Moonpie

Feb 20, 2022

Contents:

1 Getting Started **3**

2 moonpie **5**
- 2.1 moonpie.class . . . 5
- 2.2 moonpie.keyboard . . . 5
- 2.3 moonpie.event_system . . . 5

3 moonpie.collections **7**
- 3.1 Array . . . 7
- 3.2 Grid . . . 7
- 3.3 moonpie.collections.iterators . . . 8

4 moonpie.entities **9**

5 moonpie.events **11**
- 5.1 Available Events . . . 11

6 moonpie.graphics **13**
- 6.1 moonpie.graphics.colors . . . 13
- 6.2 moonpie.graphics.font . . . 13
- 6.3 moonpie.graphics.image . . . 13

7 moonpie.math **15**
- 7.1 Functions . . . 15
- 7.2 moonpie.math.cards . . . 15
- 7.3 moonpie.math.ipoint . . . 15
- 7.4 moonpie.math.rectangle . . . 16

8 moonpie.state **17**
- 8.1 Add Action Validator . . . 17
- 8.2 Binding Components . . . 17

9 moonpie.sorts **19**

10 moonpie.tables **21**

11 moonpie.test_helpers **23**
- 11.1 New Assertions . . . 23

11.2 Array Extensions . . . 23
11.3 Component Extensions . . . 23

12 moonpie.ui **25**
12.1 moonpie.ui.components . . . 25
12.2 moonpie.ui.styles . . . 27
12.3 Default Styles . . . 27
12.4 Built In Components . . . 27
12.5 body . . . 27
12.6 image . . . 28
12.7 textbox . . . 28

13 moonpie.utility **29**

14 Indices and tables **31**

Moonpie is a framework designed for easy development within the Love2D engine. It is focused around providing great UX with responsive designs for flexible layouts of components. At the same time it allows for flexibility in how components are rendered. There is also support with libraries for collections, math, entity-component-systems, etc…

CHAPTER 1

Getting Started

Moonpie-template is the easiest way to start a project using Moonpie.

```
$ There is a command that will clone this and set up project
```

CHAPTER 2

moonpie

This is the entry point into the API and provides access to the rest of the framework. This eliminates the need to require specific modules for operation of the framework.

```
moonpie = require "moonpie"
```

2.1 moonpie.class

Utilized the middleclass library for functionality: https://github.com/kikito/middleclass

2.2 moonpie.keyboard

Provides access to keyboard routines, including the ability to configure hotkeys that will trigger a function.

```
local keyboard = require "moonpie.keyboard"
keyboard:hotkey("a", function() print("Pressed A") end)
keyboard:hotkey("shift+a", function() print("Pressed shift(either)+a") end)
Keyboard:hotkey("alt+ctrl+shift+8", function() print("alt, ctrl, shift, 8 ... all at␣
↪the same time") end)
```

hotkey(keycode, function) Maps a function that will be triggered on keypress. Only one function can be mapped to the callback. _keycode_ can be formatted with context keys separated by a plus: alt, ctrl, shift **In that order** Examples: “a”, “ctrl+a”, “ctrl+shift+a”, “alt+a”

2.3 moonpie.event_system

A simple mechanism for creating loosely coupled components that can dispatch events. Events are registered onto the system to be handled. Subscribers can register to specific events. Those events can then be dispatched to with additional arguments as necessary.

Event messages are formatted the same as actions to the state management store. This keeps things consistent but allows for different purposes for dispatching.

```
local my_callback_function = function(data)
  print(data.payload)
end

local events = require "moonpie.event_system"
events.register("hello")
events.subscribe("hello", my_callback_function)
events.dispatch( { type = "hello", payload = "extra_data" })
```

CHAPTER 3

moonpie.collections

Various collections and structures to help provide functionality to lua tables.

```
local a_list = moonpie.collections.list:new()
a_list:add("stuff", 1, 4, "more-stuff")
if a_list:contains(4) then
  print "It does!"
end
```

3.1 Array

Provides an easy way to manage multidimensional arrays without knowing the bounds ahead of time. The Array is more of an "Array Bag", where you can place values based on multidimensional coordinates, but iterating over the values isn't really supported at this time. Grid is a better implementation for 2D Arrays that have known dimensions.

```
local Array = require "moonpie.collections.array"
local a = Array:new(3) -- Define a 3-dimensional array
a(3, 2, 1, "bar")
print(a(3, 2, 1)) -- # bar
```

3.2 Grid

Provides a 2D array element that makes it easier to track and assign elements for 2D lists. Initialized to a specific size and can handle default values when the value has not been set.

```
local Grid = require "moonpie.collections.grid"
local g = Grid:new(10, 10, "default")
g:set(3, 2, "hello")
print(g:get(3, 2)) -- "hello"
print(g:get(8, 2)) -- "default"
```

3.2.1 Properties & Methods

default The default returned if the location requested is empty

get(x, y) Retrieves the value at the specified location. If empty, return default or nil

height The height of the grid data

set(x, y, value) Sets the value at the specified location, overriding any previous value assigned there.

width The width of the grid data

3.3 moonpie.collections.iterators

Iterators provide a variety of functions for iterating over tables. They should work with any index based table.

