Developers of game AI are always interested in cramming more complexity into the virtual brains they build. However, complexity often has a price, or rather has many prices: poor run-time, poor scalability, a lack of directability and worst of all, a murky experience for the player in which the AIs seem to act "randomly" rather than "intentionally". We will discuss the sources of complexity, the various ways in which complexity can manifest itself and also some of the architectural approaches we took in Halo 2 to mitigate these negative effects. This discussion will center on the problem of scalable decision-making, but will also touch on questions of memory, knowledge model, and control representations for the scripting and influencing of the AI by level designers.

The Brute Force Approach to Common Sense

When it comes to game AI, more is often better, in the sense that the more well-rounded an AI's behavioral repertoire is, the more unique triggers an AI can recognize, and the more unique ways an AI can respond, the more likely we are to see the AI as a common-sensical kind of creature. A "common sense" AI is a long-standing goal for much of the research AI community. For many (for example, [Lenat95] and [Stork99]), the question of common sense is intimately connected to the problems of knowledge acquisition and representation. After all, common sense can simply be considered the massive database of mundane, everyday knowledge that is so obvious to the walking, seeing, thinking human being that it never really needs to be taught or even expressed. This makes common sense a very elusive thing indeed.

For games, or at least for Halo 2, we are far less interested in encoding factual knowledge (birds have wings, water gets things wet) than we are in encoding behavior, which is perhaps a different sort of knowledge. This is the knowledge that says that when you are sitting in a vehicle seat, you have to get out of the seat before you can walk through the door, or that in order to stop someone from shooting you, you need to move in order to place a large sturdy barrier between you and your attacker. In both styles of common sense however, the solution is the same: quantity. The more the AI knows, the better.

Quantity, of course, is complexity, especially when considered along with some of the other constraints that The Game forces upon us. It is not enough that the AI be able to do it a lot of things, it is equally important that they do all those things right, at the right times, and in a way that does not break the illusion of life, or threaten the player's understanding of the AI's intentions or motivations. In Halo 2, the AI works best when the player believes he is fighting a living breathing (evil) creature, and can respond to and predict that creature's actions accordingly. As authors of behavior, one of our primary goals is to facilitate the on-going narrative that is taking place in our player's head: "oh, the grunt just ran away screaming because I pulled out my energy sword and it was scared, but when I had it cornered, it turned around and started fighting again".

Attaining both these goals - quantity on the one hand and what we might term behavioral integrity on the other - is a huge architectural challenge. Because whatever the content of the knowledge we encode, we need an appropriate container to put it all in, hopefully a container that addresses the perennial large-system-design concerns of scalability, modularity, transparency and so on.

We pay for complexity in a number of ways:

- **Coherence:** If behavior is action over time, we need to make sure that our AIs start, stop and change actions at appropriate times. And we must avoid at all costs the problem of dithering (the rapid flipping back and forth between two or more actions).

- **Transparency:** given the AI's outward stance, it must be possible for the untrained observer to make reasonable guesses as to the AI's internal state as well as explain and predict the AI's actions.
• Run-time: The most obvious of all constraints. The AI has to run at 30Hz or more.

• Mental bandwidth: When we lose the ability to reason about what’s going on in the system, we lose control over it.

Quantity in service of common sense is not the only sources of complexity. Consider these others:

• Usability: The AI must be directable enough to support the larger fictional setting of the game. The "user" in this case is not the player but the level designer, who must craft a drama over the course of the level through character placement, scripting and high-level direction.

• Variety: Different AIs behave in different ways according to their character. How do we design a system that provides a base of robust common sense behavior but that also allows for character-to-character variety?

• Variability: AIs should behave different in different situations, especially when those situations are directed by the designers in service of the story (for example, one scene might demand that the player-ally AI be holed up, defending themselves from an onslaught that the player will ultimately rescue them from, while the next might send the same AI out on an assault with the player).

• Run-time: the one concern that can both suffer from and contribute to complexity. Much of the complexity of an architecture like Halo 2’s stems from our desire to avoid work we don’t need to do.

