# STEAD3

Peter Kosyh

August 27, 2022

# Table of contents

# Chapter 1

# General Information

The engine for INSTEAD is written in Lua language (5.1), therefore knowing this language is useful, though not necessary. The core engine also written in lua, so the Lua knowledge can be useful for in depth understanding principles of operation, of course, if you are interested this to do.

During its development, INSTEAD got loads of new functions. Now you can make games of different genres (from arcade, to games text). And also, INSTEAD you can run games written in some other engines, but the Foundation INSTEAD remains of the original core, which is focused on creating text and graphic adventure games. This documentation describes the this basic skills is necessary, even if you want to write something else… so let's Start learning INSTEAD by writing a simple game!

INSTEAD 3.0 was published in February of 2017 after 8 years of development. Version 3.0 supports a new runtime named STEAD3. The old runtime is now known as STEAD2. INSTEAD supports games written for either STEAD2 or STEAD3.

This manual will teach you everything you need to know about writing games for STEAD3. Other resources may be helpful to you such as the INSTEAD website:

https://instead-hub.github.io

We also have a chat room on gitter:

https://gitter.im/instead-hub/instead

## 1.1 The story of creation

When we say "text adventure" most of the people there one of the two familiar images. It's either text, action buttons, for example:

```
You see a table in front of you. There is an apple on the table. What to do?

1) Take the apple
2) Step away from the table
```

Or much less, this is a classic game with a text input where game control was necessary to introduce actions with the keyboard.

```
You in the kitchen. There is a table.
> inspect the table.
There is an apple on the table.
```

Both approaches have their advantages and disadvantages.

If we talk about the first approach, it is close to the genre of books, games and more convenient for literary texts that describe events, what is happening with the main character, and not very easy to create classic quests, where the main character explores modeled in-game world, moving freely on it and interacting with objects this world.

The second approach models the world, but requires considerable effort from the game author, and more importantly, more prepared player. Especially when we are dealing with the Russian language.

INSTEAD the project was created for writing other types of games combine the advantages of both approaches while trying to avoid their disadvantages.

The world of the game INSTEAD is modeled as the second approach, that is, in the game there are places (scenes or rooms) which can have access to the main the hero and the objects with which it interacts (including living characters). The player is freely exploring the world and manipulates objects. Moreover, actions with objects is not spelled out explicitly menu items, but rather reminiscent of the classic graphic quests in the style of the 90s.

Actually, INSTEAD there are many invisible at first glance things that are aimed at the development of the approach chosen, and which makes the game as dynamic and different from the usual "text quests". This is confirmed by including the fact that the engine was released a lot of great games, the interest show not only the fans of word games as such, but people do not familiar with this genre.

Before you read this guide, I recommend to play classic INSTEAD games to understand what was going on. On the other hand, since you're here, you probably did it.

However, not worth while to examine the code of these games, because old games are very often behave inefficiently, with outdated designs. Current version

INSTEAD allows you to implement code shorter, simpler and easier. About this and tells this document.

If you are interested in the history of the engine, then you can read an article about how it all began:

https://instead-hub.github.io/article/2010–05–09-history/

## 1.2 Look and feel of a typical INSTEAD game

So, What does a typical INSTEAD game look like?

*Main game window* contain a narrative, information about the static and dynamic parts of the scene, active events and a picture the scene (in the graphic interpreter) with possible transitions to other scene.

*Descriptive part of the scene* appears only once, when showing scene, or with explicit inspection of the scene (in the graphic interpreter – Staticheskaya part scene contains information about static objects scene (usually scenery) and is always displayed. This part written by the author of the game.

*Dynamic part of the scene* is composed of descriptions of objects in the scene, which are present in the moment on stage. This part is generated automatically and always displays. Usually, it presents objects that can change its location.

The player is available the features available on any stage – Inventory. The player can interact with objects of the inventory and to act the objects of the inventory on other objects in the scene or inventory.

> It should be noted that the "inventory" is a conventionalism. For example, equipment can be such objects as "open", "inspect", "use", etc.

*Actions* of the player can be:

- inspection of the scene;
- the effect on the object scene;
- the effect on the inventory object;
- the action object inventory on the inventory object;
- the unfolding of the object scene;
- switch to another scene.

## 1.3 How to start a new game project

INSTEAD will treat any directory on your computer as a game project if it contains a text file named "main3.lua". The presense of this file means that it is a STEAD3 project. Any other files you need for your game such as extra Lua scripts, images, and music should be stored within the game directory as well. Any time you reference an external resource in your code, it should be relative to the top-level game directory.

The "main3.lua" file should start with a comment block containing a list of tags. Tags are pairs of names and character strings that provide information about your game to INSTEAD.

Any sequence of characters starting with a double-dash on the same line is a Lua comment. INSTEAD tags are comment lines of the following form:

```
-- $TagName: A string of UTF-8 characters$
```

The $Name tag contains the name of the game. Here is an example:

```
-- $Name: the Most interesting game!$
```

It's good practice to follow the 'Name' tag with a few others. Here is how to specify the game's version number:

```
-- $Version: 0.5$
```

People who play your game may be interested in who wrote it:

```
-- $Author: Anonymous fan of text adventures$
```

It's oftentimes helpful to include a description about your game. Here we give an example of a multi-line tag. You specify line breaks using the "\n" escape sequence:

```
-- $Info: This is a remake of a classic\nZX Spectrum game.$
```

If you are a Windows user, make sure that your text editor can save files encoded as UTF–8 *without a BOM (byte order marker)*.

After the preamble containing your tags, you should list any external modules required by the game. Here is an example of what that might look like. We'll explain what these specific lines mean later:

```
require "fmt"   -- some formatting functions
fmt.para = true -- enable the paragraphs (indents)
```

After this, it's usually worthwhile to define default handlers. We haven't covered what handlers are yet, so it's fine if you don't know exactly what these lines do. Here is how you would define the default "act", "use", and "inv" handlers:

```
game.act = 'Not running.';
game.use = 'It does not help.';
game.inv = 'Why?';
```

When the game starts, it will set the initial game state by calling the "init" function. You can use this function to initialize the player or to perform any special starting tasks. The function may not be needed depending on your game.

```
function init() -- Put the knife and paper in the player's inventory
    take 'the knife'
    take 'paper'
end
```

The game engine only calls init() when a new game is started. It will not be called when you load a game that was previously saved. To perform a few actions whenever the game is loaded, use the start() function.

```
function start(is_loaded) -- to restore the original state?
    if is_loaded then
        dprint "Game loaded."
    else
        dprint "New game started."
    end
    -- we don't need to do anything
end
```

If you include both an init() function *and* a start() function, then them game engine will call init() first, and then call start().

The graphic interpreter looks for available games in the directory games. The Unix version of the interpreter in addition, the catalog scans also games in the directory ∼/.instead/games. Windows version: Documents and Settings/USER/

Local Settings/Application Data/instead/games. In Windows - standalone-Unix-version of the game are searched in the directory ./appdata/games, if it exists.

In some assemblies, INSTEAD (in Windows, in Linux if the project is built with gtk etc.), you can open the game in any way from the menu "Choice of games".Or, press f4. If in the directory with the game there is only one the game, INSTEAD it will be launched automatically, which is handy if you want distribute your game with the engine.

So you put the game in your directory and run INSTEAD.

**Important!**

When writing games, it is strongly recommended to use indentation for coding games, as in the example from the leadership, thereby you will reduce the number of errors and will make your code graphically!

Below is a minimal template for your first game:

```
-- $Name: My first game$
-- $Version: 0.1$
-- $Author: Anonymous$

require "fmt"
fmt.para = true

game.act = 'Hm...';
game.use = 'does Not work.';
game.inv = 'Why me?';

function init()
    -- initialization if it is needed
end
```

## 1.4 The basics of debugging

During debugging (check the health of your game) it is convenient to INSTEAD was started with-debug option, then in case of error shows more detailed information about the problem in the form of a stack calls. The-debug option can be set in the shortcut (if you are working in Windows) and for other systems, I think you already know how to pass command-line options.

In addition, the mode is debug, the debugger automatically connects. You can activate it with ctrl-d or f7. You can to connect the debugger and explicitly:

```
require "dbg"
```

In the code of your game.

When you debug a game, you usually need to frequently save the game and load the state of the game. You can use the standard mechanism of saving via menu (or via keys F2/F3), or use the quick saving/loading (press F8/F9).

Mode '-debug' you can restart the game with [ALT]+R. In combination with F8/F9 this enables you to quickly see changes to the game after it changes.

> Attention! INSTEAD's autosave feature is helpful when playing games, but may get in the way when you are in the middle of development. When restarting INSTEAD, the default behavior is to pick up where you left off by reloading the previous game state. You can disable this feature through the menu settings under "autosave". You can also explicitly reset the game whenever you edit its source code. While in debug mode (by starting with the -debug flag as described above), you can press [ALT]+R or select "start over" from the menu.

Mode '-debug' Windows version INSTEAD creates a console window (in Unix version, if you start INSTEAD from the console, the output will bedirected to it) which will be implemented by the error output. In addition using the function 'print ()', you can generate your messages with debug output. For example:

```
act = function(s)
    print ("Act is here! ");
    ...
end;
```

Don't be alarmed when you read all the guidance and start writing your game, you will most likely take a look at this example with a large enthusiasm.

You can also use the function dprint(), which sends the output in the debugger window, and you can see it when logged in mode debug.

```
act = function(s)
    dprint ("Act is here! ");
    ...
end;
```

During debugging, it is useful to examine the save file, which contain the state variables of the game. Not to search every time files saves, create a saves directory in the directory of your game ( the directory that contains main3.lua) and the game will persist saves. This mechanism will also be useful for transferring games to other computers.

It is possible (especially if you use Unix systems) you will like it the idea of checking the syntax of your script through the compiler "luac". In Windows it is also possible, you only need to install execute the lua file for Windows (http://luabinaries.sourceforge.net)/and use luac52.exe.

You can check the syntax and use INSTEAD, for this use the parameter -luac:

```
sdl-instead-debug-luac <path to the script.>
```

# Chapter 2

# Scene

Scene (or room) – is a unit game in which the player can examine all the objects in the scene and interact with them. For example, the scene can be a room in which the hero is. Or plot forest available for observation.

In any game should be a scene called "main". It will start and your game!

```
room {
    nam = 'main';
    disp = "Main room";
    dsc = [[You are in a large room.]];
}
```

The record means creation of an object (as almost all entities. objects) main room type (room). The object attribute stores the name nam room 'main', which you can access room from your code. Each object has its unique name. If you try to create two object with the same name, you will receive an error message.

To access the object by name, you can use the following entry:

```
dprint("Object: ", _'main')
```

Each object has *attributes* and *event handlers*. In this example has two attributes: nam and dsc. The attributes are separated the delimiter (in this example – a symbol, the semicolon ';').

Usually, the attributes can be text strings, functions-handlers and Boolean values. However, the attribute nam should always be a text string, if specified.

In fact, you may not specify the name when creating the object:

```
room {
    disp = ”Main room”;
    dsc = [[You are in a large room.]];
}
```

In this case, the engine itself will give the name of the object, and this name is a kind of number. Because you do not know this number, you can contact the object clearly. It is sometimes convenient to create unnamed objects, for example, for decoration. When the object is created, even if it is nameless, you unable to create variable object reference, for example:

```
myroom = room {
    disp = ”Closet”;
    dsc = [[You in the closet.]];
}
```

Myroom variable in this case becomes a synonym of object (link on the object itself).

```
dprint(”Object: ”, myroom)
```

You can stick to any one method or to apply both. For example, you can specify the name and the variable link:

```
main_room = room {
    nam = ’main’;
    disp = ”Main room”;
    dsc = [[You are in a large room.]];
}
```

It is important to understand that the engine is in any case working with object names, variables are just references-it’s just a way to simplify access to frequently used objects. So, for our first game, we owe to specify the attribute nam = ‘main’ to create the main room and we begin our adventure!

In our example, when displaying the scene, the title scene will be used the attribute ‘disp’. In fact, if we hadn’t asked it’s in the title we would see ‘nam’. But nam is not always convenient to do the title of the scene, especially if it is a string like ‘main’ or if it numeric ID that the engine assigned to the object automatically.

There is a more intuitive attribute ‘title’. If it is set, when the display room as the title will indicate it. title is used when the player is inside one of the room. In all other cases (when showing transitions in this room) use the ‘disp’ or ‘nam’.

```
mroom = room {
    nam = 'main';
    title = 'Start of adventure';
    disp = "Main room";
    dsc = [[You are in a large room.]];
}
```

Attribute 'dsc' is the description of the scene that is displayed once when entering the scene or when an explicit examination of the scene. There are no descriptions of objects present in the scene.

You can use the symbol ',' instead of ';' to separate attributes. For example:

```
room {
    nam = 'main',
    disp = 'Main room',
    dsc = 'You are in a large room.',
}
```

In this example, all attributes – a string. The string can be written in single or double quotation marks:

```
room {
    nam = 'main';
    disp = 'Main room';
    dsc = "You are in a large room.";
}
```

For long descriptions it is convenient to use the following:

```
dsc = [[ Very long description... ]];
```

The newlines are ignored. If you want in the output the description of the scene was attended by paragraphs – use the '^'symbol.

```
dsc = [[ First paragraph. ^^
The Second Paragraph.^^

Third paragraph.^
On a new line.]];
```

I recommend to always use [[ and ]] to ‘dsc’.

Let me remind you again that the name ‘nam’ of object and display it (in this the case of how the scene will look like for the player in the form of lettering at the top) you can (and often should!) share. For this there are attributes ‘disp’ and ‘title’. ‘title’ is only in the rooms and works as handle when the player is inside this room. In other cases, use the ‘disp’ (if any).

If ‘disp’ and ‘title’ is not specified, it is considered that the display equals name.

‘disp’ and ‘title’ can be set to false in this case display will not.

# Chapter 3

# Objects

*Objects* – this one scene interacting with the player.

```
obj {
    nam = 'table';
    dsc = 'In the {table}.';
    act = 'Hm... Just a table...';
};
```

Object name "nam" is used when it gets to your inventory. Though in our case, the table hardly gets there. If the object is defined 'disp', when it enters the inventory to display it will make use of this attribute. For example:

```
obj {
    nam = 'table';
    disp = 'table angle';
    dsc = 'In the {table}.';
    tak = 'I took the corner of the table';
    inv = 'I hold the corner of the table.';
};
```

Still, the table came to us in the inventory.

You can hide items in the inventory, if 'disp' the attribute will be 'false'.

'dsc' – description of the object. It will be displayed in the dynamic part the scene in the presence of the object in the scene. The braces displayed a fragment of the text to be a link in the window INSTEAD. If objects in the scene a lot, all descriptions are displayed one after the other, using the spacebar

'act' is an event handler that is called when the action user (action on object in the scene, usually – click the mouse on the the link). Its main task-the output (return) line of text which will become part of scene events, and state change to playing world.

# Chapter 4

# Add objects to the scene

There are several ways for you, the author, to populate a scene with objects.

First, when a room is created, you can assign a list containing the names of objects to the room's 'obj' attribute:

```
obj { -- an object with a name but without a variable
    nam = 'box';
    dsc = [[I see a {box} on the floor.]];
    act = [[Hard!]];
}

room {
    nam = 'main';
    disp = 'Big room';
    dsc = [[You are in a large room.]];
    obj = { 'box' };
};
```

Now, when rendering the scene, we will see the object "box" in a dynamic part.

Rather than referring to the box by its name, we could have used a variable reference. This is possible as long as the object is declared earlier in your source code:

```
apple = obj { -- object variable, but without a name
    dsc = [[there is {Apple}.]];
    act = [[Red!!]];
}
```

```
room {
    nam = 'main';
    disp = 'Big room';
    dsc = [[You are in a large room.]];
    obj = { apple };
};
```

As a syntactic convenience, you can assign room objects at the end in a ":with" block. This allows you to remove one level of indentation, and group objects at the end of a room definition:

```
room {
    nam = 'main';
    disp = 'Big room';
    dsc = [[You are in a large room.]];
}:with {
    'box',
}
```

You can even declare your objects directly within a room definition. Here is an example of how you might do that:

```
room {
    nam = 'main';
    disp = 'Big room';
    dsc = [[You are in a large room.]];
}:with {
    obj {
        nam = 'box';
        dsc = [[I see a {box} on the floor.]];
        act = [[Hard!]];
    }
};
```

This is useful for objects and scenery. But in this case, you are will be able to create objects with a variable link. Fortunately, for decorations don't have to.

If the room is placed a few objects, separate the links with commas, for example:

```
obj = { ’box’, apple };
```

You can insert line breaks for clarity, when objects many, for example:

```
obj = {
    ’table’,
    ’apple’,
    ’knife’,
};
```

Another way of placing objects is to call functions which the objects are placed in the required room. This will be discussed in further.

# Chapter 5

# Objects used as decoration

Objects which end up being moved around within different scenes or into and out of the player's inventory during play are typically given a name or assigned to a variable. Naming an object allows us to refer to and work with objects in our code wherever the player (or our code editor) happens to be.

On the other hand, many or most of our game world may consist of objects that serve no other purpose than decoration. These objects exist solely to enrich our environments with descriptive content for the player to experience.

Such objects can be numerous, and moreover, it's common for them to be duplicates of each other. A forested area might be populated by many instances of the same tree, city streets may contain the same light pole on every block. We use a different approach for creating objects such as these than we would for unique objects that the player can manipulate.

## 5.1 The same object in multiple rooms

Using our example of a forrested area, you can create a single object and place it in multiple rooms as follows:

```
obj {
    nam = 'tree';
    dsc = [[There is a {tree}.]];
    act = [[All of these trees look alike.]];
}

room {
```

```
    nam = 'Forest';
    obj = { 'tree' };
}

room {
    nam = 'Street';
    obj = { 'tree' };
}
```

## 5.2 Use tags instead of names

A decorative object may only appear in a single room. For objects like these, you can assign them a local name using the 'tag' attribute. By assigning a tag, you don't have to come up with a globally unique name. Strings assigned to the 'tag' attribute look like a name, but are preceeded by a '#' symbol. Even so, you refer to the object by its tag without a #:

```
obj {
    tag = '#flowers';
    dsc = [[there is {flowers}.]]
}
```

All objects have names whether you define one or not. In this example, the object tagged as "flowers" will be given an automatically generated name. Referring to an object by its tag is only possible within the current room. For example:

```
-- search the current room for the first object tagged as '#flowers'
dprint(_'#flowers')
```

One way of thinking about tags is that they are local names. For convenience, you can tag an object by assigning a tag to the 'nam' attribute. Remember that tags always start with a '#' character whereas names do not.

```
obj {
 nam = '#flowers';
 dsc = [[there is {flowers}.]]
}
```

By assigning a tag to the 'nam' attribute, the object's actual name will be automatically generated behind the scenes.

