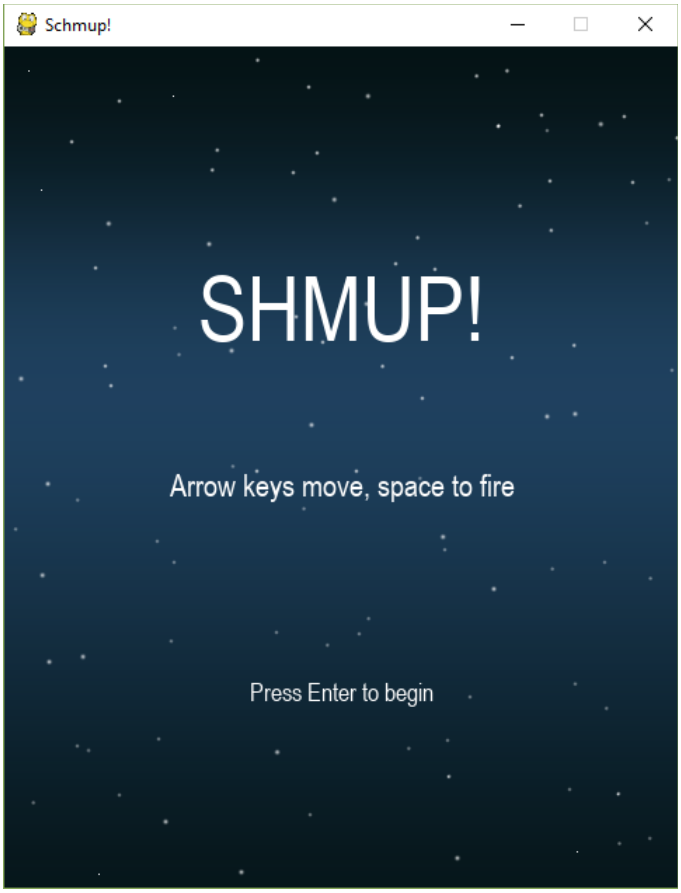
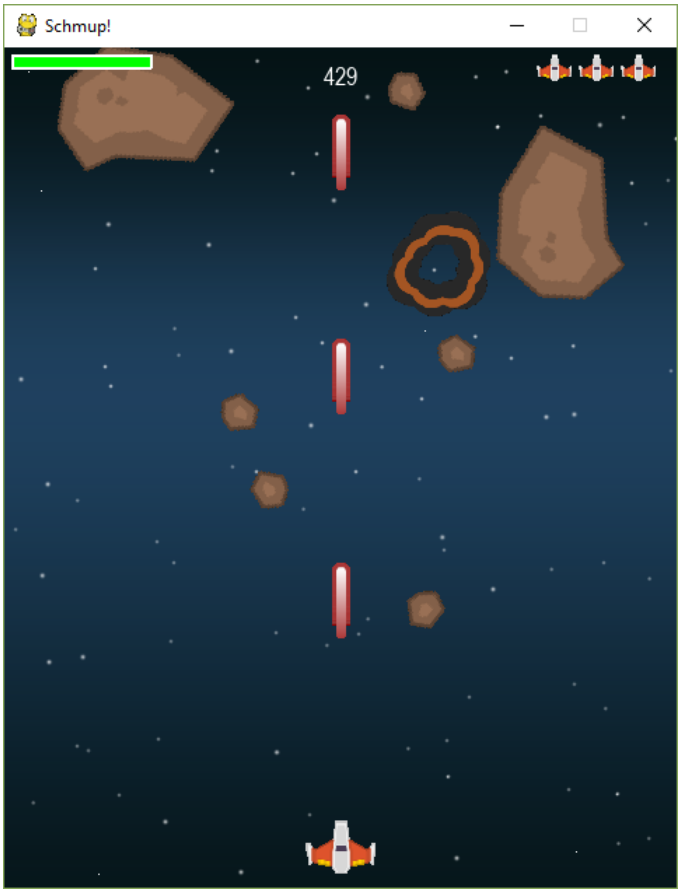
Love2D Tutorial

Shoot'Em'Up





Write a fully working game with sound effects and background music in Lua with Love2D

Love2D Tutorial Shoot'em Up

This tutorial is based on the the excellent Python Pygame Youtube series by KidsCanCode found at https://www.youtube.com/playlist?list=PLsk-HSGFjnaH5yghzu7PcOzm9NhsW0Urw

The code has been adapted to use Lua and the Love2D lua game engine, but uses the same resources:

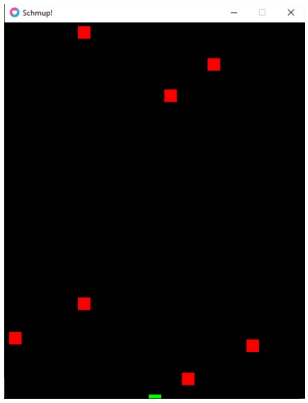
Create a new folder in your Lua/Love2D Directory called Shmup-01

Inside this folder make 3 sub-folders: 'lib', 'img' and 'snd'

If using this tutorial at school you should have these assets in the shared folder, otherwise download them: Put the images from https://github.com/kidscancode/pygame_tutorials/tree/master/shmup/img in the 'img' folder

Put the sounds from https://github.com/kidscancode/pygame_tutorials/tree/master/shmup/snd in the 'snd' folder

The first version of the game uses simple rectangles to represent the player, falling meteors and the bullets from the player.



In this screenshot, the red squares are falling at different rates and directions, the green rectangle at the bottom represents the player, and can be moved from side to side with the arrow keys.

Pressing the space bar fires a blue rectangle upwards from the player and if the rectangle hits a red square, it destroys it.

