16

Plumbing the Forbidden Depths

Scripting and AI

Mike Lewis

16.1 Introduction

Lurking deep in the mists of contemporary artificial intelligence work is a controversial and hotly debated technique—a "black magic" that has been rumored to accomplish fantastical things, but at perilous cost. This forbidden ability has been responsible for some of the most cherished game AI in the industry's history … and also for some of its worst embarrassments. I speak of scripted AI.

Scripting, however, need not be feared. Like any good black art, it must be understood and mastered to be wielded safely [Tozour 02]. Here, we will explore the good, bad, and dangerous aspects of scripting as it pertains to developing game AI—all with an eye towards delivering the kind of dynamic, adaptive, challenging-but-entertaining experiences that modern gamers demand.

There are, of course, a multitude of ways to integrate scripting techniques into a game's architecture, but for the most part they comprise two basic philosophies. These rival points of view can be thought of as the "master" and "servant" ideologies. While both have their proper place, it can be tremendously useful to consider which role scripts will play in a given game's implementation. It is also of paramount importance to maintain as much adherence to that role as possible, for reasons which will be considered later.

By the same token, there is a vast spectrum of complexity in scripting systems, ranging from simple tripwire/response mechanisms to full-blown programming languages. Once

again, each possible approach has its merits and drawbacks. As with any technology or tool, the situation at hand must be carefully examined and considered in order to decide how best to deploy scripting systems and what their implementations should look like.

16.2 The Master and the Servant

From the very beginning of working with an AI scripting mechanism, it is important to understand the system's place in the overall architecture of the game—not just how it interacts with other AI features, but how it will function in terms of the totality of the game's systems. Moreover, it is deeply beneficial to remain true to that decision for the duration of development. Without this discipline, scripts can quickly become unruly monsters that suck up huge amounts of debugging time, deliver subpar gameplay experiences, or even drag down development as a whole.

The most successful approaches to scripting will generally fall into one of two camps. First is the "scripts as master" perspective, wherein scripts control the high-level aspects of agent decision making and planning. The other method sees "scripts as servant," where some other architecture controls the overall activity of agents, but selectively deploys scripts to attain specific design goals or create certain dramatic effects.

In general, master-style systems work best in one of two scenarios. In the optimal case, a library of ready-to-use tools already exists, and scripting can become the "glue" that combines these techniques into a coherent and powerful overarching model for agent behavior. Even in cases where such a library is not immediately available, master scripts are often the superior approach when it is known in advance that agents must be able to fluidly transition between a number of different decision-making or planning techniques.

By contrast, servant scripts are most effective when design requires a high degree of specificity in agent behavior. This is the typical sense in which interactions are thought of as "scripted"; a set of possible scenarios is envisioned by the designers, and special-case logic for reacting to each scenario is put in place by the AI implementation team. Servant scripts need not be entirely reactive, however; simple scripted loops and behavioral patterns can make for excellent ambient or "background" AI.

Most game designs will naturally lend themselves to one side or the other of the master/servant divide. Even when the game itself does not clearly lean towards a preferred approach, the human factor almost always will. Different teams will work more effectively in one style or another, depending on any number of circumstances: the ratio of programmers to designers, relative experience of the engineers working on the AI, time and budget constraints, and so on.

One such factor worth considering is the investment required to implement either approach. A master system will tend to work best with developers who can draw upon a diverse bag of tricks to help flesh out the overall implementation of the AI. Servant systems are much easier to design and implement in terms of code, but require extra care from gameplay design to avoid falling into the classic trap of producing agents that feel like cardboard cutouts.

Alternative philosophies surely exist, but they can be difficult to tame. Interspersing "scripted" logic with other architectural approaches is often a recipe for creating exactly the kind of rigid, inflexible agents that have (rightfully) earned scripting a bad reputation. Without clear boundaries and responsibilities, different systems will begin competing for

dominance—either at an abstract level in the code's design and details, or in the worst case, while the game is being played.

16.2.1 Scripts as Benevolent Overlords

The master-script philosophy derives its power from one fundamental principle: delegation. It is not the responsibility of the script to dictate every detail of how an agent behaves, or even to anticipate every possible scenario that might occur during the game experience. Instead, a good master script seeks to categorize and prioritize, and hand off responsibility for the mundane details to other systems.

Categorization is all about recognizing the nature of what is going on in the simulation at a given moment. Knowledge representation is often the key to doing this effectively; it is mandatory that agents have coherent and believable ideas about the world they inhabit so they can interpret events in a manner that is consistent with their worldview, and thereby take appropriate action. Note that it is not necessary for agents to have *accurate* or *precise* beliefs, so long as they do things which seem *sensible* from the perspective of the player. It can be tempting to build highly complex knowledge systems, but this often leads to inscrutable decision-making processes that wind up feeling arbitrary or mysterious to an outside observer.

Prioritization, on the other hand, boils down to choosing what to do with the gathered knowledge of an agent's surroundings and situation. For the master script, this is not a matter of selecting behaviors or states per se. Rather, during prioritization, the master script examines a set of lower-level control mechanisms, decides which are most appropriate for the moment, and selects one or more of those mechanisms to contribute towards the final decision-making process.

For example, consider a typical open-world role-playing game in which a large number of agents populate a town. The master script shouldn't concern itself too much with the minutiae of what an agent happens to be doing at any given moment. Instead, it acts like a sort of abstract state machine, governing which systems might control an individual agent at any point in time.

During peaceful spells, the master may choose between simple idle animation loops for a stationary blacksmith agent or perhaps it might assign a utility-based system to monitor the desires and needs of that agent and select activities based on those demands. The script might gather a group of agents to take on the role of the town militia, patrolling for nearby threats and policing the streets; these agents might be controlled using group tactical reasoning and movement systems, whereas a stray dog might simply wander around randomly and attempt to eat various things it encounters in the world.