This paper will discuss some of the techniques that Bungie used in the implementation of the Halo 2 AI to handle the burgeoning complexity problem. The first half of the paper will deal mostly with questions of internal architecture, particularly as it relates to memory and decision-making. The second half of the paper will present some of the useful level-design tools we used to control AI and script levels.

Core Combat Cycle

In the beginning, it’s all very simple. It probably starts out looking something like Figure 1. This is the kind of diagram a game designer might come up with to describe the ways in which a player may interact with AI. Clearly each of the states shown describes very different modes of behavior for our characters, preferably with their own animation styles (sneaking for searching, flailing panic for flight, etc.) How might we go about implementing this scheme?

![Figure 1: The combat cycle](image)

The first thing to recognize is that the figure contains all kinds of hidden complexities. For example, for each of the arrows we have a question of "when is it appropriate to follow this transition?" Some of the transitions are voluntary (for example, the decision to give up searching and return to idle). Others are forced by perception: clearly from combat
we are forced along a transition, either to *idle* or to *search*, when our target steps behind an obstacle. In other words, the diagram is a useful conceptual tool (particularly for designers), but falls far short of being implementable.

**Behavior**

What does the actual control structure look like? Like many systems, the *Halo 2* AI implements a hierarchical finite state machine (HFSM) or a behavior tree, or even more specifically, a behavior DAG (directed acyclic graph), since a single behavior (or behavior subtree) can occupy several locations in the graph. An example is shown in Figure 2. This is a highly abbreviated version of the actual core behavior DAG of *Halo 2*, which contains on the order of 50 different behaviors.

HFSMs are a well-known and time-honored technique for decision-making. We will therefore confine our current discussion to some of the "special features" we found useful in *Halo 2*.

**Decision routines**

In a typical HFSM scheme, the role of non-leaf behaviors is to make decisions (specifically, decisions about which of its children to run), while the role of the leaf behavior is to get something done. When it comes to the decision-making process that takes place in the former, there are two general approaches: (a) the parent behavior can make the decision using custom code, or (b) the children can compete, with the parent making the final choice based on child behavior desire-to-run, or relevancy. Both options will in fact be useful to us at different times, so we leave the ability to write customized decision routines on the table.

**Design Principle #1: Everything is customizable**

Where we can, we will use the more general mechanism (option b), particularly for some of the core components of the combat cycle, each of which will be parent to many children (on the order of ten to twenty). Using this approach for these parents is a good idea, since writing hard-coded logic to choose between one of twenty options can be tedious, as well as unscalable.
Assuming we do use a child-competitive decision model, how do we actually go about picking a winner? Numerous systems feature an analog activation desire: each child provides a floating point number indicating its relevancy, and the child with the highest relevancy wins (with the previous tick’s winner given an added bonus to avoid dithering). This does, however, again face a scalability problem once the number of competing behaviors gets above a certain number, especially when a very specific set of priorities is desired (for example, “fight the target, unless the player drives up in a vehicle, in which case get in his vehicle”). Tweaking floats in order to get a specific set of rules or priorities is feasible when there are two or three choices, but when there are twenty or more it is almost impossible.

We will simplify this scheme considerably by making relevancy a binary test. Using this approach, we were able to define a small number of standard decision schemes:

- **prioritized-list**: march down a prioritized list of the children. The first one that can run, does, but higher-priority siblings can always interrupt the winner on subsequent ticks.
- **sequential**: run each of the children in order, skipping those that are not currently relevant (and never revisiting it). When we reach the end of the list, the parent behavior is finished.
- **sequential-looping**: same as above, but when we reach the end of the list, we start again.
- **probabilistic**: a random choice is made from among the relevant children.
- **one-off**: pick in a random or prioritized way but never repeat the same choice.

Of these, by far the most commonly used is the prioritized-list scheme. It has a number of great advantages, not the least of which is that it is closely in line with the way that we generally think of solving problems: we think first of the
best thing to do, but failing that we will consider the second best, the third best and so on. Whichever we choose, when a better option opens up, we immediately switch to it.

