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 2: Breakout

Today's Topics

- Sprite Sheets
  - Sprite Sheets allow us to condense all the images we need to load for our game into one big image, with each sprite assigned a specific cell in the "sheet".
- Procedural Layouts
  - We'll take a look at how to dynamically generate bricks for our breakout implementation!
- Managing State
  - This week we'll go one step further with improving our state machine in terms of code cleanliness and best practices, particularly by taking advantage of the state machine class's built-in methods to ensure we have a less polluted global namespace.
- Levels

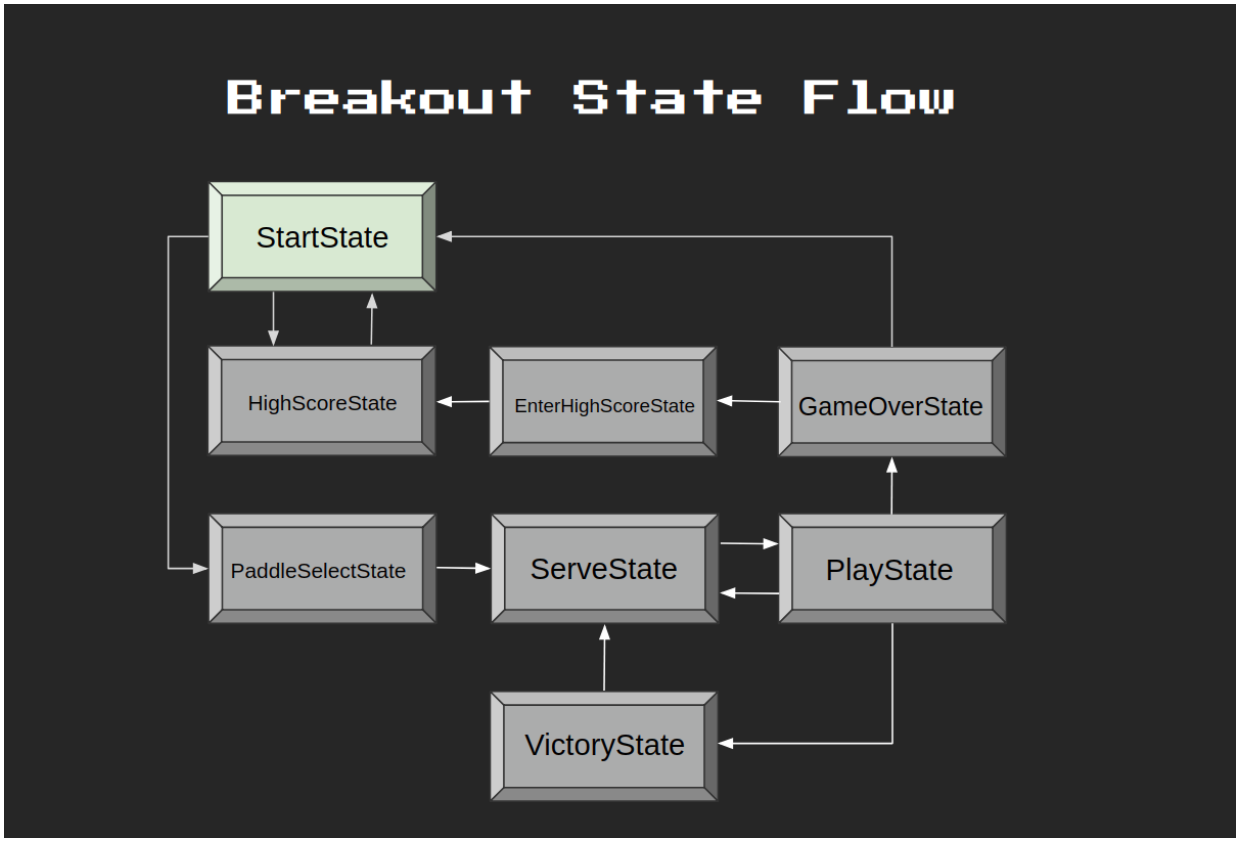
        - We'll introduce the concept of "levels" to our game, allowing a player to "level up" and changing what we're displaying to the screen accordingly.
    - Player Health
        - We'll learn how to keep track of player "health" using hearts to give them a number of chances before losing the game.
    - Particle Systems
        - We'll learn more about Particle Systems this week so as to provide more advanced aesthetic qualities to our game.
    - Collision Detection Revisited
        - Collision Detection will be a bit more advanced this week!
    - Persistent Save Data
        - In the context of High Scores, it's useful to know how to save information relevant to our game so that the next time we open it, we can still access that old information.