All of this is simply prioritization. The master oversees the "peaceful" environment and delegates responsibility for agents to specific subsystems. This works elegantly alongside traditional subsumption architectures, where the details of an agent's activities are divided into increasingly abstract layers. At the highest layer, the master script plays the role of benevolent overlord, coordinating the activities of the lower-level agents and systems, and if necessary, dealing with situations that are otherwise problematic or insurmountable for the more specific layers.

Categorization enters the picture when the peaceful little hamlet becomes embroiled in a vicious battle with the local band of roving goblins. Suddenly, the master script must detect that the once-sensible actions it has been assigning to its minions are no longer

appropriate. It is here that the abstract state machine makes a transition, and begins issuing a different set of prioritizations and orders.

The blacksmith, once happily pounding away on his anvil, might suddenly shift from making horseshoes to making swords. This might be accomplished by changing his idle animation, or simply adjusting the relative merits of sword-making in his utility inputs. The militia must cease worrying about pickpockets in the local market and begin establishing a line of defense against the marauding goblins; they still act as a tactical unit, but now with a very different set of priorities and behaviors that may not fit well with their peacekeeping patterns. Even the poor, mangy, stray dog must not be forgotten: instead of rooting around in the trash for scraps, he might be found behind the barred gates of the town, bristling and growling at the shadowy enemy that lurks outside.

Note that the master script need not transition to the wartime state in any particular fashion; in fact, it is precisely the freedom to choose how this transition occurs that makes scripting a powerful tool for this type of scenario. If the game leans towards the sandbox style, the master might simply examine how many goblins are near the town and trip the transition when their "threat level" exceeds some threshold. In a more narrative-driven title, the master script might simply wait for twenty minutes after the player enters the area and then initiate the invasion.

In any case, the important flexibility of master-scripting lies in the general abstractness of the script's activities. Throughout this example, the master script is rarely, if *ever*, in the position of actually controlling the blow-by-blow activities of any given agent. This is, of course, always an option, but should be used sparingly. The whole purpose of a master architecture is to enable more appropriate control schemes to come into play as necessary. It is through this philosophy that the master script evades the trap of brittle, predictable, boring behavior. Since the script is only guiding the broad strokes of the simulation, other systems retain the freedom to act in interesting and meaningful ways based on the details of that simulation.

16.2.2 Scripts as Humble Indentured Labor

It is, of course, hardly necessary for a scripting system to maintain the high-level "master" overview of how agents behave in a simulation. A perfectly viable alternative is to invert the priority hierarchy and place a traditional behavior control system in charge of the overarching flow of the simulation, while relegating scripts to handling certain particular details. This is the "servant" approach to scripting.

In this view, scripted logic is deployed only when other techniques cannot deliver the precise behavior desired in an easily configurable manner. Most commonly, servant scripts are used to plug the gaps where design requirements have asked for a very specific outcome. For instance, a standard behavior tree could be used to dictate an agent's actions for the majority of a simulation, while a script is activated at a particular node (typically a leaf node) of the tree in order to realize a step in a story progression, or a highly crafted response to a particular stimulus, and so on.

Servant scripts derive the bulk of their power from their specificity. Unlike master scripts, which aim for the exact opposite, servant scripts are generally designed to handle a narrow range of scenarios in a simulation. The "triggers" for these scripts are, for the most part, looking for very precise combinations of events or inputs, and the scripts

themselves create agent reactions (or even proactive behaviors) that make sense only in that exact context.

As a tool for rounding out the suite of techniques used to create robust AI, servant scripts certainly have their place. The principal danger in using scripts in this manner lies in the fact that servant scripts are, by far, most effective when used sparingly as supplements to other behavior control systems.

From the perspective of a player, overuse of servant scripts produces brittle, predictable, and—after the first few encounters with the script—stale interactions. One commonly (and fairly) criticized use of scripting is the "tripwire" technique, where agents mindlessly repeat some particular pattern of action until the player crosses an invisible line in the sand, which switches the script to the next set of behavior.

Tripwires are of course not inherently bad; the issue is in using servant scripts in a vacuum. When there is no high-level system producing interesting behavior, scripted actions are no longer exceptional. Once scripted behavior becomes the norm in a simulation, players are likely to become frustrated or bored very quickly with the repetitive and overly patterned interactions they can have with the AI.

Heavy reliance on servant scripts requires an exacting attention to detail and an exhaustive capacity for predicting the sorts of situations players are likely to create. This is certainly possible, and a few noteworthy titles have created exceedingly memorable and believable experiences using extensive scripting. A serious risk, though, is the creation of an "uncanny valley" effect. Good scripting can produce very immersive results up until the point where a player does something the scripts could not have anticipated. At that point, the realism is shattered and the player often comes away feeling disappointed at the discontinuity of the illusion.

Much like in master scripting, it is imperative to maintain a selection of alternative techniques for producing interesting and compelling AI behaviors. The key difference between the two approaches is simply a matter of which extreme the scripts control—generalized, or specialized scenarios.

16.2.3 The Evils of Cross-Pollination

As mentioned earlier, wrangling a herd of AI scripts successfully requires consistent adherence to either of the servant or master models. Straying from these recipes can have disastrous consequences.

The reasons for this are primarily organizational. Without a clear set of guidelines about how scripts are to be deployed, it is virtually inevitable that a project will become inconsistent and variegated in its views on what exactly scripts are meant to do. As a result, different interactions with AI agents within the simulation will feel very different. Some might be highly scripted and controlled to exacting detail, while others are loose and rely on other AI techniques to be interesting.

In the best case, this haphazard use of scripts will produce a disjointed and inconsistent experience for the player. In the worst case, the confusion and wildly varying level of "quality" of interactions will render the experience painful and unpleasant.

