Справочное пособие по INSTEAD

Александр Яковлев
oreolek@jabber.ru

при участии Петра Косых
gl00my@jabber.ru

8 ноября 2009 г.

Содержание

Введение

Я предполагаю, что вы, читатель, интересуетесь процессом создания игр для движка INSTEAD. Также я думаю, что вы - человек умный и вам не надо повторять дважды.

Игры для движка STEAD пишутся на языке Lua версии 5.1 (то есть, последней на данный момент). Знание этого языка будет очень полезным при написании игр, но я постараюсь сделать описание настолько подробным, что даже новичок в Lua смог программировать без проблем. Между прочим, знающим Lua будет небезынтересно посмотреть код движка.

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

Динамическая часть сцены составлена из описаний объектов сцены, она отображается всегда. Она может выглядеть так: “Стоит стол. Рядом стоит стул.”. Если динамическая часть пуста, то игроку не с чем контактировать в сцене.

Статическая часть сцены описывает саму сцену, её “декорации”. Она отображается при показе сцены (единожды или каждый раз – решает автор игры), или при повторении команды look (в графическом интерпретаторе при щелчке на названии сцены).

Игрок имеет собственный инвентарь. В нём лежат объекты, доступные на любой сцене. Чаще всего инвентарь рассматривают как некую “котомку”, в которой лежат объекты; в этом случае каждый объект считают предметом. Такая трактовка практична, обыденна и интуитивна; но не единственна. Понятие инвентаря является условным, ведь это лишь контейнер. В нём могут находиться такие объекты, как “открыть”, “потрогать”, “лизнуть”. Можно наполнить его объектами “нога”, “рука”, “мозг”. Автор игры свободен в определении этих понятий, но он также должен определить действия игрока над ними.

Действиями игрока могут быть:

- осмотр сцены
- действие на объект сцены
- действие на объект инвентаря
- действие объектом инвентаря на объект сцены
- действие объектом инвентаря на объект инвентаря

Осмотр сцены - это чаще всего неявное действие. Игрок входит в комнату, он автоматически осматривает её.

Действие на объект сцены обычно понимается как изучение объекта, или использование его. Например, если в сцене существует объект “чашка кофе”, то действием на него может быть выпивание кофе, тщательный осмотр чашки, разбивание чашки или перемещение чашки в инвентарь. Это определяется только автором и никем другим.

Действие на объект инвентаря понимается аналогично. Например, если в инвентаре лежит объект “яблоко”, его можно съесть или осмотреть. С другой стороны, если в инвентаре лежит объект “осмотреть”, то действие над ним будет трудно описать логически.

Действие объектом инвентаря на объект сцены – это чаще всего использование или передача объекта. Например, действие объектом “нож” на объект “бармен” может означать передачу ножа бармену, угрозу ножом бармену, убийство ножом бармена и многое другое.

Действие объектом инвентаря на объект инвентаря понимается так же свободно. Это может быть соединение предметов (“сарделька” + “кетчуп”) в одно (“сарделька с кетчупом”), либо использование (“открыть” + “ящик”).

Эти примеры подробно показывают первую из идей STEAD - гибкость. Автор свободен в своей фантазии и может трактовать все понятия движка как хочет.

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

Игра начинается именно с main.lua. В начале файла main.lua может быть определён заголовок, состоящий из тегов. Теги должны начинаться с символов комментария --. На данный момент существует один тег: $Name:, который должен содержать название игры. Пример использования тега:

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

Интерпретатор ищет доступные игры в каталогах:
Unix версия интерпретатора просматривает игры в каталогах:
/usr/local/share/instead/games (по умолчанию),
~/.instead/games.
WinXP версия:
Documents and Settings/USER/Local Settings/Application Data/instead/games
WinVista: Users\USER\AppData\Local\instead\games
Все Windows: куда-вы-установили-INSTEAD/games

В дальнейшем я буду отталкиваться от возможностей графической ветки интерпретатора – INSTEAD-SDL.

1 Сцена

Сцена – это единица игры, в рамках которой игрок может изучать все объекты сцены и взаимодействовать с ними. В игре должна быть хотя бы одна сцена с именем main.

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

Отмечу, что пример выше является минимальной игрой для INSTEAD. Это некий "Hello, World", который я рекомендую сохранить под именем main.lua и поместить в отдельную папку в каталоге для игр.

Атрибут nam (имя) является необходимым для любого объекта. Для сцены это – то, что будет заголовком сцены при её отображении. Имя сцены также используется для её идентификации при переходах.

Атрибут dsc – это описание статической части сцены, которое выводится при входе в сцену или выполнении команды look.

Внимание!!! Если для вашего творческого замысла необходимо, чтобы описание статической части сцены выводилось на каждом ходу (а не только при первом входе в сцену), вы можете определить для своей игры параметр forcedsc (в начале игры).

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

Или, аналогично, задать атрибут forcedsc для конкретных сцен.

