Руководство INSTEAD

Петр Косых

24.01.13

Оглавление

0. Общие сведения

Код игр под INSTEAD пишется на *lua (5.1 или 5.2)*, поэтому, знание этого языка полезно, хотя и не необходимо. Ядро движка также написано на lua, поэтому знание lua может быть полезным для углубленного понимания принципов его работы, конечно, при условии, если вам интересно этим заниматься.

За время своего развития, INSTEAD получил множество новых функций. Теперь с его помощью можно делать игры разных жанров (от аркад, до игр с текстовым вводом). А также, в INSTEAD можно запускать игры, написанные на некоторых других движках, но основой INSTEAD остается первоначальное ядро, которое ориентировано на создание текстографических приключенческих игр. В данной документации описана именно эта базовая часть, изучение которой необходимо даже в том случае, если вы хотите написать что-то другое... Начните свое изучение INSTEAD с написания простой игры!

Внимание!

Самая свежая документация всегда находится по адресу:
http://instead.syscall.ru/wiki/ru/gamedev/documentation

В формате pdf:

http://instead.googlecode.com/svn/trunk/doc/instead-manual.pdf
http://instead.googlecode.com/svn/trunk/doc/instead-modules.pdf

Если у вас возникают вопросы, вы можете зарегистрироваться на форуме INSTEAD и задавать их там:

http://instead.syscall.ru/forum

Также, вы можете посетить Jabber конференцию: instead@conference.jabber.ru

Следующие ресурсы также могут быть полезными:

- Обучающая игра: http://instead.syscall.ru/wiki/ru/gamedev/dive-into-instead
- Instead: вводный курс: http://forum.ifiction.ru/viewtopic.php?id=1867

История создания

Когда мы говорим «текстовое приключение» у большинства людей возникает один из двух привычных образов. Это либо текст с кнопками действий, например:

```
Вы видите перед собой стол. На столе лежит яблоко. Что делать?

1) Взять яблоко
2) Отойти от стола
```

Или, гораздо реже, это классические игры с текстовым вводом, где для управления игрой необходимо было вводить действия с клавиатуры.

```
Вы на кухне. Тут есть стол.
> осмотреть стол.
На столе есть яблоко.
```

У обоих подходов есть свои преимущества и недостатки.

Если говорить про первый подход, то он близок к жанру книг-игр и удобен больше для литературных текстов, которые описывают события, происходящие с главным героем, и не очень удобен для создания классических квестов, где главный герой исследует смоделированный в игре мир, свободно перемещаясь по нему и взаимодействуя с объектами этого мира.

Второй подход моделирует мир, но требует значительных усилий от автора игры, и, что более важно, более подготовленного игрока. Особенно, когда мы имеем дело с русским языком.

Проект INSTEAD был создан для написания другого типа игр, которые совмещают преимущества обоих подходов, одновременно пытаясь избежать их недостатков.

Мир игры на INSTEAD моделируется как при втором подходе, то есть в игре есть места (сцены) которые может посещать главный герой и объекты, с которыми он взаимодействует (включая живых персонажей). Игрок свободно изучает мир и манипулирует объектами. Причем, действия с объектами не прописаны в виде явных пунктов меню, а скорее напоминают классические графические квесты в стиле 90-х.

На самом деле, в INSTEAD есть множество незаметных на первый взгляд вещей, которые направлены на развитие выбранного подхода, и который делает процесс игры максимально динамичным и непохожим на привычные «текстовые квесты». Это подтверждается в том числе и тем, что на движке было выпущено множество замечательных игр, интерес к которым проявляют не только любители текстовых игр как таковых, но и люди не знакомые с данным жанром.

Перед изучением данного руководства, я рекомендую поиграть в классические игры INSTEAD, чтобы понять о чем идет речь. С другой стороны, раз вы здесь, то наверное вы уже сделали это.

Правда, не стоит пока изучать код этих игр, так как старые игры очень часто написаны неоптимально, с использованием устаревших конструкций. Текущая версия INSTEAD позволяет реализовывать код лаконичнее, проще и понятнее. Об этом и рассказывается в данном документе.

Если вас интересует история создания движка, то вы можете прочитать статью о том, как все начиналось: http://instead.syscall.ru/2010/05/history

Как выглядит классическая INSTEAD игра

Итак, как выглядит классическая INSTEAD игра?

Главное окно игры содержит информацию о статической и динамической части сцены, активные события и картинку сцены с возможными переходами в другие сцены (в графическом интерпретаторе).

Статическая часть сцены отображается только один раз, при показе сцены, или при повторении команды `look` (в графическом интерпретаторе – клик на названии сцены). *Динамическая часть сцены* составлена из описаний объектов сцены, она отображается всегда.

Игроку доступны объекты, доступные на любой сцене – *инвентарь*. Игрок может взаимодействовать с объектами инвентаря и действовать объектами инвентаря на другие объекты сцены или инвентаря.

Следует отметить, что понятие инвентаря является условным. Например, в «инвентаре» могут находиться такие объекты как «открыть», «осмотреть», «использовать» и т.д.

Действиями игрока могут быть:

- осмотр сцены;
- действие на объект сцены;
- действие на объект инвентаря;
- действие объектом инвентаря на объект сцены;
- действие объектом инвентаря на объект инвентаря;
- действие объектом сцены на объект сцены (режим scene_use);
- действие объектом сцены на инвентарь (режим scene_use);
- переход в другую сцену.

Как создавать игру

Игра представляет из себя каталог, в котором должен находиться скрипт (текстовый файл) `main.lua`. Другие ресурсы игры (скрипты на `lua`, графика и музыка) должны находиться в рамках этого каталога. Все ссылки на ресурсы делаются относительно текущего каталога – каталога игры.

В начале файла `main.lua` может быть определен заголовок, состоящий из тегов (строк специального вида). Теги должны начинаться с символов `--`: комментарий с точки зрения lua. На данный момент существует три тега.

Тег `$Name:` содержит название игры в кодировке UTF-8. Пример использования тега:

```
-- $Name: Самая интересная игра!$
```

Затем следует (желательно) задать версию игры:

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

И указать авторство:

```
-- $Author: Анонимный любитель текстовых приключений$
```

Если вы разрабатываете игру в Windows, то убедитесь, что ваш редактор поддерживает кодировку UTF-8 без BOM. Именно эту кодировку следует использовать при написании игры!

Сразу после заголовков вам необходимо указать версию STEAD API, которая требуется игре. На данный момент последняя версия **1.8.2**.

```
instead_version "1.8.2"
```

Важно!

Если instead_version отсутствует, то STEAD API будет работать в режиме совместимости (устаревшее API).

Дело в том, что начиная с версии 1.2.0, в INSTEAD для автора появились новые возможности. При этом в движок была заложена идея постепенного развития API.

Как это работает?

Если вы пишете в начале игры:

```
instead_version "версия api"
```

То вы даете знать движку, что ваш код написан для INSTEAD >= 1.2.0 и при этом вам доступна та версия API, которую вы вписали. Например:

```
instead_version "1.8.1"
```

Сейчас версии API совпадают с версиями INSTEAD.

Но есть уже много игр, написанных до версии INSTEAD 1.2.0, и они работают на новых версиях INSTEAD. В этих играх нет instead_version и движок понимает, что игра написана для старой версии API.

Новое API дает множество преимуществ автору (снапшоты, глобальные переменные, изменяемые на лету функции, автоматическое форматирование, модули и пр..), а перенос старого кода на новое API тривиален. Поэтому нет большого смысла писать новые игры на старом API, что распространено из-за того, что для изучения INSTEAD были выбраны старые игры.

Важно!

Поэтому, не изучайте код игр прежде чтения этого руководства!

Далее, обычно следует указать модули, которые требуются игре. О модулях будет рассказано отдельно.

```
require "para" -- красивые отступы;
require "dash" -- замена символов два минуса на тире;
require "quotes" -- замена простых кавычек "" на типографские «»;
```

Кроме того, обычно стоит определить обработчики по-умолчанию: game.act, game.use, game.inv, о которых также будет рассказано ниже.

```
game.act = 'Не работает.';
game.use = 'Это не поможет.';
game.inv = 'Зачем мне это?';
```

Инициализацию игры следует описывать в функции `init`, которая вызывается движком в самом начале. В этой функции удобно инициализировать состояние игрока на начало игры, или какие-то другие действия, нужные для первоначальной настройки мира игры. Впрочем, функция `init` может быть и не нужна.

```
function init() -- добавим в инвентарь нож и бумагу
    take(knife);
    take(paper);
end
```

Графический интерпретатор ищет доступные игры в каталоге `games`. Unix-версия интерпретатора кроме этого каталога просматривает также игры в каталоге `~/.instead/games`. Windows-версия: `Documents and Settings/USER/Local Settings/Application Data/instead/games`. В Windows- и standalone-Unix-версии игры ищутся в каталоге `./appdata/games, если он существует`.

В некоторых сборках INSTEAD (в Windows, в Linux если проект собран с gtk и др.) можно открывать игру по любому пути из меню «Выбор игры». Либо, нажать f4. Если в каталоге

с играми присутствует только одна игра, INSTEAD запустит ее автоматически, это удобно, если вы хотите распространять свою игру вместе с движком.

Таким образом, вы кладете игру в свой каталог и запускаете INSTEAD.

Важно!

При написании игры, настоятельно рекомендуется использовать отступы для оформления кода игры, как это сделано в примере из данного руководства, этим самым вы сократите количество ошибок и сделаете свой код наглядней!

Ниже приводится минимальный шаблон для вашей первой игры:

```
-- $Name: Моя первая игра$
-- $Version: 0.1$
-- $Author: Анонимный автор$

instead_version "1.8.2"

require "para" -- для оформления
require "dash"
require "quotes"

require "dbg" -- для отладки

game.act = 'Гм...';
game.use = 'Не сработает.';
game.inv = 'Зачем это мне?';

function init()
-- инициализация, если она нужна
end
```

Основы отладки

Во время отладки (проверки работоспособности вашей игры) удобно, чтобы INSTEAD был запущен с параметром -debug, тогда в случае ошибок будет показана более подробная информация о проблеме в виде стека вызовов. Параметр -debug можно задать в ярлыке (если вы работаете в Windows), а для других систем, я думаю вы и так знаете как передавать параметры командной строки.

При отладке игры обычно нужно часто сохранять игру и загружать состояние игры. Вы можете использовать стандартный механизм сохранений через меню (или по клавишам f2/f3), или воспользоваться быстрым сохранением/загрузкой (клавиши f8/f9).

В режиме `-debug` вы можете перезапускать игру клавишами `alt-r`. В комбинации с f8/f9 это дает возможность быстро посмотреть изменения в игре после ее правки.

В режиме `-debug` Windows-версия INSTEAD создает консольное окно (в Unix версии, если вы запускаете INSTEAD из консоли, вывод будет направлен в нее) в которое будет осуществляться вывод ошибок. Кроме того, используя функцию `print()` вы сможете порождать свои сообщения с отладочным выводом. Например:

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

Не пугайтесь, когда вы прочитаете все руководство и начнете писать свою игру, вы, скорее всего, взглянете на этот пример с большим воодушевлением.

Также удобно воспользоваться модулем отладчика, для этого после записи `instead_version` добавьте:

```
require "dbg"
```

Отладчик доступен по клавише f7.

Во время отладки бывает удобно изучать файлы сохранений, которые содержат состояние переменных игры. Чтобы не искать каждый раз файлы сохранений, создайте каталог saves в директории с вашей игрой (в том каталоге, где содержится main.lua) и игра будет сохраняться в saves. Этот механизм также будет удобен для переноса игры на другие компьютеры.

Возможно (особенно, если вы пользуетесь Unix системами) вам понравится идея проверки синтаксиса ваших скриптов через запуск компилятора `luac`. В Windows это тоже возможно, нужно только установить выполняемые файлы lua для Windows (http://luabinaries.sourceforge.net)/ и воспользоваться luac52.exe.

1. Сцена

Сцена (или комната) – это единица игры, в рамках которой игрок может изучать все объекты сцены и взаимодействовать с ними. Например, сценой может быть комната, в которой находится герой. Или участком леса, доступным для наблюдения.

В любой игре должна быть сцена с именем `main`. Именно с нее начнется и ваша игра!

```
main = room {
    nam = "Главная комната";
    dsc = [[Вы в большой комнате.]];
}
```

Запись означает создание объекта (так как почти все сущности в INSTEAD это объекты) `main` типа `room` (комната). Текстовый идентификатор `main`, по которому можно обращаться к комнате, будем в дальнейшем называть идентификатором комнаты.

У каждого объекта игры есть *атрибуты* и *обработчики событий*. В данном примере есть два атрибута: `nam` и `dsc`. Атрибуты разделяются разделителем (в данном примере – символом точка с запятой `;`).

Обычно, атрибуты могут быть текстовыми строками, функциями-обработчиками и булевыми значениями.

Например, *обязательный* атрибут `nam` для сцены это то, что будет заголовком сцены при ее отображении. Имя сцены также используется для ее идентификации при переходах. Вообще, nam это обязательный атрибут любого объекта. Часто вы можете работать с объектом по его имени, а не по идентификатору.

Атрибут `dsc` – это описание статической части сцены, которое выводится один раз при входе в сцену или при явном осмотре сцены.

Вы можете использовать символ `,` вместо `;` для разделения атрибутов. Например:

```
main = room {
        nam = 'Главная комната',
        dsc = 'Вы в большой комнате.',
}
```

Как вы помните, INSTEAD спрячет статическое описание сцены, после того, как игрок его прочитает при входе в сцену. Если для вашего творческого замысла необходимо, чтобы описание статической части сцены выводилось каждый раз, вы можете определить для своей игры параметр `forcedsc` (в начале игры).

```
game.forcedsc = true;
```

Или, аналогично, задать атрибут forcedsc для конкретных сцен.

```
main = room {
        forcedsc = true;
        nam = 'Главная комната';
        dsc = [[Вы в большой комнате.]];
}
```

Но лучше всего на данном этапе не использовать эту технику, так как классические игры INSTEAD не используют ее, а движок по-умолчанию оптимизирован именно на стандартное поведение.

В данном примере все атрибуты – строковые. Строка может быть записана в одинарных или двойных кавычках:

```
main = room {
        nam = 'Главная комната';
        dsc = "Вы в большой комнате.";
}
```

Для длинных описаний удобно использовать запись вида:

```
dsc = [[ Очень длинное описание... ]];
```

При этом переводы строк игнорируются. Если вы хотите, чтобы в выводе описания сцены присутствовали абзацы – используйте символ `^`.

```
dsc = [[ Первый абзац. ^^
Второй Абзац.^^

Третий абзац.^
На новой строке.]];
```

Я рекомендую всегда использовать [[и]] для `dsc`.

На самом деле, имя `nam` объекта и его отображение (в данном случае то, как сцена будет выглядеть для игрока в виде надписи сверху окна) можно разделять. Для этого существует атрибут `disp`. Если он не задан, то считается, что отображение равняется имени, но если это не так, то для функций отображения используется `disp`, например:

```
main = room {
        nam = 'Начало';
        disp = 'Моя комната';
        dsc = [[Я нахожусь в своей комнате.]];
}
```

2. Объекты

Объекты – это единицы сцены, с которыми взаимодействует игрок.

```
tabl = obj {
        nam = 'стол';
        dsc = 'В комнате стоит {стол}.';
        act = 'Гм... Просто стол...';
};
```

Обязательное имя объекта `nam` используется при попадании его в инвентарь, а также в текстовом интерпретаторе для адресации объекта. Если у объекта определен `disp`, то при попадании в инвентарь для его отображения будет использоваться именно этот атрибут. Например:

```
tabl = obj {
        nam = 'стол';
        disp = 'угол стола';
        dsc = 'В комнате стоит {стол}.';
        tak = 'Я взялся за угол стола';
        inv = 'Я держусь за угол стола.';
};
```

Вы можете скрывать отображение предмета в инвентаре, если `disp` атрибут будет равен `false`.

`dsc` – описание объекта. Оно будет выведено в динамической части сцены. Фигурными скобками отображается фрагмент текста, который будет являться ссылкой в окне INSTEAD. Если объектов в сцене много, то все описания выводятся одно за другим, через пробел,

`act` – это обработчик события, который вызывается при действии пользователя (действие на объект сцены, обычно – клик мышкой по ссылке). Его задача – вывод (возвращение) строки текста, которая станет частью событий сцены, или логического значения (см. раздел 5).

ВНИМАНИЕ: в пространстве имен *lua* уже существуют некоторые объекты (таблицы), например: *table*, *io*, *string*... Будьте внимательны при создании объекта. Например, в приведенном примере традиционно используется `tabl`, а не `table`. Хотя в новых версиях INSTEAD эта проблема практически полностью решена, и в данном примере можно было бы использовать и идентификатор table.

В любом случае, вы не можете использовать дескрипторы объектов, совпадающие с именами конструкторов INSTEAD, такими как: obj, game, player, list, room, dlg.

3. Добавляем объекты в сцену

Ссылкой на объект называется текстовая строка, содержащая дескриптор объекта. Например: 'tabl' – ссылка на объект с дескриптором `tabl`.

Для того, чтобы поместить в сцену объекты, существует два пути.

