How to Build a Roguelike

by Matt Mongeau

thoughtbot

Write Yourself a Roguelike

Matt Mongeau

March 13, 2015

Contents

Introduction

Chapter i - Why write this book?

The reason I decided to write this book is because game development was one of the things that first attracted me to developing software. I started programming in highschool with QBasic. QBasic made it pretty easy to enter into a graphics mode and start drawing on the screen. It wasn't long before I had written a very primitive RPG. Nowadays, it's much harder to get started due to the complexities of graphical hardware and complex operating system interactions. Roguelikes allow us to simplfy things one again.

There is a certain purity in an ASCII based game - there is very little overhead and very little math required to get started. Imagination also plays a large role. Most games these days have taken imagination out of the equation. I know what everything looks like because the game's artists have fleshed it all out. However, games like NetHack allow me to imagine what's going on and to in some ways weave my own story.

Because I've found a lot of enjoyment in playing games like NetHack, I've developed a natural curiosity for the internal workings. How would one go about creating the same kind of game? I've spent considerable time diving in and out of the C code to answer that question. I hope this book will answer that question for you as well.

Generating a Character

Chapter 1 - The Title Screen

To begin our journey, we'll first need to learn how to use the `curses` gem. If you haven't already, install the `curses` gem via:

```
gem install curses
```

Once that finishes installing, we'll continue by examining the NetHack title screen. We'll want our title screen to mimic it.

Let's start our game by writing the simplest curses example we can come up with. The program will initialize curses, read a single character, and then quit. To do this, create a file named `main.rb` and add the following:

```
require "curses" # require the curses gem
include Curses   # mixin curses

# The next three functions are provided by including the Curses module.

init_screen      # starts curses visual mode
getch            # reads a single character from stdin
close_screen     # closes the ncurses screen
```

If you run this program, you will see the terminal go black and upon pressing a character it will return back to normal.

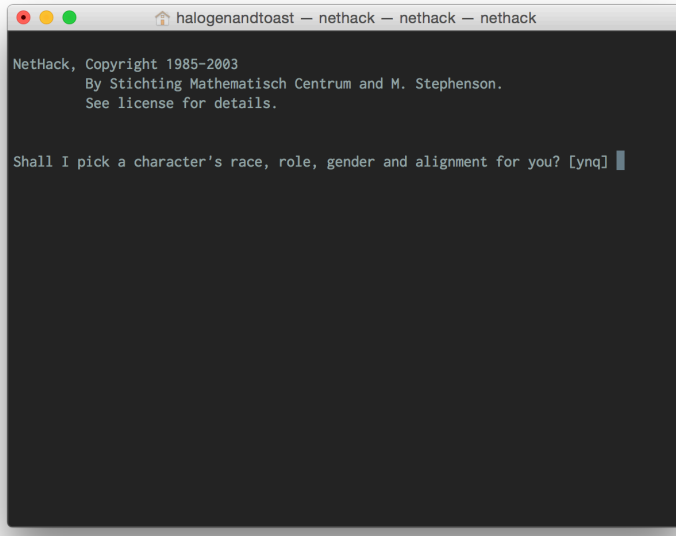

Figure 1.1: character selection

Now that we've got a simple curses example running, let's work on our title screen. We're going to break our code up into three files. The first file we'll create is named `game.rb` and it should contain the following:

```
class Game
  def initialize
    @ui = UI.new
    at_exit { ui.close } # runs at program exit
  end

  def run
    title_screen
  end

  private

  attr_reader :ui

  def title_screen
    ui.message(0, 0, "Rhack, a NetHack clone")
    ui.message(7, 1, "by a daring developer")
    ui.choice_prompt(0, 3, "Shall I pick a character's race, role, gender and alignment for
  end
end
```

