Introduction to Game Development

**Lecture 1: Pong**

* What is Lua?
  + Flexible, lightweight scripting language focused around tables (dictionary in Python)
  + Intended for embedded use in larger applications
  + Similar to JavaScript
  + Excellent for storing data as well as code (data-driven design)
* What is Love2D?
  + Fast 2D game dev framework written in C++
  + Uses Lau as script language
  + Contains modules for graphics, keyboard input, math, audio, windowing and physics
* Lecture Scope:
  + Draw shapes to the screen (paddles and Ball)
  + Control 2D position of paddles based on input
  + Collision detection between paddles and ball to deflect ball back toward opponent
  + Collision detection between ball and map boundaries to keep ball within vertical bounds and to detect score (outside horizontal bounds)
  + Sound effects when ball hits paddles/walls or when a point in scored for flavor
  + Scorekeeping to determine winner
* Important Functions:
  + love.load()
    - used for initializing our game state at the very beginning of program execution
  + love.update(dt)
    - called each frame by Love; dt will be the elapsed time in seconds since the last frame, and we can use this to scale any changes in our game for even behavior across frame rates
  + love.draw()
    - called each frame by Love after update for drawing things to the screen once they’ve changed
  + love.graphics.printf(text, x, y, [width], [align])
    - versatile print function that can align text left, right or center on the screen
  + love.window.setMode(width, height, params)
    - used to initialize the windows dimensions and to set parameters like (vertical sync), whether we’re fullscreen or not and whether the window is resizable after startup.
  + Love.graphics.setDefaultFilter(min, mag)
    - Sets the texture scaling filter when minimizing and magnifying textures and fonts; default is bilinear, which causes blurriness, and for our use cases we will typically want nearest-neighbor filtering, which results in perfect pixel upscaling and downscaling, simulating a retro feel
  + Love.keypresses(key)
    - a LOVE2D callback function that executes whenever we press a key, assuming we’ve implemented this in our main.lua, in the same vein as love.load(0), love.update(dt) and love.draw()
  + love.event.quit()
    - simple function that terminates the application
  + love.graphics.newFont(path, size)
    - loads a font file into memory at a specific path, setting it to a specific size and storing it in an object we can use to globally change the currently active font that Love2D is using to render text
  + love.graphics.setFont(font)
    - sets Love2D’s currently active font to a passed-in font object that we can create using love.graphics.newFont
  + love.graphics.clear(r, g, b, a)
    - wipes the entire screen with a color defined by an RGBA set, each component of which being from 0-255
  + love.graphics.rectangle(mode, x, y, width, height)
    - draws a rectangle onto the screen using whichever our active color is
    - mode can be set to “fill” or “line”, which result in a filled outlined rectangle, respectively, and the other four parameters are its position and dimensions.
  + Love.keyboard.isDown(key)
    - returns true or false depending on whether the specified key is currently held down; differs from love.keypressed(key) in that this can be called arbitrarily and will continuously return true if the key is pressed down, where love.keypressed(key) will only fire its code once every time the key is initially pressed down
  + math.random(num)
    - “seeds” the random number generator used by Lua (math.random) with some value such that its randomness is dependent on the supplied value, allowing us to pass in different numbers each playthrough to guarantee non-consistency across different program executions
  + os.time()
    - lua function that returns, in seconds, the time since 00:00:00 UTC, January 1, 1970 also known as Unix epoch time
  + math.random(min, max)
    - returns a random number, dependent on the seeded random number generator, between min and max, inclusive.
  + math.min(num1, num2)
    - returns the lesser of the two numbers passed in
  + math.max(num1, num2)
    - returns the greater of the two numbers passed in
* What is a class? p.1
  + Blueprints for creating bundles of data and code that are related
  + A “car” class can have attributes that describe its brand, model, color, miles and anything else descriptive; these are also known as “fields”
  + A “car” class can also have “methods” that define its behavior, such as “accelerate”, “turn”, “honk” and more, which take the form of functions
  + Objects are instantiated from these class blueprints, and its these concrete objects that are the physical “cars” you see on the road, as opposed to the blueprints that may exist in the factory.
  + Our Paddles and Ball are perfect simple use cases for taking some of our code and bundling it together into classes and objects.
* More important functions:
  + love.window.setTitle(title)
    - simply sets the title of our application window, adding a slight level of polish
  + love.timer.getFPS()
    - returns the current FPS of our application, making it easy to monitor when printed
* AABB Collision Detection p.1
  + Relies on all colliding entities to have “alix-aligned bounding boxes”, which simply means their collision boxes contain no rotation in our world space, which allows us to use a simple math formula to test for collision:
    - If rect1.x is not > rect2.x + rext2.width and

