Game Engineering

CS420-11

lua

David Galles

Department of Computer Science
University of San Francisco
11-0: **Lua**

- Scripting, or “glue” language
- Built to interface with C code
- Gaming industry, most common scripting language
  - Many modern games have a lua interpreter running inside them to handle scripting
Lua

- Lua is very small and (for an interpreted scripting language) fast
- Don’t usually write huge amounts of code in Lua
- All of the “heavy lifting” is done by C, C++ code
Global variables
- Don’t need to declare global variables
- Assigning a value to a global variable creates it
- Assigning nil to a global variable removes it
Global variables

- print(x)
- x = 3
- print(x)
- x = "A String!"
- print(x)
Statement separators

- Don’t use whitespace (like python)
- Don’t use semicolons either! (Though semicolons are optional between statements)
- Don’t need to use anything(!), lua can infer end of statements
  - If you don’t use semicolons or whitespace, you will make other people who look at your code very, very angry!
11-5: Types

- Lua types:
  - nil, boolean, number, string, userdata, function, thread, table
  - Function type returns the type of a value
    - print(type(3)), print(type("foo")), print(type(print))
    - print(type(type(print)))?
nil

- The type “nil” is essentially “I don’t exist”
- Variables that have not been defined are nil
- Setting a variable to nil removes it
Booleans

- Standard boolean values: true, false
- Anything can be used in a boolean test (like C!)
- false and nil are both false, anything else (including 0!) is true
  - print(not 0)
  - print(not nil)
Numbers

- Lua only has a single numeric type, “number”
- Equivalent to a double-precision floating point value in C
- No integers!
11-9: Strings

- Strings are immutable in Lua
  - Can’t change a string – need to create a new string instead
  - Denoted with either " or ’
    - "This is a string"
    - ’This is also a string’
  - Standard C-like escape sequences
    \n, \", \’, , etc
Strings

- Anything between [[ and ]] is a string
- "raw" string – escape characters are not interpreted
- If [[ is on a line by itself, first EOL is ignored
- examples
Strings

• Strings are automatically converted to numbers
  • print("10" + 1)
  • print("10 + 1")
  • print("-31.4e-1" * 2)
Numbers are also automatically converted to strings

- print(3 .. 4)
- print((3 .. 4) * 2)
- print(type(3 .. 4) * 2))
- print(10 == "10")
- tostring, tonumber
Userdata

• Blind data
• Used for storing C/C++ datastructures (or, more generally, data from some other language)
• Lua can’t manipulate it, only pass it around
• What good is it?
• Lua is often used as a “glue” language
• Make a call into C/C++ code, returns some userdata
  • Definition of some object in your gameworld, perhaps
• Pass that userdata on to some other C/C++ code
Functions

- Functions are first-class values in Lua.
- Anywhere you can use a number / string / etc, you can use a function.

```lua
function(params) <body> end
```
double = function(x) return 2 * x end

• Using a slightly easier to read syntax:

double = function(x)
    return 2 * x;
end
Some syntactic sugar:

```plaintext
add = function(x,y)
    return x + y
end
```

is equivalent to

```plaintext
function add(x,y)
    return x + y
end
```

Top version emphasizes that functions are values like any other value.
Tables

- Tables are associative arrays
- Essentially hash tables
- Can use any value a key, and any value as data
Tables

- Create an empty table using {}
- Use x[] notation to access table elements

```python
x = { }  # create empty table
x["foo"] = 4
x[2] = "bar"
x["foo"] = x["foo"] + 1
```
Tables

- Table entries that have not been assigned values have the value nil
- Can remove elements from a table by assigning them nil
  - Doesn’t create a table entry with value nil
  - Removes entry from the table completely
- Just like global variables (which are implemented using a table)
• Fields (syntactic sugar)
  • For keys that are strings, we can access them using C struct notation

```
x = {}
x["foo"] = 3  -- equivalent to the next line:
x.foo = 3      -- just syntactic sugar
```
Arrays

• There are no “array”s in lua
• Tables that have integers (numbers!) as keys can function as arrays
• Can start with any index you like (just tables!), but all lua library functions start at index 1 (and not 0!)
Table Constructors (More syntactic sugar!)

