Tomas Varaneckas

Developing Games

With Ruby

For those who write code for living

Developing Games With Ruby

For those who write code for living

Tomas Varaneckas

This book is for sale at http://leanpub.com/developing-games-with-ruby

This version was published on 2014-12-16

Table of Contents

A Boy Who Wanted To Create Worlds

Once there was a boy who fell in love with this magical device that could bring things to life inside a glaring screen. He spent endless hours exploring imaginary worlds, fighting strange creatures, shooting pixelated spaceships, racing boxy cars. The boy kept pondering. “How is this made? I want to create my own worlds…”.

Then he discovered programming. “I can finally do it!” - he thought. And he tried. And failed. Then he tried harder. He failed again and again. He was too naive to realize that those worlds he was trying to create were too sophisticated, and his knowledge was too limited. He gave up creating those worlds.

What he didn’t give up is writing code for this magical device. He realized he isn’t smart enough to create worlds, yet he found out he could create simpler things like small applications - web, desktop, server side or whatnot. Few years later he found himself getting paid to make those.

Applications got increasingly bigger, they spanned across multiple servers, integrated with each other, became pats of huge infrastructures. The boy, now a grown man, was all into it. It was fun and challenging enough to spend over 10000 hours learning and building what others wanted him to build.

Some of these things were useful, some where boring and pointless. Some were never finished. There were things he was proud of, there were others that he wouldn’t want to talk about, nonetheless everything he built made him a better builder. Yet he never found the time, courage or reason to build what he really wanted to build since he was a little boy - his own worlds.

Until one day he realized that no one can stop him from following his dream. He felt that equipped with his current knowledge and experience he will be able to learn to create worlds of his own. And he went for it.

This boy must live in many software developers, who dream about creating games, but instead sell their software craftsmanship skills to those who need something else. This boy is me, and you. And it’s time to set him free.

Welcome to the world of game development that was waiting for you all these years.

Why Ruby?

When it comes to game development, everyone will tell you that you should go with C++ or some other statically typed language that compiles down to bare metal instructions. Or that you should go with full blown game development platform like Unity. Slow, dynamic languages like Ruby seem like the last choice any sane game developer would go for.

A friend of mine said “There’s little reason to develop a desktop game with Ruby”, and he was absolutely right. Perhaps this is the reason why there are no books about it. All the casual game action happens in mobile devices, and desktop games are for seasoned gamers who demand fast and detailed 3D graphics, motion-captured animations and sophisticated game mechanics - things we know we are not going to be able to build on our own, without millions from VC pockets and Hollywood grade equipment.

Now, bear with me. Your game will not be a 3D MMORPG set in huge, photo realistic representation of Middle-earth. Let’s leave those things to Bethesda, Ubisoft and Rockstar Games. After all, everyone has to start somewhere, and you have to be smart enough to understand, that even though that little boy in you wants to create an improved version of Grand Theft Auto V, we will have to go for something that resembles lesser known Super Nintendo titles instead.

Why not go mobile then? Those devices seem perfect for simpler games. If you are a true gamer at heart, you will agree that touch screen games you find in modern phones and tablets are only good for killing 10 minutes of your time while taking a dump. You *have* to feel the resistance when you click a button! Screen size also does matter. Playing anything on mobile phone is a torture for those who know what playing real games should feel like.

So, your game will have to be small enough for you to be able to complete it, it will have to have simple 2D graphics, and would not require the latest GeForce with at least 512MB of RAM. This fact gives you the benefit of choice. You don’t have to worry about performance that much. You can choose a friendly and productive language that is designed for programmer happiness. And this is where Ruby starts to shine. It’s beautiful, simple and elegant. It is close to poetry.

What You Should Know Before Reading This Book

As you can read on the cover, this book is “for those who write code for living”. It’s not a requirement, and you will most likely be able to understand everything even if you are a student or hobbyist, but this book will not teach you how to be a good programmer. If you want to learn that, start with timeless classic: The Pragmatic Programmer: From Journeyman to Master.

You should understand Ruby at least to some extent. There are plenty of books and resources covering that subject. Try Why’s Poignant Guide To Ruby or Eloquent Ruby. You can also learn it while reading this book. It shouldn’t be too hard, especially if you already write code for living. After all programming language is merely a tool, and when you learn one, others are relatively easy to switch to.

You should know how to use the command line. Basic knowledge of Git can also be handy.

You don’t have to know how to draw or compose music. We will use media that is available for free. However, knowledge of graphics and audio editing software won’t hurt.

What Are We Going To Build?

This question is of paramount importance. The answer will usually determine if you will likely to succeed. If you want to overstep your boundaries, you will fail. It shouldn't be too easy either. If you know something about programming already, I bet you can implement Tic Tac Toe, but will you feel proud about it? Will you be able to say "I've built a world!". I wouldn't.

Graphics

To begin with, we need to know what kind of graphics we are aiming for. We will instantly rule out 3D for several reasons:

- We don't want to increase the scope and complexity
- Ruby may not be fast enough for 3D games
- Learning proper 3D graphics programming requires reading a separate book that is several times thicker than this one.

Now, we have to swallow our pride and accept the fact that the game will have simple 2D graphics. There are three choices to go for:

- Parallel Projection
- Top Down
- Side-Scrolling

Parallel Projection (think Fallout 1 & 2) is pretty close to 3D graphics, it requires detailed art if you want it to look decent, so we would have a rough start if we went for it.

Top Down view (old titles of Legend of Zelda) offers plenty of freedom to explore the environment in all directions and requires less graphical detail, since things look simpler from above.

Side Scrolling games (Super Mario Bros.) usually involve some physics related to jumping and require more effort to look good. Feeling of exploration is limited, since you usually move from left to right most of the time.

Going with Top Down view will give us a chance to create our game world as open for exploration as possible, while having simple graphics and movement mechanics. Sounds like the best choice for us.

If you are as bad at drawing things as I am, you could still wonder how we are going to get our graphics. Thankfully, there is this opengameart.org. It's like GitHub of game media, we will surely find something there. It also contains audio samples and tracks.

Game Development Library

Implement it all yourself or harness the power of some game development library that offers you boilerplates and convenient access to common functions? If you're like me, you would definitely want to implement it all yourself, but that may be the reason why I failed to make a decent game so many times.

If you will try to implement it all yourself, you will most likely end up reimplementing some existing game library, poorly. It won't take long while you reach a point where you need to interface with underlying operating system libraries to get graphics. And guess if those bindings will work in a different operating system?

So, swallow your pride again, because we are going to use an existing game development library. Good news is that you will be able to actually finish the game, and it will be portable to Windows, Mac and Linux. We will still have to build our own game engine for ourselves on top of it, so don't think it won't be fun.

There are several game libraries available for Ruby, but it's a simple choice, because Gosu is head and shoulders above others. It's very mature, has a large and active community, and it is mainly written in C++ but has first class Ruby support, so it will be both fast and convenient to use.

Many of other Ruby game libraries are built on top of Gosu, so it's a solid choice.

Theme And Mechanics

Choosing the right theme is undoubtedly important. It should be something that appeals to you, something you will want to play, and it should not imply difficult game mechanics. I love MMORPGs, and I always dreamed of making an open world game where you can roam around, meet other players, fight monsters and level up. Guess how many times I started building such a game? Even if I wouldn't have lost the count, I wouldn't be proud to say the number.

This time, equipped with logic and sanity, I've picked something challenging enough, yet still pretty simple to build. Are you ready?

Drumroll…

We will be building a multi directional shooter arcade game where you control a tank, roam around an island, shoot enemy tanks and try not to get destroyed by others.

If you have played Battle City or Tank Force, you should easily get the idea. I believe that implementing such a game (with several twists) would expose us to perfect level of difficulty and provide substantial amount of experience.

We will use a subset of these gorgeous graphics which are available on opengameart.org, generously provided by Csaba Felvegi.

Preparing The Tools

While writing this book, I will be using Mac OS X (10.9), but it should be possible to run all the examples on other operating systems too.

Gosu Wiki has "Getting Started" pages for Mac, Linux and Windows, so I will not be going into much detail here.

Getting Gosu to run on Mac Os X

If you haven't set up your Mac for development, first install Xcode using App Store. System Ruby should work just fine, but you may want to use Rbenv or RVM to avoid polluting system Ruby. I've had trouble installing Gosu with RVM, but your experience may vary.

To install the gem, simply run:

```
$ gem install gosu
```

You may need to prefix it with `sudo` if you are using system Ruby.

To test if gem was installed correctly, you should be able to run this to produce an empty black window:

```
$ irb
irb(main):001:0> require 'gosu'
=> true
irb(main):002:0> Gosu::Window.new(320, 240, false).show
=> nil
```

Most developers who use Mac every day will also recommend installing Homebrew package manager, replace Terminal app with iTerm2 and use Oh-My-Zsh to manage ZSH configuration.

Getting The Sample Code

You can find sample code at GitHub: https://github.com/spajus/ruby-gamedev-book-examples.

Clone it to a convenient location:

```
$ cd ~/gamedev
$ git clone git@github.com:spajus/ruby-gamedev-book-examples.git
```

The source code of final product can be found at https://github.com/spajus/tank_island

Other Tools

All you need for this adventure is a good text editor, terminal and probably some graphics editor. Try GIMP if you want a free one. I'm using Pixelmator, it's wonderful, but for Mac only. A noteworthy fact is that Pixelmator was built by fellow Lithuanians.

When it comes to editors, I don't leave home without Vim, but as long as what you use makes you productive, it doesn't make any difference. Vim, Emacs or Sublime are all good enough to write code, just have some good plugins that support Ruby, and you're set. If you really feel you need an IDE, which may be the case if you are coming from a static language, you can't go wrong with RubyMine.

Gosu Basics

By now Gosu should be installed and ready for a spin. But before we rush into building our game, we have to get acquainted with our library. We will go through several simple examples, familiarize ourselves with Gosu architecture and core principles, and take a couple of baby steps towards understanding how to put everything together.

To make this chapter easier to read and understand, I recommend watching Writing Games With Ruby talk given by Mike Moore at LA Ruby Conference 2014. In fact, this talk pushed me towards rethinking this crazy idea of using Ruby for game development, so this book wouldn't exist without it. Thank you, Mike.

Hello World

To honor the traditions, we will start by writing "Hello World" to get a taste of what Gosu feels like. It is based on Ruby Tutorial that you can find in Gosu Wiki.

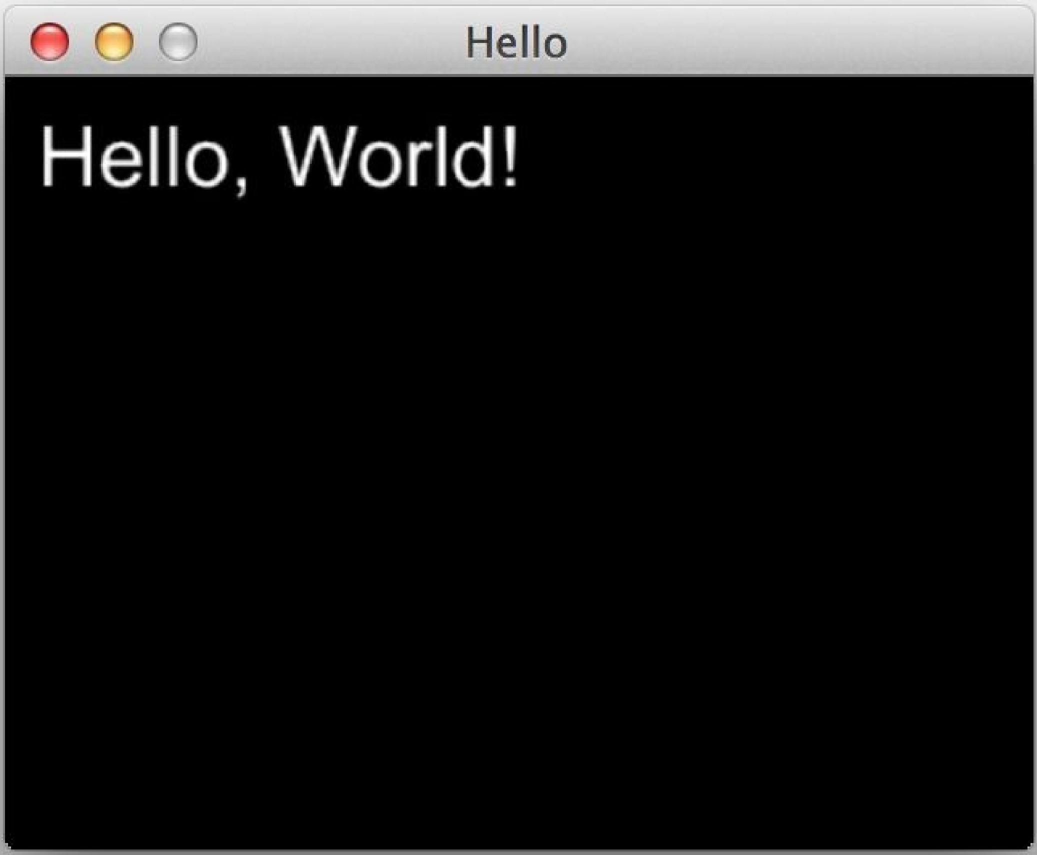
01-hello/hello_world.rb

```
require 'gosu'

class GameWindow < Gosu::Window
  def initialize(width=320, height=240, fullscreen=false)
    super
    self.caption = 'Hello'
    @message = Gosu::Image.from_text(
      self, 'Hello, World!', Gosu.default_font_name, 30)
  end

  def draw
    @message.draw(10, 10, 0)
  end
end

window = GameWindow.new
window.show
```

Run the code:

```
$ ruby 01-hello/hello_world.rb
```

You should see a neat small window with your message:



Hello World

See how easy that was? Now let's try to understand what just happened here.

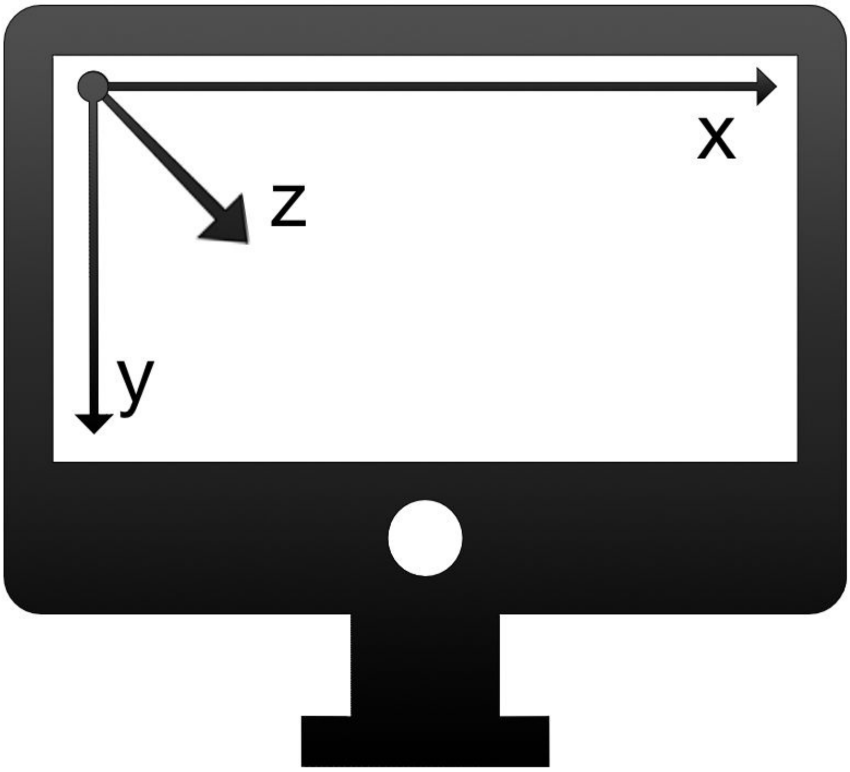
We have extended `Gosu::Window` with our own `GameWindow` class, initializing it as `320x240` window. `super` passed `width`, `height` and `fullscreen` initialization parameters from `GameWindow` to `Gosu::Window`.

Then we defined our window's `caption`, and created `@message` instance variable with an image generated from text `"Hello, World!"` using `Gosu::Image.from_text`.

We have overridden `Gosu::Window#draw` instance method that gets called every time Gosu wants to redraw our game window. In that method we call `draw` on our `@message` variable, providing `x` and `y` screen coordinates both equal to `10`, and `z` (depth) value equal to 0.

Screen Coordinates And Depth

Just like most conventional computer graphics libraries, Gosu treats `x` as horizontal axis (left to right), `y` as vertical axis (top to bottom), and `z` as order.



Screen coordinates and depth

`x` and `y` are measured in pixels, and value of `z` is a relative number that doesn't mean anything on it's own. The pixel in top-left corner of the screen has coordinates of `0:0`.

`z` order in Gosu is just like `z-index` in CSS. It does not define zoom level, but in case two shapes overlap, one with higher `z` value will be drawn on top.

Main Loop

The heart of Gosu library is the main loop that happens in [`Gosu::Window`](Gosu::Window). It is explained fairly well in Gosu wiki, so we will not be discussing it here.

Moving Things With Keyboard

We will modify our "Hello, World!" example to learn how to move things on screen. The following code will print coordinates of the message along with number of times screen was redrawn. It also allows exiting the program by hitting Esc button.

01-hello/hello_movement.rb

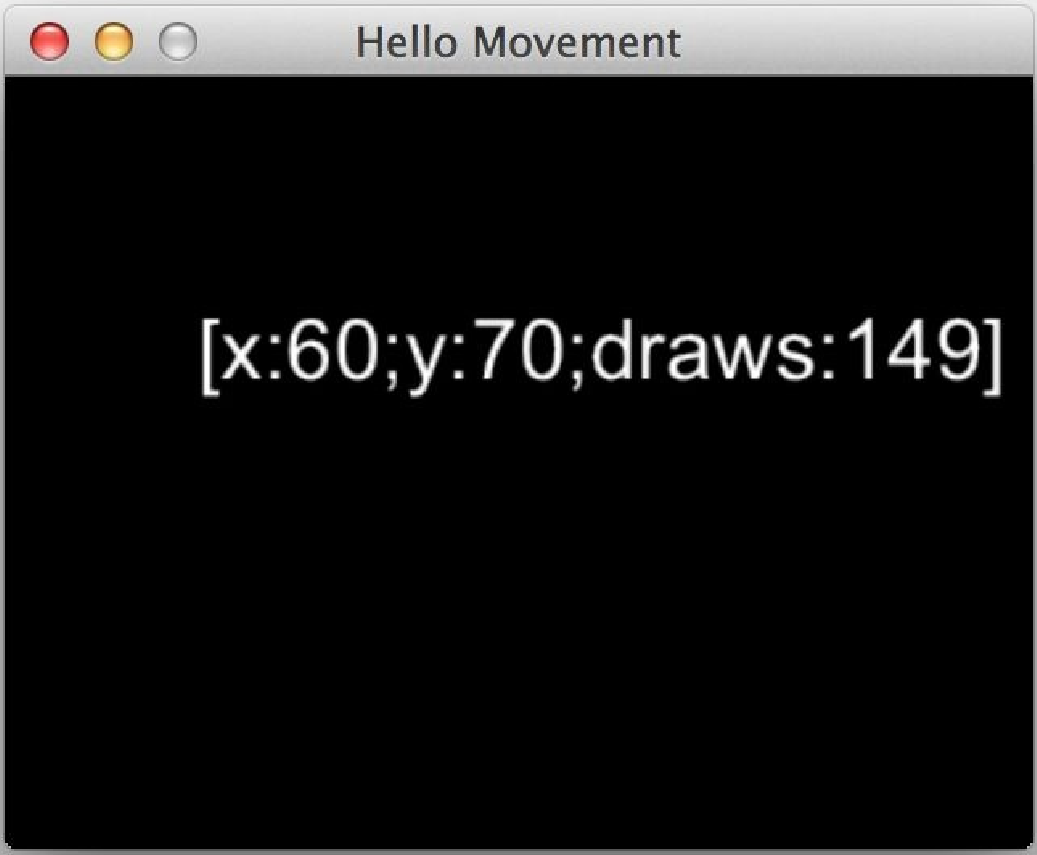
```
require 'gosu'

class GameWindow < Gosu::Window
  def initialize(width=320, height=240, fullscreen=false)
    super
    self.caption = 'Hello Movement'
    @x = @y = 10
    @draws = 0
    @buttons_down = 0
  end

  def update
    @x -= 1 if button_down?(Gosu::KbLeft)
    @x += 1 if button_down?(Gosu::KbRight)
```

```ruby
    @y -= 1 if button_down?(Gosu::KbUp)
    @y += 1 if button_down?(Gosu::KbDown)
  end

  def button_down(id)
    close if id == Gosu::KbEscape
    @buttons_down += 1
  end

  def button_up(id)
    @buttons_down -= 1
  end

  def needs_redraw?
    @draws == 0 || @buttons_down > 0
  end

  def draw
    @draws += 1
    @message = Gosu::Image.from_text(
      self, info, Gosu.default_font_name, 30)
    @message.draw(@x, @y, 0)
  end

  private

  def info
    "[x:#{@x};y:#{@y};draws:#{@draws}]"
  end
end

window = GameWindow.new
window.show
```

Run the program and try pressing arrow keys:

```
$ ruby 01-hello/hello_movement.rb
```

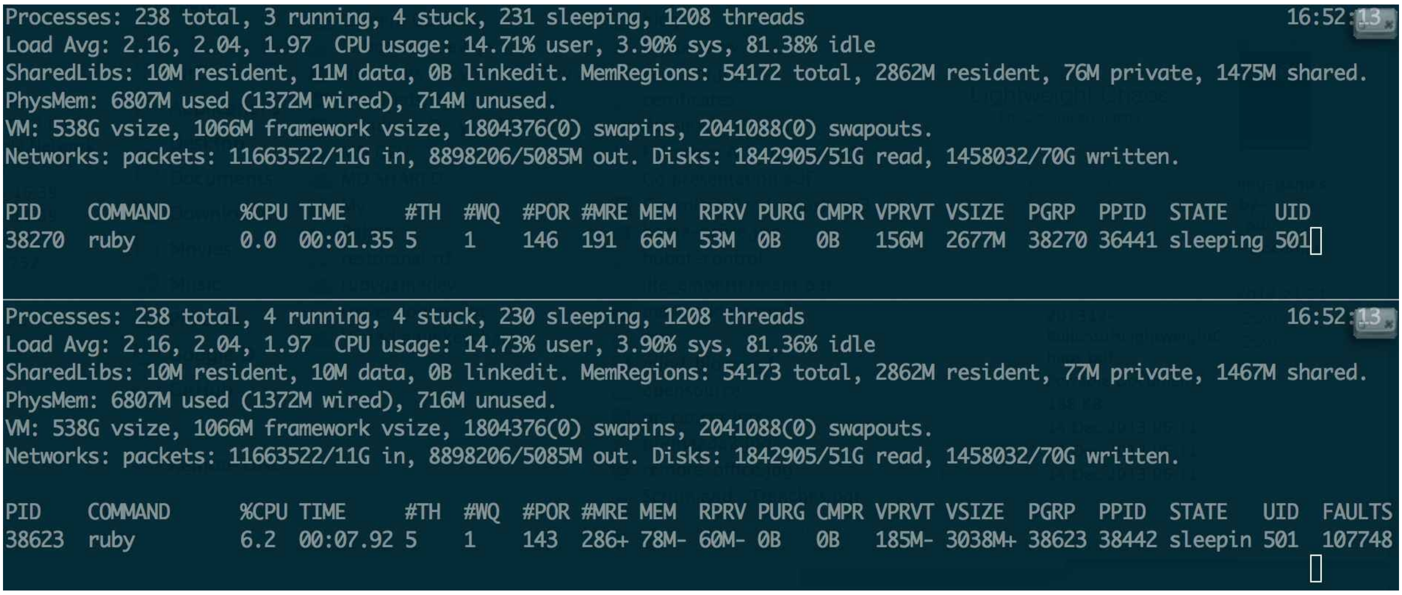
The message will move around as long as you keep arrow keys pressed.



Use arrow keys to move the message around

We could write a shorter version, but the point here is that if we wouldn’t override `needs_redraw?` this program would be slower by order of magnitude, because it would create `@message` object every time it wants to redraw the window, even though nothing would change.

Here is a screenshot of `top` displaying two versions of this program. Second screen has `needs_redraw?` method removed. See the difference?



```
Processes: 238 total, 3 running, 4 stuck, 231 sleeping, 1208 threads                                     16:52:13
Load Avg: 2.16, 2.04, 1.97  CPU usage: 14.71% user, 3.90% sys, 81.38% idle
SharedLibs: 10M resident, 11M data, 0B linkedit. MemRegions: 54172 total, 2862M resident, 76M private, 1475M shared.
PhysMem: 6807M used (1372M wired), 714M unused.
VM: 538G vsize, 1066M framework vsize, 1804376(0) swapins, 2041088(0) swapouts.
Networks: packets: 11663522/11G in, 8898206/5085M out. Disks: 1842905/51G read, 1458032/70G written.

PID    COMMAND      %CPU TIME     #TH  #WQ  #POR #MRE MEM  RPRV PURG CMPR VPRVT VSIZE  PGRP  PPID  STATE    UID
38270  ruby         0.0  00:01.35 5    1    146  191  66M  53M  0B   0B   156M  2677M  38270 36441 sleeping 501

Processes: 238 total, 4 running, 4 stuck, 230 sleeping, 1208 threads                                     16:52:13
Load Avg: 2.16, 2.04, 1.97  CPU usage: 14.73% user, 3.90% sys, 81.36% idle
SharedLibs: 10M resident, 10M data, 0B linkedit. MemRegions: 54173 total, 2862M resident, 77M private, 1467M shared.
PhysMem: 6807M used (1372M wired), 716M unused.
VM: 538G vsize, 1066M framework vsize, 1804376(0) swapins, 2041088(0) swapouts.
Networks: packets: 11663522/11G in, 8898206/5085M out. Disks: 1842905/51G read, 1458032/70G written.

PID    COMMAND      %CPU TIME     #TH  #WQ  #POR #MRE MEM  RPRV PURG CMPR VPRVT VSIZE  PGRP  PPID  STATE    UID  FAULTS
38623  ruby         6.2  00:07.92 5    1    143  286+ 78M- 60M- 0B   0B   185M- 3038M+ 38623 38442 sleepin  501  107748
```

Redrawing only when necessary VS redrawing every time

Ruby is slow, so you have to use it wisely.

Images And Animation

It's time to make something more exciting. Our game will have to have explosions, therefore we need to learn to animate them. We will set up a background scene and trigger explosions on top of it with our mouse.

