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 5: Legend of Zelda

Today's Topics

- Top-down Perspective
  - With Flappy Bird and Mario, we were looking at the screen from the side. With Zelda, we'll have a bird's eye view of the screen, so we'll discuss the logistics of this today.
- Infinite Dungeon Generation
  - We've discussed previously the concept of games as illusions, and we'll be seeing another example of that today in Zelda, where we'll seemingly generate an endless dungeon.
- Hitboxes/Hurtboxes
  - We'll discuss the difference between hitboxes and hurtboxes in the context of collisions. The former allows a character to inflict damage, whereas the latter allows a character to receive damage.

- Events
    - Events are a way of broadcasting some key or message that informs us when something happens, and allows us to call a function in response.
- Screen Scrolling
    - We'll take a look at how we can use tweening to give the appearance of transitioning from room to room in our dungeon.
- Data-Driven Design
    - Lastly, we'll see how we can model our game items, entities, and objects as data rather than logic in order to improve the design of our code.

Downloading demo code

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

Zelda Sprites

- We'll again need a sprite sheet in order to render our sprites to the screen. It's helpful for our sprites to be laid out in tile segments of 16x16 pixels, so that we can index into the sheet evenly in order to access particular sprites. However, we will inevitably encounter sprites that do not fit into our sprite sheet in this way, and these sprites will require slightly more complicated rendering logic.
    - For example, the doors we use in our Zelda program are bigger than 16x16 pixels, so instead of rendering a single quad from our sheet, we must instead render 4 quads in order to properly display each door. This would change how we monitor for collisions as well, since we'd have to adjust each door's hitbox to cover all 4 quads, or alternatively we'd have to monitor for collisions on each of the 4 quads individually.
    - A more complicated example would be our character sprite sheet, which has padding. Sometimes the sprites in a sheet will not be neatly divided into even segments. For instance, our character sprite in Zelda is 20x16 pixels and also has several animations of different dimensions, such as his sword-swing animation (32x32 pixels).
    - In order to render such a sprite properly to the screen, one often has to associate an `offset` with that sprite's coordinates, such that the sprite is shifted by that `offset` on the screen when being rendered.
- Fortunately, our sprite sheet for the remaining creatures in our game follows the more ideal layout of each sprite being 16x16 pixels.

    - It's quite large, however, so we've written a script that will print the index names for each sprite on our sheet. This is something that you may find helpful as you work on future projects with large sprite sheets.

Top-Down Perspective

- A map in top-down perspective is essentially a tile map like we've seen before, but with the subtle difference of looking the map from up above rather than from the side.
- Some of the main visual differences might be that there's less of a focus on screen scrolling, and more of a focus on things like shadows on the walls, corners of the screen, lighting, and camera angles (you want your world to be slightly skewed rather than completely vertical, for example).

Dungeon Generation

- In games such as Zelda, dungeons are generally fixed (i.e., they are preemptively created by the programmers in some predetermined layout), but in our version, we will be generating dungeons infinitely.
- Nevertheless, when done in the Zelda way, dungeons can be represented in a 2D array, such that some indexes are empty ("off") and others contain rooms ("on"), with each room connected to at least one other.
- The result is that when transitioning from room to room (via a door), we can display the new room by simply adding or subtracting `1` from the `x` or `y` index of the current room.
- This allows the programmer to potentially "lock" doors and hide "keys" in certain rooms in the dungeon, such that the player has to visit each room in a particular order to beat the level

World Classes

- Navigate to the `src/world/` subdirectory. This is where we've saved all files having to do with world generation.
- Here, we have three classes: `Doorway.lua`, `Dungeon.lua`, and `Room.lua`.
    - These are fairly self explanatory. We can think of a dungeon as a table of rooms (which themselves contain doorways), such that we have only one active room at a time, and whenever we transition between rooms, a new one is loaded and becomes the active room.

