

Extending Multi-Agent Coordination in *Neverwinter Nights*

Steven E. Matuszek

Computer Science Department

University of Maryland Baltimore County

smatus1@umbc.edu

Abstract

Many computer games include artificial intelligence components to control the opposition to the human player. When the opposition consists of multiple actors, they are often implemented as multiple instances of one agent. One such game is *Neverwinter Nights*. In this project, the behavior of the default agent and its limited communication with other agents are examined. Extensions to the agents' frameworks for representing capabilities, taking on team roles, communicating, and acting according to human "personality" are implemented, and applied to the domain of joint combat actions. Finally, the behavior of the extended agents is compared to that of the original agents on the bases of combat effectiveness and realism.

Introduction

Neverwinter Nights is a networked multi-player computer role-playing game that is based on the *Dungeons and Dragons*¹ tabletop role-playing game.² In D&D, one or more players control characters such as warriors and wizards in an imaginary world. The mechanics of gameplay are run by the *Dungeon Master*, who also controls and describes the actions of all people, monsters, and creatures with whom the player characters interact.

This game can easily degenerate into a "hack and slash" campaign in which the player characters (PCs) simply kill monster after monster. A good DM, however, will provide more depth of character, strategy, and interaction to the non-player characters.

That is easy for a human DM to do. But what about a computer program?

¹ *Neverwinter Nights* is TM and [®] BioWare Corp. *Dungeons and Dragons* and all related indicia are TM and [®] Wizards of the Coast, Inc., a subsidiary of Hasbro, Inc.

² More specifically, it is based on *Advanced Dungeons and Dragons*, 3rd edition, but we shall refer to the property generically.

Neverwinter Nights provides an excellent architecture for controlling the monsters, allies, and other non-player characters (collectively, NPCs). The same toolset that is used in-house at BioWare is provided along with the game, including its full-featured scripting language based on C [Darrah 2001], and all the scripts that encapsulate default behaviors.

In this project, we will examine the structure and default behavior of the game agents, and the emergent team behavior that results. We will then implement further stored knowledge and procedures for the agents in the areas of representing their capabilities, being assigned the most appropriate roles based on their capabilities, and carrying out those roles in combat. We will also address the issue of alignments as they relate to mainstream multi-agent systems issues of trust and rationality. We will use our roles to try to improve combat effectiveness, provide narrative realism, and allow a framework for the agents to later determine trust levels for one another based on their knowledge of the other agents' capabilities. Finally, we will evaluate our results.

Events and reactive AI

In *Neverwinter Nights*, each NPC is in fact represented by an agent with its own thread of control. A level-of-detail algorithm is used to share processing resources, with agents closer to the PCs getting more processing time [Brockington 2001].

The agent architecture is explicitly a *reactive* one. This is sensible given the domain, allowing for simple reactions such as

- If I perceive a PC, attack
- If I am down to half my health, drink a healing potion
- If I am blocked by a locked door, attempt to pick the lock
- When I am killed, yell "Argh!"

or more complex reasoning such as

- If I have been hurt, target the creature who hurt me. If that creature cannot be found,

target the creature that is attacking my ally. If there is no such creature, target the nearest creature that is an enemy. If I am a warrior, attack the target. If I am a wizard, attack the target with a spell. If I am a dragon...

Note that the decision structure allows the same script to be used by agents with different specializations.

There are thirteen events to which an NPC will respond. For each event, there is a default script, which can be replaced by a different script on a NPC-by-NPC basis. These event handlers include OnDeath, OnPerceive, OnDamaged, OnHeartbeat (which fires approximately every six seconds, providing a place to put deliberation or housekeeping), and OnUser-DefinedEvent, which allows the programmer to define his or her own events.

The OnConversation handler encapsulates two important abilities. First, the ability to respond to certain keywords for which it is always listening; these may be as simple as *attack*, but to avoid nasty surprises, the keywords more often look like NW_I_WAS_ATTACKED and are spoken “silently”—the PCs do not hear them. Second, any creature can work its way through a pre-defined conversation tree, giving the PCs choices of what to say. Any node can also have a script attached to it, making this an excellent way to request actions of NPCs.