01-hello/hello_animation.rb

```ruby
require 'gosu'

def media_path(file)
  File.join(File.dirname(File.dirname(
    __FILE__)), 'media', file)
end

class Explosion
  FRAME_DELAY = 10 # ms
  SPRITE = media_path('explosion.png')

  def self.load_animation(window)
    Gosu::Image.load_tiles(
      window, SPRITE, 128, 128, false)
  end

  def initialize(animation, x, y)
    @animation = animation
    @x, @y = x, y
    @current_frame = 0
  end

  def update
    @current_frame += 1 if frame_expired?
  end

  def draw
    return if done?
    image = current_frame
    image.draw(
      @x - image.width / 2.0,
      @y - image.height / 2.0,
      0)
  end

  def done?
    @done ||= @current_frame == @animation.size
  end
```

```ruby

  private

  def current_frame
    @animation[@current_frame % @animation.size]
  end

  def frame_expired?
    now = Gosu.milliseconds
    @last_frame ||= now
    if (now - @last_frame) > FRAME_DELAY
      @last_frame = now
    end
  end
end

class GameWindow < Gosu::Window
  BACKGROUND = media_path('country_field.png')

  def initialize(width=800, height=600, fullscreen=false)
    super
    self.caption = 'Hello Animation'
    @background = Gosu::Image.new(
      self, BACKGROUND, false)
    @animation = Explosion.load_animation(self)
    @explosions = []
  end

  def update
    @explosions.reject!(&:done?)
    @explosions.map(&:update)
  end

  def button_down(id)
    close if id == Gosu::KbEscape
    if id == Gosu::MsLeft
      @explosions.push(
        Explosion.new(
          @animation, mouse_x, mouse_y))
    end
  end

  def needs_cursor?
    true
  end

  def needs_redraw?
    !@scene_ready || @explosions.any?
  end

  def draw
    @scene_ready ||= true
    @background.draw(0, 0, 0)
    @explosions.map(&:draw)
  end
end

window = GameWindow.new
window.show
```

Run it and click around to enjoy those beautiful special effects:

```
$ ruby 01-hello/hello_animation.rb
```

Multiple explosions on screen

Now let's figure out how it works. Our `GameWindow` initializes with `@background` `Gosu::Image` and `@animation`, that holds array of `Gosu::Image` instances, one for each frame of explosion. `Gosu::Image.load_tiles` handles it for us.

`Explosion::SPRITE` points to "tileset" image, which is just a regular image that contains equally sized smaller image frames arranged in ordered sequence. Rows of frames are read left to right, like you would read a book.

Explosion tileset

Given that `explosion.png` tileset is `1024x1024` pixels big, and it has 8 rows of 8 tiles per row, it is easy to tell that there are 64 tiles `128x128` pixels each. So, `@animation[0]` holds `128x128` [`Gosu::Image`]() with top-left tile, and `@animation[63]` - the bottom-right one.

Gosu doesn't handle animation, it's something you have full control over. We have to draw each tile in a sequence ourselves. You can also use tiles to hold map graphics The logic behind this is pretty simple:

1. `Explosion` knows it's `@current_frame` number. It begins with 0.
2. `Explosion#frame_expired?` checks the last time when `@current_frame` was rendered, and when it is older than `Explosion::FRAME_DELAY` milliseconds, `@current_frame` is increased.
3. When [`GameWindow#update`]() is called, `@current_frame` is recalculated for all `@explosions`. Also, explosions that have finished their animation (displayed the last

frame) are removed from `@explosions` array.

4. [`GameWindow#draw`] draws background image and all `@explosions` draw their `current_frame`.
5. Again, we are saving resources and not redrawing when there are no `@explosions` in progress. `needs_redraw?` handles it.

It is important to understand that `update` and `draw` order is unpredictable, these methods can be called by your system at different rate, you can't tell which one will be called more often than the other one, so `update` should only be concerned with advancing object state, and `draw` should only draw current state on screen if it is needed. The only reliable thing here is time, consult `Gosu.milliseconds` to know how much time have passed.

Rule of the thumb: `draw` should be as lightweight as possible. Prepare all calculations in `update` and you will have responsive, smooth graphics.

Music And Sound

Our previous program was clearly missing a soundtrack, so we will add one. A background music will be looping, and each explosion will become audible.

01-hello/hello_sound.rb

```ruby
require 'gosu'

def media_path(file)
  File.join(File.dirname(File.dirname(
    __FILE__)), 'media', file)
end

class Explosion
  FRAME_DELAY = 10 # ms
  SPRITE = media_path('explosion.png')

  def self.load_animation(window)
    Gosu::Image.load_tiles(
      window, SPRITE, 128, 128, false)
  end

  def self.load_sound(window)
    Gosu::Sample.new(
      window, media_path('explosion.mp3'))
  end

  def initialize(animation, sound, x, y)
    @animation = animation
    sound.play
    @x, @y = x, y
    @current_frame = 0
  end

  def update
    @current_frame += 1 if frame_expired?
  end

  def draw
    return if done?
    image = current_frame
    image.draw(
      @x - image.width / 2.0,
      @y - image.height / 2.0,
      0)
  end

  def done?
    @done ||= @current_frame == @animation.size
```

```
  end

  def sound
    @sound.play
  end

  private

  def current_frame
    @animation[@current_frame % @animation.size]
  end

  def frame_expired?
    now = Gosu.milliseconds
    @last_frame ||= now
    if (now - @last_frame) > FRAME_DELAY
      @last_frame = now
    end
  end
end

class GameWindow < Gosu::Window
  BACKGROUND = media_path('country_field.png')

  def initialize(width=800, height=600, fullscreen=false)
    super
    self.caption = 'Hello Animation'
    @background = Gosu::Image.new(
      self, BACKGROUND, false)
    @music = Gosu::Song.new(
      self, media_path('menu_music.mp3'))
    @music.volume = 0.5
    @music.play(true)
    @animation = Explosion.load_animation(self)
    @sound = Explosion.load_sound(self)
    @explosions = []
  end

  def update
    @explosions.reject!(&:done?)
    @explosions.map(&:update)
  end

  def button_down(id)
    close if id == Gosu::KbEscape
    if id == Gosu::MsLeft
      @explosions.push(
        Explosion.new(
          @animation, @sound, mouse_x, mouse_y))
    end
  end

  def needs_cursor?
    true
  end

  def needs_redraw?
    !@scene_ready || @explosions.any?
  end

  def draw
    @scene_ready ||= true
    @background.draw(0, 0, 0)
    @explosions.map(&:draw)
  end
end

window = GameWindow.new
window.show
```

Run it and enjoy the cinematic experience. Adding sound really makes a difference.

```
$ ruby 01-hello/hello_sound.rb
```

We only added couple of things over previous example.

```
@music = Gosu::Song.new(
  self, media_path('menu_music.mp3'))
@music.volume = 0.5
@music.play(true)
```

`GameWindow` creates [`Gosu::Song`] with `menu_music.mp3`, adjusts the volume so it's a little more quiet and starts playing in a loop.

```
def self.load_sound(window)
  Gosu::Sample.new(
    window, media_path('explosion.mp3'))
end
```

`Explosion` has now got `load_sound` method that loads `explosion.mp3` sound effect `Gosu::Sample`. This sound effect is loaded once in `GameWindow` constructor, and passed into every new `Explosion`, where it simply starts playing.

Handling audio with Gosu is very straightforward. Use `Gosu::Song` to play background music, and `Gosu::Sample` to play effects and sounds that can overlap.

Warming Up

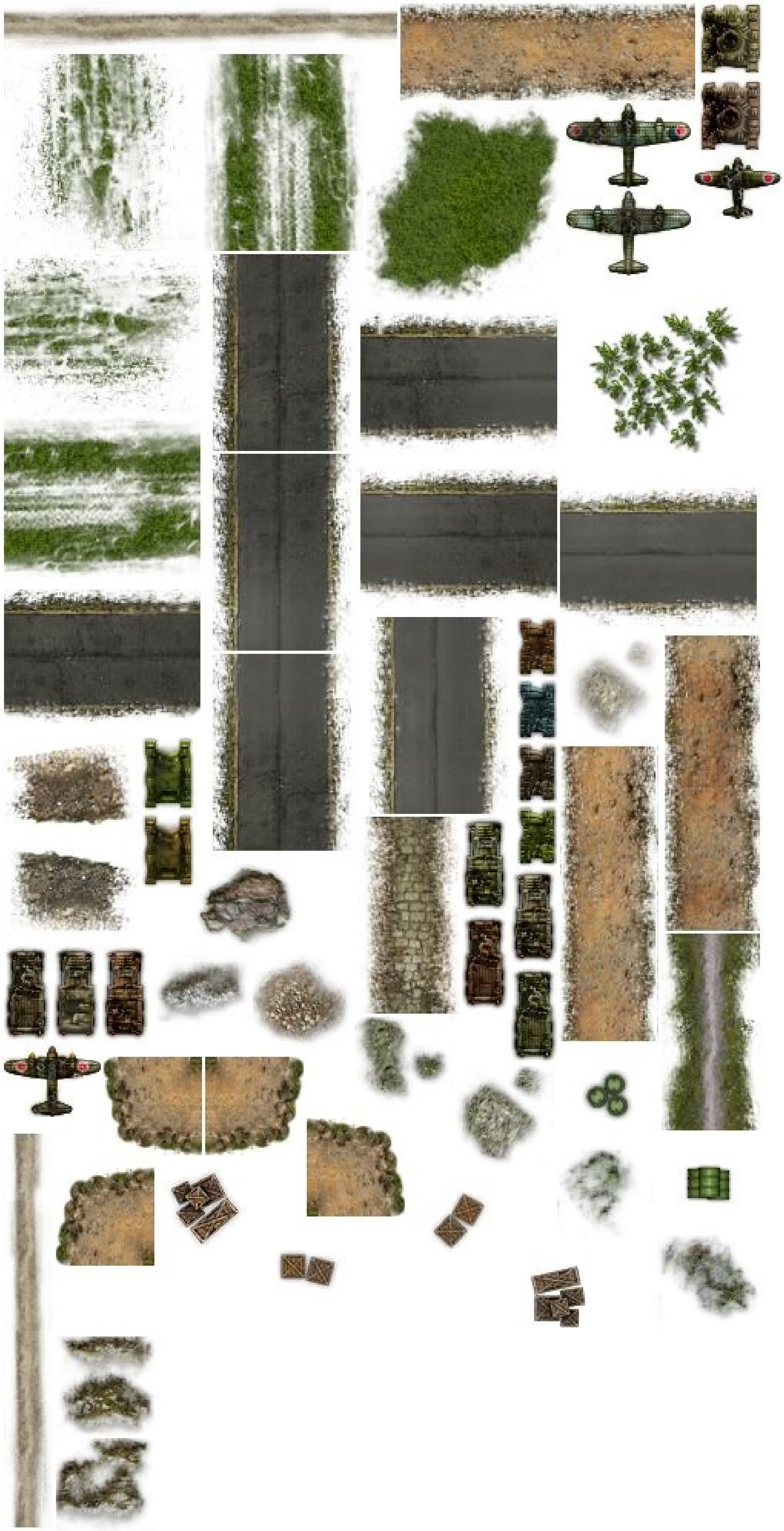
Before we start building our game, we want to flex our skills little more, get to know Gosu better and make sure our tools will be able to meet our expectations.

Using Tilesets

After playing around with Gosu for a while, we should be comfortable enough to implement a prototype of top-down view game map using the tileset of our choice. This ground tileset looks like a good place to start.

Integrating With Texture Packer

After downloading and extracting the tileset, it's obvious that `Gosu::Image#load_tiles` will not suffice, since it only supports tiles of same size, and there is a tileset in the package that looks like this:



Tileset with tiles of irregular size

And there is also a JSON file that contains some metadata:

```
{"frames": {
"aircraft_1d_destroyed.png":
{
  "frame": {"x":451,"y":102,"w":57,"h":42},
  "rotated": false,
  "trimmed": false,
  "spriteSourceSize": {"x":0,"y":0,"w":57,"h":42},
  "sourceSize": {"w":57,"h":42}
},
"aircraft_2d_destroyed.png":
{
  "frame": {"x":2,"y":680,"w":63,"h":47},
  "rotated": false,
  "trimmed": false,
  "spriteSourceSize": {"x":0,"y":0,"w":63,"h":47},
  "sourceSize": {"w":63,"h":47}
},
...
}},
"meta": {
        "app": "http://www.texturepacker.com",
        "version": "1.0",
        "image": "decor.png",
        "format": "RGBA8888",
        "size": {"w":512,"h":1024},
        "scale": "1",
        "smartupdate": "$TexturePacker:SmartUpdate:2e6b6964f24c7abfaa85a804e2dc1b05$"
}
```

Looks like these tiles were packed with [Texture Packer](). After some digging I've discovered that Gosu doesn't have any integration with it, so I had these choices:

1. Cut the original tileset image into smaller images.
2. Parse JSON and harness the benefits of Texture Packer.

First option was too much work and would prove to be less efficient, because loading many small files is always worse than loading one bigger file. Therefore, second option was the winner, and I also thought "why not write a gem while I'm at it". And that's exactly what I did, and you should do the same in such a situation. The gem is available on GitHub:

https://github.com/spajus/gosu-texture-packer

You can install this gem using `gem install gosu_texture_packer`. If you want to examine the code, easiest way is to clone it on your computer:

```
$ git clone git@github.com:spajus/gosu-texture-packer.git
```

Let's examine the main idea behind this gem. Here is a slightly simplified version that does handles everything in under 20 lines of code:

02-warmup/tileset.rb

```ruby
require 'json'
class Tileset
  def initialize(window, json)
    @json = JSON.parse(File.read(json))
    image_file = File.join(
      File.dirname(json), @json['meta']['image'])
    @main_image = Gosu::Image.new(
      @window, image_file, true)
  end

  def frame(name)
    f = @json['frames'][name]['frame']
```

```
    @main_image.subimage(
      f['x'], f['y'], f['w'], f['h'])
  end
end
```

If by now you are familiar with Gosu documentation, you will wonder what the hell is `Gosu::Image#subimage`. At the point of writing it was not documented, and I accidentally discovered it while digging through Gosu source code.

I'm lucky this function existed, because I was ready to bring out the heavy artillery and use RMagick to extract those tiles. We will probably need RMagick at some point of time later, but it's better to avoid dependencies as long as possible.

Combining Tiles Into A Map

With tileset loading issue out of the way, we can finally get back to drawing that cool map of ours.

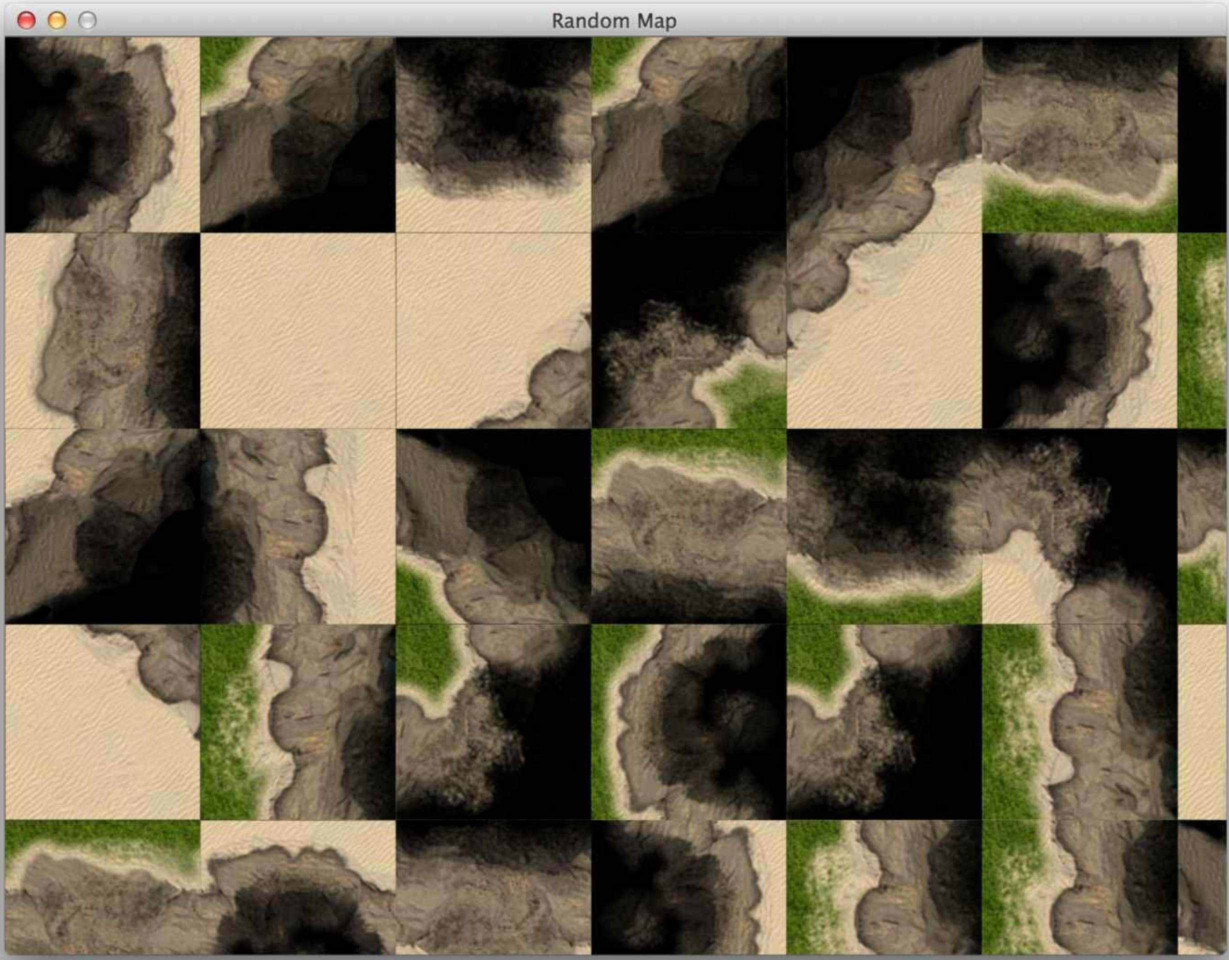
The following program will fill the screen with random tiles.

02-warmup/random_map.rb

```
require 'gosu'
require 'gosu_texture_packer'

def media_path(file)
  File.join(File.dirname(File.dirname(
    __FILE__)), 'media', file)
end

class GameWindow < Gosu::Window
  WIDTH = 800
  HEIGHT = 600
  TILE_SIZE = 128

  def initialize
    super(WIDTH, HEIGHT, false)
    self.caption = 'Random Map'
    @tileset = Gosu::TexturePacker.load_json(
      self, media_path('ground.json'), :precise)
    @redraw = true
  end

  def button_down(id)
    close if id == Gosu::KbEscape
    @redraw = true if id == Gosu::KbSpace
  end

  def needs_redraw?
    @redraw
  end

  def draw
    @redraw = false
    (0..WIDTH / TILE_SIZE).each do |x|
      (0..HEIGHT / TILE_SIZE).each do |y|
        @tileset.frame(
          @tileset.frame_list.sample).draw(
            x * (TILE_SIZE),
            y * (TILE_SIZE),
            0)
      end
    end
  end
end

window = GameWindow.new
window.show
```

Run it, then press spacebar to refill the screen with random tiles.

```
$ ruby 02-warmup/random_map.rb
```



Map filled with random tiles

The result doesn't look seamless, so we will have to figure out what's wrong. After playing around for a while, I've noticed that it's an issue with `Gosu::Image`.

When you load a tile like this, it works perfectly:

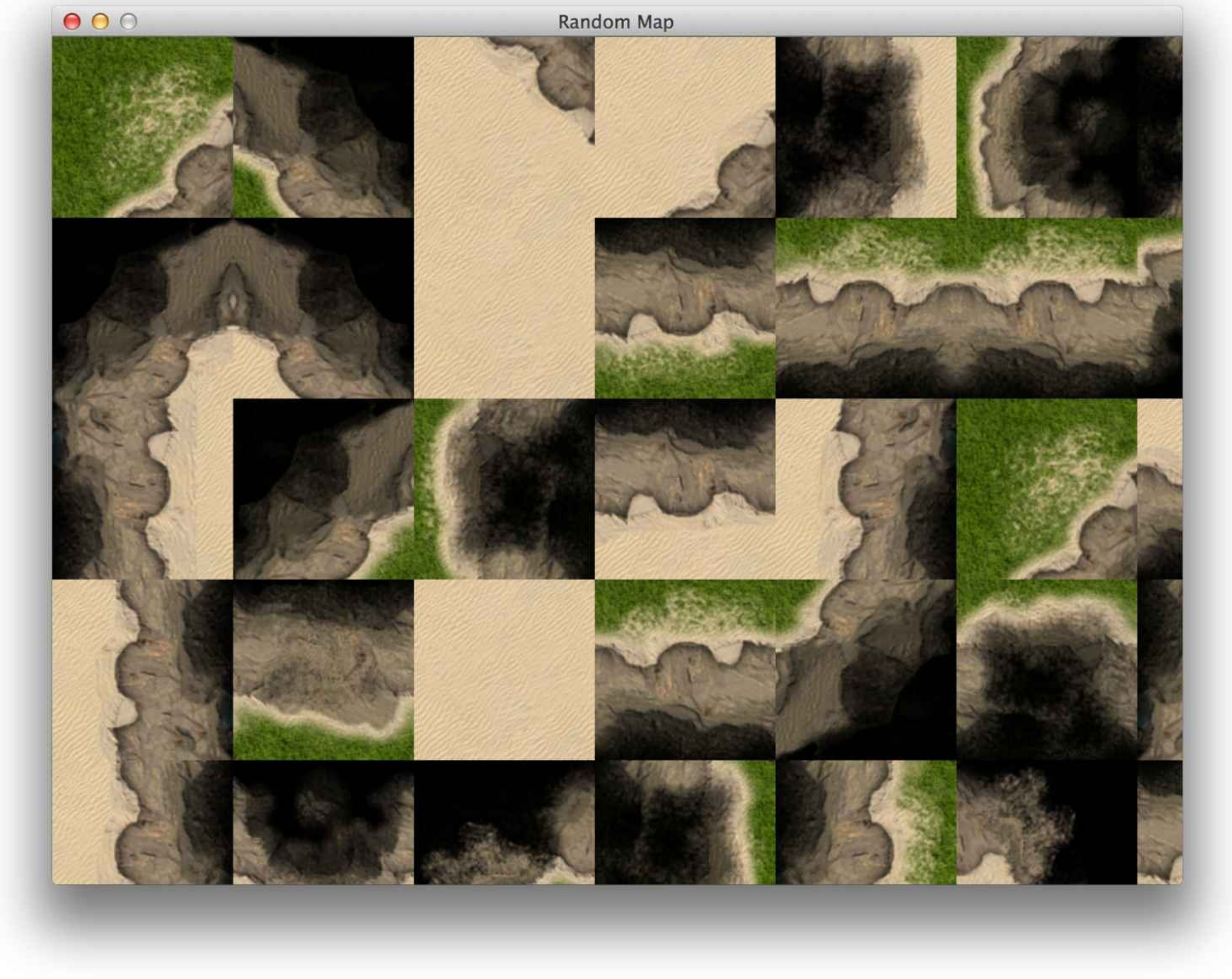
```
Gosu::Image.new(self, image_path, true, 0, 0, 128, 128)
Gosu::Image.load_tiles(self, image_path, 128, 128, true)
```

And the following produces so called "texture bleeding":

```
Gosu::Image.new(self, image_path, true)
Gosu::Image.new(self, image_path, true).subimage(0, 0, 128, 128)
```

Good thing we're not building our game yet, right? Welcome to the intricacies of software development!

Now, I [have reported my findings](), but until it gets fixed, we need a workaround. And the workaround was to use RMagick. I knew we won't get too far away from it. But our random map now looks gorgeous:

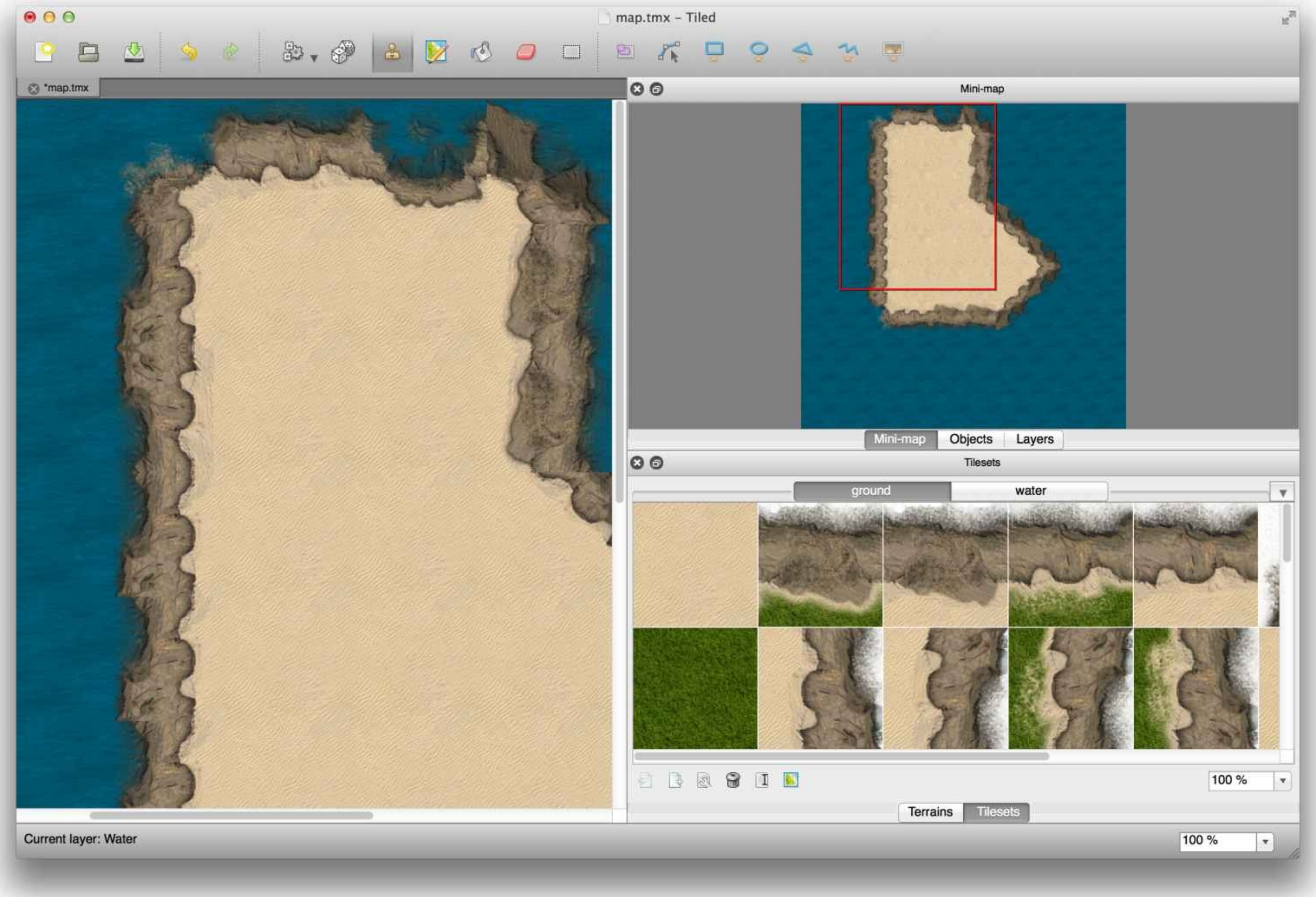


Map filled with *seamless* random tiles

Using Tiled To Create Maps

While low level approach to drawing tiles in screen may be appropriate in some scenarios, like randomly generated maps, we will explore another alternatives. One of them is this great, open source, cross platform, generic tile map editor called Tiled.

It has some limitations, for instance, all tiles in tileset have to be of same proportions. On the upside, it would be easy to load Tiled tilesets with `Gosu::Image#load_tiles`.