**Behavior Impulses**

However, this presents a new problem: what about when the priority is not fixed? In other words, under certain circumstances behavior A has priority over behavior B ("fight rather than getting into a nearby vehicle") but under other circumstances, B has priority over A? ("Unless the player is in the vehicle, in that case do get in.") To solve this problem, we use a behavior impulse. An impulse is a free-floating trigger which, like a full behavior provides a binary relevancy, but is itself merely a reference to a full behavior. When the impulse wins the child competition either the current stack of running behaviors is redirected to the position of the referenced behavior, or the referenced behavior is simply run in the position of the impulse. In the example given above, we are interested in the latter. Our priority list becomes

```
player_in_vehicle_impulse
fight_behavior
enter_vehicle_behavior
```

The important point here is that we have explicitly separated out the condition that would have made the `enter_vehicle_behavior` more desirable, into a separate impulse that nonetheless references the same behavior.

**Design Principle #2: Value explicitness above all other things**

As mentioned, impulses can also serve to redirect the current behavior stack to another portion of the tree. For example, there might be self-preservation impulses (self-preserve due to damage, self-preserve when facing a scary enemy, etc.) that are children of the `engage` behavior - thus the impulses are only considered when the AI is engaging. When one of these impulses is chosen, rather than running `self-preservation` under `engage`, we simply pop up a level in the tree and run self-preservation in its native position. The semantics for how this redirection is performed (in particular what level of the tree to search for the referenced behavior, and what limitations to place on the reference itself) are somewhat involved. Suffice it to say that impulses can at times act as "pointers" to other branches of the tree and cause execution to jump to that branch.

Impulses serve us well in another way. Consider an impulse that never returns a positive relevancy: this impulse will never provide us with a referenced behavior to run. On the other hand, this is an arbitrary, lightweight piece of code that can itself be run at very specific point in the behavior DAG. What might we use this code for? Anything. Perhaps we could make a data-logging call, to record the fact that we reach that point in the priority list. Or perhaps we wish to spew some debugging information to the console. Or perhaps we wish the character to make a certain sound whenever a certain condition is met. The fact is, that the code does not need to be explicitly part of a behavior to do something useful. Might this be considered a hack? In some cases, yes, since we are specifically bypassing the behavior performance step (which says that a behavior can only do real work when it is officially chosen), but in fact this is one of the design purposes of the impulse construct: to give us a convenient way to place arbitrary pieces of code at specific points in the behavior DAG.

**Design Principle #3: Hackability is key**

Hacks are going to happen. When they do, we must make sure we have a way of containing them. This system also imposes a healthy discipline on our hacks, since one is required to label and, in the case of the *Halo 2* codebase, list them in a global list of impulses, thus making it very unlikely that we will lose track of them and forget that they are there.

**Behavior Tagging**
As trees grow to large sizes, we can easily imagine that determining behavior relevancy would become one of the principle contributors to run-time. After all, we are often checking the relevancy of numerous behaviors and impulses that are not actually running. Often, however, we find that many of the basic relevancy conditions are the same across many candidates. For example, in *Halo 2*, vehicle status (is the actor a driver, a passenger or on-foot) and alertness status (is the AI in visual contact with a target, simply aware of a target, or unaware of any targets) are practically always checked when determining relevancy.

The idea of behavior tagging is to move these common conditions out of the relevancy function (thereby avoiding having to write the same code over and over again) and encode them in a tag for the behavior, which is checked directly at decision-time. In *Halo 2*, these conditions are encoded as a bitvector, which is then simply compared with another bitvector representing the AI's current actual state. Behaviors and impulses whose conditions are satisfied undergo the full check for relevancy. The others are ignored entirely.

While this can be considered simply a way to speed up the relevancy check, there is another interesting interpretation. We can see these conditions as locking and unlocking large portions of the behavior tree, thus modifying its fundamental structure. For a passenger of a vehicle, for example, the unlocked portions of the tree are very limited: major branches controlling fleeing, self-preservation and searching, for example, are unavailable to it. A vehicle driver has much more available to it, but still not as much as an infantry AI. If we were to look closely at the engage behavior we would find something else: that the fighting behaviors of a driver and an infantry unit are different, the infantry unit using the *fight_behavior* and the driver using the more specialized *vehicle_fight_behavior* (the latter keeps the AI moving around constantly, whereas the former tends to pick points and stay there). Similarly the process of searching is very different for a driver versus an infantry unit, mostly for the presence in the case of the latter of a number of coordination behaviors that make searching a group activity.

