Lua and Love2D Part 2

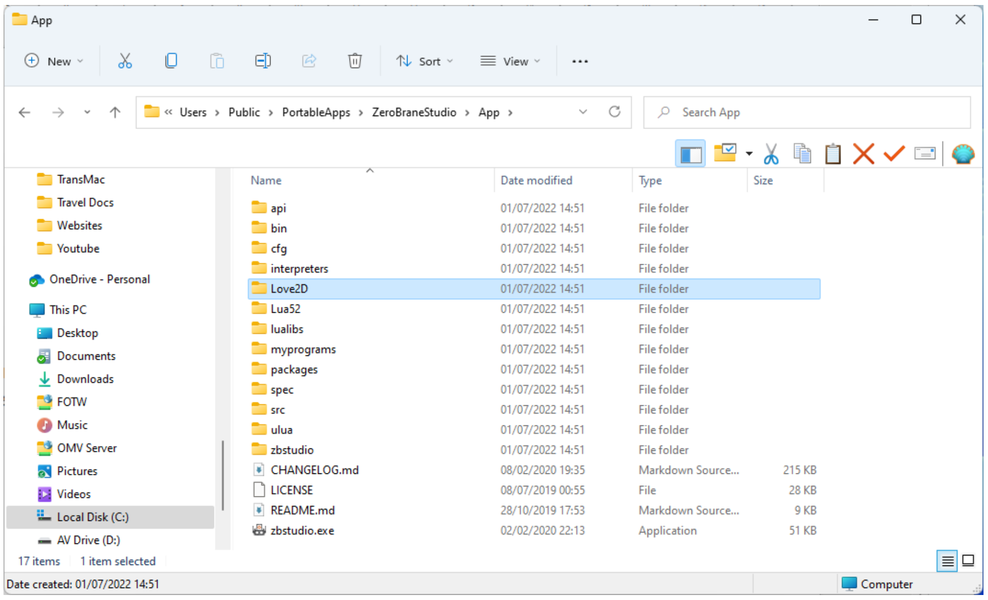
Installation

All the files in this tutorial are available from:

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials

Love2D can be downloaded from https://love2d.org/ If you have Admin permissions, use the installer. If not, or you want a portable app, then use the zipped file.

I have placed Love2D inside the ZeroBrane install folder:



Name	Date modified	Type	Size
api	01/07/2022 14:51	File folder	
bin	01/07/2022 14:51	File folder	
cfg	01/07/2022 14:51	File folder	
interpreters	01/07/2022 14:51	File folder	
Love2D	01/07/2022 14:51	File folder	
Lua52	01/07/2022 14:51	File folder	
lualibs	01/07/2022 14:51	File folder	
myprograms	01/07/2022 14:51	File folder	
packages	01/07/2022 14:51	File folder	
spec	01/07/2022 14:51	File folder	
src	01/07/2022 14:51	File folder	
ulua	01/07/2022 14:51	File folder	
zbstudio	01/07/2022 14:51	File folder	
CHANGELOG.md	08/02/2020 19:35	Markdown Source...	215 KB
LICENSE	08/07/2019 00:55	File	28 KB
README.md	28/10/2019 17:53	Markdown Source...	9 KB
zbstudio.exe	02/02/2020 22:13	Application	51 KB

Start Zerobrane
Menu → Edit → Preferences → Settings: System

```
--[[--
  Use this file to specify **System** preferences.
  Review [examples](\ZeroBraneStudio\App\cfg\user-sample.lua) or check [online documentation](http://studio.zerobrane.com/documentation.html) for details.
--]]--
activateoutput = true -- activate Output or Console window on new content added.
-- Love2D can be placed inside a Zerobrane/love2D/ folder:
path.love2d = 'love2D/love.exe' -- change this path to suit your system eg 'C:/Program Files/LOVE/love.exe'
```

Change the setting to reflect where your copy of Love2D is found. You can also change font sizes, styles, themes from this file.

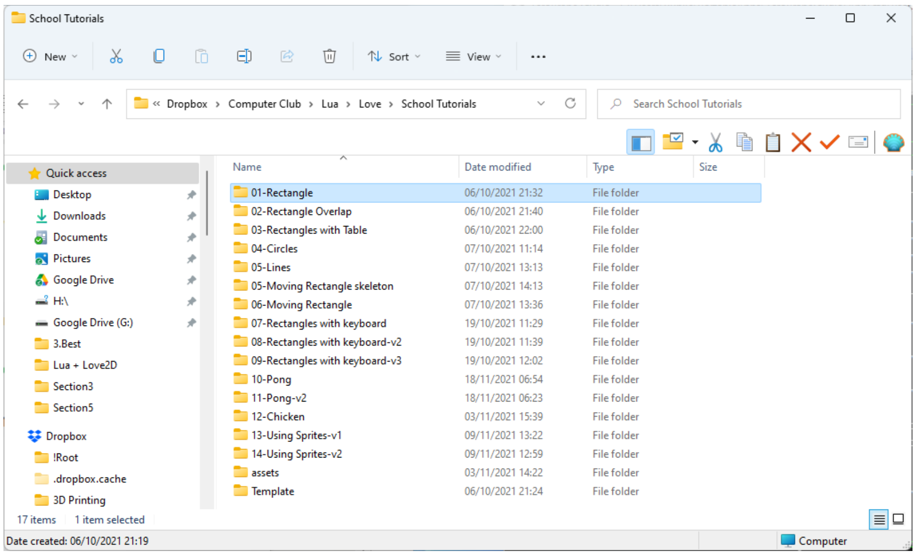
Repeat for Settings: User

If you change the path, save the user.lua files, close and restart Zerobrane.

The settings below work well, including the colour scheme Molokai

```
editor.fontsize = 14
editor.fontname = "Consolas"
editor.tabwidth = 4
editor.usetabs  = true
editor.usewrap = false
acandtip.nodynwords = false
acandtip.startat = 4
editor.checkeol = false
debugger.runonstart = true
editor.foldcompact = true
console.fontname = "Consolas"
console.fontsize = 14
console.nomousezoom = false
output.fontname = "Consolas"
output.fontsize = 14
styles = loadfile('cfg/tomorrow.lua')('Molokai')
stylesoutshell = styles
styles.auxwindow = styles.text
styles.calltip = styles.text
styles.indicator = {} -- to disable indicators (underlining)
```

Every Love2D project has to be in it's own folder, so if you have not already done so, create a new folder in your Lua folder for Love2D projects, and a new empty folder called 01-Rectangle



Name	Date modified	Type	Size
01-Rectangle	06/10/2021 21:32	File folder	
02-Rectangle Overlap	06/10/2021 21:40	File folder	
03-Rectangles with Table	06/10/2021 22:00	File folder	
04-Circles	07/10/2021 11:14	File folder	
05-Lines	07/10/2021 13:13	File folder	
05-Moving Rectangle skeleton	07/10/2021 14:13	File folder	
06-Moving Rectangle	07/10/2021 13:36	File folder	
07-Rectangles with keyboard	19/10/2021 11:29	File folder	
08-Rectangles with keyboard-v2	19/10/2021 11:39	File folder	
09-Rectangles with keyboard-v3	19/10/2021 12:02	File folder	
10-Pong	18/11/2021 06:54	File folder	
11-Pong-v2	18/11/2021 06:23	File folder	
12-Chicken	03/11/2021 15:39	File folder	
13-Using Sprites-v1	09/11/2021 13:22	File folder	
14-Using Sprites-v2	09/11/2021 12:59	File folder	
assets	03/11/2021 14:22	File folder	
Template	06/10/2021 21:24	File folder	

17 items 1 item selected
Date created: 06/10/2021 21:19

Select this folder from the Zerobrane Project menu AND select the Love interpreter.
Menu → Project → Lua interpreter → LOVE

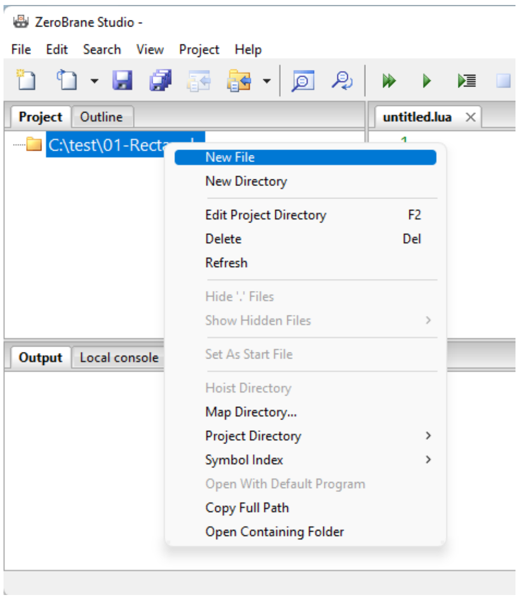
Menu → Project → Project Directory → Choose

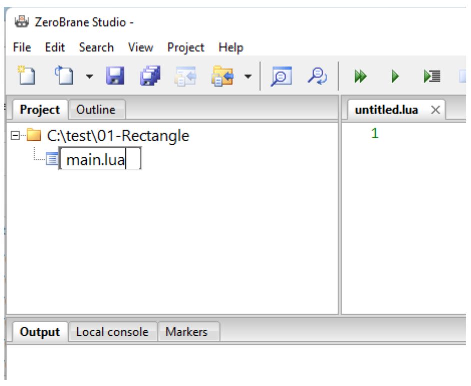
Browse to your Lua/Love2D/01-Rectangle project folder.