Tiled

Tiled uses it's own custom, XML based `tmx` format for saving maps. It also allows exporting maps to JSON, which is way more convenient, since parsing XML in Ruby is usually done with [Nokogiri](), which is heavier and it's native extensions usually cause more trouble than ones JSON parser uses. So, let's see how that JSON looks like:

02-warmup/tiled_map.json

```
{ "height":10,
 "layers":[
        {
         "data":[65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 0, 0, 65, 6\
5, 65, 65, 65, 65, 65, 65, 0, 0, 65, 65, 65, 65, 65, 65, 65, 65, 65, 0, 0, 0, 65, 65\
, 65, 65, 65, 65, 65, 0, 0, 0, 0, 65, 65, 65, 65, 65, 65, 0, 0, 0, 0, 65, 65, 65\
, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65\
, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65, 65\
],
         "height":10,
         "name":"Water",
         "opacity":1,
         "type":"tilelayer",
         "visible":true,
         "width":10,
         "x":0,
         "y":0
        },
        {
         "data":[0, 0, 7, 5, 57, 43, 0, 0, 0, 0, 0, 0, 28, 1, 1, 42, 0, 0, 0, 0,\
 0, 0, 44, 1, 1, 42, 0, 0, 0, 0, 0, 0, 28, 1, 1, 27, 43, 0, 0, 0, 0, 0, 28, 1, 1\
, 1, 27, 43, 0, 0, 0, 0, 28, 1, 1, 1, 59, 16, 0, 0, 0, 0, 48, 62, 61, 61, 16, 0,\
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\
, 0, 0, 0, 0, 0],
         "height":10,
         "name":"Ground",
         "opacity":1,
         "type":"tilelayer",
```

```
          "visible":true,
          "width":10,
          "x":0,
          "y":0
         }],
 "orientation":"orthogonal",
 "properties":
    {

    },
 "tileheight":128,
 "tilesets":[
        {
         "firstgid":1,
         "image":"media\/ground.png",
         "imageheight":1024,
         "imagewidth":1024,
         "margin":0,
         "name":"ground",
         "properties":
            {

            },
         "spacing":0,
         "tileheight":128,
         "tilewidth":128
        },
        {
         "firstgid":65,
         "image":"media\/water.png",
         "imageheight":128,
         "imagewidth":128,
         "margin":0,
         "name":"water",
         "properties":
            {

            },
         "spacing":0,
         "tileheight":128,
         "tilewidth":128
        }],
 "tilewidth":128,
 "version":1,
 "width":10
}
```

There are following things listed here:

- Two different tilesets, “ground” and “water”
- Map width and height in tile count (10x10)
- Layers with data array contains tile numbers

Couple of extra things that Tiled maps can have:

- Object layers containing lists of objects with their coordinates
- Properties hash on tiles and objects

This doesn’t look too difficult to parse, so we’re going to implement a loader for Tiled maps. And make it open source, of course.

Loading Tiled Maps With Gosu

Probably the easiest way to load Tiled map is to take each layer and render it on screen, tile by tile, like a cake. We will not care about caching at this point, and the only optimization would be not drawing things that are out of screen boundaries.

After couple of days of test driven development, I've ended up writing gosu_tiled gem, that allows you to load Tiled maps with just a few lines of code.

I will not go through describing the implementation, but if you want to examine the thought process, take a look at `gosu_tiled` gem's git commit history.

To use the gem, do `gem install gosu_tiled` and examine the code that shows a map of the island that you can scroll around with arrow keys:

02-warmup/island.rb

```ruby
require 'gosu'
require 'gosu_tiled'

class GameWindow < Gosu::Window
  MAP_FILE = File.join(File.dirname(
    __FILE__), 'island.json')
  SPEED = 5

  def initialize
    super(640, 480, false)
    @map = Gosu::Tiled.load_json(self, MAP_FILE)
    @x = @y = 0
    @first_render = true
  end

  def button_down(id)
    close if id == Gosu::KbEscape
  end

  def update
    @x -= SPEED if button_down?(Gosu::KbLeft)
    @x += SPEED if button_down?(Gosu::KbRight)
    @y -= SPEED if button_down?(Gosu::KbUp)
    @y += SPEED if button_down?(Gosu::KbDown)
    self.caption = "#{Gosu.fps} FPS. Use arrow keys to pan"
  end

  def draw
    @first_render = false
    @map.draw(@x, @y)
  end

  def needs_redraw?
    [Gosu::KbLeft,
     Gosu::KbRight,
     Gosu::KbUp,
     Gosu::KbDown].each do |b|
      return true if button_down?(b)
    end
    @first_render
  end
end

GameWindow.new.show
```

Run it, use arrow keys to scroll the map.

```
$ ruby 02-warmup/island.rb
```

The result is quite satisfying, and it scrolls smoothly without any optimizations:

Exploring Tiled map in Gosu

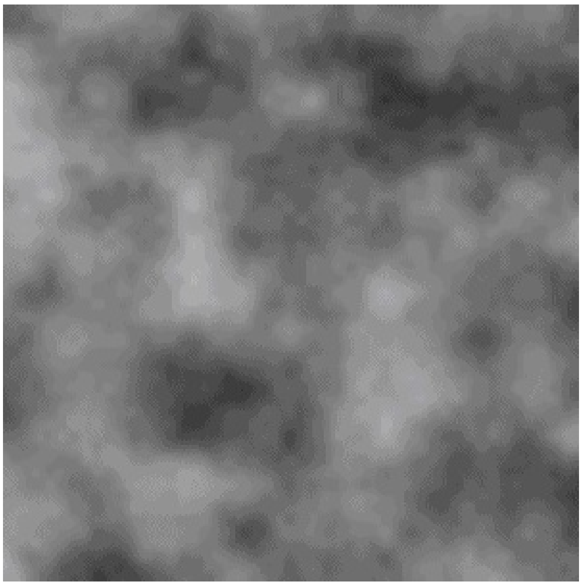
Generating Random Map With Perlin Noise

In some cases random generated maps make all the difference. Worms and Diablo would probably be just average games if it wasn't for those always unique, procedurally generated maps.

We will try to make a very primitive map generator ourselves. To begin with, we will be using only 3 different tiles - water, sand and grass. For implementing fully tiled edges, the generator must be aware of available tilesets and know how to combine them in valid ways. We may come back to it, but for now let's keep things simple.

Now, generating naturally looking randomness is something worth having a book of it's own, so instead of trying to poorly reinvent what other people have already done, we will use a well known algorithm perfectly suited for this task - Perlin noise.

If you have ever used Photoshop's Cloud filter, you already know how Perlin noise looks like:



Perlin noise

Now, we could implement the algorithm ourselves, but there is perlin_noise gem already available, it looks pretty solid, so we will use it.

The following program generates `100x100` map with 30% chance of water, 15% chance of sand and 55% chance of grass:

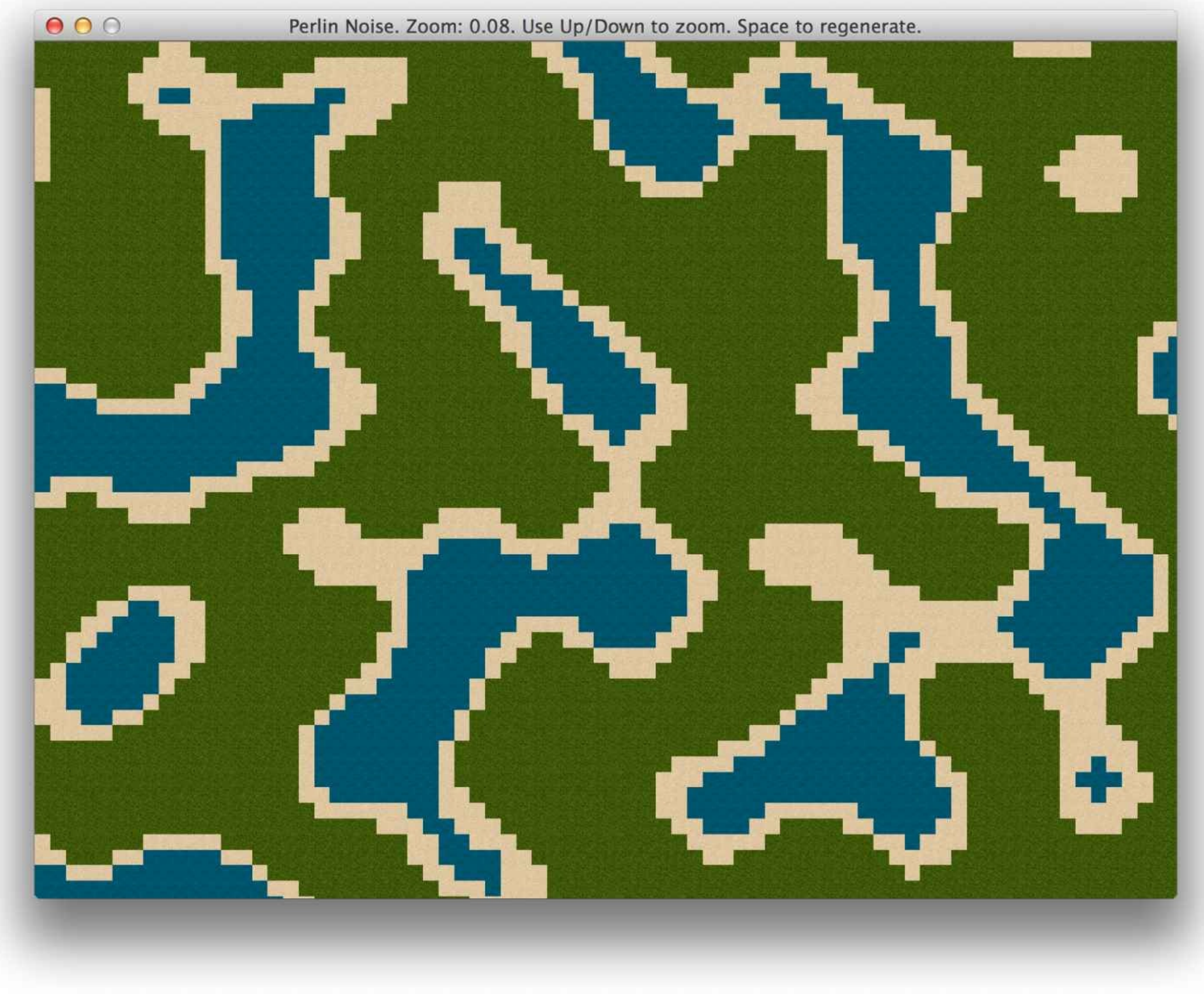
02-warmup/perlin_noise_map.rb

```
require 'gosu'
require 'gosu_texture_packer'
require 'perlin_noise'

def media_path(file)
  File.join(File.dirname(File.dirname(
    __FILE__)), 'media', file)
end

class GameWindow < Gosu::Window
  MAP_WIDTH = 100
  MAP_HEIGHT = 100
  WIDTH = 800
  HEIGHT = 600
  TILE_SIZE = 128

  def initialize
    super(WIDTH, HEIGHT, false)
    load_tiles
    @map = generate_map
    @zoom = 0.2
  end

  def button_down(id)
    close if id == Gosu::KbEscape
    @map = generate_map if id == Gosu::KbSpace
  end

  def update
    adjust_zoom(0.005) if button_down?(Gosu::KbDown)
    adjust_zoom(-0.005) if button_down?(Gosu::KbUp)
    set_caption
  end

  def draw
    tiles_x.times do |x|
      tiles_y.times do |y|
        @map[x][y].draw(
          x * TILE_SIZE * @zoom,
          y * TILE_SIZE * @zoom,
          0,
```

```ruby
          @zoom,
          @zoom)
      end
    end
  end

  private

  def set_caption
    self.caption = 'Perlin Noise. ' <<
      "Zoom: #{'%.2f' % @zoom}. " <<
      'Use Up/Down to zoom. Space to regenerate.'
  end

  def adjust_zoom(delta)
    new_zoom = @zoom + delta
    if new_zoom > 0.07 && new_zoom < 2
      @zoom = new_zoom
    end
  end

  def load_tiles
    tiles = Gosu::Image.load_tiles(
      self, media_path('ground.png'), 128, 128, true)
    @sand = tiles[0]
    @grass = tiles[8]
    @water = Gosu::Image.new(
      self, media_path('water.png'), true)
  end

  def tiles_x
    count = (WIDTH / (TILE_SIZE * @zoom)).ceil + 1
    [count, MAP_WIDTH].min
  end

  def tiles_y
    count = (HEIGHT / (TILE_SIZE * @zoom)).ceil + 1
    [count, MAP_HEIGHT].min
  end

  def generate_map
    noises = Perlin::Noise.new(2)
    contrast = Perlin::Curve.contrast(
      Perlin::Curve::CUBIC, 2)
    map = {}
    MAP_WIDTH.times do |x|
      map[x] = {}
      MAP_HEIGHT.times do |y|
        n = noises[x * 0.1, y * 0.1]
        n = contrast.call(n)
        map[x][y] = choose_tile(n)
      end
    end
    map
  end

  def choose_tile(val)
    case val
    when 0.0..0.3 # 30% chance
      @water
    when 0.3..0.45 # 15% chance, water edges
      @sand
    else # 55% chance
      @grass
    end
  end

end

window = GameWindow.new
window.show
```

Run the program, zoom with up / down arrows and regenerate everything with spacebar.

```
$ ruby 02-warmup/perlin_noise_map.rb
```



Map generated with Perlin noise

This is a little longer than our previous examples, so we will analyze some parts to make it clear.

```
def generate_map
  noises = Perlin::Noise.new(2)
  contrast = Perlin::Curve.contrast(
    Perlin::Curve::CUBIC, 2)
  map = {}
  MAP_WIDTH.times do |x|
    map[x] = {}
    MAP_HEIGHT.times do |y|
      n = noises[x * 0.1, y * 0.1]
      n = contrast.call(n)
      map[x][y] = choose_tile(n)
    end
  end
  map
end
```

`generate_map` is the heart of this program. It creates two dimensional `Perlin::Noise` generator, then chooses a random tile for each location of the map, according to noise value. To make the map a little sharper, cubic contrast is applied to noise value before choosing the tile. Try commenting out contrast application - it will look like a boring golf course, since noise values will keep buzzing around the middle.

```
def choose_tile(val)
  case val
  when 0.0..0.3 # 30% chance
```

```ruby
    @water
  when 0.3..0.45 # 15% chance, water edges
    @sand
  else # 55% chance
    @grass
  end
end
```

Here we could go crazy if we had more different tiles to use. We could add deep waters at `0.0..0.1`, mountains at `0.9..0.95` and snow caps at `0.95..1.0`. And all this would have beautiful transitions.

Player Movement With Keyboard And Mouse

We have learned to draw maps, but we need a protagonist to explore them. It will be a tank that you can move around the island with WASD keys and use your mouse to target it's gun at things. The tank will be drawn on top of our island map, and it will be above ground, but below tree layer, so it can sneak behind palm trees. That's as close to real deal as it gets!

02-warmup/player_movement.rb

```ruby
require 'gosu'
require 'gosu_tiled'
require 'gosu_texture_packer'

class Tank
  attr_accessor :x, :y, :body_angle, :gun_angle

  def initialize(window, body, shadow, gun)
    @x = window.width / 2
    @y = window.height / 2
    @window = window
    @body = body
    @shadow = shadow
    @gun = gun
    @body_angle = 0.0
    @gun_angle = 0.0
  end

  def update
    atan = Math.atan2(320 - @window.mouse_x,
                      240 - @window.mouse_y)
    @gun_angle = -atan * 180 / Math::PI
    @body_angle = change_angle(@body_angle,
      Gosu::KbW, Gosu::KbS, Gosu::KbA, Gosu::KbD)
  end

  def draw
    @shadow.draw_rot(@x - 1, @y - 1, 0, @body_angle)
    @body.draw_rot(@x, @y, 1, @body_angle)
    @gun.draw_rot(@x, @y, 2, @gun_angle)
  end

  private

  def change_angle(previous_angle, up, down, right, left)
    if @window.button_down?(up)
      angle = 0.0
      angle += 45.0 if @window.button_down?(left)
      angle -= 45.0 if @window.button_down?(right)
    elsif @window.button_down?(down)
      angle = 180.0
      angle -= 45.0 if @window.button_down?(left)
      angle += 45.0 if @window.button_down?(right)
    elsif @window.button_down?(left)
      angle = 90.0
      angle += 45.0 if @window.button_down?(up)
      angle -= 45.0 if @window.button_down?(down)
    elsif @window.button_down?(right)
```

```
      angle = 270.0
      angle -= 45.0 if @window.button_down?(up)
      angle += 45.0 if @window.button_down?(down)
    end
    angle || previous_angle
  end
end

class GameWindow < Gosu::Window
  MAP_FILE = File.join(File.dirname(
    __FILE__), 'island.json')
  UNIT_FILE = File.join(File.dirname(File.dirname(
    __FILE__)), 'media', 'ground_units.json')
  SPEED = 5

  def initialize
    super(640, 480, false)
    @map = Gosu::Tiled.load_json(self, MAP_FILE)
    @units = Gosu::TexturePacker.load_json(
      self, UNIT_FILE, :precise)
    @tank = Tank.new(self,
      @units.frame('tank1_body.png'),
      @units.frame('tank1_body_shadow.png'),
      @units.frame('tank1_dualgun.png'))
    @x = @y = 0
    @first_render = true
    @buttons_down = 0
  end

  def needs_cursor?
    true
  end

  def button_down(id)
    close if id == Gosu::KbEscape
    @buttons_down += 1
  end

  def button_up(id)
    @buttons_down -= 1
  end

  def update
    @x -= SPEED if button_down?(Gosu::KbA)
    @x += SPEED if button_down?(Gosu::KbD)
    @y -= SPEED if button_down?(Gosu::KbW)
    @y += SPEED if button_down?(Gosu::KbS)
    @tank.update
    self.caption = "#{Gosu.fps} FPS. " <<
      'Use WASD and mouse to control tank'
  end

  def draw
    @first_render = false
    @map.draw(@x, @y)
    @tank.draw()
  end
end

GameWindow.new.show
```

Tank sprite is rendered in the middle of screen. It consists of three layers, body shadow, body and gun. Body and it's shadow are always rendered in same angle, one on top of another. The angle is determined by keys that are pressed. It supports 8 directions.

Gun is a little bit different. It follows mouse cursor. To determine the angle we had to use some math. The formula to get angle in degrees is `arctan(delta_x / delta_y) * 180 / PI`. You can see it explained in more detail [on stackoverflow](#).

Run it and stroll around the island. You can still move on water and into the darkness, away from the map itself, but we will handle it later.

```
$ ruby 02-warmup/player_movement.rb
```

See that tank hiding between the bushes, ready to go in 8 directions and blow things up with that precisely aimed double cannon?

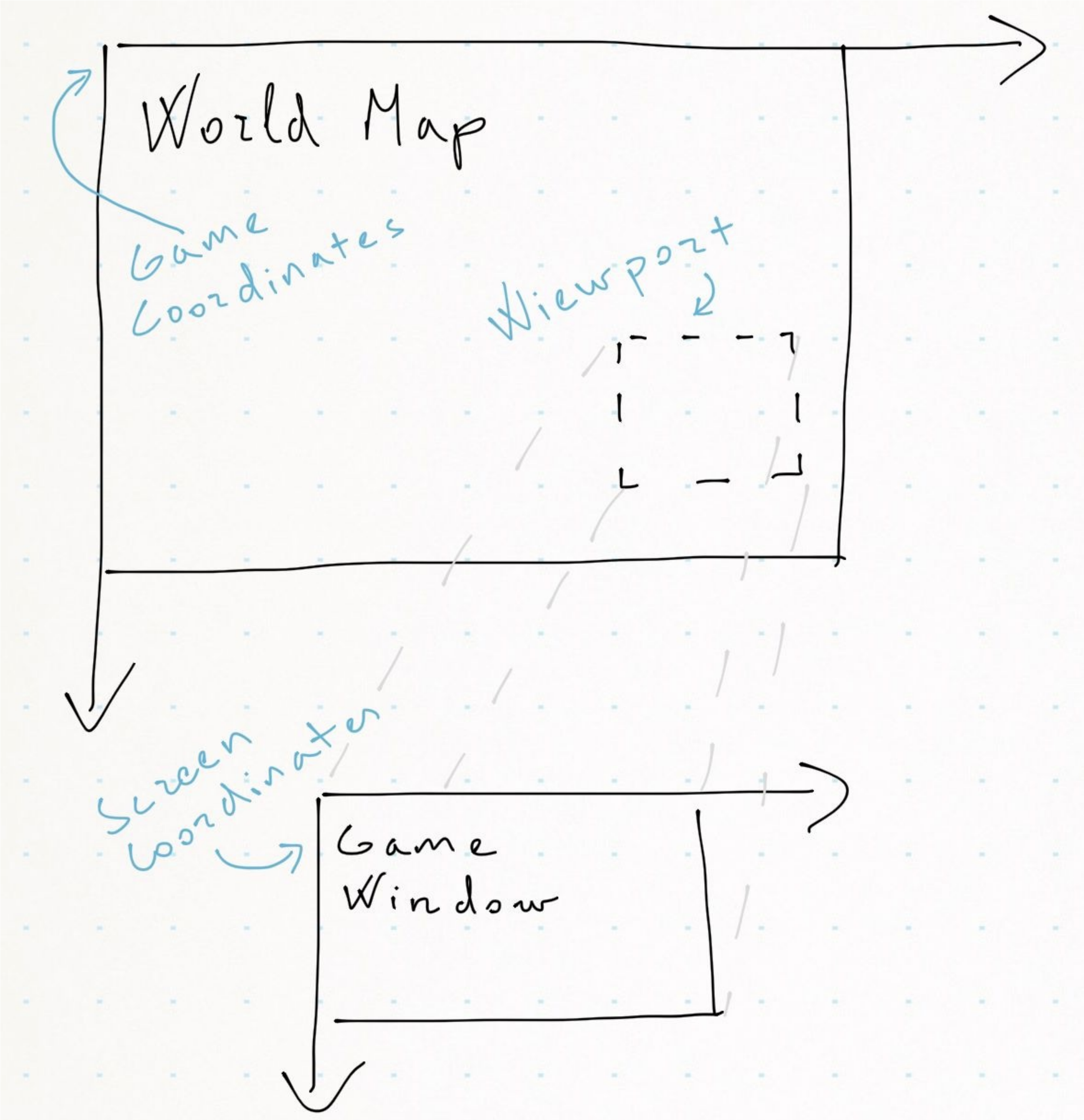
Tank moving around and aiming guns

Game Coordinate System

By now we may start realizing, that there is one key component missing in our designs. We have a virtual map, which is bigger than our screen space, and we should perform all calculations using that map, and only then cut out the required piece and render it in our game window.

There are three different coordinate systems that have to map with each other:

1. Game coordinates
2. Viewport coordinates
3. Screen coordinates



Coordinate systems

Game Coordinates

This is where all logic will happen. Player location, enemy locations, powerup locations - all this will have game coordinates, and it should have nothing to do with your screen position.

Viewport Coordinates

Viewport is the position of virtual camera, that is "filming" world in action. Don't confuse it with screen coordinates, because viewport will not necessarily be mapped pixel to pixel to your game window. Imagine this: you have a huge world map, your player is standing in the middle, and game window displays the player while slowly zooming in. In this scenario, viewport is constantly shrinking, while game map stays the same, and game window also stays the same.

Screen Coordinates

This is your game display, pixel by pixel. You will draw static information, like your HUD directly on it.

How To Put It All Together

In our games we will want to separate game coordinates from viewport and screen as much as possible. Basically, we will program ourselves a “camera man” who will be busy following the action, zooming in and out, perhaps changing the view angle now and then.

Let’s implement a prototype that will allow us to navigate and zoom around a big map. We will only draw objects that are visible in viewport. Some math will be unavoidable, but in most cases it’s pretty basic - that’s the beauty of 2D games:

02-warmup/coordinate_system.rb

```ruby
require 'gosu'

class WorldMap
  attr_accessor :on_screen, :off_screen

  def initialize(width, height)
    @images = {}
    (0..width).step(50) do |x|
      @images[x] = {}
      (0..height).step(50) do |y|
        img = Gosu::Image.from_text(
          $window, "#{x}:#{y}",
          Gosu.default_font_name, 15)
        @images[x][y] = img
      end
    end
  end

  def draw(camera)
    @on_screen = @off_screen = 0
    @images.each do |x, row|
      row.each do |y, val|
        if camera.can_view?(x, y, val)
          val.draw(x, y, 0)
          @on_screen += 1
        else
          @off_screen += 1
        end
      end
    end
  end
end

class Camera
  attr_accessor :x, :y, :zoom

  def initialize
    @x = @y = 0
    @zoom = 1
  end

  def can_view?(x, y, obj)
    x0, x1, y0, y1 = viewport
    (x0 - obj.width..x1).include?(x) &&
      (y0 - obj.height..y1).include?(y)
  end

  def viewport
```

```
    x0 = @x - ($window.width / 2)  / @zoom
    x1 = @x + ($window.width / 2)  / @zoom
    y0 = @y - ($window.height / 2) / @zoom
    y1 = @y + ($window.height / 2) / @zoom
    [x0, x1, y0, y1]
  end

  def to_s
    "FPS: #{Gosu.fps}. " <<
      "#{@x}:#{@y} @ #{'%.2f' % @zoom}. " <<
      'WASD to move, arrows to zoom.'
  end

  def draw_crosshair
    $window.draw_line(
      @x - 10, @y, Gosu::Color::YELLOW,
      @x + 10, @y, Gosu::Color::YELLOW, 100)
    $window.draw_line(
      @x, @y - 10, Gosu::Color::YELLOW,
      @x, @y + 10, Gosu::Color::YELLOW, 100)
  end
end


class GameWindow < Gosu::Window
  SPEED = 10

  def initialize
    super(800, 600, false)
    $window = self
    @map = WorldMap.new(2048, 1024)
    @camera = Camera.new
  end

  def button_down(id)
    close if id == Gosu::KbEscape
    if id == Gosu::KbSpace
      @camera.zoom = 1.0
      @camera.x = 0
      @camera.y = 0
    end
  end

  def update
    @camera.x -= SPEED if button_down?(Gosu::KbA)
    @camera.x += SPEED if button_down?(Gosu::KbD)
    @camera.y -= SPEED if button_down?(Gosu::KbW)
    @camera.y += SPEED if button_down?(Gosu::KbS)

    zoom_delta = @camera.zoom > 0 ? 0.01 : 1.0

    if button_down?(Gosu::KbUp)
      @camera.zoom -= zoom_delta
    end
    if button_down?(Gosu::KbDown)
      @camera.zoom += zoom_delta
    end
    self.caption = @camera.to_s
  end

  def draw
    off_x = -@camera.x + width / 2
    off_y = -@camera.y + height / 2
    cam_x = @camera.x
    cam_y = @camera.y
    translate(off_x, off_y) do
      @camera.draw_crosshair
      zoom = @camera.zoom
      scale(zoom, zoom, cam_x, cam_y) do
        @map.draw(@camera)
      end
    end
    info = 'Objects on/off screen: ' <<
      "#{@map.on_screen}/#{@map.off_screen}"
    info_img = Gosu::Image.from_text(
      self, info, Gosu.default_font_name, 30)
```

```
    info_img.draw(10, 10, 1)
  end
end

GameWindow.new.show
```

Run it, use WASD to navigate, up / down arrows to zoom and spacebar to reset the camera.

```
$ ruby 02-warmup/coordinate_system.rb
```

It doesn’t look impressive, but understanding the concept of different coordinate systems and being able to stitch them together is paramount to the success of our final product.

Prototype of separate coordinate systems

Luckily for us, Gosu helps us by providing `Gosu::Window#translate` that handles camera offset, `Gosu::Window#scale` that aids zooming, and `Gosu::Window#rotate` that was not used yet, but will be great for shaking the view to emphasize explosions.

Prototyping The Game

Warming up was really important, but let's combine everything we learned, add some new challenges, and build a small prototype with following features:

1. Camera loosely follows tank.
2. Camera zooms automatically depending on tank speed.
3. You can temporarily override automatic camera zoom using keyboard.
4. Music and sound effects.
5. Randomly generated map.
6. Two modes: menu and gameplay.
7. Tank movement with WADS keys.
8. Tank aiming and shooting with mouse.
9. Collision detection (tanks don't swim).
10. Explosions, visible bullet trajectories.
11. Bullet range limiting.

Sounds fun? Hell yes! However, before we start, we should plan ahead a little and think how our game architecture will look like. We will also structure our code a little, so it will not be smashed into one ruby class, as we did in earlier examples. Books should show good manners!

