STEAD3

Петр Косых

5 августа 2018 г.

Оглавление

Глава 1

Общие сведения

Код игр под INSTEAD пишется на языке Lua (5.1), 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!

В феврале 2017 года, после 8 лет разработки, INSTEAD (версия 3.0) вышел с поддержкой нового ядра STEAD3. Старое ядро получило название STEAD2. INSTEAD поддерживает работу игр написанных как на STEAD2, так и на STEAD3. Это руководство описывает STEAD3.

Если у вас возникают вопросы, вы можете посетить сайт INSTEAD:

https://instead-hub.github.io

Или подключиться к чату на gitter:

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

1.1 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 или комнаты) 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: https://instead-hub.github.io/article/2010–05–09-history/

1.2 Как выглядит классическая INSTEAD игра

Итак, как выглядит классическая INSTEAD игра?

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

Описательная часть сцены отображается только один раз, при показе сцены, или при явном осмотре сцены (в графическом интерпретаторе – *Статическая часть сцены* содержит информацию о статических объектах сцены (обычно, это декорации) и отображается всегда. Эта часть написана автором игры.

Динамическая часть сцены составлена из описаний объектов сцены, которые присутствуют в данный момент на сцене. Эта часть генерируется автоматически и отображается всегда. Обычно, в ней представлены объекты, которые могут менять свое местоположение.

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 инвентаря;
- действие объектом инвентаря на объект сцены;
- переход в другую сцену.

1.3 Как создавать игру

Игра представляет из себя каталог, в котором должен находиться скрипт (текстовый файл) main3.lua. (Обратите внимание, наличие файла main3.lua означает, что вы пишите игру на STEAD3!) 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 main3.lua может быть определен заголовок, состоящий из тегов (строк специального вида). Теги должны начинаться с символов: два минуса.

```
--
```

Два минуса это комментарий с точки зрения Lua. На данный момент существуют следующие теги.

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:

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

Дополнительную информацию об игре, можно задать тегом Info:

```
-- $Info: Это ремейк старой игры\nC ZX specturm$
```

Обратите внимание на \n в Info, это станет переводом строки, когда вы выберете пункт “Информация” в INSTEAD.

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!

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

```
require ”fmt” -- некоторые функции форматирования
fmt.para = true -- включить режим параграфов (отступы)
```

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 ’нож’
    take ’бумага’
end
```

После того как игра проинициализирована, выполняется запуск игры. Вы можете определить функцию start(), которая запускается непосредственно перед запуском игры. Это значит, например, что в случае загрузки сохраненной игры, start() вызовется после того, как сохранение будет прочитано,

```
function start(load) -- восстановить состояние?
    if load then
        dprint ”Это загрузка состояния.”
    else
        dprint ”Это старт игры.”
    end
    -- нам сейчас не нужно ничего делать
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:

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

require ”fmt”
fmt.para = true

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

function init()
```

```
-- инициализация, если она нужна
end
```

1.4 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.

Кроме того, в режиме -debug автоматически подключается отладчик. Вы можете активировать его с помощью клавиш ctrl-d или f7. Вы можете подключить отладчик и явно указав:

```
require "dbg"
```

В коде вашей игры.

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.

> Внимание! Если вы просто перезапустите INSTEAD, то скорее всего увидите старое состояние игры, так как по умолчанию работает режим автозагрузки автосохранения! Либо отключите эту настройку в меню (автосохранение), либо явно перезапускайте игру после правок. Перезапуск возможен через меню (начать заново) или 'alt-r' в режиме '-debug' как это описано выше.

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! ”);
    ...
end;
```

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

Вы также можете использовать функцию dprint(), которая посылает вывод в окно отладчика, и вы сможете посмотреть его при входе в режим отладки.

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

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 main3.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.

Вы можете проверить синтаксис и с помощью INSTEAD, для этого воспользуйтесь параметром -luac:

```
sdl-instead -debug -luac <пусть к скрипту.lua>
```

Глава 2

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 участок леса, доступный для наблюдения.

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

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

Запись означает создание объекта (так как почти все сущности в INSTEAD это объекты) main типа room (комната). Атрибут объекта nam хранит имя комнаты 'main', по которому можно обращаться к комнате из кода. Каждый объект имеет свое уникальное имя. Если вы попробуете создать два объекта с одинаковыми именами, вы получите сообщение об ошибке.

Для обращения к объекту по имени, вы можете использовать следующую запись:

```
dprint("Объект: ", _'main')
```

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. Однако, атрибут nam всегда должен быть текстовой строкой, если он задан.

На самом деле, вы можете не указывать имя при создании объекта:

```
room {
    disp = ”Главная комната”;
    dsc = [[Вы в большой комнате.]];
}
```

В таком случае, движок сам даст имя объекту, и это имя будет неким числом. Так как вы не знаете это число, вы не можете обратиться к объекту явно. Иногда удобно создавать безымянные объекты, например, для декораций. При создании объекта, даже если он “безымянный”, вы можете создать переменную - ссылку на объект, например:

```
myroom = room {
    disp = ”Чулан”;
    dsc = [[Вы в чулане.]];
}
```

Переменная myroom в таком случае становится синонимом объекта (ссылкой на сам объект).

```
dprint(”Объект: ”, myroom)
```

Вы можете придерживаться какого-то одного способа, или применять оба. Например, вы можете задать и имя и переменную-ссылку:

```
main_room = room {
    nam = ’main’;
    disp = ”Главная комната”;
    dsc = [[Вы в большой комнате.]];
}
```

Важно понять, что движок в любом случае работает с именами объектов, а переменные-ссылки – это просто способ упростить доступ к часто используемым объектам. Поэтому, для нашей первой игры мы обязаны указать атрибут

nam = ‘main’, чтобы создать комнату main с которой и начнется наше приключение!

В нашем примере, при показе сцены, в качестве заголовка сцены будет использован атрибут ‘disp’. На самом деле, если бы мы его не задали, то в заголовке мы бы увидели ‘nam’. Но nam не всегда удобно делать заголовком сцены, особенно если это строка вроде ‘main’, или если это числовой идентификатор, который движок присвоил объекту автоматически.

Есть еще более понятный атрибут ‘title’. Если он задан, то при отображении комнаты в качестве заголовка будет указан именно он. title используется тогда, когда игрок находится *внутри* комнаты. Во всех остальных случаях (при показе переходов в эту комнату) используется ‘disp’ или ‘nam’.

```
mroom = room {
    nam = ’main’;
    title = ’Начало приключения’;
    disp = ”Главная комната”;
    dsc = [[Вы в большой комнате.]];
}
```

Атрибут ‘dsc’ – это описание сцены, которое выводится один раз при входе в сцену или при явном осмотре сцены. В нем нет описаний объектов, присутствующих в сцене.

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

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

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

```
room {
    nam = ’main’;
    disp = ’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'.

Напомню еще раз, что имя 'nam' объекта и его отображение (в данном случае то, как сцена будет выглядеть для игрока в виде надписи сверху) можно (и, часто, нужно!) разделять. Для этого существуют атрибуты 'disp' и 'title'. 'title' бывает только у комнат и работает как описатель, когда игрок находится внутри данной комнаты. В остальных случаях используется 'disp' (если он есть).

Если 'disp' и 'title' не заданы, то считается, что отображение равняется имени.

'disp' и 'title' могут принимать значение false, в таком случае, отображения не будет.

Глава 3

Объекты

Объекты – это единицы сцены, с которыми взаимодействует игрок.

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

Имя объекта “nam” используется при попадании его в инвентарь. Хотя, в нашем случае, стол вряд ли туда попадет. Если у объекта определен ‘disp’, то при попадании в инвентарь для его отображения будет использоваться именно этот атрибут. Например:

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

Все-таки стол попал к нам в инвентарь.

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

‘dsc’ – описание объекта. Оно будет выведено в динамической части сцены, при наличии объекта в сцене. 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 основная задача – вывод (возвращение) строки текста, которая станет частью событий сцены, и изменение состояния игрового мира.

Глава 4

Добавляем объекты в сцену

Для того, чтобы поместить в сцену объекты, существует несколько путей.

Во-первых, при создании комнаты можно определить список ‘obj’, состоящий из имен объектов:

```
obj { -- объект с именем, но без переменной
    nam = ’ящик’;
    dsc = [[На полу я вижу {ящик}.]];
    act = [[Тяжелый!]];
}

room {
    nam = ’main’;
    disp = ’Большая комната’;
    dsc = [[Вы в большой комнате.]];
    obj = { ’ящик’ };
};
```

Теперь, при отображении сцены мы увидим объект “ящик” в динамической части.

Вместо имени объекта, вы можете использовать переменную-ссылку, если только она была определена заранее:

```
apple = obj { -- объект с переменной, но без имени
    dsc = [[Тут есть {яблоко}.]];
    act = [[Красное!!]];
}
```

```
room {
    nam = 'main';
    disp = 'Большая комната';
    dsc = [[Вы в большой комнате.]];
    obj = { apple };
};
```

Альтернативной формой записи является конструкция with:

```
room {
    nam = 'main';
    disp = 'Большая комната';
    dsc = [[Вы в большой комнате.]];
}:with {
    'ящик',
}
```

Конструкция with позволяет избавиться от лишнего уровня вложенности в коде игры.

Во-вторых, вы можете объявлять объекты прямо внутри obj или with, описывая их определение:

```
room {
    nam = 'main';
    disp = 'Большая комната';
    dsc = [[Вы в большой комнате.]];
}:with {
    obj {
        nam = 'ящик';
        dsc = [[На полу я вижу {ящик}.]];
        act = [[Тяжелый!]];
    }
};
```

Это удобно делать для объектов - декораций. Но в таком случае, вы не сможете создавать объекты с переменной-ссылкой. К счастью, для декораций это и не нужно.

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

```
obj = { ’ящик’, apple };
```

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

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

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

Глава 5

Декорации

Объекты, которые могут быть перенесены из одной сцены в другую (или попадать в инвентарь), обычно имеют имя и/или переменную-ссылку. Так как таким образом вы всегда можете найти объект где угодно и работать с ним.

Но немалую часть игрового мира составляют объекты, которые занимают конкретную локацию и служат в качестве декораций.

Таких объектов может быть очень много, и более того, обычно это однотипные объекты вроде деревьев и тому подобных объектов.

Для создания декораций можно использовать различные подходы.

5.1 Один и тот же объект в нескольких комнатах

Вы можете создать один объект, например, ‘дерево’ и помещать их в разные комнаты.

```
obj {
    nam = ’дерево’;
    dsc = [[Тут стоит {дерево}.]];
    act = [[Все деревья выглядят похожими.]];
}

room {
    nam = ’Лес’;
    obj = { ’дерево’ };
}
```

```
room {
    nam = 'Улица';
    obj = { 'дерево' };
}
```

5.2 Использование тегов вместо имен

Если вам не нравится придумывать уникальные имена для однотипных декоративных объектов, вы можете использовать для таких объектов теги. Теги задаются атрибутом tag и всегда начинаются с символа `#':

```
obj {
    tag = '#цветы';
    dsc = [[Тут есть {цветы}.]]
}
```

В данном примере, имя у объекта будет сформировано автоматически, но обращаться к объекту вы сможете по тегу. При этом объект будет искаться в текущей комнате. Например:

```
dprint(_'#цветы') -- ищем в текущей комнате первый объект с тегом '#цветы'
```

Теги, это в каком то смысле, синоним локальных имен, поэтому существует альтернативная запись создания предмета с тегом:

```
obj {
    nam = '#цветы';
    dsc = [[Тут есть {цветы}.]]
}
```

Если имя у объекта начинается с символа `#', то такой объект получает тег и автоматически сгенерированное числовое имя.

5.3 Использование атрибута сцены decor

Так как декорации не меняют свое место-положение, есть смысл сделать их частью описания сцены, а не динамической области. Это делается с помощью атрибута сцены ‘decor’. decor показывается всегда и его основная функция – описание декораций сцены.

```
room {
    nam = ’Дом’;
    dsc = [[Я у себя дома.]];
    decor = [[Тут я вижу много интересных вещей. Например, на {#стена|стене}
    висит {#картина|картина}.]];
}: with {
    obj {
        nam = ’#стена’;
        act = [[Стена как стена!]];
    };
    obj {
        nam = ’#картина’;
        act = [[Ван-Гог?]];
    }
}
```

Здесь мы видим сразу несколько приемов:

1. В decor в виде связанного текста описаны декорации;
2. В качестве ссылок используются конструкции с явным заданием объектов, к которым они относятся {имя объекта|текст};
3. В качестве имен объектов используются теги, чтобы не думать над их уникальностью;
4. У объектов-декораций в сцене отсутствуют атрибуты dsc, так как их роль играет decor.

Конечно, вы можете комбинировать все описанные приемы между собой в любых пропорциях.

Глава 6

Объекты, связанные с другими объектами

Объекты тоже могут содержать в себе атрибут ‘obj’ (или конструкцию ‘with’). 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.

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

obj {
    nam = ’стол’;
    dsc = [[В комнате стоит {стол}.]];
    act = ’Гм... Просто стол...’;
    obj = { ’яблоко’ };
};

room {
    nam = ’Дом’;
    obj = { ’стол’ };
}
```

При этом, в описании сцены мы увидим описание объектов ‘стол’ и ‘яблоко’, так как ‘яблоко’ – связанный со столом объект и движок при выводе объекта ‘стол’ 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’.

Конечно, данный пример мог бы быть написан и по другому, например, так:

```
room {
    nam = ’Дом’;
}:with {
    obj {
        nam = ’стол’;
        dsc = [[В комнате стоит {стол}.]];
        act = ’Гм... Просто стол...’;
    }: with {
        obj {
            nam = ’яблоко’;
            dsc = [[На столе лежит {яблоко}.]];
            act = ’Взять что-ли?’;
        };
    }
}
```

Выбирайте тот способ, который для вас понятней.

Глава 7

Атрибуты и обработчики как функции

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

```
disp = function()
    p ’яблоко’;
end
```

Пример не очень удачен, так как проще было бы написать disp = ‘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”);
```

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, ‘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!

Важно!

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”.

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

```
obj {
    nam = ’яблоко’;
    seen = false;
    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 _‘яблоко’. Когда вы работаете с самим объектом в функции, удобнее использовать параметр, а не явное обращение к объекту по имени, так как при переименовании объекта вам не придется переписывать вашу игру. Да и запись будет короче.

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 ‘яблоко’ 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 ’яблоко’ then
    p ’У меня есть яблоко!’
end
```

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

if have ’яблоко’ 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>

if have ’яблоко’ then
    p ’У меня есть яблоко!’
elseif have ’вилка’ then
    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. Равенство описывается как ‘==’, неравенство ‘∼=’.

```
if not have ’яблоко’ and not have ’вилка’ 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, INSTEAD даст ошибку. Вам придется заранее определять переменные, которые вы используете.

7.1 Переменные объекта

Запись ‘s.seen’ означает, что переменная ‘seen’ размещена в объекте ‘s’ (то есть ‘яблоко’). Помните, мы назвали первый параметр функции ‘s’ (от self), а первый параметр – это сам текущий объект.

