NPC Dialog, NPC Scripting and Progression Script Syntax

Document version: 0.9.1
Status: Draft
Last Modification: 2004.12.10

Created by:
Anders Reggestad

Other authors:
Andrew Craig, Keith Fulton

Table of Contents

History

2003.10.02
Anders Reggestad
First version of the document.
2003.10.06
Anders Reggestad
Added support for Progression Expressions.
2003.10.07
Anders Reggestad
Added power level function.
2003.11.18
Anders Reggestad
Added base option to stat operations.
2003.12.01
Andrew Craig
Added ‘wallet’ to <item> locations for money.
2004.01.04
Anders Reggestad
Added aim option to most operands.
Added sub/neg/div operations.
Added return of values from script to be used in engine.
2004.12.09
Keith Fulton
Updated for MathScript Integration and new ops
2005.05.16
Keith Fulton
Added NPC Dialog and Dialog Scripting to document

NPC Dialog and NPC Scripting

NPC Dialog is an area of potential differentiation between PlaneShift and commercial MMORPGs, where the commercial applications tend to be focused on quick interactions and minimal reading, but PlaneShift can be more roleplay-centric and provide more of a consistent feel to the player. To this end, players should interact to the possible with the NPC in the same manner they interact with other players. A lot of work has gone into the NPC Scripting system to enable this to work and be both fun and natural. The downside of this system is that it is a data-intensive system. If we don't supply the base dialog data, the NPCs will not appear to understand much, which will make the dialog system appear bad and unworkable.

In this section, we will explain conceptually how the system works and follow it up with reference information about each table in the database and each command accessible from the npc scripts.

Triggers and Responses

Triggers are events which cause an NPC to run a particular script. These scripts being run are called responses. Most of the time a response will include text for an NPC to 'say' back to a player, but this is only one kind of response. The trigger includes a set of words the player must say, and links to the appropriate response for the NPC if the player does say those words.

Ex: Trigger "how are you?" Response "I am fine, sir. How are you?"
Trigger "where is the stable?" Response "Third path on the right, after the tavern."

Obviously, there are numerous ways to ask how someone is doing, so triggers are defined with only their most basic required words. When the player types in a full sentence, only these 'known words' are pulled out and searched for in the list of triggers.

Ex: Player says "how are you?" which turns into a search trigger of "how you"
Player says "where is the stable sir, please" which turns into "where stable"
Player says "where can I find the stable" which turns into "where stable"

The *npc_disallowed_words* table is a list of words which should be skipped over when building the known words list from the triggers. This feature makes sure that "is" and "the" aren't added to the known word dictionary when "where is the stable" is submitted as a trigger, but allows the trigger to stay readable. Any word appearing in the *npc_disallowed_words* table will be removed from triggers in the database before adding the remaining words to the dictionary.

Synonyms

The *npc_synonyms* table allows the designer to automatically substitute one term for another in a player's sentence before searching for the trigger. Synonyms can also be used to substitute multiple words and decode slang or idioms.

Ex: npc_synonym “Barn” = “Stable”

Trigger “Where is the barn” becomes “Where is the stable” which becomes “where stable” which we defined in the previous example.

Ex 2: npc_synonym “Good afternoon” = “Hello”
“Tavern of Love” = “Stable”

Generalizations

The *npc_synonyms* table is also used for generalizations. Imagine players saying all these things:

“Give me an apple”
“Give me a pear”
“Give me an arrow”
“Give me a sword”
“Give me a banjo”

It is not good to force a Settings person to write an individualized response for each of these, but if we do not allow generalizations, we either write individualized responses or we live with the NPC saying “I don’t understand you” or somesuch because he doesn’t recognize the trigger.

In the *npc_synonyms* table, you can set up any hierarchy of terms you wish, and then have the NPC respond to a higher level in the hierarchy.

Ex: npc_synonym “Apple” = “Fruit”
“Pear” = “Fruit”
“Arrow” = “Weapon”
“Sword” = “Weapon”
“Banjo” = “Musical instrument”
“Fruit” = “Thing”
“Weapon” = “Thing”
“Musical Instrument” = “Thing”

Player says: “Give me an apple” which becomes “give apple” due to known terms.

NPC system searches for “give apple”, “give fruit” and “give thing”.

