

Declarative versus Imperative Paradigms in Games AI¹

Nathan Combs, Jean-Louis Ardoit

Independent, ILOG S.A.
ncombs@roaringshrimp.com
ardoint@ilog.fr

Abstract

Most game behavior is written using *scripts* instead of *rules*. Rules are declarative representations that, given variables in the game, encode relationships and facts about the game. From these facts one can reason about truth in the game world. Scripts are imperative representations: they provide a set of instructions that are used to process game variables and compute some conclusion. Scripting is more intuitive for game designers and developers to work with. It is easier to conceptualize and write a script that says: "go right, go left, turn around twice, go straight, then fire your gun..." then it is to formulate a set of rules that could shape an entities movement to the same effect.

While scripting allows developers to construct sophisticated behaviors, typically these behaviors are brittle outside of the environment for which they were designed. As game worlds grow in size and sophistication, a challenge for the game industry will be how to satisfy the increased demand for cost-effective, high quality content to place in those worlds.

We believe that games Artificial Intelligence (AI) and rule-based programming are important sources of games content for this future. This paper outlines the current design pattern of games AI, and introduces why current practices will likely need to change. Working from discussions within the Rule-based Systems (RBS) Workgroup of the Artificial Intelligence Interface Standards Committee (AIISC) of the International Game Developers Association (IGDA), this paper will present the general argument that rule-based programming is advantageous for scripting games AI. Furthermore, we suggest that RBS middleware may enhance the ability of developers to script AI more efficiently in the long term.

Environment-based Programming Style

Game AI development can be characterized by a three step design pattern. *First*, the developer specifies an environment – terrain, textures, a map, etc. *Second*, the developer specifies objects to embed in that environment: objects can be static, or fixed in that environment; objects can be dynamic, e.g. move around; or objects can signify episodic scenario behaviors, e.g. triggers. *Third*, in the case of dynamic objects, the developer can link them up to indicate where they should move, e.g. pathways. The

developer can coordinate objects and events graphically as well as embed logic with the objects - behavior is added to the objects by "drilling in" and associating scripts and/or changing property values.

While details differ across individual games and genres, the pattern is similar. For example, Operation Flashpoint (Codemasters) provides a graphical mission editor, whereas, Unreal Tournament uses a more complex CAD level-editor. This design pattern is an "*environment-based programming*" style that has its roots in early game systems (e.g., RuthMOO's "Programming the Environment") – where both the content and programming are intermingled.

An environment-based programming style may emphasize object-level scripting (inserting scripts into game world objects), or it may emphasize tools that can manipulate the top-level organization of those objects and events. The key point, however, is that both styles are consistent with each other, and both are imperative, rather than declarative, in their view of behavior design. This minimizes surprises by producing exact and repeatable behaviors that can be examined and debugged, whereas the alternative is to use a behavior model that is not explicit but somehow *emerges* from a set of rules.

An imperative program design lends itself well to easy-to-create and testable AI. In contrast, developing rules from which behaviors emerge is harder. Yet, rules, are likely to be more scalable in the long-term. Rules are declarative and as such they represent relationships that can be generalized across individual environments and scenarios. In contrast, the imperative nature of the environment-based programming style tends to contextualize the logic and prevents its re-use across scenarios, and it also leaves the logic brittle to changes in the scenario.

In Figure 1. we see a representative example mission editor. This illustration is simple but is suggested to be representative of a class of 2D mission editors commonly seen. With these editors a graphical tool is used to layout game entities in a map and to construct their movement in that space using modeling elements such as waypoints and

¹ Last revised: June 2005

attacker; try to run from an attacker much stronger than you... etc.

At recent Game Developer Conferences speakers (e.g. Peter Molyneux, Will Wright) indicated that games AI will play an increasingly important role - as the source of "dynamics" and "emergent behaviors" that leads to new (generated/emergent) content within games. We also believe that this trend will drive game development towards more sustainable "rules" based programming styles.

