Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment

Multi-Agent Coordination Using Dynamic Behavior-Based Subsumption

Frederick W. P. Heckel and G. Michael Youngblood
The University of North Carolina at Charlotte
Game Intelligence Group, Department of Computer Science
9201 University City Blvd, Charlotte, NC 28223-0001
{fheckel, youngbld}@uncc.edu

Abstract

Team coordination of non-player characters can create a deeper sense of immersion in real-time games by allowing characters to work together to produce better tactics and strategy. Achieving multi-agent coordination can be a difficult problem, and can incur substantial computational costs. Our goal with this work is to produce a *reactive* method for coordinating game characters that will allow computationally inexpensive team coordination. Reactive teaming creates teams of agents through the use of simple constant-time agent interactions without increasing the difficulty of authoring game characters.

Introduction

Poor or nonexistent team coordination of non-player characters in games can break the immersive experience for players. When an AI agent is surrounded by allies but behaves as though it is alone, the result is an unbelievable situation. Team coordination allows characters to work together and produce better tactics and strategy, which generates better behavior. Characters acting alone are limited in their space of action; the lone sniper in an action game can find a hidden position offering an excellent view of an ambush area, but without teammates to drive the enemy to the ambush, the sniper may end up being useless. A worse problem is when the game provides AI characters as allies for the player but the allies hinder the player.

Achieving multi-agent coordination can be a difficult problem that can incur substantial computational costs. In real-time games, the problem is compounded by the highly dynamic nature of the environment. As characters are disabled, the tactical or strategic environment changes, or team capabilities change, carefully built team plans can be completely invalidated.

When creating individual agents, reactive control methods are frequently used because of their fast response times and low computational cost. Reactive control may not generate optimal behavior, but it does allow inexpensive agents that are easy for AI designers to build. Our goal with this work is to apply the idea of reactive control to the problem of team coordination in real-time games. Reactive teaming should provide fast and computationally inexpensive agent coordination that, while not making any guarantees about the quality of the plan, are still easy to author by AI designers. In this paper, we present our formulation of a reactive teaming technique using behavior-based subsumption architecture. We show that the computational complexity of reactive teaming makes it an appropriate choice for real-time games, and present case studies that use reactive teaming in our AI testbed (seen in Figure 1).

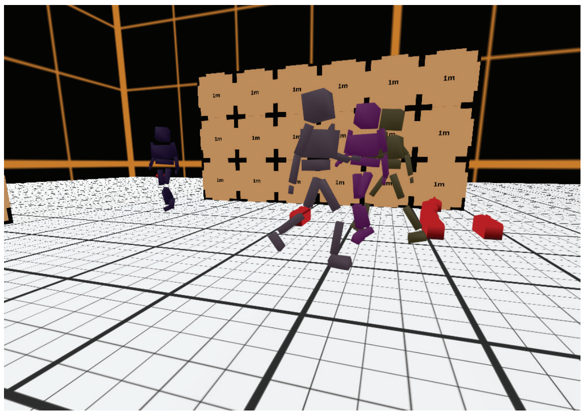
Figure 1: Teams of agents in the FI3RST environment

Background

Reactive control is a paradigm in artificial intelligence that uses the immediate world state to generate actions. The key observation underlying reactive control is that "the world is its own best model," so decisions should be based primarily on the observed state of the environment (Brooks 1990). Reactive control is a good choice for real-time game environments because they are highly dynamic and require fast responses that can be difficult to achieve with planning methods. Several different reactive architectures are commonly used, including finite state machines (FSMs), behavior trees, and subsumption architecture (Isla 2005; Fu and Houlette 2004; Brooks 1986).