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.]],
```

К текущей сцене можно обратиться через функцию here().

2 Объект

2.1 Нормальные объекты

Объекты – это единицы сцены, с которыми взаимодействует игрок.

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

Имя объекта `nam` используется для адресации объекта.

`dsc` – описатель объекта. Он будет выведен в динамической части сцены. Фигурными скобками отображается фрагмент текста, который будет являться ссылкой в графическом интерпретаторе. Если вы забудете сделать ссылку, то интерпретатор не выдаст ошибки, но игроки не смогут взаимодействовать с объектом.

`act` – это обработчик, который вызывается при действии пользователя на объект сцены. Если объект находится в инвентаре, то действие с ним будет передаваться другому обработчику – `inv`.

Настало время сделать небольшое отступление. До сих пор в примерах приводились примитивные обработчики, которые всего лишь возвращают определённую строку. В примере выше обращение к объекту вызовет банальную реакцию: интерпретатор напечатает строку "Гм... Просто стол...". Хуже того: он будет отвечать тем же образом каждый раз при обращении к объекту. Это не совсем гибкий подход, поэтому STEAD позволяет определить любой атрибут объекта как функцию. Так, возможно построить такую конструкцию:

```
apple = obj {
        nam = 'яблоко',
        dsc = function(s)
                if not s._seen then
                        return 'На столе {что-то} лежит.';
                else
                        return 'На столе лежит {яблоко}.';
                end
        end,
        act = function(s)
                if s._seen then
                        return 'Это яблоко!';
                else
                        s._seen = true;
                        return 'Я присматриваюсь и понимаю,что это - яблоко.!';
                end
        end,
};
```

Если атрибут или обработчик оформлен как функция, то обычно первый аргумент функции (s) сам объект. В данном примере, при показе сцены будет в динамической части сцены будет текст: "На столе что-то лежит". При взаимодействии с "что-то", переменная `_seen` объекта `apple` будет установлена в `true`, и мы увидим, что это было яблоко.

Запись `s._seen` означает, что переменная `_seen` размещена в объекте `s` (то есть, `apple`). В языке Lua переменные необязательно объявлять заранее, при первом обращении к ней переменная `apple._seen` появится сама; но хорошим тоном будет заранее **проинициализировать** переменную со значением `false`.

Подчёркивание в имени переменной означает, что она **попадёт** в файл сохранения игры. Сохраняются все переменные, название которых начинается с большой буквы или с подчёркивания. Вы можете переопределить функцию `isForSave(k)`, если вас это не устраивает.

Внимание!!! Переменные в любом случае не записываются в файл сохранения, если они не размещены в одном из перечисленных типов объектов: комната, объект, диалог, игра, игрок.

2.2 Облегчённые объекты

Иногда, сцену нужно наполнить декорациями, которые обладают ограниченной функциональностью, но делают игру разнообразней. Для этого можно использовать облегчённый объект. Например:

```
sside = room {
        nam = 'южная сторона',
        dsc = [[Я нахожусь у южной стены здания института. ]],
        act = function(s, w)
                if w == 1 then
                        ways():add('stolcorridor');
                        return "Я подошёл к подъезду. Хм -- зайти внутрь?";
                end
        end,
        obj = {vobj(1, "подъезд", "У восточного угла находится небольшой {подъезд}.")},
};
```

Как видим, `vobj` позволяет сделать лёгкую версию статического объекта, с которым тем не менее можно взаимодействовать (за счёт определения обработчика `act` в сцене и анализа ключа объекта). `vobj` также вызывает метод `used`, при этом в качестве третьего параметра передаётся объект, воздействующий на виртуальный объект.

Синтаксис `vobj`: `vobj(ключ, имя, описатель);` где ключ – это цифра, которая будет передана обработчикам `act/used` сцены как второй параметр.

Существует модификация объекта `vobj` – `vway`. `vway` реализует ссылку. Синтаксис и пример:

```
vway(имя, описание, сцена назначения);
obj = { vway("дальше", "Нажмите {здесь}.", 'nextroom') }
```

Вы можете динамически заполнять сцену объектами `vobj` или `vway` с помощью методов `add` и `del`.

В довершение, определена также упрощённая сцена `vroom`. Синтаксис:

```
vroom(имя перехода, сцена назначения)
```

Ниже приводится несколько примеров и трюков с подобными объектами:

```
home.objs:add(vway("next", "{Дальше}.", 'next_room');
home.objs:del("next");
home.objs:add(vroom("идти на запад", 'mountains');
if not home.obj:srch('Дорога') then
        home.obj:add(vway('Дорога', 'Я заметил {дорогу}, ведущую в лес...', 'forest'));
end
obj = {vway('Дорога', 'Я заметил {дорогу}, ведущую в лес...', 'forest'):disable()},
objs()[1]:disable();
objs()[1]:enable();
```

2.3 Динамическое создание объектов

Вы можете использовать аллокаторы `new` и `delete` для создания и удаление динамических объектов:

```
new("obj { nam = 'a' ..... }")
put(new [[obj {nam = 'test' } ]]);
put(new('myconstructor()');
```

Созданный объект будет попадать в файл сохранения. `new()` возвращает реальный объект;чтобы получить его имя,если это нужно, используйте функцию`deref`.

3 Некоторые манипуляции с объектами

3.1 Объект и сцена

Ссылкой на объект называется текстовая строка, содержащая имя объекта при его создании. Например: `'table'` – ссылка на объект `table`.

Для того, чтобы поместить в сцену объекты, нужно определить массив `obj`, состоящий из ссылок на объекты:

```
main = room {
        nam = 'главная комната',
        dsc = 'Вы в большой комнате.',
        obj = { 'tabl' },
};
```

3.2 Объекты, связанные с объектами

Объекты тоже могут содержать атрибут `obj`. При этом, список будет последовательно разворачиваться. Например, поместим на стол яблоко:

```
apple = obj {
        nam = 'яблоко',
        dsc = 'На столе лежит {яблоко}.',
};

table = obj {
        nam = 'стол',
        dsc = 'В комнате стоит {стол}.',
        obj = { 'apple' },
};
```

При этом, в описании сцены мы увидим описание объектов “стол” и “яблоко”, так как `apple` – связанный с `table` объект.

3.3 Действия объектов друг на друга

Игрок может действовать объектом инвентаря на другие объекты. При этом вызывается обработчик `use` у объекта которым действуют и `used` – на которого действуют.

Например:

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

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

Если игрок возьмёт нож и использует его на стол – то увидит текст обработчиков `knife.use` и `tabl.used`.

`use` и `used` могут быть функциями. Тогда первый параметр это сам объект, а второй – ссылка на объект, на который направлено действие в случае `use` и объект, которым действие осуществляется в случае `used`.

`use` может вернуть статус `false`, в этом случае обработчик `used` не вызывается (если он вообще был). Статус обработчика `used` – игнорируется. Это будет выглядеть как:

```
return 'Строка реакции', false;
```

Возможно также действовать объектами сцены на объекты сцены; для этого нужно установить переменную `game.scene_use = true` или поставить `scene.use=true` в нужной комнате. В этом случае использование объектов сцены будет аналогично использованию объектов инвентаря.

4 Смена сцен

Как только главный герой уходит со сцены, декорации меняются. Но чтобы игрок ушёл из нашей сцены, он должен знать, куда идти.

Для перехода между сценами используется атрибут сцены – список `way`.

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

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

При этом, вы сможете переходить между сценами `main` и `room2`. Как вы помните, `nam` может быть функцией, и вы можете генерировать имена сцен на лету, например, если вы хотите, чтобы игрок не знал название сцены, пока не попал на неё.

При переходе между сценами движок вызывает обработчик `exit` из текущей сцены и `enter` той сцены, куда идёт игрок. `exit` и `enter` могут быть функциями – тогда первый параметр это сам объект, а второй – ссылка на комнату куда игрок хочет идти (для `exit`) или из которой уходит (для `enter`). Например:

```
room2 = room {
        enter = function(s, f)
                if f == 'main' then
                        return 'Вы пришли из комнаты.';
                end
        end,
        nam = 'зал',
        dsc = 'Вы в огромном зале.',
        way = { 'main' },
        exit = function(s, t)
                if t == 'main' then
                        return 'Я не хочу назад!', false
                end
        end,
};
```

Как видим, обработчики могут возвращать два значения: строку и статус. В нашем примере функция `exit` вернёт `false`, если игрок попытается уйти из зала в `main`. `false` блокирует переход. Такая же логика работает и для `enter`. Кроме того, она работает и для обработчика `tak` (о нём чуть позже).

Важное замечание: обработчик enter вызывается не при входе в комнату, а чуть раньше. Когда игрок хочет уйти в комнату, движок вызывает её обработчик enter, чтобы убедиться в том,что тот не возвращает false и переход возможен. Поэтому если вы используете `here()` в `enter`, она будет указывать прежде всего – на предыдущую комнату. Если вы передаёте enter параметр сцены, то он будет указывать всегда на текущую сцену.

Если требуется перейти на другую сцену автоматически, можно использовать функцию `goto` со ссылкой на сцену как параметром:

```
return goto('main');
```

Почему здесь написано `return`? Дело в том,что функция `goto()` не является безусловным переходом,как можно подумать. Она возвращает описание новой сцены, поэтому почти всегда должна завершать работу обработчика таким нехитрым способом.

НО: Если вы выполните goto из обработчика exit, то получите переполнение стека, так как goto снова и снова будет вызывать метод exit. Вы можете избавиться от этого, если вставите проверку, разрушающую рекурсию. Например:

```
exit = function(s, t)
        if t == 'dialog' then return; end
        return goto('dialog');
```

Обычно движок сам пытается разорвать рекурсию. Тем не менее, автор должен понимать, что он делает и не надеяться на автоматику.

5 Специальные типы объектов

5.1 Инвентарь

Инвентарь проще всего возвращается функцией inv(). Он представлен списком, поэтому для него справедливы все их трюки (см. соответствующий раздел).

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

Если предмет сцены имеет обработчик `tak` и НЕ имеет обработчика `act` [1], то при действии на нём вызывается не `act`, а `tak`; после этого предмет перемещается в инвентарь. Это происходит вот так:

```
apple = obj {
        nam = 'яблоко',
        dsc = 'На столе лежит {яблоко}.',
        tak = 'Вы взяли яблоко.',
};
```

5.2 Игрок

Игрок в STEAD представлен объектом `pl`. Тип объекта – `player`.

Атрибут `obj` представляет собой инвентарь игрока.

[1] Пользуясь случаем: я считаю,что tak - немного неудобное имя. Именно так. - А.Я.

5.3 Игра

Игра представлена объектом game. Он хранит в себе указатель на текущего игрока ('pl') и некоторые параметры. Например, вы можете указать в начале своей игры кодировку текста следующим образом:

```
game.codepage="UTF-8";
```

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

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

На практике полезно что-то вроде:

```
game.inv = function()
 local a = rnd(7);
 local reaction = {
  [1] = 'Либо я ошибся карманом, либо мне нужно что-то другое.',
  [2] = 'Откуда у меня в кармане ЭТО?!',
  [3] = 'Сам не понял,что достал. Положу обратно.',
  [4] = 'Это что-то неправильное.',
  [5] = 'В моих карманах что только не залёживается...',
  [6] = 'Я не представляю, как я могу тащить ЭТО с собою.',
  [7] = 'Мне показалось или оно на меня смотрит?',
 };
 return reaction[a];
end;
```

6 Диалоги

Третьим важным типом в движке являются диалоги. Диалоги – это особый подвид сцен, содержащий только фразы. Например, диалог может выглядеть следующим образом:

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

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

_phr – создание выключенной фразы. Она не видна изначально, но её можно включить с помощью функции pon() (см. ниже).

Вот как создаётся фраза:

```
[1] = phr('Мне вот-этих зелёненьких... Ага -- и бобов!', 'На здоровье!', [[pon(1);]]),
```

В реакции может быть любой lua код (простейшей реакцией является возвращение строки), но в STEAD определены наиболее часто используемые функции:

pon(n...) включить фразы диалога с номерами n... (в нашем примере – чтобы игрок мог повторно взять еду того-же вида).

poff(n...) выключить фразы диалога с номерами n...

prem(n...) удалить (заблокировать) фразы диалога с номерами n... (удаление означает невозможность включения фраз. pon(n..) не приведёт к включению фраз).

Как ответ,так и реакция могут быть функциями.

Переход в диалог осуществляется как переход на сцену:

```
return goto('povardlg');
```

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

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

Вы можете включать/выключать фразы не только текущего, но и произвольного диалога, с помощью методов объекта диалог `pon/poff`. Например:

```
shopman:pon(5);
```

Если номер фразы не указан,то это означает,что действие относится к текущей фразе:

```
phr('a', 'b', [[ pon() ]]);
```

7 О списках

Каждый атрибут-список имеет методы:

add Добавление элемента в список. Необязательным вторым параметром является позиция в списке.

del Удаление элемента из списка.

look Получение индекса элемента в списке по идентификатору

srch Проверка на наличие объекта в списке по его nam. Возвращает 2 значения: идентификатор и позицию; если объекта нет, вернёт nil. Например:

```
objs():srch('Ножик')
```

set Изменение объекта по номеру. Например, этот пример присвоит заменит первый объект списка:

```
objs():set('knife',1);
```

disable Скрытие объекта в списке; отличается от удаления тем, что может быть возвращён к жизни через метод enable

enable Показ скрытого объекта

zap Обнуление списка

cat(b) Склеивает список со списком b

cat(b,pos) Добавляет в список список b на позицию pos

disable_all Аналогично disable, но массово

enable_all Смысл понятен.

Параметрами методов могут быть объекты, их идентификаторы и имена.

8 Функции

inv() возвращает список инвентаря

objs() возвращает список объектов указанной сцены; если сцена не указана, то возвращает список объектов текущей сцены.

ways() возвращает список возможных переходов из указанной сцены; если сцена не указана, то возвращает список возможных переходов из текущей сцены.

me() возвращает объект pl (объект игрока)

here() возвращает текущую сцену

where() возвращает текущую сцену как строку-имя объекта, а не сам объект

from() возвращает объект прошлой сцены

ref(nam) возвращает ссылку на указанный объект:

```
ref('home') == home
```

deref(object) возвращает ссылку строкой для объекта:

```
deref(knife) == 'knife'
```

have(object) проверяет, есть ли объект в инвентаре по имени объекта или по его nam

move(from, where) переносит объект из текущей сцены в другую

movef(from, where) действует так же, но добавляет объект в начало списка

seen(object,scene) проверяет, есть ли объект в указанной сцене (в текущей, если сцена не указана)

drop(object,scene) выбрасывает объект из инвентаря в указанную сцену (в текущую, если сцена не указана)

dropf(object,scene) то же, что drop, но объект появляется в начале списка

put(object,scene) кладёт предмет в указанную сцену; в текущую, если сцена не указана

remove(object,scene) удаляет предмет из указанной сцены; из текущей, если сцена не указана

take(object,scene) перемещает объект из указанной(текущей) сцены в инвентарь

taken(object) проверяет, взят ли уже объект

rnd(m) возвращает случайное целое значение от 1 до m

goto(destination) переносит в сцену w; используется в конструкции вида `return goto('inmycar');`

change_pl(player) переключает на другого игрока (со своим инвентарём и позицией), также используется в `return`. Может использоваться для переключения между разными инвентарями.

back() переносит в предыдущую сцену (аналогично goto)

time() возвращает текущее время игры в активных действиях.

cat(...) возвращает строку – склейку строк-аргументов. Если первый аргумент nil, то функция возвращает nil.[2]

[2]Функция легко и просто заменяется обычным оператором склейки строк Lua: `строка1 .. строка2`. Тем не менее, этот оператор выдаст ошибку при склейке nil

par(...) возвращает строку – склейку строк-аргументов, разбитых строкой-первым параметром.

Следующие записи эквивалентны:

```
ref('home').obj:add('chair');
home.obj:add('chair');
objs('home'):add('chair');
put('chair', 'home');
```

Если вы хотите перенести объект из произвольной сцены, вам придётся удалить его из старой сцены с помощью метода del. Для создания сложно перемещающихся объектов, вам придётся написать свой метод, который будет сохранять текущую позицию объекта в самом объекте и делать удаление объекта из старой сцены. Вы можете указать исходную позицию (комнату) объекта в качестве третьего параметра move.

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

Отдельно следует упомянуть функции альтернативного вывода текста. Их назначение можно понять по следующим формам записи:

```
act = function(s)
       p "Я осмотрел его...';
       p "Гммм...."
end

act = function(s)
       return "Я осмотрел его... Гммм...."
end
```

p Добавляет строку и пробел в буфер

pn Добавляет строку и перевод строки в буфер

pclr Очистка буфера

pget Получение содержимое буфера на текущий момент

Другой пример:

```
life = function(s)
     p 'Ветер дует мне в спину.'
     return pget(), true
end
```

9 Добавление динамики в игру

Игра измеряет время в своих единицах - в шагах, или активных действиях. Каждое действие игрока - это его шаг, пусть даже он тратит его на то,чтобы ещё раз осмотреться. Что он увидит нового? Что изменится в мире игры за это время?

Именно для того,чтобы задать динамику мира, существует система с говорящим названием life.

Вы можете определять обработчики, которые выполняются каждый раз, когда время игры увеличивается на 1. Например:

```
mycat = obj {
        nam = 'Барсик',
        lf = {
                [1] = 'Барсик шевелится у меня за пазухой.',
                [2] = 'Барсик выглядывает из за пазухи.',
                [3] = 'Барсик мурлычет 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(6);
                if r > 2 then
                        return;
                end
                r = rnd(6);
                return s.lf[r];
        end,
```

В этом примере кот по имени Барсик, сидя в инвентаре у игрока, будет на каждом шагу показывать свою активность. Но приведённый код пока что не будет работать. Для того,чтобы объявить объект "живым", следует добавить его в соответствующий список с помощью функции `lifeon()`.

```
inv():add('mycat');
lifeon('mycat');
```

Любой объект или сцена могут иметь свой обработчик `life`, который вызывается каждый раз при смене текущего времени игры, если объект или сцена были добавлены 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.

Вы можете вернуть из обработчика life второй код возврата, важность. (true или false). Если он равен true, то возвращаемое значение будет выведено ДО описания объектов; по умолчанию значение равно false.

Если вы хотите "очистить экран", то можно воспользоваться вот каким хаком. Из метода `life` доступна переменная `ACTION_TEXT` – это тот текст, который содержит реакцию на действие игрока. Соответственно, сделав `ACTION_TEXT = nil`,можно "запретить" вывод реакции. Например,для перехода на конец игры можно сделать:

```
ACTION_TEXT = nil
return goto('theend'), true
```

10 Краски и звуки

В чём очевидное преимущество графического интерпретатора над текстовой веткой – это то, что он может говорить и показывать. Проще говоря, вы можете добавить в игру графику и музыку.

Графический интерпретатор анализирует атрибут сцены `pic`, и воспринимает его как путь к картинке-иллюстрации для сцены. Если в текущей сцене не определён атрибут `pic`, то берётся `game.pic`. Если не определён и он, то картинка не отображается.

Unix-версия движка поставляется в исходных кодах, поэтому поддержка форматов графики и музыки определяется на стадии его компиляции; если при сборке в системе присутствуют нужные библиотеки для SDL, то формат будет поддерживаться. Я рекомендую использовать форматы PNG, JPG и GIF. Новые версии движка поддерживают GIF-анимацию. Те же правила действуют для музыки. Здесь строгих рамок нет, поэтому старайтесь не использовать редких форматов. Проверена поддержка WAV, MP3, OGG, FLAC, MIDI, XM, MOD, IT, S3M.

Вы также можете встраивать графические изображения в текст или в инвентарь с помощью функции `img`. Например:

```
knife = obj {
        nam = 'Нож'..img('img/knife.png'),
}
```

Теперь о звуке. Фоновая музыка задаётся с помощью функции:

```
set_music(имя музыкального файла, количество проигрываний)
```

Она проигрывается циклически бесконечно,если количество проигрываний не задано.

`get_music()` возвращает текущее имя трека.

Функция `get_music_loop` возвращает, на каком витке повторения находится мелодия. 0 - означает вечный цикл. n – количество проигрываний. -1 – проигрывание текущего трека закончено.

Помимо фонового сопровождения, `set_sound()` позволяет проиграть звуковой файл. `get_sound()` возвращает имя звукового файла, который будет проигран.

11 Трюки

11.1 Форматирование

SDL-INSTEAD поддерживает простое форматирование текста с помощью функций:

txtc(текст) – разместить текст по центру

txtr(текст) – разместить текст справа

txtl(текст) – разместить слева

txtb(текст) – полужирное начертание

txtem(текст) – начертание курсивом

txtu(текст) – подчёркнутый текст

11.2 Проверка правописания

Проверка правописания готовой игры - это большая головная боль. У вас есть примерно 100 Кб кода, в которых находятся около 80 Кб текста. Любая программа проверки орфографии будет сильно ругаться на синтаксис Lua и мешать. Один из способов проверки – использовать редактор Emacs.

Для проверки нужно установить сам Emacs и поддержку Lua к нему (lua-mode); дальнейшие операции с редактором:

1. Открыть нужный файл
2. Если русские буквы выглядят кракозябрами – выбираете в меню Options – Set Font/FontSet... шрифт fixed
3. Tools – Spell Checking – Select Russian Dict
4. Tools – Spell Checking – Spell-Check Buffer
5. Пробел для того,чтобы пропустить слово; Английская a,чтобы игнорировать слово вообще; i чтобы добавить слово в словарь пользователя; цифры, чтобы заменить слово на один предложенных вариантов.

Если вы впервые видите этот редактор, я настоятельно НЕ рекомендую нажимать что-нибудь и щёлкать на что-нибудь непонятное. Будет только непонятнее.

11.3 Меню

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

```
pocket = menu {
        State = false,
        nam = function(s)
                if s.State then
                        return txtu('карман');
                end
                return 'карман';
        end,
        gen = function(s)
                if s.State then
                        s:enable_all();
                else
                        s:disable_all();
                end
        end,
        menu = function(s)
                if s.State then
                        s.State = false;
                else
                        s.State = true;
                end
                s:gen();
        end,
};

knife = obj {
        nam = 'нож',
        inv = 'Это нож',
};

inv():add(pocket);
put(knife, pocket);
pocket:gen();

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

11.4 Статус

Ниже представлена реализация статуса игрока в виде текста, который появляется в инвентаре, но не может быть выбран.

```
pl.Life = 10;
pl.Power = 10;

status = obj {
        nam = 'Жизнь: '..pl.Life..',Сила: '..pl.Power,
};
```

```
inv():add('status');
status.object_type = false
```

Вы можете использовать конструктор `stat` для создания статуса:

```
status = stat {
       nam = function(s)
               return 'Статус!!!';
       end
};

inv():add('status');
```

11.5 Кодирование исходного кода

Если вы не хотите показывать исходный код своих игр, вы можете закодировать его с помощью команды

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

и использовать его с помощью lua функции `doencfile(путь к файлу)`.

Важное замечание: Лучше не использовать компиляцию игр с помощью luac, так как luac создаёт платформозависимый код. Таким образом, вам придётся выдавать сразу две скомпилированных версии: для 32-битных и 64-битных машин[3]. Однако, компиляция игр может быть использована для поиска ошибок в коде.

11.6 Создание собственного плейлиста

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

```
tracks = {"mus/astro2.mod", "mus/aws_chas.xm", "mus/dmageofd.xm", "mus/doomsday.s3m"}
mplayer = obj {
        nam = 'плеер',
        life = function(s)
                local n = get_music();
                local v = get_music_loop();
                if not n or not v then
                        set_music(tracks[2], 1);
                elseif v == -1 then
                        local n = get_music();
                        while get_music() == n do
                                n = tracks[rnd(4)]
                        end
                        set_music(n, 1);
                end
        end,
};
lifeon('mplayer');
```

[3] А в будущем, возможно, SDL-INSTEAD будет поддерживать больше платформ, поэтому использовать предложенный метод будет ещё выгоднее

11.7 Отладка

Для того, чтобы во время ошибки увидеть стек вызовов функций lua, вы можете запустить

```
sdl-instead -debug
```

Вы можете отлаживать свою игру вообще без instead. Например, вы можете создать следующий файл game.lua:

```
dofile("/usr/share/games/stead/stead.lua"); -- путь к stead.lua
dofile("main.lua"); -- ваша игра
game:ini();
iface:shell();
```

И запустите игру в lua: `lua game.lua`. При этом игра будет работать в примитивном shell окружении. Полезные команды: ls, go, act, use.... Теоретически движок можно таким образом привязать даже к CGI окружению.