Во-первых, при создании комнаты можно определить список `obj`, состоящий из ссылок на объекты:

```
main = room {
    nam = 'Главная комната';
    dsc = [[Вы в большой комнате.]];
    obj = { 'tabl' };
};
```

Теперь, при отображении сцены мы увидим объект *«стол»* в динамической части.

Вы можете использовать дескрипторы объектов (в данном примере, указав tabl без кавычек) но только в том случае, если объект был определен ранее (перед определением комнаты). По этой причине использование кавычек всегда безопасней и рекомендуется как предпочтительное.

Если в комнату помещаются несколько объектов, разделяйте их ссылки запятыми, например:

```
  obj = { 'tabl', 'apple' };
```

Вы можете вставлять переводы строк для наглядности, когда объектов много, например, так:

```
  obj = {
                'tabl',
                'apple',
                'knofe',
        };
```

Второй способ размещения предметов заключается в вызове функций, которые поместят объекты в требуемые комнаты. Он будет рассмотрен в дальнейшем.

4. Объекты, связанные с другими объектами

Объекты тоже могут содержать атрибут `obj`. При этом, при выводе объектов, INSTEAD будет разворачивать списки последовательно. Такая техника может использоваться для создания объектов-контейнеров или просто для связывания нескольких описаний вместе. Например, поместим на стол *яблоко*.

```
apple = obj {
    nam = 'яблоко';
    dsc = [[На столе лежит {яблоко}.]];
    act = 'Взять что-ли?';
};

tabl = obj {
    nam = 'стол';
    dsc = [[В комнате стоит {стол}.]];
    act = 'Гм... Просто стол...';
    obj = { 'apple' };
};
```

При этом, в описании сцены мы увидим описание объектов *стол* и *яблоко*, так как `apple` – связанный с `tabl` объект и движок при выводе объекта `tabl` вслед за его `dsc` выведет последовательно `dsc` всех вложенных в него объектов.

Также, следует отметить, что оперируя объектом стол (например, перемещая его из комнаты в комнату) мы автоматически будем перемещать и вложенный в него объект яблоко.

5. Атрибуты и обработчики как функции

Большинство атрибутов и обработчиков могут быть *функциями*. Так, например:

```
nam = function()
    p 'яблоко';
end;
```

Пример не очень удачен, так как проще было бы написать nam = 'яблоко', но показывает синтаксис записи функции.

Основная задача такой функции – это возврат *строки* или булевого значения. Сейчас мы рассматриваем возврат строки. Для возврата строки вы можете использовать явную запись в виде:

```
    return "яблоко";
```

При этом ход выполнения кода функции прекращается и она возвращает движку строку. В данном случае «яблоко».

Более привычным способом вывода являются функции:

- `p («текст»)` – вывод текста и пробела;
- `pn («текст»)` – вывод текста с переводом строки;
- `pr («текст»)` – вывод текста как есть;

Если `p`/`pn`/`pr` вызывается с одним текстовым параметром, то скобки можно опускать.

```
pn "Нет скобкам!";
```

Все эти функции дописывают текст в буфер и при возврате из функции возвращают его движку. Таким образом вы можете постепенно формировать вывод за счет последовательного выполнения `p/pn/pr`. Имейте в виду, что автору крайне редко необходимо явно форматировать текст, особенно если это описание объектов, движок сам расставляет необходимые переводы строк и пробелы для разделения информации разного рода и делает это унифицированным способом.

Вы можете использовать `..` или `,` для склейки строк. Тогда (и) обязательны. Например:

```
pn ("Строка 1".." Строка 2");
pn ("Строка 1", "Строка 2");
```

Для очистки буфера (если это нужно), используйте `pclr()`. Если вам нужно получить содержимое текущего буфера – `pget()`.

Основное отличие атрибутов от обработчиков событий состоит в том, что обработчики событий могут менять состояние игрового мира, а атрибуты нет. Поэтому, если вы оформляете атрибут (например, `nam` или `dsc`) в виде функции, помните, что задача атрибута это возврат значения, а не изменение состояния игры! Дело в том, что движок обращается к атрибутам в те моменты времени, которые обычно четко не определены, и не связаны явно с какими-то игровыми процессами!

Еще одной особенностью обработчиков является тот факт, что вы не должны ждать каких то событий внутри обработчика. То есть, не должно быть каких-то циклов ожидания, или организации задержек (пауз). Дело в том, что задача обработчика – изменить игровое состояние и отдать управление INSTEAD, который визуализирует эти изменения и снова перейдет в ожидание действий пользователя. Если вам требуется организовать задержки вывода, вам придется воспользоваться модулем «timer» или «cutscene».

Функции практически всегда содержат условия и работу с переменными. Например:

```
apple = obj {
    nam = 'яблоко';
```

```
    dsc = function(s)
        if not s._seen then
            p 'На столе {что-то} лежит.';
        else
            p 'На столе лежит {яблоко}.';
        end
    end;
    act = function(s)
        if s._seen then
            p 'Это яблоко!';
        else
            s._seen = true;
            p 'Гм... Это же яблоко!';
        end
    end;
};
```

Если атрибут или обработчик оформлен как функция, то всегда *первый аргумент* функции (`s`) – сам объект. То есть, в данном примере, `s` это синоним `apple`. Когда вы работаете с самим объектом в функции, удобнее использовать параметр, а не дескриптор, так как при переименовании дескриптора вам не придется переписывать вашу игру.

В данном примере при показе сцены в динамической части сцены будет выведен текст: 'На столе что-то лежит'. При взаимодействии с 'что-то', переменная `_seen` объекта `apple` будет установлена в `true` – истина, и мы увидим, что это было яблоко.

Как видим, синтаксис оператора `if` довольно очевиден. Для наглядности, несколько примеров.

if <выражение> then <действия> end

```
if have (apple) then
    p 'У меня есть яблоко!'
end
```

if <выражение> then <действия> else <действия иначе> end

```
if have (apple) then
    p 'У меня есть яблоко!'
else
    p 'У меня нет яблока!'
end
```

if <выражение> then <действия> elseif <выражение 2> then <действия 2> else <иначе> end и т.д.

```
if have (apple) then
    p 'У меня есть яблоко!'
elseif have (fork)
    p 'У меня нет яблока, но есть вилка!'
else
    p 'У меня нет ни яблока, ни вилки!'
end
```

Выражение в операторе if может содержать логическое «и» (and), «или» (or), «отрицание» (not) и скобки (,) для задания приоритетов. Запись вида if <переменная> then означает, что переменная не равна false и определена. Равенство описывается как ==, неравенство ~=.

```
if not have (apple) and not have(fork) then
    p 'У меня нет ни яблока, ни вилки!'
end

...
if w ~= apple then
   p 'Это не яблоко.';
end
...

if time() == 10 then
   p '10 й ход настал!'
end
```

Важно

В ситуации когда переменная не была определена, но используется в условии, считается, что переменная равна пустоте (`nil`). Так, проверку на существование переменной можно было бы написать следующим образом:

```
if z == nil then
        p "Глобальная переменная z не существует."
end
```

В то же время, при анализе несуществующих переменные в условиях, считается, что они дают «ложь». То есть, если z не была определена, то:

```
if not z then
       p "Переменная z или не определена, или равна false."
end
```

С другой стороны:

```
if z == false then
       p "Переменная z равна false."
end
```

Учитывайте это при отладке своей игры, так как если вы описались в имени переменной при задании условия, то условие будет работать (без выдачи ошибки), но игровая логика станет некорректной.

Запись `s._seen` означает, что переменная `_seen` размещена в объекте `s` (то есть `apple`). Помните, мы назвали первый параметр функции `s` (от self), а первый параметр – это дескриптор текущего объекта. *Подчеркивание* означает, что эта переменная *попадет в файл сохранения* игры. Файлы сохранения содержат в себе изменения, которые произошли с игровым миром относительно первоначальной инициализации.

На самом деле, есть два способа определения переменных, попадающих в файл сохранения игры. Либо вы работаете с переменными, имена которых начинаются с символа `_`, и тогда вы можете создавать такие переменные на лету, либо вы должны явно определять (инициализировать) переменные с помощью конструкций `var` и `global`.

```
global { -- определение глобальных переменных
    global_var = 1; -- число
    some_number = 1.2; -- число
    some_string = 'строка';
    know_truth = false; -- булево значение
}
main = room {
    var { -- определение переменных комнаты
        i = "a";
        z = "b";
    };
    nam = 'Моя первая комната';
    var {
        new_var = 3;
    };
    dsc = function(s)
```

```
        p ("i == ", s.i);
        p ("new_var == ", s.new_var);
        p ("global_var == ", global_var);
    end;
```

Если вы пользуетесь `var` или `global`, обязательно присваивайте переменным первоначальные значения! Вы можете инициализировать переменную значением null, если переменная в дальнейшем должна будет хранить дескриптор объекта, а при инициализации этот объект неизвестен. Другими словами, `null` – это объект-пустышка.

Итак, переменные записываются в файл сохранения, если они размещены в одном из перечисленных типов объектов: комната, объект, игра, игрок, глобальное пространство, при этом начинаются с символа `_` или определены с помощью `var` и `global`. При разработке пользуйтесь простым правилом: всегда объявляйте переменные в блоках `var` или `global`, если они будут меняться. Данный способ наиболее нагляден.

В файл сохранения могут быть записаны переменные следующих типов:

- строки;
- булевы величины;
- числовые величины;
- дескрипторы объектов;
- конструкции `code`;

Конструкция `code` – это другой способ определения функций

```
dsc = code [[
      if not self._seen then
          p 'На столе {что-то} лежит.';
      else
          p 'На столе лежит {яблоко}.';
      end
  ]],
```

Обратите внимание, что текст функции задан в скобках [[ ]], и это значит, что текстовые параметры внутри скобок придется задавать кавычками или апострофами.

При вызове code автоматически создаются некоторые переменные. При этом в `self` записан текущий объект, `arg1 ... arg9` и массив `args[]` – параметры.

Запись `code` имеет смысл определять в случае, если функция очень короткая, например:

```
    act = code [[ walk(sea) ]];
```

Либо, если вы хотите переопределять функции на лету (Конструкции `code` сохраняются, если присвоены сохраняемым переменным).

```
    var {
       act = code [[ walk(sea) ]];
    };
...
    s.act = code [[ walk(ground) ]];
```

Правда, обычно это очень плохой стиль программирования игры. Иногда при написании функции вам могут понадобиться вспомогательные переменные, которые хранят какие-то промежуточные значения, например:

```
kitten = obj {
        nam = 'котенок';
        var { state = 1 };
        act = function(s)
                s.state = s.state + 1
                if s.state > 3 then
                        s.state = 1
                end
                p [[Муррр!]]
        end;
        dsc = function(s)
                local dsc = {
                        "{Котенок} мурлычет.",
                        "{Котенок} играет.",
                        "{Котенок} облизывается.",
                };
                p(dsc[s.state])
        end;
end
```

Как видим, в функции dsc мы определили массив dsc. `local` указывает на то, что он действует в пределах функции dsc. Если вам нужны вспомогательные переменные в функциях, всегда пишите перед их определением `local`. Конечно, данный пример можно было написать и так:

```
dsc = function(s)
        if s.state == 1 then
                p "{Котенок} мурлычет."
        elseif s.state == 2 then
                p "{Котенок} играет."
        else
                p "{Котенок} облизывается.",
        end
```

Вы можете писать свои вспомогательные функции и использовать их из своей игры, например:

```
function mprint(n, ...)
        local a = {...}; -- временный массив с аргументами к функции
        p(a[n]) -- выведем n-й элемент массива
end
....
        dsc = function(s)
                mprint(s.state, {
                        "{Котенок} мурлычет.",
                        "{Котенок} играет.",
                        "{Котенок} облизывается." });
        end;
```

Пока не обращайте внимания на данный пример, если он кажется вам сложным.

Иногда может понадобиться обработчик, который совершал бы некоторое действие, но не выводил никакого описания. Например:

```
button = obj {
    nam = "кнопка";
        var {
                on = false;
        };
    dsc = "На стене комнаты видна большая красная {кнопка}.";
    act = function (s)
            s.on = true
                return true
        end;
}
```

```
r12 = room {
    nam = 'Комната';
        forcedsc = true;
    dsc = function (s)
                if not button.on then
                        p [[Я нахожусь в комнате.]];
                else
                        p [[Я нахожусь в комнате с нажатой кнопкой.]];
                end
        end,
    obj = {'button'}
}
```

В данном случае обработчик `act` нужен для того, чтобы поменять описание комнаты, и не нужно, чтобы он выводил результат действия. Если честно, это плохой пример, и его искусственность показывает, что вам вряд ли понадобится делать обработчики без реакции. Действительно, почему бы в приведенном примере не вывести в `act` что-то вроде: «Я нажал на кнопку»? Кроме того, нам пришлось еще включить режим `forcedsc`. Тем не менее, необходимость пустой реакции может возникнуть.

Для отключения результата можно вернуть из обработчика значение `true` – это будет означать, что действие успешно выполнено, но не требует дополнительного описания.

Если необходимо показать, что действие невыполнимо, ничего не возвращайте. При этом будет отображено описание по умолчанию, заданное с помощью обработчика `game.act`. Обычно описание по умолчанию содержит описание невыполнимых действий. Что-то вроде:

```
game.act = 'Гм... Не получается...';
```

Обратите внимание, что для создания динамического описания сцены в рассмотренном выше примере используется функция `dsc`. Почему бы не менять значение `dsc` на лету? На самом деле такой вариант сработает только в том случае, если `dsc` объявить в блоке `var`. Например, данный пример мог бы выглядеть так:

```
button = obj {
    nam = "кнопка";
    dsc = "На стене комнаты видна большая красная {кнопка}.";
    act = function (s)
            here().dsc = [[Теперь комната выглядит совсем по-другому!!!]];
            pn [[После того как я нажал на кнопку, комната преобразилась.
          Книжный шкаф куда-то исчез вместе со столом и комодом, а на его месте
          появился странного вида аппарат.]];
```

```
        end,
}

r12 = room {
    nam = 'Комната';
        var {
            dsc = [[Я нахожусь в комнате.]];
        };
    obj = {'button'}
}
```

Правда, такой стиль программирования не рекомендуется. Во-первых, вы запутываете код игры, так как тексты описаний перестают быть локализованными в объектах, которые они описывают. Во-вторых, файлы сохранений будут занимать гораздо больший объем. Я советую всегда использовать функции для изменяющихся атрибутов и реакций, а не менять их значение динамически извне.

Иногда возникает необходимость вызвать обработчик вручную. Для этого используется `lua` синтаксис вызова метода. `Объект:метод(параметры)`. Например:

```
apple:act() -- вызовем обработчик ''act'' у объекта ''apple''.
```

На самом деле, это синоним следующей записи:

```
apple.act(apple) -- вызовем обработчик ''act'' у объекта ''apple''
-- (явно указав 1-й параметр)
```

Однако, такой метод работает только в том случае, если вызываемый метод оформлен как функция. Вы можете воспользоваться `stead.call()` для вызова обработчика тем способом, каким это делает сам INSTEAD. (Будет описано в дальнейшем).

6. Инвентарь

Простейший вариант сделать объект, который можно брать – определить обработчик `tak`.

Например:

```
apple = obj {
        nam = 'яблоко';
        dsc = 'На столе лежит {яблоко}.';
        inv = function(s)
                remove(s, me()); -- удалить яблоко из инвентаря
                p 'Я съел яблоко.'
        end;
        tak = 'Вы взяли яблоко.';
};
```

При этом, при действии игрока на объект «яблоко» (щелчок мыши на ссылку в сцене) – яблоко будет убрано из сцены и добавлено в инвентарь. При действии игрока на инвентарь (двойной щелчок мыши на названии объекта) – вызывается обработчик `inv`.

В нашем примере, при действии игроком на яблоко в инвентаре – яблоко будет съедено.

Конечно, мы могли бы реализовать код взятия объекта в `act`, например, так:

```
apple = obj {
        nam = 'яблоко';
        dsc = 'На столе лежит {яблоко}.';
        inv = function(s)
                remove(s, me()); -- удалить яблоко из инвентаря
                p 'Я съел яблоко.'
        end;
        act = function(s)
                take(s)
                p 'Вы взяли яблоко.';
        end
};
```

Если у объекта в инвентаре не объявлен обработчик `inv` будет вызван `game.inv`.

7. Переходы между сценами

Традиционные переходы в INSTEAD выглядят как ссылки над описанием сцены. Для определения таких переходов между сценами используется атрибут сцены – список `way`. В списке определяются комнаты, в виде ссылок на комнаты или дескрипторов комнат (аналогично списку `obj`). Например:

```
room2 = room {
    nam = 'Зал';
    dsc = 'Вы в огромном зале.';
    way = { 'main' };
};

main = room {
    nam = 'Главная комната';
    dsc = 'Вы в большой комнате.';
    obj = { 'tabl' };
    way = { 'room2' };
};
```

При этом, вы сможете переходить между сценами `main` и `room2`. Как вы помните, `nam` (или `disp`) может быть функцией, и вы можете генерировать имена сцен на лету, например, если вы хотите, чтобы игрок не знал название сцены, пока не попал на нее. Хотя для этой цели есть более удачные средства, вроде модуля `wroom`, о котором будет рассказано позже.