“Array” table constructor

```plaintext
x = {4, 5, "foo", "cat"}
```

equivalent to

```plaintext
x = { }; x[1] = 4; x[2] = 5;
```
• Table Constructors (More syntactic sugar!)
  • “Record” table constructor

\[
x = \{\text{red} = 1, \text{green} = 2, \text{blue} = 3, \text{purple} = 3.7\}
\]
equivalent to

\[
x = \{\}; x[\"red\"] = 1; x[\"green\"] = 2; x[\"blue\"] = 3; x[\"purple\"] = 3.7;
\]
Table Constructors (More syntactic sugar!)

opnames = {
    "+" = "add",
    "*" = "multiply",
    "-" = "subtract",
    "/" = "divide"
}

genConstr = {
    [3] = "foo",
    [4] = 5,
    "key" = "keyval",
    "pi" = 3.14159
}
We can store anything in tables

```lua
x = { }
x["a"] = function(x) return x * x end
print(x["a"](2))
```
We can store anything in tables

```python
x = { }
x["a"] = {"dog", "cat"}
x["b"] = { key1 = 3, key2 = "brown" }
print(x["a"][1])
print(x["b"]["key2"])
```
We can use anything as a key

```lua
f = function(x) return x + 2 end
x = {
}
x[f] = "Hello"
```
11-29: Tables

- We can use anything as a key

```python
f = function(x) return x + 2 end
f2 = funcion(x) return x + 5 end
x = {}
x[f] = f2
print(x[f](3))
```
How could we implement a linked structure using tables?
How could we implement a linked structure using tables?

```plaintext
> lst = nil  -- not required unless lst already defined
> for i = 1, 10 do
>> lst = {data=i, next = lst}
>> end
> print(lst.data)
> print(lst.next.data)
> print(lst.next.next.data)
```
Tables

- Tables are indexed by reference
- That means that the address of the structure is used by the hash function
- What does that say about strings?
x = "foo"
y = "f"
z = y .. "oo"

• x and z not only have the same value – they are in fact pointers to the same memory location!
11-34: Operators

- Standard Arithmetic operators, standard precedence
- Relational operators: $<$, $>$, $<=$, $>$=, ==, ~=
  - Operator == tests for equality, operator ~= tests for non-equality
  - Functions, tables, userdata are compared by reference (pointer)
Operators

- Automatic conversion between strings and numbers leads to some pitfalls in relational operators
  - "0" == 0 returns false
  - 3 < 12 returns true
  - "3" < "12" returns false (why?)
• Logical operators
  • and
    • If first argument is false (nil/false), return first argument, otherwise return second argument
  • or
    • If first argument is true (not nil/false), return first argument, otherwise return second argument
Operators

• How can we use and, or to create a C-like ? : operator
  • (test) ? value1 : value2
Operators

• How can we use and, or to create a C-like ? : operator
  • (test) ? value1 : value2
• test and value1 or value2
  • When doesn’t this work?
Operators

- How can we use and, or to create a C-like ? : operator
  - (test) ? value1 : value2
- test and value1 or value2
  - When doesn’t this work?
  - If test and value1 are both nil
11-40: **Operator Precedence**

\[
\begin{align*}
\wedge & \quad \text{(not part of core lua, needs math library)} \\
\text{not} & \quad \text{-(unary)} \\
* & / \\
+ & - \\
.. & \\
< & > & <= & >= & \sim= & == \\ \\
\text{and} \\
\text{or} \\
\end{align*}
\]

- \( \wedge \) and .. are right associative, all others are left associative.
• `dofile(<filename>)`
  • Evaluate the file as if it were typed into the interpreter
  • (there are one or two small differences between `dofile` and typing into the interpreter, we’ll cover them in a bit)
Standard Assignment:
- \( z = 10 \)
- \( y = \text{function}(x) \ return \ 2^x \ \text{end} \)

Multiple Assignment:
- \( a, b = 10, 20 \)
• Multiple Assignment
  • Values on right are calculated, then assignment is done
  • Use Multiple Assignment for swapping values
  • \( x, y = y, x \)
11-44: Assignment

• Multiple Assignment
  • Lua is relatively lax about number of values in multiple assignments

\[
\begin{align*}
x, y, z & = 1, 2 \quad \text{-- } z \text{ gets assigned nil} \\
x, y & = 1, 2, 3 \quad \text{-- } \text{value 3 is discarded} \\
x, y, z & = 0 \quad \text{-- Common mistake --} \\
& \text{what does this do?}
\end{align*}
\]
Assignment