## 5.3 Attribute usage scene decor

Since the scenery does not change its location, it makes sense to do them part of the description of the scene and not dynamic. This is done attribute scene ‘decor’. decor is always shown and the main function-description of the scenery of the stage.

```
room {
    nam = ’House’;
    dsc = [[I am at home.]];
    decor = [[I see a lot of interesting things. For example, {#wall|wall}
    hanging {#picture|picture}.]];
}: with {
    obj {
        nam = ’#wall’;
        act = [[the Wall like a wall!]];
    };
    obj {
        nam = ’#pattern’;
        act = [[van Gogh?]];
    }
}
```

Here we see several techniques:

1. To as a related text describes the scenery;
2. As references are used design with a clear mission objects to which they belong {the name of the object|text};
3. As the names of objects are tags, not to think about them uniqueness;
4. The objects of the scenery in the scene has no attributes dsc, as their the role of decor.

Of course, you can combine all of the techniques among themselves any proportions.

# Chapter 6

# Objects associated with other objects

Objects can also contain the attribute ‘obj’ (or design ‘with’). In this case, when outputting the objects, INSTEAD be deployed lists consistently. This technique can be used to create container objects or just to link a few descriptions together. For example, put on the table an Apple.

```
obj {
    nam = ’Apple’;
    dsc = [[On the table is {Apple}.]];
    act = ’Take what?’;
};

obj {
    nam = ’table’;
    dsc = [[the room is a {table}.]];
    act = ’Hm... Just a table...’;
    obj = { ’Apple’ };
};

room {
    nam = ’House’;
    obj = { ’table’ };
}
```

Thus, in the scene description we’ll see descriptions of objects ‘table’ and ‘Ap-

ple', because 'Apple' is associated with a table object and the engine in the output of the object 'table' after 'dsc' will consistently "dsc" and all its nested objects.

Also, it should be noted that in terms of object 'table' (for example, moving it from room to room) we will automatically move and invested in an object 'Apple'.

Of course, this example could also be written differently, for example, so:

```
room {
    nam = 'House';
}: with {
    obj {
        nam = 'table';
        dsc = [[the room is a {table}.]];
        act = 'Hm... Just a table...';
    }: with {
        obj {
            nam = 'Apple';
            dsc = [[On the table is {Apple}.]];
            act = 'Take what?';
        };
    }
}
```

You can choose the way that's clearer for you.

# Chapter 7

# Defining attributes and event handlers with functions

Up until now we've assigned strings to most of the attributes of our objects and rooms. You can also assign *functions* to attributes. For example:

```
disp = function()
    p 'Apple';
end
```

That wasn't a very good example, but it does show what the syntax should look like. We may as well have written `disp = 'Apple'`. The main objective of such functions is to return a string or boolean value.

Let's take a look how to return something. One way to return a string is by an explicit statement such as the following:

```
return "Apple";
```

When a statement like this is encountered, the function will exit with the given return value. In this case, that would be the string, "Apple".

The most common way to output strings is by using the following three built-in functions:

- p ("text") – output text and white space;
- pn ("text") – output text with line break;
- pr ("text") – output text as-is.

These functions do not output text directly. The engine maintains a clipboard where text is stored. The contents of this clipboard is sent to the engine all at once when your function returns. This allows you to build up a block of text to return by calling p/pn/pr multiple times. In most cases, you shouldn't concern yourself with where line breaks will go or any other formatting issue. This is especially true of descriptions. The engine will wrap text and add spacing intelligently based on what you present to the player.

As with all functions, you can leave out the parentheses if there is only a single value to supply:

```
pn "No brackets!"
```

It's likely that you will often times want to concatenate strings together. You can either use '..' or ',' for this purpose, and in that case parentheses will be required:

```
pn ("String 1".." Line 2");
pn ("String 1" "String 2");
```

> The main difference between attributes and event handlers is that only event handlers are able to change the state of the game world. When writing an attribute such as 'dsc', remember that your goal is to return a value and not to affect what happens in the game! Forgetting this rule will lead to unpredictable behavior.

**Important!**

> There is something to keep in mind when writing event handlers as well. They should execute quickly. Event handlers are called in response to user action, and the engine expects them to return immediately so that it can refresh the user interface. If you want to insert delays to control the output that users see, you will have to use the "timer" module.

Functions almost always return dynamic results that are based on the contents of variables. For example:

```
obj {
    nam = 'Apple';
    seen = false;
    dsc = function(s)
        if not s.seen then
            p '{Something} is on the table.';
        else
            p 'An {Apple} lies on the table.';
        end
    end;
    act = function(s)
        if s.seen then
            p 'It's an Apple!';
        else
            s.seen = true;
            p 'Um... It's an Apple!';
        end
    end;
};
```

When a function is assigned to an attribute, its first parameter always refers to the object itself. We typically name this variable 's' for "self". You could also write out the object by name (_'Apple'), but when writing such functions, it's generally better to use the 's' parameter than to make an explicit call to the object name. This will save you from having to rewrite functions if you ever need to rename the object.

In this example, we make text descriptions dynamic. The first time the player encounters this object, it will be referred to as "Something". Once they interract with it, the 'seen' variable becomes true. After that, they will see that the object is an "Apple".

The syntax of 'if' statements is easy to read. Here are a few examples for clarity:

```
if <expression> then <actions> end

if have 'Apple' then
    p 'I have an apple!'
end
```

```
if <expression> then <actions> else <actions otherwise> end

if have 'Apple' then
    p 'I have an apple!'
else
    p 'I don't have any apples!'
end

if <expression> then <action> elseif <expression 2> then <action 2>
else <otherwise> end -- etc.

if have 'Apple' then
    p 'I have an apple!'
elseif have 'fork' then
    p 'I don't have any apples, but there is this fork in my inventory!'
else
    p 'I don't have an apple or a fork.
end
```

The *expression* of an "if" statement is a boolean value made up of true/false terms separated by "and", "or", "not", and parenthesis to control evaluation priority. Expressions containing a single variable (ie: `if` <`variable`>) simply check that the variable does not equal to `false`. The equality operator is '==', and the inequality operator is '~='.

```
if not have 'Apple' and not have 'fork' then
    p 'I don't have an apple or a fork!'
end

...
if w ~= apple then
    p 'This is not an Apple.';
end
...

if time() == 10 then
    p 'It's your 10th turn!'
end
```

**Important!**

You need to define variables before using them. If you use a variable in a condition expression without first defining it, INSTEAD will raise an error.

## 7.1 Object variables

The entry's.seen' means that the variable 'seen' is placed in the object 's' (i.e. 'Apple'). Remember, we have called the first parameter of the function 's' (self) as the first parameter – is the current object itself.

Object variables must be defined in advance if you are to modify them. Something like we did with the seen. But variables can be many.

```
obj {
    nam = 'Apple';
    seen = false;
    eaten = false;
    color = 'red';
    weight = 10;
    ...
};
```

All variables of an object when it changes, enter the save file game.

If you don't want the variable was in a save file, you you can declare such variables in a special block:

```
obj {
    nam = 'Apple';
    {
        t = 1; -- this variable will not get to save
        x = false; -- and this too
    }
};
```

Normally you don't need to do that. However, there is a situation in which this technique will be useful. The fact that arrays and tables object are always saved. If you use arrays to store immutable values, you can write:

```
obj {
    nam = 'Apple';
    {
        text = { "one", "two", "three" }; -- never go to a save file
    }
    ...
};
```

You can access the variables of an object via s-if it is himself object. or variable - reference, for example:

```
apple = obj {
    color = 'red';
}
...
-- somewhere in another place
    apple.color = 'green'
```

Or by name:

```
obj {
    nam = 'Apple';
    color = 'red';
}
...
-- somewhere in another place
    _'Apple'.color = 'green'
```

In fact, you can create variables of the object on the fly (without pre-define them), although usually it makes no sense. For example:

```
apple 'xxx' (10) -- create a variable xxx, the object apple on the link
(_'Apple') 'xxx' (10) -- same but in the name of the object
```

## 7.2 Local variables

In addition to object variables you can use local and global variables.
Local variables are created by using office word local:

```
act = function(s)
    local w = _'light bulb'
    w.light = true
    p [[I pressed the button and the bulb lit up.]]
end
```

In this example, the variable 'w' exists only in the body the handler function act. We created a temporary reference variable 'w', which refers to the object 'light' to change feature-variable light this object.

Of course, we could write:

```
_'light bulb'.light = true
```

But imagine if we need to execute multiple actions with a the object in such cases is easier to use a temporary variable.

Local variables are never displayed in the file-save and play the role of temporary auxiliary variables.

Local variables can be created outside of functions, then this the variable is visible only within a given lua file and misses a save file.

Another example of using local variables:

```
obj {
    nam = 'kitten';
    state = 1;
    act = function(s)
        s.state = s.state + 1
        if s.state > 3 then
            s.state = 1
        end
        p [[Purr!]]
    end;
    dsc = function(s)
        local dsc = {
            "The{kitten} purrs.",
            "The{kitten} is playing.",
            "The{kitten} is licked.",
        };
        p(dsc[s.state])
    end;
end
```

As you can see, in dsc, we determined the array dsc. ‘local’ indicates that it operates within the dsc. Of course, this example you could write it as:

```
dsc = function(s)
    if s.state == 1 then
        p ”the{Kitten} is purring.”
    elseif s.state == 2 then
        p ”the{Kitten} is playing.”
    else
        p ”the{Kitten} is licked.”,
    end
end
```

## 7.3 Global variables

You can also create a global variable:

```
global { -- definition of global variables
    global_var = 1; -- number
    some_number = 1.2; -- number some_string = ’string’;
    know_truth = false; -- a Boolean value
    array = {1, 2, 3, 4}; -- array
}
```

Another form, convenient for single definitions:

```
global ’global_var’ (1)
```

Global variables always get to the file-save.

In addition to global variables you can define constants. Syntax similar to global variables:

```
const {
    A = 1;
    B = 2;
}
const ’Aflag’ (false)
```

The engine will control the consistency of constants. The constants are not enter the file-save.

Sometimes you need to work with a variable that is not defined aslocal (and visible in all of your lua files), but should not get in the save file. For such variables you can use Declaration:

```
declare {
    A = 1;
    B = 2;
}
declare 'Z' (false)
```

The Declaration is not stored in the save file. One of the important properties declarations is that you can declare functions for example:

```
declare 'test' (function()
    p "Hello world!"
end)

global 'f' (test)
```

In such case, you can assign the value of the function 'test' others variables and the state of these variables may be stored in the filesave. That is, a declared function can be used as the value of the variable!

You can declare a previously defined function, for example:

```
declare 'dprint' (dprint)
```

Thereby making such undeclared functions – declared.

The function Declaration, in fact, is the assignment of the function name, thanks what can we retain this feature as a link.

## 7.4 Helper functions

You can write your helper functions and use them from your game, for example:

```
function mprint(n, ...)
    local a = {...}; -- temporary array with the arguments to the function
    p(a[n]) -- get the n-th element of the array
end
...
dsc = function(s)
    mprint(s.state {
        ”The{kitten} purrs.”,
        ”The{kitten} is playing.”,
        ”The{kitten} is licked.” });
end;
```

Don’t pay attention to this example, if it seems to you difficult.

## 7.5 The return value handlers

If you want to show that the action is not executed (handler did anything useful), return false. For example:

```
act = function(s)
    if broken_leg then
        return false
    end
    p [[I kicked the ball.]]
end
```

This displays the default description is specified using a handler ‘game.act’. Usually the default description contains description of the undoable action. Something like:

```
game.act = ’Hm... does Not work...’;
```

So, if you don’t set the handler act or returned from it false – it is believed that there is no reaction and the engine will perform the same handler from object ‘game’.

Usually, there is no sense to return false from the act, but there are other handlers, which will be discussed further, for which the described behaviour is exactly the same.

Actually, besides ‘game.act’ and ‘act’ – the object attribute exists handler ‘on-act’ of the game object, which can interrupt execution handler ‘act’.

Before calling the handler ‘act’ of an object is called onact have game. If the handler returns false, the execution of the ‘act’ dropped. ‘onact’ convenient to use for control events in the game, for example:

```
-- called onact rooms, if they are
-- for the actions on any object

game.onact = function(s, ...)
    local r, v = std.call(here(), 'onact', ...)
    if v == false then -- if false, chop off the chain
        return r, v
    end
    return
end

room {
    nam = 'shop';
    disp = 'Shop';
    onact = function(s, w)
        p [[In the store, you can not steal!]]
        p ([[Even if it's only a ]], w, '.')
        return false
    end;
    obj = { 'ice cream', 'bread' };
}
```

In this example, when trying to “touch” any item, will be display a message prohibiting this action.

All that is described above on the example of the ‘act’ applies for other handlers: tak, inv, use, and transitions, as will be is discussed later.

> Sometimes you need to call the handler function manually. It uses the syntax of a method call object. ‘Object:method(parameters)’. For example:

```
apple:act() -- call handler 'act' of object 'apple' (if it defined as a
function!). _'Apple':act() -- same, but name, not a variable reference
```

This method works only if the called method designed as a feature. You can use ‘std.call()’ for a handler is invoked in the way that it makes itself INSTEAD. (To be described in the future).

# Chapter 8

# Inventory

The easiest way to create an object that can be picked up is to assign a handler to the ‘tak’ attribute, which is short for “take”. For example:

```
obj {
    nam = ’Apple’;
    dsc = ’On the table is {Apple}.’;
    inv = function(s)
        p ’I ate the Apple.’
        remove(s); -- remove an Apple from the inventory
    end;
    tak = ’You took the Apple.’;
};
```

In this case, the player object is “Apple” (click on link in the scene) – the Apple is removed from the scene and added to inventory. When the player uses the inventory (double click on the name of the object) – call the handler ‘inv’.

In our example, when the player uses the Apple in the inventory – Apple be eaten.

Of course, we could implement the code takes object to “act”, for example:

```
obj {
    nam = ’Apple’;
    dsc = ’On the table is {Apple}.’;
    inv = function(s)
        p ’I ate the Apple.’
        remove(s); -- remove an Apple from the inventory
```

```
    end;
    act = function(s)
        take(s)
        p 'You took the Apple.';
    end
};
```

If the object in the inventory is not declared a handler for the 'inv', will be called 'game.inv'.

If the handler is 'tak' will return false, the item will not be taken, for example:

```
obj {
    nam = 'Apple';
    dsc = 'On the table is {Apple}.';
    tak = function(s)
        p "It is wormy!"
        return false
    end;
};
```

# Chapter 9

# Transitions

The traditional transitions into INSTEAD appear as links above the description scene. To determine the transitions between scenes is used attribute scene – list ‘way’. In the list are determined by the room in the form of of the room names or variable references, similar to the list ‘obj’. For example:

```
room {
    nam = ’room2’;
    disp = ’Hall’;
    dsc = ’You are in a huge hall.’;
    way = { ’main’ };
};

room {
    nam = ’main’;
    disp = ’Main room’;
    dsc = ’You are in a large room.’;
    way = { ’room2’ };
};
```

With this, you’ll be able to go between scenes ‘main’ and ‘room2’. As you remember, ‘disp’ can be a function, and you can generate names transitions on the fly. Or use title, to separate the name of the scene as the title and how the transition name:

```
room {
    nam = ’room2’;
```

```
    disp = 'hall';
    title = 'hall';
    dsc = 'You are in a huge hall.';
    way = { 'main' };
};

room {
    nam = 'main';
    title = 'the main room';
    disp = 'the main room';
    dsc = 'You are in a large room.';
    way = { 'room2' };
};
```

When passing between rooms the engine calls the handler for the 'onexit' of the current scene and 'the onenter' in that scene where is the player. For example:

```
room {
    the onenter = 'You enter the room.';
    nam = 'Hall';
    dsc = 'You are in a huge hall.';
    way = { 'main' };
    onexit = 'You exit the room.';
};
```

Of course, like all handlers, 'onexit' and 'the onenter' can be functions. Then the first parameter is (as always) the object itself - room and the second room is where the player is going to go (for 'onexit') or from which is going to leave (for 'the onenter'). For example:

```
room {
    onenter = function(s, f)
        if f^'main' then
            p 'You go from room to main.';
        end
    end;
    nam = 'Hall';
    dsc = 'You are in a huge hall.';
```

```
    way = { 'main' };
    onexit = function(s, t)
        if t^a'main' then
            p 'I don't want to go back!'
            return false
        end
    end;
};
```

Writing:

```
if f^'main' then
```

This mapping of the object name. This alternative records:

```
if f == _'main' then
```

Or:

```
if f.nam == 'main' then
```

Or:

```
if std.nameof(f) == 'main' then
```

As you can see, for example, onexit, these handlers other than line can return a Boolean status value. Similarly, the processor onact, we can cancel the transition by returning false from onexit/the onenter.

You can also return another way, if it seems you comfortable:

```
return "I don't want to go back", false
```

If you use the function 'p'/'pn'/'pr', then just return the status of a transaction with the final 'return', as shown in the example above.

**Important!**

> It should be noted that when calling the handler 'the onenter' pointer to current scene (here()) **not yet changed**!!! In there INSTEAD handlers 'exit' (leaving the room) and 'enter' (entering the room), which are called already *posle* how the transition happened. These handlers are recommended when there is no need to forbid the transition.

Sometimes there is a need to name the transformation differed from the name of the room in which this transition leads. There are sever always to do this. For example, using ‘path’.

```
room {
    nam = ’room2’;
    title = ’Hall’;
    dsc = ’You are in a huge hall.’;
    way = { path { ’main room’, ’main’} };
};

room {
    nam = ’main’;
    title = ’Main room’;
    dsc = ’You are in a large room.’;
    way = { path {’room’, ’room2’} };
};
```

Actually, ‘path’ creates a room with the attribute ‘disp’, which equal to the first parameter, and special feature ‘the onenter’, which redirects the player to the room specified by the second argument of ‘path’.

If you specify three parameters:

```
way = { path {’#hall’, ’room’, ’room2’} };
```

The first parameter will be the name (or tag, as in the example) for such a room.

Alternative form of entry with the explicit task attribute nam:

```
way = { path { nam = ’#hall’, ’room’, ’room2’} };
```

You can change the name of the transition, after the transition occurred at least once, and you know, what is this room:

```
way = { path {’#udvari’, ’door’, after = ’living room’, ’room2’} };
```

All parameters except the transition name, can be functions.

Thus, the ‘path’ allows you to call a transitions convenient way.

