making games with lua and love

bob somers 2011

overview

Why is Lua awesome?

What is LÖVE?

How do I write Lua code?

Game programming paradigms!

How do I use LÖVE?

lua is awesome

Lua is **multi-paradigm** scripting language.

Simplicity and size.

Extensibility and embedding.

Efficiency and speed.

Can run (almost) anywhere. For realz.

what is love

Not a 1993 dance song by Haddaway.

An open source, cross platform, **2D game framework** for the Lua language.

The best open source libraries behind a simple, coherent interface.

Graphics, sound, file loading, device I/O, physics, and more.

in a nutshell

Lua makes it simple to program.

LÖVE makes is simple to program games.

how do i write

teh luas

lua basics

Lua code goes in a ***.lua file**.

Here's hello world, in **hello.lua**:

```
print("Hello, CPGD!")
```

lua basics

Variables don’t need to be declared, **just use them.**

No semicolons are necessary, unless you’re insanely attached to them.

```
age = 2011 – 1987
print(“Bob is “ .. age .. “ years old.”)
```

lua basics

Those double dots are the **string concatenation** operator**.**

Go easy with it, we'll see why later.

```
age = 2011 – 1987
print(“Bob is “ .. age .. “ years old.”)
```

lua basics

Variables are can be one of **eight types**.

You only need to know nil, booleans, numbers, strings, functions, tables.

You never need to declare a variable's type. Lua will figure it out.

lua basics

Nil means “null” or “not used”.

Numbers are integers **and** floating point.

Strings are “obviously strings”.

Booleans are **true** or **false**.

Functions are **first class values** in Lua.

lua basics

Tables are Lua's **only data structure**.

But they're so insanely flexible that you can do **nearly anything** with them.

More about tables later...

lua basics

You can explicitly convert to a particular type with the **to*() functions**.

```
-- age is a string
age = “23”

-- now age is a number
age = tonumber(age)
```

lua basics

The double dash is a **comment**.

A **block comment** adds two square brackets.

```
-- age is a string
age = “23”

--[[ commented out!
age = tonumber(age)
]]
```

lua basics

Arithmetic and relational operators are just like you would expect.

```
+ - * / ^ %
< > <= >= == ~=
```

Except that **not equal** is tilde-equal!

Logical operators are words.

```
not and or
```

lua basics

Conditionals are not surprising.

```
if age < 18 then
     print(“a youngin!”)
elseif age >= 18 or age <= 25 then
     print(“a poly student?”)
else
     print(“hopefully graduated!”)
end
```

Note the **then’s**, single world **elseif**, and final **end** keyword.

lua basics

While loops aren't surprising either.

```
while age < 25 do
      print("work work work")
end
```

Note the **do** and **end** marking the extents of the loop.

lua basics

For loops come in **numeric** and **generic** flavors.

The numeric flavor is straightforward.

```
for i = 1, 10 do
    print(i)
end

-- prints 1 through 10
```

lua basics

The three parts are the **start**, **end**, and optional **step size**.

```
-- prints my odd birthdays
for i = 1, age, 2 do
    print(i)
end
```

All three are evaluated **only once** before the loop begins.

lua basics

The three parts are the **start**, **end**, and optional **step size**.

```
-- prints my odd birthdays
for i = 1, age, 2 do
    print(i)
end
```

All three are evaluated **only once** before the loop begins.

lua basics

We'll mention the **generic** form after tables.

All loop types can use **stop looping** with the **`break`** statement.

Unfortunately there is no **`continue`**.

lua basics

Functions are pretty straightforward.

```
function calcAge(year)
     return year - 1987
end
```

Note that functions don't need a **do**.

lua basics

They can return multiple values.

```
function greatBook()
     return “the wisdom of crowds”,
                 “james surowiecki”
end

title, author = greatBook()
```

lua basics

They're first class values.

```
function nextPurchase(book)
     title, author = book()
     print("buy: " .. title)
end

nextPurchase(greatBook)

-- prints 'buy: the wisdom of crowds'
```

lua basics

A note about variable scope...