Subsumption architecture is a reactive control method originally proposed by Rodney Brooks as a robot control architecture (Brooks 1986). Agents are defined as a series of

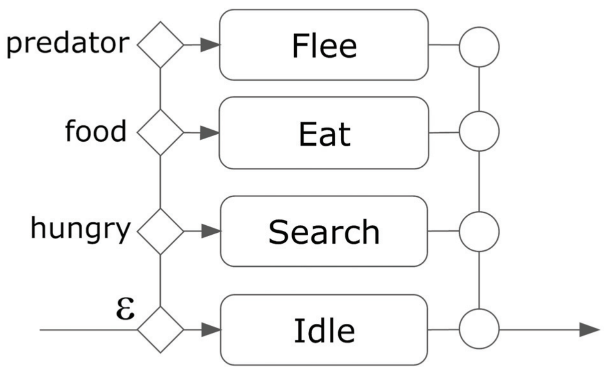

Figure 2: Subsumption architecture: layers represent behaviors, which may be simple or composed of multiple behaviors. Higher layers have higher priorities. Circles represent subsumption policies, which dictate how layers interact.

prioritized layers, as shown in Figure 2. Layers are typically composed of FSMs that make decisions based on the current environment state. Higher layers may adjust the control of lower layers, but lower layers may not interact with higher layers. Higher priority layers may completely or partially override (or *subsume*) lower layers. Transitions are not explicitly specified, though the closest analog is the subsumption policy. Subsumption is inherently parallel, eliminating the need for constructions such as ϵ transitions (in FSMs) or schedulers (in planning architectures) to achieve parallelism. Behaviors can be executed in sequence by composing a layer of multiple behaviors, but executing layers in a sequence is more complex than in a standard FSM unless each layer has a discrete effect on world state.

Reactive techniques are often adapted as behavior-based control methods. Behavior-based control was proposed by Maja Mataric as an alternative to subsumption for robot control (Matarić 1992). Behavior-based layers are more modular and act as "black boxes," in that interaction between the layers is very limited. In addition, behaviors may be more complex than typical reactive controllers. They may reason over a limited amount of world state. Aaron Khoo described the use of behavior-based subsumption as a game AI method with abstracted trigger/action modules (Khoo 2006). While subsumption in its basic form provides conflict resolution through the use of prioritized layers, Khoo uses other methods to resolve action conflicts without static priorities.

Much of the research in multi-agent planning focuses on providing near-optimal task assignments for teams. Multi-robot task assignment, a close parallel to game team coordination, can be defined as a scheduling problem (Dahl, Matarić, and Sukhatme 2009). Scheduling defines a number of jobs which must be executed by a number of machines, by a certain deadline. Different approaches have been taking to finding near-optimal teams, including planning, auctions, and free-market methods (Brumitt and Stentz 1996; Gerkey and Matarić 2002; Kalra et al. 2005).

Approaches to team coordination of behavior-based agents often focus on local approaches. Werger observed that emergent team behaviors developed from agents with no awareness of the team (Werger 1998). While this is possible, it becomes difficult to design teams which coordinate through emergent behavior. Learning and swarming approaches without explicit communication have been successful (Shell and Matarić 2005). Adding explicit communication which conveys a teammate's state provided success with a robotic box-pushing task (Matarić, Nilsson, and Simsarian 1995).

In games, teams of agents are commonly coordinated through a combination of decentralized and centralized AI. Decentralized team AI extends individual agent controllers to take their teammates' current observations and intentions into account (van der Sterran 2002a). Decentralized methods are considered to be too weak to perform sophisticated coordinated actions, so centralized methods provide an additional level that issues commands to individual agents. When centralized methods are used, the team coordinator uses either an authoritarian or coaching style, depending on the requirements (van der Sterran 2002b). Decentralized teams may use blackboards to communicate information between agents, which provide a central shared memory area that agents can read and update (Orkin 2004). The information shared by AI agents in the game *No One Lives Forever 2* includes requests for agents to move when they are blocking another agent, messages to change pathfinding behavior, notifications of events, and requests for behaviors to be executed.

Team coordination can be implemented in the Goal Oriented Action Planning architecture by providing hierarchical goal planning (Pittman 2008). When an action is chosen at a high level unit, the action includes suggestions of goals for the subordinate units under its command. Each level of the command hierarchy filters goal suggestions downward. Note that in this implementation, goals cannot be *added* to the controllers of lower level units, but instead importance modifiers are applied to the available goals. Modifiers can be either positive or negative.