In the games community a *script* loosely refers to a relatively easily modifiable program for implementing game behavior. It may be written in a "glue" language that rides above lower-level functions. For example, consider these words from a forum discussion (Game Design X):

"...have some gnarly C code... You expose functions to the scripting language; these functions do the heavy lifting and the object-oriented script provides the creative element."

To the games industry, scripting is a means of partitioning concerns between the developers and the level (content) designers. Doing so implies these advantages (Matheson):

1. Means a coder is concerned in writing engine/tool code, rather than game logic...
2. Designers like to be able to 'twiddle' with things. Scripting allows them easy access to this functionality. It also allows them more flexibility to try things out in the level that they normally would have to get a coder involved with.
3. You don't have to re-compile if you want to change functionality in the game. Simply modify the script.
4. You want to break the tie between engine code and game code. They should be two separate pieces. That way, it's easy to use the engine for multiple games (hopefully).

Scripting is favored in games AI because it allows content developers to quickly create and tweak the behaviors of game objects. The online FAQ for a massively multiplayer game (Eve-Online), for example, describes the advantages of using a variant of the Python programming language (Stackless Python) on their game servers. The advantage they emphasize is its ability to allow designers to write lots of independent bits of behavior. In other words it is seen as productivity tool for designers:

...Our game logic scripters are thereby freed from many of the mundane tasks associated with models that don't benefit from the control structures provided by Stackless. The creative

process of writing interesting game behavior is no longer bogged down by software or system limitations.

Scripts can be either compiled or interpreted. Typically, their language form falls into one of two categories: *Imperative*, or *Declarative*. Imperative scripts share properties with imperative programming languages:

- Implicit state: variables
- State modification through assignment: side effecting
- Instruction sequencing (begin-end blocks, loops,...)

Whereas declarative scripts share properties with declarative programming languages:

- No implicit state, no assignments
- Expression evaluation instead of instruction sequencing
- Chaining (recursion) instead of loops
- Can be functional

Scripts of the imperative form are most common with games. A poll conducted at a game developer's site (GameDev.Net), for example, identified these imperative languages: Lua, "C", and Python as the most commonly used scripting languages (excluding "I made my own").

In the following pseudo-code fragment of a script from Morrowind (Bethesda Softworks), the pattern we described earlier is clear: a game map is assumed and is populated with objects that are reified with scripts to respond to events (e.g. in this case, a non-player character transaction):

```
// Loop through all the Player Character's items..
// reward her, if she has a goblin ear...
While( obj= getEachItem() ) {
    IsA(obj, "goblin ear") {
        // Give 10 Gold Pieces to Player Character...
        GiveGoldToPlayerCharacter(playerChar, 10);
        // Give 100 XP to Player Character
        GiveXPToPlayerCharacter(playerChar, 100);
        // Take the goblin ear from the player character
        // and throw away...
        Destroy(obj);
    }
}
```

One can consider rules as a special declarative form whose statements are in a *Condition -> Action* format. Declarative scripts are occasionally seen in games. Anecdotally, they are more likely to be seen with "strategy" games whose logic tends to be globally applicable - across individuals and map locations. For example, the rule for selling excess resources in Age of

Empires II (Ensemble Studios) is given below. This rule can be easily applied to all of the game AI players.

```
// Rule to sell excess resources
(defrule
  (wood-amount > 1200)
  (or (food-amount < 1600)
      (or (gold-amount < 1200)
          (stone-amount < 650))))
  (can-sell-commodity wood)
  =>
  (chat-local-to-self "excess wood")
  (release-escrow wood)
  (sell-commodity wood))
```

A declarative model of game behaviors describes the relationships of the elements within a game. Whereas, an imperative model prescribes a process for computing the behavior of those entities. The former describes the *what*, leaving the *how* implicit. The latter prescribes the *how*, leaving the *what* implicit.

There are a number of reasons why shifting from an imperative style towards a declarative style may benefit games AI development long-term. The reasons are on grounds of *pragmatics*, of *scalability*, of *usability*, and of *logic expressiveness*.

