Part 2 – Basic things

Before we start working on our story we must define some basic stuff that we can reuse with other stories. Things like:

- common descriptions;
- common flags;
- common attributes;
- common triggers;
- logging;
- save and restore functions (to store progress);
- regression testing;
- the player object;
- mechanism for starting the game;
- mechanism for moving the player between locations;
- looking around;
- scoring mechanism;
- verbose function;
- quit function.

At this point it is helpful to read sections “locations objects and timers” (2 pages) and “location and object artifacts” (5 pages) from the XVAN Introduction document.

Inputs for this part of the tutorial are files part1-end.lib and part2-start.xvn.

Common descriptions

Common descriptions are descriptions that each location and object has. This is the reason they can be used with wildcards: it is guaranteed that they can be found at a later time when the wildcard is linked to an actual object or location.

We will use the following common descriptions (apart from the predefined descriptions):

d_longdescr long location or object description;
d_shortdescr short object description;

We make the following design decisions:

- For locations, d_shortdescr always is the location name only.
- Upon first visit of a location, the location's d_shortdescr and d_longdescr are printed;
- With following visits only d_shortdescr is printed;
- For an object, with each visit only d_shortdescr is printed;
- For objects the “examine” command prints d_longdescr;
- For locations the “examine” command acts as the “look” command (see further);
- We will also define a verbose() function to allow forced printing of long location descriptions at all times.

Common flags

We will use the following common flags (apart from the predefined flags):

f_seenbefore to determine whether to print long or short description.

Common attributes

We will use the following common attributes (apart from the predefined attribute):

r_be conjugation of verb ”to be” for location or object;
r_have conjugation of verb “to have” for location or object.

We must also define words for is, are, has and have in the vocabulary file. I chose to define them as verbs so they can also be used in user input (e.g. "where is the toaster"). If you only want to use them as attribute values for printing, they may have any word type.

Common triggers

We will use the following common triggers (apart from the predefined triggers):

t_i to print inventory
t_exa to examine

We make the following design decisions:

- When the player object enters a new location the predefined trigger t_entrance will be executed for the new location and all its contained objects;
- When the player object wants to exit from a location the predefined trigger t_exit will be executed for the current location and all its contained objects. If any of them responds with disagree() the player will not be allowed to leave the location;

We will now define the common triggers.

t_i

This trigger was already mentioned in part 1 of the tutorial. The inventory verb will print the "You are carrying" message and each object will print its description.

Our t_i common trigger

```
t_i
 if owns(o_player, %this) then
   indent()                          # indent level was set by the verb prologue
   printcr("[a] [this]")
 else
   nomatch()
```

Because this is a common trigger, each object and location will now have this trigger (you don’t have to worry about memory space, the code is only stored once). In case an object needs to print a different message (e.g. add text like "being worn") it suffices to redefine the trigger as a local trigger with the same name within the object body. Local triggers take preference over common triggers with the same name.

There's something new, here. The function nomatch() generates the third possible return code for a trigger (the others are agree() and disagree() as explained in part 1). With nomatch() a location or object tells the interpreter "forget that I had a match for this input". But why do we need nomatch(), we can also print nothing and return agree(), right? Wrong. If we return agree(), the interpreter would know there had been a match and no verb default code would be executed. So if the player carried nothing and all objects in the location would return agree(), the default verb text "nothing, you are empty-handed" would not be printed.

Nomatch() is quite powerful. If you redefine a common trigger and the local copy returns nomatch(). the interpreter will execute the common trigger as well. I use this sometimes to let the object do a quick test and if everything is ok, execute the common trigger after all.

t_exa

Our design decision states that we should print d_longdescr.

Our t_exa common trigger

```
t_exa
  printcr(d_longdescr)
  setflag(f_seenbefore)
```

the interpreter will know for which object or location the common trigger is executed, so it can locate the right description and flag. We could also have said printcr(%this.d_longdescr) and setflag(%this.f_seenbefore).

t_entrance

As per our design decision, t_entrance is triggered when the player object enters a new location. We want it to:

- print information about the location (long or short description);
- print information about the objects in the location;
- print information about objects in/on/under/.. other objects.

