Game Engine Architecture

Chapter 9

Human Interface Devices

Overview

- Types of input devices
- Interfacing with a HID
- Types of input
- Types of output
- Game engine HID systems

HID

- A human interface device or HID is a type of computer device usually used by humans that takes input from humans and gives output to humans.
- The term "HID" most commonly refers to the USB-HID specification.
- The term was coined by Mike Van Flandern of Microsoft when he proposed that the USB committee create a Human Input Device class working group
- The working group was renamed as the Human Interface Device class at the suggestion of Tom Schmidt of DEC because the proposed standard supported bi-directional communication.

Types of HIDs

- Consoles typically come with a Joypad device
- Computers often use the keyboard and mouse
- Arcade games have very specialized input
- Many specialized devices
  - Guitar Hero
  - Dance Dance Revolution
- Modifiers on existing devices
  - Steering wheel for WiiMote
- Nintendo Switch's hybrid console
  - Can be used as either a home console or portable device
- Azure Kinect
  - The new iteration of Kinect technology designed primarily for enterprise software and artificial intelligence usage. It is designed around the Microsoft Azure cloud platform, and is meant to "leverage the richness of Azure AI to dramatically improve insights and operations

Types of HIDs

Interfacing to a HID

- Depends on the device
- Polling – reading the state once per iteration
  - Hardware register
  - Memory mapped I/O port
  - Using a higher level function
- XInput on the XBox 360 is a good example
  - Call XInputGetState() which returns a XINPUT_STATE struct

Interfacing

- Interrupts
  - Triggered by a state change in the hardware
  - Temporarily interrupts the CPU to run a small piece of code
    - Interrupt Service Routine (ISR)
  - ISRs typically just read the change and store the data
- Wireless
  - Many of the wireless controllers use Bluetooth to communicate
  - Software requests HID state
  - Usually handled by a separate thread
  - Looks like polling to the application developer

Type of input

- Almost every HID has at least a few *digital buttons* Digital Buttons
  - Have two states
  - Pressed represented by a 1
  - Not pressed represented as a 0
  - Depends on the hardware developer
- Electrical engineers speak of a circuit containing a switch as being *closed* (meaning electricity is flowing through the circuit) or *open* (no electricity is flowing)
- Often the button state is combine into one struct

wButtons

```
#define XINPUT_GAMEPAD_DPAD_UP 0x0001 // bit 0
#define XINPUT_GAMEPAD_DPAD_DOWN 0x0002 // bit 1
#define XINPUT_GAMEPAD_DPAD_LEFT 0x0004 // bit 2
#define XINPUT_GAMEPAD_DPAD_RIGHT 0x0008 // bit 3
#define XINPUT_GAMEPAD_START 0x0010 // bit 4
#define XINPUT_GAMEPAD_BACK 0x0020 // bit 5
#define XINPUT_GAMEPAD_LEFT_THUMB 0x0040 // bit 6
#define XINPUT_GAMEPAD_RIGHT_THUMB 0x0080 // bit 7
#define XINPUT_GAMEPAD_LEFT_SHOULDER 0x0100 // bit 8
#define XINPUT_GAMEPAD_RIGHT_SHOULDER 0x0200 // bit 9
#define XINPUT_GAMEPAD_A 0x1000 // bit 12
#define XINPUT_GAMEPAD_B 0x2000 // bit 13
#define XINPUT_GAMEPAD_X 0x4000 // bit 14
#define XINPUT_GAMEPAD_Y 0x8000 // bit 15
```

Xinput_gamepad

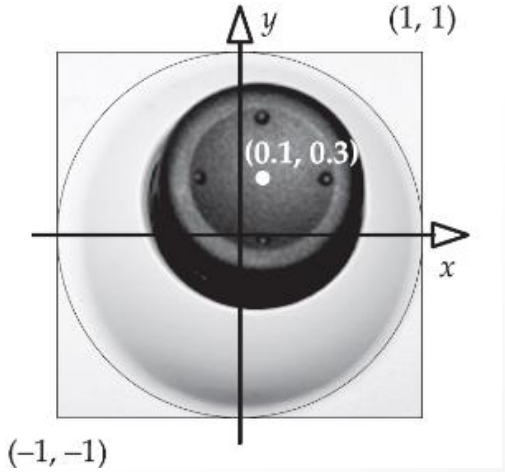
```
typedef struct _XINPUT_GAMEPAD {
WORD wButtons;
BYTE bLeftTrigger;
BYTE bRightTrigger;
SHORT sThumbLX
SHORT sThumbLY
SHORT sThumbRX
SHORT sThumbRY
}
```

- xButtons contains the state of all the buttons and you apply masks to get their current state

```
bool IsButtonADown(const XINPUT_GAMEPAD& pad)
{
// Mask off all bits but bit 12 (the A button).
return ((pad.wButtons & XINPUT_GAMEPAD_A) != 0);
}
```

Analog axes and buttons

- An *analog input* is one that can take on a range of values (rather than just 0 or 1).
- Unlike buttons, joysticks have a range of values
- Analog inputs are sometimes called *analog axes*, or just *axes*
- They aren't truly analog because they are digitized
  - Range from -32,768 to +32,767
  - or -1 to +1