Downloading demo code

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

Breakout State Flow

- Below we map out the state flow for our final version of Breakout.
- The program will begin in StartState, which can transition back and forth between itself and HighScoreState (since the user can check High Scores before playing).
- StartState can also transition to PaddleSelectState, which transitions on to ServeState.
- During gameplay, the program will transition back and forth between ServeState and PlayState as the user loses health. If the user clears the level, the program transitions onto the VictoryState and then to the ServeState for the next level.
- Alternatively, if the user loses all their lives before clearing the level, the program will transition from PlayState to GameOverState, optionally transitioning to the EnterHighScoreState (if the user achieves a High Score) and then on to the HighScoreState, or simply transitioning from GameOverState to StartState if the user does not have a High Score.
- Take a moment to examine the diagram below if in need of a visual aid:



breakout0 (“The Day-0 Update”)

- At this point, you will want to have downloaded the demo code in order to follow along.
- breakout0 displays the main screen and allows the user to toggle between the “Start” and “High Score” options.

Project Organization

- The first thing to note will be the organizational changes we’ve made to our workspace and to our code files themselves since last week.
- You’ll notice that we’ve organized our workspace using appropriate subdirectories for our different file types. This is a big improvement from the messy workspaces we’ve had in the past.
- Now, we have our `main.lua` file out in the open, while our sound, font, graphics, and other files are grouped together in their own subdirectories. This is a good example to follow when working on your own projects!
- In a similar vein, you’ll notice that we’ve moved away from our previous tendency to store related information in individual variables. Instead, we have begun modularizing our code even more by storing related values in tables, such as `gFonts`, `gTextures`, and `gSounds` from `main.lua` (recall that use of the prefix `g` is a common convention to denote that

the variable in question will have global scope).

- Furthermore, we've moved over our familiar constant globals from `main.lua` (e.g. `WINDOW_WIDTH`, `WINDOW_HEIGHT`, `VIRTUAL_WIDTH`, `VIRTUAL_HEIGHT`) into their own separate file, `src`, which we are then importing to `main.lua`
- These wholesome organizational changes may seem unnecessary at first, but as your own projects grow more complex, you will come to depend on these organizational strategies, especially once you begin to work with collaborators who may otherwise not understand your project's layout.

Important Code

- Next in `main.lua` you'll notice:
  - `love.load()` in which we set up our global tables as well as our state machine.
  - `love.update(dt)` whose logic we delegate to our state machine.
  - `love.resize(w, h)`
  - `love.keypressed(key)` which we funnel into `love.keyboard.wasPressed(key)`, our own custom function as we did last week (recall why?).
  - `love.draw()` which renders what we see on the screen.
  - `displayFPS()` a custom function (which should be a review from previous weeks).
- The remaining logic will be found in our `src/states` subdirectory, since our `love.update(dt)` function in `main.lua` is deferring to our state machine. Currently, we only have BaseState and StartState.
  - `StartState:update(dt)`: allows the user to toggle between "Start" and "High Scores" on the screen, highlighting their selection and playing a toggle sound effect.
  - `StartState:render()`: includes some graphics configurations specific to the StartState.
  - Like last week, our BaseState is essentially a skeleton for our other states so that we don't have to redefine the same methods for each State class.
- Be sure to read through each file carefully so as to understand its role in the overarching project. The code itself should look familiar, but do take the time to familiarize yourself with the new organizational layout.

breakout1 ("The Quad Update")

- breakout1 takes advantage of Sprite Sheets in order to render a Paddle sprite during PlayState.

What is a Sprite Sheet?

- A Sprite Sheet is essentially a bitmap image containing smaller images (i.e., sprites) within itself. A Sprite Sheet can be split into Quads, that is, rectangular sections of itself (each encapsulating a single sprite), so that instead of having multiple image files in our project for each sprite, we can more efficiently use a single file that we section out into Quads when wanting to render a particular sprite.