This is the first of several techniques we will present that affects the decision-making process through direct modification of the structure of the tree itself.

**Stimulus behaviors**

Here is another redundancy concern: imagine a "flee when leader dies" impulse. This impulse essentially waits until an "actor died" event occurs, then it springs into action, testing whether the actor that died was a leader, whether there are other leader actors in the vicinity, etc. If all its conditions are satisfied, it triggers a flee behavior. The problem is that given the architecture we've described, this impulse would need to be tested every single tick. We would like to avoid the need to evaluate this impulse continually when we KNOW that no "actor died" event has occurred. We would like, in some sense, to make this impulse "event-driven".

One way we might consider doing this is through a **stimulus behavior**. This is a behavior or impulse that does not appear in the static tree structure, but is instead dynamically added by an event-handler to a specific point in the tree. In the given example, the actor would receive an "actor died" event asynchronously with its main update loop (in *Halo 2* these...
sorts of event notifications happen through a callback). Assuming it is then determined that the actor that died was of a leader class, this causes a `flee_because_leader_died` stimulus impulse to be added to the behavior tree of the receiving actor. This means that for a given a period of time (one or two seconds in _Halo 2_), that impulse will be considered for execution along with all the other static behaviors and impulses.

![Figure 4: a dead-ally event is turned into a "leader dead retreat" impulse which is dynamically added to the root child list. This impulse will cause a retreat behavior, interrupting engage, postcombat and idle but not self-preservation or retreat itself (if the AI is already running it).](image)

Why is it important that the impulse be placed into the actual behavior tree? After all, we could simply force the actor to start running a behavior based on some decision local to the event-handling code. We don’t do this, because it would not, in a sense, be a well-thought-out decision unless it was made in the context of the full tree. In the above example, we would not want to consider fleeing if we were already running `enter_player_vehicle`, but could if we were simply running `engage`. It would be ludicrous, not to mention highly unscalable, to test these conditions in the event-handler. Only by placing the stimulus behavior into the tree itself can we be assured that all the higher-level and higher-priority behaviors have had their say before the stimulus behavior can consider taking action.

This is an important point, because it underlines the fact that tree-placement constitutes as large a part of the decision process for a behavior or impulse as does its relevancy function. Note also that nothing prevents the stimulus behavior from being a non-leaf behavior, thereby allowing the addition of entire behavior subtrees in an event-driven way. Thus we are again finding a way to modify the structure of the behavior tree in order to get the precise set of behaviors considered.

**Custom behaviors**

We can take a similar approach when considering the problem of character-type to character-type variety. Presumably in fashioning the repertoire of a new character, we would like high-level structure to remain the same - the kind of structure shown in Figure 1 - but the details of the transition-triggers to vary. In some cases we will simply tweak behavior parameters to get the desired effect. In other cases where more specific triggers are needed, we will use custom
behaviors. Like stimulus behaviors, custom behaviors are inserted into the tree, in this case in a preprocess step, so that the final prioritized list of children does not need to be recomputed every time. In this way, we can add any number of character-specific impulses, behaviors or behavior subtrees - and it is through these additions that a good amount of the personality of the characters comes through (for example, grunts, the cowardly creatures of Halo 2, will have an inordinate number of retreat impulses, whereas marines have extra action-coordination behaviors, to get them to work together more cohesively).

This approach - of taking a solid base and then adding stuff onto it - is one we will return to:

**Design principle #4: Take something that works and then vary from it**

Of course, if a truly fundamentally different brain is necessary, then starting from a common base may be impractical. Such was the case, in Halo 2, with the flood swarm characters, which had such wildly different basic needs from ordinary characters that they had a behavior DAG that was ENTIRELY custom.

**Memory and Memory**

With large trees, we face another challenge: storage. In an ideal world, each AI would have an entire tree allocated to it, with each behavior having a persistent amount of storage allocated to it, so that any state necessary for its functioning would simply always be available. However, assuming about 100 actors allocated at a time about 60 behaviors in the average tree, and each behavior taking up about 32 bytes of memory, this gives us about 192K of persistent behavior storage. Clearly, as the tree grows even further this becomes even more of a memory burden, especially for a platform like the Xbox.