On “give thing”, trigger is found which causes the response “I’m not giving you anything!”

Thus with one trigger/response on “give thing”, we can make the NPC appear much more understanding of the intent of the player with his requests, without enumerating huge lists of trigger permutations.

Trigger Groups

The next issue is that there are many legitimate ways to ask the same basic question. Having our scripting people remember to include all these at every stage is cumbersome, error prone and inefficient. For example, to determine the location of the stable, a player could legitimately ask:

- Where is the stable? (where stable)
- Where can I find the stable? (where find stable)
- Is the stable near by? (stable near)
- Is the stable far from here? (stable far)

What we need is a way to equate "where find stable" to the most basic version, "where stable". This is what *npc_trigger_groups* allow. You equate "where find stable", "stable near" and "stable far" to "where stable" as a trigger group. Then wherever you want an NPC to be able to answer this question, you use only "where stable" and you get the other triggers for free. This feature is particularly important with quest meta-scripts.

Knowledge Areas

Everyone has their areas of expertise, due to their life experience and training. Anyone has a good knowledge of their own life, of their hometown, their relatives, their country, occupation, school and so on. We simulate these areas with *npc_knowledge_areas* in the database. Triggers are not assigned directly to NPCs, but in fact assigned to knowledge areas. Each NPC has zero or more knowledge areas to look in to see if he understands your question. Any knowledge area can be linked to, and thus shared, by multiple NPCs. Sharing the basic Knowledge Areas saves a lot of time for the dialog writers, keeping them from wasting time with copying triggers and responses over and over between NPCs.

Here is an example of things a dwarf blacksmith in Hydlaa might know:

- Things about life in general
- Things about Hydlaa
- Things about Dwarfs
- Things about blacksmithing
- Things about himself that only he knows (such as his quests)

If you imagine that each of those bullets is an "area of knowledge" or a "knowledge area" in our parlance, you could see how each one could have a number of triggers and responses within it. Having a knowledge area (KA) assigned to him gives an NPC a whole new category of things he can understand when people say them and a whole new set of responses to those things.

It is recommended that the "General" KA be assigned to all NPCs and for it to be fairly extensive. The more triggers are understoon by the npcs, the less frustrating it will be for players to talk to npcs. Personalizing every NPC to have his own response to "Hello" might be beneficial to the game, but it probably unnoticed by most players. Thus the costs of excessive personalization should be weighed against the benefits of having fewer, broader and more widely shared KA's.

Summary

When a player says something, the following steps happen:

1. The words he says are scanned for known synonyms, and any synonyms are replaced.

2. The known words are all pulled out of the sentence, forming the base trigger string.

3. This trigger string is searched to see if it is part of a trigger group, and if so, the primary trigger of the group is substituted.

4. This trigger is searched for within all the Knowledge Areas in the order specified for that NPC, first with the prior response link and then without if not already found.

5. If the response is found, run the associated response script. (This response script usually is nothing more than “tell the player the response” but can do much more. See next section for a full explanation of response scripting.)

6. If the response is not found, find the next available generalization of the trigger phrase and go back to step 3.

7. If all generalizations are searched for and not found, then the npc will return an error, the text said by the player will be logged to the npc_bad_text table, so Settings people can analyze how people are attempting to phrase things and improve the database for the future.

NPC Response Scripts

Introduction

NPC's can be triggered by several things:

- Things said by players
- Items given to them by players
- Players simply approaching near them (coming soon)

Whenever a recognized event trigger occurs, the trigger is linked to a certain response. This response is a mini-script which tells PS what the npc should do. Most of the time, the response is merely an instruction to say something to the player, but it can be much more than that. These scripts allow you to have an NPC...

- Give a player an item or money
- Offer the player a choice of items
- Perform animations
- Award experience points or affect stats by running a Progression Script (see next section).
- Assign and Complete Quests

This section will document and explain all the various commands at your disposal in npc response scripts.

respond

Makes the npc send the player one of the 5 responses specified at random. If less than the maximum of 5 response texts are specified, it chooses randomly between the ones that are present.

Syntax:

<respond />

Attributes:

None.

Example:

```
<response>
  <respond />
</response>
```

This operation executes and makes the npc send a /tell to the player with something he is saying.

action

Makes the npc perform a certain animation. All clients in proximity to the npc can see the animation perform.

