HERITAGE Game Creation Documentation

Release 1

SylvieLorxu

May 13, 2015

This documentation will guide you to creation of Text Adventure games in the HERITAGE game format.

HERITAGE is available on https://notabug.org/SylvieLorxu/HERITAGE.

Contents:

CHAPTER 1

Introduction

HERITAGE stands for *HERITAGE Equals Retro Interpreting Text Adventure Game Engine*.

It is both an engine (implemented in Javascript) and a text adventure game file format, readable by any application implementing the HERITAGE standard.

The HERITAGE file format is characterized by being simple to learn and write, yet being powerful for even more complicated text adventure games.

CHAPTER 2

Tutorial: HERITAGE Game creation

This page will guide you to creating a HERITAGE game. After completing this guide, you will have a basic game, which you can then tweak to your liking with the knowledge gained from following this guide.

To start with following this documentation, create a text file named *main.heritage* using your favourite text editor.

Please avoid using office applications such as LibreOffice, as these by default do not save plain text. They often can, but save yourself the headache and use a text editor like gedit.

When choosing a text editor, please note that HERITAGE game engines are only required to support UNIX line endings. Therefore, it is good to make sure your text editor of choice supports UNIX line endings.

Please also note that, although this guide will indent code, the indentation is not required to be a valid HERITAGE game file. We feel, however, that it greatly improves readability and want to promote it as a best practice.

Okay, you have main.heritage file open in your favourite text editor? Time to start!

2.1 Writing down information

So, you're writing a cool new text adventure game in HERITAGE. Obviously, you want people to know you made it. HERITAGE info statements can contain any information you want, but you are recommended to add at least an author name, the name of the game, and some license information. We do this by adding the following lines to your game:

```
info():
    title: Your game name
    author: Your name
    license: Your preferred license
    version: A version identifier
```

Choosing a license is often hard. We recommend you to choose either the Creative Commons Zero 1.0 or the GNU GPLv3+ license. Whatever you choose, make sure to choose a GPL-compatible license, as the main parser is GPL-licensed. Licensing, however, falls out of the scope of this documentation.

So, let's say your name is John Doe, the game you're creating is named "Super Cool Text Adventure Game" and the license is Creative Commons Zero 1.0. Your info statement will look like this:

```
info():
    title: Super Cool Text Adventure Game
    author: John Doe
    license: Creative Commons Zero 1.0
    version: 1
```

You can add as many additional fields to info() as you want, but it's up to engine in question how (and if) these are displayed.

2.2 Creating a room

HERITAGE games require at least one room to be present. Rooms are stored on a basic grid, and have an X, Y and Z coordinate.

Each game starts off in a room with the X, Y and Z coordinate 0.

So, let's say we want the player to start in a grassy field, where they can go north, with a tree. This would look as follows:

```
room(0.0.0):
    description: You are in a grassy field, surrounded by trees.
    items: tree
    exits: north
```

That's it, your first room, with X.Y.Z coordinates 0.0.0!

You may wonder how the game knows what happens when the player types `go north`. The answer is simple: north is one Y position up from the current location, so the player will be moved to room 0.1.0 (the Y coordinate incremented by 1). HERITAGE understands the following exits:

- east: Increments the X coordinate by 1 (1.0.0)
- west: Decreases the X coordinate by 1 (-1.0.0)
- north: Increments the Y coordinate by 1 (0.1.0)
- south: Decreases the Y coordinate by 1 (0.-1.0)
- up: Increments the Z coordinate by 1 (0.0.1)
- down: Decreases the Z coordinate by 1 (0.0.-1)
- northeast: Increments the X and Y coordinate by 1 (1.1.0)
- northwest: Decreases the X coordinate by 1, increments the Y coordinate by 1 (-1.1.0)
- southeast: Increments the X coordinate by 1, decreases the Y coordinate by 1 (1.-1.0)
- southwest: Decreases the X and Y coordinate by 1 (-1.-1.0)

It is possible to have more flexible exits than this. They can be called with more than one name, they can only work in certain cases, they can trigger specific actions, they can move to another location, or they can have another name than described above. This will be talked about in more detail later. For now, we are going to concentrate on the tree.

