
Next Basic AI Technique:

Scripting

References: *Buckland, Chapter 6*
CACM, 52(3), "Better Scripts, Better Games"

Scripting

- Two senses of the word
 - “scripted behavior”
 - having agents follow pre-set actions
 - rather than choosing them dynamically
 - “scripting language”
 - using a dynamic language
 - to make the game easier to modify
- The senses are related
 - a scripting language is good for writing scripted behaviors (among other things)

Scripted Behavior

- One way of building AI behavior
- What's the *other* way?
- Versus **simulation-based** behavior
 - e.g., goal/behavior trees
 - genetic algorithms
 - machine learning
 - etc.

Scripted vs. Simulation-Based AI Behavior

- Example of scripted AI behavior
 - fixed trigger regions
 - when player/enemy enters predefined area
 - send pre-specified waiting units to attach
 - doesn't truly simulate scouting and preparedness
 - player can easily defeat AI once she figures it out
 - mass outnumbering force just outside trigger area
 - attack all at once

Scripted vs. Simulation-Based AI Behavior

- Non-scripted (“simulation-based”) version
 - send out patrols
 - use reconnaissance information to influence unit allocation
 - adapts to player’s behavior (e.g., massing of forces)
 - can even vary patrol depth depending on stage of the game

Advantages of Scripted AI Behavior

- Much faster to execute
 - apply a simple rule, rather than run a complex simulation
- Easier to write, understand and modify
 - than a sophisticated simulation

Disadvantages of Scripted AI Behavior

- Limits player creativity
 - players will try things that “should” work (based on their own real-world intuitions)
 - will be disappointed when they don’t
- Allows degenerate strategies
 - players will learn the limits of the scripts
 - and exploit them
- Games will need *many* scripts
 - predicting their interactions can be difficult
 - complex debugging problem

Stage Direction Scripts

- Controlling camera movement and “bit players”
 - create a guard at castle drawbridge
 - lock camera on guard
 - move guard toward player
 - etc.
- Better application of scripted behavior than AI
 - doesn’t limit player creativity as much
 - improves visual experience
- Stage direction also be done by sophisticated simulation
 - e.g., camera system in God of War

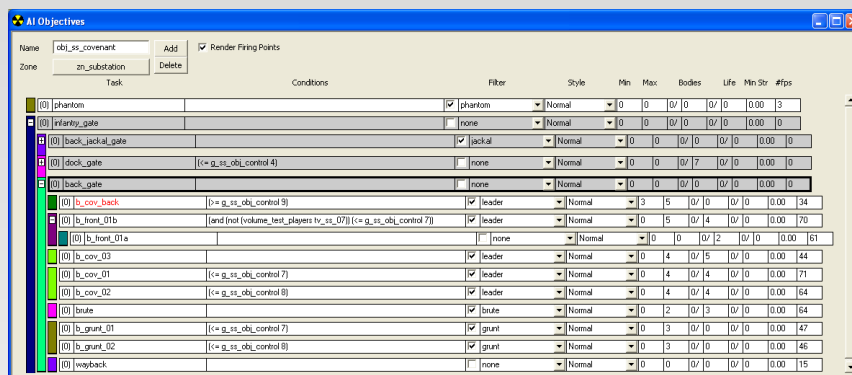
Scripting Languages

You can probably name a bunch of them:

- custom languages tied to specific games/engines
 - UnrealScript, QuakeC, HaloScript, LSL, ...
- general purpose languages
 - Tcl, Python, Perl, Javascript, Ruby, Lua, ...
 - the “modern” trend, especially with Lua

Often (mostly) used to write scripted (AI) behaviors.

Custom Languages and Tools



Name	Zone	Task	Conditions	Filter	Style	Min	Max	Bodies	Life	Min Str	#fps
[0] phantom	obi_ss_covenant			phantom	Normal	0	0	0/0	0/0	0.00	3
[0] phantom_gate	zn_substation			none	Normal	0	0	0/0	0/0	0.00	0
[0] back_jackal_gate				jackal	Normal	0	0	0/0	0/0	0.00	0
[0] deck_gate			[<= g_ss_obj_control 4]	none	Normal	0	0	0/7	0/0	0.00	0
[0] back_gate				none	Normal	0	0	0/0	0/0	0.00	0
[0] b_cov_back			[>= g_ss_obj_control 5]	leader	Normal	3	5	0/0	0/0	0.00	34
[0] b_front_01b			[and [not [volume_test_players lv_ss_07]] [<= g_ss_obj_control 7]]	leader	Normal	0	5	0/4	0/0	0.00	70
[0] b_front_01a				none	Normal	0	0	0/2	0/0	0.00	61
[0] b_cov_03				leader	Normal	0	4	0/5	0/0	0.00	44
[0] b_cov_01			[<= g_ss_obj_control 7]	leader	Normal	0	4	0/4	0/0	0.00	71
[0] b_cov_02			[<= g_ss_obj_control 8]	leader	Normal	0	4	0/4	0/0	0.00	64
[0] brute				brute	Normal	0	2	0/3	0/0	0.00	64
[0] b_grunt_01			[<= g_ss_obj_control 7]	grunt	Normal	0	3	0/0	0/0	0.00	47
[0] b_grunt_02			[<= g_ss_obj_control 8]	grunt	Normal	0	3	0/0	0/0	0.00	46
[0] wayback				none	Normal	0	0	0/0	0/0	0.00	15

