

Let's code Lua!

Dein Einstieg in die Spieleprogrammierung

» Hier geht's
direkt
zum Buch

DIE LESEPROBE

Kapitel 4

Daten, Daten, Daten

Daten sind der Rohstoff, mit dem Computerprogramme arbeiten. Sei es der Punktestand, dein Level oder die Dialoge in deinem Game: Zahlen und Text sind die Elemente, aus dem unsere Programme ihren Output generieren. In diesem Kapitel lernst du den Umgang mit diesen Elementen: wie du sie speicherst und wie du sie mit Rechenoperationen kombinierst.

In diesem Kapitel wirst du einige Grundlagen der Programmierung erlernen. Wir werden in diesem Kapitel noch nicht in der Lage sein, ein echtes Game zu programmieren, das wird sich aber bald ändern.

Wie sehen Lua-Programme aus?

Öffne noch mal die Datei *HalloWelt.lua*, die du im letzten Kapitel angelegt hast, und füge eine weitere Zeile hinzu:

```
print("Hallo Welt!")  
  
print("Auf Wiedersehen Welt!")
```

Listing 4.1 Hallo und tschüs Welt!

Wenn du das Programm ausführst, sollte Folgendes erscheinen:

```
C:\Users\LetsCodeLua\Documents\LuaPrograms>lua HalloWelt.lua  
Hallo Welt!  
Auf Wiedersehen Welt!
```

Allgemein gesprochen, sind Lua-Programme eine Folge von Anweisungen an den Computer. Eine Anweisung kann sein, dass der Computer etwas ausrechnen soll, eine Eingabe von der Tastatur einlesen soll oder, wie in diesem Beispiel, etwas auf dem Bildschirm ausgeben soll.

So eine Anweisung nennt man in der Fachsprache auch ein *Statement*. In der Regel steht jede Anweisung in einer eigenen Zeile im Programm. Das Programm in Listing 4.1 besteht aus zwei Anweisungen. Leere Zeilen werden vom Lua-Interpreter ignoriert. Sie dienen nur der Lesbarkeit des Programms. Und auch bei Leerzeichen ist Lua tolerant. Wo ein Leerzeichen erlaubt ist, sind auch mehrere Leerzeichen oder sogar Zeilenwechsel erlaubt. (Diese »unsichtbaren« Zeichen nennt man in der Programmierung *White-space*.) Man könnte das obere Programm also auch so schreiben:

```
    print(
"Hallo Welt!"

) print(  "Auf Wiedersehen Welt!"
)
```

Listing 4.2 Hallo und tschüs, verwirrte Welt!

Dem Lua-Interpreter ist das zwar egal, aber dem Menschen, der deinen Code liest, nicht. Stell dir vor, du möchtest später in einem längeren Programm eine Änderung vornehmen, beispielsweise die Anzahl der Leben oder den Namen deines Hauptcharakters ändern. Je lesbarer der Code, desto leichter wirst du die relevante Stelle finden. Bemühe dich also, Leerzeichen und Leerzeilen so zu verwenden, dass sie die Lesbarkeit deines Codes erhöhen.

Kommentare

Manchmal kann es sinnvoll sein, in deinen Programmen kleine Notizen zu hinterlassen. Vielleicht arbeitest du mit einem Freund zusammen an deinem Code, und die nächsten Zeilen sind so kompliziert, dass es besser ist, eine Erinnerung dazu zu schreiben, wie sie funktionieren.

Solche Notizen nennt man *Kommentare*. In Lua ist alles, was nach einem doppelten Minuszeichen (--) kommt, bis zum Ende der Zeile ein Kommentar. Kommentare werden vom Lua-Interpreter einfach ignoriert.

```
-- Dieses Programm gibt zwei Zeilen Text aus.
print("Hallo Welt!")          --> Hallo Welt!

print("Auf Wiedersehen Welt!") --> Auf Wiedersehen Welt!
```

Listing 4.3 Ein Programm mit drei Kommentaren

Übrigens: Wenn in diesem Buch hinter einer Programmzeile ein Kommentar steht, der mit einem `-->` anfängt, dann siehst du dort, was das Programm in dieser Zeile ausgibt. So musst du nicht jedes Programm extra eintippen.