Open the Shmup directory in ZeroBrane Studio and add 5 files with the following names:

1. main.lua
2. conf.lua
3. bullet.lua
4. mob.lua
5. player.lua

If you are in a school setting, look in your shared drive for a pre-configured folder called Schmup which has all the assets and some of the code included.

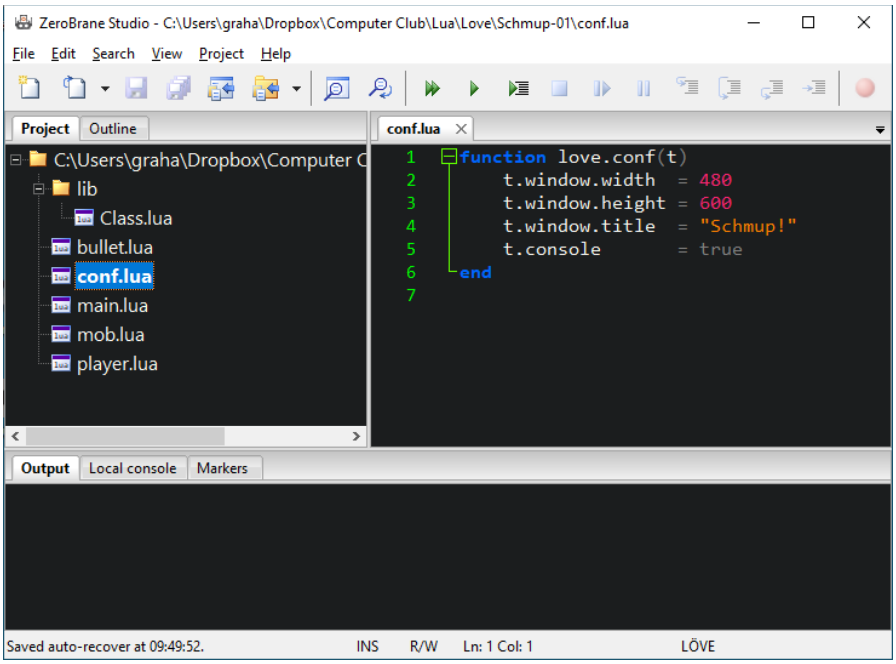
Right-Click on the lib folder and add a new file called Class.lua

Add this code to conf.lua:

```
function love.conf(t)

    t.window.width  = 480
    t.window.height = 600
    t.window.title  = "Schmup!"
    t.console       = true
end
```

Your setup should now look like this. Make sure Class.lua is inside the lib folder:



Variable setup:

If you have copied a main.lua template you will already have the 3 love functions defined, otherwise your main.lua file will be empty.

Put the following code at the top of the file (NOT in any of the functions)

```
_G.WIDTH =  love.graphics.getWidth()
_G.HEIGHT = love.graphics.getHeight()

-- colour constants: (Red, Green, Blue) 0-1 each value
local RED =     {1, 0, 0}
local GREEN =   {0, 1, 0}
local BLUE =    {0 ,0, 1}

local player     = {}               -- simple table for player. As there is only one, class not required
local bulletList = {}               -- store all bullets created in this table
local mobList    = {}               -- store all meteors (Mobs) in this table

local Bullet = require "bullet"     -- import the bullet class
local Mob = require "mob"           -- import the Mob class

local newBulletTimerInterval = 0.5  -- allow a new bullet every 0.5 seconds
local newBulletTimer = 0            -- start timer at 0, update(dt) increases its value
local allowNewBullet = true         -- change to false as soon as a new bullet is added to the bulletList
```

Explaining the variables:

It is useful to have the width and height of the window in a simple format so calculations can be made on the position of the game objects.

It is conventional to declare variables whose value is not going to change in CAPS. These are known as Constants.

It is ok to use WIDTH and HEIGHT, but the addition of _G. is a Lua convention that allows them to be used globally across all files in the project. You will see them used in the code later, some without the _G. part – just WIDTH and HEIGHT.

Colours in lua are expressed as 4 values: red, green, blue, alpha.
Each has a value between 0 and 1. if you have a tutorial showing numbers between 0 and 255, just divide them by 255 e.g. `local COLOUR = {128/255, 53/255, 0}`
Alpha is the tranparency of the colour, which defaults to fully opaque, and can be left out.

It is easier to set a few colours as constants using the CAPS convention, and supply a table of values for each one:

```
local RED = {1, 0, 0}
```

The keyword `local` is only found in Lua. It is used to increase memory efficiency and execution speed by restricting the 'scope' of a variable.

`local RED` can be used anywhere inside this file, including inside functions and loops, but cannot be seen from another file. It will not matter if you miss off the local declaration on all the variables declared in the body of the script, but is a good habit to get familiar with.

Player, bulletList and mobList are just empty tables. They will be used to store values and objects later.

The keyword require is the lua equivalent of Python import or C# using. It imports the library used to create bullets and meteors.

The remaining variables are used to control the rate bullets can be fired from the player.

Class.lua

If using this tutorial at school you should have Class.lua in the shared folder.If not you need to create it!
Open Class.lua and type the following code: Note: __ is 2 underscore characters (shift -)

```
local Class = {}
Class.__index = Class --metamethod to index itself
function Class:new() end

-- create new class type
function Class:derive(classType)
    assert(classType ~= nil, "parameter classType must not be nil")
    assert(type(classType) == "string", "parameter classType must be string")
    local cls = {}
    cls["__call"] = Class.__call
    cls.__index = cls
    cls.super = self
    setmetatable(cls, self) -- allows inheritance
    return cls
end

-- allow table to be treated as a function
function Class:__call(...)
    local inst = setmetatable({}, self) --create instance of Class
    inst:new(...)
    return inst
end

return Class
```

This code is very complex, but it is a way of making Lua behave as if it was an Object Oriented Program (OOP)

Do not concern yourself how it works at this stage.