При переходе между сценами движок вызывает обработчик `exit` из текущей сцены и `enter` в той сцене, куда идет игрок. Например:

```
room2 = room {
    enter = 'Вы заходите в зал.';
    nam = 'Зал';
    dsc = 'Вы в огромном зале.';
    way = { 'main' };
    exit = 'Вы выходите из зала.';
};
```

Конечно, как и все обработчики, `exit` и `enter` могут быть функциями. Тогда первый параметр это (как всегда) *сам объект*, а второй это комната куда игрок *хочет идти* (для `exit`) или из которой *уходит* (для `enter`). Например:

```
room2 = room {
    enter = function(s, f)
        if f == main then
            p 'Вы пришли из комнаты.';
        end
    end;
    nam = 'Зал';
    dsc = 'Вы в огромном зале.';
```

```
    way = { 'main' };
    exit = function(s, t)
        if t == main then
            p 'Я не хочу назад!'
                        return false
        end
    end;
};
```

Как видим, эти обработчики могут возвращать два значения: *строку* и *статус*. В нашем примере функция `exit` вернет `false`, если игрок попытается уйти из зала в комнату `main`. `false` означает, что переход *не будет* выполнен. Такая же логика работает и для `enter`. Кроме того, она работает и для обработчика `tak`.

Вы можете сделать возврат статуса и другим способом, если это кажется вам удобным:

```
        return "Я не хочу назад", false
```

Если же вы используете функции `p`/`pn`/`pr`, то просто возвращайте статус операции с помощью завершающего `return`, как показано в примере выше.

Следует отметить, что при вызове обработчика `enter` указатель на текущую сцену (here()) может быть **еще не изменен**!!! В INSTEAD есть обработчики `left` (уход из комнаты) и `entered` (заход в комнату), которые вызываются уже после того, как переход произошел. Эти обработчики рекомендованы к использованию всегда, когда нет необходимости запрещать переход.

Иногда есть необходимость, чтобы название перехода отличалось от названия комнаты, в которую ведет этот переход. Существует несколько способов сделать это. Например, с помощью `vroom`.

```
room2 = room {
    nam = 'Зал';
    dsc = 'Вы в огромном зале.';
    way = { vroom('В главную комнату', 'main') };
};

main = room {
    nam = 'Главная комната';
    dsc = 'Вы в большой комнате.';
    obj = { 'tabl' };
    way = { vroom('В зал', 'room2') };
};
```

На самом деле, функция `vroom` возвращает комнату с именем, который равен первому параметру, и специальной функцией `enter`, которая перенаправляет игрока в комнату заданную вторым параметром `vroom`.

Таким образом, `vroom` позволяет именовать переходы удобным способом. Обратите внимание, что в `room2` вторая комната задана через ссылку, так как на данный момент `main` еще не объявлена. Во второй комнате переход также описан в виде ссылки, для общности, хотя в этот момент комната `room2` уже объявлена, и мы можем убрать апострофы вокруг `room2` во втором параметре `vroom`.

Если вам не хватает возможностей `vroom`, вы можете воспользоваться модулем `wroom`.

Иногда вам может потребоваться включать и выключать переходы. На самом деле это требуется не часто. Идея переходов состоит в том, что переход виден даже тогда, когда он невозможен. Например, представим себе сцену перед домом у входной двери. Войти в дом нельзя, так как дверь закрыта.

Нет никакого смысла прятать переход «дверь». Просто в функции `enter` сцены внутри дома мы проверяем, а есть ли у героя ключ? И если ключа нет, говорим о том, что дверь закрыта и запрещаем переход. Это повышает интерактивность и упрощает код. Если же вы хотите сделать дверь объектом сцены, поместите ее в комнату, но в `act` обработчике сделайте осмотр двери, или дайте возможность игроку открыть ее ключом (как это сделать - мы рассмотрим позже), но сам переход дайте сделать игроку привычным способом через строку переходов.

Тем не менее, бывают ситуации, когда переход не очевиден и он появляется в результате каких-то событий. Например, мы осмотрели часы и увидели там секретный лаз.

```
clock = obj {
        nam = 'часы';
        dsc = [[Тут есть старинные {часы}.]];
        act = function(s)
                path('В часы'):enable()
                p [[Вы видите, что в часах есть потайной ход!]];
        end;
}
hall = room {
    nam = 'Зал';
    dsc = 'Вы в огромном зале.';
        obj = { 'clock' };
    way = { vroom('В часы', 'inclock'):disable() };
};
```

В данном примере, мы создали *отключенный* переход, за счет вызова метода `disable` у комнаты созданной с помощью `vroom`. Метод `disable` есть у всех объектов, он переводит объект в отключенное состояние, которое означает, что объект перестает рассматриваться движком как существующий. Кроме того, этот метод снова возвращает объект (уже в

выключенном состоянии). Замечательным свойством отключенного объекта является то, что его можно *включить* с помощью метода `:enable()`;

Кстати, если вам удобнее, вы можете использовать более привычную запись функций:

```
way = { disable(vroom('В часы', 'inclock')) };
```

Далее, когда игрок нажимает на ссылку, описывающую часы, вызывается обработчик `act`, который с помощью функции `path()` находит в `way` текущей комнаты переход 'В часы' и вызывает для него метод `enable()`. Альтернативный вариант записи, если он кажется вам удобней:

```
        act = function(s)
                enable(path('В часы'))
                -- вы можете опускать () если у функции 1 параметр,
                -- таким образом запись enable( path 'В часы' )
                -- тоже будет корректной (и более простой)
                p [[Вы видите, что в часах есть потайной ход!]];
        end;
```

Если бы нам было необходимо проделать такую процедуру с другой комнатой, то мы бы указали второй параметр при вызове `path`. Например:

```
        path('В часы', room312):enable();
```

Если вам не нравится, что в вашей игре есть привязка к имени перехода 'В часы', то вы можете использовать переменные:

```
path_clock = vroom('В часы', 'inclock');

clock = obj {
        nam = 'часы';
        dsc = [[Тут есть старинные {часы}.]];
        act = function(s)
                path_clock:enable()
                p [[Вы видите, что в часах есть потайной ход!]];
        end;
}
hall = room {
```

```
    nam = 'Зал';
    dsc = 'Вы в огромном зале.';
        obj = { 'clock' };
    way = { path_clock:disable() };
};
```

Если вы не используете `vroom`, вы можете включать и выключать сами комнаты:

```
inclock = room {
        nam = 'Внутри';
        dsc = [[Тут темно.]];
}:disable();
-- вместо приписки :disable() можно написать
-- inclock:disable()
-- или disable(inclock)

clock = obj {
        nam = 'часы';
        dsc = [[Тут есть старинные {часы}.]];
        act = function(s)
                inclock:enable()
                p [[Вы видите, что в часах есть потайной ход!]];
        end;
}
hall = room {
    nam = 'Зал';
    dsc = 'Вы в огромном зале.';
        obj = { 'clock' };
    way = { 'inclock' };
};
```

На самом деле, данными вещами не стоит злоупотреблять. Простые игры могут вполне обходиться обычными переходами в виде списка комнат.

8. Действие объектов друг на друга

Игрок может действовать объектом инвентаря на другие объекты. Для этого он щелкает мышью на предмет инвентаря, а затем, на предмет сцены. При этом вызывается обработчик `use` у объекта, которым действуют, и `used` – на который действуют.

Например:

```
knife = obj {
    nam = 'нож';
    dsc = 'На столе лежит {нож}';
    inv = 'Острый!';
    tak = 'Я взял нож!';
    use = 'Вы пытаетесь использовать нож.';
};

tabl = obj {
    nam = 'стол';
    dsc = 'В комнате стоит {стол}.';
    act = 'Гм... Просто стол...';
    obj = { 'apple', 'knife' };
    used = 'Вы пытаетесь сделать что-то со столом...';
};
```

Если игрок возьмет нож и использует его на стол – то он увидит текст обработчиков `use` (у объекта `knife`) и `used` (у объекта `tabl`). `use` и `used`, очевидно, могут быть функциями.

Тогда, в случае функции `use`, первый параметр к функции – сам объект, которым осуществляется действие, а второй параметр – объект, на который направлено действие. В случае функции `used`, первый параметр – сам объект, на который производится действие, а второй параметр – объект, которым осуществляется действие.

То есть, *первый* параметр это как всегда *сам объект*, а *второй* – объект на который *направлено действие* в случае `use` и объект, которым действие *осуществляется* в случае `used`.

`use` может вернуть статус `false`, в этом случае обработчик `used` не вызовется (если он вообще был). Статус обработчика `used` **игнорируется**.

Пример:

```
knife = obj {
    nam = 'нож';
    dsc = 'На столе лежит {нож}';
    inv = 'Острый!';
    tak = 'Я взял нож!';
    use = function(s, w)
        if w ~= tabl then
            p 'Не хочу это резать.'
                        return false
        else
            p 'Вы вырезаете на столе свои инициалы.'
```

```
        end
    end
};
```

В примере выше нож можно использовать только на стол.

Если и `use` и `used` ничего не вернут (или их нет), вызовется обработчик по-умолчанию `game.use`.

Использовать `use` или `used` это вопрос личных предпочтений, однако, всегда следует стараться локализовать код рядом с объектом, к которому он относится. Например, если сделать объект «мусорка» и дать возможность игроку выбрасывать в нее все предметы, то очевидным решением здесь будет обработчик `used` у мусорки.

```
trash = obj {
        nam = 'мусорка';
        dsc = [[Я вижу {мусорку}.]];
        act = 'Не полезу я туда.';
        used = function(s, w)
                remove(w, me())
                p [[Мне это не нужно больше.]];
        end
}
```

При одновременном использовании `use` и `used` могут быть проблемы. Например, пусть у игрока есть нож, при использовании которого на все предметы, кроме яблока, выдается сообщение «Я не хочу это резать.».

При применении ножа на мусорку, игроку будет выдано «Я не хочу это резать.», а потом нож навсегда сгинет в ее недрах (так мы написали обработчик `used`); Конечно, в `use` ножа можно написать:

```
p "Я не хочу это резать."
return false -- прервать цепочку (не вызывать used)
```

Но это не очень удобно. Для решения этой проблемы можно воспользоваться модулем `nouse`.

```
...
require "nouse"
```

```
...
knife = obj {
    nam = 'нож',
    use = function(s, w)
                if w ~= apple then -- неравенство
                        return
                end
                if w.knife then
                        return "Я уже почистил его."
                end
                w.knife = true
            p 'Я почистил яблоко.'
    end;
        nouse = [[Не хочу это резать.]];
};
```

Обработчик `nouse` объекта вызывается в том случае, если ни `use` ни `used` не вернули реакции. Если и `nouse` объекта не содержит реакции, то вызовется `noused` страдательного объекта. Если и в таком случае реакция не прописана, вызовется `game.nouse`;

Конечно, все обработчики могут быть функциями с тремя параметрами. Сам объект (или объект `game` в случае `game.nouse`), объект которым действуем и страдательный объект.

Модуль «nouse» переопределяет функцию `game.use`, поэтому используйте `game.nouse` если вы включаете модуль 'nouse' в свою игру.

Я рекомендую использовать модуль «nouse» всегда, так как код игры становится более наглядным.

На самом деле, в INSTEAD играх можно действовать не только предметом инвентаря на предмет сцены, но и предметом сцены на предмет сцены (и даже предметом сцены на инвентарь). Этот механизм вряд ли действительно стоит использовать, но тем не менее, определив булевый атрибут `scene_use` в объекте или у игры (`game.scene_use`), вы можете управлять моментом, когда этот режим доступен.

Например: `game.scene_use = true`.

Атрибут может быть задан функцией, возвращающей булево значение (`true` или `false`).

9. Объект "player"

Игрок в STEAD представлен объектом `pl`. Тип объекта – `player`. В движке объект создается следующим образом:

```
pl = player {
        nam = "Incognito";
```

```
        where = 'main';
        obj = { };
};
```

Атрибут `obj` представляет собой инвентарь игрока. Обычно, нет смысла переопределять объект типа player, однако, если вы хотите создать переменные, которые бы хранились в игроке, вы можете сделать это:

```
pl = player {
        nam = "Василий";
        where = 'main';
        var { power = 100 };
        obj = { 'apple' }; -- заодно добавим яблоко в инвентарь
};
```

В INSTEAD есть возможность создавать нескольких игроков и переключаться между ними. Для этого служит функция `change_pl()`. В качестве параметра передайте функции требуемый объект типа `player`. Функция переключит текущего игрока, и при необходимости, осуществит переход в комнату, где находится новый игрок.

Функция `me()` всегда возвращает текущего игрока. В большинстве игр `me() == pl`.

10. Объект ''game''

Игра также представлена объектом `game` с типом `game`. В движке он определяется следующим образом:

```
game = game {
        codepage = "UTF-8",
    nam = "INSTEAD -- Simple Text Adventure interpreter v"..
                    stead.version.." '2013 by Peter Kosyh",
    dsc = [[
Commands:^
    look(or just enter), act <on what> (or just what), use <what> [on what],
    go <where>,^
    back, inv, way, obj, quit, save <fname>, load <fname>.]],
    pl ='pl',
    showlast = true,
        _scripts = {},
};
```

Как видим, объект хранит в себе указатель на текущего игрока (`pl`) и некоторые параметры. Не существует возможности пересоздать объект `game`, к счастью, это и не требуется. Однако, вам придется иногда менять некоторые атрибуты.

Например, вы можете указать в начале своей игры кодировку текста следующим образом:

```
game.codepage="cp1251";
```

На самом деле, не стоит этого делать. Лучше настройте свой редактор на использование UTF-8. Переопределение кодировки это экстренная мера, например, для игр написанных `URQL`, которые могут быть запущены в INSTEAD с помощью соответствующего модуля.

Кроме того, объект `game` может содержать обработчики по умолчанию `act`, `inv`, `use`, которые будут вызваны, если в результате действий пользователя не будут найдены никакие другие обработчики. Например, вы можете написать в начале игры:

```
game.act = 'Не получается.';
game.inv = 'Гм.. Странная штука..';
game.use = 'Не сработает...';
```

Всегда определяйте эти обработчики в своей игре. В случае, если вы используете модуль «nouse», вместо `game.use` задавайте `game.nouse`.

11. Атрибуты-списки

Атрибуты-списки (такие как `way` или `obj`) позволяют работать со своим содержимым с помощью набора методов. Атрибуты-списки призваны сохранять в себе списки объектов. На самом деле, вы можете создавать списки для собственных нужд, при этом нет необходимости вставлять список в `var` или `global`, например:

```
treasures = list { 'gold', 'silver' };
```

Хотя обычно, это не требуется.

Методы списков: `add`, `del`, `look`, `srch`, `purge`, `replace`. Из них наиболее часто используемые: `add` и `del`.

- `add` – добавляет в список объект;
- `cat` – `cat(b, [pos])` – добавляет в список содержимое списка `b` в позицию `pos`;

- `zap` – обнулить список;
- `del` – удаляет объект из списка (если он не выключен);
- `purge` – удаляет даже выключенный объект;
- `srch` – выполняет поиск объекта. Возвращает два значения: элемент списка и индекс (если элемент найден в списке);
- `replace` – замена объекта другим `replace(old, new)`;
- `enable` – включение объекта (если он найден в списке);
- `disable` – выключение объекта (если он найден в списке);
- `enable_all` – включить все объекты в списке;
- `disable_all` – выключить все объекты в списке;

Следует отметить, что параметром `add`, `del`, `purge`, `replace` и `srch` и других методов может быть не только сам объект (дескриптор), но и имя объекта (атрибут `nam`).

Самый часто используемый пример для работы со списками это конструкция вида: `inv():del('apple')`;

`inv()` – это функция, которая возвращает список-инвентарь. `del` после `:` – метод, удаляющий элемент инвентаря.

Аналогично, собственная реализация `tak` может быть такой:

```
knife = obj {
    nam = 'нож',
    dsc = 'На столе лежит {нож}',
    inv = 'Острый!',
    act = function(s)
        objs():del(s);
        inv():add(s);
    end,
};
```

`objs()` – это функция, которая возвращает список объектов текущей комнаты. Либо любой другой комнаты, если указать ее в качестве параметра к `objs()`.

Для получения списка `way` используйте функцию `ways()`.

Кроме удаления / добавления объектов из списков вы можете использовать *выключение / включение* объектов с помощью методов *объекта* `disable()` и `enable()`. Например: `knife:disable()`. При этом объект `knife` пропадает из описания сцены (предполагаем, что он находится в списке `obj` сцены), но в последствии может быть опять быть включен,

с помощью `knife:enable()`. Для проверки того факта, что объект выключен, можно использовать метод `:disabled()`. Если вам привычней использовать синтаксис функций, вы можете пользоваться записью вида:

```
disable(knife) -- или knife:disable()
...
if not disabled(knife) -- или not knife:disabled()
...
```

Методы *списков* `enable` и `disable` работают аналогично одноименным методам объектов, при условии что искомый объект находится в списке. Например, учитывая, что поиск объекта в списке осуществляется в том числе и по имени, можно написать:

```
inv():disable 'нож';
```

Внимание!!! Для работы с инвентарем, объектами и переходами рекомендуется использовать более высокоуровневые функции: `put` / `get` / `take` / `drop` / `remove` / `seen` / `have` и др. функции, которые будут описаны в дальнейшем. Тем не менее, иногда работа со списками оказывается полезной.

12. Функции, которые возвращают объекты

В INSTEAD определены некоторые функции, которые возвращают различные объекты или списки. При описании функции используются следующие соглашения о параметрах.