rect1.x + rext1.width is not < rect2.x and

rect1.y is not > rect2.y + rect2.height and

rect1.y + rect1.height is not < rect2.y:

collision is true

else

collision is false

* State Machine: the overall conceptual look at what your different states are and their transitions
* More important functions:
  + love.audio.newSource(path, [type])
    - Creates a LOVE2D audio object that we can play back at any point in our program. Can also be given a “type” of “stream” or “static”; streamed assets will be streamed from disk as needed, whereas static assets will be preserved in memory. For larger sounds effects and music tracks, streaming is more memory-effective; in our example, audio assets are static, since they’re so small that they won’t take much memory at all (TRY USING bfxr: bfxr.net)
  + love.resize(width, height)
    - Called by LOVE every time we resize the application; logic should go in here if anything in the game (like a UI) is sized dynamically based on the window size. push:resize() needs to be called here for our use case so that it can dynamically rescale its internal canvas to fit our new window dimensions

**Lecture 2: Flappy Bird**

* <https://howtomakeanrpg.com/>
* <https://hameprogrammingpatterns.com/>
* Important functions:
  + love.graphics.newImage(path)
    - loads an image from a graphics file (JPEG, PNG, GIF, etc.), storing it in an object we can draw to the screen
  + love.graphics.draw(texture, x, y)
    - texture is the image to be loaded; x and y are the positions to draw the image on
  + table.insert(table, object)
    - table is the table name to add the object into
  + iterate through a table
    - for k, pipe in pairs(pipes) do

pipe:update(dt)

if pipe.x < -pipe.width then

table.remove(pipes, k)

end

end

* + love.mousepressed(x, y, button)
    - callback fired by LOVE2D every time a mouse button is pressed; also gives us the (X,Y) of where the mouse cursor was at the time of the button press

**Lecture 3: Breakout**

* Sprite Sheets: a bitmap image file that contains several smaller graphics in a titled grid arrangement. Basically one file for all images to be used in a game.
* Important functions:
  + love.graphics.newQuad(x, y, width, height, dimensions):
    - Specify rectangle boundaries for our Quad and pass in the dimensions (returned via image: getDimensions on whichever texture we want to make a quad for).
  + love.graphics.draw(texture, quad, x, y):
    - Variant of love.graphics.draw, which we’ve seen, but this time we can pass in a Quad to draw just the specific part of the texture we want, not the entire thing
* Paddle collision:
  + Take the diff between the balls x and paddles center, which is paddle.x + paddle.width / 2 – ball.x; use this to scale the balls dx in the negative direction
  + Perform the operation on either side of the paddle based on paddles dx; if on the right side the differential will be negative, so we need to call math.abs to make it positive, then scale it by a positive amount so dc becomes positive
* Brick collision:
  + If left edge of ball is outside brick and dx is positive:

trigger left-side collision

elseif right edge of ball is outside brick and dx is negative:

trigger right-side collision

elseif top edge of ball is outside brick:

trigger top-side collision

else

trigger bottom-side collision

* Better way to deal with collisions:
  + <https://github.com/noooway/love2d_arkanoid_tutorial>
  + <https://github.com/noooway/love2d_arkanoid_tutorial/wiki/Resolving-Collisions>
* More important functions:
  + love.graphics.newParticleSystem(texture, particles)
    - Takes in a particle texture and maximum number of particles we can emit and creates a particle system we can emit from, update and render