Variables are **global by default**, but can be made local to the scope of first use with the **`local`** keyword.

lua basics

```
age = 1

function calcAge(year)
     local birthyear = 1987
     age = year - birthyear
end

calcAge(2011)

-- age has been changed to 23
-- birthyear is NOT visible out here!
```

lua basics

Tables are **associative arrays**, or basically a key-value store.

Any Lua type except nil can be a key.

Any Lua type including nil can be a value.

lua basics

They're created with curly braces...

```
point = {}
```

...and indexed with square brackets.

```
point["x"] = 42
point["y"] = 10

print(point["x"])
```

lua basics

The **record syntax** makes string keys nicer.

```
point = {}
point[“x”] = 42
point.y = 10

print(point.x)
print(point[“y”])
```

Either way, **`tab[“abc”]`** and **`tab.abc`** mean **the same thing.**

lua basics

If you index with integers, you have **an array**.

```
tens = {}

for i = 1, 10 do
    tens[i] = 10 * i
end
```

For historical reasons, Lua arrays are **one-based**, not zero based.

lua basics

The hash character is shorthand to **get the length** of a one-based table array.

```
tens = {}
for i = 1, 10 do
    tens[i] = 10 * i
end

for i = 1, #tens do
    print(tens[i])
end
```

lua basics

But it only counts **consecutive numeric keys**, starting at 1.

```
vals = {}
vals[1] = 42
vals[2] = 100
vals[3] = 9001
vals.name = “Edward Cullen”
vals.gender = “Unsure”

print(#vals) -- prints 3
```

lua basics

You can use it cleverly to easily add items to the end of an array.

```
vals = {42, 100, 9001}

vals[#vals + 1] = 12345

-- vals now contains the number
-- 12345 at index 4 and #vals
-- would return 4
```

lua basics

You can **initialize keys** at creation time in the curly braces.

```
vals = {
     42,       -- key is 1
     100,      -- key is 2
     9001,     -- key is 3
     name = “Edward Cullen”,
     gender = “Unsure”
}
```

lua basics

Because tables can hold any data (including functions!), it's common to use them as **structs** or **namespaces**.

```
player = {
    x = 10,
    y = 50,
    img = "run.png",
    state = "running"
}
```

lua basics

Because tables can hold any data (including functions!), it's common to use them as **structs** or **namespaces**.

```
function player.jump()
    player.y = player.y + 10
end
```

lua basics

Lua comes with a small **standard library** of useful functions.

They're all inside their respective library's tables.

```
math.sin()
table.concat()
```

Math, table, string, I/O, OS, and debug.

lua basics

Remember when I said go easy on string concatenation?

`table.concat` is more memory efficient.

```
str = “This ” .. “generates ” ..
      “a ” .. “lot ” .. “of ” ..
      “intermediate ” .. “garbage.”)

str = table.concat({“This ”, “is ”,
      “far ”, “more ”, “efficient.”})
```

lua basics

The **generic for** uses a more iterator-like syntax.

```
vals = {
     name = “Edward Cullen”,
     gender = “Unsure”
}

for key, value in pairs(vals) do
     print(key .. “ = ” .. value)
end
```

lua basics

The **pairs** function is an iterator for all keys, while **ipairs** is an iterator for just integer indices.

```
vals = {42, 100, 9001, 12345}

for i, value in ipairs(vals) do
    print(i .. “ = ” .. value)
end

-- prints ‘1 = 42’, etc.
```

lua basics

Lua is silent on the issue of **object oriented programming.**

It's easy to use **tables**, which can hold both **data** and **functions** as objects.

The **colon syntax** gives us some nice syntactic sugar.

lua basics

```
player = {}

function player:move(x, y)
	self.x = x
	self.y = y
end

function player.move(obj, x, y)
	obj.x = x
	obj.y = y
end
```

lua basics

We can use a Lua feature called **metatables** to simulate **inheritance**.

We won't talk about it. It's boring.

There are many ways to **do classes** in Lua.

We'll use a helper library later and forget about the details.

game programming

parawats

paradigms

The game loop.

Vectors ftw.

Coordinate frames.

State machines.

paradigms

The game loop is an **infinite loop** where the state of things in the game is **continuously updated** and drawn to the screen.

paradigms

The **update** loop and **draw loop** are usually separated.

