
Steven Matuszek UMBC CMSC 691M Spring 2003

# Background

- *Neverwinter Nights* is a computer role-playing game based on the *Dungeons and Dragons* tabletop role-playing game.
  - *Neverwinter Nights* is ™ © ® BioWare Corp.
  - *Dungeons and Dragons* is ™ © ® Wizards of the Coast, which is owned by Hasbro.
- D&D is very open-ended, but canonically, several adventurers traverse a dungeon, killing monsters and collecting treasure.

# Game AI

- The game AI controls individual agents (monsters, *henchmen*, *familiars*, and *non-player characters*).
- The game AI is a *reactive* architecture, which works very well in this context.
  - If I am attacked, fight back.
  - If my employer says “Hold your ground,” wait in the same place until told to follow.
  - If I *perceive* someone I hate, attack him.
  - If I die, scream and flop onto my back.

# Multi-agent

- These are all implemented in a very sophisticated scripting language.
- There is some coordinated action already built into the AI:
  - Henchmen and familiars will follow orders, and reactively perform obvious tasks (joining the attack, healing comrades).
  - NPCs and monsters can yell for help and can reactively perform obvious tasks (joining the attack, healing comrades).

# Project

- The programming language is general purpose, and the environment very programmatically accessible, so we should be able to implement any type of behavior.

- The purpose of this project, then, is to see what can be done to extend the multi-agent coordination in *NWN*.

# Approach

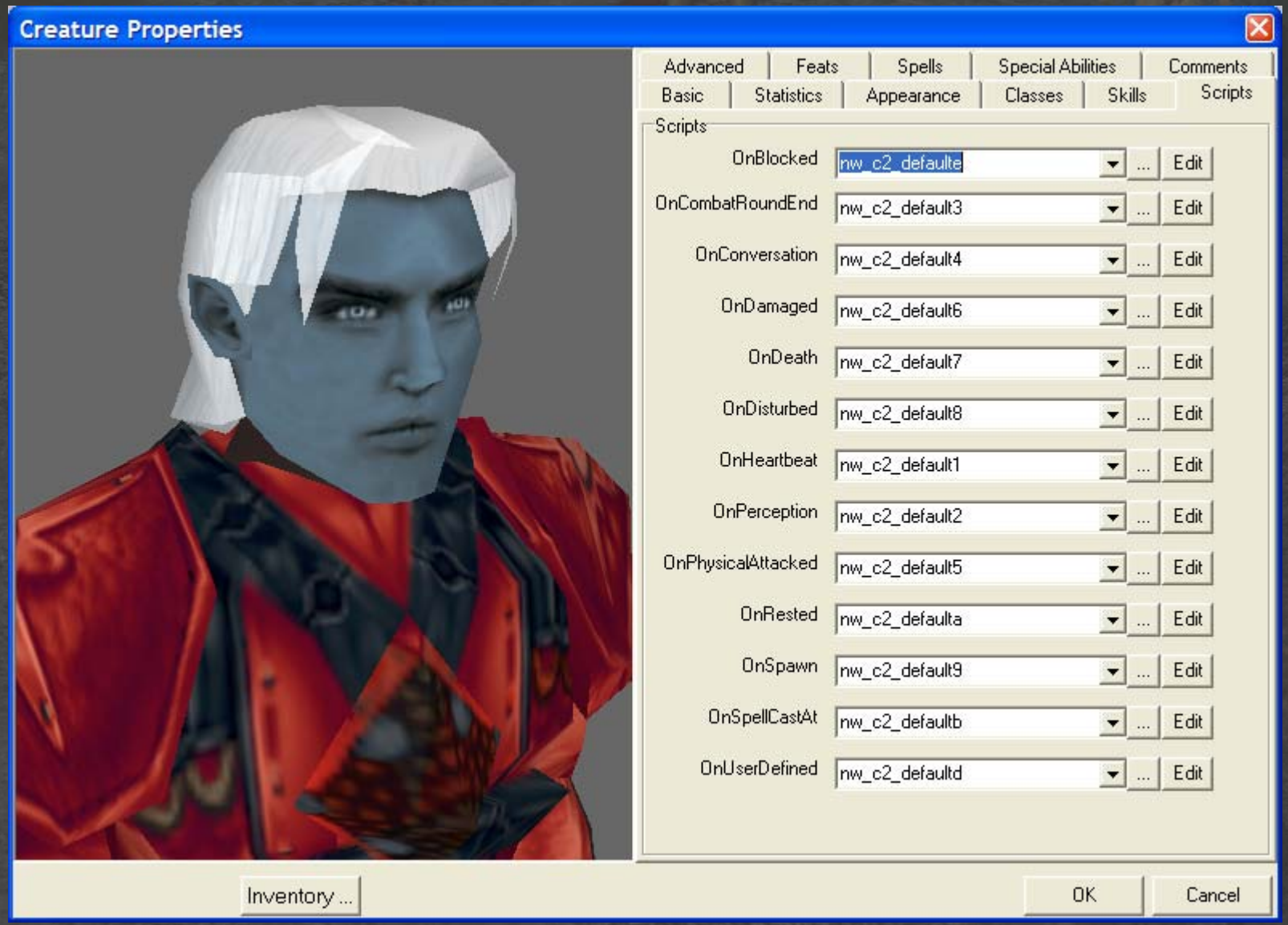
- Learn NWScript
- Analyze and grok the existing AI scripts
- Implement and test* new behaviors:
  - Awareness of allies' capabilities
  - Joint planning
  - Communication
  - Effective joint combat
  - Acting according to *alignment*
- Philosophize about alignment

