##Game Programming Patterns_Revisited_State##

"Complex branching & mutable state — fields that change over time — are two of those error-prone kinds of code..."

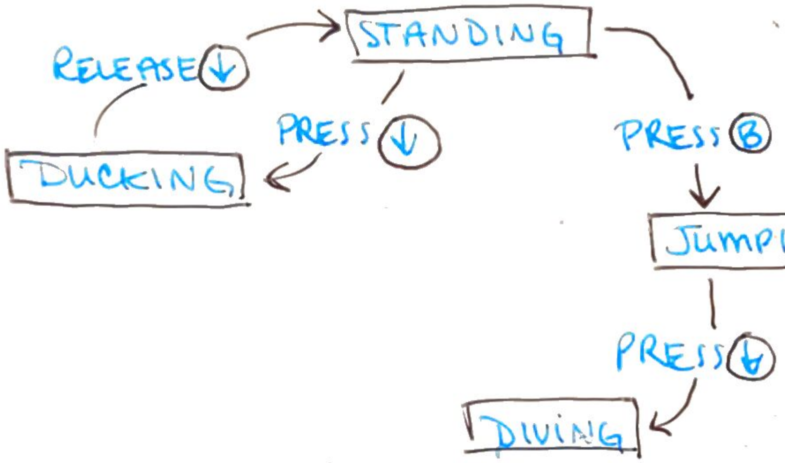
Finite State Machines

① You have a fixed set of states that the machine can be in

② The machine can only be in <u>one</u> state at a time

③ A sequence of inputs/events is sent to the machine

④ Each state has a set of transitions, each associated w/an input & pointing to a state



Ex. Zork "Each room is a state. The room you're in is your current state. Each room's exits are its transitions. The navigation commands are the inputs".

① Create enum for state

② SWITCH on state then check for input

STATE Pattern: Allow an obj. to alter its behavior when its internal state changes. The object will appear to change its class

① def. state interface, every behavior that was state dependent is a virtual method in the interface

② Class for each state, that implements interface

③ delegate to the state; get ptr to current state & delegate to each state class instead

"With State the goal is for the main obj. to <u>change</u> its behavior by <u>changing</u> the obj. it delegates to."

* your state ptr. could be: *static class, *top level function, instantiated class

↳ could have eac FSM have its own instance of the state & delete the old one once it has a new state. Be wary of fragmentation

④ Enter & Exit Actions

*want each state to control its own graphics, so we can give the ~~states~~ states an entry action (and, if helpful, an exit action)

Concurrent State Machines

⚠ what if you want to track what the character is doing & carrying?

"If we want to cram n states for what she's doing & m states for what she's carrying into a single machine, we need n x m states. With two machines, it's just n + m"

① define 2 states: state_ & equipment_

② when delegating, hand input to both FSMs

↳ works well when the two states are independent. If not, you can check on the status of one state from the other? to coordinate

Hierarchical State Machines

*likely have a bunch of similar states. A state can have a superstate, making itself a substate.

↳ when an event comes in if the substate doesn't handle it it rolls up the chain of superstates (like overriding inherited methods)

↳ you can also use a stack of states instead of a single state in the main class "the current state is the one on the top of the stack, under that is its immediate superstate, and the that state's superstate & so on" & you walk down the stack

Pushdown Automata

*FSMs have no concept of history

"where a (FSM) has a single ptr to a state, a pushdown automaton has a stack of them." So you can push or pop state.

OVERALL

Good for: entity whose behavior changes based on some internal state, that state can be rigidly divided into one of a relatively small # of distinct options & it responds to a series of inputs over time

*See behavior trees & planning systems