Intro

A nice intro of the workshop exists in docs

Resources

- Love Wiki
- Love Tutorials
- Lua Manual and Wiki

Overview

The demos folder contains fully working source code for each demo, it can be used as a reference to get your program right, but try and figure out the solution on your own

The templates folder contains some templates for you to use when building pong as well as the classic.lua file

Donwloading Lua and LOVE2D

LOVE2D can be installed from its websites homepage for Windows, Mac, and Linux. (For a treat input the Konami code on the webpage)

Love projects can ran via a .love file(we will cover this later) or by putting a main.lua in a folder and calling that folder with the love command: `love example-folder`.

Lua can be acqured via its download page or via a simple `sudo apt install lua` if on linux. Lua can be ran by `lua` which will bring up the interpreter or you can invoke a file like `lua example.lua` to run that code.

Lua Basics

Lua is similar to python in that it is dynamically typed and an interpreted langauge. That means there is no main function we must write and any functions need to be called manually if not called by something else.

```
-- This file exists in lua_basics
-- comment
-- functions
function example()
      -- variables
      name = "World!"
      -- unless the keyword local is used variables
      -- are considered global values
```

```
            local a = 5
        print("Hello,", name)
    end

    example()
```

Now run it with `lua hello-world.lua`

LOVE2D Basics

- In video game development we generally are having our code called by the engine, in this case LOVE2D is designed to called specific functions when certain actions happen. We will leverage these callback functions to run our games and do a variety of actions.
- `love.load` (only called once)
  - Used to setup variables, settings, and load resources
  - saves resources
- `love.update`
  - Constantly called and is where most math is done. This is where the concept of FPS comes into play. We will create a dt, delta time that is cacluated from the previous
- `love.draw`
  - where all the drawing happens, none of the love.graphics.draws will happen if not done inside of this functions
    - `love.graphics`
      - Has multiple methods that can draw and print different shapes, images, and workloads
      - **CHALLENGE**: Can you print hello world? `love.graphics.print` - demo1
      - `love.graphics.rectangle`
      - and more!
- `love.mousepressed`, `love.mousereleased`, `love.keypressed`, `love.keyreleased`
  - mouse/key press and release
- The chain: love.load(once) -> love.update -> love.draw -> love.update -> love.draw -> love.update, etc.
- `love.focus` - when user clicks off Love window
- `love.quit` - when user clicks the close button
- `love.resize(w, h)` - called when user resizes the window

We can run these files with `love <folder_name>`

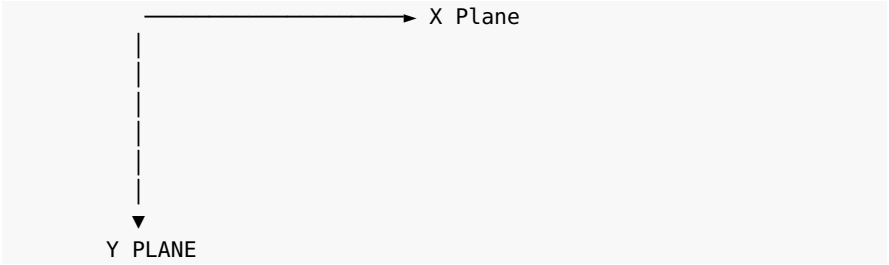
Drawing Shapes

- `love.graphics.rectangle(mode, x, y, width, height)`
  - DrawMode (mode0) will indicate whether to fill in the shape or just outline the shape
  - x, y gives the x and y offsets
  - width and height give the width and height of the shape we are drawing

- **CHALLENGE**: Lets draw a rectangle! demo2
- Lets use `love.load` to store our X and Y demo3

Okay how do we move stuff

Basic 2D game development coordinate field



- `love.update` is where we should be making changes to the coordinate field
- Okay well that makes things move really fast!
  - Things moving at different speeds because they process faster on different computers computers. Therefore you need some kind of factor to make it so that things will update at the same time. This is commonly called delta 20:35 which is the amount of time that happened between updates. If we multiply it by our "speed" then we essentially create a factor that is different on every computer That makes the overall speed move the same!
  - Faster computers will have a smaller dt therefore they will move less pixels b/w each update and slower computers will have a larger dt and will move more pixels b/w each update.
- **CHALLENGE**: `love.update(dt)`, where dt is the amount of time that is passed between each update. demo4 (uncomment the block with dt)

Moving things ourselves

- How do we move things ourselves?
  - By tracking key presses of course!
  - **CHALLENGE**: Lets just track the right key, can you do all of them?
  - `love.keyboard.isDown("key")` demo5 (all keys in the comments)
- Okay we have the basic loop of the love game engine and surpisingly this is the basic loop of most game engines!
  - We normally always have an update function that it is constantly running and does our frame by frame updates
  - we have a draw function that may be called multiple times to render more assets
  - and we have a load function that can setup global variables and objects

