Organic HFSM / AI System Growth

A case study in evolving a hierarchical FSM system used to ship (literally) dozens of medium-budget projects

What is this talk about (and why do I care?)

- I'm going to walk you through the evolution of a HFSM system

- Questions to answer:
  - Should you do this?
  - Is this still a relevant technology to use?
  - Are scripting languages a trap?
  - Are tools only for sissies?

Battlefield Proven

- This talk describes the evolution of a SHPFSM* system and toolset over many projects, three generations of engines, and more than a decade of time
  - * Scripted Hierarchical Probabilistic Finite State Machine ☺
- Most games it was used for were platformers, but it has also been used for:
  - “action” adventure games (Leisure Suit Larry: MCL), etc.
  - turn-based arena combat game
  - real-time RPG
  - eventually used in FPS
- *At least* 15 titles (33+ SKUs) shipped using this tech.

Credit and Thanks

- Credit to those who did much of this work...
  - Jerry Karaganis
  - Brant French
  - Frank Wilson
  - Mike Henry
  - Jason Petersohn
  - (others at High Voltage Software)

Cold Realities

- Non-"AAA" titles:
  - no dedicated AI staff
  - one attempt to “get it ‘right’ (aka. good enough)”
  - many developers using system at once
  - 1/2 of team typically inexperienced, no “technical designers” (programmers disguised as design staff)
  - very little schedule for up front “technology development”
  - expectation of “reusable for other projects *and game types*” (that last part is killer)

The Good, the Bad, …

- FSM's are "antiquated", but they have certain advantages
  - understandable with very modest training (mastery is a different story)
  - deterministic behavior
  - speedy (to execute)
  - integrates easily with other game systems
- Disadvantages
  - number of states can explode when handling asynchronous sequences or slightly different situations
  - "stiff" and predictable agent behavior (these first two are inversely correlated)
  - may have trouble scaling to complex characters/situations
  - explicit coding of new state selection routines can get rough and can burn lots and lots of CPU (and development) time