Things in your game should **update** based on **real world time.**

The draw loop draws the state of the world as **fast as it can.**

paradigms

Vectors store **groups of related numbers**, like position, velocity, or direction.

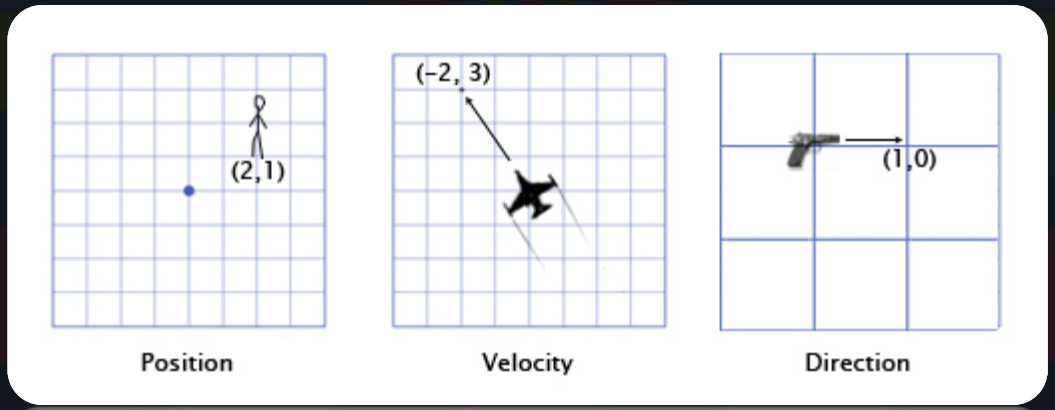


Image credit: Wolfire Games Blog (blog.wolfire.com)

paradigms

Vectors are **meaningless without units**, and can change over time.

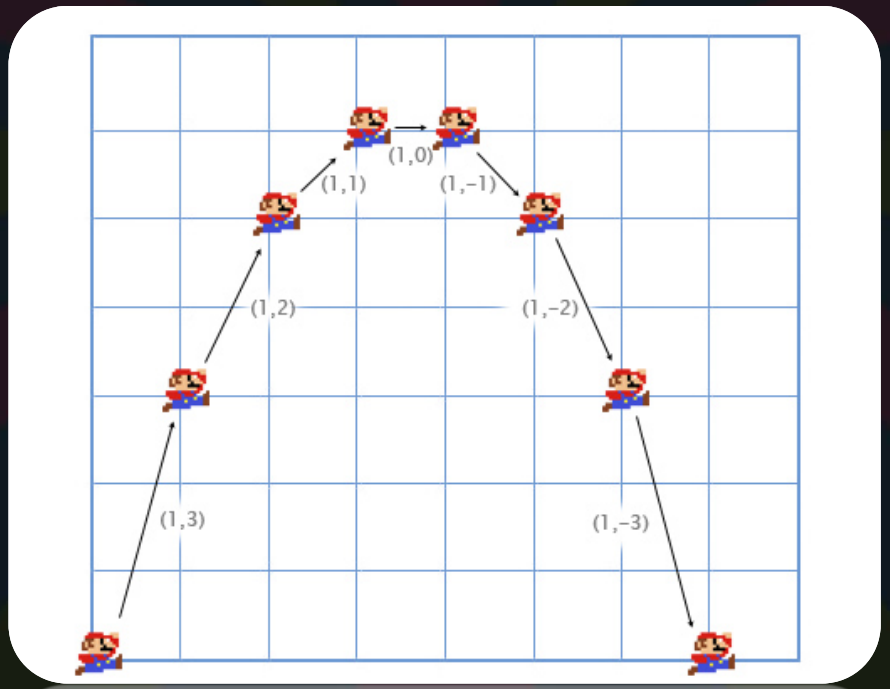


Image credit: Wolfire Games Blog (blog.wolfire.com)

paradigms

Position and direction vectors can be **added** and **subtracted**.

