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 7: Pokémon

Today's Topics

- StateStacks
  - We'll see how a StateStack, which supports running multiple states in parallel, is like a more advanced version of a State Machine.
- Turn-Based Systems
  - We'll implement our battle mechanics using a turn-based system, which is a core feature of Pokémon and other RPGs.
- GUIs
  - Graphical User Interfaces, or GUIs for short, are what bring these games to life. GUIs can include panels, scrollbars, textboxes, and many more visual ecosystems for navigating a game.
- RPG Mechanics
  - Leveling up, experience, damage calculations- these are all a part of the RPG

experience and we'll be taking a look at how to implement these features today.

Downloading demo code

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

RPG Mechanics

- For those curious, you can read more about what goes into making an RPG at the following link:
  - howtomakeanrpg.com (https://howtomakeanrpg.com)

StateStack

- The StateStack is the foundational class for this project; every other part of this program revolves around it.
- Previously, this had been the role of the State Machine. However, while a State Machine allows us to work with one state at a time, a StateStack will allow us to render multiple states at once.
  - For example, if you envision our states as a stack, you can imagine having the PlayState in the bottom of the stack as our `player` walks around, when suddenly a DialogueState is pushed onto the stack.
  - Rather than having to transition from PlayState to DialogueState (as we would've had to with a State Machine), we can simply "pause" the PlayState and render the DialogueState on top of it.
  - This allows us to return back to previous states as they were when we left them, rather than create new ones.
  - Only the top-most state on the stack is being updated at once, though this could be changed if we wanted it to.

Important Code

- With this in mind, open up `src/states/StateStack.lua` from our `pokemon` distro:
  - In `init`, we create a `states` table which is going to be how we implement our stack.

```
function StateStack:init()
```

```
    self.states = {}
end
```

- In `update`, we only update the stack at the end of the `states` table (which will be our "top" of the stack), since we only want to update the top state in the stack.

```
function StateStack:update(dt)
    self.states[#self.states]:update(dt)
end
```

- However, as you can see in `render` we do make sure we loop through our stack from bottom to top and render each state, such that removing the top state results in immediately continuing to update the state under it.

```
function StateStack:render()
    for i, state in ipairs(self.states) do
        state:render()
    end
end
```

- We also include methods to `clear` the stack, `push` and `pop` states to the stack (i.e., insert and remove).

```
function StateStack:clear()
    self.states = {}
end

function StateStack:push(state)
    table.insert(self.states, state)
    state:enter()
end

function StateStack:pop()
    self.states[#self.states]:exit()
    table.remove(self.states)
end
```

- You might've noticed we also implement a `processAI` method, which similarly to `update`, only interacts with the top state in the stack.
  - We don't actually use this method in this program, but that would be the way to update any AI you might have in your program

```
function StateStack:processAI(params, dt)
self.states[#self.states]:processAI(params, dt)
end
```

FadeInState

- Open up `src/states/game/FadeInState.lua`:
- FadeInState's sole purpose is to transition us into another state with a fade-in.
- `init` takes in a `color`, `time`, and an `onFadeComplete` callback function, and proceed to tween from transparency to the given `color` in `time` amount of seconds.
- Once that process completes, FadeInState is popped from the StateStack and the `onFadeComplete` callback function is executed.

```
function FadeInState:init(color, time, onFadeComplete)
    self.r = color.r
    self.g = color.g
    self.b = color.b
    self.opacity = 0
    self.time = time

    Timer.tween(self.time, {
        [self] = {opacity = 1}
    })
    :finish(function()
        gStateStack:pop()
        onFadeComplete()
    end)
end
```

- The only other method in this class, `render`, simply draws the fade-in to the screen.

```
function FadeInState:render()
    love.graphics.setColor(self.r, self.g, self.b, self.opacity)
    love.graphics.rectangle('fill', 0, 0, VIRTUAL_WIDTH, VIRTUAL_HEIGHT)
    love.graphics.setColor(1, 1, 1, 1)
end
```

StartState

- Our StartState consists of some text fields and a carousel of sprites that are displayed on the screen. There is then a fading transition to the next screen.

