Quick answers to common problems

Lua Game Development Cookbook

Over 70 recipes that will help you master the elements and best practices required to build a modern game engine using Lua

Mário Kašuba

[PACKT] open source*
PUBLISHING
community experience distilled

Lua Game Development Cookbook

Over 70 recipes that will help you master the elements and best practices required to build a modern game engine using Lua

Mário Kašuba

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Lua Game Development Cookbook

First published: July 2015

Production reference: 1200715

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-550-4

`www.packtpub.com`

Credits

Author
Mário Kašuba

Reviewers
Victor Andrade de Oliveira
Anthony Zhang

Commissioning Editor
Sarah Cullington

Acquisition Editor
Kevin Colaco

Content Development Editor
Govindan K

Technical Editor
Vivek Arora

Copy Editors
Merilyn Pereira
Laxmi Subramanian

Project Coordinator
Sanjeet Rao

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Jason Monteiro
Abhinash Sahu

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Mário Kašuba achieved a master's degree in applied informatics at Slovak Technical University in Bratislava, where he used the Lua language in 3D robotics simulations and developed various multimedia applications along with a few computer games.

Currently, he is the co-owner and chief information officer of an IT Academy company, while he also leads courses on C/C++, PHP, Linux, Databases, Typo3, Silverstripe CMS, VMware virtualization, and the Microsoft Windows Server operating system.

He also works as the head web developer and system administrator for the web portal `http://www.rodinka.sk/`.

I would like to acknowledge the support provided by my family, friends, and colleagues. I am particularly grateful to the members of the Lua users community who helped me with the reviewing process. I would also like to thank the staff at Packt Publishing for their guidance and valuable support.

About the Reviewer

Victor Andrade de Oliveira is a Brazilian computer engineer who graduated from the Institute for Higher Studies of the Amazon (IESAM) with a vast knowledge of the Lua language and has worked for more than 5 years in the development of interactive and embedded applications for Ginga—the middleware of the Japanese-Brazilian Digital TV System (ISDB-TB) and ITU-T Recommendation for IPTV services.

Anthony Zhang is a programmer, electronics hobbyist, and digital artist who has an unhealthy obsession with robotics. If you want to hear him talk for hours on end, ask him something about AI, physical computing, obscure processor functionality, and computer graphics. These days, you'll find him working on video games, doing ridiculous things with microcontrollers, and attempting to add LEDs where they don't belong.

Table of Contents

Preface

Game development is one of the most complex processes in the world as it requires a wide set of skills such as programming, math, physics, art, sound engineering, management, marketing, and many more. Even with modern technologies, it may take from a few hours to several years to create a game. This depends on the game complexity and tools available.

Computer games are usually based on a mix of simple concepts, which are turned into an enjoyable experience. The first step in making a good game is a game prototype. These can be made with the help of various game engines. However, learning how to use a game engine to the full extent may require you to study how it actually works. This way you have to rely on the available documentation and features that the game engine provides. Many game engines today provide a scripting language as a tool to implement certain game mechanics or to extend the game engine itself with new features.

The Lua programming language is gaining popularity in the game industry mainly due to its simplicity and efficiency. Most of the time, it's used only for simple tasks such as NPC dialogs, user interface, or custom game events. However, with additional Lua modules, you can create your own full-fledged game engine that can use almost all the capabilities of the modern computer hardware.

In this book, you'll find a set of recipes with solutions to the most common problems you may encounter while creating games with the Lua language.

The best way to learn something is to play with it. Therefore, each recipe is paired with simple demo applications that will help you understand the topic covered. You may even use these demo samples to create your own game prototype in no time.

All sample applications are available in the digital content of this book.

What this book covers

Chapter 1, *Basics of the Game Engine*, covers important algorithms and the basic design of a game engine written in the Lua programming language, as well as LuaSDL multimedia module preparation, which is the main part of all the recipes in this book.

Chapter 2, *Events*, deals with handling input events that are an important part of any game engine.

Chapter 3, *Graphics – Common Methods*, contains basic concepts used in computer graphics. You'll learn how to initialize the graphics mode, use basic OpenGL functions, load images, create textures, and draw text on the screen.