Sometimes you may need to turn on and off transitions. Reallyit is not often required. The idea of transitions is that the transition visible even when it's impossible. For example, imagine the scene in front of the house by the front door. To enter the house because the door closed.

It makes little sense to hide the transition. Just in the function of 'the onenter' the scene inside the house, we check whether a character key? And if the key no, talking about the fact that the door is closed, and prohibit the transition. It increases interactivity and simplifies the code. If you want to do the door object in the scene, place it in the room, but in the 'act' handlerd o the inspection doors, or allow the player to open it with a key (how to do it - we will look at later), but the transition itself give make the player in the usual way through the line transitions.

However, there are times when the transition is not obvious and it appears as a result of some events. For example, a clock and saw a secret tunnel behind it.

```
obj {
    nam = 'clock';
    dsc = [[you see an old {clock}.]];
    act = function(s)
        enable '#clock'
        p [[You see that the watch is a secret passage!]];
    end;
}

room {
    nam = 'Hall';
    dsc = 'You are in a huge hall.';
    obj = { 'clock' };
    way = { path { '#watch', 'watch', 'inclock' }:disable() };
};
```

In this example, we created *disabled* transition, by calling method 'disable' of the room created using the 'path'. Method 'disable' have all items (not only rooms), it translates the object in disabled state, which means that the object ceases to be available to the player. A remarkable property disabled facility is that it can *enabled* with 'enable()';

Further, when the player clicks on the link that describes the watch, called handler, 'act', using the function 'enable()' makes the transition visible.

The alternative is not in shutdown and 'close' object:

```
obj {
    nam = 'clock';
    dsc = [[you see an old {clock}.]];
    act = function(s)
        open '#clock'
        p [[You see that the watch is a secret passage!]];
    end;
}

room {
    nam = 'Hall';
    dsc = 'You are in a huge hall.';
    obj = { 'clock' };
    way = { path { '#watch', 'watch', 'inclock' }:close() };
};
```

What's the difference? Disabling an object means that the object ceases to be available to the player. If the object is nested other objects, and these objects become inaccessible. The closure of the facility makes unavailable the contents of this object, but not the object itself.

However, in the case of rooms and closing rooms, and disabled room lead to the same result – the transition to them is not available.

Another option:

```
room {
    nam = 'inclock';
    dsc = [[I in hours.]];
}:close()

obj {
    nam = 'clock';
    dsc = [[you see an old {clock}.]];
    act = function(s)
        open 'inclock'
        p [[You see that the watch is a secret passage!]];
    end;
}
```

```
room {
    nam = ’Hall’;
    dsc = ’You are in a huge hall.’;
    obj = { ’clock’ };
    way = { path { ’watch’, ’inclock’ } };
};
```

Here we close and open did not move, and the room, which is transition. path shows himself if the room in which he leads disabled or closed.

# Chapter 10

# The effect of objects on each other

The player may use an inventory object on other objects. For he clicks the mouse on the item and then for scene. When this handler is invoked ‘used’ object which function, and handler ‘use’ of the object which apply.

For example:

```
obj {
    nam = ’knife’;
    dsc = ’On the table is a {knife}’;
    inv = ’Sharp!’;
    tak = ’I took the knife!’;
    use = ’You try to use the knife.’;
};

obj {
    nam = ’table’;
    dsc = ’In the {table}.’;
    act = ’Hm... Just a table...’;
    obj = { ’knife’ };
    used = function(s)
        p ’You are trying to do something with a table...’;
        return false
    end;
};
```

In this example, used handler returns false. Why? If you remember, returning false means that the handler instructs the engine about what event he is not

treated. If we would have returned false, the queue to handler ‘use’ of object ‘knife’ simply would not come. In fact, the reality is usually you will use or use or used, it is unlikely it makes sense to do both the handler during the action of the subject on the subject of.

Another example, when it is convenient to return false:

```
use = function(s, w)
    if w^’Apple’ then
        p [[I cleaned up the Apple.]]
        w.cut = true
        return
    end
    return false;
end
```

In this case, use for knife only handles one situation – effect on using knife on Apple. In other cases, the handler returns false and the engine will call the default: game.use.

But it is better if you will add default message in use handler:

```
use = function(s, w)
    if w^’Apple’ then
        p [[I cleaned up the Apple.]]
        w.cut = true
        return
    end
    p [[it is Not necessary to brandish a knife!]]
end
```

This example also demonstrates the fact that the second argument u use is the subject on which we act. The method ‘used’ accordingly, the second – argument is the entity that acts on us:

```
obj {
    nam = ’trash’;
    dsc = [[In the corner is a {trash bin}.]];
    used = function(s, w)
        if w^’Apple’ then
            p [[I threw the Apple in the trash.]]
```

```
            remove(w)
            return
        end
        return false;
    end
}
```

As you remember, before calling use onuse handler is invoked from of the game object, then the object ‘player’, and then my current room. Youcan block ‘use’, returning from any of the following methods ‘onuse’ – false.

Use ‘use’ or ‘used’ (or both) is a matter of personal preference, however, the method used is called earlier and therefore has a higher priority.

# Chapter 11

# The “Player” Object

Player in the world INSTEAD represented by an object of type ‘player’. You can to create multiple players, but one player is present by default.

The name of this object is ‘player’. There is a variable-reference pl which points to this object.

Usually, you do not need to work with this object directly. But sometimes it may be necessary.

By default, the attribute ‘obj’, the player represents the inventory. Usually, it makes no sense to override an object of type player, however, you can do it:

```
game.player = player {
    nam = ”Basil”;
    room = ’kitchen’; -- the starting room of the player
    power = 100;
    obj = { ’Apple’ }; -- let’s give him an Apple
};
```

To INSTEAD have the ability to create multiple players and to switch between them. This is the ‘change_pl()’. In as parameter pass function required an object of type ‘player’ (or his name). Function will switch the current player, andnecessary, will move into the room where the new player.

The ‘me()’ always returns the current player. Therefore, in most games and me() == pl.

# Chapter 12

# The “World” Object

The game world is represented by an object of type world. The name of this object the ‘game’. There is a reference variable, also called game.

Usually you don’t work with this object directly, but sometimes you can call its methods, or change variable values in this object.

For example, the variable game.codepage contains the encoding of the source code games, and by default to “UTF–8”. I do not recommend using other encodings, but sometimes, the choice of encoding can be necessity.

The variable game.player – contains the current player.

In addition, as you already know, the object of the ‘game’ may contain default handlers: ‘act’, ‘inv’, ‘use’, ‘tak’, which will called if the actions of the user are not found no other handlers (or all of them returned false). For example, you you can write in the beginning of the game:

```
game.act = ’does Not work.’;
game.inv = ’hmm ... Odd thing...’;
game.use = ’does Not work...’;
game.tak = ’I don’t need this...’;
```

Of course, they can all be functions.

Also, the game object may contain handlers: onact, ontak, onuse, oninv, onwalk – which can interrupt the action, in case of return false.

Still, the game object you can set handlers: afteract, afterinv, afteruse, afterwalk – that are invoked in case of successful to perform the appropriate action.

# Chapter 13

# List Attributes

List attributes (such as ‘way’ or ‘obj’) allow you to work with its content with a set of methods. Attributes-a list designed keep a list of objects. In fact, you can create lists for their own needs, and place them in objects, for example:

```
room {
    nam = ’fridge’;
    frost = std.list { ’ice cream’ };
}
```

Although usually it is not needed. Listed below are methods for objects of type ‘list’. You can call them for any lists, although these will usually be way and obj, for example:

```
ways():disable() -- disable all transitions
```

- disable() - disables all objects in the list;
- enable() - enables all objects of the list.
- close() - close all objects of the list.
- open() - open the list objects;
- add(object|name [position]) - add an object;
- for_each(function, args) - to call for each object feature arguments;

- lookup(name/tag or object) is object search in the list. Returns the object and the index;
- srch(name/tag or the object) - the search for a visible object in the list;
- empty() - returns true if the list is empty;
- zap() - clear the list;
- replace(what, what) - replace the object in the list;
- cat(list, [position]) - add the contents of the list into the current list position;
- del(name/object) - delete object from the list.

There are functions that return the objects lists:

- inv([player]) - return the player's inventory;
- objs([where]) - return objects of the room;
- ways([room]) - return transitions of the room.

Of course, you can refer to the lists directly:

```
pl.obj:add 'knife'
```

The objects in the lists are stored in the order in which they add. However, if the object is present numeric attribute pri he plays the role of priority in the list. If pri is not specified, the value priority 0 is considered. Thus, if you want some the object was first on the list, give priority pri $<$ 0. If the end of the list – $>$ 0.

```
obj {
    pri = -100;
    nam = 'thing';
    disp = 'Very important item';
    inv = [[Careful with this subject.]];
}
```

# Chapter 14

# Functions that return objects

To INSTEAD define some functions that return different objects. In the description of the function uses the following parameters agreement.

- the characters [ ] describe the optional parameters;
- ‘what’ or ‘where’ - means an object (including the room) is specified by the tag name or a variable reference;

Thus, the main function:

- ‘_(what) ’ - get the object;
- ‘me()’ returns the current object to the player;
- ‘here()’ returns the current scene.
- ‘where ()’ returns a room or an object which is the specified object if the object is in multiple places, you can pass in a second parameter – a Lua table that will be added these objects;
- ‘inroom ()’ similar to where(), but returns the room in which the facility is located (this is important for objects in the objects);
- ‘from([where])’ returns the last room the player goes into given room. Optional parameter – to the last room not for the current room, and for a given;
- ‘seen(what [, where])’ returns an object or transition, if itis present and can see, there is a second optional parameter – select the scene or object/list in which to search;

- ‘lookup(what, [where])’ returns an object or transition, if it there is in the scene or object/list;
- ‘inspect ()’ returns the object if it is visible/available on stage. The search is performed for transitions and objects, including, in the object of the player;
- ‘have ()’ returns the object if it is in the inventory and not disabled;
- ‘live ()’ returns the object if it is present among the living objects ( described below);

These functions are mostly used in the conditions, or to search object from further modification. For example, you can use ‘seen’ for writing terms:

```
onexit = function(s)
    if seen ’monster’ then -- if a function has 1 parameter
        --- I’m not hungry ...
        p ’the Monster’s in the way!’
        return false
    end
end
```

And also, for finding the object in the scene:

```
use = function(s, w)
    if w^’window’ then
        local ww = lookup ’dog’
        if not ww then
            p [[where’s my dog?]]
            return
        end
        place(ww, ’street’)
        p ’I broke the window! My dog jumped on the street.’
        return
    end
    return false
end
```

Example with ‘have’:

```
...
act = function(s)
    if have a 'knife' then
        p 'But I have a knife!';
        return
    end
    take 'the knife'
end
...
```

The question may arise, what is the difference between lookup function and _ ()? The fact that lookup() looks up the object and if the object is not found – just did not return. And the record is _ () assumes that you just you know that the item you get. In other words, _ () is unconditional receipt of the object by name. This function does not, in general, makes the search. Only if the specified tag> will be searched from the available objects. If you use _ () on a non-existent object or unavailable – you will get an error!

# Chapter 15

# Other standard library functions

In the stdlib module INSTEAD, which always connects automatically defined the functions offered by the author as the main work tool for working with the game world. Let us consider them in this Chapter.

In the description of the functions most of the functions under the parameter ‘w’ refers to an object or room, specified by name, tag or variable link. [ wh] - indicates an optional parameter.

- include(file) - file to include in the game; include “lib” – will include the lib file.lua from the current directory with the game;
- loadmod(module) to connect the module of the game;

```
loadmod ”module” -- will include the module module.lua from the current directory;
```

- rnd(m) - a random integer value from ‘1’ to ‘m’;
- rnd(a, b) - a random integer value from ‘a’ to ‘b’ where ‘a’ and ‘b’ are integers >= 0;
- rnd_seed () - set seed of random number generator;
- p (…) output a string to the buffer handler/attribute (with a space at the end);
- pr (…) output a string to the buffer handler/attribute “as is”;- pn (…) output a string to the buffer handler/attribute (with a newline at the end);

- pf(fmt, …) - output formatted string to the buffer handler/attribute;

```
local text = 'hello';
pf("String: %q: %d\n", text, 10);
```

- pfn(…)(…)… "line" - creating a simple handler; This feature simplifies the creation of simple handlers:

```
act = pfn(walk, 'bathroom') "I decided to go to the bathroom.";
act = pfn(enable, '#transition') "I noticed a hole in the wall!";
```

- obj {} - create object;
- stat {} - create status;
- room {} - create a room;
- menu {} - create a menu;- dlg {} - create a dialogue;
- me() - returns the current player;
- here() - returns the current scene.
- from([w]) - returns the room from which the transition to your current scene.
- new(constructor, arguments) - creates a new dinamicheskogo object (to be described later);
- delete(w) - deletes the dynamic object;
- gamefile(file, [reset?]) - load dynamically the file with the game;

```
gamefile("part2.lua", true) -- reset the game state (remove
objects and variables), load part2.lua and start with the main room.
```

- player {} - create a player;- dprint(…) - debug output;
- visits([w]) - the number of visits to the bathroom (or 0 if visits);
- visited([w]) - the number of visits to the room, or false if not visits was;

```
if not visited() then
    p [[it’s my first time.]]
end
```

- walk(w, [Boolean exit], [enter Boolean], [Boolean to change from]) - transition in the scene;

```
-- unconditional jump (to ignore onexit/the onenter/exit/enter);
walk(’end’, false, false)
```

- walkin(w) is a transition in the scene (without calling exit/onexit current);
- walkout([w], [dofrom]) - return from sub-scene (without calling enter/the onenter);
- walkback([w]) - a synonym walkout([w], false);
- _(w) - receiving object;
- for_all(fn, ….) - to perform the function for all arguments;

```
for_all(enable, ’window’, ’door’);
```

- seen(w, [where]) - search for the visible object;
- lookup(w, [where]) is a search object;
- ways([where]) - get list of transitions;
- objs([where]) - get the list of objects;
- search(w) - search for the player object;
- have(w) - search for items in the inventory;
- inroom(w) - the return of the room/rooms, in which the object resides;
- where(w, [table]) - return the object/objects in which the object resides;

```
local list = {}
local w = where('Apple', list)
-- if the Apple is in more than one place, then list will contain an
-- array of these places. If you only need one location, then:
where 'Apple' -- will be enough
```

- closed(w) - true if the object is closed;
- disabled(w) - true if the object is off;
- enable(w) is to include an object;
- disable(w) - off object;
- open(w) - open;
- close(w) - close the object;
- actions(w, string, [value]) - returns (or sets) the number of actions of type t to an object w.

```
if actions(w, 'tak') > 0 then -- object w was taken at least 1 time;
if actions(w) == 1 then -- act it w was called 1 times;
```

- pop(tag) - return to the last branch of the dialog;
- push(tag) - the transition to the next branch of dialogue
- empty([w]) - empty right branch of the dialogue? (or object)
- lifeon(w) - add an object to the list of the living;
- lifeoff(w) - to remove an object from the list of the living;
- live(w) - object is alive?;
- change_pl(w) - change a player;
- player_moved([pl]) is the current player moved in this manner?;
- inv([pl]) - get a list of the inventory;

- remove(w, [wh]) - delete the object from object or room; Removes object obj from list and way (leaving all the rest, for example, game.lifes);
- purge(w) - destroy the object (from all lists); Removes the object from all lists in which it is present;
- replace(w, ww, [wh]) is to replace one object to another;
- place(w, [wh]) is to put the object in the object/room (removing it from the old object/rooms);
- put(w, [wh]) - put object without removing it from the old location;
- take(w) - pick up object;
- drop(w, [wh]) - to throw object;
- path {} - create a transition;
- time () is the number of moves from the beginning of the game.

**Important!**

In fact, many of these functions also are able to work not only with rooms and facilities, but also with lists. That is, 'remove(apple inv())' works like 'remove(apple, me())"; However, remove(apple) also work and remove the object from the places where he present.

Consider a few examples.

```
act = function()
    pn "I'm going to next room..."
    walk (nextroom);
end

obj {
    nam = 'my car';
    dsc = 'in Front of the cabin is my old {pickup} Toyota.';
    act = function(s)
        walk 'inmycar';
    end
};
```

**Important!**

After a call to ‘walk’ the execution of the handler continues until it completed. Therefore, usually, after the ‘walk’ is always followed ‘return’, unless it is the last line of the function, although in this case, it is safe to put ‘return’.

```
act = function()
    pn ”I’m going to next room...”
    walk (nextroom);
    return
end
```

Don’t forget also that when you call ‘walk’ will be called handlers ’onexit/the onenter/exit/enter” and if they forbid the transition, it is not will happen.

# Chapter 16

# Dialogue

Dialogues-the scene of a special type ‘dlg’ containing objects –phrase. When entering the dialogue the player sees a list of phrases that can to choose, getting some kind of reaction game. The default is already selected phrase hidden. The exhaustion of all options, the conversation ends exit to the previous room (of course, if in the dialogue there is constantly visible phrases, which usually occurs something like ‘Finish the conversation’ or ‘Ask again’). When re-entering the dialog, all the hidden phrase again become visible and the dialogue is reset to the initial state (unless, of course, the author of the game specifically make an effort to change the form of a dialogue).

The transition in the dialogue of the game is how the transition to the stage:

```
obj {
    nam = ’cook’;
    dsc = ’I see a {cook}.’;
    act = function()
        walk ’povardlg’
    end
};
```

Although I recommend to use ‘walkin’, as in the case of ‘walkin’ not called ‘onexit/exit’ the current room, and the character with which we can talk, usually to be in the same room, where the main hero. That is:

```
obj {
    nam = ’cook’;
    dsc = ’I see a {cook}.’;
```

```
    act = function()
        walkin 'povardlg'
    end
};
```

If you don't like the prefix of phrases in the form of a hyphen you can specify a string variable:

```
std.phrase_prefix = '+';
```

And get prefixed with a '+' before each phrase. You can also to make a prefix function. The function in this case will be to enter a parameter the number of the phrase. The purpose of the function – to return a string prefix.

Please note that 'std.phrase_prefix' is not saved if you you need to override it on the fly, you will have to restore it state in 'start()' function manually!