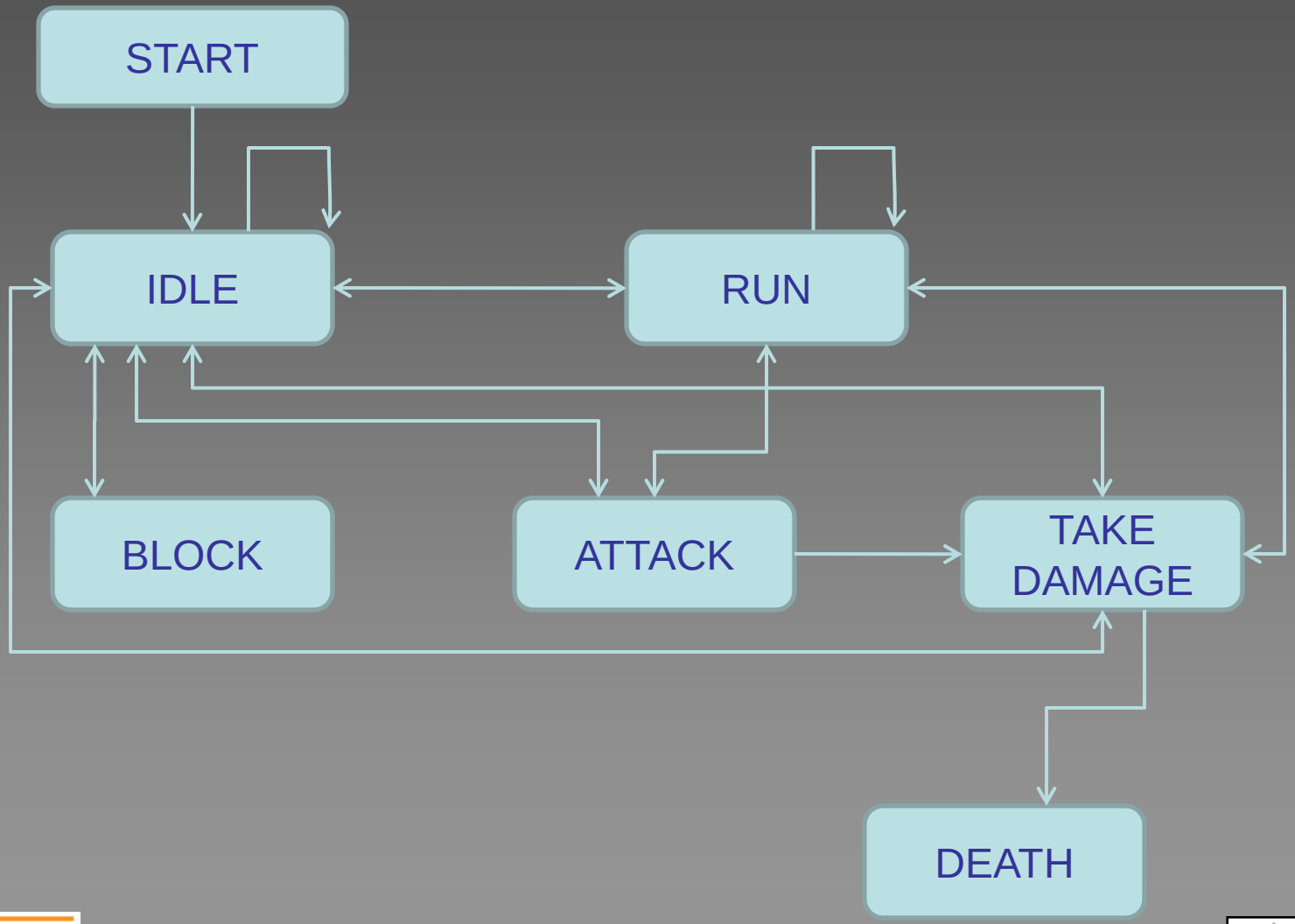
… and the Ugly

- Ugly
  - adding new states or triggers can be error prone
  - big FSMs are hard to visualize and understand

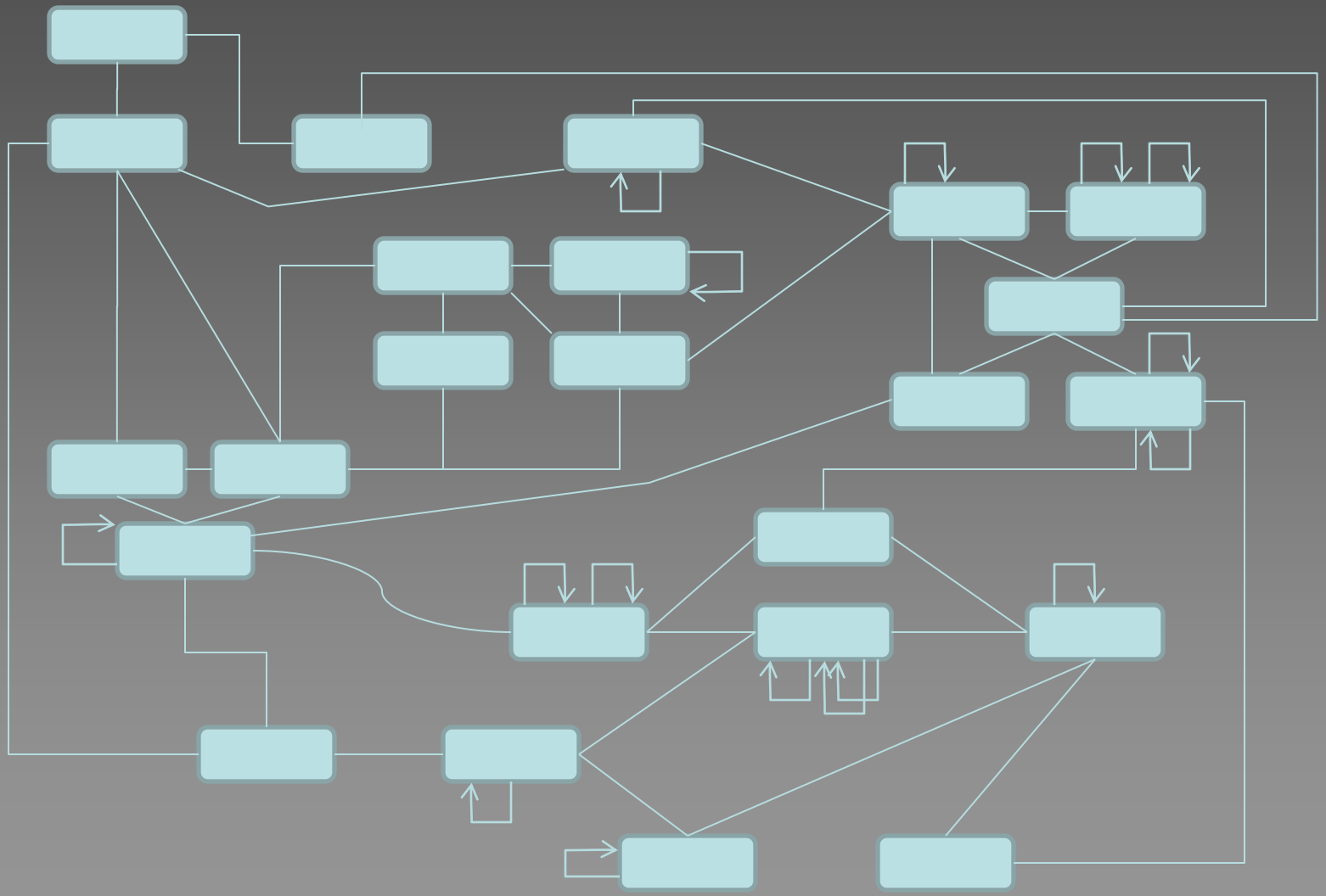
- Picture a big FSM with a dozen states and 40 events…
  - Put that into a big nested double switch block
  - Add ifs, loops and other non-trivial control logic
  - Let 2 other (junior) engineers work on it
  - Now add a new event…
    … and make sure that every state handles it properly