Переменные объекта должны быть определены заранее, если вы собираетесь их модифицировать. Примерно так, как мы поступили с seen. Но переменных может быть много.

```
obj {
    nam = ’яблоко’;
    seen = false;
    eaten = false;
    color = ’красный’;
    weight = 10;
    ...
};
```

Все переменные объекта, при их изменении, попадают в файл сохранения игры.

Если вы не хотите, чтобы переменная попала в файл сохранения, вы можете объявить такие переменные в специальном блоке:

```
obj {
    nam = ’яблоко’;
    {
       t = 1; -- эта переменная не попадет в сохранения
       x = false; -- и эта тоже
    }
};
```

Обычно, вам не стоит так делать. Однако есть ситуация, при которой этот прием будет полезным. Дело в том, что массивы и таблицы объекта всегда сохраняются. Если вы используете массивы для хранения неизменяемых значений, вы можете написать так:

```
obj {
    nam = ’яблоко’;
    {
       text = { ”раз”, ”два”, ”три” }; -- никогда не попадет в файл сохранения
    }
    ...
};
```

Вы можете обращаться к переменным объекта через s – если это сам объект. или по переменной - ссылке, например:

```
apple = obj {
    color = ’красный’;
}
...
-- где-то в другом месте
    apple.color = ’зеленый’
```

Или по имени:

```
obj {
    nam = ’яблоко’;
    color = ’красный’;
}
...
-- где-то в другом месте
    _’яблоко’.color = ’зеленый’
```

На самом деле, вы можете создавать переменные-объекта на лету (без предварительного их определения), хотя обычно в этом нет смысла. Например:

```
apple ’xxx’ (10) -- создали переменную xxx у объекта apple по ссылке
(_’яблоко’) ’xxx’ (10) -- то же самое, но по имени объекта
```

7.2 Локальные переменные

Кроме переменных объекта вы можете использовать локальные и глобальные переменные.

Локальные переменные создаются с помощью служебного слова local:

```
act = function(s)
    local w = _'лампочка'
    w.light = true
    p [[Я нажал на кнопку и лампочка загорелась.]]
end
```

В данном примере, переменная w существует только в теле функции-обработчика act. Мы создали временную ссылку-переменную w, которая ссылается на объект 'лампочка', чтобы изменить свойство-переменную light у этого объекта.

Конечно, мы могли написать и:

```
_'лампочка'.light = true
```

Но представьте себе, если нам нужно произвести несколько действий с объектом, в таких случаях проще воспользоваться временной переменной.

Локальные переменные никогда не попадают в файл-сохранение и играют роль временных вспомогательных переменных.

Локальные переменные можно создавать и вне функций, тогда данная переменная видима только в пределах данного lua файла и не попадает в файл сохранения.

Еще один пример использования локальных переменных:

```
obj {
    nam = 'котенок';
    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. 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
end
```

7.3 Глобальные переменные

Вы также можете создать глобальную переменную:

```
global { -- определение глобальных переменных
    global_var = 1; -- число
    some_number = 1.2; -- число
    some_string = ’строка’;
    know_truth = false; -- булево значение
    array = {1, 2, 3, 4}; -- массив
}
```

Еще одна форма записи, удобная для одиночных определений:

```
global 'global_var' (1)
```

Глобальные переменные всегда попадают в файл-сохранение.

Кроме глобальных переменных вы можете задавать константы. Синтаксис аналогичен глобальным переменным:

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

Движок будет контролировать неизменность констант. Константы не попадают в файл-сохранение.

Иногда вам нужно работать с переменной, которая не определена как local (и видима во всех ваших lua файлах игры), но не должна попадать в файл сохранения. Для таких переменных вы можете использовать декларации:

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

Декларации не попадают в файл сохранения. Одно из важных свойств деклараций состоит в том, что вы можете декларировать функции, например:

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

global 'f' (test)
```

В таком случае, вы можете присваивать значение функции 'test' другим переменным и состояние этих переменных может быть сохранено в файле сохранения. То-есть, декларированную функцию можно использовать как значение переменной!

Вы можете декларировать ранее определенные функции, например:

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

Тем самым делая такие недекларированные функции – декларированными.

Декларация функции, по сути, это присвоение функции имени, благодаря чему мы можем сохранить эту функцию как ссылку.

7.4 Вспомогательные функции

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.

7.5 Возвращаемые значения обработчиков

Если необходимо показать, что действие не выполнено (обработчик не сделал ничего полезного), возвращайте значение false. Например:

```
act = function(s)
    if broken_leg then
        return false
    end
    p [[Я ударил ногой по мячу.]]
end
```

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...’;
```

Итак, если вы не задали обработчик act или вернули из него false – считается, что реакции нет и движок выполнит аналогичный обработчик у объекта ‘game’.

Обычно, нет никакого смысла возвращать false из act, но существуют другие обработчики, о которых будет рассказано дальше, для которых описанное поведение точно такое же.

На самом деле, кроме ‘game.act’ и ‘act’ – атрибута объекта существует обработчик ‘onact’ у объекта game, который может прервать выполнение обработчика ‘act’.

Перед тем как вызвать обработчик ‘act’ у объекта, вызывается onact у game. Если обработчик вернет false, выполнение ‘act’ обрывается. ‘onact’ удобно использовать, для контроля событий в комнате или игре, например:

```
-- вызываем onact комнат, если они есть
-- для действий на любой объект

game.onact = function(s, ...)
    local r, v = std.call(here(), ’onact’, ...)
    if v == false then -- если false, обрубаем цепочку
        return r, v
    end
    return
end

room {
    nam = ’shop’;
    disp = ’Магазин’;
    onact = function(s, w)
        p [[В магазине нельзя воровать!]]
        p ([[Даже, если это всего-лишь ]], w, ’.’)
        return false
    end;
```

```
    obj = { 'мороженное', 'хлеб' };
}
```

В данном примере, при попытке "потрогать" любой предмет, будет выведено сообщение о запрете данного действия.

Все, что описано выше на примере 'act' действует и для других обработчиков: tak, inv, use, а также при переходах, о чем будет рассказано далее.

> Иногда возникает необходимость вызвать функцию - обработчик вручную. Для этого используется синтаксис вызова метода объекта. 'Объект:метод(параметры)'. Например:

```
apple:act() -- вызовем обработчик 'act' у объекта 'apple' (если он
определен как функция!).  _'яблоко':act() -- то же самое, но по
имени, а не по переменной-ссылке
```

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

Глава 8

Инвентарь

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

Например:

```
obj {
    nam = ’яблоко’;
    dsc = ’На столе лежит {яблоко}.’;
    inv = function(s)
        p ’Я съел яблоко.’
        remove(s); -- удалить яблоко из инвентаря
    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:

```
obj {
    nam = ’яблоко’;
    dsc = ’На столе лежит {яблоко}.’;
```

```
    inv = function(s)
        p 'Я съел яблоко.'
        remove(s); -- удалить яблоко из инвентаря
    end;
    act = function(s)
        take(s)
        p 'Вы взяли яблоко.';
    end
};
```

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

Если обработчик 'tak' вернет false, то предмет не будет взят, например:

```
obj {
    nam = 'яблоко';
    dsc = 'На столе лежит {яблоко}.';
    tak = function(s)
        p "Оно же червивое!"
        return false
    end;
};
```

Глава 9

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 имен комнат или переменных-ссылок, аналогично списку ‘obj’. Например:

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

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

При этом, вы сможете переходить между сценами ‘main’ и ‘room2’. Как вы помните, ‘disp’ может быть функцией, и вы можете генерировать имена переходов на лету. Или использовать title, для разделения имени сцены как заголовка и как имени перехода:

```
room {
    nam = 'room2';
```

```
    disp = 'В зал';
    title = 'В зале';
    dsc = 'Вы в огромном зале.';
    way = { 'main' };
};

room {
    nam = 'main';
    title = 'В главной комнате';
    disp = 'В главную комнату';
    dsc = 'Вы в большой комнате.';
    way = { 'room2' };
};
```

При переходе между сценами движок вызывает обработчик 'onexit' из текущей сцены и 'onenter' в той сцене, куда идет игрок. Например:

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

Конечно, как и все обработчики, 'onexit' и 'onenter' могут быть функциями. Тогда первый параметр это (как всегда) сам объект - комната, а второй – это комната куда игрок собирается идти (для 'onexit') или из которой собирается уйти (для 'onenter'). Например:

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

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

Запись вида:

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

Это сопоставление объекта с именем. Это альтернатива записям:

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

Или:

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

Или:

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

Как видим на примере onexit, эти обработчики, кроме строки могут возвращать булевое значение статуса. Аналогично обработчику onact, мы может отменить переход, вернув false из onexit/onenter.

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.

Важно!

> Следует отметить, что при вызове обработчика ‘onenter’ указатель на текущую сцену (here()) **еще не изменен**!!! В INSTEAD есть обработчики ‘exit’ (уход из комнаты) и ‘enter’ (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 ‘path’.

```
room {
    nam = ’room2’;
    title = ’Зал’;
    dsc = ’Вы в огромном зале.’;
    way = { path { ’В главную комнату’, ’main’} };
};

room {
    nam = ’main’;
    title = ’Главная комната’;
    dsc = ’Вы в большой комнате.’;
    way = { path {’В зал’, ’room2’} };
};
```

На самом деле, ‘path’ создает комнату с атрибутом ‘disp’, который равен первому параметру, и специальной функцией ‘onenter’, которая перенаправляет игрока в комнату заданную вторым параметром ‘path’.

Если вы укажете три параметра:

```
way = { path {’#взал’, ’В зал’, ’room2’} };
```

То первый параметр станет именем (или тегом, как в приведенном примере) такой комнаты.

Альтернативная форма записи с явным заданием атрибута nam:

```
way = { path { nam = ’#взал’, ’В зал’, ’room2’} };
```

Вы можете менять название перехода, после того, как переход происходил хотя бы раз, и вы узнали, что же это за комната:

```
way = { path {’#вдверь’, ’В дверь’, after = ’В гостиную’, ’room2’} };
```

Все параметры, кроме имени перехода, могут быть функциями.

Таким образом, ‘path’ позволяет именовать переходы удобным способом.

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 особого смысла прятать переход “дверь”. Просто в функции ‘onenter’ 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.

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

room {
    nam = ’Зал’;
    dsc = ’Вы в огромном зале.’;
    obj = { ’часы’ };
    way = { path { ’#часы’, ’В часы’, ’inclock’ }:disable() };
};
```

В данном примере, мы создали *отключенный* переход, за счет вызова метода ‘disable’ у комнаты созданной с помощью ‘path’. Метод ‘disable’ есть у всех

объектов (не только комнат), он переводит объект в отключенное состояние, которое означает, что объект перестает быть доступным игроку. Замечательным свойством отключенного объекта является то, что его можно *включить* с помощью ‘enable()’;

Далее, когда игрок нажимает на ссылку, описывающую часы, вызывается обработчик ‘act’, который с помощью функции ‘enable()’ делает переход видимым.

Альтернативный вариант заключается не в выключении, а ‘закрытии’ объекта:

```
obj {
    nam = ’часы’;
    dsc = [[Тут есть старинные {часы}.]];
    act = function(s)
        open ’#часы’
        p [[Вы видите, что в часах есть потайной ход!]];
    end;
}

room {
    nam = ’Зал’;
    dsc = ’Вы в огромном зале.’;
    obj = { ’часы’ };
    way = { path { ’#часы’, ’В часы’, ’inclock’ }:close() };
};
```

В чем разница? Выключение объекта означает то, что объект перестает быть доступным для игрока. Если в объекте вложены другие объекты, то и эти объекты становятся недоступными.

Закрытие объекта делает недоступным содержимое данного объекта, но не сам объект.

Однако, в случае комнат, и закрытие комнаты и отключенные комнаты приводят к одному результату – переход на них становится недоступным.

Еще один вариант:

```
room {
    nam = ’inclock’;
    dsc = [[Я в часах.]];
```

```
}:close()

obj {
    nam = ’часы’;
    dsc = [[Тут есть старинные {часы}.]];
    act = function(s)
        open ’inclock’
        p [[Вы видите, что в часах есть потайной ход!]];
    end;
}

room {
    nam = ’Зал’;
    dsc = ’Вы в огромном зале.’;
    obj = { ’часы’ };
    way = { path { ’В часы’, ’inclock’ } };
};
```

Здесь мы закрываем и открываем не переход, а комнату, в которую ведет переход. path не показывает себя, если комната в которую он ведет отключена или закрыта.

Глава 10

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 ‘used’ у объекта, на который действуют, и обработчик ‘use’ объекта, которым действуют.

Например: “‘ obj { nam = ‘нож’; dsc = ‘На столе лежит {нож}’; inv = ‘Острый!’; tak = ‘Я взял нож!’; use = ‘Вы пытаетесь использовать нож.’; };

obj { nam = ‘стол’; dsc = ‘В комнате стоит {стол}.’; act = ‘Гм… Просто стол…’; obj = { ‘нож’ }; used = function(s) p ‘Вы пытаетесь сделать что-то со столом…’; return false end; }; “‘ В данном примере, обработчик used возвращает false. Зачем? Если вы помните, возврат false означает, что обработчик сообщает движку о том, что событие он не обработал. Если бы мы не вернули бы false, очередь до обработчика ‘use’ объекта ‘нож’ просто бы не дошла. На самом деле, в реальности обычно вы будете пользоваться или use или used, вряд ли имеет смысл выполнять оба обработчика во время действия предмета на предмет.

Еще один пример, когда удобно вернуть false:

```
use = function(s, w)
    if w^’яблоко’ then
        p [[Я почистил яблоко.]]
        w.cut = true
        return
    end
    return false;
end
```

В данном случае use у яблока обрабатывает только одну ситуацию – действие на яблоко. В остальных случаях, обработчик возвращает false и движок вызовет метод по-умолчанию: game.use.

Но лучше, если вы пропишете действие по-умолчанию для ножа:

```
use = function(s, w)
    if w^’яблоко’ then
        p [[Я почистил яблоко.]]
        w.cut = true
        return
    end
    p [[Не стоит размахивать ножом!]]
end
```

Этот пример также демонстрирует тот факт, что вторым параметром у use является предмет на который мы действуем. У метода ‘used’, соответственно, второй параметр – это объект, который действует на нас:

```
obj {
    nam = ’мусорка’;
    dsc = [[В углу стоит {мусорка}.]];
    used = function(s, w)
        if w^’яблоко’ then
            p [[Я выбросил яблоко в мусорку.]]
            remove(w)
            return
        end
        return false;
    end
}
```

Как вы помните, перед вызовом use вызывается обработчик onuse у объекта game, потом у объекта ‘игрок’, а потом у текущей комнаты. Вы можете блокировать ‘use’, вернув из любого из перечисленных методов ‘onuse’ – false.

Использовать ‘use’ или ‘used’ (или оба) это вопрос личных предпочтений, однако, метод used вызывается раньше и, следовательно, имеет больший приоритет.

