To Script, or Not Script, That is the Question

Artificial Intelligence for
Interactive Media and Games

Professor Charles Rich
Computer Science Department
rich@wpi.edu

[Based on Buckland, Chapter 6 and lecture by Robin Burke]

Outline

- Scripting
- Lua Language
- Connecting Lua and C++ (LuaBind)
- Scripted State Machine
- Scripting Homework (due Sunday)

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 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

- Typically less computation
  - 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 physical intuitions)
  - will be disappointed when they don’t
- Allows degenerate strategies (“exploits”)
  - 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 logic
  - doesn't limit player creativity as much
  - improves visual experience
- Can 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.

Scripting Languages

- Easier to learn and use than C/C++ to write small procedures
  - dynamically typed ("untyped")
  - garbage collected
  - simpler syntax
- Slower to execute (becoming less relevant with JIT compilation)
- Many popular applications and languages
  - robotics (Python)
  - web pages (JavaScript)
  - system administration (Perl)
  - games (Lua), etc.

Scripting Languages in Games

- 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++)
    - parameter files (cf. Raven Params.ini)

Lua in Games

- Has come to dominate other choices
  - Powerful and fast
  - Lightweight and simple
  - Easily extended
  - Portable and free
- Currently Lua 5.3.3
- 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 **0** and **nil** coerced to true!, e.g., "" is true
- 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 <u>small</u> 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;
  - note key is *evaluated*

    x = “chuck”
    mytable[x] = false
  - alternative “dot” syntax for constant string key

    mytable.chuck = false

Lua Table Constructor Syntax

- "curly bracket" constructor (for constant keys)
  mytable = { 17 = "hello", chuck = false };
- alternative syntax to evaluate keys
  x = 17; y = "chuck";
  mytable = { [x] = "hello", [y] = 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 (*order undefined*)
  data = { a=1, b=2, c=3 };
  for k,v in data do print(k,v) end;

  e.g., can produce
  a 1
  c 3
  b 2

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
```


Optional Syntax for Functions

- alternative colon syntax for calling functions

```
x:foo(a, b)
```

is equivalent to

```
x.foo(x, a, b)
```

Why?

Object-Oriented Pgming in Lua

- No ‘class’ construct per se (cf. LuaBind)
- But *tables of functions* behave very similarly

```
Account = { withdraw = function(self, amt)
                          self.balance = self.balance – amt
                        end,
            deposit = function(self, amount) ... end,
            ... }
a = { balance = 200,
      withdraw = Account.withdraw, deposit = Account.deposit, ...}

a.withdraw(a, 100);
a:withdraw(100)
```


Lua Features not Covered

- 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/5.3

Running Lua 5.3 in VS 2015 C++

In Project > Properties
- > C/C++ > General
  - Additional Include Directories: ..\Common\lua\include
- > Linker > General
  - Additional Library Directories: ..\Common\lua\lib

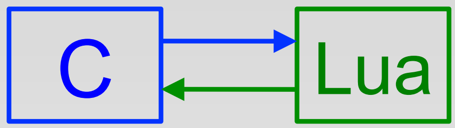
C++ Header:

```
#pragma comment(lib, "lua.lib")
extern "C"
{
    #include <lua.h>
    #include <lualib.h>
    #include <luaxlib.h>
}
```


Running Lua 5.3.3 in VS 2015 C++