Switching Between Game States

First, let's think how to hook into `Gosu::Window`. Since we will have two game states, State pattern naturally comes to mind.

So, our `GameWindow` class could look like this:

03-prototype/game_window.rb

```ruby
class GameWindow < Gosu::Window

  attr_accessor :state

  def initialize
    super(800, 600, false)
  end

  def update
    @state.update
  end

  def draw
    @state.draw
  end

  def needs_redraw?
    @state.needs_redraw?
  end

  def button_down(id)
    @state.button_down(id)
```

```ruby
  end

end
```

It has current `@state`, and all usual main loop actions are executed on that state instance. We will add base class that all game states will extend. Let's name it `GameState`:

03-prototype/states/game_state.rb

```ruby
class GameState

  def self.switch(new_state)
    $window.state && $window.state.leave
    $window.state = new_state
    new_state.enter
  end

  def enter
  end

  def leave
  end

  def draw
  end

  def update
  end

  def needs_redraw?
    true
  end

  def button_down(id)
  end
end
```

This class provides `GameState.switch`, that will change the state for our `Gosu::Window`, and all `enter` and `leave` methods when appropriate. These methods will be useful for things like switching music.

Notice that `Gosu::Window` is accessed using global `$window` variable, which will be considered an anti-pattern by most good programmers, but there is some logic behind this:

1. There will be only one `Gosu::Window` instance.
2. It lives as long as the game runs.
3. It is used in some way by nearly all other classes, so we would have to pass it around all the time.
4. Accessing it using Singleton or static utility class would not give any clear benefits, just add more complexity.

[Chingu](https://github.com/ippa/chingu), another game framework built on top of Gosu, also uses global `$window`, so it's probably not the worst idea ever.

We will also need an entry point that would fire up the game and enter the first game state - the menu.

03-prototype/main.rb

```ruby
require 'gosu'
require_relative 'states/game_state'
require_relative 'states/menu_state'
require_relative 'states/play_state'
```

```ruby
require_relative 'game_window'

module Game
  def self.media_path(file)
    File.join(File.dirname(File.dirname(
      __FILE__)), 'media', file)
  end
end

$window = GameWindow.new
GameState.switch(MenuState.instance)
$window.show
```

In our entry point we also have a small helper which will help loading images and sounds using `Game.media_path`.

The rest is obvious: we create `GameWindow` instance and store it in `$window` variable, as discussed before. Then we use `GameState.switch)` to load `MenuState`, and show the game window.

Implementing Menu State

This is how simple `MenuState` implementation looks like:

03-prototype/states/menu_state.rb

```ruby
require 'singleton'
class MenuState < GameState
  include Singleton
  attr_accessor :play_state

  def initialize
    @message = Gosu::Image.from_text(
      $window, "Tanks Prototype",
      Gosu.default_font_name, 100)
  end

  def enter
    music.play(true)
    music.volume = 1
  end

  def leave
    music.volume = 0
    music.stop
  end

  def music
    @@music ||= Gosu::Song.new(
      $window, Game.media_path('menu_music.mp3'))
  end

  def update
    continue_text = @play_state ? "C = Continue, " : ""
    @info = Gosu::Image.from_text(
      $window, "Q = Quit, #{continue_text}N = New Game",
      Gosu.default_font_name, 30)
  end

  def draw
    @message.draw(
      $window.width / 2 - @message.width / 2,
      $window.height / 2 - @message.height / 2,
      10)
    @info.draw(
      $window.width / 2 - @info.width / 2,
      $window.height / 2 - @info.height / 2 + 200,
      10)
  end

```

```
  def button_down(id)
    $window.close if id == Gosu::KbQ
    if id == Gosu::KbC && @play_state
      GameState.switch(@play_state)
    end
    if id == Gosu::KbN
      @play_state = PlayState.new
      GameState.switch(@play_state)
    end
  end
end
```

It’s a [Singleton](), so we can always get it with `MenuState.instance`.

It starts playing `menu_music.mp3` when you enter the menu, and stop the music when you leave it. Instance of [`Gosu::Song`]() is cached in `@@music` class variable to save resources.

We have to know if play is already in progress, so we can add a possibility to go back to the game. That’s why `MenuState` has `@play_state` variable, and either allows creating new `PlayState` when `N` key is pressed, or switches to existing `@play_state` if `C` key is pressed.

Here comes the interesting part, implementing the play state.

Implementing Play State

Before we start implementing actual gameplay, we need to think what game entities we will be building. We will need a `Map` that will hold our tiles and provide world coordinate system. We will also need a `Camera` that will know how to float around and zoom. There will be `Bullets` flying around, and each bullet will eventually cause an `Explosion`.

Having all that taken care of, `PlayState` should look pretty simple:

03-prototype/states/play_state.rb

```
require_relative '../entities/map'
require_relative '../entities/tank'
require_relative '../entities/camera'
require_relative '../entities/bullet'
require_relative '../entities/explosion'
class PlayState < GameState

  def initialize
    @map = Map.new
    @tank = Tank.new(@map)
    @camera = Camera.new(@tank)
    @bullets = []
    @explosions = []
  end

  def update
    bullet = @tank.update(@camera)
    @bullets << bullet if bullet
    @bullets.map(&:update)
    @bullets.reject!(&:done?)
    @camera.update
    $window.caption = 'Tanks Prototype. ' <<
      "[FPS: #{Gosu.fps}. Tank @ #{@tank.x.round}:#{@tank.y.round}]"
  end

  def draw
    cam_x = @camera.x
    cam_y = @camera.y
    off_x =  $window.width / 2 - cam_x
    off_y =  $window.height / 2 - cam_y
    $window.translate(off_x, off_y) do
      zoom = @camera.zoom
```

```ruby
      $window.scale(zoom, zoom, cam_x, cam_y) do
        @map.draw(@camera)
        @tank.draw
        @bullets.map(&:draw)
      end
    end
    @camera.draw_crosshair
  end

  def button_down(id)
    if id == Gosu::MsLeft
      bullet = @tank.shoot(*@camera.mouse_coords)
      @bullets << bullet if bullet
    end
    $window.close if id == Gosu::KbQ
    if id == Gosu::KbEscape
      GameState.switch(MenuState.instance)
    end
  end

end
```

Update and draw calls are passed to the underlying game entities, so they can handle them the way they want it to. Such encapsulation reduces complexity of the code and allows doing every piece of logic where it belongs, while keeping it short and simple.

There are a few interesting parts in this code. Both `@tank.update` and `@tank.shoot` may produce a new bullet, if your tank's fire rate is not exceeded, and if left mouse button is kept down, hence the `update`. If bullet is produced, it is added to `@bullets` array, and they live their own little lifecycle, until they explode and are no longer used. `@bullets.reject!(&:done?)` cleans up the garbage.

`PlayState#draw` deserves extra explanation. `@camera.x` and `@camera.y` points to *game* coordinates where `Camera` is currently looking at. [`Gosu::Window#translate`] creates a block within which all [`Gosu::Image`] draw operations are translated by given offset. [`Gosu::Window#scale`] does the same with `Camera` zoom.

Crosshair is drawn without translating and scaling it, because it's relative to screen, not to world map.

Basically, this draw method is the place that takes care drawing only what `@camera` can see.

If it's hard to understand how this works, get back to "Game Coordinate System" chapter and let it sink in.

Implementing World Map

We will start analyzing game entities with `Map`.

03-prototype/entities/map.rb

```ruby
require 'perlin_noise'
require 'gosu_texture_packer'

class Map
  MAP_WIDTH = 100
  MAP_HEIGHT = 100
  TILE_SIZE = 128

  def initialize
    load_tiles
    @map = generate_map
```

```ruby
  end

  def find_spawn_point
    while true
      x = rand(0..MAP_WIDTH * TILE_SIZE)
      y = rand(0..MAP_HEIGHT * TILE_SIZE)
      if can_move_to?(x, y)
        return [x, y]
      else
        puts "Invalid spawn point: #{[x, y]}"
      end
    end
  end

  def can_move_to?(x, y)
    tile = tile_at(x, y)
    tile && tile != @water
  end

  def draw(camera)
    @map.each do |x, row|
      row.each do |y, val|
        tile = @map[x][y]
        map_x = x * TILE_SIZE
        map_y = y * TILE_SIZE
        if camera.can_view?(map_x, map_y, tile)
          tile.draw(map_x, map_y, 0)
        end
      end
    end
  end

  private

  def tile_at(x, y)
    t_x = ((x / TILE_SIZE) % TILE_SIZE).floor
    t_y = ((y / TILE_SIZE) % TILE_SIZE).floor
    row = @map[t_x]
    row[t_y] if row
  end

  def load_tiles
    tiles = Gosu::Image.load_tiles(
      $window, Game.media_path('ground.png'),
      128, 128, true)
    @sand = tiles[0]
    @grass = tiles[8]
    @water = Gosu::Image.new(
      $window, Game.media_path('water.png'), true)
  end

  def generate_map
    noises = Perlin::Noise.new(2)
    contrast = Perlin::Curve.contrast(
      Perlin::Curve::CUBIC, 2)
    map = {}
    MAP_WIDTH.times do |x|
      map[x] = {}
      MAP_HEIGHT.times do |y|
        n = noises[x * 0.1, y * 0.1]
        n = contrast.call(n)
        map[x][y] = choose_tile(n)
      end
    end
    map
  end

  def choose_tile(val)
    case val
    when 0.0..0.3 # 30% chance
      @water
    when 0.3..0.45 # 15% chance, water edges
      @sand
    else # 55% chance
      @grass
    end
```

```
  end
end
```

This implementation is very similar to the `Map` we had built in “Generating Random Map With Perlin Noise”, with some extra additions. `can_move_to?` verifies if tile under given coordinates is not water. Pretty simple, but it’s enough for our prototype.

Also, when we `draw` the map we have to make sure if tiles we are drawing are currently visible by our camera, otherwise we will end up drawing off screen. `camera.can_view?` handles it. Current implementation will probably be causing a bottleneck, since it brute forces through all the map rather than cherry-picking the visible region. We will probably have to get back and change it later.

`find_spawn_point` is one more addition. It keeps picking a random point on map and verifies if it’s not water using `can_move_to?`. When solid tile is found, it returns the coordinates, so our `Tank` will be able to spawn there.

Implementing Floating Camera

If you played the original Grand Theft Auto or GTA 2, you should remember how fascinating the camera was. It backed away when you were driving at high speeds, closed in when you were walking on foot, and floated around as if a smart drone was following your protagonist from above.

The following `Camera` implementation is far inferior to the one GTA had nearly two decades ago, but it’s a start:

03-prototype/entities/camera.rb

```ruby
class Camera
  attr_accessor :x, :y, :zoom

  def initialize(target)
    @target = target
    @x, @y = target.x, target.y
    @zoom = 1
  end

  def can_view?(x, y, obj)
    x0, x1, y0, y1 = viewport
    (x0 - obj.width..x1).include?(x) &&
      (y0 - obj.height..y1).include?(y)
  end

  def mouse_coords
    x, y = target_delta_on_screen
    mouse_x_on_map = @target.x +
      (x + $window.mouse_x - ($window.width / 2)) / @zoom
    mouse_y_on_map = @target.y +
      (y + $window.mouse_y - ($window.height / 2)) / @zoom
    [mouse_x_on_map, mouse_y_on_map].map(&:round)
  end

  def update
    @x += @target.speed if @x < @target.x - $window.width / 4
    @x -= @target.speed if @x > @target.x + $window.width / 4
    @y += @target.speed if @y < @target.y - $window.height / 4
    @y -= @target.speed if @y > @target.y + $window.height / 4

    zoom_delta = @zoom > 0 ? 0.01 : 1.0
    if $window.button_down?(Gosu::KbUp)
      @zoom -= zoom_delta unless @zoom < 0.7
    elsif $window.button_down?(Gosu::KbDown)
      @zoom += zoom_delta unless @zoom > 10
```

```ruby
    else
      target_zoom = @target.speed > 1.1 ? 0.85 : 1.0
      if @zoom <= (target_zoom - 0.01)
        @zoom += zoom_delta / 3
      elsif @zoom > (target_zoom + 0.01)
        @zoom -= zoom_delta / 3
      end
    end
  end

  def to_s
    "FPS: #{Gosu.fps}. " <<
      "#{@x}:#{@y} @ #{'%.2f' % @zoom}. " <<
      'WASD to move, arrows to zoom.'
  end

  def target_delta_on_screen
    [(@x - @target.x) * @zoom, (@y - @target.y) * @zoom]
  end

  def draw_crosshair
    x = $window.mouse_x
    y = $window.mouse_y
    $window.draw_line(
      x - 10, y, Gosu::Color::RED,
      x + 10, y, Gosu::Color::RED, 100)
    $window.draw_line(
      x, y - 10, Gosu::Color::RED,
      x, y + 10, Gosu::Color::RED, 100)
  end

  private

  def viewport
    x0 = @x - ($window.width / 2)  / @zoom
    x1 = @x + ($window.width / 2)  / @zoom
    y0 = @y - ($window.height / 2) / @zoom
    y1 = @y + ($window.height / 2) / @zoom
    [x0, x1, y0, y1]
  end
end
```

Our `Camera` has `@target` that it tries to follow, `@x` and `@y` that it currently is looking at, and `@zoom` level.

All the magic happens in `update` method. It keeps track of the distance between `@target` and adjust itself to stay nearby. And when `@target.speed` shows some movement momentum, camera slowly backs away.

`Camera` also tels if you `can_view?` an object at some coordinates, so when other entities draw themselves, they can check if there is a need for that.

Another noteworthy method is `mouse_coords`. It translates mouse position on screen to mouse position on map, so the game will know where you are targeting your guns.

Implementing The Tank

Most of our tank code will be taken from “Player Movement With Keyboard And Mouse”:

03-prototype/entities/tank.rb

```ruby
class Tank
  attr_accessor :x, :y, :body_angle, :gun_angle
  SHOOT_DELAY = 500

  def initialize(map)
    @map = map
    @units = Gosu::TexturePacker.load_json(
```

```ruby
      $window, Game.media_path('ground_units.json'), :precise)
    @body = @units.frame('tank1_body.png')
    @shadow = @units.frame('tank1_body_shadow.png')
    @gun = @units.frame('tank1_dualgun.png')
    @x, @y = @map.find_spawn_point
    @body_angle = 0.0
    @gun_angle = 0.0
    @last_shot = 0
    sound.volume = 0.3
  end

  def sound
    @@sound ||= Gosu::Song.new(
      $window, Game.media_path('tank_driving.mp3'))
  end

  def shoot(target_x, target_y)
    if Gosu.milliseconds - @last_shot > SHOOT_DELAY
      @last_shot = Gosu.milliseconds
      Bullet.new(@x, @y, target_x, target_y).fire(100)
    end
  end

  def update(camera)
    d_x, d_y = camera.target_delta_on_screen
    atan = Math.atan2(($window.width / 2) - d_x - $window.mouse_x,
                      ($window.height / 2) - d_y - $window.mouse_y)
    @gun_angle = -atan * 180 / Math::PI
    new_x, new_y = @x, @y
    new_x -= speed if $window.button_down?(Gosu::KbA)
    new_x += speed if $window.button_down?(Gosu::KbD)
    new_y -= speed if $window.button_down?(Gosu::KbW)
    new_y += speed if $window.button_down?(Gosu::KbS)
    if @map.can_move_to?(new_x, new_y)
      @x, @y = new_x, new_y
    else
      @speed = 1.0
    end
    @body_angle = change_angle(@body_angle,
      Gosu::KbW, Gosu::KbS, Gosu::KbA, Gosu::KbD)

    if moving?
      sound.play(true)
    else
      sound.pause
    end

    if $window.button_down?(Gosu::MsLeft)
      shoot(*camera.mouse_coords)
    end
  end

  def moving?
    any_button_down?(Gosu::KbA, Gosu::KbD, Gosu::KbW, Gosu::KbS)
  end

  def draw
    @shadow.draw_rot(@x - 1, @y - 1, 0, @body_angle)
    @body.draw_rot(@x, @y, 1, @body_angle)
    @gun.draw_rot(@x, @y, 2, @gun_angle)
  end

  def speed
    @speed ||= 1.0
    if moving?
      @speed += 0.03 if @speed < 5
    else
      @speed = 1.0
    end
    @speed
  end

  private

  def any_button_down?(*buttons)
    buttons.each do |b|
```

```ruby
      return true if $window.button_down?(b)
    end
    false
  end

  def change_angle(previous_angle, up, down, right, left)
    if $window.button_down?(up)
      angle = 0.0
      angle += 45.0 if $window.button_down?(left)
      angle -= 45.0 if $window.button_down?(right)
    elsif $window.button_down?(down)
      angle = 180.0
      angle -= 45.0 if $window.button_down?(left)
      angle += 45.0 if $window.button_down?(right)
    elsif $window.button_down?(left)
      angle = 90.0
      angle += 45.0 if $window.button_down?(up)
      angle -= 45.0 if $window.button_down?(down)
    elsif $window.button_down?(right)
      angle = 270.0
      angle -= 45.0 if $window.button_down?(up)
      angle += 45.0 if $window.button_down?(down)
    end
    angle || previous_angle
  end
end
```

`Tank` has to be aware of the `Map` to check where it's moving, and it uses `Camera` to find out where to aim the guns. When it `shoots`, it produces instances of `Bullet`, that are simply returned to the caller. Tank won't keep track of them, it's "fire and forget".

Implementing Bullets And Explosions

Bullets will require some simple vector math. You have a point that moves along the vector with some speed. It also needs to limit the maximum vector length, so if you try to aim too far, the bullet will only go as far as it can reach.

03-prototype/entities/bullet.rb

```ruby
class Bullet
  COLOR = Gosu::Color::BLACK
  MAX_DIST = 300
  START_DIST = 20

  def initialize(source_x, source_y, target_x, target_y)
    @x, @y = source_x, source_y
    @target_x, @target_y = target_x, target_y
    @x, @y = point_at_distance(START_DIST)
    if trajectory_length > MAX_DIST
      @target_x, @target_y = point_at_distance(MAX_DIST)
    end
    sound.play
  end

  def draw
    unless arrived?
      $window.draw_quad(@x - 2, @y - 2, COLOR,
                        @x + 2, @y - 2, COLOR,
                        @x - 2, @y + 2, COLOR,
                        @x + 2, @y + 2, COLOR,
                        1)
    else
      @explosion ||= Explosion.new(@x, @y)
      @explosion.draw
    end
  end

  def update
    fly_distance = (Gosu.milliseconds - @fired_at) * 0.001 * @speed
    @x, @y = point_at_distance(fly_distance)
```

```
    @explosion && @explosion.update
  end

  def arrived?
    @x == @target_x && @y == @target_y
  end

  def done?
     exploaded?
  end

  def exploaded?
    @explosion && @explosion.done?
  end

  def fire(speed)
    @speed = speed
    @fired_at = Gosu.milliseconds
    self
  end

  private

  def sound
    @@sound ||= Gosu::Sample.new(
      $window, Game.media_path('fire.mp3'))
  end

  def trajectory_length
    d_x = @target_x - @x
    d_y = @target_y - @y
    Math.sqrt(d_x * d_x + d_y * d_y)
  end

  def point_at_distance(distance)
    return [@target_x, @target_y] if distance > trajectory_length
    distance_factor = distance.to_f / trajectory_length
    p_x = @x + (@target_x - @x) * distance_factor
    p_y = @y + (@target_y - @y) * distance_factor
    [p_x, p_y]
  end
end
```

Possibly the most interesting part of `Bullet` implementation is `point_at_distance` method. It returns coordinates of point that is between bullet source, which is point that bullet was fired from, and it's target, which is the destination point. The returned point is as far away from source point as `distance` tells it to.

After bullet has done flying, it explodes with fanfare. In our prototype `Explosion` is a part of `Bullet`, because it's the only thing that triggers it. Therefore `Bullet` has two stages of it's lifecycle. First it flies towards the target, then it's exploding. That brings us to `Explosion`:

03-prototype/entities/explosion.rb

```
class Explosion
  FRAME_DELAY = 10 # ms

  def animation
    @@animation ||=
    Gosu::Image.load_tiles(
      $window, Game.media_path('explosion.png'), 128, 128, false)
  end

  def sound
    @@sound ||= Gosu::Sample.new(
      $window, Game.media_path('explosion.mp3'))
  end

  def initialize(x, y)
```

```
    sound.play
    @x, @y = x, y
    @current_frame = 0
  end

  def update
    @current_frame += 1 if frame_expired?
  end

  def draw
    return if done?
    image = current_frame
    image.draw(
      @x - image.width / 2 + 3,
      @y - image.height / 2 - 35,
      20)
  end

  def done?
    @done ||= @current_frame == animation.size
  end

  private

  def current_frame
    animation[@current_frame % animation.size]
  end

  def frame_expired?
    now = Gosu.milliseconds
    @last_frame ||= now
    if (now - @last_frame) > FRAME_DELAY
      @last_frame = now
    end
  end
end
```

There is nothing fancy about this implementation. Most of it is taken from “Images And Animation” chapter.

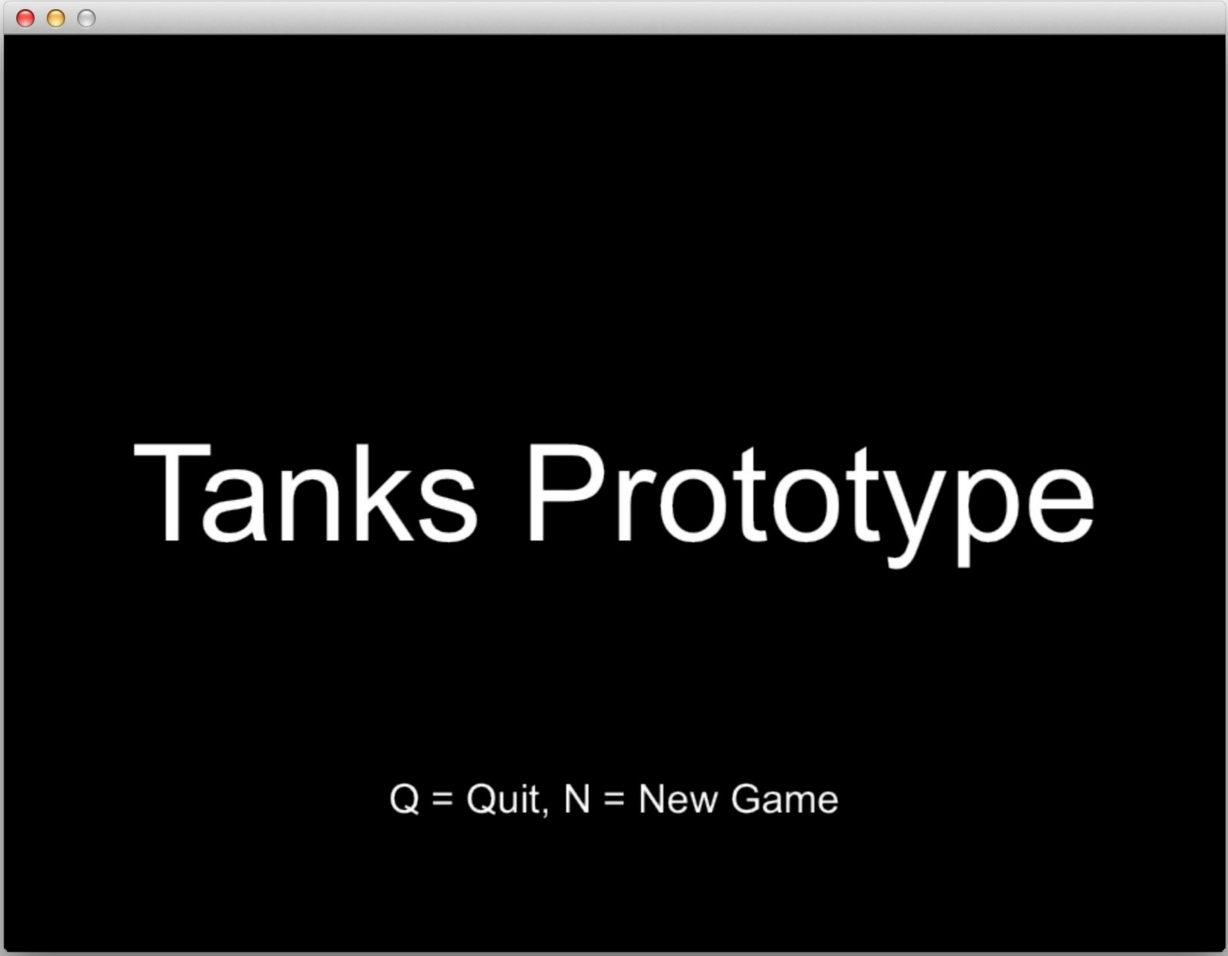
Running The Prototype

We have walked through all the code. You can get it at GitHub.

Now it’s time to give it a spin. There is a video of me playing it available on YouTube, but it’s always best to experience it firsthand. Run `main.rb` to start the game:

```
$ ruby 03-prototype/main.rb
```

Hit `N` to start new game.



Tanks Prototype menu

Time to go crazy!

Tanks Prototype gameplay

One thing should be bugging you at this point. FPS shows only 30, rather than 60. That means our prototype is slow. We will put it back to 60 FPS in next chapter.

Optimizing Game Performance

To make games that are fast and don't require a powerhouse to run, we must learn how to find and fix bottlenecks. Good news is that if you wasn't thinking about performance to begin with, your program can usually be optimized to run twice as fast just by eliminating one or two biggest bottlenecks.

We will be using a copy of the prototype code to keep both optimized and original version, therefore if you are exploring sample code, look at `04-prototype-optimized`.

Profiling Ruby Code To Find Bottlenecks

We will try to find bottlenecks in our Tanks prototype game by profiling it with [`ruby-prof`](ruby-prof).