We can cut down on this burden considerably if we note that in the vast majority of cases, we are only really interested in a small number of behaviors - those that are actually running (the current leaf, its parent, it grandparent and so on up the tree). The obvious optimization to make is to create a small pool of state memory for each actor divided into chunks corresponding to levels of the hierarchy. The tree becomes a free-standing static structure (i.e. is not allocated per actor) and the behaviors themselves become code fragments that operate on a chunk. (The same sort of memory usage can be obtained in an object oriented way if parent behavior objects only instantiate their children at the time that the children are selected. This was the approach taken in [Alt04]). Our memory usage suddenly becomes far more efficient: 100 actors times 64 bytes (an upper bound on the amount behavior storage needed) times 4 layers (in the case of Halo 2), or about 25K. Very importantly, this number only grows with the maximum depth of the tree, not the number of behaviors.

This leaves us with another problem however, the problem of persistent behavior state. There are numerous instances in the Halo 2 repertoire where behaviors are disallowed for a certain amount of time after their last successful performance (grenade-throwing, for example). In the ideal world, this information about "last execution time" would be stored in the persistently allocated grenade behavior. However, as that storage in the above scheme is only temporarily allocated, we need somewhere else to store the persistent behavior data.

There is an even worse example - what about per-target persistent behavior state? Consider the search behavior. Search would like to indicate when it fails in its operation on a particular target. This lets the actor know to forget about that target and concentrate its efforts elsewhere. However, this doesn't preclude the actor going and searching for a different target - so the behavior cannot simply be turned off once it has failed.

Memory - in the psychological sense of stored information on past actions and events, not in the sense of RAM - presents a problem that is inherent to the tree structure. The solution in any world besides the ideal one is to create a memory pool - or a number of memory pools - outside the tree to act as its storage proxy.

When we consider our memory needs more generally, we can quickly distinguish at least four different categories:
- Per-behavior (persistent): grenade throws, recent vehicle actions
- Per-behavior (short-term): state lost when the behavior finishes
- Per-object: perception information, last seen position, last seen orientation
- Per-object per-behavior: last-meleed time, search failures, pathfinding-to failures

![Figure 5: the anatomy of memory](image)

The first type is the easiest - we can simply define named variables inside the actor object that particular behaviors know how to read or manipulate - needless to say, the number of behaviors that actually NEED to keep persistent state is best kept to a minimum) and the state they DO keep is best kept small (otherwise we simply run into the same problem of exploding memory usage). The second type is the type we have been discussing. This is the volatile behavior state that is allocated and deallocated as the particular behavior starts and stops.

Things become more complicated with the third and fourth types of memory. What they suggest is that there needs to be an actor-internal reference representation for each target the actor can consider. Indeed, having such a representation has a lot of benefits.

The benefits on the perception side have already been discussed at length in [Greisemer02] and [Burke01]. In *Halo 2*, these representations are called “props”, and their primary function is as a repository for perceptual information on objects in the world. Having this state information (position, orientation, pathfinding location, etc.) distinct from the actual world-state and gated by the actor’s perception filter (an actor should not be able to see through walls, for example) allows the two representations to occasionally diverge - thus the actor can believe things that are not true, and we now enter the realm of AI that can be tricked, confused, surprised, disappointed, etc. It is on the basis of this *believed* state, of course, that the actor will be making most of its decisions.
What is new here is that there are benefits as well on the behavior side, as the "prop" can act as a convenient storage location for per-object per-behavior memory. Keeping this behavior state in the same location as the perception history also allows us to conveniently correlate the two, thus making it efficient to answer questions like "have I already searched for the enemy I'm hearing?" As before, the fewer behaviors that actually need to keep per-object persistent storage, the better, and that storage needs to be kept small.

