Image by Shane McDermott

Decision Modeling Architectures in Game Development

Shane McDermott

12/13/17

CIST 3000

Decision Modeling Architectures in Game Development

Shane McDermott

December 13, 2017

CIST 3000

Contents

Table of Figures

Executive Summary

The intent of the report is to provide a broad understanding of architectural systems that are commonly used by software engineers to aid with the artificial intelligence components of game development. These systems are used to automate menial behaviors so that more attention can be devoted to game-defining components. The viability of each of these systems is assessed based on common genre requirements as well as development environment considerations. The decision-making systems that are evaluated are decision trees, state machines, behavior trees, Goal-Oriented Action Planning, and utility systems.

Decision trees are simple, rule-based systems that explicitly state their conditional behaviors. These systems are fast to implement and require minimal programming experience to interact with. Despite being an ideal candidate for prototype stages of development, they are ill-suited for long-term development schedules.

State machines are only slightly more complex than decision trees but are much more modular. They are an ideal candidate for mobile games, sports games, and fast-paced action games. Unfortunately, games intending to emulate any kind of behavioral complexity will be hindered by the simplistic limitations of state machines.

Behavior Trees add an additional layer of complexity to state machines in order to offer a much larger behavioral design space. Games developed using a third-party engine will likely come with a basic implementation of this decision-making system. Behavior trees can be challenging for inexperienced developers, but require minimal programmer input. They are suitable for most games that are not concerned with strategy or tactics.

Goal-Oriented Action Planning is a powerful decision-making system capable of creating engaging behavior patterns. This system also enjoys the benefit of requiring very little programmer input. Action games, simulation games, and role-playing games benefit the most from this approach. However, the computational overhead is too high to be viable for mobile games and is wasted on games with a small set of possible actions.

Utility systems are another powerful decision model capable of producing high-quality results. Unfortunately, utility systems require a significantly higher initial development time investment than the other decision-making systems presented. They can also be difficult for developers to understand. Simulation, strategy, and role-playing games can make great use of utility-based systems so long as the development team can afford to devote the time necessary to properly tune them.

Any of the discussed decision-making systems can be used to produce a game. The requirements of the game and capabilities of the developers will ultimately determine which system is the optimal choice.

Introduction

Video games are a medium unlike any other. Without even considering the game design components, the modern development industry is a marriage of Hollywood and Silicon Valley. This results in a high budget, high pressure environment where the development scope is so large that designing every character's behaviors by hand is impossible (Sloan, Kelleher, & Mac Namee, 2011). To overcome this, developers often employ a decision-making system to manage the behaviors of all but the most unique non-player characters.

The decision-making system is responsible for telling game characters what to do. Despite its importance, most game developers treat this as a relatively minor component in AI design. Simple decision-making systems such as decision trees, state machines, and behavior trees are used in most scenarios (Millington & Funge, 2009).

However, as technology improves, game expectations and possibilities continue to evolve. To address this, developers have started to explore more sophisticated decision-making systems. Goal-Oriented Action Planning and utility systems are two of the more popular alternatives. Fuzzy Logic and Neural Networks have also received some attention, but are still generally avoided due to their complexity.

In the early stages of development, establishing a core decision making system is fundamental. Accomplishing this offers developers a cornerstone of consistency in what is otherwise a chaotic experience. The consistency of the decision-making system also impacts the player's experience. Overly simple decision systems are easy to exploit and can ruin a game's reception. Conversely, unpredictable decision-making systems can frustrate players. Both scenarios ultimately lead to the same result. A decision-making system must provide an adequate balance that engages players. One tool in achieving this balance is a decision model.

A decision model simulates the elements and variables related to the decision-making process (Oxford University Press, 2017). This allows developers to test and predict artificial intelligence system decisions in a variety of environments before exposing them to players. The framework which incorporates a decision-making system into decision model creation is called a decision modeling architecture.

The intent of this report is to provide a broad understanding of the decision-making systems most widely used by the game development industry. To accomplish this, the theory behind these systems will be explained before evaluating the impact of these systems in a development environment. Prior to doing this, development environment factors worth considering will be discussed. This will establish a common basis for comparison for developers with varying degrees of experience. After this has been created, a variety of decision making systems can be assessed from a surface level of understanding. Systems that will be evaluated include decision trees, state machines, behavior trees, Goal-Oriented Action Planning, and utility systems.

Agent is the term that will be used to refer to any entity that is not controlled by the player. In most cases, this is referring to non-player characters. However, this term can also be applied to other artificially intelligent game systems such as enemy tacticians found in strategy games.

Each game's requirements are unique. Any tool that can help with the development process must be considered. Decision-making systems are a powerful tool with an undeniable impact on the development process. Choosing an appropriate system can either facilitate or hinder this process.

Modeling the Decision Making Process

When evaluating a decision modeling architecture, there is a vast number of criteria to consider. Some of these factors can be generalized using traits shared with their broader genres.