Chapter 4, *Graphics – Legacy Method with OpenGL 1.x-2.1*, explains how to use the intermediate mode of OpenGL, which is intended for use on older GPUs. Even when this mode is currently deprecated, it holds important information that is vital when using modern versions of OpenGL. It may be used as a precursor to more advanced topics in *Chapter 5*, *Graphics – Modern Method with OpenGL 3.0+*.

Chapter 5, *Graphics – Modern Method with OpenGL 3.0+*, covers the basics of using the GLSL shading language with the Lua language to draw various scenes. You'll also learn how to use per-pixel lighting, render into textures and apply surface effects with normal maps.

Chapter 6, *The User Interface*, covers the implementation of the custom user interface from simple windows to window controls.

Chapter 7, *Physics and Game Mechanics*, explains how to prepare and use the LuaBox2D module with the Lua language for physics simulation. The Box2D library is quite popular in modern side-scrolling games mainly because it offers great flexibility.

Chapter 8, *Artificial Intelligence*, deals with pathfinding algorithms and fuzzy logic. You'll learn how pathfinding works in games with simple maze or even tiled environments. More advanced topics cover decision making with fuzzy logic. In combination with pathfinding algorithms, you can create intelligent game opponents that won't jump into a lava lake at the first opportunity.

Chapter 9, *Sounds and Networking*, covers how to initialize the sound card, play sounds, and music. The second part covers network communication with the high-performance ZeroMQ library. It contains many improvements over traditional socket communication and it's used by companies such as AT&T, Cisco, EA, Zynga, Spotify, NASA, Microsoft, and CERN.

What you need for this book

Sample demonstrations for recipes require the Microsofts Windows or Linux operating systems. Unfortunately, Mac OS is not currently supported.

If you intend to build binary Lua modules from this book, you'll need the C/C++ compiler along with the recent version of the CMake building system. Additionally, Linux users will need to install development packages for the XOrg display server in order to include the graphical output.

However, it's not necessary to do so as binary Lua modules are included in code files for this book.

The recipes in *Chapter 5*, *Graphics – Modern Method with OpenGL 3.0+*, require a graphic card released after 2010 with support for OpenGL 3.3.

Who this book is for

This book is for all programmers and game enthusiasts who want to stop dreaming about creating a game, and actually create one from scratch.

The reader should know the basics of programming and using the Lua language. Knowledge of the C/C++ programming language is not necessary, but it's strongly recommended in order to write custom Lua modules extending game engine capabilities or to rewrite parts of the Lua code into a more efficient form.

Algebra and matrix operations are required in order to understand advanced topics in *Chapter 4*, *Graphics – Legacy Method with OpenGL 1.x-2.1* and *Chapter 5*, *Graphics – Modern Method with OpenGL 3.0+*.

Sample demonstrations are coupled with binary libraries for Windows and Linux operating systems for convenience.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:
"We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
local sum_of_numbers = 0
local iterations = 3
for i=1,iterations do
  sum_of_numbers = sum_of_numbers + 1/iterations
  print(("%f"):format(sum_of_numbers))
end
-- is the result equal to 1?
print("Sum equals to 1?", sum_of_numbers == 1)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
gl.Begin(gl_enum.GL_LINES)
  -- A
  gl.Color4f(1, 0, 0, 1)
  gl.Vertex3f(-0.5, -0.5, 0)
  -- B
  gl.Color4f(0, 1, 0, 1)
  gl.Vertex3f(0.5, -0.5, 0)
  -- C
  gl.Color4f(0, 0, 1, 1)
  gl.Vertex3f(0.5, 0.5, 0)
  -- D
  gl.Color4f(1, 1, 0, 1)
  gl.Vertex3f(-0.5, 0.5, 0)
gl.End()
```

Any command-line input or output is written as follows:

```
mkdir luagl/build
cd luagl/build
cmake ..
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You can validate settings by pressing the **Configure** button"

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Basics of the Game Engine

In this chapter, we will cover the following recipes:

- Preparing a basic file structure for the game engine
- Making a stack
- Making a queue
- Making a prioritized queue
- Extending ipairs for use in sparse arrays
- Creating Lua modules
- Handling errors with pcall, xpcall, and assert
- Using Lua with existing projects written in C/C++
- Getting LuaSDL for libSDL 1.2
- Designing the main application loop with LuaSDL

Introduction

Almost every game uses a game engine to help developers in video game production. It is usually used as a base platform for the game and manages all important functions from 2D/3D graphics, physics, sound effects, and network communication to artificial intelligence, scripting, and support for various software/hardware platforms. Using the scripting language in games has gained a lot of attention in the last decade mainly because it allows you to create game prototypes faster, easier, and it's an important part of the so-called **modding** support for the game community.

For instance, you can look at *Quake* game from the id software company, which uses its own scripting language Quake C and is one of the reasons why there are so many mods for this game. However, the source code for this language must be compiled before using. Depending on the project size this means a significant amount of time spent between feature implementation and testing. The Lua language can be used without prior compilation, which allows developers to test out their code right away.

The first game that used the Lua language for scripting was *Grim Fandango* from the company LucasArts. It was successfully used in further major game titles and today it can be commonly found in many multiplatform and mobile games.

Modern game engines are one of the most complex applications that are used. This leads to the situation where game developers use game engine as a black box without knowing how it actually works. For certain game titles, this might work out quite well. However, if you want to make a quick game prototype with certain features that are not present in the game engine, you'll most probably have to write your own game engine extension or find a workaround.

Lua language is fast and mature enough to be used as a game engine base language. Time-critical portions of a code can be written in C or C++ language and accessed via a Lua/C language interface. With this approach, you can view the Lua language as a high-level glue for the game engine process design.

This book will use Lua 5.1 version mainly because it's well supported and the existing code can be ported into a newer version with minor changes. Another reason behind this choice is that Lua 5.1 API is used in LuaJIT which is Just-In-Time implementation of Lua language. It's generally regarded as a faster version of the Lua language, which gives you speed comparable to compiled C code.

Lua language itself doesn't provide access to graphical output, sound devices or even input devices except basic I/O file interface. This shortcoming can be overcome with the use of LuaSDL binding to libSDL multimedia library that gives us the power to access all the devices needed to create a game with graphics and sounds. Installation and use of this library binding will be contained in this chapter.

It's always good practice to maintain a consistent file structure in any project. The Lua language doesn't formally specify modular structure and it's often the case when each module uses its own style of module specification. This results in namespace conflicts and unpredictable behavior.

The first part of this chapter will cover the preparation of a modular file structure for your application, implementation of the most common data structures, and error handling.

The second half of this chapter will deal with more advanced stuff such as writing and using the Lua modules and using libSDL multimedia library to develop interactive applications in Lua.

Preparing a basic file structure for the game engine

When programming a larger project, it's always important to keep it maintainable. One of the common practices is to keep a modular structure. Modular structure can be achieved by keeping files separated in certain directories.

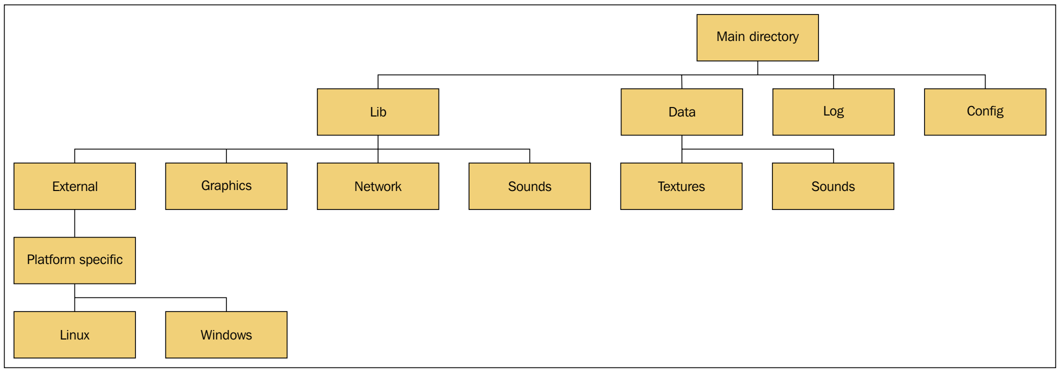
Lua language uses a `require` function to include modules in your script files. This function uses a default list of paths where it tries to find the module file. The Lua modules can be written as plain Lua scripts or use a form of binary library, which is OS and CPU architecture dependent. This is especially troublesome if want to include binary libraries for all supported operating systems and CPU architectures in one project.