**Important!**

> I recommend to use module 'noinv' and set the 'noinv' in the dialogs. The dialogs will look more beautiful and you will protect your game from mistakes and unexpected reactions when using equipment inside the dialog (as usual, the author does not imply such things). For example:

```
require "noinv"
...
dlg {
    nam = 'Guard';
    -- in the dialogues typically do not require inventory
    noinv = true;
    ...
}
```

## 16.1 Phrase

The Central concept in dialogue is the phrase. The phrase is not just the question- answer as you might think. The phrase is a tree, and in this the sense of the whole dialogue can be implemented only phrase. For example:

```
dlg {
    nam = 'conversation';
    title = [[the Conversation with the seller]];
    enter = [[I asked the seller.]];
    phr = {
        { 'You have beans?', '-- No.'},
        { 'You have chocolate?', '-- No.'},
        { 'You have a brew?', '-- Yes',
            { 'How much is it worth?', '-- 50 rubles.' },
            { 'He is cold?', '-- Fridge was broken.',
                { 'Take two!', 'Left one.',
                    { 'Give me one!', function() p [[OK!]]; take 'brew'; end };
                }
            }
        }
    }
}
```

As you can see, the phrase is specified by the attribute phr and can contain branched dialogue. The phrase contains elections, each of which can also contain choices and so on…

The phrase is in the format of pairs: descriptor – reaction. In the simplest case, this a-line. But they can also be functions. Usually, the function is reaction, which can contain code to change the game world.

Steam can be as simple as:

```
{'Question', 'Response }
```

And can contain an array of pairs:

```
{'Question', 'Answer',
    {'Sub-question1', 'Under-answer1' },
    {'Under-question2', 'Under-answer2' },
}
```

In fact, if you look carefully at the attribute phr, you notice that the array of choices is also embedded in the main sentence phr, but only the original pair is missing:

```
dlg {
    nam = 'conversation'; title = [[the Conversation with the seller]];
    enter = [[I asked the seller.]];
    phr = {
    -- there could be question answer 1 level!
    -- 'The main issue', 'Main',
        { 'You have beans?', '-- No.'},
        { 'You have chocolate?', '-- No.'},
        { 'You have a brew?', '-- Yes',
            { 'How much is it worth?', '-- 50 rubles.' },
            { 'He is cold?', '-- Fridge was broken.',
                { 'Take two!', 'Left one.',
                    { 'Give me one!', function() p [[OK!]]; take 'brew'; end };
                }
            }
        }
    }
}
```

Actually, it is. And you can add the 'Main question' and'The answer', but you will not see this question. The thing that when entering into dialogue phrase phr automatically opens, so as usual there is no point in the dialogues of a single phrase. And it is much easier to understand the dialogue as a set of election than as the only tree phrase. So there is never a phr of the initial pair question-answer, but we immediately find ourselves in the array of options that more clear.

When we talk about what the dialogue actually implemented one phrase, we are not quite right. The fact that we are dealing with the phrase,which is located inside the other phrase… It reminds us of the situation objects. Indeed, expressions are objects! Which can to be inside each other. So, take a look at the dialogue from a fresh perspective:

```
dlg {
    nam = 'conversation';
    title = [[the Conversation with the seller]];
    enter = [[I asked the seller.]];
    phr = { -- is a phrase without dsc and act
        -- is the 1st phrase inside the phrase with dsc and act
        { 'You have beans?', '-- No.'},
```

```
        --is the 2nd phrase within a phrase with dsc and act
        { 'You have chocolate?', '-- No.'},
        -- it's a 3rd phrase within a phrase with dsc and act
        { 'You have a brew?', '-- Yes',
        -- is the 1st phrase inside the 3rd phrase with dsc and act
        { 'How much is it worth?', '-- 50 rubles.' },
            { 'He is cold?', '-- Fridge was broken.',
                { 'Take two!', 'Left one.',
                    -- act here as a function
                    { 'Give me one!', function() p [[OK!]]; take 'brew'; end };
                }
            }
        }
    }
}
```

As you can see, the dialogue – it is a room, the phrase-special objects! Now you will understand the subsequent presentation.

> Attention! By default, when the player clicks on one of the questions in the list, the engine repeats it in the conclusion and then displays response. This is done to ensure that the dialogue seemed related. If you want to disable this behavior, use the option std.phrase_show:

```
std.phrase_show = false -- don't display the passphrase question when choosing
```

This setting affects all dialogs, set it in init() or the start() function.

## 16.2 The attributes of the phrases

Consider the phrase:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        {'Red', 'Hold!' },
        {'Blue', 'Here!' },
    }
}
```

If you run this dialog, after you select, say, red pills, we will have another choice of the blue pill. But our the idea, obviously not this! There are several ways to make the dialogue correct.

First, you can use pop() – return to the previous the level of dialogue:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        {'Red', function() p 'Hold!'; pop() end; },
        {'Blue', function() p 'Here!'; pop() end; },
    }
}
```

Or, in another entry:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        {'Red', pfn(pop) 'Hold up!' },
        {'Blue', pfn(pop) 'Here!' },
    }
}
```

But it is not too convenient, besides, what if these phrases contain a new phrase? In cases where the option offers a choice, and this the choice should be the only thing you can ask from the phrase attribute only:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!' },
        {'Blue', 'Here!' },
    }
}
```

In this case, after the choice of the phrase, all phrase of the current context will be closed.

Another common situation, you want the phrase was not hiding after her activation. This is done by setting flag to true:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!' },
        {'Blue', 'Here!' },
        { true, 'And which is better?', 'You choose.' }, -- phrase
        -- which will never be hidden
    }
}
```

An alternative notation, with the explicit task attribute always:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!' },
        {'Blue', 'Here!' },
        { always = true, 'And which is better?', 'You choose.' }, -- phrase
        -- which will never be hidden
    }
}
```

Another example. What if we want the phrase was presented(or hidden) on any condition? This is the handler function cond.

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!' },
        {'Blue', 'Here!' },
        { true, 'And which is better?', 'You choose.' }, -- phrase
        -- which will never be hidden
    },
    { cond = function() return have 'Apple' end
        'Do you want an Apple?', 'Thank you, no.' };
}
```

In this example, only when the player has an Apple, seem branch dialog 'do you want an Apple?'.

It is sometimes convenient to perform an action at the moment when the options current level(context) of the dialogue is exhausted. For this purpose the handler function onempty.

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!' },
        {'Blue', 'Here!' },
        onempty = function()
            p [[You made your choices.]]
            pop()
        end;
    },
    { cond = function() return have 'Apple' end
        'Do you want an Apple?', 'Thank you, no.' };
}
```

Please note that when there is a method onempty, automatic return to the previous branch is not performed, it is assumed that the method onempty will do everything you need.

All the above attributes can be set with any phrase. In fact on the 1st level:

```
phr = {
    onempty = function()
        p [[end of conversation.]]
        walkout()
    end;
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!' },
        {'Blue', 'Here!' },
        onempty = function()
            p [[You made your choices.]]
            pop()
        end;
    },
    { cond = function() return have 'Apple' end
```

```
        'Do you want an Apple?', 'Thank you, no.' };
}
```

## 16.3 Tags

We have considered mechanisms dialogues, which already allow to create quite complex dialogs. However, these funds may not be enough. Sometimes we need to be able refer to phrases from other places dialogue. For example, to selectively enable them, or analyze them state. And make transitions from one of the branches of dialogue in others.

All this is possible for phrases that have the tag. Create a sentence with tag very simple:

```
phr = {
    {'#?', 'What have you got?', 'Pills. The red and blue. You what?',
        {'#red', 'Red', 'Hold!' },
        {'#blue', 'Blue', 'Here!' },
    },
}
```

As you can see, the presence at the beginning of the phrase string, starting with symbol '#' - indicates the presence of the tag.

For such phrases employs standard methods, such as seen or enable/disable. For example, we could do without the attribute only as follows:

```
phr = {
    {'#?', 'What have you got?', 'Pills. The red and blue. You what?',
        {'#red', 'Red', 'Hold!'
            cond = function(s)
                return not closed('#blue')
            end
        },
        {'#blue', 'Blue', 'Here!',
            cond = function(s)
                return not closed('#red')
            end
        },
    },
}
```

Tags, except that allow you to learn and to change the status of a particular phrases, making possible the transitions between phrases. For this purpose the functions push and pop.

push(to) – makes the transition to the phrase about remembering the position in the stack.

pop([where]) – invoked without a parameter, goes up by 1 position in the stack history. You can specify a specific tag phrase that must be in history, in this case, the refund will be credited to her.

It should be noted that when you click on a push, we move not one the phrase, and the phrase list the phrase. That is, disclose it, as well as this is done for the main phrase phr. For example:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!', next = '#aboutpill' },
        { 'Blue', 'Here!', next = '#aboutpill' },
    },
    { false, '#aboutpill',
        {'I made the right choice?', 'Time will tell.'}
    },
}
```

Here we see several techniques:

- next attribute, instead of explicit descriptions of reaction as a function with push. next is a simple way to record push.

- false at the beginning of the phrase makes the phrase off. She is locked off until you do an explicit enable. However, inside the phrase we can go and show the contents of the elections. Alternative entry possible with the use of the hidden attribute:

  { hidden = true, '#aboutpill', {'I made the right choice?', 'Time will tell.'} },

Thus it is possible to record conversations is not a tree, and linear. More one feature of the transitions is that if the phrase is not described the reaction, when the transition is triggered by the title phrase:

```
phr = {
    { 'What have you got?', 'Pills. The red and blue. You what?',
        only = true,
        {'Red', 'Hold!', next = '#aboutpill' },
        { 'Blue', 'Here!', next = '#aboutpill' },
    },
    { false, '#aboutpill', [[I took the pill and the wizard smiled slyly.]],
        {'I made the right choice?', 'Time will tell.'},
        {'What next?', 'You're free.'},
    },
}
```

When choosing a tablet, will be called the method header phrase `#aboutpill', and then will be presented choice.

If you like linear, you might prefer the following option:

```
dlg {
    nam = 'dialog';
    phr = {
        { 'What have you got?', 'Pills. The red and blue. You what?',
            only = true,
            {'Red', 'Hold!', next = '#aboutpill' },
            { 'Blue', 'Here!', next = '#aboutpill' },
        }
    }
}: with {
    { '#aboutpill', [[I took the pill and the wizard smiled slyly.]],
        {'I made the right choice?', 'Time will tell.'},
        {'What next?', 'You're free.'},
    },
}
```

The fact that the attribute phr defines the first object of the room. But you can fill the room objects in the usual way: by setting the obj or with. Since entering the dialogue reveals the 1st phrase, then the rest phrase you will not see (pay attention to the phrase `#aboutpill' not worth itfalse), but you will be able to do transitions on these phrases.

## 16.4 Methods

As you already know, objects in an INSTEAD may be able open/closed off/ turned on. This corresponds to the phrase dialogue?

For common phrases, after activation of the choice phrase *closes*. When re-entering the dialogue, the phrases get *opened*.

For phrases with always = true (or true at the beginning of the definition) – this the closing does not occur.

For phrases with hidden = true (or false at the beginning of the definition) – the phrase will be created as disabled. It will not be visible until until is explicitly enabled.

For phrases with cond(), every time you browse phrases is called this method, and depending on the return value (true/not true) phrase turns on or off.

Knowing this behavior, you can hide/show and analyze phrase standard functions: disable /enable /empty /open /close /closed /disabled and so on…

However, to do this you can only in the dialogue, as all the phrases get identified by tags. If you want to modify condition/parse phrases from other rooms you can:

- to give the phrase name is { nam = ‘name’ }…
- search for the phrase on the tag in the other room: local ph = lookup(‘#tag’, ‘dialogue’) and then work with it;

With regard to functions push/pop then you can call it explicitly as methods of dialogue, for example:

```
_’dialog’:push ’#new’
```

But better to do it in the dialogue, for example, in enter.

In addition, there is a method :reset, which resets the stack and sets the starting phrase, for example:

```
enter = function(s)
    s:reset ’#start’
end
```

> It should be noted that when you make a enable/disable/open/close phrase, then you perform the action exactly over this phrase, not over phrases included inside. But since the showing phrases engine stop on/off /closed facility-phrase and will not be included inside, that’s enough.

# Chapter 17

# Special objects

In STEAD3 there are special objects that perform specific functions. All such objects can be divided into two class:

1. System objects @;
2. Substitution.

System objects are objects whose name begins with the character '@' or '$'. Such objects are usually modular. They are not destroyed at the death of the game world (for example, when uploading game file, when loading the game from save, and so on). Examples of objects: @timer, @prefs, @snd.

Such objects, in addition to their special functions, can be used the link, without explicitly putting the object on the stage or the stock, but the mechanism of action of these objects are special.

## 17.1 The object '@'

Usually, you do not need to work with such objects, but as example, consider the implementation of 'links'.

Suppose we want to make a link, clicking on which we will move in the other room. Of course, we could add the object to the scene, but it's worth we do it in such a simple case?

How we can help the system object?

```
obj {
```

```
    nam = '@walk';
    act = function(s, w)
        walk(w, false, false)
    end;
}
room {
    nam = 'main';
    title = 'Home';
    decor = [[Start {@start walk|adventure}]];
}
```

When you click on the link "adventure" method will be called act-object '@walk' with the parameter "start".

In fact, in the standard library stdlib is already the object filename with '@' which allows you to do your handlers the following links follows:

```
xact.walk = walk

room {
    nam = 'main';
    title = 'Home';
    decor = [[Start {@ start walk|adventure}]];
}
```

Note the space after @. This entry does the following:

- takes an object with '@' (this object is created by the library stdlib);
- takes his act;
- causes act with the parameters of the walk and start;
- act object the '@' looks at the array xact;
- walk detects a method that will be called from the array xact;
- start is a parameter of this method.

Another example:

```
xact.myprint = function(w)
    p (w)
end

room {
    nam = 'main';
    title = 'Home';
    decor = [[Push {@ myprint "hello world"|the button}]];
}
```

## 17.2 Lookup

Objects whose name begins with the character '$' are also considered system objects, but they work differently.

If the output text is found on the link:

```
{$my a b c|text}
```

The following happens:

1. Does the object $my;
2. Taken act the object $my;
3. Act is called: _'$my'(a, b, c, text);
4. The returned string replaces the entire structure {…}.

Thus, the objects play the role of a wildcard.

Why is it necessary? Imagine that you developed a module, which turns write from a text view in the graphics. You write object $math which in its act method converts text to a graphic image (sprite) and returns it in the text stream. Then to use this module extremely simple example:

```
{$math|(2+3*x)/y^2}
```

# Chapter 18

# Dynamic events

You can define handlers that run every time when the game time increments by one. Usually, it makes sense for the living characters, or any background processes of the game. The algorithm of step games looks like this: - The player clicks on the link; - Reaction, ‘act’, ’use”, ‘inv’, ‘tak’, the inspection of the scene (click on the name scene) or transition into another scene; - Dynamic events; - Output a new state of the scene.

For example, make a snow leopard alive:

```
obj {
    nam = ’Barsik’;
    {- do not save the array lf
        lf = {
            [1] = ’Barsik is moving in my bosom.’,
            [2] = ’Barsik looks out from his bosom.’,
            [3] = ’Barsik purring in my bosom.’,
            [4] = ’Barsik shivers in my bosom.’,
            [5] = ’I feel Barsik heat in his bosom.’,
            [6] = ’Barsik sticks his head out of his pocket and looks around.’,
        };
    };
    life = function(s)
        local r = rnd(5);
        if r > 2 then -- doing this is not always
            return;
        end
```

```
        r = rnd(#s.lf); -- the # symbol is the number of elements in the array
        p(s.lf[ r]); -- derive one of the 6 States of the snow leopard
    end;
....
```

And here is the point in the game, when snow leopard gets to us in thy bosom.

```
take 'snow leopard' -- add it to your inventory
lifeon 'Barsik' -- to revive the cat!
```

Any object (including the stage) can have its own handler, 'life', called each tick of the game if the object was added to the list live objects using the 'lifeon'. Don't forget to remove live objects from the list using the 'lifeoff' when they are no longer needed. It is possible to do this, for example, in the handler of 'exit' or by any other method.

> If your game is a lot of "live" objects, you can ask them a clear position in the list, adding. For this, use second numerical parameter (non-negative integer) 'lifeon', the lower the number the higher the priority. 1 – the highest. Or you you can use the attribute of the pri object. However, this attribute will affect the priority of the object in any list.

If you need a background process in some room, send it in 'enter' and remove the 'exit', for example:

```
room {
    nam = 'In the basement';
    dsc = [[it's dark in Here!]];
    enter = function(s)
        lifeon(s);
    end;
    exit = function(s)
        lifeoff(s);
    end;
    life = function(s)
        if rnd(10) > 8 then
            p [[I heard something rustling!]];
            -- occasionally to scare the player rustles
        end
```

```
    end;
    way = { 'Home' };
}
```

If you need to determine whether the transfer of the player from one scene to another, use 'player_moved()'.

```
obj {
    nam = 'flashlight';
    on = false;
    life = function(s)
        if player_moved() then -- put out the torch transitions
            s.on = false
            p "I turned off the flashlight."
            return
        end;
    end;
...
}
```

For tracking moving events, use 'time()' or an auxiliary counter variable. To determine the location player – 'here()'. For determining that the object is "alive" – the 'live()'.