To establish a common frame of reference, five genres will be considered as reference points: action games, simulation games, strategy games, sports games, and role-playing games. Each of these game archetypes have a unique set of expectations associated with them that define how game entities should behave.

Decision Model Standards

In the classic AI model, the decision-making system is provided with an information set representing the character's knowledge. Typically, this knowledge can be categorized as either external or internal. External knowledge is the information that is known about the environment around the agent: the position of other characters, the layout of the level, the direction that a sound came from, and so on. Internal Knowledge is information about the agent's inner state or motivations. Examples include an agent's health, goals, and previous actions (Millington & Funge, 2009).

The expected output of a decision-making system is an action request. Action requests can also be composed of external and internal components. External action requests are expected to change the external state of the character. Firing a weapon or moving into cover are examples of external actions. Internal changes will usually be less noticeable to the player, but might impact future decisions. Internal requests are highly context-dependent, but examples include changing the agent's emotional state or opinion of another character.

Common Game Genres and their Qualities

Although each game's specific qualities and needs will be different, they will often share at least some of the requirements of a broader genre. To establish a common frame of reference, five genres will be considered as reference points: action games, simulation games, strategy games, sports games, and role-playing games. Each of these game archetypes have a unique set of expectations associated with them that define how game entities should behave.

Action Games

The action genre encompasses a wide variety of sub-genres. Shooters, stealth, and adventure games are the most prominent examples.

Action games are designed to test the motor skills and hand-eye coordination of the player. In fast-paced variations such as *Call of Duty* (Infinity Ward, 2002), the average enemy agent lifespan can be as little as 7 seconds (Mark, 2009). In these scenarios, simple decision-making systems are ideal.

This is not the case for all action games. Slower paced action games such as *Halo* (Bungie Software, 2001) and stealth/action hybrids such as *Metal Gear Solid* (Kojima, 1998) use their agents as pieces in environmental puzzles. While agent behaviors can still be reasonably simple, the decision model driving those behaviors often requires more foresight in design.

Simulation Games

Simulation games attempt to emulate lifelike systems. *Minecraft* (Mojang, 2011), *Kerbal Space Program* (Squad, 2015), and *RimWorld* (Ludeon Studios, 2016) are popular examples of simulation games.

Instead of following a predetermined storyline from beginning to end, simulation games commonly rely on the concept of *emergent narrative* as entertainment. Emergent narrative is the manifestation of complex scenarios from the interaction of relatively simple components.

One particularly noteworthy quality of simulation games is the reduced importance of combat. While it may still exist at some level, it is rarely the focal point. This means that combat decision making can be drastically simplified. A consequence of this is that the expected agent lifespan will be much greater than in combat heavy games. Agents are expected to not only interact with their environment, but to do so intelligently for the extent of their screen time.

Strategy Games

Strategy games build upon the concept of simulation games by heightening the importance of combat, strategy, and tactics. Skillful planning and thinking is rewarded in these games. *Sid Meier's Civilization* (Meier, 1991), *Starcraft* (Blizzard Entertainment, 1998), and *Halo Wars* (Ensemble Studios, 2009) are some of the better-known examples.

Often played from a godlike view of the level, players are responsible for indirectly controlling units under their command (Rollings & Adams, 2003). Agents are responsible for indirectly controlling their own units under the same premise. Individual units are rarely expected to exhibit much intelligence beyond the ability to move.

Commanding agents persist for the duration of the level, and the factors they consider are drastically different from those in Action or Adventure games.

Sports Games

Another extension of simulation games, sports games augment the requirements by enforcing the ruleset of their representative sport.

Team based sports games place players in the role of team manager as well as team player. One of the most popular examples is the *FIFA* (EA Sports, 2017) series. Locomotion is a critical component in these games, as action possibilities tend to be confined to the ruleset of the sport that is being played.

Role-playing Games

Mechanically, role-playing games are often a combination of several other genres. Players are responsible for navigating one or more characters through a simulated environment. Environmental puzzles often require a combination of abilities to solve.

Depending on the sub-genre, combat will usually adapt components of action or strategy games. Real-time combat games such as *Diablo* (Blizzard Entertainment, 1996) have agent characteristics similar to action games. Most enemy agents have limited skillsets and low expected lifespans. Turn-based combat can generally be broken into two categories: classical and strategy.

Although the classical turn-based combat system has fallen out of favor recently, it remains an iconic mechanic of role-playing games. *Pokémon* games are the most popular series to continue to use this system. Games utilizing this system only allow one character to act at a time, while the rest remain motionless and wait for their turn. Characters are allowed to perform one action per turn. Distance between characters and time are rarely considerations when using this system.

Strategy role-playing games adapt the classical turn-based combat system to a grid system often seen in strategy games. *Fire Emblem* and *The Banner Saga* are two examples of strategy role-playing games. Character movement ranges and ability sets are often defined by archetype.