`Dungeon.lua`

- You'll notice, here that in our `Dungeon:init()` function, we have variables for our `player` and our `rooms` table, with `currentRoom` and `nextRoom` also initialized (as well as other relevant information):

```
function Dungeon:init(player)
    self.player = player
    self.rooms = {}
    self.currentRoom = Room(self.player)
    self.nextRoom = nil

    self.cameraX = 0
    self.cameraY = 0
    self.shifting = false

    ...

end
```

  You'll also notice some `Event` logic - we'll revisit this in more detail later.

`Room.lua`

- Similarly to how a dungeon needs to keep track of its rooms, a room needs to keep track of its components. `Room.lua` concerns itself with the `player`, the `doorways`, other `objects` and `entities` in the room such as door switches and enemies, as well as a set of `tiles` to actually display the room itself:

```
function Room:init(player)

    ...

    self.tiles = {}
    self:generateWallsAndFloors()

    self.entities = {}
    self:generateEntities()

    self.objects = {}
    self:generateObjects()

    self.doorways = {}
    table.insert(self.doorways, Doorway('top', false, self))
```

```
        table.insert(self.doorways, Doorway('bottom', false, self))
        table.insert(self.doorways, Doorway('left', false, self))
        table.insert(self.doorways, Doorway('right', false, self))

        self.player = player

        ...
    end
```

- In our program, we've basically hardcoded the doorway generation such that the doors will always be in the same position in every room, and they will always be locked until the switch is triggered. However, you can imagine that it might be nicer design to vary the doorway generation a bit, which is certainly the case in the real Zelda game.
- We also include some offset variables to help implement our infinite dungeon algorithm:

```
function Room:init(player)
  self.width = MAP_WIDTH
  self.height = MAP_HEIGHT

  ...

  self.renderOffsetX = MAP_RENDER_OFFSET_X
  self.renderOffsetY = MAP_RENDER_OFFSET_Y

  self.adjacentOffsetX = 0
  self.adjacentOffsetY = 0
end
```

Infinite Dungeon Generation

- Here's what we want to do to simulate infinitely transitioning from one room to another: load up a new room whenever the user collides with a doorway, and render it off-screen with some offset depending on which direction the user is going, such that the new room is adjacent to the current room.
- This offset will be a negative or positive `VIRTUAL_WIDTH` or `VIRTUAL_HEIGHT` depending on the doorway.
- At this point, we can tween from the coordinates of the current room to those of the new room, resetting the new room to have the coordinates of the current room (`(0, 0)`) once the tweening animation terminates.
- We can repeat this process the next time a user collides with a doorway.

`Room.lua` (continued)

- `generateWallsAndFloors()` (line 113) uses some screen coordinate logic to infer which tile sprite needs to be rendered to the screen.
    - It loops through each coordinate on the screen, and gives it an `id` which will map that coordinate to the appropriate tile sprite.

    ```
    function Room:generateWallsAndFloors()
        for y = 1, self.height do
            table.insert(self.tiles, {})

            for x = 1, self.width do
                local id = TILE_EMPTY

                ...

            end
        end
    end
    ```

    - First, we check the corners of the screen, since there is only one tile wich maps to each corner, then we check along the edges of the screen, generating a random `id` of a wall tile given the constraint that the wall tile correspond to the appropriate side of the screen.

    ```
    if x == 1 and y == 1 then
        id = TILE_TOP_LEFT_CORNER
    elseif x == 1 and y == self.height then
        id = TILE_BOTTOM_LEFT_CORNER
    elseif x == self.width and y == 1 then
        id = TILE_TOP_RIGHT_CORNER
    elseif x == self.width and y == self.height then
        id = TILE_BOTTOM_RIGHT_CORNER

    elseif x == 1 then
        id = TILE_LEFT_WALLS[math.random(#TILE_LEFT_WALLS)]
    elseif x == self.width then
        id = TILE_RIGHT_WALLS[math.random(#TILE_RIGHT_WALLS)]
    elseif y == 1 then
        id = TILE_TOP_WALLS[math.random(#TILE_TOP_WALLS)]
    elseif y == self.height then
        id = TILE_BOTTOM_WALLS[math.random(#TILE_BOTTOM_WALLS)]
    else
    ```

```
        id = TILE_FLOORS[math.random(#TILE_FLOORS)]
    end
```

