5

Structural Architecture—Common Tricks of the Trade

Kevin Dill

5.1 Introduction

When discussing game AI, developers often get hyper-focused on advocating for a particular approach or a particular architecture, but many of the problems that make AI programming so hard appear regardless of the architecture you use, and many of the most common solutions have been reinvented time after time in architecture after architecture. For example, we can use *hierarchy* to divide up the AI logic, simplifying both configuration and execution. We can use *option stacks* to allow one option to temporarily suspend another, without changing the overall plan of attack. We can use a *blackboard* to share information and ideas between AI components or between characters. We can *move the intelligence* into the objects, the terrain, the abilities, or the events in order to more logically divide the code and data, and to improve our ability to extend the game. (Some games, such as *The Sims* franchise, have used this approach to ship downloadable content or even entire expansion packs without a change to the executable.) Finally, we can use modularity to extract reusable pieces of AI logic, thus simultaneously eliminating duplicate code and enabling the AI's author to think at a coarser level of granularity.

In the remainder of this chapter we describe each of these ideas. The intent is to provide enough information to paint the big picture, to spark new ideas and solutions to problems that you, the reader, might face, but not to dig deeply into the technical details of any given solution. For that, we refer you to the bibliography and the references contained therein.

5.2 Definitions

The ideas that we will address in this paper are fairly universal, but there aren't necessarily universal terms to describe them—and in some cases, terms mean one thing to one person and something different to another. As a result, this section will define terms that might not be clear, so that we can have a common vocabulary to work with as we forge ahead.

The *AI architecture* is the underlying code—typically in C++—that controls the process by which the AI evaluates the situation and makes a decision. Several examples of architectures will be given below, and the other articles in this section describe many of them in detail.

A *configuration* is the behavioral specification that maps from specific sensory data (i.e., specific inputs) to specific decisions or actions (i.e., specific outputs). For example, the configuration for a First-Person Shooter (FPS) character would contain all of the logic necessary to decide what that character should do at each point in the game—e.g., should it attack, should it run away, should it reload its weapon, and so on. The configuration is built on top of the architecture and usually specified in data (i.e., XML or similar).

We will often refer to the thing that an AI controls as the *character*. Not all AIs control single characters—for example, you might be working on the AI for an opposing player in a strategy game (which controls many characters), or the AI for a missile in a flying game, or even an AI that controls some aspect of the user interface. You will also hear AI-controlled entities referred to as *agents*, *enemies*, or simply as *AIs*. The generic term *character*, however, is a convenient term, and most AI does apply to characters, so we will stick with it in this chapter.

It is often the case that a character will contain more than one decision-maker. For example, a character might have one algorithm that selects which weapon to use, another that selects the target to shoot at, and a third that decides what emotion he should display. We refer to these decision makers as *reasoners*. Reasoners may have clear cut responsibilities (such as those described above), or they may be organized in a more ad hoc fashion (such as the selectors in a behavior tree [Isla 05]).

Similarly, we will refer to the things that reasoners choose as *options*. A reasoner functions by first evaluating the situation, and then selecting one or more options to execute. When a reasoner picks a new option and starts executing it, we say that it has *selected* the option, and when it stops executing an option we say that the option is *deselected*. These options might be *actions* (that is, physical things that the AI does), but they might also be more abstract. For example, a character's emotional reasoner might simply change the AI's internal state (perhaps by setting an enumerated value to the appropriate state—`eHappy`, `eSad`, `eAngry`, etc.—or by assigning an intensity value to each emotion) so that other reasoners can then select appropriate actions (smiling, frowning, attacking the player, etc.) based on that state.

5.3 Common Architectures

Although the techniques described here are intended to be architecture-agnostic, it is often useful to discuss a specific architecture when describing them. All of these architectures are described in detail elsewhere, so we give just a very high level definition of each one.

Scripting is perhaps the most basic architecture possible. In it, the designer or AI programmer specifies the sequence of options that the AI will select and when they will be selected (e.g., wait 48 seconds, and then spawn three units and attack the player). Scripts may include very simple sensory input, such as trigger zones, but in general the idea is that every decision is fully specified by the AI's author [Berger et al. 02].