The cycle iterator allows continuous looping over an array. It also provides the ability to move backwards through the list.

moonpie.collections.iterators.cycle(array_table, count) array_table = any table with an index list count = the limit of cycles to perform

```
local set = { 1, 2, 3, 4 }

for value, index in moonpie.collections.iterators.cycle(set, 2) do
  print(value)
end

--
-- Output:
--
-- 1
-- 2
-- 3
-- 4
-- 1
-- 2
-- 3
-- 4
```

```
local set = { 1, 2, 3, 4 }
local cycle_iter = moonpie.collections.iterators.cycle(set)
print(cycle_iter.previous()) -- Output: 4
print(cycle_iter.previous()) -- Output: 3
```

CHAPTER 4

moonpie.entities

An ECS style framework that works with moonpie state management

CHAPTER 5

moonpie.events

Events are triggered at various times during interactions with Love2d. These are designed to provide a way of looping in components and behaviors to certain timings without relying on a difficult to maintain call sequence.

Love2D is the ultimate trigger source for events and these do need to be passed manually into Moonpie. This allows the most control possible for engineering solutions while keeping code upstream clean.

5.1 Available Events

```
local function my_callback()
  print("called")
end

moonpie.events.beforeUpdate:add(my_callback)

function love.update()
  moonpie.update()
end

-- Output
  called
```

CHAPTER 6

moonpie.graphics

6.1 moonpie.graphics.colors

Provides access to a color library with hundreds of default colors. Also allows for certain functionality such as lightening colors or making gradients.

6.2 moonpie.graphics.font

Manages fonts making it easy to load and reuse font resources. Fonts can be referenced by a name that makes it easy to switch out the font without impacting code or ui elements.

```
local Font = require "moonpie.graphics.font"
Font:register("assets/fonts/my_font.ttf", "title")
local f = Font:get("title")
local text = love.graphics.newText(f, "Hello World")
love.graphics.draw(text, 20, 20)
```

6.3 moonpie.graphics.image

Provides functionality to access and manage images in a way that limits creating duplicate copies

moonpie.graphics.image.load(path)

CHAPTER 7

moonpie.math

A basic library for some math functions.

7.1 Functions

floor(...) Returns the floor value for a list of values. Useful for flooring return values from a function

line(x0, y0, x1, y1) Returns an iterator that will plot lines along the requested points.

```
local maths = require "moonpie.math"

for x, y in maths.line(1, 3, 8, 18) do
  print(x, y)  -- Output each x, y coordinate
end
```

7.2 moonpie.math.cards

Provides a basic implementation of a deck of cards with a Fisher-Yates shuffler to randomize the elements.

```
local maths = require "moonpie.math"
local deck = maths.cards.newDeck { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
deck:shuffle()
local hand = deck:deal(3)
-- hand == { 4, 9, 2 }
-- deck == { 5, 1, 3, 7, 6, 8 }
```

7.3 moonpie.math.ipoint

iPoint is a simple 3d coordinate point that locks to integer based coordinates. It attempts to be efficient by only having

7.4 moonpie.math.rectangle

Provides a basic implementation for rectangles with additional helpers

new(x,y,width,height) Returns a new rectangle with the specified dimensions

left(self) Returns the leftmost coordinate (x)

right(self) Returns the rightmost coordinate (x+width)

top(self) Returns the topmost coordinate (y)

bottom(self) Returns the bottommost coordinate (y+height)

[true] intersects(self, rect) Returns true if the two rectangles intersects

[rect] overlap(self, rect) Returns a new rectangle that is the overlapping region between 2 rectangles

[x,y] center(self) Returns the coordinates in the middle of the rectangle

CHAPTER 8

moonpie.state

An implementation of a concept similar to redux that is used in react and javascript implementations. This provides a store that can be configured with reducers that handle state. Actions can be dispatched to set up changing the the status of state.

