A PROJECT REPORT ON

FLAPPY BIRDS USING LUA

A DISSERTATION
SUBMITTED FOR THE MID TERM PROJECT EVALUATION (MTE) FOR
THE COMPLETION OF COURSE COMPUTER GRAPHICS (CO-313)

SUBMITTED BY
VISHRUTH KHARE
(2K18/CO/393)
VISHWAS AGARWAL
(2K18/CO/395)

UNDER THE SUPERVISION OF
Ms. CHINGMUANKIM NAULAK

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi- 110042
Session: November, 2020

DECLARATION

We, **Vishruth Khare**, Roll No. 2K18/CO/393 and **Vishwas Agarwal**, Roll No. 2K18/CO/395 of B.Tech Computer Engineering, hereby declare that the project titled **"Flappy Birds Using Lua"** which is submitted by us to the Department of Computer Engineering, Delhi Technological University (DTU), Delhi for the mid-term project evaluation (MTE), is complete and working and has properly cited all it's quotations along with a working model.

New Delhi
India
November '20

ABSTRACT

Flappy bird is a mobile game turned Web Variation in which a player controls a bird's flight to refrain from colliding with obstacles. Tapping the keys for a longer period of time allows the bird to fly higher, while letting go causes the bird to fall down. This project will bring the mobile game to life ny creating a web variant which is unprecedented. The FPGA will render an image of the bird flying through an environment, and display the flapping motion of the wings according to the player's ability to tap the relevant keys. We will be imploring Lua Love2D framework for the implementation of a aforementioned game development. The necessary gaming design and development have been kept in mind while deploying the game features. We believe that this first attempt of the employment of Web frame application in a Mobile Based game will be an important step for a wider deployment in the research area of computer games.

INTRODUCTION

The game is a side-scroller where the player controls a bird, attempting to fly between rows of green pipes, which are equally sized gaps placed at random heights, without coming into contact them. Each successful pass through a pair of pipes awards the player one point. If the player touches the pipes, it ends the game. The bird briefly flaps upward each time the player taps the key; if the key is not tapped, the bird falls due to gravity. The player is awarded with several milestones, such as a bronze medal if they reached twenty points, a silver medal from Forty points, a gold medal from Fifty, and a platinum medal from Hundred points. The achievements get stored in the collectible haul.

Compatibility : Any system with Love2D framework installed can compile, execute and play this game.

TECHNOLOGY USED

We have used **Lua Programming Language** for the development of the project. Lua is an extensible, lightweight programming language written in C. It started as an in-house project in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. It was designed from the beginning to be a software that can be integrated with the code written in C and other conventional languages. It does not try to do what C can already do but aims at offering what C is not good at: a good distance from the hardware, dynamic structures, no redundancies, ease of testing and debugging. For this, Lua has a safe environment, automatic memory management, and good facilities for handling strings and other kinds of data with dynamic size.

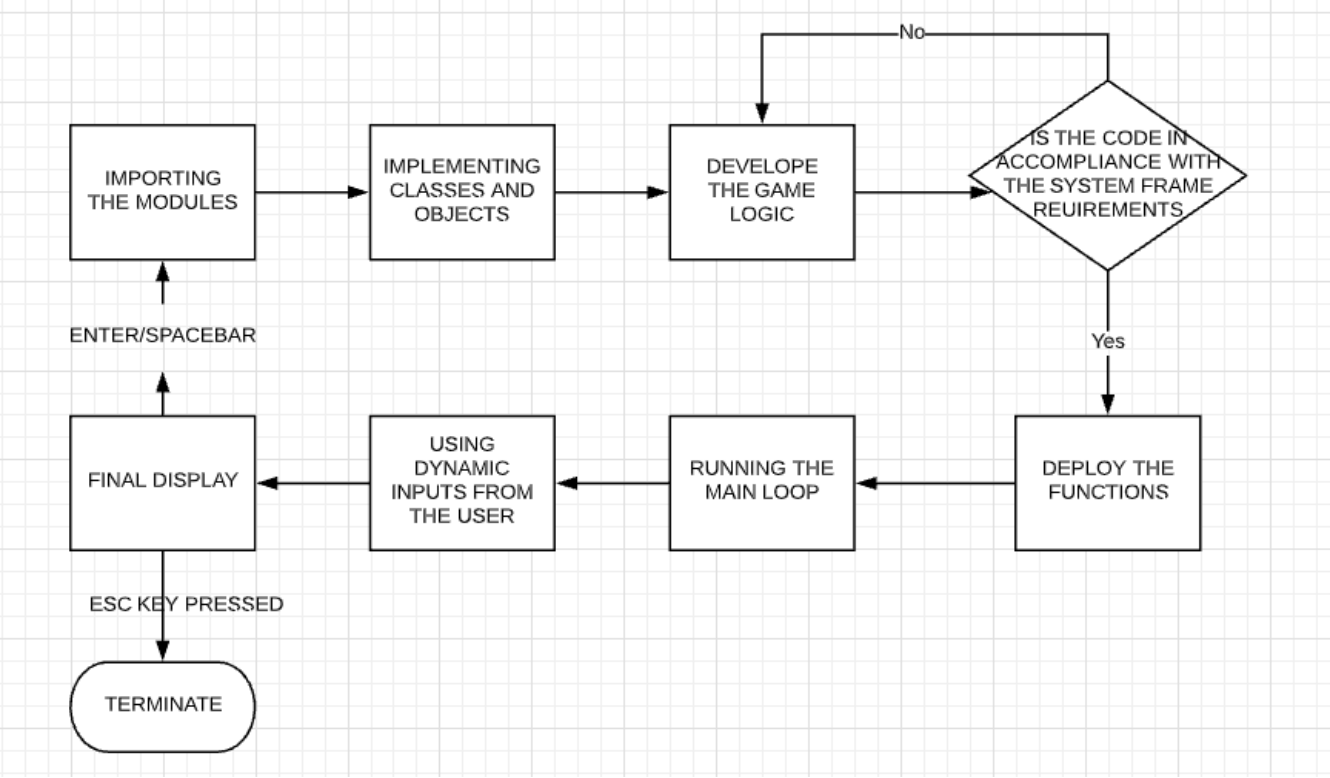
Features : Lua provides a set of unique features that makes it distinct from other languages. These include:

- Extensible
- Simple
- Efficient
- Portable
- Free and open

Lua Interpreter : The Lua Interpreter is written in ANSI C, hence it is highly portable and can run on a vast spectrum of devices from high-end network servers to small devices. Both Lua's language and its interpreter are mature, small, and fast. It has evolved from other programming languages and top software standards. Being small in size makes it possible for it to run on small devices with low memory.

METHODOLOGY

- We first import the desired modules and import the Love2D framework in out Script
- Then the classes and objects are implemented for the bird, pipes, background, Achievements, Pause Screen and the final display screen.
- We develop the relevant game logic that will ensure smooth play

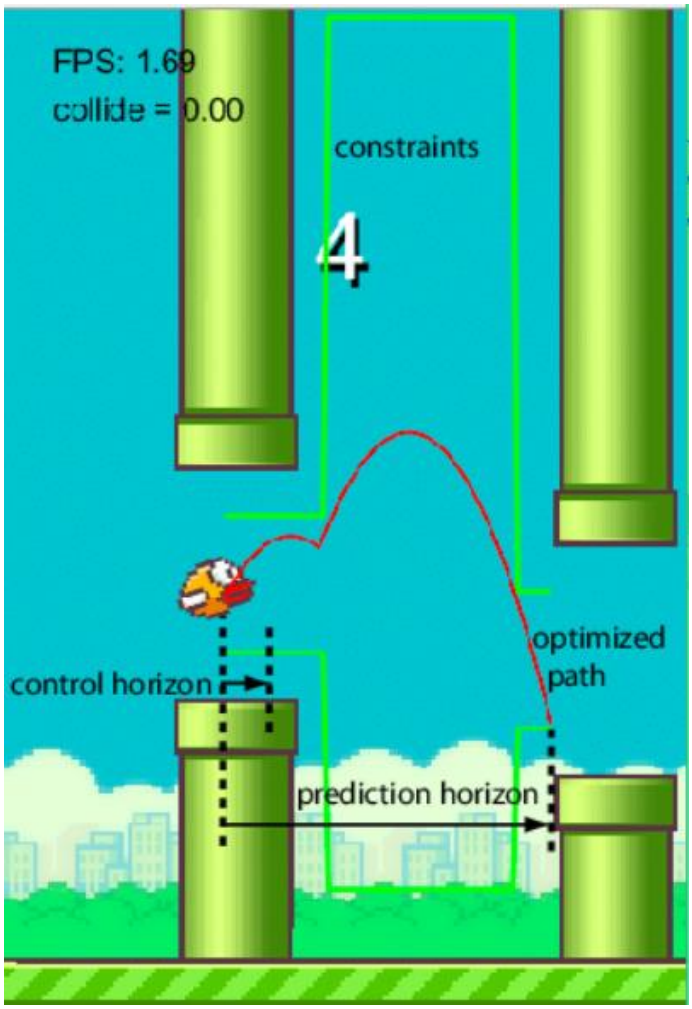


and in accomplice with the screen resolution and system settings.

- Once everything is in check, the player executes the code which means the functions get deployed in the main.lua code.
- Once the main function gets compiled, it keeps on executing until the player presses hit an obstacle or makes an illegal move.
- Finally the score along with the trophy is displayed on the screen and the main.lua function keeps on looping until the Player presses the Escape Key to trigger an escaping sequence that closes the window.