Wenn du mal einen längeren Kommentar schreiben willst, der über mehrere Zeilen geht, kannst du mit `--[[` einen solchen Kommentar beginnen. Ist der Kommentar fertig, beendest du ihn mit `]]` und schreibst ganz normal deinen Code weiter.

```
--[[
    Längere Kommentare können zum Beispiel hilfreich sein, um am Anfang deines
    Codes eine kurze Beschreibung zu geben. Jemand, der deinen Code liest, hat
    dann sofort einen Überblick.
]]
```

Listing 4.4 Ein mehrzeiliger Kommentar

Variablen und Ausdrücke

Stell dir vor, du möchtest ein Spiel programmieren, in dem der Held gegen Orks, Ratten und Fledermäuse kämpfen muss. Der Spieler hat 7 Orks, 11 Ratten und 13 Fledermäuse getötet. Getötete Orks bringen je 100 Punkte und getötete Ratten und Fledermäuse je 10 Punkte. Diese Punktzahl willst du nun ausrechnen und in einer *Variablen* mit dem Namen `score` speichern. Eine Variable ist so etwas wie eine Kiste, in der wir Werte zwischenspeichern können.

Um eine Variable zu benutzen, müssen wir sie erst deklarieren. Das heißt, wir müssen dem Lua-Interpreter mitteilen, dass er für uns eine Kiste mit dem Namen `score` anlegen soll. Das geht in Lua so:

```
local score
```

Listing 4.5 Hallo Lua. Wir hätten gern eine Variable mit dem Namen »score«.

Jetzt, da die Variable deklariert ist, können wir sie mit Inhalt füllen und sie uns anschließend mit `print()` ausgeben lassen:

```
-- Punkte für 7 Orks, 11 Ratten und 13 Fledermäuse
score = 7 * 100 + (11 + 13) * 10
print(score)    --> 940

-- Noch ein Ork getötet ...
score = score + 100
print(score)    --> 1040
```

Listing 4.6 Nur ein toter Ork ist ein guter Ork.

Sicher hast du schon eine ungefähre Vorstellung davon, was im obigen Code passiert. Trotzdem lohnt es sich, diese Zeile noch mal genau unter die Lupe zu nehmen:

Variable	Zuweisungszeichen	Ausdruck
score	=	$7 * 100 + (11 + 13) * 10$

Links vom Gleichheitszeichen steht unsere *Variable*.

Zuweisungen werden in Lua mit dem Gleichheitszeichen ausgedrückt. Das ist etwas irreführend, weil die Zuweisung wenig mit der Gleichheit aus der Mathematik zu tun hat. Sieh dir zum Beispiel Zeile 5 an: `score = score + 100`. Mathematisch gesehen, ist das einfach falsch. In Lua bedeutet es aber: Nimm die Variable `score`, addiere 100 und speichere das Ergebnis wieder in der Variablen `score`. Leider hat sich das Gleichheitszeichen für Zuweisungen in der Programmierung eingebürgert.

Rechts vom Zuweisungszeichen steht ein *Ausdruck*. Ein Ausdruck ist eine Rechnung, bei der hinterher ein Wert herauskommt, den wir in einer Variablen speichern können.

Vermutlich hast du es dir schon gedacht: Mit dem `*` in dem Ausdruck oben kannst du Zahlen miteinander malnehmen. Zum Teilen müsstest du ein `/` verwenden. Lua beachtet übrigens die Regel »Punkt- vor Strichrechnung«.

Deklaration und Zuweisung

Übrigens können wir Deklaration und Zuweisung auch miteinander kombinieren. Anstelle von

```
local score
score = 7 * 100 + (11 + 13) * 10
```

hättest du also auch schreiben können:

```
local score = 7 * 100 + (11 + 13) * 10
```

Das Ergebnis ist genau dasselbe, aber wir sparen etwas Platz, wenn wir Deklaration und Zuweisung in dieselbe Zeile schreiben. Welche Variante du besser findest, bleibt dir überlassen.

Wieso »local«?

Tatsächlich kannst du in Lua auch Variablen benutzen, ohne sie vorher mit `local` zu deklarieren. Lua geht dann davon aus, dass es sich um eine *globale Variable* handelt. Weil die Verwendung von globalen Variablen aber zu unerwarteten Fehlern in deinem Code führen kann, solltest du so oft wie möglich *lokale Variablen* benutzen – also solche, die du mit `local` deklarierst.