Important Functions

- `love.graphics.newQuad(x, y, width, height, dimensions)`
  - This function allows us to specify rectangle boundaries of our Quad and pass in the dimensions (returned via `image:getDimensions` on whichever texture we want to make a Quad for).
- `love.graphics.draw(texture, quad, x, y)`
  - This is a variant of `love.graphics.draw()`, which we've seen, but in this case we can pass in a Quad to draw *just* the specific part of the texture we want, not the entire thing!

Important Code

- We've added a few files to our `src` subdirectory (which, recall, imports all its files to `main.lua` via `Dependencies.lua`).
- Open up `Util.lua`, which handles the logic for generating our Quads. You should find the following functions:
  - `GenerateQuads(atlas, tilewidth, tileheight)`
    - We've written this function, which takes in an `atlas` (synonymous with Sprite Sheet) and our desired tile dimensions to determine the dimensions of our Quads. With this info, we can loop over our atlas (treating it as a 2D coordinate system with origin at the upper left corner) and generate our Quads along the way.
  - `table.slice(tbl, first, last, step)`
    - This is just a helper function to allow us to slice Lua tables in the same way that we can slice Python lists.
  - `GenerateQuadsPaddles(atlas)`
    - This function is specifically made to piece out the paddles from the sprite sheet. For this, we have to piece out the paddles a little more manually, since they are all different sizes.

- Next, over in `Paddle.lua`, we've created our Paddle class.
  - `Paddle:init` initializes the Paddle.
  - `Paddle:update(dt)` allows the user to interact with the Paddle by moving it side to side.
  - `Paddle:render()` displays the Paddle
- Lastly, take a look at `PlayState.lua`.
  - `PlayState:init`: instantiates a Paddle.
  - `PlayState:update(dt)` allows the user to pause and quit the game, delegating the Paddle interactivity logic to the Paddle's own `update` method.
  - `PlayState:render()` calls the Paddle's own `render()` method and also displays paused text.

breakout2 ("The Bounce Update")

- breakout2 uses AABB Collision Detection so that the Ball can bounce when it collides with the Paddle or the walls.

Important Code

- In `main.lua` you'll notice that we've modified the `gFrames` table to include Quads for our Ball.
- Check out `Util.lua` to see how we've extracted these.
- `function GenerateQuadsBalls(atlas)`
  - In this function, we're finding the offset for our Ball sprites in our Sprite Sheet and looping over it, generating Quads for the Balls as we go. This is essentially the analogous function for how we extracted our Paddle sprites.
  - The end result is that we can now access our Paddle and Ball sprites without needing to loop over the entire Sprite Sheet each time, which will prove useful going forward.
- Now we can read through `Ball.lua`, which creates our Ball class.
  - `Ball:init(skin)` initializes the Ball given a specific Ball sprite.
  - `Ball:collides(target)` checks for collisions using AABB Collision Detection.
  - `Ball:reset()` resets the Ball to the middle of the screen.
  - `Ball:update(dt)` implements behavior for bouncing off walls.
  - `Ball:render()` renders the Ball to the screen.
- Again, we conclude with some additions to `PlayState.lua`:

- `PlayState:init()` now instantiates the Ball.
- `PlayState:update()` calls the Ball's `update` method and naively implements behavior for bouncing off the Paddle. Can you think of a potential issue with our implementation? (Hint: think about what else we might want to do besides reversing the Ball's velocity). You might be able to observe the issue by trying to bounce the Ball off the Paddle at an angle.
- `PlayState:render()` calls the Ball's `render()` method.

breakout3 ("The Brick Update")

- breakout3 renders Bricks onto the screen.

Important Code

- Notice that the `gFrames` table in `main.lua` has been modified again, this time to include Quads for our Bricks.
- `Util.lua` contains the implementation details of extracting the Brick sprites from our Sprite Sheet.
  - In this case, since the Brick sprites are at the top of our Sprite Sheet, the extraction process becomes much simpler:

```
function GenerateQuadsBricks(atlas)
    return table.slice(GenerateQuads(atlas, 32, 16), 1, 21)
end
```