2.3 Creating an item

So, we added a tree to the room, but how do we interact with it? Simple, we write a statement for the item.

Say, we want the tree to look old and describe the leaves when the player shakes the tree. We do that as follows:

```
item(tree):
    on_examine: The tree looks rather old.
    on_shake: The leaves move slightly as you shake the tree...
```

We have succesfully made some statements to be shown on the screen when the player types `look at tree` or `shake tree` when in the room the tree is at.

For any other actions, such as `cut down tree`, HERITAGE will display the text `I do not know how to cut down tree`. We recommend you to always define `on_examine`, because it sounds rather silly if the game tells you it does not know how to look at tree.

If you want to allow players to take the item, make sure to put the following in the statement: `allow_take: true`. They can then take the item by typing `take itemname`, in this case, `take tree`.

2.4 Looking back: A very basic game

If you have been following the guide, you now have a very basic game, looking like this:

```
info():
    title: Super Cool Text Adventure Game
    author: John Doe
    license: Creative Commons Zero 1.0
    version: 1

room(0.0.0):
    description: You are in a grassy field, surrounded by trees.
    items: tree
    exits: north

item(tree):
    on_examine: The tree looks rather old.
    on_shake: The leaves move slightly as you shake the tree...
```

Obviously, this game is not yet complete.

It's time to get into more advanced functionality of the HERITAGE format. We will be adding a key to the first room, add another room with a closed door to go north.

For the key, we already know all we need to know, we add a statement for it:

```
item(key):
    on_examine: It's a key, it probably opens something.
    allow_take: true
```

For the room, however...

2.5 Inline Conditions

So, we want to add a key to the grassy field the player starts in, but only display text for it when the key is still on the ground, but not when the player has picked up the key. We will change the statement for room(0.0.0) to this:

```
room(0.0.0):
    description: You are in a grassy field, surrounded by trees. $(require_here:key;A key lies on the
    items: tree, key
    exits: north
```

This probably looks weird, and may take a bit getting used to, but it is one of the most powerful features HERITAGE has to offer. They are called "Inline Conditions".

An inline condition allows you to specify things which are only to be displayed or done in certain cases. In this case, we require the item "key" to be "here" (meaning: in this room), to display the text "A key lies on the ground".

The following types of inline conditions exist:

- require_location: Requires a (list of) items to be in a specific location. The first value in the list defines the location. Example: `$(require_location:0.1.2,goblin;This text is only displayed if there is a goblin in a room with X.Y.Z coordinates 0.1.2)$`

- require_here: Requires a (list of) items to be in the current room. Example: `$(require_here:key;This text in only displayed in there is a key in the current room)$`
- require_inventory: Requires a (list of) items to be in the user's inventory. Example: `$(require_inventory:pen,paper;This text is only displayed if the items pen and paper are in the user's inventory)$`
- equals: Requires a variable to equal a certain value. We will revisit variables later. Example: `$(equals:test,1;This text is only displayed if the variable test equals one)$`
- less_than: Requires a variable to be less than a certain value. Example: `$(less_than:test,3;This text is only displayed if the variable test is less than three)$`
- more_than: Requires a variable to be more than a certain value. Example: `$(more_than:test,2;This text is only displayed if the variable text is more than two)$`

2.6 Special exits

So, let's create a castle, with a door on the north in front of us. This will be a bit special, because we want the door to be locked normally:

```
room(0.1.0):
    description: You stand in front of a castle, a grassy field to your south, a door to the north.
    exits: south, north|castle.1

exit(1):
    fail: The door is locked.
    equals: door_opened,1
```

There are two meaningful things here, the exit list in the room, and the exit statement. Let's focus on them one at a time.