- If all else fails, a floor tile `id` is randomly generated in similar fashion:

```
    table.insert(self.tiles[y], {
        id = id
    })
```

- `generateEntities()` (line 48) uses the `ENTITY_DEFS` table in `src/entity_defs.lua`, paired with a local `types` table containing the different entities in our game, to select 10 random entities and animate them on the screen.

```
function Room:generateEntities()
    local types = {'skeleton', 'slime', 'bat', 'ghost', 'spider'}

    for i = 1, 10 do
        local type = types[math.random(#types)]

        table.insert(self.entities, Entity {
            animations = ENTITY_DEFS[type].animations,
            walkSpeed = ENTITY_DEFS[type].walkSpeed or 20,

            x = math.random(MAP_RENDER_OFFSET_X + TILE_SIZE,
                VIRTUAL_WIDTH - TILE_SIZE * 2 - 16),
            y = math.random(MAP_RENDER_OFFSET_Y + TILE_SIZE,
                VIRTUAL_HEIGHT - (VIRTUAL_HEIGHT - MAP_HEIGHT * TILE_SIZE) +

            width = 16,
            height = 16,

            health = 1
        })

        self.entities[i].stateMachine = StateMachine {
            ['walk'] = function() return EntityWalkState(self.entities[i]) en
            ['idle'] = function() return EntityIdleState(self.entities[i]) en
        }

        self.entities[i]:changeState('walk')
    end
end
```

A particularly nice thing about `entity_defs.lua` is that it allows us to store information about our different entities in a file free from any programming logic, such that an Entity object can be instantiated using the information in this file, without needing a separate class for each entity. Be sure to look through it to understand how we're creating Entities for our game.

- `generateObjects()` (line 82) behaves very similarly to `generateEntities`, but using the `GAME_OBJECT_DEFS` table in `src/game_objects.lua` to generate objects (e.g., the switch) instead of entities:

```
function Room:generateObjects()
    local switch = GameObject(
        GAME_OBJECT_DEFS['switch'],
        math.random(MAP_RENDER_OFFSET_X + TILE_SIZE,
                    VIRTUAL_WIDTH - TILE_SIZE * 2 - 16),
        math.random(MAP_RENDER_OFFSET_Y + TILE_SIZE,
                    VIRTUAL_HEIGHT - (VIRTUAL_HEIGHT - MAP_HEIGHT * TILE_SIZI
    )

    switch.onCollide = function()
        if switch.state == 'unpressed' then
            switch.state = 'pressed'

            -- open every door in the room if we press the switch
            for k, doorway in pairs(self.doorways) do
                doorway.open = true
            end

            gSounds['door']:play()
        end
    end

    table.insert(self.objects, switch)
end
```

- `update(dt)` (line 149) tracks updates in the player, entities, and objects, most notably monitoring collisions and responding appropriately. For example, if the player collides with an entity, the player will take damage and go briefly invulnerable until they run out of lives. On the other hand, if an entity collides with the player's sword, that entity is marked `dead` once its `health` reaches `0`. If the player collides with an object, that object's `onCollide()` function is called to determine what happens:

```
function Room:update(dt)

    if self.adjacentOffsetX ~= 0 or self.adjacentOffsetY ~= 0 then return end

    self.player:update(dt)
```

```
    for i = #self.entities, 1, -1 do
        local entity = self.entities[i]

        if entity.health <= 0 then
            entity.dead = true
        elseif not entity.dead then
            entity:processAI({room = self}, dt)
            entity:update(dt)
        end

        if not entity.dead and self.player:collides(entity) and not self.pla
            gSounds['hit-player']:play()
            self.player:damage(1)
            self.player:goInvulnerable(1.5)

            if self.player.health == 0 then
                gStateMachine:change('game-over')
            end
        end
    end

    for k, object in pairs(self.objects) do
        object:update(dt)

        -- trigger collision callback on object
        if self.player:collides(object) then
            object:onCollide()
        end
    end
end
```