First Love2D project

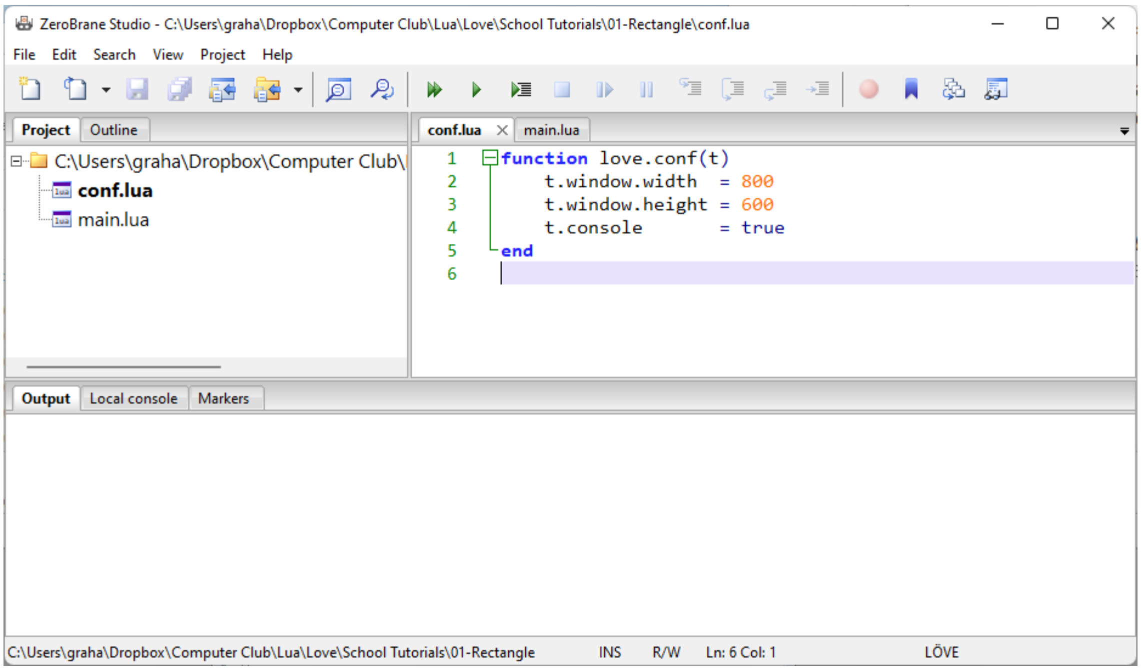
https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/01-Rectangle

Right-Click on the open project title and select New File. Name it main.lua:





Repeat for a new file called conf.lua
Close the empty "untitled.lua" tab
Double-click the conf.lua file


ZeroBrane Studio - C:\Users\graha\Dropbox\Computer Club\Lua\Love\School Tutorials\01-Rectangle\conf.lua
File Edit Search View Project Help
Project Outline
conf.lua main.lua
C:\Users\graha\Dropbox\Computer Club\
conf.lua
main.lua
Output Local console Markers
C:\Users\graha\Dropbox\Computer Club\Lua\Love\School Tutorials\01-Rectangle INS R/W Ln: 6 Col: 1 LÖVE

```lua
function love.conf(t)
    t.window.width  = 800
    t.window.height = 600
    t.console       = true
end

```

Add the lines as above.

This will create a window 800 pixels wide by 600 pixels high.
The Console will open allowing debugging print() calls to be performed.

01-Rectangle/conf.lua

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/01-Rectangle

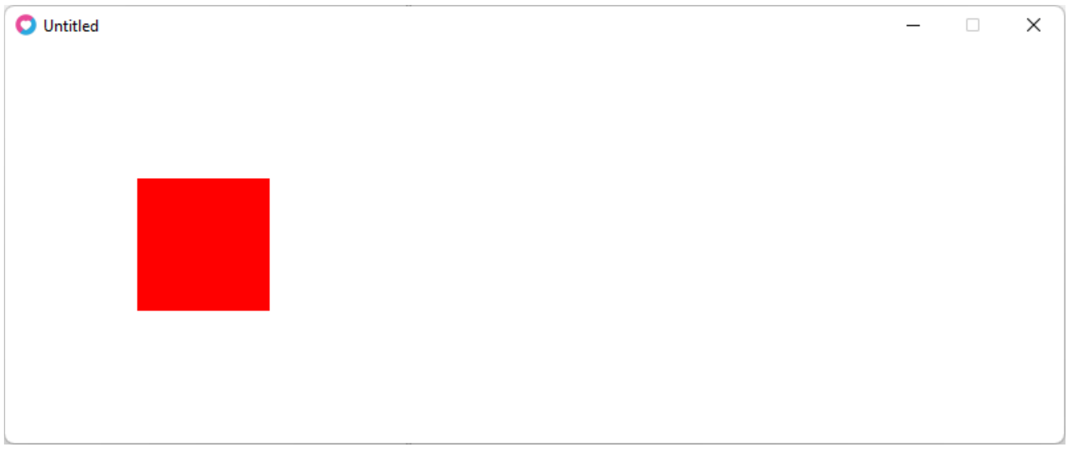
```lua
function love.conf(t)
    t.window.width  = 800
    t.window.height = 600
    t.console       = true
end
```

See https://love2d.org/wiki/Config_Files for further information

01-Rectangle/main.lua

```lua
function love.load()

end

function love.update(dt)

end

function love.draw()
   love.graphics.setColor(1, 0, 0, 1) -- set colour to red
   love.graphics.rectangle("fill", 100, 100, 100, 100)
end
```

Run the main.lua file from Zerobrane. (You did Menu → Project → Lua interpreter → LOVE, didn't you?)



The background colour has been changed to white to save ink on printing and the window size has been reduced to 800 x 300.
If you want to change the background use this line as the first in love.draw() for mid-grey.

```lua
love.graphics.setBackgroundColor(0.5,0.5,0.5,1)
```

If you are used to colours using 0-255 values then you can also use this for a mid-grey background:

```lua
love.graphics.setBackgroundColor(128/255, 128/255, 128/255, 255/255)
```

The first 3 numbers are the red, green, blue components and the fourth is transparency, where 0 is fully transparent and 1 is fully opaque.

If there is a file called conf.lua present in the game folder, it will be read and used before the game starts.

There MUST be a file called main.lua, otherwise the game will not run.

There are 3 functions in the barebones main.lua:

```lua
function love.load()

end
```

This is used to load in any sounds, graphics, and other assets, and setup variables and or classes. It runs once on loading.

The other 2 are run 60 x per second once the game has loaded

```lua
function love.update(dt)

end
```

This function is not used here, as all that happens is a static rectangle is drawn in the next function:

```lua
function love.draw()
   love.graphics.setColor(1, 0, 0, 1) -- set colour to red
   love.graphics.rectangle("fill", 100, 100, 100, 100)
end
```

This does 2 things at 60 x per second

1. It sets the drawing colour to red, fully opaque
2. It draws a filled red rectangle at coordinates 100, 100 width 100 and height 100

Coordinates are measured from TOP LEFT.