Aesthetically, role-playing games tend to focus on storyline and character development. While most of this is predetermined and told through dialogue and cinematic events, it is critical that player immersion is not lost during non-scripted events. This means that agents are expected to not only act intelligently, but to act as their portrayed character would. This often requires a great deal of agent specific tuning to fully accomplish.

Other Development Considerations

Game features and mechanics are not the only factors to consider when choosing a decision model architecture. Hardware limitations, developer experience, and scalability are all affected by their game’s decision model.

Each decision modeling architecture is going to require a unique balance of input from programmers and designers. This can have many ramifications on the development process. Programmer time could be spent implementing or improving upon other game features, while designer time could just as easily spent creating more playable content.

This balance can also impact game performance issues. Because performance optimization is usually a programmer’s responsibility, writing optimized code is not a skill set that is typically expected of designers.

Another important consideration is the entire development team’s level of experience with various architectures. Each decision-making system is going to have its own requirements for technical understanding, which can easily become a bottleneck in the development process.

Most modern games are supplemented by additional content after release. To help drive sales, this content usually features new gameplay mechanics that expand upon already existing ones. Each decision-making system reacts to these kinds of changes differently.

Although hardware continues to improve, these gains are rarely allocated to the decision-making process. This means that both processing requirements and memory requirements must be considered when evaluating a decision model. Minimalistic models will be much better suited for mobile platforms, as power consumption will be much more relevant than realism.

Decision Trees

Decision Trees, also referred to as rule-based systems, are simple decision models that are fast to implement. A decision tree is composed of connected decision points. Starting from an initial decision, referred to as the root of the tree, one of the potential options is selected. This option will either lead to an action request or another set of options. The outcome of each step in the selection process is determined by some number of conditions. This option selection process is repeated until an action request is encountered.

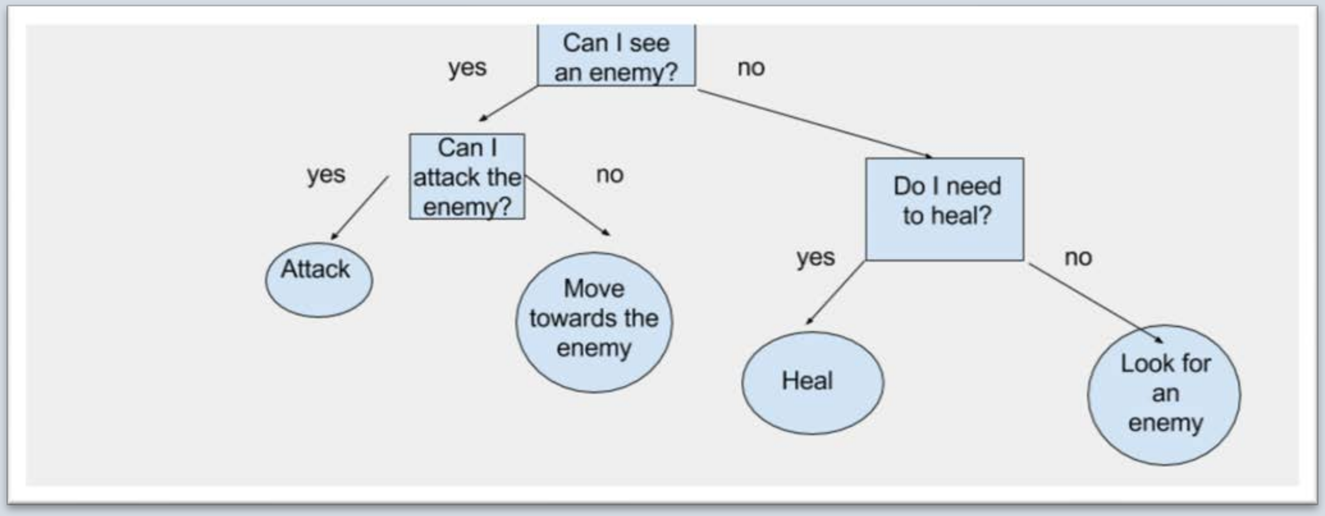

Figure 1: An example of a decision tree followed by a simple enemy.

Figure 1 provides an example decision model for an agent following a decision tree. Squares represent decision points. Arrows represent the connection to the next node in the tree. Once a circle is reached, that action is requested.

Decision Trees in Games

Games that will be able to benefit most from decision trees are those with small rule sets and performance in mind. Agents in the digital card game *Hearthstone* follow a rule-based decision model as the set of considerations is small and reasonably consistent (Dill, Rabin, & Schwab, 2010). Decision trees are also a common choice during the incredibly early stages of development due to their direct and simple nature.

Noteworthy Attributes

The greatest strength of rule-based systems is the speed at which they can be developed. They are also relatively easy for non-programmers to understand and implement on their own. Their simplicity also allows them to be highly modular.