Syntax:

<action anim="name" />

Attributes:

anim This is the name of the animation from the .cal3d file. If the animation does not exist for that particular race, the command is ignored.

Example:

```
<response>
  <respond />
  <action anim="greet"  />
</response>
```

This script makes the npc say something back to the player who triggered the response, then makes the npc appear to wave.

give

Makes the npc give the player a specified item.

Syntax:

<give item="id|item_name"/>

Attributes:

item This attribute specifies either the id number or the official name of the item to give to the player. Names must match exactly.

Example:

```
<response>
  <respond />
  <give item="Short Bronze Sword" />
</response>
```

This script has the npc say something to the player, then give him a Short Bronze Sword. The player will see a message that he has been given the item and he will see it when he opens his inventory.

offer

Makes the npc offer the player a choice of items to pick from. This allows players to receive rewards which are more tailored for their needs, depending on their class and what they already own, which makes quests more flexible and useful.

Syntax:

```
<offer>
  <item id= “id” />
  <item name=“name”  />
</offer
```

Attributes:

The <offer> tag has no attributes, but instead specifies a variable length list of <item> tags, which can specify either the id or the name of the item(s) to offer.

Example:

```
<response>
  <respond />
  <offer>
    <item id= “Jeweled Dagger” />
    <item name=“Golden Shield”  />
  </offer
</response>
```

This script makes the npc say something in response to the player, then offer the player a choice of whether to accept the Jeweled Dagger or the Golden Shield. The player will get a popup window in their GUI with names, icons and descriptions of these with the chance to pick one for himself/herself.

money

Makes the npc give the player the specified amount of money. The player will also get a system message informing him of the gift.

Syntax:

<money value="circles,octas,hexas,tria" />

Attributes:

value Specifies the amount of money to give to the player. Money in PS is always formatted as a csv string with each amount of each currency in a field.

Example:

```
<response>
  <respond />
  <money value="0,0,0,5" />
</response>
```

This script makes the npc respond to the player talking to him, then gives the player 5 tria. The player also receives a system message informing him of what he has received.

guild_award

Makes the npc award guild karma points to the guild of the player who triggered the script.

Syntax:

<guild_award karma="amount" />

Attributes:

karma This attribute specifies how many points to award to the guild of the player who triggered this response.

Example:

```
<response>
  <respond />
  <guild_award karma="50" />
</response>
```

This script makes the npc respond to the player verbally, then awards the guild of that player 50 additional guild points.

train

Makes the npc offer training in the named skill. The player will receive a confirmation popup box and must confirm to actually receive the training. The player will be charged according to the standard formula for training costs.

Syntax:

<train skill="skillname" />

Attributes:

skill The name of the skill the npc should offer to increase. Must be an exact match to the database name for the skill.

Example:

```
<response>
  <respond />
 <train skill="Swords" />
</response>
```

This script makes the npc say a response back to the player, then calculates the monetary cost of increasing the player's skill in Swords and offers the player the chance to pay that money to get his skill increased. If the player accepts, the Swords skill is incremented and the monetary cost is deducted from the player's inventory.

run

Invokes a named progression script for the player. Progression scripts can change or affect almost any stat or ability of a player, permanently or temporarily. (See the next section of this document for all reference information on progression commands.) NPC scripts should use this command when they want to buff a player, heal him, hurt him, etc.

Syntax:

<run scr="scriptname" [param0="#"] [param1="#"] [param2="#"] />

Default values are bold.

Attributes:

scr	The name of the progression script to run. (Required.)
param0	An optional numeric value which can be referenced in the progression script as "param0".
param1	An optional numeric value which can be referenced in the progression script as "param1".
param2	An optional numeric value which can be referenced in the progression script as "param2".

If an optional parameter is not specified and the progression script attempts to use it, the progression script will use the value of 0.

Example:

```
<response>
  <respond />
  <run scr="Award Exp" param0="500" />
</response>
```

This script makes the npc respond to the player verbally, then runs a script to give the player more experience points. It assumes the progression script refers to the variable 'param0'. If the progression script does not refer to this variable, the npc script has no ability to affect the outcome of the progression script.

assign

Assigns a specified quest to the player.

