CS50's
Introduction to Game Development

OpenCourseWare

Colton Ogden (https://www.linkedin.com/in/colton-ogden-0514029b/)
cogden@cs50.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/) (https://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan) (https://www.tiktok.com/@davidjmalan) (https://davidjmalan.t.me/) (https://twitter.com/davidjmalan)

Lecture 3: Match 3

Today's Topics

- Anonymous Functions
  - We'll see how we can define functions that operate as data types!
- Tweening
  - Tweening will allow us to interpolate a value within a range over time.
- Timers
  - We will learn how to make things happen within a range of time and/or after a certain amount of time has passed.
- Solving Matches
  - This is the heart of Match 3; we'll discuss how we can actually solve matches in order to progress the game.
- Procedural Grids
  - We'll dicuss how to randomly generate levels using procedural grids for our Match 3

game.

- Sprite Art and Palettes
    - This is a fundamental part of 2D game development. We'll discuss how to create sprites and color palettes for our game!

Downloading demo code

- github.com/games50/match3 (https://github.com/games50/match3)

timer0 ("The Simple Way")

- As usual, be sure to have downloaded the code for this week in order to follow along.
- timer0 renders a blank screen with the following text drawn in the center: "Timer: `x` seconds", where `x` is an integer counting up from 0 once per second.

Important Code

- As usual, in `main.lua` we require the `push` library and set up our screen dimension variables.
- `love.load()` initializes two variables to `0` (a counter `currentSecond` and a timer `secondTimer`) and sets up our screen dimensions.
- `love.resize()` allows us to resize our screen.
- `love.keypressed(key)` listens for keyboard input that quits the game when the `escape` key is pressed.
- `love.update()` increments our counter using our timer:

```
function love.update(dt)
    secondTimer = secondTimer + dt

    if secondTimer > 1 then
        currentSecond = currentSecond + 1
        secondTimer = secondTimer % 1
    end
end
```

- Finally, `love.draw()` displays our text and counter to the screen.
- This implementation is not ideal.. Can you think of why?

timer1 ("Also the Ugly Way")

- timer1 renders a screen with the text "Timer: `x` seconds (every `y`)" displayed in five rows, where `x` is an integer counting up from 0, and `y` is an integer representing how many seconds have passed between each increment of `x`.

Important Code

- Take a look again at `main.lua`. This should make the problem with our implementation a little clearer. This program simulates a situation in which we might need several sprites to perform some action at different times. With the way we've chosen to approach this problem, you'll notice that our code begins to get messier and messier as we introduce more and more counter and timer variables:

```
function love.update(dt)
    secondTimer = secondTimer + dt

    if secondTimer > 1 then
        currentSecond = currentSecond + 1
        secondTimer = secondTimer % 1
    end

    secondTimer2 = secondTimer2 + dt

    if secondTimer2 > 2 then
        currentSecond2 = currentSecond2 + 1
        secondTimer2 = secondTimer2 % 2
    end

    secondTimer3 = secondTimer3 + dt

    if secondTimer3 > 4 then
        currentSecond3 = currentSecond3 + 1
        secondTimer3 = secondTimer3 % 4
    end

    ...
end
```

- Surely there is a better way to do this than initializing all these variables in `love.load()` and cramming our `love.update()` with separate logic for each of them?

timer2 ("The Clean Way")

- timer2 behaves exactly like timer1 but is written with much better design.

Important Functions

- You'll notice at the top of `main.lua` that we are borrowing a `Timer` class from the `Knife` library, which we've now added to our project directory:

```
Timer = require 'knife.timer'
```

- This will simplify our code greatly, but before we dive into the rest of it, take a moment to familiarize yourself with the following functions:
  - `Timer.every(interval, callback)`
    - Calls `callback`, which is a function, every `interval`, where `interval` is measured in seconds; this happens indefinitely.
  - `Timer.after(interval, callback)`
    - Calls `callback` after `interval`, but only does this one time.
  - For more documentation, refer to the below links:
    - [github.com/airstruck/knife/blob/master/readme/timer.md (https://github.com/airstruck/knife/blob/master/readme/timer.md)](https://github.com/airstruck/knife/blob/master/readme/timer.md)
    - [github.com/airstruck/knife (https://github.com/airstruck/knife/)](https://github.com/airstruck/knife/)