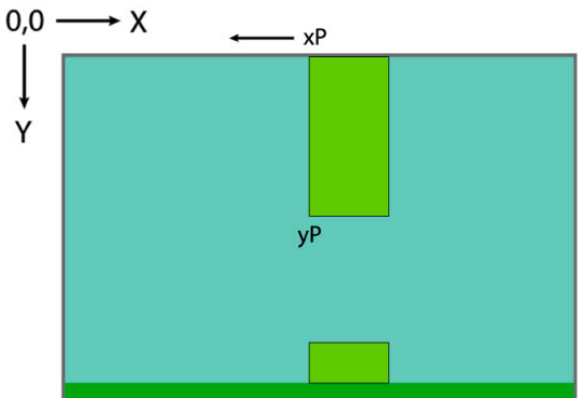
GAME LOGIC

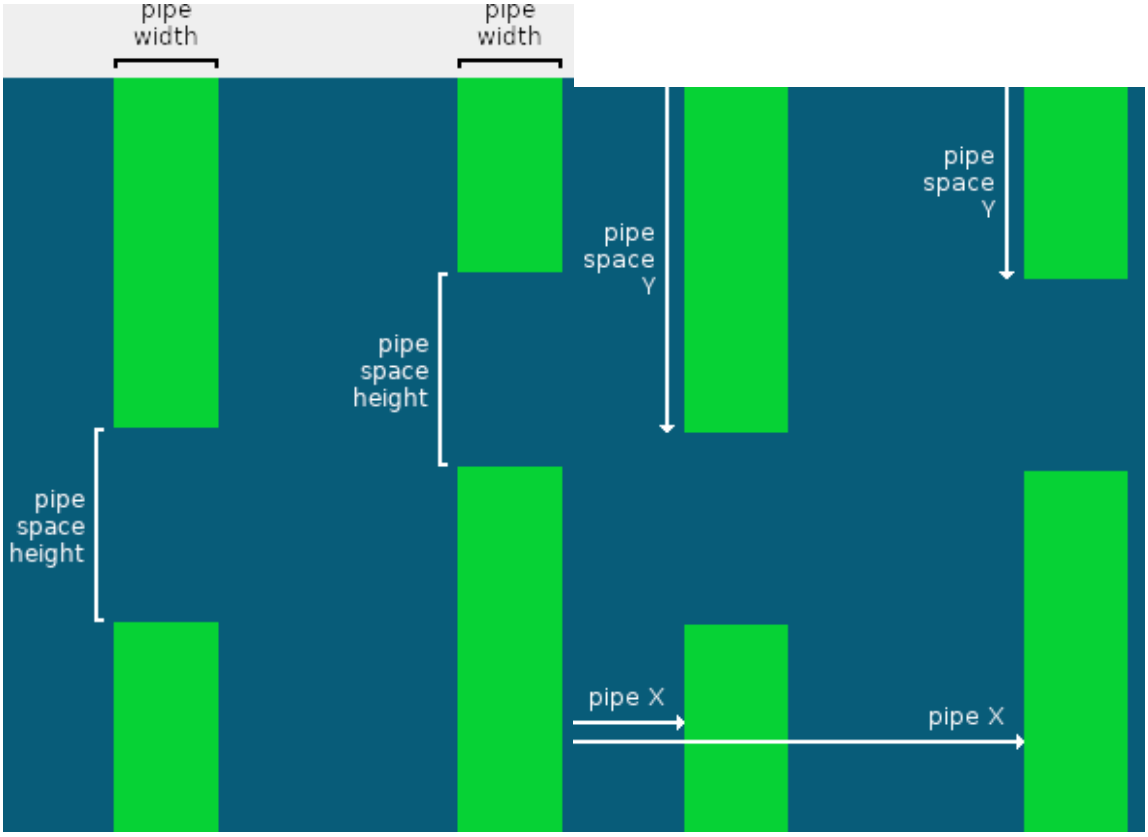
For the successful execution of any game, The game logic has to beconcrete and accordance with the fundamentals of the game.



We have 50 pixels wide pillars which move from right to left and every next pillar has a different random height. In order to make them moving, logically, after each iteration we need to clear the screen and redraw the graphic with the pillars at their new position. However, we cannot do that because of the low refresh rate of the screen, which would cause flickering of the graphics. In order to activate all of its pixels the screen needs a bit more time.

The graphics we follow is the standard system where the screen moves in the X-Direction and the user will have to trigger the up-down movement by the press of the key.





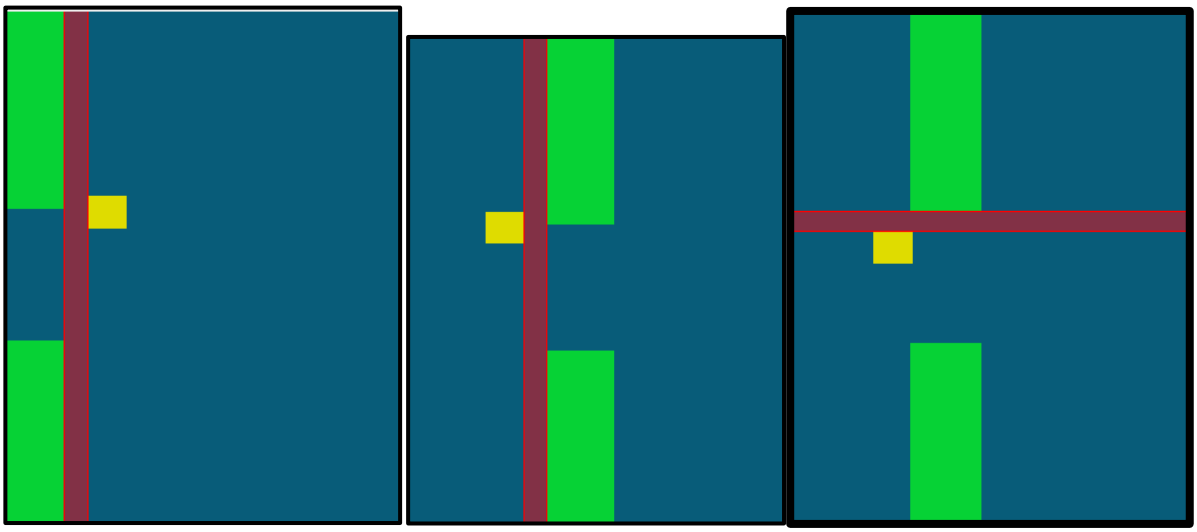
The bird's Y position moves down by a certain speed each frame. Each frame this speed is increased to simulate gravity.
When a key is pressed, the bird's speed is set to a high number in the upward direction.
The pipes share the same width, and the same space height between the segments. Each pipe has its own X position, and Y position the space.