While fast to implement, rule-based systems do not scale well beyond simple decisions. Adding or adjusting one action possibility means that every other possible action must have its own considerations adjusted as well. While this is true for most decision models, decision trees are the most resistant to this process.

Conclusion

Decision trees are great for simple games with short agent lifespans or lean performance requirements. However, they are ill-suited for games with more than a few action possibilities.

They can still be useful during the prototyping phase of development, but in most cases a more flexible decision-making system will be needed.

State Machines

State machines limit action possibilities by mapping each possible agent state to a specific course of action. Before starting the decision making process, the agent's current state is determined by considering its internal and external knowledge. Once the state has been determined, the corresponding action is performed for as long as the agent remains in that state.

Each state is connected to other possible states by any number of transition conditions. Similar to decision tree options, each of the transition conditions are considered. However, if none of these conditions are fulfilled, the agent remains in its current state.

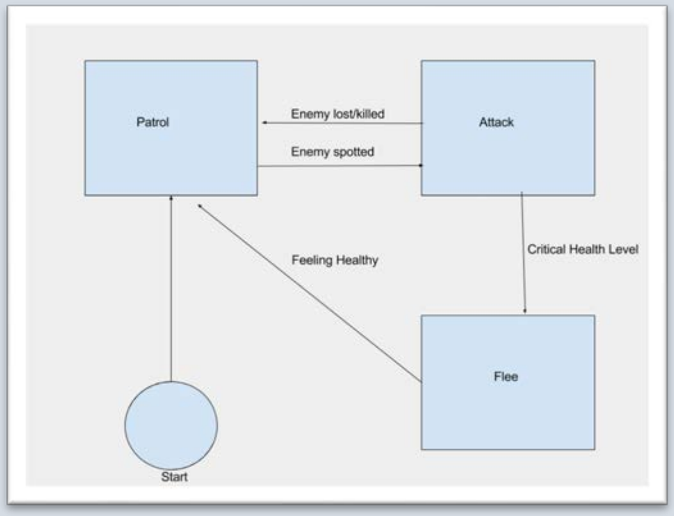

Figure 2: An example of a "Guard" state machine.

Figure 2 shows an example of a state machine that might be used by a typical guard agent. Each square represents a possible state. The arrows represent transitions that can be made be their appropriate conditions are fulfilled. An agent can only be in a single state at any given time.

State Machines in Games

Covenant enemies in *Halo: Combat Evolved* (Bungie Software, 2001) will stand at their posts until they notice the player. Once this happens, the agent transitions into a combat state where it takes cover and begins to fire. Characters in Sony's NBA series also use an advanced version of State Machines (Dill, Rabin, & Schwab, 2010). State machines are suitable for these kinds of games due to the limited scope of the environment that must be considered.

Noteworthy Attributes

State machines retain the benefit of being simple to understand and computationally lightweight. Their initial development time is also short. As more complex features and considerations are added, state machine capabilities scale better than rule-based systems. This is due in large part to their ability to categorize logic components so that only relevant considerations are made.

Although state machines are more scalable than decision trees, their complexity limit is still incredibly shallow. All transitions between states must be specified within code during the design of the state machine. This means that the designer must know all possible transitions from any given state at design time (Sloan, Kelleher, & Mac Namee, 2011). If any of these components are changed after the state machine has been designed, the designer will likely need to re-evaluate the transition set.

It is also not uncommon for a designer to want an agent to make a state transition under very specific circumstances. While the number of possible states might remain small, the number of transition rules can quickly grow beyond comprehension. This can lead to unexpected behaviors that are difficult to detect without a large amount of testing.

State machines also necessitate environment specific programming. The consumes additional developer time that could be spent elsewhere. As a result, state machines are ill-suited for games with a wide range of potential scenarios.

Conclusion

State machines remain a core decision model due to their fast design time and fast response time. They are a great option for both games with short agent lifespans as well as those with a simple ruleset. However, games with expected agent lifespans longer than a few minutes should likely consider more powerful techniques such as behavior trees.

Behavior Trees

Behavior trees expand upon the concept of state machines by replacing the state component with tasks. Tasks can be composed of smaller tasks which, when combined, represent higher level behaviors. Each task is allocated an appropriate amount of computation time to reach a conclusion. While a typical conclusion is binary, some behavior tree models allow for a greater range of outcomes.

Although behavior tree tasks share a common structure, each task's purpose is often further categorized as part of the encompassing decision model. A common model will consist of three types of tasks: conditions, actions, and composites (Millington & Funge, 2009).

Condition tasks are responsible for testing some property of the game. This property can be internal or external knowledge, but it is important that each property is designated its own task. If a more complex condition is desired, this can be achieved by creating a task composed of these smaller condition tasks. Doing so allows for a modular system with manageable complexity and enhancement capabilities. Condition tasks are executed in a means similar to the option selection process of decision trees.