Important Code

- Open up `src/states/game/StartState.lua`:

- `init`:
  - Takes care of playing some background music and implementing the sprite carousel by tweening sprites from coordinates on the screen to off the screen and vice versa.

```
function StartState:init()
    gSounds['intro-music']:play()

    self.sprite = POKEMON_DEFS[POKEMON_IDS[math.random(#POKEMON_IDS)]].
    self.spriteX = VIRTUAL_WIDTH / 2 - 32
    self.spriteY = VIRTUAL_HEIGHT / 2 - 16

    self.tween = Timer.every(3, function()
        Timer.tween(0.2, {
            [self] = {spriteX = -64}
        })
        :finish(function()
            self.sprite = POKEMON_DEFS[POKEMON_IDS[math.random(#POKEMON
            self.spriteX = VIRTUAL_WIDTH
            self.spriteY = VIRTUAL_HEIGHT / 2 - 16

            Timer.tween(0.2, {
                [self] = {spriteX = VIRTUAL_WIDTH / 2 - 32}
            })
        end)
    end)
end
```

- `update`:
  - Monitors whether the user has pressed the "enter" key (or "return" if on a Mac), and if so, pushes FadeInState onto the stack, which recall takes in a color table, duration, and callback function for when the transition is complete.
  - Our callback function in this case takes care of cleaning up and pushing the next states onto the stack: first the PlayState, so that it is on the bottom of the Stack, then the DialogueState, so that rather than jumping head first into the game, the user can read some instructions, and lastly the FadeOutState to transition the screens nicely.

```
function StartState:update(dt)
    if love.keyboard.wasPressed('enter') or love.keyboard.wasPressed('r
        gStateStack:push(FadeInState({
            r = 1, g = 1, b = 1
        }, 1,
        function()
            gSounds['intro-music']:stop()
```

```
                self.tween:remove()

                gStateStack:pop()

                gStateStack:push(PlayState())
                gStateStack:push(DialogueState("" ..
                    "Welcome to the world of 50Mon! To start fighting monst
                    " monster, just walk in the tall grass! If you need to
                    "Good luck! (Press Enter to dismiss dialogues)"
                ))
                gStateStack:push(FadeOutState({
                    r = 1, g = 1, b = 1
                }, 1,
                function() end))
            end))
        end
    end
```

- Many of our states will be written in this way so that we can support asynchronous behavior.
  - For example, if we want something to happen after a dialogue screen is closed, but we don't know when the user will close it, we can just include the desired functionality in a callback function which will be executed once the user finally closes the dialogue screen.

PlayState

- In the PlayState, the `player` will be able to walk around the screen and encounter enemy Pokémon in the tall grass.
- The mechanics are somewhat similar to those in `zelda`, with a notable difference of having grid-aligned movement.
- You'll notice that our `player` in this program is always perfectly aligned with the tilemap grid. This, while not a strictly necessary feature of the game, allows for an easier implementation.

Important Code

- Our implementation of grid-aligned movement can be found in `src/states/entity/EntityWalkState.lua`.
  - `attemptMove`:

```
function EntityWalkState:attemptMove()

    ...

end
```

- Checks that the `player` is within the bounds of the map, and if so, adjusts our coordinates appropriately.
- In order to implement grid-aligned movement, we've given the `player` two sets of coordinates: one is a regular `(x, y)` pair; the other is a map `(x, y)` pair.

```
self.entity:changeAnimation('walk-' .. tostring(self.entity.direct

local toX, toY = self.entity.mapX, self.entity.mapY

if self.entity.direction == 'left' then
    toX = toX - 1
elseif self.entity.direction == 'right' then
    toX = toX + 1
elseif self.entity.direction == 'up' then
    toY = toY - 1
else
    toY = toY + 1
end

if toX < 1 or toX > 24 or toY < 1 or toY > 13 then
    self.entity:changeState('idle')
    self.entity:changeAnimation('idle-' .. tostring(self.entity.di
    return
end
```

- This allows us to center the `player` on any tile in our grid using the map coordinates, which we can then use to determine the corresponding regular coordinates to render the `player` on the screen.

```
self.entity.mapY = toY
self.entity.mapX = toX
```