8.1 Add Action Validator

A helper to append validations to actions. Sometimes there is value in reusing previous existing actions, but providing an additional method to validate the action.

8.2 Binding Components

A common issue is to bind components to state changes to refresh or respond to updated data. Sometimes components are not designed from the ground up to be connected to the store. Binding allows taking advantage of dynamic updates from the store to refresh the component.

```
local Components = require "moonpie.ui.components"
local c = bind(
  Components.text { text = "My Name" }, -- The created component to bind
  function(component, state) -- the binding routine
    -- Any logic and behavior could be applied here
    component:update { text = state.name }
  end)

  .
  .

  store.dispatch({ updateName = "foobar" })
```

CHAPTER 9

moonpie.sorts

CHAPTER 10

moonpie.tables

The tables utilities provides various mini-functions for helpful operations.

tables.count(set, func) Returns the count of items that match the comparison function passed in.

```
local set = { 1, 2, 3, 4, 5, 6 }
local compare = function(v) return v % 2 == 0 end

print(tables.count(set, compare))
-- 3
```

tables.deepCompare(tbl1, tbl2, ignoreMT) Tests the values in the the 2 tables to see if they look the same without having to be the same table instance.

tables.keysToList(tbl) Returns a table in array form where all the entries from tbl are outputted into a list formats.

tables.popRandom(list) Selects a random item out of the list, removes it and returns the selected item.

tables.slice(list, start, [end]) Returns a slice of the elements from the array. If end is not provided, defaults to end of array. If a negative number is passed into start, it takes from the end of the array.

tables.swap(tbl, i, j) Swaps two elements positions in the table

tables.take(tbl, count) Takes the specified count of elements from the front of the table. Because of the reshuffle of the table performance is not optimal for many operations during critical cycles.

tables.toString(tbl) Outputs a human readable form of the table. Useful for debugging purposes.

tables.unpack(tbl) The Lua unpack routine provided by the library but because sometimes it's based on table and sometimes global this just simplifies tracking it.

CHAPTER 11

moonpie.test_helpers

Test helpers and extra assertions are provided for the Busted <http://olivinelabs.com/busted/> testing framework.

11.1 New Assertions

callable(expectedCallable) Returns true if the value passed in is a callable table or a function.

```
assert.callable(function() end) -- true
assert.callable({}) -- false
assert.callable(setmetatable({}, { __call = function() end })) -- true
```

11.2 Array Extensions

array_includes(value, table, compare) Checks whether the value exists in the table. A custom comparison function can be provide to search for the value

```
it("has some array elements", function()
  local test = { 1, 2, 3, 4 }
  assert.array_includes(1, test)
end)
```

11.3 Component Extensions

```
-- Code to test
local components = require "moonpie.ui.components"
local my_comp = components("my_comp", function()
  return {
    components.text(),
```

(continues on next page)

(continued from previous page)

```
    components.text { id = "12345" }
  }
end)

it("contains a component", function()
  assert.contains_component("text", my_comp())
end)

it("contains a component with id", function()
  assert.contains_component_with_id("12345", my_comp())
end)
```

11.3.1 Matchers

Matchers are used when validating arguments to spies.

matches.in_range(low, high) Returns true if the value is in the range specified. Automatically sorts the low/high values by size when passed in.

11.3.2 Mock Store

Mocks the redux style store that manages state. Allowing easier testing of components that are dependent on the store.

```
describe("My test harness", function()
  local mock_store = require "moonpie.test_helpers.mock_store"
  local initial_state = { values = true }
  local store = mock_store(initial_state)

  it("can track dispatches", function()
    system_under_test.do_thing_that_dispatches()
    assert.equals(1, #store.get_actions("action_type"))
  end)
end)
```

11.3.3 General helpers

spy_to_func Converts a spy routine into a pure function. This can be helpful in situations where the code under test responds differently to tables vs functions.

CHAPTER 12

moonpie.ui

12.1 moonpie.ui.components

Components represent any kind of control or ui element. They are designed similar to React components. Each component should handle a specific demand on the UI. These components should be nested and reused as appropriate.

Default components are defined for very commonly used elements, but you should plan on extending the components with ones specific for your game.