```
obj {
    nam = 'dynamite';
    timer = 0;
    used = function(s, w)
        if w^'match' then -- match?
            if live(s) then
                return "it's Already lit!"
            end
            p "I have lit the dynamite."
            lifeon(s)
            return
        end
        return false-if not match
    end;
```

```
    life = function(s)
        s.timer = s.timer + 1
        if s.timer == 5 then
            lifeoff(s) if here() == where(s) then
            p [[the Dynamite exploded right next to me!]]
        else
            p [[I heard an explosion somewhere.]];
        end
    end
end;
...
}
```

If ‘life’ handler returns the event text, it is printed after the description of the scene.

You can return from a handler for ‘life’ second return code (‘true’ or ‘false’). If you return true-it is a sign of important the event, which will be displayed to describe the objects in the scene, for example:

```
p ’In walked the guard.’
return true
```

Or:

```
return ’you entered the room the security guard.’, true
```

If you return false, the chain of life methods will fail on you. It easy to do when performing a walk from the life, for example:

```
life = function()
    walk ’theend’
    return false -- this is the last life
end
```

If you want to block ‘life’ handlers in some of the rooms, use the module ‘nolife’. For example:

```
require ”noinv”
require ”nolife”

dlg {
    nam = ’Guard’;
    noinv = true;
    nolife = true;
...
}
```

We should also consider the transition of the player from ‘life’ handler. If you are going to use the function ‘walk’ inside’life’, then you should consider the following behavior.

If ‘life’ takes the player to a new location, it is usually assumed you:

1. Clear withdrawal reactions: game:reaction(false);
2. Cleanse withdrawal live methods at the moment: game:events(false, false)
3. Do the walk.
4. Stop the chain of life of the call using return false;

Some points required clarification.

game:reaction() – allows you to take/modify the output of the reaction user, if set to false this means to reset the reaction.

game:events () – allows you to take/change /withdrawal of life methods. Inas choices are made a priority and not a priority criteria false we cancelled the whole output of the previous life methods.

The standard library has a function life_walk(), which makes the described actions. You just have to return false.

# Chapter 19

# Graphics

Graphic interpreter INSTEAD analyzes the attribute of the scene ‘pic’, sees it as the path to the picture, for example:

```
room {
    pic = ’gfx/home.png’;
    nam = ’Home’;
    dsc = ’I am home’;
};
```

**Important!**

Use in ways only direct ‘/’. Also, it is strongly recommended to use the names of directories and files only Latin lowercase characters. This way you will protect your game from compatibility problems and it will work on all architectural platforms where ported INSTEAD.

Of course, ‘pic’ can be function, expanding the possibilities of the developer. If the current scene does not defined the attribute ‘pic’ attribute is taken ‘game.pic’. If the picture is not displayed.

Supports all common image formats, but I I recommend you to use ‘png’ and where () ‘jpg’.

You can use as images animated GIF files. Be sure that they are GIF files, and not WEBP format or PNG format, which are unsupported.

You can embed the graphic images right in the text in the inventory, transitions, titles rooms and ‘dsc’ using the function ‘fmt.img’ (this will switch on the module fmt).

For example:

```
require "fmt"

obj {
    nam = 'Apple'
    disp = 'Apple'..fmt.img('img/apple.png');
}
```

That, at least, the scene picture always should be made in the form of 'pic' attribute, not insertion 'fmt.img' to 'dsc'.

The fact that the picture of the scene scales on other algorithm. Pictures 'fmt.img' get scaled in accordance with the settings INSTEAD (the scale theme), and 'pic' – also takes into account the size of the image.

In addition, the images 'pic' have other properties, for example, the ability to track coordinates of mouse clicks.

If you put 'fmt.img' inside { and }, you get a graphical link.

```
 obj { nam = 'Apple'; disp = 'Apple' ..img('img/apple.png');
dsc = function(s) p ("the floor is {Apple",fmt.img 'img/
apple.png', "}"); -- other options: -- return "On the floor lies
{Apple"..fmt.img('img/apple.png')..."}"; -- p "On the floor lies
{Apple"..fmt.img('img/apple.png')..."}"; -- or dsc = "the floor is
{Apple"..fmt.img('img/apple.png')..."}"; end; }
```

INSTEAD supports wrapping images with text. If the picture is inserted using the function 'fmt.imgl'/'fmt.imgr', it will located at the left/right edge.

**Important!**

> Images inserted into text using 'fmt.imgl/fmt.imgr' can't be links!!! Use them only for decorative purposes.

To set spacing around an image, use the 'pad' example:

```
-- indentation 16 from each edge
fmt.imgl 'pad:16,picture.png'
-- margins: top 0, right 16, bottom 16, left 4
fmt.imgl 'pad:0 16 16 4,picture.png'
-- margins: top 0, right 16, bottom 0, left 16
fmt.imgl 'pad:0 16,picture.png'
```

You can use pseudo-files for images and rectangles empty areas:

```
dsc = fmt.img 'blank:32x32'..[[Line with blank image.]];
dsc = fmt.img 'box:32x32,red,128'..[[Line with red semi-transparent square.]];
```

INSTEAD can process compound images, for example:

```
pic = 'gfx/mycat.png;gfx/milk.png@120,25;gfx/fish.png@32,32';
```

Thus, the composite image is a set of paths to images, separated by ';'. The second and subsequent components can contain Postfix in the form @x\coordinate,y\coordinate%, where the coordinate 0,0 corresponds to the left the top corner of the image. The total size of the image is considered equal to the total amount of the first component of the composite image, that is, the first component (in our example-gfx/mycat.png) plays the role of canvas, and subsequent components are superimposed on this canvas.

The overlay is for the upper left corner of the overlaypictures. If you need to overlay was the center of the overlay images, use before the coordinate prefix "c", for example:

```
pic = 'gfx/galaxy.png;gfx/star.png@c128,132';
```

Having as a function the formation of the path of the composite pictures, you can generate an image based on the game state.

If you are in the game tied to any coordinates images or their sizes, do it in relation to the original of image sizes. Scaling topic under specified playerthe resolution INSTEAD he will convert the coordinates (with the coordinates for the game look like the game is running without scaling). However, there may be a small error of computation.

If you do not have enough of the features described in this Chapter, I examine the module "sprite", which provides more opportunities for graphic design. But I highly recommend not to do it in his the first game.

# Chapter 20

# Music

To work with music and sounds you will need the snd module.

```
require ”snd”
```

The interpreter plays in the cycle of the current music that is set using the function: ‘snd.music(music file)’.

**Important!**

Use in ways only direct ‘/’. Also, it is strongly it is recommended to use the names of directories and files only Latin lowercase characters. This way you will protect your game from compatibility problems and it will work on all architectural platforms where ported INSTEAD.

Supports most music formats, but we strongly it is recommended to use the format ‘ogg’ as it supported in the best way in all versions INSTEAD (for different platforms).

**Important!**

Caution should be exercised when using tracker music as some Linux distributions can be a problem when playing certain files (error in bundle libraries SDL_mixer and libmikmod).

Also, if you use the ‘mid’ files, be prepared for the fact that the player will hear them only in the Windows version INSTEAD (as in most cases Unix version of SDL_mixer compiled without support “timidity”).

As the frequency of the music files, use a frequency multiple of 11025.

```
room {
    pic = ’gfx/street.png’;
    enter = function()
```

```
        snd.music 'mus/rain.ogg'
    end;
    nam = 'on the street';
    dsc = 'it is raining outside.';
};
```

'snd.music()' without a parameter returns the current track name.

In function 'snd.music()' you can pass second parameter-the number playout (cycles). To get the current counter you can use 'snd.music()' no parameters – the second return value. 0 – it is eternal cycle. 1..n-the number of playbacks. –1 – the playback of the current track is finished.

In order to cancel the music, you can use 'snd.stop_music()'

In order to know if the music is:

```
snd.music_playing()
```

You can set the rise time and attenuation of music, by calling:

```
snd.music_fading(o [i])
```

Here o is the time in milliseconds. for attenuation and i is the time in milliseconds. to rise music. If you specify only one parameter – both times are considered as same. After the call, the settings will affect playback of all music files.

For playback of sounds using snd.play()'. Highly it is recommended to use the format 'ogg', although most common sound formats will also work.

The distinction between music and sound file is that the engine monitoring the process of playing music and saves/restores the currently played track. Exiting the game and downloading it again, the player hear the same music that you heard when you exit. Sounds generally indicate short-term effects, and the engine does not store or restores sound events. So, if a player did not have time to listen the sound of the shot and left the game, after downloading the file, save it not hear the sound (or end) again.

However, if you consider the fact that 'snd.play()' allows you to run looped sounds, the difference between music and sound becomes not so unambiguous.

So, the definition of the function: 'snd.play(file [channel], [loop])', where:

- file – path and /or name of the sound file;
- channel – channel number [0..7]; If not specified, get first free.

- cycle – the number of playbacks 1..n, 0 – looping.

To stop playback you can use ‘snd.stop()’. For stop sound in a certain channel ‘snd.stop(channel)’.

**Important!**

> If you’re using looped sounds, you will have to reset their state (run again with ‘snd.sound()’) in function ‘start()’

For example:

```
global ’wind_blow’ (false)
...
function start()
    if wind_blow then
        snd.play(’snd/wind.ogg’, 0)
    end
end
```

If you are not fairly described here functions for working with sound use the full description of the module “snd”.

# Chapter 21

# The formatting of the output

Usually INSTEAD deals with formatting and design output. For example, the stage separates static from dynamic. Highlights italicize the actions of the player. Moves focus to the change in the text and etc Modules like "fmt" improve the output quality of the game without additional effort on the part of the author.

For example:

```
require 'fmt'
fmt.para = true -- enable indentation of paragraphs
```

And your game will look much better. If you need some automatic processing of the displayed text, you can enable the module "fmt" and define the function 'fmt.filter'. For example:

```
require "fmt"
fmt.filter = function(s, state)
    -- s -- output
    -- state -- true if this is the beat game (output stage)
    if state then
        return 'This string will be added to the beginning of output\n'..s;
    end
    return s
end
```

A good game for INSTEAD not do your design in addition to splitting the text 'dsc' paragraphs using the characters '^^',so think, and whether you want to design your games manually?

However, sometimes it's still necessary.

Attention! By default, all trailing and starting newlines, spaces and tabs are cut from the output handlers. So as usual, they are meaningless and even harmful. In rare cases, the author may want more control over the output, then it maybe ask std.strip_call as false in init() or start(), for example:

```
std.strip_call = false

obj {
    dsc = [[There is {Apple}.^^^^]] --- now the line
    -- will not be clipped, even though it's weird
}
```

But usually such manual formatting indicates a bad the style. For the design stage it is better to use decor and/or substitution.

## 21.1 Formatting

You can do simple text formatting using the functions:

- fmt.c(string) - place in the center.
- fmt.r(string) - place right;
- fmt.l(line) - place left;
- fmt.top(row) - top row;
- fmt.bottom(string) - the bottom line;
- fmt.middle(line) - middle line (default).

For example:

```
room {
    nam = 'main';
    title = 'Welcome';
    dsc = fmt.c 'Welcome!'; -- if a function has only 1 parameter,
    -- you can omit the parentheses;
}
```

These features affect not only text but also images, inserted using ”fmt.img()’.

It should be noted that if you use several functions formatting, it is assumed that they belong to different rows (paragraphs). Otherwise, the result is undefined. Or abuse the text into paragraphs with ‘^’ or ‘pn()’.

INSTEAD in the derivation removes the extra spaces. This means that no matter how many spaces you insert between words, are all the same at the conclusion they will not be counted for calculating the distance between words. Sometimes it could be a problem.

You can create *non-breaking line* using: fmt.nb(row). For example, the module “fmt” uses a continuous line to create indentation at the beginning of paragraphs. Also, ‘fmt.nb’ can to be useful to output special characters. We can say that the whole string parameter ‘fmt.nb’ is perceived by the engine as one big word.

Another example. If you use the underline text, the spaces between words are not underlined. When using ‘fmt.nb’ the gaps will also be highlighted.

INSTEAD not support the display of tables, but the conclusion is simple tabular data you can use ‘fmt.tab()’. This function used for absolute positioning in the line (tab delimited).

```
fmt.tab(position, [center])
```

*Position*, is a text or numeric parameter. If you specify a numeric parameter, it is treated as position in pixels. If it is set in a string parameter ‘number%’, it is perceived as position, expressed in percentage of the width of the output window scene.

The optional parameter center specifies the position in the following’fmt.tab’ word that will be placed at the specified offset in the line. Positions can be as follows:

- left;
- right;
- center.

By default, it is considered that the option “left”.

So, for example:

```
room {
    nam = 'main';
```

```
    disp = 'Start';
    -- the location of the 'Start!' in the middle of the line
    dsc = fmt.tab('50%', 'center')..'Start!';
}
```

Of course, not a very good example, as the same could make using 'fmt.c()'. A more successful example.

```
 dsc = function(s)
    p(fmt.tab '0%')
    p "Left";
    p(fmt.tab '100%', 'right')
    p Right;
end
```

In fact, the only situation where the use of 'fmt.tab()' justified – is the output of table data.

It should be noted that in a situation when we write something like:

```
-- the location of the 'Times' on the center line
dsc = fmt.tab('50%', 'center')..'one two three!';
```

Only the word 'Time' is placed in the centre of the line, the rest of the words will be appended to the right of this word. If you want to center the 'Times two, three!' as one unit, use 'fmt.nb()'.

```
-- place 'one two three!' on the center line
dsc = fmt.tab('50%', 'center')..fmt.nb ('one two three!');
```

To INSTEAD it is also possible to perform a simple vertical formatting. To do this, use the vertical tab:

```
fmt.y(position, [center])
```

As in the case of fmt.tab position is text or numeric parameter. Here it is perceived as a position of the line expressed in pixels or percentage of the height region of the scene. For example, 100% – corresponds to the lower boundary region of the scene. 200% corresponds – the lower boundary of the second page of output (two the height of the output pane scene).

The optional parameter center specifies the position withinline on which you position:

- top (upper edge);
- middle (center);
- bottom (for bottom-the default).

It should be noted that ‘fmt.y’ works entirely for the line. If the line will meet a few of fmt.y, act will be the last one tab.

```
-- the location of the ’CHAPTER I’ - center stage
dsc = fmt.y(’100%’)..”CHAPTER I”;
```

*Esli position specified by the tab, is already occupied by another row, the tab is ignored.*

By default, the static part of the scene is separated from the dynamic double newline. If you do not fit, you can override ‘std.scene_delim’, for example:

```
std.scene_delim = ’^’ -- single line
```

You cannot change this variable in the handlers, as it is not remains, but you can set it for the entire game, or fix it manually in the function ‘start()’.

If you absolutely do not like how INSTEAD forms conclusion (sequence of paragraphs), you can override the function ‘game.display()’, which by default looks like the following:

```
game.display = function(s, state)
    local r, l, av, pv
    local reaction = s:reaction() or nil -- reaction
    r = std.here()
    if state then -- the beat of the game?
        reaction = iface:em(reaction) -- italic font
        av, pv = s:events()
        av = iface:em(av) - the output of ”important” life
        pv = iface:em(pv) - output background life
        l = s.player:look() -- objects [and scene] -- the objects and scene
    end
    l = std.par(std.scene_delim,
        reaction or false, av or false, l or false
        pv or false) or ”
    return l
end;
```

The fact that I have brought here this code does not mean that I recommend to override this feature. On the contrary, I categorically against such strong binding to the text formatting. However, sometimes there is a situation where full control over the sequence output needed. If you write your first game, just skip this text.

## 21.2 Making

You can change the font style of text using combinations of features:

- fmt.b(string) - bold text;
- fmt.em(string) - italic;
- fmt.u(string) - text that is underlined;
- fmt.st(string) - text crossed out.

For example:

```
room {
    nam = 'Intro';
    title = false;
    dsc = function(s)
        p ('You are in a room: ')
        p (fmt.b(s))
    end;
}
```

> Using the function 'fmt.u' and 'fmt.st' on strings containing spaces you will get broken lines in these places. What to avoid it, to turn the text into *non-breaking string*:

```
fmt.u(fmt.nb "now the unabridged" )
```

Strictly speaking, INSTEAD does not support simultaneous display of different fonts in the window scene (except for the different font styles), so if you require a more flexible output control, you can do the following:

- Use the graphic insert 'fmt.img()';

- Use the module ‘fonts’, which implements the rendering different fonts at the expense module ‘sprite’;
- Use another engine, because most likely you use INSTEAD for other purposes.

# Chapter 22

# Constructors and inheritance

**Caution!**

If you write your first game, it would be better if she was simple. For a simple game, the information in this Chapter do not need. Moreover, 90% say it's not a good described in this Chapter!

If you're writing a game, where many similar objects mayyou will want to simplify their creation. You can do one of the following ways:

- Create your designer;
- Create a new feature class.

## 22.1 Designers

Constructor-a function that creates the object for you and fills his attributes as you need it. Let's consider an example. For example, in your game will be a lot of Windows. You need to create a window, any window can be break. We can write constructor 'window'.

```
window = function(v)
    v.window = true
        v.broken = false
    if v.dsc == nil then
        v.dsc = 'there is a {window}.'
    end
    v.act = function(s)
```

```
        if s.broken then
            p [[broken Window.]]
        else
            p [[it's dark outside.]]
        end
    end
    if v.used == nil then
        v.used = function(s, w)
            if w^'hammer' then
                if s.broken then
                    p [[the Window is already broken.]]
                else
                    p [[I broke the window.]]
                    s.broken = true;
                end
                return
            end
            return false
        end
    end
    return obj(v)
end
```

As you can see, the idea of the designers is simple. You just create the function which receives the input of the table with the attributes {} which is the constructor can fill out the desired attributes. Then this table is passed constructor obj/room/dlg and returns the resulting object.

Now, create a window was easy:

```
window {
    dsc = [[there is a {window}.]];
}
```

Or, because the window is usually a static object, you can create it directly in 'obj'.

```
obj = { window {
        dsc = 'In the Eastern wall there is a {window}.';
    }
};
```

Our window will be ready used method and act method. You can to check the fact that the object is a window-just checking the attribute window:

```
use = function(s, w)
    if w.window then
    p [[the Action.]]
return
end
    return false
end
```

The condition "weakness" of the window, this attribute is broken.

How to implement inheritance in the constructors?

In fact, in the example above is already used inheritance. Indeed, the designer 'window" causes the other constructor 'obj', thus obtaining all the properties of conventional object. Also, the 'window' defines a variable attribute 'window', to the game, we could understand that we are dealing with a window. For example:

To illustrate the inheritance mechanism, create a class of objects 'treasure', those. treasures.

```
global { score = 0 }
treasure = function()
    local v = {}
    v.disp = 'treasure'
    v.treasure = true
    v.points = 100
    v.dsc = function(s)
        p ('there is {', std.dispof(s), '}.')
    end;
    v.inv = function(s)
        p ('This is ', std.dispof(s), '.');
    end;
    v.tak = function(s)
        score = score + s.points; -- increase the score
        p [[Trembling hand I took the treasure.]];
    end
    return obj(v)
end
```

Now, let's create gold, diamond and treasure chest.

```
gold = function(dsc)
    local v = treasure();
    v.disp = 'gold';
    v.gold = true;
    v.points = 50;
    v.dsc = dsc;
    return v
end

diamond = function(dsc)
    local v = treasure();
    v.disp = 'diamond';
    v.diamond = true;
    v.points = 200;
    v.dsc = dsc;
    return v
end

chest = function(dsc)
    local v = treasure();
    v.disp = 'chest';
    v.chest = true
    v.points = 1000;
    v.dsc = dsc;
    return v
end
```

Now, in the game you can create a treasure using constructors:

```
diamond1 = diamond("In the mud, I noticed {the diamond}.")
diamond2 = diamond(); -- here is the standard description of the diamond
gold1 = gold("In the corner I noticed the glitter {gold}.");

room { nam = 'cave';
    obj = {
        diamond1,
```

```
        gold1,
        chest("I can see {chest}!")
    };
}
```

In fact, how to write constructor functions and implement the inheritance principle, depends only on you. Choose the most simple and intuitive way.

When writing constructors, it is sometimes useful to make a call handler the way it does INSTEAD. To do this, use 'std.call(object, method, options)', this function will return the reaction of the attribute as a string. For example, consider a modification 'window', which is that you can define your own the reaction to the inspection window, which will be executed after the standard posts about what is the broken window (if it is broken).