Finite-state machines (FSMs) were once the most popular game AI architecture, but have been largely replaced by behavior trees in recent years. An FSM is a collection of states and transitions. The states represent the options that the reasoner can select, whereas the transitions represent the conditions under which the AI will change from one state to another. For example, a very simple first-person shooter (FPS) character might have four states: *Attack*, *Flee*, *Reload*, and *Search for Enemy*. The *Attack* state would contain transitions to *Flee* (which fires if the character is nearly dead), *Reload* (which fires when the character is out of ammo), and *Search for Enemy* (which fires when the character loses sight of the enemy) [Buckland 05, Rabin 00].

A *rule-based AI* is one that consists of a sequence of predicate-option pairs. The AI evaluates the predicate for each rule in order. When it gets to a rule whose predicate is true, it executes the option for that rule and stops evaluating additional rules. Thus, a rule-based AI for our simple FPS character would have four rules. The first would make it flee if its health is low. The second would make it reload if it is out of ammo. The third would make it attack the player if the player is in sight. Finally, the fourth would search for the player [Millington et al. 09a, Nilson 94].

A *utility-based AI* uses a *heuristic function* to assign a floating-point value (typically called a *weight*, *priority*, or *utility*) to each option. It then selects the option to execute based on those values—for example, by taking the option with the highest utility, or by assigning a weight to each option and using that to guide the probability of random selection. Our simple FPS character would still have the same four possible options, but now it would decide which one to do by evaluating the heuristic function for each option and using the resulting values to guide its final selection [Mark 09, Dill 06, Dill et al. 12a, Dill 12c].

Planners such as *goal-oriented action planners (GOAP)* or *hierarchical task-network (HTN) planners* build a sequence of options that will get them to some goal state. For example, our FPS character might have the goal *Enemy Dead*. It would search through its possible options and the ways in which they can change the state of the world in order to find a sequence of options which will get it into the goal state. That plan might be something like *Search for Enemy–Attack–Reload–Attack* (if it expects two magazines of ammo to be enough to get the job done). Planners typically have the ability to replan if the situation changes. So if our character finds itself nearly dead then it might replan with a new goal, such as *Don't Die*. This new plan might have only a single option: *Flee* [Orkin 04, Gorniak et al. 07].

The *behavior tree (BT)* architecture is a bit of a special case, because it is an architecture that can contain other architectures. A BT is a tree of *selectors*, each of which makes a single piece of the overall decision. In their original formulation [Isla 05], the selectors were all exceedingly simple and not really architectures in their own right. However, more recent work has discussed the ability to use nearly any architecture in a selector [Dill 11a], making the behavior tree more of a framework (or meta-architecture) than an architecture in its own right.

5.4 Hierarchical Reasoning

The difficulty of building an AI configuration generally scales worse than linearly with the size of the configuration. In other words, the more situations your AI can handle, the more things it takes into account, the more options it contains, etc., the more complex it is to add yet another of one of those things. The reason for this should be fairly intuitive. Whenever you add something new to your AI you need to put at least some effort into considering how that new thing interacts with each thing already in existence. Thus, the cost of adding a new thing increases the more things you have.

The severity of that increase depends in large part on the architecture you are using. FSMs, for example, scale exponentially because the number of transitions that is exponential on the number of states. This becomes unmanageable very quickly. This is one of the principal reasons that FSMs have largely passed out of use for complex problems. Utility-based AI, on the other hand, only requires you to balance the heuristic functions appropriately, while rule-based AI typically just requires you to place your new rule at the proper place in the list. With that said, even a rule-based AI will become brittle when it contains hundreds or thousands of rules (which is not an unreasonable size for many games).

One common way to address this challenge—an approach that has been applied to nearly every architecture, both in games and in academia—is to break the decision making up hierarchically. That is, have a high-level reasoner that makes the big, overarching decisions, and then one or more lower-level reasoners that handle implementation of the higher-level reasoners' decisions. For example, a high-level reasoner might decide whether to execute a daily schedule of ambient tasks (e.g., get up, get breakfast, go to work, etc.), start a conversation with the player, go into combat, and so forth. Each of those options would contain another reasoner, which decides how to go about accomplishing the goal.