The next code to add to main.lua is the function to detect collisions. Put it below the variable list you added above:

```
local function collides(rect1, rect2)

    --[[ check whether rectangles are NOT colliding ]]

    -- if left side of rect1 is beyond rect2 right side
    -- OR left side of rect2 is beyond rect1 right side

    if rect1.x > rect2.x + rect2.w or rect2.x > rect1.x + rect1.w then
        return false
    end
    -- if top side of rect1 is beyond bottom of rect2
    -- OR top side of rect2 is beyond bottom of rect1
    if rect1.y > rect2.y + rect2.h or rect2.y > rect1.y + rect1.h then
        return false
    end
    -- code only gets this far if both if statements above fail
    return true
end
```

This works by checking whether the coordinates of the 2 rectangles passed in as parameters (rect1, rect2) are intersecting with each other in 2 separate if statements.

The first if statement checks whether the left side of rect1 is beyond the right side of rect2 or vice-versa. If so, the rectangles cannot intersect, so return false and the function exits.

The second if statement checks if the top of rect1 is beyond the bottom of rect2, and vice-versa. If so, the rectangles cannot intersect, so return false and the function exits.

If neither of these if statements are true, then the rectangles are intersecting, so return true.

function love.load()

If you are working from a template, you will already have an empty love.load() function. Add these lines:

```
function love.load()

    if arg[#arg] == "-debug" then
        print("running in debug mode")
        require("mobdebug").start()
    end

    -- setup player
    player.w = 20                       -- width = 20 pixels
    player.h = 6                        -- height = 6 pixels
    player.speed = 300                  -- how fast the player can move
    player.x = WIDTH / 2 - player.w / 2 -- player starting position x based on window width and player width
    player.y = HEIGHT - player.h        -- player starting position y based on window height and player height

    --[[This is the clever bit! assign the love.graphics.rectangle() function to player.draw
    player.draw() is now a function, and can be called in the love.draw() built-in function]]
    player.draw = function() love.graphics.rectangle("fill", player.x, player.y, player.w, player.h) end

    -- make 8 Mobs (Dangerous rectangles) and store them in a list
    for i = 1, 8 do
        table.insert(mobList, Mob()) -- This one-liner creates a new Mob object and adds it to the mobList
    end

end
```

The if statement allows ZeroBrane to run a Love2D project in debug mode, so you can trace through the code.

It will not affect the game if you leave it out.

The player is given default values for the position and size of the rectangle and it's speed.

Player.draw is setup as follows:

```
player.draw = function() love.graphics.rectangle("fill", player.x, player.y, player.w, player.h) end
```

This is unique to Lua's table datatype. In Python and Lua you can assign a function to a variable, eg

`p = Print → assignment. No output`

`p("Hello World") -> outputs "Hello World"`

but you cannot assign p to print() with any brackets or parameters included.

In lua you can use the function() keyword to assign a fully parameterised function to a variable, eg:

`p = function() print("Hello World") end → assignment, no output`

`p() -> outputs "Hello World"`

In the love.draw() function, just use player.draw() to draw the player rectangle, as all the parameters have already been assigned.

The for loop creates 8 rectangles representing falling meteors and stores them in the mobList table.

Time to look at the mob.lua class:

mob.lua class

If using this tutorial at school you should have mob.lua in the shared folder. If not you need to create it! open mob.lua and add the following lines:

```
local Class = require("lib.Class")

local M = Class:derive("Mob")

function M:new()
    --[[ class constructor ]]
    self.w = 20
    self.h = 20
    M.setProperties(self)
end

function M.setProperties(self)
    -- set speed to a random amount, so some will move faster than others
    self.x = math.random(0, _G.WIDTH - self.w)  -- make the mob appear on screen randomly
    self.y = math.random(-150, -100)            -- start off the top of the screen by random amount
    self.speedX = math.random(-10, 10)
    self.speedY = math.random(10, 80)
end

function M:getRect()
    rect = {}
    rect.x = self.x
    rect.y = self.y
    rect.w = self.w
    rect.h = self.h
    return rect
end

function M:update(dt)
    self.y = self.y + self.speedY * dt
    self.x = self.x + self.speedX * dt
    -- check if Mob has gone off botom or sides of screen
    if  self.y > _G.HEIGHT + self.y or self.x < 0 - self.w or self.x > _G.WIDTH + self.w then
        M.setProperties(self)
    end
end

function M:draw()
    love.graphics.rectangle("fill", self.x, self.y, self.w, self.h)
end

return M
```

This code will allow you to create as many Mobs as you want, each one having different properties, such as size, position, rotation, speed etc. At the moment these properties are set randomly in the class code.

You can also update and draw them by using their own update() and draw() methods, called from the same functions in main.lua

Creation is done in main.lua in a number of ways:

1. create 1 mob called fred → `fred = Mob()`

2. create 8 mobs and put them in a list called mobList. They will not have named variables, just an index:

```
for i = 1, 8 do
    table.insert(mobList, Mob())
end
```

bullet.lua class

If using this tutorial at school you should have bullet.lua in the shared folder.If not you need to create it!

```
local Class = require("lib.Class")
local B = Class:derive("Bullet")


function B:new(x, y)
    --[[ class constructor, takes x and y integer values ]]
    self.x = x
    self.y = y
    self.w = 5
    self.h = 10
    self.speedY = -500
    self.active = true
end

function B:getRect()
    rect = {}
    rect.x = self.x
    rect.y = self.y
    rect.w = self.w
    rect.h = self.h
    return rect
end

function B:update(dt)
    self.y = self.y + self.speedY * dt
    if self.y <= 0 then
        self.active = false
    end
    return self.active -- when false, bullet can be set to nil
end

function B:draw()
    love.graphics.rectangle("fill", self.x, self.y, self.w, self.h)
end

return B
```

This code will allow you to create as many bullets as you want, but restricted to a time limit, so you cannot spam the spacebar to produce a continuous stream!