(good luck with that…) ☹

The Simple FSM



The Actual FSM…



Introducing the State Chart

- aka. Behavior Chart, FSM chart, state chart
- In this talk, I will refer to this as the “primary finite state machine” or PFSM

	Init (state entry event)	**Stop** (motion event)	**Run** (motion event)	**AnimOver** (animation event)	**Fire** (motion event)	**Bang** (animation event)	**Hit** (simulation event)

START (state)	model(“grunt”);
	update(user);
	state(IDLE);						

IDLE (state)	animLoop(“idle”);
	update(user);
			state(RUN);		state(FIREGUN);	// should never happen	takeDamage();
							if(hit_points < 0)
							{ state(DIE); }

RUN (state)	animLoop(“run”);
	update(user);
		state(IDLE);			state(FIREGUN);	// should never happen	takeDamage();
							if(hit_points < 0)
							{ state(DIE); }

FIREGUN (state)	update(stopped);
	anim(“firegun”);
				// no bang?
				state(IDLE);
						playVfx(“bang”);
						state(IDLE);
							takeDamage();
							if(hit_points < 0)
							{ state(DIE); }

DIE (state)	anim(“die”); update(none);			state(DEAD);		// should never happen	// already dying
DEAD (state)	model(“corpse”); anim(“none”);					// should never happen	model(“gibs”); playVfx(“spray”); exitChart();

Events

- The PFSM receives “events” which cause the contents of a “cell” to be executed
- This may change the PFSM’s state or any of the dependent states* (*next slide)
- Incoming events may be sent by the event system, other agents, from the engine (collision, hit by projectile, etc.), from triggers embedded in animation / audio files, etc.
  - Some standard events include the current animation/sound starting a new loop, collision with another agent, collision with the ground, non-looping animation end, nav point arrival, etc.
- New state selection routines may also result in state change request events

Dependent States

- Dependent states are parameters of the agent independent of the main state machine's state
  - e.g. the "walking" state would have a walking animation, a sound loop, a collision model, a visual mesh, a collision mesh, etc. which may be used in other places in the PFSM
- A key dependent state is what to do (what function to call) for the every-frame update of the agent (simulation, collision, etc.)