For example, a possible hierarchy of components on a title screen:

- Title screen
  - Background Animation
  - Title
  - Menu
    - Button (New Game)
    - Button (Resume)
    - Button (Quit)

12.1.1 Defining Components

Components are defined by calling the components initializer and passing a name for the component and a function block that returns a table to represent a new instance of the component.

```
local components = require "moonpie.ui.components"
local widget = components("widget", function(props)
  return {
    -- Properties can be defined on the component
    styles = "custom-style",
    width = "75%",
```

(continues on next page)

(continued from previous page)

```
    -- This nests a child component within this component
    components.h1 { text = "Heading" },
    components.text { text = "Hello World!" },
  }
end)

moonpie.render("ui", widget()) -- sets the UI to render this component
```

12.1.2 Component Methods

These are methods that can be used or overridden to provide additional behavior for the UI

addStyle(self, style) Adds a new style tag to the component.

drawComponent(self) A method for executing custom drawing commands. Love will already be configured to translate to the appropriate x/y coordinates on the screen so all drawing commands should be assumed to start based on the top-left of the content area for the component.

findByID(self, id) Searches the component's child hierarchy to find the first matching identifier.

```
local found = component:findByID("sampleComponentID")
```

hide(self) Marks a component as hidden and removes from layout and rendering

isHidden(self) Returns true if a component is marked as hidden.

mounted(self) A method that is called when a component has been added to the render tree. Layout and other information will not be calculated at this point but the node should be aware of its place in the render tree.

remove(self) Flags the component to be removed from the render tree.

removeStyle(self, style) Removes a style tag from the component.

show(self) Marks a component as visible and will show up in layouts and rendering.

unmounted(self) A method called when a component is destroyed from the render tree. Used for any kind of global cleanup necessary when the component is removed that would be difficult for the garbage collector to know about. For example, global event handlers or lambdas.

Component Properties

data This can be used to pass in customized initialization data that will be stored in the component.

```
local c = Components.h1 { data = { a = "a", b = 3 } }
print(c.data.a) -- "a"
print(c.data.b) -- 3
```

logger Easy access to the logger library

Component Events

onUpdate(component, changes) Called whenever the component receives an update call.

```lua
local callbackRoutine = function(component, changes) print(changes.newValue) end
local c = component { onUpdate = callbackRoutine }
c:update({ newValue = "foo" })
-- prints "foo"
```

onMouseMove(component, x, y) Called whenever the mouse moves around over the component. *x*, *y* are screen coordinates

12.2 moonpie.ui.styles

Styles are a way of setting common properties that are easy to change across the site. These work similar to CSS in HTML though without the full selector behavior. Styles are applied directly to an element. When calculating values some properties do inherit from the parent to make it easier to specify items like fonts to be defaulted through.

12.2.1 Style Properties

display [inline, inline-block, block] Describes how the component should calculate its width. The main ones to use our `inline` and `block`. `block` is the default display setting, this will expand the component to the maximum width available. Determined by the parent. `inline` will size the component based on the width of the children.

textwrap specifies that whether text should wrap. Default behavior if nil is to wrap text. If set to 'none' will disable wrapping

12.3 Default Styles

12.3.1 Buttons

button-small Makes a smaller button for those tinier button needs

button-primary A style that uses the primary color for the background of the button

button-warning A style that uses a gold/yellow background color

button-danger A style that uses a red/fuschia background color

12.4 Built In Components

12.5 body

The *body* component defaults to a full screen component that uses the *background* color by default. This will create a clean empty background for the rest of the components to render upon. The only custom parameter takes the contents to render.

12.5.1 Properties

contents A table that will be rendered out within the body

Example

```
local Components = require "moonpie.ui.components"

local body = Components.body {
  content = {
    -- custom screen elements
  }
}
```

12.6 image

12.6.1 Properties

source The path to the image to be loaded

12.7 textbox

12.7.1 Methods

getText(self) Returns the text currently in the text box

setText(self, value, skipUpdateCursor) Sets the text within the textbox to the specific value. By default, the cursor will move to the end of the string, passing true to skipUpdateCursor will bypass this.

CHAPTER 13

moonpie.utility

ensureKey(tbl, key, default) Makes sure that a table contains a key with a default value. Useful for state management in the store to make sure that values exist

isCallable(val) Returns true if the parameter is either a function or a table with a metatable that implements __call.

```
local utility = require "moonpie.utility"
utility.isCallable(function() end) -- true
utility.isCallable({}) -- false
utility.isCallable(setmetatable({}, { __call = function() end })) -- true
```

swapFunction(tbl, functionName, override) Replaces a table function with a new routine. This is most useful for testing scenarios to mock an API. `:revert()` can be used to unwind the swapped function.

```
local utility = require "moonpie.utility
local tbl = { f = function() end }
local new = function() end
utility.swapFunction(tbl, "f", new)
tbl:f() -- calls new(tbl)
```

CHAPTER 14

Indices and tables

- genindex
- modindex
- search