Глава 11

Объект “Игрок”

Игрок в мире INSTEAD представлен объектом типа ‘player’. Вы можете создавать несколько игроков, но один игрок присутствует по-умолчанию.

Имя этого объекта – ‘player’. Существует переменная-ссылка pl, которая указывает на этот объект.

Обычно, вам не нужно работать с этим объектом напрямую. Но иногда это может быть необходимым.

По умолчанию, атрибут ‘obj’ у игрока представляет собой инвентарь. Обычно, нет смысла переопределять объект типа player, однако, вы можете это сделать:

```
game.player = player {
    nam = ”Василий”;
    room = ’кухня’; -- стартовая комната игрока
    power = 100;
    obj = { ’яблоко’ }; -- 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’ (или его имя). Функция переключит текущего игрока, и при необходимости, осуществит переход в комнату, где находится новый игрок.

Функция ‘me()’ всегда возвращает текущего игрока. Следовательно, в большинстве игр me() == pl.

Глава 12

Объект “Мир”

Игровой мир представлен объектом типа world. Имя такого объекта ‘game’. Существует ссылка-переменная, которая также называется game.

Обычно вы не работаете с этим объектом напрямую, однако иногда вы можете вызывать его методы, или менять значения переменных этого объекта.

Например, переменная game.codepage содержит кодировку исходного кода игры, и по-умолчанию равна “UTF–8”. Я не рекомендую использовать другие кодировки, но иногда, выбор кодировки может стать необходимостью.

Переменная game.player – содержит текущего игрока.

Кроме того, как вы уже знаете, объект ‘game’ может содержать обработчики по умолчанию: ‘act’, ‘inv’, ‘use’, ‘tak’, которые будут вызваны, если в результате действий пользователя не будут найдены никакие другие обработчики (или все они вернули false). Например, вы можете написать в начале игры:

```
game.act = ’Не получается.’;
game.inv = ’Гм.. Странная штука..’;
game.use = ’Не сработает...’;
game.tak = ’Не нужно мне это...’;
```

Конечно, все они могут быть функциями.

Также, объект game может содержать обработчики: onact, ontak, onuse, oninv, onwalk – которые могут прерывать действия, в случае возврата false.

Еще, у объекта game можно задать обработчики: afteract, afterinv, afteruse, afterwalk – которые вызываются в случае успешного выполнения соответствующего действия.

Глава 13

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, и размещать их в объектах, например:

```
room {
    nam = ’холодильник’;
    frost = std.list { ’мороженное’ };
}
```

Хотя, обычно, это не требуется. Ниже перечислены методы объектов типа ‘список’. Вы можете вызывать их для любых списков, хотя обычно это будут way и obj, например:

```
ways():disable() -- отключить все переходы
```

- disable() - отключает все объекты списка;
- enable() - включает все объекты списка;
- close() - закрыть все объекты списка;
- open() - открыть все объекты списка;
- add(объект|имя, [позиция]) - добавить объект;
- for_each(функция, аргументы) - вызвать для каждого объекта функцию с аргументами;

- lookup(имя/тег или объект) - поиск объекта в списке. Возвращает объект и индекс;
- srch(имя/тег или объект) - поиск видимого объекта в списке;
- empty() - вернет true, если список пуст;
- zap() - очистить список;
- replace(что, на что) - заменить объект в списке;
- cat(список, [позиция]) - добавить содержимое списка в текущий список по позиции;
- del(имя/объект) - удалить объект из списка.

Существуют функции, возвращающие объекты-списки:

- inv([игрок]) - вернуть инвентарь игрока;
- objs([комната]) - вернуть объекты комнаты;
- ways([комната]) - вернуть переходы комнаты.

Конечно, вы можете обращаться к спискам и напрямую:

```
pl.obj:add 'нож'
```

Объекты в списках хранятся в том порядке, в котором вы их добавите. Однако, если у объекта присутствует числовой атрибут pri, то он играет роль *приоритета* в списке. Если pri не задан, значением приоритета считается 0. Таким образом, если вы хотите, чтобы какой-то объект был первым в списке, давайте ему приоритет pri $<$ 0. Если в конце списка – $>$ 0.

```
obj {
    pri = -100;
    nam = 'штука';
    disp = 'Очень важный предмет инвентаря';
    inv = [[Осторожней с этим предметом.]];
}
```

Глава 14

Функции, которые возвращают объекты

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

- в символах [ ] описаны необязательные параметры;
- ‘что’ или ‘где’ - означает объект (в том числе комнату), заданный тегом, именем или переменной-ссылкой;

Итак, основные функции:

- ‘_(что)’ - получить объект;
- ‘me()’ возвращает текущего объекта-игрока;
- ‘here()’ возвращает текущую сцену;
- ‘where(что)’ возвращает комнату или объект в котором находится заданный объект, если объект находится в нескольких местах, то можно передать второй параметр – таблицу Lua, в которую будут добавлены эти объекты;
- ‘inroom(что)’ аналогично where(), но вернет комнату, в которой расположен объект (это важно для объектов в объектах);

- ‘from([где])’ возвращает прошлую комнату, из которой игрок перешел в заданную комнату. Необязательный параметр – получить прошлую комнату не для текущей комнаты, а для заданной;
- ‘seen(что, [где])’ возвращает объект или переход, если он присутствует и видим, есть второй необязательный параметр – выбрать сцену или объект/список в котором искать;
- ‘lookup(что, [где])’ возвращает объект или переход, если он существует в сцене или объекте/списке;
- ‘inspect(что)’ возвращает объект, если он виден/доступен на сцене. Поиск производится по переходам и объектам, в том числе, в объектах игрока;
- ‘have(что)’ возвращает объект, если он есть в инвентаре и не отключен;
- ‘live(что)’ возвращает объект, если он присутствует среди живых объектов (описано далее);

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

```
onexit = 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^’окно’ then local ww = lookup ’собака’ if not ww then p [[А где моя собака?]] return end place(ww, ’улица’) p ’Я разбил окно! Моя собака выпрыгнула на улицу.’ return end return false end`

Пример с функцией ‘have’:

```
...
act = function(s)
```

```
    if have ’нож’ then
        p ’Но у меня же есть нож!’;
        return
    end
    take ’нож’
end
...
```

Может возникнуть вопрос, в чем разница между функциями lookup и _ ()? Дело в том, что lookup() ищет объект, и в случае, если объект не найден – просто ничего не вернет. А запись _ () предполагает, что вы точно знаете, что за предмет вы получаете. Другими словами, _ () это безусловное получение объекта по имени. Эта функция в общем случае не занимается *поиском*. Только если в качестве параметра задан тег, будет осуществлен поиск среди доступных объектов. Если вы используете _ () на несуществующий объект или недоступный тег – вы получите ошибку!

Глава 15

Другие функции стандартной библиотеки

В INSTEAD в модуле stdlib, который всегда подключается автоматически, определены функции, которые предлагаются автору как основной рабочий инструмент по работе с миром игры. Рассмотрим их в этой главе.

При описании функции в большинстве функций под параметром ‘w’ понимается объект или комната, заданная именем, тегом или по переменной-ссылке. [wh] - означает необязательный параметр.

- include(файл) - включить файл в игру;

```
include ”lib” -- включит файл lib.lua из текущего каталога с игрой;
```

- loadmod(модуль) - подключить модуль игры;

```
loadmod ”module” -- включит модуль module.lua из текущего каталога;
```

- rnd(m) - случайное целочисленное значение от ‘1’ до ‘m’;
- rnd(a, b) - случайное целочисленное значение от ‘a’ до ‘b’, где ‘a’ и ‘b’ целые >= 0;
- rnd_seed(что) - задать зерно генератора случайных чисел;
- p(…) - вывод строки в буфер обработчика/атрибута (с пробелом в конце);

- pr(…) - вывод строки в буфер обработчика/атрибута “как есть”;
- pn(…) - вывод строки в буфер обработчика/атрибута (с переводом строки в конце);
- pf(fmt, …) - вывод форматной строки в буфер обработчика/атрибута;

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

- pfn(…)(…)… “строка” - формирование простого обработчика; Данная функция упрощает создание простых обработчиков:

```
act = pfn(walk, 'ванная') "Я решил зайти в ванную.";
act = pfn(enable, '#переход') "Я заметил отверстие в стене!";
```

- obj {} - создание объекта;
- stat {} - создание статуса;
- room {} - создание комнаты;
- menu {} - создание меню;
- dlg {} - создание диалога;
- me() - возвращает текущего игрока;
- here() - возвращает текущую сцену;
- from([w]) - возвращает комнату из которой осуществлен переход в текущую сцену;
- new(конструктор, аргументы) - создание нового *динамического* объекта (будет описано далее);
- delete(w) - удаление динамического объекта;
- gamefile(файл, [сбросить состояние?]) - подгрузить динамически файл с игрой;

```
gamefile(”part2.lua”, true) -- сбросить состояние игры (удалить
объекты и переменные), подгрузить  part2.lua и начать с main комнаты.
```

- player {} - создать игрока;
- dprint(…) - отладочный вывод;
- visits([w]) - число визитов в данную комнату (или 0, если визитов не было);
- visited([w]) - число визитов в комнату или false, если визитов не было;

```
if not visited() then
    p [[Я тут первый раз.]]
end
```

- walk(w, [булевое exit], [булевое enter], [булевое менять from]) - переход в сцену;

```
walk(’конец’, false, false) -- безусловный переход (игнорировать
onexit/onenter/exit/enter);
```

- walkin(w) - переход в под-сцену (без вызова exit/onexit текущей комнаты);
- walkout([w], [dofrom]) - возврат из под-сцены (без вызова enter/onenter);
- walkback([w]) - синоним walkout([w], false);
- _(w) - получение объекта;
- for_all(fn, ….) - выполнить функцию для всех аргументов;

```
for_all(enable, ’окно’, ’дверь’);
```

- seen(w, [где]) - поиск видимого объекта;
- lookup(w, [где]) - поиск объекта;

- ways([где]) - получить список переходов;
- objs([где]) - получить список объектов;
- search(w) - поиск доступного игроку объекта;
- have(w) - поиск предмета в инвентаре;
- inroom(w) - возврат комнаты/комнат, в которой находится объект;
- where(w, [таблица]) - возврат объекта/объектов, в котором находится объект;

```
local list = {}
local w = where('яблоко', list)
-- если яблоко находится в более, чем одном месте, то
-- list будет содержать массив этих мест.
-- Если вам достаточно одного местоположения, то:
where 'яблоко' -- будет достаточно
```

- closed(w) - true если объект закрыт;
- disabled(w) - true если объект выключен;
- enable(w) - включить объект;
- disable(w) - выключить объект;
- open(w) - открыть объект;
- close(w) - закрыть объект;
- actions(w, строка, [значение]) - возвращает (или устанавливает) число действий типа t для объекта w.

```
if actions(w, 'tak') > 0 then -- предмет w был взят хотя бы 1 раз;
if actions(w) == 1 then -- act у предмета w был вызван 1 раз;
```

- pop(тег) - возврат в прошлую ветвь диалога;
- push(тег) - переход в следующую ветвь диалога

- empty([w]) - пуста ли ветвь диалога? (или объект)
- lifeon(w) - добавить объект в список живых;
- lifeoff(w) - убрать объект из списка живых;
- live(w) - объект жив?;
- change_pl(w) - смена игрока;
- player_moved([pl]) - текущий игрок перемещался в этом такте?;
- inv([pl]) - получить список-инвентарь;
- remove(w, [wh]) - удалить объект из объекта или комнаты; Удаляет объект из списков obj и way (оставляя во всех остальных, например, game.lifes);
- purge(w) - уничтожить объект (из всех списков); Удаляет объект из *всех* списков, в которых он присутствует;
- replace(w, ww, [wh]) - заменить один объект на другой;
- place(w, [wh]) - поместить объект в объект/комнату (удалив его из старого объекта/комнаты);
- put(w, [wh]) - поместить объект без удаления из старого местоположения;
- take(w) - забрать объект;
- drop(w, [wh]) - выбросить объект;
- path {} - создать переход;
- time() - число ходов от начала игры.

Важно!

На самом деле, многие из этих функций 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())”; Впрочем, remove(apple) тоже сработает и удалит объект из тех мест, где он присутствует.

Рассмотрим несколько примеров.

```
act = function()
    pn ”Я иду в следующую комнату...”
    walk (nextroom);
end

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

Важно!

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 = function()
        pn ”Я иду в следующую комнату...”
        walk (nextroom);
        return
end
```

Не забывайте также, что при вызове ‘walk’ вызовутся обработчики ‘onexit/onenter/exit/enter” и если они запрещают переход, то он не произойдет.

Глава 16

Диалоги

Диалоги – это сцены специального типа ‘dlg’, содержащие объекты – фразы. При входе в диалог игрок видит перечень фраз, 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’). При повторном входе в диалог, все скрытые фразы снова становятся видимыми и диалог сбрасывается в начальное состояние (если, конечно, автор игры специально не прикладывал усилия по изменению вида диалога).

Переход в диалог в игре осуществляется как переход на сцену:

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

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

```
obj {
    nam = ’повар’;
```

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

Если вам не нравится префикс у фраз в виде дефиса, вы можете определить строковую переменную:

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

И получить префикс в виде '+' перед каждой фразой. Вы также можете сделать префикс функцией. На вход функции в таком случае будет поступать в виде параметра номер фразы. Задача функции – вернуть строковый префикс.

Обратите внимание, что 'std.phrase_prefix' не сохраняется, если вам нужно переопределять ее на лету, вам придется восстанавливать ее состояние в 'start()' функции вручную!

Важно!

> Я рекомендую использовать модуль 'noinv' и задавать свойство 'noinv' 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:

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

16.1 Фразы

Центральным понятием в диалогах является *фраза*. Фразы это не просто вопрос-ответ, как можно подумать. Фраза является деревом, и в этом смысле, весь диалог может быть реализован единственной фразой. Например:

```
dlg {
    nam = ’разговор’;
    title = [[Разговор с продавцом]];
    enter = [[Я обратился к продавцу.]];
    phr = {
        { ’У вас есть бобы?’, ’-- Нет.’},
        { ’У вас есть шоколад?’, ’-- Нет.’},
        { ’У вас есть квас?’, ’-- Да’,
            { ’А сколько он стоит?’, ’-- 50 рублей.’ },
            { ’А он холодный?’, ’-- Холодильник сломался.’,
                { ’Беру два!’, ’Остался один.’,
                    { ’Дайте один!’, function() p [[Ок!]]; take ’квас’; end };
                }
            }
        }
    }
}
```

Как видно из примера, фраза задается атрибутом phr и может содержать разветвленный диалог. Фраза содержит в себе выборы, каждый из которых тоже может содержать в себе выборы и так далее…

Фраза имеет формат пары: описатель – реакция. В простейшем случае, это строки. Но они могут быть и функциями. Обычно, функцией бывает реакция, которая может содержать код по изменению игрового мира.