A default set of paths might not always be appropriate for your project, mainly if you bundle many third-party modules with it.

This recipe shows how to set up the Lua interpreter so that it can find correct files in a systematic and user-definable way. This recipe should be used at the beginning of your main Lua script file so that further calls to the `require` function in Lua will use your file path structure.

Getting ready

You can use a directory structure as shown in the following diagram. If you intend to implement your application for multiple platforms, always divide platform-specific files into separate directories.



The Lib directory contains all the Lua module files and binary libraries.

However, each operating system uses its own file naming convention for binary libraries. The Lua language doesn't have an easy way to obtain the OS name. For this purpose, you can download and use the Lua script module `os_name.lua` from `https://gist.github.com/soulik/82e9d02a818ce12498d1`.

You should copy this file into your project directory so that the Lua interpreter can find it.

How to do it...

The `require` function in the Lua language uses a set of default paths defined in `package.path` and `package.cpath` string variables. With your new directory structure, you'd have to change those two variables manually for each operating system, which could be cumbersome.

Instead, you can define a Lua script to build up these two string variables from a generic list of paths for both Lua script files and binary libraries.

In the first step, you need to create a list of directories:

```
-- A list of paths to lua script modules
local paths = {
  '{root}/{module}',
  '{root}/lib/{module}',
  '{root}/lib/external/{module}',
  '{root}/lib/external/platform-specific/{platform}/{module}',
}
-- A list of paths to binary Lua modules
local module_paths = {
  '?.{extension}',
  '?/init.{extension}',
  '?/core.{extension}',
}
```

Strings enclosed with curly brackets will be substituted with the following values:

Name	Value
root	This is the application's root directory
platform	This is the current platform
module	This is the module's file path
extension	This is the module's filename extension, which is platform dependent

Binary module filename extensions that are platform dependent are also set in a table:

```
-- List of supported OS paired with binary file extension name
local extensions = {
  Windows = 'dll',
  Linux = 'so',
  Mac = 'dylib',
}
```

Now, you need to set `root_dir` which is the current working directory of the application and the current platform name as follows:

```
-- os_name is a supplemental module for
-- OS and CPU architecture detection
local os_name = require 'os_name'

-- A dot character represent current working directory
local root_dir = '.'
local current_platform, current_architecture = os_name.getOS()

local cpaths, lpaths = {}, {}
local current_clib_extension = extensions[current_platform]
```

Before you start building the path list, you need to check whether the current platform has defined binary module extensions as follows:

```
if current_clib_extension then
  -- now you can process each defined path for module.
  for _, path in ipairs(paths) do
    local path = path:gsub("{(%w+)}", {
      root = root_dir,
      platform = current_platform,
    })
    -- skip empty path entries
    if #path>0 then
      -- make a substitution for each module file path.
      for _, raw_module_path in ipairs(module_paths) do
        local module_path = path:gsub("{(%w+)}", {
          module = raw_module_path
        })
        -- add path for binary module
        cpaths[#cpaths+1] = module_path:gsub("{(%w+)}", {
          extension = current_clib_extension
        })
        -- add paths for platform independent lua and luac modules
```

```
          lpaths[#lpaths+1] = module_path:gsub("{(%w+)}", {
            extension = 'lua'
          })
          lpaths[#lpaths+1] = module_path:gsub("{(%w+)}", {
            extension = 'luac'
          })
        end
      end
    end
    -- build module path list delimited with semicolon.
    package.path = table.concat(lpaths, ";")
    package.cpath = table.concat(cpaths, ";")
  end
```

With this design, you can easily manage your module paths just by editing `paths` and `module_paths` tables.