- The strategy will be to tween from one set of map coordinates to the next, such that a step in any direction will preserve our alignment with the map.

```
Timer.tween(0.5, {
    [self.entity] = {x = (toX - 1) * TILE_SIZE, y = (toY - 1) * TI
```

```
    }):finish(function()
        if love.keyboard.isDown('left') then
            self.entity.direction = 'left'
            self.entity:changeState('walk')
        elseif love.keyboard.isDown('right') then
            self.entity.direction = 'right'
            self.entity:changeState('walk')
        elseif love.keyboard.isDown('up') then
            self.entity.direction = 'up'
            self.entity:changeState('walk')
        elseif love.keyboard.isDown('down') then
            self.entity.direction = 'down'
            self.entity:changeState('walk')
        else
            self.entity:changeState('idle')
        end
    end)
```

- With that in mind, open up `src/states/game/PlayState.lua`.
  - `init`:
    - Creates the level and plays some background music.

```
    function PlayState:init()
        self.level = Level()

        gSounds['field-music']:setLooping(true)
        gSounds['field-music']:play()

        self.dialogueOpened = false
    end
```

  - `update`:
    - Allows the user to heal their Pokémon by pressing the "p" key when not in DialogueState, and then defers to the `Level` class to update the world.

```
    function PlayState:update(dt)
        if not self.dialogueOpened and love.keyboard.wasPressed('p') t
            gSounds['heal']:play()
            self.level.player.party.pokemon[1].currentHP = self.level.

            gStateStack:push(DialogueState('Your Pokemon has been heal
            function()
                self.dialogueOpened = false
            end))
        end
```

```
        self.level:update(dt)
    end
```

- When reading through the code, you'll notice that DialogueState is similar in spirit to FadeInState in that it takes some information in order to perform its job, and additionally takes in a callback function to run upon completion.
- In this case, we pass in the text we want to display in DialogueState, and our callback function simply sets `dialogueOpened` back to `false`.
- You can double check what's happening in the DialogueState by opening it up (`src/states/game/DialogueState.lua`):
  - `init`:
    - Initializes the textbox that will contain the dialogue and the callback function that will run upon termination of this state.

```
function DialogueState:init(text, callback)
    self.textbox = Textbox(6, 6, VIRTUAL_WIDTH - 12, 64, text, gFo
    self.callback = callback or function() end
end
```

  - `update`:
    - Defers to the textbox's own `update` method and executes the callback function once the textbox is closed.

```
function DialogueState:update(dt)
    self.textbox:update(dt)

    if self.textbox:isClosed() then
        self.callback()
        gStateStack:pop()
    end
end
```

  - `render`:
    - Defers to the textbox's own `render` function, but obviously its purpose is to draw the textbox to the screen.

```
function DialogueState:render()
    self.textbox:render()
end
```

Level maps and Pokémon encounters

Important Code

- Open up `src/world/Level.lua`:
  - You'll notice that `init` creates separate tilemaps for the base layer of normal grass and for the patch of tall grass on the bottom of the screen.
  - This is done because our tall grass sprites have a transparent background.

```
function Level:init()
    self.tileWidth = 50
    self.tileHeight = 50

    self.baseLayer = TileMap(self.tileWidth, self.tileHeight)
    self.grassLayer = TileMap(self.tileWidth, self.tileHeight)
    self.halfGrassLayer = TileMap(self.tileWidth, self.tileHeight)

    self:createMaps()

    self.player = Player {
        animations = ENTITY_DEFS['player'].animations,
        mapX = 10,
        mapY = 10,
        width = 16,
        height = 16,
    }

    self.player.stateMachine = StateMachine {
        ['walk'] = function() return PlayerWalkState(self.player, self)
        ['idle'] = function() return PlayerIdleState(self.player) end
    }
    self.player.stateMachine:change('idle')
end
```

