KORAX' HERITAGE
cross-breeding a 3D first person shooter with a role playing game in the world of Hexen

KRPG Conversation Script

Root-level statements

import

Imports one or more script lumps and registers their contents. Basicaly that means that these script lumps would be merged with current one and parsed altogether.

Syntax:

```
import lump_name

 import
 {
 lump_name
 lump_name2
 ...
 }
```

conitem

Begins a person-specific conversation block, which contains conversation data.

Syntax:

```
conitem [person_name] person_id [, con_name con_id]
 {
 ...
 }
```

*Person_name*is an optional convenient name of the person entry, under this the character can be adressed in conversation scripts. *Person_id* is the corresponding person's numeric id to use in DECORATE and map scripts. *Con_name*is an optional name under which this conversation item can be adressed in conversation scripts. *Con_id* is an optional id of this conversation; it allows to specify numerous "conitems" attached to one person. If there are more than one "conitem" with same *person_id*, conversation with lesser *con_id* is counted as active, unless this is explicitly changed from either conversation script or map script.

Conversation block root statements

bye

Begins a speech block that is run automatically when conversation ends.

Syntax:

```
bye
 [if (condition)]
 {
 ...
 }
```

Condition part is optional; it makes the 'bye' block active or inactive depending on check result. First active 'bye' found is played during the conversation.

choice

Begins a speech block that could be selected and "spoken" by player.

Syntax:

```
choice <choice_name | none> choice_text [hidden]
 [if (condition)]
 {
 ...
 }
```

Choice_name is an optional name under which this choice block can be adressed in conversation script. *Choice_text* is a common text that represent this choice block in game during the conversation. ***Hidden*** flag makes choice unaccessible all the time regardless of block condition, unless it is ordered to be shown explicitly by ***choices*** command (see speech block statements for more info).
Condition part is optional; it makes the 'choice' block active or inactive depending on check result. Only active choices are shown for player as speech options during the conversation.

flat

Specifies flat texture drawn on conversation screen background.

Syntax:

```
flat [if (condition)] flat_name;
```

Flat_name is a name (case insensitive) of an flat patch lump in WAD or file in PK3 (without extension).
Condition part is optional; it makes the picture active or inactive depending on check result. First active flat found is used during the conversation.

hello

Begins a speech block that is run automatically when conversation commences.

Syntax:

```
hello
```

```
[if (condition)]
{
...
}
```

Condition part is optional; it makes the ‘hello’ block active or inactive depending on check result. First active ‘hello’ found is played during the conversation.

music (not yet supported)

Specifies theme music to play during conversation.

Syntax:

```
music [if (condition)] music_name;
```

Condition part is optional; it makes the name active or inactive depending on check result. First active music found is played during the conversation. *Music_name* is is a name of a music entry in SNDINFO or file in PK3 (without extension) to be played.

name

Specifies current person’s name, which would be shown in game during the conversation.

Syntax:

```
name [if (condition)] text_string;
```

Condition part is optional; it makes the name active or inactive depending on check result. First active name found is shown during the conversation.

pic

Specifies current person’s picture, which would be shown in game during the conversation.

Syntax:

```
pic [if (condition)] pic_name;
```

Pic_name is a name (case insensitive) of an image lump in WAD or file in PK3 (without extension).
Condition part is optional; it makes the picture active or inactive depending on check result. First active picture found is used during the conversation.

speech

Begins a custom speech block that could be run by script command from any other block.

Syntax:

```
speech speech_name
```

```
[if (condition)]
{
...
}
```

Speech_name is a name under which this speech block can be adressed in conversation script.

Condition part is optional; it makes the 'speech' block active or inactive depending on check result. First active 'speech' of defined name found is played when corresponding command is given (see speech block statements for more info).

var

Registers a local person's variable, that could store some value and be used in conditions all around the conversation script. They can be accessed in map scripts as well to get their value or set a new one; this makes them very useful in designing Quests.