The aforementioned uncanny valley effect is nearly certain to appear if agents have highly variable degrees of believability. Certainly, for practical reasons, we may not be able to craft every single agent's entire life in crystal-clear detail for the duration of the player's experience. Indeed, most large-scale simulations are forced to embrace the idea of

"ambient" or "throw-away" agents who exist solely as background or filler and generally are low level-of-detail creatures.

However, there is a fine line between level of detail and level of fidelity. Players are, for the most part, willing to accept agents who forego excruciatingly detailed existences within the simulation. What causes problems, however, is lack of fidelity in the behaviors of those agents. A particularly egregious flaw is to intersperse carefully scripted behavior of a particular agent with other control techniques acting on the same agent, without regard to the player's perception of the changes. This is a classic recipe for ruined immersion. Agents will feel inconsistent, arbitrary, or even random in their responses to simulation stimuli, and the net effect is almost universally negative.

This is not to say that all scripting must remain at one polar extreme or the other; to be sure, it is possible to inject scripted logic at various degrees of generalized or specialized control over a simulation, which can even be done to great effect if planned carefully. The key is to remain mindful of the effect that this can have on the end-user experience. To that end, heavy design, oversight, and constant testing and iteration are essential.

16.2.4 Contrasting Case Studies: The X Series

In the "*X* Series" of space-simulation games by Egosoft, we used a range of scripting technologies to implement the agent AI. Across the span of three titles, the overall philosophy of scripting shifted dramatically, although it never really landed on a clear footing with regards to the master/servant paradigm.

For *X²: The Threat* agents were controlled via a haphazard blend of servant scripts. Each individual "quest" or mini-story in the game was a separate scripted encounter. Moreover, content could be implemented in any of three layers: the C++ core engine, a bytecode-based language called KC, or a third layer known only as "the script engine" which was in turn implemented on top of KC.

This rapidly created the sort of uncanny valley experience described earlier; with no consistent quality or degree of fidelity to the agents and their behavior, the game offered a typically less-than-compelling patchwork of content. As a result, players tended to gravitate towards a tiny fraction of the in-game world that had been implemented with the most attention to detail.

The game's sequel, *X3: Reunion*, saw the introduction of the so-called "God Module" which represented a shift towards master-style scripting, at least in part. The purpose of this module was to observe the state of the simulated game universe and arbitrarily introduce events which would change the world to some degree. At the most extreme, entire swaths of space stations could be built or destroyed by this master script throughout the course of gameplay.

Unfortunately, due to limitations on time, the master philosophy was not consistently applied to all AI development. Much of the old servant style remained, and a large part of the game's content again suffered from a consistent lack of quality and detail. This was addressed post-ship via a number of downloadable patches. Even after extensive updates, however, the disjointed fidelity of agent behavior remained a common complaint among players.

Finally, in *X3: Terran Conflict*, we introduced a system known as the "Mission Director." Interestingly, this represented far more of a shift back towards servant-style scripting than the name might suggest. Instead of attempting to control the experience from a high level,

the Mission Director allowed developers—and, eventually, enterprising players—to write highly concise and expressive scripts using a common framework and tool set.

The net result of the Mission Director was to allow the servant philosophy to flourish. With access to rapid development tools and a unified policy for how to create new game content, *Terran Conflict* shipped with substantially more hand-crafted AI behavior and quests than any of the game's predecessors.

Ultimately, the most compelling conclusion to be drawn from this experience is that the actual selection of a design philosophy is not as important as consistent application of that decision. Once again, it is also extremely beneficial to select a design approach that fits nicely with the development resources available for a given project.

16.3 Implementation Techniques

Once a guiding philosophy for scripting's role has been chosen for a given project, it's time to begin considering implementation details. There are numerous mechanisms that could be considered degrees of "scripting" to some extent. These range from simple observation and reaction systems all the way to full-fledged programming languages embedded within the larger simulation itself.

For the most part, implementation choices are orthogonal to the philosophical choices outlined previously. Generally speaking, any of the techniques detailed in the following sections can be applied equally well to master or servant script models. There are a few exceptions to this, which will be specifically called out, but the predominant factors involved in choosing a script implementation approach pertain to the team involved more than the master/servant distinction itself.

Before embarking on the journey of building a scripting system, it is worth taking some time to evaluate the situation's particulars in order to ensure that the most appropriate techniques are selected. For example, observation/reaction systems have a distinct advantage when the majority of the simulation "script" logic needs to be emplaced by designers or nonprogrammer staff. At the other extreme, rolling a custom language is best reserved for engineers with prior language creation experience—although depending on the nature of the custom language, the audience may not necessarily need much technical programming experience, as we shall see later.

In the realm of implementation decisions, there are far more potentially profitable approaches than can be exhaustively enumerated here. Instead, we'll look at both extremes of the implementation spectrum, and then examine a happy medium that can be deployed to balance out the strengths and weaknesses of other approaches.

16.3.1 Observation and Reaction Systems

The canonical observation/reaction system is the "tripwire." This is a simple mechanism which observes the location of an agent within the simulation's world space, and when the agent enters (or exits) a particular locale, a reaction is triggered [Orkin 02].

Tripwires are trivial to implement in the context of most engines, because they rely only on testing intersection of an agent's bounding volume with some other (typically invisible) bounding volume within the simulation space. Such functionality is almost always necessary for other aspects of the simulation, such as generalized physics or collision detection and response, so very little new code needs to be written to accomplish a simple tripwire.

The simplest case of a tripwire resembles the ubiquitous automatic sliding doors found at the entrances to supermarkets; when something moves into the sensor volume, the door opens, and when nothing has moved there for a time, the door shuts again. This is fine for trivial interactions—where things get interesting with tripwires is in selective reaction.

Suppose we want to have a security system on the door, so that it only opens if someone carrying the appropriate keycard walks into the sensor volume. This can become an arbitrarily complex task depending on the nature of the rest of the simulation. Is a "keycard" just a Boolean flag on an agent, or might agents have inventories which need to be enumerated in order to find a key? To selectively activate the tripwire, it is suddenly necessary to interface with a large part of the rest of the simulation logic.

