XVAN 2.6

-- tutorial --

-- everything is a location, an object or a timer --

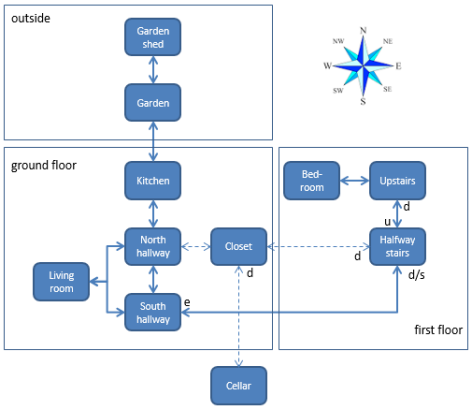


Table of Contents

About XVAN…

XVAN is a text adventure authoring system. It consists of a compiler, an interpreter and an authoring language.

The XVAN compiler is a console application. It takes a story file (text file) as input and compiles it to a binary with byte code.
The XVAN interpreter comes in 3 versions: a console version, a Glk version and a GUI-version. The interpreter executes the binary file created by the compiler.

As of version 2.3.4, IFI-XVAN is available. This version of the interpreter uses the IFI (Interactive Fiction Interface) to connect to the Brahman GUI, which supports in-game graphics, graphical map display, clickable links etc.

This tutorial does not (yet) handle the GUI-version, IFI_XVAN.

Platforms – Linux and Windows
XVAN is available for Linux and Windows. Story files and save files are portable, what you compile or save on one platform will work on the other platforms.

The XVAN Library
As of XVAN version 2.3.2, the XVAN Library is available. The Libraryt contains predefined XVAN code (verbs and artefacts) to offer a head start. Because the purpose of this tutorial is to demonstrate all XVAN concepts by creating a story from scratch, the Librrayt is not used in the tutorial.

About this tutorial

This tutorial teaches:

- setting up the basic things for a story;
- vocabulary;
- working with objects, locations and timers.

Basic things[1]

XVAN was developed based on the idea that the story author must have flexibility in writing his stories. This means very little is predefined, so the author initially must invest some time to take care of some basic things (moving around, inventory, save, restore, etc). But this needs only be done once, because the basics can be reused in other stories.

Vocabulary

The vocabulary is one of the basic things to set up, but it's important enough to have its own section in the tutorial. XVAN comes with one predefined word: "go". All other words must be defined by the author. The vocabulary is meant to be story independent so it can be reused and grow with next stories. It is advised to have the vocabulary in a dedicated file, separate from the story file. The XVAN Librray comes with a predefined vocabulary.

[1] the basic things are also covered in the Library, but as this is a tutorial the Library is not used here.

Object, locations and timers

As described in the introduction document, an XVAN story file is a collection of locations, objects and timers. The tutorial shows how to define these and how to model the behavior.

Structure of this tutorial

In this tutorial we will write a sample story through a set of exercises. At the start and end of each part we will have predefined vocabulary and story files. Of course you can create your own files, but for the tutorial we need predefined file sets that match with the tutorial text.

Note: the XVAN compiler requires unformatted text files. Copy/pasting source code from other file types may introduce invisible control characters that generate compiler errors.

Part 1 – the vocabulary

So, we are going to think about the vocabulary before we designed our story? Yes, we want to be able to reuse the vocabulary for other games so there's no problem making a first version without having a story yet.
We start with file "part 1 start.lib". The sections for nouns, adjectives, prepositions, adverbs, articles, question words and conjunctions are straightforward: the more words you put here, the more words the interpreter will know. A word can be in more than 1 section. E.g. 'orange' can be an adjective (color) but also a noun (fruit). The parser will combine word types in a sentence until it has a valid English syntax in the user input (or not).

For the remainder of this section we will focus on verbs and directions.

Verbs (and default code)

A verb can be defined as a single word:
$VERB take ENDVERB

But a verb can also be used as a last resort when a user command cannot be handled by a location or an object. As described in the XVAN introduction document, the user input is offered to all locations and objects in scope. If none of them can process the user input, we can fall back to the verb default code to respond to the message. Likewise, actions that will be always be handled in the same way (e.g. score) can be coded in the verb defaults.

For our tutorial game we want at least the following verbs:

- look or l;
- examine or x;
- get or take;
- drop;
- inventory or I;
- and the predefined verb 'go'.

Note: technically speaking, inventory is not a verb, but by defining it as one, the user can enter it as a single word on the command line.

look

With verb default code we are only interested in default replies, when none of the locations or objects have responded (which is very unlikely for 'look', by the way).

So, what kind of messages would do as default? The user could have entered different look commands, like "look", "look at something", "look in/on/under/… something". The verb syntax allows to define responses for all these situations. Responses could be "You see nothing special.", "You see nothing special about the something." and "There's nothing in/on/under/… the something."

In XVAN code:

```
$VERB look
  "look"
    printcr("You see nothing special.")

  "look at [o_subject]"
    printcr("You see nothing special about [the] [o_subject].")

  "look [prepos] [o_subject]"
    printcr("There's nothing [prepos] [the] [o_subject].")
ENDVERB
```

A little bit about [o_subject], [prepos] and [the]. These are called wildcards. During play, they are replaced with the user input. If the user types "look behind the couch", [prepos] will be replaced by 'behind' and [o_subject] by 'couch'. [the] means that the interpreter must print the article in case there is one defined with the subject. There also is [a] which will print either "a" or "an".

But 'at' also is a preposition. Because the "look at [o_subject]" line is defined before the "look [prepos] [o_subject]" line, this line will be executed first. If the lines were reversed, "look at" would never be executed because the interpreter stops after a matching line is found.

But we're not there yet. What if the user enters a look-sentence with a syntax that we didn't cover here?, like for example "look carefully" (carefully being an adverb)? We cannot cover everything, so we have the DEFAULT section. This usually prints a very generic response that fits all user input like "I only understood you as far as wanting to look."

Finally, suppose we're in the dark and can't see anything. Than all off the above is useless. We need some sort of initial test possibility, before anything is done with the verb. This is the verb prologue. The prologue is executed right after the user input is parsed, even before the input is handed over to the objects or locations.

Our verb ‘look’ with DEFAULT and PROLOGUE sections:

```
$VERB look
  PROLOGUE
    if not(islit(o_player)) then
      printcr(“It is pitch black.”)
      disagree()
    else
      agree()

  “look”
    printcr(“You see nothing special.”)

  “look at [o_subject]”
    printcr(“You see nothing special about [the] [o_subject].”)

  “look [prepos] [o_subject]”
    printcr(“There’s nothing [prepos] [the] [o_subject].”)

  DEFAULT
    printcr(“I only understood you as far as wanting to look.”)
ENDVERB
```

The function disagree() tells the interpreter to stop further execution of the user input. Likewise, agree() tells to continue. By default, the compiler will add agree() at the end of code.

examine

The verb examine differs from look in the way that it always requires a subject. If the user only enters ‘examine’, we want to ask for additional information about the subject to be examined. XVAN has the getsubject() function to do this.

In XVAN code:

```
$VERB examine SYNONYM x
  “examine”
    printcr(“What do you want to examine?”)
    getsubject()

  “examine [o_subject]”
    printcr(“You see nothing special about [the] [o_subject].”)

  DEFAULT
    printcr(“I only understood you as far as wanting to examine something.”)
ENDVERB
```

The SYNONYM keyword tells the interpreter that ‘examine’ and ‘x’ are identical.

There’s no prologue here to check if we are in the dark. Why? The parser will try to map the user input to a subject to be examined. In this process it will take into account whether the subject is visible to the player. So, when the user refers to a subject he cannot see, the parser will generate a “you don’t see that here” message. With the ‘look’ verb, there was no subject so we had to check it ourselves.

get

The verb 'get' is similar to 'examine', but we can make it a bit more complicated. We have to make a design decision for "get something":

- Either we say that all get actions should be defined by the objects and the verb will print a default message like "you can't get the [o_subject]."
  or
- We say that default behavior for "get something" is that it will be picked up and that any exceptions must be handled by the objects.

To make things a bit more interesting, let's implement the second option, so we give more intelligence to the verb.

First of all, before each execution of a get command, we must make sure the subject can be picked up. This can be implemented in a prologue:

```
PROLOGUE
  If equal(o_subject, %none) then
    agree()
  else
    if not(testflag(o_subject.f_takeable)) then
      printcr("[the] [o_subject] is not something that can be taken.")
      disagree()
  endif
```

Some new things here.

In the else part we test whether the subject can actually be picked up. We do this by testing the flag f_takeable. If it is set, the subject can be carried. If not, the verb will tell the interpreter to stop by calling the disagree() function.

As we will see in the story file, f_takeable is a common flag, each object and location has it. Common as opposed to local flags that are specific to an object or location. Testing a wildcard object (like o_subject) or location for a local flag will generate a compiler error, because it is not guaranteed at compile time that the subject will indeed have the local flag.

In the if part we test if the user actually entered a subject with his command. Suppose he only entered "take", then testing for flag o_subject.f_takeable would not be possible. We know that the user did not enter a subject when the o_subject wildcard has value %none.

Our get verb

```
$VERB get SYNONYM take SYNONYM grab
 PROLOGUE
  If equal(o_subject, %none) then
   agree()
  else
   if not(testflag(o_subject.f_takeable)) then
    printcr("[the] [o_subject] is not something that can be taken.")
    disagree()
  endif

 "get"
  printcr("What do you want to get?")
  getsubject()

 "get [o_subject]"
  If not(owns(o_actor, o_subject)) then
   move(o_subject, o_actor)
   setflag(o_subject.f_bypass)
   printcr("[o_subject]: taken.")
  else
   printcr("But [o_actor] already [o_actor.r_have] [the] [o_subject].")

 DEFAULT
  printcr("I only understood you as far as wanting to get something.")
END_VERB
```

The verb will generate an error response if the actor already carries the subject.

But what's with this o_actor thing? Why not use o_player? Well, there's a good explanation for that. Usually, the player will be the actor, but we can also give commands to an npc (non-player character). After parsing the user input, the interpreter will set o_actor to the entity who must perform the command.

Examples:

- "get lamp", o_actor = o_player
- "Fred, get lamp" , o_actor = o_fred

We don't know beforehand who the actor will be and we want the verb default code to work with all actors. Hence the [o_actor]. As a side effect we must also define an r_have attribute to store the actor's correct conjugation for the verb "to have" for printing (see part 2 about attributes).

But you don't use o_actor with the verbs Look and Examine? That's right, in my opinion the you-replies will do, even for other actors. It's my design decision, but feel free to change it.

There's one other flag we have not seen before: f_bypass. This is also a common flag and it tells the interpreter to bypass the visibility check for this subject. Remember from the examine verb that the interpreter will only consider objects that the player can see? Sometimes we want the player to be able to refer to objects he cannot see. Suppose he enters a dark location and wants to turn on the lamp in his inventory? We don't want a "You don't see a lamp here"-like response.

Our design decision is that all items in the player's inventory must be usable in darkness, so we set f_bypass when he picks something up and we clear it when he drops something.

drop

The drop verb is similar to get. But it will sometimes be the case that we want to put things *in* other things. To prevent us from having to code this functionality for each object we create default verb code that will not print a default "You cannot …"message but that will handle the "put something *in* something else" command. We must define a new common flag f_container that tells whether an object can contain other objects.

Actions to put or drop things under/on/besides/… other objects must be handled by the objects themselves. For these scenarios, the verb default code will print a rejection message. We could also have decided to handle these actions in the verb code but for this tutorial we will only do the "in" preposition.

Our drop verb, *almost finished*

```
$VERB drop SYNONYM put
 "drop"
  printcr("What do you want to drop?")
  getsubject()

 "drop [o_subject]"
  move(o_subject, owner(o_actor))
  clearflag(o_subject.f_bypass)
  printcr("[o_subject]: dropped.")

 "drop [o_subject] in [o_spec]"
   if not(testflag(o_spec.f_container)) then
    printcr("[the] [o_spec] cannot contain things.")
    disagree()   # exit
   endif
   if testflag(o_spec.f_locked) then
    printcr("[the] [o_spec] is locked.")
   else
    if not(testflag(o_spec.f_open)) then
     printcr("[[opening [the] [o_spec] first]")
     setflag(o_spec.f_open)
     clearflag(o_spec.f_opaque)
    endif
    clearflag(o_spec.f_bypass)
    move(o_spec, o_subject, in)
    printcr("[the] [o_subject] is now in [the] [o_spec].")

 "drop [o_subject] [prepos] [o_spec]"
  printcr("[the] [o_subject] is not something that can be put [prepos] [the] [o_spec].")

 DEFAULT
  printcr("I only understood you as far as wanting to drop something.")
ENDVERB
```

As you see, dropping the subject is moving it to the object (or location) that contains the player.

