LÖVE for Newbies

Sol Bekic, Ricardo Gomes

Table of Contents

Preface

> So you want to make a game?

1. World 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:

1.1.1. 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 1-4 we will see that the actual way of saving the information we use differs, but we could as well do with this "definition".

1.1.2. 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.

1.1.3. 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

1.2.1. 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 * 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*

```
function love.conf(t)
    t.identity = nil                    -- The name of the save directory (string)
    t.version = "0.9.1"                 -- The L  VE 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`

```
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.

1.3.1. 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

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

2. World 2: The Redo

So that was World 1: 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 World 1: 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. World 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.

3.1.1. 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:

```
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.

3.1.2. 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()`!

3.1.3. 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. <<<

4. License

IMPORTANT	Find consensus on a license

4.1. Libraries & Tools

- AsciiDoctor renders this book
- Moonshine, licensed under the GNU GPL License, and
- punchdrunk by Tanner Rogalsky make LÖVE run in **your** browser
- ...as does of course LÖVE, which this book is all about

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

The MIT License

```
Copyright (C) held by the respective authors, see the LICENSEs in the links provided
above

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```