The advantage here is that the complexity of AI configuration scales worse than linearly on the number of options *in a particular reasoner*. To give a sense of the relevance, imagine that the cost of configuring the AI is $O(n^2)$ on the number of options (as it is for FSMs). If we have 25 options, then the cost of configuring the AI is on the order of $25^2 = 625$. On the other hand, if we have five reasoners, each with five options, then the cost of configuring the AI is only $5 \times (5^2) = 125$. Conceptually, this makes sense. When we add a new option, we only need to consider how it relates to other options within the same reasoner—which is much simpler than comparing it to every other option anywhere in the AI.

Examples of this approach abound, from hierarchical FSMs [Millington et al. 09b] to HTN planners [Gorniak et al. 07] and GOAP implementations [Cerpa et al. 08] to strategy game AIs that break the opposing player AI into command hierarchies [Pittman 08]. Behavior trees are perhaps the archetypical example—a BT is really nothing more than a hierarchical infrastructure in which you can place whatever sorts of reasoning architectures best encapsulate the decisions to be made.

5.5 Option Stacks

Most reactive AIs function by evaluating the situation very frequently (often every frame), and deciding what is the best thing to do right at that particular moment. They may have a history of past decisions that guides their choices, they may even have some greater plan that they're following, but decisions are made moment-to-moment. This is what allows the AI to respond if the situation changes.

Of course, we do want options to be persistent. That is, we don't want the AI to be constantly flip-flopping between different decisions—attacking on one frame, fleeing on the next, and then attacking again on the frame after. Or, for another example, attacking with a shotgun, and then a flamethrower, and then right back to the shotgun after only a frame or two, switching weapons too fast to even get a shot off. That sort of indecisiveness makes the AI look stupid, even if there is a good reason *at that moment* for the decision being made. As we discussed in an earlier chapter [Dill 13], looking stupid is the single worst thing that an AI can do. It breaks the player's suspension of disbelief—that is, their immersion in the experience. As a result, most architectures have some form of *inertia* built in, which keeps the AI doing the same thing unless there is a good reason for the change.

When the AI does change options, one of two possible situations pertains. In most cases, the AI's decision is lasting—that is, it has decided to stop the old option and start a new one, and it's not expecting to go back to what it was doing before. For example, if the AI is in a fight and it kills one enemy, now it can pick a new enemy to attack (or pick something else to do if the fight is over). The decision can be lasting even if the AI wasn't done with the previous option. For example, when an AI decides to flee, that's a lasting decision, even though it typically happens *before* the AI finishes its attack. Regardless, the AI has made a deliberate decision to stop doing what it was doing and do something else instead. In this case, we should stop applying inertia to the deselected option, and in fact may even want to apply a *cooldown* which will prevent us from returning to it for a short period of time.

There are situations, however, when the AI needs to react to an immediate need or opportunity, but once that reaction is complete it should return to its previous option. For example, if an AI needs to reload, it should return to the same action (presumably firing a particular weapon) when it finishes reloading. It wouldn't make sense to reload your shotgun, only to then immediately switch weapons to a flamethrower (or decide to flee). That's not to say that the AI can't change its mind, but it should have to overcome the option's inertia to do so, just as if that option were still executing. Thus, we might switch to the flamethrower immediately after reloading the shotgun—but only if we suddenly spotted some new enemy who is highly vulnerable to fire.

One common trick which has been applied to a great many architectures is to have a stack of currently executing options. This stack is sometimes referred to as a *state stack* [Tozour 04], or a *goal stack* [Cerpa 08], or *subsumption* [Heckel et al. 09], depending on the underlying architecture, but we will simply call it an *option stack*, since that is an architecture-agnostic term. Option stacks allow us to push a new, high priority option on top of the stack, *suspending* the currently executing option but retaining its internal state. When the high priority option completes execution, it will pop itself back off of the stack, and the previously executing option will *resume* as if nothing had ever happened.

There are a myriad of uses for option stacks, and they can often be several levels deep. For example, a high-level strategic reasoner might have decided to send a unit to attack a distant enemy outpost. Along the way, that unit could be ambushed—in which case, it might push a *React to Ambush* option on top of its option stack. While responding to the ambush, one of the characters in the unit might notice that a live grenade has just been thrown at its feet. That character might then push an *Avoid Grenade* option on top of

React to Ambush. Once the grenade has gone off (assuming the character lives) it can pop *Avoid Grenade* off the stack, and *React to Ambush* will resume. Once the enemy ambush is over, it will be popped as well, and the original *Attack* option will resume.

