



The Game Development Process

Game Programming




WORCESTER POLYTECHNIC INSTITUTE



Outline

- Teams and Processes
- Select Languages
- Debugging
- Misc (as time allows)
 - AI
 - Multiplayer



WORCESTER POLYTECHNIC INSTITUTE



Introduction

- Used to be programmers created games
 - But many great programmers not great game makers
- With budget shift, emphasis has shifted
 - Game content creators are artist and designers
- Programmers can be thought of as providing services for content
 - But fate of entire game rests in their hands

Based on Chapter 3.1, *Introduction to Game Development*



Programming Areas - Game Code

- Everything directly related to game itself
 - How camera behaves, score is kept, AI for bots, etc.
- Often in scripting language (rest is in C++, more on languages next)
 - Produce faster iterations
 - Allow technical designers/artists to change behaviors
 - More appropriate language for domain (ex: AI probably not easiest in C++)

Based on Chapter 3.1, *Introduction to Game Development*





Programming Areas - Game Engine

- Support code that is not game specific
 - More than just drawing pretty 3d graphics (that is actually the graphics engine, part of the game engine)
 - Isolate game code from hardware
 - ex: controller, graphics, sound
 - Allows designers to concentrate on game
 - Common functionality needed across game
 - Serialization, network communication, pathfinding, collision detection

Based on Chapter 3.1, *Introduction to Game Development*



Programming Areas - Tools

- Most involve content creation
 - Level editors, particle effect editors, sound editors
- Some to automate repetitive tasks (ex: convert content to game format)
 - These usually have no GUI
- Sometimes written as plug-ins for off-the-shelf tools
 - Ex: extensions to Maya or 3dStudio or Photoshop
- If no such extension available, build from scratch

Based on Chapter 3.1, *Introduction to Game Development*





Programming Team Organization

- Programmers often *specialize*
 - Graphics, networking, AI
- May be *generalists*, know something about everything
 - Often critical for "glue" to hold specialists together
 - Make great lead programmers
- More than 3 or 4, need some organization
 - Often lead programmer, much time devoted to management
- More than 10 programmers, several leads (graphics lead, AI lead, etc.)

Based on Chapter 3.1, *Introduction to Game Development*



Software Methodologies

- Code and Fix
- Waterfall
- Iterative
- Agile

- (Take cs3733, Software Engineering)





Methodologies - Code and Fix

- Really, lack of a methodology
 - And all too common
- Little or no planning, diving straight into implementation
- Reactive, no proactive
- End with bugs. If bugs faster than can fix, "death spiral" and may be cancelled
- Even those that make it, must have "crunch time"
 - viewed after as badge of honor, but results in burnout

Based on Chapter 3.1, *Introduction to Game Development*



Methodologies - Waterfall

- Plan ahead
- Proceed through various planning steps *before* implementation
 - requirements analysis, design, implementation, testing (validation), integration, and maintenance
- The waterfall loops back as fixes required
- Can be brittle to changing functionality, unexpected problems in implementation
 - Going back to beginning

Based on Chapter 3.1, *Introduction to Game Development*





Methodologies - Iterative

- Develop for a period of time (1-2 months), get working game, add features
 - Periods can coincide with publisher milestones
- Allows for some planning
 - Time period can have design before implementation
- Allows for some flexibility
 - Can adjust (to new technical challenges or producer demands)

Based on Chapter 3.1, *Introduction to Game Development*



Methodologies - Agile

- Admit things will change, avoid looking too far in the future
- Value simplicity and the ability to change
- Can scale, add new features, adjust
- Relatively new for game development
- Big challenge is hard to convince publishers

Based on Chapter 3.1, *Introduction to Game Development*





Common Practices - Version Control

- Database containing files and past history of them
- Central location for all code
- Allows team to work on related files without overwriting each other's work
- History preserved to track down errors
- Branching and merging for platform specific parts

Based on Chapter 3.1, *Introduction to Game Development*



Common Practices - Quality (1 of 2)

- *Code reviews* - walk through code by other programmer(s)
 - Formal or informal
 - "Two eyes are better than one"
 - Value is programmer aware others read
- *Asserts*
 - Force program to crash to help debugging
 - Ex: Check condition is true at top of code, say pointer not NULL before following
 - Removed during release

Based on Chapter 3.1, *Introduction to Game Development*



Common Practices - Quality (2 of 2)