The verb also checks if the container is locked or closed and will open the container automatically if possible. Opening the container will also clear the common flag f_opaque so the player can see what's in the container.

o_spec is the specifier wildcard.

We see that we have rules for “drop [o_subject] in [o_spec]” and “drop [o_subject] [prepos] [o_spec]”. The rules are matched top to bottom, so it is important that the … in … rule is before the … [prepos] … rule. If not, then ‘in’ would be caught by the ‘prepos’ rule and the ‘in’ rule would never be executed.

We still forgot two things: handling the case where the user wants to drop a subject he is not carrying and putting an object into itself. We will handle both situations in the prologue. And yes, for the “get” verb we did not handle the getting-something-you-already-have scenario in the prologue. But because this is a tutorial we want to show different ways of doing things and hence we show how to handle the drop-something-you-don’t-have in the prologue.

Our final drop verb

```
$VERB drop SYNONYM put
  PROLOGUE
    if equal(o_subject, %none) then
      Agree()
    else
if not(owns(o_actor, o_subject)) then
       printcr(“But [o_actor] [o_actor.r_be] not holding [the] [o_subject].”)
       disagree()
     endif
     if not(equal(o_spec, %none) then
       if equal(o_subject, o_spec) then
         printcr("You cannot put something into itself.")
 endif
     endif
    endif    # endifs at the end are not necessary

  "drop"
    printcr("What do you want to drop?")
    getsubject()

  "drop [o_subject]"
    move(o_subject, owner(o_actor))
    clearflag(o_subject.f_bypass)
    printcr("[o_subject]: dropped.")

  "drop [o_subject] in [o_spec]"
     if not(testflag(o_spec.f_container)) then
       printcr("[the] [o_spec] cannot contain things.")
       disagree()   # exit
     endif
     if testflag(o_spec.f_locked) then
```

```
    printcr("[the] [o_spec] is locked.")
   else
    if not(testflag(o_spec.f_open)) then
     printcr("[[opening [the] [o_spec] first]")
     setflag(o_spec.f_open)
     clearflag(o_spec.f_opaque)
    endif
    clearflag(o_spec.f_bypass)
    move(o_spec, o_subject, in)
    printcr("[the] [o_subject] is now in [the] [o_spec].")

 "drop [o_subject] [prepos] [o_spec]"
  printcr("[the] [o_subject] is not something that can be put into [the] [o_spec].")

 DEFAULT
  printcr("I only understood you as far as wanting to drop something.")
ENDVERB
```

As with to 'get' where we needed the r_have attribute, with 'drop' we need an r_be attribute to store the actor's correct conjugation for to be.

inventory

Inventory illustrates XVAN's object oriented architecture. Each object in the player's possession will respond to the inventory command by printing its own description as we will see later in the story file.

The verb prologue will print the "You are carrying:" message. In case the player is carrying nothing (no object will react), the verb default code will print the "nothing, you are empty-handed." Message.

Our inventory verb

```
$VERB inventory SYNONYM i
 SCOPE player_only

 PROLOGUE
  printcr("You are carrying:")
  indent(2)

 EPILOGUE
  Indent(-2)

 DEFAULT
  Indent()
  printcr("Nothing, you are empty-handed.")
END_VERB
```

Some new things here:

SCOPE player_only means that the user input is only offered to the player and the objects that are contained in the player. Other scope values are:

- curr_loc (player's current location and all contained objects). This is the default scope value;
- all_locs (every location and object in the game).

The indent function, when called with a parameter increases or decreases the indent level. When called without a parameter the function prints the a number of spaces equal to the indent level.

The EPILOGUE is similar to the prologue, but is executed as the last item of processing the user input. Here we use it to reset the indent level to its old value.

go

The actual functionality of the go verb is defined in the story file with the player object. The go default code only prints the "you cannot go there" message.

Our go verb

```
$VERB go
  # do not remove this verb
  "go [dir]",
  "go to [dir]"
    printcr("You can't go that way.")

  DEFAULT
    printcr("I don't understand that sentence.")
END_VERB
```

What's new here?

[dir] is the wildcard for an arbitrary direction. Whenever the user enters only a direction, the interpreter will prefix it with the verb 'go', which is predefined by the compiler.

There are 2 command strings followed by 1 piece of code. This means that both commands will execute the same code. So, both "go [dir]" and "go to [dir]" will print "You can't go that way." as a default message when no objects reply to the command.

Directions

As the last activity of part 1 of the tutorial, we will define the directions that our story will understand.

Any word can be used as a direction and there can be as many or as few as you want. Compass directions, left, right, in, out, up, down, all are possible. Left and right are a bit tricky because they require knowledge about the direction the player is facing.

The actual story map will defined in the story file. For each location we will indicate which direction leads where.

For our tutorial we will use compass directions and up/down.

```
$DIRECTIONS
north       SYNONYM n,
south       SYNONYM s,
east        SYNONYM e,
west        SYNONYM w,
northeast   SYNONYM ne,
northwest   SYNONYM nw,
southeast   SYNONYM se,
southwest   SYNONYM sw,
up          SYNONYM u,
down        SYNONYM d
```

When printing a word that has synonyms, the interpreter will use the first word for printing. So, advice is to not use the abbreviations as the first words.

End of part 1

This is the end of part 1 of the tutorial. Everything we did is in file part1-end.lib. This file is the starting point for part 2 of the tutorial.

This file won't compile to an XVAN game, because we have not made the story file yet. In part 2 and part 3 we will create the story file.

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.

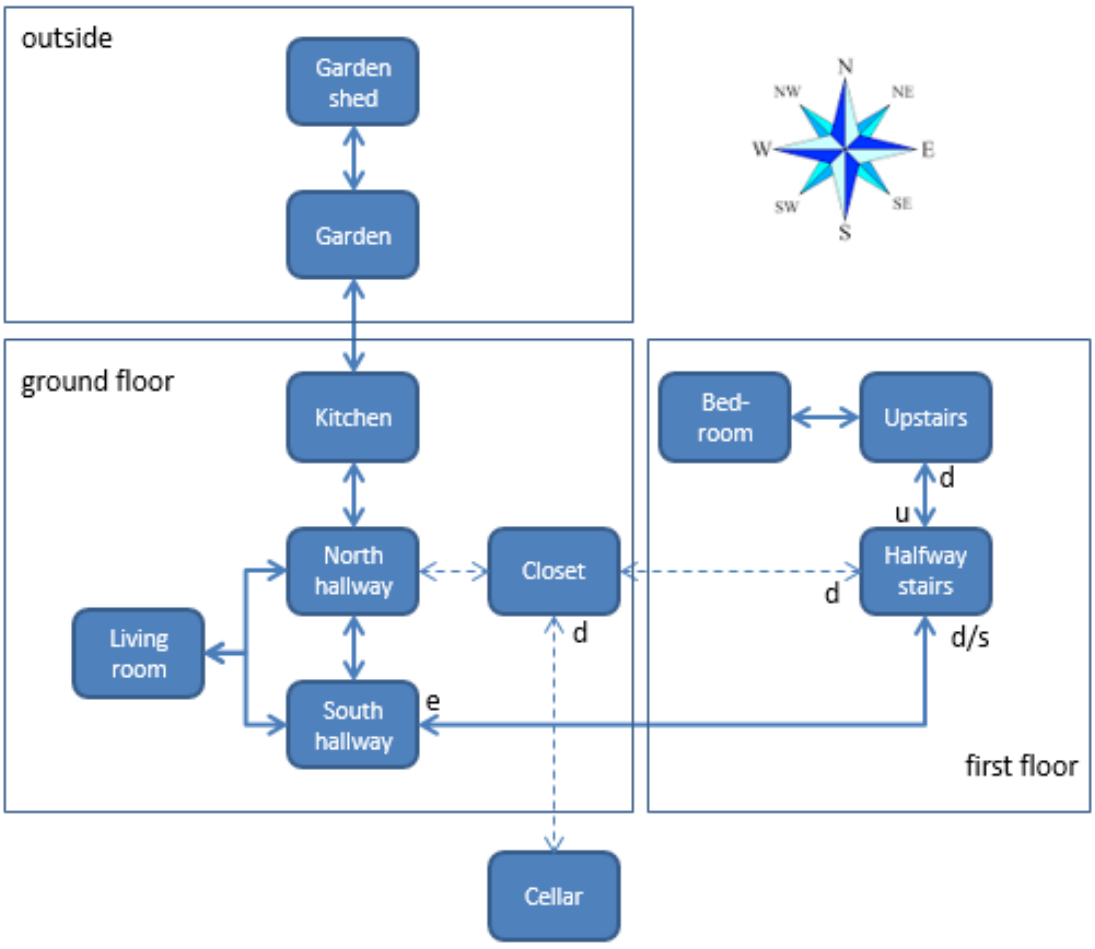
The files we created in part 2 will not compile to an XVAN story. This is because part2-end.xvn is not yet complete. By the end of the next part 3 we will have a playable story.

Part 3 – a sample story

Now that we've defined a vocabulary and took care of some basic requirements, we are ready to develop a sample story. Our purpose is not to create an award winning story, but to illustrate how to make locations, objects and timers and how they interact.

We won't make a full size adventure game (I'm not a great author anyway). It will be sort of a first level where you are in a house and must make your way to the cellar. The tutorial game will end when you descend the stairs to the cellar.

The map:



The locations

There are 11 locations. We will enter them in the story file with their long descriptions, short descriptions and directions. Actually, there are 12, we reserve on location, l_storage, to store objects that are removed from play. l_storage is not accessible to the player.

Some locations require non-standard handling of certain directions:

- when going east from the living room, we want to go back to the location we came from, south hallway or north hallway (we use an attribute to remember where we came from);

- going east from north hallway is not possible;
- going north from kitchen to garden is not possible when the kitchen door is locked;
- when the user has not examined the loose step on halfway stairs, going down from halfway stairs will lead to south hallway. After the user examines the loose step, going down will lead to the closet and south will lead to the south hallway.

The locations' code is below. With each location, we will explain new functionality, if any. Some locations also refer to objects, this code will be discussed with the object descriptions.

south hallway

```
$LOCATION l_hallway_south
 DESCRIPTIONS
  d_sys            "the south hallway"

  d_longdescr      "You are in the south hallway. To the west is a passage to the /
                    living room. To the east are stairs leading up. The hallway /
                    continues to the north."

  d_shortdescr     "South hallway"

 EXITS
  n -> l_hallway_north
  w -> l_living_room
  u -> l_halfway
  e -> l_halfway

 TRIGGERS
  "examine [l_hallway_south]"    -> o_player.t_look
  "west"                         -> t_west
  “north”                        -> t_north

  t_entrance
   move(o_stairs, %this)  # must be able to refer to stairs
   nomatch()

  t_west
   # remember where we came from
   l_living_room.r_back = %this
   nomatch()

  t_north
   # must be able to refer to the closet door
   o_closet_door.r_direction = east
   o_closet_door.r_access    = o_closet_door.d_closet
   move(o_closet_door, l_hallway_north)
   nomatch()

END_LOC
```

Description d_sys is the system description. It is a predefined common description and is used by the parser to map the user input to objects and locations.
To elaborate a bit, the user input is translated from a text string to separate words. The words are looked up in XVAN's word table and replaced by their word id (a number). Next, groups of word ids are held against the location and object tables and mapped on location or object ids. To map the word ids to object/location ids the parser compares them to the word ids from d_sys. As an example, the combination of two word ids for "south" and "hallway" will be mapped to one location id for l_hallway_south. When an object or location has no d_sys description, it cannot be referred to by the user.

Do not forget to include the article in the system description (d_sys). The compiler will strip it and store it separately. Whenever you include [the] or [a] wildcards in a string followed by a location or an object, the interpreter will check whether it has to print an article or not. If you did not include the article in the system description it won't print it. If you did include the article in d_sys but don't use [the] or [a] in a string, the article will not be printed either.
A slash '/' in a string tells the compiler to skip the next <cr> and spaces. It is used for formatting long text strings so they are better readable in the source file.