- Multiple assignment is useful for returning multiple values

```python
f = function() return 1,2 end
x = f()
y, z = f()
```
If statements

if <test> then <statement block> [else <statement block>] end

So:

if a < 0 then a = -a end
if a < b then return a else return b end
if a < b then
  a = a + 1
  b = b - 1
end
11-47: If statements

- elseif avoids multiple ends

```python
if op == "+" then
    r = a + b
elseif op == "-" then
    r = a - b
elseif op == "*" then
    r = a * b
elseif op == "/" then
    r = a / b
else
    error("invalid operation")
end
```
While statements

while test do <statement block> end

To print out all of the element in an array:

i = 1
while a[i] do
    print(a[i])
    i = i + 1
end
Repeat statements are just like do-whiles in C/Java/C++

Note that the <test> tells us when to stop, not when to keep going.

repat <statement block> until <test>
for var = exp1, exp2, exp3 do
    <statement block>
end

• Where exp1 is the starting value, exp2 is the ending value, and exp3 is the increment (defaults to 1 if not given)

for i = 1, 10 do
    print i
end
for i = 10, 1, -2 do
    print i
end
Some wrinkles:

- The bounds expressions are only evaluated once, before the for loop starts.
- The index variable is locally declared for only the body of the loop.
- Never assign a value to the loop variable – undefined behavior.
Numeric For statement

i = 37
for i = 1, 10 do
    print(i)
end
print(i)
for key, val in pairs(t) do
  print(key, val)
end

- pairs() is a function used to iterate through the list
- Execute the loop once for each element of the table (order is not determined)
- More on iterators (including building your own) in a bit...
for x in pairs(t) do
    print(x)
end

- Can you figure out what does this does?
- Hint ... think about how multiple assignment works...
11-55: **Generic For statement**

```.lua
for x in pairs(t) do
    print(x)
end
```

- Prints out all of the keys in a table
- The iterator returns a key and value, value is ignored
Generic For statement

- Let’s say we have a table, and we want to know what key is associated with a particular value
  - For instance, we have a phonebook table that matches names to phone numbers
- We could build a “Reverse-lookup” table
  - Table that matches phone numbers to names
- How can we do this using generic for and pairs?
tRev = {}
for key, val in pairs(t) do
    tRev[val] = key
end
Generic For statement

- `ipairs()` is like `pairs`, but it only iterates through integer values.
- Starts with 1, keeps going until a nil is hit.
- Ignores non-integer values, and stops for gaps.
- (examples)
Generic For statement

Fun with for statement and function tables ...

t = {}
t[function(x) return 2*x end] = function(x) return x + 1 end
t[function(x) return x + 1 end] = function(x) return x + 2 end

for x,y in pairs(t) do print(x(y(10))) end
for x,y in pairs(t) do print(y(x(10))) end
11-60: Local Variables

- Variables can be declared as local
- Local variables are only live for the scope in which they are contained
  - Function body
  - Loop body
  - ... etc
Local Variables

function sumTable(t)
    local sum = 0
    for key, val in pairs(t) do
        sum = sum + val
    end
    return sum
end
If we have the following file sum.lua:

```lua
local x = 0
for i = 1, 10 do
    x = x + i
end
print(x)
```

and we call `dofile('sum.lua')`, it will print out 55 (as expected)
However, if we type this into the interpreter:

```plaintext
> local x = 0
> for i = 1, 10 do
  >> x = x + i
  >> end
ERROR!
```

Because `local x = 0` is local to just that statement that was typed in
function foo(<params>)
  <body>
end

equivalent to
foo = function(<params>)
  <body>
end
Functions

- If we pass too many arguments to a function, the extra ones are ignored.
- If we don’t pass enough arguments to a function, the extra ones get the value nil.

```
function test(a,b,c) print(a,b,c) end

test(1)
test(1,2)
test(1,2,3)
test(1,2,3,4)
```
We can use this behavior to get default parameters

```
function printArray(array, startIndex)
    startIndex = startIndex or 1
    while(array[startIndex]) do
        print(array[startIndex])
        startIndex = startIndex + 1
    end
end
```
Functions can return multiple values

```plaintext
function fib(n)
    if n == 1 or n == 2 then
        return 1, 1
    end
    prev, prevPrev = fib(n - 1)
    return prev+prevPrev, prev
end

x = fib(10)
print(x)
```
11-68: Functions