- `render()` (line 188) loops through all the components of the room and draws them on the screen. First the tiles, then the doorways, objects, entities, and finally the player. An interesting tidbit in this function is the use of stenciling, which we'll cover in more detail briefly. Essentially, the purpose of stenciling here is to aid us in the visual effect of making the player's transition through each doorway look as realistic as possible:

```
function Room:render()
    for y = 1, self.height do
        for x = 1, self.width do
            local tile = self.tiles[y][x]
            love.graphics.draw(gTextures['tiles'], gFrames['tiles'][tile.id]
                (x - 1) * TILE_SIZE + self.renderOffsetX + self.adjacentOffs
                (y - 1) * TILE_SIZE + self.renderOffsetY + self.adjacentOffs
```

```
            end
        end

        for k, doorway in pairs(self.doorways) do
            doorway:render(self.adjacentOffsetX, self.adjacentOffsetY)
        end
        for k, object in pairs(self.objects) do
            object:render(self.adjacentOffsetX, self.adjacentOffsetY)
        end
        for k, entity in pairs(self.entities) do
            if not entity.dead then entity:render(self.adjacentOffsetX, self.adj
        end

        love.graphics.stencil(function()
            love.graphics.rectangle('fill', -TILE_SIZE - 6, MAP_RENDER_OFFSET_Y -
                TILE_SIZE * 2 + 6, TILE_SIZE * 2
            love.graphics.rectangle('fill', MAP_RENDER_OFFSET_X + (MAP_WIDTH * TI
                MAP_RENDER_OFFSET_Y + (MAP_HEIGHT / 2) * TILE_SIZE - TILE_SIZE, T
            love.graphics.rectangle('fill', MAP_RENDER_OFFSET_X + (MAP_WIDTH / 2)
                -TILE_SIZE - 6, TILE_SIZE * 2, TILE_SIZE * 2 + 12
            love.graphics.rectangle('fill', MAP_RENDER_OFFSET_X + (MAP_WIDTH / 2)
                VIRTUAL_HEIGHT - TILE_SIZE - 6, TILE_SIZE * 2, TILE_SIZE * 2 + 12
        end, 'replace', 1)

        love.graphics.setStencilTest('less', 1)

        if self.player then
            self.player:render()
        end

        love.graphics.setStencilTest()

    end
```

- A stencil is an arbitrary, invisible shape that you can draw to the screen. It can decide whether anything that gets drawn on top of it, is actually rendered. In the function above, we've drawn a stencil on each of the doorways to ensure that the player does not get rendered when walking through doors. This prevents funky visual behavior from happening when the player walks through a door.
    - To see this more clearly, try commenting out the stenciling logic and notice the weird visual behavior that occurs when walking through doorways, especially the top and bottom ones.

Hitboxes and Hurtboxes

- Hitboxes and hurtboxes are two types of collision boxes for sprites that are especially relevant in fighting scenarios.
- A hitbox would, upon collision with another sprite, register a positive hit (e.g., the player deals damage to an enemy).
- Conversely, a hurtbox would register a negative hit upon collision (e.g., the player takes damage).
- In our Zelda program, you can imagine that the player's hitbox would cover their sword, whereas the hurtbox would cover their body.
- Open up `Hitbox.lua` in `src/`. You'll notice that our Hitbox class only has one method: `init`. All it does is instantiate a Hitbox object with an `(x, y)` coordinate pair, a `width` and a `height`. It might seem simple, but when you think about it, this is really all the information needed to create a hitbox!

```
Hitbox = Class{}

function Hitbox:init(x, y, width, height)
    self.x = x
    self.y = y
    self.width = width
    self.height = height
end
```