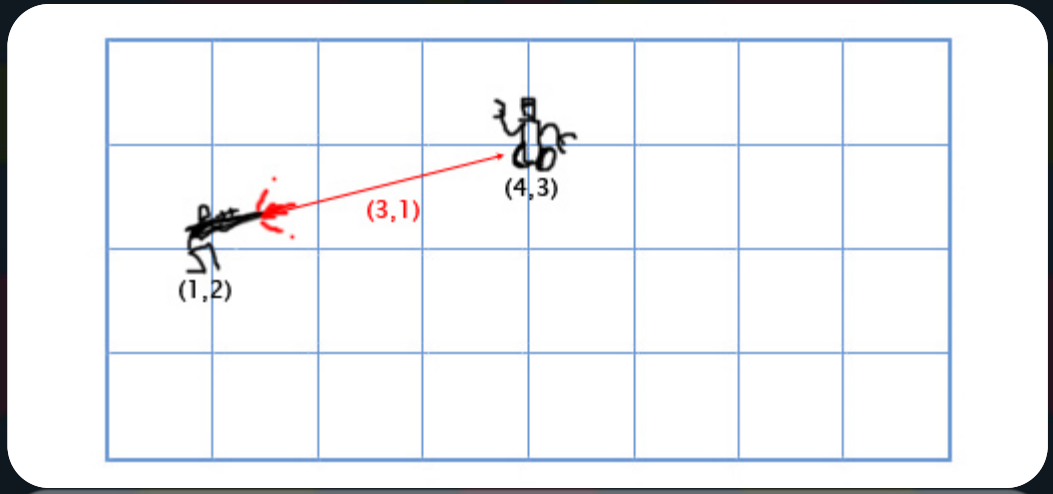


Image credit: Wolfire Games Blog (blog.wolfire.com)

paradigms

The **length** of a vector can be found using the Pythagorean theorem.

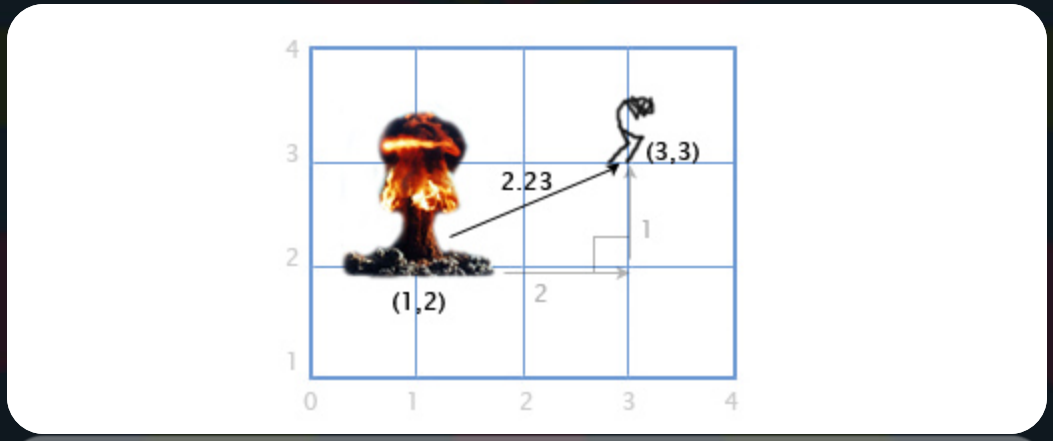


Image credit: Wolfire Games Blog (blog.wolfire.com)

paradigms

The **dot product** of two vectors gives you an idea of how they are oriented to each other.