```
lua_State* pLua = luaL_newstate();
luaL_openlibs(pLua);
luaL_dofile(pLua, script_name);
...
lua_close(pLua);
```

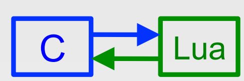
Connecting Lua and C++



- Accessing Lua from C++
  - global variables
  - tables (with/without LuaBind)
  - functions (with/without LuaBind)
- Accessing C++ from Lua (with LuaBind)
  - functions
  - classes
- LuaBind definitions for Lua "classes"

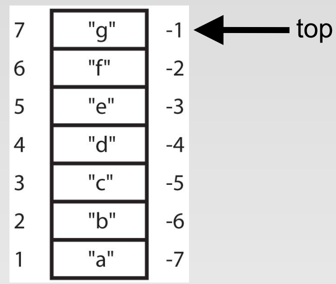
Connecting Lua and C++

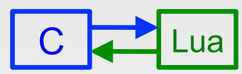
- Lua virtual stack
  - bidirectional API/buffer between two environments
  - preserves garbage collection safety
- data wrappers
  - UserData – Lua wrapper for C data
  - luabind::object – C wrapper for Lua data



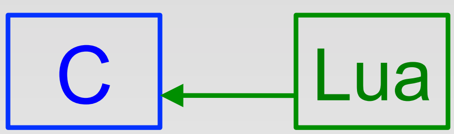
Lua Virtual Stack

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





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*

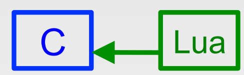
    `string s = lua_tostring(pLua, 1);`

  - *or can check for type*

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

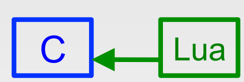
- *C clears value from stack*

  `lua_pop(pLua, 1);`

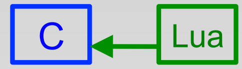


Accessing Lua Global Variables from C

- Common\script\LuaHelperFunctions.h
  - T PopLuaNumber(pLua, "foo")
  - std::string PopLuaString(pLua, "foo")
  - bool PopLuaBool(pLua, "foo")



Accessing Lua Tables from C



- *C asks Lua to push table object onto stack*

  lua_getglobal(pLua, "some_table");

- *C pushes key value onto stack (using appropriate API function for key type)*

  lua_pushstring(pLua, "myKey");

- *C asks Lua to replace given key on stack with corresponding value from given table*

  lua_gettable(pLua, -2);

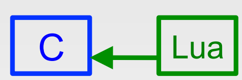
- *C retrieves value from stack (w. appropriate API)*

  string myvalue = lua_tostring(pLua, -1);

- *C clears value from stack:* lua_pop(pLua, 1);
- *Optional: C clears table from stack:* lua_pop(pLua, 1);

Accessing Lua Tables from C

- Common\script\LuaHelperFunctions.h
  - T LuaPopNumberFieldFromTable(pLua,"myKey")
  - std::string LuaPopStringFieldFromTable(pLua, "myKey")



Calling Lua Function from C

- *C asks Lua to push function object onto stack*

  lua_getglobal(pLua, "some_function");

- *C pushes argument values onto stack (using appropriate API function for each argument type)*

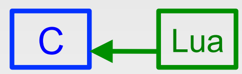
  lua_pushnumber(pLua, 17);

  lua_pushstring(pLua, "myarg");

- *C asks Lua to replace given args and function object on stack with specified number of return value(s)*

  lua_call(pLua, 2, 1);

- *C retrieves and clears values from stack*



LuaBind 0.9

- Handy utility
- for connecting Lua and C
- without explicitly manipulating Lua virtual stack
- uses luabind::object "wrapper" class in C++
- overloads [ ] and ( ) syntax and = in C++
- http://luabind.sf.net

Running LuaBind 0.9 in VS 2015 C++

In Project > Properties
> C/C++ > General
Additional Include Directories: ..\Common\luabind\include;
..\Common\boost\include
> Linker > General
Additional Library Directories: ..\Common\luabind\lib

C++:

```
#pragma comment(lib, "luabind-0.9.lib")
#include <luabind/luabind.hpp>
luabind::open(pLua);
```


Accessing Lua Global Variables from C (w. LuaBind)

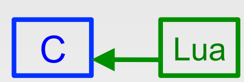
- *C asks Lua for global values table*

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

- *C accesses global table using <u>overloaded [ ] syntax</u> and casting*

```
global_table["foo"] = "hello";

string s =
    luabind::object_cast<string>(global_table["foo"]);
```



Accessing Lua Tables from C (w. LuaBind)

- *C asks Lua for global values table*

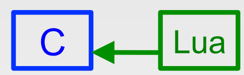
```
luabind::object global_table = globals(pLua);
```

- *C accesses global table using* ***overloaded [ ] syntax***

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

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

```
mytab[17] = 5;
int val = luabind::object_cast<int>(mytab["key"]);
```



Calling Lua Functions from C (w. LuaBind)

- *C asks Lua for global values table*

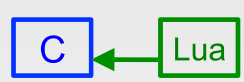
```
luabind::object global_table = globals(pLua);
```

- *C accesses global table using overloaded [ ] syntax*

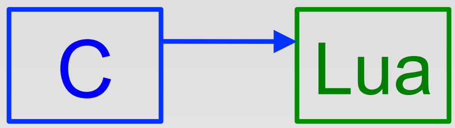
```
luabind::object myfunc = global_table["myfunction"];
```

- *C calls function using* ***overloaded ( ) syntax***

```
int val =
    luabind::object_cast<int>(myfunc(2, "hello"));
```



Accessing C from



...using LuaBind only

Calling C Function from Lua (w. LuaBind)

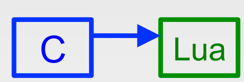
- *C "exposes" function to Lua*