Important Code

- You'll notice in `love.load()` that we now store all our counters and intervals using two tables, which allows us to simply loop over our `intervals` table and use `Timer.every` on each interval, passing in our own custom function as the `callback`. Our anonymous function takes care of incrementing each counter per each interval:

```
function love.load()

    intervals = {1, 2, 4, 3, 2, 8}
    counters = {0, 0, 0, 0, 0, 0}

    for i = 1, 6 do
        Timer.every(intervals[i], function()
            counters[i] = counters[i] + 1
        end)
    end
```

```
        ...

    end

    ...
```

- Now, our previously cluttered `love.update()` can simply defer all timer related logic to the `Timer` class.

```
function love.update(dt)
    Timer.update(dt)
end
```

- This design is incredibly scalable. Now, if we wanted to add in additional counters and timers, we need only make a handful of tweaks to our existing tables and loops, whereas previously we would've needed to introduce additional overhead in the form of additional variables and loops.

tween0 ("The Simple Way")

- If unfamiliar with the term, "tweening" (short for "in-betweening") refers to "the process of generating intermediate frames between two images to give the appearance that the first image evolves smoothly into the second image" (definition taken from webopedia.com/TERM/T/tweening.html (https://www.webopedia.com/TERM/T/tweening.html)).
  - The goal, of course, is to create the illusion of motion.
- With that in mind, tween0 renders a screen in which our fifty-bird sprite moves across the screen from left to right, with a timer being displayed in the top-left corner.

Important Code

- Take special note of the global `MOVE_DURATION` constant we've defined atop `main.lua`. We use it in `love.update(dt)` to ensure that our sprite takes approximately that amount of time (in seconds) to move across the screen each time the program is run.
- `love.load()` creates our bird sprite (note than we've added its image file to our project directory), assigns it initial coordinate values, and initializes an `endX` variable to hold the value of the `x` coordinate we want our sprite to have at the end of its movement.
- Notice now our use of `math.min` in `love.update(dt)`:

```
function love.update(dt)
```

```
        if timer < MOVE_DURATION then
            timer = timer + dt
            flappyX = math.min(endX, endX * (timer / MOVE_DURATION))
        end
    end
```

- By using `math.min` here, we ensure that we don't go past our desired `endX`. Our `timer` variable increases by `dt` each frame, so `timer / MOVE_DURATION` is the ratio by which we multiply our `endX` each turn to make it seem as if we're moving towards the right during the desired `MOVE_DURATION`.
- This program works, but again, it is not the optimal implementation. Take a moment to brainstorm for a potentially better approach…

tween1 ("A Better Way")

- tween1 operates under the same premise as tween0, but instead of moving one sprite across the screen, it moves 1000 sprites across the screen, each at different rates, albeit with the same maximum timer.

Important Code

- `love.load()` now sets up the sprites on the screen using a `birds` table and looping through it. Also sets up an individual `rate` and `y` position for each sprite:

```
function love.load()
    flappySprite = love.graphics.newImage('flappy.png')

    birds = {}

    for i = 1, 100 do
        table.insert(birds, {
            x = 0,
            y = math.random(VIRTUAL_HEIGHT - 24),
            rate = math.random() + math.random(TIMER_MAX - 1)
        })
    end

    ...

end
```

- Each sprite's `rate` is calculated randomly. Recall that `math.random()` by itself will generate a random float between 0 and 1, so we add that to `math.random(TIMER_MAX - 1)` to get a number between 0 and 10, floating-point.

- `love.update(dt)` loops over `birds` and moves each sprite across the screen using its individual `rate`:

```
function love.update(dt)
    if timer < TIMER_MAX then
        timer = timer + dt
        for k, bird in pairs(birds) do
            bird.x = math.min(endX, endX * (timer / bird.rate))
        end
    end
end
```

- Again, this is a more scalable design, which is always better

tween2 (“The `Timer.tween` way”)

- tween2 behaves similarly to tween1 but with the added component of tweening the opacity of the sprites.

Important Functions

- `Timer.tween(duration, definition)`
  - Interpolates values specified in the `definition` table over some length of time (`duration`), where the values in `definition` are the final values of the transformation.

Important Code

- Not much has changed in `main.lua`, but we’ve introduced an `opacity` value in `love.load()`, which, along with the `endX` position, gets passed into `Timer.tween`.

```
function love.load()
  ...

  for k, bird in pairs(birds) do
      Timer.tween(bird.rate, {
          [bird] = { x = endX, opacity = 255 }
      })
  end
```

```
    ...
end
```

- This again allows us to defer our `love.update()` function to the `Timer` class:

```
function love.update(dt)
    Timer.update(dt)
end
```

chain0 ("The Simple (and Hard... and Ugly) Way")

- chain0 moves a fifty-bird sprite along the perimeter of the screen in clockwise fashion.