`north|castle.1` is rather special, and it means the following:

- There is an exit north, which is special exit number 1
- castle is a synonym for north, with special exit number 1

It is important to know that `castle` is the synonym, not `north`. This allows the player to type `go castle`, and it will move the player north, just like they would if they were to type `go north`.

However, because north is a special exit, HERITAGE will first make sure the conditions set for that exit (by `exit(1)`) are all satisfied. If they are not, HERITAGE will display the text "The door is locked.". Otherwise, the player will be moved north.

2.7 Variable Management and Actions

As you can see, we want to make sure the variable `door_opened` equals `1`. That is all nice and dandy, but this means that we will need to learn variable management.

Variable management consists of two parts:

- Variable creation
- Inline expressions

First, we will create a variable:

```
var(door_opened,0)
```

That's all we need to do to create a variable named `door_opened` with the value `0`. HERITAGE supports number, names of other variables, or strings. Do note to put strings between "double quotes".

Now, let's create an action to allow players to open the door. This action will required the key to be in the user's inventory, will make us lose it (we admit, losing the key makes no sense, but we don't want to clutter the user's inventory) and set the variable door_opened to 1, so that we can enter the castle:

```
action(open_door):
    succeed: The door opens.#(door_opened=1)#
    require_inventory: key
    lose: key
```

So, if the player types `open door`, and they have a key in their inventory, they will lose it, the text "The door opens" will be displayed on the screen, and the variable door_opened will be set to one.

Instead of =, we could have also used + or − to increment or decrease the value on the variable door_opened.

Another interesting thing to notice is `lose`, this makes the user lose an item.

The following item management statements are supported:

- lose: causes an item to disappear from the player inventory
- gain: puts an item into the player inventory
- drop: makes the player drop the item in the current room
- disappear: causes the item to disappear from the current room

2.8 Putting it together

We have learned a lot, and have created a fairly sophisticated game already. So far, this is our code:

```
info():
    author: John Doe
    title: Super Cool Text Adventure Game
    license: Creative Commons Zero 1.0
    version: 1

room(0.0.0):
    description: You are in a grassy field, surrounded by trees. $(require_here:key;A key lies on the
    items: tree, key
    exits: north

item(tree):
    on_examine: The tree looks rather old.
    on_shake: The leaves move slightly as you shake the tree...

item(key):
    on_examine: It's a key, it probably opens something.
    allow_take: true

room(0.1.0):
    description: You stand in front of a castle, a grassy field to your south, a door to the north.
    exits: south, north|castle.1

exit(1):
    fail: The door is locked.
    equals: door_opened,1

action(open_door):
    succeed: The door opens.#(door_opened=1)#
    require_inventory: key
    lose: key

var(door_opened,0)
```

We can move the statements around as we wish, sort it in the way that makes it easiest for us to maintain the game. Now, let's finish the game.

2.9 Finishing the game

So, the player has played our amazing game, and solved all our cool challenges, and now we want to end our game.

To be more exact, we want the player to be eaten by a grue as they enter the castle. For that, we do create a new room for them to enter through the door we previously created:

```
room(0.2.0):
    first_enter: You are eaten by a grue. #(_game_over=1)#
```

first_enter is a special statement, displayed only when the player first enters the room. We could have just used description here as well, as the user won't be able to enter the room again.

By setting _game_over to 1, we tell HERITAGE that the game is over. It locks all input and ends the game, forcing the player to load a new game or restart.