Here we go, comments added for clarification.

```
t_entrance
  if not(islit(o_player)) then
    printcr("It is pitch black.")
    disagree()   # ready, exit
  endif
  if equal(%this, l_location) then
    # l_location is wildcard for the current location
    printcr(d_shortdescr)            # print location name
    if not(testflag(f_seenbefore)) then
      # first visit
      setflag(f_seenbefore)
      printcr(d_longdescr)
    endif
  else
    # it´s not the current location but an object in the location
    if cansee(o_player, %this) then
      if owns(owner(o_player), %this) then
```

```
    # object is at the same containment level as player
    setflag(f_seenbefore)
    printcr(d_shortdescr)
   else
    if not(owns(o_player, %this, 0))  then
     # it´s not (in) some object the player carries (0 means all levels of containment)
     setflag(f_seenbefore)
     print("There is [a] [this] [r_preposition] [the] ")
     print(owner(%this))
     printcr(".")
    endif
   endif
  endif
endif
# all the endifs are not necessary at the end of a trigger.
```

So, what t_entrance does:

- check whether it's dark;
- check whether it's executed for the location (as opposed to an object in the location);
- check whether it's executed for an object in the location that the player can see;
- check whether this object is contained in another object not carried by the player if the object is carried by the player, we don't want to mention it).

Note: make sure to close the current IF-statement with an ENDIF when starting a new if statement. If you find that parts of your trigger should be executed but are not, you may have forgotten an ENDIF statement. If you forget the first ENDIF (line 5), nothing of the trigger will be executed when the player is lit because all lines will be considered part of the if not(islit(o_player)) branch.

In case the common t_entrance should not be executed for an object, define an empty t_entrance trigger locally with the object. Empty as in that it only contains an agree() function. We do this for example for the player object.

t_exit

As per our design decision, t_exit is triggered when the player object exits the current location. We want it to:

- check whether the player object is free to go.
- if not, it must return disagree()

As leaving a location is game specific, in our basic definitions the t_exit trigger will always return agree().

```
t_exit
 agree()
```

But how does it work then? We'll come to that when we define the player object, but here's a heads-up: XVAN has a function called exit(par). This function will call t_exit for par and all its contained objects. If one of the t_exit triggers returns disagree(), the exit function will return false and we know there's some object not allowing the player to exit the current location. Likewise, there is also an entrance(par) function that calls all t_entrance triggers.

logging

To log your game session, XVAN has the built-in transcript() function. This function copies user input and the game's response to a file called transcript.txt in the directory that the game is running from. Calling transcript for the second time will turn off logging. We will define transcriptas a verb in the vocabulary file so it can be used for all games.

```
VERB transcript
 "transcript"
  transcript()
 DEFAULT
  printcr("use 'transcript' to log your session.")
ENDVERB
```

save and restore functions

To store and load game progress, XVAN has functions save() and restore(). The save() function stores the current story progress in a file called save.dat in the directory that the game is running from. The restore() functions scans the directory for save.dat and loads it.

In order to use the functions, we create verbs "save" and "restore" in the vocabulary file. Restoring a game is pretty straightforward and always allowed, so we will define the restore functionality in the vocabulary file. The code for saving will be a local trigger in the player object. Why? Well, there may be game specific situations when we do not allow the user to save. For example in a maze or to prevent trial-and-error guessing when solving a puzzle.

```
$VERB save
 # define your save functionality in the story file
 DEFAULT
  printcr("Use 'save' to save your progress.")
ENDVERB
$VERB restore
 "restore"
  restore()
  printcr("restored.")
 DEFAULT
  printcr("use 'restore' to restore a previously saved game.")
ENDVERB
```

quit function

```
$VERB quit SYNONYM q
 "quit"
  print("Do you really want to quit? ")
  if yesno() then
   quit()
 DEFAULT
  printcr("use 'quit' to leave the game.")
ENDVERB
```

The yesno() function requires the user to enter "yes", "no", "y" or "n". It is not case sensitive.

o_player object

The player object is mandatory in each XVAN story file.

Before we start, the player object in part2-start.xvn looks like this:

```
$OBJECT o_player
 # The o_player object is predefined and represents the human player.
 DESCRIPTIONS
   d_sys
 CONTAINED
 FLAGS
 ATTRIBUTES
 TRIGGERS
END_OBJ
```

For starters, we will:

- set the system descriptions (d_sys) to “you” and "me";
- set common attribute r_be to “are”;
- set common attribute r_have to "have";
- define the nouns "you"and "me" and the "are" verb in the vocabulary file;
- override common triggers t_entrance and t_exit with local ones that do nothing.

```
$OBJECT o_player
 # The o_player object is predefined and represents the human player.
 DESCRIPTIONS
   d_sys              “You”, "me"
 CONTAINED        # don’t know where the player starts  until we have the game map
 FLAGS
 ATTRIBUTES
   r_be = are     # you are
   r_have = have  # you have
 TRIGGERS
   t_entrance
    agree()
   t_exit
    agree()
END_OBJ
```

Next, we will define the following basic stuff in the player object:

- starting the game;
- moving the player between locations;
- looking around;
- keeping the score;
- save and restore commands to store progress;
- logging;
- verbose function.

starting the game

If we do nothing, the game will just start with a “> “ prompt. However, we want to print some introductory text when the game starts. XVAN has no default starting mechanism, so we make our own. We define a timer m_init that reaches its threshold when the game starts.

```
m_init
 init              0
 step              1
 direction       up
 interval          1
 state           go
 trigger_at        1
 execute         o_player.t_init
```

With the player object, we will define a trigger t_init and a description d_init. The trigger prints the description.

```
$OBJECT o_player
 # The o_player object is predefined and represents the human player.
 DESCRIPTIONS
   d_sys              “You”, "me"
   d_init             “*** XVAN tutorial ***”
 CONTAINED        # don’t know where the player starts  until we have the game map
 FLAGS
 ATTRIBUTES
   r_be    = are     # you are
```

```
  r_have = have  # you have
TRIGGERS
  t_entrance
   agree()
  t_exit
   agree()
 t_init
  printcr(d_init)
  printcr("")
  entrance(owner(o_player))
END_OBJ
```

After starting the game, timer m_init expires and triggers o_player.t_init. This will print d_init, our opening message and describe the player's initial location.

Moving the player around

XVAN has no default mechanism to let the player move between locations. We will create our own mechanism. We will use several of XVAN's built-in functions to implement moving around. XVAN's built in functions are described in detail in a separate document.

In order to move around the player we must:

- make the player object catch user input about moving around;
- check if the direction indicated in the user input is a valid direction;
- check with all objects in scope whether the player may leave;
- move the player object to the new location and execute the t_entrance triggers.

This is our new player object

```
$OBJECT o_player
 # The o_player object is predefined and represents the human player.
 DESCRIPTIONS
   d_sys              “You”, "me"
   d_init             “*** XVAN tutorial ***”
 CONTAINED        # don’t know where the player starts  until we have the game map
 FLAGS
 ATTRIBUTES
   r_be    = are    # you are
   r_have = have  # you have
 TRIGGERS
     "[dir]"        -> t_move
     “go to [dir]”-> t_move
   t_entrance
     agree()
   t_exit
     agree()
  t_init
    printcr(d_init)
    printcr("")
    entrance(owner(o_player))
   t_move
     if valdir(l_location, %dir) then
       # it's a valid direction
       if exit(l_location) then
         # no object objects to the player leaving the room
         move(o_player, %dir) # move updates current location
         entrance(l_location)
       endif
     else
       nomatch()  # let other objects or verb default code react.
     endif
     agree()
END_OBJ
```

What do we see here? Right below the TRIGGERS keyword we see two possible user inputs that will fire the t_move trigger. In the t_move trigger, the valdir() function checks if the direction is a valid direction. If not, the t_move trigger returns nomatch(). In case none of the other objects react, the default verb code will be executed which will print “you can’t go that way”.

If the direction is valid, the exit() function will execute the t_exit triggers from all objects in the current location (and from the current location itself). If all return agree(), then the player object will be moved in the direction indicated by the user input. Finally, for the new location and all its contained objects, the t_entrance trigger will be executed.

You may have noted that move() accepts different kinds of parameters. With the verbs take and drop, we used move(object1, object2) which moved object2 in object1. With t_move we used move(object,

direction) which moved object to the location that is reached by going into the direction. Possible parameters combinations are listed in the function description document.

looking around

To enable the player to look around we define trigger t_look locally with the player object.

This trigger will call the entrance() function for the player's location. This means that for the player's location and each containing object, the t_entrance trigger will be called.

The player's t_look trigger

```
t_look
 if equal(owner(o_player, l_location)) then
  clearflag(l_location.f_seenbefore)
  entrance(l_location)
 else
  # the player is in some object. Print this information
  print("[[[prepos] [the] ")
  print(owner(o_player)
  print("].")
    clearflag(owner(o_player).f_seenbefore)
entrance(owner(o_player))
```

Because we (ab)use the entrance() function in our look command, we must clear the f_seenbefore flag before calling the entrance() function. The reason is that the t_entrance trigger will check for f_seenbefore and if it's set it will print the short description.

Why are there 3 '[' in the print statement? We want to print something like "[in the boat]". However, for the compiler, a '[' in a string means that a parameter will follow. We tell the compiler to print one '[' by entering '[['. So, '[[[' tells the compiler to print a '[' and that a parameter will follow.