It's a ruby gem, just install it like this:

```
$ gem install ruby-prof
```

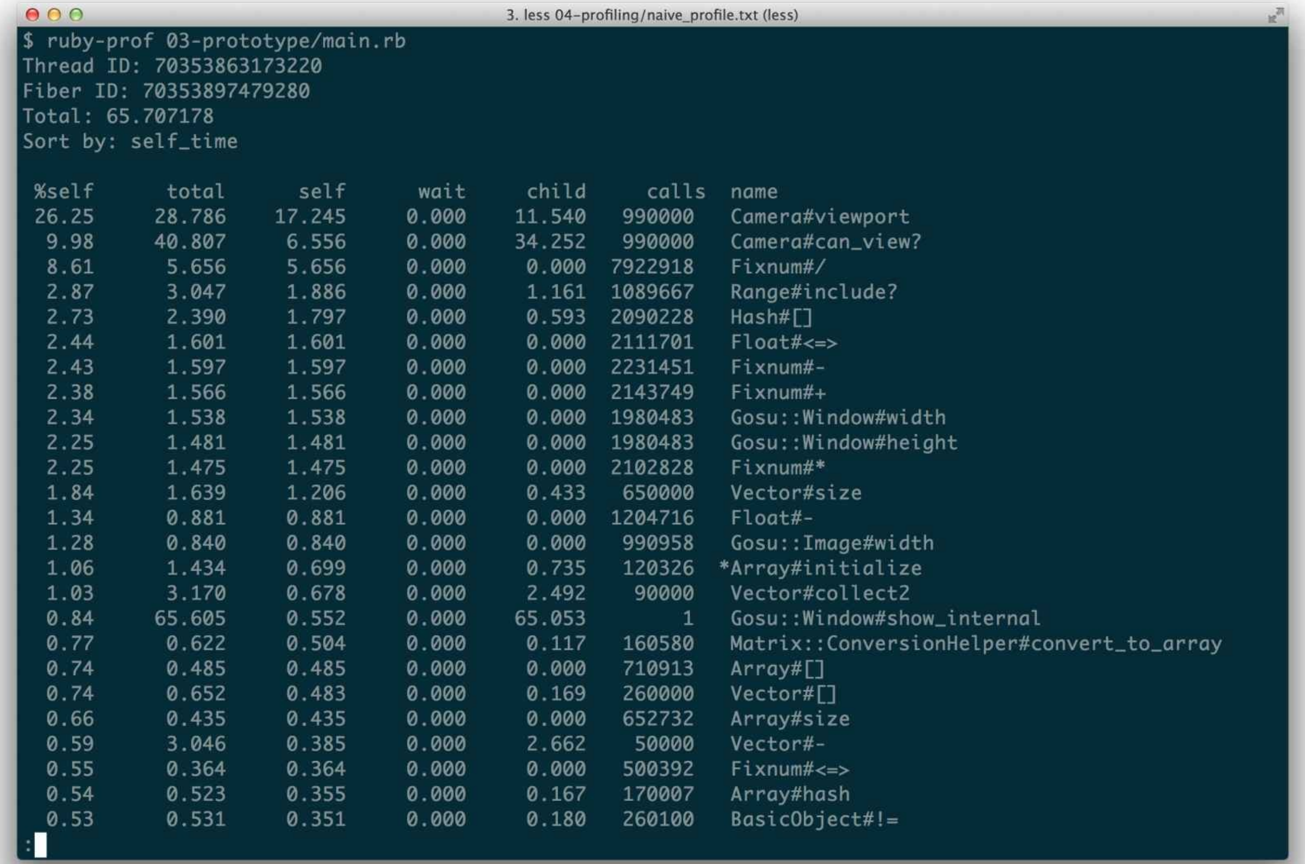
There are several ways you can use `ruby-prof`, so we will begin with the easiest one. Instead of running the game with `ruby`, we will run it with `ruby-prof`:

```
$ ruby-prof 03-prototype/main.rb
```

The game will run, but everything will be ten times slower as usual, because every call to every function is being recorded, and after you exit the program, profiling output will be dumped directly to your console.

Downside of this approach is that we are going to profile everything there is, including the super-slow map generation that uses Perlin Noise. We don't want to optimize that, so in order to find bottlenecks in our play state rather than map generation, we have to keep playing at dreadful 2 FPS for at least 30 seconds.

This was the output of first "naive" profiling session:




```
$ ruby-prof 03-prototype/main.rb
Thread ID: 70353863173220
Fiber ID: 70353897479280
Total: 65.707178
Sort by: self_time

 %self      total      self      wait     child     calls   name
 26.25     28.786    17.245     0.000    11.540    990000   Camera#viewport
  9.98     40.807     6.556     0.000    34.252    990000   Camera#can_view?
  8.61      5.656     5.656     0.000     0.000   7922918   Fixnum#/
  2.87      3.047     1.886     0.000     1.161   1089667   Range#include?
  2.73      2.390     1.797     0.000     0.593   2090228   Hash#[]
  2.44      1.601     1.601     0.000     0.000   2111701   Float#<=>
  2.43      1.597     1.597     0.000     0.000   2231451   Fixnum#-
  2.38      1.566     1.566     0.000     0.000   2143749   Fixnum#+
  2.34      1.538     1.538     0.000     0.000   1980483   Gosu::Window#width
  2.25      1.481     1.481     0.000     0.000   1980483   Gosu::Window#height
  2.25      1.475     1.475     0.000     0.000   2102828   Fixnum#*
  1.84      1.639     1.206     0.000     0.433    650000   Vector#size
  1.34      0.881     0.881     0.000     0.000   1204716   Float#-
  1.28      0.840     0.840     0.000     0.000    990958   Gosu::Image#width
  1.06      1.434     0.699     0.000     0.735    120326  *Array#initialize
  1.03      3.170     0.678     0.000     2.492     90000   Vector#collect2
  0.84     65.605     0.552     0.000    65.053         1   Gosu::Window#show_internal
  0.77      0.622     0.504     0.000     0.117    160580   Matrix::ConversionHelper#convert_to_array
  0.74      0.485     0.485     0.000     0.000    710913   Array#[]
  0.74      0.652     0.483     0.000     0.169    260000   Vector#[]
  0.66      0.435     0.435     0.000     0.000    652732   Array#size
  0.59      3.046     0.385     0.000     2.662     50000   Vector#-
  0.55      0.364     0.364     0.000     0.000    500392   Fixnum#<=>
  0.54      0.523     0.355     0.000     0.167    170007   Array#hash
  0.53      0.531     0.351     0.000     0.180    260100   BasicObject#!=
:
```

Initial profiling results

It's obvious, that `Camera#viewport` and `Camera#can_view?` are top CPU burners. This means either that our implementation is either very bad, or the assumption that checking if camera can view object is slower than drawing the object off screen.

Here are those slow methods:

```
class Camera
  # ...
  def can_view?(x, y, obj)
    x0, x1, y0, y1 = viewport
    (x0 - obj.width..x1).include?(x) &&
      (y0 - obj.height..y1).include?(y)
  end
  # ...
  def viewport
    x0 = @x - ($window.width / 2)  / @zoom
    x1 = @x + ($window.width / 2)  / @zoom
    y0 = @y - ($window.height / 2) / @zoom
    y1 = @y + ($window.height / 2) / @zoom
    [x0, x1, y0, y1]
  end
  # ...
end
```

It doesn't look fundamentally broken, so we will try our "checking is slower than rendering" hypothesis by short-circuiting `can_view?` to return `true` every time:

```
class Camera
  # ...
  def can_view?(x, y, obj)
    return true # short circuiting
    x0, x1, y0, y1 = viewport
    (x0 - obj.width..x1).include?(x) &&
      (y0 - obj.height..y1).include?(y)
```

```ruby
  end
  # ...
end
```

After saving `camera.rb` and running the game without profiling, you will notice a significant speedup. Hypothesis was correct, checking visibility is more expensive than simply rendering it. That means we can throw away `Camera#can_view?` and calls to it.

But before doing that, let’s profile once again:

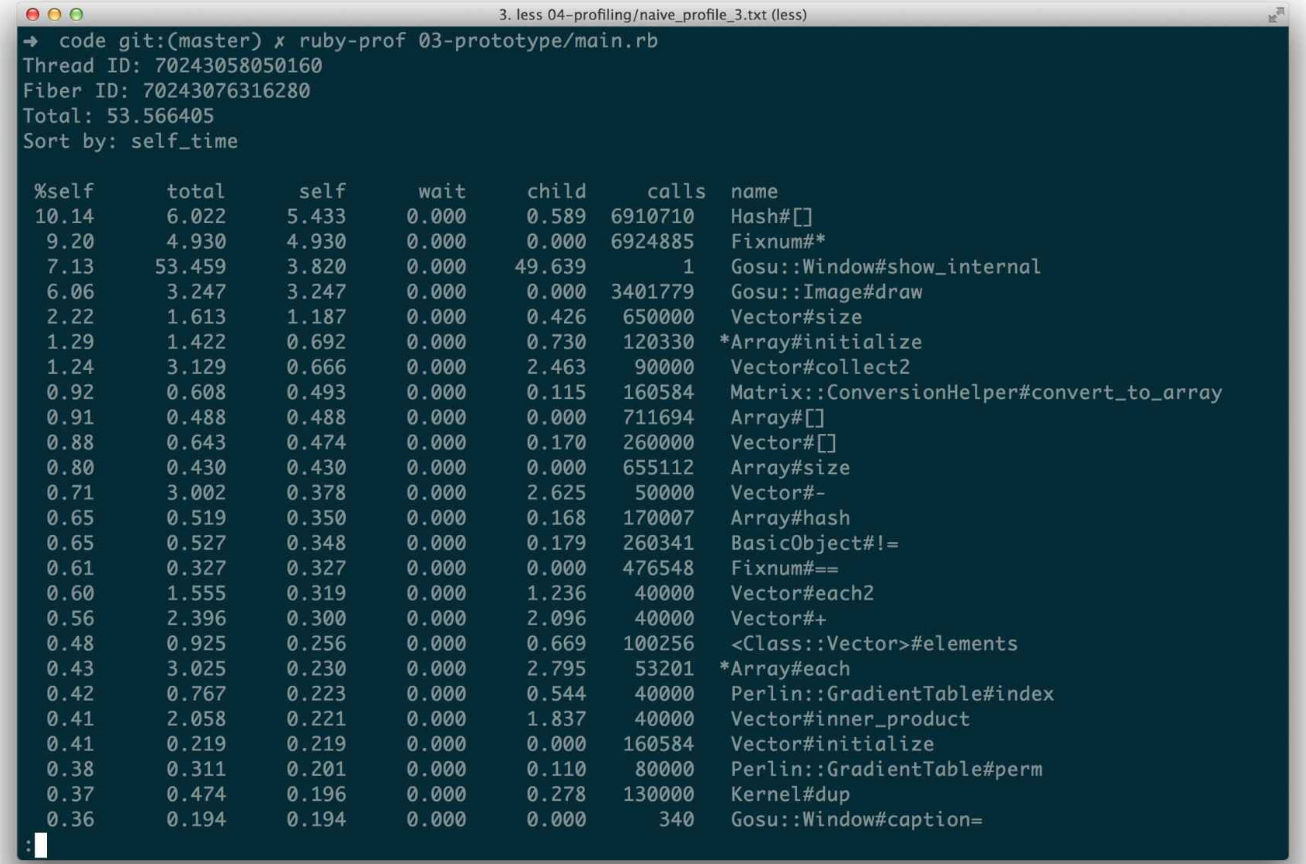
```
3. less 04-profiling/naive_profile_2.txt (less)
➜  code git:(master) ✗ ruby-prof 03-prototype/main.rb
Thread ID: 70300440323180
Fiber ID: 70300470320220
Total: 52.392213
Sort by: self_time

 %self      total      self      wait     child     calls  name
  8.43      5.036     4.418     0.000     0.618   5250544  Hash#[]
  7.15      3.748     3.748     0.000     0.000   5263582  Fixnum#*
  6.72      3.522     3.522     0.000     0.000   2570000  Camera#can_view?
  5.81     52.287     3.046     0.000    49.241         1  Gosu::Window#show_internal
  4.85      2.542     2.542     0.000     0.000   2571293  Gosu::Image#draw
  2.29      1.633     1.201     0.000     0.432    650000  Vector#size
  1.34      1.445     0.703     0.000     0.743    120331 *Array#initialize
  1.28      3.171     0.672     0.000     2.499     90000  Vector#collect2
  0.96      0.619     0.504     0.000     0.114    160585  Matrix::ConversionHelper#convert_to_array
  0.94      0.490     0.490     0.000     0.000    711238  Array#[]
  0.93      0.654     0.486     0.000     0.168    260000  Vector#[]
  0.83      0.435     0.435     0.000     0.000    653725  Array#size
  0.73      3.043     0.385     0.000     2.659     50000  Vector#-
  0.69      0.541     0.360     0.000     0.181    170007  Array#hash
  0.67      0.537     0.350     0.000     0.187    260258  BasicObject#!=
  0.64      0.338     0.338     0.000     0.000    474832  Fixnum#==
  0.62      1.579     0.325     0.000     1.254     40000  Vector#each2
  0.58      2.433     0.306     0.000     2.127     40000  Vector#+
  0.50      0.940     0.261     0.000     0.679    100256  <Class::Vector>#elements
  0.45      3.100     0.233     0.000     2.866     52744 *Array#each
  0.43      2.093     0.226     0.000     1.867     40000  Vector#inner_product
  0.43      0.774     0.225     0.000     0.548     40000  Perlin::GradientTable#index
  0.42      0.220     0.220     0.000     0.000    160585  Vector#initialize
  0.39      0.314     0.203     0.000     0.110     80000  Perlin::GradientTable#perm
  0.38     13.908     0.197     0.000    13.711     10000  Perlin::Noise#[]
:
```

Profiling results after short-circuiting `Camera#can_view?`

We can see `Camera#can_view?` is still in top 3, so we will remove `if camera.can_view?(map_x, map_y, tile)` from `Map#draw` and for now keep it like this:

```ruby
class Map
  # ...
  def draw(camera)
    @map.each do |x, row|
      row.each do |y, val|
        tile = @map[x][y]
        map_x = x * TILE_SIZE
        map_y = y * TILE_SIZE
        tile.draw(map_x, map_y, 0)
      end
    end
  end
  # ...
end
```

After completely removing `Camera#can_view?`, profiling session looks like dead-end - no more low hanging fruits on top:



Profiling results after removing `Camera#can_view?`

The game still doesn't feel fast enough, FPS occasionally keeps dropping down to ~45, so we will have to do profile our code in smarter way.

Advanced Profiling Techniques

We would get more accuracy when profiling only what we want to optimize. In our case it is everything that happens in `PlayState`, except for `Map` generation. This time we will have to use ruby-prof API to hook into places we need.

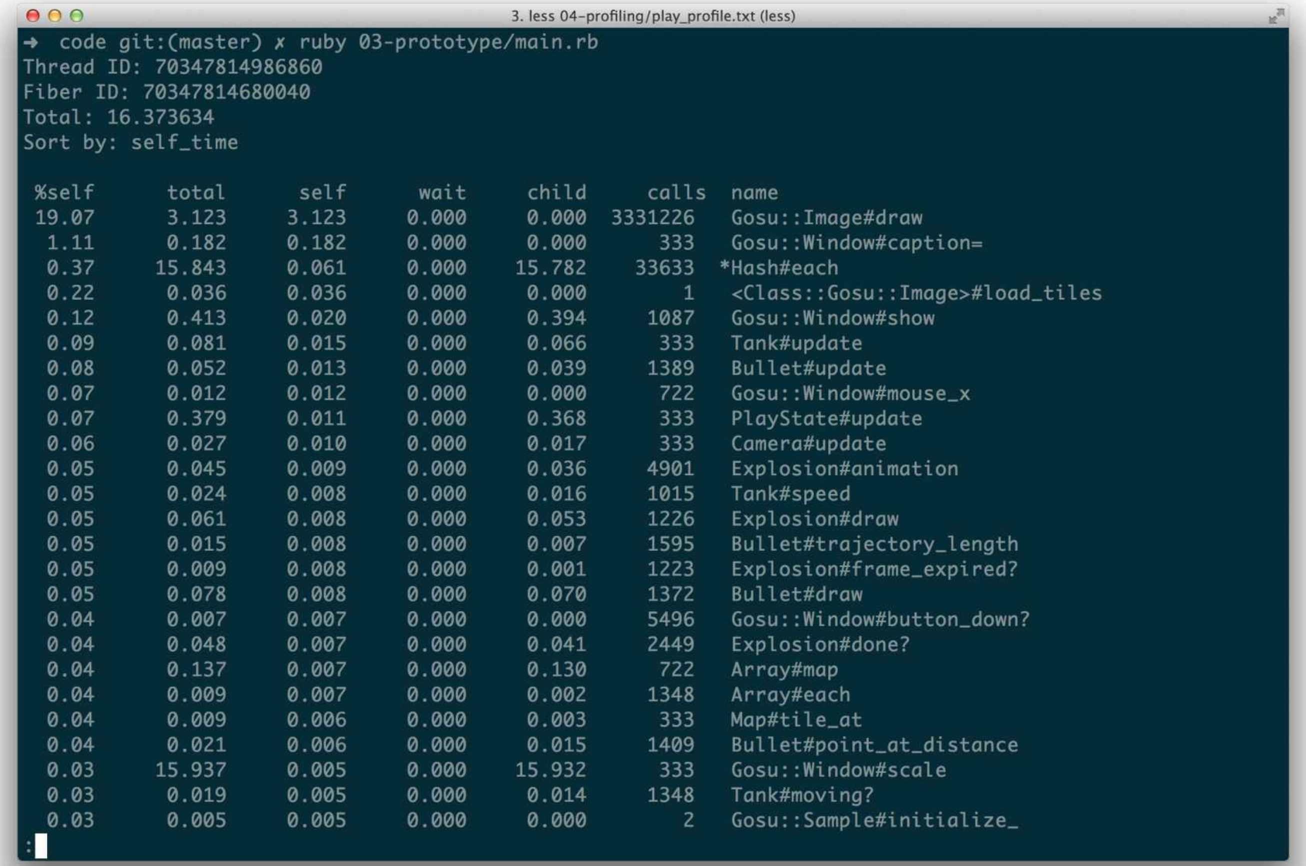
`Map` generation happens in `PlayState` initializer, so we will leverage `GameState#enter` and `GameState#leave` to start and stop profiling, since it happens after state is initialized. Here is how we hook in:

```ruby
require 'ruby-prof'
class PlayState < GameState
  # ...
  def enter
    RubyProf.start
  end

  def leave
    result = RubyProf.stop
    printer = RubyProf::FlatPrinter.new(result)
    printer.print(STDOUT)
  end
  # ...
end
```

Then we run the game as usual:

```
$ ruby 04-prototype-optimized/main.rb
```

Now, after we press `N` to start new game, `Map` generation happens relatively fast, and then profiling kicks in, FPS drops to 15. After moving around and shooting for a while we hit `Esc` to return to the menu, and at that point `PlayState#leave` spits profiling results out to the console:



```
3. less 04-profiling/play_profile.txt (less)
➜  code git:(master) ✗ ruby 03-prototype/main.rb
Thread ID: 70347814986860
Fiber ID: 70347814680040
Total: 16.373634
Sort by: self_time

 %self      total      self      wait     child     calls  name
 19.07      3.123     3.123     0.000     0.000   3331226   Gosu::Image#draw
  1.11      0.182     0.182     0.000     0.000       333   Gosu::Window#caption=
  0.37     15.843     0.061     0.000    15.782     33633  *Hash#each
  0.22      0.036     0.036     0.000     0.000         1   <Class::Gosu::Image>#load_tiles
  0.12      0.413     0.020     0.000     0.394      1087   Gosu::Window#show
  0.09      0.081     0.015     0.000     0.066       333   Tank#update
  0.08      0.052     0.013     0.000     0.039      1389   Bullet#update
  0.07      0.012     0.012     0.000     0.000       722   Gosu::Window#mouse_x
  0.07      0.379     0.011     0.000     0.368       333   PlayState#update
  0.06      0.027     0.010     0.000     0.017       333   Camera#update
  0.05      0.045     0.009     0.000     0.036      4901   Explosion#animation
  0.05      0.024     0.008     0.000     0.016      1015   Tank#speed
  0.05      0.061     0.008     0.000     0.053      1226   Explosion#draw
  0.05      0.015     0.008     0.000     0.007      1595   Bullet#trajectory_length
  0.05      0.009     0.008     0.000     0.001      1223   Explosion#frame_expired?
  0.05      0.078     0.008     0.000     0.070      1372   Bullet#draw
  0.04      0.007     0.007     0.000     0.000      5496   Gosu::Window#button_down?
  0.04      0.048     0.007     0.000     0.041      2449   Explosion#done?
  0.04      0.137     0.007     0.000     0.130       722   Array#map
  0.04      0.009     0.007     0.000     0.002      1348   Array#each
  0.04      0.009     0.006     0.000     0.003       333   Map#tile_at
  0.04      0.021     0.006     0.000     0.015      1409   Bullet#point_at_distance
  0.03     15.937     0.005     0.000    15.932       333   Gosu::Window#scale
  0.03      0.019     0.005     0.000     0.014      1348   Tank#moving?
  0.03      0.005     0.005     0.000     0.000         2   Gosu::Sample#initialize_
:
```

Profiling results for `PlayState`

We can see that `Gosu::Image#draw` takes up to 20% of all execution time. Then goes `Gosu::Window#caption`, but we need it to measure FPS, so we will leave it alone, and finally we can see `Hash#each`, which is guaranteed to be the one from `Map#draw`, and it triggers all those Gosu::Image#draw calls.

Optimizing Inefficient Code

According to profiling results, we need to optimize this method:

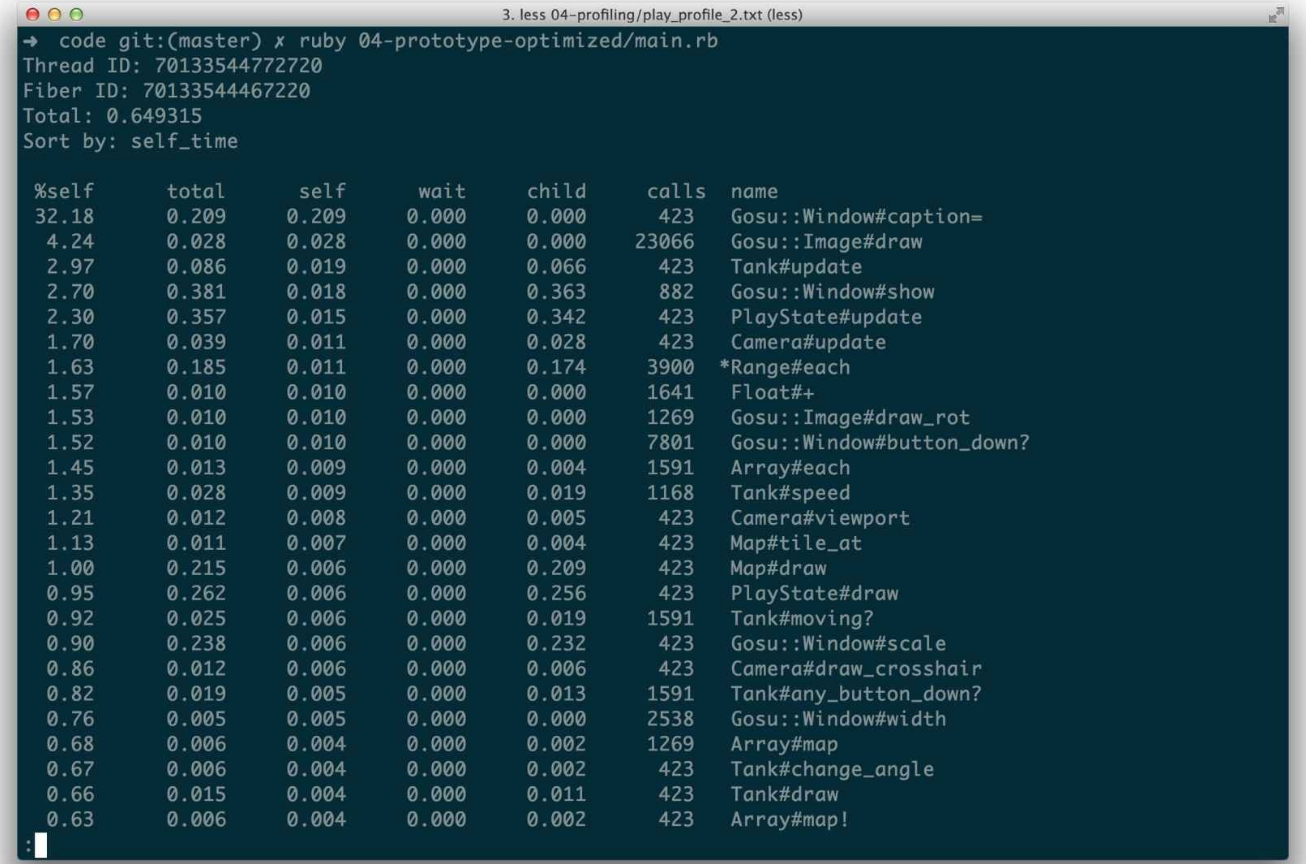
```
class Map
  # ...
  def draw(camera)
    @map.each do |x, row|
      row.each do |y, val|
        tile = @map[x][y]
        map_x = x * TILE_SIZE
        map_y = y * TILE_SIZE
        tile.draw(map_x, map_y, 0)
      end
    end
  end
  # ...
end
```

But we have to optimize it in more clever way than we did before. If instead of looping through all map rows and columns and blindly rendering every tile or checking if tile is visible we could calculate the exact map cells that need to be displayed, we would reduce method complexity and get major performance boost. Let’s do that.

We will use `Camera#viewport` to return map boundaries that are visible by camera, then divide those boundaries by `Map#TILE_SIZE` to get tile numbers instead of pixels, and retrieve them from the map.

```
class Map
  # ...
  def draw(camera)
    viewport = camera.viewport
    viewport.map! { |p| p / TILE_SIZE }
    x0, x1, y0, y1 = viewport.map(&:to_i)
    (x0..x1).each do |x|
      (y0..y1).each do |y|
        row = @map[x]
        if row
          tile = @map[x][y]
          map_x = x * TILE_SIZE
          map_y = y * TILE_SIZE
          tile.draw(map_x, map_y, 0)
        end
      end
    end
  end
```

This optimization yielded astounding results. We are now getting nearly stable 60 FPS even when profiling the code! Compare that to 2 FPS while profiling when we started.



```
3. less 04-profiling/play_profile_2.txt (less)
➜  code git:(master) ✗ ruby 04-prototype-optimized/main.rb
Thread ID: 70133544772720
Fiber ID: 70133544467220
Total: 0.649315
Sort by: self_time

 %self      total      self      wait     child     calls  name
 32.18      0.209     0.209     0.000     0.000      423   Gosu::Window#caption=
  4.24      0.028     0.028     0.000     0.000    23066   Gosu::Image#draw
  2.97      0.086     0.019     0.000     0.066      423   Tank#update
  2.70      0.381     0.018     0.000     0.363      882   Gosu::Window#show
  2.30      0.357     0.015     0.000     0.342      423   PlayState#update
  1.70      0.039     0.011     0.000     0.028      423   Camera#update
  1.63      0.185     0.011     0.000     0.174     3900  *Range#each
  1.57      0.010     0.010     0.000     0.000     1641   Float#+
  1.53      0.010     0.010     0.000     0.000     1269   Gosu::Image#draw_rot
  1.52      0.010     0.010     0.000     0.000     7801   Gosu::Window#button_down?
  1.45      0.013     0.009     0.000     0.004     1591   Array#each
  1.35      0.028     0.009     0.000     0.019     1168   Tank#speed
  1.21      0.012     0.008     0.000     0.005      423   Camera#viewport
  1.13      0.011     0.007     0.000     0.004      423   Map#tile_at
  1.00      0.215     0.006     0.000     0.209      423   Map#draw
  0.95      0.262     0.006     0.000     0.256      423   PlayState#draw
  0.92      0.025     0.006     0.000     0.019     1591   Tank#moving?
  0.90      0.238     0.006     0.000     0.232      423   Gosu::Window#scale
  0.86      0.012     0.006     0.000     0.006      423   Camera#draw_crosshair
  0.82      0.019     0.005     0.000     0.013     1591   Tank#any_button_down?
  0.76      0.005     0.005     0.000     0.000     2538   Gosu::Window#width
  0.68      0.006     0.004     0.000     0.002     1269   Array#map
  0.67      0.006     0.004     0.000     0.002      423   Tank#change_angle
  0.66      0.015     0.004     0.000     0.011      423   Tank#draw
  0.63      0.006     0.004     0.000     0.002      423   Array#map!
:
```

Profiling results for `PlayState` after `Map#draw` optimization

Now we just have to do something about that [`Gosu::Window#caption`](), because it is consuming 1/3 of our CPU cycles! Even though game is already flying so fast that we will have to reduce tank and bullet speeds to make it look more realistic, we cannot let ourselves leave this low hanging fruit remain unpicked.