_XINPUT_GAMEPAD

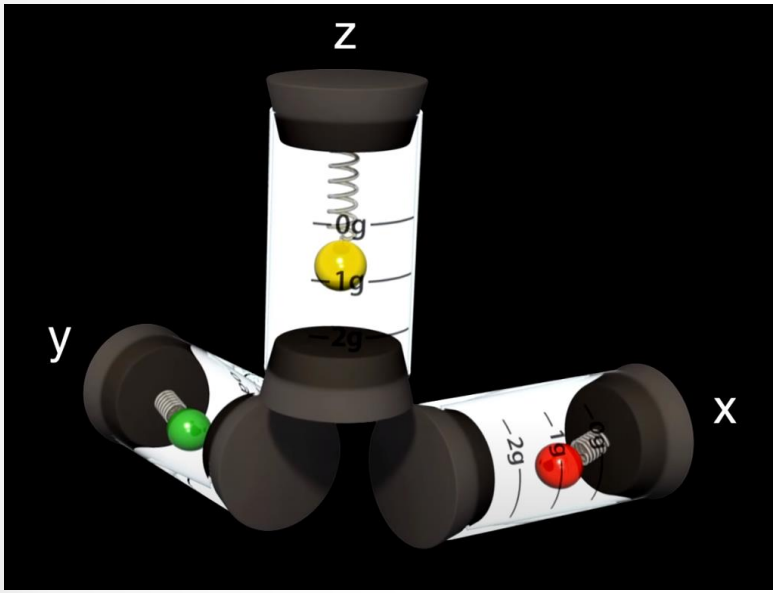
- XINPUT_GAMEPAD: Microsoft represents the deflections of the left and right thumb sticks on the Xbox 360 gamepad using 16-bit signed integers (sThumbLX and sThumbLY for the left stick and sThumbRX and sThumbRY for the right).
- these values range from -32,768 (left or down) to 32,767 (right or up).
- To represent the positions of the left and right shoulder triggers, Microsoft chose to use eight-bit unsigned integers (bLeftTrigger and bRightTrigger, respectively).
- These input values range from 0 (not pressed) to 255 (fully pressed).
- Different game machines use different digital representions for their analog axes.

```
typedef struct
_XINPUT_GAMEPAD
{
WORD wButtons;
// 8-bit unsigned
BYTE bLeftTrigger;
BYTE bRightTrigger;
// 16-bit signed
SHORT sThumbLX;
SHORT sThumbLY;
SHORT sThumbRX;
SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

Relative axes

- Most of the time the position of the trigger, joystick, or thumb stick is absolute – relative to a 0,0 point
- Some input devices provide information in a relative format
  - They return 0 if the device has not moved from its past position
- Examples include mice, mice wheels, and track balls

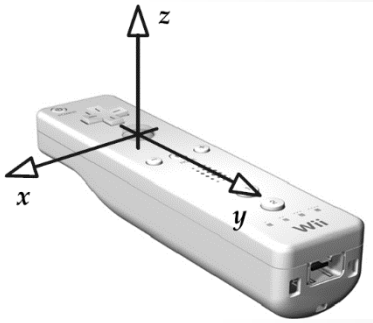
Accelerometers



- Newer controllers have accelerometers built in
- These provide information about g-force in three directions (1 g ~ 9.8 m/s2).
- These are *relative* analog inputs, much like a mouse's two-dimensional axes.
- When the controller is not accelerating these inputs are zero.
- When the controller is accelerating, they measure the acceleration up to 3 g along each axis, quantized into three signed eight-bit integers, one for each of x, y and z.

WiiMote and DualShock

- One issue is that the accelerometers don't give you orientation information
- One way to do it is to remember that gravity has 1g of force in the downward direction
- If the player is holding the device, is not holding still, the accelerometer inputs will include this acceleration in their values, invalidating our math.
- The z-axis of the accelerometer has been calibrated to account for gravity, but the other two axes have not!
- you can calibrate the device based on which axis is experiencing 1g of force

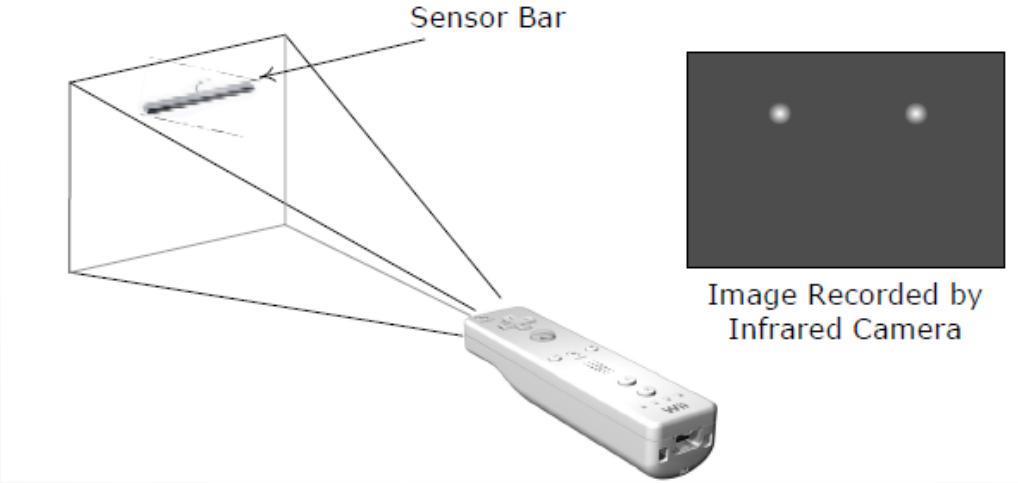


Dual Sense

- The DualSense is the PlayStation 5's controller and was unveiled on April 7, 2020.
- It is based on the DualShock 4 controller
- It incorporates a more ergonomic design that is somewhat bigger and rounder than the DualShock 4

Cameras

- The Wii uses a very low resolution IR camera to read the location of the WiiMote
  - WiiMote has two IR LEDs
- The distance between the two dots, the size of the dots, and the relative position of the dots provides a lot of information about where the mote is pointing
- Other examples include the Sony EyeToy and Xbox Kinect



Camera input

Types of output

- Rumble
  - Vibrations are usually produced by one or two motors with an unbalanced weight at various speed.
  - The game can turn these motors on and off, and control the speed to create different feels
- Force feedback
  - A motor attached to the HID that resists input from the user
  - In arcade driving games, where the steering wheel resists the player's attempt to turn it
  - The Sixaxis was succeeded by the DualShock 3, an updated version of the controller that, like the DualShock and DualShock 2 controllers, incorporates haptic technology – also known as force feedback
- Audio
  - WiiMote has has low-quality speaker
  - DualShock 4, Xbox 360 and Xbox One controllers have headphone jack
- Others
  - Some have LEDs that can be controlled via software

Game Engine HID Systems

- Most game engines don't use "raw" HID inputs directly
- Most engines introduce at least one additional level of indirection between the HID and the game in order to abstract HID inputs in various ways
- Most game engine massage the data coming in from the controllers
- For example, a button-mapping table might be used to translate raw button inputs into logical game actions, so that human players can reassign the buttons' functions as they see fit.

Typical requirements

- Dead zones
- Analog signal filtering
- Event detection
- Detection of sequences and chords
- Gesture detection
- Management of multiple HIDs
- Multiplatform HID support
- Controller input re-mapping
- Context sensitive inputs
- The ability to disable certain inputs

Dead zones

- HIDs are analog devices by nature
- A joystick, thumb stick, shoulder trigger, or any other analog axis produces input values that range between a predefined minimum and maximum value, $I_{\min}$ and I_{max}.
- When the control is not being touched, we would expect it to produce a steady and clear "undisturbed" value I_0.
- I_0 is usually numerically equal to zero, and it either lies halfway between I_{min} and I_{max} for a centered, two-way control like a joystick axis, or it coincides with I_{min} for a one-way control like a trigger
- the voltage produced by the device is noisy , the actual inputs we observe may fluctuate slightly around I_0
- The most common solution to this problem is to introduce a small *dead zone* around I_0
- The dead zone might be defined as $[I_0 - \delta, I_0 + \delta]$ for a joystick or $[I_0, I_0 + \delta]$ for a trigger
- Dead zones allow the developer to clamp values within a small range $[I_0, I_0 + \delta]$to the value I_0 for a trigger