Was es genau mit lokalen und globalen Variablen auf sich hat, erfährst du in Kapitel 7, »Entscheidungen treffen«.

Variablenamen

Die Variable im vorherigen Listing hatte den Namen `score`. Grundsätzlich darfst du deinen Variablen jeden beliebigen Namen geben, allerdings gelten dabei ein paar Regeln: Variablenamen dürfen in Lua aus Groß- und Kleinbuchstaben, Zahlen und dem Unterstrich (`_`) bestehen. Sie dürfen allerdings nicht mit einer Zahl anfangen. In der Regel solltest du einen Variablenamen verwenden, der gut beschreibt, was in der Variablen gespeichert wird.

```
-- Gültige Variablenamen
local killedOrcs = 11
local getoeteteOrks = 11
local POINTS_PER_RAT = 10
local weaponPower1 = 170
local weaponPower2 = 310
local b = weaponPower1 + weaponPower2
local B = b * b

-- Ungültige Variablenamen
local getöteteOrks = 1      -- keine Umlaute oder andere Sonderzeichen
local POINTS PER RAT = 10  -- keine Leerzeichen
local 2orks                -- nicht mit einer Zahl anfangen
```

Listing 4.7 Es gibt viele Möglichkeiten, eine Variable zu benennen.

Game: Ein Höhlenabenteurer



Nach einigen Wochen der Reise erreichte Marvin einen Ort namens Ka-ieh. Dieser war dafür bekannt, dass dort eine außergewöhnlich hohe Anzahl von Menschen wohnten, denen hellseherische Kräfte zugeschrieben wurden. Auch Magier sollten dort leben. Ein Ort, der schräg, aber friedlich ist. Vor dem Ortseingang sah Marvin eine Höhle, die in einen Hügel hineinzugehen schien. Davor saß eine alte Frau.

Das Game

Das nächste Spiel ist ein kleines Abenteuer. Die Spielerinnen und Spieler werden vor verschiedene Entscheidungen gestellt. Je nachdem, wofür sie sich entscheiden, geht die Handlung unterschiedlich weiter.

Man nennt das ein *Text-Adventure* oder manchmal auch *Interactive Fiction*, was frei übersetzt so etwas wie »interaktiver Roman« bedeutet. Dieses Genre gibt es seit den 1970ern, und bis heute lebt es in Form von *Visual Novels* weiter.

Frau: "Ah, du bist genau richtig hier."

Marvin: "Warum sagst du das?"

```
Frau: "Ich spuere so etwas einfach. Komm in die Hoehle, es wird
sich fuer dich lohnen."
```

```
1 - "Okay."
```

```
2 - "Aeh, nee, danke."
```

```
>
```

Listing

Zu Beginn wird Marvin von der alten Frau angesprochen und kann sich für eine von zwei Optionen entscheiden. Achte darauf, dass wir hier häufig einfache Anführungszeichen benutzen, weil im Text doppelte Anführungszeichen vorkommen!

```
io.write('Frau: "Ah, du bist genau richtig hier."\n')
io.write('Marvin: "Warum sagst du das?"\n')
io.write('Frau: "Ich spuere so etwas einfach. Komm in die Hoehle, es wird\n')
io.write('sich fuer dich lohnen."\n')
io.write("\n")
io.write('1 - "Okay."\n')
io.write('2 - "Aeh, nee, danke."\n')
io.write("\n")

io.write("> ")
local input = io.read()
io.write("\n")
```

Wenn er Antwort zwei gewählt hat, wird ein entsprechender Text ausgegeben. Wichtig ist, dass wir als Bedingung `input == "2"` schreiben und *nicht* `input == 2`, denn `io.read()` gibt immer einen String zurück.

```
if input == "2" then
    io.write("Marvin geht an der Frau vorbei. Ihn beschleicht das komische\n")
    io.write("Gefuehl, eine wichtige Gelegenheit verpasst zu haben...\n")
end
```

Die einzige Antwort, die den Spieler weiterführt, ist Antwort eins. Jede andere Antwort führt zur Beendigung des Programms.

```

if input ~= "1" then
    os.exit()
end