2.10 Our complete game

That's it, we have created a complete game in HERITAGE, and it looks like this:

```
info():
    title: Super Cool Text Adventure Game
    author: John Doe
    license: Creative Commons Zero 1.0
    version: 1

room(0.0.0):
    description: You are in a grassy field, surrounded by trees. $(require_here:key;A key lies on the
    items: tree, key
    exits: north

item(tree):
    on_examine: The tree looks rather old.
    on_shake: The leaves move slightly as you shake the tree...

item(key):
    on_examine: It's a key, it probably opens something.
    allow_take: true

room(0.1.0):
    description: You stand in front of a castle, a grassy field to your south, a door to the north.
    exits: south, north|castle.1
```

```
exit(1):
    fail: The door is locked.
    equals: door_opened,1

action(open_door):
    succeed: The door opens.#(door_opened=1)#
    require_inventory: key
    lose: key

var(door_opened,0)

room(0.2.0):
    first_enter: You are eaten by a grue. #(_game_over=1)#
```

There are some features we have not documented in this guide yet, these will be documented later. For now, please look at the example directory shipped with the engine, it contains an example game.

All that rests now is to upload your game somewhere, so people can play it. For example, if you upload your game to `http://example.com/mygame/main.heritage`, it can be loaded by typing `load http://example.com/mygame`. Don't forget to name it main.heritage!

Have fun, and create something cool!

CHAPTER 3

HERITAGE statements

3.1 action

Syntax:

```
action(name):
    succeed:
```

Example:

```
action(open_door):
    succeed: You open the door
    fail: You don't have a key
    require_inventory: key
```

action allows the developer to create specific commands that are not tied to a specific item and can always be executed (this specific example is better used as an on_open substatement on a door item).

name is the command the user will type. If it is more than one word, it should be seperated by an underscore.

action supports the following substatements:

```
succeed: statement to print when the action succeeds
fail: statement to print when the action fails
require_location: locationnumber(X.Y.Z.) followed by a comma-seperated list of items that should be
require_here: comma-seperated list of items that should be on the player's current location
require_inventory: comma-seperated list of items that should be in the player's inventory
equals: check if variable is equal to value (variable, value)
less_than: check if variable is less than a certain value (variable, value)
more_than: check if variable is more than a certain value (variable, value)
lose: comma-seperated list of items in inventory to disappear
gain: comma-seperated list of items to be added to the inventory
drop: comma-seperated list of items to be moved from the inventory to the current room
```

3.2 exit

Syntax:

```
exit(id):
```

Example:

```
exit(1):
    fail: The door is locked
    require_inventory: key
```

exit allows the developer to create special exits. These are exits that do not use a standard direction (north, east, south, west, northeast, southeast, northwest, southwest, up and down) or exits on a standard direction that have special requirements. id is an unique number. The same id should not be re-used for multiple exits.

exit supports the following substatements:

```
succeed: statement to print when the action succeeds
fail: statement to print when the action fails
require_location: locationnumber(X.Y.Z.) followed by a comma-seperated list of items that should be
require_here: comma-seperated list of items that should be on the player's current location
require_inventory: comma-seperated list of items that should be in the player's inventory
equals: check if variable is equal to value (variable, value)
less_than: check if variable is less than a certain value (variable, value)
more_than: check if variable is more than a certain value (variable, value)
lose: comma-seperated list of items in inventory to disappear
gain: comma-seperated list of items to be added to the inventory
drop: comma-seperated list of items to be moved from the inventory to the current room
new_location: X.Y.Z value of the room this exit should take you to
```

3.3 info

Syntax:

```
info():
    title:
    author:
    license:
```

Example:

```
info():
    title: Your game name
    author: Your name
    license: Your preferred license
```

info allows the developer to store game-specific information such as title, author or license.

The title, author and license fields are required. The developer may add as many additional fields as they want, but it is up to the HERITAGE implementation if these are shown and how these are shown.

3.4 import

Syntax:

```
import(filename)
```

Example:

```
import(items)
```

import allows the developer to import other HERITAGE files in the current directory or a subdirectory. This is mostly to ease organising game data, but can also be used to import “libraries” adding a set of standard items. Do note that the statements in the imported file are inserted on the location the import statement is. If you want to overwrite imported items, make sure to do so AFTER the import statement.

