29

Designing a Robust Input Handling System for Games

Peer Reviewed by **Michael Tanczos (/user/5-michael-tanczos)**, **jbadams (/user/36615-jbadams)**, **Nercury (/user/208401-nercury)**

Mike Lewis (/user/33873-apochpiq) | Mar 19 2013 03:14 AM |
Game Programming (https://www.gamedev.net/resources/_/technical/game-programming/)

Input (https://www.gamedev.net/index.php?app=search&module=tags&section=view&tag=Input)

Input Handling (https://www.gamedev.net/index.php?app=search&module=tags&section=view&tag=Input%2BHandling)

Key Bindings (https://www.gamedev.net/index.php?app=search&module=tags&section=view&tag=Key%2BBindings)

Ever wonder how to allow reconfigurable keybinds in your game? Ever scratch your head as to how to handle various types of input device in a single game? Curious how the pros build input handling mechanisms? This will answer all those questions and more!

A common question for those designing a new game engine is "how do I handle input?" Typically, there's a few core issues that pretty much every game faces, and unlike many areas of game implementation, input is one where we can more or less build a one-size-fits-all framework. Fortunately, this is a pretty straightforward thing to do, and even is portable across APIs and platforms pretty easily.

For the purposes of this article, we'll focus on the platform-independent side of things. Getting input from the underlying hardware/OS is up to you; for Windows, Raw Input is pretty much the de facto standard; XInput might be useful if you want joystick/controller support. For other platforms, research and select the API or library of your choice.

Got a way to get pure input data? Good. Let's take a look at the overall input system architecture.

Designing a Robust Input Handling System

We have a few goals here which should make this system (or ones based on it) applicable for pretty much any game, from a simple 2D platformer to an RTS to a 3D shooter:

- Performance is important; input lag is a bad thing.
- It should be easy to have new systems tap into the input stream.

- The system must be very flexible and capable of handling a wide variety of game situations.
- Configurability (input mapping) is essential for modern games.

Thankfully, we can hit all of these targets with fairly minimal effort.

We will divide the system into three layers:

1. Raw input gathering from the OS/etc.
2. Input mapping and dispatch to the correct high-level handlers
3. High level handler code

The first layer we have already decided to gloss over; its specifics aren't terribly important. What matters is that you have a way to pump pure input data into the *second* layer, which is where most of the interesting stuff happens. Finally, the third layer will implement your specific game's responses to the input it receives.

Contexts

The central concept of this system is the *input context*. A context defines what inputs are available for the player at a given time. For instance, you may have a different context for when a game menu is open versus when the game is actually being played; or different modes might require different contexts. Think of games like Final Fantasy where you have a clear division between moving around the game world and combat, or the Battlefield series where you get a different set of controls when flying a helicopter versus when running around on the ground.

Contexts consist of three different types of input:

1. Actions
2. States
3. Ranges

An action is a single-time thing, like casting a spell or opening a door; generally if the player just holds the button down, the action should only happen once, generally when the button is first pressed, or when it is finally released. "Key repeat" should not affect actions.

States are similar, but designed for continuous activities, like running or shooting. A state is a simple binary flag: either the state is on, or it's off. When the state is active, the corresponding game action is performed; when it is not active, the action is not performed. Simple as that. Other good examples of states include things like scrolling through menus.

Finally, a range is an input that can have a number value associated with it. For simplicity, we will assume that ranges can have any value; however, it is common to define them in *normalized* spans, e.g. 0 to 1, or -1 to 1. We'll see more about the specifics of range values later. Ranges are most useful for dealing with analog input, such as joysticks, analog controller thumbsticks, and mice.

Input Mapping

The next feature we'll look at is input mapping. Simply put, this is the process of going from a raw input datum to an action, state, or range. In terms of implementation, input mapping is very simple: each context defines an *input map*. For many games, this map can be as straightforward as a C++ map object (aka a dictionary or table in other languages). The goal is simply to take an identified type of hardware input and convert it to the final type of input.

One twist here is that we might need to handle things like key-repeat, joysticks, and so on. It is especially important to have a mapping layer that can handle ranges intelligently, if we need normalized range values in the high-level game logic (and I strongly recommend using normalized values anywhere possible). So an input mapper is really a set of code that can convert raw input IDs to high-level context-dependent IDs, and optionally do some normalization for range values.

Remember that we need to handle the situation where different contexts provide different available actions; this means that each context needs to have its own input map. There is a one-to-one relationship between contexts and input maps, so it makes sense to implement them as a single class or group of functions.

Dispatching

There are two basic options for dispatching input: callbacks, and polling. In the callback method, every time some input occurs, we call special functions which handle that input. In the polling method, code is responsible for asking the input management system each frame for what inputs are occurring, and then reacting accordingly.

For this system, we will favor a callback-based approach. In some situations it may make more sense to use polling, but if you're writing game code for those scenarios, chances are you don't need any advice on how to build your input system

The basic design looks like this:

- Every frame, raw input is obtained from the OS/hardware
- The currently active contexts are evaluated, and input mapping is performed
- Once a list of actions, states, and ranges is obtained, we package this up into a special data structure and invoke the appropriate callbacks

Note that we specifically might want to allow more than one context to be valid at once; this is often useful for cases where basic activities (running around) are always available to the player, but specific activities need to be restricted based on the current scenario (what weapons I'm carrying, perhaps).