State Transition Extras

- Every PFSM state change generates “automatic” events: ‘Init’, ‘Exit’
  - changing state causes the Exit of the current state to be called followed by the Init of the new state
- These allow for standardized and reusable PFSM state initialization & cleanup
- Example state change from state X to state Y
  - state change function called when currently in state X
  - X-Exit is called followed by Y-Init automatically

Easy State and/or Trigger Addition

- Added “PATH” state: note clarity in seeing which events are handled and what is done as a result

	Init (state entry event)	**Stop** (motion event)	**Path** (motion event)	**Run** (motion event)	**AnimOver** (animatn event)	**Fire** (motion event)	**Bang** (animatn event)	**Hit** (simulation event)

START (state)	model(“grunt”);
	update(user);
	state(IDLE);							

IDLE (state)	animLoop(“idle”);
	update(user);
			state(PATH);	state(RUN);		state(
						FIREGUN);
								takeDamage();
								if(hit_points < 0)
								{ state(DIE); }

PATH (state)	animLoop(“walk”);
	path(“patrol”);
	update(path);
		path(NULL);
		state(IDLE);
				state(RUN);				takeDamage();
								if(hit_points < 0)
								{ state(DIE); }
								else
								{ state(IDLE); }

RUN (state)	animLoop(“run”);
	update(user);
		state(IDLE);	state(PATH);			state(
						FIREGUN);
								takeDamage();
								if(hit_points < 0)
								{ state(DIE); }

FIREGUN (state)	motion(none);
	anim(“firegun”);
					// no bang?
					state(IDLE);
							playVfx(
							“bang”);
							state(
							IDLE);
								takeDamage();
								if(hit_points < 0)
								{ state(DIE); }

… (state)								

Easy Debugging…

- Added debug error messages (and colors!) for invalid combinations…

	Init (state entry event)	**Stop** (motion event)	**Run** (motion event)	**AnimOver** (animation event)	**Fire** (motion event)	**Bang** (animation event)	**Hit** (simulation event)

START (state)	model("grunt");
	update(user);
	state(IDLE);
		invalid();	invalid();	invalid();	invalid();	invalid();	invalid();

IDLE (state)	animLoop("idle");
	update(user);
		// ignored	state(RUN);	invalid("non-looping anim?");	state(FIREGUN);	invalid();	takeDamage();
							if(hit_points < 0)
							{ state(DIE); }

RUN (state)	animLoop("run");
	update(user);
		state(IDLE);	// ignored	invalid("non-looping anim?");	state(FIREGUN);	invalid();	takeDamage();
							if(hit_points < 0)
							{ state(DIE); }

FIREGUN (state)	update(stopped);
	anim("firegun");
		// ignored	// ignored	invalid("no bang?");
				state(IDLE);
					invalid();	playVfx("bang");
						state(IDLE);
							takeDamage();
							if(hit_points < 0)
							{ state(DIE); }

DIE (state)	anim("die"); update(none);	invalid();	invalid();	state(DEAD);	invalid();	invalid();	// already dying // ignored

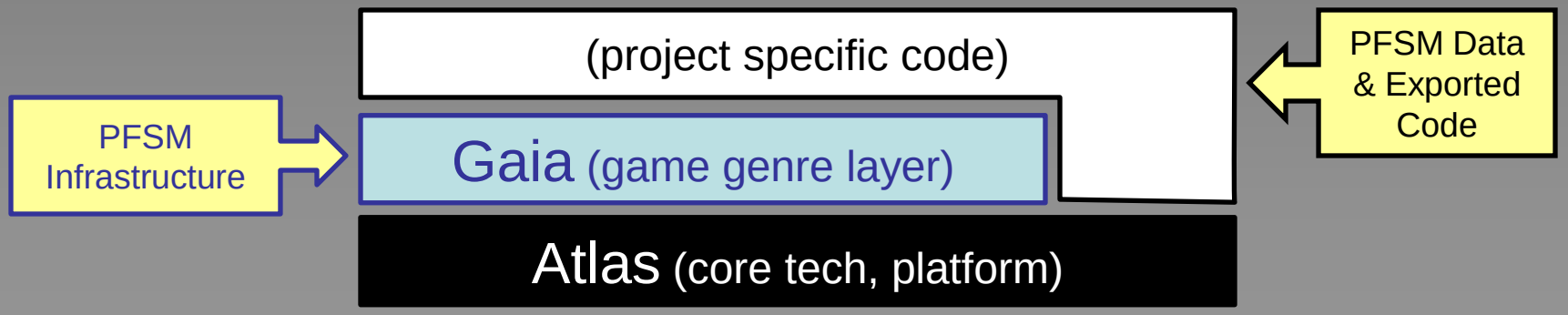
DEAD (state)	model("corpse");
	anim("none");
		invalid();	invalid();	invalid();	invalid();	invalid();	model("gibs");
							playVfx("spray");
							exitChart();