Since there are only two pipes in the playing area at one time, the information for only two pipes needs to stored. Once a pipe goes out of the playing area, it gets a new space position and its X position is reset to the right edge of the playing area.

The bird has collided with the top segment if...

- The left edge of the bird is to the left of the right edge of the pipe
- The right edge of the bird is to the right of the left edge of the pipe

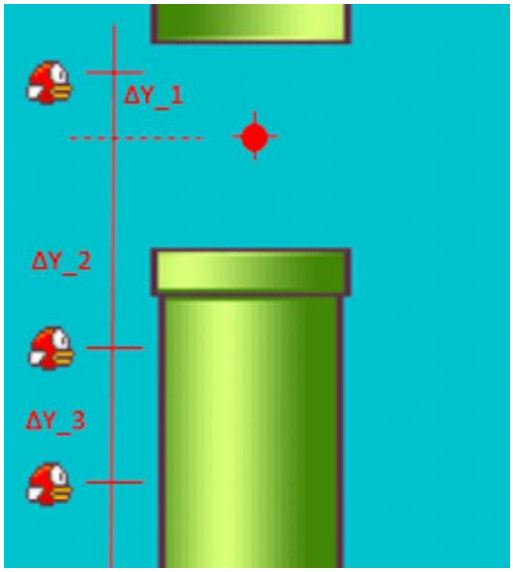
- The top edge of the bird is above the bottom edge of the pipe segment.



(Collision for the upper Pipe)

BIRD and GRAVITY

The bird's Y position is made into a variable, and every frame it increases by 30 multiplied by dt, making the bird move down 30 pixels per second. Instead of moving at a constant speed, the number added to the bird's Y position also increases over time.



The bird's speed is made into a variable which starts at 0, and every frame it increases by a fixed scalar (Gravity) times the frame height.

PIPES

Pipes are the most important part of the game because they have to be made sensitive to collision with the bird. Their appearance in the game is totally randomized thereby making it exceedingly difficult for the player to cope as the game proceeds.

Two rectangles are drawn for the upper and lower segments.

The top rectangle's height is set to where the space between the two segments starts.
The bottom rectangle starts at the Y position of the top rectangle's height plus the amount of space between them

The bottom rectangle's height is the playing area's height, minus the height of the top rectangle and the space.
The pipe width is the same for both segments, so this is made into a variable.

DISPLAY SCREEN

We also have several achievements as Trophies as the player keeps on crossing the hurdles, his score gets incremented and on

the basis of the score. Digital trophies are given to the player. Also the background is a png file that is attached and it keeps on re-looping as the bird progresses throughout the game. Various other facades are there in the game that give a complete look of the dynamism of the game.

DESIGN AND REQUIREMENTS

The following sections are going to be focusing on the core development of the game along with implementation techniques and the code.

User Actions –

KEY	ACTION
Spacebar (in-game) / Left Click	Jump
Spacebar on Pause Screen	Restart

Technical Game Details –

PARAMETER	VALUE
Frames per second	60
Resolution	1280x720 (below this is self-adjusting)
Framework	Love2D
Pipe Gap	200px
Gravity	30px per second

As already mention, the game continues until the user wants to. The resolution is of self-adjusting nature as the Virtual Width and Height have been set to 512x218. Below this value, the pixels start to distort and collapse. The frame rate ideally is made accustomed to 60 frames per second but that can be changed according to the need of the user. Gravity works at 30 pixel per second which when decreased increases the difficulty of the game.

SOURCE CODE

- MAIN

```
push = require 'push'

-- classic OOP class library
Class = require 'class'

-- bird class we've written
require 'Bird'

require 'Pipe'

require 'PipePair'

require 'StateMachine'
require 'states/BaseState'
require 'states/PlayState'
require 'states/TitleScreenState'
require 'states/ScoreState'
require 'states/CountdownState'
require 'states/PauseState'

-- physical screen dimensions
WINDOW_WIDTH = 1280
WINDOW_HEIGHT = 720

-- virtual resolution dimensions
VIRTUAL_WIDTH = 512
VIRTUAL_HEIGHT = 288

-- background image and starting scroll location (X axis)
local background = love.graphics.newImage('images/background.png')
local backgroundScroll = 0

-- ground image and starting scroll location (X axis)
local ground = love.graphics.newImage('images/ground.png')
local groundScroll = 0
```

```
-- speed at which we should scroll our images, scaled by dt
local BACKGROUND_SCROLL_SPEED = 30
local GROUND_SCROLL_SPEED = 60

-- point at which we should loop our background back to X 0
local BACKGROUND_LOOPING_POINT = 413

function love.load()
    -- initialize our nearest-neighbor filter
    love.graphics.setDefaultFilter('nearest', 'nearest')

    math.randomseed(os.time())

    -- app window title
    love.window.setTitle('Fifty Bird')

    smallFont = love.graphics.newFont("fonts/font.ttf", 8)
    mediumFont = love.graphics.newFont("fonts/flappy.ttf", 14)
    flappyFont = love.graphics.newFont("fonts/flappy.ttf", 28)
    hugeFont = love.graphics.newFont("fonts/flappy.ttf", 56)
    love.graphics.setFont(flappyFont)

    -- initialize our virtual resolution
    push:setupScreen(VIRTUAL_WIDTH, VIRTUAL_HEIGHT, WINDOW_WIDTH, WINDOW_HEIGHT, {
        vsync = true,
        fullscreen = false,
        resizable = true
    })

    gStateMachine = StateMachine {
        ['title'] = function () return TitleScreenState() end,
        ['play'] = function () return PlayState() end,
        ['score'] = function () return ScoreState() end,
        ['countdown'] = function () return CountdownState() end,
        ['pause'] = function () return PauseState() end
    }
    gStateMachine:change('title')