A declarative style is more *pragmatic* than an imperative one because it separates implementation from the logic of the behavior. This can lead to more maintainable engineering and code. Furthermore, a rules-based representation provides a more concise and direct relationship between specification and implementation that would simplify testing. Separating the game engine from the game rules allows independent simulation and testing of each (for related discussion see Combs – note supporting comments by online game executives). Analogously, one could develop a declarative annotation or specification (for an imperative language, say) that is then analyzed, e.g. a discussion here with first order predicates (Makela). The problem with this is that it is cumbersome and prone to error, e.g. the specification can drift from the implemented behavior.

A declarative style can lead to more *scalable AI* in that it makes it easier to separate logic about type or class from logic pertaining to the instance. So, for example, rules about *cars* and *8_cylinder_engines*, can be left distinct from logic about *my_red_ford_thunderbird*. This facilitates componentization of the building blocks of games AI. Separating the logic from the implementation also enables reasoning about the logic by itself:

- Is it complete?
- Are all rules reachable?
- Can we use look-ahead?

- Can learning algorithms be applied?

For example, rules promise to be more easily validated by analyzing their conditions and actions than by examining script code. Consider rule subsumption: under what circumstances is a rule subsumed by another rule? This is in contrast to a much more difficult (and often likely intractable) analysis proposition: under what circumstances is a Lua/Python/C script fragment subsumed by another Lua/Python/C script fragment?

Beyond scaling the AI, a declarative style of programming where logic maintained separately from code is likely more *usable* to mod developers - witness this trend with business software. Increasingly games builders are looking to appeal to *4th party developers* (Sawyer) to develop outside content to extend the life of the product as well as to broaden its appeal. Such, we speculate, could lead to a “virtuous cycle” equivalent to one that developed in the applications sector. There we saw emerge for commercial use tools geared towards development of large rule-based systems (e.g. ILOG) as well as products such event simulators that support testing application logic independent of the middleware implementation (e.g. Tivoli). We anticipate that equivalent tools for game-oriented rule-systems can be developed.

The *expressive* need for both declarative and imperative forms is straightforward: sometimes it is just easier to think in rules and compute consequences; sometimes it is vice versa. Consider two different approaches for specifying behavior: the first approach (imperative) is to describe the *consequences* or the *process* first; the second (declarative) is to describe the *goals* or *rules* first:

A.) "I want Buck Rogers to run here and fire his gun" (*process, consequences first*)

B.) "Sun-tzu said (The Art of War): The grounds are accessible, entrapping, stalemated, narrow, steep, and expansive... For entrapping ground, if the enemy is unprepared, advance and defeat him." (*goals, rules first*)

Developers shouldn't have to write in a style of (A.) to say (B.). Both language styles have their place, depending upon what needs to be said. (A.) is preferable to express behaviors that are specific to location, objects, and scenario. (B.) is preferable for general behaviors that transcend individual scenarios, objects and instances.

The ability to interchange rules with scripts is useful for a final reason. It will help designers and developers choose the right language to build behaviors for their game. More options can lead to better trade-offs when considering game design, the technical platform, etc.

Wright and Marshall (2000) suggest that mixing imperative and declarative forms can improve the *expressiveness of game logic* by helping design of *different* (and more appropriate) representations:

... scripting languages for game AI based on procedural languages, such as Java or subsets of C, are misguided: they are too similar to the main game language. The extra complexity introduced to the development process is not offset by any new language advantages. And any benefits obtained from the extra level of indirection between AI code and game engine code introduced by using a procedural scripting language can be also be obtained by simply implementing a good AI/game engine interface.

Keep in mind that a hybrid between imperative games AI scripting and a declarative approach could be found with *functional declarative* programming styles. A common pattern in games AI is to delegate low-level state and computation to the game-engine and reserve high-level logic (and state) to the scripting language. A functional programming style while descriptive (declarative) can indicate ways of computing through the function semantics – thereby preserving some of the imperative control. In this way a functional declarative language can represent a useful bridge between the declarative and imperative styles.