- *Unit tests*
 - Low level test of part of game (Ex: see if physics computations correct)
 - Tough to wait until very end and see if bug
 - Often automated, computer runs through combinations
 - Verify before assembling
- *Acceptance tests*
 - Verify high-level functionality working correctly (Ex: see if levels load correctly)
- Note, above are programming tests (ie- code, technical). Still turned over to testers that track bugs, do gameplay testing.
- *Bug database*
 - Document and track bugs
 - Can be from programmers, publishers, customers
 - Classify by severity
 - Keeps bugs from falling through cracks
 - Helps see how game is progressing

Based on Chapter 3.1, *Introduction to Game Development*



Outline

- Teams and Processes (done)
- Select Languages (next)
- Debugging
- Misc (as time allows)
 - AI
 - Multiplayer



C++ (1 of 3)

- Mid-late 1990's, C was language of choice
- Since then, C++ language of choice for games
 - First commercial release in 1985 (AT&T)
- List *pros* (+) and *cons* (-)
- (Take *cs2102 OO Design Concepts* or *cs4233 OOAD*)
- + C Heritage
 - Learning curve easier
 - Compilers wicker fast
- + Performance
 - Used to be most important, but less so (but still for core parts)
 - Maps closely to hardware (can "guess" what assembly instructions will be)
 - Can not use features to avoid cost, if want (ie- virtual function have extra step but don't have to use)
 - Memory management controlled by user

Based on Chapter 3.2, *Introduction to Game Development*



C++ (2 of 3)

- + High-level
 - Classes (objects), polymorphism, templates, exceptions
 - Especially important as code-bases enlarge
 - Strongly-typed (helps reduce errors)
 - ex: declare before use, and `const`
- + Libraries
 - C++ middleware readily available
 - OpenGL, DirectX, Standard Template Library (*containers*, like "vectors", and *algorithms*, like "sort")

Based on Chapter 3.2, *Introduction to Game Development*



C++ (3 of 3)

- Too Low-level
 - Still force programmer to deal with low-level issues
 - ex: memory management, pointers
- Too complicated
 - Years of expertise required to master (other languages seek to overcome, like Java and C#)
- Lacking features
 - No built-in way to look at object instances
 - No built-in way to serialize
 - Forces programmer to build such functionality (or learn custom or 3rd party library)
- Slow iteration
 - Brittle, hard to try new things
 - Code change can take a looong time as can compile

Based on Chapter 3.2, *Introduction to Game Development*



C++ (Summary)

- When to use?
 - Any code where performance is crucial
 - Used to be all, now game engine such as graphics and AI
 - Game-specific code often *not* C++
 - Legacy code base, expertise
 - When also use middle-ware libraries in C++
- When not to use?
 - Tool building (GUI's tough)
 - High-level game tasks (technical designers)

Based on Chapter 3.2, *Introduction to Game Development*





Java (1 of 3)

- Java popular, but only recently so for games
 - Invented in 1990 by Sun Microsystems
- + Concepts from C++ (objects, classes)
 - Powerful abstractions
- + Cleaner language
 - Memory management built-in
 - Templates not as messy
 - Object functions, such as virtualization
- + Code portability (JVM)
(Hey, draw picture)
- + Libraries with full-functionality built-in

Based on Chapter 3.2, *Introduction to Game Development*



Java (2 of 3)

- Performance
 - Interpreted, garbage collection, security
 - So take 4x to 10x hit
 - + Can overcome with JIT compiler, Java Native Interface (not interpreted)
- Platforms
 - JVM, yeah, but not all games (most PC games not, nor consoles)
 - + Strong for browser-games, mobile

Based on Chapter 3.2, *Introduction to Game Development*



Java (3 of 3)

- Used in:
 - Downloadable/Casual games
 - PopCap games
 - *Mummy Maze, Seven Seas, Diamond Mine*
 - Yahoo online games (WorldWinner)
 - *Poker, Blackjack*
 - PC
 - *Star Wars Galaxies* uses Java (and simplified Java for scripting language)
 - *You Don't Know Jack* and *Who Wants to be a Millionaire* all Java

Based on Chapter 3.2, *Introduction to Game Development*



Scripting Languages (1 of 3)

- Not compiled, rather specify (script) sequence of actions
- Most games rely upon some
 - Trigger a few events, control cinematic
- Others games may use it lots more
 - Control game logic and behavior (Game Maker has GML)
- + Ease of development
 - Low-level things taken care of
 - Fewer errors by programmer
 - But script errors tougher, often debuggers worse
 - Less technical programming required
 - Still, most scripting done by *programmers*
 - Iteration time faster (don't need to re-compile all code)
 - Can be customized for game (ex: just AI tasks)