12 Создание тем для SDL-INSTEAD

Графический интерпретатор поддерживает механизм тем.

Тема – это инструкция к оформлению игры. Она задаёт внешний вид и положения всех информационных блоков на экране.

Тема представляет из себя каталог, с файлом `theme.ini` внутри. Файл `theme.ini` здесь является ключевым.

Тема, которая является минимально необходимой – это тема `default`. Эта тема всегда загружается первой. Все остальные темы наследуются от неё и могут частично или полностью заменять её 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>
; комментарий
```

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

Цвет задаётся в форме `#rgb`, где `r`, `g` и `b` — компоненты цвета в шестнадцатеричном виде. Кроме того некоторые основные цвета распознаются по своим именам:

параметр		параметр		параметр			
aliceblue	T.	forestgreen	T.	mediumvioletred	T.	violet	T.
antiquewhite	T.	fuchsia	T.	midnightblue	T.	violetred	T.
aqua	T.	gainsboro	T.	mintcream	T.	wheat	T.
aquamarine	T.	ghostwhite	T.	mistyrose	T.	white	T.
azure	T.	gold	T.	moccasin	T.	whitesmoke	T.
beige	T.	goldenrod	T.	navajowhite	T.	yellow	T.
bisque	T.	gray	T.	navy	T.	yellowgreen	T.
black		green	T.	oldlace	T.		
blanchedalmond	T.	greenyellow	T.	olive	T.		
blue	T.	honeydew	T.	olivedrab	T.		
blueviolet	T.	hotpink	T.	orange	T.		
brown	T.	indianred	T.	orangered	T.		
burlywood	T.	indigo	T.	orchid	T.		
cadetblue	T.	ivory	T.	palegoldenrod	T.		
chartreuse	T.	lavender	T.	palegreen	T.		
chocolate	T.	lavenderblush	T.	paleturquoise	T.		
coral	T.	lawngreen	T.	palevioletred	T.		
cornflowerblue	T.	lemonchiffon	T.	papayawhip	T.		
cornsilk	T.	lightblue	T.	peachpuff	T.		
crimson	T.	lightcoral	T.	peru	T.		
cyan	T.	lightcyan	T.	pink	T.		
darkblue	T.	lightgoldenrodyellow	T.	plum	T.		
darkcyan	T.	lightgrey	T.	powderblue	T.		
darkgoldenrod	T.	lightgreen	T.	purple	T.		
darkgray	T.	lightpink	T.	red	T.		
darkgreen	T.	lightsalmon	T.	rosybrown	T.		
darkkhaki	T.	lightseagreen	T.	royalblue	T.		
darkmagenta	T.	lightskyblue	T.	saddlebrown	T.		
darkolivegreen	T.	lightslateblue	T.	salmon	T.		
darkorange	T.	lightslategray	T.	sandybrown	T.		
darkorchid	T.	lightsteelblue	T.	seagreen	T.		
darkred	T.	lightyellow	T.	seashell	T.		
darksalmon	T.	lime	T.	sienna	T.		
darkseagreen	T.	limegreen	T.	silver	T.		
darkslateblue	T.	linen	T.	skyblue	T.		
darkslategray	T.	magenta	T.	slateblue	T.		
darkturquoise	T.	maroon	T.	slategray	T.		
darkviolet	T.	mediumaquamarine	T.	snow	T.		
deeppink	T.	mediumblue	T.	springgreen	T.		
deepskyblue	T.	mediumorchid	T.	steelblue	T.		
dimgray	T.	mediumpurple	T.	tan	T.		
dodgerblue	T.	mediumseagreen	T.	teal	T.		
feldspar	T.	mediumslateblue	T.	thistle	T.		
firebrick	T.	mediumspringgreen	T.	tomato	T.		
floralwhite	T.	mediumturquoise	T.	turquoise	T.		

12.1 Параметры окна изображений

Окно изображений — область, в которой располагается картинка сцены. Интерпретация зависит от режима расположения.

Для окна изображений заданы следующие параметры:

параметр	тип	описание
`scr.w`	число	ширина игрового пространства, пиксели
`scr.h`	число	высота игрового пространства, пиксели
`scr.col.bg`	цвет	цвет фона
`scr.gfx.bg`	строка	путь к файлу фонового изображения
`scr.gfx.cursor.x`	число	абсцисса центра курсора, пиксели
`scr.gfx.cursor.y`	число	ордината центра курсора, пиксели
`scr.gfx.cursor.normal`	строка	путь к картинке-курсору
`scr.gfx.cursor.use`	строка	путь к картинке-курсору режима использования
`scr.gfx.use`	строка	путь к картинке-индикатору режима использования
`scr.gfx.pad`	число	размер отступов к скролл-барам и краям меню, пиксели
`scr.gfx.x`	число	абсцисса окна изображений, пиксели
`scr.gfx.y`	число	ордината окна изображений, пиксели
`scr.gfx.w`	число	ширина окна изображений, пиксели
`scr.gfx.h`	число	высота окна изображений, пиксели
`win.gfx.h`	число	синоним `scr.gfx.h`
`scr.gfx.mode`	строка	режим расположения

Параметр `scr.gfx.mode` может принимать одно из значений: `fixed`, `embedded` или `float`.