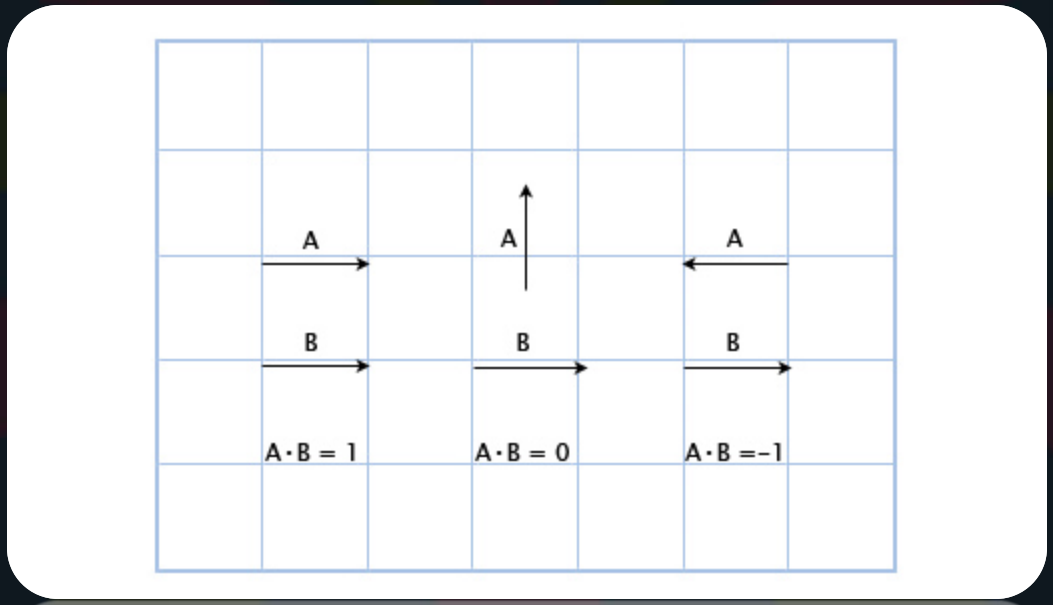
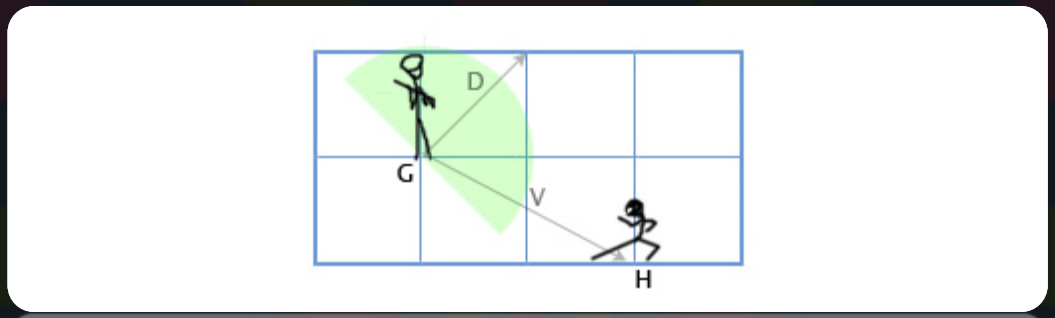


Image credit: Wolfire Games Blog (blog.wolfire.com)

paradigms

This is useful for lots and lots of things.



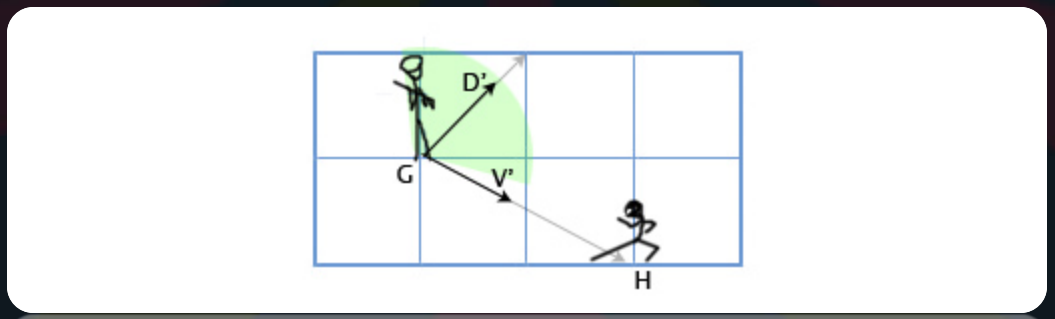


Image credit: Wolfire Games Blog (blog.wolfire.com)

paradigms

For more info on vectors and matrices, see the *Linear Algebra for Game Developers* series on the Wolfire Games Blog.

It’s really awesome.

paradigms

Coordinate frames let you describe position or motion **relative** to something else.