“Designer UI” from Halo 3

Custom Scripting Languages

- A custom scripting language tied to a specific game, which is just idiosyncratically “different” (e.g., QuakeC) doesn’t have much to recommend it
- However, a game-specific scripting language that is **truly natural** for non-programmers can be very effective:

```
if enemy health < 500 && enemy distance < our bigrange
    move ...
    fire ...
else
    ...
return
```

(GalaxyHack)



IMGD 400X (B 09)

11

General Purpose Scripting Languages

What makes a general purpose scripting language different from any other programming language?

- interpreted (byte code, virtual machine)
 - faster development cycle
 - safely executable in “sandbox”
- simpler syntax/semantics:
 - untyped
 - garbage-collected
 - builtin associative data structures
- plays well with other languages
 - e.g., LiveConnect, .NET, Lua stack



IMGD 400X (B 09)

12

General Purpose Scripting Languages

But when all is said and done, it looks pretty much like “code” to me...☺

e.g. *Lua*

```
function factorial(n)
  if n == 0 then
    return 1
  end
  return n * factorial(n - 1)
end
```

Scripting Languages in Games

So it must be about something else...

*Namely, the **game development process**:*

- For the technical staff
 - data-driven design (scripts viewed as data, not part of codebase)
 - script changes do not require game recompilation
- For the non-technical staff
 - allows parallel development by designers
 - allows end-user extension

A Divide-and-Conquer Strategy

- implement part of the game in C++
 - the time-critical inner loops
 - code you don't change very often
 - requires complete (long) rebuild for each change
- and part in a scripting language
 - don't have to rebuild C++ part when change scripts
 - code you want to evolve quickly (e.g, AI behaviors)
 - code you want to share (with designers, players)
 - code that is not time-critical (can migrate to C++)

General Purpose Scripting Languages

But to make this work, you need to successfully address a number of issues:

- Where to put boundaries (APIs) between scripted and “hard-coded” parts of game
- Performance
- Flexible and powerful debugging tools
 - even more necessary than with some conventional (e.g., typed) languages
- Is it **really** easy enough to use for designers!?

Lua in Games

- Has come to dominate other choices
 - Powerful and fast
 - Lightweight and simple
 - Portable and free
- Currently Lua 5.1
- See <http://lua.org>



Lua in Games

per Wikipedia

- * [Aleph One](#) (an open-source enhancement of *Marathon 2: Durandal*) supports Lua, and it's been used in a number of scenarios (including *Excalibur* and *Eternal*).
- * [Bobby Volley](#), in which bots are written in Lua.
- * *Company of Heroes*, a WW2 RTS. Lua is used for the console, AI, single player scripting, win condition scripting and for storing unit attributes and configuration information.
- * [Crysis](#), a first-person shooter & spiritual successor to *Far Cry*.
- * [Dawn of War](#), uses Lua throughout the game.
- * [Destroy All Humans!](#) and [Destroy All Humans! 2](#) both use Lua.
- * [Escape from Monkey Island](#) is coded in Lua instead of the SCUMM engine of the older titles. The historic "SCUMM Bar" is renovated and renamed to the "Lua Bar" as a reference.
- * [Far Cry](#), a first-person shooter. Lua is used to script a substantial chunk of the game logic, manage game objects' (Entity system), configure the HUD and store other configuration information.
- * [Garry's Mod](#) and [Fortress Forever](#), mods for *Half-Life 2*, use Lua scripting for tools and other sorts of things for full customization.
- * [Grim Fandango](#) and [Escape from Monkey Island](#), both based on the GrimE engine, are two of the first games which used Lua for significant purposes.