Пара может быть простой:

```
{’Вопрос’, ’Ответ }
```

А может содержать в себе массив пар:

```
{’Вопрос’, ’Ответ’,
    {’Под-вопрос1’, ’Под-ответ1’ },
    {’Под-вопрос2’, ’Под-ответ2’ },
}
```

На самом деле, если вы посмотрите внимательно на атрибут phr, то вы заметите, что массив выборов тоже является вложенным в главную фразу phr, но только первоначальная пара отсутствует:

```
dlg {
    nam = 'разговор';
    title = [[Разговор с продавцом]];
    enter = [[Я обратился к продавцу.]];
    phr = {
    -- тут мог бы быть вопрос ответ 1-го уровня!
    -- 'Главный вопрос', 'Главный ответ',
        { 'У вас есть бобы?', '-- Нет.'},
        { 'У вас есть шоколад?', '-- Нет.'},
        { 'У вас есть квас?', '-- Да',
            { 'А сколько он стоит?', '-- 50 рублей.' },
            { 'А он холодный?', '-- Холодильник сломался.',
                { 'Беру два!', 'Остался один.',
                    { 'Дайте один!', function() p [[Ок!]]; take 'квас'; end };
                }
            }
        }
    }
}
```

На самом деле, так и есть. И вы можете добавить 'Главный вопрос' и 'Главный ответ', но только вы не увидите этот главный вопрос. Дело в том, что при входе в диалог фраза phr автоматически раскрывается, так как обычно нет никакого смысла в диалогах из одной единственной фразы. И гораздо проще понять диалог как набор выборов, чем как единственную древовидную фразу. Так что у phr никогда нет первоначальной пары вопрос-ответ, но мы сразу попадаем в массив вариантов, что более понятно.

Когда мы говорим о том, что диалог на самом деле реализован одной фразой, мы не совсем правы. Дело в том, что мы имеем дело с фразой, внутри которой находятся другие фразы… Это напоминает нам ситуацию с объектами. Действительно, фразы – это объекты! Которые могут находиться внутри друг-друга. Итак, взглянем на диалог свежим взглядом:

```
dlg {
    nam = 'разговор';
    title = [[Разговор с продавцом]];
    enter = [[Я обратился к продавцу.]];
    phr = { -- это объект типа фраза, без dsc и act
```

```
        -- это 1-я фраза, внутри фразы с dsc и act
        { 'У вас есть бобы?', '-- Нет.'},
        -- это 2-я фраза, внутри фразы с dsc и act
        { 'У вас есть шоколад?', '-- Нет.'},
        -- это 3-я фраза, внутри фразы с dsc и act
        { 'У вас есть квас?', '-- Да',
        -- это 1-я фраза внутри 3й фразы с dsc и act
        { 'А сколько он стоит?', '-- 50 рублей.' },
            { 'А он холодный?', '-- Холодильник сломался.',
                { 'Беру два!', 'Остался один.',
                    -- здесь act в виде функции
                    { 'Дайте один!', function() p [[Ок!]]; take 'квас'; end };
                }
            }
        }
    }
}
```

Как видим, диалог – это комната, а фразы – специальные объекты! Теперь вам станет понятным дальнейшее изложение.

> Внимание! По умолчанию, когда игрок нажимает на один из вопросов в списке, движок повторяет его в выводе и только потом выводит ответ. Это сделано для того, чтобы диалог выглядел связанным. Если вы хотите отключить такое поведение, используйте настройку std.phrase_show:

```
std.phrase_show = false -- не выводить фразу-вопрос при выборе
```

Эта настройка действует на все диалоги, устанавливайте ее в init() или start() функции.

16.2 Атрибуты фраз

Рассмотрим вариант фразы: `phr = { { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?', {'Красную', 'Держите!' }, {'Синюю', 'Вот!' }, } }`

Если запустить этот диалог, то в после выбора, скажем, красной таблетки, у нас останется еще один выбор синей таблетки. Но наш замысел, явно не в этом! Существует несколько способов сделать диалог правильным.

Во первых, вы можете воспользоваться pop() – возвратом на предыдущий уровень диалога:

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        {'Красную', function() p 'Держите!'; pop() end; },
        {'Синюю', function() p 'Вот!'; pop() end; },
    }
}
```

Или, в другой записи:

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        {'Красную', pfn(pop) 'Держите!' },
        {'Синюю', pfn(pop) 'Вот!' },
    }
}
```

Но это не слишком удобно, кроме того, что если эти фразы содержат в себе новые фразы? В случаях, когда вариант предлагает выбор, и этот выбор должен быть единственным, вы можете задать у фразы атрибут only:

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!' },
        {'Синюю', 'Вот!' },
    }
}
```

В таком случае, после выбора фразы, все фразы текущего контекста будут закрыты.

Еще одна частая ситуация, вы хотите, чтобы фраза не пряталась после ее активации. Это делается заданием флага true:

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!' },
        {'Синюю', 'Вот!' },
        { true, 'А какая лучше?', 'Тебе выбирать.' }, -- фраза
        -- которая никогда не будет скрыта
    }
}
```

Альтернативная запись, с явным заданием атрибута always:

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!' },
        {'Синюю', 'Вот!' },
        { always = true, 'А какая лучше?', 'Тебе выбирать.' }, -- фраза
        -- которая никогда не будет скрыта
    }
}
```

Еще один пример. Что-если мы хотим, чтобы фраза была показана(или спрятана) по какому-либо условию? Для этого есть функция-обработчик cond.

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!' },
        {'Синюю', 'Вот!' },
        { true, 'А какая лучше?', 'Тебе выбирать.' }, -- фраза
        -- которая никогда не будет скрыта
    },
    { cond = function() return have 'яблоко' end,
        'А хотите яблоко?', 'Спасибо, нет.' };
}
```

В данном примере, только при наличии у игрока яблока, покажется ветка диалога 'А хотите яблоко?'.

Иногда бывает удобно выполнить действие в тот момент, когда варианты текущего уровня(контекста) диалога исчерпаны. Для этого служит функция-обработчик onempty.

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!' },
        {'Синюю', 'Вот!' },
        onempty = function()
            p [[Ты сделал свой выбор.]]
            pop()
        end;
    },
    { cond = function() return have 'яблоко' end,
        'А хотите яблоко?', 'Спасибо, нет.' };
}
```

Обратите внимание, что когда есть метод onempty, автоматический возврат в предыдущую ветку не производится, предполагается, что метод onempty сделает все, что нужно.

Все описанные атрибуты могут быть установлены у любой фразы. В том числе и на 1-м уровне:

```
phr = {
    onempty = function()
        p [[Вот и поговорили.]]
        walkout()
    end;
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!' },
        {'Синюю', 'Вот!' },
        onempty = function()
            p [[Ты сделал свой выбор.]]
            pop()
        end;
    },
```

```
    { cond = function() return have 'яблоко' end,
        'А хотите яблоко?', 'Спасибо, нет.' };
}
```

16.3 Теги

Только что мы рассмотрели механизмы диалогов, которые уже позволяют создавать довольно сложные диалоги. Однако, и этих средств может не хватить. Иногда нам нужно уметь обращаться к фразам из других мест диалога. Например, выборочно включать их, или анализировать их состояние. А также делать переходы из одних ветвей диалога в другие.

Все это возможно для фраз, у которых есть тег. Создать фразу с тегом очень просто:

```
phr = {
    { '#что?', 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        {'#красная', 'Красную', 'Держите!' },
        {'#синяя', 'Синюю', 'Вот!' },
    },
}
```

Как видим, наличие в начале фразы строки, которая начинается на символ '#' - означает наличие тега.

Для таких фраз работают стандартные методы, такие как seen или enable/disable. Например, мы могли бы обойтись без атрибута only следующим образом:

```
phr = {
    { '#что?', 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        {'#красная', 'Красную', 'Держите!'
            cond = function(s)
                return not closed('#синяя')
            end
        },
        {'#синяя', 'Синюю', 'Вот!',
            cond = function(s)
                return not closed('#красная')
            end
```

```
        },
    },
}
```

Теги, кроме того, что позволяют узнавать и менять состояние конкретных фраз, делают возможным переходы между фразами. Для этого используются функции push и pop.

push(куда) – делает переход на фразу с запоминанием позиции в стеке.

pop([куда]) – вызванная без параметра, поднимается на 1 позицию в стеке истории. Можно указать конкретный тег фразы, которая должна быть в истории, в таком случае возврат будет осуществлен на нее.

Нужно отметить, что при переходе по push, мы переходим не на одну фразу, а на список фраз этой фразы. То-есть раскрываем ее, также как это сделано для главной фразы phr. Например:

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!', next = '#отаблетке' },
        { 'Синюю', 'Вот!', next = '#отаблетке' },
    },
    { false, '#отаблетке',
        {'Я сделал верный выбор?', 'Время покажет.'}
    },
}
```

Тут мы видим сразу несколько приемов:

- атрибут next, вместо явного описания реакции в виде функции с push. next – это простой способ записать push.

- false в начале фразы, делает фразу выключенной. Она находится в состоянии выключена, пока не сделать явный enable. Однако внутрь фразы мы можем перейти, и показать содержимое выборов. Альтернативная запись возможна с использованием атрибута hidden:

```
    { hidden = true, '#отаблетке',
        {'Я сделал верный выбор?', 'Время покажет.'}
    },
```

Таким образом можно записывать диалоги не древовидно, а линейно. Еще одна особенность переходов состоит в том, что если у фразы не описана реакция, то при переходе будет вызван заголовок фразы:

```
phr = {
    { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
        only = true,
        {'Красную', 'Держите!', next = '#отаблетке' },
        { 'Синюю', 'Вот!', next = '#отаблетке' },
    },
    { false, '#отаблетке', [[Я взял таблетку и мастер хитро улыбнулся.]],
        {'Я сделал верный выбор?', 'Время покажет.'},
        {'Что делать дальше?', 'Ты свободен.'},
    },
}
```

При выборе таблетки, будет вызван заголовочный метод фразы `#отаблетке', а уже потом будет представлен выбор.

Если вам нравится линейная запись, вы можете предпочесть следующий вариант:

```
dlg {
    nam = 'диалог';
    phr = {
        { 'Что это у вас?', 'Таблетки. Красная и синяя. Вам какую?',
            only = true,
            {'Красную', 'Держите!', next = '#отаблетке' },
            { 'Синюю', 'Вот!', next = '#отаблетке' },
        }
    }
}:with {
    { '#отаблетке', [[Я взял таблетку и мастер хитро улыбнулся.]],
        {'Я сделал верный выбор?', 'Время покажет.'},
        {'Что делать дальше?', 'Ты свободен.'},
    },
}
```

Дело в том, что атрибут phr диалога задает первый объект комнаты. Но вы можете заполнить объекты комнаты обычным образом: задав obj или with.

Так как при входе в диалог раскрывается 1-я фраза, то остальные фразы вы не увидите (обратите внимания, у фразы `#отаблетке' не стоит false), но вы сможете делать переходы на эти фразы.

16.4 Методы

Как вы уже знаете, объекты в INSTEAD могут находиться в состоянии открыт/закрыт и выключен/включен. Как это соответствует фразам диалога?

Для обычных фраз, после активации выбора фраза *закрывается*. При повторном входе в диалог все фразы *открываются*.

Для фраз с always = true (или true в начале определения) – такого закрытия не происходит.

Для фраз с hidden = true (или false в начале определения) – фраза будет создана как выключенная. Она не будет видима до тех пор, пока не будет явно включена.

Для фраз с cond(), каждый раз при просмотре фраз вызывается этот метод, и в зависимости от возвращаемого значения (true/не true) фраза включается или выключается.

Зная это поведение, вы можете прятать/показывать и анализировать фразы обычными функциями вида: disable /enable /empty /open /close /closed / disabled и так далее…

Однако, делать вы это можете только в самом диалоге, так как все фразы идентифицируются по тегам. Если вы хотите модифицировать состояние/ анализировать фразы из других комнат вы можете:

- дать фразе имя { nam = `имя' }…
- искать фразу по тегу в другой комнате: local ph = lookup(`#тег', `диалог') и потом работать с ней;

Что касается функций push/pop, то вы можете вызывать их явно как методы диалога, например:

```
_'диалог':push '#новая'
```

Но лучше это делать в самом диалоге, например, в enter.

Кроме того есть метод :reset, который сбрасывает стек переходов и устанавливает стартовую фразу, например:

```
enter = function(s)
    s:reset '#начало'
end
```

Следует отметить, что когда вы делаете enable/disable/open/close фразы, то вы выполняете действие именно над этой фразой, а не над фразами включенными внутрь. Но так как при показе фраз движок остановится на выключенном/закрытом объекте-фразе и не войдет внутрь, этого достаточно.

Глава 17

Специальные объекты

В STEAD3 существуют специальные объекты, которые выполняют специфические функции. Все такие объекты можно разделить на два класса:

1. Системные объекты @;

2. Подстановки $.

Системные объекты, это объекты, чье имя начинается с символа ‘@’ или ‘$’. Такие объекты обычно создаются в *модулях*. Они не уничтожаются при смерти игрового мира (например, при подгрузке gamefile, при загрузке игры из сохранения, и так далее). Примеры объектов: @timer, @prefs, @snd.

Такие объекты, кроме своих специальных функций, могут быть использованы по ссылке, без явного помещения объекта в сцену или инвентарь, но механизм действия таких объектов – особенный.

17.1 Объект ‘@’

Обычно, вам не нужно работать с такими объектами, но в качестве примера рассмотрим реализацию ‘ссылок’.

Пусть мы хотим сделать ссылку, при нажатии на которую мы перейдем в другую комнату. Конечно, мы могли бы добавить объект в сцену, но стоит ли это делать в таком простом случае?

Как нам может помочь системный объект?

```
obj {
```

```
    nam = '@walk';
    act = function(s, w)
        walk(w, false, false)
    end;
}
room {
    nam = 'main';
    title = 'Начало';
    decor = [[Начать {@walk старт|приключение}]];
}
```

При нажатии на ссылку "приключение" будет вызван метод act объекта '@walk' с параметром "старт".

На самом деле, в стандартной библиотеке stdlib уже есть объект, с именем '@', который позволяет делать свои обработчики ссылок следующим образом:

```
xact.walk = walk

room {
    nam = 'main';
    title = 'Начало';
    decor = [[Начать {@ walk старт|приключение}]];
}
```

Обратите внимание, на пробел после @. Данная запись делает следующее:

- берет объект '@' (такой объект создан библиотекой stdlib);
- берет его act;
- вызывает act с параметрами walk и старт;
- act объекта '@' смотрит в массив xact;
- walk определяет метод, который будет вызван из массива xact;
- старт – параметр этого метода.