Syntax:

```
var var_name [var_id] <boolean | integer> initial_value;
```

Var_name is a name of variable it could be accessed by in conversation script. *Var_id* is an optional numeric id that could be used to access this variable in map script. ***Boolean*** and ***integer*** are two available types for variable; variable can have only one type. *Initial_value* initializes newly registered variable.

General conditions syntax

if / else (limited support)

If statement orders script to make a check of some condition. Depending on condition check result script may execute differently afterwards. There are two general types of conditions: header condition and normal condition. Header conditions are controlling availability of a speech block, person name and pic properties. Normal conditions control speech blocks execution by implementing 'forks' in them.

Unlike ***if***, ***else*** could be used only inside speech blocks; and it should always be in pair with ***if***. In speech blocks ***if*** statement precedes commands that are executed in case condition check resulted in TRUE, ***else*** precedes commands that are executed in the opposite case.

Syntax:

```
if (condition)

if (condition) <single command here>;
 [else <single command here>;]

if (condition)
 {
 ...
 }
 [else
 {
 ...
 }]
```

First syntax style is used for header conditions.

Second and third syntax styles can be used only inside speech blocks.

Condition should be enclosed in round brackets and may consist of logical functions and expressions.

and

And logical function results in TRUE only if both defined conditions result in TRUE.

Syntax:

(first condition) and (second condition)

is

Is logical function results in TRUE only if left part of the expression has same result as the right part.

Syntax:

<left expression> is <right expression>

is not

Is not logical function results in TRUE only if left part of the expression does not have same result as the right part.

Syntax:

<left expression> is not <right expression>

not

Not logical function results in TRUE only if following condition results in FALSE.

Syntax:

not (condition)

or

Or logical function results in TRUE if at least one of the defined conditions results in TRUE.

Syntax:

(first condition) or (second condition)

Speech block statements and keywords

choices

Explicitly sets a number of speech options that should be given to player next time and terminates execution of current speech; those choice blocks that fail condtion check are ignored though. Meanwhile all other choice blocks available are considered unaccessible regardless of their conditions. These rules work until player makes a selection, after that conversation continues normally.

Syntax:

```
choices [strict]
 {
 choice_name
 choice_name2
 ...
 }
```

Strict flag makes choices be listed strictly in the order of their definition in ***choices*** block. Otherwise an order of 'choice' blocks appearance in the conversation script is used.
Only 'choice' blocks can be selected this way, other block types will be ignored.

cls

Orders to clean conversation screen from previous cues (sayings). This action has only user-side effect.

Syntax:

```
cls;
```

comment

Displays a commentary text. If player participates in conversation this command will also suspend futher script execution until user order to continue.

Syntax:

```
comment commentary_text;
```

end

Executes available 'bye' speech block and terminates conversation afterwards.

Syntax:

```
end;
```

everybody (not yet supported)

A keyword that allows to adress every person participating in conversation at once. It can be used in the cases where some command should adress a person and to execute their common member functions.

executems

Executes map script.

Syntax:

executems map_index script_index[(arg1[, arg2[, arg3]])];

All parameters are similar to ACS_Execute action special.

Game

Reference to Game object that allows to execute its member functions. See object references and Appendix A.

jump

Breaks execution of current speech block and proceeds to another one. If there are no blocks of given name accessable at the moment, conversation terminates. If a ***none*** keyword is given at a place of a speech name, jump does nothing.

Syntax:

jump speech_name;

jump (condition) speech_name1 speech_name2;

Second syntax executes *speech_name1* speech if condition check results in TRUE and *speech_name2* speech in opposite case.

initiator

Initiator keyword is a reference to person who initiated the conversation (usually – player). It can be used in the cases where some command should adress such person and to execute his/her member functions.