Lua in Games (cont'd)

- * [Gusanos](#) (Version 0.9) supports Lua Scripting for making the whole game modable.
- * [Homeworld 2](#) uses Lua scripting for in-game levels, AI, and as a Rules Engine for game logic.
- * [Incredible Hulk: Ultimate Destruction](#) uses Lua for all mission scripting
- * [JKALua](#), A game modification for the game JK3: Jedi Academy.
- * [Multi Theft Auto](#), a multi-player modification for the Grand Theft Auto video game series. The recent adaptation for the game Grand Theft Auto San Andreas uses Lua.
- * [Painkiller](#)
- * [Ragnarok Online](#) recently had a Lua implementation, allowing players to fully customize the artificial intelligence of their homunculus to their liking, provided that they have an Alchemist to summon one.
- * [ROBLOX](#) is an online Lego-like building game that uses Lua for all in-game scripting.
- * [SimCity 4](#) uses Lua for some in-game scripts.
- * [Singles: Flirt Up Your Life](#) uses Lua for in-game scripts and object/character behavior.
- * [Spring](#) (computer game) is an advanced open-source RTS engine, which is able to use Lua for many things, including unit/mission scripting, AI writing as well as interface changes.
- * [S.T.A.L.K.E.R.: Shadow of Chernobyl](#)
- * [Star Wars: Battlefront](#) and [Star Wars: Battlefront 2](#) both use Lua.



Lua in Games (cont'd)

- * [Star Wars: Empire at War](#) uses Lua.
- * [Supreme Commander](#) allows you to edit almost all its aspects with Lua.
- * [Toribash](#), a turn-based fighting game, supports Lua scripting.
- * [Vendetta Online](#), a science fiction MMORPG, lets users use Lua to customize the user interface, as well as create new commands and react to events triggered by the game.
- * [Warhammer Online](#) uses Lua.
- * [The Witcher](#).
- * [World of Warcraft](#), a fantasy MMORPG. Lua is used to allow users to customize its user interface.
- * [Xmoto](#), a free and open source 2D motocross platform game, supports Lua scripting in levels.



Lua Language Data Types

- **Nil** – singleton default value, nil
- **Number** – internally double (no int's!)
- **String** – array of 8-bit characters
- **Boolean** – true, false
 - Note: *everything* except nil coerced to false!, e.g., "", 0
- **Function** – unnamed objects
- **Table** – key/value mapping (any mix of types)
- **UserData** – opaque wrapper for other languages
- **Thread** – multi-threaded programming (reentrant code)

Lua Variables and Assignment

- **Untyped:** any variable can hold any type of value at any time
 - A = 3;
 - A = "hello";
- **Multiple values**
 - in assignment statements
 - A, B, C = 1, 2, 3;
 - multiple return values from functions
 - A, B, C = foo();

“Promiscuous” Syntax and Semantics

- *Optional* semi-colons and parens
 - A = 10; B = 20;
 - A = 10 B = 20
 - A = foo();
 - A = foo
- *Ignores* too few or too many values
 - A, B, C, D = 1, 2, 3
 - A, B, C = 1, 2, 3, 4
- Can lead to a debugging *nightmare!*
- *Moral:* Only use for small procedures

Lua Operators

- arithmetic: + - * / ^
- relational: < > <= >= == ~=
- logical: and or not
- concatenation: ..

... *with usual precedence*

Lua Tables

- heterogeneous associative mappings
- used a lot
- standard array-ish syntax
 - except any object (not just int) can be “index” (key)
`mytable[17] = “hello”;`
`mytable[“chuck”] = false;`
 - curly-bracket constructor
`mytable = { 17 = “hello”, “chuck” = false };`
 - default integer index constructor (starts at 1)
`test_table = { 12, “goodbye”, true };`
`test_table = { 1 = 12, 2 = “goodbye”, 3 = true };`

Lua Control Structures

- Standard **if-then-else**, **while**, **repeat** and **for**
 - with **break** in looping constructs
- Special **for-in** iterator for tables
`data = { a=1, b=2, c=3 };`
`for k,v in data do print(k,v) end;`
produces, e.g.,
`a 1`
`c 3`
`b 2`
(order undefined)

Lua Functions

- standard parameter and return value syntax

```
function (a, b)
  return a+b
end
```
- inherently unnamed, but can assign to variables

```
foo = function (a, b) return a+b; end
foo(3, 5) → 8
```
- convenience syntax

```
function foo (a, b) return a+b; end
```



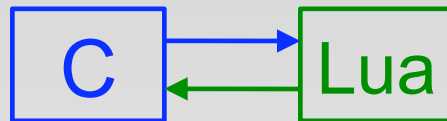
Lua Features not Covered

- object-oriented style (alternative dot/colon syntax)
- local variables (default global)
- libraries (sorting, matching, etc.)
- namespace management (using tables)
- multi-threading (thread type)
- compilation (bytecode, virtual machine)
- features primarily used for language extension
 - metatables and metamethods
 - fallbacks

See <http://www.lua.org/manual/5.1>



Connecting Lua and C++



- Accessing Lua from C++
- Accessing C++ from Lua

*See more details and examples in
Buckland, Ch 6.*

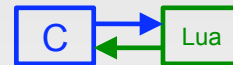
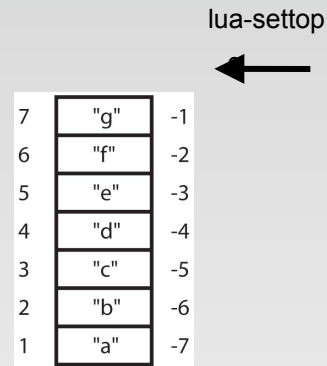
Connecting Lua and C++