Action tasks are expected to change a game property. Much like condition tasks, these properties can be internal or external. After the action has been performed, its success or failure helps to determine the next task to perform.

A behavior tree composed solely of condition tasks and action tasks would operate in a manner similar to a state machine or decision tree. However, composite tasks empower sophisticated behaviors that are much more difficult to model with a state machine or decision tree. Composite tasks are composed of a collection of other tasks. Composite tasks are categorized by how they process their contained tasks.

It is not uncommon for a wide variety of action and condition tasks to be designed for use in behavior trees. Composite tasks usually have a much smaller set of types designed as each can encompass any number of sub-tasks. Two common types of composite tasks are Selectors and Sequences. Both Selectors and Sequences will execute each of their contained tasks until a desired result is achieved (Millington & Funge, 2009).

Selector tasks will execute each of their composite tasks as long as the task is unsuccessful. One a task succeeds the Selector task will report a successful result. If a Selector runs out of tasks to attempt, it reports a failure.

Sequence tasks are the inverse of Selector tasks. So long as each of its contained tasks are successful, the Sequence will continue. If this sequence of tasks is ever interrupted by a failure, the Sequence halts execution and reports a failure. Sequence tasks only report success once all of their sub-tasks have been successfully performed.

Behavior Trees in Games

Halo 2 (Bungie Software, 2004) was one of the first high-profile games to utilize behavior trees. Since their mainstream debut, behavior trees have become a staple in game development. Both Unreal Engine 4 and Unity feature behavior tree implementations as they can be utilized by almost any genre of game.

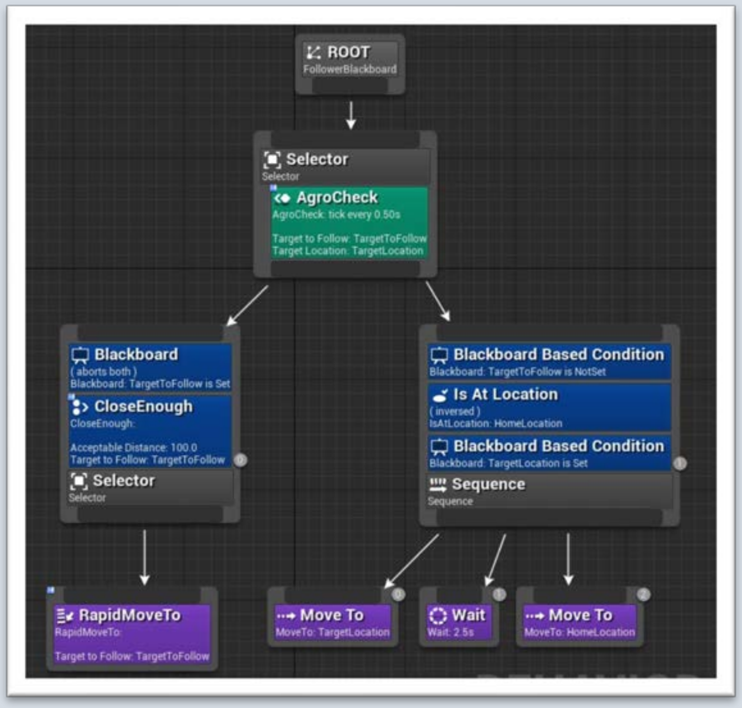
ROOT
FollowerBlackboard
Selector
Selector
AgroCheck
AgroCheck: tick every 0.50s
Target to Follow: TargetToFollow
Target Location: TargetLocation
Blackboard
(aborts both)
Blackboard: TargetToFollow is Set
CloseEnough
CloseEnough:
Acceptable Distance: 100.0
Target to Follow: TargetToFollow
Selector
Selector
Blackboard Based Condition
Blackboard: TargetToFollow is NotSet
Is At Location
(inversed)
IsAtLocation: HomeLocation
Blackboard Based Condition
Blackboard: TargetLocation is Set
Sequence
Sequence
RapidMoveTo
RapidMoveTo:
Target to Follow: TargetToFollow
Move To
MoveTo: TargetLocation
Wait
Wait: 2.5s
Move To
MoveTo: HomeLocation

Figure 3: Epic Games' Unreal Engine 4 features a behavior tree development tool suite.

A game agent following the behavior tree modeled in Figure 3 would behave in the following manner:

- If an enemy has not yet been detected, the agent will stand in place.
- Once an enemy is detected, the agent runs toward the targeted enemy.
- When the agent is close enough to the enemy, the chain of execution will return to the root of the behavior tree. If it can no longer detect the enemy, it will move to the last known location of the enemy.
- After 2.5 seconds have passed, if the agent has still not seen the enemy, it will return to its original position.
- At any point during the task sequence on the right, if an enemy is detected, the agent will abort its current action in favor of chasing its target.

