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

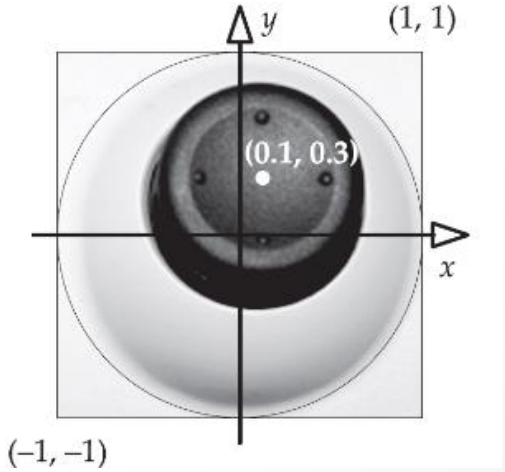
```
typedef struct _XINPUT_GAMEPAD {
WORD wButtons;
BYTE bLeftTrigger;
BYTE bRightTrigger;
SHORT sThumbLX
SHORT sThumbLY
SHORT sThumbRX
SHORT sThumbRY
}
```

- 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

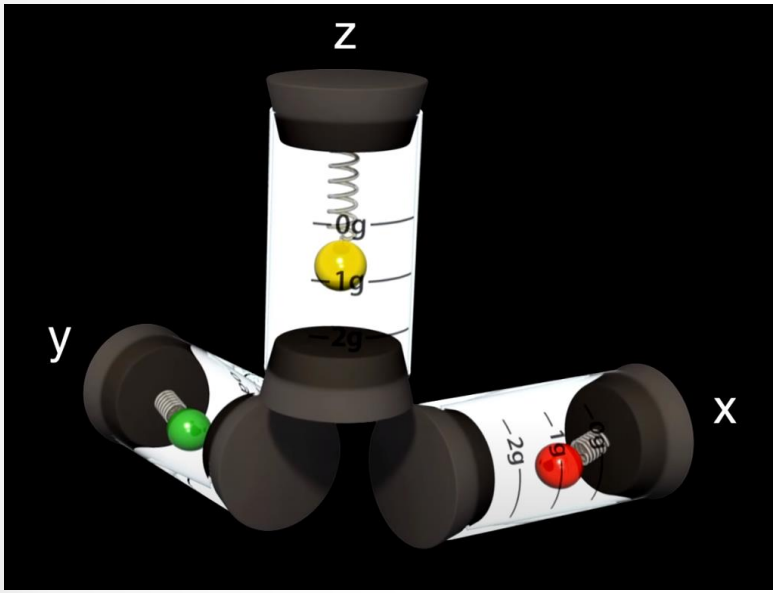
- XINPUT_GAMEPAD: Microsoft represents the deflections of the left and right thumb sticks on the Xbox 360 gamepad using 16-bit signed integers (sThumbLX and sThumbLY for the left stick and sThumbRX and sThumbRY for the right).
- these values range from -32,768 (left or down) to 32,767 (right or up).
- 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

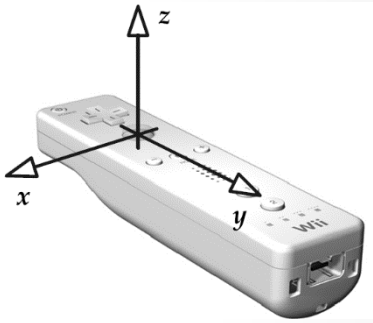
Accelerometers



- 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

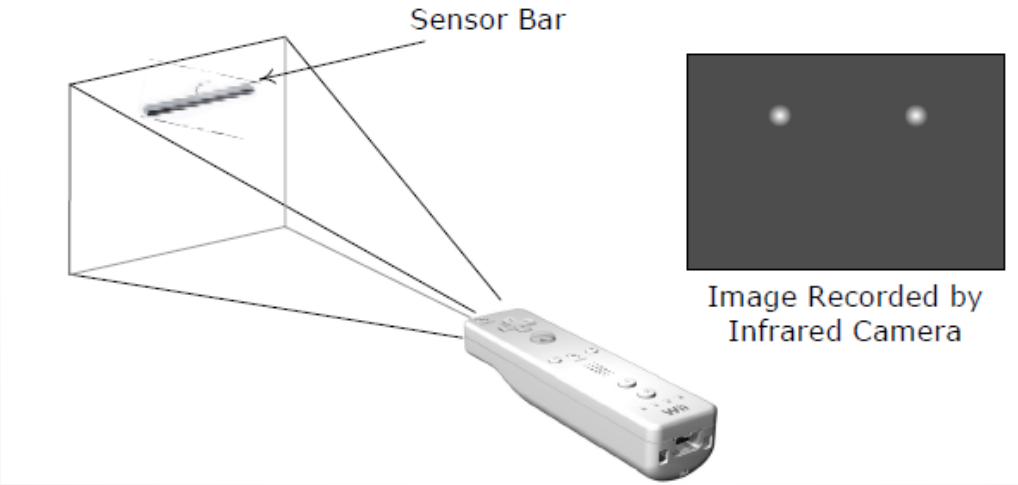


Dual Sense

- 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();
}
}
};
```

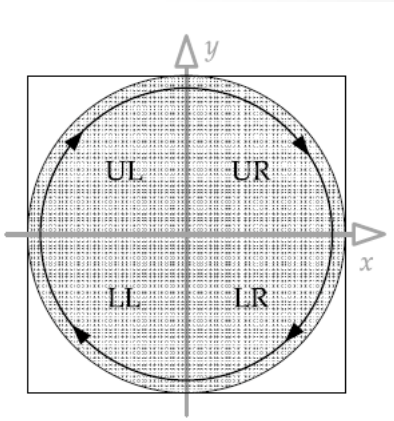
Multibutton Sequence

```
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