In **Figure 1** is an excerpt from the default OnDamaged handler. GetMaxHitPoints, GetIsEnemy, GetHitDice, GetFleeToExit, ActionMoveToLocation, ActionDoCommand, and GetLastHostileActor are just a few of the approximately 700 methods that are available to access the game environment and order its agents around.

The first thing one notices, on going through this code, is that on the first line the agent tries to flee! This is not quite as cowardly as it looks, because there is decision logic hidden in the GetFleeToExit method, and in all methods of its ilk. The logic is of the form:

```
if now is an appropriate time to Foo
{
```

```
    try to Foo;
    if I was successful
        return TRUE;
    else
        return FALSE;
}
else return FALSE;
```

That is, the method will return TRUE if it decided that it would be appropriate to Foo *and* was successful in doing so (a side effect), and will return FALSE if it was unable to Foo *or* decided not to.

In the case of GetFleeToExit, the method will return false immediately unless the creature thinks it is in deep trouble, so the behavior falls through to the subsequent code.

It turns out that the vast majority of predefined behaviors work this way—they are at their hearts a list of priorities.

Barring that decision to flee or an invalid argument,

```
void main()
{
    if (!GetFleeToExit())
    {
        if (!GetIsObjectValid(GetAttemptedAttackTarget())
            && !GetIsObjectValid(GetAttemptedSpellTarget()))
        {
            if(GetBehaviorState(NW_FLAG_BEHAVIOR_SPECIAL))
            {
                DetermineSpecialBehavior(GetLastDamager());
            }
            else if(GetIsObjectValid(GetLastDamager()))
            {
                DetermineCombatRound();
                if(!GetIsFighting(OBJECT_SELF))
                {
                    object oTarget = GetLastDamager();
                    if(!GetObjectSeen(oTarget) &&
                        GetArea(OBJECT_SELF) == GetArea(oTarget))
                    {
                        ActionMoveToLocation(GetLocation(oTarget), TRUE);
                        ActionDoCommand(DetermineCombatRound());
                    }
                }
            }
        }
        else if (!GetIsObjectValid(GetAttemptedSpellTarget()))
        {
            object oTarget = GetAttackTarget();
            if (!GetIsObjectValid(oTarget))
            {
                oTarget = GetAttemptedAttackTarget();
            }
            object oAttacker = GetLastHostileActor();
            if (GetIsObjectValid(oAttacker) && oTarget != oAttacker
                && GetIsEnemy(oAttacker) &&
                (GetTotalDamageDealt() > (GetMaxHitPoints(OBJECT_SELF) / 4)
                 || (GetHitDice(oAttacker) - 2) > GetHitDice(oTarget)))
            {
                DetermineCombatRound(oAttacker);
            }
        }
    }
}
}
```

Figure 1. Excerpt from default OnDamaged script. © BioWare.

this code—which, remember, is invoked when the creature in question gets injured by someone—will usually end up calling `DetermineCombatRound()`. In this method, all combat decisions occur.

Character classes

Before discussing the decisions an agent makes in combat, we should stress that different agents have very different sets of skills.

A creature's *character class* in D&D is its "job description." The traditional four classes are that of wizard, warrior, rogue, and priest (a.k.a. magic-user, fighter, thief and cleric).

The warrior can deal and withstand the most physical damage, and use all kinds of armor and weapons, but has no magical abilities.

The wizard is nearly useless in a physical fight, but can cast magical fireballs, lightning, and so forth (as well as many useful non-combat spells, which we will ignore.)

The priest can banish the undead, can cast healing spells plus some offensive spells, and is almost as good in a fight as a warrior.

The rogue relies on sneakiness; his best bet in a fight is to backstab an enemy that is otherwise occupied.

Plainly, each of these characters is going to have different tactics in combat.

Complicating the issue are the fact that characters can have ranks in multiple classes, and the existence of all the variant classes such as the paladin, a holy warrior with some priestly powers; the druid, a priest of nature, and so forth.

Combat

`DetermineCombatRound` is the method in which, at each "round" (about six seconds) of combat, the creature decides what it shall do next. It is to be hoped that the previous two sections have prepared the reader for the idea that the body of this method shall be a long series of if-elses, each containing a list of priorities.