Based on Chapter 3.2, *Introduction to Game Development*



Scripting Languages (2 of 3)

- + Code as an asset
 - Ex: consider Peon in C++, with behavior in C++, maybe art as an asset. Script would allow for behavior to be an asset also
 - Can be easily modified, even by end-user in "mod"
- Performance
 - Parsed and executed "on the fly"
 - Hit could be 10x or more over C++
 - Less efficient use of instructions, memory management
- Tool support
 - Not as many debuggers, IDEs
 - Errors harder to catch
- Interface with rest of game
 - Core in C++, must "export" interface
 - Can be limiting way interact
 - (Hey, draw picture)

Based on Chapter 3.2, *Introduction to Game Development*



Scripting Languages (3 of 3)

- *Python*
 - Interpreted, OO, many libraries, many tools
 - Quite large (bad when memory constrained)
 - Ex: *Blade of Darkness*, *Earth and Beyond*, *Eve Online*, *Civilization 4* (Table 3.2.1 full list)
- *Lua* (pronounced: *Loo-ah*)
 - Not OO, but small (memory). Embed in other programs. Doesn't scale well.
 - Ex: *Grim Fandango*, *Baldur's Gate*, *Far Cry* (Table 3.2.2 full list)
- Others:
 - *Ruby*, *Perl*, *JavaScript*
 - Custom: *GML*, *QuakeC*, *UnrealScript*
 - Implementing own tough, often performs poorly so careful!

Based on Chapter 3.2, *Introduction to Game Development*



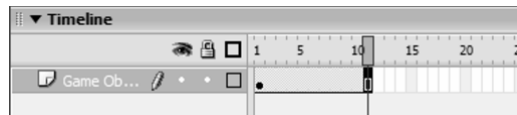
Macromedia Flash (1 of 2)

- More of a platform and IDE (ala *Game Maker*) than a language (still, has ActionScript)
 - "Flash" refers to 1) authoring environment, 2) the player, or 3) the application files
 - Released 1997, popular with Browser bundles by 2000
- Advantages
 - Wide audience (nearly all platforms have Flash player)
 - Easy deployment (embed in Web page)
 - Rapid development (small learning curve, for both artists and programmers)
- Disadvantages
 - 3D games
 - Performance (interpreted, etc.)

Based on Chapter 3.3, *Introduction to Game Development*



Macromedia Flash (2 of 2)



- Timeline Based
 - Frames and Frame rate (like animations)
 - Programmers indicate when (time) event occurs (can occur across many frames)
- Vector Engine
 - Lines, vertices, circles
 - Can be scaled to any size, still looks crisp
- Scripting
 - ActionScript similar to JavaScript
 - Classes (as of Flash v2.0)
 - Backend connectivity (load other Movies, URLs)



Based on Chapter 3.3, *Introduction to Game Development*





Outline

- Teams and Processes (done)
- Select Languages (done)
- Debugging (next)
- Misc (as time allows)
 - AI
 - Multiplayer



Debugging Introduction

- New Integrated Development Environments (IDEs) have debugging tools
 - Trace code, print values, profile
- But debugging frustrating
 - Beginners not know how to proceed
 - Even advanced can get "stuck"
- Don't know how long takes to find
 - Variance can be high
- Mini-outline
 - 5-step debugging process
 - Debugging tips
 - Touch scenarios and patterns
 - Prevention

Based on Chapter 3.5, *Introduction to Game Development*





Step 1: Reproduce the Problem Consistently

- Find case where always occurs
 - "Sometimes game crashes after kill boss" doesn't help much
- Identify steps to get to bug
 - Ex: start single player, skirmish map 44, find enemy camp, use projectile weapon ...
 - Produces systematic way to reproduce

Based on Chapter 3.5, *Introduction to Game Development*



Step 2: Collect Clues

- Collect clues as to bug
 - But beware that some clues are false
 - Ex: if bug follows explosion may think they are related, but may be from something else
 - Ex: if crash using projectile, what about that code that makes it possible to crash?
- Don't spend too long, get in and observe
 - Ex: see reference pointer from arrow to unit that shot arrow should get experience points, but it is NULL
 - That's the bug, but why is it NULL?