- в символах [ ] описаны необязательные параметры;
- `что` – означает объект (в том числе комнату), заданный дескриптором, ссылкой или именем;
- `где` – означает объект (в том числе комнату), заданный дескриптором или ссылкой;
- `комната` – означает объект типа `room`, заданный дескриптором или ссылкой;
- `объект` – означает объект типа `obj`;
- `переход` – означает текстовое имя перехода в `way`, его дескриптор или ссылку;

Функции, возвращающие списки:

- `inv()` возвращает список инвентаря;

- `objs([где])` возвращает список объектов текущей сцены; Второй необязательный параметр – комната или объект, для которой возвращается список `obj`;
- `ways([комната])` возвращает список возможных переходов из текущей сцены; Второй необязательный параметр – комната, для которой возвращается список;

На самом деле, функции, которые возвращают списки, редко бывают необходимы при написании игры, так как для работы с объектами и переходами существует набор специальных функций, которые описаны в следующей главе.

Теперь посмотрим на функции, возвращающие объекты:

- `me()` возвращает текущего объекта-игрока;
- `here()` возвращает текущую сцену;
- `where(объект)` возвращает комнату или объект в котором находится заданный объект, если он был помещен туда с помощью функций `put/move/drop/replace` и т.д.)
- `from([комната])` возвращает прошлую комнату, из которой игрок перешел в текущую комнату. Второй необязательный параметр – получить прошлую комнату не для текущей комнаты, а для заданной;
- `seen(что, [где])` возвращает объект, если он присутствует и не отключен на сцене, есть второй необязательный параметр – выбрать сцену или объект в котором искать `что`;
- `have(что)` возвращает объект, если он есть в инвентаре и не отключен;
- `exist(что, [где])` возвращает объект, если он присутствует на сцене (даже если он выключен!), есть второй необязательный параметр – выбрать сцену или объект, в котором искать `что`;
- `live(что)` возвращает объект, если он присутствует среди живых объектов (описано далее);
- `path(переход, [комната])` – найти элемент в `way`, даже если он выключен. Второй параметр используется, если интересует не текущая, а заданная комната;

Эти функции в основном используются в условиях, либо для поиска объекта с последующей модификацией. Например, вы можете использовать `seen` для написания условия:

```
exit = function(s)
        if seen 'монстр' then -- если у функции 1 параметр,
                --- скобки писать не обязательно
                p 'Монстр загораживает проход!'
                return false
        end
end
```

А также, для нахождения объекта в сцене:

```
use = function(s, w)
        if w == window and path 'В окно':disabled() then
                -- действие на окно и нет перехода 'В окно'
                path 'В окно':enable();
                p 'Я разбил окно!'
        end
end
```

Пример с функцией `have`:

```
...
act = function(s)
    if have('knife') then
        p 'Но у меня же есть нож!';
                return
    end
end
...
```

Следующие варианты тоже будут работать:

```
...
    if have 'knife' then
...
    if have (knife) then
...
```

Еще одна функция, которая получает объект по ссылке: `stead.ref(ссылка)`.

На самом деле, вам вряд ли понадобится эта функция, по крайней мере для первой игры. Эта функция получает объект по ссылке, другими словами:

```
stead.ref 'apple' == apple
```

При условии, конечно, что `apple` определена.

Функция `stead.deref(дескриптор)`, наоборот, возвращает ссылку-строку для объекта;

```
act = function(s)
        p('Вы кликнули на объект: ', stead.deref(s));
end
```

13. Другие функции INSTEAD

В INSTEAD определено множество функций, которые манипулируют состоянием предмета, или используются для написания игровой логики. Многие из них вы уже встречали по мере чтения данного руководства. Теперь рассмотрим их подробней.

- `move(что, куда, [где])` переносит объект из текущей сцены в другую сцену или объект; Если задать параметр `где`, то объект будет искаться не в текущей сцене, а в сцене или объекте `где`.

```
move('mycat','inmycar');
```

Если вы хотите перенести объект из произвольной сцены, вам придется знать о его местоположении. Для создания сложно перемещающихся объектов, вам придется написать свой метод, который будет сохранять текущую позицию объекта в самом объекте и делать удаление объекта из старой сцены. Либо, если для перемещения объектов вы пользуетесь функциями из данной главы, вы можете использовать `where`. Например:

```
move(mycat, here(), where(mycat)); -- мой кот ходит со мной;
```

Правда при этом и первоначальное размещение объекта `mycat` должно производиться методом `put` (или `place`).

Существует также функция `movef`, аналогичная `move`, но добавляющая объект в начало списка.

- `drop(что, [где])` – положить объект из инвентаря на сцену:

```
drop (knife);
```

Существует также функция `dropf`, аналогичная `drop`, но добавляющая объект в начало списка объектов. Второй необязательный параметр – комната, куда помещается предмет.

- `place(что, [где]) и placef(...)` как drop/dropf, но не удалять предмет из инвентаря;
- `put(что, [где]) и putf(...)` устаревшая форма записи place/placef;
- `replace(что, объект, [где])` – заменить один объект на другой в сцене или объекте `где`;
- `remove(что, [где])` удаляет объект из текущей сцены или сцены/объекта `где`, если он не выключен;
- `purge (объект, [откуда])` – см. `remove`, удаляет даже выключенные объекты;
- `take(что, [где])` – взять объект с текущей сцены или объекта/сцены заданной `где`. `takef` – вариант с добавлением в начало инвентаря. На самом деле, если объекта нет на сцене, он все равно будет добавлен в инвентарь, таким образом `take`, например, часто используется для первоначального наполнения инвентаря в `init` функции.

```
take('knife');
```

Внимание!!! На самом деле, эти функции также умеют работать не только с комнатами и объектами, но и со списками. То есть `remove(apple, inv())` сработает также как и `remove(apple, me())`; Кроме того, у некоторых описанных выше функций есть варианты с постфиксом `to`: `placeto`, `putto`, `taketo`, `dropto`. У этих функций есть дополнительный параметр – индекс позиции в списке, для точного управления позицией предмета. Вы также можете вписывать позицию прямо в списках, например;

```
obj = { [1] = 'apple', [1000] = 'floor' };
```

Но я не рекомендую пользоваться такими вещами в своих играх, если вам нужно точное позиционирование, лучше используйте объекты в объектах.

- `lifeon(объект, [приоритет])` добавить объект в список динамических («живых») объектов (будет описано дальше), приоритет – числовой параметр (положительное число), 1 – наивысший приоритет;
- `lifeoff(что)` удалить объект из списка динамических объектов;
- `taken(объект)` – если объект взят – возвратить `true` (взят с помощью `tak` или `take()`);
- `rnd(m)` – случайное целочисленное значение от `1` до `m`;
- `walk(куда)` – перейти в сцену `куда`:

```
act = code [[
        pn "Я иду в следующую комнату..."
        walk (nextroom);
]]
```

```
mycar = obj {
    nam = 'моя машина';
    dsc = 'Перед хижиной стоит мой старенький {пикап} Toyota.';
    act = function(s)
        walk('inmycar');
    end
};
```

Внимание!!!

После вызова `walk` выполнение обработчика продолжится до его завершения. Поэтому обычно, после `walk` всегда следует `return`, если только это не последняя строка функции, хотя и в этом случае безопасно поставить `return`.

```
act = code [[
        pn "Я иду в следующую комнату..."
        walk (nextroom);
        return
]]
```

Не забывайте также, что при вызове `walk` вызовутся обработчики `exit/enter/left/entered` и если они запрещают переход, то он не произойдет.

- `change_pl(игрок)` – переключиться на другого игрока (со своим инвентарем и позицией). При этом функция переносит действие в сцену нового игрока без вызова `exit/enter/left/entered`. Для указания сцены, на которой находится игрок, вы можете использовать атрибут `where`:

```
pupkin.where = 'kitchen'
```

... или явно вызывать функцию `walk()` после change_pl().

- `walkback([куда])` – возвращается из сцены в прошлую (если не задано `куда`). Возврат означает, что свойство `from` не будет изменено;
- `back([куда])` – возвращается из сцены в прошлую (если не задано `куда`). Если это переход из диалога в комнату, то не вызываются: `dsc`, `enter`, `entered` у комнаты. `exit`/`left` диалога вызываются. В других случаях аналогична `walkback`.
- `walkin(куда)` – перейти в сцену, при этом `exit`/`left` текущей комнаты не вызывается;
- `walkout()` – вернуться в прошлую сцену, при этом `enter`/`entered` этой сцены не вызовется;
- `time()` – возвращает текущее время игры. Время игры считается в активных действиях игрока.
- `cat(...)` – возвращает строку – склейку строк-аргументов. Если первый аргумент `nil` – пустота, то функция возвращает `nil` – пустоту.
- `par(...)` – возвращает строку – склейку строк-аргументов, разбитых строкой-первым параметром.
- `disable`/`enable`/`disable_all`/`enable_all` – аналог одноименных методов у объекта;
- `visited([комната])` счетчик посещений комнаты или `nil` - если мы в ней не были, если `комната` не задана, то проверяется текущая;
- `visits([комната])` аналогична `visited` за исключением того, что если посещений не было, возвращает 0, а не пустоту;
- `player_moved()` возвращает true если в данном такте игры был переход игрока между комнатами, предназначена для использования в `life` методах (описано далее);
- `stead.need_scene()` если вам необходимо, чтобы следующий такт игры вывел статическую часть сцены (и вы не хотите использовать `forcedsc`), вы можете воспользоваться этой функцией. Другим способом перерисовки сцены является переход в эту-же комнату;
- `stead.nameof(объект)` – вернуть имя объекта (`nam` атрибут);
- `stead.dispof(объект)` – вернуть результат `disp`, а если его нет – `nam`;
- `disabled(объект)` – возвращает `true`, если объект отключен;
- `stead.call(дескриптор, строка - имя атрибута/обработчика, параметры...)` – вызов обработчика или получение значения атрибута (описано далее);

14. Диалоги

Диалоги – это сцены специального типа `dlg`, содержащие объекты – фразы. В INSTEAD на данный момент существует два способа описания диалогов: *расширенный* и *простой*. Простой способ считается устаревшим, и не рекомендуется для новых игр. Рассмотрим оба способа.

Общим для обоих способов является сама архитектура, по которой работает диалог.

При входе в диалог игрок видит перечень фраз (по-умолчанию, пронумерованных), которые может выбирать, получая какую-то реакцию игры. По умолчанию, уже выбранные фразы скрываются. При исчерпании всех вариантов, диалог завершается выходом в предыдущую комнату (конечно, если в диалоге нет постоянно видимых фраз, среди которых обычно встречается что-то типа `Завершить разговор` или `Спросить еще раз`).

Переход в диалог в игре осуществляется как переход на сцену:

```
povar = obj {
    nam = 'повар';
    dsc = 'Я вижу {повара}.';
    act = function()
        walk 'povardlg'
    end,
};
```

Хотя я рекомендую использовать `walkin`, так как в случае `walkin` не вызываются `exit/left` текущей комнаты, а персонаж, с которым мы можем поговорить, обычно находиться в этой же комнате, где и главный герой. То есть:

```
povar = obj {
    nam = 'повар';
    dsc = 'Я вижу {повара}.';
    act = function()
        walkin 'povardlg'
    end,
};
```

Вы можете делать переход из одного диалога в другой, организуя иерархичность диалогов (при необходимости делая возврат на предыдущий уровень с помощью `back()`), впрочем, в расширенных диалогах иерархичность организована более просто.

Если вам не нравится префикс у фразы в виде цифры, вы можете определить строковую переменную:

```
stead.phrase_prefix = '--';
```

И получить префикс в виде тире перед каждой фразой.

Обратите внимание, что `stead.phrase_prefix` не сохраняется, если вам нужно переопределять ее на лету, вам придется восстанавливать ее состояние в `start` функции вручную!

Внимание!!! Если в диалоге нет ни одной фразы, движок по умолчанию блокирует переход в такой диалог (ведь из него нет выхода!). Имейте это в виду при отладке своей игры.

Внимание!!! Я настоятельно рекомендую использовать модуль `hideinv` и задавать свойство `hideinv` в диалогах. Диалоги будут выглядеть красивей и вы обезопасите свою игру от ошибок и непредсказуемых реакций при использовании инвентаря внутри диалога (так как обычно автор не подразумевает такие вещи). Например:

```
instead_version "1.8.2"
require "hideinv"
...
guarddlg = dlg {
        nam = 'Охранник';
        -- в диалогах обычно не нужен инвентарь
        hideinv = true;
        ...
}
```

Расширенные диалоги

Начиная с версии INSTEAD 1.7.0 поддерживается новый более простой и мощный синтаксис диалогов, по сравнению с устаревшим традиционным. Фразы определяются в атрибуте `phr` диалога, например:

```
povardlg = dlg {
    nam = 'На кухне';
        hideinv = true;
    entered = [[Передо мной полное лицо женщины -
                повара в белом колпаке и усталым взглядом...]];
    phr = {
        { always = true, 'Мне вот-этих зелененьких... Ага -- и бобов!',
                            'На здоровье!'};
```

```
        { always = true, 'Картошку с салом, пожалуйста!',
                            'Приятного аппетита!'};
        { always = true, 'Две порции чесночного супа!!!',
                            'Прекрасный выбор!' };
        { always = true, 'Мне что-нибудь легонькое, у меня язва...',
                            'Овсянка!' };
            { always = true, 'Спасибо, мне ничего не нужно.',
                            'Как пожелаете.', [[ back() ]] };
    };
};
```

Имейте в виду, что если в диалоге не определен атрибут `dsc`, то он формируется движком таким образом, чтобы отражать последнюю реакцию диалога, то есть, если игрок нажмет на заголовок сцены он увидит последний ответ на свою реплику еще раз. Если вы рассчитываете на такое поведение диалога, то первоначальную реакцию диалога удобнее всего вписать в `entered`, как в примере выше. Я не рекомендую переопределять `dsc` у диалогов расширенного типа.

Каждая фраза имеет вид:

```
{ [номер или tag=тэг,][false если выключена,][always = true],
         "Вопрос", "Ответ", [[ необязательный код  - реакция]] },
```

Фраза содержит *вопрос*, *ответ* и *реакцию*. Когда игрок выбирает одну из фраз (кликнув на нее), выводится ответ, фраза отключается, а затем срабатывает *реакция* (если она есть). Когда все фразы отключатся, *ветвь* диалога заканчивается. *Реакция* – это строка кода на lua, который выполнится после отключения фразы.

В реакции может быть любой lua код, но обычно, он содержит в себе логику по работе с фразами.

INSTEAD предоставляет следующие функции по работе с фразами:

- `pon(t..)` – включить фразы диалога с номерами или тегами t...
- `poff(t...)` – выключить фразы диалога с номерами или тегами t...
- `prem(t...)` – удалить (заблокировать) фразы диалога с номерами или тегами t... (удаление означает невозможность включения фраз. `pon(t...)` не приведет к включению фраз).
- `pseen(t...)` – вернет `true`, если все заданные фразы диалога (заданными номерами или тегами) видимы.
- `punseen(t...)` – вернет `true`, если все заданные фразы диалога (заданные номерами или тегами) невидимы.

Если параметр `t` не указан, действие относится к текущей фразе (в контексте которой был вызван код).

Если вам необходимо работать с фразами другого диалога (который не является в данный момент текущей комнатой), используйте вызовы в виде методов: `комната:метод()`, например, `guard_dlg:pon('show_card')`.

Вы можете определить в диалоге выключенную фразу, а затем, включить ее:

```
povardlg = dlg {
    nam = 'На кухне';
        hideinv = true;
    entered = [[Передо мной полное лицо женщины -
                повара в белом колпаке и усталым взглядом...]];
    phr = {
            -- выключенная фраза
            { 1, false, always = true,
                -- для наглядности, вы можете использовать переносы строк
                [[Дайте мне французских булок!]],
                [[Конечно...]] };
            -- знаем про булки, включить фразу
            { [[А что у вас там, на полке?]],
              [[Это французские булки.]],
              [[ pon(1) ]] };
            { always = true, 'Мне вот-этих зелененьких... Ага -- и бобов!',
                             'На здоровье!'};
            { always = true, 'Картошку с салом, пожалуйста!',
                             'Приятного аппетита!'};
            { always = true, 'Спасибо, мне ничего не нужно.',
                             'Как пожелаете.', [[ back() ]] };
    };
};
```

Итак, как вы уже поняли, для идентификации фразы вы можете использовать номер, например:

```
{ 2, "Вопрос?", "Ответ!" };
```

Для сложных диалогов более удобными являются теги, например:

```
{ tag = 'exit', "Ну ладно, я пошел!", code [[ back() ]] };
```

Если вам не нужно идентифицировать фразу, просто опускайте первое поле:

```
{ "Вопрос?", "Ответ!" };
```

Как видно, `тег` – это текстовая метка фразы. Как уже было замечено, вы можете делать `pon/poff/pseen/punseen` как с пронумерованными фразами, так и с фразами, имеющими тег. В случае, если один и тот же тег стоит у нескольких фраз, то действие применяется на все фразы с одинаковым тегом. Для функции `pseen`, видимость тега означает наличие хотя бы одной фразы с таким тегом, для функции `punseen` – отсутствие включенных фраз с заданным тегом.

Вы можете присваивать тег и пронумерованной фразе, если это требуется.