It may seem a bit unusual to move around the stairs (in trigger t_entrance) but this is just how we model the world. There are several locations from which the stairs are accessible. We could have created individual stair objects in different locations but that would require more code to keep them in sync. The net effect for the person playing the story will be the same and this makes our coding effort easier.

Trigger t_north is used to move around the closet door. It's like moving the stairs but a bit more complicated and will be explained with the closet door object.

north hallway

```
$LOCATION l_hallway_north
 DESCRIPTIONS
  d_sys              "the north hallway"

  d_longdescr        "You are in the north hallway. To the west is a passage to the /
                      living room. The hallway continues north to the kitchen."

  d_shortdescr       "North hallway"

 EXITS
  n -> l_kitchen
  s -> l_hallway_south
  w -> l_living_room

 TRIGGERS
  "examine [l_hallway_north]"  -> o_player.t_look
  "west"                       -> t_west

  t_west
   # remember where we came from
   l_living_room.r_back = %this
   nomatch()

END_LOC
```

living room

```
$LOCATION l_living_room
 DESCRIPTIONS
  d_sys              "the living room"

  d_longdescr        "This is the living room. It's completely abandoned. There is an /
                       exit to the hallway to the east."

  d_shortdescr       "Living room"

 EXITS
  # no exits

 ATTRIBUTES
  r_back = l_hallway_south

 TRIGGERS
  "examine [l_living_room]"    -> o_player.t_look
  "[dir]"                      -> t_go

  t_go
   if equal(%dir, east) then
     move(o_player, r_back)
     entrance(r_back)
```

```
    disagree()
  else
    nomatch()
  endif

END_LOC
```

We see that the living room location has a “%dir” trigger, as does the player object. We don’t know the exact order in which objects that are in scope get the user input, but it is ensured that a containing object gets it before its contained objects do. So the location always is the first to get the user input. In our case, because we know the location gets to process the %dir command first, our setup with the r_back attribute will work. In case the user enters any other direction than East, the nomatch() will make the player object’s t_move trigger to further process the user input.

kitchen

```
$LOCATION l_kitchen
 DESCRIPTIONS
  d_sys          "the kitchen"
  d_longdescr    "This is the kitchen. There is not much here. The /
                  hallway is to the south."
  d_shortdescr   "Kitchen"

 EXITS
  s -> l_hallway_north

 TRIGGERS
  "examine [l_kitchen]"   -> o_player.t_look
  "n"                     -> t_north
  “s”                     -> t_south

 t_entrance
  printcr(d_shortdescr)
  if not(testflag(f_seenbefore) AND not(testflag(o_player.f_verbose))) then
   printcr(d_longdescr)
  endif
  move(o_kitchen_door, %this)  # must be able to refer to the door

 t_north
  if testflag(o_kitchen_door.f_locked) then
   printcr("The kitchen door is locked.")
  else
   if not(testflag(o_kitchen_door.f_open)) then
    printcr("[[opening the kitchen door first]")
    setflag(o_kitchen_door.f_open)
    newexit(l_kitchen, north, l_garden)
   endif
   move(o_player, n)    # also updates current location
   entrance(l_location)
  endif
  disagree()  # prevent o_player.t_move to execute the "n" command

 t_south
  # must be able to refer to the closet door
  o_closet_door.r_direction = east
  o_closet_door.r_access    = o_closet_door.d_closet
  move(o_closet_door, l_hallway_north)
  nomatch()

END_LOC
```

The valdir() function checks for a valid direction (exit from the current location).

When the user wants to go north, the t_north trigger is fired. If the door is locked, we print a rejection message. If it is unlocked but closed, we don't print a "the door is closed" rejection message, but open the door for the player. Note the last disagree(). It tells the interpreter to stop and not offer the user's command to other objects. If we forget it, the command will also be sent to the o_player object who will execute it. Since at that moment we already are in the garden (t_north has already moved the player object to the north), the player will finally end up in the shed.

Trigger t_south is used to move around the closet door. It's like moving the stairs but a bit more complicated and will be explained with the closet door object..

closet

```
$LOCATION l_closet
 DESCRIPTIONS
  d_sys             "the closet"

  d_longdescr       "You are in a dark closet below the staircase. To the west is /
                     the closet door, which is closed."

  d_shortdescr      "Closet"

 EXITS
  u -> l_halfway
  d -> l_cellar

 TRIGGERS
  "examine [l_closet]"   -> o_player.t_look
  "down"                 -> t_down

  t_entrance
   printcr(d_shortdescr)
   printcr(d_longdescr)
   if not(testflag(o_trapdoor.f_hidden)) then
    printcr("Visible exits are up and down.")
   else
    printcr("The only visible exit is up.")
   endif

  t_down
   if testflag(o_trapdoor.f_hidden) then
    printcr("The carpet is blocking your way down.")
    disagree()
   else
    if not(testflag(o_trapdoor.f_open)) then
     printcr("The trapdoor is closed.")
     disagree()
    else
     nomatch()  # let o_player.t_move handle this

END_LOC
```

Flag f_hidden is a predefined common flag. When set, the object or location is treated by the parser as not visible, so the player won't be able to refer to it.

cellar

```
$LOCATION l_cellar
 DESCRIPTIONS
  d_sys                " the cellar"

  d_burning            "You walk down the stairs into the cellar. Down below you see /
                        the red glow of a fire. As you walk down further, it gets hotter /
                        and hotter. You realize you will be fried if you continue and you /
                        hurry back up the stairs."

  d_not_burning        "There is still a lot of smoke in the cellar, but through the /
                        hazes you can make out an old workbench to the east and a door /
                        to the north."

  d_shortdescr         "Cellar"

  d_end                "
/                      ***** this is the end of the tutorial *****
/                      “

EXITS
  up -> l_closet

 FLAGS
  f_tried_before = 0

 TRIGGERS

  "examine [l_cellar]" -> o_player.t_look

  t_entrance
   if testflag(o_flames.f_extinguished) then
    printcr(d_not_burning)
    printcr(d_end)
    quit()
   else
    if not(testflag(f_tried_before)) then
     setflag(f_tried_before)
     printcr(d_burning)
    else
     printcr("There's flames down there, remember?")
    endif
    move(o_player, u)
   endif
   agree()

END_LOC
```

halfway stairs

```
$LOCATION l_halfway
 DESCRIPTIONS
   d_sys              "halfway"

   d_longdescr        "You are now halfway up the stairs. The stairs continue up /
                      to the north and down to the south."

   d_shortdescr       "Halfway stairs"

   d_up_closed        "When you walk further up the stairs one of the steps makes /
                      a hollow sound. You try to pinpoint it but get no further /
                      than that it is somewhere in the upper half of the stairs.
/                     "

   d_up_open          "You carefully step over step 11, so you don't fall down /
                      into the closet.
/                     "

 EXITS
   n -> l_upstairs
   u -> l_upstairs
   s -> l_hallway_south
   d -> l_hallway_south

 TRIGGERS
   "examine [l_halfway]"  ->o_stairs.t_exa
   "up"                   -> t_up
   "north"                -> t_up
   “down”                 -> t_down

   t_entrance
     print(d_longdescr)
     move(o_stairs, %this)  # must be able to refer to stairs
     agree()

   t_up
      if testflag(o_button.f_pressed) then
      # step 11 is open
      printcr(d_up_open)
     else
      printcr(d_up_closed)

   t_down
     # must be able to refer to the closet door
     o_closet_door.r_direction = west
     o_closet_door.r_access    = o_closet_door.d_hallway
     move(o_closet_door, l_closet)
     nomatch()

END_LOC
```

Location halfway stairs has its own local t_entrance trigger, because we must move the stairs object to this location when the player enters. If we don't do this, then the user won't be able to refer to the stairs.

Trigger t_down is used to move around the closet door. It's like moving the stairs but a bit more complicated and will be explained with the closet door object..

upstairs

```
$LOCATION l_upstairs
 DESCRIPTIONS
  d_sys              "upstairs"

  d_longdescr        "You are upstairs. Behind you, the stairs lead down. There /
                     is an exit to the west."

  d_shortdescr       "Upstairs"

  d_down_closed      "When you walk down, one of the steps makes a hollow sound. /
                     You try to pinpoint it but get no further than that it is /
                     at the top half of the stairs.\n"

  d_down_open        "You carefully step over step 11, so you don't fall down /
                      into the closet.\n"

 EXITS
  s -> l_halfway
  d -> l_halfway
  w -> l_bedroom

 TRIGGERS
  "examine [l_upstairs]"  -> o_player.t_look
  "down"                  -> t_down
  "south"                 -> t_down

  t_entrance
   move(o_stairs, %this) # must be able to refer to stairs
   nomatch()

  t_down
   if testflag(o_button.f_pressed) then
    # step 11 is open
    printcr(d_down_open)
   else
    printcr(d_down_closed)

END_LOC
```

Again, we move the stairs object in a local t_entrance trigger because the player must be able to refer to the stairs. The local t_entrance trigger returns nomatch(), so the common t_entrance trigger will be executed as well.

bedroom

```
$LOCATION l_bedroom
 DESCRIPTIONS
  d_sys            "the bedroom"

  d_longdescr      "This location used to be a bedroom a long time ago. But /
                    now, there is nothing there. All furniture has been /
                    removed."

  d_shortdescr     "Bedroom"

  d_exa            "Mounted to the west wall are a sink and a tap."

 EXITS
  e -> l_upstairs

 TRIGGERS
  "examine [l_bedroom]" -> o_player.t_look

  t_entrance
   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

END_LOC
```

garden

```
$LOCATION l_garden
 DESCRIPTIONS
  d_sys             "the garden", "the hedges", "the hedge"

  d_longdescr       "You are in the garden at the back of the house. East and west /
                     there are hedges. To the north is a garden shed."

  d_shortdescr      "garden"

 EXITS
  s -> l_kitchen
  n -> l_shed

 TRIGGERS
  "examine [l_garden]" -> o_player.t_look

 t_entrance
  printcr(d_shortdescr)
  if not(testflag(f_seenbefore) AND not(testflag(o_player.f_verbose))) then
    printcr(d_longdescr)
  endif
  move(o_kitchen_door, %this)  # must be able to refer to the door

END_LOC
```

The garden is also described as the hedge and hedges. When we use [l_garden] in a string, the interpreter will always print it as “garden”, even if the user referred to it as hedge. If we want the interpreter to print the system description that the player used last, then we must set the predefined flag f_swap. Printing l_garden.d_sys will always print the first system description, regardless of f_swap.

shed

```
$LOCATION l_shed
 DESCRIPTIONS
  d_sys              "the garden shed"

  d_longdescr        "You are now in the garden shed. The shed hasn't been cleaned /
                      for a long time. Maybe never. On the walls you see the nails /
                      that were used to hang the garden utensils to. Almost all of /
                      them are gone now."

  d_shortdescr       "Garden shed"

 EXITS
  s -> l_garden

 TRIGGERS
  "examine [l_shed]" -> o_player.t_look

END_LOC
```

Why is the shed a location and not an object in the garden? That's just a design choice, it could have been an object as well. Making it an object in the garden is a bit more work though, because all user input will then be offered to the garden as well and we may have to write extra code for t_entrance and to move around (e.g. when in the shed object, "s" will take us to the kitchen).

The objects

Now that we've got the map, let's take a look at the objects.

We have the following objects:

- player
- nst (no such thing)
- kitchen door
- kitchen window
- toaster
- key hole
- rusty key
- glass fragment
- hacksaw
- stairs
- steps
- button (on stairs)
- closet door
- floor (in closet)
- carpet
- trapdoor
- drain pipe in bedroom
- drain pipe in closet
- flames
- tap
- water
- sink

Before going into the object descriptions, we'll briefly describe the plot of this tutorial game:

- go to the kitchen
- get the toaster and throw it through the window
- look through the kitchen door and notice the key on the outside;
- open the kitchen door
- go inside the shed and get the hacksaw
- go back into the kitchen and get the window fragment
- go to the stairs and find the button near step 11
- move the step and go down into the closet
- cut the carpet with the fragment
- open the trapdoor and see the flames
- cut the drain pipe with the hacksaw
- go up to the bedroom and open the tap
- go back into the closet and see that the flames are extinguished by the water
- enter the cellar
- end of first level

In the next sections we'll describe the objects, list the code and clarify where necessary.

player
We already addressed the player object in section 2 of the tutorial.

nst
o_nst is the 'no-such-thing' object. It's predefined by the compiler and must be in de story file. It is used with disambiguation rules as explained in part 5 of this tutorial. We'll leave it for now.

it
We want the player to be able to refer to a previous object by "it". We won't use it in this tutorial, but the o_it object is predefined by the compiler and must be in the story file. Don't worry about it.