Другой пример:

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

room {
    nam = 'main';
    title = 'Начало';
    decor = [[Нажми {@ myprint "hello world"|на кнопку}]];
}
```

17.2 Подстановки

Объекты, чье имя начинается на символ '$' тоже считаются системными объектами, но работают они по-другому.

Если в выводе текста встречается "ссылка" вида:

```
{$my a b c|текст}
```

То происходит следующее:

1. Берется объект $my;
2. Берется act объекта $my;
3. Вызывается act: _'$my':(a, b, c, текст);
4. Возвращаемая строка заменяет собой всю конструкцию {...}.

Таким образом, объекты играют роль подстановки.

Зачем это нужно? Представьте себе, что вы разработали модуль, который превращает записи формул из текстового вида в графические. Вы пишете объект $math который в своем act методе превращает текст в графическое изображение (спрайт) и возвращает его в текстовый поток. Тогда пользоваться таким модулем крайне просто, например:

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

Глава 18

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: - Игрок нажимает на ссылку; - Реакция ‘act’, ‘use”, ‘inv’, ‘tak’, осмотр сцены (клик по названию сцены) или переход в другую сцену; - Динамические события; - Вывод нового состояния сцены.

Например, сделаем Барсика живым:

```
obj {
    nam = ’Барсик’;
    { -- не сохранять массив lf
        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 'Барсик' -- добавить в инвентарь
lifeon 'Барсик' -- 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 'exit', или любым другим способом.

> Если в вашей игре много "живых" объектов, вы можете задавать им явную позицию в списке, при добавлении. Для этого, воспользуйтесь вторым числовым параметром (целое неотрицательное число) 'lifeon', чем меньше число, тем выше приоритет. 1 – самый высокий. Или вы можете использовать атрибут pri у объекта. Правда, этот атрибут будет влиять на приоритет объекта в любом списке.

Если вам нужен фоновый процесс в какой-то комнате, запускайте его в 'enter' и удаляйте в 'exit', например:

```
room {
        nam  = 'В подвале';
        dsc = [[Тут темно!]];
        enter = function(s)
                lifeon(s);
        end;
        exit = 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 =  { 'Дом' };
}
```

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

```
obj {
    nam  = 'фонарик';
    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()'.

```
obj {
    nam  = 'динамит';
    timer = 0;
    used = function(s, w)
        if w^'спичка' then -- спичка?
            if live(s) then
                return "Уже горит!"
            end
            p "Я поджег динамит."
            lifeon(s)
            return
        end
        return false -- если не спичка
```

```
    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, (‘true’ или ‘false’). Если вы вернете true – то это будет признаком важного события, которое выведется до описания объектов сцены, например:

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

Или: `return ’В комнату вошел охранник.’, true`

Если вы вернете false, то цепочка life методов прервется на вас. Это удобно делать при выполнении walk из метода life, например:

```
life = function()
    walk ’theend’
    return false -- это последний life
end
```

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

```
require ”noinv”
require ”nolife”
```

```
dlg {
        nam = ’Охранник’;
        noinv = 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 обычно предполагается что вы:

1. Очищаете вывод реакций: game:reaction(false);
2. Очищаете вывод живых методов на данный момент: game:events(false, false)
3. Делаете walk.
4. Останавливаете цепочку life вызовов с помощью return false;

Некоторые моменты требуют пояснений.

game:reaction() – позволяет взять/изменить вывод реакции пользователя, если задать его в false это означает сбросить реакцию.

game:events() – позволяет взять/изменить вывод life методов. В качестве параметров принимаются приоритетные и не приоритетные сообщения, задав false, false мы отменили весь вывод предыдущих life методов.

В стандартной библиотеке уже есть функция life_walk(), которая делает описанные действия. Вам остается только вернуть false.

Глава 19

Графика

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

```
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 ‘fmt.img’ (Для этого включите модуль fmt).

Например:

```
require ”fmt”
```

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

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

Дело в том, что картинка сцены масштабируется по другому алгоритму. Картинки 'fmt.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 'fmt.img' внутрь { и }, то получите графическую ссылку.

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

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

Важно!

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

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

```
fmt.imgl ’pad:16,picture.png’ -- отступы по 16 от каждого края
fmt.imgl ’pad:0 16 16 4,picture.png’ -- отступы: вверху 0, справа 16, внизу 16, слева 4
fmt.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 = fmt.img ’blank:32x32’..[[Строка с пустым изображением.]];
dsc = fmt.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 “sprite”, который предоставляет более широкие возможности по графическому оформлению. Но я крайне не рекомендую делать это в своей первой игре.

Глава 20

Музыка

Для работы с музыкой и звуками вам понадобится модуль snd.

```
require ”snd”
```

Интерпретатор проигрывает в цикле текущую музыку, которая задается с помощью функции: ‘snd.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”).

В качестве частоты музыкальных файлов используйте частоты кратные 11025.

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

`snd.music()' без параметра возвращает текущее имя трека.

В функцию `snd.music()' можно передавать второй параметр – количество проигрываний (циклов). Получить текущий счетчик можно с помощью `snd.music()' без параметров – второе возвращаемое значение. 0 – означает вечный цикл. 1..n – количество проигрываний. –1 – проигрывание текущего трека закончено.

Для того, чтобы отменить проигрывание музыки, вы можете использовать `snd.stop_music()'

Для того, чтобы узнать, играет ли музыка:

```
snd.music_playing()
```

Вы можете задавать время нарастания и затухания музыки, с помощью вызова:

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

Здесь o - время в мс. для затухания и i - время в мс. для нарастания музыки. Если задан только один параметр – оба времени считаются одинаковыми. После вызова, установленные параметры будут влиять на проигрывание всех музыкальных файлов.

Для проигрывания звуков используйте `snd.play()'. 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 ‘snd.play()’ позволяет запускать зацикленные звуки, то различие между музыкой и звуками становится уже не таким однозначным.

Итак, определение функции: ‘snd.play(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 ‘snd.stop()’. Для остановки звука в определенном канале ‘snd.stop(канал)’.

Важно!

> Если вы используете зацикленные звуки, вам придется самим восстанавливать их состояние (запускать снова с помощью ‘snd.sound()’) в функции ‘start()’

Например:

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

Если вам не достаточно описанных здесь функций по работе со звуком, используйте полное описание модуля “snd”.

Глава 21

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 “fmt” улучшают качество вывода игры без дополнительных усилий со стороны автора.

Например:

```
require 'fmt'
fmt.para = true -- включить отступы параграфов
```

И ваша игра будет выглядеть гораздо лучше. Если вам нужна какая-то автоматическая обработка выводимого текста, вы можете включить модуль “fmt” и определить функцию ‘fmt.filter’. Например:

```
require "fmt"
fmt.filter = function(s, state)
    -- s -- вывод
    -- state -- true, если это такт игры (вывод сцены)
    if state then
        return 'Эта строка будет добавлена к началу вывода\n'..s;
    end
    return s
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.

> Внимание! По умолчанию, все конечные и начальные переводы строк, пробелы и символы табуляции вырезаются из вывода обработчиков. Так как обычно они не имеют смысла и даже вредны. В редких случаях, автору может понадобиться более полный контроль над выводом, тогда он может задать std.strip_call как false в init() или start(), например:

```
std.strip_call = false

obj {
    dsc = [[Тут лежит {яблоко}.^^^^]] -- теперь переводы строк
    -- не будут вырезаны, хотя это странное желание
}
```

> Но обычно такое ручное форматирование свидетельствует о плохом стиле. Для оформления сцены лучше использовать decor и/или подстановки $.

21.1 Форматирование

Вы можете делать простое форматирование текста с помощью функций:

- fmt.c(строка) - разместить по центру;
- fmt.r(строка) - разместить справа;
- fmt.l(строка) - разместить слева;
- fmt.top(строка) - сверху строки;
- fmt.bottom(строка) - снизу строки;
- fmt.middle(строка) - середина строки (по умолчанию).

Например: `room { nam = 'main'; title = 'Добро пожаловать'; dsc = fmt.c '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 "fmt.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: fmt.nb(строка). Например, модуль "fmt" использует неразрывные строки для создания отступов в начале параграфов. Также, 'fmt.nb' может оказаться удобной для вывода служебных символов. Можно сказать, что вся строка-параметр 'fmt.nb' воспринимается движком как одно большое слово.

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

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

```
fmt.tab(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 'fmt.tab' 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:

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

Конечно, не очень удачный пример, так как то же самое можно было сделать с помощью ‘fmt.c()’. Более удачный пример.

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

На самом деле, единственная ситуация, когда применение ‘fmt.tab()’ оправдано – это вывод табличных данных.

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

```
-- размещение 'Раз' по центру строки
dsc = fmt.tab('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 ‘fmt.nb()’.

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

В INSTEAD также существует возможность выполнять простое вертикальное форматирование. Для этого используйте вертикальный табулятор:

```
fmt.y(позиция, [центр])
```

Как и в случае с fmt.tab *позиция*, это текстовый или числовой параметр. Здесь он воспринимается как позиция строки, выраженная в пикселях или процентах от высоты области сцены. Например, 100% – соответствует нижней границе области сцены. 200% – соответствует нижней границе второй страницы вывода (две высоты области вывода сцены).

Необязательный строковой параметр *центр* задает позицию внутри строки, относительно которой выполняется позиционирование:

- top (по верхнему краю);
- middle (по центру);
- bottom (по нижнему краю – значение по умолчанию).

Следует отметить, что ‘fmt.y’ работает целиком для строки. Если в строке встретится несколько fmt.y, действовать будет последний из табуляторов.

```
-- размещение ’ГЛАВА I’ - в центре сцены
dsc = fmt.y(’100%’)..”ГЛАВА I”;
```

Если позиция, указанная табулятором, уже занята другой строкой, табулятор игнорируется.

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

```
std.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 ‘game.display()’, которая по умолчанию выглядит следующим образом:

```
game.display = function(s, state)
    local r, l, av, pv
    local reaction = s:reaction() or nil -- реакция
    r = std.here()
    if state then -- такт игры?
```

```
        reaction = iface:em(reaction) -- курсив
        av, pv = s:events()
        av = iface:em(av) -- вывод ”важных” life
        pv = iface:em(pv) -- вывод фоновых life
        l = s.player:look() -- objects [and scene] -- объекты и сцена
    end
    l = std.par(std.scene_delim,
        reaction or false, av or false, l or false,
        pv or false) or ''
    return l
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.

21.2 Оформление

Вы можете менять начертание текста с помощью комбинаций функций:

- fmt.b(строка) - жирный текст;
- fmt.em(строка) - курсив;
- fmt.u(строка) - подчеркнутый текст;
- fmt.st(строка) - перечеркнутый текст.

Например: `room { nam = 'Intro'; title = false; dsc = function(s) p ('Вы находитесь в комнате: ') p (fmt.b(s)) end; }`

> Используя функции ‘fmt.u’ и ‘fmt.st’ на строках, содержащих пробелы, вы получите разрывы линий в этих местах. Что избежать этого, можно превратить текст в *неразрывную строку*:

```
fmt.u(fmt.nb ”теперь текст без пропусков” )
```

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 ‘fmt.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.

Глава 22

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. Это можно сделать одним из следующих способов:

- Создать свой конструктор;
- Создать новый класс объектов.

22.1 Конструкторы

Конструктор – это функция, которая создает за вас объект и заполняет его атрибуты так, как вам это нужно. Рассмотрим пример. Допустим, в вашей игре будет много окон. Нужно создавать окна, любое окно можно разбить. Мы можем написать конструктор ‘window’.

```
window = function(v)
    v.window = true
        v.broken = false
    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^'молоток' then
                if s.broken then
                    p [[Окно уже разбито.]]
                else
                    p [[Я разбил окно.]]
                    s.broken = true;
                end
                return
            end
            return false
        end
    end
    return obj(v)
end
```

Как видим, идея конструкторов проста. Вы просто создаете функцию, которая получает на вход таблицу с атрибутами {}, которую конструктор может дозаполнить нужными атрибутами. Затем эта таблица передается конструктору obj/room/dlg и возвращается полученный объект.

Теперь, создавать окна стало легко:

```
window {
    dsc = [[Тут есть {окно}.]];
}
```

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

```
obj = { window {
```

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

У нашего окна будет готовый used метод и act метод. Вы можете проверить тот факт, что объект окно – просто проверив признак window:

```
use = function(s, w)
    if w.window then
        p [[Действие на окно.]]
        return
    end
    return false
end
```

Состояние "разбитости" окна, это атрибут broken.

Как реализовать наследование в конструкторах?

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

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

```
global { score = 0 }
treasure = function()
        local v = {}
        v.disp = 'сокровище'
        v.treasure = true
        v.points = 100
        v.dsc = function(s)
                p ('Здесь есть {', std.dispof(s), '}.')
        end;
        v.inv = function(s)
                p ('Это же ', std.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.disp = ’золото’;
        v.gold = true;
        v.points = 50;
        v.dsc = dsc;
        return v
end

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

chest = function(dsc)
        local v = treasure();
        v.disp = ’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}.”);

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 ‘std.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.what = what
    v.broken = false
    if dsc == nil then
        v.dsc = ’Здесь есть {окно}’
    end
    v.act = function(s)
        if s.broken then
                p [[Окно разбито.]]
        end
        local r, v = stead.call(s, ’what’)
        if v then -- обработчик выполнился?
```

```
                p(r)
        else
                p [[За окном темно.]]
        end
    end
    return obj(v)
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.

22.2 Класс объектов

Конструкторы объектов широко использовались в STEAD2. В STEAD3 obj/dlg/room реализованы как классы объектов. Класс объектов удобно создавать для тех случаев, когда поведение создаваемого объекта не укладывается в стандартные объекты obj/room/dlg и вы хотите поменять методы класса. Изменив метод класса, например, вы можете вообще изменить то, как выглядит предмет в сцене. В качестве примера, рассмотрим создание класса “контейнер”. Контейнер может хранить в себе другие объекты, быть закрытым и открытым.

```
-- create own class container
cont = std.class({ -- создаем класс cont
    __cont_type = true; -- для определения типа объекта
    display = function(s) -- переопределяем метод показа предмета
        local d = std.obj.display(s)
        if s:closed() or #s.obj == 0 then
            return d
        end
        local c = s.cont or 'Внутри: ' -- описатель содержимого
        local empty = true
        for i = 1, #s.obj do
            local o = s.obj[i]
            if o:visible() then
                empty = false
```

```
                if i > 1 then c = c .. ’, ’ end
                c = c..’{’..std.nameof(o)..’|’..std.dispof(o)..’}’
            end
        end
        if empty then
            return d
        end
        c = c .. ’.’
        return std.par(std.space_delim, d, c)
    end;
}, std.obj) -- мы наследуемся от стандартного объекта
```

После этого, вы можете создавать контейнеры так:

```
cont {
    nam = ’ящик’;
    dsc = [[Тут есть {ящик}.]];
    cont = ’В ящике: ’;
}: with {
    ’яблоко’, ’груша’;
}
```

Когда контейнер будет открыт, вы увидите описание ящика, а также содержимое ящика в виде строки ссылок: В ящике: яблоко, груша. dsc объектов яблоко и груша будут тоже показаны, если они заданы.

Если необходимо прятать dsc объектов при открытии контейнера, оставляя лишь имена объектов, то можно выполнить следующую модификацию:

```
-- заменим функцию показа любого объекта
-- если объект внутри контейнера, не вызывать его dsc
std.obj.display = function(self)
    local w = self:where() -- где объект?
    if not std.is_obj(w, ’cont’) then -- если не в контейнере
        local d = std.call(self, ’dsc’)
        return d
    end
end
```

К сожалению, подробное описание классов выходит за рамки данного руководства, эти вещи будут описаны в другом руководстве для разработчиков модулей. А пока, для вашей первой игры, вам не стоит писать свои классы объектов.

Глава 23

Полезные советы

23.1 Разбиение на файлы

Когда ваша игра становится большой, размещение ее кода целиком в ‘main3.lua’ – плохая идея.

Для разбиения текста игры на файлы вы можете использовать ‘include’. Вы должны использовать ‘include’ в глобальном контексте таким образом, чтобы во время загрузки ‘main3.lua’ загрузились и все остальные фрагменты игры, например.