- If a function returns multiple values, and is passed as the last parameter to another function, then the return values are used to fill in the parameter list (best seen with an example)

```plaintext
function two() return 1,2 end
function add(a, b) return a + b end
function add3(a, b, c) return a + b + c end

print(add(two()))
print(add(two(),10))
print(add3(1,two()))
print(add3(two(),1)) -- ERROR! Why?
```
11-69: Functions

- Putting an extra set of parenthesis around a function discards all but the first return value

function two() return 1, 2 end
x,y = two()
print(x,y)
x,y = nil, nil
x,y = (two())
print(x,y)
Putting an extra set of parenthesis around a function discards all but the first return value

function two() return 1, 2 end
function goodtwo() return two() end
function badtwo() return (two()) end
print(goodtwo())
print(badtwo())
print((goodtwo()))
11-71: **Unpack**

- Built-in function `unpack` converts an array (table with integer keys) into multiple values:

```lua
function unpack (t, i)
    i = i or 1
    if t[i] ~= nil then
        return t[i], unpack(t, i+1)
    end
end
```
x = {"first", "second", "third"}
a, b, c = unpack(x)
print(a)
print(b)
print(c)
We can use unpack to call a function using an array as a parameter list

```python
function AddThree(a, b, c)
    return a + b + c
end

x = {5, 4, 2}
print(AddThree(unpack(x)))
```
We can write functions that take a multiple number of parameters (like built-in print function)

Use ... as the parameter list, parameters are collected into an array named \texttt{arg}

```lua
sum = function(...) 
  local result = 0 
  for i,v in ipairs(arg) do 
    result = result + v 
  end 
  return result 
end
```
11-75: Function Parameters

• We can combine “standard” parameters with the ... syntax

```python
function foo(a, b, ...)
    print("a =", a)
    print("b =", b)
    print("rest = ", unpack(arg))
end

foo(1)
foo(1,2)
foo(1,2,3)
foo(1,2,3,4)
```
Functions are first-class values - can be passed as parameters to other functions.

Built in function `table.sort`, takes as parameters a table (really, and array) and a function, sorts the table using the function.

```javascript
database = {
    { name = "Smith, John", ID = 3122475 },
    { name = "Doe, Jane", ID = 4157214 },
    { name = "X, Mister", ID = 0 },
}

table.sort(database, function(a,b) return a.name < b.name)
```
database = {
  { name = "Smith, John", ID = 3122475 },
  { name = "Doe, Jane", ID = 4157214 },
  { name = "X, Mister", ID = 0 },
}

table.sort(database, function(a,b) return a.ID < b.ID)
names = {"peter", "paul", "mary" }
gpa = {mary=3.8, paul = 3.2, peter = 3.5}

table.sort(names, function(a,b) return gpa[a] < gpa[b] end)
11-79: Closures

- We can declare functions within functions
- The “nested function” can see everything local to the enclosing function

```lua
function newCounter()
    local i = 0
    return function()
        i = i + 1
        return i
    end
end

c = newCounter()
print(c())
print(c())
c2 = newCounter()
pint(c2())
pint(c())
pint(C)
```
function newCounter()
    local i = 0
    return function()
        i = i + 1
        return i
    end
end

- i is not a global variable
- i is an external local variable, or upvalue
- but i is out of scope before the counter functions are called!
A closure is everything a function needs to function properly:

- Function code
- All upvalues

Under the hood, we are returning not just a function, but a closure:

- Including the value of \( i \), which can be modified
11-82: Function Tables

- Functions are first-class values
- We can store them in tables

MyLib = {}
MyLib.add = function(a,b) return a + b end
MyLib.mult = function(a,b) return a * b end

a = MyLib.add(3,4)
b = MyLib.mult(5,7)
Functions are first-class values

We can store them in tables

MyLib = {}
MyLib["add"] = function(a,b) return a + b end
MyLib["mult"] = function(a,b) return a * b end

a = MyLib.add(3,4)
b = MyLib.mult(5,7)
Functions are first-class values
We can store them in tables