Syntax:

<assign q1="questname1" q2="questname2" q3="questname3" q4="questname4"
q5="questname5" timeout_msg="timeout message" />

Attributes:

q1-q5 Names of possible quests to be assigned in this step. It is not required to specify 5 quests every time with this operation. If you only have 1 quest to assign, just use q1 and leave the others out. The quest will be randomly assigned with equal probability to as many q# choices as have been specified.

timeout_msg This message will be displayed to the player on failure when he runs out of time to complete the quest. *This feature is not implemented yet.*

Example:

<response>
<respond />
<assign q1="Fetch Sandwich" q2="Kill Dragon" />
</response>

This script allows the npc to respond verbally to the player, then assigns a quest to the player with a 50/50 chance of whether it is to get a sandwich or kill a dragon.

complete

Marks the named quest as completed on the server. If the quest is not marked to be saved for the player's permanent list, the quest assignment is deleted from the player's Quest Notebook.

Syntax:

<complete quest_id="questname" />

Attributes:

quest_id The name of the quest to be completed.

Example:

```
<response>
  <respond />
  <complete quest_id="Fetch Sandwich">
  <money value= "0,0,0,5" />
</response>
```

This script allows the npc to respond the player by saying something, marks the 'Fetch Sandwich' quest complete, and pays the player 5 tria for his trouble.

complete

Marks the named quest as completed on the server. If the quest is not marked to be saved for the player's permanent list, the quest assignment is deleted from the player's Quest Notebook.

Syntax:

<complete quest_id="questname" />

Attributes:

quest_id The name of the quest to be completed.

Example:

```
<response>
  <respond />
  <complete quest_id="Fetch Sandwich">
  <money value= "0,0,0,5" />
</response>
```

This script allows the npc to respond the player by saying something, marks the 'Fetch Sandwich' quest complete, and pays the player 5 tria for his trouble.

verifyquestassigned

This operation ensures that the named quest is actually assigned to the player in question. If assigned, then the rest of the script proceeds normally. If not assigned, the npc says the error_msg back to the player and the script is cancelled. Normally, this operation should be the first operation in a script if the script is for a quest.

Syntax:

<verifyquestassigned quest=”questname” error_msg=“Error msg goes here”/>

Attributes:

quest The name of the quest to check for assignment to this player.

error_msg The text of what to say back to the player if this quest is not assigned to the player in question.

Example:

```
<response>
  <verifyquestassigned quest=”Fetch Sandwich” error_msg= “Crazy people these days...handing
        out random sandwiches.”/>
  <respond />
  <complete quest_id=“Fetch Sandwich”>
  <money value= “0,0,0,5” />
</response>
```

This script allows the npc to respond the player by checking to make sure he really has the Sandwich Fetching quest. If he does, he is rewarded after giving it to the NPC by the NPC saying something, marking the ‘Fetch Sandwich’ quest complete, and paying the player 5 tria for his trouble. If he does not have the quest assigned, the NPC says “Crazy people these days...handing out random sandwiches,” and exits the script immediately.

Note: Almost always you will want verifyquestassigned as the first op of a script, so it has the effect of blocking the script from executing if the quest is not assigned.

verifyquestcompleted

This operation ensures that the named quest has actually been completed. Normally, this operation should be the first operation in a script if the script is for a quest.

NB: Remember that not all quests are stored after completion. Only quests which are marked to be saved, are saved. These are the only ones which will be remembered and useful for this script operation.

Syntax:

<verifyquestcompleted quest="questname" error_msg="Error msg goes here"/>

Attributes:

quest The name of the quest to check for quest completion by this player.

error_msg The text of what to say back to the player if this quest has not been completed by the player in question.

Example:

```
<response>
  <verifyquestcompleted quest="Fetch Sandwich" error_msg= "You haven't even delivered a
          sandwich yet. What makes you think you can deliver the gold?"/>
  <respond />
  <assign quest_id="Fetch Gold">
</response>
```

This script allows the npc to respond the player by checking to make sure he really has completed the Sandwich Fetching quest. If he has, he is considered qualified to handle the next quest and is assigned the Gold Fetching quest. If he has not completed the Sandwich Fetching quest, the NPC says "You haven't even delivered a sandwich yet. What makes you think you can deliver the gold?" and exits the script immediately.