```
-- main3.lua
include ”episode1” -- .lua можно опускать
include ”npc”
include ”start”

room {
    nam = ’main’;
....
```

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 доопределять объекты). Для этого вы можете воспользоваться функцией ‘gamefile’:

```
...
act = function() gamefile (”episode2”) end -- .lua можно опускать
...
```

> Внимание! Если в вашей игре определена функция init(), то в подгружаемых частях она также должна присутствовать! В противном случае, после подгрузки файла, будет вызвана текущая функция init(), что обычно не является желательным.

‘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’. Имейте в виду, что существующие модули остаются и переживают операцию gamefile в обоих случаях. ‘gamefile()’ можно использовать только в обработчиках.

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

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

23.2 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 (‘act’) будет вызван после одного клика мыши. Если обработчик возвращает false, то состояние игры не изменяется. Например, реализация кармана:

```
menu {
    state = false;
    nam = ’карман’;
    disp = function(s)
        if s.state then
            return fmt.u(’карман’); -- подчеркиваем активный карман
        end
        return ’карман’;
    end;
    gen = function(s)
        if s.state then
            s:open(); -- показать все предметы в кармане
        else
            s:close(); -- спрятать все предметы в кармане
        end
        return s
    end;
    act = function(s)
        s.state = not s.state -- изменить состояние
        s:gen(); -- открыть или закрыть карман
    end;
}: with {
    obj {
        nam = ’нож’;
        inv = ’Это нож’;
    };
}

function init()
    take ’карман’:gen()
end
```

23.3 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;
}

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

function init()
    take 'статус'
end
```

23.4 walk из обработчиков onexit и onenter

Вы можете делать 'walk' из обработчиков 'onenter' и 'onexit'. Например, 'path' реализован как комната с обработчиком 'onenter', который переносит игрока в другую комнату.

Рекомендуется возвращать из onexit/onenter false в случае, если вы делаете walk из этих обработчиков.

23.5 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 обычных include/ gamefile. Однако, для этого вы должны написать в начале main3.lua:

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

При этом главный файл 'main3.lua' необходимо оставлять открытым. Таким образом, схема выглядит следующим образом ('game.lua' – закодированный файл):

```
-- $Name: Моя закрытая игра!$
std.dofile = std.doencfile
include ”game”; -- никто не узнает, как ее пройти!
```

Важно!

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

23.6 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
```

23.7 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.

23.8 Использование параметров обработчика

Пример кода.

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

23.9 Специальные статусы обработчиков

Из обработчика обычно возвращается текст, в виде return “текст сообщения”. Или с помощью функций p()/pr()/pn()/pf(). Кроме этого, есть специальные статусы, которые могут пригодиться при разработке игры.

Возвращение статуса false:

```
return false
```

Такой статус означает что обработчик не выполнил свою функцию и должен быть проигнорирован. Обычно движок в таком случае вызовет обработчик по умолчанию.

Вы можете также вернуть специальный статус:

```
return true, false
```

В этом режиме перерисуется только область инвентаря (но не сцена). Данный статус удобно использовать для реализации меню в области инвентаря.

Существует еще один специальный статус: std.nop(). Он может быть использован просто как вызов функции в конце обработчика или совместно с return.

```
return std.nop()
-- ... или ...
std.nop()
-- далее конец функции или return
```

В этом случае, содержимое сцены останется таким же, как и в прошлый такт игры (даже строка реакции останется старой). Данный статус удобно использовать совместно с модулем theme, когда нужно изменить оформление игры на лету и перерисовать кадр с учетом новых параметров темы.

23.10 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 false, 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’. Если он возвращает false – вызывается game.timer.

Например:

```
game.timer = function(s)
    if time() > 10 then
        return false
    end
    snd.play ’gfx/beep.ogg’;
    p (”Timer:”, time())
end

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

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

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

Вы можете вернуть из таймера специальный статус:

```
return true, false
```

В этом режиме перерисуется только область инвентаря. Это можно использовать для статусов вроде часов.

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

23.11 Музыкальный плеер

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

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

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 };

obj {
    nam = ’player’;
    pos = 0;
    {
        playlist = { ’01 Frozen sun.ogg’, 0,
            ’02 Thinking.ogg’, 0,
            ’03 Melancholy.ogg’, 0,
```

```
            '04 Everyday happiness.ogg', 0,
            '10 Good morning again.ogg', 1,
            '15 [Bonus track] The end (demo cover).ogg', 1
        };
    };
    tick = function(s)
        if snd.music_playing() and ( track_time < 120 or not player_moved() ) then
            return
        end
        track_time = 0
    if s.pos == 0 then
        s.pos = 1
    else
        s.pos = s.pos + 2
    end
        if s.pos > #s.playlist then
            s.pos = 1
        end
        snd.music('mus/'..s.playlist[s.pos], s.playlist[s.pos + 1]);
    end;
}

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

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

23.12 Живые объекты

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

```
obj {
```

```
    nam = 'лошадь';
    dsc = 'Рядом со мной стоит {лошадь}.';
    act = [[Моя лошадка.]];
    life = function(s)
        if player_moved() then
            place(s);
        end
    end;
}

function init()
    lifeon 'лошадь'; -- сразу оживим лошадь
end
```

23.13 Вызов меню

Вы можете вызвать из игры меню INSTEAD с помощью функции 'istead.menu()'. Если в качестве параметра задать: 'save', 'load' или 'quit', то будет вызван соответствующий подраздел меню.

23.14 Динамическое создание объектов

Обычные объекты и комнаты нельзя создавать "на лету". Обычно вы создаете их в глобальном пространстве lua файла. Однако, существуют игры в которых количество объектов неизвестно заранее, или количество объектов велико и они добавляются по ходу игры.

В INSTEAD существует способ создания любого объекта на лету. Для этого вам понадобится написать *конструктор* вашего объекта и воспользоваться функцией 'new'.

Конструктор должен быть декларирован.

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

```
declare 'box' (function()
    return obj {
        dsc = [[Тут лежит {коробка}.]];
        tak = [[Я взял коробку.]];
```

```
    }
end)

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

declare 'box' (function(dsc)
    return obj {
        dsc = dsc;
        tak = [[Я взял коробку.]];
    }
end)
take(new(box, 'В углу стоит {коробка}'))
```

`new' воспринимает первый аргумент – как задекларированную функцию-конструктор, а все остальные параметры – как аргументы конструктору. Результатом выполнения конструктора должен быть объект.

```
function myconstructor()
    local v = {}
    v.disp = 'тестовый объект'
    v.act = 'Тестовая реакция'
    return obj(v)
end
```

> Внимание! При создании объекта конструктор должен опираться только на информацию, полученную из параметров! Дело в том, что создание объекта при загрузке происходит в самом начале, когда окружение мира еще не загружено полностью! В качестве параметров поддерживаются простые типы: строки, таблицы, числа, булевые значения. Недекларированные функции и списки не будут работать.

Если вы хотите уничтожить объект по его имени или ссылке-переменной, воспользуйтесь:

```
purge(o) -- удалить из всех списков
delete(o) -- освободить объект
```

При этом, delete это именно удаление объекта из INSTEAD, а не аналог remove() или purge(). Обычно, нет особого смысла делать delete. Только если предмет больше никогда не понадобится в игре, или вы собираетесь пересоздать объект с тем же именем, имеет смысл освободить его с помощью delete().

Более практически-полезный пример:

```
-- декларируем конструктор
-- path

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

-- динамический переход
-- создали новый объект
-- и положили его в ways()

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

23.15 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 'instead.nosave'.

Например:

instead.nosave = true – запретить сохранения

Если вы хотите запрещать сохранения не везде, а в некоторых сценах, оформите 'instead.nosave' в виде функции, или же меняйте состояние атрибута на лету – он попадает в файл сохранений.

```
-- запретить
-- сохранения в комнатах, которые содержат атрибут nosave.
instead.nosave = function()
    return here().nosave
end
```

Следует отметить, что запрет на сохранения не означает запрета на автосохранение. Для управления автосохранением воспользуйтесь аналогичным атрибутом ‘instead.noautosave’.

> Вы можете явно сохранять игру с помощью вызова: ‘instead.autosave([номер слота])’; Если номер слота не задан, то игра будет сохранена под слотом ‘автосохранение’. Имейте в виду, что сохраняется состояние **после** завершение текущего такта игры.

23.16 Определение типа объекта

В INSTEAD существует два способа определить тип объекта. Первый - с помощью функции std.is_obj(переменная, [тип]).

Например: “‘ a = room { nam = ‘объект’; };

dprint(std.is_obj(a)) – выведет true dprint(std.is_obj(‘объект’)) – выведет false dprint(std.is_obj(a, ‘room’)) – выведет true dprint(std.is_obj(a.obj, ‘list’)) – выведет true “‘

std.is_obj() удобная для определения типа пременной или аргумента функции.

Второй способ связан с использованием метода type:

```
a = room {
    nam = ’объект’;
};

dprint(a:type ’room’) -- выведет true
```

Глава 24

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 - масштабируемая без сглаживания), начиная с версии 2.2.0 доступны дополнительно [4|5|6]: 4 - полностью не масштабируемая (с не масштабируемыми шрифтами), 5 - масштабируемая, с не масштабируемыми шрифтами, 6 - масштабируемая без сглаживания, с не масштабируемыми шрифтами
- 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.icon = пусть к файлу-иконке игры (ОС зависимая опция, может работать некорректно в некоторых случаях)
- 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 scr.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)

- win.ways.mode = top/bottom (задать расположение списка переходов, по умолчанию top – сверху сцены)

- 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. Существуют 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

Кроме этого, новые версии INSTEAD поддерживают механизм множественных тем в одной игре. Давая возможность игроку через стандартное меню INSTEAD выбрать подходящее оформление, из предусмотренных автором игры. Для этого, все темы должны располагаться в игре в подкаталоге themes. В свою очередь, каждая тема – это подкаталог в каталоге themes. В каждом таком подкаталоге должен находится свой файл theme.ini и ресурсы темы (картинки, шрифты, звуки). При этом обязательно наличие темы-каталога themes/default - эта тема будет загружена по умолчанию. Формат файлов theme.ini мы только что рассмотрели. Однако, пути к файлам с ресурсами в theme.ini пишутся не относительно корневого каталога игры, а относительно текущего каталога темы. Это означает, что обычно они содержат только имя самого файла, без пути к каталогу. Например:

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

theme.ini

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

При этом, все игровые темы наследуются от темы themes/default. Поддерживается механизм include. При этом, INSTEAD сначала пытается найти одноименную тему игры, и если такой темы не находится, будет загружена тема из стандартных тем INSTEAD (если она существует). Далее, в theme.ini можно изменять только те параметры, которые требуют изменения.

Глава 25

Модули

Дополнительная функциональность часто реализована в INSTEAD в виде модулей. Для использования модуля необходимо написать:

```
require ”имя модуля”
```

Или:

```
loadmod ”имя модуля”
```

Если модуль поставляется вместе с игрой.

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’.

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

- ‘dbg’ — модуль отладки;
- ‘click’ — модуль перехвата кликов мыши по картинке сцены;
- ‘prefs’ — модуль настроек (хранилище данных настроек);
- ‘snapshots’ — модуль поддержки снапшотов (для откатов игровых ситуаций);

- ‘fmt’ — модуль оформления вывода;
- ‘theme’ — управление темой на лету;
- ‘noinv’ - модуль работы с инвентарем;
- ‘key” - модуль обработки событий срабатывания клавиш;
- ‘timer’ - таймер;
- ‘sprite’ — модуль для работы со спрайтами;
- ‘snd’ — модуль работы со звуком;
- ‘nolife’ – модуль блокировки методов life;

Пример загрузки модулей:

```
--$Name: Моя игра!$
require ”fmt”
require ”click”
```

> Некоторые дополнительные модули, которые не входят в стандартную поставку, вы можете скачать из репозитория модулей[1]. Просто скачайте нужный вам модуль и положите его в каталог с игрой. Включайте такой модуль с помощью loadmod().

25.1 Модуль keys

Вы можете перехватывать события клавиатуры с помощью модуля “keys”.

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

Для задания того, какие именно клавиши необходимо отслеживать, определите функцию keys:filter(press, key). Эта функция должна возвращать true в том случае, если вы хотите отследить данное событие. Например:

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

```
require "keys"

function keys:filter(press, key)
    return press -- ловим нажатия любых клавиш
end
```

В примере мы просто возвращаем параметр press, который равен true для события нажатия клавиши и false – отжатия. В key передается символическое название клавиши (в виде строки).

Обычно, нам нужно выбрать какие именно клавиши мы хотим перехватывать:

```
require "keys"

function keys:filter(press, key)
    if key == '0' or key == '1' or key == 'z' then
        return press -- ловим нажатия клавиш z, 1 и 0
    end
end
```

Итак, keys:filter позволяет выбрать нужные события клавиатуры. А сами события приходят в игру в виде вызова обработчика 'onkey' для текущей комнаты или (если он не задан в комнате) для объекта 'game'.

Обработчик onkey действует как обычный обработчик STEAD3. Он может вернуть false и тогда считается, что клавиша не была обработана игрой. Либо он может выполнить какое-нибудь действие.

Внимание: Если игра будет обрабатывать *все* события клавиш, то даже те комбинации, которые используются самим INSTEAD будут обрабатываться игрой, а не интерпретатором. Например, если игра будет перехватывать клавишу "escape" (и не возвращать false из обработчика), то клавиша "escape" перестанет обрабатываться интерпретатором INSTEAD (escape – вызов меню).

Ниже приводится простой пример вывода на экран символических имен клавиш:

```
require "keys"

function keys:filter(press, key)
    return press -- ловим все нажатия
```

```
end

game.onkey = function(s, press, key)
    dprint(”pressed: ”, key)
    p(”Нажата: ”, key)
    return false -- давать обрабатывать клавиши интерпретатору INSTEAD
end
```

Этот пример можно использовать для того, чтобы выяснить символическое имя конкретной клавиши.

При написании аркадных игр бывает полезным не получать событие от клавиатуры, а опрашивать ее (как правило, в таймере). Для этого вы можете использовать функцию keys:state(имя клавиши).

Эта функция возвращает true для нажатой клавиши и false – для отжатой, например:

```
require ”timer”
require ”keys”

game.timer = function(s) -- показываем состояние клавиши курсор вправо
    dprint(”state of ’right’ key: ”, keys:state ’right’)
    p(”Состояние клавиши ’вправо’:”, keys:state ’right’)
end

timer:set(30)
```

25.2 Модуль click

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:

```
instead.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).

```
require ”click”
function click:filter(press, btn, x, y, px, py)
    dprint(press, btn, x, y, px, py)
    return press and px -- ловим только нажатия на картинку
end
room {
    nam = ’main’;
    pic = ”box:320x200,red”;
    onclick = function(s, press, btn, x, y, px, py)
        pn(”Вы нажали на картинку: ”, px, ”, ”, py)
        pn(”Абсолютные координаты: ”, x, ”, ”, y)
        p(”Кнопка: ”, btn)
    end;
}
```

Внимание! В INSTEAD по умолчанию включен фильтр кликов мыши, который гасит быстрые клики. Это сделано для исключения эффекта дребезга клавиш мыши. В некоторых случаях фильтр может оказаться нежелательным. В таком случае, используйте функцию instead.mouse_filter(), которая может быть использована для определения текущего значения фильтра мыши и установки нового (в том числе - выключения), например:

```
function start()
    dprint(”Mouse filter delay: ”, instead.mouse_filter())
    instead.mouse_filter(0) -- выключили фильтр
end
```

Или так:

```
old_filter = instead.mouse_filter(0) -- выключили
...
instead.mouse_filter(old_filter) -- восстановили
```

25.3 Модуль theme

Модуль theme позволяет изменять параметры темы на лету.