Keep in mind that you need to execute this code before any `require` command.

How it works...

This recipe builds content for two variables that are used in the `require` function—`package.path` and `package.cpath`.

Both variables use a semicolon as a delimiter for individual paths. There's also a special character—the question mark which is substituted with the module name. Note that the path order might not be as important in this case as with our default list of paths. The path order might cause problems if you expect to use a module out of the project directory structure. Therefore, a customized set of paths from this recipe should always be used before the default set of paths.

The Lua language allows the use of hierarchical structure of modules. You can specify a submodule with package names delimited by a dot.

```
require 'main_module.submodule'
```

A dot is always replaced with the correct directory separator.

Making a stack

Stack data structure can be defined in the Lua language as a closure that always returns a new table. This table contains two functions defined by keys, `push` and `pop`. Both operations run in constant time.

Getting ready

Code from this recipe will be probably used more than once in your project so that it can be moved into the Lua module file with similar algorithms. The module file can use the following structure:

```
-- algoritms.lua

-- Placeholder for a stack data structure code

return {
  stack = stack,
}
```

This module structure can be used with algorithms from other recipes as well to keep everything organized.

How to do it…

The following code contains a local definition of the `stack` function. You can remove the `local` statement to make this function global or include it as part of the module:

```
local function stack()
  local out = {}
  out.push = function(item)
    out[#out+1] = item
  end
  out.pop = function()
    if #out>0 then
      return table.remove(out, #out)
    end
  end
  out.iterator = function()
    return function()
      return out.pop()
    end
  end
  return out
end
```

This stack data structure can be used in the following way:

```
local s1 = stack()
-- Place a few elements into stack
for _, element in ipairs {'Lorem','ipsum','dolor','sit','amet'} do
  s1.push(element)
end

-- iterator function can be used to pop and process all elements
for element in s1.iterator() do
     print(element)
end
```

How it works...

Calling the `stack` function will create a new empty table with three functions. `Push` and `pop` functions use the property of the length operator that returns the integer index of the last element. The `iterator` function returns a closure that can be used in a `for` loop to pop all the elements. The `out` table contains integer indices and no holes (without empty elements). Both the functions are excluded from the total length of the `out` table.

After you call the `push` function, the element is appended at the end of the `out` table. The `Pop` function removes the last element and returns the removed element.

Making a queue

The queue data structure can be constructed in a similar way as a stack with the `table.insert` and `table.remove` functions. However, this will add unnecessary overhead because each element insertion at the beginning of the list will need to move other elements as well. A better solution is using two indices that indicate the beginning and the end of the list.

Getting ready

The code from this recipe can be placed into the `algorithms.lua` file as in the *Making a stack* recipe.

How to do it...

The queue data structure will consist of a constructor that returns a new table with three functions: a `push`, a `pop`, and an `iterator`. The resulting table uses the modified version of the length operator to get the right length of the queue:

```
local function queue()
```

```
    local out = {}
    local first, last = 0, -1
    out.push = function(item)
      last = last + 1
      out[last] = item
    end
    out.pop = function()
      if first <= last then
        local value = out[first]
        out[first] = nil
        first = first + 1
        return value
      end
    end
    out.iterator = function()
      return function()
        return out.pop()
      end
    end
    setmetatable(out, {
      __len = function()
        return (last-first+1)
      end,
    })
    return out
  end
```

A new queue data structure can be created by calling the `queue` function:

```
local q1 = queue()
-- Place a few elements into queue
for _, element in ipairs {'Lorem','ipsum','dolor','sit','amet'} do
  q1.push(element)
end

-- You can use iterator to process all elements in single for loop
for element in q1.iterator() do
  -- each queue element will be printed onto screen
  print(element)
end
```

How it works...

This algorithm uses a pair of integer indices that represent positions of the first and the last element of the queue. This approach provides element insertion and deletion in constant time. Because the original length operator isn't suitable for this case, a modified one is provided.

The iterator function creates a new closure that is used in a `for` loop. This closure is called repeatedly until the `pop` function returns an empty result.