```

Die nächste Entscheidung steht an:

```

io.write("Marvin betritt die daemmerige Hoehle, vor ihm sind drei Tueren.\n")
io.write('Ueber jeder der Tueren steht ein Wort: "ENNOS", "DNOM", "ENRETS".\n')
io.write("Welche der Tueren soll Marvin nehmen?\n")
io.write("\n")
io.write('1 - Die Tuer "ENNOS"\n')
io.write('2 - Die Tuer "DNOM"\n')
io.write('3 - Die Tuer "ENRETS"\n')
io.write("\n")

io.write("> ")
input = io.read()
io.write("\n")

```

```

if input == "1" then
    io.write("Marvin durchschreitet einen Gang. Zeichnungen von Kreisen, von\n")
    io.write("denen Linien ausgehen, schmuecken die Wand. Nach einiger Zeit\n")
    io.write("tritt er wieder ans Tageslicht. Die Sonne scheint ihn an.\n")
    io.write("Irgendwie hat er den Eindruck, dass dieser Gang nicht fuer ihn\n")
    io.write("bestimmt war.\n")

```

```

elseif input == "3" then
    io.write("Marvin betritt einen langen Gang. Lichtpunkte finden sich an\n")
    io.write("der Decke. Der Gang zieht sich, und die Sterne leuchten schon\n")
    io.write("am Himmel, als er wieder ins Freie kommt. Irgendwie hat Marvin\n")
    io.write("den Eindruck, dass dieser Gang nicht der richtige fuer ihn\n")
    io.write("war.\n")

```

```
end
```

```

if input ~= "2" then
    os.exit()
end

```

Wenn der Spieler bis hierhin richtig gewählt hat, wird er vor das finale Rätsel gestellt:

```
io.write("Marvin betritt einen Raum. Aus einem Loch in der Decke tritt ein\n")
io.write("wenig Licht und beleuchtet einen kreisrunden Stein. An den Waenden\n")
io.write("erkennt er schwach geometrische Zeichnungen von Sichel und\n")
io.write("Kreisen, die einem ihm unverständlichen System zu folgen\n")
io.write("scheinen. Eine innere Stimme lässt ihn nach dem Stein greifen.\n")
io.write("Er fühlt sich gut an. Marvin verlässt die Höhle wieder.\n")
io.write("\n")
io.write('Frau: "Ich sehe, du hast den Stein gefunden. Behalte ihn, er war\n')
io.write("für dich bestimmt. Nun weißt du, welches Gestirn dich auf deinen\n")
io.write('Reisen beschützen wird."\n')
io.write("\n")
io.write('Haeh?', denkt Marvin.\n')
io.write("\n")

io.write("(Welches Gestirn ist gemeint?) > ")
input = io.read()
io.write("\n")
```

Wir wollen jede Antwort gelten lassen, die das Wort »Mond« enthält, egal ob der Nutzer »Mond«, »der Mond« oder »ES IST DER MOND!« eingibt. Dazu wandeln wir die Antwort erst mal in Kleinbuchstaben und prüfen, dann mit `string.find()`, ob sie den String "mond" enthält. Erinnerung: `string.find()` nur dann `nil` zurückgibt, wenn der String nicht gefunden wurde! Das heißt, wenn `string.find()` etwas anderes als `nil` zurückgibt, ist der String in der Antwort enthalten.

```
input = string.lower(input)

if string.find(input, "mond") ~= nil then
    io.write("Ein zufriedenes Gefühl macht sich in Marvin breit. Er blickt\n")
    io.write("in den Himmel, und tatsächlich hat er das Gefühl, dass der\n")
    io.write("ruhige Mond über ihn wacht. Gestärkt setzt er seine Reise\n")
    io.write("fort.\n")
else
    io.write('Komische Alte', denkt Marvin, "keine Ahnung, wovon sie\n")
    io.write('spricht."\n')
    io.write("Froh setzt seine Reise fort.\n")
end
```

Charaktere und Animation

Als nächstes wenden wir uns der Ausgabe der Charaktere zu. Alle Löve-Funktionen, die dafür nötig sind, hast du bereits kennengelernt. Später wollen wir die Charaktere animieren. Sie sollen sich ein wenig bewegen, auch wenn der Spieler keine Eingabe macht. So kann man die Charaktere besser vom Hintergrund unterscheiden und das Spiel wirkt viel lebendiger. Aber alles der Reihe nach ...