Then create the file `ui.rb` with:

```
class UI
  include Curses

  def initialize
    noecho # do not print characters the user types
    init_screen
  end

  def close
    close_screen
```

```
  end

  def message(x, y, string)
    setpos(y, x) # positions the cursor - pay attention to the argument order here
    addstr(string) # prints a string at cursor position
  end

  def choice_prompt(x, y, string, choices)
    message(x, y, string + " ")

    loop do
      choice = getch
      return choice if choices.include?(choice)
    end
  end
end
```

Finally, change your `main.rb` to:

```
$LOAD_PATH.unshift "." # makes requiring files easier

require "curses"
require "ui"
require "game"

Game.new.run
```

I've chosen to break the UI into its own class for a few reasons. First, in game development, it's easy to produce code that is difficult to understand. We want to avoid this by trying to employ the single-responsibility pattern as much as possible. Tangentally, if we decide to replace our UI implementation with a different one, the isolation here makes doing that far easier.

The responsibility of our `Game` class will be to manage all of our global state and the execute the main run loop.

If you run the program now, it will look very much like the initial NetHack screen.

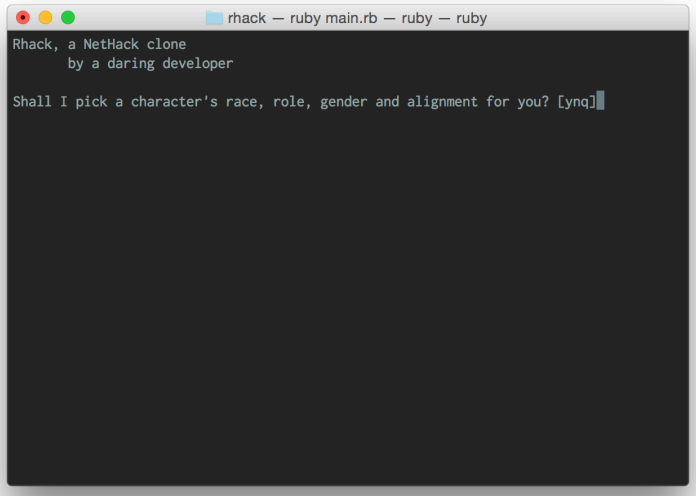

Figure 1.2: Rhack

Moving forward, we're going to want to show more than a title screen. Let's start by refactoring our current code into something more adaptable. Refactor `game.rb` to the following:

```
class Game
  def initialize
    @ui = UI.new
    @options = { quit: false, randall: false } # variable for options
    at_exit { ui.close; p options } # See selected options at exit
  end

  def run
    title_screen
  end

  private

  attr_reader :ui, :options # Add attr_reader for options

  def title_screen
    TitleScreen.new(ui, options).render
    quit?
  end

  def quit?
    exit if options[:quit]
  end
end
```

Now we'll create a `title_screen.rb` file with the following:

```
class TitleScreen
  def initialize(ui, options)
    @ui = ui
    @options = options
  end
```

```
  def render
    ui.message(0, 0, "Rhack, a NetHack clone")
    ui.message(7, 1, "by a daring developer")
    handle_choice prompt
  end

  private

  attr_reader :ui, :options

  def prompt
    ui.choice_prompt(0, 3, "Shall I pick a character's race, role, gender and alignment for
  end

  def handle_choice(choice)
    case choice
    when "q" then options[:quit] = true
    when "y" then options[:randall] = true
    end
  end
end
```

You can see here how we'll be making use of the `options` variable created in `game.rb`. It will store the selections the user makes during setup. We need to do this in order to communicate between the different selection screens about what choices the player has made. If the user selects "q" we'll store that we need to quit, if they choose "y" then we'll randomly assign the rest of the traits. In order to keep our application working you'll also need to add:

```
require "title_screen"
```

to `main.rb`. Now when running the program and choose an option you'll see set in the output that is printed. For instance, if I select yes, then I'll see the following:

```
{:quit=>false, :randall=>true}
```