- Open up `PlayerSwingSwordState.lua` in `src/states/entity/player`. This file will implement the interactions between the player's hitbox and the rest of the world.
    - In the `init` method, we set the hitbox boundaries:

```
function PlayerSwingSwordState:init(player, dungeon)
    self.player = player
    self.dungeon = dungeon

    self.player.offsetY = 5
    self.player.offsetX = 8

    local direction = self.player.direction
    local hitboxX, hitboxY, hitboxWidth, hitboxHeight

    if direction == 'left' then
        hitboxWidth = 8
        hitboxHeight = 16
        hitboxX = self.player.x - hitboxWidth
```

```
            hitboxY = self.player.y + 2
        elseif direction == 'right' then
            hitboxWidth = 8
            hitboxHeight = 16
            hitboxX = self.player.x + self.player.width
            hitboxY = self.player.y + 2
        elseif direction == 'up' then
            hitboxWidth = 16
            hitboxHeight = 8
            hitboxX = self.player.x
            hitboxY = self.player.y - hitboxHeight
        else
            hitboxWidth = 16
            hitboxHeight = 8
            hitboxX = self.player.x
            hitboxY = self.player.y + self.player.height
        end

        self.swordHitbox = Hitbox(hitboxX, hitboxY, hitboxWidth, hitboxHeig
        self.player:changeAnimation('sword-' .. self.player.direction)
    end
```

- In the `update(dt)` method, we loop through all the entities in the map and check if they are colliding with the player's hitbox while the player is swinging their sword. Once finished swinging, the player is transitioned back to the `idle` state. If the player presses `space`, they can re-enter the `PlayerSwingSwordState`:

```
function PlayerSwingSwordState:update(dt)

    for k, entity in pairs(self.dungeon.currentRoom.entities) do
        if entity:collides(self.swordHitbox) then
            entity:damage(1)
            gSounds['hit-enemy']:play()
        end
    end

    if self.player.currentAnimation.timesPlayed > 0 then
        self.player.currentAnimation.timesPlayed = 0
        self.player:changeState('idle')
    end

    if love.keyboard.wasPressed('space') then
        self.player:changeState('swing-sword')
    end
end
```

Events

- An event is registered to trigger via some name, implemented via an anonymous function.
- Events are useful when something in the game warrants an event being triggered, or "dispatched".
- The anonymous callback function tied to the Event, the handler, is passed arguments via the event's dispatch.

Important Functions

- Take note of the following important functions from the Knife Event library:
  - `Event.on(name, callback)`
    - Calls `callback`, which is a function, whenever the message by its `name` is dispatched via `Event.dispatch()`.
  - `Event.dispatch(name, [params])`
    - Calls the callback function registered to `name`, set by `Event.on()`, with some optional `params` that will be sent to that callback function as arguments.
- For more extensive documentation, do reference the link below:
  - github.com/airstruck/knife/blob/master/readme/event.md (https://github.com/airstruck/knife/blob/master/readme/event.md)