You can also update and draw them by using their own update() and draw() methods, called from the same functions in main.lua

Creation is done in main.lua in the love.keyboard.isDown() function:

```
if love.keyboard.isDown("space") then             -- has the player hit the space key to fire a bullet?
    if allowNewBullet then                        -- has enough time passed to fire a new bullet?
                                                  -- add a new bullet to the bulletList
        table.insert(bulletList, Bullet(player.x + player.w / 2, player.y))
        allowNewBullet = false                    -- prevent new bullets being made
        newBulletTimer = 0                        -- reset newBulletTimer to 0
    end
end
```

The way this works is:

1. allowNewBullet is true at the start of the game.
2. If the spacebar is down, the code next checks if a new bullet is allowed: `if allowNewBullet then`
3. Assuming this is ok, a new bullet object is made from the bullet class, and is given the current player's x and y coordinates: `Bullet(player.x + player.w / 2, player.y)`
4. This new bullet is added to the table bulletList `table.insert(bulletList,`
5. allowNewBullet is set to false, and newBulletTimer is reset to 0
6. This prevents another bullet being generated until allowNewBullet is set to true

Resetting allowNewBullet is done in the love.update(dt) function:

```
newBulletTimer = newBulletTimer + dt                -- increase newBulletTimer by dt
if newBulletTimer >= newBulletTimerInterval then    -- check if a new bullet can be created
    allowNewBullet = true                           -- YAY! new bullet can be created
    newBulletTimer = 0                              -- reset newBulletTimer to 0
end
```

dt is delta-time. This is the time interval between the last time the update() function was called. It is supposed to run 60x per second, but if a lot of code runs to calculate or draw the graphics, this could be much longer than 1/60th of a second.

The variable newBulletTimer only increases by the value of dt every frame, so it will eventually reach newBulletTimerInterval, which was set to 0.5 seconds at love.load() at exactly 0.5 seconds delay.

When that happens, allowNewBullet is set to true, and the timer reset to 0, so you can fire another bullet. Changing the value of newBulletTimerInterval at love.load() can increase or reduce the waiting time, to make the game harder or easier as required.

function love.update(dt)

```
function love.update(dt)
    newBulletTimer = newBulletTimer + dt                -- increase newBulletTimer by dt
    if newBulletTimer >= newBulletTimerInterval then    -- check if a new bullet can be created
        allowNewBullet = true                           -- YAY! new bullet can be created
        newBulletTimer = 0                              -- reset newBulletTimer to 0
    end

    if love.keyboard.isDown("left") then                -- move player left
        player.x = player.x - player.speed * dt
    end
    if love.keyboard.isDown("right") then               -- move player right
        player.x = player.x + player.speed * dt
    end
    if player.x < 0 then                                -- check if player x position is off-screen left side
        player.x = 0                                    -- whoops! change it to 0: left side of screen
    end
    if player.x > WIDTH - player.w then                 -- check if player x position is off-screen right side
        player.x = WIDTH - player.w                     -- whoops! change it to right side of screen
    end


    if love.keyboard.isDown("space") then               -- has the player hit the space key to fire a bullet?
        if allowNewBullet then                          -- has enough time passed to fire a new bullet?
                                                        -- add a new bullet to the bulletList
            table.insert(bulletList, Bullet(player.x + player.w / 2, player.y))
            allowNewBullet = false                      -- prevent new bullets being made
            newBulletTimer = 0                          -- reset newBulletTimer to 0
        end
    end

    -- update all bullets. remove any non-active
    for i = #bulletList, 1, -1 do                       -- Go through the bulletList in reverse order.
        if not bulletList[i]:update(dt) then            -- update the bullet. If it is too far up the screen: (false)
            bulletList[i] = nil                         -- delete it
            table.remove(bulletList, i)                 -- remove it from the table
        end
    end


    -- update all mobs
    for i = #mobList, 1, -1 do                          -- Go through the bulletList in reverse order
        mobList[i]:update(dt)
    end

    -- check if any bullets are colliding with any Mobs
    for i = #bulletList, 1, -1 do                       -- outer loop checks bullets
        local destroy = false                           -- local boolean set to false
        for j = #mobList, 1, -1 do                      -- inner loop checks Mobs
            -- destroy set to true if rectangles are colliding (bullet + Mob)
            destroy = collides(bulletList[i]:getRect() , mobList[j]:getRect())
            if destroy then                             -- destroy mob first
                mobList[j] = nil
                table.remove(mobList, j)
            end
        end
        if destroy then                                 -- destroy bullet
            bulletList[i] = nil
            table.remove(bulletList, i)
        end
    end
end
```

The update() function is heavily commented, but some further points below:

- The bulletTimer is described above.
- The left/right key detection could be simplified using math.min() and math.max()
- The bullet firing is discussed earlier.

Updating bullets:

```
    -- update all bullets. remove any non-active
    for i = #bulletList, 1, -1 do                       -- Go through the bulletList in reverse order.
        if not bulletList[i]:update(dt) then            -- update the bullet. If it is too far up the screen: (false)
            bulletList[i] = nil                         -- delete it
            table.remove(bulletList, i)                 -- remove it from the table
        end
    end
```

The for loop iterates the bulletList, and calls the object update() function on each bullet in turn.
The bullet:update(dt) function returns a true / false value based on whether the bullet is still on-screen:

```
function B:update(dt)
    self.y = self.y + self.speedY * dt
    if self.y <= 0 then
        self.active = false
    end
    return self.active -- when false, bullet can be set to nil
end
```

This returns false if the value of y is 0 or less (It has gone up past the top of the screen)
In this case the bullet is set to nil and removed from the list.

