A visual DSL toolkit in Lua
Past, present and future

LogicEditor

Alexander Gladysh <ag@logiceditor.com>

Lua Workshop 2013
Toulouse
Outline

Introduction

The Problem

Classic third-party alternatives

Past generations

The present generation

The future

Questions?
Alexander Gladyshev

- CTO, co-founder at LogicEditor
- In löve with Lua since 2005
Use Lua to develop:
- Visual DSL toolkit (subject of this talk)
- Big-data analytics
- Intensive-load web-services
- In-browser and mobile games

- 600+ KLOC of private Lua codebase
- Some contributions to open-source Lua projects
The Problem

- Business-logic is highly volatile.
- Programmers are not domain area specialists.
- Specialist $\Leftrightarrow$ Manager $\Leftrightarrow$ Programmer loop is slow.
- Specialist $\Leftrightarrow$ Programmer loop is expensive.
- Let specialists do the business-logic!
OMG WTF
Non-programmers aren’t programmers

It is not enough to be able to compose an algorithm and even implement it with some simple programming language. For commercial programming you’ll also need, at least:

- Technical background
- Debugging skills
- Team coding skills
Solution

- A tool that prevents *technical* mistakes
- While limiting creativity as little as possible
- And is within grasp of a non-programmer.
Ad-hoc implementations

- One-shot, very limited flexibility
- Full of crutches
- Hard to maintain
Classic third-party alternatives
Descent Freespace Editor Events

Mission Event Edit

- Gorre directive
- Conditions to depart
  - OP when
    - OP or
      - OP is-destroyed-delay
        - 0
        - Gorre
      - OP has-departed-delay
        - 0
        - Gorre
    - OP send-message
      - Command
      - High
      - RTB message
- RTB directive
  - OP when
    - OP has-departed-delay
      - 0
      - Alpha 1
    - OP do-nothing

Directive text: Return to base

Repeat Count: 1
Interval time: 1
Score: 0
Chain Delay: 0

Chained: [on]
Apple Automator
What is in common?

- A visual domain-specific language,
- that allows user to describe
- the control-flow.
A retrospective of ideas

Screenshots shown here are from editors done by me and/or my colleagues for different companies we worked for, over time. Only an idea was re-used and improved between generations.
Video-adventure game editor

(No screenshots available)

➤ Legacy, circa 2002—2004
➤ Graph of in-game dialog remarks and answer choices
➤ Allowing to tie-in video-loop to remark
➤ No Lua.
Adventure game dialog editor, I, II

- Graph of in-game dialog remarks and answer choices
- With ability to add custom Lua code logic (in II)
- Generated data (in Lua) for a state machine (also in Lua)
Browser MMO quest editor, I, II
Цели
неинтерактивно [ # x ] [+]

Мгновенные эффекты
Игнорировать активацию в статистике: DA [#]

Действия: HET [+]

Овертайм-эффекты

Цель: на себя [#]
Время жизни: 255 [ ] (≥255 — беспрерывно)
Период: 0 [ ]
Изначальный кулдаун: 0 [ ]
Сброс в конце боя: HET [#][0 ]
Остается при снятии всех эффектов вручную: DA [#]
Максимальное число одновременно активных эффектов: 1 [ ] (0 — не ограничено)
Игровые режимы: дуэль [#]

При изменении набора характеристик

1. Если (изменения инициированы целью овертайм-эффекта [ # x ] и (жизнь #) в наборе изменений противника [ # ][ ] < уровней с 1 [ ] до 10 [ ] (учитывая уровень в счетчике: DA [#][(0)][1][x][+(+)][1][0][1][x][+(+)][0]), то
   1. Играть эффект абилки ID: 50402 [ ] [ # x ]
   2. Активировать OT-эффект №1 [ ], передав ключи + [ # x ]
   3. Увеличить у себя [#] статистику «исп. автоабилки [#]» эффекта №0 [ ] (0 — текущий) на 1 [ ] [ # x ][+(+)]

В конце хода цели
HET [+]

Временные модификаторы (кроме жизни)
HET [+]

1. Дополнительный OT-эффект

   Цель: на противника [#]
   Время жизни: 5 [ ] (≥255 — беспрерывно)
   Период: 0 [ ]
   Изначальный кулдаун: 0 [ ]
   Сброс в конце боя: HET [#][0 ]
   Остается при снятии всех эффектов вручную: DA [#]
Some non-programmers prefer visual control-flow editors.
Some — textual representation.
(Programmers hate to use both kinds.)
All editors were very useful, some — invaluable.
But, in retrospective, some should have been replaced by dedicated coders.
None of the past-generation editors were flexible enough to be used outside its immediate domain (but this never was an official goal for them).
The Visual Business Logic Editor Toolkit

If $\color{N} = 1$:
- Create object:
  - Icon:
  - Custom properties:
    - color: color
- And place → a cell with coords {x: $\color{N}$, y: $\color{N}$}

Else if $\color{N} = 2$:
- Create object:
  - Icon:
  - Custom properties:
    - color: color
- And place → a cell with coords {x: $\color{N}$}

Else if $\color{N} = 3$:
- Create object:
  - Icon:
  - Custom properties:
    - color: color
- And place → a cell with coords {x: $\color{N}$}

Else if $\color{N} = 4$:
- Create object:

---

Change Numeric expression:
- Numeric value
- Random number
- $a + b$
- $a - b$
- $a \times b$
- $a / b$
- Length of string
- Size of set of cells
- Other math ops
- Variable
- Game property
- Object property
- Game map properties
- Cell properties
- Returns numeric

Divides $a$ by $b_1 \ldots b_N$

Parameters:
- numeric $a$
- numeric $b_1$
- ... numeric $b_N$
Design goals

▶ Easy to create new editors.
▶ Easy to support existing editors.
▶ Easy to integrate with "any" other project on "any" technology.
▶ Easy *enough* to learn and use by end-users.
Editor use-cases

For example:

▶ A dialog editor for a game scenario writer.
▶ A magic system editor for a game designer.
▶ A mission logic editor for a game level designer.
▶ A DB query editor for a data analyst (Hadoop, anyone?).
▶ An advertising campaign targeting editor for a marketer.
▶ ...and so on.
The data is a tree corresponding to the control flow (or to anything tree-like, actually).

The output is structured text (code or data).

Editor code, UI and backend, is generated by Lua code in the Toolkit, from the data "schema".

Editor UI is in JavaScript / HTML, backend is in Lua.
The Data Schema

- Embedded Lua DSL (see my talk on Lua WS’11).
- Describes how to:
  - check data validity,
  - generate default data,
  - render data to editor UI,
  - change data in editor UI,
  - render the conforming data to the output code (or data).
- Two layers: human-friendly and machine-friendly
See also: http://bit.ly/le7-schema

lang:root "lua.print-string"

lang:value "lua.string.value" {
  data_type = "string";
  default = "Hallo, world!";
  render:js [[String Value]] { [[${1}]] };
  render:lua { [[${1}]] };
}
lang:func "lua.print-string" {
    "lua.string.value";
    render:js [[Print string]] {
        [[Print: ${1}]];
    };
    render:lua {
        [[print(${1})]];
    };
}
Default Data

```lua
{
    id = "lua.print-string";
    {
        id = "lua.string.value";
        "Hallo, world!"
    }
}

Renders to Lua as:

`print("Hallo, world!")`
UI for default data (simplified)

```html
<div id="lua.print-string">
  Print: <span id="lua.string.value">Hallo, world!</span>
</div>

NB: That <span> turns to edit-box on click.
```
Extending string type

```lua
lang:func "lua.string.reverse" {
    type = "lua.string";
    render:js [[Reverse string]] { [[Reverse: ${1}]] };
    render:lua { [[(${1}):reverse()]] };
}
```
Print with multiple arguments

lang:list "lua.print"
{
    "lua.string";
    render:js [[Print]] {
        empty = [[Print newline]];
        before = [[Print values: <ul><li>]];
        glue = [[</li><li>]];
        after = [[</li></ul>]];
    };
    render:lua {
        before = [[print([])];
        glue = [[,]];
        after = [[]]];
    };
}
Main primitives

- lang:const
- lang:value
- lang:enum
- lang:func
- lang:list
- lang:type
Machine-friendly schema

- node:literal
- node:variant
- node:record
- node:value
- node:list
Data-upgrade routines

- A set of hooks for data tree traversal.
- Transformations between two given data versions.
- In terms of node schema.
- Semi-automatic, template code is generated.
What else?

- Scopes in the schema.
- External and internal data-sources.
Several points of improvement

Current generation does its job well, but we see several ways on how to make it better

Several points to improve

▶ Better, modern HTML (at the cost of support of IE6).
▶ Lua in browser for a server-less integration option.
▶ Even more flexible and expressive Schema DSL.

NB: We’ll probably go for a control-flow diagram UI first, not text-based one (current text-based is cool enough).
Problems with the current DSL

- One language for three separate concepts:
  - data tree structure,
  - editor UI,
  - final output.

- Data tree structure gets a bit synthetic and convoluted at times.

- Should be easier to add alternative editor UIs.
Solution

- Three separate sets of languages:
  - data tree format,
  - render to output (per output format),
  - render to editor (per editor kind).

- CSS-like rules instead of pre-defined set of node types
Early examples


data:root "script"
data:type "script" ("*", "action")
data:type "action" "print-var" "var-name"

to:text "script" :T [[
local _VARS = {}
${indent(concat(children))}
]]
to:text "print-var" "var-name"
  :T [[print(_VARS[${quote:lua(node)}])]]

to:ui "print-var" "var-name"
  :T [[Print: ${child(1)}]]
An alternative approach to the Embedded DSLs in Lua

```lua
foo:bar "baz" { "quo" }
local proxy = foo
proxy = proxy["bar"]
proxy = proxy(foo, "baz")
proxy = proxy({ "quo" })
```
foo:bar "baz" { "quo" }

If proxy is as a FSM, indexes and calls — state transitions.

INIT | index "bar" -> foo.bar
    foo.bar | call -> foo.bar.name
    foo.bar.name | call -> foo.bar.name.param
FINAL <- foo.bar.name.param

Easier to code complex DSL constructs

play:scene [[SCENE II]]
.location [[Another room in the castle.]]
:enter "HAMLET"
:remark "HAMLET" [[
Safely stowed.
]]
:remark { "ROSENCRANTZ", "GILDERSTERN" }
  .cue [[within]] [[
Hamlet! Lord Hamlet!
]]
:remark "HAMLET" [[
What noise? who calls on Hamlet?
0, here they come.
]]
Questions?

Alexander Gladyshev, ag@logiceditor.com