X coordinate starts at 0 on the LEFT, and increases as you go RIGHT
Y coordinate starts at 0 on the TOP of the window and increases as you go DOWN

Assignment:

1. Change the background colour
2. Change the rectangle x,y coordinates
3. Change the rectangle dimensions
4. Use “line” instead of “fill”

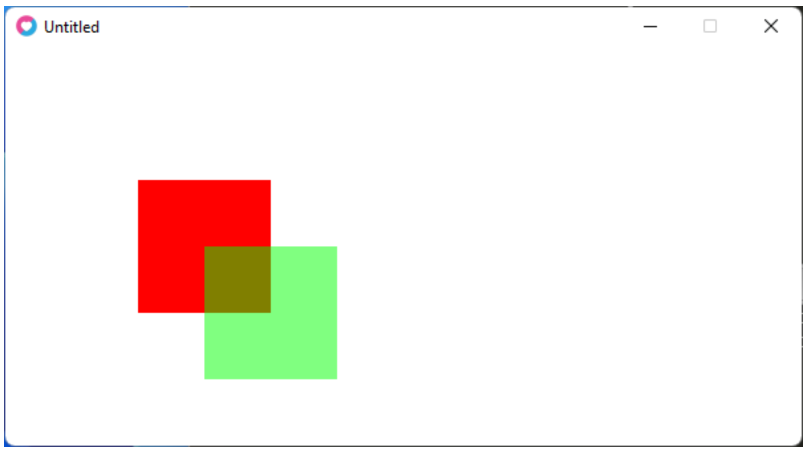
02-Rectangle Overlap/conf.lua and 02-Rectangle Overlap/main.lua

Create a new folder in your Lua/Love2D folder called 02-Rectangle Overlap. Copy/Paste the code from

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/02-Rectangle%20Overlap

conf.lua is same as 01-Rectangle/conf.lua and can be copy/pasted from your previous project.
main.lua

```lua
function love.load()
    -- set variables for 2 rectangles
    x1 = 100
    y1 = 100
    w1 = 100
    h1 = 100

    x2 = 150
    y2 = 150
    w2 = 100
    h2 = 100
end

function love.update(dt)

end

function love.draw()
    love.graphics.setBackgroundColor(1, 1, 1, 1)
    love.graphics.setColor(1, 0, 0, 1) -- set colour to red
    love.graphics.rectangle("fill", x1, y1, w1, h1)
    love.graphics.setColor(0, 1, 0, 0.5) -- set colour to green, transparency to 50%
    love.graphics.rectangle("fill", x2, y2, w2, h2)
end
```



- This time love.load() is used to create 8 variables for 2 rectangles.
- The draw colour is set to red and the first rectangle drawn.
- The draw colour is set to green, but with transparency set at 50% then drawn on top of the red rectangle.
- The transparency allows some of the red to show through.

Assignment

1. Change the rectangle coordinates / dimensions
2. Change the colours
3. Change the transparency

03-Rectangles With Table (Reminder create a new folder for each Love2D project)

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/03-Rectangles%20with%20Table

conf.lua is same as 01-Rectangle/conf.lua and can be copy/pasted
main.lua:

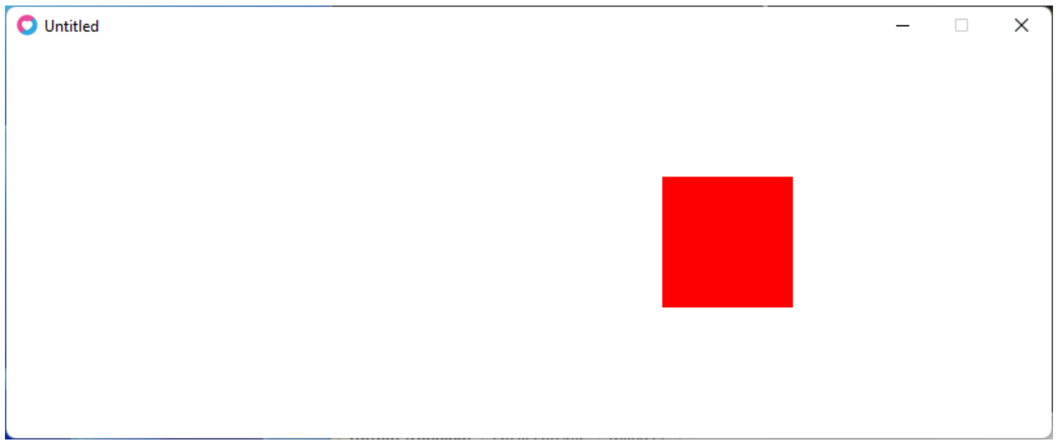
```lua
function love.load()
-- set variables for rectangle in a table
    rect1 = {}
    rect1.x = 100
    rect1.y = 100
    rect1.w = 100
    rect1.h = 100
end

function love.update(dt)
    rect1.x = rect1.x + 1
end

function love.draw()
    love.graphics.setBackgroundColor(1, 1, 1, 1)
    love.graphics.setColor(1, 0, 0, 1) -- set colour to red
    love.graphics.rectangle("fill", rect1.x, rect1.y, rect1.w, rect1.h)
end
```

This time, a red rectangle travels slowly from left to right across the screen

love.load() uses a table to hold the x, y, width and height of the rectangles

love.update() changes the position of the rectangle 60 pixels per second by changing it's x coordinate value

love.draw() paints a white background and the current position of the rectangle 60 x per second, so it appears to move across the screen.



Assignment:

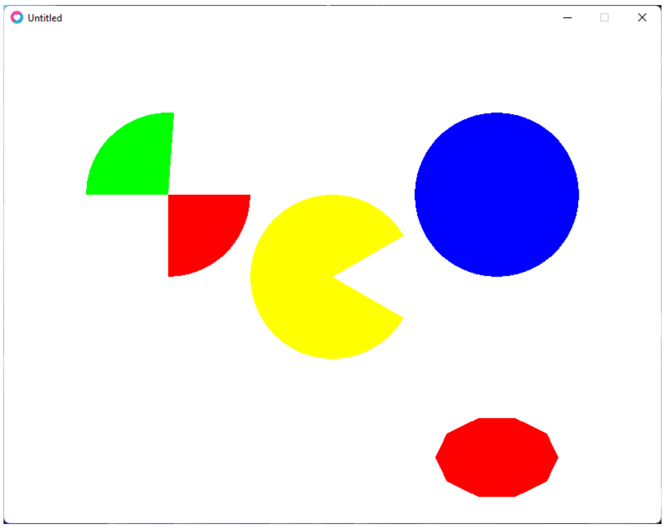
5. Change the background colour
6. Change the rectangle x,y coordinates
7. Change the rectangle dimensions
8. Use "line" instead of "fill"

04-Circles

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/04-Circles

main.lua

```lua
function love.load()

end

function love.update(dt)

end

function love.draw()
    love.graphics.setBackgroundColor(1, 1, 1, 1)
    -- love.graphics.circle( mode, x, y, radius )
    -- draw a blue circle at 600, 200 with a radius of 100
    love.graphics.setColor(0, 0, 1, 1) -- set colour to blue
    love.graphics.circle("fill", 600, 200, 100)


    -- love.graphics.arc( drawmode, x, y, radius, angle1, angle2, segments )
    -- Draws a filled or unfilled arc at position (x, y). The arc is drawn from angle1 to angle2 in
radians
    -- draw red 1/4 circle at  200,200 radius 100 clook position 12 to 3
    love.graphics.setColor(1, 0, 0, 1) -- set colour to red
    love.graphics.arc( "fill", 200, 200, 100, 0, math.pi/2)

    -- draw green 1/4 circle at  200,200 radius 100 clock poition 9 to 12
    love.graphics.setColor(0, 1, 0, 1) -- set colour to green
    love.graphics.arc( "fill", 200, 200, 100, -3.142, -1.5)

    -- draw pacman
    love.graphics.setColor(1, 1, 0, 1) -- set colour to yellow
    pacwidth = math.pi / 6 -- size of his mouth
    love.graphics.arc( "fill", 400, 300, 100, pacwidth, (math.pi * 2) - pacwidth )

    -- draw ellipse
    -- love.graphics.ellipse( mode, x, y, radiusx, radiusy, segments )
    love.graphics.setColor(1, 1, 1)
    love.graphics.ellipse("fill", 600, 400, 75, 50, 100) -- Draw white ellipse with 100 segments.
    love.graphics.setColor(1, 0, 0)
    love.graphics.ellipse("fill", 600, 520, 75, 50, 10)   -- Draw red ellipse with 10 segments.
end
```



The comments in the code say it all.
Looks better on a black background. Change the code to do that

05-Lines

https://github.com/Inksaver/LuaForSchools/blob/main/Love2D%20School%20Tutorials/05-Lines/main.lua

main.lua

```lua
function love.load()
    colour = love.graphics.setColor -- use a variable to store a function
    line = love.graphics.line
    red = {1,0,0,1}
    green = {0,1,0,1}
    blue = {0,0,1,1}
end

function love.update(dt)

end

function love.draw()
    love.graphics.setBackgroundColor(1, 1, 1, 1)
    -- love.graphics.line( x1, y1, x2, y2, ... ) You can continue passing points to draw a polyline.
    colour(green)
    love.graphics.line(0, 20, 800, 20)
    colour(red)
    line(200,50, 400,50, 500, 250, 100,250, 200,50)   -- last pair repeats to complete the trapezoid

    local mx, my = love.mouse.getPosition()
    local windowWidth, windowHeight = love.graphics.getDimensions()
    colour(blue)
    love.graphics.line(windowWidth/2, windowHeight/2, mx, my)
end
```

Useful tips:

The function `love.graphics.setColor` has been assigned to a variable `colour` , so to call the function later, just use the variable. Note when assigning do not use the brackets!