The loop has to run in reverse, so bullets are removed from the end of the list, otherwise the loop iterator runs into an error.

Updating meteors (Mobs)

```
    -- update all mobs
    for i = #mobList, 1, -1 do
        mobList[i]:update(dt)
    end
```

The loop does not have to run in reverse, as they are not removed within the loop

The mob:update(dt) function calculates a new position and if they have gone off-screen, are simply returned to a random position above the top of the window:

```
function M:update(dt)
    self.y = self.y + self.speedY * dt
    self.x = self.x + self.speedX * dt
    -- check if Mob has gone off botom or sides of screen
    if  self.y > _G.HEIGHT + self.y or self.x < 0 - self.w or self.x > _G.WIDTH + self.w then
        M.setProperties(self) -- reset to position above top of the screen
    end
end
```

The number does not increase in this early test version, but will be altered to allow new Mob(s) to spawn in later versions.

Checking collisions:

```
-- check if any bullets are colliding with any Mobs
    for i = #bulletList, 1, -1 do                       -- outer loop checks bullets
        local destroy = false                           -- local boolean set to false
        for j = #mobList, 1, -1 do                      -- inner loop checks Mobs
            -- destroy set to true if rectangles are colliding (bullet + Mob)
            destroy = collides(bulletList[i]:getRect() , mobList[j]:getRect())
            if destroy then                             -- destroy mob first
                mobList[j] = nil
                table.remove(mobList, j)
            end
        end
        if destroy then                                 -- destroy bullet
            bulletList[i] = nil
            table.remove(bulletList, i)
        end
    end
```

This uses nested for loops:

The outer loop (index i) iterates each bullet and uses an inner loop (index j) that iterates all the mobs.
If the inner loop finds a collision beween bullet and mob, the mob is removed from the list.
On completion of the inner loop, the bullet is also removed from the list if it hit a mob, so it will not continue to travel up the screen and hit another mob.

Love.draw()

```
function love.draw()
    love.graphics.setColor(GREEN)   -- change colour to green ready for player
    player.draw()

    love.graphics.setColor(BLUE)    -- change colour to blue ready for any bullets
    for i = 1, #bulletList do
        bulletList[i]:draw()
    end

    love.graphics.setColor(RED)     -- change colour to red ready for any mobs
    for i = 1, #mobList do
        mobList[i]:draw()
    end
end
```

This changes the rectangle colour to green before drawing the player.
Next the colour is changed to blue and the bullet:draw() method is called
Finally the colour is changed to red, and the mobs are drawn using their mob:draw() method

The game should now play, and allow you to shoot all the 8 red squares. There are no sound effects, no background music and no animated graphics.

It is a good start!

Version 2 – Graphics!

Close the project if it is open in Zerobrane.
Copy the Shmup-01 folder and name it Shmup-02.
Open the Shmup-02 folder in ZeroBrane.

The images found in the img folder are now going to be imported into the game. They will be stored in a new table called ‘sprites’.

Add this line in the variable declaration section, after the colour tables, and before the player, bulletList and mobList:

```
local sprites = {}
```

This will be used in the love.load() function.

Function love.load()

Add these 3 new lines above the player setup lines:

```
  sprites.background = love.graphics.newImage("img/starfield.png")
  sprites.player = love.graphics.newImage("img/playerShip1_orange.png")
  sprites.bullet = love.graphics.newImage("img/laserRed16.png")
```

These store the images for the background, player and bullets.

Next change the height and width of the player, which is now a sprite, not a rectangle:
from

```
    player.w = 20
    player.h = 6
```

to

```
   player.w = sprites.player:getWidth() / 2
   player.h = sprites.player:getHeight()
```

The player’s draw method also needs to change from

```
   player.draw = function() love.graphics.rectangle("fill", player.x, player.y, player.w, player.h) end
```

to

```
   player.draw = function() love.graphics.draw(sprites.player, player.x, player.y, nil, 0.5, 0.5) end
```

The values 0.5, 0.5 are the scale factors (x and y) to halve the image size.

Next a bit of code is used to store the 7 meteor images:

1. Create a table for the images:

```
sprites.meteorImages = {}
```

2. Create a table of the file names of the images:

```
local meteorList = {'meteorBrown_big1.png',
            'meteorBrown_big2.png',
            'meteorBrown_med1.png',
            'meteorBrown_med3.png',
            'meteorBrown_small1.png',
            'meteorBrown_small2.png',
            'meteorBrown_tiny1.png'}
```

3. Loop through the list of image filenames and load their images into the image table:

```
for i = 1, #meteorList do
    table.insert(sprites.meteorImages, love.graphics.newImage("img/".. meteorList[i]))
end
```

4. Now change the 8 mobs created from rectangles to meteors!

From:

```
for i = 1, 8 do
    table.insert(mobList, Mob())
end
```

to:

```
-- make 8 Mobs (Meteors) and store them in a list
for i = 1, 8 do
    table.insert(mobList, Mob(sprites.meteorImages)) -- creates a new meteor and adds it
end
```

For this to work, the mob class has to be modified, as you are now passing a set of images when creating a new mob.