One handy trick is to use option stacks to handle your hit reaction. If a character is hit by an enemy attack (e.g., a bullet), we typically want them to play a visible reaction. We also want the character to stop whatever it was doing while it reacts. For instance, if an enemy is firing their weapon when we hit them, they should not fire any shots while the hit reaction plays. It just looks wrong if they do. Thus, we push an *Is Hit* option onto the option stack, which suspends all previously running options while the reaction plays, and then pop it back off when the reaction is done.

We mentioned this above, but it's worth reemphasizing that the option stack is not meant to *prevent* the AI from changing what it's doing, but simply to preserve its previous state so that can be figured into the decision appropriately. To extend the previous example, imagine that the character was hit in the arm and as a result lost the use of that arm—and therefore could no longer fire its weapon. In that case, it should certainly pick a different option. The option stack simply ensures that the AI has the context of the previous actions available to it so that it can make an informed decision.

Option stacks are surprisingly simple to implement for most architectures. In the AI configuration, each option can specify whether it should suspend the previous option (i.e., push the new option on the stack) or deselect it (i.e., this is a lasting decision). When an option that suspended its predecessor finishes execution, it automatically pops the stack and resumes the previous option. There are a few edge cases to handle (e.g., What to do if another option is selected while there are options on the stack?), but they're not difficult to manage. An example of the interfaces to support this can be found in our GAIA architecture [Dill 12c].

5.6 Knowledge Management

Knowledge is the key to good decision making. This is true in the real world, and it is doubly true in the realm of game AI, where most decisions boil down to relatively simple checks against the current situation. Of course, there are two different kinds of knowledge implied in that statement—knowledge of the situation itself, and knowledge of how to evaluate the situation in order to make a decision. Looked at that way, there isn't much to an AI beyond knowledge.

Given knowledge's central role in game AI, it's worth putting effort into thinking about how to best store and access our knowledge.

5.6.1 Blackboards

In the academic AI community, blackboard architectures typically refer to a specific approach in which multiple reasoners propose potential solutions (or partial solutions) to a problem, and then share that information on a *blackboard* [Wikipedia 12, Isla et al. 02]. Within the game community, however, the term is often used simply to refer to a shared memory space which various AI components can use to store knowledge that may be of use to more than one of them, or may be needed multiple times. In our architecture, for example, every character has access to two blackboards. The *character blackboard* stores information specific to the character, and is only accessible by that character's AI. The *global blackboard* is accessible by all characters, and is used to store general information.

There are many types of information that can be stored on a blackboard. One common use is to store expensive checks on the blackboard, so as to avoid the cost of running them more than once. Line of sight (LOS) checks are a common example. Quite often, more than one component in the AI (or more than one character's AI) will want to check for visibility between the same two objects. These checks can be extremely expensive. To lessen the impact of this problem, we can run the check once and then cache it on the blackboard. Path-planning checks are similar.

Another common use is to store information used to coordinate AI components. This could include partial solutions such as those found in classic blackboard systems, but it could also simply be information used to coordinate between multiple characters or between different AI components within a character. For example, if you want your characters to focus their attacks on a single enemy or to ensure that every enemy is attacked, you can store target assignments on the blackboard. If you want to coordinate spatial motion—perhaps flanking, or placing the tanks in front and the DPS and healers in back, then you can store movement plans on the blackboard. If you want to ensure that two characters don't try to use the same cover spot, then you can have them reserve it on the blackboard. If you have one reasoner whose output is the input for another—for example, the emotional reasoner that we discussed earlier in this paper—then that output can be placed on the blackboard.

There is an interview with Damián Isla on AIGameDev.com, which gives an excellent introduction to blackboard architectures as they are commonly used in games [Isla 10].

5.6.2 Intelligent Everything

When we think about the AI for a character, it seems intuitive to put all of the knowledge needed in the character. This can result in a monolithic, difficult to extend AI, however. It can also result in considerable duplication between similar (but not identical) characters.