  - Apart from generating the map, the `Level` class relies heavily on the `player`'s state machine (which is separate from the game's StateStack).
- On that note, let's open up `src/states/entity/PlayerWalkState.lua`:
  - Here, we're mostly interested in our `checkForEncounter` method.
  - When the `player` tries to move, we check if they are standing in tall grass, and if so, we generate a 1 in 10 chance of encountering a Pokémon.
  - In the event of an encounter, the `player` is forced back into IdleState and a BattleState is pushed onto the StateStack.

```
function PlayerWalkState:checkForEncounter()
```

```
        local x, y = self.entity.mapX, self.entity.mapY

        if self.level.grassLayer.tiles[y][x].id == TILE_IDS['tall-grass'] a
            self.entity:changeState('idle')

            gSounds['field-music']:pause()
            gSounds['battle-music']:play()

            gStateStack:push(
                FadeInState({
                    r = 1, g = 1, b = 1,
                }, 1,

                function()
                    gStateStack:push(BattleState(self.entity))
                    gStateStack:push(FadeOutState({
                        r = 1, g = 1, b = 1,
                    }, 1,

                    function()
                    end))
                end)
            )

            self.encounterFound = true
        else
            self.encounterFound = false
        end
    end
```

GUIs

- Short for “graphical user interface”.
- Common widgets and elements include Panels, Labels, Textboxes, Scrollbars, and others.

Important Code

- Open up `src/gui/Panel.lua`:
  - This file provides the framework for creating a Panel for our GUI.
  - We do this by drawing two rectangles on top of each other with slightly different sizes and colors.

```
function Panel:init(x, y, width, height)
    self.x = x
    self.y = y
    self.width = width
    self.height = height
    self.visible = true
end

function Panel:render()
    if self.visible then
        love.graphics.setColor(1, 1, 1, 1)
        love.graphics.rectangle('fill', self.x, self.y, self.width, sel
        love.graphics.setColor(56/255, 56/255, 56/255, 1)
        love.graphics.rectangle('fill', self.x + 2, self.y + 2, self.wi
        love.graphics.setColor(1, 1, 1, 1)
    end
end

function Panel:toggle()
    self.visible = not self.visible
end
```

- Open up `src/gui/Textbox.lua`:
  - This file is understandably more complex, since textboxes must divide up their text based on their size.
  - For example, if the length of text surpasses the height of the textbox, then the text would ideally be divided up into multiple pages.
  - `init`:

```
    function Textbox:init(x, y, width, height, text, font)

        ...

    end
```

    - Creates a panel for the text (i.e., the "box") and sets the properties of the textbox in their own variables.

```
self.panel = Panel(x, y, width, height)
self.x = x
self.y = y
self.width = width
self.height = height
```

- It then determines the size of the "chunks" needed to display the text, based on the amount of text and textbox size.

```
    self.text = text
    self.font = font or gFonts['small']
    _, self.textChunks = self.font:getWrap(self.text, self.width - 12)
    self.chunkCounter = 1
    self.endOfText = false
    self.closed = false
```

- Finally, it calls `next` to determine whether the text has been displayed and the textbox can be closed, or if there are still chunks of text that must be displayed in additional pages.

```
    self:next()
```

- `nextChunks` is the method that keeps track of how many chunks of text we've displayed, and whether we're ready to close the textbox.

```
    function Textbox:nextChunks()
        local chunks = {}
        for i = self.chunkCounter, self.chunkCounter + 2 do
            table.insert(chunks, self.textChunks[i])
            if i == #self.textChunks then
                self.endOfText = true
                return chunks
            end
        end
        self.chunkCounter = self.chunkCounter + 3

        return chunks
    end
```

- `update` monitors whether the user has pressed the "space", "enter", or "return" key in order to continue calling the `next` method.

```
    function Textbox:update(dt)
        if love.keyboard.wasPressed('space') or love.keyboard.wasPressed('e
            self:next()
        end
    end
```

- Open up `src/gui/Selection.lua`:
  - A Selection is essentially a list of textual items (e.g., "Fight", "Run", etc.) that link to callbacks.