- `Brick.lua` creates our Brick class.
  - `Brick:init(x, y)` initializes a Brick. Importantly, we include an `inPlay` flag to serve as a signal for whether a Brick is still in play or if it should disappear from the screen. In the context of our breakout program, this is an effective shortcut, but do note that in larger programs, it would be better practice to free memory that is not being used instead of just hiding it from view.
  - `Brick:hit()` hides a Brick by toggling the `inPlay` flag to `false`.
  - `Brick:render()` renders a Brick to the screen.
- `PlayState.lua` references a new class, `LevelMaker.lua`, which encapsulates all the logic for generating new levels (which boil down to Brick layouts). It also checks for collisions between the Ball and the Bricks (hiding Bricks as needed) and renders the "in play" Bricks to the screen.
- To create a level, `LevelMaker.lua` randomly generates a table of Bricks that can be

rendered to the screen. Read through the `LevelMaker.createMap(level)` function carefully, paying close attention to the comments.

breakout4 ("The Collision Update")

- breakout4 implements bouncing behavior for the Ball upon a collision with a Brick. It also fixes our previous naive implementation of bouncing behavior between the Ball and the Paddle.

Important Algorithms

- To fix our Paddle collision, we need to take the difference between the Ball's `x` value and the Paddle's center, which is:

  ```
  paddle.x + (paddle.width / 2) * ball.x
  ```

  We use this formula to scale the ball's `dx` in the negative direction.
- We perform this operation on either side of the Paddle based on the Paddle's `dx`. If on the right side, the differential will be negative, so we need to call `math.abs` to make it positive, then scale it by a positive amount so `dx` becomes positive.
- For Brick collision, we must check which edge of the Ball is not inside the Brick. Below is our algorithm:

  ```
  if left edge of ball is outside brick and dx is positive:
      trigger left-side collision
  else if right edge of ball is outside brick and dx is negative:
      trigger right-side collision
  else if top edge of ball is outside brick:
      trigger top-side collision
  else
      trigger bottom-side collision
  ```

  This is a fairly simple collision algorithm, so it is not the most accurate, particularly when faced with corner-cases, but it works essentially 99% of the time. For a more robust solution, do check out the following alternative solution in the links below, if curious:

  - github.com/noooway/love2d_arkanoid_tutorial (https://github.com/noooway/love2d_arkanoid_tutorial)
  - github.com/noooway/love2d_arkanoid_tutorial/wiki/Resolving-Collisions (https://github.com/noooway/love2d_arkanoid_tutorial/wiki/Resolving-Collisions)

Important Code

- In `PlayState.lua`, we've updated the `update()` function to reflect the collision algorithms mentioned above.
- You'll notice our new collision logic for Ball/Paddle collisions as discussed:

```
if self.ball.x < self.paddle.x + (self.paddle.width / 2) and self.paddle.dx <
    self.ball.dx = -50 + -(8 * (self.paddle.x + self.paddle.width / 2 - self
elseif self.ball.x > self.paddle.x + (self.paddle.width / 2) and self.paddle
    self.ball.dx = 50 + (8 * math.abs(self.paddle.x + self.paddle.width / 2
end
```

- And below that, our Ball/Brick collision detection, albeit with a small modification to account for corner-collisions. Namely, we insert a `break` statement to ensure the Ball's collision mechanics are only evaluated against a single Brick in the event that it collides at the corner of two Bricks.
- And of course, we slightly increase the Ball's velocity after a collision.

```
for k, brick in pairs(self.bricks) do
    if brick.inPlay and self.ball:collides(brick) then
        brick:hit()
        if self.ball.x + 2 < brick.x and self.ball.dx > 0 then
            self.ball.dx = -self.ball.dx
            self.ball.x = brick.x - 8
        elseif self.ball.x + 6 > brick.x + brick.width and self.ball.dx < 0 t
            self.ball.dx = -self.ball.dx
            self.ball.x = brick.x + 32
        elseif self.ball.y < brick.y then
            self.ball.dy = -self.ball.dy
            self.ball.y = brick.y - 8
        else
            self.ball.dy = -self.ball.dy
            self.ball.y = brick.y + 16
        end
        self.ball.dy = self.ball.dy * 1.02
        break
    end
end
```

breakout5 ("The Hearts Update")

- breakout5 implements the idea of "health" for the user, which is displayed on the screen

as hearts.