One trick is to put the intelligence in the world, rather than in the character. This technique was popularized by *The Sims*, though earlier examples exist. In *The Sims* (and its sequels), objects in the world not only advertise the benefits that they offer (for example, a TV might advertise that it's entertaining, or a bed might advertise that you can rest there), they also contain information about how to go about performing the associated actions [Forbus et al. 01].

Another advantage of this approach is that it greatly decreases the cost of expansion packs. In the *Zoo Tycoon 2* franchise, for example, every other expansion pack was "content only." Because much of the intelligence was built into the objects, we could create new objects that would be used by existing animals, and even entirely new animals, without having to make any changes to the source code. This greatly reduced the cost of developing those expansions, allowing us to focus our efforts on the larger expansions and put out two expansions a year instead of just one.

Intelligence can also be placed in the world itself. For example, in *Red Dead Redemption* the people who populate the towns have very little intelligence of their own—but the town has hundreds of hotspots. Each hotspot has information about who can use it, what time of day the hotspots is valid, and the behavior tree for characters who are on that hotspot. Some hotspots can even require multiple characters. So, for example, a chair in a tavern might have a hotspot for sitting and drinking and another hotspot for playing poker—but the latter is only valid if there are four people at the table (and includes a mechanism for coordinating their actions). The piano bench has a hotspot that only the piano player can use, and the bar has multiple hotspots for the bartender (some that require other characters to join him, some not). Even the conversation AI works by creating a dynamic hotspot for the two characters that are going to have a conversation.

Of course, intelligence can go anywhere, not just in physical objects or locations. For example, games that have a wide range of special abilities can put the intelligence into those abilities. *Darkspore*, a game that had hundreds of abilities—many of them quite unique—took this approach [McHugh et al. 11]. Similarly, events can carry information about appropriate responses. For example, a fire in a school could carry information about how different categories of people (e.g., teachers, children, firemen, parents, etc.) should react [Stocker et al. 10].

5.7 Modularity

Component systems for characters have become commonplace in the games community. For example, a character might have a *Movement* component, an *Animation* component, an *AI* component, a *Weapon* component, and so forth. Each component encapsulates one aspect of the character's functionality behind a shared interface. This sort of modularity—breaking a large piece of code into small, reusable pieces with a consistent interface—can be tremendously powerful [Dill et al. 12b]. It can allow us to greatly reduce code duplication and to reuse more of our code both within a project and across projects. In addition, we can more rapidly implement our characters, because we simply have to plug in the appropriate module, rather than reimplement the functionality in code. It is tremendously liberating to be able to think at the level of broad concepts (i.e., entire modules), rather than having to concentrate on individual lines of code.

One extremely powerful use of modules is to create *considerations*, which are modules that can be combined to evaluate the validity of an option [Dill 11b, Dill 12c]. Hearkening back to our simple FPS character, the *Reload* option would have only a single consideration, which checks how much ammo is left in the current weapon. The *Flee* option, on the other hand, might have considerations to evaluate the current health, the number of allies left, how much health the enemy has left, what weapons are available, and so forth. The AI would combine the output of these considerations into a final evaluation of how important it is to flee, given the current situation. The output of these considerations sets might be Boolean (for example, to drive a rule-based reasoner or the transition in an FSM), or it might be continuous (for example, to drive a utility-based AI). The big advantage of considerations is that those exact same considerations can be used for other decisions—such as whether to attack aggressively, whether to use a health pack, etc. What's more, the considerations themselves can be reused across projects. Even a very different game (say, a real-time strategy game or a role-playing game) might require health checks, counts of surviving allies, distance checks, and so forth. Furthermore, many meta-concepts such as option inertia and cooldowns (described in a previous section) can be expressed easily as considerations.

Once you embrace modularity, you will find that it can be applied throughout your code base. For example, we often have an action that needs a target. The target could be a particular character (perhaps the player), or the camera, or a specific (*x*, *y*, *z*) position. It could even be the output of another reasoner—perhaps one which evaluates all enemies and selects the best one to attack. By having a modular *target* class, we can decouple the logic for specifying and updating the target from the actions that use it [Dill 12c]. Furthermore, targets can also be used elsewhere, such as in considerations—for example, a distance consideration might measure the distance between two targets without knowing or caring what types of targets they are. Targets can even be stored on the blackboard, allowing characters to communicate about them (as described in a previous section).