Analog signal filtering

- Even with dead-zone clamping, the noise from the analog input can cause the motion to be jerky
- For this reason, many games *filter* the raw inputs coming from the HID.
- Noise is usually high-frequency so we can isolate user input by using a low-pass filter

$$f(t) = (1 - a)f(t - \Delta t) + au(t)$$

filtered

unfiltered

$$a = \frac{\Delta t}{RC + \Delta t}$$

Low-pass filter

```
F32 lowPassFilter(F32 unfilteredInput,
                  F32 lastFramesFilteredInput,
                  F32 rc, F32 dt)
{
    F32 a= dt / (rc +dt);
    return (1-a) * lastFramesFilteredInput
           + a * unfilteredInput;
}
```

Moving average

- An equally interesting way of filtering the data is to use a moving average
- This can be done using a circular queue that maintains the last *n* values
- You can also keep the running sum by adding the new value and subtracting the one it is replacing in the queue

```
template< typename TYPE, int SIZE >
class MovingAverage
{
TYPE m_samples[SIZE];
TYPE m_sum;
U32 m_curSample;
U32 m_sampleCount;
public:
MovingAverage() :
m_sum(static_cast<TYPE>(0)),
m_curSample(0),
m_sampleCount(0)
{
}
void addSample(TYPE data)
{
if (m_sampleCount == SIZE)
{
m_sum -= m_samples[m_curSample];
}
else
{
m_sampleCount++;
}
m_samples[m_curSample] = data;
m_sum += data;
m_curSample++;
if (m_curSample >= SIZE)
{
m_curSample = 0;
}
}
F32 getCurrentAverage() const
{
if (m_sampleCount != 0)
{
return static_cast<F32>(m_sum)
/ static_cast<F32>(m_sampleCount);
}
return 0.0f;
}
};
```

Input event detection

- The easiest way to detect a change to states is to keep the previous state and XOR it with the current state
  - XOR produces a 1 when the state has changed
- In `m_buttonDowns` and `m_buttonUps` , the bit corresponding to each button will be 0 if the event has not occurred in this frame and 1 if it has.
- We can then mask out the up events and the down events by using AND
  - ANDing the original state with the changed state gives us the DOWNs
  - NANDing gives us the UPs

```
class ButtonState
{
U32 m_buttonStates; // current frame's button states
U32 m_prevButtonStates; // previous frame's states
U32 m_buttonDowns; // 1 = button pressed this frame
U32 m_buttonUps; // 1 = button released this  frame
void DetectButtonUpDownEvents()
{
// Assuming that m_buttonStates and
// m_prevButtonStates are valid, generate
// m_buttonDowns and m_buttonUps.

// First determine which bits have changed via XOR.
U32 buttonChanges = m_buttonStates ^ m_prevButtonStates;

// Now use AND to mask off only the bits that are DOWN.
m_buttonDowns = buttonChanges & m_buttonStates;

// Use AND-NOT to mask off only the bits that are UP.
m_buttonUps = buttonChanges & (~m_buttonStates);
}
// ...
};
```

Chords

- Chords are combinations of button presses
  - A+B at the same time
- Easy to create masks for chords, but there are some subtle issues
  - Users aren't perfect in the timing of the presses
  - The code must be robust to not triggering the non-chord events too
- Lots of ways to resolve these
  - Design button inputs so the chord does the individual actions plus something else
  - Introduce a delay in processing individual inputs so a chord can form
  - Wait for button release to start an event
  - Start a single button move right away and override it with a chord move

Sequences and gestures

- The idea of introducing a delay between when a button actually goes down and when it really "counts" as down is a special case of *gesture detection*.
- A gesture is a sequence of actions performed via a HID by the human player over a period of time.
- Sequences are usually done by storing the input events and tagging them with time
- When the next button is pressed, determine if the time for the sequence has expired or if the sequence is invalid
- Can be used for
  - Rapid button tapping
  - Button sequences
  - Thumb stick rotations

Rapid Button Tapping

- Many games require the user to tap a button rapidly in order to perform an action.
- The frequency of the button presses may or may not translate into some quantity in the game, such as the speed with which the player character runs or performs some other action
- The frequency *f* is then just the inverse of the time interval between presses
  - $\Delta T = T_{cur} - T_{last}$
  - $f = \frac{1}{\Delta T}$
- To implement a minimum valid frequency, we simply check *f* against the minimum frequency f_{min}
- or we can just check ΔT against the maximum period $\Delta T_{max} = 1/ f_{min}$ directly

ButtonTapDetector

```
class ButtonTapDetector
{
U32 m_buttonMask; // which button to observe (bit  mask)
F32 m_dtMax; // max allowed time between  presses
F32 m_tLast; // last button-down event, in seconds
public:
// Construct an object that detects rapid tapping of
// the given button (identified by an index).
ButtonTapDetector(U32 buttonId, F32 dtMax) :
m_buttonMask(1U << buttonId),
m_dtMax(dtMax),
m_tLast(CurrentTime() - dtMax) // start out invalid
{
}
// Call this at any time to query whether or not
// the gesture is currently being performed.
bool IsGestureValid() const
{
F32 t = CurrentTime();
F32 dt = t - m_tLast;
return (dt < m_dtMax);
}
// Call this once per frame.
void Update()
{
if (ButtonsJustWentDown(m_buttonMask))
{
m_tLast = CurrentTime();
}
}
};
```

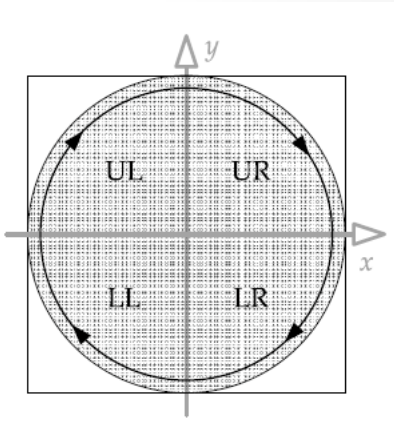
Multibutton Sequence