Note: the o_nst and o_it objects are predefined in the XVAN Library (o_it as of version 1.1). If you use the Library, nst and it are taken care of automatically.

kitchen door
Is in the kitchen and leads to the garden. The door is locked and the key is in the key hole on the other side of the door. In the door is a window. The window and key hole are also defined as objects with their own t_entrance triggers.

It was a design decision to not mention the window in the door descriptions, because the window must be broken at some point which would result in outdated descriptions. It is better to let the window object handle this by itself.

```
$OBJECT o_kitchen_door
 DESCRIPTIONS
  d_sys              "the kitchen door"

  d_longdescr        "The door is made of wood; it gives access to the garden."

  d_longdescr1       "The door is made of wood; it leads back into the kitchen."

  d_shortdescr       "To the north is a door that leads to the garden."

  d_shortdescr1      "To the south is a door that leads to the kitchen."

  d_no_window        "In the upper half of the door is an opening where /
                       a window used to be "

 CONTAINED in l_kitchen

 FLAGS
  f_openable  = 1
  f_lockable  = 1
  f_locked    = 1

 TRIGGERS
  "examine [o_kitchen_door]"                    -> t_exa
  "look through [o_kitchen_door]"               -> o_kitchen_window.t_look_through
  "unlock [o_kitchen_door] with [o_rusty_key]"  -> t_unlock
  "turn [o_rusty_key]"                          -> t_unlock
  "open [o_kitchen_door]"                       -> t_open
  "close [o_kitchen_door]"                      -> t_close

  t_entrance
    if owns(l_kitchen, %this) then
      printcr(d_shortdescr)
    else
      printcr(d_shortdescr1)

  t_exa
    if owns(l_kitchen, %this) then
      print(d_longdescr)
    else
      print(d_longdescr1)
    endif
    if testflag(f_open) then
      print(" The door is open. ")
    else print(" The door is closed. ")
    endif
    # print info about window and keyhole
    if testflag(o_kitchen_window.f_broken) then
      print(d_no_window)
    else
      print(o_kitchen_window.d_shortdescr)
    endif
```

```
  printcr(o_keyhole.d_shortdescr)
  contents(o_keyhole)

 t_unlock
  if not(owns(o_player, o_rusty_key)) and not(owns(o_keyhole, o_rusty_key)) then
   printcr("[[picking up the rusty key first]")
   move(o_rusty_key, o_player)
  endif
  # verb prologue will check if already unlocked
  if not(owns(o_keyhole, o_rusty_key)) then
   printcr("[[putting the rusty key in the keyhole]")
  endif
  printcr("Ok, the kitchen door is now unlocked.")
  clearflag(f_locked)

 t_open
  # test for already open is done by verb prologue
  if not(testflag(f_locked)) then
   printcr("Ok, the kitchen door is now open")
   setflag(f_open)
   newexit(l_kitchen, north, l_garden)
  else
   printcr("The door seems to be locked.")

 t_close
  # test for already closed is done by verb prologue
  printcr("Ok, the kitchen door is now closed.")
  clearflag(f_open)
  blockexit(l_kitchen, n)

END_OBJ
```

To create and delete exits we use functions newexit() and blockexit().

For this object, we also need verbs "unlock" and "open". And while we're at it, we will create "lock" and " close" as well.
With these verbs we test as many general things (already open/closed/locked/unlocked) in the verb prologue, so we don't have to repeat the same tests in the objects. The general tests do require some additional common flags: f_openable, f_open, f_lockable, f_locked.

verb unlock

```
$VERB unlock
 PROLOGUE
  if not(equal(o_subject, %none)) then
   if not(testflag(o_subject.f_lockable)) then
    printcr("[the] [o_subject] is not something that can be unlocked.")
    disagree()
   else
    if not(testflag(o_subject.f_locked)) then
     printcr("But [the] [o_subject] [o_subject.r_be] not locked.")
     disagree()
```

```
      endif
    endif
  endif    # endifs at the end of code  may be omitted

 "unlock"
  printcr("What do you want to unlock?")
  getsubject()

 "unlock [o_subject]"
  printcr("How do you want to unlock [the] [o_subject]?")
  getspec()

 "unlock [o_subject] with [o_spec]"
  printcr("[the] [o_actor] cannot unlock [the] [o_subject] with [the] [o_spec].")

 DEFAULT
  printcr("I only understood you as far as wanting to unlock something.")
ENDVERB
```

verb open

```
$VERB open
 PROLOGUE
if not(equal(o_subject, %none)) then
    if not(testflag(o_subject.f_openable)) then
      printcr("[the] [o_subject] is not something that can be opened.")
      disagree()
    else
      if testflag(o_subject.f_open) then
        printcr("But [the] [o_subject] [o_subject.r_be] is already open.")
        disagree()
    endif
  endif
 endif

 "open"
  printcr("What do you want to open?")
  getsubject()

 "open [o_subject]"
  printcr("[the] [o_actor] can't open that.")
ENDVERB
```

verb lock

```
$VERB lock
  PROLOGUE
    if not(equal(o_subject, %none)) then
      if not(testflag(o_subject.f_lockable)) then
        printcr("[the] [o_subject] is not something that can be locked.")
        disagree()
      else
        if testflag(o_subject.f_locked) then
          printcr("But [the] [o_subject] [o_subject.r_be] is already locked.")
          disagree()
        endif
      endif
    endif

  "lock"
    printcr("What do you want to lock?")
    getsubject()

  "lock [o_subject]"
    printcr("How do you want to lock [the] [o_subject]?")
    getspec()

  "lock [o_subject] with [o_spec]"
    printcr("[the] [o_actor] cannot lock [the] [o_subject] with [the] [o_spec].")

  DEFAULT
    printcr("I only understood you as far as wanting to lock something.")
ENDVERB
```

verb close

```
$VERB close
  PROLOGUE
    if not(equal(o_subject, %none)) then
      if not(testflag(o_subject.f_openable)) then
        printcr("[the] [o_subject] is not something that can be closed.")
        disagree()
      else
        if not(testflag(o_subject.f_open)) then
          printcr("But [the] [o_subject] [o_subject.r_be] is already closed.")
          disagree()
      endif
    endif
  endif

  EPILOGUE
    if not(islit(o_player)) then
      # they may have closed a container with the light source
      printcr("It is now pitch black.")
      disagree()
```

```
 "close"
   printcr("What do you want to close?")
   getsubject()

 "close [o_subject]"
   printcr("[the] [o_actor] can't close that.")
ENDVERB
```

You notice that for verb close we also have an epilogue. In the epilogue we check if closing the subject made the light source invisible. For example, if the player puts his flashlight in a box and closes it, it will become dark. The epilogue will detect this.

kitchen window

As already mentioned with the kitchen door object, the kitchen window is an autonomous object, because it will have different behavior once it is broken. It makes less sense to code this all with the door object.

```
$OBJECT o_kitchen_window
 DESCRIPTIONS
  d_sys              "the kitchen window"

  d_longdescr        "The window is made of glass, which somehow doesn't /
                      surprise you."

  d_shortdescr       "In the upper half of the door is a window "

  d_smash_no         "You smash the window with your fist, but with no /
                     success. You need something heavy to break the /
                     window."

  d_broken           "\nScattered over the floor is a broken window that /
                      once was a part of the door."

  d_look_glass       "Through the window you see the garden. At the far /
                      face to the window and try to look down but can't /
                      see right behind the door. If you could only stick /
                      your face further through."

  d_look_no_glass    "Because the window is no longer there, you can stick /
                      your head through the hole. There's a rusty key in /
                      the outside of the keyhole!"

  d_climb            "You don't fit through the window. It's way too small (or /
                      you are too big)."

 CONTAINED in o_kitchen_door

 FLAGS
  f_broken = 0
```

```
TRIGGERS
  "examine [o_kitchen_window]"                    -> t_exa
  "look through [o_kitchen_window]"               -> t_look_through
  "break [o_kitchen_window]"                      -> t_break_no
  "break [o_kitchen_window] with [o_spec]"        -> t_break
  "throw [o_subject] [prepos] [o_kitchen_window]" -> t_throw
"climb through [o_kitchen_window]"                -> t_climb
  "go through [o_kitchen_window]"                 -> t_climb

  t_entrance
    if testflag(f_broken) then
      printcr(d_broken)

  t_look_through
    if testflag(f_broken) then
      printcr(d_look_no_glass)
      clearflag(o_rusty_key.f_hidden)
    else
      printcr(d_look_glass)

  t_break_no
    printcr(d_smash_no)

  # break and throw cannot be the same trigger because they
  # have their subject and specifier reversed.
  t_break
    print("You throw [the] [o_spec] at the window")
    if not(testflag(o_spec.f_heavy)) then
      printcr(", but it bounces back. It obviously isn't /
            heavy enough.")
      move(o_spec, l_kitchen)
    else
      printcr(" and it goes straight through. The window /
            is shattered all over the floor. One of the /
            glass fragments is a bit larger than the rest.")
      move(o_spec, l_garden)
      setflag(f_broken)
      move(%this, l_kitchen)
      move(o_fragment, l_kitchen)
    endif

  t_throw
    # may only work for 'at' and 'through'
    if equal(%prepos, at) OR equal(%prepos, through) then
      print("You throw [the] [o_subject] at the window")
      if not(testflag(o_subject.f_heavy)) then
        printcr(", but it bounces back. It obviously isn't /
              heavy enough.")
        move(o_subject, l_kitchen)
      else
        printcr(" and it goes straight through. The window /
              is shattered all over the floor. One of the /
```

```
            glass fragments is a bit larger than the rest.")
      move(o_subject, l_garden)
      setflag(f_broken)
      move(%this, l_kitchen)
      move(o_fragment, l_kitchen)
    endif
   else
    nomatch()

 t_climb
  printcr(d_climb)
  disagree()

END_OBJ
```

Triggers t_break and t_throw are almost identical. But because they have subject and specifier reversed, we must code separate triggers. We also need an additional common flag for these triggers: f_heavy.

We must also define some additional verbs in our vocabulary: break, throw and climb.

verb climb

```
$VERB climb
 "climb"
  printcr("What do you want to climb?")
  getsubject()

 "climb [o_subject]"
  printcr("[the] [o_subject] is not something to climb.")

 "climb [prepos] [o_subject]"
  printcr("[the] [o_actor] cannot climb [prepos] [the] [o_subject].")

 DEFAULT
  printcr("I only understood you as far as willing to climb something.")
ENDVERB
```

verb break

```
VERB break SYNONYM destroy
 "break"
  printcr("What do you want to break?")
  getsubject()

 "break [o_subject]"
  printcr("[the] [o_actor] can't break [the] [o_subject].")

 "break [o_subject] with [o_spec]"
  printcr("[the] [o_actor] can't break [the] [o_subject] with [the] [o_spec].")

 DEFAULT
  printcr("I only understood you as far as wanting to break something.")
ENDVERB
```

verb throw

```
VERB throw
 PROLOGUE
  # actor must hold the subject
  if not(equal(o_subject, %none)) then
   if not(owns(o_actor, o_subject)) then
    printcr("[the] [o_actor] must be holding [the] [o_subject] first.")
    disagree()
   endif
  endif

 "throw [o_subject] [dir]",
 "throw [o_subject] to [dir]"
  if valdir(l_location, %dir) then
   move(o_subject, %dir)
   printcr("Thrown.")
  else
   printcr("[the] [o_subject] bumps to the [dir] wall and falls on the floor.")
   move(o_subject, l_location)

 "throw [o_subject] [prepos] [o_spec]"
  printcr("Throwing [the] [o_subject] [prepos] [the] [o_spec] won't work.")


 DEFAULT
  printcr("I only understood you as far as wanting to throw something.")
ENDVERB
```

We also coded some standard functionality in the throw verb for throwing objects in a particular direction.

The strings "throw [o_subject] to [dir]" immediately follows "throw [o_subject] [dir]". This means that the code that follows applies to both commands.