I recommend implementing this as a simple ordered list: each context in the list is given the raw input for the frame. If the context can validly map that raw input to an action, state, or range, it does so; otherwise, it passes on to the next context in the list. This can be done effectively using something like a Chain of Responsibility pattern. This allows us to prioritize certain contexts to make sure they always get first crack at mapping input, in case the same raw input might be valid in multiple active contexts. Generally, the more specific the context, the higher priority it should carry.

The other half of this scenario is the callback system. Again there are several ways to approach this, but in my experience, the most powerful and flexible method is to simply register a set of general callbacks that are given input every frame (or whenever input is available). Again, a chain of responsibility works well here: certain callbacks might want first crack at handling the mapped input. This is again useful for special situations like debug modes or chat windows.

Have the input mapper wrap up all of its mapped inputs into a simple data structure: one list of valid actions, one list of valid states, and one list of valid ranges and their current values. Then pass this data on to each callback in turn. If a callback handles a piece of input, it should generally remove it from the data structure so that further callbacks don't issue duplicate commands. (For instance, suppose the M key is handled by two registered callbacks; if both callbacks respond to the key, then two things will happen every time the player presses the M key! Oops! So if the first callback to handle the key "eats" it from the list, then we don't have to worry, and we can use a simple priority system to make sure that the most sensible callback gets dibs on the input.)

High Level Handling

Once the input is available, we simply need to act on it. For actions and states, this is just a matter of having our callbacks investigate the data list and take action appropriately. Ranges are similar but slightly more complex in that we have to turn the input value into something useful. For things like joysticks, this is easy: use a normalized -1 to 1 value and just multiply that by your sensitivity factor, and poof, you have a mapped range of input. (Try using a logistical S-curve or other interpolator for better results than just multiplication.) For mice, you can use the value to tell you how far to move the cursor/camera, again possibly by using a scaling factor for sensitivity purposes.

The specifics of this third layer are really up to your game's design and your imagination.

Putting Everything Together

So, let's recap the basic flow of data through the system:

1. The first layer gathers raw input data from the hardware, and optionally normalizes ranged inputs
2. The second layer examines what game contexts are active, and maps the raw inputs into high-level actions, states, and ranges. These are then passed on to a series of callbacks
3. The third layer receives the callbacks and processes the input in priority order, performing game activity as needed

That's all there is to it!

A Word on Data Driven Designs

So far I've been vague as to how all this is actually coded. One option is certainly to hard-code everything: in context A, key Q corresponds to action 7, and so on. A far better option is to make everything *data driven*. In this approach, we write code once that can be used to handle *any* context and any input mapping scheme, and then feed it data from a simple file to tell it what contexts exist, and how the mappings work.

The basic layout I typically use looks something like this:

- rawinputconstants.h (a code file) specifies a series of ID codes, usually in an enumeration, corresponding to each raw input (from hardware) that we might handle. These are divided up into "buttons" and "axes." Buttons can map to states or actions, and axes always map to ranges.
- inputconstants.h (a code file) specifies another set of ID codes, this time defining each action, state, and range available in the game.
- contexts.xml (a data file) specifies each context in the game, and provides a list of what inputs are valid in each individual context.
- inputmap.xml (a data file) carries one section per context. Each context section lists out what raw input IDs are mapped to what high-level action/state/range IDs. This file also holds sensitivity configurations for ranged inputs.
- inputranges.xml (a data file) lists each range ID, its raw value range (say, -100 to 100), and how to map this onto a normalized internal value range (such as -1 to 1).
- A code class called RangeConversions loads inputranges.xml and handles converting a raw value to a mapped value.
- A code class called InputContext encapsulates all of the functionality of mapping a single context worth of inputs from raw to high-level IDs, including ranges. Sensitivity configurations are applied here. This class basically just exists to act on the data from inputmap.xml.
- A code class called InputMapper encapsulates the process of holding a list of valid (active) InputContexts. Input is passed into this class from the first-layer code, and out into the third-layer code.
- A code class (usually a POD struct in C++ versions of the system) called MappedInput holds a list of all the input mapped in the current frame, as covered above.
- Each frame (or whenever input is available), the first layer of input code takes all of the available input and packs it into an InputMapper object. Once this is finished, it calls InputMapper.Dispatch() and the InputMapper then calls InputContext.MapInput() for each active context and input. Once the final list of mapped input is compiled into a MappedInput object, the MappedInput is passed into each registered callback, and the high-level game code gets a chance to react to the input.

And there you have it! Complete, end-to-end input handling. The system is fast, easily extended to handle new game functionality, easily configurable, and simple to use.

Go forth and code some games!

If you'd like to see an example of how this works in action, check out the Input Mapping Demo at my Google Code repository (http://scribblings-by-apoch.googlecode.com/).