LÖVE for Newbies

Sol Bekic, Ricardo Gomes

Table of Contents

Preface

> So you want to make a game?

1. The Prototype

Welcome to World 1!
Here we will build a prototype for a game, up from the ground. Let’s jump right in!

1.1. The Concept

For the Purposes of this Book, we have decided to recreate the game Asteroids.

Before we start we need to think about how exactly this game works and what we want to include. Thinking about the game before we start will keep us from getting lost in the development process or adding feature after feature and losing interest before it is done.

So, how does Asteroids work?

In asteroids, the Player controls a small spaceship that is flying around. He has to shoot asteroids that fly around the screen. If he gets hit by an asteroid, the player loses a life and has to restart the round. When he hits an asteroid, the asteroid splits into two smaller ones.

Alright, from these few rules we can already extract some things that we will need:

- a Spaceship / Player
- Asteroids
- Bullets

Next, let’s see what each of these do or can do:

The Spaceship

The spaceship is the main thing the player can control in our game. The player can use two keys to accelerate or decelerate in the direction the ship is currently facing. He can also turn the spaceship with two other keys. Lastly the player can press a button to fire a shot.

When the spaceships leaves the screen on one side, it will enter from the opposite side, keeping it’s velocity.

We can see that the spaceship needs to have a `position`, `velocity` and `rotation`. In World 1-5 we will see that the actual way of saving the information we use differs, but we could as well do with this "definition".

The Asteroids

Asteroids move in a straight line, also wrapping around the Screen in the same way the spaceship does. They larger an asteroid is, the slower it should move. Therefore all we need to know about every asteroid is his `position`, `size` and his `rotation`.

Whenever an asteroid is hit, we split it into two smaller ones, unless it is already the smallest possible size. In that case we remove it altogether.

The Bullets

Bullets travel along a straight path, much like asteroids. However they are removed from the game as soon as they hit an asteroid or leave the screen.

All bullets share the same speed, we only need to know the bullet's `position` and `rotation`. <<<

1.2. The Tools

LÖVE

So, what makes a LÖVE Project?
At core a LÖVE Project is just a folder containing everything needed to make the game; Code, Images, Sound and Video files and everything else you might need.

Code

Obviously, a game contains code. LÖVE Games are programmed in Lua, which you should be familiar with already. Generally Lua Source Files (`.lua`) can lay around anywhere in the project directory and have arbitrary names… except for two special ones: `main.lua` and `conf.lua`.

These two are the only files that the LÖVE Framework runs; they are the starting points of every game or project you build.

main.lua

As the name implies this file will contain all your **main code** - what exactly that will be and how you organize your code is up to you. Usually this file contains the used *Callback Routines*, which will be covered in the next Level. For smaller projects and the next levels in this book this will be the only file (except for `conf.lua`) you will need.

conf.lua

conf is short for **Conf**iguration, and thats what `conf.lua` is all about. You can fill this file with a function called `love.conf( t )` that accepts a table as it's only parameter. In that function you can then modify certain fields of the table and thereby change the configuration the LÖVE Framework uses when it first creates your window.

Here is a function that sets every possible value to it's default value - and thereby does nothing:

Full `conf.lua` example

```lua
function love.conf(t)
    t.identity = nil                   -- The name of the save directory (string)
    t.version = "0.9.1"                -- The LÖVE version this game was made for
(string)
    t.console = false                  -- Attach a console (boolean, Windows only)

    t.window.title = "Untitled"        -- The window title (string)
    t.window.icon = nil                -- Filepath to an image to use as the window's
icon (string)
    t.window.width = 800               -- The window width (number)
    t.window.height = 600              -- The window height (number)
    t.window.borderless = false        -- Remove all border visuals from the window
(boolean)
    t.window.resizable = false         -- Let the window be user-resizable (boolean)
    t.window.minwidth = 1              -- Minimum window width if the window is
resizable (number)
    t.window.minheight = 1             -- Minimum window height if the window is
resizable (number)
    t.window.fullscreen = false        -- Enable fullscreen (boolean)
    t.window.fullscreentype = "normal" -- Standard fullscreen or desktop fullscreen
mode (string)
    t.window.vsync = true              -- Enable vertical sync (boolean)
    t.window.fsaa = 0                  -- The number of samples to use with multi-
sampled antialiasing (number)
    t.window.display = 1               -- Index of the monitor to show the window in
(number)
    t.window.highdpi = false           -- Enable high-dpi mode for the window on a
Retina display (boolean). Added in 0.9.1
    t.window.srgb = false              -- Enable sRGB gamma correction when drawing to
the screen (boolean). Added in 0.9.1

    t.modules.audio = true             -- Enable the audio module (boolean)
    t.modules.event = true             -- Enable the event module (boolean)
    t.modules.graphics = true          -- Enable the graphics module (boolean)
    t.modules.image = true             -- Enable the image module (boolean)
    t.modules.joystick = true          -- Enable the joystick module (boolean)
    t.modules.keyboard = true          -- Enable the keyboard module (boolean)
    t.modules.math = true              -- Enable the math module (boolean)
    t.modules.mouse = true             -- Enable the mouse module (boolean)
    t.modules.physics = true           -- Enable the physics module (boolean)
    t.modules.sound = true             -- Enable the sound module (boolean)
    t.modules.system = true            -- Enable the system module (boolean)
    t.modules.timer = true             -- Enable the timer module (boolean)
    t.modules.window = true            -- Enable the window module (boolean)
    t.modules.thread = true            -- Enable the thread module (boolean)
end
```

NOTE You don’t need to use a `conf.lua` or specify every key in the conf table; everything you leave out will remain at it’s default value.

You will mostly be using this to set a different resolution for your game and set the game title.

Usual `conf.lua`

```lua
function love.conf( t )
    t.identity      = "GtGLG"
    t.version       = "0.9.1"

    t.window.title  = "Gary, the green-legged Giraffe"
    t.window.width  = 1200
    t.window.height = 720

    t.window.fsaa   = 4
    t.window.vsync  = true
end
```

Other files

Everything else will need to be `require`d by `main.lua` in some way (direct or indirect).

Images, Animations, Sounds and other Assets

All of these files need to be somewhere in the project directory aswell. You will learn to load and draw or play these files throughout this World.

Even though you can just have all the files in one directory, it is advised that you structure your files in a logical hierarchy, for example like this:

```
- mygame/
    + main.lua
    + conf.lua
    + lib/
        + library1.lua
        + library2.lua
        + sometool.lua
    + assets/
        + images/
            + player.png
            + rock.png
        + sounds/
            + impact.wav
            + menumusic.mp3
        + videos/
            + intro.mp4
```

1.3. Starting Small

Alright, now that we now the file structure, we can get started on the actual code.

Drawing Circles

In the last Level we learned that our main code goes in a file called `main.lua` and that we are supposed to define *Callback Routines* there.

A *Callback Routine* is nothing but a function with a specific name the LÖVE framework knows. For example, whenever a key is pressed, LÖVE will attempt to call a function called `love.keypressed`. If you have not written than function (yet), the keypress will be ignored, but if you choose to write one, you can do something in reaction to that *Event*.

One of the most important Callbacks is `love.draw`. As the name implies, this is where any drawing to the screen should take place.

main.lua

```
function love.draw()
  -- draw a sparkling unicorn
end
```

Inside this function we can now draw everything we want the player to be able to see. To get started, let's draw a circle. To draw a circle, we will use the function `love.graphics.circle`. Go and see what you can find on the wiki page now!

NOTE

The wiki is **the** most important resource for programming in LÖVE. You can find information and examples for all available functions there.

When you get stuck somewhere, first consult the wiki. If you cannot find a solution there, **search the forum**. If you *still* do not find anything, start a new thread in the **Support & Development** forum. Provide a clear, concise title and *be patient!*

As you can see, `love.graphics.circle` accepts five parameters:

- mode, whether to *fill* or *stroke* the circle (`"line"` is used to stroke)
- x, the horizontal position of the circle's center
- y, the vertical position
- radius, the circle's radius (half of the width)
- segments, how detailed to draw the circle. This can be left out.