Next, we'll continue with the keyhole object.

keyhole

```
$OBJECT o_keyhole
 DESCRIPTIONS
  d_sys            "the keyhole"

  d_longdescr      ""

  d_shortdescr     "and you also see a keyhole."

  d_peek           "You peek through the keyhole but you cannot see a thing. /
                    something on the other side of the keyhole blocks your view."

  d_look           "You see the garden. At the far east end, there is a /
                    garden shed."

 CONTAINED in o_kitchen_door

 TRIGGERS
  "examine [o_keyhole]"        -> t_look_through
  "look through [o_keyhole]"   -> t_look_through

  t_entrance
   # don't print there is a keyhole when entering the room
   agree()

  t_look_through
   if owns(o_keyhole, o_rusty_key) then
    if cansee(o_player, o_rusty_key) then
     printcr("You can't, since there is a key in the keyhole.")
    else
     printcr(d_peek)
    endif
   else
    printcr(d_look)
   endif
   disagree()

END_OBJ
```

With the keyhole object we don't use d_longdescr and d_shortdescr. For examining we use t_look_through and default t_entrance will always use d_sys because the keyhole is a part of the door that cannot be removed ("…. [this]…." Will print d_sys from the current object).

This part of the common t_entrance trigger applies for the keyhole object:

```
    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
```

rusty key

```
$OBJECT o_rusty_key
 DESCRIPTIONS
  d_sys              "the rusty key"

  d_longdescr      "An old rusty metal key."

  d_shortdescr     "An old rusty metal key."

 CONTAINED in o_keyhole

 FLAGS
  f_takeable = 1
  f_hidden   = 1   # key is in the other side of the keyhole

 TRIGGERS
  "inventory"                    -> t_i
  "examine [o_rusty_key]"    -> t_exa

END_OBJ
```

glass fragment

```
$OBJECT o_fragment
 DESCRIPTIONS
  d_sys             "the glass fragment", "the shard", "the splinter"

  d_longdescr       "The fragment is about 5 inches long and has a sharp edge."

  d_shortdescr      "There is a glass fragment here."

  d_carpet          "You cut the carpet along its sides and it comes loose /
                     from the floor, revealing a trapdoor!"

 CONTAINED in l_storage

 FLAGS
  f_takeable = 1

 TRIGGERS
  "inventory"                          -> t_i
  "examine [o_fragment]"               -> t_exa
  "cut [o_carpet] with [o_fragment]"   -> t_cut

  t_cut
    if not(owns(o_player, %this)) then
      printcr("[[picking up the fragment first]")
    endif
    if not(testflag(o_carpet.f_cut)) then
      setflag(o_carpet.f_cut)
      move(o_trapdoor, l_cellar)
      move(o_carpet, o_player)
      setflag(o_carpet.f_bypass)
      printcr(d_carpet)
    else
     printcr("You already did that.")

END_OBJ
```

We don't want the carpet lying around after cutting it, so we make the player pick it up in the cut action.

For the glass fragment we need to define the verb "cut".

verb cut

```
$VERB cut SYNONYM saw
  "cut"
    printcr("What do you want to cut?")
    getsubject()

  "cut [o_subject]"
    printcr("How do you want to cut [the] [o_subject]?")
    getspec()

  "cut [o_subject] with [o_spec]"
    printcr("[the] [o_actor] cannot cut [the] [o_subject] with [the] [o_spec].")

  DEFAULT
    printcr("I only understood you as far as wanting to cut something.")
ENDVERB
```

toaster

The toaster object is in the kitchen. We need the toaster to break the window in the kitchen door so we can reach the key that is on the outside of the door

The toaster object

```
$OBJECT o_toaster
 DESCRIPTIONS
   d_sys              "the toaster"

   d_longdescr      "An old toaster, quite heavy. The power cord /
                       has been cut off."

   d_shortdescr     "There's a toaster here."

 CONTAINED in l_kitchen

 FLAGS
   f_takeable = 1
   f_heavy     = 1

 TRIGGERS
   "inventory"                -> t_i
   "examine [o_toaster]"   -> t_exa

END_OBJ
```

We set flag f_heavy for the toaster so it can be used to break the window in the kitchen door.

hacksaw

```
$OBJECT o_hacksaw
 DESCRIPTIONS
  d_sys           "the hacksaw", "the saw"

  d_longdescr   "This is just an ordinary hacksaw. It can be used /
                 to saw metal objects. The saw looks a bit worn, /
                 but it probably will last for one more saw job."

  d_shortdescr  "There is a hacksaw here."

  d_no_saw      "The saw is pretty worn. It will probably last for /
                 one more saw job and your planned action is unlikely /
                 to be that job."

  d_worn        "The saw is completely worn out. Whatever you are going to /
                 do with it, it won't be a saw job."

 CONTAINED in l_shed

 FLAGS
  f_takeable = 1
  f_worn     = 0

 TRIGGERS
  "inventory"                          -> t_i
  "examine [o_hacksaw]"                -> t_exa
  "saw [o_subject] with [o_hacksaw]"   -> t_saw

  t_exa
   if testflag(f_worn) then
    printcr(d_worn)
   else
    printcr(d_longdescr)
   endif
   disagree()

  t_saw
   if not(equal(o_subject, o_drain_pipe_closet)) then
    if testflag(f_worn) then
     printcr(d_worn)
    else
     printcr(d_no_saw)
END_OBJ
```

We only allow the user to use the hacksaw once, to cut the drain pipe in the closet. For all other situations we have defined rejection messages.

We do not make a separate “saw” verb but define a synonym for the “cut” verb instead.

stairs

The stairs is an object that will be available in the following locations:

- l_hallway_north
- l_halfway;
- l_upstairs.

From within these locations, the player must be able to refer to the stairs. The stairs will be moved to the location once the player enters it.

The first time the examine command is given, it will only work if the player is in location l_halfway (halfway up the stairs). Once examined from location l_halfway, the examine command will also work from the other two locations (we use flag f_exa to check for this).

```
$OBJECT o_stairs
 DESCRIPTIONS
  d_sys              "the stairs", "the staircase"

  d_exa              "You see nothing special about the stairs."

  d_exa_hollow       "It looks just like a staircase with one /
                      step that sounds hollow when stepped on."

  d_longdescr        "It's a wooden staircase. There are 15 steps. You can refer /
                      to a particular step with 'step <number>'."

  d_shortdescr       ""  # included in room description for hallway south and halfway

  d_count            "There are 15 steps. You can refer to a particular step with /
                      'step <number>'."

  d_cant_see         "It's hard to get a good view from here. If you were halfway /
                      the stairs you would have a better view."

 CONTAINED in l_hallway_south

 FLAGS
  f_exa = 0  # Not yet examined.

 TRIGGERS
  "examine [o_stairs]"   -> t_exa
  "look at [o_stairs]"    ->t_exa
  "count [o_steps]"       -> t_count

  t_entrance
   agree()  # Must execute t_entrance for contained objects (steps).

  t_exa
   if (equal(l_location, l_hallway_south) OR equal(l_location, l_upstairs))
       AND not(testflag(f_exa)) then
  printcr(d_cant_see)
```

```
   else
    # we are halfway
    # if they have not yet heard the hollow sound, we don't mention it
    if testflag(l_upstairs.f_seenbefore) then
     setflag(f_exa)
     printcr(d_exa_hollow)
    else
     printcr(d_longdescr)

 t_count
  printcr(d_count)

END_OBJ
```

Now, we also need a verb 'count':

```
$VERB count
 "count"
  printcr("1 2 3")

 "count [o_subject]"
  printcr("[the] [o_subject] is not something that can be counted.")

 DEFAULT
  printcr("I only understood you as far as wanting to count something.")
ENDVERB
```

steps

The steps object is part of the stairs. There are 15 steps and they can be referred to individually (but there is only one steps object). Referring to steps goes by “step <number>”. The number entered by the player is captured in the %ord wildcard, where ord stands for ordinal.

A little something about number wildcards
XVAN has two number wildcards: %value and %ord. The difference is best explained with some examples:
%ord captures ordinal numbers, something with a certain order. “examine step 5” will cause the number 5 to be stored in %ord.
%value captures values, all other numbers. “set dial to 1234” or “enter 1234 on keypad” will store the number 1234 in %value.

Step 11 is a special step, as soon as the player examines it, he will be notified that there is a button next to the step.

the steps object

```
$OBJECT o_steps
 DESCRIPTIONS
  d_sys          "the steps", "the step"

  d_longdescr    "There's a tiny button on the side of the step."
```

```
 d_shortdescr    ""

 d_15            "There are only 15 steps."

 d_which         "If you want to do something to a specific step, please refer to /
                 the step as 'step <number>'."

 d_moved_11      "Step 11 has disappeared, revealing a passage down."

CONTAINED in o_stairs

FLAGS
 f_swap = 1  # always print the d_sys last referred to by the user

TRIGGERS
 "examine [o_steps]"          -> t_exa
 "examine [o_steps] [ord]"    -> t_exa_step

t_entrance
 agree()

t_exa
 if not(trigger(o_stairs.t_exa)) then
  disagree()

t_exa_step
 if (equal(l_location, l_hallway_south) OR equal(l_location, l_upstairs))
     AND not(testflag(o_stairs.f_exa)) then
  printcr(o_stairs.d_cant_see)
  disagree() # stop
 endif

 if lt(%ord, 1) or gt(%ord, 15) then
  printcr("Steps are numbered from 1 to 15.")
 else
  # step 11 gives access to the closet
  if equal(%ord, 11) then
   if not(testflag(o_button.f_pressed)) then
    printcr(d_longdescr)
    clearflag(o_button.f_hidden)
   else
    printcr(d_moved_11)
  else
   printcr("You see nothing special about step [ord].")

t_default
 if equal(o_subject, o_steps) then
  printcr(d_which)
  disagree()
 else
 nomatch()  # this is important for default verb code
```

```
END_OBJ
```

The trigger() function is used to execute a trigger from another object or location. It returns true or false. When the trigger to be executed returns disagree, the trigger() function will return false.

The t_default trigger is a special system defined trigger. If none of the triggers of an object fired, the t_default trigger - if present - will fire. We use it here to catch all actions on the steps that we did not foresee and print a message on how to refer to the steps. Since the o_steps object receives ALL user input, it must check the subject and only reply if the subject is o_steps. If not, it is very important to return a nomatch() result because otherwise the interpreter will see that atrigger fired and it will not call verb code.

button

The button is hidden until the player examines step 11.

```
$OBJECT o_button
 DESCRIPTIONS
  d_sys              "the button"

  d_longdescr        "A round button in the same color as the stairs. You have to look /
                      really close to notice it."

  d_shortdescr       "
/                     There's a tiny button on the side of step 11."

  d_press            "As you press the button, step 11 retracts a bit, lowers /
                      about an inch and then slides backwards out of sight, /
                      revealing a passage down into the closet!"

 CONTAINED in l_halfway

 FLAGS
  f_hidden    = 1
  f_pressed   = 0

 TRIGGERS
  "examine [o_button]" -> t_exa
  "examine [o_stairs]" -> t_exa_stairs
  "press [o_button]"   -> t_press

  t_entrance
    if not(testflag(f_hidden)) then
      # the button is visible
      if not(testflag(f_pressed)) then
        printcr(d_shortdescr)
      else
        printcr(o_steps.d_moved_11)

  t_exa_stairs
```

```
  if not(testflag(f_hidden)) then
    printcr(d_shortdescr)

 t_press
   if testflag(f_pressed) then
     printcr("Nothing happens.")
   else
     printcr(d_press)
     o_player.r_score += 50
     printcr("")
     printcr("[[Your score just went up by 50 points!]")
     setflag(f_pressed)
     blockexit(l_halfway, d)
     newexit(l_halfway, d, l_closet)
   endif
   disagree()

END_OBJ
```

closet door

The closet door cannot be opened. Access to the closet is through the staircase when step 11 is open.

```
$OBJECT o_closet_door
 DESCRIPTIONS
   d_sys              "the closet door"

   d_longdescr        "The closet door seems to be locked."


   d_shortdescr       "To the [r_direction] is a door that gives access /
                      to [r_access]."

   d_closet           "a closet under the stairs"

   d_hallway          "the north hallway"

   d_no_unlock        "[the] [o_spec] does not fit."

 CONTAINED in l_hallway_north

 ATTRIBUTES
   r_direction     = east
   r_access        = d_closet

 FLAGS
   f_openable      = 1
   f_lockable      = 1
   f_locked        = 1

 TRIGGERS
   "east"                                    -> t_east
   "examine [o_closet_door]"                 -> t_exa
```

```
  "open [o_closet_door]"                          ->t_locked
  "unlock [o_closet_door] with [o_rusty_key]"     ->t_unlock

  t_east
   printcr("The closet door is closed.")
   disagree()

  t_locked
   printcr(d_longdescr)

  t_unlock
   printcr(d_no_unlock)

END_OBJ
```

The closet door is moved around between locations l_hallway_north and l_closet. We see that its shortdescr description contains two attributes: direction and access. Depending on whether the closet door object is in the north hallway or the closet, we change the value of the attributes. This ensures that in t_entrance the correct description will be printed:
“To the east is a door that gives access to a closet under the stairs.”
Or
“To the west is a door that gives access to the north hallway.”