Note that just defining t_look does not mean that the t_look trigger will be executed when the user enters "look". We must yet link the trigger to a user input. This is done at the beginning of the TRIGGERS section in the player object:

"look" -> t_look

Now the interpreter knows that whenever the user enters "look", it must execute the t_look trigger.

keeping the score

To keep track of the score, we define a local attribute r_score with the player object. We also define a verb "score" in the vocabulary and a local trigger t_score for the player object that prints the score.

save function

We create a local trigger t_save with the player object. Additionally, we create a local flag f_no_save with the player object. The t_save trigger will check the flag and if it is set, it will not save game progress. This can be used to prevent cheating. For example, when the player enters a maze, an object may set the o_player.f_no_save flag and thus prevent the player from saving progress while he is in de maze.

verbose function

By invoking the verbose functionality, the t_entrance triggers will always print the long location descriptions (d_longdescr). We define the verb 'verbose' in the vocabulary file and a local flag f_verbose with the player object. Next we change the common t_entrance trigger so it will test for the f_verbose flag before printing the location description.

This is what we have right now:

In the vocabulary file we've added:

```
$VERB score
  DEFAULT
    printcr("Use 'score' to get information about your score.")
ENDVERB
VERB verbose
  DEFAULT
    printcr("use 'verbose' to toggle long room descriptions.")
ENDVERB
```

'score' and 'save' will be handled by the player object. All other input with these verbs will print the default message.

Our player object with all the basic stuff we wanted:

```
$OBJECT o_player
 # The o_player object is predefined and represents the human player.
 DESCRIPTIONS
   d_sys              "You", "me"
   d_init             "*** XVAN tutorial ***"
 CONTAINED          # don't know where the player starts  until we have the game map
 FLAGS
f_no_save = 0
   f_verbose  = 0
 ATTRIBUTES
   r_be     = are     # you are
   r_have = have  # you have
 TRIGGERS
   "[dir]"-> t_move
   "go to [dir]"    -> t_move
   "look"           -> t_look
   "score"          -> t_score
   "save"           -> t_save
   "verbose"        -> t_verbose
 t_entrance
    agree()
  t_exit
    agree()
  t_init
    printcr(d_init)
    printcr("")
    entrance(owner(o_player))
  t_look
    if equal(owner(o_player), l_location) then
      clearflag(l_location.f_seenbefore)
```

```
    entrance(l_location)
   else
    # the player is in some object. Print this information
    print("[[[prepos] [the] ")
    print(owner(o_player))
    printcr("].")
   endif
   if cansee(o_player, owner(owner(o_player))) then
    entrance(owner(owner(o_player)))
   else
    entrance(owner(o_player))
  t_move
   if valdir(l_location, %dir) then
    # it's a valid direction
    if exit(l_location) then
     # no object objects to the player leaving the room
     move(o_player, %dir) # move updates current location
     entrance(l_location)
    endif
   else
    nomatch()  # let other objects or verb default code react.
   endif
   agree()
  t_score
   printcr("Your score is [r_score] points.")
  t_save
   if testflag(f_no_save) then
    printcr("Saving at this point would be like cheating.")
   else
    save()
    printcr("saved.")
  t_verbose
   if testflag(f_verbose) then
    clearflag(f_verbose)
    printcr("Verbose mode turned off.")
   else
    setflag(f_verbose)
    printcr("Verbose mode turned on.")
END_OBJ
```

And our common t_entrance trigger, adapted for verbose functionality:

```
t_entrance
 if not(islit(o_player)) then
  printcr("It is pitch black.")
  disagree()   # ready, exit
 endif
 if equal(%this, l_location) then
  # l_location is wildcard for the current location
  printcr(d_shortdescr)
  if not(testflag(f_seenbefore)) or testflag(o_player.f_verbose) then
   # first visit or verbose mode
   setflag(f_seenbefore)
```

```
    printcr(d_longdescr)
  endif
 else
  # it´s not the current location but an object in the location
  if cansee(o_player, %this) then
   if owns(owner(o_player), %this) then
    # object is at the same containment level as player
    setflag(f_seenbefore)
    printcr(d_shortdescr)
   else
    if not(owns(o_player, %this, 0))  then
     # it´s not (in) some object the player carries (0 means all levels of containment)
     setflag(f_seenbefore)
     print("There is [a] [this] [r_preposition] [the] ")
     print(owner(%this))
     printcr(".")
    endif
   endif
  endif
endif
# all the endifs are not necessary at the end of a trigger.
```

End of part 2

This ends part 2 of the tutorial. We've finished our preliminary work that we can use as a starting point for future stories.

Everything we did is in files part2-end.lib and part2-end.xvn. These files are the starting point for part 3 of the tutorial where we will write our sample story.

The files we created in part 2 will not compile to an XVAN story. This is because part2-end.xvn is not yet complete. By the end of the next part 3 we will have a playable story.