Based on Chapter 3.5, *Introduction to Game Development*



Step 3: Pinpoint Error

- Propose a hypothesis and prove or disprove
 - Ex: suppose arrow pointer corrupted during flight. Add code to print out values of arrow in air. But equals same value that crashes. *Wrong*. But have new clue.
 - Ex: suppose unit deleted before experience point. Print out values of all in camp before fire and all deleted. *Yep, that's it.*
- Or, divide-and-conquer method (note, can use in conjunction with hypo-test above, too)
 - Sherlock Holmes "when you have eliminated the impossible, whatever remains, however improbably, must be the truth"
 - Setting breakpoints, look at all values, until discover bug
 - The "divide" part means break it into smaller sections
 - Ex: if crash, put breakpoint $\frac{1}{2}$ way. Is it before or after? Repeat.
 - Look for anomalies, NULL or NAN values

Based on Chapter 3.5, *Introduction to Game Development*



Step 4: Repair the Problem

- Propose solution. Exact solution depends upon stage of problem.
 - Ex: late in code cannot change data structures. Too many other parts use.
 - Worry about "ripple" effects.
- Ideally, want original coder to fix. At least, talk with original coder for insights.
- Consider other similar cases, even if not yet reported
 - Ex: other projectiles may cause same problem as arrows did

Based on Chapter 3.5, *Introduction to Game Development*





Step 5: Test Solution

- Obvious, but can be overlooked if programmer is sure they have fix (but programmer can be wrong!)
- So, test that fix repairs bug
 - Best by independent tester
- Test if other bugs introduced (beware "ripple" effect)

Based on Chapter 3.5, *Introduction to Game Development*



Debugging Tips (1 of 3)

- *Question your assumptions* - don't even assume simple stuff works, or "mature" products
 - Ex: libraries can have bugs
- *Minimize interactions* - systems can interfere, make slower so isolate the bug to avoid complications
- *Minimize randomness* -
 - Ex: can be caused by random seed or player input. Fix input (script player) so reproducible

Based on Chapter 3.5, *Introduction to Game Development*





Debugging Tips (2 of 3)

- Break complex calculations into steps - may be equation that is at fault or "cast" badly
- Check boundary conditions - classic "off by one" for loops, etc.
- Disrupt parallel computations - "race conditions" if happen at same time (cs3013)
- Use debugger - breakpoints, memory watches, stack ...
- Check code recently changed - if bug appears, may be in latest code (not even yours!)

Based on Chapter 3.5, *Introduction to Game Development*



Debugging Tips (3 of 3)

- *Take a break* - too close, can't see it. Remove to provide fresh prospective
- *Explain bug to someone else* - helps retrace steps, and others provide alternate hypotheses
- *Debug with partner* - provides new techniques
 - Same advantage with code reviews, peer programming
- *Get outside help* - tech support for consoles, libraries, ...

Based on Chapter 3.5, *Introduction to Game Development*





Tough Debugging Scenarios and Patterns (1 of 2)

- Bug in *Release* but not in *Debug*
 - Often in initialized code
 - Or in optimized code
 - Turn on optimizations one-by-one
- Bug in *Hardware* but not in *Dev Kit*
 - Usually dev kit has extra memory (for tracing, etc.). Suggest memory problem (pointers), stack overflow, not checking memory allocation
- Bug Disappears when Changing Something Innocuous
 - Likely timing problem (race condition) or memory problem
 - Even if looks like gone, probably just moved. So keep looking

Based on Chapter 3.5, *Introduction to Game Development*



Tough Debugging Scenarios and Patterns (2 of 2)

- Truly Intermittent Problems
 - Maybe best you can do is grab all data values (and stack, etc) and look at ("Send Error Report")
- Unexplainable Behavior
 - Ex: values change without touching. Usually memory problem. Could be from supporting system. Retry, rebuild, reboot, re-install.
- Bug in Someone Else's Code
 - "No, it is not". Be persistent with own code first.
 - It's not in hardware. (Ok, very, very rarely, but expect it not to be). Download latest firmware, drivers.
 - If really is, best bet is to help isolate to speed them in fixing it.