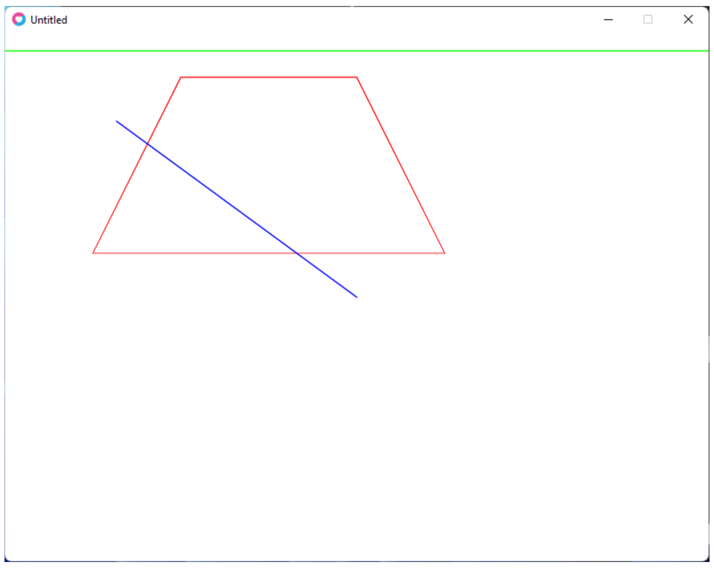
Same with `line = love.graphics.line`

red and blue are variables of type table, containing all four rgba elements

Two of the assignments above can be used together `colour(green)` is the same as:

`love.graphics.setColor(0,1,0,1)`

Note how the blue line drawn on screen follows the mouse with a line starting at the centre.



This is achieved with:

```
local mx, my = love.mouse.getPosition()
local windowWidth, windowHeight = love.graphics.getDimensions()
colour(blue)
love.graphics.line(windowWidth/2, windowHeight/2, mx, my)
```

The line is drawn from the calculated window centre to the current x,y position of the mouse via `love.mouse.getPosition()`

Assignment:

There is an obvious inefficiency in this code with the line:

```
local windowWidth, windowHeight = love.graphics.getDimensions()
```

Why is it inefficient?

What can be done about it?

Hint: some love functions run at 60fps...

06-Moving Rectangle

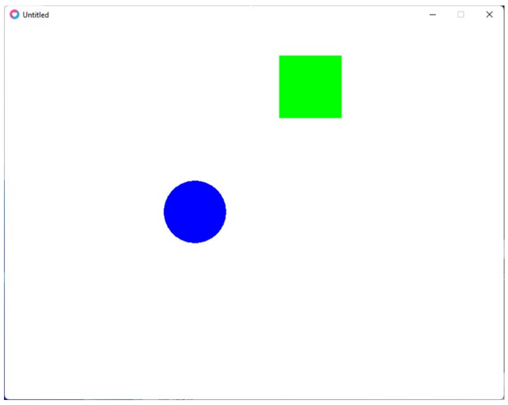
https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/06-Moving%20Rectangle

This project moves a circle and a rectangle across the screen and they bounce back and forth when they hit the margins.
main.lua

```lua
function love.load()
    rect = {x = 0, y = 50, w = 100, h = 100, speed = 5}
    circle = {x = 25, y = 300, r = 50, speed = 7}
end

function love.update(dt)
    rect.x = rect.x + rect.speed

    if rect.x > 800 - rect.w then
        rect.speed = rect.speed * -1
    end
    if rect.x < 0 then
        rect.speed = math.abs(rect.speed)
    end

    circle.x = circle.x + circle.speed
    if circle.x > 800 - circle.r then
        circle.speed = circle.speed * -1
    end
    if circle.x < circle.r -1 then
        circle.speed = math.abs(circle.speed)
    end
end

function love.draw()
    love.graphics.setBackgroundColor(1, 1, 1, 1)
    love.graphics.setColor(0,1,0,1)
    love.graphics.rectangle("fill", rect.x, rect.y, rect.w, rect.h)
    love.graphics.setColor(0,0,1,1)
    love.graphics.circle("fill", circle.x, circle.y, circle.r)
end
```

The shapes are defined as tables in love.load()

In love.update() the rect x coordinate is updated by rect.speed (5).
If the x coordinate is greater than the width (800) then the speed is made negative by multiplying by -1
If the x coordinate is less than 0 then speed is set to positive using math.abs



The circle is treated in a similar fashion, but using the radius.

Both are drawn in the usual way in love.draw()

07-Rectangles With Keyboard

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/07-Rectangles%20with%20keyboard

The WASD keys are used to move a rectangle around the screen.
If it hits any window border, it stops.

main.lua

```lua
function love.load()
    rect = {x = 0, y = 0, w = 100, h = 100, speed = 10}
end

function love.update(dt)
    if love.keyboard.isDown('w') then
        rect.y = rect.y - rect.speed
    end
    if love.keyboard.isDown('a') then
        rect.x = rect.x - rect.speed
    end
    if love.keyboard.isDown('s') then
        rect.y = rect.y + rect.speed
    end
    if love.keyboard.isDown('d') then
        rect.x = rect.x + rect.speed
    end
    if rect.x > 800 - rect.w then
        rect.x = 800 - rect.w
    end
    if rect.x < 0 then
        rect.x = 0
    end
    if rect.y > 600 - rect.h then
        rect.y = 600 - rect.h
    end
    if rect.y < 0 then
        rect.y = 0
    end
end

function love.draw()
    love.graphics.setBackgroundColor(1, 1, 1, 1)
    love.graphics.setColor(1, 0, 0) -- set colour to red
    love.graphics.rectangle("fill", rect.x, rect.y, rect.w, rect.h)
end
```

The key inputs are read with the `love.keyboard.isDown('<key>')` functions, one for each of the WASD keys.

The rectangle coordinates are changed depending on the key(s) pressed.

Once the new coordinates are calculated, separate checks are made on the positions, and movement is stopped by ‘clamping’ the coordinates to the min / max values.

Note how the width / height of the rectangle is used in the calculations, as the coordinates are measured from the top left.

08-Rectangles With Keyboard-v2

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/08-Rectangles%20with%20keyboard-v2

Works the same as 07-Rectangles With Keyboard, but with some variations on the code:

```lua
local WIDTH = love.graphics.getWidth()
local HEIGHT = love.graphics.getHeight()
function love.load()
    -- rect table declared in a more compact form
    rect1 =
    {
        x = 0,
        y = 0,
        w = 100,
        h = 100,
        speed = 10
    }
    -- assign functions to variables
    key = love.keyboard.isDown
    colour = love.graphics.setColor
    rectangle = love.graphics.rectangle
end

function love.update(dt)
    -- combining selections to check for key press and ability to move
    if (key('w') or key('up')) and rect1.y > 0 then
        rect1.y = rect1.y - rect1.speed
    end
    if (key('a')  or key('left')) and rect1.x > 0 then
        rect1.x = rect1.x - rect1.speed
    end
    if (key('s') or key('down')) and rect1.y < HEIGHT - rect1.h then
        rect1.y = rect1.y + rect1.speed
    end
    if (key('d') or key('right')) and rect1.x < WIDTH - rect1.w then
        rect1.x = rect1.x + rect1.speed
    end
end

function love.draw()
    colour(1, 0, 0, 1) -- set colour to red
    rectangle("fill", rect1.x, rect1.y, rect1.w, rect1.h)
End
```

Note:
Assigning functions to variables saves time and space.

```lua
  key = love.keyboard.isDown
```

Checking for which key has been pressed **and** if the rectangle is in a position allowing movement.

```lua
  if (key('s') or key('down')) and rect1.y < HEIGHT - rect1.h then
```

09-Rectangles with keyboard-v3

There is another alternative with changing rectangle colours at:

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/09-Rectangles%20with%20keyboard-v3

10-Pong

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/10-Pong

main.lua part 1

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

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()
    -- set variables for left paddle in a table
    --[[
    -------------------------------------------------------------
    |         tables of variables for paddles and ball          |
    -------------------------------------------------------------
    |   name   |x position | y position | width | height | speed |
    -------------------------------------------------------------
    | paddleL |      10    |     260    |   15  |   80   |   10  |
    -------------------------------------------------------------
    | paddleR | WIDTH - 25|     260    |   15  |   80   |   10  |
    -------------------------------------------------------------
    |   ball  | WIDTH/2-5 | HEIGHT/2-5 |   10  |   10   |    5  |
    -------------------------------------------------------------
    ]]
    paddleL =
    {
        x = 10,          -- start 10 pixels in from left edge
        y = 260,         -- start 260 pixels down from top
        w = 15,          -- paddle is 15 pixels wide
        h = 80,          -- paddle is 80 pixels tall
        speed = 10
    }
    -- set variables for right paddle in a table
    paddleR =
    {
        x = WIDTH - 25, -- start 10 pixels from the right edge (10 + width of paddle)
        y = 260,
        w = 15,
        h = 80,
        speed = 10
    }
    -- set variables for ball in a table
    ball =
    {
        x = WIDTH / 2 - 5,  -- start in centre of the screen (half of WIDTH minus ball width)
        y = HEIGHT / 2 - 5, -- start in centre of the screen (half of HEIGHT minus ball height)
        w = 10,
        h = 10,
        speedx = math.random(-5, 5),
        speedy = math.random(-2, 2)
    }
    rectangle = love.graphics.rectangle
    key = love.keyboard.isDown
end
```

WIDTH and HEIGHT are constant values of the window dimensions, so are represented by convention in CAPS. (like a variable, but the contents do not change). They are useful in the code for calculating the positions of game assets (rectangles, circles etc).

The code above contains a new function called collides() which handles collisions between the ball and the paddles.

This function could be used to check for collisions between any 2 rectangles, as long as they have x, y, w, and h propeties.

Three tables are created, one for each of the paddles, and one for the ball.
Note the ball table has 2 speed properties, one for the up/down direction and one for left/right. These are initialised with random values. Negative numbers make it move the opposite direction.