MyLib = {}
function MyLib.add(a,b) return a + b end
function MyLib.mult(a,b) return a * b end

a = MyLib.add(3,4)
b = MyLib.mult(5,7)
Iterators

- We can use closures to create iterators
- Recal that iterators need to keep track of some internal data – where in the list we are
- Closures can do that for us
function rev_itr(l)
    local i = 1
    while (l[i]) do
        i = i + 1
    end
    return function()
        i = i - 1
        if l[i] then return l[i] end
    end
end
11-87: **Iterators**

```python
iter = rev_itr(l)
while true do
    local elem = iter()
    if elem == nil then break end
    print(elem)
end
```
iterators

iter = rev_itr(l)
while true do
    local elem = iter()
    if elem == nil then break end
    print(elem)
end

for elem in rev_iter(l) do
    print(elem)
end
Iterators

- The generic for loop is actually a little more complicated
  - We’re not using all of the features of the for loop
  - Technically, the iterator returns an iterator function, and a data structure to be iterated over, and the current element
  - The iterator function also takes as input parameters the data structure and current element
  - We’re using closures to handle these values – magic of functions being OK with too many / not enough arguments makes it all work
In an object-oriented programming language, what is an object?
Objects

- In an object-oriented programming language, what is an object?
  - Collection of data and methods
- How could we “fake” objects in Lua?
In an object-oriented programming language, what is an object?
- Collection of data and methods

How could we “fake” objects in lua?
- Tables can contain anything
- Create a table that contains both objects and data
11-93: **Objects**

- Simple Object example: 3D Vector
  - Stores 3 points, x, y,z
  - Method to print out vector
  - Method to add to vectors
  - Way to create a vector (table that contains all of the appropriate values)
Objects

- We will create a vector “class”
  - Table that contains all necessary functions
  - Method to create a new vector, with all necessary methods
  - Since lua doesn’t “know” about classes, we’ll have to pass the “this” pointer explicitly
Vector3 = { }

Vector3.print = function(v)
    print("( " .. v.x .. ", " .. v.y .. ", " .. v.z .. ")")
end

Vector3.add = function(v1, v2)
    return Vector3.create(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z)
end

Vector3.create = function(x, y, z)
    local v = { }
    v.x = x
    v.y = y
    v.z = z
    v.print = Vector3.print
    v.add = Vector3.add
    return v
end
We need to pass in the “this” pointer explicitly, which is a little cumbersome:

```lua
> v1 = Vector3.create(1,2,3)
> v1.print(v1)
(1, 2, 3)
```

Lua gives us some (more!) syntactic sugar

```
foo:bar(x,y) == foo.bar(foo,x,y)
```
So, with nothing more than tables, we get object oriented programming!

- Need to pass in the “this” pointer explicity, but : syntax helps out with that
- Creating an object is a little cumbersome – need to copy all of the methods and values over by hand
- Can’t do fun things like overload +, - for vectors
- Can fix last two issues using metatables
What will the following do?

```lua
> v1 = Vector3.create(1, 2, 3)
> v2 = Vector3.create(7, 8, 9)
> v3 = v1.add(v1, v2)
> v4 = v1:add(v2)
> v5 = v1 + v2
```
11-99: **Metatables**

- What will the following do?

```language=nil
> v1 = Vector3.create(1,2,3)
> v2 = Vector3.create(7,8,9)
> v3 = v1.add(v1, v2)  -- set v3 = v1 + v2
> v4 = v1:add(v2)      -- set v3 = v1 + v2
> v5 = v1 + v2         -- Error!
```

- We can’t add tables!
- metatables to the rescue!
11-100: Metatables

- A metatable is a special table that can be stored as a subtable of a table
- Stores the function definitions for built-in functions (+, *, -, /, etc)
- If we try to add two tables, see if the first table has a metatable
  - No metatable (the default) throw an error
  - Metatable exists, no `__add` field in metatable, throw an error
  - Metatable exists, `__add` field, execute function in `__add` field
Vector3 = { }
Vector3.print = function(v)
    print("(" .. v.x .. ", " .. v.y .. ", " .. v.z .. ")")
end
Vector3.add = function(v1, v2)
    return Vector3.create(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z)
end
Vector3.mt = { __add = Vector3.add }
Vector3.create = function(x, y, z)
    local v = { }
    v.x = x
    v.y = y
    v.z = z
    v.print = Vector3.print
    v.add = Vector3.add
    setmetatable(v,Vector3.mt)
    return v
end
11-102: Metatables

- One more piece that can help use out
- What happens when we try to access a field that doesn’t exist in a table?
  - Check themetatable to see if there is an `__index` field
  - If not, return nil
  - If `__index` field exists and stores a table, look in that table for the value