The "prop" representation is one of the cornerstones of the Halo 2 AI, and essentially forms the entirety of the AI's knowledge model - its view and understanding of the world around it. This model unfortunately remains extremely rudimentary. It is after all, simply a flat list of object references, with no form of spatial-relation (is-next-to or is-on-top-of), or structural-relation information (is-a-part-of) and very little time-based analysis (a series of position-reading suggesting a certain trajectory, for example). Furthermore, a major limitation of our implementation is that only potential targets are allowed on the list, like bipeds and vehicles, thus excluding other potentially behavior-relevant objects, such as pathfinding obstacles, machines, elevators and weapons.

Nonetheless, giving AI an internal mental image of its world not only results in interesting and more "realistic" behavior, it also allows us to overcome one of the major storage problems associated with the behavior tree.

**Designer Control**

Let us consider complexity from another point of view, namely, that of usability and directability. In this case, we are concerned not with whether the AI is acting believably, but rather with how easy it is for the users of the AI system - the level designers - to make use of the system to put together a dramatic and fun experience for the player.

This may seem like a dramatic shift in pace, but keep in mind that this is an area that is equally beset by the problems of complexity. Consider, for example, the problem of parameter creep. There are many different types of characters. There are many different behaviors that each of them can execute. Each of those behaviors is controlled by a small number of parameters. Combine these factors and what we have is an explosion of inscrutable floats. Which one of the hundreds of potential numbers is it that is making a particular enemy "feel wrong"? It is very difficult to tell indeed.

The designer needs to tell the AI what to do - but at what level? Clearly we are not interested in a scripting system in which the designer specifies EVERYTHING the AI does and where it goes - that would be too complex. We do need, however, the AI to be able to handle high-level direction: direct them to behave generally aggressively, or generally cowardly. Similarly, when it comes to position-control, we want the direction to be vague: "occupy this general area".

As in the preceding sections, the solution to all these problems lies in a few extremely useful representations.

**Position Direction: Orders and Firing Positions**

As in the case of Halo 1 (see [Greisemer02]), AI position is controlled through designer-placed firing positions. Firing positions are simply discrete points which the AI can consider as destinations when performing spatial behaviors. For example, if a target takes cover behind an obstacle, the AI can try to uncover the target by going to a firing position which has a clear line of sight to the target's current presumed position ("presumed" because the target may have moved). Similarly when running the fight_behavior, an appropriate firing position is chosen from which to shoot at the target (a position which again has a clear line of sight, and which also puts the AI at an appropriate range from the target based on the kind of weapon being used and other factors).

Firing positions become an extremely useful control mechanism when we begin to script the set of firing positions available to the AI at a given time. In Halo1, AIs were grouped into encounters, which also contained a set of firing positions. Various subsets of this set were made available to the AI depending on the state of their encounter (have many of their allies been killed? Are they winning? Are they losing? This was a mapping that was created by a designer). In Halo 2, the basic ideas remain the same, although the representations are different. Instead of having a
single encounter structure, we now have squads, groupings of AI and areas, groupings of firing positions. Forming the mapping between the two is a new structure called the **order**.

Fundamentally, an order is simply a reference to a grouping of firing positions. When the order is "assigned" to a squad, the firing positions referenced by the order become available to the AI in the squad. This is a simple mechanism, which is made slightly more complex by the fact that orders also incorporate some rudimentary scripting functionality that allows for automatic transitioning between orders. A set number of possible trigger-types are available to the designers (for example, "have x or more squad members been killed?" "has the squad seen the player?"). When a trigger condition is satisfied, the squad is assigned a new order associated with that trigger. Thus, designers can script the general flow of a battle using very simple high-level representations.

**Behavior Direction: Orders and Styles**

The idea behind the term "order" is that it should indeed embody the same sort of level of direction that might an order given by a company commander to his soldiers. "We're going to take that hill!" "We're going to occupy that bunker and hole up until the cavalry arrives." Most of these orders are of the "go here and do this" variety, or, to be more precise, "go here and **behave this way**."

So far we have only described how our order representation encodes the first part of that directive. However, the order is a fantastically useful level of representation for the second part as well. In *Halo 2*, designers can, for example, allow or disallow vehicle use, engage stealth and control the rules of engagement (don't-fire-until-fired-upon, versus free-for-all) through special-purpose flags contained in the order.