Based on Chapter 3.5, *Introduction to Game Development*



Debugging Prevention (1 of 2)

- Understand underlying system
 - Knowing language not enough
 - Must understand underlying system
 - At least one level down
 - Engine for scripters
 - OS for engine
 - Maybe two layers down (hardware, assembly)
- Add infrastructure, tools to assist
 - Make general
 - Alter game variables on fly (speed up)
 - Visual diagnostics (maybe on avatars)
 - Log data (events, units, code, time stamps)
 - Record and playback capability

Based on Chapter 3.5, *Introduction to Game Development*



Debugging Prevention (2 of 2)

- Set compiler on highest level warnings
 - Don't ignore warnings
- Compile with multiple compilers
 - See if platform specific
- Write own memory manager (for console games, especially, since tools worse)
- Use asserts
- Always initialize when declared
- Indent code, use comments
- Use consistent style, variable names
- Avoid identical code - harder to fix if bug
- Avoid hard-coded (magic numbers) - makes brittle
- Verify coverage (test all code) when testing

Based on Chapter 3.5, *Introduction to Game Development*



Outline

- Teams and Processes (done)
- Select Languages (done)
- Debugging (done)
- Misc (as time allows)
 - AI (next)
 - Multiplayer



Introduction to AI

- Opponents that are challenging, or allies that are helpful
 - Unit that is credited with acting on own
- Human-level intelligence too hard
 - But under narrow circumstances can do pretty well (ex: chess and Deep Blue)
- Artificial Intelligence (around in CS for some time)

Based on Chapter 5.3, *Introduction to Game Development*





AI for CS different than AI for Games

- Must be smart, but purposely flawed
 - Loose in a fun, challenging way
- No unintended weaknesses
 - No "golden path" to defeat
 - Must not look dumb
- Must perform in real time (CPU the bottleneck)
- Configurable by designers
 - Not hard coded by programmer
- "Amount" and type of AI for game can vary
 - RTS needs global strategy, FPS needs modeling of individual units at "footstep" level
 - RTS most demanding: 3 full-time AI programmers
 - Puzzle, street fighting: 1 part-time AI programmer

Based on Chapter 5.3, *Introduction to Game Development*



AI for Games - Mini Outline

- Introduction (done)
- Agents (next)
- Finite State Machines
- Common AI Techniques
- Promising AI Techniques



Game Agents (1 of 2)

- Most AI focuses around game agent
 - think of agent as NPC, enemy, ally or neutral
- Loops through: sense-think-act cycle
 - Acting is event specific, so talk about sense+think
- *Sensing*
 - Gather current world state: barriers, opponents, objects
 - Needs limitations
 - Avoid "cheating" by looking at game data
 - Typically, same constraints as player (vision, hearing range)
 - Often done simply by distance direction (not computed as per actual vision)
 - Model communication (data to other agents) and reaction times (can build in delay)

Based on Chapter 5.3, *Introduction to Game Development*



Game Agents (2 of 2)

- *Thinking*
 - Evaluate information and make decision
 - As simple or elaborate as required
 - Two ways:
 - Precoded expert knowledge, typically hand-crafted "if-then" rules + randomness to make unpredictable
 - Search algorithm for best (optimal) solution

Based on Chapter 5.3, *Introduction to Game Development*





Game Agents - Thinking (1 of 3)

- Expert Knowledge
 - finite state machines, decision trees, ... (FSM most popular, details next section)
 - Appealing since simple, natural, embodies common sense
 - Ex: if you see enemy weaker than you, attack. If you see enemy stronger, then go get help
 - Often quite adequate for many AI tasks
 - Trouble is, often does not scale
 - Complex situations have many factors
 - Add more rules, becomes brittle

Based on Chapter 5.3, *Introduction to Game Development*



Game Agents - Thinking (2 of 3)

- Search
 - Look ahead and see what move to do next
 - Ex: piece on game board, pathfinding (ch 5.4)
- Machine learning
 - Evaluate past actions, use for future
 - Techniques show promise, but typically too slow
 - Need to learn and remember

Based on Chapter 5.3, *Introduction to Game Development*





Game Agents - Thinking (3 of 3)

- Making agents stupid
 - Many cases, easy to make agents dominate
 - Ex: bot always gets head-shot
 - Dumb down by giving "human" conditions, longer reaction times, make unnecessarily vulnerable
- Agent cheating
 - Ideally, don't have unfair advantage (such as more attributes or more knowledge)
 - But sometimes might to make a challenge
 - Remember, that's the goal, AI lose in challenging way
 - Best to let player know

Based on Chapter 5.3, *Introduction to Game Development*

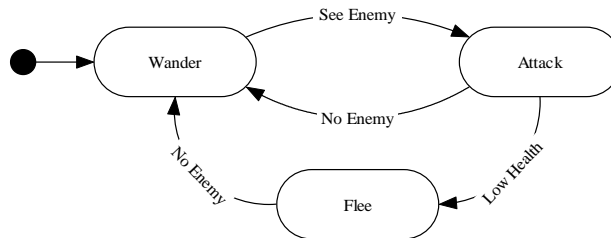


AI for Games - Mini Outline

- Introduction (done)
- Agents (done)
- Finite State Machines (next)
- Common AI Techniques
- Promising AI Techniques