```
class ButtonSequenceDetector
{
U32* m_aButtonIds; // sequence of buttons to watch for
U32 m_buttonCount; // number of buttons in sequence
F32 m_dtMax; // max time for entire sequence
U32 m_iButton; // next button to watch for in seq.
F32 m_tStart; // start time of sequence, in seconds
public:
// Construct an object that detects the given button
// sequence. When the sequence is successfully
// detected, the given event is broadcast so that the
// rest of the game can respond in an appropriate way.
ButtonSequenceDetector(U32* aButtonIds,
U32 buttonCount,
F32 dtMax,
EventId eventIdToSend) :
m_aButtonIds(aButtonIds),
m_buttonCount(buttonCount),
m_dtMax(dtMax),
m_eventId(eventIdToSend), // event to send when
complete
m_iButton(0), // start of sequence
m_tStart(0) // initial value
// irrelevant
{
}
```

```
// Call this once per frame.
void Update()
{
ASSERT(m_iButton < m_buttonCount);
// Determine which button we're expecting next, as
// a bitmask (shift a 1 up to the correct bit index).
U32 buttonMask = (1U << m_aButtonId[m_iButton]);
// If any button OTHER than the expected button just went down,
invalidate the sequence. (Use the bitwise NOT operator to check for
all other buttons.)
if (ButtonsJustWentDown(~buttonMask))
{
m_iButton = 0; // reset
}
// Otherwise, if the expected button just went
// down, check dt and update our state appropriately.
else if (ButtonsJustWentDown(buttonMask))
{
if (m_iButton == 0)
{
// This is the first button in the sequence.
m_tStart = CurrentTime();
m_iButton++; // advance to next button
}
else
{
F32 dt = CurrentTime() - m_tStart;
if (dt < m_dtMax)
{
// Sequence is still valid.
m_iButton++; // advance to next button Is the sequence complete?
if (m_iButton == m_buttonCount)
{
BroadcastEvent(m_eventId);
m_iButton = 0; // reset
}
}
else
{
// Sorry, not fast enough.
m_iButton = 0; // reset
}}}}};
```

Thumb Stick Rotation

- How to detect when the player is rotating the left thumb stick in a clockwise circle.
- Divide the two-dimensional range of possible stick positions into quadrants
- In a clockwise rotation, the stick passes through the upper-left quadrant, then the upper-right, then the lower-right and finally the lower-left.
- We can treat each of these cases like a button press and detect a full rotation with a slightly modified version of the sequence detection code



Multiple HIDs

- This just involves mapping the correct input to the correct player
- Also involves detecting HID changes due to serious events
  - Controller unplugged
  - Batteries dying
  - Player being assaulted

Cross platform HIDs

- Two main ways to handle this
  - Conditional compilations – YUCK messy code

```
#if TARGET_XBOX360
if (ButtonsJustWentDown(XB360_BUTTONMASK_A))
#elif TARGET_PS3
if (ButtonsJustWentDown(PS3_BUTTONMASK_TRIANGLE))
#elif TARGET_WII
if (ButtonsJustWentDown(WII_BUTTONMASK_A))
#endif
{
// do something...
}
```

- Hardware abstraction layer
  - Create your own input classes and add a translation layer
  - Certainly superior because it is easy to add new HIDs

Hardware abstraction layer

```
enum AbstractControlIndex
{
// Start and back buttons
AINDEX_START, // Xbox 360 Start, PS3 Start
AINDEX_BACK_SELECT, // Xbox 360 Back, PS3 Select
// Left D-pad
AINDEX_LPAD_DOWN,
AINDEX_LPAD_UP,
AINDEX_LPAD_LEFT,
AINDEX_LPAD_RIGHT,
// Right "pad" of four buttons
AINDEX_RPAD_DOWN, // Xbox 360 A, PS3 X
AINDEX_RPAD_UP, // Xbox 360 Y, PS3 Triangle
AINDEX_RPAD_LEFT, // Xbox 360 X, PS3 Square
AINDEX_RPAD_RIGHT, // Xbox 360 B, PS3 Circle
// Left and right thumb stick buttons
AINDEX_LSTICK_BUTTON, // Xbox 360 LThumb, PS3 L3, Xbox white
AINDEX_RSTICK_BUTTON, // Xbox 360 RThumb, PS3 R3, Xbox black
// Left and right shoulder buttons
AINDEX_LSHOULDER, // Xbox 360 L shoulder, PS3 L1
AINDEX_RSHOULDER, // Xbox 360 R shoulder, PS3 R1
// Left thumb stick axes
AINDEX_LSTICK_X,
AINDEX_LSTICK_Y,
// Right thumb stick axes
AINDEX_RSTICK_X,
AINDEX_RSTICK_Y,
// Left and right trigger axes
AINDEX_LTRIGGER, // Xbox 360 -Z, PS3 L2
AINDEX_RTRIGGER, // Xbox 360 +Z, PS3 R2
};
```

Input re-mapping

- We can provide a very nice feature by adding in a few classes that create a level of indirection – input remapping
- If we give names to events that have nothing to do with the input that triggers it, then we simple create an input to event lookup table
- Requires us to normalize the input a bit
  - Think about keyboard versus mouse for aircraft control

Input values

- Different controls produce different kinds of inputs.
- Analog axes may produce values ranging from -32,768 to 32,767, or from 0 to 255, or some other range
- A reasonable set of classes for a standard console joypad and their normalized input values:

  - *Digital buttons*. States are packed into a 32-bit word, one bit per button.
  - *Unidirectional absolute axes (e.g., triggers, analog buttons)*. Produce floating-point input values in the range [0, 1].
  - *Bidirectional absolute axes (e.g., joy sticks)*. Produce floating-point input values in the range [-1, 1].
  - *Relative axes (e.g., mouse axes, wheels, track balls)*. Produce floating-point input values in the range [-1, 1], where 1 represents the maximum relative offset possible within a single game frame (i.e., during a period of 1/30 or 1/60 of a second).

Context sensitive controls

- Often the controls will trigger different events based on the situation
  - The “use” button is a classic case
- One way to implement this is to use a state machine
  - In state “walking” L1 means fire primary weapon
  - In state “driving” L1 means turbo
- When mixed with user re-mapping the system can become quite complex

Disabling inputs

- One simple, but Draconian approach is to introduce a mask
  - When a control is disabled, set the bit to 0
  - The mask is then ANDed with the control input before further process
- Make sure the bit gets set back to 1 again otherwise you loose control forever
- Sometimes you want to disable it for a particular set of actions, but other systems might want to see it...

In practice

- It takes a serious amount of time and effort to build an effective HID system.
- It is super important because it radically affects game play

Unity HID Support

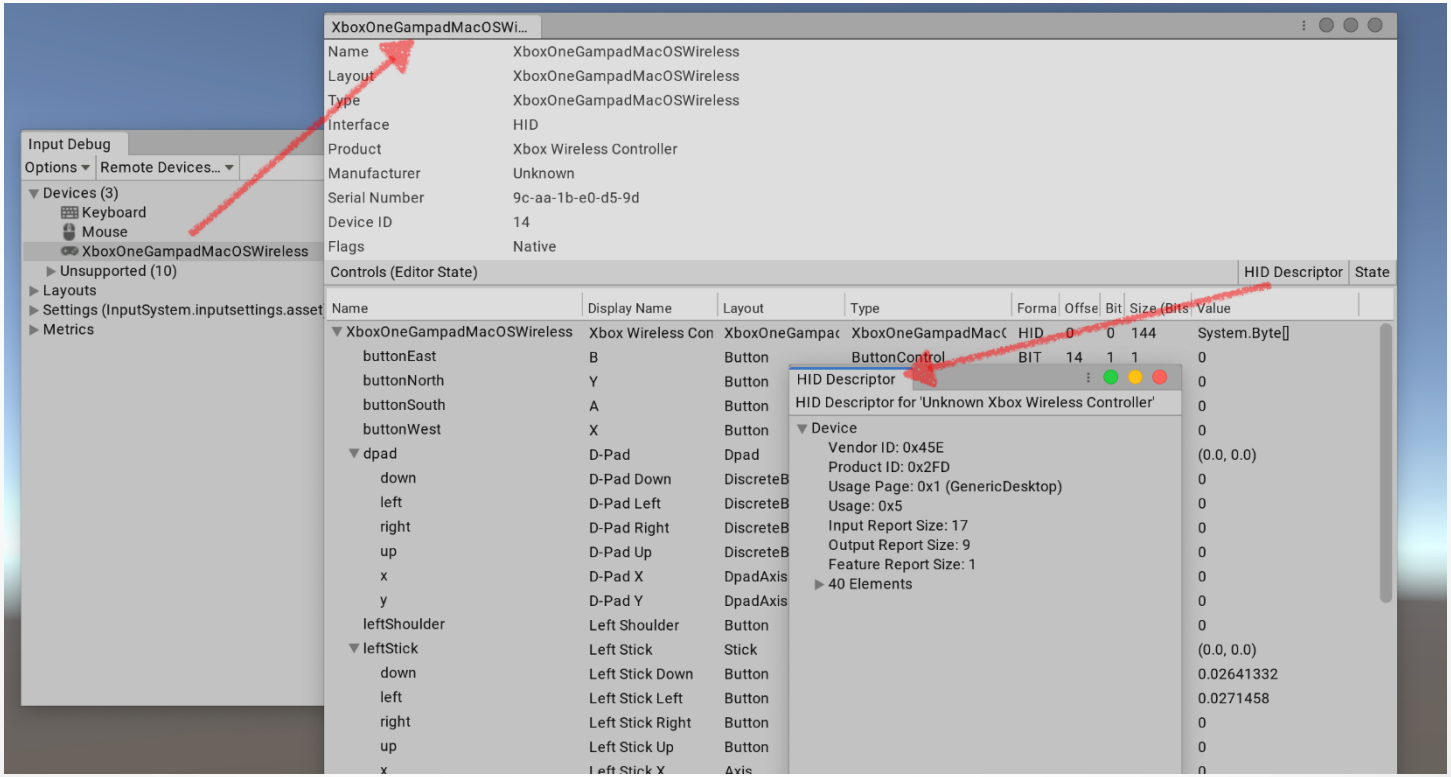
- Human Interface Device (HID) is a specification to describe peripheral user input devices connected to computers via USB or Bluetooth.
- HID is commonly used to implement devices such as gamepads, joysticks, or racing wheels.
- The Input System directly supports HID (connected via both USB and Bluetooth) on Windows, MacOS, and the Universal Windows Platform (UWP).
- The system might support HID on other platforms, but not deliver input through HID-specific APIs.
- For example, on Linux, the system supports gamepad and joystick HIDs through SDL, but doesn't support other HIDs.

HID Descriptor

- Every HID comes with a device descriptor.
- Install the Input System
- Window→ Analysis → Input Debugger
- Double click on Mouse or Keyboard
- Keyboard and Mouse have "Raw Input" interfaces
- Regular game control has "XInput" interface
- If you have an HID, you'll see HID as an interface
- To browse through the descriptor of an HID from the Input Debugger, click the **HID Descriptor** button in the device debugger window(HID Descriptor appears if you have an HID interface device!)
- To specify the type of the device, the HID descriptor reports entry numbers in the HID usage tables, and a list of all controls on the device, along with their data ranges and usages.

HID Layout

- The Input System handles HIDs in one of two ways:
  1. The system has a known layout for the specific HID.
  2. If the system does not have a known layout, it auto-generates one for the HID.



Monitoring Devices

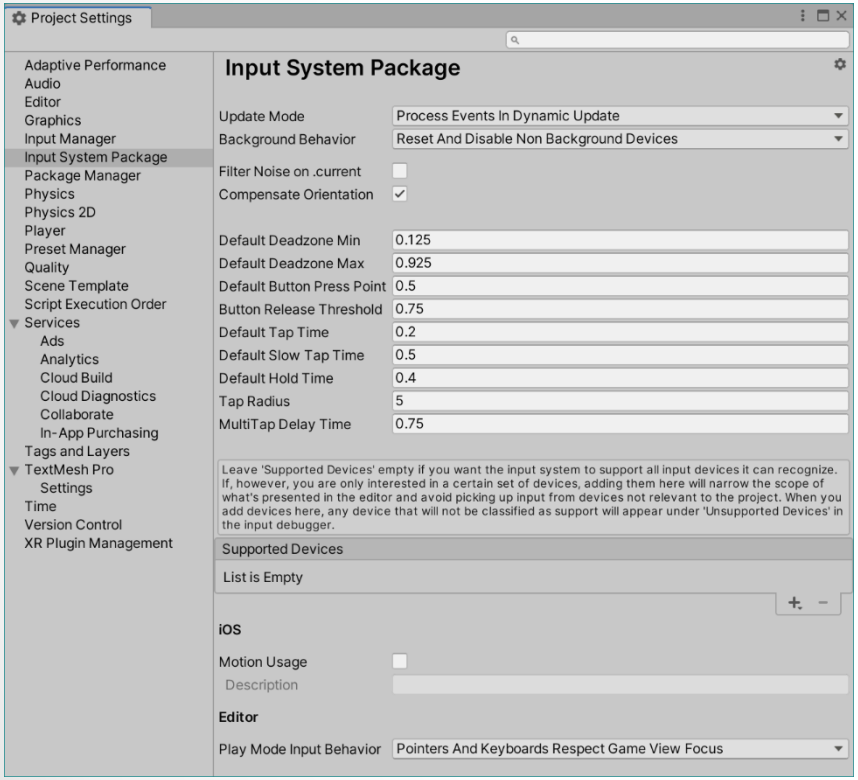
- To be notified when new Devices are added or existing Devices are removed, use InputSystem.onDeviceChange.