We will update the caption once per second, it should remove the bottleneck:

```ruby
class PlayState < GameState
  # ...
  def update
    # ...
    update_caption
  end
  # ...
  private

  def update_caption
    now = Gosu.milliseconds
    if now - (@caption_updated_at || 0) > 1000
      $window.caption = 'Tanks Prototype. ' <<
        "[FPS: #{Gosu.fps}. " <<
        "Tank @ #{@tank.x.round}:#{@tank.y.round}]"
      @caption_updated_at = now
    end
  end
end
```

Now it’s getting hard to get FPS to drop below 58, and profiling results show that there are no more bottlenecks:

```
3. less 04-profiling/play_profile_3.txt (less)
➜  code git:(master) ✗ ENABLE_PROFILING=1 ruby 04-prototype-optimized/main.rb
Thread ID: 70210468308080
Fiber ID: 70210468002100
Total: 1.284961
Sort by: self_time

 %self      total      self      wait     child     calls  name
  5.13      0.066     0.066     0.000     0.000     47363   Gosu::Image#draw
  4.20      0.552     0.054     0.000     0.498      3014   Gosu::Window#show
  4.15      0.273     0.053     0.000     0.219      1046   Tank#update
  2.65      0.034     0.034     0.000     0.000         1   <Class::Gosu::Image>#load_tiles
  2.31      0.082     0.030     0.000     0.052      1046   Camera#update
  2.05      0.432     0.026     0.000     0.406      8840  *Range#each
  2.04      0.026     0.026     0.000     0.000     18328   Gosu::Window#button_down?
  1.93      0.034     0.025     0.000     0.009      3843   Array#each
  1.84      0.077     0.024     0.000     0.053      2797   Tank#speed
  1.77      0.471     0.023     0.000     0.449      1046   PlayState#update
  1.61      0.032     0.021     0.000     0.011      1046   Map#tile_at
  1.58      0.020     0.020     0.000     0.000      6601   Gosu::Window#width
  1.55      0.032     0.020     0.000     0.012      1046   Camera#viewport
  1.47      0.024     0.019     0.000     0.005      1046   Tank#change_angle
  1.46      0.153     0.019     0.000     0.134      3463   Array#map
  1.37      0.018     0.018     0.000     0.000      5418   Fixnum#-
  1.35      0.066     0.017     0.000     0.049      3843   Tank#moving?
  1.33      0.017     0.017     0.000     0.000      1046   <Module::Math>#atan2
  1.27      0.511     0.016     0.000     0.494      1046   Map#draw
  1.25      0.715     0.016     0.000     0.699      1046   PlayState#draw
  1.23      0.635     0.016     0.000     0.620      1046   Gosu::Window#scale
  1.21      0.031     0.016     0.000     0.016      1046   Camera#draw_crosshair
  1.16      0.049     0.015     0.000     0.034      3843   Tank#any_button_down?
  0.90      0.023     0.012     0.000     0.012      1046   Tank#draw
  0.88      0.048     0.011     0.000     0.037      1047   Bullet#update
:
```

Profiling results for `PlayState` after introducing `Gosu::Window#caption` cache

We can now sleep well at night.

Profiling On Demand

When you develop a game, you may want to turn on profiling now and then. To avoid commenting out or adding and removing profiling every time you want to do so, use this trick:

```ruby
# ...
require 'ruby-prof' if ENV['ENABLE_PROFILING']
class PlayState < GameState
  # ...
  def enter
    RubyProf.start if ENV['ENABLE_PROFILING']
  end

  def leave
    if ENV['ENABLE_PROFILING']
      result = RubyProf.stop
      printer = RubyProf::FlatPrinter.new(result)
      printer.print(STDOUT)
    end
  end

  def button_down(id)
    # ...
    if id == Gosu::KbQ
      leave
      $window.close
    end
  end
  # ...
end
```

Now, to enable profiling, simply start your game with `ENABLE_PROFILING=1` environmental variable, like this:

```
$ ENABLE_PROFILING=1 ruby-prof 03-prototype/main.rb
```

Adjusting Game Speed For Variable Performance

You should have noticed that our optimized Tanks prototype runs way too fast. Tanks and bullets should travel same distance no matter how fast or slow the code is.

One would expect `Gosu::Window#update_interval` to be designed exactly for that purpose, but it returns `16.6666` in both original and optimized version of the prototype, so you can guess it is the desired interval, not the actual one.

To find out actual update interval, we will use `Gosu.milliseconds` and calculate it ourselves. To do that, we will introduce `Game#track_update_interval` that will be called in `GameWindow#update`, and `Game#update_interval` which will retrieve actual update interval, so we can use it to adjust our run speed.

We will also add `Game#adjust_speed` method that will take arbitrary speed value and shift it so is as fast as it was when the game was running at 30 FPS. The formula is simple, if 60 FPS expects to call `Gosu::Window#update` every `16.66` ms, our speed adjustment will divide actual update rate from `33.33`, which roughly equals to `16.66 * 2`. So, if bullet would fly 100 pixels per update in 30 FPS, adjusted speed will change it to 50 pixels at 60 FPS.

Here is the implementation:

```ruby
# 04-prototype-optimized/main.rb
module Game
  # ...
  def self.track_update_interval
    now = Gosu.milliseconds
    @update_interval = (now - (@last_update ||= 0)).to_f
    @last_update = now
  end

  def self.update_interval
    @update_interval ||= $window.update_interval
  end

  def self.adjust_speed(speed)
    speed * update_interval / 33.33
  end
end

# 04-prototype-optimized/game_window.rb
class GameWindow < Gosu::Window
  # ...
  def update
    Game.track_update_interval
    @state.update
  end
  # ...
end
```

Now, to fix that speed problem, we will need to apply `Game.adjust_speed` to tank, bullet and camera movements.

Here are all the changes needed to make our game run at roughly same speed in different conditions:

```ruby
# 04-prototype-optimized/entities/tank.rb
class Tank
  # ...
  def update(camera)
    # ...
    shift = Game.adjust_speed(speed)
    new_x -= shift if $window.button_down?(Gosu::KbA)
    new_x += shift if $window.button_down?(Gosu::KbD)
    new_y -= shift if $window.button_down?(Gosu::KbW)
    new_y += shift if $window.button_down?(Gosu::KbS)
    # ...
  end
  # ...
end

# 04-prototype-optimized/entities/bullet.rb
class Bullet
  # ...
  def update
    # ...
    fly_speed = Game.adjust_speed(@speed)
    fly_distance = (Gosu.milliseconds - @fired_at) * 0.001 * fly_speed
    @x, @y = point_at_distance(fly_distance)
    # ...
  end
  # ...
end

# 04-prototype-optimized/entities/camera.rb
class Camera
  # ...
  def update
    shift = Game.adjust_speed(@target.speed)
    @x += shift if @x < @target.x - $window.width / 4
    @x -= shift if @x > @target.x + $window.width / 4
    @y += shift if @y < @target.y - $window.height / 4
    @y -= shift if @y > @target.y + $window.height / 4

    zoom_delta = @zoom > 0 ? 0.01 : 1.0
    zoom_delta = Game.adjust_speed(zoom_delta)
    # ...
  end
  # ...
end
```

There is one more trick to make the game playable even at very low FPS. You can simulate such conditions by adding `sleep 0.3` to `GameWindow#draw` method. At that framerate game cursor is very unresponsive, so you may want to start showing native mouse cursor when things get ugly, i.e. when update interval exceeds 200 milliseconds:

```ruby
# 04-prototype-optimized/game_window.rb
class GameWindow < Gosu::Window
  # ...
  def needs_cursor?
    Game.update_interval > 200
  end
  # ...
end
```

Frame Skipping

You will see strange things happening at very low framerates. For example, bullet explosions are showing up frame by frame, so explosion speed seems way too slow and unrealistic. To avoid that, we will modify our `Explosion` class to employ frame skipping if update rate is too slow:

```ruby
# 04-prototype-optimized/explosion.rb
class Explosion
  FRAME_DELAY = 16.66 # ms
  # ...
```

```ruby
  def update
    advance_frame
  end

  def done?
    @done ||= @current_frame >= animation.size
  end
  # ...
  private
  # ...
  def advance_frame
    now = Gosu.milliseconds
    delta = now - (@last_frame ||= now)
    if delta > FRAME_DELAY
      @last_frame = now
    end
    @current_frame += (delta / FRAME_DELAY).floor
  end
end
```

Now our prototype is playable even at lower frame rates.

Refactoring The Prototype

At this point you may be thinking where to go next. We want to implement enemies, collision detection and AI, but design of current prototype is already limiting. Code is becoming tightly coupled, there is no clean separation between different domains.

If we were to continue building on top of our prototype, things would get ugly quickly. Thus we will untangle the spaghetti and rewrite some parts from scratch to achieve elegance.

Game Programming Patterns

I would like to tip my hat to Robert Nystrom, who wrote this amazing book called Game Programming Patterns. The book is available online for free, it is a relatively quick read - I've devoured it with pleasure in roughly 4 hours. If you are guessing that this chapter is inspired by that book, you are absolutely right.

Component pattern is especially noteworthy. We will be using it to do major housekeeping, and it is great time to do so, because we haven't implemented much of the game yet.

What Is Wrong With Current Design

Until this point we have been building the code in monolithic fashion. `Tank` class holds the code that:

1. Loads all ground unit sprites. If some other class handled it, we could reuse the code to load other units.
2. Handles sound effects.
3. Uses `Gosu::Song` for moving sounds. That limits only one tank movement sound per whole game. Basically, we abused Gosu here.
4. Handles keyboard and mouse. If we were to create AI that controls the tank, we would not be able to reuse `Tank` class because of this.
5. Draws graphics on screen.
6. Calculates physical properties, like speed, acceleration.
7. Detects movement collisions.

`Bullet` is not perfect either:

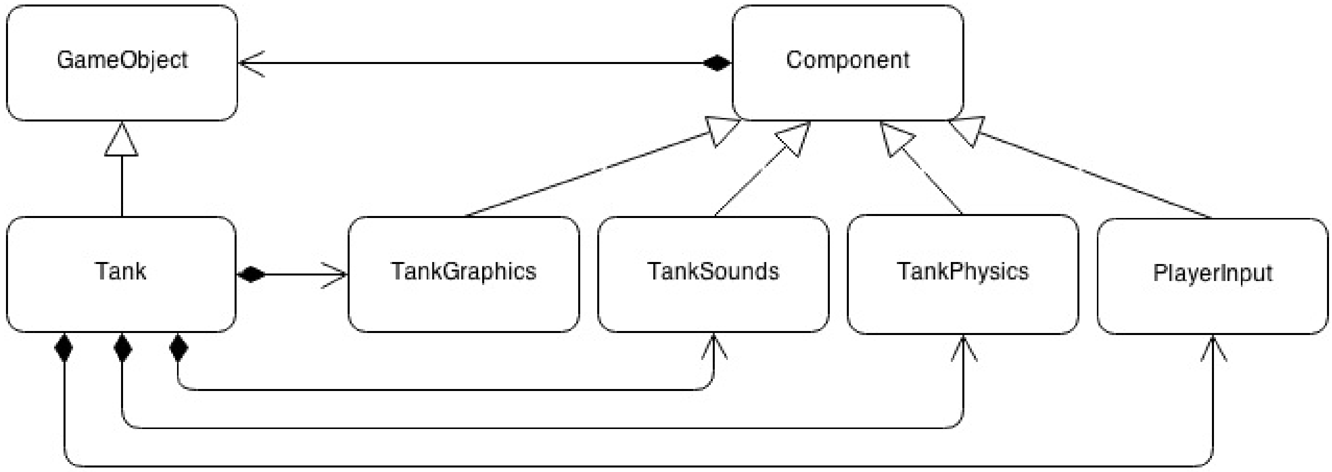
1. It renders it's graphics.
2. It handles it's movement trajectories and other physics.
3. It treats `Explosion` as part of it's own lifecycle.
4. Draws graphics on screen.
5. Handles sound effects.

Even the relatively small `Explosion` class is too monolithic:

1. It loads it’s graphics.
2. It handles rendering, animation and frame skipping
3. It loads and plays it’s sound effects.

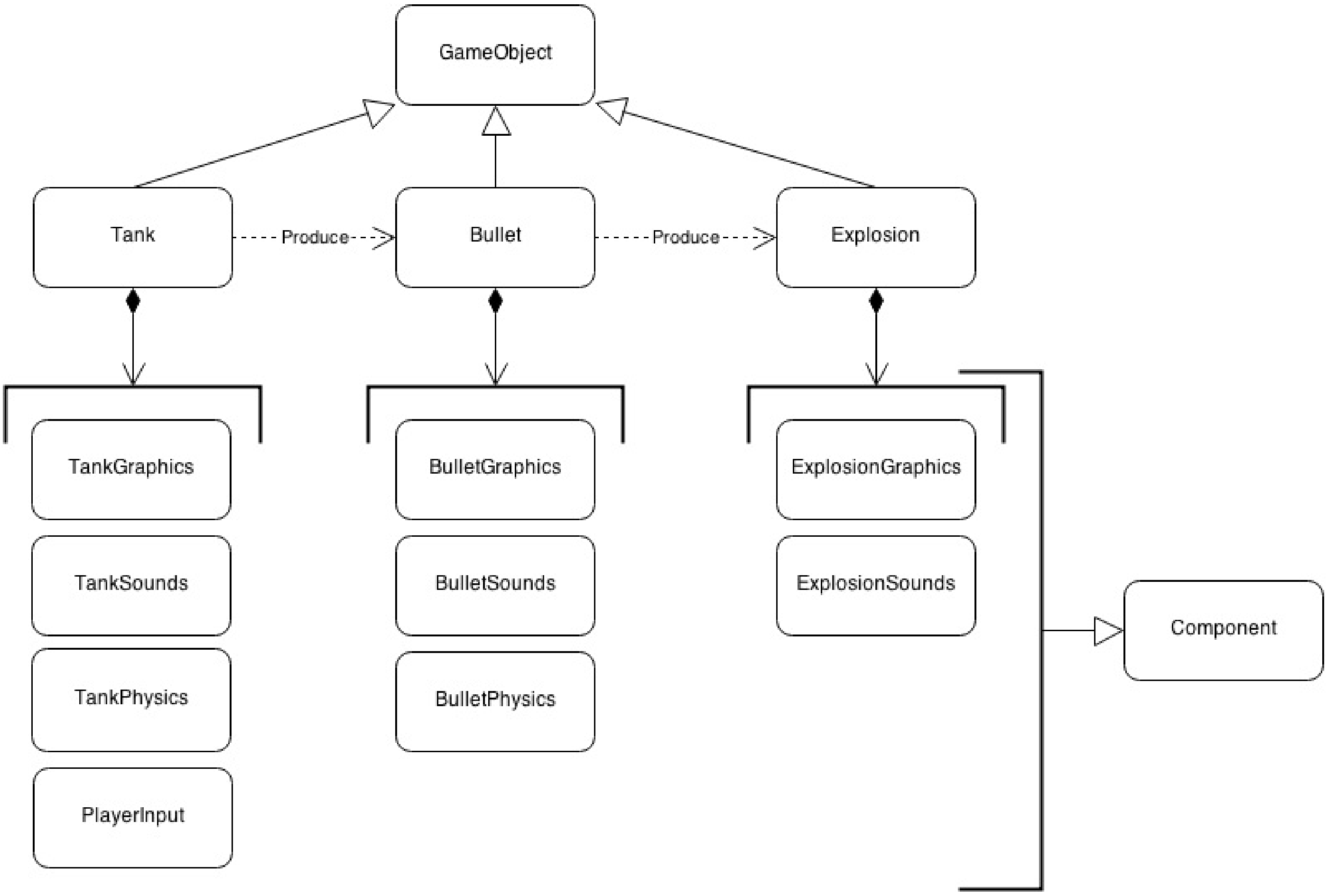
Decoupling Using Component Pattern

Best design separates concerns in code so that everything has it’s own place, and every class handles only one thing. Let’s try splitting up `Tank` class into components that handle specific domains:



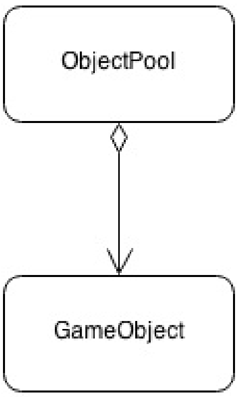
Decoupled Tank

We will introduce `GameObject` class will contain shared functionality for all game objects (`Tank`, `Bullet`, `Explosion`), each of them would have it’s own set of components. Every component will have it’s parent object, so it will be able to interact with it, change it’s attributes, or possibly invoke other components if it comes to that.



Game objects and their components

All these objects will be held within [`ObjectPool`](), which would not care to know if object is a tank or a bullet. Purpose of `ObjectPool` is a little different in Ruby, since GC will take care of memory fragmentation for us, but we still need a single place that knows about every object in the game.



Object Pool

`PlayState` would then iterate through `@object_pool.objects` and invoke `update` and `draw` methods.

Now, let's begin by implementing base class for `GameObject`:

05-refactor/entities/game_object.rb

```ruby
class GameObject
  def initialize(object_pool)
    @components = []
    @object_pool = object_pool
    @object_pool.objects << self
```

```
  end

  def components
    @components
  end

  def update
    @components.map(&:update)
  end

  def draw(viewport)
    @components.each { |c| c.draw(viewport) }
  end

  def removable?
    @removable
  end

  def mark_for_removal
    @removable = true
  end

  protected

  def object_pool
    @object_pool
  end
end
```

When `GameObject` is initialized, it registers itself with `ObjectPool` and prepares empty `@components` array. Concrete `GameObject` classes should initialize `Components` so that array would not be empty.

`update` and `draw` methods would cycle through `@components` and delegate those calls to each of them in a sequence. It is important to `update` all components first, and only then `draw` them. Keep in mind that `@components` array order has significance. First elements will always be updated and drawn before last ones.

We will also provide `removable?` method that would return `true` for objects that `mark_for_removal` was invoked on. This way we will be able to weed out old bullets and explosions and feed them to GC.

Next up, base `Component` class:

05-refactor/entities/components/component.rb

```
class Component
  def initialize(game_object = nil)
    self.object = game_object
  end

  def update
    # override
  end

  def draw(viewport)
    # override
  end

  protected

  def object=(obj)
    if obj
      @object = obj
      obj.components << self
    end
  end

  def x
```

```ruby
    @object.x
  end

  def y
    @object.y
  end

  def object
    @object
  end
end
```

It registers itself with `GameObject#components`, provides some protected methods to access parent object and it's most often called properties - `x` and `y`.

Refactoring Explosion

`Explosion` was probably the smallest class, so we will extract it's components first.

05-refactor/entities/explosion.rb

```ruby
class Explosion < GameObject
  attr_accessor :x, :y

  def initialize(object_pool, x, y)
    super(object_pool)
    @x, @y = x, y
    ExplosionGraphics.new(self)
    ExplosionSounds.play
  end
end
```

It is much cleaner than before. `ExplosionGraphics` will be a `Component` that handles animation, and `ExplosionSounds` will play a sound.

05-refactor/entities/components/explosion_graphics.rb

```ruby
class ExplosionGraphics < Component
  FRAME_DELAY = 16.66 # ms

  def initialize(game_object)
    super
    @current_frame = 0
  end

  def draw(viewport)
    image = current_frame
    image.draw(
      x - image.width / 2 + 3,
      y - image.height / 2 - 35,
      20)
  end

  def update
    now = Gosu.milliseconds
    delta = now - (@last_frame ||= now)
    if delta > FRAME_DELAY
      @last_frame = now
    end
    @current_frame += (delta / FRAME_DELAY).floor
    object.mark_for_removal if done?
  end

  private

  def current_frame
    animation[@current_frame % animation.size]
  end

  def done?
```

```ruby
    @done ||= @current_frame >= animation.size
  end

  def animation
    @@animation ||=
    Gosu::Image.load_tiles(
      $window, Utils.media_path('explosion.png'),
      128, 128, false)
  end
end
```

Everything that is related to animating the explosion is now clearly separated. `mark_for_removal` is called on the explosion after it's animation is done.

05-refactor/entities/components/explosion_sounds.rb

```ruby
class ExplosionSounds
  class << self
    def play
      sound.play
    end

    private

    def sound
      @@sound ||= Gosu::Sample.new(
        $window, Utils.media_path('explosion.mp3'))
    end
  end
end
```

Since explosion sounds are triggered only once, when it starts to explode, `ExplosionSounds` is a static class with `play` method.

Refactoring Bullet

Now, let's go up a little and reimplement our `Bullet`:

05-refactor/entities/bullet.rb

```ruby
class Bullet < GameObject
  attr_accessor :x, :y, :target_x, :target_y, :speed, :fired_at

  def initialize(object_pool, source_x, source_y, target_x, target_y)
    super(object_pool)
    @x, @y = source_x, source_y
    @target_x, @target_y = target_x, target_y
    BulletPhysics.new(self)
    BulletGraphics.new(self)
    BulletSounds.play
  end

  def explode
    Explosion.new(object_pool, @x, @y)
    mark_for_removal
  end

  def fire(speed)
    @speed = speed
    @fired_at = Gosu.milliseconds
  end
end
```

All physics, graphics and sounds are extracted into individual components, and instead of managing `Explosion`, it just registers a new `Explosion` with `ObjectPool` and marks itself for removal in `explode` method.

05-refactor/entities/components/bullet_physics.rb

```ruby
class BulletPhysics < Component
  START_DIST = 20
  MAX_DIST = 300

  def initialize(game_object)
    super
    object.x, object.y = point_at_distance(START_DIST)
    if trajectory_length > MAX_DIST
      object.target_x, object.target_y = point_at_distance(MAX_DIST)
    end
  end

  def update
    fly_speed = Utils.adjust_speed(object.speed)
    fly_distance = (Gosu.milliseconds - object.fired_at) * 0.001 * fly_speed
    object.x, object.y = point_at_distance(fly_distance)
    object.explode if arrived?
  end

  def trajectory_length
    d_x = object.target_x - x
    d_y = object.target_y - y
    Math.sqrt(d_x * d_x + d_y * d_y)
  end

  def point_at_distance(distance)
    if distance > trajectory_length
      return [object.target_x, object.target_y]
    end
    distance_factor = distance.to_f / trajectory_length
    p_x = x + (object.target_x - x) * distance_factor
    p_y = y + (object.target_y - y) * distance_factor
    [p_x, p_y]
  end

  private

  def arrived?
    x == object.target_x && y == object.target_y
  end
end
```

`BulletPhysics` is where the most of `Bullet` ended up at. It does all the calculations and triggers `Bullet#explode` when ready. When we will be implementing collision detection, the implementation will go somewhere here.

05-refactor/entities/components/bullet_graphics.rb

```ruby
class BulletGraphics < Component
  COLOR = Gosu::Color::BLACK

  def draw(viewport)
    $window.draw_quad(x - 2, y - 2, COLOR,
                      x + 2, y - 2, COLOR,
                      x - 2, y + 2, COLOR,
                      x + 2, y + 2, COLOR,
                      1)
  end

end
```

After pulling away `Bullet` graphics code, it looks very small and elegant. We will probably never have to edit anything here again.

05-refactor/entities/components/bullet_sounds.rb

```ruby
class BulletSounds
  class << self
    def play
      sound.play
```

```ruby
    end

    private

    def sound
      @@sound ||= Gosu::Sample.new(
        $window, Utils.media_path('fire.mp3'))
    end
  end
end
```

Just like `ExplosionSounds`, `BulletSounds` are stateless and static. We could make it just like a regular component, but consider it our little optimization.

Refactoring Tank

Time to take a look at freshly decoupled `Tank`:

05-refactor/entities/tank.rb

```ruby
class Tank < GameObject
  SHOOT_DELAY = 500
  attr_accessor :x, :y, :throttle_down, :direction, :gun_angle, :sounds, :physics

  def initialize(object_pool, input)
    super(object_pool)
    @input = input
    @input.control(self)
    @physics = TankPhysics.new(self, object_pool)
    @graphics = TankGraphics.new(self)
    @sounds = TankSounds.new(self)
    @direction = @gun_angle = 0.0
  end

  def shoot(target_x, target_y)
    if Gosu.milliseconds - (@last_shot || 0) > SHOOT_DELAY
      @last_shot = Gosu.milliseconds
      Bullet.new(object_pool, @x, @y, target_x, target_y).fire(100)
    end
  end
end
```

`Tank` class was reduced over 5 times. We could go further and extract `Gun` component, but for now it’s simple enough already. Now, the components.

05-refactor/entities/components/tank_physics.rb

```ruby
class TankPhysics < Component
  attr_accessor :speed

  def initialize(game_object, object_pool)
    super(game_object)
    @object_pool = object_pool
    @map = object_pool.map
    game_object.x, game_object.y = @map.find_spawn_point
    @speed = 0.0
  end

  def can_move_to?(x, y)
    @map.can_move_to?(x, y)
  end

  def moving?
    @speed > 0
  end

  def update
    if object.throttle_down
      accelerate
    else
```

```
      decelerate
    end
    if @speed > 0
      new_x, new_y = x, y
      shift = Utils.adjust_speed(@speed)
      case @object.direction.to_i
      when 0
        new_y -= shift
      when 45
        new_x += shift
        new_y -= shift
      when 90
        new_x += shift
      when 135
        new_x += shift
        new_y += shift
      when 180
        new_y += shift
      when 225
        new_y += shift
        new_x -= shift
      when 270
        new_x -= shift
      when 315
        new_x -= shift
        new_y -= shift
      end
      if can_move_to?(new_x, new_y)
        object.x, object.y = new_x, new_y
      else
        object.sounds.collide if @speed > 1
        @speed = 0.0
      end
    end
  end

  private

  def accelerate
    @speed += 0.08 if @speed < 5
  end

  def decelerate
    @speed -= 0.5 if @speed > 0
    @speed = 0.0 if @speed < 0.01 # damp
  end
end
```

While we had to rip player input away from it's movement, we got ourselves a benefit - tank now both accelerates and decelerates. When directional buttons are no longer pressed, tank keeps moving in last direction, but quickly decelerates and stops. Another addition that would have been more difficult to implement on previous `Tank` is collision sound. When `Tank` abruptly stops by hitting something (for now it's only water), collision sound is played. We will have to fix that, because metal bang is not appropriate when you stop on the edge of a river, but we now did it for the sake of science.

05-refactor/entities/components/tank_graphics.rb

```
class TankGraphics < Component
  def initialize(game_object)
    super(game_object)
    @body = units.frame('tank1_body.png')
    @shadow = units.frame('tank1_body_shadow.png')
    @gun = units.frame('tank1_dualgun.png')
  end

  def draw(viewport)
    @shadow.draw_rot(x - 1, y - 1, 0, object.direction)
    @body.draw_rot(x, y, 1, object.direction)
    @gun.draw_rot(x, y, 2, object.gun_angle)
  end
```

```ruby

  private

  def units
    @@units = Gosu::TexturePacker.load_json(
      $window, Utils.media_path('ground_units.json'), :precise)
  end
end
```

Again, graphics are neatly packed and separated from everything else. Eventually we should optimize `draw` to take `viewport` into consideration, but it's good enough for now, especially when we have only one tank in the game.

05-refactor/entities/components/tank_sounds.rb

```ruby
class TankSounds < Component
  def update
    if object.physics.moving?
      if @driving && @driving.paused?
        @driving.resume
      elsif @driving.nil?
        @driving = driving_sound.play(1, 1, true)
      end
    else
      if @driving && @driving.playing?
        @driving.pause
      end
    end
  end

  def collide
    crash_sound.play(1, 0.25, false)
  end

  private

  def driving_sound
    @@driving_sound ||= Gosu::Sample.new(
      $window, Utils.media_path('tank_driving.mp3'))
  end

  def crash_sound
    @@crash_sound ||= Gosu::Sample.new(
      $window, Utils.media_path('crash.ogg'))
  end
end
```

Unlike `Explosion` and `Bullet`, `Tank` sounds are stateful. We have to keep track of `tank_driving.mp3`, which is no longer `Gosu::Song`, but `Gosu::Sample`, like it should have been.

When `Gosu::Sample#play` is invoked, `Gosu::SampleInstance` is returned, and we have full control over it. Now we are ready to play sounds for more than one tank at once.