Note: Almost always you will want verifyquestcompleted as the first op of a script, so it has the effect of blocking the script from executing if the quest is not assigned.

Progression Scripts

A progression script is either stored in the DB or executed by entering a script right into the progression manager. If possible a progression event should be created in the db because these events are preprocessed and than the xml script will not have to be parsed.

Every progression event script starts and ends in <evt>(script goes here)</evt>. In between these tags go the command tags. You can have as many as you wish, but you must have at least one.

Examples:

1. The effect on the target from a heal spell where target is healed with 5 HP.

```
<evt>
        <hp aim=”target” value=”5” />
</evt>”
```

2. Pass in a parameter indicating the damage amount, because the damage is calculated outside the script in the combat code. Note the variable called “Param0” which is used in the HP adjustment and the system messages sent to the target and the actor.

```
<evt>
        <hp aim="target" value="-Param0" save="Result" />
        <msg aim="target" text="You lost $Param0 hitpoints"/>
        <msg aim="actor" text="Target hit for $Param0 hitpoints"/>
</evt>
```

3. This is a spell healing script to add to HP again, but this time the amount to adjust isn’t passed as a parameter, but is calculated in the progression script here. Actor:PowerLevel is a named property of the spell caster. Actor and Target are two variables which are always set to the player causing the script, and the player affected by the script. Actor and Target each have a large number of properties like PowerLevel which are accessible in this fashion at runtime.

```
<evt>
        <msg text="You feel fresh"/>
        <hp aim="target" value="2*Actor:PowerLevel+3" />
</evt>
```

4. This script calculates the damage amount in the script, and saves that amount in a variable called “Result”, which is then used with a dollar sign in the text of the message sent to the spell caster.

```
<evt>
        <hp aim="target" value="-15+rnd(5)" save=”Result” />
        <msg text="Your spell did $Result damage."/>
</evt>
```

Progression Operations

AGI (StatOp)

Adjust the agility stat of a character.

Syntax:

<agi [aim=”**actor**|target”] [adjust=”**adjust**|set|mul”] [base=”yes|**no**”] value=”mathscript” [delay=”mathscript”] [undomsg=”text goes here”] />

Default values are bold.

Attributes:

aim	If set to ‘actor’, the stat to be adjusted is on the main actor. If it is set to “target”, then the agility of the target is what is adjusted.
adjust	If set also to “adjust”, then value added relative to the current stat. If it is set to “mul”, the current value of the stat is multiplied by the op value (for percentage adjustments). Otherwise it is directly overriding the current value with the script value.
base	If set to “yes”, then the script will modify the base value of the Agility stat instead of modified values.
value	Any valid one-line mathscript can be specified here, including any predefined or user defined variables. This is the value which the operation uses in the adjustment calculations.
delay	Another mathscript to calculate the duration of this adjustment. If any delay is specified, the effect of this operation will be automatically undone after ‘delay’ seconds.
undomsg	Specify a message to be sent as a system message to the affected player when the effect is undone after ‘delay’ seconds.

Example:

<agi aim=”target” value=”-5” delay=”60” undomsg=”You feel more agile again.” />

This operation decreases the modified value of the target’s agility by 5 points for 1 minute, then announces to the target player that he feels more agile again to let him know the Undo has worked.

NOTE: The following script operations have the same syntax and capabilities as <agi> here.

<cha>	Charisma
<int>	Intelligence
<str>	Strength
<wil>	Willpower
<sta>	Stamina
<con>	Constitution
<end>	Endurance
<fatigue>	Fatigue
<hp>	Hit Points
<mana>	Mana (Magic Capacity)
<hprate>	Hit Point Regeneration Rate
<attack>	Attack Value Modifier
<defense>	Defense Value Modifier

Faction (FactionOp)

Adjust the target's faction rating relative to a named faction.

Syntax:

<faction [aim="**actor**|target"] name="name" value="mathscript" />

Default values are bold.

Attributes:

aim Are we adjusting the faction score of the actor or the target?

name Every faction in PS is specified by name. The name attribute must be a legal faction on the list for this script operation to work.

value Any valid one line mathscript used to calculate the amount of the adjustment.

Example:

<faction value="-1" name="Orcs" />

This operation subtracts 1 from the current faction standing of the actor with the Orcs faction. (This number essentially is a measure of how much Orcs like or dislike the actor.)