Important Code

- The "chain" behavior in `main.lua` comes from knowing when the sprite has finished one animation (e.g., moving from top-left to top-right) and must begin the next (e.g., moving from top-right to bottom-right)
- A naive implementation of this behavior might be to keep track of our sprite's progress using a series of `if` statements along the lines of:

```
if sprite is at top-left then
    move to top-right
elseif sprite is at top-right then
    move to bottom-right
elseif sprite is at bottom-right then
    move to bottom-left
elseif sprite is at bottom-left then
    move to top-left
end
```

  Although this is an intuitive and easy-to-understand way of solving the problem, you can imagine it would quickly become very messy as the path of the sprite increases in complexity.
- To avoid littering our code with `if` statements, we create a `destinations` table in `love.load()` that holds the position of each corner of the screen (i.e., our sprite's destinations):

```
destinations = {
    [1] = {x = VIRTUAL_WIDTH - flappySprite:getWidth(), y = 0},
    [2] = {x = VIRTUAL_WIDTH - flappySprite:getWidth(), y = VIRTUAL_HEIGHT -
    [3] = {x = 0, y = VIRTUAL_HEIGHT - flappySprite:getHeight()},
```

```
        [4] = {x = 0, y = 0}
    }
```

- We attribute a `reached` flag to each of the destinations that can be toggled from `false` to `true` as fifty-bird reaches each corner:

```
    for k, destination in pairs(destinations) do
        destination.reached = false
    end
```

- What this allows us to do is check in `love.update()` for each destination's `reached` flag to be toggled, and move fifty-bird accordingly, which can be done in a loop:

```
function love.update(dt)
  timer = math.min(MOVEMENT_TIME, timer + dt)

  for i, destination in ipairs(destinations) do
      if not destination.reached then
          flappyX, flappyY =
              baseX + (destination.x - baseX) * timer / MOVEMENT_TIME,
              baseY + (destination.y - baseY) * timer / MOVEMENT_TIME

          if timer == MOVEMENT_TIME then
              destination.reached = true
              baseX, baseY = destination.x, destination.y
              timer = 0
          end

          break
      end
  end
end
```

- Although this solution is cleaner than the naive implementation, it still requires us to manually handle all the interpolation from one destination to the next. We can arrive at an even better solution when we make use of the `Timer` class.

chain1 ("The Better Way")

- chain1 behaves the same way as chain0 but is written with better design.

Important Functions

- `Timer:finish(callback)`

- A function we can call after any `Timer` function (`tween`, `every`, `after`, etc.), which calls `callback` once that function has completed. Useful for chaining any of the aforementioned function types together.

Important Code

- By making use of the `Timer` class's methods, we can defer our `love.update()` function to `Timer.update()`. As a result, most of the logic in the code happens in `love.load()`.
- We can use `Timer.tween` to take care of the sprite's movement from the first destination to the second, and then apply `Timer.finish` to that function call. Within `Timer.finish`, we can make a subsequent call to `Timer.tween` to take care of the next animation. This is analogous to telling the computer: "Move the sprite from point 1 to point 2. When you're finished, move it from point 2 to point 3 (and so on)".

```
function love.load()
    flappySprite = love.graphics.newImage('flappy.png')
    flappy = {x = 0, y = 0}

    Timer.tween(MOVEMENT_TIME, {
        [flappy] = {x = VIRTUAL_WIDTH - flappySprite:getWidth(), y = 0}
    })
    :finish(function()
        Timer.tween(MOVEMENT_TIME, {
            [flappy] = {x = VIRTUAL_WIDTH - flappySprite:getWidth(), y = VIR
        })
        :finish(function()
            Timer.tween(MOVEMENT_TIME, {
                [flappy] = {x = 0, y = VIRTUAL_HEIGHT - flappySprite:getHeig
            })
            :finish(function()
                Timer.tween(MOVEMENT_TIME, {
                    [flappy] = {x = 0, y = 0}
                })
            end)
        end)
    end)

    ...
end
```

- The downside of this approach is that, similarly to the issue with our endless `if` statements example, it suffers from a certain lack of scalability, since we need to make an additional nested call each time we want to include a new chain. This is sometimes referred to in programming circles as "Callback Hell". Thankfully, the `Timer` class provides some helpful workarounds to this problem, which we may revisit later.