So, to draw a filled 30px circle at (100, 100) and then a stroked one 20px further down, this is what we need to do:

main.lua

```lua
function love.draw()
  love.graphics.circle("fill", 100, 100, 15)
  love.graphics.circle("line", 100, 120, 15)
end
```

Quite similarly, we can also draw rectangles using `love.graphics.rectangle`. The six parameters are:

- mode, same as for `circle`
- x, this time it is not the center position of the top-left corner
- y, again, relative to the top left corner
- width, the width of the rectangle being drawn
- height, the height of the rectangle

Another function you will use in probably every game is `love.graphics.setColor`. It accepts three numbers in the range of `0` to `255`, specifying the intensity of red, green and blue respectively. A fourth value, used as alpha (transparency) can optionally be supplied.

The last drawing function for now is `love.graphics.line`. You can pass it four values, each representing the x and y values of a point.

Putting it all together, we can draw a little stickman:

main.lua

```lua
function love.draw()
  love.graphics.setColor(255, 255, 255)           -- white
  love.graphics.line(40, 90, 35, 120)             -- left foot
  love.graphics.line(60, 90, 65, 120)             -- right foot

  love.graphics.setColor(0, 255, 0)               -- green
  love.graphics.rectangle("fill", 30, 30, 40, 60) -- body

  love.graphics.setColor(255, 0, 0)               -- red
  love.graphics.circle("fill", 50, 20, 15)        -- head
end
```

NOTE Note how the head is in front of and the feet are behind the body because of the order we chose to draw them in.

LÖVE provides a lot more drawing operations and related functions, but for now we will stick with those above. You can find the whole list on the `wiki`. <<<

1.4. Creating Motion

Last time we managed to draw a terrible-looking stickman, but he seemed rather… lifeless. If we want to make a game, it might be a good idea to have things moving around.

A Ship

As we are working towards an Asteroids clone, let's start out with a spaceship. Before we can make a spaceship move, we need a spaceship though:

main.lua

```lua
function love.draw()
  love.graphics.setColor(200, 200, 200)
  love.graphics.rectangle("fill", 20, 20, 100, 20)
  love.graphics.setColor(80, 80, 80)
  love.graphics.line(70, 30, 120, 30)
end
```

I know, I know, that doesn't look very aerodynamic. But I'll tell you what, there's no air resistance in space anyway - at least there's a line to indicate which direction we are facing.

An Engine

Alright, so how do we make this thing move? We will obviously need a variable to keep track of it's current position. In fact we will be using two, one for each coordinate we are going to simulate the ship on:

main.lua

```lua
x, y = 20, 20

function love.draw()
  love.graphics.setColor(80, 80, 80)
  love.graphics.rectangle("fill", x, y, 100, 20)
  love.graphics.setColor(200, 200, 200)
  love.graphics.line(x+20, y+10, x+100, y+10)
end
```

Now we only need to change the values of `x` and `y` and the ship will follow. We could for example make them always point to the current mouse position:

main.lua

```lua
x, y = 20, 20

function love.update(dt)
  x, y = love.mouse.getPosition()
end

function love.draw()
  love.graphics.setColor(80, 80, 80)
  love.graphics.rectangle("fill", x, y, 100, 20)
  love.graphics.setColor(200, 200, 200)
  love.graphics.line(x+20, y+10, x+100, y+10)
end
```

Wait, what happened here? First of all, I introduced a new LÖVE callback: `love.update`. As the name implies, this is where we *update* the current game state. `love.update` is where things *happen*; This is where the player moves, the enemies shoot and the lighting is calculated. One thing you might notice immediately is that unlike `love.draw`, `love.update` receives a parameter: `dt`.

`dt` is short for **d**elta**time** (*time difference*). `dt` is always the time (in seconds) since the last time `love.update` was called. In this small example `dt` is not used, but it is very useful, as we will see later on.

Secondly, another LÖVE function has come into play: `love.mouse.getPosition`. It should again be rather obvious what this function is doing for us; it returns the mouse's current position along the x and y axis (in pixels).