```lua
function love.update(dt)
    -- w and s keys control left paddle
    if key('w') then -- move left paddle up
        paddleL.y = math.max(0, paddleL.y - paddleL.speed)
    elseif key('s') then -- move left paddle down
        paddleL.y = math.min(HEIGHT - paddleL.h, paddleL.y + paddleL.speed)
    end

    -- up and down keys control right paddle
    if key('up') then -- move right paddle up
        paddleR.y = math.max(0, paddleR.y - paddleR.speed)
    elseif key('down') then -- move right paddle down
        paddleR.y = math.min(HEIGHT - paddleR.h, paddleR.y + paddleR.speed)
    end

    ball.x = ball.x + ball.speedx
    ball.y = ball.y + ball.speedy
    if ball.x < 0 or ball.x > WIDTH then -- if ball has hit the edge of the screen-> back in the centre
        ball.x = WIDTH / 2 - ball.w / 2
    end
    if ball.y < 0 or ball.y > HEIGHT - ball.h then
        ball.speedy = ball.speedy * -1
    end

    if collides(ball, paddleR) then         -- has the ball hit the right paddle?
        ball.speedx = ball.speedx * -1      -- reverse the direction right -> left
        -- alter the vertical speed between 1 and 2 randomly in the same direction
        if ball.speedy < 0 then
            ball.speedy = -math.random(-2, 2)
        else
            ball.speedy = math.random(-2, 2)
        end
    elseif  collides(ball, paddleL) then    -- has the ball hit the right paddle?
        ball.speedx = math.abs(ball.speedx) -- reverse the direction left -> right
        if ball.speedy < 0 then
            ball.speedy = -math.random(-2, 2)
        else
            ball.speedy = math.random(-2, 2)
        end
    end
end

function love.draw()
    --love.graphics.clear(0.18, 0.16, 0.2)  -- clear screen with a dark grey colour
    love.graphics.clear(1, 1, 1)    -- clear screen with white
    love.graphics.setColor(0, 0, 0, 1)
    rectangle('fill', paddleL.x, paddleL.y, paddleL.w, paddleL.h)   -- draw left paddle
    rectangle('fill', paddleR.x, paddleR.y, paddleR.w, paddleR.h)   -- draw right paddle
    rectangle('fill', ball.x, ball.y, ball.w, ball.h)               -- draw ball
end
```

The update() and draw() functions perform as usual.

These lines will check if either the “w” or “s” keys are being held down for the left paddle and up / down keys for the right paddle.

If “w” then move the paddle up the screen by decreasing paddleL.y by the number of pixels held in paddleL.speed. As this was set to 10 in love.load(), this means that every frame, the paddles y coordinate decreases by 10. If running at 60 frames / sec, the paddle will move by 600 pixels every second.

Similarly, if the “s” key is held down, the y coordinate increases by 10 every frame, and the paddle moves down.

Note when moving either paddle a check is made to see whether the paddle has reached it’s limits. e.g. the w key to move the left panel upwards:

```
paddleL.y = math.max(0, paddleL.y - paddleL.speed)
```

The value of the y coordinate is set using `math.max(value1, value2)` which selects the greatest value between 0 and the calculated new postion of the paddle, so it never goes above the top of the screen.

Similarly the s key to move down uses:

```
paddleL.y = math.min(HEIGHT - paddleL.h, paddleL.y + paddleL.speed)
```

The `math.min(value1, value2)` function returns the lowest value from the screen height minus paddle height or paddle.y plus speed.

The ball movement is calculated from it’s current position, speed and direction.
That movement is then taken into consideration when checking whether the ball has gone past the left or right edge of the screen. If it has, it is returned to the centre:

```
if ball.x < 0 or ball.x > WIDTH then -- if ball has hit the edge of the screen-> back in the centre
    ball.x = WIDTH / 2 - ball.w / 2
end
```

Next check is if the ball has hit the top or bottom of the screen, when it will have its y direction reversed effectively making it bounce in the opposite direction:

```
if ball.y < 0 or ball.y > HEIGHT - ball.h then
    ball.speedy = ball.speedy * -1
end
```

Finally ball collision with a paddle is checked, which reverses the horizontal direction and resets the vertical direction to a random amount:

```
if collides(ball, paddleR) then         -- has the ball hit the right paddle?
    ball.speedx = ball.speedx * -1      -- reverse the direction right -> left
    -- alter the vertical speed between 1 and 2 randomly in the same direction
    if ball.speedy < 0 then
        ball.speedy = -math.random(-2, 2)
    else
        ball.speedy = math.random(-2, 2)
    end
```

Assignment

You could now make some improvements to the game:

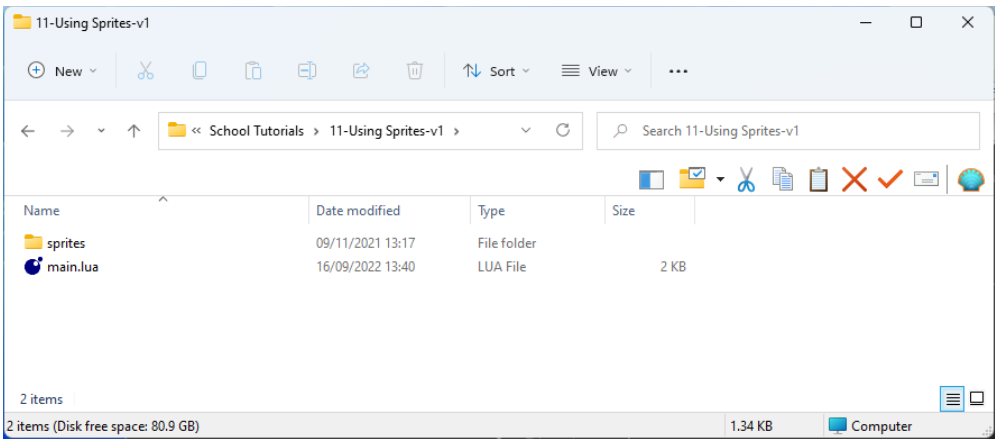
1. Keep a score and display it using love.graphics.print(text, x, y)
2. Change the colour of the paddles and ball by setting the colour with love.graphics.setColor() before each rectangle is drawn

11-Using Sprites-v1

As always create a new folder called 11-Using Sprites-v1.
Add a new folder inside: 11-Using Sprites-v1/sprites/

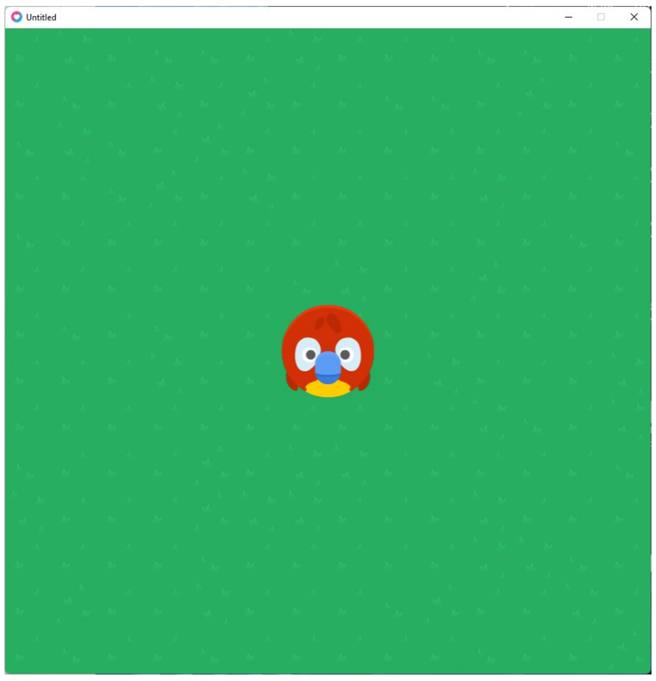
https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/11-Using%20Sprites-v1

There is a folder of assets (2 images) and main.lua in this link
Put the images inside the sprites folder
Put main.lua in the root of 11-Using Sprites-v1


11-Using Sprites-v1
New
Sort
View
School Tutorials › 11-Using Sprites-v1 ›
Search 11-Using Sprites-v1

Name	Date modified	Type	Size
sprites	09/11/2021 13:17	File folder	
main.lua	16/09/2022 13:40	LUA File	2 KB

2 items
2 items (Disk free space: 80.9 GB) 1.34 KB Computer



love.load()

