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 4: Super Mario Bros.

Today's Topics

- Tile Maps
  - How can we take a series of numbers (e.g., tile IDs) and turn them into a game world?
- 2D Animation
  - How can we "animate" a sprite and give its body the appearance of movement (e.g., walking?)
- Procedural Level Generation
  - How can we generate our levels randomly so that we're not limited to a specific number of different levels we've created?
- Platformer Physics
  - How can we implement basic platformer physics so that we don't have to iterate through our entire world to determine, for example, collisions?

- Basic AI
  - How can we program an adversary to attack the player on its own?
- Powerups
  - How can we create "powerups" that affect "Mario" and his abilities?

Downloading demo code

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

Tilemaps

- A tilemap can be thought of as a giant 2D array of numbers.
- It can be a little more complicated, of course, since some numbers represent tiles that are solid and others represent tiles that are not. This determines whether or not a player can collide with a certain tile, which will obviously be very relevant to mario.
- In mario, each level is comprised of many small tiles that give the appearance of some larger whole.
- As we alluded to earlier, tiles often have an ID of some kind to differentiate their appearance or behavior.

tiles0 ("Static Stiles")

- tiles0 displays a colorful, static background on the screen with some tiles in the foreground.

Important Code

- In `love.load()`, we define a `tiles` table for storing each of our tiles, which will be represented as tables themselves, and we will render them on the screen using the sprite sheet in `tiles.png`.
  - It might seem like our sprite sheet contains just one sprite (the tile that is visible in `tiles.png`), but it also contains an additional sprite of the same tile, but transparent.
  - This will allow us to differentiate our tiles in our program with IDs (`SKY` and `GROUND`), which is how we can have half the screen contain tiles, and the other half be a simple color
- The rest of `love.load()` populates the map (i.e., the screen) with a random-color

background, filling the top half of the screen with transparent tile sprites and the lower half with opaque tile sprites:

```
tilesheet = love.graphics.newImage('tiles.png')
quads = GenerateQuads(tilesheet, TILE_SIZE, TILE_SIZE)

mapWidth = 20
mapHeight = 20

backgroundR = math.random(255) / 255 / 255
backgroundG = math.random(255) / 255 / 255
backgroundB = math.random(255) / 255 / 255

for y = 1, mapHeight do
    table.insert(tiles, {})

    for x = 1, mapWidth do
        table.insert(tiles[y], {
            id = y < 5 and SKY or GROUND
        })
    end
end
```

And of course, after the above block of code, it also sets up our screen dimensions, as always.

- If the loop in `love.load()` seems odd, take a look at `love.draw()` and notice how the map is actually rendered. We are looping over the `tiles` table and drawing them on the screen, which is why their IDs from `love.load()` are relevant:

```
function love.draw()
    push:start()
        love.graphics.clear(backgroundR, backgroundG, backgroundB, 1)

        for y = 1, mapHeight do
            for x = 1, mapWidth do
                local tile = tiles[y][x]
                love.graphics.draw(tilesheet, quads[tile.id], (x - 1) * TILE_
            end
        end
    push:finish()
end
```

tiles1 (“Scrolling Tiles”)

- tiles1 looks the same as tiles0, but allows the user to shift the map from left to right using the arrow keys.

Important Functions

- `love.graphics.translate(x, y)`
  - Shifts the coordinate system by `x, y`; useful for simulating camera behavior.

Important Code

- You'll notice we've added in some logic to `love.update()`, whose purpose for now is to update a `cameraScroll` variable whenever the user presses the left or right arrow key:

```
function love.update(dt)
    if love.keyboard.isDown('left') then
        cameraScroll = cameraScroll - CAMERA_SCROLL_SPEED * dt
    elseif love.keyboard.isDown('right') then
        cameraScroll = cameraScroll + CAMERA_SCROLL_SPEED * dt
    end
end
```

- Now, in `love.draw()`, we have to translate the map in the opposite direction as `cameraScroll`, since the map needs to move to the left when the user scrolls to the right, and vice versa:

```
love.graphics.translate(-math.floor(cameraScroll), 0)
love.graphics.clear(backgroundR, backgroundG, backgroundB, 1)
```

  We make sure to round down our `cameraScroll` using `math.floor()` in order to prevent any weird blurring or other such visual effects that might result from our screen trying to handle floating-point numbers in a small, virtual canvas.

character0 ("The Stationary Hero")

- character0 behaves the same way as tiles1, but with the simple addition of introducing a stationary character sprite that stands on the tiles.