```

```
    sounds = {
        ['explosion'] = love.audio.newSource('sounds/explosion.wav', 'static'),
        ['jump'] = love.audio.newSource('sounds/jump.wav', 'static'),
        ['hurt'] = love.audio.newSource('sounds/hurt.wav', 'static'),
        ['score'] = love.audio.newSource('sounds/score.wav', 'static'),
        ['music'] = love.audio.newSource('sounds/marios_way.mp3', 'static'),
        ['pause'] = love.audio.newSource('sounds/pause.wav','static')
    }

    medals = {
        ['rock'] = love.graphics.newImage('images/medals/rock.png'),
        ['bronze'] = love.graphics.newImage('images/medals/bronze.png'),
        ['silver'] = love.graphics.newImage('images/medals/silver.png'),
        ['gold'] = love.graphics.newImage('images/medals/gold.png'),
        ['crown'] = love.graphics.newImage('images/medals/crown.png')
    }

    sounds['music']:setLooping(true)
    sounds['music']:play()

    love.keyboard.keysPressed = {}

    love.mouse.buttonsPressed = {}
end

function love.resize(w, h)
    push:resize(w, h)
end

function love.keypressed(key)
    love.keyboard.keysPressed[key] = true

    if key == 'escape' then
        love.event.quit()
    end

end

function love.keyboard.wasPressed( key)
```

```
    return love.keyboard.keysPressed[key]
end

function love.mousepressed(x, y, button)
    love.mouse.buttonsPressed[button] = true
end

function love.mouse.wasPressed(button)
    return love.mouse.buttonsPressed[button]
end

function love.update(dt)

    -- scroll background by preset speed * dt, looping back to 0 after the looping point
    backgroundScroll = (backgroundScroll + BACKGROUND_SCROLL_SPEED * dt)
        % BACKGROUND_LOOPING_POINT

    -- scroll ground by preset speed * dt, looping back to 0 after the screen width passes
    groundScroll = (groundScroll + GROUND_SCROLL_SPEED * dt)
        % VIRTUAL_WIDTH

    gStateMachine:update(dt)



    love.keyboard.keysPressed = {}

    love.mouse.buttonsPressed = {}

end

function love.draw()
    push:start()

    -- draw the background at the negative looping point
    love.graphics.draw(background, -backgroundScroll, 0)

    gStateMachine:render()

```

```
    -- draw the ground on top of the background, toward the bottom of the screen,
    -- at its negative looping point
    love.graphics.draw(ground, -groundScroll, VIRTUAL_HEIGHT - 16)

    push:finish()
end
```

- BIRD

```
Bird = Class{}

local GRAVITY = 20

function Bird:init()
    -- load bird image from disk and assign its width and height
    self.image = love.graphics.newImage('images/bird.png')
    self.width = self.image:getWidth()
    self.height = self.image:getHeight()

    -- position bird in the middle of the screen
    self.x = VIRTUAL_WIDTH / 2 - (self.width / 2)
    self.y = VIRTUAL_HEIGHT / 2 - (self.height / 2)

    self.dy = 0

end

function Bird:collides( pipe )
    if (self.x + 2) + (self.width - 4) >= pipe.x and self.x + 2 <= pipe.x + PIPE_WIDTH then
        if (self.y + 2) + (self.height - 4) >= pipe.y and self.y + 2 <= pipe.y + PIPE_HEIGHT then
            return true
        end
    end

    return false
    -- body
end

function Bird:render()
    love.graphics.draw(self.image, self.x, self.y)
```

```
end

function Bird:update(dt)
    self.dy = self.dy + GRAVITY * dt
    if love.keyboard.wasPressed('space') or love.mouse.wasPressed(1) then
        self.dy = -5
        sounds['jump']:play()
    end
    self.y = self.y  + self.dy

end

```

- PIPE-PAIR

```
Pipe = Class{}

local PIPE_IMAGE = love.graphics.newImage('images/pipe.png')

local PIPE_SCROLL = -60

PIPE_SPEED = 60

PIPE_HEIGHT = 288
PIPE_WIDTH = 70

function Pipe:init(orientation, y)
    self.x = VIRTUAL_WIDTH

    self.y = y

    self.width = PIPE_IMAGE:getWidth()
    self.height = PIPE_HEIGHT

    self.orientation = orientation
end

function Pipe:update(dt)
end

```

```
function Pipe:render()
    love.graphics.draw(PIPE_IMAGE, self.x, (self.orientation == 'top' and self.y + PIPE_HEIGHT or self.y), 0, 1, self.orientation == 'top' and -1 or 1)
end
```

- STATE MACHINE

```
StateMachine = Class{}

function StateMachine:init(states)
    self.empty = {
        render = function() end,
        update = function() end,
        enter = function() end,
        exit = function() end
    }
    self.states = states or {} -- [name] -> [function that returns states]
    self.current = self.empty
end

function StateMachine:change(stateName, enterParams)
    assert(self.states[stateName]) -- state must exist!
    self.current:exit()
    self.current = self.states[stateName]()
    self.current:enter(enterParams)
end

function StateMachine:update(dt)
    self.current:update(dt)
end

function StateMachine:render()
    self.current:render()
end
```