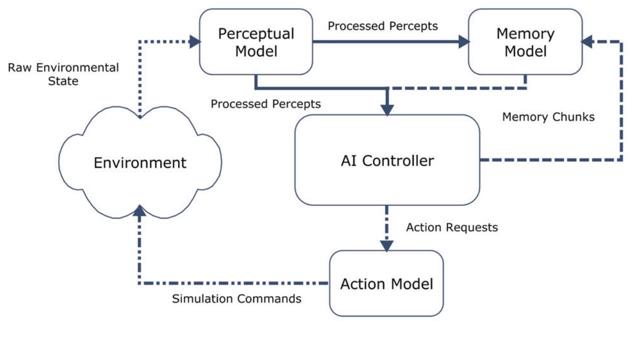

Figure 3: The BEHAVEngine Architecture.

Our approach to teaming draws from reactive control methods to create team coordination that is more structured than emergent approaches, but without the complexity of planning. The work described in this paper is implemented as part of the Behavior Emulating Hierarchical Agent Vending Engine (BEHAVEngine) (Heckel, Youngblood, and Hale 2009b). BEHAVEngine provides a behavior-based subsumption engine for controlling game characters, and uses our FI3RST (FIrst and 3rd-person Realtime Simulation Testbed) environment to provide a game testbed. The subsumption architecture implemented by BEHAVEngine is a hybrid system and includes elements inspired by cognitive models, including modular perceptual, memory, and action systems which are independent of the subsumption controller, as shown in Figure 3.

Methodology

The primary mechanism used by our reactive teaming method is a dynamic layer stack. Agents can add and remove entire behavior layers at runtime, and layers are monitored for failures. Agents on teams can coordinate with one another by exchanging behavior layers. These layers can be transferred in their entirety, or subdivided to reduce the burden of each agent and possibly improve the speed of the task. Reactive teaming can also make use of environmental information to provide convincing cooperation without explicit task division.

Behavior Transfer

The core of reactive teaming is the agent-to-agent transfer process. Typically agents in teams will be selected to perform a particular behavior based on their suitability. This requires evaluation of each agent with respect to each behavior. If all behaviors are assigned at once, the computational complexity is typically $O(mn)$ for n agents and m tasks. The number of messages required to communicate utility values is also typically the same. Online assignments, where one task is assigned at a time, will still require $O(n)$ complexity and communication to assign tasks.

Just as reactive control greatly reduces the complexity of individual agent controllers, reactive teaming reduces the complexity of team coordination. Instead of assigning behaviors globally, all assignments are local. Furthermore, for an agent to be assigned a task, it must request one.

All task assignments are one-to-one transfers. Take two agents in a virtual environment, A and B. Agent A has no current behaviors, or may have simple placeholder behaviors (we will refer to A as a *generic agent*, and assume that all generic agents have a single default *wander* behavior). Agent B has one or more defined behaviors (we will refer to B as a *fully-defined agent*). When A encounters B, it will request a behavior from B. Then B will choose a behavior to transfer to A, offer the transfer, and if A can execute the behavior, it will accept it. Otherwise, A will reject the behavior. No utility value is calculated, and no other agents are considered during the process. The result is that while it may take up to $O(n^2)$ agent interactions to distribute behaviors in the worst case ($n-1$ generic agents, and each generic agent queries all other generic agents before the fully-defined agent, with no child behavior transfers), these interactions are amortized over multiple game ticks. This amortization is important as it prevents the task distribution process from dominating the limited CPU resources available during each frame of the game.

In heterogeneous environments, this method of behavior assignment may result in non-optimal assignments. Related to this problem is that this technique does not allow behaviors to be *pushed* to other agents, as receiving agents must request behaviors. Behaviors may also be distributed unevenly, depending on the policy for choosing which behavior to transfer. Additional refinements can address each of these problems in different ways. Behavior pushing may be important in situations where teams of agents are organized in hierarchies. These hierarchies can be created through high-level user logic to issue team commands through the environment. The exact behavior of agents with regard to transfers is represented as a *policy*. The policy is a three-tuple, (r, t, i), defining a strategy for when to *request* a layer, how to decide which behavior should be *transferred*, and where to *insert* the new layer.

