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 attack
  - 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.

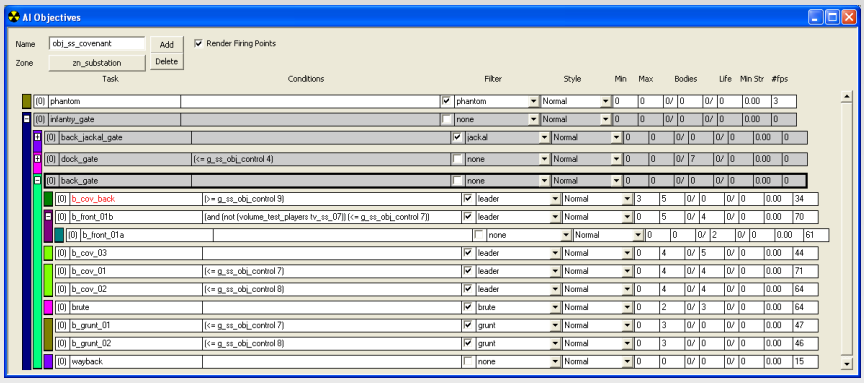
Custom Scripting Languages

- 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)

Custom Languages and Tools



"Designer UI" from Halo 3

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"
  - recently JIT native compilation also (see http://www.mono-project.com/Scripting_With_Mono)
- simpler syntax/semantics:
  - untyped
  - garbage-collected
  - builtin associative data structures
- plays well with other languages
  - e.g., LiveConnect, .NET, Lua stack

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 more 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 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
```

why is this important/useful?

- convenience syntax

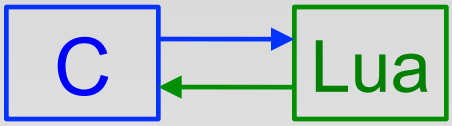
```
function foo (a, b) return a+b; end
```

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)
- 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++

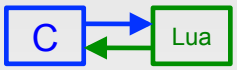


- Accessing Lua from C++
- Accessing C++ from Lua

See more details and examples in Buckland, Ch 6.

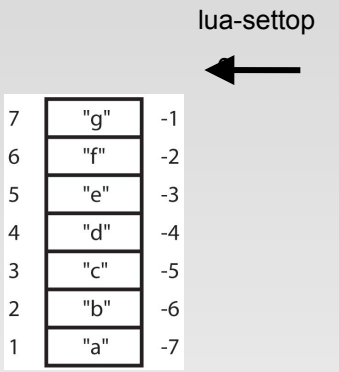
Connecting Lua and C++

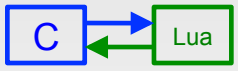
- Lua virtual stack
  - bidirectional API/buffer between two environments
  - preserves garbage collection safety
- data wrappers
  - UserData – Lua wrapper for C data
  - luabind::object – C wrapper for Lua data



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);
```

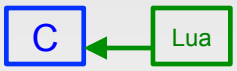
Accessing Lua Tables from C (w. LuaBind)

- *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 <u>any</u> table using overloaded [ ] syntax and casting*
  int val = luabind::object_cast<int>(tab[“key”]);

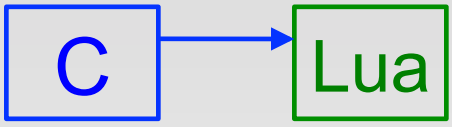
  tab[17] = “shazzam”;

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



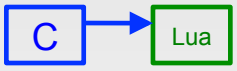
Calling C Function from Lua (w. LuaBind)

- *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);
```



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)

Go ye forth and script!