But, wait a second. I understand you want to move the closet door to the locations where it must be in scope. I compared it to the stairs object that is moved around as well, and the stairs object is moved in the t_entrance trigger from the location where it must end up whereas the closet door object is moved in a special trigger from the location that the player is leaving.
=> when the player is moving from south hallway to halfway stairs, the stairs object is moved to halfway stairs in t_entrance from halfway stairs.
=> when the player is moving from the kitchen to hallway north, the closet door object is moved in t_south from the kitchen and NOT in t_entrance from hallway north.
Why?

There’s a good reason for that. The stairs object has no actions for its t_entrance trigger (other than agree). The closet door’s t_entrance trigger must print a description. Remember that in the player’s t_move trigger the entrance(l_location) function is called? This function creates a list of all objects whose t_entrance must be called. If one of these t_entrance triggers adds another object (like moving the stairs or the closet door) this object will not be on the list and its t_entrance trigger will not be called. For the stairs this is not an issue, because its t_entrance doesn’t do anything, but for the closet door it is. We solved it by moving the closet door from the current location if the player is going to a location from where he must be able to refer to the closet door.
But, the living room also leads to the north hallway does not have a trigger to move the closet door to the north hallway? Right, but the only way you can go from the living room to the north hallway is when you came from the north hallway first. So the closet door will already be there.

floor

The floor is sort of a scenery object. We want the user to be able to refer to the floor, but is has all the default replies. We override the common t_entrance trigger with a local one that doesn't do anything, because we don't want the floor to be mentioned when entering the closet or when looking around.
When necessary, the carpet and the trapdoor will respond to "examine floor". The floor object will check whether carpet or trapdoor are visible and if not, it will make sure (through nomatch()) that the examine verb prints the default message.

```
$OBJECT o_floor
 DESCRIPTIONS
  d_sys "the floor"

 CONTAINED in l_closet

TRIGGERS
  "x [o_floor] " -> t_exa

  t_entrance  # don't call common t_entrance
   agree()

  t_exa
   if owns(l_closet, o_carpet) OR not(testflag(o_trapdoor.f_hidden)) then
    # do nothing, carpet and/or trapdoor will print a message
     agree()
   else
    # let verb print default message
    nomatch()

END_OBJ
```

Next are the carpet and the trapdoor.

Carpet

The carpet hides the trapdoor. The sides of the carpet are glued to the floor. To reveal the trapdoor, the player has to cut the sides of the carpet with the glass fragment. After cutting the carpet we don't want it to lay around, so we move it into the player's inventory.

```
$OBJECT o_carpet
 DESCRIPTIONS
  d_sys          "the old carpet"

  d_longdescr    "The carpet doesn't seem very expensive. It just /
                  about covers the floor. On a closer examination, it /
                  turns out that its sides are glued to the floor."
```

```
 d_shortdescr    "On the floor is an old carpet."

 d_cut           "You use the [o_fragment] to cut along the glued /
                  sides of the carpet. You grab the middle part /
                  of the carpet that now is no longer attached to /
                 the floor and lift it. Removing the carpet /
                  reveals a trapdoor in the floor!."

 d_no_move       "The carpet won't move. On closer examination /
                  you find that its edges are glued to the floor."

 d_exa_moved     "It's just an old carpet with the edges cut off /
                  by a sharp object."

CONTAINED on o_floor

FLAGS
  f_takeable     = 1
  f_moveable     = 1
  f_cut          = 0

 TRIGGERS
  "inventory"                          -> t_i
  "examine [o_carpet]"                 -> t_exa
  "examine [o_floor]"                  -> t_exa
  "lift [o_carpet]"                    -> t_move
  "take [o_carpet]"                    -> t_move
  "move [o_carpet]"                    -> t_move
  "cut [o_carpet] with [o_fragment]"   -> t_cut

 t_exa
  if not(testflag(f_cut)) then
    nomatch()
  else
    printcr(d_exa_moved)

 t_move
  if testflag(f_cut) then
    printcr("You already cut the carpet loose.")
  else
    printcr(d_no_move)

 t_cut
  if not(testflag(f_cut)) then
    printcr(d_cut)
    setflag(f_cut)
    clearflag(o_trapdoor.f_hidden)
    move(o_carpet, o_player)
  else
    printcr("You already cut the carpet.")
  endif
  disagree()
```

```
END_OBJ
```

trapdoor

When the player opens the trapdoor while the flames are not extinguished, we only allow him three more turns in the closet before it gets too hot. We define a timer m_heat that counts down and fires after three moves

timer m_heat

```
m_heat
 init          3
 step          1
 direction     down
 interval      1
 state         stop
 trigger_at    0
 execute       l_closet.t_leave
```

We must define a local trigger t_leave with the closet object.

Situations when the timer is started/stopped/updated:

- when the player enters the closet with trapdoor open and flames not extinguished: timer started;
- when the player is in the closet and opens the trapdoor and flames not extinguished: timer started;
- when the player leaves the closet: timer stopped and set to 3 in trigger t_exit;
- when the player is in the closet and closes the trapdoor: timer stopped and set to 3.

object trapdoor

```
$OBJECT o_trapdoor
 DESCRIPTIONS
  d_sys          "the trapdoor", “the trap door”

  d_longdescr    "The trapdoor is made of laminated wood. It seems large /
                  enough for a person to fit through.."

  d_shortdescr   "In the middle of the floor is a trapdoor, "

  d_open         "The trapdoor gives access to the cellar. Through the open /
                  trapdoor you see a stairway leading down."

 CONTAINED in l_closet

 FLAGS
  f_hidden       = 1
  f_openable     = 1
```

```
TRIGGERS
  "examine [o_trapdoor]"     -> t_exa
  "open [o_trapdoor]"        -> t_open
  "close [o_trapdoor]"       -> t_close

  t_entrance
    if not(testflag(f_hidden)) then
      print(d_shortdescr)
      setflag(f_seenbefore)
      if testflag(f_open) then
     printcr("which is open.")
        # player cannot see the flames
        if not(testflag(o_flames.f_extinguished)) then
          printcr(o_flames.d_flames)
          starttimer(m_heat)  # will count down to 0
        endif
      else
        printcr("which is closed.")

  t_exit
    if testflag(o_flames.f_extinguished) then
      # stop and reset the heat timer
      stoptimer(m_heat)
      m_heat = 3

t_open
    setflag(f_open)
print(d_open)
    if not(testflag(o_flames.f_extinguished)) then
      starttimer(m_heat)
      printcr(o_flames.d_flames)
    else
      printcr("")

  t_close
    if not(testflag(o_flames.f_extinguished)) then
      printcr("It's less hot now. This feels much better.")
      stoptimer(m_heat)
      m_heat = 3
    else
      printcr("closed.")
    endif
    clearflag(f_open)
END_OBJ
```

And we also need a trigger t_leave that we will code in location l_closet. Why in l_closet and not in the trapdoor? Well, both are possible, we chose l_closet because leaving seems like a location thing.

new version of l_closet

```
$LOCATION l_closet
 DESCRIPTIONS
  d_sys             "the closet"

  d_longdescr       "You are in a dark closet below the staircase. To the west is /
                     the closet door, which is closed."

  d_shortdescr      "Closet"

  d_leave           "\nThe heat is getting too much for you. You hurry back up to the /
                     stairs where it is much cooler."

 EXITS
  u -> l_halfway

 TRIGGERS
  "examine [l_closet]" -> o_player.t_look

  t_entrance
   printcr(d_shortdescr)
   printcr(d_longdescr)
   if not(testflag(o_trapdoor.f_hidden)) then
    printcr("Visible exits are up and down.")
   else
    printcr("The only visible exit is up.")
   endif

  t_leave
   # timer m_heat has fired
   stoptimer(m_heat)
   m_heat = 3
   printcr(d_leave)
   move(o_player, u)
   printcr("")
   printcrbold(l_halfway.d_shortdescr)

END_LOC
```

flames

```
$OBJECT o_flames
 DESCRIPTIONS
  d_sys          "the flames", "the fire"

  d_longdescr    "Because of the heat you cannot get close enough for
                 a good examination."

  d_shortdescr   ""  # flame entrance printed by the trapdoor

  d_flames       "A tremendous heat is coming through the open trapdoor. /
                 You look down and see a dark red glow deep down in /
                 the cellar."

  d_extinguish   "As soon as the water touches the flames, you hear a loud hissing /
                 sound, followed by the appearance of lots of steam. After a /
                 while, the hissing gets less until it completely stops. The fire /
                  has died.\nIt seems safe to go down into the cellar now."

 CONTAINED in l_cellar

 FLAGS
  f_extinguished = 0

 TRIGGERS
  "examine [o_flames]"   -> t_exa
  "extinguish [o_flames]"-> t_extinguish

  t_entrance
   agree()

  t_extinguish
   printcr("It's up to you to find a way how to do that.")

END_OBJ
```

tap

The tap is in the bedroom. The tap can be opened and closed. “Turn tap” checks the current position and then does the opposite.

When the following prerequisites have been fulfilled when opening the tap:

- trapdoor is open;
- drain pipe in closet is cut with the hacksaw;
- fire is not extinguished.

The fire in the cellar will be extinguished.

If the trapdoor is closed but the drain pipe has been cut, there will be water in the north hallway, pouring from under the closet door.

```
$OBJECT o_tap
 DESCRIPTIONS
  d_sys              "the tap"

  d_longdescr        "It's a tap for cold water."

  d_shortdescr       ""  # printed in t_entrance from sink.

  d_open             "As you turn the tap to open it, water starts /
                      pouring into the sink."

  d_extinguish       "After a little while, you faintly here a hissing /
                      sound, coming from somewhere below."

 CONTAINED in l_bedroom

 FLAGS
  f_openable = 1  # for open prologue

 TRIGGERS
  "examine [o_tap]"   -> t_exa
  "open [o_tap]"      -> t_open
  "close [o_tap]"     -> t_close
  "turn [o_tap]"      -> t_turn
  "turn on [o_tap]"   -> t_open
  "turn off [o_tap]"  -> t_close

  t_entrance
    # tap is handled by sink, because we want
    # to execute the sink t_entrance first
    agree()

  t_exa
    if testflag(f_open) then
      printcr("Water is pouring out of the tap into the sink.")
    else
      printcr("The tap is closed.")
    endif
    disagree()

  t_open
    if testflag(f_open) then
      printcr("The water is already running.")
    else
      setflag(f_open)
      clearflag(o_water_bedroom.f_hidden)
      if testflag(o_drain_pipe_closet.f_cut) then
        clearflag(o_water_closet.f_hidden)
        if not(testflag(o_trapdoor.f_open)) then
       # put water in the hallway north
          clearflag(o_water_hall_n.f_hidden)
        endif
```

```
    endif
    printcr(d_open)
    if not(testflag(o_flames.f_extinguished)) and
      testflag(o_trapdoor.f_open) and testflag(o_drain_pipe_closet.f_cut) then
      setflag(o_flames.f_extinguished)
      printcr(d_extinguish)
    endif
  endif

 t_close
  if not(testflag(f_open)) then
    printcr("It's already closed.")
  else
    clearflag(f_open)
    setflag(o_water_bedroom.f_hidden)
    setflag(o_water_closet.f_hidden)
    # water in hallway north remains
    printcr("The waterflow stops when you close the tap.")

 t_turn
  if testflag(f_open) then
    if not(trigger(t_close)) then
      disagree()
    endif
  else
    if not(trigger(t_open)) then
      disagree()
    endif
  endif

END_OBJ
```

sink object

The sink is there because we need the drain pipe. It's a scenery object.

```
$OBJECT o_sink
 DESCRIPTIONS
   d_sys              "the sink"

   d_longdescr        "The sink is connected to a drain pipe, which disappears /
                       into the floor."

   d_shortdescr       "There is a sink mounted to the wall. Above the sink /
                       is a tap."

 CONTAINED in l_bedroom

 TRIGGERS
   "examine [o_sink]" -> t_exa

END_OBJ
```

We're almost there. All we must do now is describe water objects to make the game more realistic. We want to allow the player to refer to the water when he opens the tap. There are three locations where the player can refer to the water: in the bedroom, in the closet and in the north hallway when the waters comes from under the closet door when the trapdoor is closed.