```
InputSystem.onDeviceChange +=
(device, change) =>
  {
      switch (change)
      {
          case
InputDeviceChange.Added:
Debug.Log("New device added: "
+ device);
              break;

          case
InputDeviceChange.Removed:
Debug.Log("Device removed: " +
device);
              break;
      }
  };
```

HID output

- HIDs can support output
  - for example, to toggle lights or force feedback motors on a gamepad
- Unity controls output by sending HID Output Report commands to a Device.
- Output reports use Device-specific data formats.
- To use HID Output Reports, call InputDevice.ExecuteCommand to send a command struct with the typeStatic property set as "HIDO" to a Device.
- The command struct contains the Device-specific data sent out to the HID.

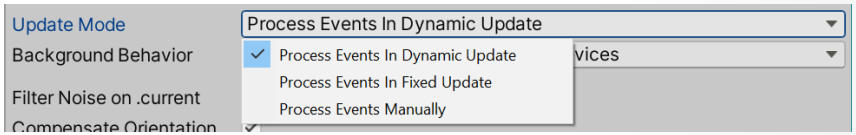
Input Settings



- To configure the Input System individually for each project, go to **Edit > Project Settings… > Input System Package** from Unity's main menu.
- The Input System stores input settings in Assets.
- If your Project doesn't contain an input settings Asset, click **Create settings asset** in the Settings window to create one.
- If your Project contains multiple settings Assets, use the gear menu in the top-right corner of the window to choose which one to use.
- You can also use this menu to create additional settings Assets.

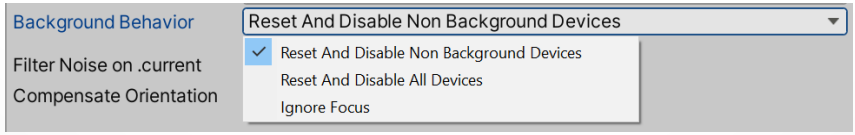
Update Mode

Type	Description
Process Events In Dynamic Update	The Input System processes events at irregular intervals determined by the current framerate.
Process Events In Fixed Update	The Input System processes events at fixed-length intervals. This corresponds to how MonoBehaviour.FixedUpdate operates. The length of each interval is determined by Time.fixedDeltaTime.
Process Events Manually	The Input System does not process events automatically. Instead, it processes them whenever you call InputSystem.Update().



Background Behavior

- what happens when focus is lost or gained and how input behaves while the application is not in the foreground.
- This setting is only relevant when "Run In Background" is enabled in the Player Settings for the project.
- Note that in the editor, "Run In Background" is considered to always be enabled as the player loop will be kept running regardless of whether a Game View is focused or not.
- When a Device is reset, Actions bound to Controls on the device will be cancelled. This ensures, for example, that a user-controlled character in your game doesn't continue to move when focus is lost while the user is pressing one of the W, A, S or D keys.



Background Behavior

Setting	Description
Reset And Disable Non Background Devices	When focus is lost, perform a soft reset on all Devices that are not marked as canRunInBackground and also subsequently disable them. Does not affect Devices marked as being able to run in the background. When focus is regained, re-enable any Device that has been disabled and also issue a sync request on these Devices in order to update their current state. If a Device is issued a sync request and does not respond to it, soft-reset the Device. This is the default setting.
Reset And Disable All Devices	When focus is lost, perform a soft reset on all Devices and also subsequently disable them. When focus is regained, re-enable all Devices and also issue a sync request on each Device in order to update it to its current state. If a device does not respond to the sync request, soft-reset it.
Ignore Focus	Do nothing when focus is lost. When focus is regained, issue a sync request on all Devices.

Filter Noise On Current

- Whenever there is input on a Device, the system makes the respective Device .current.
- For example, if a Gamepad receives new input, Gamepad.current is assigned to that gamepad.
- Some Devices have noise in their input, and receive input even if nothing is interacting with them.
- If the user is “not” using the Xbox controller/the PS4 controller, they still push themselves to the front continuously and makes themselves current.
- This setting is disabled by default, and it's only relevant for apps that use the .current properties (such as Gamepad.current) in the API.

Compensate Orientation

- If this setting is enabled, rotation values reported by sensors are rotated around the Z axis as follows:
  - Screen orientation — Effect on rotation values
  - ScreenOrientation.Portrait — Values remain unchanged
  - ScreenOrientation.PortraitUpsideDown — Values rotate by 180 degrees.
  - ScreenOrientation.LandscapeLeft — Values rotate by 90 degrees.
  - ScreenOrientation.LandscapeRight — Values rotate by 270 degrees.

- This setting affects the following sensors:
  - Gyroscope
  - GravitySensor
  - AttitudeSensor
  - Accelerometer
  - LinearAccelerationSensor

Default value properties

- Default Deadzone Min
  - The default minimum value for Stick Deadzone or Axis Deadzone processors when no min value is explicitly set on the processor.
- Default Deadzone Max
  - The default maximum value for Stick Deadzone or Axis Deadzone processors when no max value is explicitly set on the processor.
- Default Button Press Point
  - The default press point for Button Controls, and for various Interactions. For button Controls which have analog physics inputs (such as triggers on a gamepad), this configures how far they need to be held down for the system to consider them pressed.
- Default Tap Time
  - Default duration for Tap and MultiTap Interactions. Also used by by touchscreen Devices to distinguish taps from to new touches.
- Default Slow Tap Time
  - Default duration for SlowTap Interactions.
- Default Hold Time
  - Default duration for Hold Interactions.
- Tap Radius
  - Maximum distance between two finger taps on a touchscreen Device for the system to consider this a tap of the same touch (as opposed to a new touch).
- Multi Tap Delay Time
  - Default delay between taps for MultiTap Interactions. Also used by touchscreen Devices to count multi-taps (See TouchControl.tapCount).

Supported Devices

- A Project usually supports a known set of input methods. For example, a mobile app might support only touch, and a console application might support only gamepads. A cross-platform application might support gamepads, mouse, and keyboard, but might not require XR Device support.

- To narrow the options that the Editor UI presents to you, and to avoid creating input Devices and consuming input that your application won't use, you can restrict the set of supported Devices on a per-project basis.

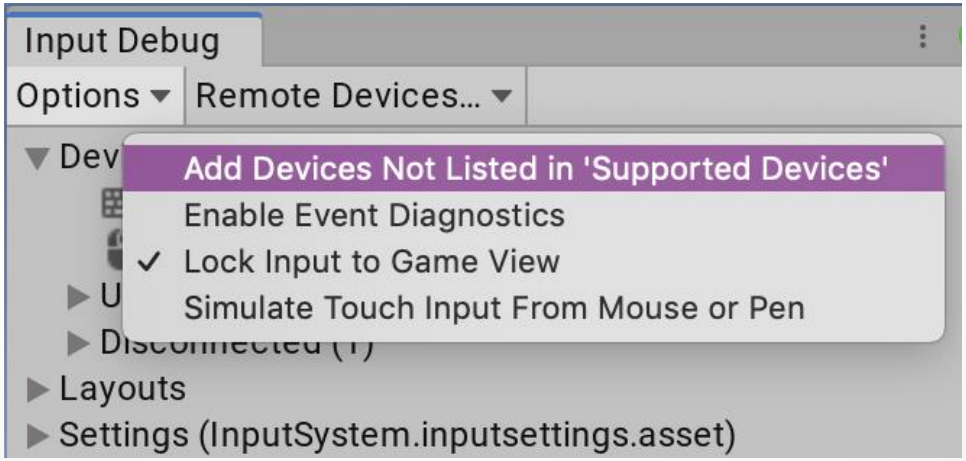
- If Supported Devices is empty, no restrictions apply, which means that the Input System adds any Device that Unity recognizes and processes input for it.

- However, if Support Devices contains one or more entries, the Input System only adds Devices that are of one of the listed types.

Override in Editor

- In the Editor, you might want to use input Devices that the application doesn't support.
  - For example, you might want to use a tablet in the Editor even if your application only supports gamepads.

- To force the Editor to add all locally available Devices, even if they're not in the list of Supported Devices,
  - open the Input Debugger (menu: Window > Analysis > Input Debugger), and select Options > Add Devices Not Listed in 'Supported Devices'.



Platform-specific settings

- iOS/tvOS
- Motion Usage
  - Governs access to the pedometer (or step-counter) on the device. If enabled, the Description string supplied in the settings will be added to the application's Info.plist

Play Mode Input Behavior

- Determines how input is handled in the Editor when in play mode.
- Unlike in players, in the Editor Unity (and its input backends) will keep running for as long as the Editor is active regardless of whether a Game View is focused or not.
- Pointers And Keyboards Respect Game View Focus: This setting is the default.
  - Only Pointer and Keyboard Devices require the Game View to be focused. Other Devices will route their input into the application regardless of Game View focus.

Supported Input Devices

Device	Windows	Mac	Linux	UWP	Android	iOS	tvOS	Xbox(3)	PS4(3)	Switch(3)	WebGL
Mouse	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	Yes
Keyboard	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	No	Yes
Pen	Yes	No (1)	No	Yes	Yes	Yes	No	No	No	No	No
Touchscreen	Yes	No	No	Yes	Yes	Yes	Yes(4)	No	No	No	Yes
Sensors	No	No	No	No	Yes	Yes	No	No	No	No	Yes(5)
Joystick	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	Yes

Pointers

- Pointer Devices are defined as InputDevices that track positions on a 2D surface.
- The Input System supports three types of pointers:
  - Touch
  - Mouse
  - Pen
- Each of these types implements a common set of Controls.

Pointer Controls

Control	Type	Description
position	Vector2Control	The current pointer coordinates in window space.
delta	Vector2Control	Provides motion delta in pixels accumulated (summed) over the duration of the current frame/update. Resets to (0,0) each frame. Note that the resolution of deltas depends on the specific hardware and/or platform.
press	ButtonControl	Whether the pointer or its primary button is pressed down.
pressure	AxisControl	The pressure applied with the pointer while in contact with the pointer surface. This value is normalized. This is only relevant for pressure-sensitive devices, such as tablets and some touch screens.
radius	Vector2Control	The size of the area where the finger touches the surface. This is only relevant for touch input.

Keyboard support

- The Keyboard class defines a Device with a set of key Controls defined by the Key enumeration.
- The location of individual keys is agnostic to keyboard layout. This means that, for example, the A key is always the key to the right of the Caps Lock key.
- To query which (if any) character is generated by a given key, use the key Control's displayName property.

The old input manager

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour
{

    public Rigidbody rb;
    public float moveSpeed = 5f;

    Vector3 moveDirection = Vector3.zero;

    void Start()
    {
        Debug.Log(Time.fixedDeltaTime);
    }

    void Update()
    {
        float moveX = Input.GetAxis("Horizontal");
        float moveY = Input.GetAxis("Vertical");

        moveDirection = new Vector2(moveX, moveY).normalized;

    }

private void FixedUpdate()
    {

        rb.velocity = new Vector3(moveDirection.x * moveSpeed, moveDirection.y * moveSpeed, moveDirection.z * moveSpeed);
    }

}
```

The old input manager

- Go to Project Setting → Input Manager
- Expand the Axes and check out Horizontal/Vertical inputs
- You had to use verbatims

```
float moveX = Input.GetAxis("Horizontal");
float moveY = Input.GetAxis("Vertical");
```

Getting input directly from an Input Device Using New Input System Manager

- The quickest way to get started in script is to read the current state directly from Input Devices.
- For example, the following code gets the gamepad/keyboard/mouse that a player last used, and reads its current state:

```
var keboard = Keyboard.current;
        if (keboard == null)
            return; // No keyboard connected.


        if (Keyboard.current[Key.A].wasPressedThisFrame)
        {

            moveX = -10;
            Debug.Log(Keyboard.current[Key.A].ReadValue()); // +1.

        }
```

Input Actions

- Input Actions are designed to separate the logical meaning of an input from the physical means of input (that is, activity on an input device) that generate the input.

Input Actions

- Instead of writing input code like this:

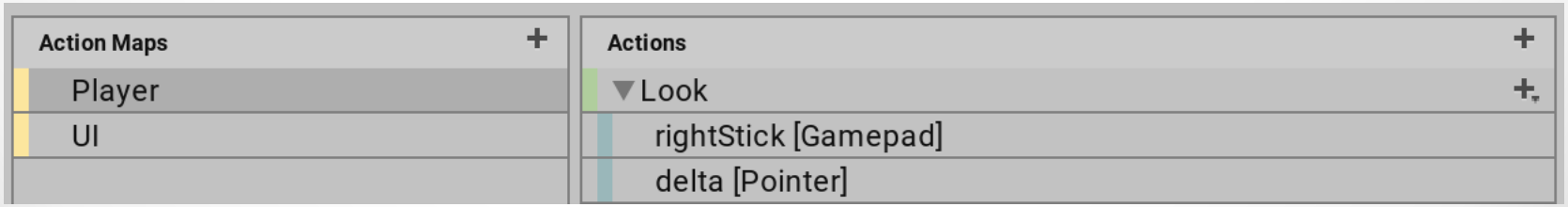
```
var look = new Vector2();

var gamepad =
Gamepad.current;
    if (gamepad != null)
        look =
gamepad.rightStick.ReadValu
e();

var mouse = Mouse.current;
    if (mouse != null)
        look =
mouse.delta.ReadValue();
```

- You can write code that is agnostic to where the input is coming from:

```
myControls.gameplay.look.perfor
med +=
context => look =
context.ReadValue<Vector2>();
```

Input Action

- You can then use the visual editor to establish the mapping:
- Actions use `InputBinding` to refer to the inputs they collect.
- Each Action has a name (`InputAction.name`), which must be unique within the Action Map that the Action
- Each Action also has a unique ID (`InputAction.id`), which you can use to reference the Action.