05-refactor/entities/components/player_input.rb

```ruby
class PlayerInput < Component
  def initialize(camera)
    super(nil)
    @camera = camera
  end

  def control(obj)
    self.object = obj
  end

  def update
    d_x, d_y = @camera.target_delta_on_screen
    atan = Math.atan2(($window.width / 2) - d_x - $window.mouse_x,
```

```
                     ($window.height / 2) - d_y - $window.mouse_y)
    object.gun_angle = -atan * 180 / Math::PI
    motion_buttons = [Gosu::KbW, Gosu::KbS, Gosu::KbA, Gosu::KbD]

    if any_button_down?(*motion_buttons)
      object.throttle_down = true
      object.direction = change_angle(object.direction, *motion_buttons)
    else
      object.throttle_down = false
    end

    if Utils.button_down?(Gosu::MsLeft)
      object.shoot(*@camera.mouse_coords)
    end
  end

  private

  def any_button_down?(*buttons)
    buttons.each do |b|
      return true if Utils.button_down?(b)
    end
    false
  end

  def change_angle(previous_angle, up, down, right, left)
    if Utils.button_down?(up)
      angle = 0.0
      angle += 45.0 if Utils.button_down?(left)
      angle -= 45.0 if Utils.button_down?(right)
    elsif Utils.button_down?(down)
      angle = 180.0
      angle -= 45.0 if Utils.button_down?(left)
      angle += 45.0 if Utils.button_down?(right)
    elsif Utils.button_down?(left)
      angle = 90.0
      angle += 45.0 if Utils.button_down?(up)
      angle -= 45.0 if Utils.button_down?(down)
    elsif Utils.button_down?(right)
      angle = 270.0
      angle -= 45.0 if Utils.button_down?(up)
      angle += 45.0 if Utils.button_down?(down)
    end
    angle = (angle + 360) % 360 if angle && angle < 0
    (angle || previous_angle)
  end
end
```

We finally come to a place where keyboard and mouse input is handled and converted to `Tank` commands. We could have used `Command` pattern to decouple everything even further.

Refactoring PlayState

05-refactor/game_states/play_state.rb

```
require 'ruby-prof' if ENV['ENABLE_PROFILING']
class PlayState < GameState
  attr_accessor :update_interval

  def initialize
    @map = Map.new
    @camera = Camera.new
    @object_pool = ObjectPool.new(@map)
    @tank = Tank.new(@object_pool, PlayerInput.new(@camera))
    @camera.target = @tank
  end

  def enter
    RubyProf.start if ENV['ENABLE_PROFILING']
  end

  def leave
    if ENV['ENABLE_PROFILING']
```

```ruby
      result = RubyProf.stop
      printer = RubyProf::FlatPrinter.new(result)
      printer.print(STDOUT)
    end
  end

  def update
    @object_pool.objects.map(&:update)
    @object_pool.objects.reject!(&:removable?)
    @camera.update
    update_caption
  end

  def draw
    cam_x = @camera.x
    cam_y = @camera.y
    off_x =  $window.width / 2 - cam_x
    off_y =  $window.height / 2 - cam_y
    viewport = @camera.viewport
    $window.translate(off_x, off_y) do
      zoom = @camera.zoom
      $window.scale(zoom, zoom, cam_x, cam_y) do
        @map.draw(viewport)
        @object_pool.objects.map { |o| o.draw(viewport) }
      end
    end
    @camera.draw_crosshair
  end

  def button_down(id)
    if id == Gosu::KbQ
      leave
      $window.close
    end
    if id == Gosu::KbEscape
      GameState.switch(MenuState.instance)
    end
  end

  private

  def update_caption
    now = Gosu.milliseconds
    if now - (@caption_updated_at || 0) > 1000
      $window.caption = 'Tanks Prototype. ' <<
        "[FPS: #{Gosu.fps}. " <<
        "Tank @ #{@tank.x.round}:#{@tank.y.round}]"
      @caption_updated_at = now
    end
  end
end
```

Implementation of `PlayState` is now also a little simpler. It doesn't update `@tank` or `@bullets` individually anymore. Instead, it uses `ObjectPool` and does all object operations in bulk.

Other Improvements

05-refactor/main.rb

```ruby
#!/usr/bin/env ruby

require 'gosu'

root_dir = File.dirname(__FILE__)
require_pattern = File.join(root_dir, '**/*.rb')
@failed = []

# Dynamically require everything
Dir.glob(require_pattern).each do |f|
  next if f.end_with?('/main.rb')
  begin
```

```
    require_relative f.gsub("#{root_dir}/", '')
  rescue
    # May fail if parent class not required yet
    @failed << f
  end
end

# Retry unresolved requires
@failed.each do |f|
  require_relative f.gsub("#{root_dir}/", '')
end

$window = GameWindow.new
GameState.switch(MenuState.instance)
$window.show
```

Finally, we made some improvements to `main.rb` - it now recursively requires all `*.rb` files within same directory, so we don't have to worry about it in other classes.

05-refactor/utils.rb

```
module Utils
  def self.media_path(file)
    File.join(File.dirname(File.dirname(
      __FILE__)), 'media', file)
  end

  def self.track_update_interval
    now = Gosu.milliseconds
    @update_interval = (now - (@last_update ||= 0)).to_f
    @last_update = now
  end

  def self.update_interval
    @update_interval ||= $window.update_interval
  end

  def self.adjust_speed(speed)
    speed * update_interval / 33.33
  end

  def self.button_down?(button)
    @buttons ||= {}
    now = Gosu.milliseconds
    now = now - (now % 150)
    if $window.button_down?(button)
      @buttons[button] = now
      true
    elsif @buttons[button]
      if now == @buttons[button]
        true
      else
        @buttons.delete(button)
        false
      end
    end
  end
end
```

Another notable change is renaming `Game` module into `Utils`. The name finally makes more sense, I have no idea why I put utility methods into `Game` module in the first place. Also, `Utils` received `button_down?` method, that solves the issue of changing tank direction when button is immediately released. It made very difficult to stop at diagonal angle, because when you depressed two buttons, 16 ms was enough for Gosu to think "he released W, and S is still pressed, so let's change direction to S". `Utils#button_down?` gives a soft 150 ms window to synchronize button release. Now controls feel more natural.

Simulating Physics

To make the game more realistic, we will spice things up with some physics. This is the feature set we are going to implement:

1. Collision detection. Tank will bump into other objects - stationary tanks. Bullets will not go through them either.
2. Terrain effects. Tank will go fast on grass, slower on sand.

Adding Enemy Objects

It's boring to play alone, so we will make a quick change and spawn some stationary tanks that will be deployed randomly around the map. They will be stationary in the beginning, but we will still need a dummy AI class to replace `PlayerInput`:

06-physics/entities/components/ai_input.rb

```ruby
class AiInput < Component
  def control(obj)
    self.object = obj
  end
end
```

A quick and dirty way to spawn some tanks would be when initializing `PlayState`:

```ruby
class PlayState < GameState
  # ...
  def initialize
    @map = Map.new
    @camera = Camera.new
    @object_pool = ObjectPool.new(@map)
    @tank = Tank.new(@object_pool, PlayerInput.new(@camera))
    @camera.target = @tank
    # ...
    50.times do
      Tank.new(@object_pool, AiInput.new)
    end
  end
  # ...
end
```

And unless we want all stationary tanks face same direction, we will randomize it:

```ruby
class Tank < GameObject
  # ...
  def initialize(object_pool, input)
    # ...
    @direction = rand(0..7) * 45
    @gun_angle = rand(0..360)
  end
  # ...
end
```

Fire up the game, and wander around frozen tanks. You can pass through them as if they were ghosts, but we will fix that in a moment.

Brain dead enemies

Adding Bounding Boxes And Detecting Collisions

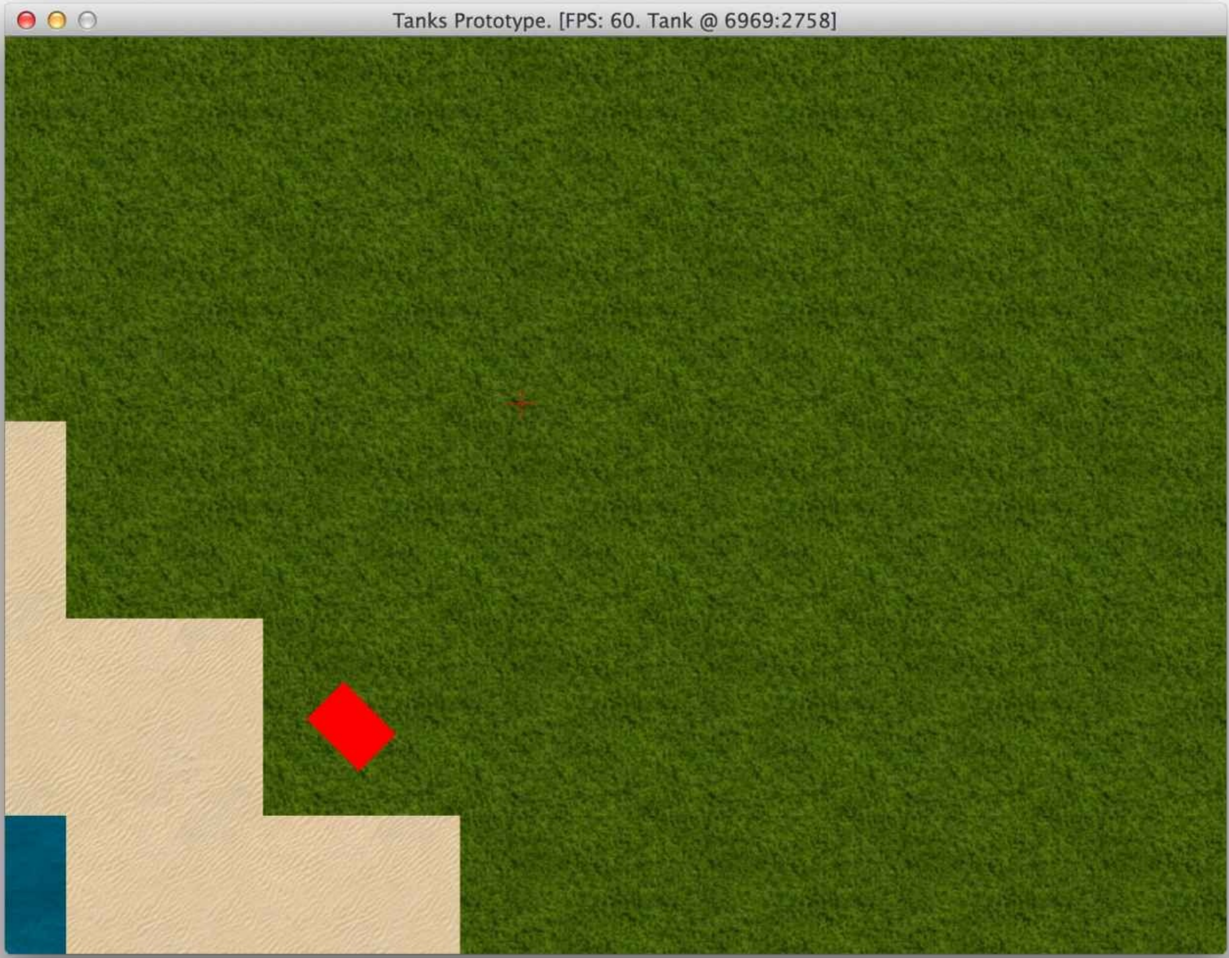
We want our collision detection to be pixel perfect, that means we need to have a bounding box and check colisions against it. Get ready for some math!

First, we need to find a correct way to construct a bounding box. Tank has it's body image, so let's see how it's boundaries look like. We will add some code to `TankGraphics` component to see it:

```
class TankGraphics < Component
  def draw(viewport)
    # ...
    draw_bounding_box
  end

  def draw_bounding_box
    $window.rotate(object.direction, x, y) do
      w = @body.width
      h = @body.height
      $window.draw_quad(
        x - w / 2, y - h / 2, Gosu::Color::RED,
        x + w / 2, y - h / 2, Gosu::Color::RED,
        x + w / 2, y + h / 2, Gosu::Color::RED,
        x - w / 2, y + h / 2, Gosu::Color::RED,
        100)
    end
  end
```

```
  # ...
end
```

Result is pretty good, we have tank shaped box, so we will be using body image dimensions to determine our bounding box corners:



Tank's bounding box visualized

There is one problem here though. `Gosu::Window#rotate` does the rotation math for us, and we need to perform these calculations on our own. We have four points that we want to rotate around a center point. It's not very difficult to find how to do this. Here is a Ruby method for you:

```
module Utils
  # ...
  def self.rotate(angle, around_x, around_y, *points)
    result = []
    points.each_slice(2) do |x, y|
      r_x = Math.cos(angle) * (x - around_x) -
        Math.sin(angle) * (y - around_y) + around_x
      r_y = Math.sin(angle) * (x - around_x) +
        Math.cos(angle) * (y - around_y) + around_y
      result << r_x
      result << r_y
    end
    result
  end
  # ...
end
```

We can now calculate edges of our bounding box, but we need one more function which tells if point is inside a polygon. This problem has been solved million times before, so just poke the internet for it and drink from the information firehose until you understand how to do this.

If you wasn't familiar with the term yet, by now you should discover what vertex is. In geometry, a vertex (plural vertices) is a special kind of point that describes the corners or intersections of geometric shapes.

Here's what I ended up writing:

```
module Utils
  # ...
  # http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html
  def self.point_in_poly(testx, testy, *poly)
    nvert = poly.size / 2 # Number of vertices in poly
    vertx = []
    verty = []
    poly.each_slice(2) do |x, y|
      vertx << x
      verty << y
    end
    inside = false
    j = nvert - 1
    (0..nvert - 1).each do |i|
      if (((verty[i] > testy) != (verty[j] > testy)) &&
         (testx < (vertx[j] - vertx[i]) * (testy - verty[i]) /
         (verty[j] - verty[i]) + vertx[i]))
        inside = !inside
      end
      j = i
    end
    inside
  end
  # ...
```

It is Jordan curve theorem reimplemented in Ruby. Looks ugly, but it actually works, and is pretty fast too.

Also, this works on more sophisticated polygons, and our tank is shaped more like an H rather than a rectangle, so we could define a pixel perfect polygon. Some pen and paper will help.

```
class TankPhysics < Component
  #...

  # Tank box looks like H. Vertices:
  # 1   2   5   6
  #     3   4
  #
  #    10   9
  # 12 11   8   7
  def box
    w = box_width / 2 - 1
    h = box_height / 2 - 1
    tw = 8 # track width
    fd = 8 # front depth
    rd = 6 # rear depth
    Utils.rotate(object.direction, x, y,
                 x + w,      y + h,      #1
                 x + w - tw, y + h,      #2
                 x + w - tw, y + h - fd, #3

                 x - w + tw, y + h - fd, #4
                 x - w + tw, y + h,      #5
                 x - w,      y + h,      #6

                 x - w,      y - h,      #7
                 x - w + tw, y - h,      #8
                 x - w + tw, y - h + rd, #9

                 x + w - tw, y - h + rd, #10
                 x + w - tw, y - h,      #11
                 x + w,      y - h,      #12
                )
  end
```

```ruby
  # ...
end
```

To visually see it, we will improve our `draw_bounding_box` method:

```ruby
class TankGraphics < Component
  # ...
  DEBUG_COLORS = [
    Gosu::Color::RED,
    Gosu::Color::BLUE,
    Gosu::Color::YELLOW,
    Gosu::Color::WHITE
  ]
  # ...
  def draw_bounding_box
    i = 0
    object.box.each_slice(2) do |x, y|
      color = DEBUG_COLORS[i]
      $window.draw_triangle(
        x - 3, y - 3, color,
        x,     y,     color,
        x + 3, y - 3, color,
        100)
      i = (i + 1) % 4
    end
  end
  # ...
```

Now we can visually test bounding box edges and see that they actually are where they belong.

High precision bounding boxes

Time to pimp our `TankPhysics` to detect those collisions. While our algorithm is pretty fast, it doesn't make sense to check collisions for objects that are pretty far apart. This is why we need our `ObjectPool` to know how to query objects in close proximity.

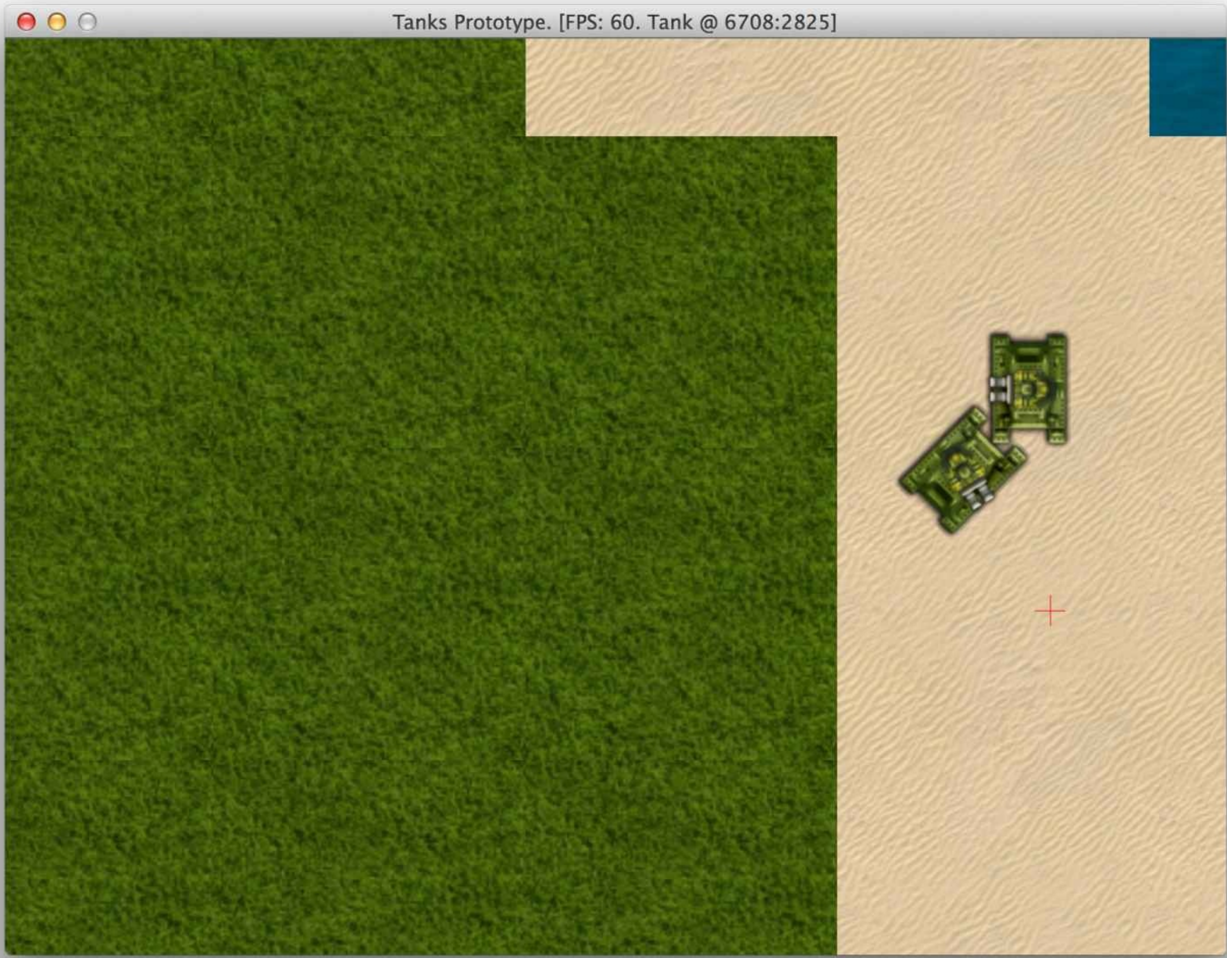
```
class ObjectPool
  # ...
  def nearby(object, max_distance)
    @objects.select do |obj|
      distance = Utils.distance_between(
        obj.x, obj.y, object.x, object.y)
      obj != object && distance < max_distance
    end
  end
end
```

Back to `TankPhysics`:

```
class TankPhysics < Component
  # ...
  def can_move_to?(x, y)
    old_x, old_y = object.x, object.y
    object.x = x
    object.y = y
    return false unless @map.can_move_to?(x, y)
    @object_pool.nearby(object, 100).each do |obj|
      if collides_with_poly?(obj.box)
        # Allow to get unstuck
        old_distance = Utils.distance_between(
          obj.x, obj.y, old_x, old_y)
        new_distance = Utils.distance_between(
```

```
          obj.x, obj.y, x, y)
        return false if new_distance < old_distance
      end
    end
    true
  ensure
    object.x = old_x
    object.y = old_y
  end
  # ...
  private

  def collides_with_poly?(poly)
    if poly
      poly.each_slice(2) do |x, y|
        return true if Utils.point_in_poly(x, y, *box)
      end
      box.each_slice(2) do |x, y|
        return true if Utils.point_in_poly(x, y, *poly)
      end
    end
    false
  end
  # ...
end
```

It’s probably not the most elegant solution you could come up with, but `can_move_to?` temporarily changes `Tank` location to make a collision test, and then reverts old coordinates just before returning the result. Now our tanks stop with banging sound when they hit each other.



Tanks colliding

Catching Bullets

Right now bullets fly right through our tanks, and we want them to collide. It's a pretty simple change, which mostly affects `BulletPhysics` class:

```
# 06-physics/entities/components/bullet_physics.rb
class BulletPhysics < Component
  # ...
  def update
    # ...
    check_hit
    object.explode if arrived?
  end
  # ...
  private

  def check_hit
    @object_pool.nearby(object, 50).each do |obj|
      next if obj == object.source # Don't hit source tank
      if Utils.point_in_poly(x, y, *obj.box)
        object.target_x = x
        object.target_y = y
        return
      end
    end
  end
  # ...
end
```

Now bullets finally hit, but don’t do any damage yet. We will come back to that soon.

Bullet hitting enemy tank

Implementing Turn Speed Penalties

Tanks cannot make turns and go into reverse at full speed while keeping it's inertia, right? It is easy to implement. Since it's related to physics, we will delegate changing `Tank`'s `@direction` to our `TankPhysics` class:

```
# 06-physics/entities/components/player_input.rb
class PlayerInput < Component
  # ...
  def update
    # ...
    motion_buttons = [Gosu::KbW, Gosu::KbS, Gosu::KbA, Gosu::KbD]

    if any_button_down?(*motion_buttons)
      object.throttle_down = true
      object.physics.change_direction(
        change_angle(object.direction, *motion_buttons))
    else
      object.throttle_down = false
    end
    # ...
  end
  # ...
end

# 06-physics/entities/components/tank_physics.rb
class TankPhysics < Component
  # ...
```

```ruby
  def change_direction(new_direction)
    change = (new_direction - object.direction + 360) % 360
    change = 360 - change if change > 180
    if change > 90
      @speed = 0
    elsif change > 45
      @speed *= 0.33
    elsif change > 0
      @speed *= 0.66
    end
    object.direction = new_direction
  end
  # ...
end
```

Implementing Terrain Speed Penalties

Now, let's see how can we make terrain influence our movement. It sounds reasonable for `TankPhysics` to consult with `Map` about speed penalty of current tile:

```ruby
# 06-physics/entities/map.rb
class Map
  # ...
  def movement_penalty(x, y)
    tile = tile_at(x, y)
    case tile
    when @sand
      0.33
    else
      0
    end
  end
  # ...
end

# 06-physics/entities/components/tank_physics.rb
class TankPhysics < Component
  # ...
  def update
    # ...
      speed = apply_movement_penalty(@speed)
      shift = Utils.adjust_speed(speed)
    # ...
  end
  # ...

  private

  def apply_movement_penalty(speed)
    speed * (1.0 - @map.movement_penalty(x, y))
  end
  # ...
end
```

This makes all tanks move 33% slower on sand.

Implementing Health And Damage

I know you have been waiting for this. We will be implementing health system and most importantly, damage. Soo we will be ready to blow things up.

To implement this, we need to:

1. Add `TankHealth` component. Start with 100 health.
2. Render tank health next to tank itself.
3. Inflict damage to tank when it is in explosion zone
4. Render different sprite for dead tank.
5. Cut off player input when tank is dead.