Herewith an excerpt, for the character classes that have no (or few) magical abilities:

```
else if ( nClass == CLASS_TYPE_FIGHTER ||
nClass == CLASS_TYPE_ROGUE ||
nClass == CLASS_TYPE_PALADIN ||
nClass == CLASS_TYPE_RANGER ||
nClass == CLASS_TYPE_MONK ||
nClass == CLASS_TYPE_BARBARIAN )
{
    // Use healing potions to not die
    if(TalentHealingSelf()) {return;}
    // Use potions of enhancement and protection
    if(TalentBuffSelf()) {return;}
    // Check if the character can enhance itself
    if(TalentUseEnhancementOnSelf()) {return;}
    // Check for Paladins who can turn undead
    if(TalentUseTurning()) {return;}
    // Sneak Attack Flanking attack
    if(TalentSneakAttack()) {return;}
    // Use melee skills and feats
    if(TalentMeleeAttack(oIntruder)) {return;}
    return;
}
```

This doughty warrior will

1. heal herself if necessary and possible,
2. better her chances with magic potions,
3. use any other self-enhancing tricks she has,
4. turn undead (if she happens to be a paladin and happens to be fighting zombies or ghosts),
5. sneak attack (if she's a rogue),
6. or, usually, just wade in and start bashing.

Simple enough—but the wizard must worry about which spell to cast, the priest about whom to heal. This method is about 500 lines of code.

For an agent on its own, this method provides highly sophisticated combat ability. But when a party of creatures is thrown together, there is no coordination among them as to who shall perform what roles in combat. Other than the fact that healers will generally heal whoever is hurt, each agent is still acting entirely on its own.

We can say that a group of agents has *emergent* behavior if it accomplishes something because of the way that the actions of the agents interact, which would not be obvious from the actions of any individual agent. van der Sterren proposes [van der Sterren 2002a] that

Squad behavior might already emerge when squad members share their individual observations and intentions, and base their decisions on their individual state and the perceived states of nearby squad members.

In this case, there is no need for squad members to share observations, since all the agents have access to all the same data (and since it is very difficult to pass around one's own data structures in any case!). But the agents are definitely making decisions (healing) based on the states of their teammates. Perhaps we could further this to help our teammates achieve other intentions, such as defeating a particular enemy.

There is another fantasy role-playing game extant, which, unlike *NWN*, has a persistent online world. This means that the players' characters exist only on the company's servers, as do the monsters, items, and environments. We refer, of course, to *EverQuest*.³

Your author finds that this game becomes repetitive and tiresome, since no player has the ability to create anything new. But it must be noted that a very specific and well-known canon of tactics has emerged from this consistent environment. Could these same tactics work in *NWN*?

In *EQ*, a player has one or more of these basic responsibilities to her group in combat.

- **Tanking:** stepping into the thick of the *melée*, dealing and absorbing physical punishment.
- **Healing:** tracking and maintaining everyone's level of health, especially the tanks'.
- **Buffing:** casting spells that will increase allies' health, strength, speed, defense, etc.
- **Nuking:** casting spells that will physically damage the enemy—but not so much that it comes after you.

Since you are so often in groups with people you don't know, you are expected to toe the line and perform as would generally be expected of those in your character class. If you are a wizard, you'll be nuking. If you are a cleric, you'll be healing and buffing. If you are a warrior, you'll be tanking.

This is important because battles last several minutes—unless something goes wrong, doing anything but

following the optimal tactics will just result in inefficiency.

The question, then, is whether we can use similar tactics in *NWN*. The first step is to help the agents to decide something that human *EQ* players do as a matter of course: decide who should take on what role.

Capability representation

Because the potential actions each agent might take have already been categorized, we could use this as a basis for representing each agent's capabilities. As a first pass, we can do this by duplicating the logic, but capturing all possible actions rather than performing one immediately.

Unfortunately, we run into an abstraction barrier here. The source code is available for many of the provided functions, but not all of them. Those that are missing are presumably written in a lower-level language, since they do things that would make a scripter's head spin, such as performing a simulation to see which weapon would do a monster most damage [Lexicon 2003].

In any case, all that would have given us was a yes-no answer for most attempted "talents."