The challenge here is in providing appropriately rich tools. Those responsible for creating the AI technology must ensure that those actually using the technology have all the hooks, knobs, levers, and paraphernalia necessary for accomplishing the design goals of the project. While it can be tempting to throw in the kitchen sink, there is tremendous benefit in careful up-front design of both the game systems themselves and their interactions with the tripwire AI systems. Otherwise, the tools can become overwhelmingly complex and detailed, even to the point of obscuring the most commonly needed functionality.

Context is supremely important when making these decisions. What is appropriate for a team creating a general-purpose engine for licensing purposes will be dramatically different from what makes the most sense for a small, nimble team producing mobile titles at the rate of several per year. While the two systems may bear a striking resemblance to one another in the broad strokes, it is generally straightforward to keep the feature set of a tripwire system minimalistic if the scope of the simulation is well defined up front.

Any number of considerations may be useful for an observation/reaction system. Again, context is extremely important in selecting them. However, there are a few patterns that are so broadly applicable that they are worth considering—even if only to mutate them into something more specifically useful for the project at hand.

The first and most common consideration is **classification**. Put simply, this consideration examines the "kind" of thing that has just tripped the sensor: perhaps it only examines player agents, or only AI agents, or only agents on the Blue Team, and so forth. An even more powerful option is to allow things besides agents to trip the sensors. If a sensor volume moves along with an agent, and the volume is "tuned" to trip a response when a grenade enters it, it becomes trivial to build grenade evasion logic into an agent using nothing but observation/reaction architecture.

A sister technique to classification is **attribute checking**. An attribute check might look for the presence (or absence) of a particular inventory item, or compare values based on some threshold, and so on. Attribute checking is also convenient when needing to make decisions that are not strictly binary. For example, an attribute check might look at a player's health and ammunition levels before triggering a reaction that sends out squads of enemies in response to the perceived threat.

Another useful consideration is **sequencing**. A sequence requires that one tripwire be activated before another can become active. Sequencing allows designers to create linear flows of connected events. Combined with configurable timings, sequencing can be used to unfold entire story arcs based simply on having one event follow logically after another.

Deduplication is yet another handy technique. This is a trivial state flag which simply ensures that a particular tripwire cannot be triggered more than once, or more often than

at some prescribed rate. This avoids the classic blunder of AI systems that repeatedly greet the hero as he steps back and forth across the threshold of the city gates.

It is worth noting that observation/reaction does not necessarily lead to strictly linear behavior, in contrast to the images that the term "scripted AI" typically conjures up. Branching logic can be accomplished easily with the use of attribute checks and sequences. Deduplication can be applied to ensure that logic does not become repetitively applied to the simulation. Last but not least, there is the potential for movable trigger zones to be employed, as suggested in the grenade evasion example from earlier.

If a single agent has a set of tripwires applied to itself, it can quickly become prepared to handle all kinds of contingencies in the simulation world. Indeed, the limitations are predominantly found in the foresight and creativity of the designers rather than technical details.

Obviously, however, if everything in a complex simulation is handled by tripwires of various kinds—and especially if intricate, nonlinear storytelling becomes involved—the number of tripwires required can explode exponentially very easily. This is the primary weakness of simple observation/reaction systems; since they are essentially data driven mechanisms, they scale proportionally with the number of situations that the simulation must present and/or respond to. Even if the feature set of the tripwire technology is minimalistic and elegant, the number of actual triggers needed to realize a sophisticated project might be prohibitive.

16.3.2 Domain-Specific Languages

At the opposite extreme of implementation technique lies the **domain-specific language**, or DSL. A DSL is simply some kind of tool for expressing specialized types of logic. Strictly speaking, DSLs can run the gamut from little more than textually defined observation/response systems, to intricate monstrosities that rival the complication of full-blown traditional programming languages.

As the name hopefully suggests, however, domain-specific languages should be precisely that: constrained heavily to accomplish one particular task—or *domain*. The further a language strays from this self-imposed limitation, the more likely it is to become a liability rather than an asset [Brockington et al. 02]. General-purpose languages require extraordinary amounts of effort to develop to the point where they are suitable for general-purpose tasks; rolling a custom general-purpose language almost automatically entails giving up on existing tools, programmer knowledge, and battle-tested code. As such, it pays to keep the "domain-specific" part in mind at all times.

In a nutshell, the goal of a good DSL is to allow implementers to talk (in code) about what they want to happen using the same vocabulary and mental patterns that they use to think about it. DSLs are by nature very heavily tied to their intended audience; a language for helping helicopter pilots navigate automatically is going to look very different from a language used to help physicists calibrate particle accelerator experiments.

A key realization in the creation of DSL-based AI is that it is not necessary to lump *all* functionality of the AI into a single language. In fact, it is almost universally detrimental to do so, given that such accumulation of features will by necessity cause the language to stop being specific and start being more general.

Another important thing to keep in mind is that DSLs are often most useful for people who are not primarily programmers [Poiker 02]. There is no need for a DSL to be littered with squiggly symbols and magical words; on the contrary, a good DSL will tend

to resemble both the terminology and the overall structure of a design diagram of the intended logic. If the intended audience tends to use certain symbols or incantations to describe their goals, then those things should be considered perfectly at home in a DSL. However, a good language design will avoid bending over backwards to "look" or "feel" like a general-purpose programming language.

Put simply: a good DSL should not look like C++ code, nor should it require a complex parser, compiler, interpreter, virtual machine, or any other trappings of a typical general-purpose language. In fact, a simple whitespace-based tokenizer should be amply sufficient to power most DSLs. Another common option entails using existing file formats such as XML or JSON to encode the logic, and providing thin user interfaces on top of these formats for creating the actual programs in the language. End users need not write XML by hand; they can use comfortable, visually intuitive tools to craft their logic [McNaughton et al. 06]. Meanwhile, there is no need to roll yet another parsing system just to load the generated scripts. Good DSLs are about leveraging existing technologies in new ways, not reinventing wheels.