Exp (ExperienceOp)

This operation adds to the experience points of one or more players.

Syntax:

<exp [type="allocate_dmg"] value="mathscript" />

Attributes:

value Any valid one line mathscript used to calculate the amount of the adjustment.

type If this attribute is set to 'allocate_dmg', the operation uses the damage history on the target to allocate the exp value proportionally across all the people listed in the damage history, by the weighted average of each one's damage.

Example:

<faction value="10" type="allocate_dmg" />

This operation awards 10 W points to the actor(s) who dealt damage to the target monster. If multiple players are on the damage history list of who hurt the creature, this 10 points will be spread across them all, proportionately to how much damage each one did. *Side Note:* Rather than use the entire damage history of the monster, it only goes back to the first time gap in damage of more than 10 seconds. This enforces that only members of the same group that killed the monster get credit for the kill.

Item(ItemOp)

Create an item and either give it to the player or place it on the ground for anyone to pick up.

Syntax:

<item name="item name" location="inventory | wallet | ground" [aim="**actor**|target"]
[count="stack_count"] />

Default values are bold.

Attributes:

name Specifies the exact name of the item to create. If the "wallet" location is used, only names of trias, hexas, octas or circles can be used.

location Specifies where to put the item(s) created. If "inventory" is specified, the script will attempt to put the item in the actor's inventory. If his inventory is full, or if "ground" is specified, the item will appear on the ground at the actor's feet. If "wallet" is specified, only money objects can be given.

count The number of items in the stack given to the player. If not specified, a count of 1 is used.

aim Normally, items are given to the actor, but target is also specifiable here.

Example:

<item name="Battle Axe" location="inventory" />

This operation gives the actor a new Battle Axe and he will see it when he opens his inventory window seconds or hours later.

Msg (MsgOp)

Send a system message to a player.

Syntax:

<msg [aim="**actor**|target"] text="text" />

Default values are bold.

Attributes:

aim The message is sent to the actor if this is specified as 'actor' or otherwise it goes to the target.

text The string of text to send to the actual player. This text can have mathscript variables from other parts of the progression script contained in it, each prepended with a $ sign.

Example:

<msg text="You just got $Exp experience points." />

This command sends a message to the actor telling him about his experience point. This example assumes that a variable called 'Exp' was created and saved by a preceding operation.

Block (BlockOp)

Mark a named category of scripts as blocked for a particular character. This is used to keep buff spells from being cast repetitively on a player for multiple effects.

Syntax:

<block [operation="**add** | remove"] category="category name" delay="mathscript" />

Default values are bold.

Attributes:

operation 'Add' means that this category should be added to the blocked list. 'Remove' means that it should be removed. The 'Remove' version is really for Undo scripts which are generated automatically. Scripters should only need 'add'.

category The name of the category to block. PS does not validate these names, but just keeps a list of the active blocks. Thus it is up to the scripter to ensure that the blocked names are the names of the categories used by the spells, or those spells will never get blocked.

delay Calculates the length of time for this block. Normally this will be equivalent to the delay times for other StatOps, etc. When the time expires, the block for the named category is removed again.

Example:

<block category="+STR Buff" delay="60" />

This command blocks any spell in the category of "+STR Buff" from being successfully cast on this player until 60 seconds have expired.

Purify(PurifyOp)

This command is used to purify a glyph in the actor's inventory. Mainly used inside a persistent script to be able to continue the purifying process after a player has been offline and reconnects. Scripters should not need this operation in their own scripts.

Syntax:

<purify glyph="glyphUID" />

glyphID is the item_instances.id for that glyph in inventory.

Script(ScriptOp)

Start a sequence of events after a delay in milliseconds. If persistent and target is a character and the client disconnect when waiting for the script to execute it is saved and restarted when the character reconnect to the world. This can be used to make scripts with delayed reactions.

Syntax:

<script delay=”mathscript” [persistent=”yes|**no**”]><op1/>…<opN/></script>

Default values are bold.

Where op1 to opN is any number of other script operations, including other scripts.

Attributes:

persistent If yes, then this script will be saved with the current timer value if the player logs out while the timer is counting down. Thus, when the player logs back in, his script timer picks up where it left off instead of just being gone.

delay Any single line mathscript used to define the time, in milliseconds, until the sub script is executed.