3 classes for Actions

Class	Description
InputActionAsset	An Asset that contains one or more Action Maps and, optionally, a sequence of Control Schemes.
InputActionMap	A named collection of Actions.
InputAction	A named Action that triggers callbacks in response to input.

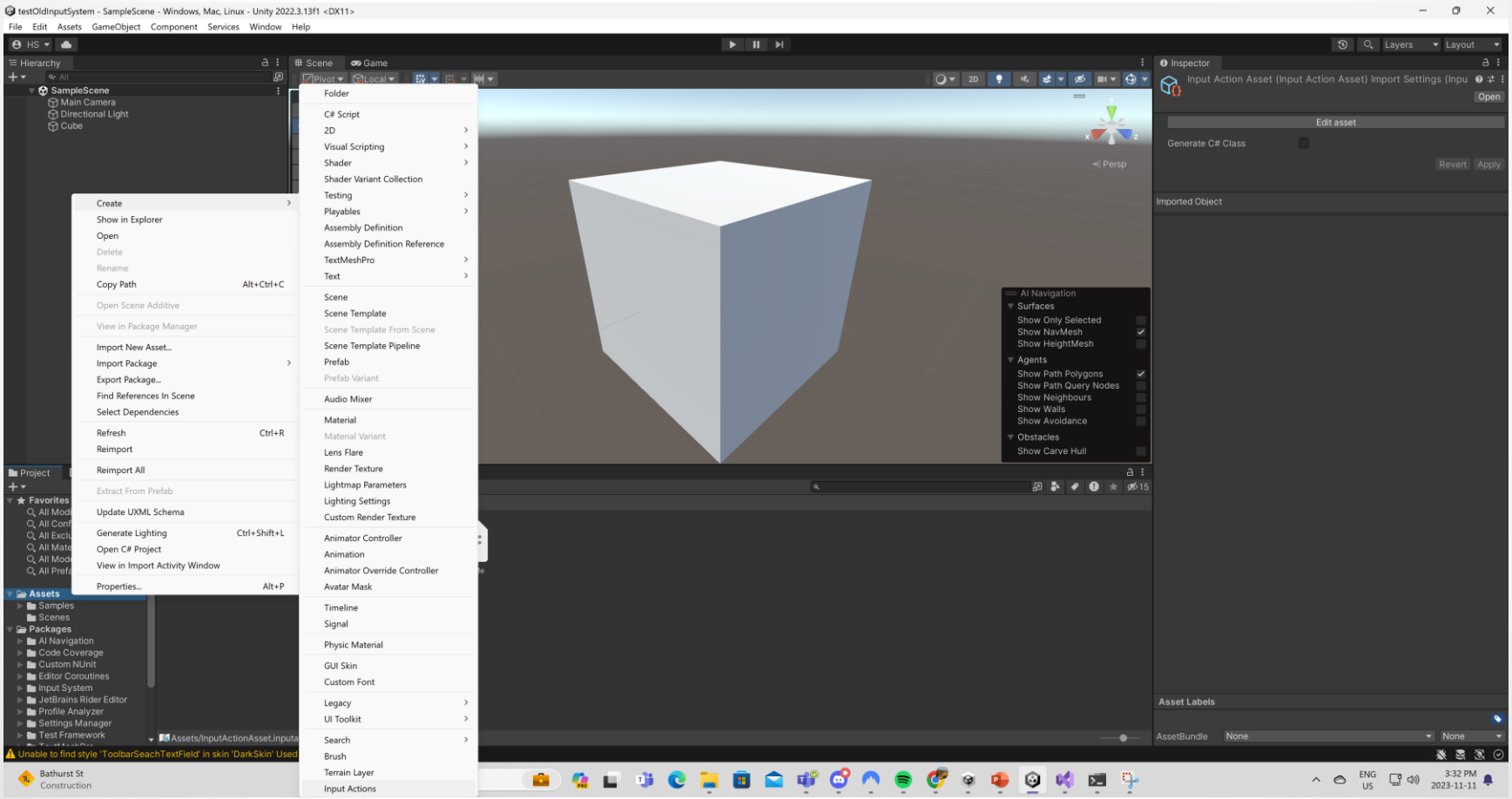
Creating Actions

- You can create Actions in one of the following ways:
  1. Use the dedicated editor for `.inputactions` Assets.
  2. Embed them in MonoBehaviour components.
  3. Manually load them from JSON.
  4. Create them directly in code.

Creating Input Action Assets

- To create an Asset that contains Input Actions in Unity, right-click in the **Project** window or go to **Assets > Create > Input Actions** from Unity's main menu.
- To bring up the Action editor, double-click an `.inputactions` Asset in the Project Browser, or select the **Edit Asset** button in the Inspector for that Asset.

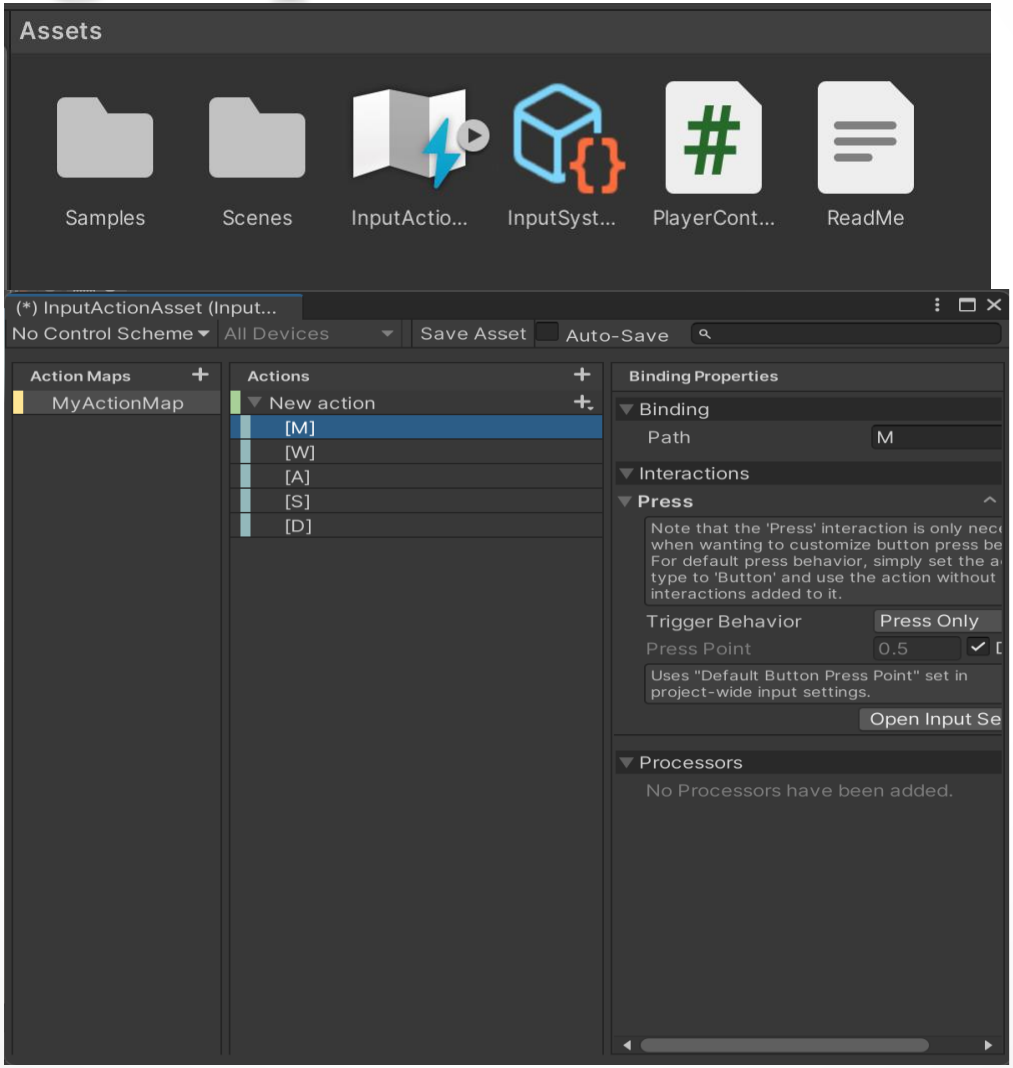
Create Input an Action Asset



Setting Up the Action Map

- To add a new Action Map, select the Add (+) icon in the header of the Action Map column.
- To add a new Action, select the Add (+) icon in the header of the Action column.
- To add a new Binding, select the Add (+) icon on the action you want to add it to, and select the binding type from the menu that appears.
- The most important property of any Binding is the control path it's bound to. To edit it, open the **Path** drop-down list. This displays a Control picker window.
- One of the most convenient ways to work with `.inputactions` Assets in scripts is to automatically generate a C# wrapper class for them. This removes the need to manually look up Actions and Action Maps using their names, and also provides an easier way to set up callbacks.

Setting Up an Action Map



Using Action Assets with `PlayerInput`

- The `PlayerInput` component provides a convenient way to handle input for one or multiple players.
- It requires you to set up all your Actions in an Input Action Asset, which you can then assign to the `PlayerInput` component.
- `PlayerInput` can then automatically handle activating Action Maps and selecting Control Schemes for you.