For most DSL implementations, the real work is in specifying a compact yet usable language; actually parsing and executing the code is relatively straightforward. Simple techniques include large `switch` statements, or groups of "executable" classes derived from a simple abstract base class or interface, where virtual dispatch is used to trigger the corresponding code for each language "keyword" or operation.

Of course, from a theoretical standpoint, there exists the possibility of writing an entire virtual machine architecture just for executing game logic; this has in fact been explored in numerous successful titles, including the *X Series* described earlier. However, rolling a true, custom VM is almost always a serious crime of excess when a DSL is concerned.

An effective guideline for designing DSLs is to create a language that expresses the sort of things that might be useful in a more simplistic observation/reaction architecture. All the standard considerations apply: classification, attribute checking, sequencing, and so on are all fundamental control flow techniques for the language.

In sharp contrast to a general-purpose language, DSLs need not worry about handling every contingency under the sun in terms of writing agent behavior logic. Rather, the language designers craft the vocabulary and syntax with which the AI implementers assemble the final resulting scripts—the language is the bricks and mortar, and the actual building is up to the AI programmer or designer to accomplish.

The nature of those bricks can have profound consequences for the final constructed building. As such, just as with a tripwire architecture, DSLs require a large degree of context-specific decision making to be effective. It is exceedingly unlikely that a DSL from one genre of game could be readily reused in a totally different kind of simulation experience, for example.

It is worth mentioning again that confining a game to a single DSL is usually a mistake. Even within the AI system, it can be highly effective to deploy multiple DSLs in concert. Subsumption architectures are a perfect match for this approach, and master-script systems may see tremendous benefit from deploying DSLs for various subsets of the fine-detail control mechanisms.

The basic idea is to divide up agent behavior into discrete, well-defined, compact groupings, and write a language for each grouping. Describing the behavior of an agent wandering freely about the world might require a very different linguistic framework than

Listing 16.1. This fragment of a hypothetical DSL shows how a clearly readable, minimalistic language can be used to dictate the behavior of a simple robot. The robot's basic life goal is to collect widgets until he needs to recharge his batteries, at which point he will return to the charging station and wait until he has enough power to get more widgets.

```
; Robot.dsl
set energy = get energy of robot
set mypos = get position of robot
set chargepos = get position of charger
compute homedist = distance mypos to chargepos
trigger if energy <= homedist
    path robot to chargepos
    wait until energy equals 100
trigger if energy > homedist
    set mywidget = get closest widget to robot
    set targetpos = get position of mywidget
    path robot to targetpos
    wait until mypos equals targetpos
    pickup mywidget
repeat
```

describing the exact same agent's split-second reactions during intense combat. Moreover, DSLs can even be nested—a high-level `enter combat` command in one DSL might invoke a far more detailed script implemented in a different, lower-level language.

Listing 16.1 illustrates a simple DSL fragment used to control a widget-gathering robot. For sake of brevity, the robot isn't terribly intelligent, but it should have enough basic logic to accomplish its mission and get lots of widgets. Unfortunately, it might eventually try for a widget that is too far from home, and run out of power before it can return to recharge; but extending the logic to safely avoid such widgets should be straightforward.

The key advantage of using a DSL for this logic is that any number of robot behaviors can be crafted without having to write any general-purpose control code in a more traditional programming language. As alluded to earlier, this enables a far wider audience to create AI scripts for the simulation project—a very effective force multiplier.

Note that the DSL snippet can be parsed using a trivial tokenizer; it looks readable enough, but the vocabulary is carefully chosen so that the code can be parsed and broken down into a sequence of simple command objects in program memory at runtime. For example, consider the line, `set mywidget = get closest widget to robot`. We can split this into a series of whitespace-delimited tokens using the string parsing facilities of our implementation language of choice. In most modern languages, this is no more than a line of code or a single library function call.

Next, we traverse the list from left to right. The intention is always clear without having to peek ahead to additional tokens in the stream—we want to `set` a variable called `widget`. The equals sign can be thought of as decoration to help make the program more readable. It can simply be discarded by the parser.

Once we have ascertained that the statement is a variable assignment, we proceed—again, from left to right. We determine that we will perform a lookup of some kind (`get`). This

lookup should specifically find the `closest` of two entities in the simulation. Lastly, we realize that the two entities we want to look for are of type `widget` and `robot`. Each section of the phrase can be converted directly into some kind of in-memory representation for later execution without complicated parsing algorithms or nightmarish flashbacks to compiler architecture courses.

A typical approach to executing DSL code is to process the scripts once during load time and then store them in an in-memory format that is easy to handle for fast execution. If load time performance is an issue, most DSLs can be trivially preprocessed into binary formats that are faster to convert into the in-memory format.

There are many subtly different approaches to execution models, but for the most part, they boil down to a simple decision: should execution of a single instruction in the DSL code be accomplished by virtual function dispatch or by a `switch` statement? The particulars of making this decision will of course tend to vary widely between teams, levels of experience, architectural preferences and policies, and platform performance considerations.

In the virtual dispatch model, individual instructions are implemented as classes which derive from a common interface or abstract base class. During loading, instructions from the raw source are converted into instances of these classes as appropriate. During execution, the scripting engine simply stores a container of these objects and sequentially invokes a virtual function such as `Execute` on each one.

Parameters to each instruction can be stored in a variety of formats, but typically a simple typeless DSL will only need to store strings (or enumeration "tokens" for built-in strings) that represent each parameter's value. This allows each instruction to have a simple interface that accepts a generic, untyped container of parameters which it interprets according to the semantics of the instruction itself. Implementation of a full type system is certainly possible, but it is important to weigh the work of building a type system against the (typically marginal) gains that this offers for the sort of code likely to be written in the DSL.