  - github.com/airstruck/knife/blob/master/readme/chain.md (https://github.com/airstruck/knife/blob/master/readme/chain.md)

- For now, however, merely note that it's favorable to trade some scalability in exchange for not reinventing the wheel.

swap0 ("Just a Board")

- swap0 creates our Match 3 board and renders it to the screen.

Important Code

- Like last week, you'll notice we again make use of a separate file, `Util.lua`, to store our own helper functions for generating Quads.
- In `love.load()` we're generating our tile Quads (by calling our helper method, `GenerateQuads()`, from `Util.lua`) and setting up the board (by calling a helper function, `generateBoard()`, from within `main.lua`):

```
function love.load()
  tileSprite = love.graphics.newImage('match3.png')
  tileQuads = GenerateQuads(tileSprite, 32, 32)

  board = generateBoard()

  love.graphics.setDefaultFilter('nearest', 'nearest')

  push:setupScreen(VIRTUAL_WIDTH, VIRTUAL_HEIGHT, WINDOW_WIDTH, WINDOW_HEIGHT,
      fullscreen = false,
      vsync = true,
      resizable = true
  })
end
```

- Unlike the bricks in breakout, the tiles in Match 3 will always remain in a full grid, which we implement as an 8-by-8 2D array. Within the 2D array, we represent each tile as a table containing `x` and `y` coordinates as well as a Quad:

```
function generateBoard()
  local tiles = {}

  for y = 1, 8 do
```

```
        table.insert(tiles, {})

        for x = 1, 8 do
            table.insert(tiles[y], {
                x = (x - 1) * 32,
                y = (y - 1) * 32,
                tile = math.random(#tileQuads)
            })
        end
    end

    return tiles
end
```

- `love.draw()` then calls another helper function, `drawBoard()`, to actually render the board by looping through the 2D array and drawing each individual Quad at its `x,y` coordinates.

swap1 ("The Static Swap")

- swap1 allows the user to swap two tiles on the board. The implementation uses keyboard behavior (move among tiles using the arrow keys, select tiles with the "enter" key) and does not enforce any rules for which tiles can be swapped.

Important Code

- `love.load()` looks very similar to the swap0 version, but we've added a few new variables to keep track of the highlighting behavior that will be useful for the user to see which tile they're currently selecting:

```
highlightedTile = false
highlightedX, highlightedY = 1, 1
selectedTile = board[1][1]
```

- `love.keypressed(key)` contains several new additions, as might be expected. Here we've begun to monitor whether the user has pressed any arrow keys, and if so, we move them around our 2D tile array accordingly:

```
local x, y = selectedTile.gridX, selectedTile.gridY

if key == 'up' then
    if y > 1 then
        selectedTile = board[y - 1][x]
```

```
    end
elseif key == 'down' then
    if y < 8 then
        selectedTile = board[y + 1][x]
    end
elseif key == 'left' then
    if x > 1 then
        selectedTile = board[y][x - 1]
    end
elseif key == 'right' then
    if x < 8 then
        selectedTile = board[y][x + 1]
    end
end
```

Recall that the first table index in Lua is 1, not 0, and also recall that the way to index into a 2D array (i.e., a table of tables) is by specifying `table[row][column]`, so `table[1][1]` would be the top-left element in the 2D array, and `table[2][3]` would be the third element in the second row.

- We also monitor whether the user has pressed `enter` (or `return`) to select a tile. Once two tiles have been selected, we swap them by swapping their place in our 2D array as well as swapping their coordinates on the grid:

```
if key == 'enter' or key == 'return' then
    if not highlightedTile then
        highlightedTile = true
        highlightedX, highlightedY = selectedTile.gridX, selectedTile.gridY
    else
        local tile1 = selectedTile
        local tile2 = board[highlightedY][highlightedX]

        local tempX, tempY = tile2.x, tile2.y
        local tempgridX, tempgridY = tile2.gridX, tile2.gridY
        local tempTile = tile1

        board[tile1.gridY][tile1.gridX] = tile2
        board[tile2.gridY][tile2.gridX] = tempTile

        tile2.x, tile2.y = tile1.x, tile1.y
        tile2.gridX, tile2.gridY = tile1.gridX, tile1.gridY
        tile1.x, tile1.y = tempX, tempY
        tile1.gridX, tile1.gridY = tempgridX, tempgridY

        highlightedTile = false
```

```
            selectedTile = tile2
        end
    end
```

- Finally, in addition to handling the rendering of the board as before, `drawBoard()` now contains some additional logic in order to display the highlighting of selected tiles.

swap2 ("The Tween Swap")

- swap2 behaves the same way as swap1, with the graphical difference of tweening during a swap.