Присутствие в фразе `always = true` означает, что фраза не будет автоматически выключена при ее срабатывании:

```
{ tag = 'exit', always = true, "Ну ладно, я пошел!", code [[ back() ]] }
```

Если необходимо опустить ответ фразы и всю реакцию описать в параметре «необязательный код», то следующие варианты записи являются допустимыми:

```
{ tag = 'exit', always = true, "Ну ладно, я пошел!", nil, [[ back() ]] },
{ tag = 'exit', always = true, "Ну ладно, я пошел!", code = [[ back() ]] }
```

Вы также можете задавать *вопрос* и *ответ* в виде функций или `code`.

```
{ tag = 'exit', code [[ p "Ну ладно, я пошел!" ]],
                code [[ p "Может, останешься?"; pon 'really' ]] },
{ tag = 'really', false,
  always = true,
  "Я точно пошел!",
  function() back() end } -- эта фраза выключена и включается предыдущей
```

Вы можете группировать фразы диалога в *ветви*, тем самым организуя иерархические диалоги без необходимости массового использования `pon/poff` и перехода между несколькими `dlg`.

Группа фраз – это набор фраз, отделенная от другой группы фразой, у которой нет реакции (пример такой простейшей фразы это пара скобок {}). Например:

```
{ 'Расскажи что-нибудь о погоде?',
    'Хорошо, что тебя интересует?', [[ psub 'погода' ]] },
{ always=true, [[Пока!]], code = [[ back() ]] },
{ tag = 'погода' },
{ 'Какая температура?', '25 градусов!' },
{ 'Какая влажность?', '80%' },
```

В диалоге показывается только текущая группа фраз. В примере выше мы видим две группы. При входе в диалог, игрок увидит выбор из двух фраз: «Расскажи что-нибудь. . . » и «Пока!». Выбрав первую фразу, он попадет в подветку с тегом 'погода', в которой увидит два вопроса (о температуре и влажности). Когда он задаст оба вопроса, то переместится снова на первую ветку, где останется активной только одна фраза: «Пока!».

В данном примере группы разделены фразой: { tag = 'погода' }, но также точно разделителем могла стать пустая фраза без тега:

```
{ 'Расскажи что-нибудь о погоде?',
    'Хорошо, что тебя интересует?', [[ psub 'погода' ]] },
{ always="true", [[Пока!]], code = [[ back() ]] },
{ },
{ tag = 'погода', 'Какая температура?', '25 градусов!' },
{ 'Какая влажность?', '80%' },
```

Переход на ветку осуществляется с помощью команд:

- `psub` – переход с возвратом. Если все ответы ветки были исчерпаны или явно, с помощью `pret()`;
- `pjump` - безусловный переход;
- `pstart` - безусловный переход с обнулением истории переходов по `psub`.

В качестве аргумента `psub/pstart/pjump` может быть указан номер или тег. Вы можете использовать эти функции и извне диалога, аналогично `pon/poff` и т.д, с помощью записи: `диалог:метод()`, например: `shopdlg:pstart(1)`

Для того, чтобы узнать текущую подветку, используйте методы диалога `диалог:current()` и `диалог:curtag()`. Первый всегда возвращает номер, а второй – тег.

Проверку состояния ветки можно осуществить с помощью функций:

- `диалог:empty([t])`;

- `диалог:visible([t])`;

Обе функции могут получать параметр – номер или тег фразы, с которой начинается анализ группы. `:empty()` возвращает `true`, в случае, если в группе нет активных фраз. `:visible()` возвращает число видимых фраз (0 – если группа пуста). В случае, если параметр не указан, анализируется текущая группа.

В случае перехода по `psub/pstart/pjump`, первая фраза, на который выполняется переход, может служить заголовком группы фраз.

Например:

```
{ 'Расскажи что-нибудь о погоде?', code = [[ psub 'погода' ]] },
{ always=true, [[Пока!]], code = [[ back() ]] },
{ },
{ tag = 'погода', "Хорошо, что тебя интересует?" },
{ 'Какая температура?', '25 градусов!' },
{ 'Какая влажность?', '80%' },
```

Фраза с тегом 'погода' не содержит в себе реакцию, и выполняет роль заголовка ветки. При переходе на ветку 'погода' с помощью psub будет выведен текст «Хорошо, что тебя интересует?».

Как вы знаете, вопрос может быть функцией, тем самым позволяя выполнять код при переходе между ветками:

```
{ 'Расскажи что-нибудь о погоде?', code = [[ psub 'погода' ]] },
{ always=true, [[Пока!]], code = [[ back() ]] },
{ },
{ tag = 'погода', function()
          p "Хорошо, что тебя интересует?";
          weather_asked = true;
end },
{ 'Какая температура?', '25 градусов!' },
{ 'Какая влажность?', '80%' },
```

Кроме того, заголовок группы может содержать метод `empty`, который вызывается в ситуации, когда все вопросы данной группы исчерпаны:

```
{ 'Расскажи что-нибудь о погоде?', code = [[ psub 'погода' ]] },
{ always=true, [[Пока!]], code = [[ back() ]] },
{ },
```

```
{ tag = 'погода', "Хорошо, что тебя интересует?",
    empty = code [[ p "Хватит о погоде..."; pret() ]] },
{ 'Какая температура?', '25 градусов!' },
{ 'Какая влажность?', '80%' },
```

`empty` вызывается в ситуации, когда в ветке не остается фраз. Если empty не определена, то действие по-умолчанию это возврат по `pret()`. Если вы переопределяете `empty`, вам придется вызвать `pret()` явно, если это требуется.

На самом деле, написание диалога не такая сложная штука, как может показаться. В большинстве случаев используется небольшое подмножество возможностей движка.

Законченный пример реализации *сложного* диалога, вы можете посмотреть здесь: http://instead.googlecode.com/svn/trunk/doc/examples/dialog/main.lua

Простые диалоги

Данный фрагмент документации содержит описание устаревшего синтаксиса, тем не менее, если вам не понятна идея диалогов из предыдущей части, возможно, этот фрагмент вам поможет, так как некоторые вещи являются общими для обоих вариантов диалогов.

Простейший диалог в старом синтаксисие может выглядеть следующим образом:

```
povardlg = dlg {
    nam = 'На кухне';
    dsc = [[Передо мной полное лицо женщины -
                повара в белом колпаке и усталым взглядом...]];
    obj = {
    [1] = phr('Мне вот-этих зелененьких... Ага -- и бобов!', 'На здоровье!'),
    [2] = phr('Картошку с салом, пожалуйста!', 'Приятного аппетита!'),
    [3] = phr('Две порции чесночного супа!!!', 'Прекрасный выбор!'),
    [4] = phr('Мне что-нибудь легонькое, у меня язва...', 'Овсянка!'),
    };
};
```

`phr` – создание фразы. Фраза содержит *вопрос*, *ответ* и *реакцию* (реакция в данном примере отсутствует). Когда игрок выбирает одну из фраз, фраза отключается. Когда все фразы отключатся диалог заканчивается. *Реакция* – это строка кода на lua, который выполнится после отключения фразы. Например:

```
food = obj {
    nam = 'еда',
```

```
    inv = function (s)
        iremove('food', inv());
        p 'Я ем.';
    end
};

gotfood = function(w)
    take 'food';
    food._num = w;
    back();
end

povardlg = dlg {
    nam = 'На кухне';
    dsc = [[Передо мной полное лицо женщины -
                повара в белом колпаке и усталым взглядом...]];
    obj = {
    [1] = phr('Мне вот-этих зелененьких... Ага -- и бобов!',
                'На здоровье!', [[pon(); gotfood(1);]]),
    [2] = phr('Картошку с салом, пожалуйста!',
                'Приятного аппетита!', [[pon(); gotfood(2);]]),
    [3] = phr('Две порции чесночного супа!!!',
                'Прекрасный выбор!', [[pon();gotfood(3);]]),
    [4] = phr('Мне что-нибудь легонькое, у меня язва...',
                'Овсянка!', [[pon(); gotfood(4);]]),
    };
};
```

В данном примере, игрок выбирает еду. Получает ее (запомнив выбор в переменной food._num) и возвращается обратно (в ту сцену откуда попал в диалог).

В реакции может быть любой lua код, но, как и в случае с расширенными диалогами, обычно в нем содержится логика по управлению фразами. `pon/poff/prem/pseen/punseen` работают только с номерами (так как тегов здесь нет).

Вы можете переходить из одного диалога в другой диалог, организовывая иерархические диалоги.

Также, вы можете прятать некоторые фразы при инициализации диалога и показывать их при некоторых условиях.

```
facectrl = dlg {
    nam = 'Фэйсконтроль';
    dsc = 'Я вижу перед собой неприятное лицо полного охранника.';
    obj = {
```

```
        [1] = phr('Я пришел послушать лекцию Белина...',
        [[-- Я не знаю кто вы -- ухмыляется охранник --
                    но мне велели пускать сюда только приличных людей.]],
        [[pon(2);]]),
        [2] = _phr('У меня есть приглашение!',
        [[-- А мне плевать! Посмотри на себя в зеркало!!!
                    Ты пришел слушать самого Белина -- правую руку самого...
                    -- охранник почтительно помолчал -- Так что пошел вон..]], [[pon(3,4)]]),
        [3] = _phr('Сейчас я дам тебе по роже!',
                        '-- Ну все... Мощные руки выталкивают меня в коридор...',
            [[poff(4)]]),
        [4] = _phr('Ты, кабан! Я же тебе сказал -- у меня есть приглашение!',
            [[-- Чтоооооо? Глаза охранника наливаются кровью...
                        Мощный пинок отправляет меня в коридор...]],
            [[poff(3)]]),
    };
    exit = function(s,w)
        s:pon(1);
    end;
};
```

_phr – создает выключенную фразу, которую можно включить. Данный пример показывает также возможность использования методов pon, poff, prem для диалога (см. exit).

15. Облегченные объекты

Иногда сцену нужно наполнить декорациями, которые обладают ограниченной функциональностью, но делают игру разнообразней. Или вы хотите сделать что-то вроде кнопки «дальше». Для таких вещей можно использовать облегченный объект. Например:

```
sside = room {
    nam = 'Южная сторона';
    dsc = [[Я нахожусь у южной стены здания института. ]];
    act = function(s, w)
        if w == "подъезд" then
            ways():add('stolcorridor');
            p [[Я подошел к подъезду. На двери подъезда надпись --
                            'Столовая'. Хм -- зайти внутрь?]];
        elseif w == "люди" then
            p 'Те, кто выходят, выглядят более довольными...';
        end
```

```
    end;
    obj = { vobj("подъезд", "У восточного угла находится небольшой {подъезд}."),
        vobj("люди", [[Время от времени дверь подъезда хлопает
                                впуская и выпуская {людей}.]])},
};
```

Как видим, `vobj` позволяет сделать легкую версию статического объекта, с которым, тем не менее, можно взаимодействовать (за счет определения обработчика `act` в сцене и анализа имени объекта). `vobj` также вызывает метод `used`, при этом в качестве третьего параметра передается объект, воздействующий на виртуальный объект. Если вы используете предмет на `vobj`, то как и с обычными объектами, у предмета инвентаря вызовется `use`. Но объекты `vobj` обычно не имеют дескриптора, поэтому, для определения страдательного объекта можно воспользоваться `stead.nameof`.

```
use = function(s, w)
        if stead.nameof(w) == "люди" then
                p "Не стоит беспокоить людей."
                return
        end
end;
```

Синтаксис `vobj` прост: `vobj(имя, описатель);` `vobj` можно добавлять в сцену динамически, например:

```
put(vobj("дальше", "{Дальше}"));
```

Хотя я не рекомендую этот стиль, он больше характерен для старых версий движка. Нагляднее использовать `disable/enable`;

```
...
exist 'дальше':enable();
...
obj = { vobj("дальше", "{Дальше}"):disable() };
```

Существует модификация объекта `vobj` – `vway`. `vway` реализует ссылку-переход. Синтаксис `vway`: `vway(имя, описатель, сцена назначения);` например:

```
  obj = { vway("дальше", "Нажмите {здесь}.", 'nextroom') };
        -- при нажатии - перейдем в nextroom
```

На самом деле, если вы пишите что-то вроде книги-игры, где игровой процесс представляет из себя переход по ссылкам, то (если не считать, что это неудачная идея для вашей первой игры) вам следует воспользоваться модулем «xact», в котором реализован более простой механизм создания ссылок.

Вы можете динамически заполнять сцену объектами `vway` (аналогично `vobj`). Например:

```
  put(vway("next", "{Дальше}.", 'next_room'));
-- другой способ, явно вызывая метод списка
        objs():add(vway("next", "{Дальше}.", 'next_room'))
```

Следует понимать, что и `vobj` и `vway` это обычные объекты, с заранее определенными обработчиками и функциями сохранения (что позволяет создавать эти объекты на лету, как показано в примерах выше). Когда вы узнаете архитектуру движка INSTEAD, вы сможете писать свои варианты объектов с требуемыми свойствами.

Говоря об облегченных объектах, хотелось бы обратить внимание еще на один способ описания декораций. Если объект в сцене является статическим, то его можно определить непосредственно в `obj`, без присваивания дескриптора. Например:

```
hall = room {
        nam = 'Гостинная';
        dsc = [[Я в просторной гостинной.]];
        obj = {
                obj {
                        nam = 'стол';
                        dsc = [[Посреди гостинной находится {стол}.]];
                        act = [[Из красного дерева.]];
                };
        };
}
```

В обработчике `use` вы можете идентифицировать такие объекты также, как и `vobj`:

```
use = function(s, w)
        if stead.nameof(w) == 'стол' then
```

```
                p [[Не хочется портить красивую вещь.]]
                return
        end
end
```

Использовать или нет такую форму решать вам, многие считают, что присвоение дескриптора всем объектам делает код более понятным. Я в своих играх использую оба подхода.

Наконец, еще один способ для создания декораций, это использование одного и того-же объекта в разных сценах. Например, можно создать объект «гильзы дробовика», и выбрасывать его на сцену всегда, когда герой стреляет. Понятно, что в таком случае, гильзы могут служить только в качестве декораций, их нельзя будет взять и вообще произвести какое-то действие, меняющее состояние.

16. Динамические события

Вы можете определять обработчики, которые выполняются каждый раз, когда время игры увеличивается на 1. Обычно, это имеет смысл для живых персонажей, или каких-то фоновых процессов игры. Алгоритм шага игры выглядит примерно так:

1. Игрок нажимает на ссылку;
2. Реакция `act`, `use`, `inv`, осмотр сцены (клик по названию сцены) или переход в другую сцену;
3. Динамические события;
4. Вывод состояния сцены (если нужно статическая часть, и всегда – динамическая).

Например, сделаем Барсика живым:

```
mycat = obj {
    nam = 'Барсик';
    lf = {
        [1] = 'Барсик шевелится у меня за пазухой.',
        [2] = 'Барсик выглядывает из-за пазухи.',
        [3] = 'Барсик мурлычит у меня за пазухой.',
        [4] = 'Барсик дрожит у меня за пазухой.',
        [5] = 'Я чувствую тепло Барсика у себя за пазухой.',
        [6] = 'Барсик высовывает голову из-за пазухи и осматривает местность.',
    };
    life = function(s)
```

```
        local r = rnd(5);
        if r > 2 then -- делать это не всегда
            return;
        end
        r = rnd(#s.lf); -- символ # -- число элементов в массиве
        p(s.lf[r]); -- выводим одно из 6 состояний Барсика
    end;
....
-- и вот момент в игре, когда Барсик попадает к нам за пазуху!
take 'mycat' -- добавить в инвентарь
lifeon 'mycat' -- оживить Барсика!
....
```

Любой объект (в том числе и сцена) могут иметь свой обработчик `life`, который вызывается каждый такт игры, если объект был добавлен в список живых объектов с помощью `lifeon`. Не забывайте удалять живые объекты из списка с помощью `lifeoff`, когда они больше не нужны. Это можно сделать, например, в обработчике `left`, или любым другим способом.

Если в вашей игре много «живых» объектов, вы можете задавать им приоритеты. Для этого, воспользуйтесь вторым числовым параметром (целое неотрицательное число) `lifeon`, чем меньше число, тем выше приоритет. 1 – самый высокий.

Если вам нужен фоновый процесс в какой-то комнате, запускайте его в `entered` и удаляйте в `left`, например:

```
podval = room {
        nam  = 'В подвале';
        dsc = [[Тут темно!]];
        entered = function(s)
                lifeon(s);
        end;
        left = function(s)
                lifeoff(s);
        end;
        life = function(s)
                if rnd(10) > 8 then
                        p [[Я слышу какие-то шорохи!]];
                        -- изредка пугать игрока шорохами
                end
        end;
        way =  { 'upstair' };
}
```

Если вам нужно определить, был ли переход игрока из одной сцены в другую, воспользуйтесь `player_moved`.

```
flash = obj {
        nam  = 'фонарик';
        var { on = false };
        life = function(s)
                if player_moved() then -- гасить фонарик при переходах
                        s.on = false
                        p "Я выключил фонарик."
                        return
                end
        end;
...
}
```

Для отслеживания протекающих во времени событий, используйте `time()` или вспомогательную переменную-счетчик. Для определения местоположения игрока – `here()`. Для определения факта, что объект «живой» – `live()`.

```
dynamite = obj {
        nam  = 'динамит';
        var {
                timer = 0;
        };
        used = function(s, w)
                if w == fire then
                        if live(s) then
                                return "Уже горит!"
                        end
                        p "Я поджег динамит."
                        lifeon(s)
                end
        end;
        life = function(s)
                s.timer = s.timer + 1
                if s.timer == 5 then
                        lifeoff(s)
                        if here() == where(s) then
                                p [[Динамит взорвался рядом со мной!]]
                        else
                                p [[Я услышал, как взорвался динамит.]];
                        end
```

```
                end
        end;
...
}
```

Если `life` обработчик возвращает текст события, он печатается после описания сцены.