Level

Reference to Level object that allows to execute its member functions. See object references and Appendix C.

myself

Myself keyword is a reference to person whose conversation script is currently executed. It can be used in the cases where some command should adress such person and to execute his/her member functions. ***Myself*** is equal to ***initiator*** if this person him/her-self commenced the conversation.

none

Null-reference keyword. It can be used in cases when some command must adress a speech block or a person, to tell that no speech/person should be adressed.

Player

Player keyword is a reference to player (player-controlled character) and can be used in the cases where some command should adress player character and to execute his/her member functions. Player's actual participation in conversation is not required to use this reference.

random (not yet supported)

Executes one of the commands in following block at random. Futher script execution fully depends on what command was executed. A sub-block of commands enclosed in braces counts as single "command".

Syntax:

```
random
 {
 <command1>;
 <command2>;
 {
 <multiple commands 1>
 }
 {
 <multiple commands 2>
 }
 ...
 }
```

say

Plays a single person's cue (saying). If player participates in conversation this command will also suspend futher script execution until user order to continue.

Syntax:

```
say [who [to_whom]] cue_text [<cue_voice | none>];
```

Who is a reference to person who is saying, *to_whom* is a reference to person who is being adressed. *Cue_voice* is an optional name (case insensitive) of a sound entry in SNDINFO or file in PK3 (without extension) to be played.
If *who* is not explicitly defined, ***myself*** reference is used by default; if *to_whom* is not defined ***initiator*** reference is used.

Cue_text should be represented by a quoted line or a number of quoted lines of text. Numerous lines can be separated either by '+' (plus), ',' (comma) or a number of commas. A plus character makes two lines be concatenated and a single space inserted between last symbol in first line and first symbol in second line. Comma character concatenates text lines and inserts linebreak between. Multiple commas make corresponding number of linebreaks inserted.

Examples:

```
"This cue displays only this short line of text"

"My name is Victor"+
 "and I am your friend."
```

```
"A text before two linebreaks.",,
 "A text after two linebreaks."

"First part of the phrase"+
 "second part of the phrase",
 "Second phrase after a linebreak."
```

A notice should be made, that in game cue text may be wrapped according to user-interface settings (and the screen width), so there’s no need to plan manual text wrapping when writing cue texts.

set

Applies new value to variable. This person’s local variables, other person’s local variables and and global variables could be changed this way.

Syntax:

```
set [<global | person_name>::]var_name <expression>;
```

Global keyword tells script to adress global variable; *person_name* makes it adress other person’s variable. *Var_name* is a name variable was registered under. Expression’s result will be applied as a new variable value.

skip (limited support)

Skips the rest of the speech block or sub-block (a number of commands enclosed in braces).

Syntax:

```
skip;
```

sound (not yet supported)

Plays a sound.

Syntax:

```
sound sound_name[ wait];
```

Sound_name is a name of a sound entry in SNDINFO or file in PK3 (without extension) to be played. ***Wait*** keyword makes script execution suspend until sound finish playing.

terminate

Terminates conversation without executing ‘bye’ speech.

Syntax:

terminate;

wait (not yet supported)

Suspends script execution for defined time.

Syntax:

wait time_ms;

Time_ms is a delay time in milliseconds;

World

Reference to World object that allows to execute its member functions. See object references and Appendix B.

yesno

Gives player an immediate selection between two common options: “yes” and “no”, – and executes single action depending on player’s choice. Action can be setting new variable value, jump or skip.

Syntax:

yesno var_name (yes_expression) (no_expression);

yesno jump yes_speech_name no_speech_name;

yesno <continue | end | skip | terminate> <continue | end | skip | terminate>;

First syntax style applies a value to variable with name *variable_name* depending on player choice; if player selected “yes” *yes_expression* result is applied, otherwise *no_expression* result is applied.

Second syntax style executes ***jump*** command either with *yes_speech_name* or *no_speech_name* as parameter correspondingly.

Third syntax executes ***end***, ***skip***, ***terminate*** or nothing (first command defined for “yes” selection, second for “no”).