Important Code

- Let's take a look at some examples of events in our Zelda program.
- Open up `PlayerWalkState.lua` in `src/states/entity/player`.
  - In this file, among other things, we're monitoring player collisions with the room walls, since the only way for the player to move to a new room is by colliding with a doorway.
  - When we do detect a collision between player and doorway, we shift the player to the center of the door and dispatch a `shift-DIRECTION` name, where DIRECTION is left, right, up or down:

```
if self.bumped then
    if self.entity.direction == 'left' then

        ...

        for k, doorway in pairs(self.dungeon.currentRoom.doorways) do
```

```
            if self.entity:collides(doorway) and doorway.open then
                self.entity.y = doorway.y + 4
                Event.dispatch('shift-left')
            end
        end

        ...

    elseif self.entity.direction == 'right' then

        ...

        for k, doorway in pairs(self.dungeon.currentRoom.doorways) do
            if self.entity:collides(doorway) and doorway.open then

                -- shift entity to center of door to avoid phasing thro
                self.entity.y = doorway.y + 4
                Event.dispatch('shift-right')
            end
        end

        ...
    elseif self.entity.direction == 'up' then

        ...

        for k, doorway in pairs(self.dungeon.currentRoom.doorways) do
            if self.entity:collides(doorway) and doorway.open then

                -- shift entity to center of door to avoid phasing thro
                self.entity.x = doorway.x + 8
                Event.dispatch('shift-up')
            end
        end

        ...

    else

        ...

        for k, doorway in pairs(self.dungeon.currentRoom.doorways) do
            if self.entity:collides(doorway) and doorway.open then

                -- shift entity to center of door to avoid phasing thro
```

```
                    self.entity.x = doorway.x + 8
                    Event.dispatch('shift-down')
                end
            end

            ...

        end
    end
```

- Now open up `Dungeon.lua` in `src/world/`. Here, you'll notice that we have Event listeners waiting for each Event `name` that we dispatch in `PlayerWalkState.lua`, with a callback function that will be triggered upon dispatch.
  - In this case, our callback function begins the shifting process of our room as discussed previously in the infinite dungeon algorithm:

```
function Dungeon:init(player)

    ...

    Event.on('shift-left', function()
        self:beginShifting(-VIRTUAL_WIDTH, 0)
    end)

    Event.on('shift-right', function()
        self:beginShifting(VIRTUAL_WIDTH, 0)
    end)

    Event.on('shift-up', function()
        self:beginShifting(0, -VIRTUAL_HEIGHT)
    end)

    Event.on('shift-down', function()
        self:beginShifting(0, VIRTUAL_HEIGHT)
    end)
end
```

- As you can see, Events are quite powerful tools that we can use to write our code in a modularized way.

Stenciling Revisited

- Similarly to how stencils in real life can be useful when drawing difficult shapes (such as a perfect circle), so, too, can stencils programmatically facilitate our graphic design.

Important Functions

- `love.graphics.stencil(func, [action], [value], [keepvals])`
  - Performs all stencil drawing within `func`; anything drawn during that time will act as the stencil pixels during `love.graphics.setStencilTest`.
  - `action` defines how those pixels will behave with pixels drawn onto them during `love.graphics.setStencilTest`.
  - `value` is the value `action` is reliant upon.
- `love.graphics.setStencilTest(compare_mode, compare_value)`
  - Compares pixels drawn via `compare_mode` with that of `compare_value`, only drawing pixels whose result of this mode is true.

Game Design via Data

- If you'll recall from earlier, in this project, we're representing our entities as specific data instantiated by a general class.
- In other words, instead of having separate classes for each entity in the game, we can have a single `Entity` class, with a separate Lua file denoting the properties of each specific entity in a big table.
- This is great design. Not only does it reduce redundancy in our code, it also serves to separate programming logic from descriptive data regarding our entities.
- This would facilitate the process of adding more entities or properties to the game, for example, especially if you're working as part of a team whereby perhaps some members are in charge of game design, and others are in charge of implementation.

NES Homebrew

- Homebrew is a term used to describe games and/or software based on older games/software which were subject to hardware restrictions, that have expanded the function of the previously restricted hardware.
  - For example, a version of an old NES Mario game on a modern emulator.
- Below are some links that you may find useful if curious to learn more:
  - [wiki.nesdev.com/w/index.php/Nesdev_Wiki (http://wiki.nesdev.com/w/index.php/Nesdev_Wiki)](http://wiki.nesdev.com/w/index.php/Nesdev_Wiki)
  - wiki.nesdev.com/w/index.php/Programming_guide (http://wiki.nesdev.com

/w/index.php/Programming_guide)

- wiki.nesdev.com/w/index.php/Installing_CC65 (http://wiki.nesdev.com/w/index.php/Installing_CC65)