Вы можете вернуть из обработчика `life` второй код возврата, важность. (`true` или `false`). Например:

```
    p 'В комнату вошел охранник.'
    return true
```

Или:

```
    return 'В комнату вошел охранник.', true
```

При этом текст события будет выведен *до* описания объектов.

Если вы хотите блокировать `life` обработчики в какой-то из комнат, воспользуйтесь модулем `nolife`. Например:

```
instead_version "1.8.2"
require "hideinv"
require "nolife"

guarddlg = dlg {
        nam = 'Охранник';
        hideinv = true;
        nolife = true;
...
}
```

Отдельно стоит рассмотреть вопрос перехода игрока из `life` обработчика. Если вы собираетесь использовать функции `walk...` внутри `life`, то вам следует учитывать следующее поведение.

Если `life` переносит игрока в новую локацию, то подавляется весь вывод, который произведен `life` обработчиками объектов с прошлой сцены (то есть, все предыдущие `life` выводы). Остается только вывод `life` обработчиков, сработавших после перехода. Это сделано специально, так как вывод прошлых `life` относился к прошлой сцене, например:

1. `life` сцены `скала` вывел текст о том, что герою страшно, когда он висит на тросе;
2. `life` объекта `трос` вывел текст о том, что трос оборвался и герой упал вниз и сделал `walk` в новую локацию `море`;

Здесь `life1` выполнялся в ином контексте и его вывод подавляется.

Кроме всего прочего, обработчик `life` может влиять на текст реакций действий игрока, которые произошли в этом игровом такте. Например, рассмотрим такую ситуацию:

1. Игрок осмотрел окно («Я выглянул в окно. Унылый пейзаж.»);
2. `life` обработчик `гоблин` сообщил, что внезапно дверь открылась и в комнату влетел гоблин;

Автору игры может показаться, что информация о пейзаже, когда перед игроком стоит свирепый гоблин, неуместна. Тогда он пишет в `life` обработчике:

```
    p [[Свирепый гоблин влетел в комнату!]];
    ACTION_TEXT = nil
    -- текст реакции пуст (раньше он был равен
    -- "Я выглянул в окно. Унылый пейзаж.")
```

Таким образом, `ACTION_TEXT` это текстовая переменная, доступная в `life` обработчике для модификации. Обычно, имеет смысл или не трогать ее, или обнулять, как в примере выше.

17. Графика

Графический интерпретатор INSTEAD анализирует атрибут сцены `pic`, и воспринимает его как путь к картинке, например:

```
home = room {
    pic = 'gfx/home.png';
    nam = 'Дома';
    dsc = 'Я у себя дома';
};
```

Важно!

Используйте в путях только прямые '/'. Также, настоятельно рекомендуется использовать в именах каталогов и файлов только латинские строчные символы. Этим самым вы

обезопасите свою игру от проблем с совместимостью и она будет работать на всех архитектурных платформах, куда портирован INSTEAD.

Конечно, `pic` может быть функцией, расширяя возможности разработчика. Если в текущей сцене не определен атрибут `pic`, то берется атрибут `game.pic`. Если не определен и он, то картинка не отображается.

Поддерживаются все наиболее распространенные форматы изображений, но я рекомендую вам использовать `png` и (когда важен размер) `jpg`.

Вы можете использовать в качестве картинок анимированные gif файлы.

Вы можете встраивать графические изображения прямо в текст, в том числе в инвентарь, переходы, заглавия комнат и `dsc` с помощью функции `img`. Например:

```
apple = obj {
        -- склеить текстовую строку с изображением
        nam = 'яблоко '..img ('img/apple.png');
}
```

Хотя, в данном случае предпочтительнее воспользоваться `disp`:

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

Теперь мы разделили имя объекта и его отображение.

Тем-не менее, картинку сцены всегда следует оформлять в виде `pic` атрибута, а не вставки `img` в `dsc` комнаты.

Дело в том, что картинка сцены масштабируется по другому алгоритму. Картинки `img` масштабируются в соответствии с настройками INSTEAD (масштаб темы), а `pic` – учитывает также размер картинки.

Кроме того, картинки `pic` обладают и другими свойствами, например, возможностью отслеживания координат кликов мышью.

Если вы поместите `img` внутрь { и }, то получите графическую ссылку.

```
apple = obj {
    nam = 'яблоко';
        disp = 'яблоко ' ..img('img/apple.png');
        dsc = function(s)
```

```
                p ("На полу лежит {яблоко",
                       img 'img/apple.png',
                        "}");
                -- другие варианты:
                -- return "На полу лежит {яблоко"..img('img/apple.png')..."}";
                -- p "На полу лежит {яблоко"..img('img/apple.png')..."}";
                -- или dsc = "На полу лежит {яблоко"..img('img/apple.png')..."}";
        end;
}
```

INSTEAD поддерживает обтекание картинок текстом. Если картинка вставляется с помощью функции `imgl`/`imgr`, она будет расположена у левого/правого края.

Важно!

Картинки, вставленные в текст с помощью `imgl/imgr` не могут быть ссылками!!! Используйте их только в декоративных целях.

Для задания отступов вокруг изображения используйте `pad`, например:

```
imgl 'pad:16,picture.png' -- отступы по 16 от каждого края
imgl 'pad:0 16 16 4,picture.png' -- отступы: вверху 0, справа 16, внизу 16, слева 4
imgl 'pad:0 16,picture.png' -- отступы: вверху 0, справа 16, внизу 0, слева 16
```

Вы можете использовать псевдо-файлы для изображений прямоугольников и пустых областей:

```
dsc = img 'blank:32x32'..[[Строка с пустым изображением.]];
dsc = img 'box:32x32,red,128'..[[Строка красным полупрозрачным квадратом.]];
```

INSTEAD может обрабатывать составные картинки, например:

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

Таким образом, составная картинка представляет собой набор путей к изображениям, разделенных символом `;`. Вторая и последующие компоненты могут содержать постфикс в виде @x_координата,y_координата, где координате 0,0 соответствует левый верхний угол всего изображения. Общий размер картинки считается равным общему размеру первой компоненте составной картинки, то есть, первый компонент (в нашем примере – gfx/mycat.png) играет роль холста, а последующие компоненты накладываются на этот холст.

Наложение происходит для левого верхнего угла накладываемой картинки. Если вам нужно, чтобы наложение происходило относительно центра накладываемой картинки, используйте перед координатами префикс `c`, например:

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

Оформив в виде функции формирование пути составной картинки, вы можете генерировать изображение на основе игрового состояния.

Если вы в своей игре привязываетесь к каким-то координатам изображений, или к их размерам, делайте это относительно оригинальных размеров изображений. При масштабировании темы под заданное игроком разрешение, INSTEAD сам будет осуществлять пересчёт координат (при этом координаты для игры выглядят так, как будто игра запущена без масштабирования). Однако, возможны небольшие погрешности вычислений.

Если вам не хватает функций, описанных в этой главе, изучите модуль «sprites», который предоставляет более широкие возможности по графическому оформлению. Но я крайне не рекомендую делать это в своей первой игре.

18. Музыка

Интерпретатор проигрывает в цикле текущую музыку, которая задается с помощью функции: `set_music(путь к музыкальному файлу)`.

Важно!

Используйте в путях только прямые '/'. Также, настоятельно рекомендуется использовать в именах каталогов и файлов только латинские строчные символы. Этим самым вы обезопасите свою игру от проблем с совместимостью и она будет работать на всех архитектурных платформах, куда портирован INSTEAD.

Поддерживается большинство музыкальных форматов, но настоятельно рекомендуется использовать формат `ogg`, так как именно он поддерживается наилучшим образом во всех версиях INSTEAD (для различных платформ).

Важно!

Следует проявлять осторожность при использовании трекерной музыки, так как в некоторых дистрибутивах Linux могут быть проблемы при проигрывании определенных файлов (ошибки в связке библиотек SDL_mixer и libmikmod).

Также, если вы используете `mid` файлы, будьте готовы к тому, что игрок услышит их только в Windows версии INSTEAD (так как в большинстве случаев, Unix версии SDL_mixer собраны без поддержки `timidity`).

Например:

```
street = room {
    pic = 'gfx/street.png';
    enter = function()
        set_music('mus/rain.ogg')
    end;
    nam = 'на улице';
    dsc = 'На улице идет дождь.';
};
```

`get_music()` возвращает текущее имя трека.

В функцию `set_music()` можно передавать второй параметр – количество проигрываний (циклов). Получить текущий счетчик можно с помощью `get_music_loop`. 0 – означает вечный цикл. 1..n – количество проигрываний. -1 – проигрывание текущего трека закончено.

Часто бывает необходимым сменить музыку на время, а затем восстановить предыдущий трек. Для этого можно воспользоваться функциями `save_music()/restore_music()`. Эта пара функций запоминает/восстанавливает трек в/из переменных объекта текущего контекста. Например, для обработчиков `enter/exit/entered/left` это будет текущая комната.

Если вы хотите явно задать объект, в котором будет сохранено состояние (из которого будет восстановлено состояние) трека, укажите его в качестве необязательного параметра. Данные функции не работают с ссылками, поэтому вы не можете передавать в них текстовые строки-указатели на объекты.

Например:

```
street = room {
    pic = 'gfx/street.png';
    entered = function()
                save_music();
        set_music('mus/rain.ogg')
    end;
        left = restore_music;
    nam = 'на улице';
    dsc = 'На улице идет дождь.';
};
```

Обратите внимание, что в примере `left` обработчику присваивается значение `restore_music`. Отсутствие () после `restore_music` означает, что это присваивание самого кода функции, а не ее вызов. Таким образом, при вызове обработчика `left`, будет вызвана `restore_music` с параметром равным `street`, что нам и требуется. Впрочем, можно было написать и такой код:

```
left = function()
        restore_music()
end;
```

Или:

```
restore_music(street)
```

Для того, чтобы отменить проигрывание музыки, вы можете использовать `stop_music()` Функция `is_music()` позволяет узнать, проигрывается ли музыка в данный момент.

Для проигрывания звуков используйте `set_sound()`. Настоятельно рекомендуется использовать формат `ogg`, хотя большинство распространенных звуковых форматов также будет работать.

Различие между музыкой и звуковым файлом заключается в том, что движок следит за процессом проигрывания музыки и сохраняет/восстанавливает текущий проигрываемый трек. Выйдя из игры и загрузив ее снова, игрок услышит то же музыкальное оформление, что слышал при выходе. Звуки обычно означают кратковременные эффекты, и движок не сохраняет и не восстанавливает звуковые события. Так, если игрок не успел дослушать звук выстрела и вышел из игры, после загрузки файла сохранения он не услышит звук (или его окончание) снова.

Тем не менее, если учесть то, что `set_sound` позволяет запускать зацикленные звуки, то различие между музыкой и звуками становится уже не таким однозначным.

Итак, определение функции: `set_sound(файл, [канал], [цикл])`, где:

- файл – путь и\или имя звукового файла;
- канал – номер канала [0..7]; Если не указан, то выберется первый свободный.
- цикл – количество проигрываний 1..n, 0 – зацикливание.

Для остановки проигрывания звука можно использовать `stop_sound()` Для остановки звука в определенном канале `stop_sound(канал)`.

На самом деле, `set_sound` имеет одну особенность. Если вы вызовете эту функцию подряд несколько раз, то эффект принесет только последний вызов. Если вам нужно запускать несколько одновременных звуков за один такт игры, используйте `add_sound()`. Параметры к функции имеют тот же смысл, что и у `set_sound()`.

Если вы используете зацикленные звуки, вам придется самим восстанавливать их состояние (запускать снова с помощью `set_sound()/add_sound()`) в функции `start()`.

Если вам не достаточно описанных здесь функций по работе со звуком, используйте модуль «sound».

19. Форматирование и оформление вывода

Обычно INSTEAD сам занимается форматированием и оформлением вывода. Например, отделяет статическую сцену от динамической. Выделяет курсивом действия игрока. Переводит фокус на изменение в тексте и т.д. Модули вроде «quotes», «para» и подобные улучшают качество вывода игры без дополнительных усилий со стороны автора.

Например, я рекомендую всегда включать в вашу игру следующие модули:

```
instead_version "1.8.2"
require "para" -- отступы параграфов
requure "dash" -- замена двойного минуса на длинное тире
require "quotes" -- красивые кавычки
```

И ваша игра будет выглядеть гораздо лучше. Если вам нужна какая-то автоматическая обработка выводимого текста, вы можете включить модуль «format» и определить функцию `stead.filter`. Например:

```
instead_version "1.8.2"
require "format"
stead.filter = function(s)
        return s..'^Эта строка будет добавлена к выводу';
end
```

Многие хорошие игры на INSTEAD никак не занимаются своим оформлением, кроме разбиения текста `dsc` на параграфы с помощью символов '^^', поэтому подумайте, а так ли вам хочется заниматься оформлением своей игры вручную?

Тем не менее, иногда это все-таки необходимо.

Форматирование

Вы можете делать простое форматирование текста с помощью функций:

- `txtc(строка)` - разместить по центру;
- `txtr(строка)` - разместить справа;
- `txtl(строка)` - разместить слева;
- `txttop(строка)` - сверху строки;
- `txtbottom(строка)` - снизу строки;

- `txtmiddle(строка)` - середина строки (по умолчанию);

Например:

```
main = room {
    nam = 'Intro';
    dsc = txtc 'Добро пожаловать!'; -- если у функции только 1 параметр,
        -- скобки можно опускать;
}
```

Вышеописанные функции влияют не только на текст, но и на изображения, вставленные с помощью `img()`. Следует отметить, что если вы используете несколько функций форматирования, то предполагается, что они относятся к разным строкам (параграфам). В противном случае, результат не определен. Разбивайте текст на абзацы символами ^ или `pn()`.

INSTEAD при выводе удаляет лишние пробелы. Это значит, что неважно сколько пробелов вы вставляете между словами, все равно при выводе они не будут учитываться для расчета расстояния между словами. Иногда это может стать проблемой.

Вы можете создавать *неразрывные строки* с помощью: `txtnb(строка)`. Например, модуль «para» использует неразрывные строки для создания отступов в начале параграфов. Также, `txtnb` может оказаться удобной для вывода служебных символов. Можно сказать, что вся строка-параметр `txtnb` воспринимается движком как одно большое слово.

Еще один пример. Если вы используете подчеркивание текста, то промежутки между словами не будут подчеркнуты. При использовании `txtnb` промежутки также будут подчеркнуты.

INSTEAD не поддерживает отображение таблиц, однако для вывода простых табличных данных можно воспользоваться `txttab()`. Эта функция используется для абсолютного позиционирования в строке (табулятор).

`txttab(позиция, [центр])`

Позиция, это текстовый или числовой параметр. Если задан числовой параметр, он воспринимается как позиция в пикселях. Если он задан в виде строкового параметра `число%`, то он воспринимается как позиция, выраженная в процентах от ширины окна вывода сцены.

Необязательный строковой параметр *центр* задает позицию в следующем за `txttab` слове, которая будет размещена по указанному смещению в строке. Позиции могут быть следующими:

- left;
- right;
- center;

По-умолчанию считается что задан параметр «left».

Так, например:

```
main = room {
        nam = 'Начало';
        -- размещение 'Начало!' по центру строки
        dsc = txttab('50%', 'center')..'Начало!';
}
```

Конечно, не очень удачный пример, так как то же самое можно было сделать с помощью `txtc()`. Более удачный пример.

```
main = room {
        nam = 'Начало';
        dsc = function(s)
                p(txttab '0%')
                p "Слева";
                p(txttab '100%', 'right')
                p "Справа";
        end
}
```

На самом деле, единственная ситуация, когда применение `txttab` оправдано – это вывод табличных данных.

Следует отметить, что в ситуации, когда мы пишем что-то вроде:

```
        -- размещение 'Раз' по центру строки
        dsc = txttab('50%', 'center')..'Раз два три!';
```

Только слово 'Раз' будет помещено в центр строки, остальные слова будут дописаны справа от этого слова. Если вы хотите центрировать 'Раз два три!' как одно целое, воспользуйтесь `txtnb()`.

```
        -- размещение 'Раз два три!' по центру строки
        dsc = txttab('50%', 'center')..txtnb 'Раз два три!';
```

По умолчанию, статическая часть сцены отделяется от динамической двойным переводом строки. Если вам это не подходит, вы можете переопределить `stead.scene_delim`, например:

```
instead_version "1.8.2"
stead.scene_delim = '^' -- одинарный перевод строки
```

Вы не можете менять эту переменную в обработчиках, так как она не сохраняется, но вы можете задать ее для игры целиком, или восстанавливать ее вручную в функции `start()`.