Convergence of Scripts and Rules?

Building upon Wright and Marshall's suggestion to mix imperative and declarative forms for maximum effect - can we assert a stronger claim: that perhaps rules processing and script processing can be merged? Consider the earlier example (Age of Empires II), where scripting can take on a simple rules form. Baldur's Gate has also been cited as an example of "a rules-based approach that operates in a strictly linear fashion" (Woodcock).

At the level of integration, scripting and rules pose similar problems. They both have to relate to the game engine. Is their relationship synchronous? Where is the game state? Will they support functional call-backs? Etc. Should the game engine be able to reach-in and tune performance? In the AIISC working group we have hypothesized how these approaches might ultimately converge behind an interface rooted in an industry standard, JSR-94 (Java Specification Requests). We start with the view that an RBS is a rules-engine that (words adapted from JSR-94):

1. Acts as an if/then statement interpreter. Statements are rules.
2. Promotes declarative programming by externalizing ...game logic.

3. Acts upon input objects to produce output objects. Input objects are often referred to as facts and are a representation of the state of the ...game. Output objects can be thought of as conclusions or inferences and are grounded by the game in the... game domain.
4. Executes actions directly and affect the game, the input objects, the execution cycle, the rules, or the rule engine.
5. Creates output objects or may delegate the interpretation and execution of the output objects to the caller.

We believe that a JSR-94 based interface can support a range of RBS middleware (more powerful to less powerful RBS components). An RBS can script game logic in a functional declarative style language that is more familiar to current game designers (versus other declarative expressions). Evaluating rules against an RBS in a stateless session (e.g. JSR-94 interface) can be *functional* when the rules call functions in the game engine (call-backs). Thus, one might choose a functional declarative rules language to serve as an entry-point into rules-based scripting for games.

From the applications sector there is a precedent for merging rules with scripting. This sector has evolved a range of products that uses rules-based scripting to customize middleware (e.g. for two different examples, see Kozlenkov et. al., and Siliware Rules Engine). Consider large system architectures servicing business processes. Typically such architectures integrate a range of products and contain much "glue code" (including scripts) whose behavior is conditioned on the values in the data as it passes through, e.g. real-time data feeds etc. Such systems often represent metadata using rules: by separating the representation from the implementation (code) it simplifies maintenance.

Where might we see the convergence between declarative and imperative forms first in the games industry? In the AIISC working group we considered the possibility that it might occur first with server based games (e.g. online multiplayer games), for a couple of reasons. First, server architectures tend to be already largely componentized (e.g. database, login, etc.) - adding a new RBS component on the server is likely more feasible than on highly optimized clients. Second, server-resident games share a number of requirements with commercial server-applications, e.g., performance, scalability, and supporting dynamic loading and "hot swapping" of AI logic to achieve *agility* (Sinur).

While rules-scripting for single-seat games can be simplistic (e.g., the earlier Age of Empires II example) – RBS integration with multiplayer online game servers will require considerations beyond a JSR-94 based interface. For example, discussion on the Wworld-tech mailing list

suggests that RBS optimizations that work well in commercial business domains may need to be adapted for use in online games (e.g. Rete-based rule matching). By first agreeing upon the interface, however, the games industry can then let the middleware providers compete on specific solutions.

Conclusions

Games AI will be important sources of games content for this future. Rules and more generally declarative scripting practices will enhance the ability of game developers and designers to script AI more efficiently in the long term. Online multiplayer games are likely to be the first beneficiaries of rule-based scripting. However, the industry as a whole can benefit by increased use of declarative programming practices. We also hypothesize upon the convergence of declarative and imperative scripting behind a common interface.

Acknowledgements

Special thanks to Alexander Nareyek and to the members of the Rule-Based Systems Workgroup of the Artificial Intelligence Interface Standards Committee (AIISC) of the International Game Developers Association.