Important Code

- In `love.load()` you'll notice that we're accessing a new sprite sheet for our character sprite, and setting the `x` and `y` coordinates for our character sprite:

```
characterSheet = love.graphics.newImage('character.png')
characterQuads = GenerateQuads(characterSheet, CHARACTER_WIDTH, CHARACTER_HE

characterX = VIRTUAL_WIDTH / 2 - (CHARACTER_WIDTH / 2)
characterY = ((7 - 1) * TILE_SIZE) - CHARACTER_HEIGHT
```

- The only other change is in `love.draw()`, where we render our character to the screen (for now, we're only rendering the sprite in the first Quad):

```
love.graphics.draw(characterSheet, characterQuads[1], characterX, characterY
```

character1 ("The Moving Hero")

- character1 allows our character sprite to move from left to right on the map. However, it no longer allows us to scroll the camera.

Important Code

- The main difference here is in `love.update()`. What we've done is instead of updating `cameraScroll`, we update our character's `x` coordinate when the user presses the left or right arrow keys:

```
function love.update(dt)
    if love.keyboard.isDown('left') then
        characterX = characterX - CHARACTER_MOVE_SPEED * dt
    elseif love.keyboard.isDown('right') then
        characterX = characterX + CHARACTER_MOVE_SPEED * dt
    end
end
```

Can you think of a better solution?

character2 ("The Tracked Hero")

- character2 addresses the camera issue from character1.
- Now, the camera is fixed on our character sprite, allowing the user to move the sprite while scrolling the camera. However, there is still no animation for the sprite, and the user can still scroll off the edge of the map.

Important Code

- The main change here is achieved by simply updating `cameraScroll` in `love.update()` to track the character sprite's `x` coordinate, making sure to center it on the screen as well:

```
function love.update(dt)
    if love.keyboard.isDown('left') then
        characterX = characterX - CHARACTER_MOVE_SPEED * dt
    elseif love.keyboard.isDown('right') then
        characterX = characterX + CHARACTER_MOVE_SPEED * dt
    end

    cameraScroll = characterX - (VIRTUAL_WIDTH / 2) + (CHARACTER_WIDTH / 2)
end
```

character3 ("The Animated Hero")

- character3 behaves the same way as character2 but with the added feature of animating our sprite's movement.
    - Animations can be achieved by simply displaying a series of frames from a sprite sheet one after the other, akin to a flip book.

Important Code

- `Animation.lua` will take care of handling animations for our program.
    - `Animation:init(def)` will take in an argument which will contain information about the animation in question. In particular, it will specify the number of frames in the animation as well as the interval between each frame:

```
function Animation:init(def)
    self.frames = def.frames
    self.interval = def.interval
    self.timer = 0
    self.currentFrame = 1
end
```

    - `Animation:update(dt)` takes care of toggling among the frames at the proper intervals:

```
function Animation:update(dt)
    if #self.frames > 1 then
        self.timer = self.timer + dt
```

```
            if self.timer > self.interval then
                self.timer = self.timer % self.interval

                self.currentFrame = math.max(1, (self.currentFrame + 1) % (
            end
        end
    end
```

- `Animation:getCurrentFrame()` returns the current frame for the animation.

- Back in `main.lua`, `love.load()` is where we actually instantiate our animations (one for moving and the other for standing still) and set our sprite's direction:

```
idleAnimation = Animation {
    frames = {1},
    interval = 1
}
movingAnimation = Animation {
    frames = {10, 11},
    interval = 0.2
}

currentAnimation = idleAnimation
direction = 'right'
```

- `love.update()` takes care of toggling the moving animations when the user moves the character and setting the animation back to idle when the user is not moving the character:

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

    if love.keyboard.isDown('left') then
        characterX = characterX - CHARACTER_MOVE_SPEED * dt
        currentAnimation = movingAnimation
        direction = 'left'
    elseif love.keyboard.isDown('right') then
        characterX = characterX + CHARACTER_MOVE_SPEED * dt
        currentAnimation = movingAnimation
        direction = 'right'
    else
        currentAnimation = idleAnimation
    end

    cameraScroll = characterX - (VIRTUAL_WIDTH / 2) + (CHARACTER_WIDTH / 2)
end
```

- `love.draw()` renders everything on the screen as before.

character4 ("The Jumping Hero")

- character4 behaves similarly to character3, but with the added feature of allowing the character to jump

Important Code

- In `love.load()`, you'll notice we've added an additional animation for jumping. Another addition to the code is the introduction of a `characterDY` for jumping and applying gravity:

```
jumpAnimation = Animation {
    frames = {3},
    interval = 1
}
characterDY = 0
```

  As a side note, it's important to recognize there are two separate states when jumping: jumping and falling. In mario, for example, you can destroy a block by hitting it while you jump, which you can't do by merely falling on it. Similarly, you can defeat an enemy by falling on its head, but not by jumping towards it.

- `love.keypressed(key)` now monitors whether the user has hit the "space" key and responds by setting `characterDY` to an arbitrary jump velocity and triggering the jump animation, although it enforces that the character can only jump if they are on the ground (no double jumps mid-air, etc.):

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

    if key == 'space' and characterDY == 0 then
        characterDY = JUMP_VELOCITY
        currentAnimation = jumpAnimation
    end
end
```

- `love.update(dt)` uses `characterDY` to apply velocity to the sprite's `y` coordinate, setting `characterDY` to `0` once the sprite hits the floor (since we haven't implemented tile collision yet). This implementation is a bit hacky, but it works for now:

```
characterDY = characterDY + GRAVITY
characterY = characterY + characterDY * dt

if characterY > ((7 - 1) * TILE_SIZE) - CHARACTER_HEIGHT then
    characterY = ((7 - 1) * TILE_SIZE) - CHARACTER_HEIGHT
    characterDY = 0
end
```

Procedural Level Generation

- Platformer levels can be procedurally generated like anything else.
- These levels can be more easily generated per column rather than per row, given things like gaps, though there are multiple ways to do it.
- Most easily, tiles can foundationally be generated and act as the condition upon which GameObjects and Entities are generated.

level0 (“Flat Levels”)

- level0 builds off from character4, but it allows the user to change the tile configuration on the map floor by pressing the `r` key on the keyboard.
- Each tile configuration consists of a “topper” and a “tileset”, where the topper is a distinct color from the tileset and denotes the topmost surface of the tiles.

Important Code

- In `love.load()` you’ll notice we’re generating Quads from a sprite sheet in order to generate the tiles and toppers, selecting a random set as the starting configuration:

```
tilesheet = love.graphics.newImage('tiles.png')
quads = GenerateQuads(tilesheet, TILE_SIZE, TILE_SIZE)

topperSheet = love.graphics.newImage('tile_tops.png')
topperQuads = GenerateQuads(topperSheet, TILE_SIZE, TILE_SIZE)

tilesets = GenerateTileSets(quads, TILE_SETS_WIDE, TILE_SETS_TALL, TILE_SET_
toppersets = GenerateTileSets(topperQuads, TOPPER_SETS_WIDE, TOPPER_SETS_TAL

tileset = math.random(#tilesets)
topperset = math.random(#toppersets)
```

- In `love.keypressed(key)` we've added a check for the `r` key, which selects another random tile and topper set to display:

```
if key == 'r' then
    tileset = math.random(#tilesets)
    topperset = math.random(#toppersets)
end
```

- In `love.draw()` we're drawing the tileset and then checking the `topper` flag on each tile to determine whether or not to draw a topper on top of it. Only the tiles with a `y` of 7 will need a topper:

```
for y = 1, mapHeight do
    for x = 1, mapWidth do
        local tile = tiles[y][x]
        love.graphics.draw(tilesheet, tilesets[tileset][tile.id],
            (x - 1) * TILE_SIZE, (y - 1) * TILE_SIZE)

        if tile.topper then
            love.graphics.draw(topperSheet, toppersets[topperset][tile.id],
                (x - 1) * TILE_SIZE, (y - 1) * TILE_SIZE)
        end
    end
end
```

- `generateLevel()` is a custom function we wrote to modularize the map-generation process:

```
function generateLevel()
    local tiles = {}
    for y = 1, mapHeight do
        table.insert(tiles, {})
        for x = 1, mapWidth
            table.insert(tiles[y], {
                id = y < 7 and SKY or GROUND,
                topper = y == 7 and true or false
            })
        end
    end

    return tiles
end
```

- Now, instead of having all that logic in `love.load()`, we can simply substitute it with a function call:

```
tiles = generateLevel()
```

level1 (“Pillared levels”)

- level1 behaves the same way as level0 but allows for pillars to be drawn on the map.