For a representation that goes into more detail, we will also be informed by each agent's *suitability* to perform a given action. For example, a wizard could *melée*-attack an enemy with her dagger, but this would not be nearly as effective as a wizard casting a fireball, nor as effective as a barbarian *melée*-attacking.

The formulas to calculate these numbers encapsulate expert knowledge of the domain. Relevant statistics include

- Strength and other damage modifiers
- Amount of damage typically healed by healing spell
- Armor class (measures how well protected)
- Hit points (measure of how much health)
- Character class (we cannot escape it entirely!)

These capability representations will be used for two things.

³ *EverQuest* is © ® ™ Sony Computer Entertainment America Inc.

First, to supply a basis for better coordination in combat. We are examining both how the *emergent* tactics can be improved by communication, and how *directed* tactics can be used by having one party leader issue commands. Both of these will require agents to have knowledge of what the other agents can do.

Additionally, we will create a framework that will later have the agents use this knowledge to determine a level of *trust* in one another. The thrust of this is that if a certain agent does not seem to be contributing as much to the fight as it should be capable of, one loses trust in that agent.

Here is the data structure that is used to contain our assessments of each capability:

```
struct sCapabilities
{
    // summon
    int SUMMON_ALLIES;

    // attack
    int MELEE_ATTACK;
    int RANGED_ATTACK;
    int SNEAK_ATTACK;

    // aoe
    int TURN_UNDEAD;
    int AOE_SPELL_DISCRIMINATE;
    int AOE_SPELL_INDISCRIMINATE;

    // heals
    int HEAL_SELF;
    int HEAL_OTHER;
    int HEAL_MANY;
    int OVERALL_HEAL;

    // buff self
    int ENHANCE_SELF;
    int PROTECT_SELF;
    // buff single
    int ENHANCE_OTHER;
    int PROTECT_OTHER;
    // buff many
    int ENHANCE_MANY;
    int PROTECT_MANY;
    int PERSISTENT_AURA;
    int BARD_SONG;

    // defense
    int ARMOR_CLASS;

    // overall tank ability
    int OVERALL_TANK;
};
```

The healing levels are calculated based on the level of the priest that would be required to cast the best spell that the creature has, or the spell that is equivalent to the best potion that the creature has. Most of the capabilities are simply either present or absent—if present, they have the value of the character’s experience level.

The MELEE_ATTACK capability is a little more involved. It is based on the creature’s experience level as well, but weighted by class:

```
result.MELEE_ATTACK =
10 * GetLevelByClass(CLASS_TYPE_FIGHTER) +
10 * GetLevelByClass(CLASS_TYPE_BARBARIAN) +
10 * GetLevelByClass(CLASS_TYPE_PALADIN) +
9 * GetLevelByClass(CLASS_TYPE_MONK) +
9 * GetLevelByClass(CLASS_TYPE_RANGER) +
8 * GetLevelByClass(CLASS_TYPE_CLERIC) +
7 * GetLevelByClass(CLASS_TYPE_BARD) +
6 * GetLevelByClass(CLASS_TYPE_ROGUE) +
4 * GetLevelByClass(CLASS_TYPE_SORCERER) +
4 * GetLevelByClass(CLASS_TYPE_WIZARD) +
4 * GetLevelByClass(CLASS_TYPE_COMMONER);

result.MELEE_ATTACK +=
GetAbilityScore(OBJECT_SELF, ABILITY_STRENGTH);
```

The RANGED_ATTACK capability is handled similarly. Finally, the creature’s overall fitness for tanking is estimated:

```
result.OVERALL_TANK =
    result.MELEE_ATTACK +
    (2 * result.ARMOR_CLASS) +
    (2 * GetCurrentHitPoints(OBJECT_SELF));
```

as a function of the creature’s melée attack ability, ability to avoid damage (armor class), and ability to withstand damage (hit points).

Role Assignment

To test our extensions, we created a very standard sort of party: one dwarven fighter, one elven wizard, one human paladin, one half-elven cleric, and one halfling rogue, all sixth-level. Then we pitted them against a group of two skeleton warriors, one skeleton priest, and one skeleton mage, of the same approximate level as the characters.