The switch-based model is slightly more involved. Instructions are simple constant numerical values, typically stored in the host language's notion of an enumeration. Executing a stream of DSL code consists of reading an instruction value, performing the `switch` to invoke the correct functionality, and then parsing off parameters for the instruction.

This approach generally requires the notion of an explicit execution stack as well as other forms of storage. One powerful model is to have parameters to each DSL instruction passed via a stack, and other state accessible via a blackboard mechanism. The blackboard can be shared between different DSLs and even the core language of the engine, allowing seamless passing of knowledge between various AI layers. This can be especially useful if certain routines need to be implemented directly in low-level code for performance or storage reasons.

Flow control (conditions, loops, and so on) can also be more difficult in a switch-based implementation. It is generally advantageous to have assembly language experience when working with this model, as many of the same concepts are in play—explicit jumps to certain addresses for flow control, typeless storage, an execution stack, and so on.

By contrast, flow control in a virtual dispatch model is trivial: simply create an instruction class such as `loop` (for example) that stores its own container of attached instruction objects, and executes them repeatedly based on the loop conditions.

In general, virtual dispatch-based execution models are simpler to implement and maintain—particularly to extend—but come at the obvious cost of requiring virtual

functions to operate. If the DSL is focused on high-level behavior rather than per-frame behavior, however, this may not be a significant loss; if an agent only needs to complete a "thought tick" every 3 seconds on average, the cost of virtual dispatch and storing the code as objects in memory is well worth it for the advantages in ease of use.

16.3.3 Integrated Architectures

Either way, though, rolling a DSL execution engine is a considerable undertaking, and should be carefully considered against other options. One particularly effective alternative is to use existing scripting engines—or even the low-level implementation language itself—in concert with specially crafted code structures that look and feel like a DSL but require none of the implementation investment.

These "integrated architectures" are built by constructing a library of classes or functions (or both) which represent the sum total of functionality that should be available to the scripted system. The "scripts" are then simply modules in the program that access only that limited subset of functionality, and nothing else.

Listing 16.2 illustrates the same robot and widget logic, implemented entirely in C++. This is the high-level "script" only—the implementation of each of the invoked functions is left up to the imagination. Obviously, some elements such as the `Wait();` function would require some thought to implement successfully, but for the most part, this is simply good code architecture put into practice. There is no reaching out to the renderer, or the physics model, or even much of the world state itself. Everything is implemented in terms of simple routines that have been deemed appropriate for use by the AI.

Listing 16.2. This is the robot from Listing 16.1, reimplemented in C++ as part of an integrated architecture. Note that the logic looks very similar aside from superficial syntax differences; the idea is that the script logic is implemented in terms of lower-level functionality provided by a library.

```
//Robot.cpp
while (robot.IsActive()) {
    FloatValue energy = robot.GetEnergy();
    Position mypos = robot.GetPosition();
    Position chargepos = charger.GetPosition();
    FloatValue homedist = mypos.DistanceTo(chargepos);
    if(energy <= homedist) {
        robot.PathTo(chargepos);
        while(energy < 100.0f)
            Wait();
    }
    else {
        Widget mywidget = AllWidgets.GetClosest(mypos);
        Position targetpos = mywidget.GetPosition();
        robot.PathTo(targetpos);
        while(robot.GetPosition() != targetpos)
            Wait();
        robot.PickUp(mywidget);
    }
}
```

Clearly, this is predicated heavily on programmer discipline. There is little stopping a programmer from accessing functionality directly in the rendering system, or even the operating system itself, for example. Certain measures can be taken (limiting the use of `#include` in C and C++, or `using` in C# and `import` in Java, and so on) but ultimately it is up to the leadership of the team to ensure that all code complies with the restrictions on what functionality should be used from the "script" modules.

At first blush, integrated architecture may not sound like a scripting solution at all—there is no special language in use, no external tools, and only a minimum of specially crafted logic to support the system. However, upon closer examination, integrated architectures still fit into the paradigms of master and servant architectures described earlier and can accomplish precisely the same things as a separate scripting language.

There are several major advantages to using an integrated architecture over a separate language implementation. First, it allows programmers to use their existing language knowledge and skills without much modification. Second, using an existing language opens up access to all of the existing tools for working in that language—IDEs, debuggers, compilers, profilers, and so on. Third, it removes a layer of execution abstraction between the logic and the underlying platform, which can be a substantial performance win on lower-end hardware or when the team lacks an experienced optimizations engineer to work on the scripting language implementation.

Last, but certainly not least, integrated architectures provide an illustrative method for writing almost any large-scale code. The layered approach has been heavily encouraged for decades, with notable proponents including Fred Brooks and the SICP course from MIT. Learning to structure code in this way can be a powerful force multiplier for creating clean, well separated modules for the rest of the project, even well outside the scope of AI systems.

Although the example in Listing 16.2 uses C++, it is not necessarily the most effective language for building an integrated architecture. Lua is an immensely popular option, and provides a powerful and high-performance framework on which to build scripted systems. An embedded Python implementation can use `yield` statements instead of the `Wait()` function to accomplish cooperative multitasking between agents with very little work. For programmers who happen to be familiar with JavaScript, an embedded implementation of that language can easily use a callback-oriented event model for interleaving agent processing, as used in notable stacks such as `Node.js`.

The critical tradeoff here is giving up access to existing debugging and instrumentation tool support, in exchange for a bit of extra safety (the high-level scripting language can be trivially prevented from accessing unrelated functionality such as the renderer) and a potentially large productivity boost. This is yet another judgment call that must be made on a per-team and even per-project basis, taking into account the team's skill levels, experience levels, development preferences, and so on as well as the requirements of the target platform and the scope of the project itself.