```
function love.load()
    -- using an image as background, so use the image as window size.
    background = love.graphics.newImage('sprites/background.png')
    WIDTH = background:getWidth()
    HEIGHT = background:getHeight()

    love.window.setMode(WIDTH, HEIGHT) -- change window size to match background
    player =
    {
        x = 0,
        y = 0,
        speed = 5,
        sprite = love.graphics.newImage('sprites/parrot.png'),
    }
    player.width = player.sprite:getWidth()   -- get player width  from image
    player.height = player.sprite:getHeight() -- get player height from image
    player.x = (WIDTH - player.width) / 2     -- centre the player x
    player.y = (HEIGHT - player.height) / 2   -- centre the player y


    key = love.keyboard.isDown
    draw = love.graphics.draw
end
```

A different approach for this project:

- There is no conf.lua
- An image is drawn to the background instead of a plain colour
- An image (sprite) is used as the player instead of a rectangle

The variable `background` is used to store an image using `love.graphics.newImage('sprites/background.png')`
The location of the image is important, so if you did not put the images in a folder called sprites, then change the location in this line of code.
The WIDTH and HEIGHT of the window will be set from the size of the background image.
To do this first set the constants:

```
    WIDTH = background:getWidth()
    HEIGHT = background:getHeight()
```

Then set the window size

```
  love.window.setMode(WIDTH, HEIGHT) -- change window size to match background
```

The player is a table, but this time it has an image as well as x, y, width, height and speed.
The x, y values are set to 0 in the table constructor, and modified outside the constructor along with adding the width and height properties:

```
    player.width = player.sprite:getWidth()   -- get player width  from image
    player.height = player.sprite:getHeight() -- get player height from image
    player.x = (WIDTH - player.width) / 2     -- centre the player x
    player.y = (HEIGHT - player.height) / 2   -- centre the player y
```

It has to be done this way, as trying to use the above statements in the constructor gives an error, as the player table is not fully initialised until the closing } bracket.

Key and draw are assigned functions as in earlier projects:

```
    key = love.keyboard.isDown
    draw = love.graphics.draw
```

love.update() and love.draw()

```lua
function love.update(dt)
    if key("right") then
        player.x = math.min(WIDTH - player.width, player.x + player.speed)
    end

    if key("left") then
        player.x = math.max(0, player.x - player.speed)
    end

    if key("down") then
        player.y = math.min(HEIGHT - player.height, player.y + player.speed)
    end

    if key("up") then
        player.y = math.max(0, player.y - player.speed)
    end
end

function love.draw()
    draw(background, 0, 0)
    draw(player.sprite, player.x, player.y)
end
```

The code here is similar to that used for moving rectangles.

The x, y coordinates are changed depending on which key is pressed, and are clamped between fixed min / max values

12-Using Sprites-v2

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/12-Using%20Sprites-v2

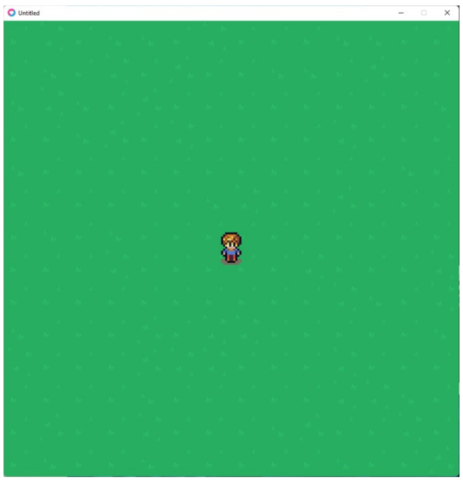
There is a library called anim8 from https://github.com/kikito/anim8/blob/master/anim8.lua which has been copied and used here.

This project is an introduction to animation and sprite sheets.

There is background music/sounds playing continuously.

As the player is moved with the keyboard, footstep sound effects are heard.

The player is animated and faces the appropriate direction during movement.



This is the spritesheet:

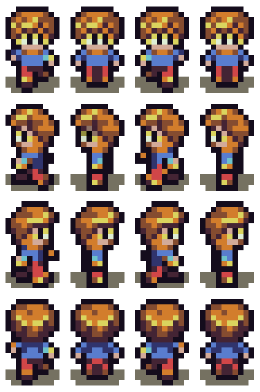
Each frame is 12 pixels wide x 18 pixels high.

The top row is four frames of walking down (forward)

Row 2 is 4 frames walking left.

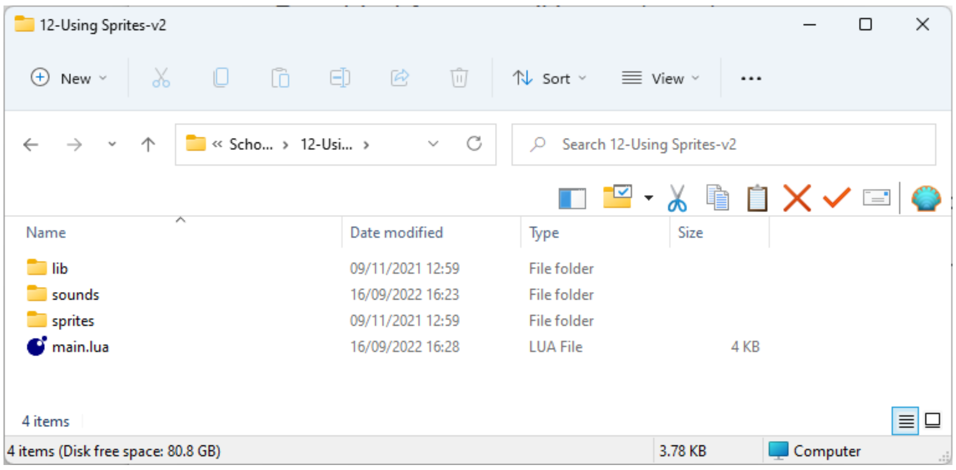
Row 3 is 4 frames walking right.

Row 4 is 4 frames walking up (away)



When the player is moved, the correct animation for the direction of travel plays at 5fps.

You will need 3 folders inside your project: lib, sounds and sprites



Name	Date modified	Type	Size
lib	09/11/2021 12:59	File folder	
sounds	16/09/2022 16:23	File folder	
sprites	09/11/2021 12:59	File folder	
main.lua	16/09/2022 16:28	LUA File	4 KB

4 items

4 items (Disk free space: 80.8 GB) 3.78 KB Computer

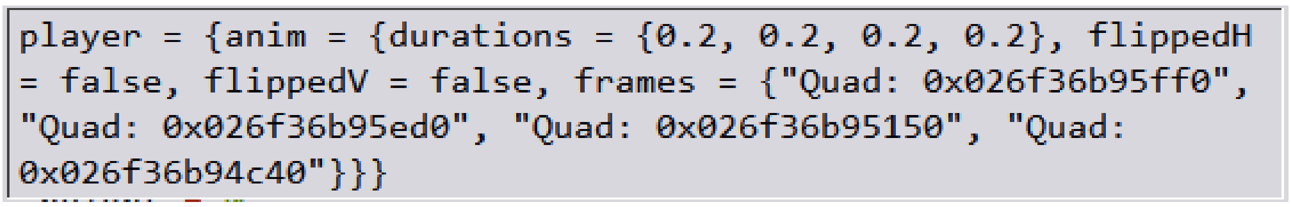
love.load()