It scales nicely

- Charts with 5-10 states and 10-20 events are very easy to work with
  - State charts with 15-20 states and 20-40 events are quite manageable
- It can get out of hand…
  - PFSMs for major characters were made with 30+ states and 50 events…

Evolution of the System

- This system evolved over many projects and several game technologies (engine and toolsets)
- Major “generations” usually occurred to address a specific project’s design and anticipated or observed problems, but these in improvements were embraced and refined by the future projects
- PFSM code in game genre layer
- PFSM data part of game project specific code/data



Version 1: GAUL Engine (PS1, Win95, N64)

- Export to native C (almost assembly)
- Single layer PFSM managing all agent (character, prop, "level" itself) states as a single PFSM state (i.e. every dependent state is 1-1 with the PFSM state)
- "motion states" to handle per-frame simulation updates
- Exported from Excel (I'll get back to this later)

☹ too many "similar" states

☺ easy to visualize state flow, event handling

Version 2: Atlas tech. (PS2, Xbox, GameCube)

- Export to native C++
- Added automatic transitions and "every frame" events
- Added "dependent states" managed by FSM
- Optimized state re-use
- External parameter (simple) hierarchical DB for per-agent or per-type parameters
- Ad-hoc probabilistic state transitions and state selection functions
- Many in-game tools to support system: (watch particular agents in detail, state "floating over their head", etc.)
- Late in this version: finally an importer!

Version 2: Atlas tech. (cont)

- ☺ FAST !! Easy to optimize for both performance and memory
- ☺ Visualizing agent states, flow between them, handling new events
- ☹ Every agent, no matter how simple, required to have PFSM and motion state
- ☹ Linking and restart for small changes on GC & PS2 (parameter DB had helped with this)
- ☹ Before the importer: lack of in-IDE editing

Version 2: Atlas tech. (cont)

Editing code in Excel == Unspeakable Horror

Really.

I'm not kidding.

(adding the importer fixed 80% of this)

Version 3: Scripted State Machines (SSMs)

- Conversion to "small C" scripting language
  (but you could still use C++ PFSMs if wanted)
- Native method signatures (bindings) exported
- Many convenience utilities added; e.g. probabilistic state selection and/or external “select next state” calls
- Used existing cascading agent parameter storage
  (so old in-game editor tools worked “automatically”)

☹ Trained designers to use: but this was abandoned
(only 2 of 6 were effective with it)

☺ No recompile to reload state logic (huge win on systems with long link and executable startup times)

☺ Acceptable performance: profiled “cold”

SSM's: before scripting…

- The temptation to put complex C++ code into cells was too great!
  - Long dereference chains
  - Complex loops, branching, etc.
  - Hardwired paths to resources, magic numbers, string literals…
  - Data access / lookup workarounds
- This caused many debugging issues
- Reading / understanding / debugging complex PFSMs became very difficult

SSM's: after…

- Required to use interface to engine
  - Simpler charts
  - Enforced sandboxing
  - No more NULL pointers, etc.
  - Fixing problematic functions or refactoring became an order of magnitude simpler

"Could this be done without the scripting and just an enforced interface?"