Finite State Machines (1 of 2)



- Abstract model of computation
- Formally:
 - Set of states
 - A starting state
 - An input vocabulary
 - A transition function that maps inputs and the current state to a next state

Based on Chapter 5.3, *Introduction to Game Development*



Finite State Machines (2 of 2)

- Most common game AI software pattern
 - Natural correspondence between states and behaviors
 - Easy to diagram
 - Easy to program
 - Easy to debug
 - Completely general to any problem
- Problems
 - Explosion of states with more variables, conditions
 - Often created with ad hoc structure

Based on Chapter 5.3, *Introduction to Game Development*





Finite-State Machine: Approaches

- Three approaches
 - Hardcoded (switch statement)
 - Scripted
 - Hybrid Approach

Based on Chapter 5.3, *Introduction to Game Development*



Finite-State Machine: Hardcoded FSM

```
void RunLogic( int * state ) {
    switch( state )
    {
        case 0: //Wander
            Wander();
            if( SeeEnemy() ) { *state = 1; }
            break;

        case 1: //Attack
            Attack();
            if( LowOnHealth() ) { *state = 2; }
            if( NoEnemy() ) { *state = 0; }
            break;

        case 2: //Flee
            Flee();
            if( NoEnemy() ) { *state = 0; }
            break;
    }
}
```

Based on Chapter 5.3, *Introduction to Game Development*



Finite-State Machine: Problems with switch FSM

1. Code is ad hoc
 - Language doesn't enforce structure
2. Transitions result from polling
 - Inefficient - event-driven sometimes better
3. Can't determine 1st time state is entered
4. Can't be edited or specified by game designers or players

Based on Chapter 5.3, *Introduction to Game Development*



Finite-State Machine: Scripted with alternative language

```
AgentFSM
{
    State( STATE_Wander )
        OnUpdate
            Execute( Wander )
            if( SeeEnemy ) SetState( STATE_Attack )
        OnEvent( AttackedByEnemy )
            SetState( Attack )
    State( STATE_Attack )
        OnEnter
            Execute( PrepareWeapon )
        OnUpdate
            Execute( Attack )
            if( LowOnHealth ) SetState( STATE_Flee )
            if( NoEnemy ) SetState( STATE_Wander )
        OnExit
            Execute( StoreWeapon )
    State( STATE_Flee )
        OnUpdate
            Execute( Flee )
            if( NoEnemy ) SetState( STATE_Wander )
}
```

Based on Chapter 5.3, *Introduction to Game Development*





Finite-State Machine: Scripting Advantages

1. Structure enforced
2. Events can be handed as well as polling
3. OnEnter and OnExit concept exists
4. Can be authored by game designers
 - Easier learning curve than straight C/C++



Finite-State Machine: Scripting Disadvantages

- Not trivial to implement
- Several months of development
 - Custom compiler
 - With good compile-time error feedback
 - Bytecode interpreter
 - With good debugging hooks and support
- Scripting languages often disliked by programmers
 - Can never approach polish and robustness of commercial compilers/debuggers



Based on Chapter 5.3, *Introduction to Game Development*



Finite-State Machine: Hybrid Approach

- Use a class and C-style macros to approximate a scripting language
- Allows FSM to be written completely in C++ leveraging existing compiler/debugger
- Capture important features/extensions
 - OnEnter, OnExit
 - Timers
 - Handle events
 - Consistent regulated structure
 - Ability to log history
 - Modular, flexible, stack-based
 - Multiple FSMs, Concurrent FSMs
- Can't be edited by designers or players

Based on Chapter 5.3, *Introduction to Game Development*



Finite-State Machine: Extensions

- Many possible extensions to basic FSM
 - OnEnter, OnExit
 - Timers
 - Global state, substates
 - Stack-Based (states or entire FSMs)
 - Multiple concurrent FSMs
 - Messaging

Based on Chapter 5.3, *Introduction to Game Development*





AI for Games - Mini Outline

- Introduction (done)
- Agents (done)
- Finite State Machines (done)
- Common AI Techniques (next)
- Promising AI Techniques



Common Game AI Techniques

- Whirlwind tour of common techniques
- (See Ch 5.3, p 554)



Based on Chapter 5.3, *Introduction to Game Development*