```lua
function love.load()
    -- add these lines to debug in ZeroBrane Studio
    if arg[#arg] == "-debug" then
        print("running in debug mode")
        require("mobdebug").start()
    end
    -- load background image
    background = love.graphics.newImage('sprites/background.png')
    WIDTH = background:getWidth()    -- get width and height of background image
    HEIGHT = background:getHeight()
    love.window.setMode(WIDTH, HEIGHT) -- change window size to match background

    anim8 = require 'lib/anim8'
    love.graphics.setDefaultFilter("nearest", "nearest") -- graphics filter for pixel scaling

    player =
    {
        x = 0,
        y = 0,
        width = 0,
        height = 0,
        speed = 5,
        spriteSheet = love.graphics.newImage('sprites/player-sheet.png'),
        grid = {},
        anim = {},
        animations = {},
        scale = {}
    }
    -- newGrid(frameWidth, frameHeight, imageWidth, imageHeight, left, top, border)
    -- create a new grid from the supplied spritesheet
    -- you will need to know the size of each frame in advance. player-sheet.png has 12x18 px sprites
    -- in a grid 4 frames wide, 4 frames deep
    player.grid = anim8.newGrid( 12, 18, player.spriteSheet:getWidth(), player.spriteSheet:getHeight())
    -- newAnimation(frames, durations, onLoop)
    -- player.grid('1-4', 1) use frames 1-4 on the top row
    player.animations.down  = anim8.newAnimation( player.grid('1-4', 1), 0.2 ) -- top row
    player.animations.left  = anim8.newAnimation( player.grid('1-4', 2), 0.2 ) -- row 2
    player.animations.right = anim8.newAnimation( player.grid('1-4', 3), 0.2 ) -- row 3
    player.animations.up    = anim8.newAnimation( player.grid('1-4', 4), 0.2 ) -- row 4

    player.anim = player.animations.down  -- start facing down (towards screen)
    player.width, player.height = player.anim:getDimensions()  -- needed for controlling player
    player.scale.x = 4    -- change to enlarge / shrink sprite display width
    player.scale.y = 4    -- change to enlarge / shrink sprite display height
    player.x = (WIDTH - (player.width * player.scale.x)) / 2    -- centre the player
    player.y = (HEIGHT - (player.height * player.scale.y)) / 2
    step = love.audio.newSource("sounds/FootstepGrass01.wav","static")
    bgmusic = love.audio.newSource("sounds/AmbientNatureOutside.wav","stream")
    love.audio.play(bgmusic)
end
```

This is just love.load(). It has got a lot more complex.

The first 4 lines allow you to put stop points in the code when running in ZeroBrane Studio, so you can keep track of changes to variables, tables etc.

If a stop point is placed at the end of this function, and the mouse hovered over any instance of the word player:



```
player = {anim = {durations = {0.2, 0.2, 0.2, 0.2}, flippedH
= false, flippedV = false, frames = {"Quad: 0x026f36b95ff0",
"Quad: 0x026f36b95ed0", "Quad: 0x026f36b95150", "Quad:
0x026f36b94c40"}}}
```

The contents of the table can be seen in the tooltip.

Now doesn’t that make things so much clearer…

The background image is used as in the previous project to set the window size.

The anim8.lua library is loaded ready for use. It is used to produce animations from the spritesheet shown.

Start the library by creating a new grid:

```
player.grid = anim8.newGrid(12, 18, player.spriteSheet:getWidth(), player.spriteSheet:getHeight())
```

This is based on:

```
newGrid(frameWidth, frameHeight, imageWidth, imageHeight, left, top, border)
```

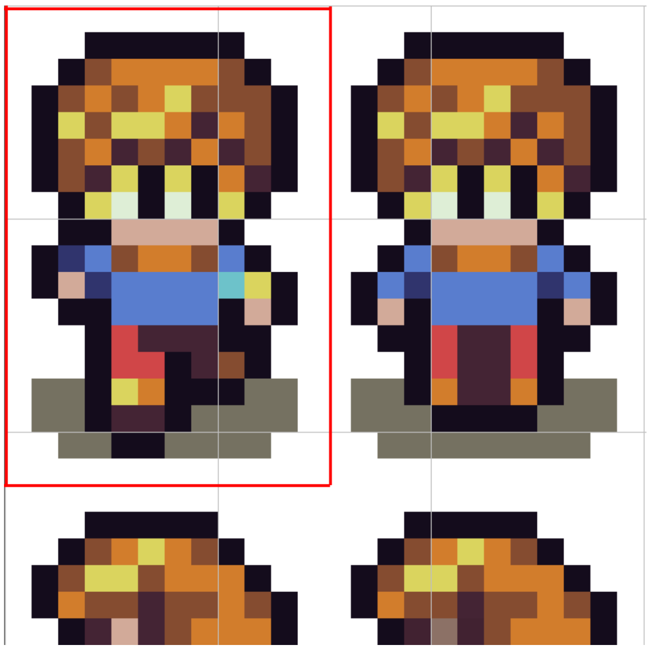
The frameWidth, frameHeight is the width, and height in pixels of each frame.

It has to be calculted by inspecting the spritesheet in an image editor:

The grid shown here is 8x8, so you can count the size of each frame (including any border round it) is 12 x 18

The imageWidth and imageHeight are calculated from the size of the spritesheet itself, which is 48 x 72 pixels, but this can be read with code.

The remaining parameters are not used here, but allow a starting point away from the top left, and a border if there is one.



The player table is created as in earlier projects, but has additional properties:

- `spriteSheet = love.graphics.newImage('sprites/player-sheet.png'),`
- `grid = {},`
- `anim = {},`
- `animations = {},`
- `scale = {}`

The next stage is to create a series of animations from this grid.
The first is:

```
player.animations.down = anim8.newAnimation(player.grid('1-4', 1), 0.2 )
```

This is based on

```
newAnimation(frames, durations, onLoop)
```

so the `player.grid('1-4', 1)` specifies:

“use frames 1 to 4 inclusive from row 1, with a framerate of 0.2 seconds per frame”
and store it in the player.animations.down variable.

Repeat for the remaining 3 rows and store in .left, .right and .up
onLoop is a boolean flag whether the animation should loop or not (default false): not used here.

Set the default animation when the game starts:

```
player.anim = player.animations.down  -- start facing down (towards screen)
```

Get the width and height of the player:

```
player.width, player.height = player.anim:getDimensions()  -- needed for controlling player
```

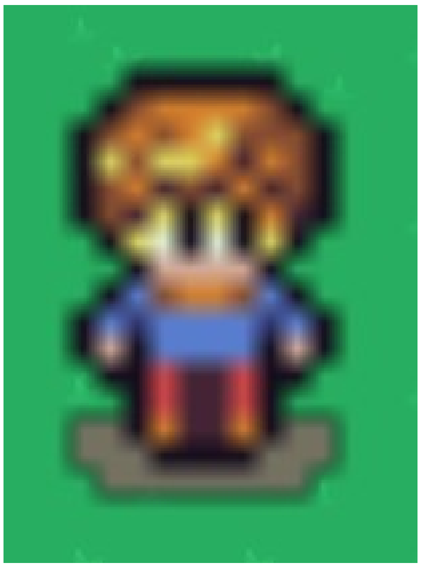
The sprite is very small, and would look totally insignificant unless scaled up:

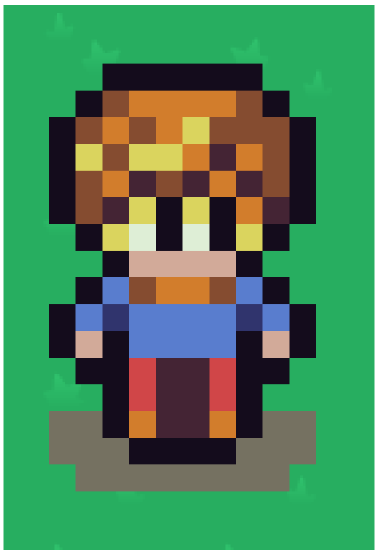
```
player.scale.x = 4     -- change to enlarge / shrink sprite display width
player.scale.y = 4     -- change to enlarge / shrink sprite display height
```

This is where the line much earlier is important:

```
love.graphics.setDefaultFilter("nearest", "nearest") -- graphics filter for pixel scaling
```

This makes sure the pixellated appearance is kept.
Without it the sprite looks like this at scale of 10/10:





The sprite is centred in the background with:

```
player.x = (WIDTH - (player.width * player.scale.x)) / 2    -- centre the player
player.y = (HEIGHT - (player.height * player.scale.y)) / 2
```

Note how the scale factor is taken into account.

Finally the background music and sound effect for taking steps are loaded in:

```
step = love.audio.newSource("sounds/FootstepGrass01.wav","static")
bgmusic = love.audio.newSource("sounds/AmbientNatureOutside.wav","stream")
love.audio.play(bgmusic)
```

The footsteps effect is a very small file, so is loaded into memory with the “static” parameter.
The background is a bigger, longer file so is loaded in as a “stream”.
This file is started straight away.

love.update() and love.draw()

```lua
function love.update(dt)
    local isMoving = false

    if love.keyboard.isDown("right") then
        player.x = math.min(WIDTH - (player.width * player.scale.x), player.x + player.speed)
        player.anim = player.animations.right
        isMoving = true
    end

    if love.keyboard.isDown("left") then
        player.x = math.max(0, player.x - player.speed)
        player.anim = player.animations.left
        isMoving = true
    end

    if love.keyboard.isDown("down") then
        player.y = math.min(HEIGHT - player.height * player.scale.y, player.y + player.speed)
        player.anim = player.animations.down
        isMoving = true
    end

    if love.keyboard.isDown("up") then
        player.y = math.max(0, player.y - player.speed)
        player.anim = player.animations.up
        isMoving = true
    end

    if isMoving then
        love.audio.play(step)
    else
        player.anim:gotoFrame(2)
    end

    player.anim:update(dt)
    if not bgmusic:isPlaying() then
        love.audio.play(bgmusic)
    end
end

function love.draw()
    love.graphics.draw(background, 0, 0)
    -- Animation:draw(image, x, y, r, sx, sy, ox, oy, kx, ky)
    player.anim:draw(player.spriteSheet, player.x, player.y, nil, player.scale.x, player.scale.y)
end
```

This is not too different to the love.update() function in the last project.

There is a local boolean variable isMoving created every frame, and set to false.
If any of the movement keys are pressed this variable is set to true.
If it is true, the footstep audio is played

```lua
love.audio.play(step)
```

else the animation is stopped on frame 2, which is the standing posture.