* (cough) and punt on

# Existing AI

- Every agent can have a script associated with these thirteen *events*:
  - OnBlocked
  - OnCombatRoundEnd
  - OnConversation
  - OnDamaged
  - OnDeath
  - OnUserDefined
    - can generate own events
  - OnPerception
  - OnDisturbed
  - OnPhysicalAttacked
  - OnRested
  - OnSpellCastAt
  - OnSpawn
    - when first created
  - OnHeartbeat
    - fires every six seconds. Useful for deliberating, updating knowledge, anything else that shouldn't happen during combat.

# Assigning event handlers



# Example OnDamaged

```
void main()
{
    if( !GetFleeToExit() )
    {
        if( !GetSpawnInCondition( NW_FLAG_SET_WARNINGS ) )
        {
            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() );
```

# continued

```
        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 );
            }
        }
    }
  }
  if( GetSpawnInCondition( NW_FLAG_DAMAGED_EVENT ) )
  {
    SignalEvent( OBJECT_SELF, EventUserDefined( 1006 ) );
  }
}
```

# Combat methods

- There are ~600 methods that are accessible from within any event handler, and ~100 more defined just in the combat include:

```
GetEnemyHD          CreateSignPostNPC     GetPercentageHPLoss     MatchCombatProtections
TalentFlee          GetCharacterLevel     GetToughestAttacker     EquipAppropriateWeapons
TalentHeal          GetFollowDistance     SetNPCWarningStatus     GetLastGenericSpellCast
GetAlliedHD         TalentHealingSelf     StartProtectionLoop     SetLastGenericSpellCast
GetHasEffect        TalentMeleeAttack     TalentCureCondition     SetSummonHelpIfAttacked
VerifyDisarm        TalentSneakAttack     TalentEnhanceOthers     VerifyCombatMeleeTalent
BashDoorCheck       TalentSpellAttack     TalentMeleeAttacked     GetToughestMeleeAttacker
GetFleeToExit       ActivateFleeToExit    TalentRangedEnemies     CheckFriendlyFireOnTarget
GetIsFighting       GetIsPostOrWalking    UniversalSpellMatch     GetAssociateStartLocation
CheckWayPoints      GetRacialTypeCount    CompareLastSpellCast    GetNumberOfMeleeAttackers
RespondToShout      MatchReflexAttacks    DetermineCombatRound    MatchElementalProtections
TalentBardSong      RemoveAmbientSleep    GetRangedAttackGroup    SetAssociateStartLocation
TalentBuffSelf      ResetHenchmenState    SetListeningPatterns    TalentAdvancedProtectSelf
CheckIsUnlocked     TalentAdvancedBuff    GetMatchCompatibility   TalentPersistentAbilities
GetLockedObject     TalentDragonCombat    GetMostDangerousClass   TalentUseProtectionOnSelf
StartAttackLoop     TalentSeeInvisible    MatchSpellProtections   TalentUseProtectionOthers
DetermineEnemies    TalentSummonAllies    TalentRangedAttackers   CheckEnemyGroupingOnTarget
MatchFortAttacks    DetermineClassToUse   AnalyzeCombatSituation  GetNearestSeenOrHeardEnemy
TalentUseTurning    GetNPCWarningStatus   FindSingleRangedTarget  GetNumberOfRangedAttackers
GetAttackCompatibility                    TalentUseEnhancementOnSelf
```