Request Policies

Agents must decide when to request a behavior transfer. We have defined four approaches to deciding when to request a transfer: failure, cooperative, greedy, and command.

An agent using the *failure* policy will request a behavior transfer when one or more layers are failing to execute properly. This may occur in a number of different cases. First, a layer may no longer be able to activate correctly because one or more prerequisites are no longer available. An example of when this may occur is if a behavior requires a specific object (or object with specific affordance), and that object is no longer available. The subsumption controller can check behavior prerequisites to discover this without attempting to execute the behavior, so that behaviors that are failing to trigger at all can be detected. In other cases, a behavior may attempt to execute and fail; this can occur when an agent attempts to navigate to a part of the world that is no longer accessible. Another more rare condition that can occur is if too many agents are attempting to use a limited resource in the world, such as an object. An example is a scenario in which several agents are attempting to perform an object removal behavior. If the agents are grouped together and repeatedly attempt to pick up the same object, the behavior will fail. Repeated failures will result in the behavior being chosen for replacement, and so the team can self-correct if too many agents have the same behavior in this case. Failure cases will not monitor the behavior that failed to check if it is available once more.

The *greedy* policy is similar to the failure policy. It will request a behavior if one of its own is failing, but unlike the failure policy, it will monitor the failed behavior. In case of a prerequisite failure, it will restore the behavior if the prerequisites are restored. In other types of failures, it will periodically attempt to restore the failed behavior; if it succeeds, the replacement behavior will be removed.

The *cooperative* policy will request a behavior transfer if it does not currently have a transferred behavior. If the transferred behavior fails, it will look for a new behavior transfer. It will not monitor behaviors that have been replaced due to failures.

A final policy is possible, which is the *command* policy. In this case, the agent is part of a command hierarchy, and will respond to requests from commanding agents. When the agent receives an order from a commanding agent, it will request a layer transfer from the commander. Note that the actual layer transfer is still initiated by the receiving (subordinate) agent, and the command-response layer could potentially be placed below other layers, allowing for insubordinate agents.

Transfer Policies

Once a request to transfer a behavior is received by an agent, it must decide which behavior to transfer. Before transferring a behavior, the agent must have at least one layer marked as transferable and which has not exceeded the maximum number of transfers. The transferable flag and the maximum number of transfers are defined when the agent is authored. We define four possible transfer policies: priority, distribution, failure, and command.

The *priority* policy will always choose the highest priority transferable layer. A variation on this will always transfer the lowest priority transferable layer. The layer transferred may be one that was transferred to the agent from another. Behaviors that have exceeded their maximum number of transfers may not be transferred.

A *distribution* policy is one that aims for even coverage of its layers. The distribution policy will transfer the highest priority transferable layer that has the least number of transfers. Alternatively, the lowest priority layer may be chosen.

Failure policies will transfer the highest priority transferable layer that is currently failing. If no layers are failing, it will fall back on the priority transfer policy.

Finally, the *command* policy is the complement to the request policy of the same name. Command policies will transfer behaviors according to the mapping determined by a command behavior layer. Note that this effectively allows auction and planning teaming methods to be used as a layer of abstraction over reactive teaming.

Insertion Policies

While placement of received layers is defined primarily by the request policy, the subsumption policy must also be defined. Possible strategies to choose subsumption policies are override all, override none, replacement, or modality. The default policy is for a transferred layer to *override all*. The opposite of this policy is *override none*, which will only override conflicting output produced by lower layers. *Replacement* uses the subsumption policy of the layer being replaced, or falls back on *override all* if the transfer is not a replacement. *Modality* policies override one or more of the effector modalities to suppress only specific channels of action.

Behavior Coordination

While layer transfer provides a simple level of team coordination, it generally does not create tight interactions between agents to execute a task. Further coordination methods are needed to achieve this.