- Yes, but it didn't happen…
  - It was too easy to just slam chunks of C++ code into the cells: the necessity of having to write and maintain an interface was enough to bring this discipline
  - (… but that doesn't mean a C++ interface wouldn't solve the problem)

Version 4: Hierarchical SSMs (Atlas)

- Added hierarchy and child state management
- Added ‘start’ and ‘exit’ events to all SSMs
- Child SSMs would receive events first, if unhandled then sent to parent (could also be explicitly sent to parent)

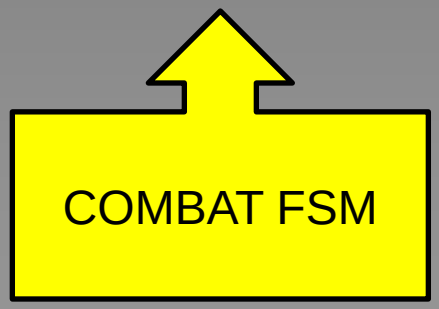
😮 when does this stop being a HFSM and start being a Behavior Tree?

☺ greatly simplified complex agents

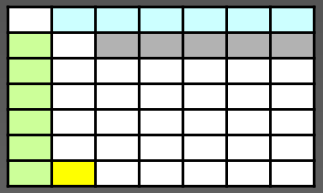
☹ more expertise required for “good” use: solved by having all complex agents done by 1-2 “semi” dedicated engineers

HSSM Visual Example

	Init	Target Lost	Target Seen	Summon	Melee Range	Die
START	state(ROAM)					
DIE	FSM(none) anim(“die”)					
ROAM	FSM(roam)		state(CHASE)	state(NAV)	state(COMBAT)	state(DIE)
NAV	setNavGoal() FSM(nav)		state(CHASE)	setNavGoal()	state(COMBAT)	state(DIE)
CHASE	FSM(pursue)	state(ROAM)		state(NAV)		state(DIE)
COMBAT	sound(“brains!”) **FSM(combat)**	state(ROAM)	If(noMeleeTarget()) { state(CHASE) }			state(DIE)



HSSM Visual Example



Unhandled events go up to parent FSM

	Init	Target Down	Hit	Miss	Grab	Damaged
START	model(melee) setMeleeTarget() state(GET CLOSE)					
END	model(default) clearMeleeTarget()					
GET CLOSE	update(close) anim(chase)	state(EAT)				If(dead()) { parentEvent(Die) }
ATTACK	update(attack) anim(attack)	state(EAT)	playHitVfx() sound("rarr!") damageTarget() anim(attack)	sound("woe") anim(attack)	state(GRAB)	If(dead()) { parentEvent(Die) }
EAT	soundLoop("eating") animLoop("eating") update(stopped)					If(dead()) { parentEvent(Die) }
GRAB	**FSM(grab)**	state(EAT)				If(dead()) { parentEvent(Die) } else state(LET GO)

GRAB FSM

Oh Noes!

New Engine: Quantum 3

The end of the SSM ?!?

Version “5”: Hierarchical SSMs for Quantum 3 (Wii, PSP, PS2, Windows)

- Heavily modified version of Argonaut's engine...
  - strat, strat, everything is a strat! (Brett Laming can back me up here)
  - proprietary scripting language (Steve Rabin ***loves*** these!)
  - language supports co-routines, very *very* rapid prototyping and “simple” entity development
- New life as a “FSM add-on” for Quantum
- Shipped and in-use on ***The Conduit***, others

☺ now an optional use element: no longer required

☹ integrated engine tool support gone…

  - (engine still had agent labeling, debug text, etc.)

The Bottom Line(s)

Should you do this? That depends...

- Evolving a system and tools applies to any technique
- How familiar are you with other methods? Will you have time to get that understanding?
- How “expert” will the users of this system be? Will it be only the author or lots of variable experience engineers?

- Don't underestimate having a good tool !!

(more) Bottom Line(s)

Does this apply to more modern techniques? Yes!

- Meta FSM selecting from more interesting child controllers
- Continued use for more deterministic game elements even when core agents have “moved on”
- Use for games where more determinism desired
- Many animation or audio controller / blending systems are based upon FSMs or HPFSMs
- Low power platforms like phones, flash games, etc.

Let's rap about AI tools!

A shameless plug for:

Technical Issues in Tools Development

round-tables

(which I just happen to chair)

Thur: 4:30-5:30 || Fri: 1:30-2:30 || Sat: 9-10

John Walker -- john@w4lker.net