```
window = function(nam, dsc, what)
    local v = {} -- creates an empty table
    -- fill it
    v.window = true
    v.what = what
    v.broken = false
    if dsc == nil then
        v.dsc = 'there is a {window}'
    end
    v.act = function(s)
        if s.broken then
            p [[broken Window.]]
        end
        local r, v = stead.call(s, 'what')
        if v then -- the handler is executed?
            p(r)
        else
            p [[it's dark outside.]]
        end
    end
    return obj(v)
end
```

Thus, we can create a window to specify the third parameter, where to define a function or a string that will be occurring at the same time the inspection window.

The message that the window was broken (if it really broken), you will see before this response.

## 22.2 Feature class

The constructors of objects are widely used in STEAD2. IN STEAD3 obj/dlg/room are implemented as object classes. Class of objects is to create for those occasions when the behavior of the created object is not fits into standard objects obj/room/dlg and you want to change methods of the class. Changing a class method, for example, you can generally to change how the object looks in the scene. As an example, consider creating a class "container". The container can store other objects to be closed and open.

```
-- create own class container
cont = std.class({ -- create the class cont
    __cont_type = true; -- to determine the type of the object
    display = function(s) -- overridable method of displaying the subject
        local d = std.obj.display(s)
        if s:closed() or #s.obj == 0 then
            return d
        end
        local c = s.cont or 'Inside:' -- descriptor content
        local empty = true
        for i = 1, #s.obj do
            local o = s.obj[i]
            if o:visible() then
                empty = false
                if i > 1 then c = c .. ', ' end
                c = c..'{'..std.nameof(o)..'|'..std.dispof(o)..'}'
            end
        end
        if empty then
            return d
        end
        c = c .. '.'
        return std.par(std.space_delim, d, c)
    end;
} std.obj) -- we inherit from the default object
```

After that, you can create containers like this:

```
cont {
    nam = 'box';
    dsc = [[there is {box}.]];
    cont = 'box: ';
}: with {
    'Apple', 'pear';
}
```

When the container is opened, you will see a description box, and the contents of the box in the row of links: In the box: Apple, pear. dsc objects Apple and pear are also shown if they are set.

If you want to hide dsc objects when opening the container, leaving only the names of the objects, you can perform the following modification:

```
-- replace the function of display of any object
-- if the object is inside the container, don't call it dsc
std.obj.display = function(self)
    local w = self:where () - where is the object?
    if not std.is_obj(w, 'cont') then -- if not in a container
        local d = std.call(self, 'dsc')
        return d
    end
end
```

Unfortunately, a detailed description of classes is beyond the scope of this manual, these things will be discussed in another guide module developers. In the meantime, for your first game, you don't need to write your own object classes.

# Chapter 23

# Useful tips

## 23.1 Split files

When your game gets large, the location of its code entirely in ‘main3.lua’ – bad idea.

To break the text of the game files you can use ‘include’. You must use ‘include’ in a global context so that while loading ‘main3.lua’ is loaded and all the remaining fragments of the game, for example.

```
-- main3.lua
include ”episode1” -- .lua can be omitted
include ”npc”
include ”start”

room {
    nam = ’main’;
....
```

How to split a source text file depends on you. I use the files in accordance with the episodes of the game (which is usually mild linked), but you can create files to store separately rooms, objects, dialogues, etc. Is a matter of personal convenience.

There is also the possibility to dynamically load parts of the game (with the ability to redefine objects). To do this, you you can use the ‘gamefile’:

```
...
```

```
act = function() gamefile ("episode2") end -- .lua can be omitted
...
```

> Attention! If your game defines a function init(), loadable parts of it must be present! Otherwise case, after uploading the file, will be called the current function init () that is not usually desirable.

'gamefile()' also allows you to upload a new file and forget stackprevious downloads by launching this new file as an independent game. To do this, set the second parameter as 'true'. Keep in mind that existing modules remain and survive the operation gamefile in both cases. 'gamefile()' can only be used in handlers.

```
act = function() gamefile ("episode3.lua", true); end;
```

In the second option 'gamefile()' can be used to design multilingual games, or game collections, where the actual shell running a standalone game.

## 23.2 Menu

The default behavior of the item of inventory is that the playerneed to make two mouse clicks. This is necessary because each any inventory item can be used on another object or scene inventory. After the second click happens games the beat games. Sometimes this behavior may be undesirable. You might want to make a game in which game mechanics differs from the classical INSTEAD games. Then you may need menu.

Menu is an element of inventory, which is triggered on the first click. When this menu can inform the engine that the action is not a game tact. Thus, using the menus you can create in the area inventory management a game of any complexity. For example, there is module "proxymenu" that implements control of the game style quests on the ZX-spectrum. In the game "Mansion" his control, which introduces a few action modifiers, etc.

So, you can do the menu in the inventory area, identifying objects with type 'menu'. In this case, the menu handler ('act') will be called after one click of the mouse. If the handler returns false, the game state does not change. For example, the implementation of the pocket:

```
menu {
    state = false;
```

```
    nam = 'pocket';
    disp = function(s)
        if s.state then
            return fmt.u('pocket'); -- emphasize active pocket
        end
        return 'pocket';
    end;
    gen = function(s)
        if s.state then
            s:open(); -- show all the items in the pocket
        else
            s:close(); -- hide all the items in the pocket
        end
        return s
    end;
    act = function(s)
        s.state = not s.state -- change state
        s:gen (); - to open or close the pocket
    end;
}: with {
    obj {
        nam = 'knife';
        inv = 'This is knife';
    };
}

function init()
 take 'pocket':gen()
end
```

## 23.3 The status of the player

Sometimes there is a desire displays some status. For example,the number of points the state of the hero or finally time days. INSTEAD does not provide any other areas of withdrawal, except scenes and inventory, therefore, the simplest way to output the status is output to the zone equipment.

Below is an implementation of player status as a text that appears in the in-

ventory, but cannot be selected, that is, looks just as text.

```
global {
    life = 10;
    power = 10;
}

stat { -- stat-object status
    nam = 'status';
    disp = function(s)
        pn ('Life: ', life)
        pn ('Power: ', power)
    end
};

function init()
    take the 'status'
end
```

## 23.4 walk from handlers to the onenter and onexit

You can do the 'walk' of the handlers 'the onenter' and 'onexit'. For example, 'path' implemented with a handler 'the onenter', which puts the player in the other room.

It is recommended to return from onexit/false in the onenter event, if you do the walk of these handlers.

## 23.5 Encoding of the source code of the game

If you do not want to show the source code of their games, you can to encode source code using the command-line option the '-encode':

```
sdl-instead-encode <file path> [output path]
```

And use the encoded file using the normal include/gamefile. However, for this you have to write at the beginning of main3.lua:

```
std.dofile = std.doencfile
```

The main file ‘main3.lua’ must be left open. This by the way, the diagram looks as follows (‘game.lua’ – encoded file):

```
-- $Name: My closed game!$
std.dofile = std.doencfile
include ”game”; -- no one knows how to pass it!
```

**Important!**

Do not use the following with ‘luac’, as ‘luac’ creates platform-dependent code! However, the following can be used to find errors in your code.

## 23.6 Boxing resources

You can package the game resources (graphics, music, themes) in the file resources ‘.idf’ placing all the resources in the directory ‘data’ and start INSTEAD:

```
sdl-instead-idf <path to data>
```

Thus, in the current directory will be created file ‘data.idf’. Place it in the directory with the game. Now resources game separate files can be removed (of course leaving himself the original files).

You can pack in format.idf’ the whole game:

```
sdl-instead-idf <path to game>
```

Game ‘idf’ can be run as a normal game ‘instead’ (as if it were a directory) and also from the command line:

```
sdl-instead game.idf
```

## 23.7 Switching between players

You can create a game with multiple characters and from time to time to switch between them (see ‘change_pl()’). But you can also to use this trick in order to be able to switch between different types of inventory.

## 23.8 Use settings handler

The code example.

```
obj {
    nam = 'stone';
    dsc = 'On the edge of lies {rock}.';
    act = function()
        remove 'stone';
        p 'I pushed the stone, it fell and flew down...';
end
```

Act handler could look simpler:

```
act = function(s)
    remove(s);
    p 'I pushed the stone, it fell and flew down...';
end
```

## 23.9 Special status handlers

Handler usually returns the text in the form return "text messages." Or with the functions p()/pr()/pn()/pf(). In addition, there are special statuses that can be useful when developing game.

The return status is false:

```
return false
```

This status means that the handler has not fulfilled its function and needsto be ignored. Typically, the engine in this case, it will call the handler by default.

You can also return a special status:

```
return true, false
```

In this mode only equipment (but not stage) will be redrawn. This status is useful for implementing menu the field inventory.

There is another special status: std.nop(). It can be used just like a function call at the end of the handler or together with the return.

```
return std.nop()
-- ... or ...
std.nop()
-- then the end of the function or return
```

In this case, the contents of the scene will remain the same as lastthe beat game (even string a reaction will be old). This status conveniently be used in conjunction with the module theme, when you need to change the design of the game on the fly and redraw the frame with the new parameters theme.

## 23.10 Timer

For asynchronous events bound to real time. it is possible to use the timer. In fact, you should have a good think about whether in the adventure game to use timer. Usually, it is not perceived too favorably. With the other hand, the timer can be used to control music or design purposes.

To use the timer, you should connect the module "timer".

```
require "timer"
```

The timer is programmed using the object "timer".

- timer:set(MS) – set timer interval in milliseconds;
- timer:stop() – stop the timer.

When the timer fires, calls a handler game.timer. If game.timer returns false, the scene is not redrawn. In otherwise, the return value is output as the response.

You can do local for room handlers 'timer'. If the room declared the handler "timer", it is invoked instead'game.timer'. If it returns false – the game is called.timer.

For example:

```
game.timer = function(s)
    if time() > 10 then
        return false
    end
    snd.play 'gfx/beep.ogg';
```

```
    p (”Timer:”, time())
end

function init()
    timer:set(1000) -- time in second
end

room {
    enter = function(s)
        timer:set(1000);
    end;
    timer = function(s)
        timer:stop();
        walk ’комната2’;
    end;
    nam = ’timer Test’;
    dsc = [[Wait.]];
}
```

As the timer gets to the save file, so you don’t you need to take care to restore it.

You can return from timer special status:

```
return true, false
```

In this mode only the area of inventory will be redrawn. It is possible use of statuses like hours.

In addition, INSTEAD there is the ability to track intervals the time in milliseconds. To do this, use the function instead.ticks(). The function returns the number of milliseconds elapsed since the start of the game.

## 23.11 Music player

You can write your game music player, created on the basis of a living object, for example:

```
-- playing tracks in random order
require ”snd”
```

```
obj {
    {
        tracks = {”mus/astro2.mod”,
            ”mus/aws_chas.xm”,
            ”mus/dmageofd.xm”,
            ”mus/doomsday.s3m”};
    };
    nam = ’player’;
    life = function(s)
        if not snd.music_playing() then
            local n = s.tracks[rnd(#s.tracks)]
            snd.music(n, 1);
        end
    end;
}:lifeon();
```

Below is an example of a more complex player. Change the track if it ended or it has been more than 2 minutes and the player goes from room to room. In each track you can specify the number of playbacks (0 - a looped track):

```
require ”timer”
global { track_time = 0 };

obj {
    nam = ’player’;
    pos = 0;
    {
        playlist = { ’01 Frozen sun.ogg’, 0,
            ’02 Thinking.ogg’, 0,
            ’03 Melancholy.ogg’, 0,
            ’04 Everyday happiness.ogg’, 0,
            ’10 Good morning again.ogg’, 1,
            ’15 [Bonus track] The end (demo cover).ogg’, 1
        };
    };
    tick = function(s)
        if snd.music_playing() and ( track_time < 120 or not player_moved() ) then
```

```
            return
        end
        track_time = 0
    if s.pos == 0 then
        s.pos = 1
    else
        s.pos = s.pos + 2
    end
        if s.pos > #s.playlist then
            s.pos = 1
        end
        snd.music('mus/'..s.playlist[s.pos], s.playlist[s.pos + 1]);
    end;
}

game.timer = function(s)
    track_time = track_time + 1
    music_player:tick();
end

function init()
    timer:set(1000)
end
```

## 23.12 Live objects

If your hero needs a friend, one of the ways can be a method 'life' of this character, which always moves the object to the location player:

```
obj {
    nam = 'horse';
    dsc = 'I am standing Next to {the horse}.';
    act = [[My horse.]];
    life = function(s)
        if player_moved() then
            place(s);
        end
```

```
    end;
}

function init()
    lifeon 'horse'; -- immediately revive the horse
end
```

## 23.13 Menu

You can call from the game menu INSTEAD of using the function'istead.menu()'. If the option ask: 'save', 'load' or 'quit', it will be caused by the relevant section on the menu.

## 23.14 Dynamic object creation

Ordinary objects and rooms cannot be created "on the fly". Usually you create them in the global namespace lua file. However, there are games in which the number of objects is unknown in advance, or the number objects large and they are added as the game progresses.

In INSTEAD, there is a way to create any object on the fly. For this you will need to write constructor your object and to use the function 'new'.

The constructor must be declared.

So, you can use the functions 'new' and 'delete' for creating and removal of dynamic objects. Examples:

```
declare 'box' (function()
    return obj {
        dsc = [[There is {box}.]];
        tak = [[I picked up a box.]];
    }
end)

local o = new (box);
take(o);

declare 'box' (function(dsc)
    return obj {
```

```
        dsc = dsc;
        tak = [[I picked up a box.]];
    }
end)
take(new(box 'In the corner {box}'))
```

'new' takes first argument is declared as a constructor function, and all other parameters-as arguments constructor. The result of the constructor should be object.

```
function myconstructor()
    local v = {}
    v.disp = 'test object'
    v.act = 'Test response'
    return obj(v)
end
```

> Attention! When the object is created the constructor should only rely on information derived from parameters! The fact that the creation of object when loading occurs in the beginning, when the environment of the world still not fully loaded! As parameters are supported simple types: strings, tables, numbers, Booleans values. Undeclared functions and lists will not work.

If you want to destroy the object by its name or the link-variable use:

```
purge(o) -- remove from all lists
delete(o) - release the object
```

In this case, delete this is the delete the object from INSTEAD, rather than analogue remove() or purge(). Usually, it makes little sense to do the delete. Only if the subject will never be needed in a game, or are you going to recreate an object with the same name, it makes sense to release it with using delete().

A more practically useful example:

```
-- declare the constructor path

declare 'make_path' (function(v) return path(v) end)
```

```
-- dynamic transition
-- create a new object
-- and put it in a ways()

put( new (make_path, { 'transition', 'комната2'}, ways())
```

## 23.15 The ban on saving a game

Sometimes you may want to forbid the player to do conservation in game. For example, if we are talking about scenes where an important element is case, or for short games where the loser needs to be fatal and require restarting the game.

To control the save function uses the attribute 'instead.nosave'.

For example:

instead.nosave = true – disable the preservation of

If you want to prevent save not everywhere, but in some scenes, make 'instead.nosave' to the view function or change the condition attribute on the fly – it gets to a file saves.

```
-- prohibit
-- save the rooms that contain the nosave attribute.
instead.nosave = function()
    return here().nosave
end
```

It should be noted that the ban on conservation does not mean prohibition AutoSave. To control the AutoSave, use the same attribute 'instead.noautosave'.

> You can explicitly save a game with a call to: 'instead.autosave([slot number])'; If the slot number is not specified, then the game will be saved under slot 'AutoSave'. Keep in mind that saves the state of **after** the completion of the current step of game.

## 23.16 Definition of an object type

In INSTEAD, there are two ways to determine the type of the object. First - using the function std.is_obj(variable, [type]).

For example:

```
a = room {
    nam = 'object';
};

dprint(std.is_obj(a)) -- will print true
dprint(std.is_obj('object')) -- will print false
dprint(std.is_obj(a, 'room')) -- will print true
dprint(std.is_obj(a.obj, 'list')) -- will print true
```

std.is_obj() is useful for determining the type of variable or argument function. The second method is to use the type method:

```
a = room {
    nam = 'object';
};

dprint(a:type 'room') -- will print true
```

# Chapter 24

# Topics for sdl-instead

Graphic interpreter supports the theme engine. Tema is a directory, the file ‘theme.ini’ inside.

The theme, which is the minimum required – is the theme ‘default’. This topic is always loaded first. All other topics inherit from it and can partially or completely replace it settings. Themes are chosen by the user via menu settings, but the game may contain its own subject and thus the influence on your appearance. In this case, the directory the game should be a file ‘theme.ini’. However, the user is free to disable this mechanism, while the interpreter will warn about violation of the creative concept the author of the game.

The syntax of ‘theme.ini’ is very simple.