More Advanced Lua Concepts

Tables and Loops

- Lets look at some more lua concepts for a bit
  - `tables` and `for` loop
    - Tables = `lists` and `dictionaries` ! (they are multipurpose and a bit strange)
    - **CHALLENGE**: Make a table loop through it and print those elements tables_and_loops
  - We can access table elements in two ways obj.x or obj[“x”]
- cool who cares?
  - Well because tables can be dicts we can create arbitrary objects out of tables, like a circle object demo6
    - Tables can also hold functions as sub elements

Quick Game Dev Challenge

- **CHALLENGE**: Can you make circles appear whenever someone presses a space?

Using Multiple Files and Libraries

- How can we split development across multiple files?
  - `require("example")`
  - Similarly you can include a file as a library do `example = require "example"`
- Lua already has some builtin libaries like the math library, but others must have their code moved into your code base and referenced there.

Classes in Lua and Structured Video game programming

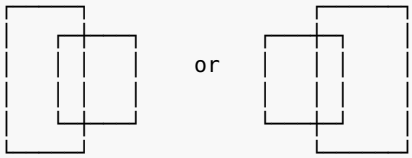
- Cool, when can we make games?? Soon! There is another important game development concept we must cover first!
  - Object Oriented Programming! Everyone’s favorite topic, in this case it is actually useful in game-development.
  - OOP at a glance:
    - Inheritance is extremely useful when we are defining two things that are different but share some characteristics. I.e. all objects will have an X and Y position, but a player might have an inventory and a monster might have items to drop. A wizard would have a mana bar, but a warrior wouldn’t, things like that.
    - Data Hiding: is useful so that people can’t cheat by modifying traits of objects they shouldn’t have access to.
    - Polymorphism: we don’t see too much of.
    - Encapsulation: allows us to define methods per function that don’t have to worry about other contexts.
  - OOP also reduces code reuse for menial task like creating classes and objects, which when you make a game you will do **A-LOT** of, being able to replicate this easily will reduce workloads and time spent writing code.

        - OOP also hopefully provides us one place to edit something that can impact all of our objects reducing the amount of edits and errors we will make
    - In our case we can utilize an external library to mimic the creation of objects in Lua and with that will come a-lot of niceties. I have some template code already classic
    - A great separation we have already seen is that circles and rectangles both have x and y coordiantes, but rectangles have height and width, whereas circles have radiuses
    - Lets make that example
- Some new things
    - Using `:` allows lua to pass the Object Object that is calling the function to the function as the `self` variable this is extremely similar to python's self. This can only be used with an instance of that class and static methods should still be called with a . and passing the object itself. For example when calling a super function
    - What is this `super`??
        - Super is a reference to the parent Object the one that we extended from so in this case super will call the Shapes new
- At all costs we should try our best to reduce how much abstraction we are doing. That will only make things more complicated and less performant so just keep that in mind
- demo7 has examples of a class structure no need to try and recreate it, we will be doing that soon enough

Lets get back to video games

- Detecting Collisions:
    - This is where the idea of a hitbox will come into play and honestly this is a subjective implementation it is all about what you want to define as a collision, where should the hitboxes interact. Thankfully this is much easier with objects that we have drawn ourselves and are just squares/rectangles
    - The way to think about it in this case is what are the boolean statements that combined would say the ball is colliding with a paddle?

Below are both valid Collisions



Lets Make Pong

- We can write to the screen to keep track of a score
- We can draw paddles and the ball to the screen and we can control

those paddles using the keyboard
- That is all of the makings of a simple pong game.

Extra Challenges

- How about an AI to play against?
- How can we handle a window resize (hint: love.resize(w,h))?
- How about a menu to enable multiplayer mode?
- Can we add music?

Packaging the game

LOVE2D's wiki has a guide on packaging games here, but here is a brief overview:

LOVE2D games are actually just zipped up files from the code base(zipped up into a file called `<game_name>.love`).

Linux:

```
love game.love
```

If using an appimage you can extract the love binary and package it with the .love file to create a standalone binary.

Windows: we can zip the love.exe and .love file together to make full standalone binary

Caveats to this workshop

- We talked very briefly about OOP and that is great, but I am here to tell you that it is not perfect!
- OOP is great when we have a small game that is relatively simple, but as soon as it becomes bigger OOP begins to hinder the performance. And will most definitely lead to scope creep where objects inherit way more than they actually need
- What a-lot of developers have switched to instead is Data Oriented Design or Entity Component System. Essentially you design components and methods around that data that matters and include those components into the entities that need them only
- OOP is how a human thinks and DOD is how the computer thinks