Noteworthy Attributes

The greatest advantage of behavior trees is that they are much more capable of modeling complex logic than decision trees and state machines. They also help to encourage a goal-driven level of abstraction (Dill, Rabin, & Schwab, 2010). By decomposing behaviors into a series of tasks, these tasks can be easily reused as part of future behavior designs. This composition also makes it much easier to organize the decision model into components that are easy for developers to digest.

Behavior trees also have the benefit of a simple form of planning called reactive planning (Millington & Funge, 2009). Selector tasks allow agents to attempt a series of tasks while retaining the ability to default to broader behaviors in the event of failure. While this feature is not unique to behavior trees, reactive planning is much less intuitive for designers if implemented through a simpler system.

While goal-based abstraction is great at saving programmer time and encouraging a modular decision-making approach, it has the disadvantage of requiring non-technical staff to approach behavior design from a perspective that they are likely unfamiliar with. Although goal-based abstraction is not the only way to approach behavior tree design, it is arguably the best use of behavior trees (Dill, Rabin, & Schwab, 2010). Another issue with behavior trees is that it can be difficult to assign priority to different transitions. These can lead to developers creating unnecessarily complex task sequences as a way to “force” priority.

Conclusion

Behavior trees are a powerful architecture with the widespread embrace of the game development industry. Although they can be challenging for novice developers to use, the power and flexibility offered allows experienced teams a standard technique to follow that frees up design time for other tasks.

Goal-Oriented Action Planning

Goal-Oriented Action Planning is another decision-making system that is centered around the concept of goal-based design. Goal-Oriented Action Planning extends the goal-oriented design of behavior trees by tasking the agent with both determining what it needs to do as well as how to do it. These actions are evaluated using a planning system that attempts to find an optimal sequence of actions (Orkin, 2006). Agents using Goal-Oriented Action Planning periodically reevaluate their situation and choose the optimal behavior to achieve whatever goal they determine to be most relevant. Much like behavior trees, a goal in this context is any set of conditions that an agent wants to satisfy.

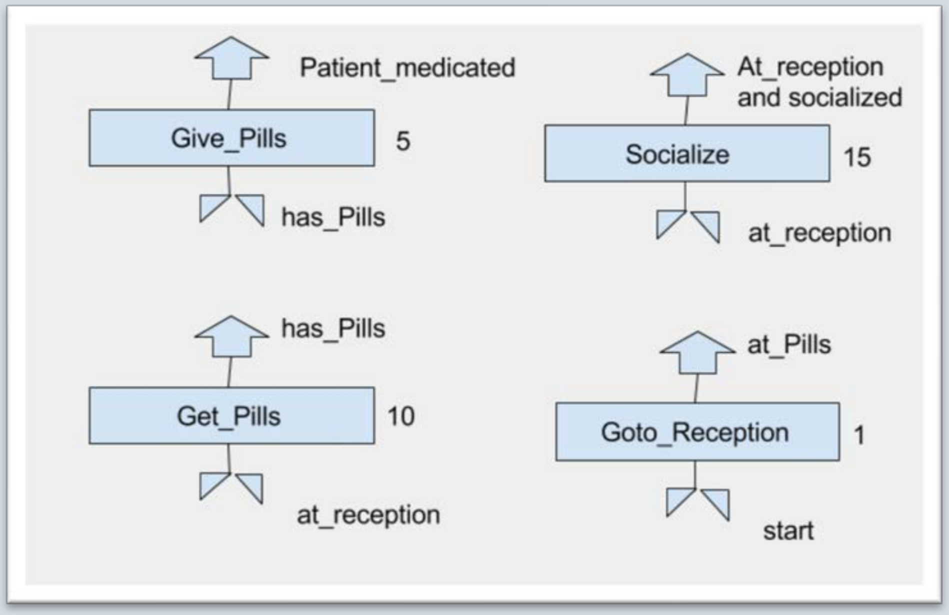

Figure 4: An example of a Goal-Oriented Action Plan.

Figure 4 shows a representation of a nurse in a simulation game whose goal is to medicate a patient. Actions are represented as rectangles. Action costs are the numbers to the right of each action. The bottom symbol is the condition required for an action to be performed. The arrow above the action represents the effects of the action. Utilizing Goal-Oriented Action Planning, an agent has two options when it arrives at reception. It can choose to either get pills or to socialize. In this example, the agent would always choose to prioritize getting pills due to the lower associated action cost. However, designers can change the agent's behavior preference by simply adjusting the cost of socializing, getting pills, or both.

A common modification to goal-oriented systems is the addition of agent personalities. Because each action has a cost associated, designers can create personalities that scale the weights of various types of actions. Applying a socially-oriented personality to the decision model in Figure 4 would lead to an agent that prioritizes socializing over getting pills without needing to change the design of the underlying action space.