Another example of modularity is a weight function. When we are using considerations to drive utility-based AI, we have found that there are a great many considerations that need to map from a floating-point value (such as a distance, the amount of health remaining, the amount of ammo remaining, the number of enemies remaining, the time since some action was taken, the character's current hunger, bathroom need, opinion of the player, etc.) to a utility value. Although there might be dozens or even hundreds of considerations like that, there are actually only a few ways to handle the mapping from input value (i.e., the value the consideration computes) to return value (i.e., the consideration's output). For example, we might simply return the input value directly, apply a mathematical function to the input value and return the result, or divide the input value into ranges and return a specific output value for each range. *Weight functions* are modular components that use one of those three techniques to do the mapping for us [Dill et al. 12b]. They allow us to decouple the mapping from the consideration, ensure that we have a consistent data specification for each type of mapping, and enable us to move massive amounts of duplicate code, some of it quite complex, into a few relatively simple classes. In addition, they allow us to add advanced features, such as hysteresis, in a consistent, well-tested way.

These are just a few of the many ways in which AI concepts can be abstracted into reusable modules. Other ideas for the widespread use of modularity can be found in our previous papers [Dill 11b, Dill et al. 12b, Dill 12c]. Our experience has been that finding ways to think about our AI in terms of reusable, pluggable modules (rather than in terms of C++ code) provides a tremendous boost in productivity, even on extremely fast-paced projects.

5.8 Conclusion

In this paper we have discussed a number of common techniques that have been used across many AI architectures, which can facilitate the creation of your game. Hierarchy can be used to divide up the decision making into reasonably-sized pieces, greatly easing the process of configuration. Option stacks can enable the AI to respond to a temporary situation or opportunity, and then return to what it was previously doing. Knowledge can be shared on a blackboard, or placed in an object, in the terrain, in an action, or in an event. Finally, modularity can be used when implementing your AI to eliminate massive amounts of duplicate code, and to allow you to think in terms of broad concepts, rather than individual lines of code, when performing configuration.

References

[Berger et al. 02] L. Berger, F. Poiker, J. Barnes, J. Hutchens, P. Tozour, M. Brockington, and M. Darrah. "Section 10: Scripting." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002, pp. 503–554.

[Buckland 05] M. Buckland. *Programming Game AI by Example*. Plano, TX: Wordware Publishing 2005, pp. 43–84.

[Cerpa 08] D. H. Cerpa. "A goal stack-based architecture for RTS AI." In *AI Game Programming Wisdom 4*, edited by Steve Rabin. Boston, MA: Course Technology, 2008, pp. 457–466.

[Cerpa et al. 08] D. H. Cerpa and J. Obelleiro. "An advanced motivation-driven planning architecture." In *AI Game Programming Wisdom 4*, edited by Steve Rabin. Boston, MA: Course Technology, 2008, pp. 373–382.

[Dill 06] K. Dill. "Prioritizing actions in a goal-based RTS AI." In *AI Game Programming Wisdom 3*, edited by Steve Rabin. Boston, MA: Charles River Media, 2006, pp. 321–330.

[Dill 11a] K. Dill. "A game AI approach to autonomous control of virtual characters." In *Proceedings of the 2011 Intraservice/Industry Training, Simulation, and Education Conference*. Available online (http://www.iitsec.org/about/PublicationsProceedings/Documents/11136_Paper.pdf).

[Dill 11b] K. Dill. "A pattern-based approach to modular AI for Games." In *Game Programming Gems 8*, edited by Adam Lake. Boston, MA: Course Technology, 2011, pp. 232–243.

[Dill et al. 12a] K. Dill, E. R. Pursel, P. Garrity, and G. Fragomeni. "Design patterns for the configuration of utility-based AI." In *Proceedings of the 2012 Intraservice/Industry Training, Simulation, and Education Conference*, 2012.

[Dill et al. 12b] K. Dill, E. R. Pursel, P. Garrity, and G. Fragomeni. "Achieving modular AI through conceptual abstractions." In *Proceedings of the 2012 Intraservice/Industry Training, Simulation, and Education Conference*, 2012.