![Figure 6: Orders and styles](image)

Orders influence behavior in another important way: they reference a **style**. The style represents the final and perhaps most direct mechanism through which we can control the structure of an AI's behavior tree. The style is really just a list of allowed and disallowed behaviors. Just as a behavior cannot be considered if its tag does not match the actor's current state, a behavior cannot be considered unless it is explicitly allowed by the order's **style**.
Given the directness of the style mechanism, it is a very powerful and very dangerous tool. In particular it is possible to give the AI a style which will literally not allow the AI to run ANY behavior, or which will leave the AI in such a debilitated state that its behavior appears essentially random. For these reasons, styles are not generally edited per-encounter. Instead, the designers have a small style library to choose from when setting up an order (each style in the library having gotten the seal of approval from both the lead designer and the AI programmer). But this caveat aside, styles allow for some interesting variability. Defensive styles do not allow *charge* or *search* behaviors. Aggressive styles do not allow self-preservation. Noncombatant style would not allow any combat behaviors at all, instead allowing only *idle* or *retreat* behaviors. Styles also allow the designer to skew some of the parameters controlling behavior one way or the other (for example, allowing characters to flee more easily in a cowardly style).

Orders and styles are one of the principle ways in which we allow for encounter-to-encounter variability in the gameplay. Using these two tools, the designer can make the same AI feel and play quite differently from one moment to the next - presumably in accordance with the dramatic needs of the story and the level progression.

**Parameter creep**

![Figure 7: the character hierarchy](image)

We will discuss a final problem related to complexity that faces our designers, a problem that we have already mentioned, namely that of parameter creep. The tendency in first authoring a behavior is to allow great customizability through the use of any number (usually about three to five) of designer-edited parameters. However, take three parameters, times 115 or so behaviors, times 30 or so character types (including fundamental types and variants, such as red versus white elites) and we have about 10,350 different numbers we need to maintain!

Clearly this is not a tenable situation. We can greatly reduce this burden on the designer, however, if we remember our design point #4: start with something that works well and then vary.

The greatest source of character types in terms of sheer number is the existence of character variants. A white elite fights more aggressively and is tougher than an ordinary red elite, and so is a different character type. In all other respects however, the white elite and the red elite are identical. It is therefore a waste to have to create an entirely new full set of behavior parameters when we’re really only interested in the *fight* and *vitality* parameters. What we therefore have, is a system which allows us to define only those parameters that are truly distinctive to a character, and then to rely on a "parent" character for the rest.
All character and behavior parameters are contained in a .character file. This file gives a character name and also specifies which geometric model to use for the body. When a designer places an AI, he or she first chooses the .character file to use.

The character file is not, however, a flat list of parameters. It is instead a list of blocks of parameters, each block grouped in a logical way to control certain aspects of behavior - the self-preservation block, the combat block, the weapon-firing block, etc. Not all blocks need be present in all character files. When an AI is attempting to run a particular behavior only to find the relevant block missing from its character file, it looks instead in the referenced parent character file. If that file does not have the block, then its parent is examined, and so on. Thus a character hierarchy is formed, in which each child defines only significant variations from its parent. The root of the entire tree is the generic character, which should define "reasonable" parameter values for all blocks.

Conclusions

The "design principles" listed in this paper are a rather transparent attempt to impose a structure on what might otherwise appear to be a random grab-bag of ideas - interesting, perhaps, in and of themselves but not terribly cohesive as a whole. In conclusion, we only hope only to drive home two major points: first, that complexity is paid for in many ways, including run-time, implementation, user-experience and usability. And second, that key to tackling the complexity problem always is the question of representation. All of the tricks described here in some way or another involve the manipulation of a convenient representation structure - be it the behavior DAG, the order/style system or the character hierarchy. This is a fitting "realization" for an AI paper, since it is, of course, nothing more than the recapitulation of an idea that academic AI has known for a long time: that hard problems can be rendered trivial through judicious use of the right representation.

References

[Alt04] G. Alt, "The Suffering: A Game AI Case Study", in the proceedings of the Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence (AAAI), 2004


[Lenat95] D. Lenat, "Cyc: A Large-Scale Investment in Knowledge Infrastructure", Communications of the ACM 38, no. 11, November 1995