For security and privacy reasons, only files in the current directory or a subdirectory may be imported.

3.5 item

Syntax:

```
item(name.instance):
```

Example:

```
item(dark_coat.2):
    on_examine: This is a dark coat.
```

name is the name of the item to create, instance is an name-unique number. itemname is required, and multi-word items should have the words separated by an underscore. instance is not required, unless you want to have multiple items with the same name; item(dark_coat) is valid syntax. An item and instance combination should only be defined once.

on_handler fields define what to do when an user types a command. In the example, the text “This is a dark coat” would be printed if the player types “look at dark coat”.

3.6 room

Syntax:

```
room(x.y.z):
```

Example:

```
room(0.0.0):
  first_enter: This text is returned the first time someone enters this room
  description: Description text, shown on entering the room or looking
  items: pen, computer.2
  exits: portal.3, east
```

room allows the user to create a room on a 3 dimensional grid. The player always starts at the room with X, Y and Z positions 0 (room(0.0.0)).

Each x.y.z combination should only be defined once.

room accepts the following substatements:

```
first_enter: The text to print upon entering this room for the first time
description: Text shown upon entering the room (also first time if first_enter is not defined) or loo
items: comma-seperated list of items in the room
exits: comma-seperated list of exits in the room
```

3.7 var

Syntax:

```
var(name, value)
```

Example:

```
var(points, 0)
```

var initializes a variable with the given value. This can be either an integer (0, 2, 14, etc.), the name of another variable that has been defined before, or a string (do note to use “double quotes”).

CHAPTER 4

HERITAGE functions

4.1 Calculate

Syntax:

```
@(variable operator value)@
```

Returns:

```
Result of calculation
```

Example:

```
@(points+10)@
```

Calculates the new value and returns it. This action is non-destructive (the actual value of the variable is not changed).

Accepted operators: + - / * %

4.2 Change

Syntax:

```
#(variable operator value)#
```

Returns:

```
Nothing
```

Example:

```
#(points+10)#
```

Calculates the new value and sets the variable to match this value. This action is destructive.

Accepted operators: = + - / * %

4.3 Comment

Syntax:

```
/* Text to comment out */
```

Returns:

```
Nothing
```

Example:

```
/* This text is commented out and
   will never be shown in any way */
```

Prevents text from being displayed in the interface, or interpreted. This is the only way to add text to a .heritage file without it meaning anything.

Comments can be spread over multiple lines.

4.4 Conditional

Syntax:

```
$(condition:variable,value;if_true|if_false)$
$(condition:list,of,variables;if_true|if_false)$
```

Returns:

```
if_true on true; if_false on false
```

Example:

```
$(!equals:_turn,5;Print this on any but the fifth turn)$
$(require_here:key,paper,note;Print this if the current location contains the items key, paper and no
$(equals:_yesno,1;Print on random true|Print on random false)$
```

Checks if a condition returns true, and returns the result if it does.

If you need to use | in your sentence, enter it twice: ||

Accepted conditions: require_location, require_here, require_inventory, equals, less_than, more_than

4.5 Echo

Syntax:

```
!(variable)!
```

Returns:

```
Value of variable
```

Example:

```
!(_turn)!
```

Prints the value of a variable.

CHAPTER 5

HERITAGE variables

5.1 _game_over

When read:

```
Returns 0
```

When written to:

```
If set to a non-zero value, ends the current game
```

5.2 _random

When read:

```
Returns random value from 1 through 100 (both inclusive)
```

When written to:

```
Discards input
```

5.3 _turn

When read:

```
Returns current game turn
```

When written to:

```
Discards input
```

5.4 _write_to

When read:

```
Returns the variable the next user input will write to, or 0 if no variable is defined
```

When written to:

```
Will save the next user input to the name of the variable given
```

5.5 _yesno

When read:

```
Returns either 0 or 1 (randomly chosen)
```

When written to:

```
Discards input
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search