The first method uses influence points and affordances. Influence points provide a mechanism for placing information in the world using navigation meshes as the underlying model instead of the grid-based maps used by influence maps (Tozour 2001; Heckel, Youngblood, and Hale 2009a). Characters can place influence points to provide information for the entire team without explicitly sending messages to every other character.

Affordances represent opportunities for action, and are derived from the concept of the same name in psychology (Gibson 1977). Affordances can be attached to objects to provide hints to agents about how to use the objects, and were used in The Sims (Simpson 2005). Affordances can be particularly valuable for teams of agents if they are augmented to include multiple action slots. An action that requires multiple agents to execute, such as moving a large object, will create multiple affordances that must be filled for the action to complete successfully. Even if multiple agents are not required, using multiple affordances allows agents to easily cooperate to perform an action without additional logic in the behavior.

Finally, some tasks are not easily shared with affordances or influences and must be modified when transferred. Behavior division provides an additional solution. When the primitive behavior is written, the programmer can include divide and merge methods. Divide methods split the behavior into two sub-behaviors, while merge methods take two sub-behaviors and re-combine them. Division is used when a behavior is transferred from one agent to another, while merging is used if an agent removes a transferred layer from its controller. Some behaviors in the BEHAVEngine behavior library are written to include these methods. The simplest example of this is division of an explore behavior, which maintains a list of regions to visit. When the explore behavior is transferred to an additional agent, the current list of regions is broken into two parts, so that two agents will work together to cover the area faster. If later the second agent decides to stop performing the task, it will notify the first agent, which will then restore its list of regions to explore. The division task is most appropriate to spatial tasks, though it is possible to apply to others.

Evaluation

Reactive teaming is implemented as part of BEHAVEngine. Failure strategies are activated once a layer has activated at least 5 times and failed 50% of the activations. Generic agents are implemented as characters with one behavior, either a simple *wander* behavior that causes the agent to wander around the world, or a *null* behavior, which provides no behavior. The maximum number of behavior transfers is set per-behavior.

Evaluation focuses on the computational complexity of creating and maintaining the team and development of scenarios in which reactive teaming can be used. Evaluating teams in games is a difficult task, because metrics for team performance are game-specific and vary widely. Depending on the type of game and the audience, the metric may be team score, adaptability to player behavior, or even just *fun*. While we do plan to evaluate the reactive teaming method in games, for this paper we focus on showing that our methods are computationally feasible and demonstrating the types of team behavior that we have generated in our testbed.

Complexity

In terms of agent complexity, each behavior that is necessary for the team can be specified just for a single agent. Other techniques require that each agent has a fully specified set of behaviors for tasks that it is capable of performing. For a set of n agents that can perform m behaviors, the overall complexity of the team specification is $n * m$ for fully specified agents, instead of $n + m$. This reduces the memory requirements per-agent, as agents only maintain information for behaviors that are actually used. In addition, runtime complexity is reduced, as agents have a smaller number of behaviors to choose from, rather than having to evaluate triggering conditions for all behaviors to decide which are active.

Another aspect of runtime complexity that is perhaps more important is the cost of allocating tasks. Alternative task assignment methods include a central authority choosing task assignments for all agents, often through an auction method. The auction requires first that each agent determines a score reflecting how well it can perform each available task. This generates $n * m$ scores to be considered. Alternatively, agents that have only limited communication but full knowledge of the capabilities of other agents can generate a full plan locally to determine roles for other agents, and then act on their own role. There are significant processing requirements for these techniques, and agents will not be active for as long as it takes to assign the tasks.

In contrast, reactive teaming requires only constant time interactions between pairs of agents. The task assignment procedure only delays the agent that is requesting a behavior, rather than all agents. The effect of this is that the assignment task, which is $O(n^2)$ in the worst case, is amortized over n game ticks, adding only a linear amount of computation, $O(n)$, to each game tick for complete team assignments. This is compared to incremental team coordination methods that add $O(n)$ computation for assigning a single behavior in one tick, or $O(n * m)$ for methods which assign the entire team at once.