Chapter 2 - Messages

There are going to be a lot of in-game messages and to make things more fluid we should extact them into a yaml file. This makes them far easier to change (or to internationalize) later. Let's start by creating a `data` directory. This directory will hold some yaml files that will contain in game text and other data. In that directory, let's create a file for holding our in-game messages. Name the file `messages.yaml` and add the following to it:

```
---
title:
  name: Rhack, a NetHack clone
  by: by a daring developer
  pick_random: "Shall I pick a character's race, role, gender and alignment for you? [ynq]"
```

Next, we'll want to update our `TitleScreen` class to make use of these messages. In order to do that, we'll first need some way to load the yaml file. In this situation, it's a good idea to isolate an external dependency like YAML in order to make it easier to replace or modify in the future. We took this exact approach with Curses by extracting the UI into its own class. Let's extract a `DataLoader` class that knows how to load our data for us. Create a `data_loader.rb` file with the following:

```
class DataLoader
  def self.load_file(file)
    new.load_file(file)
  end

  def load_file(file)
    symbolize_keys YAML.load_file("data/#{file}.yaml")
  end

  private

  def symbolize_keys(object)
    case object
    when Hash
      object.each_with_object({}) do |(key, value), hash|
```

```
        hash[key.to_sym] = symbolize_keys(value)
      end
    when Array
      object.map { |element| symbolize_keys(element) }
    else
      object
    end
  end
end
```

The reason behind `symbolize_keys` is that YAML will parse all the keys as strings and I prefer symbols for this. Even though `ActiveSupport` has a similar method, we're going to leave it out because it won't work directly with arrays. Our implementation will symbolize the keys correctly for hashes or arrays even if they are nested.

Now we'll create a global way to access these messages. Create a file called `messages.rb` with the following:

```
module Messages
  def self.messages
    @messages ||= DataLoader.load_file("messages")
  end

  def self.[](key)
    messages[key]
  end
end
```

It's evident here that our Messages module knows nothing about the YAML backend, instead it simply asks our DataLoader to load the messages. Now that we have a way to get our messages let's change our `title_screen.rb` to make use of it. In initialize add the following:

```
@messages = Messages[:title]
```

Make sure to add `:messages` to the `attr_reader` line and then change `render` to the following:

```
def render
  ui.message(0, 0, messages[:name])
  ui.message(7, 1, messages[:by])
  handle_choice prompt
end
```

And change `prompt` to:

```
def prompt
  ui.choice_prompt(0, 3, messages[:pick_random], "ynq")
end
```

Now to finish up, add `require`s in `main.rb` for `yaml`, `data_loader`, and `messages`. When you run the program again it should still function like our previous implementation.

Chapter 3 - Role call

For a game like NetHack, there is a lot of information that goes in to creating a character. From a top level, a character will have a role, race, gender and alignment. Each of these traits will determine how a game session will play.

We'll start by allowing the player to choose their role. In NetHack, these are the roles a player can select:

We will implement all of these. Looking at this list, "data" should immediately come to mind. We're going to create another data file to hold the information for our roles. To start with, we're going to give each role a `name` and a `hotkey`. Create `data/roles.yaml` with the following:

```
---
- name: Acheologist
  hotkey: a
- name: Barbarian
  hotkey: b
- name: Caveman
```