```lua
player.anim:gotoFrame(2)
```

If a movement key is pressed then the position of the sprite is updated using the clamping as before, and the correct animation for that direction is started:

```
    player.x = math.min(WIDTH - (player.width * player.scale.x), player.x + player.speed)
    player.anim = player.animations.right
    isMoving = true
```

The anim8 library is then updated:

```
    player.anim:update(dt)
```

Finally there is a check to see if the background music is still playing, and if not, it is re-started:

```
    if not bgmusic:isPlaying() then
        love.audio.play(bgmusic)
    end
```

Drawing the sprite is passed to the anim8 library to do all the work!

```
player.anim:draw(player.spriteSheet, player.x, player.y, nil, player.scale.x, player.scale.y)
```

Again the scale factor is passed through to the drawing routine.

Callback functions

https://love2d.org/wiki/Category:Callbacks

So far the functions used to determine keypresses and mouse position have been where we have asked for information eg:

```
 if love.keyboard.isDown('w') then
```

This asks “Is the ‘w’ key pressed?”

```
 local mx, my = love.mouse.getPosition()
```

This assigns the mouse x and mouse y coordinates to the variables mx, my at the time the function is called.

There is another type: the callback function.

These work in reverse. From https://love2d.org/wiki/Tutorial:Callback_Functions

In a regular function like love.graphics.draw or math.floor, you call it and LÖVE or Lua does something. A callback, on the other hand, is a function that you code and LÖVE calls at certain times. This makes it easy to keep your code organized and optimal.

An example of this is the love.keypressed(key) function

```
function love.keypressed(key)
   if key == 'b' then
      text = "The B key was pressed."
   elseif key == 'a' then
      a_down = true
   end
end
```

If this function exists in your code, every time a key is pressed, it will run. The engine tells you something has happenned, not you asking it.

This can be used to get text input from the user.

The mouse has similar functions:

```
love.mousepressed( x, y, button, istouch )
love.mousereleased( x, y, button, istouch)
```

The next project uses a mouse callback.

13-Button Clicker

https://github.com/Inksaver/LuaForSchools/tree/main/Love2D%20School%20Tutorials/13-Button%20Clicker

```lua
local HEIGHT = love.graphics.getHeight()
local WIDTH = love.graphics.getWidth()
local wait = 3
local colour = {math.random(), math.random(), math.random()}

local function distanceBetween(x1, y1, x2, y2)
    return math.sqrt((y2 - y1)^2 +(x2 - x1)^2)
end

function love.mousepressed(x, y, b, istouch)
    if b == 1 and gameState == 2 then
      if distanceBetween(button.x, button.y, love.mouse.getX(), love.mouse.getY()) < button.size then
        score = score + 1
        button.x = math.random(button.size, love.graphics.getWidth() - button.size)
        button.y = math.random(button.size, love.graphics.getHeight() - button.size)
        colour = {math.random(), math.random(), math.random()}
      end
    end
    if gameState == 1 then
        gameState = 2
        timer = 10
        score = 0
        wait = 3
    end
end
```

The function `distanceBetween()` is a mathematical calculation to determine the distance between two points, given the x and y coordinates of both.

`love.mousepressed(x, y, b, istouch)` is a callback function that runs every time the user clicks the mouse.

It checks which gamestate is current, where 1 = not started, 2 = playing, 3 = 3 second pause after game finished.
If left button and game running, the position of the mouse is checked to see if it is within the area of the circle called button, ie a ‘hit’. If hit, then score is increased, a new button colour and position chosen ready for the love.draw() function.

If game in not started, then the gamestate is changed to 2 (playing), timer, score and waiting variables reset.

love.load()

```lua
function love.load()
    if arg[#arg] == "-debug" then require("mobdebug").start() end
    love.graphics.setBackgroundColor(1, 1, 1)
    print("Window background set to white")

    button =
    {
        x = 200,
        y = 200,
        size = 50
    }
    score = 0
    timer = 10
    gameState = 1
    myFont = love.graphics.newFont(40)
end
```

A ‘button’ table is created, and variables set for score, timer and gameState set.

A font object is created at size 40. It is possible to use any font you have a file for, but will not automatically find fonts installed on the system, so only use those you have inside the project folder structure, eg “fonts/calibri.ttf” by copying it from System fonts, or downloading one from an online source eg:

https://www.1001fonts.com/

```
myFont = love.graphics.newFont("fonts/calibri.ttf", 40)
```

This font is used to display text to the screen in `love.draw()`

```
function love.update(dt)
    if gameState == 2 then
        if timer > 0 then
            timer = timer - dt
        end
        if timer < 0 then
            timer = 0
            gameState = 3
        end
    elseif gameState == 3 then
        wait = wait - dt
        if wait < 0 then
            gameState = 1
        end
    end
end
```

The update() function checks the current gameState and takes appropriate action.

gameState 1: no action, nothing to update.

gameState 2: update the timer by reducing by dt (delta time) This is the time in seconds passed since the last time the update() function was called, normally around 0.016 seconds.
If the timer goes below 0 then change the gameState to 3 (pause for 3 seconds) while the final score is displayed and no more circles are drawn.

gameState 3: (pause 3 seconds) the variable wait is decreased by dt, and when it goes below 0 (it started at 3) then the gameState is reset to 1 so the game starts again.

```
function love.draw()
    love.graphics.setFont(myFont)
    love.graphics.setColor(0, 0, 0) -- should be 1,1,1 if on black background

    if gameState == 1 then -- menu
        love.graphics.printf("Click anywhere to begin",
                             0,
                             HEIGHT / 2 - 20,
                             WIDTH,
                             "center")
    else
        love.graphics.print("Score: "..score, WIDTH * 0.2, HEIGHT * 0.9)
        love.graphics.print("Time: "..math.ceil(timer), WIDTH * 0.6, HEIGHT * 0.9)
        if gameState == 2 then
            love.graphics.setColor(colour)
            love.graphics.circle('fill', button.x, button.y, button.size)
        end
    end
end
```

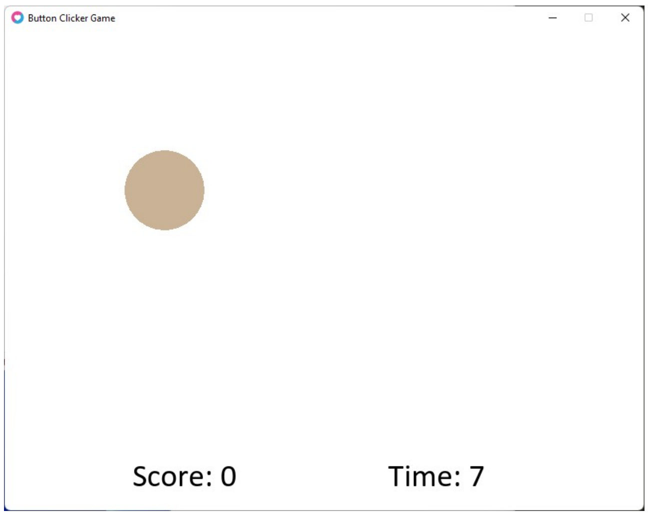
In the same way as colours are set in love.draw(), so are fonts for printing text

Again the behaviour of love.draw() is dependant on the current gameState.

gameState = 1: use love.printf() which is 'formatted' print with a number of parameters see https://love2d.org/wiki/love.graphics.printf The most useful aspect is the alignment, rotation and scaling capabilities

```
love.graphics.printf( text, x, y, limit, align, r, sx, sy, ox, oy, kx, ky )

text = "Click anywhere to begin"
x = 0,
y = HEIGHT / 2 - 20,
limit = WIDTH
align = "center"
```

gameState 2 / 3: print the score and timer to screen.
   If gameState == 2 then draw the button


Button Clicker Game
Score: 0
Time: 7