Case Studies

The system described above was implemented as an extension for BEHAVEngine, and evaluated with several scenarios. Cooperative policies were used for requests, and priority policies for transfers. Our first scenario included four generic agents and a fully specified agent with behaviors describing five patrol routes in the game environment. The generic agents started up, requested behaviors from different agents until the fully specified patrol agent transferred patrol routes. Communication was treated as independent of distance, so the agents were able to immediately receive behaviors without finding the fully specified agent initially. Since each patrol layer had a transfer limit of one, each agent received one of the patrol routes, and each patrol route was satisfied.

An additional test scenario created a single garbage-cleaning robot which was to wander through the world and pick up objects. Twelve generic agents were added that wandered the world, and in this case, would request behavior transfers only when within range of one another. Once a generic agent received the garbage-cleaning behavior, it would continue to wander, and transfer this behavior to other agents if requested. This scenario was extended to place two types of garbage-cleaning bots to pick up different types of objects. Two types of objects were added at different times. As the first type of object was cleared, failure and inactivity transfers occurred, adapting the team to gather the second type of object. The end result is that, as the environment changed, the team composition adapted to the changes.

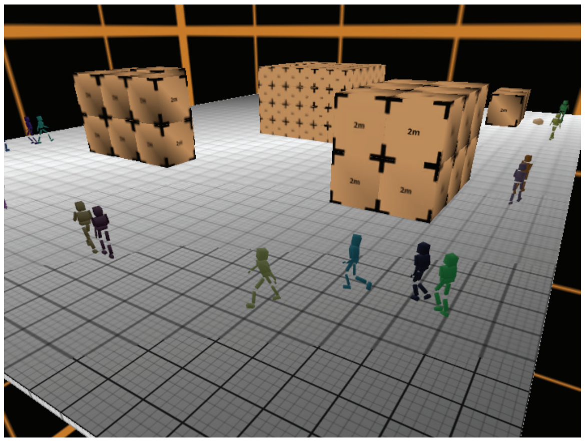

Figure 4: Screenshot of patrol agents in the FI3RST simulation environment. In this scenario, a single agent with knowledge of a large (12 point) patrol route is placed in the world. Generic agents are added, and the patrol behavior is divided into segments so that the workload is shared among the team members.

More advanced coordination was tested with the scenario shown in Figure 4. In this case, a patrol agent with a large patrol route was placed in the world, and varying number of generic agents were added to the world. The patrol behavior used behavior division to split the patrol route into segments, halving it each time it was transferred until each agent was walking a two-location segment of the route. The AI engine was able to run over 150 agents at 10hz with a low CPU load (approximately 15% for the agent controllers) on an Intel Core 2 6400 system running at 2.13ghz. Each agent used A* pathfinding through the navigation mesh for navigation between patrol points. Larger numbers of agents were not tested as the graphics pipeline became overloaded.

Conclusions and Future Work

In this paper, we described a method for creating teams of characters in games using reactive methods. We have shown that our reactive teaming method is computationally feasible for larger numbers of agents, and provided case studies to show how the method can be applied to achieve adaptive and scalable teams.

Further exploration is still needed for reactive teaming. Two major areas need additional development. First, agents may need to have self-preservation behaviors which cannot be overridden. One possibility is to merely allow the placement of a high priority hierarchical layer which must be the top layer. The second area is in behavior coordination. Influences and behavior division are useful techniques for enabling tighter coordination than behavior transfers alone, but do not provide elegant solutions for certain common situations. For example, if a group of agents is fighting a group of hostile agents, the situation may call for all friendly agents to focus on one particular hostile. This is not an unusual situation, but there is currently no mechanism to easily allow agents to convey the message to focus on a certain target. One possible technique is to enable the agents to exchange behavior state by transferring memory chunks.

Thus far, we have used reactive teaming in first/third-person shooter environments, where the primary mode of cooperation is tactical rather than strategic. Another novel use of our teaming method is in real-time strategy (RTS) games: production buildings can be agents that decide what new units to create, and units are treated as generic agents. Units can query the production building for behaviors to execute. We are investigating the feasibility of evaluating reactive teaming using shipped commercial games, with a plan to use BEHAVEngine for RTS games as well as first/third-person shooters.