Spritesheets und Sprites

Für die Levels deines Games gibt es Tilesets. Für die Charaktere eines 2D-Games benutzt man eine ähnliche Technik. Allerdings spricht man hier anstelle von einem Tileset von einem *Spritesheet*. Ein einzelner Teil des Spritesheets nennt sich *Sprite*. Dieses Spritesheet stammt wieder vom Nutzer Pipoya auf *itch.io*. Pipoya hat mir freundlicherweise erlaubt, seine Arbeit zu verwenden. Du findest das Spritesheet im *assets*-Ordner des Projekts unter *graphics/Male 16-2.png*.¹



Abbildung 17.6 Das Spritesheet für Marvin

In diesem Fall haben die einzelnen Sprites eine Größe von 32×32 Pixeln – genau wie die Tiles, die wir für die Map verwendet haben.

¹ <https://pipoya.itch.io/pipoya-free-rpg-character-sprites-32x32>

Die Charakter-Objekte lernen, sich zu malen

Die Charakter-Objekte können schon prüfen, ob sie noch lebendig sind (`char:isDead()`), eine kurze Beschreibung von sich zurückgeben (`char:description()`) und einen anderen Charakter angreifen (`char:attack()`). Jetzt lernen sie noch, sich selbst auf den Bildschirm zu malen. Dafür wollen wir sie um eine `draw()`-Methode erweitern.

Aber wie schon erwähnt: Das Laden der Grafiken sollte am besten nur einmal passieren und nicht jedes Mal, wenn wir einen Charakter ausgeben, weil es relativ zeitaufwendig ist. Daher schreiben wir zuerst eine `load()`-Funktion für unser `characterFactory`-Modul, in der wir die Grafiken laden, die wir brauchen, und die passenden Quads für unser Sprite-sheet anlegen:

```
local util = require("util")
local statusBar = require("statusBar")

local spriteSheetMarvin, spriteSheetsEnemies
local spriteQuads = {}

local tileset, corpseQuad

local function load()
  -- Wir laden das Spritesheet für Marvin.
  spriteSheetMarvin = love.graphics.newImage("assets/graphics/Male 16-2.png")
  -- Für die Dämonen gibt es sieben verschiedene Spritesheets, die
  -- wir in einem Array speichern.
  spriteSheetsEnemies = {
    love.graphics.newImage("assets/graphics/Enemy 01-1.png"),
    love.graphics.newImage("assets/graphics/Enemy 02-1.png"),
    love.graphics.newImage("assets/graphics/Enemy 03-1.png"),
    love.graphics.newImage("assets/graphics/Enemy 04-1.png"),
    love.graphics.newImage("assets/graphics/Enemy 05-1.png"),
    love.graphics.newImage("assets/graphics/Enemy 06-1.png"),
    love.graphics.newImage("assets/graphics/Enemy 07-1.png")
  }

  -- Die Quads speichern wir diesmal in einem zweidimensionalen Array.
  -- spriteQuad[2][1] soll das Sprite in der 2. Zeile und 1. Spalte enthalten.
  for i = 1, 4 do
    spriteQuads[i] = {}
    for j = 1, 3 do
```

```

        local quad = love.graphics.newQuad(
            (j - 1) * 32,
            (i - 1) * 32,
            32,
            32,
            spriteSheetMarvin
        )
        spriteQuads[i][j] = quad
    end
end

-- Um die Überreste besiegter Charaktere auszugeben, benutzen wir eine
-- Grafik aus dem Tileset.
tileset = love.graphics.newImage("assets/graphics/tileset_dungeon.png")
corpseQuad = love.graphics.newQuad(7 * 32, 2 * 32, 32, 32, tileset)
end
...
return {
    load = load,
    createMarvin = createMarvin,
    createDemon = createDemon
}

```

Listing 17.11 Laden der Spritesheets für Marvin und die Dämonen (characterFactory.lua)

Weil die Spritesheets alle die gleiche Struktur haben, müssen wir die Quads nur einmal anlegen. Wir können dieselben Quads dann für alle Spritesheets verwenden. Anstatt die Quads wie bei den Tiles in einem langen Array anzulegen, benutzen wir diesmal ein zweidimensionales Array. So können wir mit `spriteQuad[2][3]` das Sprite ausgeben, das sich in der zweiten Zeile und der dritten Reihe befindet. Das wird später noch nützlich sein.