16.4 Writing the Actual Scripts

With the dominating technical issues considered, it is time to move on to actually building a complete and functional AI using the scripting technologies of choice. Although the selection of a master or servant architecture and the details of the scripting system's implementation can play a significant role in the outcome of a scripted AI,

the real artistry—and the real black magic—lies in building effective scripts on top of the technical foundation.

There are a number of challenges to creating a compelling gameplay experience via scripting. First and foremost, it is important to realize that scripting cannot and should not be used for everything. Whether in a master or servant role, scripting is most effective when used in concert with other techniques, as the project in question requires.

The chief problem with overusing scripting is combinatorial explosion. Trying to anticipate every possible circumstance that needs to be scripted for is a losing proposition. Inevitably, players will encounter some situation that the AI was not preprogrammed to handle, and the immersion will be lost entirely as the AI does something incredibly stupid. Worse, the more robust the AI seems to be, the harder these failures hit when they do occur; perfection cannot be achieved, and for the most part, the closer a scripted system comes to appearing perfect, the more disappointing its shortcomings will seem.

This can be mitigated by relying on other technologies where scripting is liable to become too hard to manage manually. The selection of these supplemental techniques is highly sensitive to the demands of each individual and unique project, but the primary guideline to keep in mind is that general situations call for general solutions. Scripting is, for the most part, a highly specific and focused approach to creating agent behavior, particularly in the servant role. It is not generally all that effective at responding to an unpredictable diversity of situations. Even in the master configuration, scripting must be carefully planned to anticipate the categories of behavior to be selected, and deferring to alternative systems is advisable when the lines get fuzzy.

16.4.1 Iteration

When implementing an AI that involves heavy use of scripting, rapid iteration is vital. As the scripts take shape, there will inevitably be gaps in the range of scenarios that the scripts are prepared to handle. Moreover, there will also be discrepancies between what the player expects an agent to rationally or believably do, and what actually occurs.

Because of this, it is crucial to test and refine scripted logic as often and thoroughly as possible. Teams with work environments set up for rapid iteration will benefit greatly, as the tiny problems can be ironed out and turned into a cohesive result. Slow or nonexistent iteration is a recipe for disaster. Because of scripting's inherent tendency to slide towards hard-coded and fixed solutions, neglecting iteration will almost inevitably result in brittle and boring behavior.

The flip side of this, of course, is that once a system passes a certain degree of flexibility and adaptability, it becomes increasingly difficult to test exhaustively. Substantial changes at this point run the serious risk of introducing more potential side effects than can be reliably checked and validated. So while iteration remains important, it is also key to limit the scope of changes made in each pass, so as to avoid constantly creating wildly different experiences that are virtually impossible to test to a satisfactory degree.

16.4.2 Transitions

There are any number of transitions that may occur during the course of a simulation unfolding: transitions between behavior control systems when scripting is used in the master role, transitions between scripts in the servant approach, transitions between levels of detail when using a subsumption architecture, and so on.

All of these transitions represent a significant challenge to creating seamless and believable scripted experiences. Any time the control of an agent undergoes such a change, it opens the possibility of a discontinuity in the perception of that agent on the part of the player.

To help alleviate this, it is important to know what transitions will occur up front during the design of the AI itself. Each transition must be carefully considered to ensure that the handoff is smooth and convincing. There are a few tricks for assisting with this, such as shared knowledge systems. If both the outgoing and incoming scripts have access to the exact same set of information, it becomes simpler to ensure that they behave consistently. Another useful variant on this technique is to explicitly communicate between the two scripts, such as handing off internal state data from one to the other.

Going a step further, it can be useful to inject special-case logic for certain transitions. For example, when moving between levels of detail, there can be great gains in believability to be had by adding extra transitory levels. These exist specifically to smooth out the jump from one degree of fidelity to another; they should be designed to come and go quickly, and give way to more long-lasting levels. In general, the larger the potential discrepancy between the behavior generated by two systems, the more appropriate it can be to use interstitial logic.

16.4.3 Variety

One of the harshest but most applicable criticisms leveled against a typical scripted AI system concerns lack of variety. Overly simplistic use of tripwire systems, for example, can easily lead to this undesirable result: if the hero is hailed by another character—using the exact same dialogue—every time he walks back and forth over some invisible line, players will inevitably lose their immersion into the simulation and come away with a less than optimal impression of the AI. Larger scale examples tend to be even more disruptive to the experience.

For this reason, it is well worth spending some time up front in design to make sure that the scripted behavior can avoid feeling canned or stale. Deduplication of tripwire activations, as discussed previously, is one simple but effective technique for doing this. In general, making sure that events do not repeat (or do not repeat too often) is a good policy for improving the feel of a scripted AI system.

Of course, past a point, adding more variety to a game's content becomes a practical problem. Content is not free to produce, and the increase in required assets may be prohibitive. In this case, simple limitations on the frequency of scripted events repeating themselves can go a long way.

Another useful trick, especially with master-style control systems, is to stagger the scripts and their execution across time. In other words, rather than having all agents begin their scripts at the exact same moment, they are dropped into the action at various points midway through the script. This avoids the eerily robotic look of dozens of ambient AI agents performing the exact same routine in a sort of zombie lockstep. A great advantage of this approach is that it requires no additional content assets; just trigger behaviors on different timetables for each agent, and over time their routines will tend to spread out and form interesting emergent interactions.

Master systems tend to be a little easier to use when it comes to creating variety. Because the master script is delegating to one or more of a handful of actual control systems for

agents, a natural variegation will emerge from the fact that not every agent is executing in the same control scheme at a given time. Servant style scripting can be just as diverse and interesting, but at the expense of requiring much more special-case logic and design work up front.