В режиме `embedded` картинка является частью содержимого главного окна, параметры главного окна (см. ниже) `win.x`, `win.y`, `win.w` игнорируются.

В режиме `float` картинка расположена по указанным координатам (`win.x`, `win.y`) и масштабируется к размеру `win.w` x `win.h` если превышает его.

В режиме `fixed` – картинка является частью сцены как в режиме `embedded`, но не скроллируется вместе с текстом, а расположена непосредственно над ним.

12.2 Параметры главного окна

Главное окно — область, в которой располагается описание сцены. Для главного окна заданы следующие параметры:

параметр	тип	описание
`win.x`	число	абсцисса главного окна, пиксели
`win.y`	число	ордината главного окна, пиксели
`win.w`	число	ширина главного окна, пиксели
`win.h`	число	высота главного окна, пиксели
`win.fnt.name`	строка	путь к файлу шрифта
`win.fnt.size`	число	размер шрифта главного окна, пункты
`win.gfx.up`	строка	путь к файлу изображения скроллера вверх для главного окна
`win.gfx.down`	строка	путь к файлу изображения скроллера вниз для главного окна
`win.gfx.h`	число	синоним `scr.gfx.h`
`win.col.fg`	цвет	цвет текста главного окна
`win.col.link`	цвет	цвет ссылок главного окна
`win.col.alink`	цвет	цвет активных ссылок главного окна

12.3 Параметры области инвентаря

Для области инвентаря заданы следующие параметры:

параметр	тип	описание
`inv.x`	число	абсцисса области инвентаря,пиксели
`inv.y`	число	ордината области инвентаря,пиксели
`inv.w`	число	ширина области инвентаря,пиксели
`inv.h`	число	высота области инвентаря,пиксели
`inv.col.fg`	цвет	цвет текста инвентаря
`inv.col.link`	цвет	цвет ссылок инвентаря
`inv.col.alink`	цвет	цвет активных ссылок инвентаря
`inv.fnt.name`	строка	путь к шрифту инвентаря
`inv.fnt.size`	число	размер шрифта инвентаря,пункты
`inv.gfx.up`	строка	путь к изображению скроллера вверх для инвентаря
`inv.gfx.down`	строка	путь к изображению скроллера вниз для инвентаря
`inv.mode`	строка	режим инвентаря

Параметр `inv.mode` может принимать значение `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.

12.4 Параметры главного меню

Для главного меню INSTEAD-SDL заданы следующие параметры:

параметр	тип	описание
`menu.col.bg`	цвет	цвет фона меню
`menu.col.fg`	цвет	цвет текста меню
`menu.col.link`	цвет	цвет ссылок меню
`menu.col.alink`	цвет	цвет активных ссылок меню
`menu.col.alpha`	цвет	прозрачность меню (0—255)
`menu.col.border`	цвет	цвет границы меню
`menu.bw`	число	толщина границы меню, пиксели
`menu.fnt.name`	строка	путь к шрифту меню
`menu.fnt.size`	число	размер шрифта меню, пункты
`menu.gfx.button`	строка	путь к значку меню
`menu.button.x`	число	абсцисса кнопки меню, пиксели
`menu.button.y`	число	ордината кнопки меню, пиксели
`snd.click`	строка	путь к звуку щелчка
`include`	строка	имя темы

Напоминаю,что имя темы есть имя каталога с нею.

12.5 Прочее

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

```
; $Name:Новая тема$
; модификация темы book
include = book
scr.gfx.h = 500
```

Интерпретатор ищет доступные темы в каталогах:

Unix версия интерпретатора просматривает игры в:
`/usr/local/share/instead/themes` (по умолчанию),

`~/.instead/themes`.
WinXP версия:
`Documents and Settings/USER/Local Settings/Application Data/instead/themes`
WinVista: `Users\USER\AppData\Local\instead\themes`
Все Windows: `куда-вы-установили-INSTEAD/themes`

Игра может задавать собственную тему; для этого в каталоге с игрой должен лежать тот самый `theme.ini`. Его формат никак при этом не меняется, просто он загружается в первую очередь вместе с игрой.

13 Дополнительные источники документации

Вот и закончен справочник по INSTEAD. Напомним, что INSTEAD расшифровывается (и переводится) как “Интерпретатор простых текстовых приключений”. Официальная документация находится в каталоге doc и поставляется с instead. Дополнительную информацию вы можете получить в Интернете:

- Сайт программы
- Форум программы