- Unsurprisingly, we set up the properties of our Selection in `init`:

```
function Selection:init(def)
    self.items = def.items
    self.x = def.x
    self.y = def.y
    self.height = def.height
    self.width = def.width
    self.font = def.font or gFonts['small']
    self.gapHeight = self.height / #self.items
    self.currentSelection = 1
end
```

- We monitor user interactions in `update` and respond accordingly:

```
function Selection:update(dt)
    if love.keyboard.wasPressed('up') then
        if self.currentSelection == 1 then
            self.currentSelection = #self.items
        else
            self.currentSelection = self.currentSelection - 1
        end
        gSounds['blip']:stop()
        gSounds['blip']:play()

    elseif love.keyboard.wasPressed('down') then
        if self.currentSelection == #self.items then
            self.currentSelection = 1
        else
            self.currentSelection = self.currentSelection + 1
        end
        gSounds['blip']:stop()
        gSounds['blip']:play()

    elseif love.keyboard.wasPressed('return') or love.keyboard.wasPress
        self.items[self.currentSelection].onSelect()
        gSounds['blip']:stop()
        gSounds['blip']:play()
    end
end
```

- We display our Selection to the screen in `render`.

```
function Selection:render()
    local currentY = self.y
    for i = 1, #self.items do
```

```
            local paddedY = currentY + (self.gapHeight / 2) - self.font:get
            if i == self.currentSelection then
                love.graphics.draw(gTextures['cursor'], self.x - 8, paddedY
            end
            love.graphics.printf(self.items[i].text, self.x, paddedY, self.
            currentY = currentY + self.gapHeight
        end
    end
```

- Open up `src/gui/Menu.lua`:
  - In this program, we're defining a Menu to be a Selection layered onto a Panel.

```
function Menu:init(def)
    self.panel = Panel(def.x, def.y, def.width, def.height)
    self.selection = Selection {
        items = def.items,
        x = def.x,
        y = def.y,
        width = def.width,
        height = def.height
    }
end

function Menu:update(dt)
    self.selection:update(dt)
end

function Menu:render()
    self.panel:render()
    self.selection:render()
end
```

Party and Pokémon

Important Code

- In our current distro, our Pokémon Party trivially consists of a single Pokémon.
- Nonetheless, take a look at `src/Party.lua`, which is the file we'd modify if we were going to add more metadata to our Party code.

```
function Party:init(def)
    self.pokemon = def.pokemon
end
```

```
function Party:update(dt)
end

function Party:render()
end
```

- Now, open up `src/Pokemon.lua`:
  - We've implemented this class as essentially a collection of stats.
  - `init`:
    - Is where we unsurprisingly set all the stats.
    - `base` stats are stats which all level 0 Pokémon of each species share.
    - `IV` attributes are attributes that pertain to each individual Pokémon.
      - This is how two Pokémon of the same level and species might have differing stats (perhaps one has higher HP, while the other has higher speed

```
function Pokemon:init(def, level)
self.name = def.name

self.battleSpriteFront = def.battleSpriteFront
self.battleSpriteBack = def.battleSpriteBack

self.baseHP = def.baseHP
self.baseAttack = def.baseAttack
self.baseDefense = def.baseDefense
self.baseSpeed = def.baseSpeed

self.HPIV = def.HPIV
self.attackIV = def.attackIV
self.defenseIV = def.defenseIV
self.speedIV = def.speedIV

self.HP = self.baseHP
self.attack = self.baseAttack
self.defense = self.baseDefense
self.speed = self.baseSpeed

self.level = level
self.currentExp = 0
self.expToLevel = self.level * self.level * 5 * 0.75

self:calculateStats()
```

```
    self.currentHP = self.HP
    end
```

- `statsLevelUp`:
  - Determines how much growth a Pokémon gets for each stat based on their corresponding `IV` attribute.
  - In short, a 6-sided dice is rolled 3 times for each stat (HP, attack, defense, speed).
  - After each roll, the resulting number is compared to that Pokémon's corresponding `IV` attribute.
  - If the dice roll is higher than the `IV`, there is no growth.
  - Otherwise, the Pokémon's stat is incremented by 1.
  - Thus, each time a Pokémon levels up, they have a chance to have each stat boosted by 3 points (in practice this never happens, since the `IV` values are capped at 5, but the result is that some Pokémon are boosted more than others per level-up).

```
function Pokemon:statsLevelUp()
    local HPIncrease = 0

    for j = 1, 3 do
        if math.random(6) <= self.HPIV then
            self.HP = self.HP + 1
            HPIncrease = HPIncrease + 1
        end
    end

    ...

end
```

- Open up `src/pokemon_defs.lua` to take a look at the different Pokémon definitions in our distro:
  - This file is essentially a collection of names and stats, which would make it very easy (as discussed last week) for a non-programmer to create additional Pokémon and help out in the overall design of the game.