Keyboard Controls

Alright, now things are moving, but in our game we don't want the ship to be controlled by the mouse. Also, with the current code players with a faster mouse would have an advantage, they could move around much more quickly. Instead, let's introduce a fixed speed for our spaceship, and add keyboard controls:

main.lua

```lua
player = {
  x = 20,
  y = 20
}

SPEED = 300

function love.update(dt)
  if love.keyboard.isDown("right") then
    player.x = player.x + SPEED*dt
  end
  if love.keyboard.isDown("left") then
    player.x = player.x - SPEED*dt
  end
  if love.keyboard.isDown("down") then
    player.y = player.y + SPEED*dt
  end
  if love.keyboard.isDown("up") then
    player.y = player.y - SPEED*dt
  end
end

function love.draw()
  love.graphics.setColor(80, 80, 80)
  love.graphics.rectangle("fill", player.x, player.y, 100, 20)
  love.graphics.setColor(200, 200, 200)
  love.graphics.line(player.x+20, player.y+10, player.x+100, player.y+10)
end
```

Great, let's go through the changes: You can immediately see that `x` and `y` have disappeared, instead there is now a table called `player.` `player` stores the position under the keys `"x"` and `"y"`, so nothing has really changed except the varible names are longer, but I have done this so our code stays readable as we progress; As we add enemies and bullets we will need to keep track of a lot more `x`s and `y`s.

As we can see, I have also added a new variable called `SPEED`. `SPEED` is never changed in the code, so I could've also replaced every occurence further down with it's value, `300` right away. However keeping it at the top like this makes it easy to modify the speed, also I cannot forget to modify it in all the places should I change it some time. `SPEED` is uppercased because I use this style to name constant values, but this is just a stylistic choice by me.

Now on to the actual keyboard code! There is another new LÖVE function used here: `love.keyboard.isDown`. Once again the name should make the purpose of the function very clear, you should start to see a pattern here. `love.keyboard.isDown` checks if the key given is down. In LÖVE keys are identified by a string, in this example we are using `"up"`, `"down"` etc., but there are many more. You can view a table of all the keys on the wiki: `KeyConstant`.

Using `if` and `love.keyboard.isDown` we add or substract `SPEED * dt` from the fitting coordinates, when the *left* key is pressed, we substract from `player.x`; when *down* is pressed we add to the y-coordinate.

Now, why do we multiply `SPEED` with `dt`? As I said above, the time step between each `love.update` execution may vary; for example a scene that has to draw a lot of enemies might take longer do draw than one that just features a few. Other factors like the power of the graphics card and processor or how busy the computer is with other things might also impact the updaterate.

It is important that we care about this, a game that runs twice as fast on better hardware is unacceptable. By multiplying with `dt` we can scale the speed by the time that we are actually simulating.

NOTE This means we are also always "lagging a frame behind" in update-time, but that doesn't really make a difference in practice.

Because `dt` is measured in seconds, it also makes specifying movement speeds etc. very convenient, `SPEED` is now measured in **pixels per second**. This gives it a workable size and something you can think about; if we had not used `dt` we would have to deal with tiny values in **pixels per frame**. <<<

1.5. Better Steering

Great, so now we can fly around! A little realism would be nice though, spaceships don't really fly like that. It is time too look back to World 1-1 and look at our notes for the spaceship:

> The spaceship is the main thing the player can control in our game. The player can use two keys to accelerate or decelerate in the direction the ship is currently facing. He can also turn the spaceship with two other keys. Lastly the player can press a button to fire a shot.
>
> When the spaceships leaves the screen on one side, it will enter from the opposite side, keeping it's velocity.
>
> We can see that the spaceship needs to have a `position`, `velocity` and `rotation`. in World 1-5 we will see that the actual way of saving the information we use differs, but we could as well do with this "definition".
>
> — World 1-1, The Spaceship

Okay, so let's get to work:

main.lua

```lua
player = {
  x = 150,
  y = 150,
  xvel = 0,
  yvel = 0,
  rotation = 0
}

local ANGACCEL      = 4
local ACCELERATION  = 20

function love.update(dt)
  if love.keyboard.isDown"right" then
    -- rotate clockwise
    player.rotation = player.rotation + ANGACCEL*dt
  end
  if love.keyboard.isDown"left" then
    -- rotate counter-clockwise
    player.rotation = player.rotation - ANGACCEL*dt
  end
  if love.keyboard.isDown"down" then
    -- decelerate / accelerate backwards
    -- (left out for now)
  end
  if love.keyboard.isDown"up" then
    -- accelerate
    -- (left out for now)
  end
end

function love.draw()
  love.graphics.setColor(80, 80, 80)
  love.graphics.translate(player.x, player.y)
  love.graphics.rotate(player.rotation)
  love.graphics.rectangle("fill", -50, -10, 100, 20)
  love.graphics.setColor(200, 200, 200)
  love.graphics.line(20, 0, 50, 0)
end
```

Okay, some changes. We now store values for `xvel`, `yvel` and `rotation` in our `player` table. `x` and `y` are still used to track the current position, but our `love.draw` has changed, `x` and `y` now refer to the center of the spaceship. This is important so that the rotation looks realistic and the physics we introduce later work as expected.