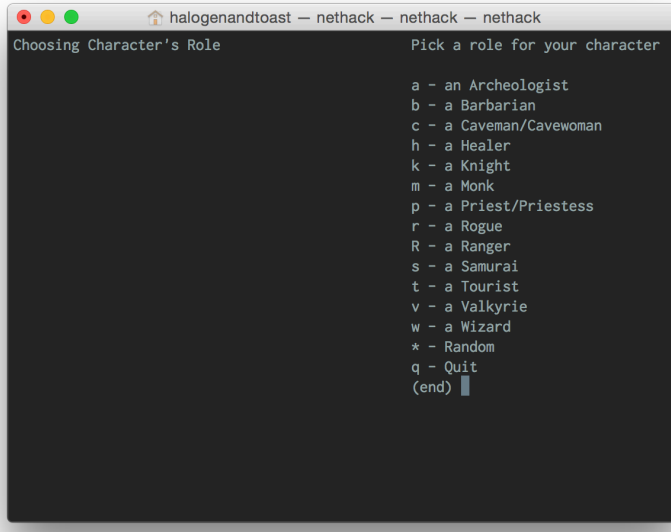

Figure 1.3: selection

```
  hotkey: d
- name: Healer
  hotkey: h
- name: Knight
  hotkey: k
- name: Monk
  hotkey: m
- name: Priest
  hotkey: p
- name: Rogue
  hotkey: r
- name: Ranger
  hotkey: R
- name: Samurai
  hotkey: s
- name: Tourist
  hotkey: t
- name: Valkyrie
  hotkey: v
- name: Wizard
  hotkey: w
```

Now we're going to create a `Role` class that can load all of this data. Create a file named `role.rb` with the following:

```
class Role
  def self.for_options(_)
    all
  end

  def self.all
    DataLoader.load_file("roles").map do |data|
      new(data)
    end
  end

  attr_reader :name, :hotkey
```

```
  def initialize(data)
    data.each do |key, value|
      instance_variable_set("@#{key}", value)
    end
  end

  def to_s
    name
  end
end
```

We're using for_options here to unify the interface across all of our characteristics, since race and alignment will be dependent on role. We'll see shortly why this abstraction makes sense.

Now we're going to write a generic `SelectionScreen` class. It's job will be to print two messages and display a list of options that can be selected by a hotkey. Create the file `selection_screen.rb` with:

```
class SelectionScreen
end
```

Now let's add some methods one by one. First we'll add our `initialize` and some `attr_reader`s:

```
def initialize(trait, ui, options)
  @items = trait.for_options(options)

  @ui = ui
  @options = options

  @key = trait.name.downcase.to_sym
  @messages = Messages[key]
end

private
```

```
attr_reader :items, :ui, :options, :key, :messages
```

When we create a our selection screen we'll call it from `game.rb` with:

```
SelectionScreen.new(Role, ui, options).render
```

So in this case, `trait` will be the class `Role`. On the first line we fetch all the relevant roles by calling `for_options`. If you recall, `for_options` just reads the yaml file of roles and returns all of them. Next we assign the `ui` and `options` variables. Then, we determine a key that we'll use for a couple of things. If `Role` is our trait, then we want `:role` to be our key. Finally, we grab a hash of messages related to our key (:role in this case).

Now we'll implement our only **public** method `render` (make sure this goes above the `private` line):

```
def render
  if random?
    options[key] = random_item
  else
    render_screen
  end
end
```

In this function we check to see if we need to randomly select an item. If we do we don't want to render the screen, so it simply sets the option and returns. Otherwise we'll render the screen. The implementation for `random?` and `random_item` look like this:

```
def random?
  options[:randall]
end

def random_item
  items.sample
end
```

For now, `random?` simply checks if `randall` was set and `random_item` just chooses a random element form our items array. Now we can implement `render_screen` and `instructions`:

```
def render_screen
  ui.clear
  ui.message(0, 0, messages[:choosing])
  ui.message(right_offset, 0, instructions)
  render_choices
  handle_choice prompt
end

# instructions has been pulled out into it's own method for a reason
# you will see later

def instructions
  messages[:instructions]
end
```

Here we clear the screen, display the message on the left - “Choosing Role”, display the message on the right - “Pick the role of your character”, display the choices, and then prompt and handle the player's selection. For convenience, I've pulled out `right_offset` into a method since we'll use it a few times:

```
def right_offset
  @right_offset ||= (instructions.length + 2) * -1
end
```

This method returns a negative number representing how far left from the right side we should be when printing the right half of our screen. We'll need to update our `UI` class to handle negative numbers, but let's finish our `SelectionScreen` class first.