Changes to mob.lua

Change the constructor:

From:

```
    function M:new()
        --[[ class constructor ]]
        self.w = 20
        self.h = 20
        M.setProperties(self)
    end
```

To:

```
    function M:new(meteorImages)
        --[[ class constructor, meteorImages is a table of images ]]
        self.image = meteorImages[math.random(1, #meteorImages)]
        self.w = self.image:getWidth()
        self.h = self.image:getHeight()
        self.rotation = 0
        self.rotationSpeed = math.random(-2,2)
        M.setProperties(self)
    end
```

This chooses a random image from the set of 7 every time a new meteor is created.
A random rotation between -2 (anticlockwise) to 2 (clockwise) is also added, so some will slowly turn as they fall.

Add this line to M:update(dt) after changing the x and y values:

```
    self.rotation = self.rotation + self.rotationSpeed * dt
```

Change the function M:draw()
From:

```
    function M:draw()
        love.graphics.rectangle("fill", self.x, self.y, self.w, self.h)
    end
```

To:

```
function M:draw()
    love.graphics.draw(self.image, self.x, self.y, self.rotation, nil, nil, self.image:getWidth() / 2,
                       self.image:getHeight() / 2)
end
```

Function love.update()

Some changes are needed here as well.

Change the bullet creation from:

```
    if love.keyboard.isDown("space") then               -- has the player hit the space key to fire a bullet?
        if allowNewBullet then                          -- has enough time passed to fire a new bullet?
                                                        -- add a new bullet to the bulletList
            table.insert(bulletList, Bullet(player.x + player.w / 2, player.y))
            allowNewBullet = false                      -- prevent new bullets being made
            newBulletTimer = 0                          -- reset newBulletTimer to 0
        end
    end
```

To:

```
    if love.keyboard.isDown("space") then              -- has the player hit the space key to fire a bullet?
        if allowNewBullet then                          -- has enough time passed to fire a new bullet?
                                                        -- add a new bullet to the bulletList
            table.insert(bulletList, Bullet(player.x + player.w / 2, player.y, sprites.bullet))
            allowNewBullet = false                      -- prevent new bullets being made
            newBulletTimer = 0                          -- reset newBulletTimer to 0
        end
    end
```

This means some changes are needed to the **bullet.lua** class:

Change the constructor
from:

```
function B:new(x, y)
    --[[ class constructor, takes x and y integer values ]]
    self.x = x
    self.y = y
    self.w = 5
    self.h = 10
    self.speedY = -500
    self.active = true
end
```

To:

```
function B:new(x, y, img)
    --[[ class constructor, takes x and y integer values and image]]
    self.x = x
    self.y = y
    self.w = 5
    self.h = 10
    self.speedY = -500
    self.active = true
    self.image = img
end
```

The B:draw() function needs changing as well:

From:

```
function B:draw()
    love.graphics.rectangle("fill", self.x, self.y, self.w, self.h)
end
```

To:

```
function B:draw()
    love.graphics.draw(self.image, self.x, self.y, nil, nil, nil, self.image:getWidth() / 2,
                       self.image:getHeight() / 4 * 3)
end
```

Next make changes to main.lua function love.draw()

From:

```
function love.draw()
    love.graphics.setColor(GREEN)   -- change colour to green ready for player
    player.draw()

    love.graphics.setColor(BLUE)    -- change colour to blue ready for any bullets
    for i = 1, #bulletList do
        bulletList[i]:draw()
    end

    love.graphics.setColor(RED)     -- change colour to red ready for any mobs
    for i = 1, #mobList do
        mobList[i]:draw()
    end
end
```

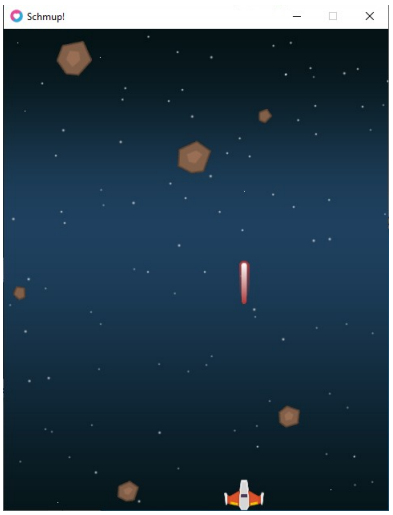
To:

```
function love.draw()
    love.graphics.draw(sprites.background, 0, 0)    -- draw background image
    player.draw()                                   -- draw player

    for i = 1, #bulletList do
        bulletList[i]:draw()                        -- draw bullet(s)
    end

    for i = 1, #mobList do
        mobList[i]:draw()                           -- draw meteors
    end
end
```

The game should run exactly the same as version 1, but with graphics instead of rectangles, on a starfield background.



The full code is listed below so you can look through to check any errors:

mob.lua:

```
local Class = require("lib.Class")
local M = Class:derive("Mob")

function M:new(meteorImages)
    --[[ class constructor, meteorImages is a table of images]]
    self.image = meteorImages[math.random(1, #meteorImages)]
    self.w = self.image:getWidth()
    self.h = self.image:getHeight()
    self.rotation = 0
    self.rotationSpeed = math.random(-2,2)
    M.setProperties(self)
end

function M.setProperties(self)
    -- set speed to a random amount, so some will move faster than others
    self.x = math.random(0, _G.WIDTH - self.w)  -- make the mob appear on screen randomly across it's width
    self.y = math.random(-150, -100)            -- start off the top of the screen by random amount
    self.speedX = math.random(-10, 10)
    self.speedY = math.random(10, 80)
end

function M:getRect()
    rect = {}
    rect.x = self.x
    rect.y = self.y
    rect.w = self.w
    rect.h = self.h
    return rect
end

function M:update(dt)
    self.y = self.y + self.speedY * dt
    self.x = self.x + self.speedX * dt
    self.rotation = self.rotation + self.rotationSpeed * dt

    if  self.y > _G.HEIGHT + self.y or self.x < 0 -self.w or self.x > _G.WIDTH +  self.w then
        M.setProperties(self)
    end
end

function M:draw()
    love.graphics.draw(self.image, self.x, self.y, self.rotation, nil, nil, self.image:getWidth() / 2,
self.image:getHeight() / 2)
end

function M:debug()
    print("Mob: x = "..self.x)
end

return M
```