```
<parameter> = <value>
```

or

```
; comment
```

Values can be of the following types: string, color, number.

The color is specified in the form #rgb where r g and b color components in hexadecimal. Additionally, some basic colorsrecognized by their names. Example: yellowgreen, or violet.

Parameters can take values:

- scr.w = width of the playing space, in pixels (number)
- scr.h = height of the playing space, in pixels (number)

- scr.col.bg = background color
- scr.gfx.scalable = [0/1/2] (0 - don't scale topic 1 - scalable 2 scalable without smoothing), since version 2.2.0 available optional [4/5/6]: 4 - fully scalable (non-scalable fonts), 5 - scalable, with non-scalable fonts, 6 - scaled without smoothing, with not a scalable font
- scr.scale_aware = [0|1|2] (adaptive themes support: 0 - virtual resolution, 1 - real proportions, 2 - real resolution)
- scr.dpi = [dpi|dpi1-dpi2] (theme dpi, 96 by default)
- scr.gfx.scale = scale (image scaling, 1.0 by default)
- scr.gfx.bg = path to the background image (string)
- scr.gfx.cursor.x = the x coordinate of the cursor center (number)
- scr.gfx.cursor.y = the y coordinate of the cursor center (number)
- scr.gfx.cursor.normal = path to the cursor picture file (string)
- scr.gfx.cursor.use = path to the cursor picture file usage (string)
- scr.gfx.use = path to the image-indicator of use (string)
- scr.gfx.pad = padding for scrollbars and menu edges (number)
- scr.gfx.x, scr.gfx.y, scr.gfx.w, scr.gfx.h = coordinates, width and the height of the window images. Region in which is located the picture scene. Interpretation depends on the mode of visit (number)
- win.gfx.h - synonymous with scr.gfx.h (for compatibility)
- scr.gfx.icon = path to the file-the icon of the game (OS dependent option, may to work correctly in some cases)
- scr.gfx.mode = mode location (line fixed, embedded or float). Sets the mode of the image. embedded-the picture is part of the content of the main window, the parameters of the scr.gfx.x, scr.gfx.y, scr.gfx.w are ignored. float-the picture located at the specified coordinates (scr.gfx.x, scr.gfx.y) and scales to the size scr.gfx.w x scr.gfx.if h exceeds it. fixed-the picture is part of a scene in embedded mode, but does not scroll along with text

and located directly above it. Available modifications mode modifiers float ‘left/right/center/middle/bottom/top’, specifies how to place an image in the field scr.gfx. For example: float-top-left;

- win.scroll.mode = [0/1/2/3] mode scrolling region of the scene. 0 - no auto-scroll 1 - scroll to change the text, 2 scroll to change only if the change is not visible, 3 - always in the end;

- win.x, win.y, win.w, win.h = coordinates, width and height of the main window. Region which is a description of the scene (number)

- win.fnt.name = path to the font file (string). Hereinafter, font may contain a description of all styles, for example: {sans,sans-b and sans-i,sans-bi}.ttf (the font style set to regular, bold, italic and bold-italic). You can omit some of the faces, and the engine itself will generate them based on the normal style, for example: {sans, sans-i}.ttf ( set only regular and italic);- win.align = center/left/right/justify (text alignment in the window scene);

- win.fnt.size = the font size of the main window (size)

- win.fnt.height = line spacing as the floating the decimal point (1.0 is default)

- win.gfx.up, win.gfx.down = paths to the image files skallerup up/down for the main window (string)

- win.up.x, win.up.y, win.down.x, win.down.y = coordinates. (coordinate or –1)

- win.col.fg = text color of the main window (color)

- win.col.link = link color for the main window (color)

- win.col.alink = active link color for the main window (color)- win.ways.mode = top/bottom (to specify the location of the jump list, default top – on top of the scene)

- inv.x, inv.y, inv.w, inv.h = coordinates, width and height of region inventory. (number)

- inv.mode = string mode inventory (horizontal or vertical). In horizontal inventory mode in one line can be several items. In vertical mode, each

line of the inventory contains only one thing. There are modifications to (-left/right/center). You can set the mode to disabled if your the game doesn't need an inventory;

- inv.col.fg = text color tools (color)- inv.col.link = link color tools (color)
- inv.col.alink = active link color for the inventory (the color)
- inv.fnt.name = path to font file the inventory (row)
- inv.fnt.size = the font size of the inventory (size)
- inv.fnt.height = line spacing as the floating the decimal point (1.0 is default)
- inv.gfx.up, inv.gfx.down = paths to the image files skallerup up/down of equipment (line)
- inv.up.x, inv.up.y, inv.down.x, inv.down.y = coordinates. (coordinate or –1)
- menu.col.bg = background (color)
- menu.col.fg = text color menu (color)- menu.col.link = menu link color (color)
- menu.col.alink = active link color menu (color)
- menu.col.alpha = the transparency of the menu 0–255 (number)
- menu.col.border = border color menu (color)
- menu.bw = thickness of the menu border (number)
- menu.fnt.name = path to the font file menu (string)
- menu.fnt.size = the font size menu (size)
- menu.fnt.height = line spacing as the floating the decimal point (1.0 is default)
- menu.gfx.button = the file path to the icon image menu (string)
- menu.button.x, menu.button.y = coordinates of the menu buttons (number)
- snd.click = the path to the click sound file (string)

- include = theme name (the last component in the directory path) (string)

In addition, the subject header may include comments with tags. At the moment there is only one tag: $Name:, containing UTF–8 string with the theme name. For example:

```
; $Name:New theme$
; modification themes book
include = the book-to use the theme of the Book
scr.gfx.h = 500 -- replace it one parameter
```

> The interpreter searches for themes in the "themes" directory. Unix version in addition to this directory, scan also the directory ∼/.instead/themes/Windows version – Documents and Settings/USER/Local Settings/Application Data/instead/themes

In addition, the new version INSTEAD support multiple those in one game. Allowing the player via the standard menu INSTEAD choose a suitable appearance, provided by the author game. For this, all threads should be in the game in a subdirectory themes. In turn, each theme – is a subdirectory in the directory themes. In each such subdirectory should be a file theme.ini and theme resources (images, fonts, sounds). In this casea themes catalog themes/default - this theme will loaded by default. The format of the theme files.ini we just considered. However, the file paths to resources in theme.ini file are not written relative to the root directory of the game, and relative to the current directory theme. This means that they usually contain only the name of the file without the directory path. For example:

```
mygame/
  themes/
    default/
      theme.ini
      bg.png
    widescreen/
      theme.ini
  main3.lua
```

theme.ini

```
scr.gfx.bg = bg.png
; ...
```

In this case, all themes in the game are inherited from the themethemes/default. Supported mechanism include. At the same time, INSTEAD first tries to find the theme of the game, and if such topics are not is will downloaded the theme from the standard themes INSTEAD (if it exist). Further, in theme.ini you can only change those settings changes were required.

# Chapter 25

# Modules

Additional functionality is often implemented INSTEAD in the form modules. To use a module, you must write:

```
require ”module name”
```

Or:

```
loadmod ”module name”
```

If the module shipped with the game.

Modules included INSTEAD, but there are those that you can download separately and put in the directory with the game. You can replace any standard module as its if you put it in the directory game under the same file name as the standard.

The module is actually ‘lua’ file name: ‘module_name.lua’.

Following are the basic standard modules, indicating the functionality that they provide.

- ‘dbg’ module debugging.
- ‘click’ — module intercept mouse clicks on the image of the scene;
- ‘prefs’ module settings (data warehouse settings);
- ‘snapshots’ — a plugin to support snapshots (for kickbacks of game situations);
- ‘fmt’ module of withdrawal;

- ‘theme’ — the office theme on the fly;
- ‘noinv’ module to operate the equipment;
- ’key” - module event processing of the operation keys;
- ‘timer’ - timer;
- ‘sprite’ — module for working with sprites;
- ‘snd’ module audio;
- ‘nolife’ module locking methods life;

Example load the modules:

```
--$Name: My game!$
require ”fmt”
require ”click”
```

> Some additional modules which are not included in the standard supply, you can download it from repository modules[1]. Just download the desired module and put it in the directory the game. Enable this module by using the loadmod().

## 25.1 Module keys

You can intercept keyboard events using the module keys.

Usually, the interception of keys makes sense for text input.

To specify what keys you want to monitor define the function keys:filter(press, key). This function must to return true if you want to track this event. For example:

```
require ”keys”

function keys:filter(press, key)
    press return -- catch pressing any keys
end
```

[1] https://github.com/instead-hub/stead3-modules

In the example we simply return the parameter, press which is true for keypress events and false – squeezing. Key is passed the symbolic name of a key (as a string).

Usually, we need to choose what keys we want to intercept:

```
require "keys"

function keys:filter(press, key)
    if key == '0' or key == '1' or key == 'z' then
        press return -- catch keystrokes z, 1, and 0
    end
end
```

So, keys:filter allows one to choose keyboard events. And doevents come into play in the form of call handler 'onkey' for the current room or (if not specified in the room) object 'game'.

The onkey handler acts as a normal handler STEAD3. It can to return false and then it is considered that the key was not processed game. Or it can perform some action.

*Внимание:* If the game will handle all key events, even those combinations that are used by the INTERPRETER are processed the game, not the interpreter. For example, if the game will catch press escape (and return false from the handler), pressing"escape" will no longer be handled by the interpreter INSTEAD (escape – the call menu).

Below is a simple example to display the symbolic names of keys:

```
require "keys"

function keys:filter(press, key)
    press return -- catch all the clicking
end

game.onkey = function(s, press, key)
    dprint("pressed: ", key)
    p("Pressed: ", key)
    return false -- allow to handle the keys to the interpreter INSTEAD
end
```

This example can be used to determine the symbolic the name of the specific keys.

When writing arcade games can be useful not get event fromthe keyboard and scan it (usually the timer). To do this, you can use the function keys:state(name key).

This function returns true for pressed and false – for pressed, for example:

```
require ”timer”
require ”keys”

-- display the state of the key the cursor to the right
game.timer = function(s)
    dprint(”state of ’right’ key: ”, keys:state ’right’)
    p(”As keys ’right’:”, keys:state ’right’)
end

timer:set(30)
```

## 25.2 Module click

You can track in your game clicks on the picture of the scene, and background. To do this, use the module “click”. Also, you can keep track of the mouse state by the function:

```
instead.mouse_pos([x, y])
```

Which returns the coordinates of the cursor. If you set the parameters (x, y), you can move the cursor to the specified position (all coordinates are calculated relative to the upper left corner of the window INSTEAD).

```
require ”click”
function click:filter(press, btn, x, y, px, py)
    dprint(press, btn, x, y, px, py)
    return press and px -- only catch clicking on the picture
end
room {
    nam = ’main’;
    pic = ”box:320x200,red”;
```

```
    onclick = function(s, press, btn, x, y, px, py)
        pn(”You pressed the image: ”, px, ”, ”, py)
        pn(”Absolute coordinates: ”, x, ”, ”, y)
        p(”Button: ”, btn)
    end;
}
```

*Внимание!* You can INSTEAD use the default filter mouse clicks, which extinguishes quick clicks. This is done to eliminate the effects of bounce buttons on a mouse. In some cases, the filter may be undesirable. In this case, use the function instead.mouse_filter(), which can be used for determine the current filter values of the mouse and install new ( the number - off), for example:

```
function start()
    dprint(”Mouse filter delay: ”, instead.mouse_filter())
    instead.mouse_filter(0) -- turn off the filter
end
```

Or this:

```
old_filter = instead.mouse_filter(0) -- turn off
...
instead.mouse_filter(old_filter) -- restored
```

## 25.3 The module theme

The theme module allows you to change the theme settings on the fly.

> Keep in mind that changing the theme settings on the fly for games which does not contain its own subject-a source of potential problems! The fact that your game needs to be ready to work with any permissions and settings the fact that it is extremely difficult to achieve. So if you are going to change the theme settings from code – create your own theme and turn it into a game!

In this case, the saving changes topic to save file not supported. The author of the game should restore the theme settings in function start () as you do when working with the module sprites.

To change the settings of the current theme, use the following functions:

```
-- configuration of the output window
theme.win.geom(x, y, w, h)
theme.win.color(fg, link, alink)
theme.win.font(name, size, and height)
theme.win.gfx.up(pic, x, y)
theme.win.gfx.down(pic, x, y)

-- configuration inventory
theme.inv.geom(x, y, w, h)
theme.inv.color(fg, link, alink)
theme.inv.font(name, size, and height)
theme.inv.gfx.up(pic, x, y)
theme.inv.gfx.down(pic, x, y)
theme.inv.mode(mode)

-- configuration menu
theme.menu.bw(w)
theme.menu.color(fg, link, alink)
theme.menu.font(name, size, and height)
theme.menu.gfx.button(pic, x, y)

-- configuration graphics
theme.gfx.cursor(norm, use, x, y)
theme.gfx.mode(mode)
theme.gfx.pad(pad)
theme.gfx.bg(bg)

-- sound settings
theme.snd.click(name);
```

There is the ability to read the current settings:

```
theme.the 'get' variable name themes';
```

The return value is always in text form.

```
theme.set ('variable name theme', value);
```

You can reset the value of the parameter of the topic, which was installed in the built-in theme of the game:

```
theme.reset 'variable name';
theme.win.reset();
```

There is a function, in order to know the current selected theme.

```
theme.name()
```

The function returns a string – the directory name of the theme. If the game uses own the file 'theme.ini', the function will return a point. This is useful for determine whether the mechanism of its own for those games:

```
if theme.name() ~= '.' then
    error "Please enable own theme mode in menu!"
end
```

If the game uses the mechanism of multiple themes, then the theme name starts with a dot, for example:

```
if theme.name() == '.default' then
    -- our built-in default theme
elseif theme.name() == 'default' then
    -- standard default theme.
end
```

Example usage:

```
theme.gfx.bg "dramatic_bg.png";
theme.win.geom (0,0, theme.get 'scr.w' theme.get 'scr.h');
theme.inv.mode 'disabled'
```

Get the dimensions of the current theme:

```
theme.scr.(w) -- width
theme.scr.w () - height
```

## 25.4 The module sprite

The sprite module allows to work with graphic images.To enable the email module:

```
require ”sprite”
```

Sprites can’t get into the save file so that recovery status of sprites-the task of the author of the game. Generally, this use the functions init() or start(). start() is called after download the game, so in this function you can use the variables of the game.

In fact, in module sprite implements two modules: sprites and pixels. But since they are used together, they are placed in one module. Let’s start with the sprites:

### 25.4.1 Sprites

To create a sprite, use the sprite.new, for example:

```
declare ’my_spr’ (sprite.new ’gfx/bird.png’)
local heart = sprite.new ’heart.png’
local blank = sprite.new (320, 200) -- an empty sprite 320x200
```

Have created a sprite object has the following methods:

- :alpha(alpha) - creates a new sprite with specified opacity alpha (255 means transparent). This is a very slow function;
- :dup() - creates a copy of the sprite;
- :scale(xs, ys, [smooth]) – scales sprite for reflections using scale –1.0 ( slow! not for real time). Creates a new sprite.
- :rotate(angle [, smooth]) – rotates the sprite through a specified angle in the degrees (slowly! not for real time). Creates a new sprite.
- :size() – Returns the width and height of the sprite in pixels.
- :draw(fx, fy, fw, fh, dst_spr, x, y, [alpha]) – Drawing the field src of the sprite in the sprite area dst (job alpha much slows down the execution of the function).

- :draw(dst_spr, x, y, [alpha]) – drawing the sprite, cropped option; (job alpha slows down the execution of the function).
- :copy(fx, fy, fw, fh, dst_spr, x, y) – Copying rectangle fw-fh from the sprite in the sprite dst_spr by coordinates [x,y] (drawing substitution). There a shortened version (as :draw).
- :compose(fx, fy, fw, fh, dst_spr, x, y) – Drawing with given the transparency of the two sprites). There is a shortened version (as in :draw).
- :fill(x, y, w, h, [col]) – Fill sprite with a color.
- :fill([col]) – Fill sprite with a color.
- :pixel(x, y, col, [alpha]) - Filling the pixel of a sprite.
- :pixel(x, y) – the capture of the pixel of the sprite (returns four color component).
- :colorkey(color) – Sets the sprite color, which acts the role of a transparent background. Thus, during subsequent operation :copy from this sprite will be copied only those pixels whose color doesn't match the background transparent.

As the "color" methods receive string: 'green', 'red', 'yellow' or '#333333', '#2d80ff'…

Example:

```
local spr = sprite.new(320, 200)
spr:fill 'blue'
local spr2 = sprite.new 'fish.png'
spr2:draw(spr, 0, 0)
```

In addition, there is the possibility of working with fonts. The font is created with the help of sprite.fnt(), for example:

```
local font = sprite.fnt('sans.ttf', 32)
```

Have created object defines the following methods:

- :height() – height of the font in pixels;

- :text(text, col, [style]) – create a sprite from text col - here the color in the text format (in the format `#rrggbb' or `text color').
- :size(text) – computes the size the text will occupy sprite, without creating a sprite.

You may also find it useful to:

```
sprite.font_scaled_size(size)
```

It returns font size taking into account the scaling that put the player in the settings INSTEAD. If you are in the game you want note this setting use this function to determine the size of the font.

Example:

```
local f = sprite.fnt('sans.ttf', 32)
local spr = sprite.new('box:320x200,black')
f:text("HELLO!", 'white'):draw(spr, 0, 0)
```

Now, consider the application of the module sprite.

### 25.4.2 Function pic

A function can return a pic for a sprite. You can shape every time new sprite (which is not very efficient), or may return preallocated sprite. If the sprite changes, these changes will be reflected in the next frame of the game. So, changing the sprite the timer to do the animation:

```
require "sprite"
require "timer"
local spr = sprite.new(320, 200)

function game:timer()
    local col = { 'red', 'green', 'blue'}
    col = col[rnd(3)]
    spr:fill(col)
    return false -- Important! So, the scene will not be changed
end
```

```
game.pic = function() return spr end -- function: as
-- sprite is a special object (not a string)

function start()
    timer:set(30)
end

room {
    nam = 'main';
    decor = [[HYPNOSIS]];
}
```

### 25.4.3 Rendering in the background

Function sprite.scr() returns the sprite - background. You can perform render this sprite in any handler, for example, in the timer. The most seeking to change background on the fly, without the use of the module theme. For example:

```
--$Author: Andrew Lobanov

require 'sprite'
require 'theme'
require 'timer'

declare {
    x = 0,
    y = 0,
    dx = 10,
    dy = 10,
}

const {
    w = theme.scr.w(),
    h = theme.scr.h(),
}

instead.fading = false
```

```
local bg, red, green

function init()
    theme.set('scr.col.bg', '#000000')
    theme.set('win.col.fg', '#aaaaaa')
    theme.set('win.col.link', '#ffaa00')
    theme.set('win.col.alink', '#ffffff')

    bg = sprite.new(w, h)
    bg:fill('black')
    red = sprite.new(w, h)
    red:fill('#ff0000')
    red = red:alpha(128)
    green = sprite.new(w, h)
    green:fill('#00ff00')
    green = green:alpha(64)
    bg:copy(sprite.scr())
    timer:set(25)
end

function game:timer()
    bg:copy(sprite.scr())
    red:draw(sprite.scr(), x, 0, 128)
    green:draw(sprite.scr(), 0, y, 64)
    x = x + dx
    if x >= w or x == 0 then
        dx = -dx
    end
    y = y + dy
    if y >= h or y == 0 then
        dy = -dy
    end
    return false -- Important!
end

room {
    nam = 'main',
    disp = 'Test. Test? Test!',
```

```
    decor = 'Lorem ipsum';
}
```

*Caution!* Interpreter INSTEAD to use item on the object transforms itself into the mode "pause". This means that at the moment when the selected item from inventory (the cursor change gear) timer events will not be processed until the until the player apply the subject. This is done in order to not break the beat game. If your creative idea this is an obstacle (for example, you don't like the fact that the animation background freezes), you can change it by calling:

```
instead.wait_use(false)
```

As usual, place the call in the init() or start() function.

### 25.4.4 Lookup

You can create your system object - substitution, and forming graphics output of the game, with img, for example:

```
require "sprite"
require "timer"
require "fmt"

obj {
    nam = '$spr';
    {
        ["square"] = sprite.new 'box:32x32,red';
    };
    act = function(s, w)
        return fmt.img(s[w])
    end
}

room {
    nam = 'main';
    decor = [[Now we insert the sprite: {$spr|square}.]];
}
```

### 25.4.5 direct mode

In INSTEAD, there is direct access to the chart. In the subject he is specified using the parameter:

```
scr.gfx.mode = direct
```

This option can be pre-set theme.ini, or use module theme. Or (better) special function:

```
sprite.direct(true)
```

If the regime managed to include – the function will return true. sprite.direct() without the parameter – returns the current mode (true-if direct included.)

In this mode, the game has direct access to the entire window and can perform the rendering in the procedure of the timer. The screen presents a special sprite:

```
sprite.scr()
```

For example:

```
require ”sprite”
require ”timer”
require ”theme”

sprite.direct(true)

local stars = {}
local w, h
local colors = {
    ”red”,
    ”green”,
    ”blue”,
    ”white”,
    ”yellow”,
    ”cyan”,
    ”gray”,
    ”#002233”,
}
```

```
function game:timer()
    local scr = sprite.scr()
    scr:fill 'black'
    for i = 1, #stars do local s = stars[i]
        scr:pixel(s.x, s.y, colors[s.dy])
        s.y = s.y + s.dy
        if s.y >= h then
            s.y = 0
            s.x = rnd(w) - 1
            s.dy = rnd(8)
        end
    end
end

function start()
    w, h = theme.scr.w(), theme.scr.h()

    w = std.tonum(w)
    h = std.tonum(h)

    for i = 1, 100 do
        table.insert(stars, { x = rnd(w) - 1, y = rnd(h) - 1, dy = rnd(8) })
        end
    timer:set(30)
end
```

Another example:

```
require "timer"
require "sprite"
require "theme"

local spr = sprite

declare {
    fnt = false, ball = false, ballw = 0,
    ballh = 0, bg = false, line = false,
```

```
    G = false, by = false, bv = false,
    bx = false, t1 = false,
}

function init()
    fnt = spr.fnt(theme.get 'win.fnt.name', 32);
    ball = fnt:text("INSTEAD of 3.0", 'white', 1);
    ballw, ballh = ball:size();
    bg = spr.new 'box:640x480,black';
    line = spr.new 'box:320x8,lightblue';
    spr.direct(true)
end

function start()
    timer:set(20)
    G = 9.81
    by = -ballh
    bv = 0
    bx = 320
    t1 = instead.ticks()
end

function phys()
    local t = timer:get() / 1000;
    bv = bv + G * t;
    by = by + bv * t;
    if by > 400 then
        bv = - bv
    end
end

function game:timer(s)
    local i
    for i = 1, 10 do
        phys()
    end
    if instead.ticks() - t1 >= 20 then
        bg:copy(spr.scr(), 0, 0);
```

```
        ball:draw(spr.scr(), (640 - ballw) / 2, by - ballh/2);
        line:draw(spr.scr(), 320/2, 400 + ballh / 2);
        t1 = instead.ticks()
    end
end
```

*Caution!* direct mode can be used to create a simple arcade games. In some cases, you may want to remove the pointer mouse. For example, when the game is controlled only with keyboard.

To do this, use function instead.mouse_show()

```
instead.mouse_show(false)
```

On the menu of the interpreter INSTEAD of the mouse pointer will still visible.

### 25.4.6 Using the sprite together with the module theme

In start and in the handler you can change the theme settings in with as graphics the sprites, for example:

```
require ”sprite”
require ”theme”

function start() -- replace the background sprite
    local spr = sprite.new(800, 600)
    spr:fill ’blue’
    spr:fill (100, 100, 32, 60, ’red’)
    theme.set(’scr.gfx.bg’, spr)
end
```

Using this technique, you can apply to the background image statuses, controls, or just change the substrate.

Take note that in this case `theme.get(’scr.gfx.bg’)` call returns a string similar to `spr:xxxxxxxxxxx`, and not a sprite object. If you want to change background dinamically, use `sprite.scr()` or multiple calls to `theme.set(’scr.gfx.bg’, spr)` with new sprites.

### 25.4.7 Pixels

The sprite module also supports work with pixel graphics. You can create objects – sets of pixels, modify them and to draw the sprites.

Creating pixels is a function of pixels.new().

Examples:

```
local p1 = pixels.new(320, 200) -- has created a 320x200 pixels
local p2 = pixels.new 'gfx/apple.png' -- created pixels from image
local p3 = pixels.new(320, 200, 2) -- created a 320x200 pixels,
-- that when you render them to a sprite -- will be scaled to
-- 640x400
```

The pixels object has the following methods:

> in the description of used symbols: r, g, b, a – components of a pixel: red, green, blue, and transparency. All values from 0 to 255). x, y coordinates of the upper left corner, w, h – width and height of the area.

- :size() – return the size and scale (3 values);
- :clear(r, g, b, [a]) – quick cleanup of pixels;
- :fill(r, g, b, [a]) – fill (with transparency);
- :fill(x, y, w, h, r, g, b, [a]) – fill area (with transparency);
- :val(x, y, r, g, b, a) - set pixel value
- :val(x, y) – to obtain the components r, g, b, a
- :pixel(x, y, r, g, b, a) – draw a pixel (taking into account transparency of the existing pixel);
- :line(x1, y1, x2, y2, r, g, b, a)–;
- :lineAA(x1, y1, x2, y2, r, g, b, a) – line AA;
- :circle(x, y, radius, r, g, b, a) – circle;
- :circleAA(x, y, radius, r, g, b, a) is the circle with AA;

- :poly({x1, y1, x2, y2, …}, r, g, b, a) - polygon;
- :polyAA({x1, y1, x2, y2, …}, r, g, b, a) - polygon with AA;
- :blend(x1, y1, w1, h1, pixels2, x, y) – draw a pixel area in another object pixels, full form;
- :blend(pixels2, x, y) – short form;
- :fill_circle(x, y, radius, r, g, b, a) – filled circle;
- :fill_triangle(x1, y1, x2, y2, x3, y3, r, g, b, a) – filled triangle;
- :fill_poly({x1, y1, x2, y2, …}, r, g, b, a) – filled polygon;
- :copy (…) - as a blend, but not to draw, and to copy (quickly);
- :scale(xscale, yscale, [smooth]) – scale the new object pixels;
- :rotate(angle [, smooth]) – the turn of the new object pixels;
- :draw_spr(…) – like draw, but in the sprite, not the pixels;
- :copy_spr(…) – like copy, but in the sprite, not the pixels;
- :compose_spr(…) – same thing, but in compose mode;
- :dup() – duplicate pixels;
- :sprite() – create a sprite with pixels.

Also, it is possible to work with fonts:

- pixels.fnt(fnt(font.ttf, size) – create a font;

In this case, the created object is “font” there is the text method:

- :text(text, color(as in sprites), style) – create pixel text;

For example:

```
local fnt = pixels.fnt(”sans.ttf”, 64)
local t = fnt:text(”HELLO, INSTEAD!”, ’black’)
pxl:copy_spr(sprite.scr())
pxl2:draw_spr(sprite.scr(), 100, 200);
t:draw_spr(sprite.scr(), 200, 400)
```

Another example (by example, Andrei Lobanov):

```
require ”sprite”
require ”timer”

sprite.direct(true)

declare ’pxl’ (false)
declare ’t’ (0)

function game:timer()
    local x, y, i
    t = t + 1
    for x = 0, 199 do
        for y = 0 to 149 do
            i = (x * x + y * y + t)
            pxl:val(x, y, 0, i, i / 2)
        end
    end
    pxl:copy_spr(sprite.scr())
end

function start(load)
    pxl = pixels.new(200, 150, 4)
    timer:set(20)
end
```

When the procedural generation using pixels it is conveniently to use Perlin noise. In INSTEAD there are functions:

- instead.noise1(x) - 1D Perlin noise;
- instead.noise2(x, y) - 2D Perlin noise;

- instead.noise3(x, y, z) is a 3D Perlin noise;
- instead.noise4(x, y, z, w) - 4D Perlin noise;

All these functions return a value in the range [−1; 1] and at the entrance get the coordinates of the floating point.

## 25.5 Module snd

We have already discussed the basic features to work with sound. Module snd has some functions for working with sound.

You can load the sound and keep it in memory as long as he you need.

```
require 'snd'
local wav = snd.new 'bark.ogg'
```

Besides uploading files, you can load the sound from an array of lua:

```
local wav = {}
for i = 1, 10000 do
    table.insert(wav, rnd() * 2 - 1) -- random values from -1 to 1
end

function start()
    -- frequency, number of channels and sound
    local p = snd.new(22050, 1, wave)
    p:play()
end
```

The sound is specified in normalized format: [−1 .. 1]

The sound you can play method :play([chan], [loop]), where chan-channel (0 - 7), loop - cycles (0 - infinity).

Other functions of the module:

- snd.stop([channel]) – to stop playback of the selected channel or all channels. The second parameter you can set the decay time of the sound in MS. when it is muted;

- snd.playing([channel]) – to know whether the sound is playing on any channel or on the selected channel; if the selected specific channel the function will return handle the currently played sound or nil. Attention! The sound of a click is not taken into account and usually takes 0 channel;
- snd.pan(chan, l, r) – set panning. Channel, volume left[0–255], the volume of the right[0–255] channels. You must call before playing the sound to have effect;
- snd.vol(vol) – sets the volume of sound (music and effects) from 0 to 127.

Another interesting possibility – sound generation on the fly (yet is in experimental status):

```
require ”snd”

function cb(hz, len, data)
    for i = 1, len do
        data[i] = rnd() * 2 - 1
    end
end
function start()
    snd.music_callback(cb)
end
```

## 25.6 The module prefs

This module allows you to save game settings. In other words, the saved information does not depend on the state of the game. Such a mechanism can be used, for example, to implement an achievements system or the count of the number of passages of play.

Essentially prefs is an object, all the variables which will be saved.

Save settings:

```
prefs:store()
```

The settings are automatically saved when you save the game, but you can to control this process, causing the prefs:store().

Destroy configuration file:

```
prefs:purge()
```

To download settings automatically when you start the game (before you call the start () function), but you can initiate the download and manually:

```
prefs:load()
```

Example usage:

```
-- $Name: Test module prefs$
-- $Version: 0.1$
-- $Author: instead of$

-- plug-in click
require ”click”
-- the plug-in prefs
require ”prefs”

-- set starting value of the counter
prefs.counter = 0;

-- define a function for tracking the number of ”clicks”
game.onclick = function(s)
    -- increment counter
    prefs.counter = prefs.counter + 1;
    -- persistent counter
    prefs:store();
    -- print message
    p(”currently done ”, prefs.counter” clicks”);
end;

-- added image by which to produce clicks
game.pic = ’box:320x200,black’;

room {
    nam = ’main’,
    title = ”house of clicks”,
    -- make the static parts of the description
```

```
    -- added description for the scene
    decor = [[ This test was written specifically
            to check the operation of the module <<prefs>>.
    ]];
};
```

Please note that after starting the game again, the number of made clicks don't reset!

## 25.7 Module snapshots

Module snapshots provides the ability to restore the previously saved game state. As an example, it is possible to bring the situation when the player performs in the game action, leading to loss. The module allows the author to write the game code so that the player will return to the previously saved state of the game.

To create a snapshot use: snapshots:make(). In the parameter can be set to the name of the slot.

*Caution!!!* Snapshot will be generated after completing the current step of game, because only in this case, the guaranteed consistency the saved game state.

Is download snapshot snapshots:restore(). As parameter can be set to the name of the slot.

Snapshot deletion is done using snapshots:remove(). Should to remove unnecessary snapshots, since they take up extra space in files save.

Example usage:

```
require "snapshots"

room {
    nam = 'main';
    title = 'Game';
    the onenter = function()
    snapshots:make () - created a restore point
    end;
    decor = [[{#red|Red} or {#black|black}?]];
}: with {
    obj {
        nam = '#red';
        act = function()
```

```
            p [[You won!]]
        end;
    };
    obj {
        nam = '#black';
        act = function()
            walk 'end'
        end;
    }
}

room {
    nam = 'end';
    title = 'End';
}: with {
    obj {
        dsc = [[{Replay?}]];
        act = function()
            snapshots:restore() -- recovered
        end;
    }
}
```

# Chapter 26

# Object methods

All objects STEAD3 there are methods that are used in the implementation of the standard library and usually are not used by the author games directly. However, it is sometimes useful to know the composition of these methods, though would be to not name your variables and methods names already of the existing methods. Below are a list of methods with a brief description.

## 26.1 Object (. obj)

- :with({…}) - set list obj;
- :new (…) constructor;
- :actions(type, [value]) - set/read the number of event object the preset type;
- :inroom([{}]) - in which room (room) the object is located;
- :where([{}]) - what the object (objects) is a object;
- :purge() - removes the object from all lists.
- :remove() - remove object from all objects/rooms/equipment;
- :close ()/open() - close/open;
- :disable()/:enable() - disable/enable;

- :closed() – returns true if closed;
- :disabled() – returns true if disabled;
- :empty() – returns true if empty;- :save(fp, n) – save function;
- :display() – display function in the scene;
- :visible() – returns true if it is visible;
- :srch(w) - the search for the visible object;
- :lookup(w) - search any object;
- :for_each(fn, …) – iterator over the objects;
- :lifeon()/:lifeoff() – add/remove from the list of living;
- :live() – returns true if the list alive.

## 26.2 Room (room)

In addition to the methods of obj, added the following methods:

- :from() – where I came into the room;
- :visited() – was there a room you visited earlier?;
- :visits() – number of visits (0-if not);
- :scene() – display scene (no objects);
- :display() – display of objects in the scene;

## 26.3 Dialogs (dlg)

In addition to the methods room, added the following methods:

- :push(phrase) - go to phrase, remembering it in the stack;
- :reset(phrase) - same, but reset the stack;

- :pop(phrase (section 16.1)) – return stack;
- :select(phrase (section 16.1)) – select current phrase;
- :ph_display() – display the selected phrase;

## 26.4 The game world (game object)

In addition to the methods of obj, added the following methods:

- :time([v]) – set/get the number of game cycles;
- :lifeon([v])/:lifeoff([v]) – add/remove an object from the list alive, or enable/disable live listings globally (if not specified argument);
- :live([v]) – check the activity of the life object;
- :set_pl(pl) – switch player
- :life () - the iteration of live objects;
- :step() – tact of the game;
- :lastdisp([v]) - set/get the latest output;
- :display(state) – display output;
- :lastreact([v]) - set/get the latest reaction;
- :reaction([v]) – set/get current response;
- :events(pre, bg) – set/get events of live objects;
- :cmd(cmd) – the command INSTEAD;

## 26.5 Player (player)

In addition to the methods of obj, added the following methods:

- :moved () - the player made a move in the current cycle of the game;

- :need_scene([v]) - need to render the scene in this cycle;
- :inspect(w) - find an object (visible) in the current scene or really;
- :have(w) - search in the inventory;
- :useit(w) - use item;
- :useon(w, ww) – use the item on the subject;
- :call(m, …) – calling a method of the player;
- :action(w) - action on the subject (act);
- :inventory() – return the inventory (list, default is obj);
- :take(w) – take the object;
- :walk/walkin/walkout – transitions;
- :go(w) - team go (checking the availability of transitions);

# Chapter 27

# List of attributes and handlers

Each object has a set of attributes and handlers associated with it. Most often attributes and handlers can be specified as text strings or functions. But some attributes are set in a special way. For example, the list of embedded objects is set as a ‘obj’ list {}, and ‘noinv’ is a boolean value.

The main difference between an attribute and a handler is that the handler is a reaction on a game event. But attributes are used by the engine while generating a description of the scene and inventory.

If the handler or attribute is set as a function, the first parameter (s) is always the object itself. Function-attributes MUST NOT change state of the game world. This can be done in handlers only.

A list of attributes and handlers is provided below for reference purposes.

## 27.1 Objects and rooms (obj, room)

- nam - attribute;
- tag - attribute;
- ini - handler, called for an object /room during construction game world, can only be a function;
- dsc - attribute, called to output the description;
- disp - attribute, information about an item in the inventory or a room in way list;

- title - attribute of the room, the name of the room displayed when found inside this room;
- decor - attribute of the room, called to display a description of the static decorations in scene;
- nolife - room attribute, do not call live object handlers;
- noinv - room attribute, do not show inventory;
- obj - attribute, list of nested objects;
- way - attribute of room, list of rooms to walk;
- life - handler, called for “live” (background) objects;
- act - object handler, called when acting on a scene object;
- tak - handler for taking an object from the scene (if act is not specified);
- inv - object handler, called when acting on an inventory object;
- use(s, on what) - object handler, called when using inventory item on another item;
- used (s, that) - object handler, called before use when using this item (passive form);
- onenter(s, from where) - room handler, called when entering the the room, can cancel when returning false;
- enter(s, where) - room handler, is called after a successful entering the room;
- onexit(s, where) - room handler, called when leaving room, can cancel when returning false;
- exit(s, where) - room handler, is called after a successful exit from the room.

## 27.2 Game world (game)

- use (s, what, on what) - handler, default action for the use of the object;
- act(s, that) - handler, default action while object click;
- inv(s, that) - handler, default action on inventory click;
- on{use, act, tak, inv, walk} - handler, response before call appropriate handlers, can cancel the chain;
- after{use, act, tak, inv, walk} - handler, reaction after plaayer action.

# Chapter 28

# Epilogue

So that's where the documentation ends. But maybe starts the most interesting – your story!

I made the first version of INSTEAD in 2009. At that time, I never thought that my toy (and the engine) would have survived so many changes. Text adventures still exist as I write this epilogue in 2017. Even so, their impact on our culture remains minimal, and well-written adventures are far and few between.

Games featuring a mixture of text and graphics have huge potential in my opinion. Though they are less interractive, they are also less demanding of the player. Enjoying a text adventure will not consume days of your life spent in front of the monitor suffering frustration or unhealthy anxiety as your desires are left unsatisfied. Textual games borrow from the best that the two worlds of liturature and computer gaming have to offer. As a bonus, the greater part of games in this genre can be enjoyed without cost.

The history of INSTEAD is, in my opinion, proof of this claim. Many games have been released for INSTEAD which can safely be called great! Their authors may retire, but they created works are already living their lives, reflected in the consciousness of people, who play them or remember. Let the "circulation" of these games is not so great but what I saw completely "justified" all efforts spent on engine. I know it's time well spent. So I found a power, and made the engine even better, releasing STEAD3. I hope he and like you.

So if you've read this far, I can only wish you to add your first story. Creativity – this is freedom. :)

Thank you and good luck. Peter Skew, of March 2017.