Important Code

- Notice in `StartState.lua` we are continuing our shift away from global variables (with global tables being an exception) by integrating relevant data fields into our State Machine.
  - On line 35, we see that when we transition from StartState to ServeState, we are passing along `paddle`, `bricks`, `health`, and `score` through `gStateMachine:change()`, which we might've alternatively implemented as global values.
  - This design is cleaner since it allows us to remove unnecessary values from our files. For example, it makes sense to have `paddle` and `bricks` when we are in ServeState, but not so much when we are in HighScoreState.
- On that note, take a look at `ServeState.lua`, which serves a very similar purpose to the ServeState from Pong. The code should look familiar, as all we're doing here is providing a state in which the user can hit the `enter` key to transition to the PlayState.
- In `main.lua` you'll notice we've added `hearts` to our `gFrames` table and included a helper function for rendering the user's `health` on the screen. It simply draws the corresponding number of full hearts followed by empty hearts per the user's `health`.

```
function renderHealth(health)
    local healthX = VIRTUAL_WIDTH - 100

    for i = 1, health do
        love.graphics.draw(gTextures['hearts'], gFrames['hearts'][1], healthX
        healthX = healthX + 11
    end

    for i = 1, 3 - health do
        love.graphics.draw(gTextures['hearts'], gFrames['hearts'][2], healthX
        healthX = healthX + 11
    end
end
```

- `PlayState.lua` now also takes care of keeping score, monitoring the user's `health`, and transitioning to other States as needed. You should be able to find the `health`-tracking code in `PlayState:update()`, which simply decreases health and reverts to ServeState when the Ball goes past the Paddle beyond the bottom of the screen:

```
if self.ball.y >= VIRTUAL_HEIGHT then
    self.health = self.health - 1
```

```
        gSounds['hurt']:play()

        if self.health == 0 then
            gStateMachine:change('game-over', {
                score = self.score
            })
        else
            gStateMachine:change('serve', {
                paddle = self.paddle,
                bricks = self.bricks,
                health = self.health,
                score = self.score
            })
        end
    end
```

The score tracking can also be found in `PlayState:update()`, where we simply add 10 points to the `score` every time a Ball/Brick collision is detected.

- `GameOverState.lua`, which is unsurprisingly called when the user loses all health, creates our new GameOverState that simply renders a "Game Over" screen with the final score. When a user presses the `enter` key in this state, they're taken back to the StartState.

breakout6 ("The Pretty Colors Update")

- breakout6 updates the levels to include different colors and layouts of Bricks.

Important Code

- We've made a few modifications to `LevelMaker.lua` in order to allow for a more varied gaming experience.
- The changes mostly consist of adding some new flags so that we can display different colors and layouts for our bricks.
- Read through the changes to this file carefully, paying special attention to the comments, to understand how we are generating our new map layouts.

breakout7 ("The Tier Update")

- breakout7 differentiates between the different tiers and colors of Bricks, making updates to the gameplay and the scoring as a result.

Important Code

- Take a look at `Brick.lua`. You'll notice that we've updated the `Brick:hit()` method such that it only toggles the `inPlay` flag if the Brick being hit is of the lowest tier and color. Otherwise, it simply decrements the tier and/or color of the Brick.

```
if self.tier > 0 then
    if self.color == 1 then
        self.tier = self.tier - 1
        self.color = 5
    else
        self.color = self.color - 1
    end
else
    if self.color == 1 then
        self.inPlay = false
    else
        self.color = self.color - 1
    end
end
```

- In `PlayState.lua`, we've updated the scoring algorithm to attribute higher values to Bricks of higher tiers and colors. Whereas previously we always added 10 points for any Ball/Brick collision, we now take color and tier into account:

```
self.score = self.score + (brick.tier * 200 + brick.color * 25)
```

breakout8 ("The Particle Update")

- breakout8 takes advantage of LÖVE2D's Particle System features to create a nicer visual effect when the Ball collides with a Brick.

Important Functions

- `love.graphics.newParticleSystem(texture, particles)`
  - This function takes in a particle texture and maximum number of particles we can emit and creates a particle system we can emit from, update, and render.
- For more LÖVE particle system functions and info, check out this link:
  - love2d.org/wiki/ParticleSystem (https://love2d.org/wiki/ParticleSystem)

Important Code

- You'll notice, when running the program, that the particles adopt the color of the Brick being hit.
- Atop our `Brick` class, we define `paletteColors` as a table of RGB values corresponding to blue, green, red, purple, and gold.
- In `Brick:init()`, we create a new particle system:

```
self.psystem = love.graphics.newParticleSystem(gTextures['particle'], 64)
self.psystem:setParticleLifetime(0.5, 1)
self.psystem:setLinearAcceleration(-15, 0, 15, 80)
self.psystem:setEmissionArea('normal', 10, 10)
```