Making a prioritized queue

A prioritized queue or simple priority queue extends basic queue with the entry sorting feature. Upon entry insertion, you can set what will be the priority of the entry. This data structure is often used in job queuing where the most important (highest priority) jobs must be processed before the jobs with lower priority. Priority queues are often used in artificial intelligence as well.

This version of the prioritized queue allows you to obtain entries with minimal or maximal priority at constant time. Element priority can be updated. However, priority queue insertion, update, and removal might use linear time complexity in worst case scenarios.

There are two rules that should be noted:

- Each entry of this queue should be unique
- The order of retrieving elements with the same priority is not defined

Getting ready

This recipe will use the following shortcuts:

```
local ti = table.insert
local tr = table.remove

-- removes element from table by its value
local tr2 = function(t, v)
  for i=1,#t do
    if t[i]==v then
      tr(t, i)
      break
    end
  end
end
```

It's recommended to put it all together in one Lua module file.

How to do it...

The priority queue can be defined as in the following code:

```
return function pqueue()
  -- interface table
  local t = {}

  -- a set of elements
  local set = {}
  -- a set of priority bags with a elements
  local r_set = {}
  -- sorted list of priorities
  local keys = {}

  -- add element into storage, set its priority and sort keys
  --  k - element
  --  v - priority
  local function addKV(k, v)
    set[k] = v
    -- create a new list for this priority
    if not r_set[v] then
      ti(keys, v)
      table.sort(keys)
      local k0 = {k}
      r_set[v] = k0
      setmetatable(k0, {
        __mode = 'v'
      })
    -- add element into list for this priority
    else
      ti(r_set[v], k)
    end
  end

  -- remove element from storage and sort keys
  local remove = function(k)
    local v = set[k]
    local prioritySet = r_set[v]
    tr2(prioritySet, k)
    if #prioritySet < 1 then
      tr2(keys, v)
      r_set[v] = nil
      table.sort(keys)
      set[k] = nil
    end
  end; t.remove = remove

  -- returns an element with the lowest priority
  t.min = function()
    local priority = keys[1]
    if priority then
      return r_set[priority][1] or {}
    else
      return {}
    end
  end

  -- returns an element with the highest priority
  t.max = function()
    local priority = keys[#keys]
    if priority then
      return r_set[priority][1] or {}
    else
      return {}
    end
  end

  -- is this queue empty?
  t.empty = function()
    return #keys < 1
  end
  -- simple element iterator, retrieves elements with
  -- the highest priority first
  t.iterate = function()
    return function()
      if not t.empty() then
        local element = t.max()
        t.remove(element)
        return element
      end
    end
  end
  -- setup pqueue behavior
  setmetatable(t, {
    __index = set,
    __newindex = function(t, k, v)
      if not set[k] then
        -- new item
        addKV(k, v)
      else
        -- existing item
        remove(k)
        addKV(k, v)
      end
    end,
  })
  return t
end
```

How it works...

This priority queue algorithm uses three tables: `set`, `r_set`, and `keys`. These tables help to organize elements into so-called **priority bags**. The first one, `set` contains elements paired with their priorities. It's also used when you try to obtain element priority from the queue. The second one, `r_set` contains priority bags. Each bag represents a priority level. The last one `keys` contains a sorted list of priorities, which is used in the extraction of elements from a minimal or maximal priority bag.

Each element can be inserted in a way similar to the Lua table with the exception that the inserted element is used as a key and `priority` is stored as a value:

```
priority_queue[element] = priority
```

This form of access can be used to update element priority. Elements with minimal or maximal priority can be extracted using the `min` or `max` function respectively;

```
local min_element = priority_queue.min()
local max_element = priority_queue.max()
```

Note that elements remain in the priority queue until you delete them with the `remove` function;

```
priority_queue.remove(element)
```

Priority queue can be queried with the `empty` function that returns true if there's no element in the queue;

```
priority_queue.empty()
```

You can use the iterator function in `for` loop to process all queue elements sorted by their priority:

```
for element in priority_queue.iterator() do
  -- do something with this element
end
```

Extending ipairs for use in sparse arrays

The `ipairs` function in the Lua language is used to iterate over entries in a sequence. This means every entry must be defined by the pair of key and value, where the key is the integer value. The main limitation of the `ipairs` function is that the keys must be consecutive numbers.