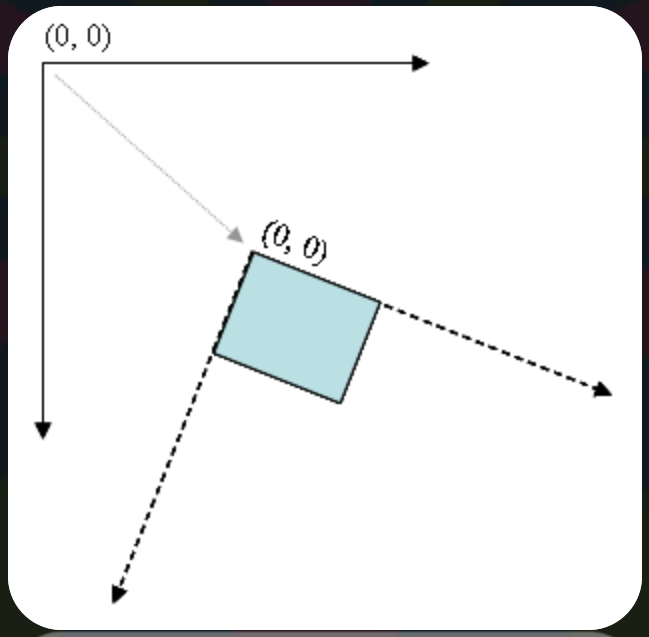


Image credit: Piccolo 2D (piccolo2d.org)

paradigms

State machines help you describe **different situations** in your game and **how you can get there**.

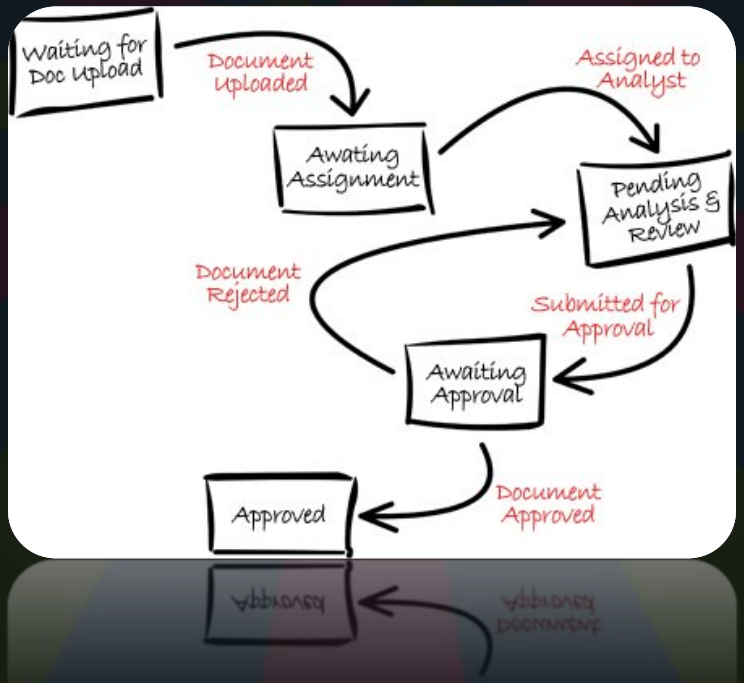


Image credit: Core.NET Blog (coredotnet.blogspot.com)

now i want to make

teh loves

love framework

To run a game, just drag its **folder** or ***.love** file onto the love executable.

Folders are great for development.

*.love files are **just *.zip files** of your game folder that have been renamed.

*.love files can also be **embedded** in the executable for distribution.

love framework

All your game's source code and assets (artwork, music, etc.) resides in the **game folder**.

It must have, at bare minimum, a **main.lua** file.

Additional configuration can be added with a **conf.lua** file.