- DYNAMIC DISPLAY AND PUSH

```
local push = {

  defaults = {
    fullscreen = false,
    resizable = false,
    pixelperfect = false,
    highdpi = true,
    canvas = true
  }

}
setmetatable(push, push)

--TODO: rendering resolution?
--TODO: clean up code

function push:applySettings(settings)
  for k, v in pairs(settings) do
    self["_" .. k] = v
  end
end

function push:resetSettings() return self:applySettings(self.defaults) end

function push:setupScreen(WWIDTH, WHEIGHT, RWIDTH, RHEIGHT, settings)

  settings = settings or {}

  self._WWIDTH, self._WHEIGHT = WWIDTH, WHEIGHT
  self._RWIDTH, self._RHEIGHT = RWIDTH, RHEIGHT

  self:applySettings(self.defaults) --set defaults first
  self:applySettings(settings) --then fill with custom settings

  love.window.setMode( self._RWIDTH, self._RHEIGHT, {
    fullscreen = self._fullscreen,
    resizable = self._resizable,
    highdpi = self._highdpi
```

```
    } )

    self:initValues()

    if self._canvas then
      self:setupCanvas({ "default" }) --setup canvas
    end

    self._borderColor = {0, 0, 0}

    self._drawFunctions = {
      ["start"] = self.start,
      ["end"] = self.finish
    }

    return self
  end

  function push:setupCanvas(canvases)
    table.insert(canvases, { name = "_render" }) --final render

    self._canvas = true
    self.canvases = {}

    for i = 1, #canvases do
      self.canvases[i] = {
        name = canvases[i].name,
        shader = canvases[i].shader,
        canvas = love.graphics.newCanvas(self._WWIDTH, self._WHEIGHT)
      }
    end

    return self
  end

  function push:setCanvas(name)
    if not self._canvas then return true end
    return love.graphics.setCanvas( self:getCanvasTable(name).canvas )
  end
```

```
function push:getCanvasTable(name)
  for i = 1, #self.canvases do
    if self.canvases[i].name == name then
      return self.canvases[i]
    end
  end
end
function push:setShader(name, shader)
  if not shader then
    self:getCanvasTable("_render").shader = name
  else
    self:getCanvasTable(name).shader = shader
  end
end

function push:initValues()
  self._PSCALE = self._highdpi and love.window.getPixelScale() or 1

  self._SCALE = {
    x = self._RWIDTH/self._WWIDTH * self._PSCALE,
    y = self._RHEIGHT/self._WHEIGHT * self._PSCALE
  }

  if self._stretched then --if stretched, no need to apply offset
    self._OFFSET = {x = 0, y = 0}
  else
    local scale = math.min(self._SCALE.x, self._SCALE.y)
    if self._pixelperfect then scale = math.floor(scale) end

    self._OFFSET = {x = (self._SCALE.x - scale) * (self._WWIDTH/2), y = (self._SCALE.y - scale) *
(self._WHEIGHT/2)}
    self._SCALE.x, self._SCALE.y = scale, scale --apply same scale to X and Y
  end

  self._GWIDTH = self._RWIDTH * self._PSCALE - self._OFFSET.x * 2
  self._GHEIGHT = self._RHEIGHT * self._PSCALE - self._OFFSET.y * 2
end

--[[ DEPRECATED ]]--
```

```
function push:apply(operation, shader)
  if operation == "start" then
    self:start()
  elseif operation == "finish" or operation == "end" then
    self:finish(shader)
  end
end

function push:start()
  if self._canvas then
    love.graphics.push()
    love.graphics.setCanvas(self.canvases[1].canvas)
  else
    love.graphics.translate(self._OFFSET.x, self._OFFSET.y)
    love.graphics.setScissor(self._OFFSET.x, self._OFFSET.y, self._WWIDTH*self._SCALE.x,
self._WHEIGHT*self._SCALE.y)
    love.graphics.push()
    love.graphics.scale(self._SCALE.x, self._SCALE.y)
  end
end

function push:finish(shader)
  love.graphics.setBackgroundColor(unpack(self._borderColor))
  if self._canvas then
    local _render = self:getCanvasTable("_render")

    love.graphics.pop()

    love.graphics.setColor(255, 255, 255)

    --draw canvas
    love.graphics.setCanvas(_render.canvas)
    for i = 1, #self.canvases - 1 do --do not draw _render yet
      local _table = self.canvases[i]
      love.graphics.setShader(_table.shader)
      love.graphics.draw(_table.canvas)
    end
    love.graphics.setCanvas()