# More combat code

- So it turns out the professional games programmers know what they're doing.

```
struct sSpellSelect
{
    int RANGED;
    int MELEE;
    object GROUP_TARGET;
    object MOB_TARGET;
    object MELEE_TOUGHEST;
    object TOUGHEST_TARGET;
    int ENEMY_HD;
    int ALLIED_HD;
};
```

- Structs used for assessing the enemy strength and selecting a response

```
struct sEnemies
{
    int FIGHTERS;
    int FIGHTER_LEVELS;
    int CLERICS;
    int CLERIC_LEVELS;
    int MAGES;
    int MAGE_LEVELS;
    int MONSTERS;
    int MONTERS_LEVELS;
    int TOTAL;
    int TOTAL_LEVELS;
};
```

# DetermineCombatRound()

- All the real work happens inside the DetermineCombatRound() method…

```
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 themselves
         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;
}
```

# DetermineCombatRound()

- It’s just a straightforward checklist of things to do in order of preference!
- Now there is a lot of logic hidden inside each of those Talent_____() calls.
  - They will all return FALSE if the action is not appropriate after all, or it fails.
- The blocks for magic-using creatures are rather more complicated.
- Hey, we’re getting somewhere…

# Represent allies' capabilities

- Things that the creature might do in combat have already been categorized.
- Melee attack, heal others, banish undead, sneak attack, use magic item…
- Duplicate the logic without actually doing it, and record the result

# Effective joint combat

- Excellent ideas presented in [van der Sterren 2001]
  - Improve emergent behavior with communication
  - A single leader making tactical decisions
- So in each combat round,
  - Instead of just following checklist,
  - Take allies' communications, requests, observations into account
  - Or follow orders from leader

# Alignment

| | | |
|---|---|---|
| Lawful Good | Neutral Good | Chaotic Good |
| Lawful Neutral | True Neutral | Chaotic Neutral |
| Lawful Evil | Neutral Evil | Chaotic Evil |

- Lots of people have lots of opinions on what exactly these mean, and some are harder to define than others.
- But obviously they have something to do with how you interact with others.
- So let’s have it affect priorities on the checklist.

# Alignment

- Heal others or heal self?
  - *Good*: heal others before healing oneself.
  - *Neutral*: heal yourself first.
  - *Evil*: don’t heal others unless there’s something in it for you.
- Follow orders/fulfill requests?
  - *Lawful*: even if it causes you trouble.
  - *Neutral*: acknowledge; do if possible.
  - *Chaotic*: quite possibly ignore altogether

# Communication

- How can agents send messages?
  - By generating UserDefinedEvents
  - By speaking or shouting
- How can agents react to messages?
  - In the OnUserDefinedEvent handler
  - In the OnConversation handler
  - With the RespondToShout method
- Either way, programmers can define their own constants

# Conclusions

- Code will be finalized 3 hrs. before ship
- Look at these pretty screenshots!