To draw the spaceship, we now use two new functions: `love.graphics.translate` and `love.graphics.rotate`.

Using them is pretty straight-forward, `love.graphics.translate` moves everything that is drawn after it is called by the amount in `x` and `y`, `love.graphics.rotate` rotates everything around the current point (0,0). These two functions are called **transformation functions** because they **transform** everything that is drawn afterwards. There are some more, like `love.graphics.scale`, but these will do for now.

Note the order in which we call them, it is very important for the correct result! You can try this by yourself: starting from the same point, try rotating 90° to the left, then walking two steps, then do those two movements in the opposite order. You will end up in two different places.

Our new `love.update` is also really simple for now, we just modify `rotation`. I have introduced a new constant `ANGACCEL` (**ang**ular **accel**eration). This is the angle that it should turn per second, note the use of `dt`, just like in World 1-4.

Okay, so what about `xvel` and `yvel`? I skipped the actual *move*ment above, but those are the **vel**ocities along the x- and y-axis. We will store them seperately so that the player can drift around (if we only stored the total velocity and assumed that it was always pointing the same way the ship does then it would behave more like a car, with full traction).

To implement those two left-out pieces of code we need some very basic trigonometry. When the `up` key is pressed, we want to add velocity in the direction the ship is currently pointing. When we press `down` the opposite should happen.

We will need to use `math.sin` and `math.cos` to calculate the parts of the acceleration that apply to each axis, based on the angle the ship is rotated to:

main.lua

```lua
player = {
  x = 150,
  y = 150,
  xvel = 0,
  yvel = 0,
  rotation = 0
}

local ANGACCEL      = 4
local ACCELERATION  = 400

function love.update(dt)
  if love.keyboard.isDown"right" then
    -- rotate clockwise
    player.rotation = player.rotation + ANGACCEL*dt
  end
  if love.keyboard.isDown"left" then
    -- rotate counter-clockwise
    player.rotation = player.rotation - ANGACCEL*dt
  end
  if love.keyboard.isDown"down" then
    -- decelerate / accelerate backwards
    player.xvel = player.xvel - ACCELERATION*dt * math.cos(player.rotation)
    player.yvel = player.yvel - ACCELERATION*dt * math.sin(player.rotation)
  end
  if love.keyboard.isDown"up" then
    -- accelerate
    player.xvel = player.xvel + ACCELERATION*dt * math.cos(player.rotation)
    player.yvel = player.yvel + ACCELERATION*dt * math.sin(player.rotation)
  end
  player.x = player.x + player.xvel*dt
  player.y = player.y + player.yvel*dt
  player.xvel = player.xvel * 0.99
  player.yvel = player.yvel * 0.99
end

function love.draw()
  love.graphics.setColor(80, 80, 80)
  love.graphics.translate(player.x, player.y)
  love.graphics.rotate(player.rotation)
  love.graphics.rectangle("fill", -50, -10, 100, 20)
  love.graphics.setColor(200, 200, 200)
  love.graphics.line(20, 0, 50, 0)
end
```

NOTE Note how again `ACCELERATION` is multiplied by `dt` before being scaled by `math.cos` and `math.sin` for the corresponding axis.

This already feels a lot like Asteroids!

2. The Redo

So that was The Prototype. In the last few levels we went from our idea to an actual playable game… so what's left to do?

Well, it turns out there is *a lot!*

Even though the game so far is perfectly playable, some of the code has gotten messy during the development process. Also we haven't added any real art, sound effects or menu yet - everything that makes your game visually appealing to users is missing. This is because in The Prototype we only created a *prototype*, a smaller version of the game to explore the concepts we want to be in our final game. Now it's time to take a step further and build **the *actual* game**…
So lets **redo** this thing!