bullet.lua

```
local Class = require("lib.Class")
local B = Class:derive("Bullet")


function B:new(x, y, img)
    --[[ class constructor, takes x and y integer values and image]]
    self.x = x
    self.y = y
    self.w = 5
    self.h = 10
    self.speedY = -500
    self.active = true
    self.image = img
end

function B:getRect()
    -- to use circle collision: calculate half the image max dimension
    local width = math.max(self.w, self.h)

    rect = {}
    rect.x = self.x
    rect.y = self.y
    rect.w = self.w
    rect.h = self.h
    return rect
end

function B:update(dt)
    self.y = self.y + self.speedY * dt
    if self.y <= 0 then
        self.active = false
    end
    return self.active -- when false, bullet can be set to nil
end

function B:draw()
    love.graphics.draw(self.image, self.x, self.y, nil, nil, nil, self.image:getWidth() / 2, self.image:getHeight() /
4 * 3)
end

return B
```

main.lua

```
_G.WIDTH = love.graphics.getWidth()
_G.HEIGHT = love.graphics.getHeight()

-- colour constants: (Red, Green, Blue) 0-1 each value
local RED =     {1, 0, 0}
local GREEN =   {0, 1, 0}
local BLUE =    {0 ,0 , 1}

local sprites = {}
local player = {}                   -- simple table for player as there is only one player
local bulletList = {}               -- store all bullets created in this list
local mobList = {}

local Bullet = require "bullet"     -- allow any number of bullets to be created
local Mob = require "mob"

local newBulletTimerInterval = 0.5  -- allow a new bullet every 0.5 seconds
local newBulletTimer = 0            -- start timer at 0, update(dt) increases its value
local allowNewBullet = true         -- change to false as soon as a new bullet is added to the bulletList
```

```lua
local function collides(rect1, rect2)
    --[[ check whether rectangles are NOT colliding ]]

    -- if left side of rect1 is beyond rect2 right side
    -- OR left side of rect2 is beyond rect1 right side

    if rect1.x > rect2.x + rect2.w or rect2.x > rect1.x + rect1.w then
        return false
    end
    -- if top side of rect1 is beyond bottom of rect2
    -- OR top side of rect2 is beyond bottom of rect1
    if rect1.y > rect2.y + rect2.h or rect2.y > rect1.y + rect1.h then
        return false
    end
    -- code only gets this far if both if statements above fail
    return true
end

function love.load()
    --[[ load fuction runs once on loading. Use to initialise variables ]]

    --[[
    in ZeroBrane, using single green triangle or F5 or Project-> Start Debugging adds -debug to the command line.
    This can be used to print out data to help debugging.
    If working ok the line "running in debug mode" appears in the console
    ]]
    if arg[#arg] == "-debug" then
        print("running in debug mode")
        require("mobdebug").start()
    end
    sprites.background = love.graphics.newImage("img/starfield.png")
    sprites.player = love.graphics.newImage("img/playerShip1_orange.png")
    sprites.bullet = love.graphics.newImage("img/laserRed16.png")

    -- setup player
    player.w = sprites.player:getWidth() / 2
    player.h = sprites.player:getHeight()
    player.speed = 300                      -- how fast the player can move
    player.x = WIDTH / 2 - player.w / 2     -- player starting position x based on window width and player width
    player.y = HEIGHT - player.h / 2            -- player starting position y based on window height and player height
    --[[This is the clever bit! assign the love.graphics.rectangle() function to player.draw
    player.draw() is now a function, and can be called in the love.draw() built-in function]]
    --player.draw = function() love.graphics.rectangle("fill", player.x, player.y, player.w, player.h) end
    --player.draw = function() love.graphics.draw(sprites.player, player.x, player.y) end
    player.draw = function() love.graphics.draw(sprites.player, player.x, player.y, nil, 0.5, 0.5) end

    sprites.meteorImages = {}
    local meteorList = {'meteorBrown_big1.png',
                'meteorBrown_big2.png',
                'meteorBrown_med1.png',
                'meteorBrown_med3.png',
                'meteorBrown_small1.png',
                'meteorBrown_small2.png',
                'meteorBrown_tiny1.png'}

    for i = 1, #meteorList do
        table.insert(sprites.meteorImages, love.graphics.newImage("img/".. meteorList[i]))
    end



    -- make 8 Mobs (Meteors) and store them in a list
    for i = 1, 8 do
        table.insert(mobList, Mob(sprites.meteorImages)) -- This one-liner creates a new Mob object and adds it to the
mobList
    end

end
```

```
function love.update(dt)
    newBulletTimer = newBulletTimer + dt                -- increase newBulletTimer by dt
    if newBulletTimer >= newBulletTimerInterval then    -- check if a new bullet can be created
        allowNewBullet = true                           -- YAY! new bullet can be created
        newBulletTimer = 0                              -- reset newBulletTimer to 0
    end

    if love.keyboard.isDown("left") then                -- move player left
        player.x = player.x - player.speed * dt
    end
    if love.keyboard.isDown("right") then               -- move player right
        player.x = player.x + player.speed * dt
    end
    if player.x < 0 then                                -- check if player x position is off-screen left side
        player.x = 0                                    -- whoops! change it to 0: left side of screen
    end
    if player.x > WIDTH - player.w then                 -- check if player x position is off-screen right side (take
player width into account)
        player.x = WIDTH - player.w                     -- whoops! change it to right side of screen, less the player
width
    end


    if love.keyboard.isDown("space") then               -- has the player hit the space key to fire a bullet?
        if allowNewBullet then                          -- has enough time passed to fire a new bullet?
                                                        -- add a new bullet to the bulletList
            table.insert(bulletList, Bullet(player.x + player.w / 2, player.y, sprites.bullet))
            allowNewBullet = false                      -- prevent new bullets being made
            newBulletTimer = 0                          -- reset newBulletTimer to 0
        end
    end

    -- update all bullets. remove any non-active
    for i = #bulletList, 1, -1 do                       -- Go through the bulletList in reverse order.
bullet:update(dt) returns true/false
        if not bulletList[i]:update(dt) then            -- update the bullet. If it is too far up the screen: (false)
            bulletList[i] = nil                         -- delete it
            table.remove(bulletList, i)                 -- remove it from the table
        end
    end


    -- update all mobs
    for i = #mobList, 1, -1 do                          -- Go through the bulletList in reverse order
        mobList[i]:update(dt)
    end

    -- check if any bullets are colliding with any Mobs
    for i = #bulletList, 1, -1 do                       -- outer loop checks bullets
        local destroy = false                           -- local boolean set to false
        for j = #mobList, 1, -1 do                      -- inner loop checks Mobs
            -- destroy set to true if rectangles are colliding (bullet + Mob)
            destroy = collides(bulletList[i]:getRect() , mobList[j]:getRect())
            if destroy then                             -- destroy mob first
                mobList[j] = nil
                table.remove(mobList, j)
            end
        end
        if destroy then                                 -- destroy bullet
            bulletList[i] = nil
            table.remove(bulletList, i)
        end
    end
end

function love.draw()
    love.graphics.draw(sprites.background, 0, 0)        -- draw background image
    player.draw()                                       -- draw player

    for i = 1, #bulletList do
        bulletList[i]:draw()                            -- draw bullets
    end

    for i = 1, #mobList do
        mobList[i]:draw()                               -- draw meteors
    end
end
```