love framework

Your game, at a minimum, should implement **three LÖVE functions**.

```
function love.load()
    -- load things!
end

function love.update(dt)
    -- update things!
end

function love.draw()
    -- draw things!
end
```

love framework

The **load function** runs once at the beginning. Use it to… load things.

```
function love.load()
    hamster =
        love.graphics.newImage(“hamster.png”)
end
```

love framework

The **update function** gives you the fractional number of real-world seconds since it last ran, or **delta time**.

```
function love.update(dt)
      -- move the ball at 100 pixels/sec
      ball.x = ball.x + 100 * dt
end
```

love framework

The **draw function** is where you use graphics to represent the **current state** of your game world.

```
function love.draw()
    love.graphics.draw(hamster, x, y)
end
```

love framework

There are other **callbacks** you can hook for other events.

```
love.mousepressed(x, y, button)
love.mousereleased(x, y, button)
love.keypressed(key, unicode)
love.keyreleased(key, unicode)
love.focus(f)
love.quit()
```

love framework

Or you can **poll them directly** if you prefer.

```
function love.update(dt)
	if love.keyboard.isDown(“left”) then
		-- go left!
	end
end
```

love framework

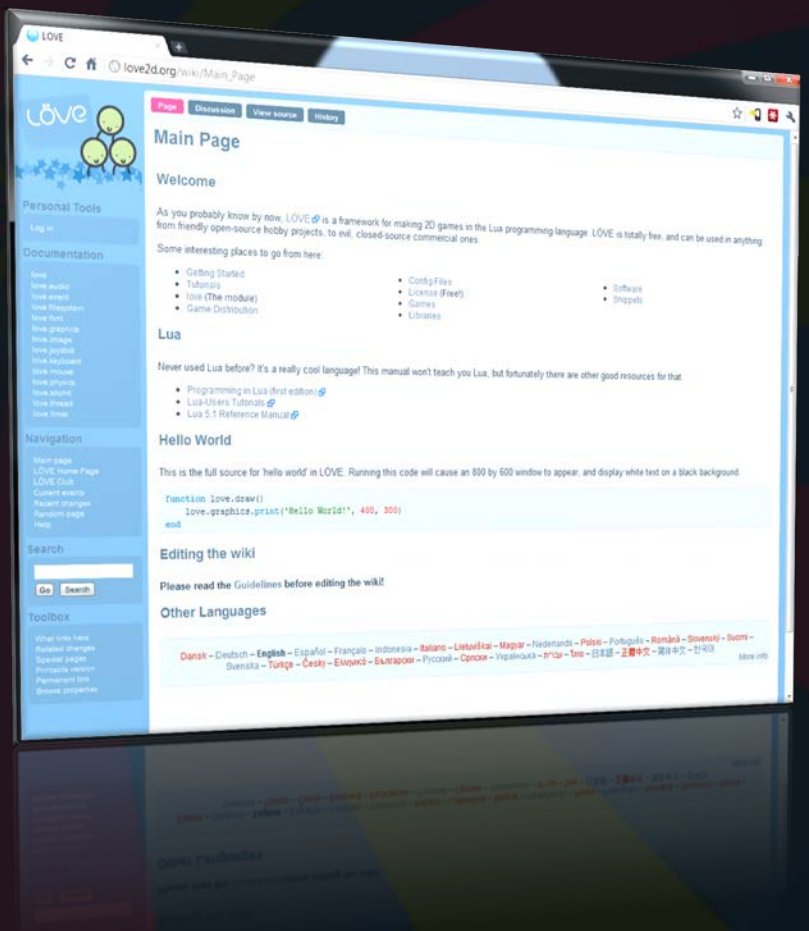
There are lots of **subsystems** in LÖVE.

```
love.audio
love.event
love.filesystem
love.font
love.graphics
love.image
love.joystick
love.keyboard
love.mouse
love.physics
love.sound
love.thread
love.timer
```

love framework

All of which are thoroughly documented on the **LÖVE wiki**.

It's at **love2d.org**



love framework

There are also **helper libraries** on the wiki for things that LÖVE does not provide.

We'll be using one today, called **Helper Utilities for More Productivty**, or HUMP.

Yes, most of them follow this... uh... naming scheme.

love framework

HUMP gives us easy to use:

- Vectors
- Timers
- Game states
- Cameras
- Classes and inheritance
- Ringbuffers

what else is there

to knows

love framework

There are lots of things we didn't talk about.

- Coroutines
- Metatables and metamethods
- Environments
- Modules
- Other ways of doing OO
- Weak tables
- The entire C API

love framework

Further resources:

Lua website at **lua.org**
(language documentation!)

LÖVE website at **love2d.org**
(wiki and forums!)

Programming in Lua
(by Roberto Ierusalimschy)


questions