Goal-Oriented Action Planning in Games

F.E.A.R. First Encounter Assault Recon (Monolith Productions, 2005) is one of the pioneers of Goal-Oriented Action Planning. At the time of its release, it was critically acclaimed for its movie-like sense of action and immersive behaviors (Herold, 2005). It remains a staple decision-making process in the game development industry due to the flexibility and scalability it permits. The programming requirements behind its implementation can, in some cases, be simpler than those of behavior trees.

Noteworthy Attributes

The underlying algorithm, A* is a keystone algorithm in the game development industry. A* is well-reputed for being easy to implement while still leaving plenty of design space for optimization (Sloan, Kelleher, & Mac Namee, 2011).

Goal-Oriented Action Planning is simple, fast, and can give great results when considering a small action set. It also has the benefit of disconnecting the dependency between states and actions. Additional actions can be created without requiring modification to existing code.

Goal-Oriented Action Planning has two critical weaknesses. The first of these weaknesses is the ability to consider action consequences. Agents utilizing Goal-Oriented Action Planning are only concerned with the most direct sequence of actions that will lead them to their goal. The second critical weakness is the failure to consider time as a factor (Orkin, 2006). Both of these weaknesses are related to the underlying issue with goal-oriented approaches- properly evaluating the context of the action can be difficult to achieve.

Another issue with Goal-Oriented Action Planning is that it is incapable of performing any actions until its entire plan has been calculated. This is due to the nature of its underlying algorithm, A*. Dynamic environments may lead to agent plans becoming quickly outdated and require additional computation time, even if the plan is unchanged.

Conclusion

While Goal-Oriented Action Planning is certainly the most powerful decision-making system discussed so far, it is not without its drawbacks. Despite touting a short development time, it is too resource-intensive to be effective in simple games.

Utility Systems

Utility systems build upon the ideas of Goal-Oriented Action Planning by addressing its two critical weaknesses: time and consequences. This is accomplished by assigning utility values to states instead of actions. Utility serves as a measure of the desirability of a state to an agent after taking its set of goals into consideration. A utility-driven agent will select actions that create states with the highest total utility score (Sloan, Kelleher, & Mac Namee, 2011).

This type of system is ideal for two particular types of situations. The first type is when an agent is assigned conflicting goals. Utility scores resolve this issue so long as the conflicting goals are not equally undesirable. The other type of situation that utility systems excel at is when there is uncertainty and the agent is pursuing multiple goals. Utility allows the agent to select an action sequence that is most likely to succeed.

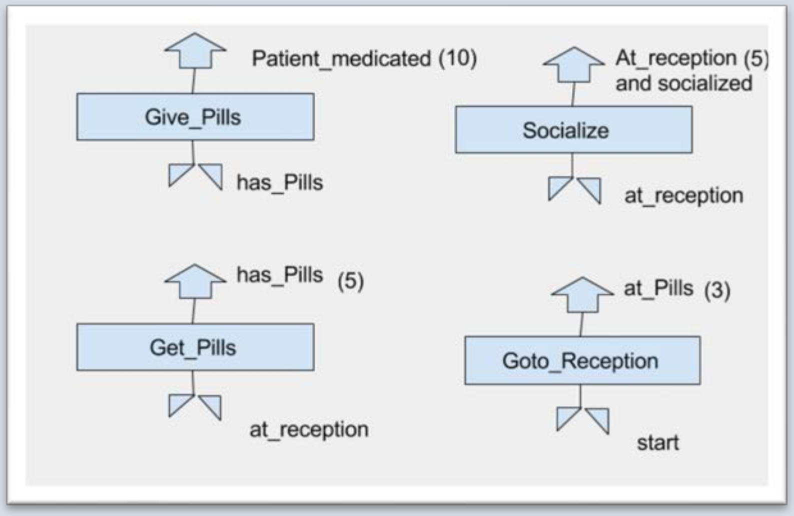

Figure 5: An example of a Utility System

Utility Systems in Games

Utility systems remain a fringe decision model in the game development industry. In recent years, however, they have started to gain traction as a viable approach for games with massive amounts of data. Large data sets empower utility systems to emulate complex systems unachievable by other decision models.

Stellaris (Paradox Interactive, 2016) employs a utility system. Each faction is assigned a personality that has its own set of attributes and goals (Bari, 2017). *Guild Wars 2* (ArenaNet, 2012) is a massively multiplayer online role-playing game that also uses utility systems to drive its agents' behaviors. Both *Stellaris* and *Guild Wars 2* feature a wide array of factions with associated personalities and opinions of one another.

Noteworthy Attributes

Utility systems are easy to add new content with minimal programmer support after release. They are also easy to balance without programmer input. When tested for performance, utility systems were more demanding than state machines, but less demanding than Goal-Oriented Action Planning (Sloan, Kelleher, & Mac Namee, 2011).