Important Code

- Everything should look essentially the same except for a few lines in `love.keypressed(key)` starting from line 99, where we tween the tile coordinates instead of instantly swapping them:

```
Timer.tween(0.2, {
    [tile2] = {x = tile1.x, y = tile1.y},
    [tile1] = {x = tempX, y = tempY}
})
```

Calculating Matches

- So, how exactly are we calculating matches in our final version of Match 3? If curious, you can find the implementation in `match-3/src/Board.lua`, in the `Board:calculateMatches()` method on line 50.
- The algorithm itself is as follows:
  - Starting from the top-left corner of the board (i.e., the first tile), set `colorToMatch` to the color of the current tile.
  - Then, check the adjacent tile on the right (i.e., the second tile). If the second tile is of the same color as `colorToMatch`, increment `matchNum` (a counter). If not, set `colorToMatch` to the color of the second tile and reset `matchNum` to 1.
  - Repeat this process for the subsequent tiles in that row. If `matchNum` reaches 3, we add that group of tiles to a `matches` table as soon as we reach an adjacent tile of a different color, or the end of the row.
  - After all rows have been checked for matches, this process is repeated for each column, going from top to bottom.

- The result of our algorithm is a table containing each group of matches, both horizontal and vertical, with each group being stored as its own table.

Removing Matches

- Now the question is how we might remove our matching tiles, once we have them all in a table. You might think we'd have to do some tricky coding with their coordinates on the grid or their position on the board, but this problem can actually be solved very straightforwardly.
- As you can see in `Board:removeMatches()` (still in `match-3/src/Board.lua`), we can simply loop over each table within our `matches` table and set the value of each tile to `nil`, after which we can also set our `matches` table to `nil`.
- Instead of breaking everything, this has the effect of erasing the tiles and the `matches` table from existence, as far as the user is concerned. The result on the screen would be the board as it was before, but with holes where previously there would've been matches.
- The next function in the file, `Board:getFallingTiles()`, takes care of shifting down the remaining tiles in each column if needed.
    - As you might imagine, the algorithm consists of starting from the bottom of each column and checking each slot in the column until a `nil` slot is found.
    - Once a `nil` slot is found, its position is marked and the next non-`nil` tile found in the column is tweened down to the marked position.
    - This process is iterated for each `nil` slot in each column.

Replacing Tiles

- After we've removed matching tiles and shifted down those left over, we need to figure out how to replace the remaining `nil` positions. This is also done in `Board:getFallingTiles()`.
- The algorithm itself is simple:
    - For each column, we count the number of `nil` spaces and generate that many new tiles.
    - Then, we can set each tile's position to be that of one of the `nil` spaces, and add it to our board array.
    - We want the animation to look like the tiles are falling into place, however, so to produce this effect, we tween each new tile from the top of the board to its final position, letting the `Timer` class take care of the actual implementation.
- One additional note is that the possibility exists that the newly-generated tiles will create

additional matches when they spawn.

- As a result, we must make sure we check for matches again after generating the new tiles.
- This is done in `match-3/src/states/PlayState.lua`, in the `PlayState:calculateMatches()` function, which uses the functions in `Board.lua` to calculate matches, remove matches, and get falling tiles, but then importantly recurses until it ensures that no new matches have been found.

Palettes

- A palette is essentially just a set of available colors.
- In our programs, we have been using DB32 - DawnBringer's 32 Color Palette (V1.0).
- The idea behind using a palette is that it allows you to "dither" (i.e., interleave two colors pixel by pixel in an attempt to approximate another color) such that your programs can look like they use many colors, when in reality you only have a small set (in this case 32) available to you.
- When done properly, the difference between a dithered image and an image that actually uses hundreds of thousands of colors can be quite minimal.

Palette Swapping

- Palette swapping is the practice of using the same sprite for two different graphics, but with a different palette for each, such that the resulting sprites are noticeably distinct from one another.
- For example, a bush and a cloud could share the same sprite, but while one would be green, the other would be white.