- Lua virtual stack
 - bidirectional API/buffer between two environments
 - preserves garbage collection safety
- data wrappers
 - `UserData` – Lua wrapper for C data
 - `luabind::object` – C wrapper for Lua data

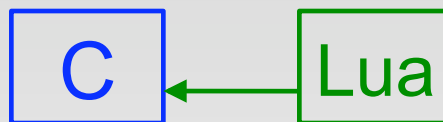


Lua Virtual Stack

- both C and Lua env'ts can put items on and take items off stack
- push/pop or direct indexing
- positive or negative indices
- current top index (usually 0)



Accessing Lua from C



Accessing Lua Global Variables from C

- C tells Lua to push global value onto stack

```
lua_getglobal(pLua, "foo");
```

- C retrieves value from stack

- using appropriate function for expected type

```
string s = lua_tostring(pLua, 1);
```

- or can check for type

```
if ( lua_isnumber(pLua, 1) )
```

```
{ int n = (int) lua_tonumber(pLua, 1) } ...
```

- C clears value from stack

```
lua_pop(pLua, 1);
```



Accessing Lua Tables from C (w. LuaBind)

- C asks Lua for global values table

```
luabind::object global_table = globals(pLua);
```

- C accesses global table using overloaded [] syntax

```
luabind::object tab = global_table["mytable"];
```

- C accesses any table using overloaded [] syntax and casting

```
int val = luabind::object_cast<int>(tab["key"]);
```

```
tab[17] = "shazzam";
```

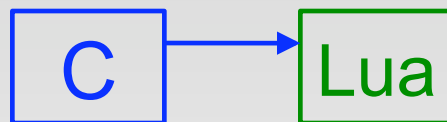


Calling Lua Functions from C (w. LuaBind)

- C asks Lua for global values table
`luabind::object global_table = globals(pLua);`
- C accesses global table using overloaded [] syntax
`luabind::object func = global_table["myfunc"];`
- C calls function using overloaded () syntax
`int val =
luabind::object_cast<int>(func(2, "hello"));`



Accessing C from Lua



Calling C Function from Lua (w. LuaBind)

- C “exposes” function to Lua

```
void MyFunc (int a, int b) { ... }  
module(pLua) [  
    def(“MyFunc”, &MyFunc)  
];
```

- Lua calls function normally in scripts

```
MyFunc(3, 4);
```



Using C Classes in Lua (w. LuaBind)

- C “exposes” class to Lua

```
class Animal { ...  
    public:  
        Animal (string ..., int ...) ... {}  
        int NumLegs () { ... }  
  
    module (pLua) [ class <Animal>(“Animal”)  
        .def(constructor<string, int>())  
        .def(“NumLegs”, &Animal::NumLegs) ];
```



- Lua calls constructor and methods

```
cat = Animal(“meow”, 4); print(cat.NumLegs())
```

JavaScript

- Technically called “ECMAScript”, according to the ECMA-262 standard
- General purpose scripting language
 - very similar syntax/semantics to Lua
 - originally developed to run in web browsers
 - included in standard Java JRE since Java 6
 - very easy connection to Java via [LiveConnect](#)

See details in O'Reilly, JavaScript—The Definitive Guide

Accessing JavaScript from Java

- `javax.script.ScriptEngine.eval(String)`
 - give it any JavaScript expression or statement
 - global variable
 - table lookup
 - function application
 - etc.
 - evaluates expression in current (persistent) state of `ScriptEngine` instance
 - primitive data types automatically coerced
 - wrapper classes for other data

Accessing Java from JavaScript

- All Java packages (and thus the classes and their methods) are directly accessible via `Packages` object

e.g.,

```
Packages.java.awt.Toolkit.getDefaultToolkit().beep()
```

Future Directions

White et al., "Better Scripts, Better Games", CACM, 52(3), March 2009.

- It's *dangerous* to put general purpose scripting languages into the hands of non-technical staff
 - destructive access to game state
 - performance degradation (e.g., infinite loops)
 - buggy synchronization (no transaction support)
- **Solution:** Design *restricted*, but *generic* languages that embody *design patterns* appropriate for games

Game Scripting Patterns

- **Restricted Iteration Pattern**
 - remove general iteration, goto and recursion
 - allow only “for each” iterations
- **Concurrency Patterns**
 - e.g., inventory management
 - instead of lock-based synchronization
- **State-Effect Pattern**
 - main game loop has effects and update phases
 - partition game object attributes into effect and state attributes (each only used in one phase)