With the default scripts, they got slaughtered. Specifically, the fighter charged headlong, which meant he was taking on three skeletons. The paladin’s

idea of backing him up was staying back and firing with his crossbow – which is perfectly fine if you’re by yourself, but which didn’t do the dwarf any good. The rogue had the same idea, throwing shuriken. The cleric tried to heal the dwarf but couldn’t keep him alive; once he was dead the skeletons all attacked the paladin, and once he was gone...

No two fights turn out the same way, due to all the pseudo-random “dice rolls” that control the action. But in none of our trials did the party persevere, always following the same general path to doom.

It seems that we could indeed benefit from a little more coordination. One of the simplest things we can do is to have as many people as possible all concentrating on the same enemy, to reduce the numbers as quickly as possible.

We decided to use our capability structures to assign one of four roles: Main Tank, Main Healer, Melée and Cover. Since all buffing can happen before battle begins, we removed this from the during-combat roles.

Main Tank is as above; the single character who takes most of the damage. This will be our one character who ends up with the highest OVERALL_TANK ability. If everyone keeps track of who the main tank is attacking, they can focus their efforts on defeating that enemy. (This would be done with communication, but it happens that the language allows all agents to access this information directly.)

Main Healer’s job is not to get too distracted. A cleric will already be healing at every opportunity, but will also get involved in combat. Our main healer will try to avoid this. (Everyone with a potion will try to heal themselves when hurt badly enough; everyone with a spell will try to heal a comrade who is hurt badly enough.)

There is only one of each of these, and there may not even be a Main Healer if no creature’s healing ability is high enough. All this role assignment is done at runtime—any of the characters in our party can be asked to assess the capabilities and assign the roles, and will do so.

The remaining party members will be designated either Melée or Cover. Melée will be for the characters who have some melée ability; they will enter the

fighting in support of the tank. They will use their melée weapons by preference, since using ranged weapons does not divide an enemy’s attention as much.

Cover is for the characters whose OVERALL_TANK is below a certain point (currently defined as 0.6 that of the main tank). These characters will try to stay out of the fight, tossing spells and missile weapons from afar. Remember that characters can be multi-classes, so it is not simple enough to say that wizards must always nuke, as we would in *EverQuest*.

If the capability assessment and role assignment method are run on our party, we find that the fighter shall be Main Tank, the Cleric Main healer, the rogue and paladin Melée, and the wizard Cover.

Role Execution

The nice thing about DetermineCombatRound is that it is simple to add new priorities to the front of it, based on the creature’s shiny new SEM_COMBAT_ROLE attribute. If none of those priorities work out, or if the creature has no shiny new attribute, the code will simply fall through to the rest of the DetermineCombatRound method.

We assign each role its new priorities as so:

```
if (role == SEM_ROLE_MAIN_TANK)
{
    // Priorities for main tank:
    // - Heal self if necessary
    // - Melee-attack most damaged enemy
    // (finish him off!)
    ...
}
else if (role == SEM_ROLE_MAIN_HEALER)
{
    // Priorities:
    // - Heal most wounded if below 1/2
    // - Fight my attacker if I am attacked
    // - Fight main tank's target
    ...
}
else if (role == SEM_ROLE_MELEE)
{
    // Priorities:
    // - Heal self if necessary
    // - Heal most wounded if below 1/4
    // - Fight main tank's target, melee only
    // - Fight my attacker if I am attacked
```

```

// - Fight most damaged enemy
...
}
else if (role == SEM_ROLE_COVER)
{
// Priorities:
// - Heal self if necessary
// - Heal most wounded if below 1/4
// - Fight my attacker if I am attacked
// - Fight main tank's target
// - Fight most damaged enemy
...
}
// if no role, fall through

```

The actual actions (omitted for brevity) include using spell attacks and ranged attacks if that's what's appropriate.

Assessment

The first time the fight was started with everything in place—the beforehand preparation, capability assessment and assignment of roles—the party actually won the battle. Only the dwarf and wizard were left alive, but it was still an improvement.

Subsequent tests were less heartening, as the skeletons went back to winning; but the heroes definitely held out longer, fought as more of a team, and came closer to winning.