```

```
    --draw render
    love.graphics.translate(self._OFFSET.x, self._OFFSET.y)
    love.graphics.setShader(shader or self:getCanvasTable("_render").shader)
    love.graphics.draw(self:getCanvasTable("_render").canvas, 0, 0, 0, self._SCALE.x, self._SCALE.y)

    --clear canvas
    for i = 1, #self.canvases do
      love.graphics.setCanvas( self.canvases[i].canvas )
      love.graphics.clear()
    end

    love.graphics.setCanvas()
    love.graphics.setShader()
  else
    love.graphics.pop()
    love.graphics.setScissor()
  end
end

function push:setBorderColor(color, g, b)
  self._borderColor = g and {color, g, b} or color
end

function push:toGame(x, y)
  x, y = x - self._OFFSET.x, y - self._OFFSET.y
  local normalX, normalY = x / self._GWIDTH, y / self._GHEIGHT

  x = (x >= 0 and x <= self._WWIDTH * self._SCALE.x) and normalX * self._WWIDTH or nil
  y = (y >= 0 and y <= self._WHEIGHT * self._SCALE.y) and normalY * self._WHEIGHT or nil

  return x, y
end

--doesn't work - TODO
function push:toReal(x, y)
  return x+self._OFFSET.x, y+self._OFFSET.y
end

function push:switchFullscreen(winw, winh)
```

```
    self._fullscreen = not self._fullscreen
    local windowWidth, windowHeight = love.window.getDesktopDimensions()

    if self._fullscreen then --save windowed dimensions for later
      self._WINWIDTH, self._WINHEIGHT = self._RWIDTH, self._RHEIGHT
    elseif not self._WINWIDTH or not self._WINHEIGHT then
      self._WINWIDTH, self._WINHEIGHT = windowWidth * .5, windowHeight * .5
    end

    self._RWIDTH = self._fullscreen and windowWidth or winw or self._WINWIDTH
    self._RHEIGHT = self._fullscreen and windowHeight or winh or self._WINHEIGHT

    self:initValues()

    love.window.setFullscreen(self._fullscreen, "desktop")
    if not self._fullscreen and (winw or winh) then
      love.window.setMode(self._RWIDTH, self._RHEIGHT) --set window dimensions
    end
  end

  function push:resize(w, h)
    local pixelScale = love.window.getPixelScale()
    if self._highdpi then w, h = w / pixelScale, h / pixelScale end
    self._RWIDTH = w
    self._RHEIGHT = h
    self:initValues()
  end

  function push:getWidth() return self._WWIDTH end
  function push:getHeight() return self._WHEIGHT end
  function push:getDimensions() return self._WWIDTH, self._WHEIGHT end

  return push
```

- CLASS

```
local function include_helper(to, from, seen)
    if from == nil then
        return to
```

```
    elseif type(from) ~= 'table' then
        return from
    elseif seen[from] then
        return seen[from]
    end

    seen[from] = to
    for k,v in pairs(from) do
        k = include_helper({}, k, seen) -- keys might also be tables
        if to[k] == nil then
            to[k] = include_helper({}, v, seen)
        end
    end
    return to
end

-- deeply copies `other' into `class'. keys in `other' that are already
-- defined in `class' are omitted
local function include(class, other)
    return include_helper(class, other, {})
end

-- returns a deep copy of `other'
local function clone(other)
    return setmetatable(include({}, other), getmetatable(other))
end

local function new(class)
    -- mixins
    class = class or {}  -- class can be nil
    local inc = class.__includes or {}
    if getmetatable(inc) then inc = {inc} end

    for _, other in ipairs(inc) do
        if type(other) == "string" then
            other = _G[other]
        end
        include(class, other)
    end
```

```
    -- class implementation
    class.__index = class
    class.init    = class.init    or class[1] or function() end
    class.include = class.include or include
    class.clone   = class.clone   or clone

    -- constructor call
    return setmetatable(class, {__call = function(c, ...)
        local o = setmetatable({}, c)
        o:init(...)
        return o
    end})
end

-- interface for cross class-system compatibility (see https://github.com/bartbes/Class-Commons).
if class_commons ~= false and not common then
    common = {}
    function common.class(name, prototype, parent)
        return new{__includes = {prototype, parent}}
    end
    function common.instance(class, ...)
        return class(...)
    end
end

-- the module
return setmetatable({new = new, include = include, clone = clone},
    {__call = function(_,...) return new(...) end})
```

RESULTS

The outcome of the above project is Phenomenal. Since its pre-requisites are nothing therefore it is compatible with all environments be it MacOS, Windows or Linux. The game is of self-adjusting nature hence the output appears evenly on all screen types as well. The game requires a minimal storage of approx. 13Mb hence can it is very space efficient as well. Booting of the game depends on the RAM but still being a 8bit game pre-dominantly, it requires a minimum of 1GB of RAM to run flawlessly. The gameplay is seamless and we have included the following new features in the game:

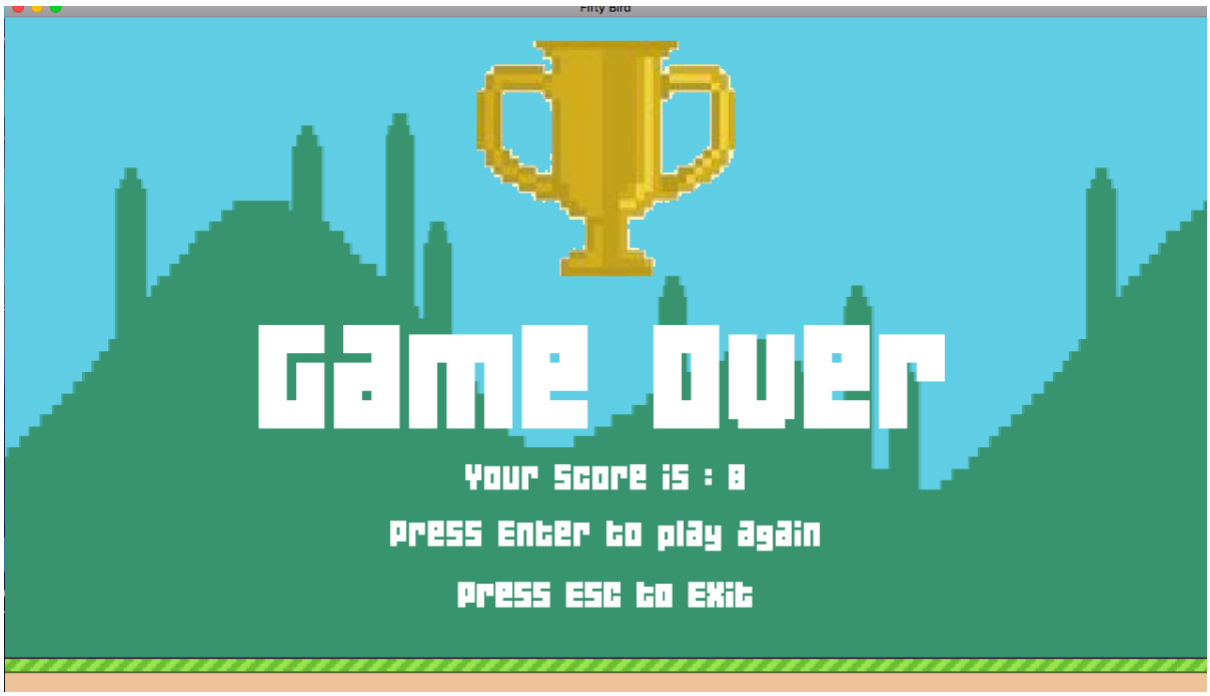
1. Background Sound, collision sound, and flapping sound.
2. Changeable Gravity to change the difficulty of the gameplay.
3. Moveable Pillars as you move up the ladder to enhance the difficulty of the game.
4. Invert color screen to break the concentration (but this has to be executed separately as only the background changes)

Gameplay

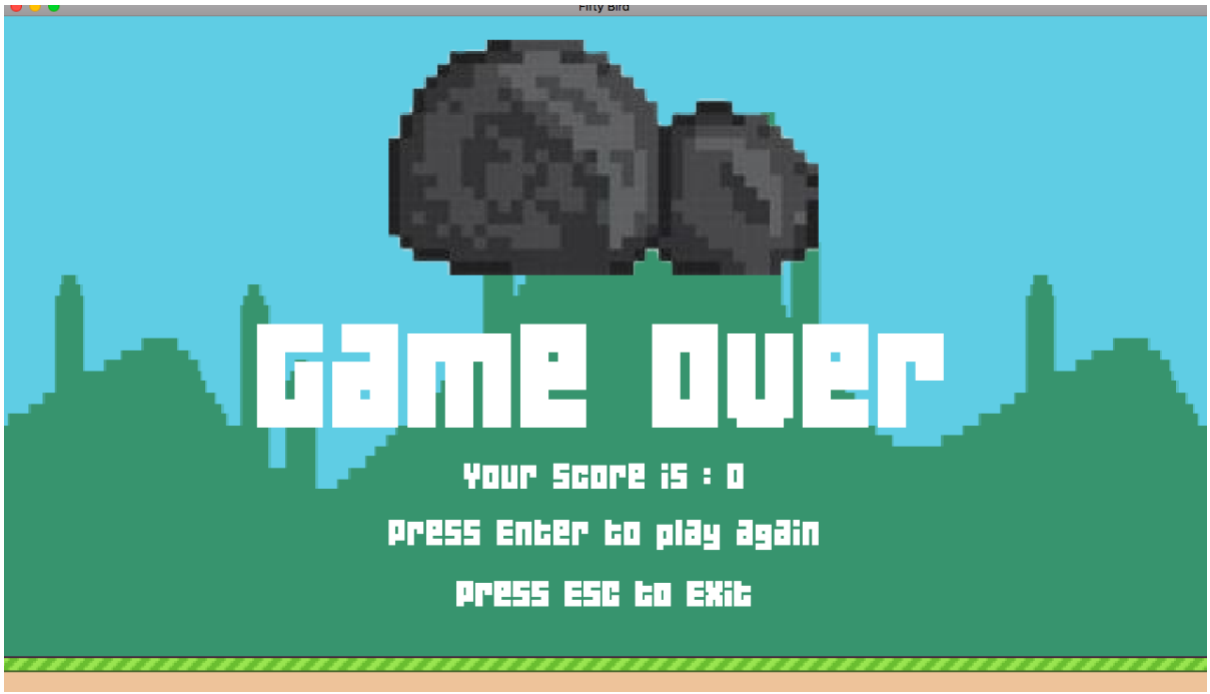
(Opening Screen)

(Timer just before the start of the game)

(In the middle of the game)



(Final display screen along with score and trophy)



(Explosion window and sound if you didn't reach a desired level)

CONCLUSION

The clone of Flappy Birds was implemented successfully using a high level scripting language. The game logic was deployed successfully and the gameplay was also smooth. The game is suitable for all age groups and its availability on Desktop broadens the scope of its usage. The game is fairly simple and we have tried to add more features to the original classic game. Nevertheless, there is still some scope to add more features like creating multiple logins or making multi-player compatible game but that is beyond the scope of the script that we have been using. Using Lua, we learned a lot about Object Oriented approach, configuration with other platforms and above all the concept of Graphics as used in Computers. Our course curriculum helped us explore more about the project that we were dwelling into and hence give us a wider perspective of how graphics and simple coding gives us a user experience so soothing and something to cherish over and over.