```
void MyFunc (int a, int b) { ... }

module(pLua) [
    def("MyFunc", &MyFunc)
];
```

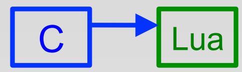
- *Lua calls function normally in scripts*

```
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())
```


Defining Lua Classes in Lua w. LuaBind

```
class 'Animal'

function Animal:__init(noise, legs)
  self.noise = noise
  self.legs = legs
 end

function Animal:getLegs () return self.legs end

cat = Animal("meow", 4); print(cat:getLegs())
```

- *see details of inheritance in Buckland*

Scripted State Machine

- *Goal:* Allow state changes and behaviors within given states to be modified without recompiling game
  - such changes can be made by non-developer
  - designer or user writes only Lua code
- Some changes will still require C coding and recompilation:
  - adding new properties of entities (e.g., Miner)
  - adding new capabilities to state machine interpreter
  - (think about extensions to cover these cases....)

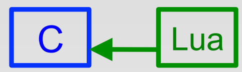
Scripted State Machine

- Each state is a Lua table with keys “Enter”, “Execute” and “Exit”
- Values are Lua functions (with entity as first arg)

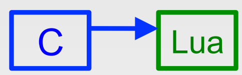
```
State_Sleep["Execute"] = function(miner)
    if miner:Fatigued() then
        print ("[Lua]: ZZZZZZ... ")
        miner:DecreaseFatigue()
    else
        miner:GetFSM():ChangeState(State_GoToMine)
    end
```

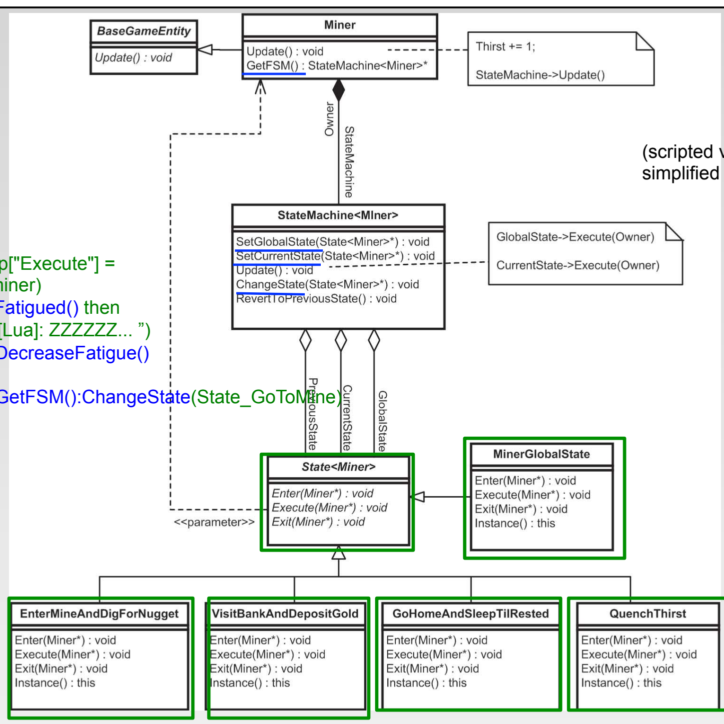
Scripted State Machine

- Which Lua objects and functions need to be accessed from C++?



- Which C++ objects and functions need to be accessed from ("exposed to") Lua?



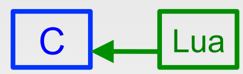


(scripted version a bit simplified from this)

```
State_Sleep["Execute"] =
 function(miner)
   if miner:Fatigued() then
     print ("[Lua]: ZZZZZZ... ")
     miner:DecreaseFatigue()
   else
     miner:GetFSM():ChangeState(State_GoToMine)
 end
```

Scripted State Machine

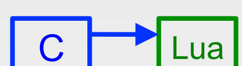
- Which Lua objects and functions need to be accessed from C++?
  - m_CurrentState holds a luabind::object which is a state table in Lua
  - accessed as
    m_CurrentState["Execute"](m_pOwner)



Scripted State Machine

- Which C++ objects and functions need to be accessed from ("exposed to") Lua?
  - ScriptedStateMachine methods (generic)
    - CurrentState, SetCurrentState, ChangeState
  - Entity methods (generic, but in Miner in SSM)
    - getFSM
  - Miner methods (used in Lua state code)
    - DecreaseFatigue, IncreaseFatigue, Fatigued
    - GoldCarried, SetGoldCarried, AddToGoldCarried

Code Walk



Scripting Homework

- Due Sunday midnight
- Add global states and blip states to Scripted State Machine
- Use these new facilities to add new “frequent urination” behavior to Miner