[Dill 12c] K. Dill. "Introducing GAIA: A Reusable, Extensible architecture for AI behavior." In *Proceedings of the 2012 Spring Simulation Interoperability Workshop*. Available online (http://www.sisostds.org/conference/download.cfm?Phase_ID=2&FileName=12S-SIW-046.docx).

[Dill 13] K. Dill. "What is game AI?" In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.

[Forbus et al. 01] K. Forbus and W. Wright. "Some Notes on Programming Objects in the Sims." Available online (http://www.qrg.northwestern.edu/papers/files/programming_objects_in_the_sims.pdf).

[Gorniak et al. 07] P. Gorniak and I. Davis. "SquadSmart: Hierarchical planning and coordinated plan execution for squads of characters." In *Proceedings, The Third Artificial Intelligence and Interactive Digital Entertainment Conference*, pp 14–19. Available online (http://petergorniak.org/papers/gorniak_aiide07.pdf).

[Heckel et al. 09] F. W. P. Heckel, G. M. Youngblood, and D. H. Hale. "BehaviorShop: An intuitive interface for interactive character design." In *Proceedings, The Fifth AAAI Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 46–51. Available online (http://www.aaai.org/ocs/index.php/AIIDE/AIIDE09/paper/viewFile/811/1074).

[Isla 05] D. Isla. "Handling complexity in the *Halo 2* AI." *2005 Game Developer's Conference*, 2005. Available online (http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php).

[Isla 10] D. Isla. "HALO Inspired Blackboard Architectures and Knowledge Representation." On AIGameDev.com, interview with Alex Champandard, 2002. Available online (http://aigamedev.com/premium/masterclass/blackboard-architecture/).

[Isla et al. 02] D. Isla and B. Blumberg. "Blackboard architectures." In *AI Game Programming Wisdom*, edited by Steve Rabin, pp. 333–342. Hingham, MA: Charles River Media, 2002.

[Mark 09] D. Mark. *Behavioral Mathematics for Game AI*. Boston, MA: Course Technology, 2009.

[McHugh et al. 11] L. McHugh, D. Kline, and R. Graham. "AI Development Postmortems: Inside Darkspore and The Sims: Medieval." Lecture, *Game Developer's Conference 2011 AI Summit*, 2011. Available online (http://twvideo01.ubm-us.net/o1/vault/gdc2011/slides/Lauren_McHugh_AI_Development_Postmortems.ppt).

[Millington et al. 09a] I. Millington and J. Funge. *Artificial Intelligence for Games*, Second Edition. Burlington, MA: Morgan Kaufmann, 2009, pp. 427–457.

[Millington et al. 09b] I. Millington and J. Funge. *Artificial Intelligence for Games*, Second Edition. Burlington, MA: Morgan Kaufmann, 2009, pp. 318–330.

[Nilson 94] N. Nilson. "Teleo-reactive programs for agent control." In *Journal of Artificial Intelligence Research* 1: 139–158, 1994. Available online (http://www.jair.org/media/30/live-30-1374-jair.pdf).

[Orkin 04] J. Orkin. "Applying goal oriented action planning to games." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2004, pp. 217–228.

[Pittman 08] D. Pittman. "Command hierarchies using goal-oriented action planning." In *AI Game Programming Wisdom 4*, edited by Steve Rabin. Boston, MA: Course Technology, 2008, pp. 383–392.

[Rabin 00] S. Rabin. "Designing a general robust AI engine." In *Game Programming Gems* 8, edited by Mark DeLoura. Rockland, MA: Charles River Media, 2000, pp. 221–236.

[Stocker et al. 10] C. Stocker, L. Sun, P. Huang, W. Qin, J. Allbeck, and N. Badler. "Smart events and primed agents." In *Proceedings of the 10th International Conference on Intelligent Virtual Agents*, pp. 15–27, 2010.

[Tozour 04] P. Tozour. "Stack-based finite-state machines." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2004, pp. 303–306.

[Wikipedia 12] Wikipedia. "Blackboard System." http://en.wikipedia.org/wiki/Blackboard_system, 2012.