DSL manual (v0.9)

November 2, 2013

This document contains a brief guide to development in the system of *DrdSim* and a summary of in-built and auxiliary functions of the system.

1 Creation of a new adventure

At first, you need to create a new folder in the main directory of the game. It's name has to start with `"DRD_"`. Otherwise, the system won't recognize it.

Next, create a description/header file int his folder. Its name is `"description.cfg"`. The syntax is very simple:

```
<attribute> = <string> | <number>
```

The following attributes can be set at this time:

TITLE	string	name of the adventure
DESCRIPTION	string	brief description
IMPORTPARTY	number (1/0)	is it possible to import a party?
NEWPARTY	number (1/0)	is it possible to create a new one?
STARTPARTYSIZE	number	maximal size of the party
ENTRYMAP	string	name fo the file with the starting map
ENTRYX	number	x-position on the starting map
ENTRYY	number	y-position on the starting map

As you can see, some of the values has the type *string*. It's possible to write it directly in place where it's used (in double quotes notation). However, if it's a text that's going to be read by a player, you should use a reference to localization file instead. You can do this by writing a dollar symbol and a name of the local string.

All the local strings has to be defined in `"stringtable.txt"`. It has a similar syntax as description files (name = string). You should create one in your adventure folder, another is already in the main one.

Afterwards, create five sub-folders: `maps`, `classes`, `scripts`, `sounds` and `music`. Notice that fodlers with the same names are in the main directory.

The next step is a creation of maps, which are stored in subdirectory `maps`. The editor for them is in directory `builder` and is started by the following command: `"java -jar mapBuild.jar"` (or simply `"runeditor.sh"` if you're in Linux). Help for this application is contained in the application itself. You can insert objects defined in the directory `classes` in maps.

You also need to define your own objects and store them in that directory and your own scripts (which are stored in the folder`scripts`). While defining classes, you need to remember that files are loaded in alphabetical order. Hence, if one class is derived from a class from another file, that file should be named in such a way that will ensure that it will be loaded first. If you define function *Init*, it will be called at the beginning of the adventure.

Folders `sounds` and `music` can contain your own audio files (dubbing for dialogues and music).

2 Programming in DSL

Files written in DSL language contain only definition of classes and functions. No other constructions are needed.

Definition of a class: Definition of a class starts with a header which consist of a name of the class and optionally a colon and a name of a mother class. Body of a class is enclosed in curly brackets and contains a set of attributes and their values (name, assign, value). The value can be a string, string-table reference or a number. Additionally, you can write another curly brackets instead of a value and write in a procedural code from the second part of the DSL language.

Formal summary:

```
<class name> [: <base class>]
"{"
   {<attribute name> = <attribute value> ;}
"}"
```

Example:

```
/// Base class for triggers.
Trigger: MapObject
{
  handler = {StdPrintLn("Base trigger triggered.");}
}

/** Base class for NPCs - encounter won't implicitly
 * start a fight but it can still go that way. */
NPC: Trigger
{
  npc = 1;
  handler = {StdPrintLn($GOODDAY);}
}
```

Note: For more examples, look at the game data.

Definition of a function: Definition of a function is started with a name of the function, normal brackets and a list of parameters. Then there are curly brackets and a body of the function.

Formal syntax:

```
<ID> "(" [ <ID> { , <ID> } ] ")"
<code block>

<code block> ::= "{"
                   { <statement> }
                 "}"

<statement> ::= <assignment> | <function call> | <return statement> |
              <if-clause> | <while cycle>

<assignment> ::= <l-value> = <expression> ;
<l-value> ::= [global] <ID> { "[" <expression> "]" }

<if-clause> ::= if "(" <expression> ")" <code block>
                { elif "(" <expression> ")" <code block> }
                [ else <code block> ]

<while cycle> ::= while "(" <expression> ")" <code block>
```

Example:

```
MyFunction(param1,param2)
{
  // body
}
```

Body of a function consists of statements, simple or composite. Simple statements are ended with semicolon.

Simple statement can be an assignment, function call or a return statement. Assignment is done by writing of a variable name, assign sign and expression. You can write keyword **global** in front of the variable name.

Function call is done by writing a name of the function and list of expressions in brackets. Return from a function is done by **return** keyword and optionally an expression. Expression consists of names of variables (this time without **global**), function calls, numbers, strings and arithmetic operators.

Example:

```
global var1 = 2*3;
var2 = "hello";
MyFunction(var1+1,var2+"world");
```

You can use composite commands **if** and **while** as shown in the example:

```
n = GetCount();
if (n==7) {
  // ...
}
elif (n>3) {
  // ...
}
else {
  // ...
}

i = 0;
while (i<n) {
  // ...
  i = i+1;
}
```

Work with arrays (/fields) is very simple. At first, you need to create an array. You'll do that by assigning empty brackets in a variable.

Then, you can assign inside the array. That is done in a standard way (see below). Just don't forget that you can index the items not only by numbers but by strings as well.

Example:

```
people = [];
people[2] = [];
people[2]["name"] = "John";
people[2]["surname"] = "Brown";
StdPrintLn(people[2]["name"]+" "+people[2]["surname"]);
```

Inbuilt functions The system has a number of inbuilt functions which enables the programmer to control the game. Apart from them, there are auxiliary functions written in DSL itself. Those functions are stored in the folder `scripts` in the root directory of the game.

In-built functions:

- **int Rand6()** Returns a number from 1 to 6 (included).
- **int Rand10()** Returns a number from 1 to 10 (included).
- **void StdPrintLn(string)** Writes a debug message on terminal.
- **string Nmr2Str(int)** Converts the number to a string.
- **int HasKey(field,int/string)** Does the array contains the given key?
- **int IsGlobalDefined(string)** Returns 1 if the global variable exists.
- **void UndefineGlobal(string)** Deletes the variable with the given name.
- **int GetType(any)** Detects the type of the parameter (0-int, 1-float, 2-string, 3-field).
- **void PlayMusic(string)** Plays a music from the given file (in a loop).
- **void StopMusic()** Turns the music off.

- **void PlaySound(string)** Plays the sound (once).
- **void StopSounds()** Stops any playing sounds.
- **void ShowDialog(string,field,string)** params: question, answers, name of the handler
  an answer: str|[str]|[str,int] (see example)
- **void ShowImageDialog()** first parameter is a name of a bmp file (string,string,field,string) (the image is then above the text)
- **void HideDialog()** Closes the currently displayed dialogue.
- **void StartCombat()** Goes into the combat mode with the current object.

Functions for controlling of AI pawns:

- **field ComputeTraverse(field,int)** The first parameter and the return value are indexed fields of two items representing x,y coordinates. The second parameter is the direction of the move. The function computes a step from the given position in the given direction.
- **Distance**
- **Direction**
- **SideCount**
- **GetMyId**
- **IsValid**
- **WhatIsIn**
- **GetPawnData**
- **Move**
- **Turn**
- **Attack**

Scripted functions:

- **void PrintNmr(int)** Prints a number on the terminal.
- **PlayDialog**

Auxiliary functions for AI scripts:

- **SpecifyAction**