Acknowledgments

This material is based on research sponsored by the US Defense Advanced Research Projects Agency (DARPA). The US Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the US Government.

References

Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* 2(1):14–23.

Brooks, R. A. 1990. Elephants don't play chess. *Robotics and Autonomous Systems* 6(1&2):3–15.

Brumitt, B. L., and Stentz, A. 1996. Dynamic mission planning for multiple mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2396–2401.

Dahl, T. S.; Matarić, M.; and Sukhatme, G. S. 2009. Multi-robot task allocation through vacancy chain scheduling. *Robot. Auton. Syst.* 57(6-7):674–687.

Fu, D., and Houlette, R. 2004. *AI Game Programming Wisdom 2*. Charles River Media. chapter 5.1: The Ultimate Guid to FSMs in Games, 283–302.

Gerkey, B. P., and Matarić, M. J. 2002. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation* 18(5):758–768.

Gibson, J. J. 1977. *Perceiving, Acting, and Knowing*. Lawrence Erlbaum Associates. chapter The theory of affordances.

Heckel, F. W. P.; Youngblood, G. M.; and Hale, D. H. 2009a. Influence Points for Tactical Information in Navigation Meshes. In *Proceedings, International Conference on Foundations of Digital Games*.

Heckel, F. W. P.; Youngblood, G. M.; and Hale, D. H. 2009b. Making Interactive Characters BEHAVE. In *Proceedings, Florida Artificial Intelligence Research Symposium*.

Isla, D. 2005. Handling Complexity in the Halo 2 AI. In *Proceedings of the 2005 Game Developers Conference*.

Kalra, N.; Dias, M. B.; Zlot, R. M.; and Stentz, A. T. 2005. Market-based multirobot coordination: A comprehensive survey and analysis. Technical Report CMU-RI-TR-05-16, Robotics Institute, Pittsburgh, PA.

Khoo, A. 2006. *AI Game Programming Wisdom 3*. Charles River Media. chapter 4.10: An Introduction to Behavior-Based Systems for Games, 351–364.

Matarić, M. J.; Nilsson, M.; and Simsarian, K. T. 1995. Cooperative multi-robot box-pushing. In *IROS '95: Proceedings of the International Conference on Intelligent Robots and Systems-Volume 3*, 3556. Washington, DC, USA: IEEE Computer Society.

Matarić, M. J. 1992. Behavior-based control: Main properties and implications. In *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, 46–54.

Orkin, J. 2004. *AI Game Programming Wisdom 2*. Charles River Media. chapter 3.2: Simple Techniques for Coordinated Behavior, 217–227.

Pittman, D. 2008. *AI Game Programming Wisdom 4*. Charles River Media. chapter 4.3: Command Hierarchies Using Goal-Oriented Action Planning, 383–391.

Shell, D. A., and Matarić, M. J. 2005. *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*. Cambridge University Press. chapter Behavior-Based Methods for Modeling and Structuring Control of Social Robots, 279–306.

Simpson, J. 2005. Scripting and Sims2: Coding the Psychology of Little People. Talk at the 2005 Game Developer's Conference. Retrieved from https://www.cmpevents.com/Sessions/GD/ScriptingAndSims2.ppt.

Tozour, P. 2001. *Game Programming Gems 2*. Charles River Media. chapter 3.6: Influence Mapping, 281–297.

van der Sterran, W. 2002a. *AI Game Programming Wisdom*. Charles River Media. chapter 5.3: Squad Tactics: Team AI and Emergent Maneuvers, 233–246.

van der Sterran, W. 2002b. *AI Game Programming Wisdom*. Charles River Media. chapter 5.4: Squad Tactics: Planned Maneuvers, 247–259.

Werger, B. B. 1998. Cooperation without deliberation: A minimal behavior-based approach to multi-robot teams. *Artificial Intelligence* 110:293–320.