Either way, scripts should generally be seen as a sort of "glue" that interconnects other techniques for creating convincing agent behavior. Some game implementations may be largely constructed out of scripts, but in the end, reliance on other mechanisms becomes paramount. Even if it is as simple as delegating to steering and navigation systems, handing off control from a purely hard-wired script to another system goes a long way toward creating good variety and ensuring that different situations can be handled by the AI agents effectively.

16.4.4 Surprise

Next to variety, surprise is one of the most commonly lacking elements in a script-heavy AI. Unfortunately, it is also one of the most ethereal and subjective qualities to pursue in game creation. Creating experiences that can surprise and entertain players is a difficult design challenge, regardless of scripting's role in the project itself.

Working closely with designers and testers (for iterative feedback) is, of course, central to accomplishing this goal. However, there are a few technical decisions that can heavily affect how practical it is to create surprising and engaging encounters with a game's AI agents. For example, excessively complex scripting systems (such as entire full-blown programming languages) can cost far more in terms of implementation effort than they deliver in terms of final game quality. This is true not only of the core technology's implementation itself, but also of the scripts built using that technology.

Because of this, it is advisable to favor simple solutions that require a little bit of creativity over excessively complex and deceptively "powerful" systems. Carefully designed tripwire systems, minimalistic DSLs, and well-crafted integrated architectures can all be set up in such a way as to provide tremendous amounts of flexibility for design purposes without becoming overly complicated.

The important factor here is giving designers and script implementers the ability to efficiently handle a diverse range of possible scenarios in which the player may find herself. However, it is just as important to resist the urge to anticipate every possible situation in advance. The specifics of this depend on whether a master or servant approach is taken.

For the master ideology, it is sufficient to categorize a broad range of situations and react to them appropriately via delegation. The servant perspective requires a little more care. Generally, the goal is to avoid creating scripts for all possible scenarios, and allowing generalized behavior control schemes to dominate as much of the time as possible. This enables the AI to provide a varied and interesting experience while retaining the potential for carefully crafted encounters created via scripts.

16.4.5 Narrative

Perhaps the most effective home for scripting is in the creation of rich narrative experiences. When specific sequences of events are meant to unfold in a particular way during gameplay, scripting is the logical choice for implementing such a design. As has hopefully become clear, this is not the only venue in which scripting can be an effective tool, but it is certainly one of the most natural.

Creating a compelling interactive narrative experience is no small task. For its part, AI must be designed to help further the experience, and not fight against it [Barnes et al. 02]. Everyone has stories of watching a hapless character stoically continue their idle animations while chaos and battle rage all around them. Slightly less common but just as egregious are situations where the AI actively seems to refuse to do what it is meant to do.

A major component of creating good narrative AI is visibility. The developer must always be able to understand why the AI has done something particular. When using scripting, this often simply boils down to keeping a trace log of the steps that have been performed by the agent, and, where applicable, what branches have been selected and how often loops have been repeated. Being able to select an agent and view a debug listing of its complete script state is also an invaluable tool.

In the end, interactive narratives are most compelling when everything works together to create a harmonious gameplay experience. Careful iteration, attention to transitions, provision of variety, and the occasional surprise all play key roles in fulfilling this objective.

16.5 Conclusion

Scripting is all too often treated as a brittle, boring, and undesirable approach to creating game AI systems. Unfortunately, this stigma is largely justified by the negative experiences of players struggling to enjoy games with little depth or variety to their AI interactions. The good news is that this is not an inevitable outcome; with proper care and investment, scripting can be a very powerful tool.

To wield this tool effectively requires intimate coordination between the design and technical concerns of the project. Knowing the design plans and requirements is critical to success, as is forethought on how to accomplish those plans.

Moreover, scripting must be viewed as simply one tool in a diverse toolbox. Over-reliance on scripts is bound to wind up producing the exact brittle, boring results that we are trying to avoid. Scripts should be seen as a sort of glue that attaches various decision-making, planning, and knowledge representation systems into a cohesive and powerful whole.

By the same token, it is important to resist the urge to try to anticipate everything that a script might need to handle. Delegation to other technologies and techniques is crucial to upholding the quality of the final game experience. Whether this is done via a "master" or "servant" approach, consistency of application is also vital. Staying true to a design philosophy, chosen early on, will help guide the creation of all of the scripted logic and underlying technologies.

As is often the case in software engineering, simpler is better when it comes to script systems. Designers have a persistent knack for creatively using (some might say abusing) any tool set before them; this can be leveraged for great effect. Instead of creating overly complicated and intricate systems, focus on creating simple ones that are flexible and can be used in creative and interesting ways.

Last but not least, never ship blind. Play testing and iteration are critical to refining and drawing out the fun in a game, and this applies just as strongly to scripted AI as anything else. Always be prepared to rapidly tweak an interaction based on player feedback or design decisions, and stay nimble.

Scripts are a sharp blade. They can accomplish amazing feats, but one false move can be hazardous. There is no need to fear, however; with proper care and control, scripting remains one of the most powerful techniques available to the AI creator. Whether it is a simple looping behavior or a complex, blossoming tree of intricate narrative possibilities, scripting is everywhere.

It takes only some imagination and some discipline to unleash.

References

[Barnes et al. 02] J. Barnes and J. Hutchens. "Scripting for undefined circumstances." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002.

[Brockington et al. 02] M. Brockington and M. Darrah. "How not to implement a basic scripting language." In *AI Game Programming Wisdom,* edited by Steve Rabin. Hingham, MA: Charles River Media, 2002.

[McNaughton et al. 06] M. McNaughton and T. Roy. "Creating a visual scripting system." In *AI Game Programming Wisdom 3*, edited by Steve Rabin. Charles River Media, 2006.

[Orkin 02] J. Orkin. "A general purpose trigger system." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002.

[Poiker 02] F. Poiker. "Creating scripting languages for non-programmers." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002.

[Tozour 02] P. Tozour. "The perils of AI scripting." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002.