Если вас категорически не устраивает то, как INSTEAD формирует вывод (последовательность абзацов текста), вы можете переопределить функцию `iface.fmt`, которая по умолчанию выглядит следующим образом:

```
iface.fmt = function(self, cmd, st, moved, r, av, objs, pv)
-- st -- changed state (main win), move -- loc changed
    local l, vv
    if st then
        av = txtem(av); -- вывод "важных" life
        pv = txtem(pv); -- вывод обычных life
        r = txtem(r) -- реакция на действие
        if isForcedsc(stead.here()) or NEED_SCENE then
            l = stead.here():scene(); -- статическая часть сцены
        end
    end
    if moved then -- компонуем вывод для случая, когда игрок перешел в новую комнату
        vv = stead.fmt(stead.cat(
                    stead.par(stead.scene_delim, r, l, av, objs, pv), '^'));
    else -- компонуем вывод, когда игрок не делал перехода
        vv = stead.fmt(stead.cat(
                    stead.par(stead.scene_delim, l, r, av, objs, pv), '^'));
    end
    return vv
end
```

Тот факт, что я привел здесь этот код, не означает, что я рекомендую переопределять эту функцию. Напротив, я категорически против такой сильной привязки к форматированию текста. Тем не менее, иногда возникает ситуация, когда полный контроль за последовательностью вывода необходим. Если вы пишите свою первую игру, просто пропустите этот текст.

Оформление

Вы можете менять начертание текста с помощью комбинаций функций:

- `txtb(строка)` - жирный текст;

- `txtem(строка)` - курсив;
- `txtu(строка)` - подчеркнутый текст;
- `txtst(строка)` - перечеркнутый текст;

Например:

```
main = room {
    nam = 'Intro';
    dsc = function()
                p ('Вы находитесь в комнате ')
                p (txtb 'main', '.');
        end;
}
```

Строго говоря, INSTEAD не поддерживает одновременный вывод разными шрифтами в окно сцены (если не считать разное начертание), поэтому если вам все-таки требуется более гибкий контроль вывода, вы можете сделать следующее:

- Использовать графические вставки `img()`;
- Использовать модуль `fonts`, в котором реализована отрисовка разными шрифтами за счет модуля `sprite`;
- Использовать другой движок, так как скорее всего вы используете INSTEAD не по назначению.

20. Конструкторы и наследование

Внимание!

Если вы пишите свою первую игру, было бы лучше, если бы она была простая. Для простой игры информация из этой главы не понадобится. Более того, 90% игр на INSTEAD не использует вещей, описанных в этой главе!

Если вы пишите игру, в которой много однотипных объектов, возможно, вам захочется упростить их создание. Конструктор, это функция, которая создает объект. На самом деле конструкции `obj`, `room`, `dlg` – все это конструкторы. Когда вы пишите что-то вроде:

```
apple = obj {
        nam = 'яблоко';
}
```

Вызывается функция `obj` в качестве параметра к которой передается таблица { nam = ‘яблоко’ }; Зная это, вы можете писать свои конструкторы. Например, рассмотрим такую задачу. Нужно создавать окна, любое окно можно разбить. Мы можем написать конструктор `window`.

```
window = function(v)
        v.window = true
        if v.nam == nil then
                v.nam = 'окно'
        end
        if v.dsc == nil then
                v.dsc = 'Здесь есть {окно}'
        end
        v.act = function(s)
                if s._broken then
                        p [[Окно разбито.]]
                else
                        p [[За окном темно.]]
                end
        end
        if v.used == nil then
                v.used = function(s, w)
                        if w == hammer then
                                if s._broken then
                                        p [[Окно уже разбито.]]
                                else
                                        p [[Я разбил окно.]]
                                        s._broken = true;
                                end
                        end
                end
        end
        return obj(v)
end
```

Мы видим, что функция `window` заполняет некоторые атрибуты и обработчики (позволяя игроку переопределить некоторые из них), а потом вызывает функцию создания объекта и возвращает новенький объект. Теперь, можно создавать объекты окна:

```
win1 = window {
        dsc = "В восточной стене есть {окно}.";
}
```

Или, так как окно это обычно статический объект, можно создавать его прямо в `obj`.

```
obj = { window {
            dsc = 'В восточной стене есть {окно}.';
      }
};
```

Если вам нравится более классический синтаксис оформления конструктора в виде функции, принимающей несколько параметров вместо одной таблицы (примеры: `vroom`, `vobj`, `vway` и подобные), то можно было бы определить конструктор так:

```
window = function(nam, dsc)
        local v = {} -- создаем пустую таблицу
        -- заполняем ее
        v.window = true
        v.nam = 'окно'
        if dsc == nil then
                v.dsc = 'Здесь есть {окно}'
        end
        v.act = function(s)
                if s._broken then
                        p [[Окно разбито.]]
                else
                        p [[За окном темно.]]
                end
        end
        v.used = function(s, w)
                if w == hammer then
                         if s._broken then
                                 p [[Окно уже разбито.]]
                         else
                                 p [[Я разбил окно.]]
                                 s._broken = true;
                         end
                end
        end
        return obj(v) -- создаем объект
end
```

Тогда вызов конструктора будет выглядеть по-другому:

```
obj = { window ('окно', 'В восточной стене есть {окно}.') }
```

На самом деле, оба подхода применимы, но в разных ситуациях. Если для создания объекта достаточно указать два-три атрибута, то проще и наглядней оформлять конструктор как функцию с несколькими параметрами (как `vroom`).

Если же предполагается, что объекту могут быть присвоены различные (в том числе и необязательные) атрибуты, то проще делать конструктор в виде функции, принимающей таблицу (как `obj/room` и др.)

Отдельно стоит рассмотреть вопрос добавления новых переменных. В нашем примере мы использовали переменную с префиксом `_`, так как такие переменные можно создавать на лету. Другим способом является использование `stead.add_var()`, например:

```
window = function(v)
        stead.add_var(v, { broken = false }) -- добавить переменные к ''v'';
        v.window = true
-- ... пропущено
        if v.used == nil then
                v.used = function(s, w)
                        if w == hammer then
                                if s.broken then
                                        p [[Окно уже разбито.]]
                                else
                                        p [[Я разбил окно.]]
                                        s.broken = true;
                                end
                        end
                end
        end
        return obj(v)
end
```

Если вы хотите добавить глобальную переменную (на лету), используйте синтаксис:

```
stead.add_var { global_var = 1 }
-- синоним записи stead.add_var ({ global_var = 1 })
```

Теперь, если вы поняли что-такое конструктор, вы можете перейти к такому понятию, как наследование.

На самом деле, в примерах выше уже используется наследование. Действительно, ведь конструктор `window` вызывает другой конструктор `obj`, тем самым получая все свойства обычного объекта. Также, `winodw` определяет переменную признак `window`, чтобы в игре мы могли понять, что мы имеем дело с окном. Например:

```
use = function(s, w)
        if w.window then
                p [[Я посветил фонариком в окно.]]
                return
        end
end
```

Для иллюстрации механизма наследования создадим класс объектов `treasure`, те. сокровищ.

```
global { score = 0 }
treasure = function()
        local v = {}
        v.nam = 'сокровище'
        v.treasure = true
        v._points = 100
        v.dsc = function(s)
                p ('Здесь есть {', stead.dispof(s), '}.')
        end;
        v.inv = function(s)
                p ('Это же ', stead.dispof(s), '.');
        end;
        v.tak = function(s)
                score = score + s.points; -- увеличим счет
                p [[Дрожащими руками я забрал сокровища.]];
        end
        return obj(v)
end
```

А теперь, на его основе создадим золото, алмаз и сундук.

```
gold = function(dsc)
        local v = treasure();
        v.nam = 'золото';
        v.gold = true;
        v.points = 50;
```

```
        v.dsc = dsc;
        return v
end
diamond = function(dsc)
        local v = treasure();
        v.nam = 'алмаз';
        v.diamond = true;
        v.points = 200;
        v.dsc = dsc;
        return v
end
chest = function(dsc)
        local v = treasure();
        v.nam = 'сундук';
        v.chest = true
        v.points = 1000;
        v.dsc = dsc;
        return v
end
```

Теперь, в игре можно создавать сокровища через конструкторы:

```
diamond1 = diamond("В грязи я заметил {алмаз}.")
diamond2 = diamond(); -- тут будет стандартное описание алмаза
gold1 = gold("В углу я заметил блеск {золота}.");
cave = room {
        nam = 'пещера';
        obj = {
                'diamond1',
                'gold1',
                chest("А еще я вижу {сундук}!")
        };
}
```

На самом деле, как именно писать функции-конструкторы и реализовывать принцип наследования, зависит только от вас. Выберете наиболее простой и понятный способ.

При написании конструкторов иногда бывает полезным сделать вызов обработчика так, как это делает INSTEAD. Для этого используется `stead.call(объект, метод, параметры)`, при этом эта функция вернет реакцию атрибута в виде строки. Например, рассмотрим модификацию `window`, которая заключается в том, что можно определять свою реакцию на осмотр окна, которая будет выполнена после стандартного сообщения о том, что это разбитое окно (если оно разбито).

```
window = function(nam, dsc, what)
        local v = {} -- создаем пустую таблицу
        -- заполняем ее
        v.window = true
        v.nam = 'окно'
        v.what = what
        if dsc == nil then
                v.dsc = 'Здесь есть {окно}'
        end
        v.act = function(s)
                if s._broken then
                        p [[Окно разбито.]]
                end
                local r = stead.call(s, 'what')
                if r then
                        p(r)
                else
                        p [[За окном темно.]]
                end
        end
...
```

Таким образом, мы можем при создании окна задать третий параметр, в котором определить функцию или строку, которая будет реакцией во время осмотра окна. При этом сообщение о том, что окно разбито (если оно действительно разбито), будет выведено перед этой реакцией.

В качестве завершающего примера, рассмотрим свою версию реализации `vway`, назовем ее `xway` (Впрочем, в `lua` можно переопределять функции. Вы можете, например, написать свою версию `vway` и она заменит ту, что определена в INSTEAD).

```
function xway(name, dsc, w)
        return  obj {
                nam = name;
                dsc = dsc;
                act = function(s)
                        walk(s.where)
                end;
                where = w;
       };
end
```

Как видим, `xway` реализован как объект, который при клике выполняет `xwalk` в заданную

при создании `xway` комнату.

На самом деле, у `xway` есть один недостаток, по сравнению с `vway`. Это невозможность создания объектов на лету, например, если вы напишите:

```
put (xway('в пещеру', '{В пещеру}', 'cave'));
```

То INSTEAD не сможет сохранить этот объект. Это произойдет потому, что вообще говоря, INSTEAD умеет сохранять только те объекты, которые имеют дескрипторы. Объекты, имеющие дескрипторы, это объекты созданные одним из следующих способов:

1. Явным присвоением дескриптору в глобальном контексте (apple = obj {...);
2. Явным созданием внутри списков `obj` и `way` (дескриптором станет элемент массива);
3. Через механизм `new` (описано ниже).

В случае же с `put` мы создаем абсолютно безымянный объект. Тем не менее существует способ сделать объекты сохраняемыми в любом случае. То, что я опишу далее, вряд ли стоит использовать при написании игр. Обычно такие вещи имеют смысл при разработке модулей или расширений движка, так что вы можете спокойно пропустить эту часть.

Итак, для сохранения можно определить свою функцию *сохранения*.

```
function xway(name, dsc, w)
        return  obj {
                nam = name;
                dsc = dsc;
                act = function(s)
                        walk(s.where)
                end;
                where = w;
                save = function(self, name, h, need)
                        -- self - текущий объект
                        -- name -- полное имя переменной
                        -- h - файловый дескриптор файла сохранения
                        -- need - признак того, что это создание объекта,
                        -- файл сохранения должен создать объект
                        -- при загрузке
                        local dsc = self.dsc;
                        local w = stead.deref(self.where);
                        if need then
                        -- в случае создания, запишем строку
                        -- с вызовом конструктора
```

```
                                h:write(stead.string.format(
                                        "%s  = xway(%s, %s, %s);\n",
                                        name,
                                        -- формирование строк
                                        stead.tostring(self.nam),
                                        stead.tostring(dsc),
                                        stead.tostring(w)));
                        end
                        stead.savemembers(h, self, name, false); -- сохраним все
                        -- остальные переменные объекта
                        -- например, состояние включен/выключен
                        -- итд
                        -- false в последней позиции означает что будет
                        -- передано в save-методы вложенных объектов в
                        -- качестве параметра need
                end
        };
end
```

На самом деле, я надеюсь, что вам никогда не понадобится разбираться с движком на таком глубоком уровне.

21. Полезные советы

Разбиение на файлы

Когда ваша игра становится большой, размещение ее кода целиком в `main.lua` – плохая идея.

Для разбиения текста игры на файлы вы можете использовать `dofile`. Вы должны использовать `dofile` в глобальном контексте таким образом, чтобы во время загрузки `main.lua` загрузились и все остальные фрагменты игры, например.

```
-- main.lua
instead_version "1.8.2"
dofile "episode1.lua"
dofile "npc.lau"
dofile "start.lua"

main = room {
....
```

Как именно разбивать исходный текст на файлы зависит только от вас. Я использую файлы в соответствии с эпизодами игры (которые обычно слабо связаны между собой), но можно создавать файлы, хранящие отдельно комнаты, объекты, диалоги и т.д. Это вопрос личного удобства.

Также есть возможность динамически подгружать части игры (с возможностью переопределения существующих объектов). Для этого вы можете воспользоваться функцией `gamefile`:

```
...
act = code [[ gamefile ("episode2.lua"); ]]
...
```

`gamefile` также позволяет загрузить новый файл и забыть стек предыдущих загрузок, запустив этот новый файл как самостоятельную игру. Для этого, задайте второй параметр функции как `true`. `gamefile` можно использовать только в обработчиках.

```
...
act = code [[ gamefile ("episode3.lua", true); ]]
...
```

Во втором варианте `gamefile` можно использовать для оформления мультиязычных игр или игр-сборников, где фактически из оболочки выполняется запуск самостоятельной игры.

Модули

Дополнительная функциональность часто реализована в INSTEAD в виде модулей. Для использования модуля необходимо после «instead_version» написать:

```
require "имя модуля"
```

Часть модулей входит в поставку INSTEAD, но есть и такие, которые вы можете скачать отдельно и положить в каталог с игрой. Вы можете заменить любой стандартный модуль своим, если положите его в каталог с игрой под тем-же именем файла, что и стандартный.

Модуль, это фактически `lua` файл с именем: `имя_модуля.lua`.

Самая свежая документация по модулям размещена здесь: http://instead.syscall.ru/wiki/ru/gamedev/modules

Подробнее о модулях

Ниже перечислены основные модули, с указанием функциональности, которые они предоставляют.

- `dbg` — модуль отладки (`require «dbg»` – включить отладчик);
- `xact` — реализация ссылок;
- `click` — модуль перехвата кликов мыши по картинке сцены;
- `prefs` — модуль настроек (хранилище данных настроек);
- `snapshots` — модуль поддержки снапшотов (для откатов игровых ситуаций);
- `format` — модуль оформления вывода;
- `theme` — управление темой на лету;
- `hideinv` - модуль работы с инвентарем;
- `kbd` - модуль обработки событий срабатывания клавиш;
- `timer` - модуль отсчета времени;
- `sprites` — модуль для работы со спрайтами;
- `sound` — модуль работы со звуком;
- `nouse` — модуль обработки реакций на не заданные действия при использовании объекта;
- `counters` — модуль счетчиков событий;
- `wroom` — модуль создания «умных» переходов;
- `nolife` – модуль блокировки методов life;
- `proxymenu` – модуль меню в стиле адвенчур на ZX-80.

Пример загрузки модулей:

```
--$Name: Моя игра!$
instead_version "1.8.2"
require "para"
require "dbg"
...
```

Меню

Стандартное поведение предмета инвентаря состоит в том, что игрок должен сделать два щелчка мышью. Это необходимо потому, что каждый предмет инвентаря может быть использован на другой предмет сцены или инвентаря. После второго щелчка происходит игровой такт игры. Иногда такое поведение может быть нежелательным. Возможно, вы захотите сделать игру в которой игровая механика отличается от классических INSTEAD игр. Тогда вам может понадобится меню.

Меню – это элемент инвентаря, который срабатывает на первый клик. При этом меню может сообщить движку, что действие не является игровым тактом. Таким образом, используя меню вы можете создать в зоне инвентаря управление игрой любой сложности. Например, существует модуль «proxymenu», который реализует управление игрой в стиле квестов на ZX-«Спектрум». В игре «Особняк» свое управление, которое вводит несколько модификаторов действий, и т.д.

Итак, вы можете делать меню в области инвентаря, определяя объекты с типом `menu`. При этом, обработчик меню (`menu`) будет вызван после одного клика мыши. Если обработчик не возвращает текст, то состояние игры не изменяется. Например, реализация кармана:

```
pocket = menu {
    var {
                state = false
        };
        nam = 'карман';
    disp = function(s)
        if s.state then
            return txtu('карман'); -- подчеркиваем активный карман
        end
        return 'карман';
    end;
    gen = function(s)
        if s.state then
            s:enable_all(); -- показать все предметы в кармане
        else
            s:disable_all(); -- спрятать все предметы в кармане
        end
    end;
    menu = function(s)
                s.state = not s.state -- изменить состояние
        s:gen(); -- открыть или закрыть карман
    end;
};

knife = obj {
    nam = 'нож';
```

```
    inv = 'Это нож';
};

function init()
    take(pocket)
    put(knife, pocket) -- нож будет в кармане
    pocket:gen() -- проинициализируем карман
end

main = room {
    nam = 'test',
};
```

Статус игрока

Иногда возникает желание выводит какой-нибудь статус. Например, количество игровых очков, состояние героя или, наконец, время суток. INSTEAD не предоставляет каких-то других областей вывода, кроме сцены и инвентаря, поэтому, самым простым способом вывода статуса является вывод его в зону инвентаря.