And of course, when we have water, we must also have a "drink" verb.

water in bedroom

When the tap is closed the water is hidden.

```
$OBJECT o_water_bedroom
 DESCRIPTIONS
  d_sys            "the water"

  d_longdescr   "Just plain ordinary water."

  d_shortdescr  "Water is running from the tap into the sink."

 CONTAINED in l_bedroom

 FLAGS
  f_hidden     = 1
  f_takeable   = 1

 TRIGGERS
  "examine [o_water_bedroom]"     -> t_exa
  "get [o_water_bedroom]"         -> t_get
"drink [o_water_bedroom]"         -> t_drink

t_get
    printcr("You have nothing with you that can hold the water.")

  t_drink
    printcr("That's refreshing! You didn't realize you were thirsty.")

END_OBJ
```

water in closet

```
$OBJECT o_water_closet
 DESCRIPTIONS
  d_sys            "the water"

  d_longdescr   "Just plain ordinary water."

  d_shortdescr  "" # printed by drain pipe

  d_no_drink    "It's better not to drink from the floor. if /
                  you are thirsty, better go to the tap in /
                  the bedroom for some fresh water."

 CONTAINED in l_closet

 FLAGS
  f_hidden      = 1
  f_takeable    = 1

 TRIGGERS
  "examine [o_water_closet]" -> t_exa
  "get [o_water_closet]"     -> t_get
  "drink [o_water_closet]"   -> t_drink

  t_entrance
   agree()   # handled by closet

  t_get
   printcr("You have nothing with you that can hold the water.")

  t_drink
   printcr(d_no_drink)

END_OBJ
```

water in hallway north

```
$OBJECT o_water_hall_n
 DESCRIPTIONS
  d_sys            "the water"

  d_longdescr      "Just plain ordinary water."

  d_shortdescr     "From underneath the closet door, water /
                    is coming into the hallway."

  d_no_drink       "It's better not to drink from the floor. if /
                    you are thirsty, better go to the tap in /
                    the bedroom for some fresh water."

 CONTAINED in l_hallway_north

 FLAGS
  f_hidden      = 1
  f_takeable    = 1

 TRIGGERS
  "examine [o_water_hall_n]"   -> t_exa
  "get [o_water_hall_n]"        -> t_get
  "drink [o_water_hall_n]"      -> t_drink
  t_get
   printcr("You have nothing with you that can hold the water.")

  t_drink
   printcr(d_no_drink)

END_OBJ
```

Verb drink

```
$VERB drink
"drink"
   printcr("What do you want to drink?")
   getsubject()

  "drink [o_subject]"
   printcr("[the] [o_actor] cannot drink [the] [o_subject].")

  DEFAULT
   printcr("I only understood you as far as wanting to drink something.")
ENDVERB
```

drain pipe in bedroom

The drain pipe in the bedroom is sort of scenery. It is used to help the player make the link between the drain pipe in the closet and the bedroom and to deduct that he should cut the pipe in the closet and turn on the water to extinguish the flames.

We have a rejection message in case the player tries to saw this drain pipe.

```
$OBJECT o_drain_pipe_bedroom
 DESCRIPTIONS
  d_sys             "the drain pipe"

  d_longdescr       "The drain pipe emerges from the sink and disappears in /
                     the floor."

  d_shortdescr      "Attached to the wall is a drain pipe."

  d_no_cut          "It makes little sense to cut the drain pipe here."

 CONTAINED in l_bedroom

 TRIGGERS
  "examine [o_drain_pipe_bedroom]"                  -> t_exa
  "cut [o_drain_pipe_bedroom] with [o_hacksaw]"     -> t_cut

  t_cut
   printcr(d_no_cut)

END_OBJ
```

drain pipe in closet

```
$OBJECT o_drain_pipe_closet
 DESCRIPTIONS
  d_sys                   "the drain pipe"

  d_longdescr             "The drainpipe comes down where the ceiling meets /
                           the west wall, goes vertically down the west /
                           wall and disappears in the floor."

  d_shortdescr            "Attached to the wall is a drain pipe."

  d_cut                   "About halfway up the wall, the pipe has been cut."

  d_cut_again             "You try to cut the pipe (again), but the hacksaw has /
                          become blunt after you used it the first time."

  d_pour                  "Water pours out of the upper half of the broken pipe /
                           on the floor, "

  d_pour_to_hallway       "where it disappears under the closet door into the hallway."

  d_pour_in_cellar        "through the open trapdoor straight into the cellar."

 CONTAINED in l_closet

 FLAGS
  f_cut = 0  # not yet cut.
```

```
TRIGGERS
 "examine [o_drain_pipe_closet]"                  -> t_exa
 "cut [o_drain_pipe_closet] with [o_hacksaw]"   -> t_cut

 t_entrance
  print(d_shortdescr)
  if testflag(f_cut) then
   print(d_cut))
   if testflag(o_tap.f_open) then
    print(d_pour)
    if testflag(o_trapdoor.f_open) then
     printcr(d_pour_in_cellar)
    else
     printcr(d_pour_to_hallway)
    endif
   endif
  else
   if testflag(o_tap.f_open) then
    printcr("You hear water running through the pipe.")

 t_exa
  if not(testflag(f_cut)) then
   printcr(d_longdescr)
  else
   printcr(d_cut)
    if testflag(o_tap.f_open) then
     print(d_pour)
     if testflag(o_trapdoor.f_open) then
      printcr(d_pour_in_cellar)
     else
      printcr(d_pour_to_hallway)
     endif
    endif
  endif

 t_cut
  if not(testflag(f_cut)) then
   setflag(f_cut)
   setflag(o_hacksaw.f_worn)
   printcr("You cut the pipe about halfway above the floor.")
   if testflag(o_tap.f_open) then
    print(d_pour)
    clearflag(o_water_closet.f_hidden)
    if not(testflag(o_trapdoor.f_open)) then
    printcr(d_pour_to_hallway)
    else
     printcr(d_pour_in_cellar)
     printcr(o_flames.d_extinguish)
     move(o_flames, l_storage)
    endif
   endif
  else
```

```
    printcr(d_cut_again)
  endif
END_OBJ
```

End of part 3

This ends part 3 of the tutorial. We now have a complete playable story. It's not the most exiting story, but the purpose of this tutorial is to show how to make an XVAN story, it's not a writing course.

Everything we've done until now is in files part3-end.lib and part3-end.xvn. To make a playable game file, run the compiler and enter part3-end.xvn as the story file name. Name the output file 'out.dat'. The output file may have any name, but if you want to use the Glk Interpreter, it must be called out.dat. The compiler will generate the output file that can be played using the interpreter. How to start the compiler and interpreter for different operating systems can be found in the XVAN installation and user guide.

In the remainder of this tutorial are two optional parts. Optional meaning that they are not necessary because we have a working story after part 3.

Part 4 goes into the look and feel. It changes background and text colors to white on blue and it makes use of the status window for the Glk version of the interpreter.

Part 5 demonstrates how to build some intelligence into verbs to parse ambiguous user input without asking the user for further clarification.

Part 4 – colors, fonts and status window

In this part of the tutorial we will look into text and background colors, fonts and the use of the status window in the Glk version of the interpreter.

Colors

By default the interpreter window shows white text on a black background. XVAN has built in functions to change text and background colors:

- background(<color>) sets the background color
- text(<color>) sets the text color.

Possible values for <color> are blue and black. Note that blue and black must be defined as words in the vocabulary file.

Setting the background color to either blue or white automatically sets the text color to white.

Setting the text color to either blue or white automatically sets the background color to black.

For our sample story we want white text on a blue background, so we change the player's t_init trigger as follows:

o_player.t_init

```
  t_init
background(blue)      # blue background with white text
   printcr(d_init)
   printcr("")
   entrance(owner(o_player))
```

Fonts (Glk interpreter)

XVAN has print functions that can print **boldface** and *italic* text:

- **printbold() and printcrbold()**
- *printitalic() and printcritalic()*

These functions work only in the GLK version of the interpreter. You do not have to make separate game source files for Glk and non-Glk. The non-Glk interpreter will handle these functions but will print normal text.

In our tutorial story, we want the location names to be printed in boldface. In the common t_entrance trigger and the various location t_entrance triggers we change the text "printcr(d_shortdescr)" to "printcrbold(d_shortdescr)". We won't copy it all here, you'll find it in the final story file part4-end.xvn.

Status window (Glk interpreter)

The Glk version of the interpreter has a three line status window on top of the game window. Following functions are available to manipulate the status window:

- clearstatus() clears all text from the status window;
- printstatus() prints text from current cursor position in status window;

- printcrstatus() same as above but adds a carriage return;
- setcursor() positions the cursor at the given position in the status window.

In the status window, we want to continuously display the player's location, number of moves and score. We create a trigger in the player object:

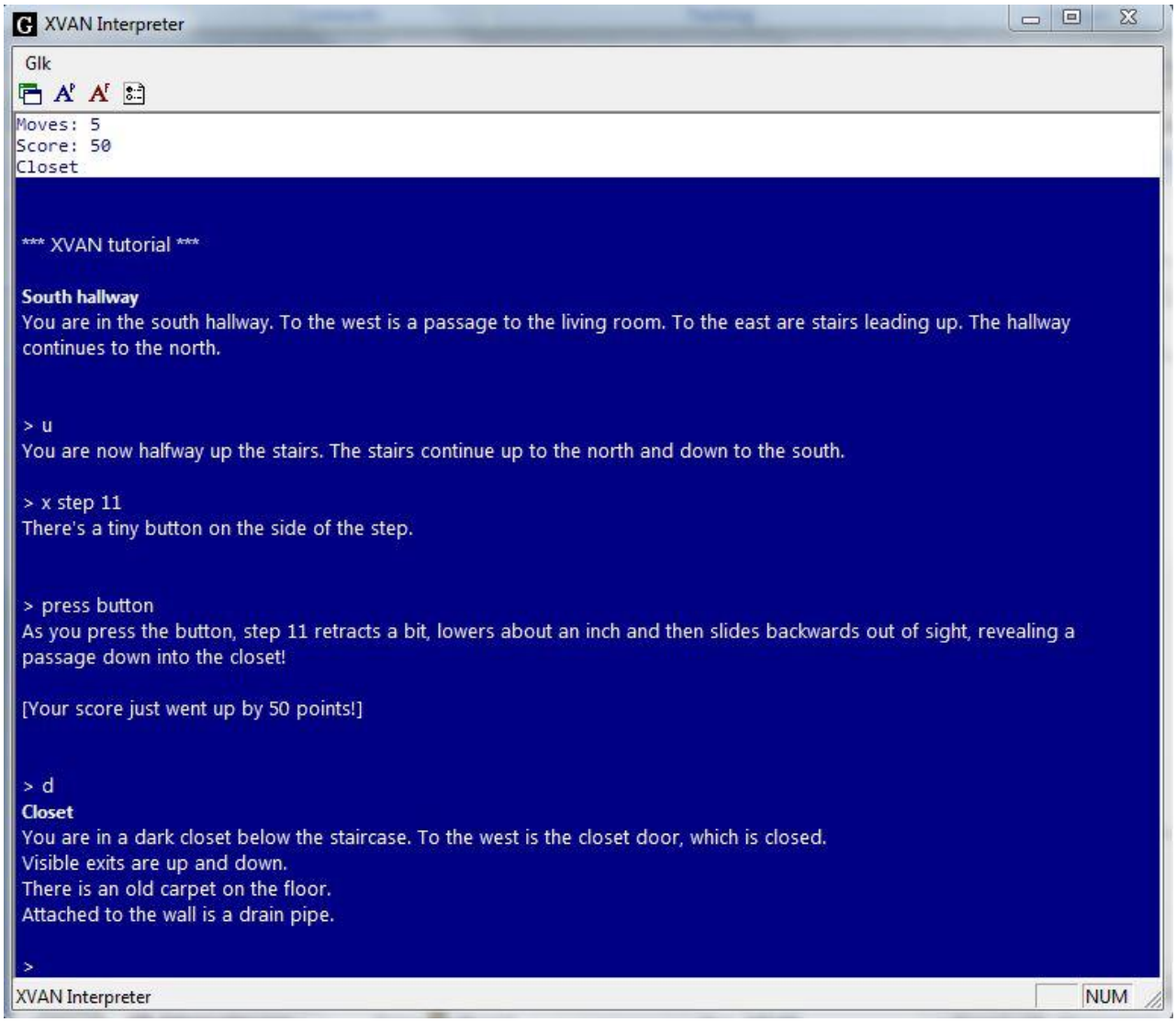
```
t_status_window
  clearstatus()
  # print the number of moves
  setcursor(0,0)
  printstatus("Moves: [m_init]")

# print the score
  setcursor(0,1)
  printstatus("Score: [r_score]")

  # print the location's name
  setcursor(0,2)
  if islit(l_location) then
printstatus(l_location.d_shortdescr)
  else
    printstatus("Darkness")
```

The non-Glk version of the interpreter will accept the commands but do nothing.