Now we'll write our method for rendering our choices

```
def render_choices
  items.each_with_index do |item, index|
```

```
    ui.message(right_offset, index + 2, "#{item.hotkey} - #{item}")
  end

  ui.message(right_offset, items.length + 2, "* - Random")
  ui.message(right_offset, items.length + 3, "q - Quit")
end
```

This function is relatively straight forward. We loop through each item and print out the hotkey and the name of the role (we're cheating here by not printing "a" or "an" in front of the name, but it's not really important).

Now let's implement `handle_choice` and `item_for_hotkey`:

```
def handle_choice(choice)
  case choice
  when "q" then options[:quit] = true
  when "*" then options[key] = random_item
  else options[key] = item_for_hotkey(choice)
end

def item_for_hotkey(hotkey)
  items.find { |item| item.hotkey == hotkey }
end
```

Here we have 3 choices. If the user presses "q" then we want to quit. If they press "*" then we want to randomly choose an item. If they press any other valid option we want to assign the corresponding role.

Finally let's implement `prompt` and `hotkeys`:

```
def prompt
  ui.message(right_offset, items.length + 4, "(end)", hotkeys)
end

def hotkeys
  items.map(&:hotkey).join + "*q"
end
```

The `hotkeys` represent our valid choices, but we need to make sure to add "*" and "q" as valid hotkeys.

Now we're ready to initialize this screen in `game.rb`. Add the following constant:

```
TRAITS = [Role]
```

Then change the `run` function to look like this:

```
def run
  title_screen
  setup_character
end
```

And then add `setup_character` and `get_traits` as a private methods:

```
def setup_character
  get_traits
end

def get_traits
  TRAITS.each do |trait|
    SelectionScreen.new(trait, ui, options).render
    quit?
  end
end
```

There are a few things left to do in order to get this working. First, in `main.rb` add:

```
require "role"
require "selection_screen"
```

Above the `require "game"` line. Next, we'll need to modify our `UI` class to have a clear function. Curses provides this function, but it's private, so we'll need to add the following to `ui.rb`:

```
def clear
  super # call curses's clear method
end
```

While we have the `ui.rb` file open we should handle our `right_offset` issue we described before. Change the implementation of `message` to the following:

```
def message(x, y, string)
  x = x + cols if x < 0
  y = y + lines if y < 0

  setpos(y, x)
  addstr(string)
end
```

Finally, we'll need to add some messages to our `data/messages.yaml` file:

```
role:
  choosing: Choosing Role
  instructions: Pick a role for your character
```

If you run the program and choose "n" for the first choice then you should see:

Choosing any role will print out the options again, but this time it will display the selected role as well. If you choose "y" at the title screen a random role will appear here. Now that we've laid down the framework for setting traits it should be fairly easy to implement the remaining ones.

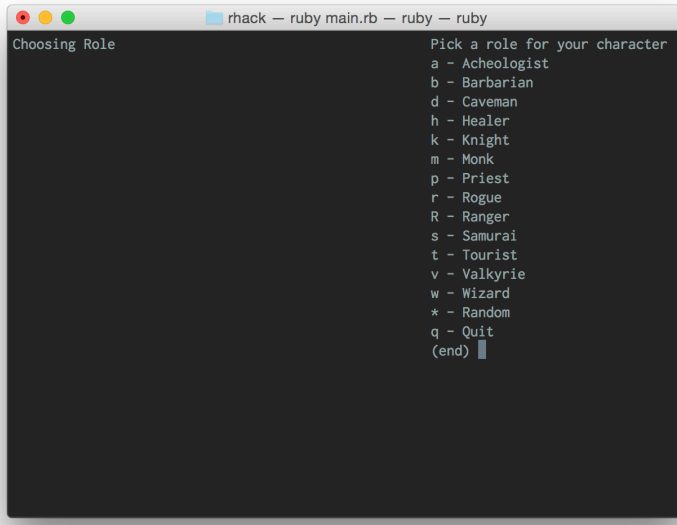

Figure 1.4: role selection example

Fin.

Thank you for checking out the sample of ***Write Yourself a Roguelike***. If you'd like to access the full content, the example game, and ongoing updates you can pick up the book at: `http://write-a-roguelike.com`.