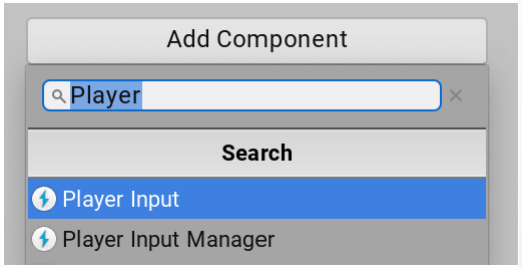
Getting input indirectly through an Input Action

- To get input directly through an Input Action, follow these steps:

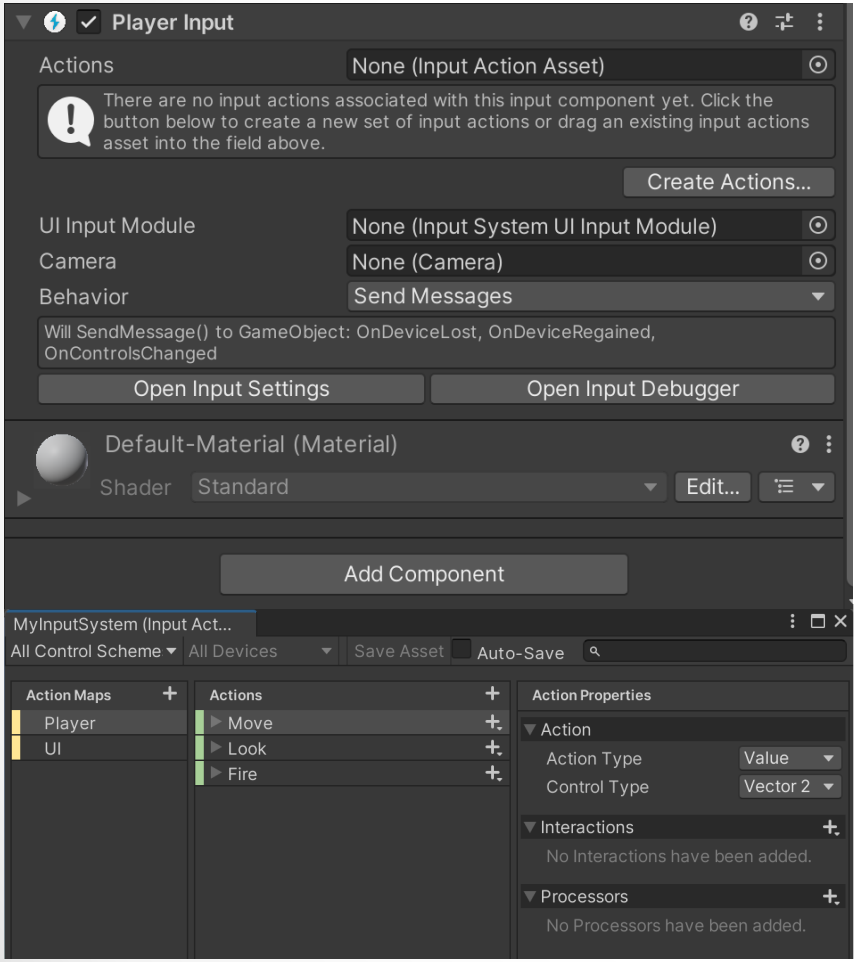
- Add a PlayerInput component.
- Create Actions.
- Script Action responses.

Step 1: Add a PlayerInput Component

- Getting input directly from an Input Device is quick and convenient, but requires a separate path for each type of Device. That also makes it harder to later change which Device Control triggers a specific event in the game.
- Alternatively, you can use Actions as an intermediary between Devices and the in-game responses they trigger. The easiest way to do this is to use the PlayerInput component. To add this component, click the Add Component button in the GameObject Inspector:



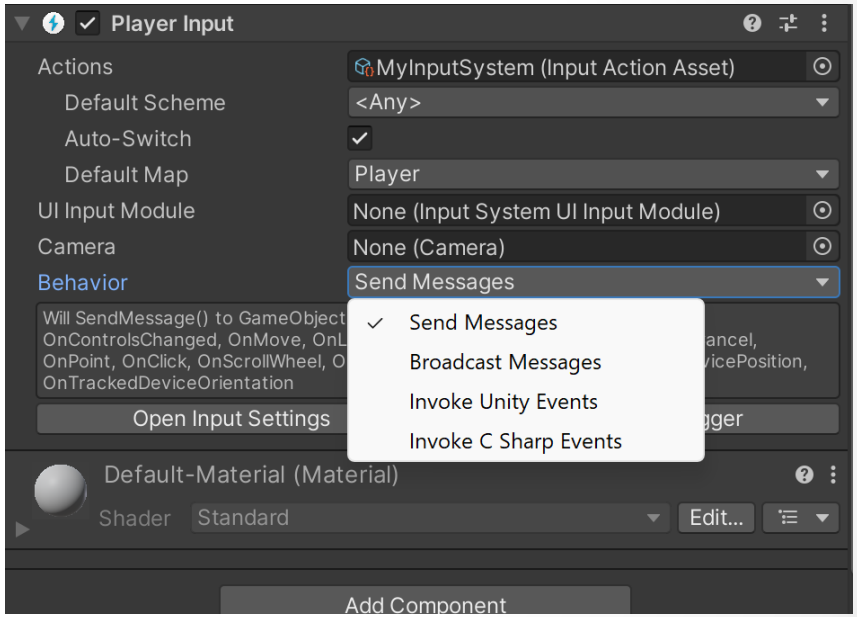
Step 2: Create Input Actions



- Each PlayerInput component represents one player in the game. To receive input, the component must be connected to a set of Input Actions. The quickest way to create a new set of Actions is to click the Create Actions… button in the Inspector window for that component. This creates an Asset pre-populated with a default set of Input Action Maps, Input Actions, and Input Bindings.

Step 3: Setting up Action responses

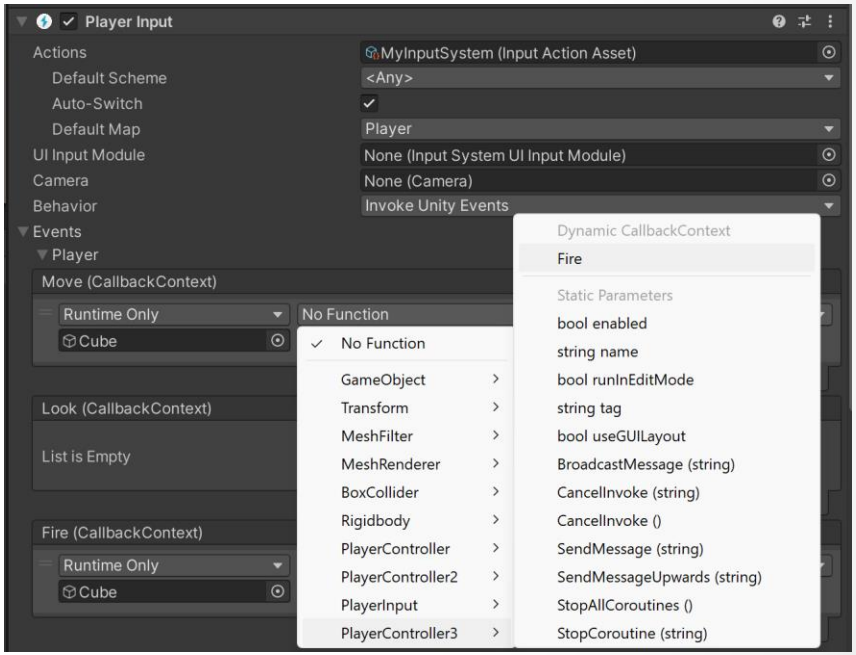
- Once the component has its Actions, you must set up a response for each Action. PlayerInput allows you to set up responses in several ways, using the Behavior property in the Inspector window.
- Select “Invoke UnityEvent” , which is in the same way the Unity UI does. Unity displays an event for each Action that is linked to the component. This allows you to directly wire in the target method for each event.



Setting Up Invoke Unity Event

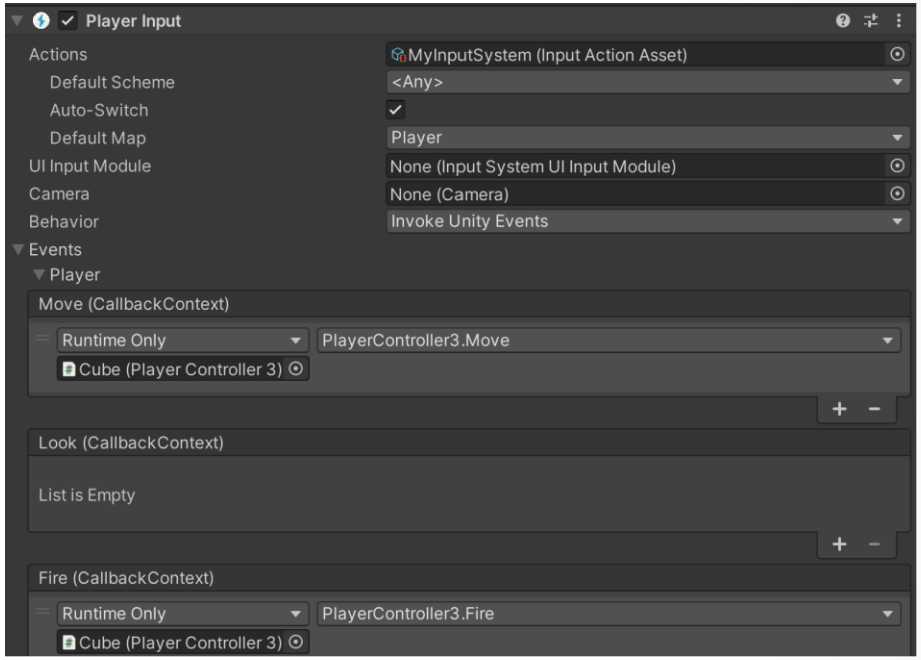
- Each method takes an InputAction.CallbackContext argument that gives access to the Control that triggered the Action and the Action's value.

```
 public void Fire(InputAction.CallbackContext context)
    {
        Debug.Log("Fire!");
    }
```