Battles

Important Code

- Open up `src/battle/BattleSprite.lua`:
  - In this file, we allow for a Pokémon "texture" to be converted into a BattleSprite, with the main difference being that a BattleSprite has an `opacity` flag associated with it that will allow it to blink repeatedly as an indication of taking damage, whereas a "texture" is simply a static image.
  - This blinking effect is produced by a Shader. In our case, we are using a relatively simple Shader whose purpose is to turn a sprite completely white.

```
function BattleSprite:init(texture, x, y)
    self.texture = texture
    self.x = x
    self.y = y
    self.opacity = 1
    self.blinking = false

    self.whiteShader = love.graphics.newShader[[
        extern float WhiteFactor;
        vec4 effect(vec4 vcolor, Image tex, vec2 texcoord, vec2 pixcoor
        {
            vec4 outputcolor = Texel(tex, texcoord) * vcolor;
            outputcolor.rgb += vec3(WhiteFactor);
            return outputcolor;
        }
    ]]
end
```

  - To see where our Shader code came from, check out love2d.org/forums/viewtopic.php?t=79617 (https://love2d.org/forums/viewtopic.php?t=79617).
- Similarly to our Party situation, the Opponent in our game trivially has a single Pokémon in their Party.
- Take a look at `src/battle/Opponent.lua`:
  - This file is where you might include additional metadata for the Opponent's Party.

```
function Opponent:init(def)
    self.party = def.party
end

function Opponent:takeTurn()

end
```

- Now, let's take a look at `src/states/game/BattleState.lua`:
  - `init`:
    - As expected, sets up our battle. We set up the `player`, the dialog screen, the `opponent`, the health bars, the exp bar, setting flags along the way to ensure nothing is rendered out of turn.

```
function BattleState:init(player)
    self.player = player
    self.bottomPanel = Panel(0, VIRTUAL_HEIGHT - 64, VIRTUAL_WIDTH
    self.battleStarted = false

    self.opponent = Opponent {
        party = Party {
            pokemon = {
                Pokemon(Pokemon.getRandomDef(), math.random(2, 6))
            }
        }
    }

    self.playerSprite = BattleSprite(self.player.party.pokemon[1].
        -64, VIRTUAL_HEIGHT - 128)
    self.opponentSprite = BattleSprite(self.opponent.party.pokemon
        VIRTUAL_WIDTH, 8)

    self.playerHealthBar = ProgressBar {
        x = VIRTUAL_WIDTH - 160,
        y = VIRTUAL_HEIGHT - 80,
        width = 152,
        height = 6,
        color = {r = 189/255, g = 32/255, b = 32/255},
        value = self.player.party.pokemon[1].currentHP,
        max = self.player.party.pokemon[1].HP
    }

    self.opponentHealthBar = ProgressBar {
        x = 8,
        y = 8,
        width = 152,
        height = 6,
        color = {r = 189/255, g = 32/255, b = 32/255},
        value = self.opponent.party.pokemon[1].currentHP,
        max = self.opponent.party.pokemon[1].HP
    }
```

```
        self.playerExpBar = ProgressBar {
            x = VIRTUAL_WIDTH - 160,
            y = VIRTUAL_HEIGHT - 73,
            width = 152,
            height = 6,
            color = {r = 32/255, g = 32/255, b = 189/255},
            value = self.player.party.pokemon[1].currentExp,
            max = self.player.party.pokemon[1].expToLevel
        }

        self.renderHealthBars = false
        self.playerCircleX = -68
        self.opponentCircleX = VIRTUAL_WIDTH + 32
        self.playerPokemon = self.player.party.pokemon[1]
        self.opponentPokemon = self.opponent.party.pokemon[1]
    end
```