Ниже представлена реализация статуса игрока в виде текста, который появляется в инвентаре, но не может быть выбран, то есть, выглядит просто как текст.

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

status = stat { -- stat -- объект "статус"
        nam = 'статус';
    disp = function(s)
        pn ('Жизнь: ', life)
                pn ('Сила: ', power)
    end
};
take(status)
```

walk из обработчиков exit и enter

Вы можете делать `walk` из обработчиков `enter` и `exit`. Например, `vroom` реализован как комната с обработчиком `enter`, который переносит игрока в другую комнату. Если бы не было `vroom`, можно было бы написать переход без него следующим образом:

```
home = room {
       nam = 'Дома';
...
        way = {
                room {
                        nam = 'На улицу';
                        enter = code [[ walk 'street' ]];
                };
        }
```

Динамически создаваемые ссылки.

Строго говоря, ссылок в INSTEAD нет. Есть объекты и переходы.

Динамически создаваемые ссылки могут быть реализованы разным способом.

1. Через включение/выключение объектов/переходов;
2. Через динамическое добавление/удаление объектов/переходов;

Следует заметить, что объекты типа `vway, vobj` и переходы `vroom, wroom` сделаны таким образом, что могут создаваться на лету, это делает возможным записи вида:

```
put(vway('Дорога', 'Я заметил {дорогу}, ведущую в лес...', 'forest'), field);
-- добавили ''vway'' в сцену field
```

Однако, я рекомендую использование `enable/disable()`.

Кодирование исходного кода игры

Если вы не хотите показывать исходный код своих игр, вы можете закодировать исходный код с помощью параметра командной строки `-encode`:

`sdl-instead -encode <путь к файлу> [выходной путь]`

И использовать закодированный файл с помощью lua функции `doencfile`. При этом главный файл `main.lua` необходимо оставлять открытым. Таким образом, схема выглядит следующим образом (`game` – закодированный `game.lua`):

```
-- $Name: Моя закрытая игра!$
instead_version "1.8.2"
doencfile("game"); -- никто не узнает, как ее пройти!
```

Осторожно!

Не используйте компиляцию игр с помощью luac, так как luac создает платформозависимый код! Однако, компиляция игр может быть использована для поиска ошибок в коде.

Запаковка ресурсов

Вы можете упаковать ресурсы игры (графику, музыку, темы) в файл ресурсов `.idf`, для этого поместите все ресурсы в каталог `data` и запустите INSTEAD:

```
sdl-instead -idf <путь к data>
```

При этом, в текущем каталоге должен будет создастся файл `data.idf`. Поместите его в каталог с игрой. Теперь ресурсы игры в виде отдельных файлов можно удалить (конечно, оставив себе оригинальные файлы).

Вы можете запаковать в формат `.idf` всю игру:

```
sdl-instead -idf <путь к игре>
```

Игры в формате `idf` можно запускать как обычные игры `instead` (как если бы это были каталоги) а также из командной строки:

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

Переключение между игроками

Вы можете создать игру с несколькими персонажами и время от времени переключаться между ними (см. `change_pl`). Но вы можете также использовать этот трюк для того, чтобы иметь возможность переключаться между разными типами инвентаря.

Использование параметров обработчика

Пример кода.

```
stone = obj {
    nam = 'камень';
    dsc = 'На краю лежит {камень}.';
    act = function()
```

```
        remove('stone');
        p 'Я толкнул камень, он сорвался и улетел вниз...';
    end
```

Обработчик act мог бы выглядеть проще:

```
act = function(s)
        remove(s);
        p 'Я толкнул камень, он сорвался и улетел вниз...';
    end
```

Таймер

Для асинхронных событий, привязанных к реальному времени, в INSTEAD есть возможность использовать таймер. На самом деле, вам следует хорошо подумать, стоит ли в приключенческой игре использовать таймер. Обычно, игроком это воспринимается не слишком благосклонно. С другой стороны, таймер вполне можно использовать для управления музыкой или в оформительских целях.

Для использования таймера, вам следует подключить модуль «timer».

```
require "timer"
```

Таймер программируется с помощью объекта `timer`.

- `timer:set(мс)` – задать интервал таймера в миллисекундах;
- `timer:stop()` – остановить таймер;

При срабатывании таймера, вызывается обработчик `game.timer`. Если game.timer возвращает пустое значение, сцена не перерисовывается. В противном случае, возвращаемое значение выводится как реакция.

Вы можете делать локальные для комнаты обработчики `timer`. Если в комнате объявлен обработчик `timer`, он вызывается вместо `game.timer`.

Например:

```
game.timer = function(s)
    set_sound('gfx/beep.ogg');
    p ("Timer:", time())
end

function init()
    timer:set(1000) -- раз в секунду
end
```

```
myroom = room  {
    entered = function(s)
        timer:set(1000);
    end;
    timer = function(s)
        timer:stop();
        walk 'myroom2';
    end;
    nam = 'Проверка таймера';
    dsc = [[Ждите.]];
}
```

Состояние таймера попадает в файл сохранения, таким образом, вам не нужно заботиться о его восстановлении.

Кроме того, в INSTEAD существует возможность отслеживать интервалы времени в миллисекундах. Для этого используйте функцию stead.ticks(). Функция возвращает число миллисекунд, прошедшее с момента старта игры.

Музыкальный плеер

Вы можете написать для игры свой проигрыватель музыки, создав его на основе живого объекта, например:

```
-- играет треки в случайном порядке, начиная со 2-го
mplayer = obj {
        tracks = {"mus/astro2.mod",
                  "mus/aws_chas.xm",
                  "mus/dmageofd.xm",
                  "mus/doomsday.s3m"};
    nam = 'плеер';
    life = function(s)
```

```
                if not is_music() then
                    local n = tracks[rnd(#s.tracks)]
            set_music(n, 1);
        end
    end;
};
lifeon('mplayer');
```

Ниже приводится пример более сложного плеера. Меняем трек только если он закончился или прошло более 2 минут и игрок перешел из комнаты в комнату. В каждом треке можно указать число проигрываний (0 - зацикленный трек):

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

music_player = obj {
    nam = 'player';
    var { 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};
    life = function(s)
        if is_music() 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
        set_music('mus/'..s.playlist[s.pos], s.playlist[s.pos + 1]);
    end;
}

game.timer = function(s)
    track_time = track_time + 1
    music_player:life();
```

```
end

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

Живые объекты

Если вашему герою нужен друг, одним из способов может стать метод `life` этого персонажа, который всегда переносит объект в локацию игрока:

```
horse = obj {
    nam = 'лошадь';
    dsc = 'Рядом со мной стоит {лошадь}.';
        act = [[Моя лошадка.]];
    life = function(s)
        if player_moved() then
            move(s, here(), where(s));
        end
    end;
};
function init()
    put (horse, main); -- только в этом случае where() будет работать
    lifeon(horse); -- сразу оживим лошадь
end
```

Клавиатура

Вы можете перехватывать события клавиатуры с помощью модуля «kbd».

Обычно, перехват клавиш имеет смысл использовать для организации текстового ввода.

За описанием, обращайтесь к документации модуля «kbd».

Мышь

Вы можете отслеживать в своей игре клики по картинке сцены, а также по фону. Для этого, воспользуйтесь модулем «click». Также, вы можете отслеживать состояние мыши с помощью функции:

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

Которая возвращает координаты курсора. Если задать параметры (x, y), то можно переместить курсор в указанную позицию (все координаты рассчитываются относительно левого верхнего угла окна INSTEAD).

Вызов меню

Вы можете вызвать из игры меню INSTEAD с помощью функции `stead.menu_toggle()`. Если в качестве параметра задать: 'save', 'load' или 'quit', то будет вызван соответствующий подраздел меню. (Поддерживается начиная с версии INSTEAD 1.8.3).

Динамическое создание объектов

Как вы уже знаете, объекты нельзя создавать на лету, если только у них не определена функция `save`, которая сможет сохранить состояние такого объекта.

Тем не менее существует способ создание любого объекта на лету. Для этого вам понадобится написать *конструктор* вашего объекта и воспользоваться функцией `new`.

Итак, вы можете использовать функции `new` и `delete` для создания и удаления динамических объектов. Примеры:

```
local o = new ([[obj { nam = 'test', act = 'test' }]]);
take(o);
...
put(new [[obj {nam = 'test' } ]]);
put(new('myconstructor()');
n = new('myconstructor()');
...
delete(n)
```

`new` воспринимает строку-аргумент, как конструктор объекта. Результатом выполнения конструктора должен быть объект. Таким образом в аргументе обычно задан вызов функции-конструктора. Например:

```
function myconstructor()
    local v = {}
    v.nam = 'тестовый объект'
```

```
    v.act = 'Тестовая реакция'
    return obj(v)
end
```

Созданный объект будет попадать в файл сохранения. `new()` возвращает сам объект.

Запрет на сохранение игры

Иногда может понадобиться запретить игроку делать сохранения в игре. Например, если речь идет о сценах, где важный элемент составляет случай, или для коротких игр, в которых проигрыш должен быть фатальным и требовать перезапуска игры.

Для управлением функции сохранения используется атрибут `game.enable_save`.

Например:

```
game.enable_save = false -- запретить сохранения
```

Если вы хотите запрещать сохранения не везде, а в некоторых сценах, оформите `game.enable_save` в виде функции:

```
-- запретить сохранения в комнатах, которые содержат атрибут nosave.
game.enable_save = function()
        if here().nosave then
                return false
        end
        return true
end
```

Следует отметить, что запрет на сохранения не означает запрета на автосохранение. Для управления автосохранением воспользуйтесь аналогичным атрибутом `game.enable_autosave`.

Вы можете явно сохранять игру с помощью вызова: `autosave([номер слота])`; Если номер слота не задан, то игра будет сохранена под слотом 'автосохранение'.

22. Темы для sdl-instead

Графический интерпретатор поддерживает механизм тем. *Тема* представляет из себя каталог, с файлом `theme.ini` внутри.

Тема, которая является минимально необходимой – это тема `default`. Эта тема всегда загружается первой. Все остальные темы наследуются от нее и могут частично или полностью заменять ее параметры. Выбор темы осуществляется пользователем через меню настроек, однако конкретная игра может содержать собственную тему и таким образом влиять на свой внешний вид. В этом случае в каталоге с игрой должен находиться свой файл `theme.ini`. Тем не менее, пользователь свободен отключить данный механизм, при этом интерпретатор будет предупреждать о нарушении творческого замысла автора игры.

Синтаксис `theme.ini` очень прост.

```
<параметр> = <значение>
```

или

```
; комментарий
```

Значения могут быть следующих типов: строка, цвет, число.

Цвет задается в форме #rgb, где r g и b компоненты цвета в шестнадцатеричном виде. Кроме того некоторые основные цвета распознаются по своим именам. Например: yellowgreen, или violet.

Параметры могут принимать значения:

- `scr.w` = ширина игрового пространства в пикселях (число)
- `scr.h` = высота игрового пространства в пикселях (число)
- `scr.col.bg` = цвет фона
- `scr.gfx.scalable` = [0|1|2] (0 - не масштабируемая тема, 1 - масштабируемая, 2 - кратно-масштабируемая)
- `scr.gfx.bg` = путь к картинке фонового изображения (строка)
- `scr.gfx.cursor.x` = x координата центра курсора (число)
- `scr.gfx.cursor.y` = y координата центра курсора (число)
- `scr.gfx.cursor.normal` = путь к картинке-курсору (строка)
- `scr.gfx.cursor.use` = путь к картинке-курсору режима использования (строка)
- `scr.gfx.use` = путь к картинке-индикатору режима использования (строка)
- `scr.gfx.pad` = размер отступов к скролл-барам и краям меню (число)

- `scr.gfx.x`, `scr.gfx.y`, `scr.gfx.w`, `scr.gfx.h` = координаты, ширина и высота окна изображений. Области в которой располагается картинка сцены. Интерпретация зависит от режима расположения (числа)
- `win.gfx.h` - синоним `scr.gfx.h` (для совместимости)
- `scr.gfx.mode` = режим расположения (строка `fixed`, `embedded` или `float`). Задает режим изображения. `embedded` – картинка является частью содержимого главного окна, параметры `scr.gfx.x`, `scr.gfx.y`, `scr.gfx.w` игнорируются. `float` – картинка расположена по указанным координатам (scr.gfx.x, scr.gfx.y) и масштабируется к размеру `scr.gfx.w` x `scr.gfx.h` если превышает его. `fixed` – картинка является частью сцены как в режиме embedded, но не скроллируется вместе с текстом а расположена непосредственно над ним. Доступны модификации режима float с модификаторами 'left/right/center/middle/bottom/top', указывающими как именно размещать картинку в области sct.gfx. Например: float-top-left;
- `win.scroll.mode` = [0|1|2|3] режим прокрутки области сцены. 0 - нет автоматической прокрутки, 1 - прокрутка на изменение в тексте, 2 прокрутка на изменение, только если изменение не видно, 3 - всегда в конец;
- `win.x`, `win.y`, `win.w`, `win.h` = координаты, ширина и высота главного окна. Области в которой располагается описание сцены (числа)
- `win.fnt.name` = путь к файлу-шрифту (строка). Здесь и далее, шрифт может содержать описание всех начертаний, например: {sans,sans-b,sans-i,sans-bi}.ttf (заданы начертания для regular, bold, italic и bold-italic). Вы можете опускать какие-то начертания, и движок сам сгенерирует их на основе обычного начертания, например: {sans,,sans-i}.ttf (заданы только regular и italic);
- `win.align` = center/left/right/justify (выравнивание текста в окне сцены);
- `win.fnt.size` = размер шрифта главного окна (размер)
- `win.fnt.height` = междустрочный интервал как число с плавающей запятой (1.0 по умолчанию)
- `win.gfx.up`, `win.gfx.down` = пути к файлам-изображениям скорллеров вверх/вниз для главного окна (строка)
- `win.up.x`, `win.up.y`, `win.down.x`, `win.down.y` = координаты скроллеров (координата или -1)
- `win.col.fg` = цвет текста главного окна (цвет)
- `win.col.link` = цвет ссылок главного окна (цвет)
- `win.col.alink` = цвет активных ссылок главного окна (цвет)
- `inv.x`, `inv.y`, `inv.w`, `inv.h` = координаты, высота и ширина области инвентаря. (числа)

- `inv.mode` = строка режима инвентаря (`horizontal` или `vertical`). В горизонтальном режиме инвентаря в одной строке могут быть несколько предметов. В вертикальном режиме, в каждой строке инвентаря содержится только один предмет. (число) Существует модификации (-left/right/center). Вы можете задать режим `disabled` если в вашей игре не нужен инвентарь;
- `inv.col.fg` = цвет текста инвентаря (цвет)
- `inv.col.link` = цвет ссылок инвентаря (цвет)
- `inv.col.alink` = цвет активных ссылок инвентаря (цвет)
- `inv.fnt.name` = путь к файлу-шрифту инвентаря (строка)
- `inv.fnt.size` = размер шрифта инвентаря (размер)
- `inv.fnt.height` = междустрочный интервал как число с плавающей запятой (1.0 по умолчанию)
- `inv.gfx.up`, `inv.gfx.down` = пути к файлам-изображениям скорллеров вверх/вниз для инвентаря (строка)
- `inv.up.x`, `inv.up.y`, `inv.down.x`, `inv.down.y` = координаты скроллеров (координата или -1)
- `menu.col.bg` = фон меню (цвет)
- `menu.col.fg` = цвет текста меню (цвет)
- `menu.col.link` = цвет ссылок меню (цвет)
- `menu.col.alink` = цвет активных ссылок меню (цвет)
- `menu.col.alpha` = прозрачность меню 0-255 (число)
- `menu.col.border` = цвет бордюра меню (цвет)
- `menu.bw` = толщина бордюра меню (число)
- `menu.fnt.name` = путь к файлу-шрифту меню (строка)
- `menu.fnt.size` = размер шрифта меню (размер)
- `menu.fnt.height` = междустрочный интервал как число с плавающей запятой (1.0 по умолчанию)
- `menu.gfx.button` = путь к файлу изображению значка меню (строка)
- `menu.button.x`, `menu.button.y` = координаты кнопки меню (числа)
- `snd.click` = путь к звуковому файлу щелчка (строка)
- `include` = имя темы (последний компонент в пути каталога) (строка)

Кроме того, заголовок темы может включать в себя комментарии с тегами. На данный момент существует только один тег: $Name:, содержащий UTF-8 строку с именем темы. Например:

```
; $Name:Новая тема$
; модификация темы book
include = book -- использовать тему ''Книга''
scr.gfx.h = 500 -- заменить в ней один параметр
```

Интерпретатор выполняет поиск тем в каталоге `themes`. Unix версия кроме этого каталога, просматривает также каталог `~/.instead/themes/` Windows версия – `Documents and Settings/USER/Local Settings/Application Data/instead/themes`