You can modify the `ipairs` function so that you can successfully iterate over entries with integer keys that are not consecutive. This is commonly seen in sparse arrays.

Getting ready

In this recipe, you'll need to define our own `iterator` function, which will return every entry of a sparse array in deterministic order. In this case, the iterator function can be included in your code as a global function to accompany `pairs` and `ipairs` functions; or you can put it in a Lua module file not to pollute the global environment space.

How to do it...

This code shows a very simple sparse array iterator without any caching:

```
function ipairs_sparse(t)
  -- tmpIndex will hold sorted indices, otherwise
  -- this iterator would be no different from pairs iterator
  local tmpIndex = {}
  local index, _ = next(t)
  while index do
    tmpIndex[#tmpIndex+1] = index
    index, _ = next(t, index)
  end
  -- sort table indices
  table.sort(tmpIndex)
  local j = 1

  return function()
    -- get index value
    local i = tmpIndex[j]
```

```
      j = j + 1
      if i then
        return i, t[i]
      end
    end
  end
```

The following lines of code show the usage example for iteration over a sparse array;

```
-- table below contains unsorted sparse array
local t = {
  [10] = 'a', [2] = 'b', [5] = 'c', [100] = 'd', [99] = 'e',
}
-- iterate over entries
for i, v in ipairs_sparse(t) do
  print(i,v)
end
```

How it works...

The Lua language uses iterator functions in the control structure called the generic `for`. The generic `for` calls the iterator function for each new iteration and stops when the iterator function returns nil. The `ipairs_sparse` function works in the following steps:

1. It builds a new index of keys from the table.
2. It sorts the index table.
3. It returns a closure where each call of the closure returns a consecutive index and a value from the sparse array.

Each call to `ipairs_sparse` prepares a new index table called `index`. The index consists of (integer, entry) pairs.

Creating Lua modules

The Lua language doesn't impose strict policies on what a module should look like. Instead, it encourages programmers to find their own style depending on the situation.

Getting ready

In this recipe, you will create three versions of a module that contains one local variable, one variable accessible from outside the module, one function that returns a simple value, and a function that uses a value from the current module.

How to do it...

There are three types of modules that are commonly used:

- A module that returns a table as a module interface
- A module in the form of an object
- A module in the form of a singleton object

The first case is used mostly with modules that contain an interface to third-party libraries. The second type of module is used less often, but it's useful if you need multiple instances of the same object, for example, a network stack. The last one uses a similar approach as in the previous case, but this time there's always only one instance of the object. Many games use the singleton object for the resource management system.

A module that returns a table as an interface

In this case, the module uses locally defined variables and functions. Every function intended for external use is put into one table. This common table is used as an interface with the outer world and is returned at the end of the module:

```
-- module1.lua
local var1 = 'ipsum'
local function local_function1()
  return 'lorem'
end

local function local_function2(self)
  return var1 .. self.var2
end
-- returns module interface
return {
  lorem = local_function1,
  ipsum = local_function2,
  var2 = 'sit',
}
```

A module in the form of an object

This module type doesn't manipulate the global namespace. Every object you create uses its own local namespace:

```
-- module2.lua
local M = function()
  local instance
  local var1 = 'ipsum'
  instance = {
    var2 = 'sit',
    lorem = function()
      return 'lorem'
    end,
    ipsum = function(self)
      return var1 .. self.var2
    end,
  }
  return instance
end

return M
```

A module in the form of a singleton object

This is a special case of object module. There is only one and the same object instance:

```
-- module3.lua
local instance

local M = function()
  if not instance then
    local var1 = 'ipsum'
    instance = {
      var2 = 'sit',
      lorem = function()
        return 'lorem'
      end,
      ipsum = function(self)
        return var1 .. self.var2
         end,
    }
  end
  return instance
end

return M
```