We want to refresh the status window at the end of every move, so we create the following timer:

```
m_status_window
  init                  0
  step                  0
  direction             up
  interval              1
  state                 go
  trigger_at            0
  execute               o_player.t_status_window
```

This is what the status window looks like (the white part above the blue window).



This is the end of part 4. Everything we did is in the files part4-end.xvn and part4-end.lib (the last file is identical to part3-end.lib).

Part 5 – disambiguation

First, a little background information on how XVAN parses user input. After the user enters a command string, the parser tries to translate it into an action, subjects, specifiers, etc. The parser looks for nouns and adjectives in the user input and compares them to the d_sys descriptions from objects and locations that are in scope.

It may occur that more than one object or location can be mapped to the user input. Let's take the following example: we now have two toasters in the kitchen, a red toaster and a blue toaster. The player enters the command "get toaster". In this situation, the parser has 2 objects that qualify as a subject: the red toaster and the blue toaster. The player has issued an ambiguous command. In such a situation, the parser needs more information from the player and would print a message like "Which toaster do you mean? The red toaster or the blue toaster?". The player would, for example, reply with "blue" and the parser would map the blue toaster as the subject. This works well.

Now consider the following. The player is in the kitchen. He carries the blue toaster and the red toaster is on the floor. The player enters "get toaster". Again, the parser will find two objects that qualify, the red toaster and the blue toaster, so it will ask the player which toaster he means.

But… considering the fact that the player is already carrying the blue toaster and that he issued a get command, it is very likely that he means to get the red toaster. The same goes for the "drop toaster" command: since he's only carrying the blue toaster it is safe to assume he wants to drop the blue toaster.

It is important to realize that the parser can only map nouns/adjectives to XVAN objects/locations. The parser has no knowledge about the context(the action) like the interpreter has. After the parser has finished the mapping, the interpreter will execute the trigger or verb default code with the objects/locations it got from the parser. So, to help the parser become better in resolving ambiguities, we must make it aware of the action.

From XVAN version 2.1 each section of default code can be extended with disambiguation rules. In case the parser finds more than one candidate when mapping the user input, it will check the disambiguation rules section for the verb syntax that it is parsing. The disambiguation rules give points to objects who – given the context - qualify better than other objects. So in our "get toaster" example, the red toaster would get points for not being carried and the blue toaster would not.

An example for the “get” verb:

```
$VERB get SYNONYM take SYNONYM grab
 PROLOGUE
  If equal(o_subject, %none) then
   agree()
  else
   if not(testflag(o_subject.f_takeable)) then
    printcr("[the] [o_subject] is not something that can be taken.")
    disagree()
  endif

 "get"
  printcr("What do you want to get?")
  getsubject()

 "get [o_subject]"
  DISAMBIGUATION_RULES
   If not(owns(o_actor, o_subject)) then score(5) endif
  END_RULES
  If not(owns(o_actor, o_subject)) then
   move(o_subject, o_actor)
   setflag(o_subject.f_bypass)
   printcr("[o_subject]: taken.")
  else
   printcr("But [o_actor] already [o_actor.r_have] [the] [o_subject].")

 DEFAULT
 printcr("I only understood you as far as wanting to get something.")
ENDVERB
```

The disambiguation rules will be evaluated for each possible subject. In our toaster example this will be the red toaster and the blue toaster. The red toaster will be awarded 5 points because it’s not held by the player. Therefore the red toaster ‘wins’. The score() function is the only function that can be used after the ‘THEN’ statement in the disambiguation rules section.

Keep in mind that disambiguation rules are only consulted in case multiple objects qualify for mapping user input. If the user is carrying the blue toaster and enters “get blue toaster”, the parser will map the subject to o_blue_toaster. There is no disambiguation here. When executing the command, the interpreter will find – when executing the “get [o_subject” default code – that the user is already carrying the blue toaster and will print the error message. So, disambiguation rules and trigger/verb default code operate at different levels.

No Such Thing

In part 3 we already saw o_nst, the no-such-thing object. As explained in the previous section, possible candidates get awarded points by the disambiguation rules. If all candidates end up with the same amount of points, the parser still has no clue and will fall back to asking the player which object he means.

All candidates start with zero points. In our toaster example, if both the red and blue toaster are on the floor and the player says "get toaster", both will end up with five points after applying the disambiguation rules and the parser will ask the player which toaster he means, which is what we want.

But if the player carries both toasters and he says "get toaster" both toasters will end up with 0 points and the parser will ask the player which toaster he means and then tell him he's already carrying it. Which is *not* what we want.

This is where the no-such-thing object comes in. o_nst is contained in the player object and it always participates in disambiguation. Where all objects start with 0 points in disambiguation, o_nst will have 1 point by default.

o_nst object

```
$OBJECT o_nst
 # this object must always be present
 # it may be modified but removal will cause a compiler error

 DESCRIPTIONS
  d_no     "no such thing"
  d_any    "any such thing"

 CONTAINED in o_player

 TRIGGERS
   t_entrance
     agree()

END_OBJ
```

In case all objects end up with 0 points, o_nst will win. The interpreter must check for this and can print an appropriate message.

verb "get" that considers o_nst

```
$VERB get SYNONYM take SYNONYM grab
 PROLOGUE
  If equal(o_subject, %none) OR equal(o_subject, o_nst) then
   agree()
  else
   if not(testflag(o_subject.f_takeable)) then
    printcr("[the] [o_subject] is not something that can be taken.")
    disagree()
  endif

 "get"
  printcr("What do you want to get?")
  getsubject()

 "get [o_subject]"
  DISAMBIGUATION_RULES
   if not(owns(o_actor, o_subject)) then score(5) endif
  END_RULES
  if equal(o_subject, o_nst) then
   printcr("There is [o_nst.d_no] to get.")
   disagree()
  endif
  If not(owns(o_actor, o_subject)) then
   move(o_subject, o_actor)
   setflag(o_subject.f_bypass)
   printcr("[o_subject]: taken.")
  else
   printcr("But [o_actor] already [o_actor.r_have] [the] [o_subject].")

 DEFAULT
  printcr("I only understood you as far as wanting to get something.")
ENDVERB
```

As an example, consider the toaster scenario where the player is holding both the blue and the red toaster and says "drop toaster".

```
Transcript from the tutorial game
XVAN transcript for: XVAN tutorial
version: 1.0

> l
Kitchen
This is the kitchen. There is not much here. The hallway is to the south.
There's a red toaster here.
There's a blue toaster here.
To the north is a door that leads to the garden.

> get toaster
Which toaster do you mean?
The red toaster or the blue toaster?

> blue
blue toaster: taken.

> l
Kitchen
This is the kitchen. There is not much here. The hallway is to the south.
There's a red toaster here.
To the north is a door that leads to the garden.

> get toaster
Red toaster: taken.

> i
You are carrying:
  a blue toaster
  a red toaster

> get toaster
There is no such thing to get.

> transcript
Turning off transcript mode.
```

Note: we used the transcript command described in section 2 to copy all screen output to file transcript.txt.

The first "get toaster" command was issued when both the red and the blue toaster were on the floor. Disambiguation rules did not help here (both toasters got 5 points an o_nst had 1) so the parser had to get back to the player for more info.

The second "get toaster" command was given when the player had the blue toaster and the red one was on the floor. Disambiguation rules gave 5 points to the red toaster, 0 to the blue and o_nst already had 1. So, the red toaster was the winner here.

The third "get toaster" command was given when the player held both toasters. So both the red and the blue toaster ended up with 0 points and o_nst had the 1 it started with and thus won. The verb code detected o_nst was the subject and printed the "There's no such thing to get." message.

Again, keep in mind that the disambiguation rules will only be consulted by the parser when t cannot map user input to a single object.

This is the end of part 5. Everything we did is in the files part5-end.xvn and part5-end.lib..

Part 6 – Play modes

As of version 2.4, XVAN supports choice and hybrid play modes.

By default, the XVAN interpreter works in interpreter mode: user input is entered as sentences via the command line. In choice and hybrid mode, at the beginning of each turn the user is presented with a numbered list of options to choose from. By entering a number, the corresponding option is selected. Hybrid mode is available for IFI-XVAN. It will create a 'quick command' list in the side bar that is updated every turn, while keyboard input is also active.

By default, XVAN operates in interpreter mode. Choice/hybrid mode can be activated in two ways:

- In the story info section from the main story file;
- During gameplay through the playmode() function.

One of the differences between choice and interpreter games is that choice is about finding out 'what' to do and that in an interpreter game one must find out 'how' to do it. Porting an interpreter game to a choice game without giving away clues is not always easy.

How does choice mode work? As an example, we will activate choice mode for our bedroom location. The list of possible choices is composed by the location and its contained objects. XVAN has a predefined trigger t_choice and a function addchoice() for this purpose.

Just before user input is required (after prologue, t_entrance etc), the interpreter checks the play mode. If it is not interpreter mode, it will tell the location and contained objects to execute their t_choice trigger. In t_choice an object can use the addchoice() function to add a possible choice to the list of choices.

E.g. addchoice("Get out", "west") will add the text "Get out" to the choice list. When this option is selected, the text "west" will be fed into the interpreter as user input.

Back to our bedroom location example. In the bedroom there are 3 contained objects:

- the sink (o_sink);
- the tap (o_tap);
- the water (o_water_bedroom);
- the drain pipe (o_drain_pipe_bedroom)

In choice mode, the bedroom, tap and water are likely to want to contribute to the choices list:
Bedroom: leave the room;

Tap: open or close the tap;
Water: drink it.

This gives us the following t_choice code:

```
$LOCATION l_bedroom
……
  t_choice
    addchoice(“Leave”, “east”)
……
END_LOC

$OBJECT  o_tap
……
  t_choice
    if testflag(f_open) then
      addchoice(“Close the tap”, “close tap”)
    else
      addchoice(“Open the tap”, “open tap”)
……
END_OBJ

$OBJECT o_water_bedroom
……
  t_choice
    if not(testflag(f_hidden)) then
      addchoice(“Drink from the water”, “drink water”)
……
END_OBJ
```

But when do we best switch on and off the choice mode?

Switching on choice mode:
Choice mode has to be switched on before the interpreter checks to see where it must get the user input from. Possible moments are in t_exit from l_upstairs or in t_entrance from l_bedroom. My choice would be t_entrance, because in t_exit from l_upstairs, the t_choice has to do an additional check to make sure we are actually going west to the bedroom and not downstairs again.

Switching off choice mode:
This can be done in t_exit from l_bedroom.

```
$LOCATION l_upstairs
……
  t_exit
    if equal(%dir, west) then
      playmode(choice)
……
END_LOC

$LOCATION l_bedroom
……
  t_entrance  # replaces code in t_exit for l_upstairs
    …..
    playmode(choice)
    …..

  t_exit
    playmode(interpreter)
……
END_LOC
```

Finally

This ends the tutorial. Everything we did is in the files part5-end.xvn and part5-end.lib. I hope the tutorial gave enough information to start implementing your own game. The XVAN distribution comes with a number of sample stories that can serve as examples.

For comments or questions, contact met at marnix@xvan.nl.

Thanks for reading.