- In `Brick:hit()`, we then use `paletteColors` to cause our particle system to interpolate between the appropriate colors, brighter for higher tiers and fading away over the particle's lifetime.

```
self.psystem:setColors(
    paletteColors[self.color].r / 255,
    paletteColors[self.color].g / 255,
    paletteColors[self.color].b / 255, 55 * (self.tier + 1) / 255,
    paletteColors[self.color].r / 255,
    paletteColors[self.color].g / 255,
    paletteColors[self.color].b / 255, 0
)
self.psystem:emit(64)
```

- Finally, we need a separate render function for our particles so it can be called after all bricks are drawn. Otherwise, some bricks would render over other bricks' particle systems:

```
function Brick:renderParticles()
    love.graphics.draw(self.psystem, self.x + 16, self.y + 8)
end
```

breakout9 ("The Progression Update")

- breakout9 allows the user to beat a level and progress to the next.

Important Code

- You'll notice in line 40 of `StartState.lua` that we are now passing in an additional field to `ServeState` upon transition, namely, `level`.

- We'll continue to do this once each level has been beaten, incrementing `level` as needed.
- On line 204 in `PlayState.lua`, we've written a new method, `checkVictory()`, that checks if the current level has been beaten, by checking if every Brick's `inPlay` flag has been toggled to `false`.
- This method is called on line 88 within `Playstate:update(dt)`, since it makes sense to check if the level has been beaten each time a Brick is hit.
- `VictoryState.lua` contains the code for producing our victory screen. It is only ever activated when the user beats a level. This is also where we increment the value of `level`. Upon increasing the `level`, we must create a new map:

```
function VictoryState:update(dt)
    self.paddle:update(dt)

    self.ball.x = self.paddle.x + (self.paddle.width / 2) - 4
    self.ball.y = self.paddle.y - 8

    if love.keyboard.wasPressed('enter') or love.keyboard.wasPressed('return
        gStateMachine:change('serve', {
            level = self.level + 1,
            bricks = LevelMaker.createMap(self.level + 1),
            paddle = self.paddle,
            health = self.health,
            score = self.score
        })
    end
end
```

breakout10 ("The High Scores Update")

- breakout10 introduces the ability to view high scores.

Important Functions

- `love.filesystem.setIdentity(identity)`
  - This function sets the active subfolder in the default LÖVE save directory for reading and writing files.
- `love.filesystem.exists(path)`
  - Checks if a file exists in our save directory.
- `love.filesystem.write(path, data)`

    - Writes data, as a string, to the file location at `path`.
- `love.filesystem.lines(path)`
    - Returns an iterator over the string lines in a file at `path`, located in our active identity path.

Important Code

- In `main.lua`, we've written a new function (`loadHighScores()`) to load existing high scores from a `.lst` file, which is saved in LÖVE2D's default save directory (called `breakout`).
- If the file doesn't exist, we create it and fill it with arbitrary values, leaving at least 10 blank entries.
- Once the file exists, we can iterate through every line, filling in names and scores to a table.

breakout11 ("The Entry Update")

- breakout11 allows users to save their own high scores.

Important Code

- In `EnterHighScoreState.lua`, we allow users to enter their high scores by choosing a 3-character name. The name is selected by toggling the 3 characters using the up and down keys. We use ASCII values to implement this behavior.
- Once the user settles on a name, we write their score to the `breakout` file, taking care to loop through the existing scores in order to replace the lowest one (if necessary).

breakout12 ("The Paddle Select Update")

- breakout12 introduces a new state that allows the user to select a Paddle sprite before starting the game.

Important Code

- In `PaddleSelectState.lua`, we render a new screen to the user which contains some text, a left arrow, a right arrow, and a Paddle.
- The user can toggle between Paddles using the arrows and can make a selection by pressing `enter`.

- Notice that this State is now in charge of transitioning over to `ServeState` and passing along the relevant values.

breakout13 (“The Music Update”)

- breakout13 adds background music to the game.
- This is done in `main.lua` by adding the `music` track to our `gSounds` table and calling `gSounds['music']:play()` as well as `gSounds['music']:setLooping(true)`.