Example:

Remove 5 from target HP after 50 seconds even if disconnecting.
<script delay=”50000” persistent=”yes”><hp value=”-5” /></hp></script>

Skill (SkillOp)

Adjust target's skill.

Syntax:

<skill name="skill name" [aim="**actor**|target"] [buffer="yes | no"] [adjust="**adjust**|set"] value="mathscript" />

Default values are bold.

Attributes:

name Text name of the skill to affect. Must be a legal skill name from the database.

aim Determines whether this operation will act on the actor or the target.

buffer Determines whether this skill modification is a buffer or not.

adjust If 'set' is set here, then the value from the mathscript will override the existing skill value. If 'adjust' is set here, this value will add on to the skill value.

Example:

<skill name="Swords" value="1" />

This adds 1 point to the player's skill in Swords.

ShowDetails (ShowDetailsOp)

Looks like it is intended to bring up a window about a player's details, but it is not implemented yet.

Syntax:

<showdetails />

Attributes:

None.

Example:

<showdetails />

AttachScript (AttachScriptOp)

Attaches a progression script to certain events in the game, such as whenever you get hit or whenever you inflict damage on someone else.

Syntax:

<attachscript scriptName=”scriptname” [aim=”**actor**|target”] [event=”attack | **damage**”] [delay=”mathscript”] [undomsg=”text msg here”] />

Default values are bold.

Attributes:

scriptName	Text name of the script to run when the event occurs to the target.
aim	Determines whether this operation will act on the actor or the target.
event	Determines whether the named script will fire on the attack event or the damage event.
delay	Any legal one-line mathscript used to calculate how long this script should stay attached to the event, in seconds.
undomsg	Text to send to the player as a system message when the script is automatically detached again, after ‘delay’ seconds.

Example:

<attachscript scriptName=”ExtraDmg” aim=”target” event=”damage” delay=”300” undomsg=”The shirt of Nettles disappears.” />

This attaches a script called “ExtraDmg” (which might deduct an extra 10 HP from someone) to run every time they are hit in combat. This script will stay attached for 5 minutes (300 seconds) and when it expires, it will unattach and send a system message to the player target to let him know.

DetachScript (DetachScriptOp)

Detaches a progression script from certain events in the game, such as whenever you get hit or whenever you inflict damage on someone else. These ops are mostly auto-generated by the system for undoing AttachScript ops. Scripters will probably not need to use these.

Syntax:

<detachscript scriptID=”scriptid” [aim=”**actor**|target”] [event=”attack | **damage**”] />

Default values are bold.

Attributes:

scriptID	ID of the script to search for and detach on the target.
aim	Determines whether this operation will act on the actor or the target.
event	Determines whether the named script will fire on the attack event or the damage event.

Example:

<detachscript scriptID=”1234” aim=”target” event=”damage” />

This finds the script with ID 1234 on the target character tied to his damage events, and removes it from the list.

IdentifyMagic (IdentifyMagicOp)

Can be used by a player to determine if an object is magical or not.

Syntax:

<identifymagic />

Attributes:

None.

Example:

<identifymagic />

This inspects the target object for magic scripts and tells the actor whether it found any magic on the object or not..

Predefined Variables for Progression-based Mathscripts

Standard Parameters

Every progression script can use parameters to customize the effect of the script to the game surroundings. It wouldn't be very nice if every Damage deducted from you was 10 points. Parameters and variables are the way that the game passes in data values to the scripts.

Param0, Param1, Param2…	These are generic and normally used for passing values from npc scripts into progression scripts.

Spells have their own variables. It is the responsibility of the script writer to know that if he uses spell variables, he must make sure the progression script is only called from a spell invocation.

Range	How far is it from the spell caster to the target?
Duration	How long does the effect of the spell last?
PowerLevel	What additional PowerLevel is being spent by the caster on this cast?
EffectRange	For Area of Effect spells, what is the radius of the effect?
EffectAngle	For Area of Effect spells, what is the arc range of the area?
EffectTypes	(Not used currently.)
ProgressionDelay	Time in seconds to wait before the spell progression event is actually fired. This is used for allowing the visual effect to be played before the effect is done.