Game object references

accessing an object reference

There is a number of object references available to use in conversation script. They are accessed by keywords Game, everybody, initiator, Level, myself, Player and World (see explanations above). For example, if you are going to refer to player character, use Player keyword.

executing object-adressed command

Commands issued to objects are also called functions or methods. There is a general syntax rule for executing any object-adressed command:

<reference>.<function>(<parameter list>);

Here *reference* is a desired object reference keyword, '.' (point) acts as object access operator, *function* is a desired command name. *Parameter list* in brackets should include all the values needed to specify command; if command does not need any futher specifications, brackets should be left empty.

Appendix A: Game object functions

Appendix B: World object functions

Appendix C: Level object functions

Map Number

Returns the current map's number, as defined in MAPINFO script. This function can be used in expressions.

Syntax:

Level.MapNumber();

NoiseAlert

Alerts all actors in the sound range of a target's presence, same as NoiseAlert action special.

Syntax:

Level.NoiseAlert([target_reference | target_TID, [emitter_reference | emitter_TID]]);

Target is an actor who is being alerted to. *Emitter* is an actor which is a centre of alert sound spreading. Both may be defined either by passing a reference, including ***everybody***, ***initiator***, ***myself*** or ***Player***, or by passing a thing's TID. If emitter is not specified, target is used as emitter. If neither target nor emitter are specified, a ***Player*** reference is used as both target and emitter.

Appendix D: Actor object functions

HasInventory

If referred character has got an inventory item or a quantity of items, this function will result in TRUE, otherwise it will result in FALSE.

Syntax:

<character>.HasInventory(item_name[, amount]);

Item name should be a name of inventory item: weapon, ammo, artifact and similar, without quotation marks; this parameter is case insensitive. *Amount* is an optional numeric value, which should be greater than zero.

This function can be used in expressions.

GiveExperience

Gives experience to referred character.

Syntax:

<character>.GiveExperience(amount);

GiveInventory

Gives an inventory item or a quantity of items to referred character.

Syntax:

<character>.GiveInventory(item_name [, amount]);

RemoveInventory

Removes an inventory item or a quantity of items from referred character. Those items are destroyed forever.

Syntax:

<character>.RemoveInventory(item_name [, amount]);

Appendix E: Player-specific object functions

GiveLevels

Gives amount of experience to referred player enough for him/her to advance by a number of levels.

Syntax:

Player.GiveLevels(amount);

A note should be made that if player already have experience earned on current level, it will be kept. For example, player is on level 2 with 1000 exps earned, that makes him closer to level 3 which requires earning 3000 exps over 2nd level cap, then you call **GiveLevels** to give him 2 levels extra, and he will receive all experience needed to get to the 4th level and his previously earned 1000 exps extra; finally he will be 4th level with 1000 exps kept to reach for 5th.

GiveCharisma

Gives amount of Charisma points to referred player.

Syntax:

Player.GiveCharisma(amount);

GiveConstitution

Gives amount of Constitution points to referred player.

Syntax:

Player.GiveConstitution(amount);

GiveDexterity

Gives amount of Dexterity points to referred player.

Syntax:

Player.GiveDexterity(amount);

GiveIntelligence

Gives amount of Intelligence points to referred player.

Syntax:

Player.GiveIntelligence(amount);

GiveStrength

Gives amount of Strength points to referred player.

Syntax:

Player.GiveStrength(amount);

GiveWisdom

Gives amount of Wisdom points to referred player.

Syntax:

Player.GiveWisdom(amount);

TeleportToMap

Teleports player to another map, same as Teleport_NewMap action special.

Syntax:

Player.TeleportToMap(map_id [, position_id [, keep_orientation]]);

Appendix F: Person-specific object functions

SetActiveConItem

Sets an active conversation item for given person.

Syntax:

<character>.SetActiveConItem(conitem_name | conitem_index);

Conversation item can be specified either by its script name, or by script index.