> Имейте в виду, что изменение параметров темы на лету для игр, которые не содержат собственную тему – источник потенциальных проблем! Дело в том, что ваша игра в таком случае должна быть готова работать с любыми разрешениями и параметрами тем, что крайне сложно добиться. Поэтому, если вы собираетесь менять параметры темы из кода – создайте свою тему и включите ее в игру!

При этом, сохранение изменений темы в файле сохранения не поддерживается. Автор игры должен сам восстановить параметры темы в функции start(), как это делается при работе с модулем спрайтов.

Для изменения параметров действующей темы, используются следующие функции:

```
-- настройка окна вывода
theme.win.geom(x, y, w, h)
theme.win.color(fg, link, alink)
theme.win.font(name, size, height)
theme.win.gfx.up(pic, x, y)
theme.win.gfx.down(pic, x, y)

-- настройка инвентаря
theme.inv.geom(x, y, w, h)
theme.inv.color(fg, link, alink)
theme.inv.font(name, size, height)
theme.inv.gfx.up(pic, x, y)
theme.inv.gfx.down(pic, x, y)
theme.inv.mode(mode)

-- настройка меню
theme.menu.bw(w)
theme.menu.color(fg, link, alink)
theme.menu.font(name, size, height)
theme.menu.gfx.button(pic, x, y)

-- настройка графики
theme.gfx.cursor(norm, use, x, y)
theme.gfx.mode(mode)
theme.gfx.pad(pad)
```

```
theme.gfx.bg(bg)

-- настройка звука
theme.snd.click(name);
```

Существует возможность чтения текущих параметров тем:

```
theme.get ’имя переменной темы’;
```

Возвращаемое значение всегда в текстовой форме.

```
theme.set (’имя переменной темы’, значение);
```

Вы можете сбросить значение параметра темы на то, которое было установлено во встроенной теме игры:

```
theme.reset ’имя переменной’;
theme.win.reset();
```

Существует функция, для того, чтобы узнать текущую выбранную тему.

```
theme.name()
```

Функция возвращает строку – имя каталога темы. Если игра использует собственный файл ‘theme.ini’, функция вернет точку. Это удобно, для определения того, включен ли механизм собственных тем игр:

```
if theme.name() ~= ’.’ then
    error ”Please, enable own theme mode in menu!”
end
```

Если в игре используется механизм множественных тем, то имя темы начинается с точки, например:

```
if theme.name() == ’.default’ then
    -- наша встроенная тема default
elseif theme.name() == ’default’ then
    -- стандартная тема default в INSTEAD
end
```

Пример использования:

```
theme.gfx.bg ”dramatic_bg.png”;
theme.win.geom (0,0, theme.get ’scr.w’, theme.get ’scr.h’);
theme.inv.mode ’disabled’
```

Получить размеры текущей темы:

```
theme.scr.w() -- ширина
theme.scr.w() -- высота
```

25.4 Модуль sprite

Модуль sprite позволяет работать с графическими изображениями. Для включения модуля напишите:

```
require ”sprite”
```

Спрайты не могут попасть в файл сохранения, поэтому восстановление состояния спрайтов – задача автора игры. Обычно, для этого используются функции init() и/или start(). start() вызывается после загрузки игры, поэтому в этой функции вы можете использовать переменные игры.

На самом деле в модуле спрайт реализованы два модуля: спрайты и пиксели. Но так как они используются совместно, они размещены в одном модуле. Начнем со спрайтов:

25.4.1 Спрайты

Для создания спрайта используйте метод sprite.new, например:

```
declare ’my_spr’ (sprite.new ’gfx/bird.png’)
local heart = sprite.new ’heart.png’
local blank = sprite.new (320, 200) -- пустой спрайт 320x200
```

У созданного спрайтового объекта существуют следующие методы:

- :alpha(alpha) - создает новый спрайт с заданной прозрачностью alpha (255 - не прозрачно). Это очень медленная функция;

- :dup() - создает копию спрайта;
- :scale(xs, ys, [smooth]) – масштабирует спрайт, для отражений используйте масштаб –1.0 (медленно! не для реального времени). Создает новый спрайт.
- :rotate(angle, [smooth]) – поворот спрайта на заданный угол в градусах (медленно! не для реального времени). Создает новый спрайт.
- :size() – Возвращает ширину и высоту спрайта в пикселях.
- :draw(fx, fy, fw, fh, dst_spr, x, y, [alpha]) – Рисование области src спрайта в область dst спрайта (задание alpha сильно замедляет выполнение функции).
- :draw(dst_spr, x, y, [alpha]) – Рисование спрайта, укороченный вариант; (задание alpha замедляет выполнение функции).
- :copy(fx, fy, fw, fh, dst_spr, x, y) – Копирование прямоугольника fw-на-fh из спрайта в спрайт dst_spr по координатам [x,y] (рисование - замещение). Существует укороченный вариант (как :copy).
- :compose(fx, fy, fw, fh, dst_spr, x, y) – Рисование - с учетом прозрачности обоих спрайтов). Существует укороченный вариант (как :compose).
- :fill(x, y, w, h, [col]) – Заполнение спрайта цветом.
- :fill([col]) – Заполнение спрайта цветом.
- :pixel(x, y, col, [alpha]) – Заполнение пикселя спрайта.
- :pixel(x, y) – Взятие пикселя спрайта (возвращает четыре компонента цвета).
- :colorkey(color) – Задает в спрайте цвет, который выступает в роли прозрачного фона. При этом, при последующем выполнении операции :copy, из рассматриваемого спрайта будут скопированы только те пиксели, цвет которых не совпадает с цветом прозрачного фона.

В качестве "цвета" методы получают строки вида: 'green', 'red', 'yellow' или '#333333', '#2d80ff'…

Пример:

```
local spr = sprite.new(320, 200)
spr:fill 'blue'
local spr2 = sprite.new 'fish.png'
spr2:draw(spr, 0, 0)
```

Кроме того, существует возможность работы с шрифтами. Шрифт создается с помощью sprite.fnt(), например:

```
local font = sprite.fnt('sans.ttf', 32)
```

У созданного объекта определены следующие методы:

- :height() – высота шрифта в пикселях;
- :text(text, col, [style]) – создание спрайта из текста, col - здесь и далее - цвет в текстовом формате (в формате '#rrggbb' или 'текстовое название цвета').
- :size(text) – вычисляет размер, который будет занимать текстовый спрайт, без создания спрайта.

Вам также может пригодиться функция:

```
sprite.font_scaled_size(size)
```

Она возвращает размер шрифта с учетом масштабирование, которое выставил игрок в настройках INSTEAD. Если вы в своей игре хотите учитывать такую настройку, используйте эту функцию для определения размера шрифта.

Пример: `local f = sprite.fnt('sans.ttf', 32) local spr = sprite.new('box:320x200,black') f:text("HELLO!'white'):draw(spr, 0, 0)`

Теперь, рассмотрим варианты применения модуля sprite.

25.4.2 Функция pic

Функция pic может вернуть спрайт. Вы можете формировать каждый раз новый спрайт (что будет не очень эффективно), или можете возвращать заранее выделенный спрайт. Если в такой спрайт вносятся изменения, то эти изменения будут отражены в следующем кадре игры. Так, меняя спрайт по таймеру, можно делать анимацию:

```
require ”sprite”
require ”timer”
local spr = sprite.new(320, 200)

function game:timer()
    local col = { ’red’, ’green’, ’blue’}
    col = col[rnd(3)]
    spr:fill(col)
    return false -- Важно! Так, сцена не будет изменена
end

game.pic = function() return spr end -- функция: так как
-- спрайт это особый объект (не строка)

function start()
    timer:set(30)
end

room {
    nam = ’main’;
    decor = [[ГИПНОЗ!]];
}
```

25.4.3 Отрисовка в фон

Функция sprite.scr() возвращает спрайт - фон. Вы можете выполнять отрисовку в этот спрайт в любом обработчике, например, в таймере. Тем самым добиваясь изменения фона на лету, без применения модуля theme. Например:

```
--$Author: Andrew Lobanov

require ’sprite’
require ’theme’
require ’timer’

declare {
   x = 0,
   y = 0,
```

```
   dx = 10,
   dy = 10,
}

const {
   w = theme.scr.w(),
   h = theme.scr.h(),
}

instead.fading = false

local bg, red, green

function init()
   theme.set('scr.col.bg', '#000000')
   theme.set('win.col.fg', '#aaaaaa')
   theme.set('win.col.link', '#ffaa00')
   theme.set('win.col.alink', '#ffffff')

   bg = sprite.new(w, h)
   bg:fill('black')
   red = sprite.new(w, h)
   red:fill('#ff0000')
   red = red:alpha(128)
   green = sprite.new(w, h)
   green:fill('#00ff00')
   green = green:alpha(64)
   bg:copy(sprite.scr())
   timer:set(25)
end

function game:timer()
   bg:copy(sprite.scr())
   red:draw(sprite.scr(), x, 0, 128)
   green:draw(sprite.scr(), 0, y, 64)
   x = x + dx
   if x >= w or x == 0 then
      dx = -dx
```

```
   end
   y = y + dy
   if y >= h or y == 0 then
      dy = -dy
   end
   return false -- Важно!
end

room {
   nam = 'main',
   disp = 'Test. Test? Test!',
   decor = 'Lorem ipsum';
}
```

Внимание! Интерпретатор INSTEAD в режиме использования предмета на предмет переводит себя в режим "паузы". Это значит, что в тот момент когда выбран предмет из инвентаря (курсор изменил вид на шестеренки) события таймера перестают обрабатываться до тех пор, пока игрок не применит предмет. Это сделано для того, чтобы не разрывать такт игры. Если для вашего творческого замысла такое поведение является помехой (например, вам не нравится тот факт, что анимация фона замирает), вы можете изменить его с помощью вызова:

```
instead.wait_use(false)
```

Как обычно, поместите вызов в init() или start() функцию.

25.4.4 Подстановки

Вы можете создать свой системный объект - подстановку, и формировать графику в выводе игры с помощью img, например:

```
require "sprite"
require "timer"
require "fmt"

obj {
    nam = '$spr';
```

```
    {
        [”квадрат”] = sprite.new ’box:32x32,red’;
    };
    act = function(s, w)
        return fmt.img(s[w])
    end
}

room {
    nam = ’main’;
    decor = [[Сейчас мы вставим спрайт: {$spr|квадрат}.]];
}
```

25.4.5 direct режим

В INSTEAD существует режим прямого доступа к графике. В теме он задается с помощью параметра:

```
scr.gfx.mode = direct
```

Этот параметр можно заранее выставить в theme.ini, или воспользоваться модулем theme. Или (что лучше), специальной функцией:

```
sprite.direct(true)
```

Если режим удалось включить – функция вернет true. sprite.direct() без параметра – возвращает текущий режим (true – если direct включен.)

В этом режиме игра имеет прямой доступ ко всему окну и может выполнять отрисовку в процедуре таймера. Экран представлен специальным спрайтом:

```
sprite.scr()
```

Например:

```
require ”sprite”
require ”timer”
require ”theme”
```

```
sprite.direct(true)

local stars = {}
local w, h
local colors = {
    ”red”,
    ”green”,
    ”blue”,
    ”white”,
    ”yellow”,
    ”cyan”,
    ”gray”,
    ”#002233”,
}
function game:timer()
    local scr = sprite.scr()
    scr:fill ’black’
    for i = 1, #stars do
        local s = stars[i]
        scr:pixel(s.x, s.y, colors[s.dy])
        s.y = s.y + s.dy
        if s.y >= h then
            s.y = 0
            s.x = rnd(w) - 1
            s.dy = rnd(8)
        end
    end
end

function start()
    w, h = theme.scr.w(), theme.scr.h()

    w = std.tonum(w)
    h = std.tonum(h)

    for i = 1, 100 do
        table.insert(stars, { x = rnd(w) - 1, y = rnd(h) - 1, dy = rnd(8) })
    end
```

```
    timer:set(30)
end
```

Еще один пример:

```
require ”timer”
require ”sprite”
require ”theme”

local spr = sprite

declare {
    fnt = false, ball = false, ballw = 0,
    ballh = 0, bg = false, line = false,
    G = false, by = false, bv = false,
    bx = false, t1 = false,
}

function init()
    fnt = spr.fnt(theme.get ’win.fnt.name’, 32);
    ball = fnt:text(”INSTEAD 3.0”, ’white’, 1);
    ballw, ballh = ball:size();
    bg = spr.new ’box:640x480,black’;
    line = spr.new ’box:320x8,lightblue’;
    spr.direct(true)
end

function start()
    timer:set(20)
    G = 9.81
    by = -ballh
    bv = 0
    bx = 320
    t1 = instead.ticks()
end

function phys()
    local t = timer:get() / 1000;
```

```
    bv = bv + G * t;
    by = by + bv * t;
    if by > 400 then
        bv = - bv
    end
end

function game:timer(s)
    local i
    for i = 1, 10 do
        phys()
    end
    if instead.ticks() - t1 >= 20 then
        bg:copy(spr.scr(), 0, 0);
        ball:draw(spr.scr(), (640 - ballw) / 2, by - ballh/2);
        line:draw(spr.scr(), 320/2, 400 + ballh / 2);
        t1 = instead.ticks()
    end
end
```

Внимание! direct режим может быть использован для создания простых аркадных игр. В некоторых случаях, вы можете захотеть убрать указатель мыши. Например, когда игра управляется только с клавиатуры.

Для этого воспользуйтесь функцией instead.mouse_show()

```
instead.mouse_show(false)
```

При этом в меню интерпретатора INSTEAD указатель мыши все еще будет виден.