3. The Cookbook

3.1. Object Pools

`Object Pools` are one of the most widely used game development concepts used throughout all sorts of games. The basic concept is very simple: when you have a lot of similar items, then handle them all at once instead of one by one.

The Basics

Imagine we are working on a Galaga-type fixed shooter. Every frame we have to simulate and draw all the enemies and the player. There are multiple types of enemies with different behaviours and graphics, but they all need to be drawn in `love.draw` and updated in `love.update`. We also need to handle all the bullets flying around.

The simples way to handle all of this is just storing all these things in a list, then iterate over it in the two callbacks:

```lua
objects = {}
for i=1,20 do
  table.insert(objects, Enemy.new())
end
for i=1,20 do
  table.insert(objects, Bullet.new()) -- let's pretend someone actually fired these
end
player = Player.new()

function love.draw()
  for _,o in ipairs(objects) do ①
    o:draw()
  end
  player:draw()
end

function love.update(dt)
  for _,o in ipairs(objects) do
    o:update(dt)
  end
  player:update(dt)
end
```

① _ here is just a varible name, but it is a common convention to use it for any value we do not care about

So far, so obvious. But what happens when we need to remove things? Whenever the player shoots an enemy down, we will have to remove it from `objects` so we don't use up all of the PCs memory in a matter of minutes. The problem with using `table.remove()` is that it will update all the indices, so that we can continue iterating over the list (which is a good thing), but because of this we will skip the next enemy, which is inacceptable.

The simplest solution iterating in reverse:

```
-- rest as above
function love.update(dt)
  for i=#objects,1,-1 do ①
    objects[i]:update(dt)
    if objects[i].dead then
      table.remove(objects, i)
    end
  end
  player:update(dt)
end
```

① because we iterate in reverse, we need to use a numeric for

This is a good start, but let's take it a step further.

Sets

Sets, in mathematics, are a like a bag of objects; the objects don't have any order, nor names or labels associated with them. In Lua Sets are usually implemented as tables where key and value are the same:

```
objects = {}
for i=1,10 do
  local n = Enemy.new()
  objects[n] = n ①
end
for i=1,10 do
  local n = Bullet.new()
  objects[n] = n
end
player = Player.new()

function love.draw()
  for _,o in pairs(objects) do ②
    o:draw()
  end
  player:draw()
end
```

① table index and value are both the enemy object itself

② we now need to use `pairs()` instead of `ipairs()`

You might ask what advantage this might have over a simple list. After all we lost the ability to order the objects! One advantage of handling the list like this is that we now only have to care about the objects themselves; whether we change, delete or add objects, we never have to know, let alone search for, the index of that object. This means that we can now delete objects from basically everywhere in code (though that generally may hurt your code structure).

Another cool thing about this way of handling Sets is that we can actually add labels to **some** objects in the table if we want to. For example there is no need to treat the player as an exception anymore:

```
objects = {}
for i=1,10 do
  local n = Enemy.new()
  objects[n] = n
end
for i=1,10 do
  local n = Bullet.new()
  objects[n] = n
end
objects.player = Player.new() ①

function love.draw()
  for _,o in pairs(objects)
    o:draw()
  end ②
end

function love.update(dt)
  for _,o in pairs(objects)
    o:update(dt)
  end
end

function love.keypressed(key)
  object.player.handleKey(key) ③
end
```

① the player is now just yet another `object`

② we do not need to treat the player seperately anymore

③ yet we can still access him easily wherever we need to

WARNING When you start adding entries that use "custom" keys, make sure you are operating on the value (the second loop parameter) when interating with `pairs()`!

Wrapping up

As always, this concept is explained here on a very small scale. In an actual game project you would usually need multiple object pools for different things or layers. Still, Sets and Object Pools are going to be a building block of more or less every game you will ever encounter, so these small tricks might still be valuable information to you. <<<

License

Libraries & Tools

- AsciiDoctor renders this book
- Moonshine and
- punchdrunk by Tanner Rogalsky make LÖVE run in **your** browser
- …as does of course LÖVE (licensed under the zlib license), which this book is all about

wherever not listed above, these are licensed under the MIT License:

The MIT License