Utility systems can be difficult for less technical designers to understand. The most common utility-based implementation, known as a Markov Decision Process, is much more complex than those used in other decision-making systems. Markov decision processes also require a machine learning process known as training (Sloan, Kelleher, & Mac Namee, 2011). Training is a time-intensive process that is ill-suited for a development environment as design components are constantly being adjusted as the game evolves. This is an unnecessary level of complexity for games with a small action space.

Conclusion

Utility systems are incredibly powerful. However, they are unnecessary for simple games and those with rapidly changing environments. They also bear the burden of being the most programmatically challenging decision-making system discussed. When used appropriately by an experienced development team, utility systems are one of the best systems available.

Conclusion

The goal of this report was to provide a broad understanding of the decision-making systems most widely used by the game development industry. To accomplish this, the theory behind these systems was explained before evaluating the impact of these systems in a development environment.

Decision trees are the perfect tool for simple games with short agent lifespans or a mobile platform target. However, they are ill-suited for games with more than a few action possibilities.

State machines remain a core decision model due to their fast design time and fast response time. They retain the benefits of decision trees at the cost of a marginal increase in complexity. However, games with expected agent lifespans longer than a few minutes should likely consider more powerful techniques such as behavior trees.

Behavior trees are the default decision architecture of the industry for a reason. Despite their steep initial learning curve, the power and flexibility they offer makes them an essential consideration for most games.

Goal-Oriented Action Planning is an incredibly powerful decision-making system. Action games, simulation games, and role-playing games benefit the most from this approach. However, the drawbacks associated with it are not negligible. Although it offers an incredibly short development time, it is too resource-intensive to be necessary for simple games.

Much like Goal-Oriented Action Planning, utility systems are a powerful decision model capable of producing high-quality results. However, their sophistication is unnecessary for simple games. Simulation, strategy, and role-playing games can make great use of utility-based systems so long as the development team can afford to devote the time necessary to properly tune them.

Ultimately, the most important factors to consider are the scope of the project and the skill set of the development team. Any of the decision-making systems discussed can be adapted for use in a variety of genres so long as the team is comfortable with the system being used.

References

ArenaNet. (2012, August 28). Guild Wars 2. Bellevue, Washington, United States of America.

Bari, M. A. (2017). Creating Complex AI Behavior in Stellaris Through Data-Driven Design. *Game Developers Conference.* San Francisco.

Blizzard Entertainment. (1996, December 31). Diablo. Irvine, California, United States of America.

Blizzard Entertainment. (1998, March 31). Starcraft. Irvine, California, United States of America.

Blizzard Entertainment. (2014, March). Hearthstone. Irvine, California, United States.

Bungie Software. (2001). Halo: Combat Evolved.

Bungie Software. (2004). Halo 2.

Dill, K., Rabin, S., & Schwab, B. (2010). *AI Architecture Mashups: Insights into Intertwined Architectures*. Retrieved from GDC Vault: http://www.gdcvault.com/play/1012409/AI-Architecture-Mashups-Insights-into

EA Sports. (2017, September 29). FIFA. Burnaby, British Columbia, Canada.

Ensemble Studios. (2009, January). Halo Wars. Dallas, Texas, United States of America.

Herold, C. (2005, October 29). If Looks Could Kill... and Here They Do. *The New York Times*.

Infinity Ward. (2002, May). Call of Duty. Woodland Hills, California, United States of America.

Kojima, H. (1998, September 3). Metal Gear Solid. Minato, Tokyo, Japan.

Ludeon Studios. (2016, July 15). RimWorld. Montreal, Quebec, Canada.

Mark, D. (2009). *Behavioral Mathematics for Game AI.* Omaha: Cengage Learning PTR.

Meier, S. (1991, September). Civilization. Hunt Valley, Maryland, United States of America.

Millington, I., & Funge, J. (2009). Decision Making. In I. Millington, & J. Funge, *Artificial Intelligence for Games* (pp. 293-491). Burlington, MA: Morgan Kaufmann Publishers.

Mojang. (2011, November 18). Minecraft. Stockholm, Sweden.

Monolith Productions. (2005). F.E.A.R. First Encounter Assault Recon.

Orkin, J. (2006). *Three States and a Plan: The A.I. of F.E.A.R.* San Francisco: Game Developers Conference.

Oxford University Press. (2017). *Definition of decision model in English*. Retrieved from Oxford Dictionaries: https://en.oxforddictionaries.com/definition/decision_model

Paradox Interactive. (2016, May 9). Stellaris. Stockholm, Sweden.

Rollings, A., & Adams, E. (2003). *Andrew Rollings and Ernest Adams on Game Design.* San Francisco: New Riders Games.

Sloan, C., Kelleher, J. D., & Mac Namee, B. (2011). *Feasibility Study of Utility-Directed Behaviour for Computer Game Agents.* Lisbon: ACM.

Squad. (2015, April 27). Kerbal Space Program. Mexico City, Mexico.