Move Action Example

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;


public class PlayerController3 : MonoBehaviour
{
    public Rigidbody rb;
    public float moveSpeed = 0.1f;

    Vector3 moveDirection = Vector3.zero;
    //float moveX = 0;
    //float moveY = 0;
    Vector2 moveXY = Vector2.zero;


    // Start is called before the first frame update
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {

    }

    public void Fire(InputAction.CallbackContext context)
    {
        Debug.Log("Fire!");
    }

    public void Move(InputAction.CallbackContext context)
    {
        Debug.Log("Move!");
        var moveXY = context.ReadValue<Vector2>();
        Debug.LogFormat("Move Direcction: ({0}, {1})", moveXY.x, moveXY.y);
        //moveDirection = new Vector2(moveX, moveY).normalized;
        moveDirection = moveXY.normalized;
    }

    private void FixedUpdate()
    {

        rb.velocity = new Vector3(moveDirection.x * moveSpeed * Time.fixedDeltaTime, moveDirection.y * moveSpeed * Time.fixedDeltaTime, moveDirection.z * moveSpeed * Time.fixedDeltaTime);
    }
}
```



Embedding Actions in MonoBehaviours

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class MyBahavior : MonoBehaviour
{
    public InputAction fireAction;
    public InputAction lookAction;

    public InputActionMap gameplayActions;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Enable/Disable Actions

- You must manually enable and disable Actions and Action Maps that are embedded in MonoBehaviour components.

```
void Awake()
    {
        fireAction.performed += OnFire;
        lookAction.performed += OnLook;

        gameplayActions["fire"].performed += OnFire;
    }

    void OnEnable()
    {
        fireAction.Enable();
        lookAction.Enable();

        gameplayActions.Enable();
    }

    void OnDisable()
    {
        fireAction.Disable();
        lookAction.Disable();

        gameplayActions.Disable();
    }
```

Loading Actions from JSON

```
// Load a set of action maps from JSON.
var maps = InputActionMap.FromJson(json);

// Load an entire InputActionAsset from JSON.
var asset = InputActionAsset.FromJson(json);
```

Creating Actions in code

```
// Create free-standing Actions.
var lookAction = new InputAction("look", binding: "<Gamepad>/leftStick");
var moveAction = new InputAction("move", binding: "<Gamepad>/rightStick");

lookAction.AddBinding("<Mouse>/delta");
moveAction.AddCompositeBinding("Dpad")
    .With("Up", "<Keyboard>/w")
    .With("Down", "<Keyboard>/s")
    .With("Left", "<Keyboard>/a")
    .With("Right", "<Keyboard>/d");

// Create an Action Map with Actions.
var map = new InputActionMap("Gameplay");
var lookAction = map.AddAction("look");
lookAction.AddBinding("<Gamepad>/leftStick");

// Create an Action Asset.
var asset = ScriptableObject.CreateInstance<InputActionAsset>();
var gameplayMap = new InputActionMap("gameplay");
asset.AddActionMap(gameplayMap);
var lookAction = gameplayMap.AddAction("look", "<Gamepad>/leftStick");
```

Default Actions

- An asset called `DefaultInputActions.inputactions` containing a default setup of Actions comes with the Input System Package.
- You can reference this asset directly in your projects like any other Unity asset. However, the asset is also available in code form through the `DefaultInputActions` class.

```
void Start()
{
    // Create an instance of the default actions.
    var actions = new DefaultInputActions();
    actions.Player.Look.performed += OnLook;
    actions.Player.Move.performed += OnMove;
    actions.Enable();
}
```

Responding to Actions

- An Action doesn't represent an actual response to input by itself. Instead, an Action informs your code that a certain type of input has occurred. Your code then responds to this information.
- There are several ways to do this:
- Each Action has a `started`, `performed`, and `canceled` callback.
- Each Action Map has an `actionTriggered` callback.
- The Input System has a global `InputSystem.onActionChange` callback.
- You can poll the current state of an Action whenever you need it.
- `InputActionTrace` can record changes happening on Actions.

Action callbacks

- You can read the current phase of an action using `InputAction.phase`.
- The `Started`, `Performed`, and `Canceled` phases each have a callback associated with them.
- Each callback receives an `InputAction.CallbackContext` structure, which holds context information that you can use to query the current state of the Action and to read out values from Controls that triggered the Action (`InputAction.CallbackContext.ReadValue`).

```
var action = new InputAction();

    action.started += ctx => /* Action was started */;
    action.performed += ctx => /* Action was performed */;
    action.canceled += ctx => /* Action was canceled */;
```

InputActionMap.actionTriggered callback

Instead of listening to individual actions, you can listen on an entire Action Map for state changes on any of the Actions in the Action Map.

```
var actionMap = new InputActionMap();
actionMap.AddAction("action1",
"<Gamepad>/buttonSouth");
actionMap.AddAction("action2",
"<Gamepad>/buttonNorth");

actionMap.actionTriggered +=
    context => { ... };
```

InputSystem.onActionChange callback

- Similar to `InputSystem.onDeviceChange`, your app can listen for any action-related change globally.

```
InputSystem.onActionChange +=
    (obj, change) =>
    {
        // obj can be either an InputAction or an InputActionMap
        // depending on the specific change.
        switch (change)
        {
            case InputActionChange.ActionStarted:
            case InputActionChange.ActionPerformed:
            case InputActionChange.ActionCanceled:
                Debug.Log($"{((InputAction)obj).name} {change}");
                break;
        }
    }
```

Polling Actions

- Instead of using callbacks, it might be simpler sometimes to poll the value of an Action where you need it in your code.

```
public InputAction moveAction;
    public float moveSpeed = 10.0f;
    public Vector2 position;

    void Start()
    {
        moveAction.Enable();
    }

    void Update()
    {
        var moveDirection = moveAction.ReadValue<Vector2>();
        position += moveDirection * moveSpeed * Time.deltaTime;
    }
```

InputAction.WasPerformedThisFrame()

To determine whether an action was performed in the current frame, you can use InputAction.WasPerformedThisFrame():

```
private InputAction action;

    void Start()
    {
        // Set up an action that triggers when the A button is
        // held for 1 second.
        action = new InputAction(
            type: InputActionType.Button,
            binding: "<Gamepad>/buttonSouth",
            interactions: "hold(duration=1)");

        action.Enable();
    }

    void Update()
    {
        if (action.WasPerformedThisFrame())
            Debug.Log("A button on gamepad was held for one second");
    }
```

poll for button presses and releases

```
public PlayerInput playerInput;

    public void Update()
    {
        // IsPressed
        if (playerInput.actions["up"].IsPressed())
            transform.Translate(0, 10 * Time.deltaTime, 0);

        // WasPressedThisFrame
        if (playerInput.actions["teleport"].WasPressedThisFrame())
            Teleport();

        // WasReleasedThisFrame
        if (playerInput.actions["submit"].WasReleasedThisFrame())
            ConfirmSelection();
    }
```

The new Input System Manager

- Install the new
- Test if the old input manager is disabled
  - using UnityEngine.Input;
- Now add the following to the script
- using UnityEngine.InputSystem;
  - If you get an error, Go to Edit --> Preferences --> External Tools -->
  - check the registry packages check box and then regenerate the project files
- Add a "Player Input" component to the "Player" Game Object
- On "Player Input" component, hit on "Create Actions" and save it as "PlayerControls"
- Notice that Player/UI action maps have already been created for you!
- Select PlayerControls in the project folder, go to inspector and check "Generate C# class" and Apply.

Update the playerController script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;


public class PlayerController : MonoBehaviour
{

    public Rigidbody rb;
    public float moveSpeed = 5f;

    Vector3 moveDirection = Vector3.zero;

    public PlayerControls playerControl;
    private InputAction move;
    private InputAction fire;

    private void Awake()
    {

        playerControl = new PlayerControls();
    }


    private void OnEnable()
    {
        playerControl.Enable();
         move = playerControl.Player.Move;
         move.Enable();

        fire = playerControl.Player.Fire;
        fire.Enable();
        //Register the fire method to the event
        fire.performed += Fire;
    }

    private void OnDisable()
    {
        playerControl.Disable();

        move.Disable();
        fire.Disable();
    }

 void Start()
    {
        Debug.Log(Time.fixedDeltaTime);
    }

    void Update()
    {
        moveDirection = move.ReadValue<Vector2>();
    }

    private void FixedUpdate()
    {

        rb.velocity = new Vector3(moveDirection.x * moveSpeed,
moveDirection.y * moveSpeed, moveDirection.z * moveSpeed);
    }


    private void Fire(InputAction.CallbackContext context)
    {
        Debug.Log("We fired");
    }

}
```