Jetzt müssen wir noch dafür sorgen, dass die `load()`-Funktion auch aufgerufen wird. Weil es das `gameArea`-Modul ist, das sich um die Verwaltung der Charaktere kümmert, ist die `load()`-Funktion des `gameArea`-Moduls ein guter Ort dafür:

```

...
local function load()
    characterFactory.load()
    tileset = love.graphics.newImage("assets/graphics/tileset_dungeon.png")

```

```

...
end
...

```

Listing 17.12 Bevor die Grafiken für das Spielfeld geladen werden, werden die Grafiken für die Charaktere geladen. (gameArea.lua)

Jetzt sind wir gut vorbereitet, um das Charakter-Objekt zu erweitern. Jedes Objekt bekommt als zusätzliche Eigenschaft ein Spritesheet zugewiesen (mit allen Grafiken für diesen Charakter). Außerdem wird das Objekt um eine `draw()`-Methode erweitert:

```

local function createCharacter(name, level, totalHp, symbol, x, y, spriteSheet)
  return {
    ...
    x = x,
    y = y,

    spriteSheet = spriteSheet,

    draw = function(self, x, y)
      if self:isDead() then
        -- Wenn der Charakter tot ist, male die Gebeine aus dem Tileset.
        love.graphics.draw(tileset, corpseQuad, x, y)
      else
        -- Sonst malen wir den Sprite oben in der Mitte des Spritesheets.
        local quad = spriteQuads[1][2]
        love.graphics.draw(self.spriteSheet, quad, x, y)
      end
    end,

    isDead = function(self)
      return self.hp <= 0
    end,

    ...
  }
end

local function createDemon(level, x, y)
  ...

```

```

-- Ein zufälliges Spritesheet wird ausgewählt.
local spriteSheet = spriteSheetsEnemies[math.random(#spriteSheetsEnemies)]

return createCharacter(name, level, totalHp, symbol, x, y, spriteSheet)
end

local function createMarvin (x, y)
-- Marvin hat Level 1 und 100 HP. Auf dem Spielfeld erscheint er als "@".
local marvin = createCharacter("Marvin", 1, 100, "@", x, y, spriteSheetMarvin)

...

return marvin
end

```

Listing 17.13 Unser Charakter-Objekt kann sich jetzt selbst ausgeben.

Diese `draw()`-Methode müssen wir nun nur noch beim Malen des Spielfelds aufrufen:

```

local function draw()
for y = 1, getMapHeight() do
for x = 1, getMapWidth() do
local tile = getTile(x, y)
local pixelX, pixelY = computeCoordinates(x, y)

love.graphics.draw(tileset, tileQuads[tile], pixelX, pixelY)

-- Wenn sich (mindestens) ein toter Charakter auf dem Feld befindet,
-- rufen wir die draw()-Methode auf, damit die Gebeine gemalt werden.
local deadChar = findDeadCharacter(x, y)
if deadChar ~= nil then
deadChar:draw(pixelX, pixelY)
end

-- Wenn sich ein lebender Charakter auf dem Feld befindet, wird er
-- gemalt.
local char = findLivingCharacter(x, y)
if char ~= nil then
char:draw(pixelX, pixelY)
end
end
end
end

```

```

        end
    end

    local pixelX, pixelY = computeCoordinates(2, 8)
    love.graphics.draw(oldManSpriteSheet, oldManQuad, pixelX, pixelY)
end

```

Listing 17.14 Beim Malen des Spielfelds werden auch die Charaktere gemalt bzw. ihre Gebeine. (gameArea.lua)

Übrigens ist die Reihenfolge, in der du die verschiedenen Elemente des Spielfelds auf den Bildschirm malst, entscheidend: Immer wenn du etwas ausgibst, wird alles, was darunterliegt, »übermalt«. Du musst also erst die Dinge malen, die im Hintergrund erscheinen sollen, und später die Dinge, die vorne erscheinen sollen.

Wenn alles funktioniert hat, solltest du jetzt eine spielbare Version von *Cave of Doom* mit allen Charakteren haben, die in etwa so aussieht:



Abbildung 17.7 »Cave of Doom« mit allen Charakteren