25.4.6 Использование sprite совместно с модулем theme

В функции start и в обработчиках вы можете менять параметры темы, в том числе, используя в качестве графики спрайты, например:

```
require ”sprite”
require ”theme”
```

```
function start() -- заменим фон на спрайт
    local spr = sprite.new(800, 600)
    spr:fill 'blue'
    spr:fill (100, 100, 32, 60, 'red')
    theme.set('scr.gfx.bg', spr)
end
```

Используя эту технику, вы можете наносить на фоновое изображение статусы, элементы управления или просто менять подложку.

25.4.7 Пиксели

Модуль спрайтов поддерживает также работу с пиксельной графикой. Вы можете создавать объекты – наборы пикселей, модифицировать их и рисовать в спрайты.

Создание пикселей осуществляется функцией pixels.new().

Примеры:

```
local p1 = pixels.new(320, 200) -- создали пиксели 320x200
local p2 = pixels.new 'gfx/apple.png' -- создали пиксели из
-- изображения
local p3 = pixels.new(320, 200, 2) -- создали пиксели 320x200,
-- которые при отрисовке их в спрайт -- будут смасштабированы до
-- 640x400
```

Объект пиксели имеет следующие методы:

> при описании использованы обозначения: r, g, b, a – компоненты пикселя: красный, зеленый, синий, и прозрачность. Все значения от 0 до 255). x, y - координаты левого верхнего угла, w, h – ширина и высота области.

- :size() – вернуть размер и масштаб (как 3 значения);
- :clear(r, g, b, [a]) – быстрая очистка пикселей;
- :fill(r, g, b, [a]) – заливка (с учетом прозрачности);
- :fill(x, y, w, h, r, g, b, [a]) – заливка области (с учетом прозрачности);

- :val(x, y, r, g, b, a) - задать значение пикселя
- :val(x, y) – получить компоненты r, g, b, a
- :pixel(x, y, r, g, b, a) – нарисовать пиксель (с учетом прозрачности существующего пикселя);
- :line(x1, y1, x2, y2, r, g, b, a) – линия;
- :lineAA(x1, y1, x2, y2, r, g, b, a) – линия с AA;
- :circle(x, y, radius, r, g, b, a) – окружность;
- :circleAA(x, y, radius, r, g, b, a) – окружность с AA;
- :poly({x1, y1, x2, y2, …}, r, g, b, a) – полигон;
- :polyAA({x1, y1, x2, y2, …}, r, g, b, a) – полигон с AA;
- :blend(x1, y1, w1, h1, pixels2, x, y) – рисовать область пикселей в другой объект пиксели, полная форма;
- :blend(pixels2, x, y) – короткая форма;
- :fill_circle(x, y, radius, r, g, b, a) – залитый круг;
- :fill_triangle(x1, y1, x2, y2, x3, y3, r, g, b, a) – залитый треугольник;
- :fill_poly({x1, y1, x2, y2, …}, r, g, b, a) – залитый полигон;
- :copy(…) – как blend, но не рисовать, а копировать (быстро);
- :scale(xscale, yscale, [smooth]) – масштабирование в новый объект pixels;
- :rotate(angle, [smooth]) – поворот в новый объект pixels;
- :draw_spr(…) – как draw, но в спрайт, а не пиксели;
- :copy_spr(…) – как copy, но в спрайт, а не пиксели;
- :compose_spr(…) – то же самое, но в режиме compose;
- :dup() – создать копию пикселей;

- :sprite() – создать спрайт из пикселей.

Также, есть возможность работы со шрифтами:

- pixels.fnt(fnt(шрифт.ttf, размер) – создать шрифт;

При этом, у созданного объекта “шрифт” существует метод text:

- :text(текст, цвет(как в спрайтах), стиль) – создать пиксели с текстом;

Например:

```
 local fnt = pixels.fnt(”sans.ttf”, 64)
 local t = fnt:text(”HELLO, INSTEAD!”, ’black’)
 pxl:copy_spr(sprite.scr())
 pxl2:draw_spr(sprite.scr(), 100, 200);
 t:draw_spr(sprite.scr(), 200, 400)
```

Еще один пример (автор примера, Андрей Лобанов):

```
require ”sprite”
require ”timer”

sprite.direct(true)

declare ’pxl’ (false)
declare ’t’ (0)

function game:timer()
   local x, y, i
   t = t + 1
   for x = 0, 199 do
       for y = 0, 149 do
           i = (x * x + y * y + t)
               pxl:val(x, y, 0, i, i / 2)
       end
   end
   pxl:copy_spr(sprite.scr())
end
```

```
function start(load)
   pxl = pixels.new(200, 150, 4)
   timer:set(20)
end
```

При процедурной генерации с помощью pixels удобно использовать шумы Перлина. В INSTEAD существуют функции:

- instead.noise1(x) - 1D шум Перлина;
- instead.noise2(x, y) - 2D шум Перлина;
- instead.noise3(x, y, z) - 3D шум Перлина;
- instead.noise4(x, y, z, w) - 4D шум Перлина;

Все эти функции возвращают значение в диапазоне [−1; 1] а на вход получают координаты с плавающей точкой.

25.5 Модуль snd

Мы уже рассматривали базовые возможности по работе со звуком. Модуль snd имеет еще некоторые функции по работе со звуком.

Вы можете подгрузить звук и держать его в памяти до тех пор, пока он вам нужен.

```
require 'snd'
local wav = snd.new 'bark.ogg'
```

Кроме подгрузки файлов, вы можете загрузить звук из массива lua:

```
local wav = {}
for i = 1, 10000 do
    table.insert(wav, rnd() * 2 - 1) -- случайные значения от -1 до 1
end

function start()
    local p = snd.new(22050, 1, wave) -- частота, число каналов и звук
    p:play()
end
```

Звук задается в нормированном формате: [−1 .. 1]

Звук можно проиграть методом :play([chan], [loop]), где chan – канал (0 - 7), loop - циклы (0 - бесконечность).

Остальные функции модуля:

- snd.stop([channel]) – остановить проигрывание выбранного канала или всех каналов. Вторым параметром можно задавать время затухания звука в мс. при его приглушении;
- snd.playing([channel]) – узнать проигрывается ли звук на любом канале или на выбранном канале; если выбран конкретный канал, функция вернет хандл проигрываемого в данный момент звука или nil. Внимание! Звук клика не учитывается и обычно занимает 0 канал;
- snd.pan(chan, l, r) – задание паннинга. Канал, громкость левого[0–255], громкость правого[0–255] каналов. Необходимо вызывать перед проигрыванием звука, чтобы имело эффект;
- snd.vol(vol) – задание громкости звука (и музыки и эффектов) от 0 до 127.

Еще одна интересная возможность – генерирование звука на лету (пока находится в экспериментальном статусе):

```
require ”snd”

function cb(hz, len, data)
    for i = 1, len do
        data[i] = rnd() * 2 - 1
    end
end
function start()
    snd.music_callback(cb)
end
```

25.6 Модуль prefs

Этот модуль позволяет сохранять настройки игры. Другими словами, сохраненная информация не зависит от состояния игры. Такой механизм можно

использовать, например, для реализации системы достижений или счетчика количества прохождений игры.

По своей сути prefs это объект, все переменные которого будут сохранены.

Сохранить настройки:

```
prefs:store()
```

Настройки сохраняются автоматически при сохранении игры, но вы можете контролировать этот процесс, вызывая prefs:store().

Уничтожить файл с настройками: `prefs:purge()` Загрузка настроек выполняется автоматически при запуске игры (перед вызовом функции start()), но вы можете инициировать загрузку и вручную:

```
prefs:load()
```

Пример использования:

```
-- $Name: Тест модуля prefs$
-- $Version: 0.1$
-- $Author: instead$

-- подключаем модуль click
require ”click”
-- подключаем модуль prefs
require ”prefs”

-- устанавливаем начальное значение счетчика
prefs.counter = 0;

-- определяем функцию отслеживания количества ”кликов”
game.onclick = function(s)
    -- увеличиваем счетчик
    prefs.counter = prefs.counter + 1;
    -- сохраняем счетчик
    prefs:store();
    -- выводим сообщение
    p(”На данный момент сделано ”, prefs.counter ,” кликов”);
end;
```

```
-- добавляем изображение, по которому можно производить клики
game.pic = 'box:320x200,black';

room {
    nam = 'main',
    title = "Комната кликов",
    -- делаем фиксацию статичной части описания
    -- добавляем описание для сцены
    decor = [[ Этот тест был написан специально
            для проверки работы модуля <<prefs>>.
    ]];
};
```

Обратите внимание, что после запуска игры заново, число выполненных кликов не обнулится!

25.7 Модуль snapshots

Модуль snapshots предоставляет возможность восстанавливать предварительно сохраненные состояния игры. В качестве примера, можно привести ситуацию, когда игрок выполняет в игре действие, ведущее к проигрышу. Модуль позволяет автору игры написать код так, что игрок вернется к предварительно сохраненному состоянию игры.

Для создания снапшота используйте функцию: snapshots:make(). В качестве параметра может быть задано имя слота.

Внимание!!! Снапшот будет создан после завершения текущего такта игры, так как только в этом случае гарантирована непротиворечивость сохраненного состояния игры.

Загрузка снапшота осуществляется snapshots:restore(). В качестве параметра может быть задано имя слота.

Удаление снапшота делается с помощью snapshots:remove(). Следует удалять ненужные снапшоты, так как они занимают лишнее место в файлах сохранения.

Пример использования:

```
require "snapshots"
```

```
room {
    nam = ’main’;
    title = ’Игра’;
    onenter = function()
        snapshots:make() -- создали точку восстановления
    end;
    decor = [[{#red|Красное} или {#black|черное}?]];
}: with {
    obj {
        nam = ’#red’;
        act = function()
            p [[Вы выиграли!]]
        end;
    };
    obj {
        nam = ’#black’;
        act = function()
            walk ’end’
        end;
    }
}

room {
    nam = ’end’;
    title = ’Конец’;
}: with {
    obj {
        dsc = [[{Переиграть?}]];
        act = function()
            snapshots:restore() -- восстановились
        end;
    }
}
```

Глава 26

Методы объектов

У всех объектов STEAD3 существуют методы, которые используются при реализации стандартной библиотеке и, обычно, не используются автором игры напрямую. Однако, иногда полезно знать состав этих методов, хотя бы для того, чтобы не называть свои переменные и методы именами уже существующих методов. Ниже представлен список методов с кратким описанием.

26.1 Объект (obj)

- :with({…}) - задание списка obj;
- :new(…) - конструктор;
- :actions(тип, [значение]) - задать/прочитать число событий объекта заданного типа;
- :inroom([{}]) - в какой комнате (комнатах) находится объект;
- :where([{}]) - в каком объекте (объектах) находится объект;
- :purge() - удалить объект из всех списков;
- :remove() - удалить объект из всех объектов/комнат/инвентаря;
- :close()/:open() - закрыть/открыть;
- :disable()/:enable() - выключить/включить;

- :closed() – вернет true, если закрыт;
- :disabled() – вернет true, если выключен;
- :empty() – вернет true, если пуст;
- :save(fp, n) – функция сохранения;
- :display() – функция отображения в сцене;
- :visible() – вернет true если считается видимым;
- :srch(w) – поиск видимого объекта;
- :lookup(w) – поиск любого объекта;
- :for_each(fn, …) – итератор по объектам;
- :lifeon()/:lifeoff() – добавить/удалить из списка живых;
- :live() – вернет true, если в списке живых.

26.2 Комната (room)

Кроме методов obj, добавлены следующие методы:

- :from() – откуда пришли в комнату;
- :visited() – была ли комната посещена ранее?;
- :visits() – число визитов (0 – если не было);
- :scene() – отображение сцены (не объектов);
- :display() – отображение объектов сцены;

26.3 Диалоги (dlg)

Кроме методов room, добавлены следующие методы:

- :push(фраза) - перейти к фразе с запоминанием ее в стеке;
- :reset(фраза) – то же самое, но со сбросом стека;
- :pop([фраза]) – возврат по стеку;
- :select([фраза]) – выбор текущей фразы;
- :ph_display() – отображение выбранной фразы;

26.4 Игровой мир (объект game)

Кроме методов obj, добавлены следующие методы:

- :time([v]) – установить/взять число игровых тактов;
- :lifeon([v])/:lifeoff([v]) – добавить/удалить объект из списка живых, или включить/выключить живой список глобально (если не задан аргумент);
- :live([v]) – проверить активность живого объекта;
- :set_pl(pl) – переключить игрока;
- :life() – итерация живых объектов;
- :step() – такт игры;
- :lastdisp([v]) – установить/взять последний вывод;
- :display(state) – отобразить вывод;
- :lastreact([v]) – установить/взять последнюю реакцию;
- :reaction([v]) – установить/взять текущую реакцию;
- :events(pre, bg) – установить/взять события живых объектов;
- :cmd(cmd) – выполнение команды INSTEAD;

26.5 Игрок (player)

Кроме методов obj, добавлены следующие методы:

- :moved() – игрок сделал перемещение в текущем такте игры;
- :need_scene([v]) – нужна отрисовка сцены в данном такте;
- :inspect(w) – найти объект (видимый) в текущей сцене или себе самом;
- :have(w) – поиск в инвентаре;
- :useit(w) – использовать предмет;
- :useon(w, ww) – использовать предмет на предмет;
- :call(m, …) – вызов метода игрока;
- :action(w) – действие на предмет (act);
- :inventory() – вернуть инвентарь (список, по умолчанию это obj);
- :take(w) – взять объект;
- :walk/walkin/walkout – переходы;
- :go(w) – команда идти (проверяет доступность переходов);

Глава 27

Послесловие

Вот и все, здесь документация заканчивается. Но, возможно, начинается самое интересное – ваша история!

Я сделал первую версию INSTEAD в 2009 году. В тот момент я никогда бы не подумал, что моя игрушка (и движок) переживут столько изменений. Сейчас, когда я пишу это послесловие, на дворе 2017 год и текстовые приключения все еще существуют. Правда, их влияние на культуру по прежнему минимально. И хороших приключений – по прежнему очень мало.

На мой взгляд, у текстографических игр огромный потенциал. Они не такие интерактивные, они не отбирают вашу жизнь взамен на вечно неудовлетворенное желание, не вынуждают вас сутками просиживать за монитором в раздражении или нездоровой нервозности… Они могут взять лучшее из мира литературы и компьютерных игр. И то, что жанр по большей части некоммерческий – даже плюс.

История INSTEAD, на мой взгляд, хорошее тому подтверждение. Было выпущено множество игр, которые можно с уверенностью назвать отличными! Их авторы могут отойти от дел, но созданные ими произведения уже живут своей жизнью, отражаясь в сознании людей, которые в них играют или помнят. Пусть "тираж" этих игр не так велик, но то что я увидел, полностью "оправдало" все потраченные усилия на движок. Я знаю, что это время потрачено не зря. Так что я нашел в себе силы, и сделал движок еще лучше, выпустив STEAD3. Я надеюсь, он понравится и вам.

Так что если вы дочитали до этого места, я могу только пожелать вам дописать вашу первую историю. Творчество – это и есть свобода. :)

Спасибо и удачи. Петр Косых, март 2017.