Version 3 Sounds!

Next step is to add sounds and do a bit of re-factoring at the same time.

Close the project if it is open in Zerobrane.
Copy the Shmup-02 folder and name it Shmup-03.
Open the Shmup-03 folder in ZeroBrane.

main.lua

Variable declaration

add:

```
local sounds = {}
local sprites = {}
local player = {}
```

Add a new function underneath function collides()

```
local function newMob()
    table.insert(mobList, Mob(sprites.meteorImages)) -- creates a new Mob and adds it to the mobList
end
```

This is re-factoring, where code is tidied up to avoid repetition or increase efficiency. It is used in love.load() and love.update()

function love.load()

after the meteorList construction add:

```
    for i = 1, #meteorList do
        table.insert(sprites.meteorImages, love.graphics.newImage("img/".. meteorList[i]))
    end

    sounds.music        = love.audio.newSource("snd/FrozenJam.ogg", "stream")
    sounds.shoot        = love.audio.newSource("snd/Laser_Shoot6.wav", "static")
    sounds.die          = love.audio.newSource("snd/PlayerDeath.wav", "static")
    sounds.shield       = love.audio.newSource("snd/pow4.wav", "static")
    sounds.power        = love.audio.newSource("snd/pow5.wav", "static")
    sounds.explosions   = {}
    sounds.explosions[1] = love.audio.newSource('snd/Explosion5.wav', 'static')
    sounds.explosions[2] = love.audio.newSource('snd/Explosion3.wav', 'static')
```

This loads the background music and sound effects into the sounds table defined earlier.

Change the creation of 8 mobs
From:

```
for i = 1, 8 do
    table.insert(mobList, Mob(sprites.meteorImages)) -- creates a new meteor and adds it
end
```

To:

```
-- make 8 Mobs (meteors) and store them in a list
for i = 1, 8 do
    newMob()
end
```

function love.update(dt)

Change the code dealing with collisions from:

```
for i = #bulletList, 1, -1 do                       -- outer loop checks bullets
    local destroy = false                           -- local boolean set to false
    for j = #mobList, 1, -1 do                      -- inner loop checks Mobs
        -- destroy set to true if rectangles are colliding (bullet + Mob)
        destroy = collides(bulletList[i]:getRect() , mobList[j]:getRect())
        if destroy then                             -- destroy mob first
            mobList[j] = nil
            table.remove(mobList, j)
        end
    end
    if destroy then                                 -- destroy bullet
        bulletList[i] = nil
        table.remove(bulletList, i)
    end
end
```

To:

```
-- check if any bullets are colliding with any Mobs
for i = #bulletList, 1, -1 do                       -- outer loop checks bullets
    local destroy = false                           -- local boolean set to false
    for j = #mobList, 1, -1 do                      -- inner loop checks Mobs
        -- destroy set to true if rectangles are colliding (bullet + Mob)
        destroy = collides(bulletList[i]:getRect() , mobList[j]:getRect())
        if destroy then                             -- destroy mob first
            -- play sound depending on size
            if mobList[j]:getType() < 4 then
                love.audio.play(sounds.explosions[1])
            else
                love.audio.play(sounds.explosions[2])
            end
            mobList[j] = nil
            table.remove(mobList, j)
            newMob()
        end
    end
    if destroy then                                 -- destroy bullet
        bulletList[i] = nil
        table.remove(bulletList, i)
    end
end
if not sounds.music:isPlaying() then
    love.audio.play(sounds.music)
end
```

This plays a sound effect if a meteor is hit by a bullet, and runs the background music continuously. Also, when a meteor is destroyed, a new one is created with newMob()