- `update`:
  - mainly depends on `triggerSlideIn` to kick off the battle, tweening in the components of the battle screen and subsequently triggering the dialogue via `triggerStartingDialogue`, which displays the dialogue and eventually pushes the BattleMenuState to the StateStack.

```
    function BattleState:update(dt)
        if not self.battleStarted then
            self:triggerSlideIn()
        end
    end

    function BattleState:triggerSlideIn()
        self.battleStarted = true
        Timer.tween(1, {
            [self.playerSprite] = {x = 32},
            [self.opponentSprite] = {x = VIRTUAL_WIDTH - 96},
            [self] = {playerCircleX = 66, opponentCircleX = VIRTUAL_WI
        })
        :finish(function()
            self:triggerStartingDialogue()
            self.renderHealthBars = true
        end)
    end
```

  - `render`: displays everything on the screen

- Open up `src/states/game/BattleMenuState.lua`:

    - This is where we present the Selection menu to the user and define what happens when the user chooses "fight" or "run".
    - Most of the logic in this file is in the `init` function, in which we create the menu and define the callback functions for each selection.
    - In the event that the user selects "fight", we push a TakeTurnState to the StateStack.
    - If the user chooses "Run", we pop the BattleMenuState off the StateStack, pushing an additional BattleMessageState to let the user know they fled successfully, after which we push a FadeInState, resume the field music, pop the StateStack twice (once for the message, once for the battle), and finally push a FadeOutState to return back to the field.
- Finally. let's take a look at `src/states/game/TakeTurnState.lua`:
    - `init`:
        - Stores the current `battleState` and determines which Pokémon should attack first (based on speed).

```
function TakeTurnState:init(battleState)
    self.battleState = battleState
    self.playerPokemon = self.battleState.player.party.pokemon[1]
    self.opponentPokemon = self.battleState.opponent.party.pokemon
    self.playerSprite = self.battleState.playerSprite
    self.opponentSprite = self.battleState.opponentSprite

    if self.playerPokemon.speed > self.opponentPokemon.speed then
        self.firstPokemon = self.playerPokemon
        self.secondPokemon = self.opponentPokemon
        self.firstSprite = self.playerSprite
        self.secondSprite = self.opponentSprite
        self.firstBar = self.battleState.playerHealthBar
        self.secondBar = self.battleState.opponentHealthBar
    else
        self.firstPokemon = self.opponentPokemon
        self.secondPokemon = self.playerPokemon
        self.firstSprite = self.opponentSprite
        self.secondSprite = self.playerSprite
        self.firstBar = self.battleState.opponentHealthBar
        self.secondBar = self.battleState.playerHealthBar
    end
end
```

    - `enter`:
        - Calls the `attack` function for each Pokémon and checks for deaths as needed.

```
function TakeTurnState:enter(params)
    self:attack(self.firstPokemon, self.secondPokemon, self.firstS
    function()
        gStateStack:pop()
        if self:checkDeaths() then
            gStateStack:pop()
            return
        end
        self:attack(self.secondPokemon, self.firstPokemon, self.se
        function()
            gStateStack:pop()
            if self:checkDeaths() then
                gStateStack:pop()
                return
            end
            gStateStack:pop()
            gStateStack:push(BattleMenuState(self.battleState))
        end)
    end)
end
```

- `attack`:
  - First pushes a BattleMessageState to let the user know who is attacking, then plays the attack animation and sound.
  - Next, the damaged Pokémon is made to blink a few times, and their health bar is decreased upon damage calculation.
- `checkDeaths`:
  - Checks whether a Pokémon's health has fallen below 1 HP, and if so, causes them to faint.

```
function TakeTurnState:checkDeaths()
    if self.playerPokemon.currentHP <= 0 then
        self:faint()
        return true
    elseif self.opponentPokemon.currentHP <= 0 then
        self:victory()
        return true
    end
    return false
end
```

- `faint`:
  - Drops the user's sprite from the screen, and pushes a BattleMessageState to

let the user know they've fainted.

- Then, it pushes a FadeInState, heals the user's Pokémon to full health, resumes the field music, and pops the StateStack before pushing a FadeOutState and DialogueState to let the user know their Pokémon has been healed.
- To signify defeat, we fade in and out with a black screen rather than a white one.

- `victory`:
  - Drops the enemy sprite from the screen, plays victory music and pushes a BattleMessageState to let the user know they've won.
  - Then, it calculates exp earned, leveling up the user's Pokémon if needed, and fading out with a white screen to signify victory.