References

AIISC of the AI SIG of the International Game Developers Association. Website URL: <http://www.igda.org/ai/>

Bethesda Softworks LLC. The Elder Scrolls (Morrowind). Website URL: <http://www.elderscrolls.com/index.php>

Bioware Corp. Baldur's Gate. Website URL: http://www.bioware.com/games/baldurs_gate/

CodeMasters. Operation Flashpoint. Website URL: <http://www.codemasters.com/flashpoint/front.htm>

Combs, Nathan. *Can We Save MMOGs From Ourselves (Using Simulation)*. March 3, 2004. Article on Terra Nova. Website URL: <http://terranova.blogs.com/>

Ensemble Studios. *Computer Player Strategy Builder Guide, AI Expert Documentation for Age of Empires II: The Age of Kings*.

Eve-Online, FAQ. 1997-2004. Website URL: http://www.eve-online.com/faq/faq_07.asp

Game Design X Forum. Jan 2, 2004. Website URL: <http://p215.ezboard.com/fgamedesignxfrm7.showMessage?topicID=1.topic>

GameDev.Net. Poll Results: Which language do you use for scripting in your game engine? 1999-2004. Website URL: <http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163>

ILOG JRules™ 4.6 *Rule Builder Tutorial*. February 2004. Website URL: <http://www.ilog.com/>

Java Specification Requests. *JSR 94: Java™ Rule Engine API*. Website URL: <http://www.jcp.org/en/jsr/detail?id=94>

Kozlenkov, Alex, Michael Schroeder. *Prova: A Language for Rule-based Java Scripting, Data Integration, and Agent Programming*. Website URL: <http://comas.soi.city.ac.uk/prova/>

Lua.org. Website URL: <http://www.lua.org>

Makela, Sami. *NPC Scripting and Reasoning about the NPC behavior*. November 2001, World Forge newsletter. Website URL: http://www.worldforge.org/project/newsletters/November2001/NPC_Scripting

Matheson, A. *Why a Scripting Language*. Website URL: <http://gamestudies.cdis.org/~amatheson/writing/LUA-Part01/Part01-section02.html>

Molyneux, Peter. *AI: Gameplay & Design: A Marriage of Heaven or Hell?* Presentation at Game Developer's Conference. March 24, 2004.

Polge, Steven. *Unreal Tournament 2003 AI for Level Designers (Version 2)*. Feb. 15 2004. Website URL: <http://unreal.epicgames.com/ut2ai.htm>

Reynolds, Craig. *Steering Behaviors for Autonomous Vehicles*. Website URL: <http://www.red3d.com/cwr/steer/>

RuthMOO: Programming the Environment. Website URL: <http://confabulation.com/~sam/ruthmoo/progging.html>

Sawyer, Ben. *The Next Ages of Game Development*. Sept 30 2002. Website URL: <http://www.avault.com/developer/avscreenshot.asp?pic=bsawyer1&num=1>

Siliware Rules Engine. Website URL: <http://design.iwebsight.com/siliware/rules-engine.htm>

Sinur, J. *Architecting Agility With Business Rules*, COM-19-9972, Gartner report, Research Note 2003

Stackless Python: *A Python Implementation That Does Not Use The C Stack*. Website URL: <http://www.stackless.com/>

Sun-Tzu *The art of war* BC. Website URL: <http://www.sonshi.com/sun10.html>

Tivoli (IBM) Rule Builder's Guide. Website URL: http://publib.boulder.ibm.com/tividd/td/tec/GC32-0669-01/en_US/HTML/RBGmst40.htm

Vworld-tech Mailing List. *VWorld Ontology*. April 14, 2004. Website URL: <http://lists.puremagic.com/pipermail/vworld-tech/>

Woodcock, Steve. *Games Making Interesting Use of Artificial Intelligence Techniques*. Website URL: <http://www.gameai.com/games.html#BGATE>

Wright, W. *Dynamics for Designers*. Game Developers Conference, 2003.

Wright, I. P. & Marshall, J. A. R. (2000) *RC++: a rule-based language for game AI*. In: Proceedings of the First International Conference on Intelligent Games and Simulation (GAME-ON 2000). SCS Europe BVBA