Important Code

- We’ve modified our `generateLevel()` function to support this new functionality.
- Now, we first populate the map entirely with empty tiles (i.e., tiles with `id` of `SKY`):

```
for y = 1, mapHeight do
    table.insert(tiles, {})

    for x = 1, mapWidth do
        table.insert(tiles[y], {
            id = SKY,
            topper = false
        })
    end
end
```

- Then, we iterate through the map column by column, generating the ground as usual:

```
for x = 1, mapWidth do
    local spawnPillar = math.random(5) == 1

    if spawnPillar then
        for pillar = 4, 6 do
            tiles[pillar][x] = {
                id = GROUND,
                topper = pillar == 4 and true or false
            }
        end
    end

    ...

end
```

- However, we now introduce the possibility of spawning a pillar (in this case it’s a 1 in 5 chance). If a pillar is to be spawned, we must modify the tiles with `y` 4 to 6 to have `id` of `GROUND` (where the tile with `y` of 4 needs a topper, rather than the tile with `y` of 7).

```
    for x = 1, mapWidth do
        local spawnPillar = math.random(5) == 1

        ...

        for ground = 7, mapHeight do
            tiles[ground][x] = {
                id = GROUND,
                topper = (not spawnPillar and ground == 7) and true or false
            }
        end
    end

    return tiles
```

level2 (Chasmed levels)

- level2 introduces chasms to the map, with everything else working the same as before.

Important Code

- Again, the changes are in `generateLevel()`.
- We've now added a chasm spawn chance (1 in 7) similar to the pillar spawn chance.
- Since the first thing we do is populate the map with empty tiles, that means our implementation for creating chasms can be to simply skip the ground generation.
- The syntax to do this in Lua is to use the `goto` keyword to point the program to a label of your choice, in our case `continue`, skipping all the code in between:

```
    for x = 1, mapWidth do
        if math.random(7) == 1 then
            goto continue
        end

        ...

        ::continue::
    end
```

Tile Collision

- AABB can be useful for detecting entities, but we can take advantage of our static coordinate system and 2D tile array and just calculate whether the pixels in the direction we're traveling are solid, saving us computing time.
- Check out `TileMap:pointToTile(x, y)` in `mario/src/TileMap.lua`:

```
function TileMap:pointToTile(x, y)
    if x < 0 or x > self.width * TILE_SIZE or y < 0 or y > self.height * TILE
        return nil
    end

    return self.tiles[math.floor(y / TILE_SIZE) + 1][math.floor(x / TILE_SIZE
end
```

  The first part of this function merely ensures that we don't return an error if we somehow go beyond the boundaries of the map. The second part of this function actually returns the tile in our `tiles` array that corresponds with the `x` and `y` coordinates in our map so that we can check if it is solid or empty.
- In terms of performance, this is notably better than having to iterate over all tiles in our array to check AABB , with some inflexibilities (tiles can't move around, for example).
- We can improve performance even further by only checking for collisions above us when in jumping state, collisions below us when in idle, walking and falling state, and collisions to the left and right when in walking, jumping, and falling state (as opposed to checking for collisions in all directions in all states)

Entities

- Entities can contain states just like the game, with their own StateMachine; states can affect input handling (for the player) or decision-making (like the Snail).
- Some engines may adopt an Entity-Component System (or ECS), where everything is an Entity and Entities are simply containers of Components, and Components ultimately drive behavior (Unity revolves around an ECS).
- Collision can just be done entity-to-entity using AABB collision detection.
- Entities in our distro represent the living things in our world (Snail and Player), but could generally represent almost anything.

Game Objects

- Game objects are separate from the tiles in our map, for things that maybe don't align perfectly with it (maybe they have different widths/heights or their positions are offset by

a different amount than `TILE_SIZE`).

- They can be tested for collision by AABB, are often just containers of traits and functions, and can be represented via Entities (but aren't in this distro).
- Powerups in mario can be represented as GameObjects:
  - Effectively a GameObject that changes some `status` or trait of the player.
  - An invincible star may flip an `invincible` flag on the player and begin an `invincibleDuration` timer.
  - A mushroom to grow the player may trigger a `huge` flag on the player that alters their `x`, `y`, `width`, and `height` and then scales their sprite.
- We allow for the generation of game objects in `mario/src/LevelMaker.lua`. Read through this file carefully in order to get a good intuition for how you could add in your own game objects.