would be the Plugin_Example project, which shows everything you need to know to write your own input plugin. The key components of a plugin are zero or more `InputDeviceService` derived objects, which are registered with the `DeviceServiceManager`, and one or more `InputCreatorBase` derived objects (usually created from the `InputCreator<Input,Modifier>` template), which create instances of your Input derived objects, optionally with an `InputModifier` (the template argument defaults to a null modifier).

Remember that your plugins must be built against the same version of HInput that your game is built against, or you will get heap corruption and other nasty things.

Writing an Editor

The system is designed to allow any client to write a rich GUI to view and modify the system configuration. This way, you have total control over the available inputs, and can let your players define their own inputs if you like.

The primary example of this is the HInput Editor, which operates on the system through the HInputCLIBridge component. An example Irrlicht GUI is also included in the SDK.

Metadata about available inputs is available from the MetaDataStore `[GetManager().GetMetaDataStore()]`, and the configuration is available from the DataStore `[GetManager().GetDataStore()]`.