I am also pleased with the narrative plausibility (a minor point), but the starting conversation goes along these lines:

Elven Wizard: What would you like me to do?
PC: Find out everyone's capabilities and assign them appropriate roles.
Elven Wizard: Half-elven Cleric, you're to be the main healer. Dwarven Fighter, you're to be the lead attacker. Human Paladin, you're to join the attack. Elven Wizard, you're to cover the attackers. Halfling Rogue, you're to join the attack.
PC: Summon allies.
Elven Wizard: I have Summon Creature II and will try to cast it.
Half-Elven Cleric: I have Summon Creature II and will try to cast it.
Dwarven Fighter: I don't have any Summon Creature spells.
Elven Wizard: I have the ability to summon a familiar, so I shall.
Human Paladin: I don't have any Summon Creature spells.

...

The actual information is passed programmatically, but I think it's nice to have conversation showing what's going on as well.

Future work

Obviously, this coordination code should be tested with different compositions of parties, and against different enemies, to see whether it still represents an improvement, or indeed whether it truly does.

The party size is currently hard-coded to five, because the language does not support arrays, let alone collections.

Although roles are assigned beforehand, each agent is still on her own once combat begins. A single leader for a squad AI might be able to assess the situation, pick a tactic and order the rest of the squad to carry it out [van der Sterren 2002b]. In our world, such a leader might notice that the skeleton mage was causing a lot of trouble, and order everyone to redirect their attacks onto that mage.

Also, I did not have time to address the issues of *alignment* and *trust*.

Alignment

In *D&D*, a creature's *alignment* represents how it interacts with others: either in a lawful or a chaotic way, or neutrally in between, and following a life of good or evil, or again neutral. These are orthogonal:

Lawful Good	Neutral Good	Chaotic Good
Lawful Neutral	True Neutral	Chaotic Neutral
Lawful Evil	Neutral Evil	Chaotic Evil

There is much discussion as to the exact meaning of these terms, just as there is in real life [Bush 2002]. Some, like Lawful Evil, are much more puzzling than others, like Chaotic Good.

I contend that these properties of D&D agents might correspond meaningfully to the social properties of traditional networked agents. For example, Evil agents might act in a way that other agent systems would call *selfish* or *deceptive*.

However, the discussion must be left for another time.

Trust

A related issue is that of trust, defined (among other ways) as the degree to which agents can be confident that other agents will do what they claim they will do.

I propose that once we have, as in this project, a model of how agents *should* behave as part of a group, a metric of trust in them can be obtained by comparing that to how they *do* behave.

But I also think that in the course of a single battle, as we have observed here, there would not be enough data to go on. This could become part of a larger testing process.

Conclusions

I most assuredly learned a lot about Neverwinter Nights scripting and module creation, and I feel I did improve the multi-agent coordination some. If the coordination was indeed improved, this shows that a model of pre-combat capability assessment and role assignments based on that capability assessment can tend to improve combat effectiveness.

Citations

[Brockington 2002]: Mark Brockington and Mark Darrah, "Level-of-Detail AI for a Large Role-Playing Game", section 8.5, *AI Game Programming Wisdom*, Steve Rabin, editor, Charles River Media, Inc., 2002.

[Bush 2002]: State of the Union Address, "President" George W. Bush, January 29, 2002.
<http://www.whitehouse.gov/news/releases/2002/01/20020129-11.html>

[Darrah 2002]: Mark Brockington and Mark Darrah, "How *Not* to Implement a Basic Scripting Language", section 10.7, *AI Game Programming Wisdom*, Steve Rabin, editor, Charles River Media, Inc., 2002.

[Lexicon 2003]: Neverwinter Nights Lexicon, NWN Lexicon Group,
<http://www.reapers.org/nwn/reference/>

[van der Sterren 2002a]: William van der Sterren, "Squad Tactics: Team AI and Emergent Maneuvers", section 5.3, *AI Game Programming Wisdom*, Steve Rabin, editor, Charles River Media, Inc., 2002.

[van der Sterren 2002b]: William van der Sterren, "Squad Tactics: Planned Maneuvers", section 5.4, *AI Game Programming Wisdom*, Steve Rabin, editor, Charles River Media, Inc., 2002.