Adding Health Component

If we didn't have `Component` system in place, it would be way more difficult. Now we just kick in a new class:

07-damage/entities/components/tank_health.rb

```ruby
class TankHealth < Component
  attr_accessor :health

  def initialize(object, object_pool)
    super(object)
    @object_pool = object_pool
    @health = 100
    @health_updated = true
    @last_damage = Gosu.milliseconds
  end

  def update
    update_image
  end

  def update_image
    if @health_updated
      if dead?
        text = '†'
        font_size = 25
      else
        text = @health.to_s
        font_size = 18
      end
      @image = Gosu::Image.from_text(
          $window, text,
          Gosu.default_font_name, font_size)
      @health_updated = false
    end
  end

  def dead?
    @health < 1
  end

  def inflict_damage(amount)
    if @health > 0
      @health_updated = true
      @health = [@health - amount.to_i, 0].max
      if @health < 1
```

```
        Explosion.new(@object_pool, x, y)
      end
    end
  end

  def draw(viewport)
    @image.draw(
      x - @image.width / 2,
      y - object.graphics.height / 2 -
      @image.height, 100)
  end
end
```

It hooks itself into the game right away, after we initialize it in `Tank` class:

```
class Tank < GameObject
  attr_accessor :health
  # ...
  def initialize(object_pool, input)
    # ...
    @health = TankHealth.new(self, object_pool)
    # ..
  end
  # ..
end
```

Inflicting Damage With Bullets

There are two ways to inflict damage - directly and indirectly. When bullet hits enemy tank (collides with tank bounding box), we should inflict direct damage. It can be done in `BulletPhysics#check_hit` method that we already had:

```
class BulletPhysics < Component
  # ...
  def check_hit
    @object_pool.nearby(object, 50).each do |obj|
      next if obj == object.source # Don't hit source tank
      if Utils.point_in_poly(x, y, *obj.box)
        # Direct hit - extra damage
        obj.health.inflict_damage(20)
        object.target_x = x
        object.target_y = y
        return
      end
    end
  end
  # ...
end
```

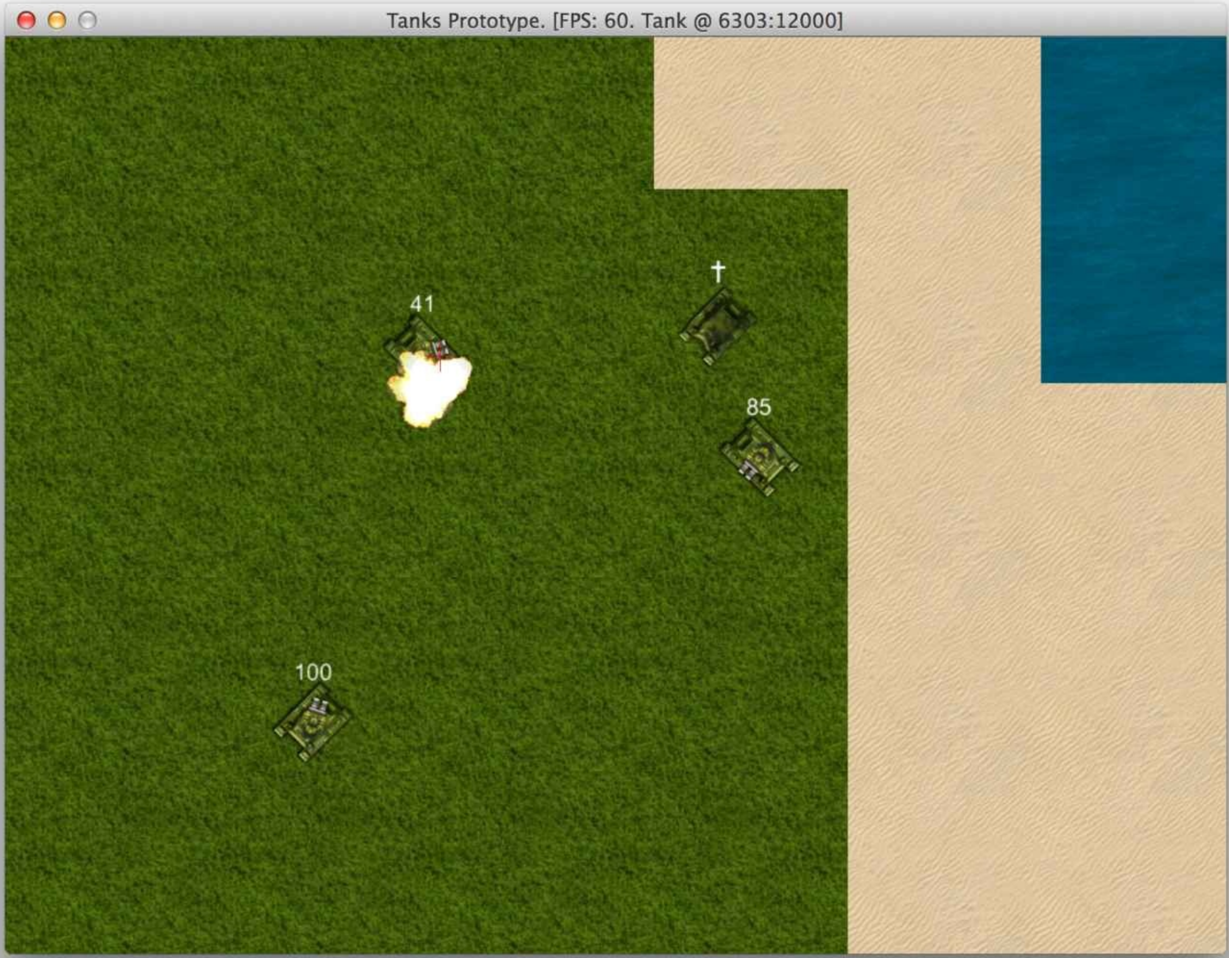
Finally, `Explosion` itself should inflict additional damage to anything that are nearby. The effect will be diminishing and it will be determined by object distance.

```
class Explosion < GameObject
  # ...
  def initialize(object_pool, x, y)
    # ...
    inflict_damage
  end

  private

  def inflict_damage
    object_pool.nearby(self, 100).each do |obj|
      if obj.class == Tank
        obj.health.inflict_damage(
          Math.sqrt(3 * 100 - Utils.distance_between(
              obj.x, obj.y, x, y)))
      end
    end
  end
end
```

This is it, we are ready to deal damage. But we want to see if we actually killed somebody, so `TankGraphics` should be aware of health and should draw different set of sprites when tank is dead. Here is what we need to change in our current TankGraphics to achieve the result:

```ruby
class TankGraphics < Component
  # ...
  def initialize(game_object)
    super(game_object)
    @body_normal = units.frame('tank1_body.png')
    @shadow_normal = units.frame('tank1_body_shadow.png')
    @gun_normal = units.frame('tank1_dualgun.png')
    @body_dead = units.frame('tank1_body_destroyed.png')
    @shadow_dead = units.frame('tank1_body_destroyed_shadow.png')
    @gun_dead = nil
  end

  def update
    if object.health.dead?
      @body = @body_dead
      @gun = @gun_dead
      @shadow = @shadow_dead
    else
      @body = @body_normal
      @gun = @gun_normal
      @shadow = @shadow_normal
    end
  end

  def draw(viewport)
    @shadow.draw_rot(x - 1, y - 1, 0, object.direction)
    @body.draw_rot(x, y, 1, object.direction)
    @gun.draw_rot(x, y, 2, object.gun_angle) if @gun
  end
  # ...
end
```

Now we can blow them up and enjoy the view:



Target practice

But what if we blow ourselves up by shooting nearby? We would still be able to move around. To fix this, we will simply cut out player input when we are dead:

```
class PlayerInput < Component
  # ...
  def update
    return if object.health.dead?
    # ...
  end
  # ...
end
```

And to prevent tank from throttling forever if the pedal was down before it got killed:

```
class TankPhysics < Component
  # ...
  def update
    if object.throttle_down && !object.health.dead?
      accelerate
    else
      decelerate
    end
    # ...
  end
  # ...
end
```

That's it. All we need right now is some resistance from those brain dead enemies. We will spark some life into them in next chapter.

Creating Artificial Intelligence

Artificial Intelligence is a subject so vast that we will barely scratch the surface. AI in Video Games is usually heavily simplified and therefore easier to implement.

There is this wonderful series of articles called Designing Artificial Intelligence for Games that I highly recommend reading to get a feeling how game AI should be done. We will be continuing our work on top of what we already have, example code for this chapter will be in `08-ai`.

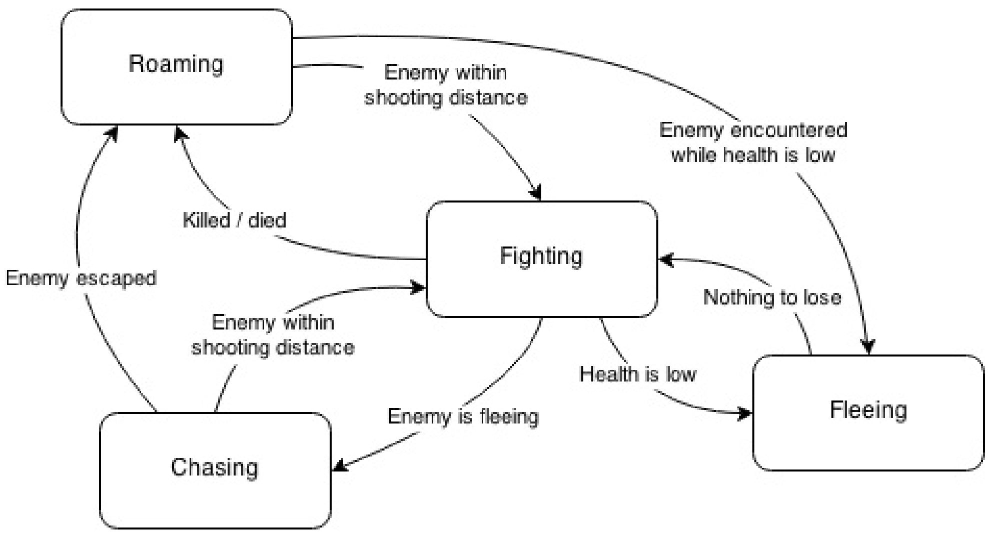
Designing AI Using Finite State Machine

Non player tanks in our game will be lone rangers, hunting everything that moves while trying to survive. We will use Finite State Machine to implement tank behavior.

First, we need to think "what would a tank do?" How about this scenario:

1. Tank wanders around, minding it's own business.
2. Tank encounters another tank. It then starts doing evasive moves and tries hitting the enemy.
3. Enemy took some damage and started driving away. Tank starts chasing the enemy trying to finish it.
4. Another tank appears and fires a couple of accurate shots, dealing serious damage. Our tank starts running away, because if it kept receiving damage at such rate, it would die very soon.
5. Tank keeps fleeing and looking for safety until it gets cornered or the opponent looks damaged too. Then tank goes into it's final battle.

We can now draw a Finite State Machine using this scenario:



Vigilante Tank FSM

If you are on a path to become a game developer, FSM should not stand for Flying Spaghetti Monster for you anymore.

Implementing AI Vision

To make opponents realistic, we have to give them senses. Let's create a class for that:

08-ai/entities/components/ai/vision.rb

```
class AiVision
  CACHE_TIMEOUT = 500
  attr_reader :in_sight

  def initialize(viewer, object_pool, distance)
    @viewer = viewer
    @object_pool = object_pool
    @distance = distance
  end

  def update
    @in_sight = @object_pool.nearby(@viewer, @distance)
  end

  def closest_tank
    now = Gosu.milliseconds
    @closest_tank = nil
    if now - (@cache_updated_at ||= 0) > CACHE_TIMEOUT
      @closest_tank = nil
      @cache_updated_at = now
    end
    @closest_tank ||= find_closest_tank
  end

  private

  def find_closest_tank
    @in_sight.select do |o|
      o.class == Tank && !o.health.dead?
    end.sort do |a, b|
      x, y = @viewer.x, @viewer.y
      d1 = Utils.distance_between(x, y, a.x, a.y)
      d2 = Utils.distance_between(x, y, b.x, b.y)
      d1 <=> d2
    end.first
  end
end
```

It uses `ObjectPool` to put nearby objects in sight, and gets a short term focus on one closest tank. Closest tank is cached for 500 milliseconds for two reasons:

1. Performance. Uncached version would do `Array#select` and `Array#sort` 60 times per second, now it will do 2 times.
2. Focus. When you choose a target, you should keep it a little longer. This should also avoid "jitters", when tank would shake between two nearby targets that are within same distance.

Controlling Tank Gun

After we made `AiVision`, we can now use it to automatically aim and shoot at closest tank. It should work like this:

1. Every instance of the gun has it's own unique combination of speed, accuracy and aggressiveness.

2. Gun will automatically target closest tank in sight.
3. If no other tank is in sight, gun will target in same direction as tank's body.
4. If other tank is aimed at and within shooting distance, gun will make a decision once in a while whether it should shoot or not, based on aggressiveness level. Aggressive tanks will be trigger happy all the time, while less aggressive ones will make small random pauses between shots.
5. Gun will have a "desired" angle that it will be automatically adjusting to, according to it's speed.

Here is the implementation:

08-ai/entities/components/ai/gun.rb

```ruby
class AiGun
  DECISION_DELAY = 1000
  attr_reader :target, :desired_gun_angle

  def initialize(object, vision)
    @object = object
    @vision = vision
    @desired_gun_angle = rand(0..360)
    @retarget_speed = rand(1..5)
    @accuracy = rand(0..10)
    @aggressiveness = rand(1..5)
  end

  def adjust_angle
    adjust_desired_angle
    adjust_gun_angle
  end

  def update
    if @vision.in_sight.any?
      if @vision.closest_tank != @target
        change_target(@vision.closest_tank)
      end
    else
      @target = nil
    end

    if @target
      if (0..10 - rand(0..@accuracy)).include?(
        (@desired_gun_angle - @object.gun_angle).abs.round)
        distance = distance_to_target
        if distance - 50 <= BulletPhysics::MAX_DIST
          target_x, target_y = Utils.point_at_distance(
            @object.x, @object.y, @object.gun_angle,
            distance + 10 - rand(0..@accuracy))
          if can_make_new_decision? && @object.can_shoot? &&
              should_shoot?
            @object.shoot(target_x, target_y)
          end
        end
      end
    end
  end

  def distance_to_target
    Utils.distance_between(
      @object.x, @object.y, @target.x, @target.y)
  end


  def should_shoot?
    rand * @aggressiveness > 0.5
  end

  def can_make_new_decision?
    now = Gosu.milliseconds
```

```
    if now - (@last_decision ||= 0) > DECISION_DELAY
      @last_decision = now
      true
    end
  end

  def adjust_desired_angle
    @desired_gun_angle = if @target
       Utils.angle_between(
        @object.x, @object.y, @target.x, @target.y)
    else
      @object.direction
    end
  end

  def change_target(new_target)
    @target = new_target
    adjust_desired_angle
  end

  def adjust_gun_angle
    actual = @object.gun_angle
    desired = @desired_gun_angle
    if actual > desired
      if actual - desired > 180 # 0 -> 360 fix
        @object.gun_angle = (actual + @retarget_speed) % 360
        if @object.gun_angle < desired
          @object.gun_angle = desired # damp
        end
      else
        @object.gun_angle = [actual - @retarget_speed, desired].max
      end
    elsif actual < desired
      if desired - actual > 180 # 360 -> 0 fix
        @object.gun_angle = (360 + actual - @retarget_speed) % 360
        if @object.gun_angle > desired
          @object.gun_angle = desired # damp
        end
      else
        @object.gun_angle = [actual + @retarget_speed, desired].min
      end
    end
  end
end
```

There is some math involved, but it is pretty straightforward. We need to find out an angle between two points, to know where our gun should point, and the other thing we need is coordinates of point which is in some distance away from source at given angle. Here are those functions:

```
module Utils
  # ...
  def self.angle_between(x, y, target_x, target_y)
    dx = target_x - x
    dy = target_y - y
    (180 - Math.atan2(dx, dy) * 180 / Math::PI) + 360 % 360
  end

  def self.point_at_distance(source_x, source_y, angle, distance)
    angle = (90 - angle) * Math::PI / 180
    x = source_x + Math.cos(angle) * distance
    y = source_y - Math.sin(angle) * distance
    [x, y]
  end
  # ...
end
```

Implementing AI Input

At this point our tanks can already defend themselves, even through motion is not yet implemented. Let's wire everything we have in `AiInput` class that we had prepared earlier.

We will need a blank `TankMotionFSM` class with 3 argument initializer and empty `update`, `on_collision(with)` and `on_damage(amount)` methods for it to work:

08-ai/entities/components/ai_input.rb

```ruby
class AiInput < Component
  UPDATE_RATE = 200 # ms

  def initialize(object_pool)
    @object_pool = object_pool
    super(nil)
    @last_update = Gosu.milliseconds
  end

  def control(obj)
    self.object = obj
    @vision = AiVision.new(obj, @object_pool,
                           rand(700..1200))
    @gun = AiGun.new(obj, @vision)
    @motion = TankMotionFSM.new(obj, @vision, @gun)
  end

  def on_collision(with)
    @motion.on_collision(with)
  end

  def on_damage(amount)
    @motion.on_damage(amount)
  end

  def update
    return if object.health.dead?
    @gun.adjust_angle
    now = Gosu.milliseconds
    return if now - @last_update < UPDATE_RATE
    @last_update = now
    @vision.update
    @gun.update
    @motion.update
  end
end
```

It adjust gun angle all the time, but does updates at `UPDATE_RATE` to save CPU power. AI is usually one of the most CPU intensive things in games, so it's a common practice to execute it less often. Refreshing enemy brains 5 per second is enough to make them deadly.

Make sure you spawn some AI controlled tanks in `PlayState` and try killing them now. I bet they will eventually get you even while standing still. You can also make tanks spawn below mouse cursor when you press `T` key:

```ruby
class PlayState < GameState
  # ...
  def initialize
    # ...
    10.times do |i|
      Tank.new(@object_pool, AiInput.new(@object_pool))
    end
  end
  # ...
  def button_down(id)
    # ...
    if id == Gosu::KbT
      t = Tank.new(@object_pool,
                   AiInput.new(@object_pool))
      t.x, t.y = @camera.mouse_coords
    end
    # ...
  end
```

```
  # ...
end
```

Implementing Tank Motion States

This is the place where we will need Finite State Machine to get things right. We will design it like this:

1. `TankMotionFSM` will decide which motion state tank should be in, considering various parameters, e.g. existence of target or lack thereof, health, etc.
2. There will be `TankMotionState` base class that will offer common methods like `drive`, `wait` and `on_collision`.
3. Concrete motion classes will implement `update`, `change_direction` and other methods, that will fiddle with `Tank#throttle_down` and `Tank#direction` to make it move and turn.

We will begin with `TankMotionState`:

08-ai/entities/components/ai/tank_motion_state.rb

```
class TankMotionState
  def initialize(object, vision)
    @object = object
    @vision = vision
  end

  def enter
    # Override if necessary
  end

  def change_direction
    # Override
  end

  def wait_time
    # Override and return a number
  end

  def drive_time
    # Override and return a number
  end

  def turn_time
    # Override and return a number
  end

  def update
    # Override
  end

  def wait
    @sub_state = :waiting
    @started_waiting = Gosu.milliseconds
    @will_wait_for = wait_time
    @object.throttle_down = false
  end

  def drive
    @sub_state = :driving
    @started_driving = Gosu.milliseconds
    @will_drive_for = drive_time
    @object.throttle_down = true
  end

  def should_change_direction?
    return true unless @changed_direction_at
    Gosu.milliseconds - @changed_direction_at >
```

```
      @will_keep_direction_for
  end

  def substate_expired?
    now = Gosu.milliseconds
    case @sub_state
    when :waiting
      true if now - @started_waiting > @will_wait_for
    when :driving
      true if now - @started_driving > @will_drive_for
    else
      true
    end
  end

  def on_collision(with)
    change = case rand(0..100)
    when 0..30
      -90
    when 30..60
      90
    when 60..70
      135
    when 80..90
      -135
    else
      180
    end
    @object.physics.change_direction(
      @object.direction + change)
  end
end
```

Nothing extraordinary here, and we need a concrete implementation to get a feeling how it would work, therefore let's examine `TankRoamingState`. It will be the default state which tank would be in if there were no enemies around.

Tank Roaming State

08-ai/entities/components/ai/tank_roaming_state.rb

```
class TankRoamingState < TankMotionState
  def initialize(object, vision)
    super
    @object = object
    @vision = vision
  end

  def update
    change_direction if should_change_direction?
    if substate_expired?
      rand > 0.3 ? drive : wait
    end
  end

  def change_direction
    change = case rand(0..100)
    when 0..30
      -45
    when 30..60
      45
    when 60..70
      90
    when 80..90
      -90
    else
      0
    end
    if change != 0
      @object.physics.change_direction(
        @object.direction + change)
    end
```

```
    @changed_direction_at = Gosu.milliseconds
    @will_keep_direction_for = turn_time
  end

  def wait_time
    rand(500..2000)
  end

  def drive_time
    rand(1000..5000)
  end

  def turn_time
    rand(2000..5000)
  end
end
```

The logic here:

1. Tank will randomly change direction every `turn_time` interval, which is between 2 and 5 seconds.
2. Tank will choose to drive (80% chance) or to stand still (20% chance).
3. If tank chose to drive, it will keep driving for `drive_time`, which is between 1 and 5 seconds.
4. Same goes with waiting, but `wait_time` (0.5 - 2 seconds) will be used for duration.
5. Direction changes and driving / waiting are independent.

This will make an impression that our tank is driving around looking for enemies.

Tank Fighting State

When tank finally sees an opponent, it will start fighting. Fighting motion should be more energetic than roaming, we will need a sharper set of choices in `change_direction` among other things.

08-ai/entities/components/ai/tank_fighting_state.rb

```
class TankFightingState < TankMotionState
  def initialize(object, vision)
    super
    @object = object
    @vision = vision
  end

  def update
    change_direction if should_change_direction?
    if substate_expired?
      rand > 0.2 ? drive : wait
    end
  end

  def change_direction
    change = case rand(0..100)
    when 0..20
      -45
    when 20..40
      45
    when 40..60
      90
    when 60..80
      -90
    when 80..90
      135
    when 90..100
      -135
    end
```

```
    @object.physics.change_direction(
      @object.direction + change)
    @changed_direction_at = Gosu.milliseconds
    @will_keep_direction_for = turn_time
  end

  def wait_time
    rand(300..1000)
  end

  def drive_time
    rand(2000..5000)
  end

  def turn_time
    rand(500..2500)
  end
end
```

We will have much less waiting and much more driving and turning.

Tank Chasing State

If opponent is fleeing, we will want to set our direction towards the opponent and hit pedal to the metal. No waiting here. `AiGun#desired_gun_angle` will point directly to our enemy.

08-ai/entities/components/ai/tank_chasing_state.rb

```
class TankChasingState < TankMotionState
  def initialize(object, vision, gun)
    super(object, vision)
    @object = object
    @vision = vision
    @gun = gun
  end

  def update
    change_direction if should_change_direction?
    drive
  end

  def change_direction
    @object.physics.change_direction(
      @gun.desired_gun_angle -
      @gun.desired_gun_angle % 45)

    @changed_direction_at = Gosu.milliseconds
    @will_keep_direction_for = turn_time
  end

  def drive_time
    10000
  end

  def turn_time
    rand(300..600)
  end
end
```

Tank Fleeing State

Now, if our health is low, we will do the opposite of chasing. Gun will be pointing and shooting at the opponent, but we want body to move away, so we won't get ourselves killed. It is very similar to `TankChasingState` where `change_direction` adds extra 180 degrees to the equation, but there is one more thing. Tank can only flee for a while. Then it gets itself together and goes into final battle. That's why we provide `can_flee?` method that `TankMotionFSM` will consult with before entering fleeing state.

We have implemented all the states, that means we are moments away from actually playable prototype with tank bots running around and fighting with you and each other.

Wiring Tank Motion States Into Finite State Machine

Implementing `TankMotionFSM` after we have all motion states ready is surprisingly easy:

08-ai/entities/components/ai/tank_motion_fsm.rb

```ruby
class TankMotionFSM
  STATE_CHANGE_DELAY = 500

  def initialize(object, vision, gun)
    @object = object
    @vision = vision
    @gun = gun
    @roaming_state = TankRoamingState.new(object, vision)
    @fighting_state = TankFightingState.new(object, vision)
    @fleeing_state = TankFleeingState.new(object, vision, gun)
    @chasing_state = TankChasingState.new(object, vision, gun)
    set_state(@roaming_state)
  end

  def on_collision(with)
    @current_state.on_collision(with)
  end

  def on_damage(amount)
    if @current_state == @roaming_state
      set_state(@fighting_state)
    end
  end

  def update
    choose_state
    @current_state.update
  end

  def set_state(state)
    return unless state
    return if state == @current_state
    @last_state_change = Gosu.milliseconds
    @current_state = state
    state.enter
  end

  def choose_state
    return unless Gosu.milliseconds -
      (@last_state_change) > STATE_CHANGE_DELAY
    if @gun.target
      if @object.health.health > 40
        if @gun.distance_to_target > BulletPhysics::MAX_DIST
          new_state = @chasing_state
        else
          new_state = @fighting_state
        end
      else
        if @fleeing_state.can_flee?
          new_state = @fleeing_state
        else
          new_state = @fighting_state
        end
      end
    else
      new_state = @roaming_state
    end
    set_state(new_state)
  end
end
```

All the logic is in `choose_state` method, which is pretty ugly and procedural, but it does the job. The code should be easy to understand, so instead of describing it, here is a picture worth thousand words:

First real battle

You may notice a new crosshair, which replaced the old one that was never visible:

```
class Camera
  # ...
  def draw_crosshair
    factor = 0.5
    x = $window.mouse_x
    y = $window.mouse_y
    c = crosshair
    c.draw(x - c.width * factor / 2,
           y - c.height * factor / 2,
           1000, factor, factor)
  end
  # ...
  private

  def crosshair
    @crosshair ||= Gosu::Image.new(
      $window, Utils.media_path('c_dot.png'), false)
  end
end
```

However this new crosshair didn't help me win, I got my ass kicked badly. Increasing game window size helped, but we obviously need to fine tune many things in this AI, to make it smart and challenging rather than dumb and deadly accurate.

Making The Prototype Playable

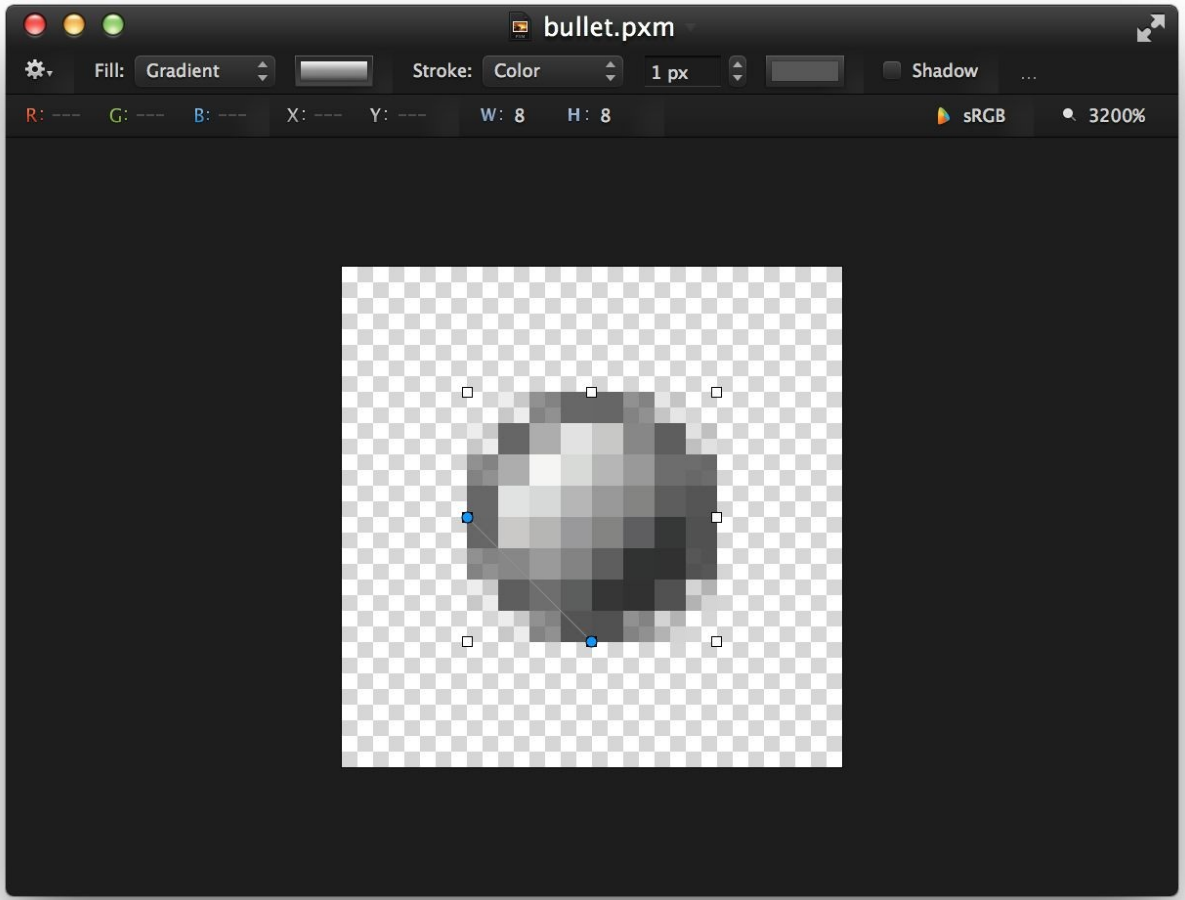
Right now we have a somewhat playable, but boring prototype without any scores or winning conditions. You can just run around and shoot other tanks. Nobody would play a game like this, hence we need to to add the missing parts. There is a crazy amount of them. It is time to give it a thorough play through and write down all the ideas and pain points about the prototype.

Here is my list:

1. Enemy tanks do not respawn.
2. Enemy tanks shoot at my current location, not at where I will be when bullet hits me.
3. Enemy tanks don't avoid collisions.
4. Random maps are boring and lack detail, could use more tiles or random environment objects.
5. Bullets are hard to see on green surface.
6. Hard to tell where enemies are coming from, radar would help.
7. Sounds play at full volume even when something happens across the whole map.
8. My tank should respawn after it's dead.
9. Motion and firing mechanics seem clumsy.
10. Map boundaries are visible when you come to the edge.
11. Enemy tank movement patterns need polishing and improvement.
12. Both my tank and enemies don't have any identity. Sometimes hard to distinguish who is who.
13. No idea who has most kills. HUD with score and some state that displays score details would help.
14. Would be great to have random powerups like health, extra damage.
15. Explosions don't leave a trace.
16. Tanks could leave trails.
17. Dead tanks keep piling up and cluttering the map.
18. Camera should be scouting ahead of you when you move, not dragging behind.
19. Bullets seem to accelerate.

This will keep us busy for a while, but in the end we will probably have something that will hopefully be able to entertain people for more than 3 minutes.

Some items on this list are easy fixes. After playing around with Pixelmator for 15 minutes, I ended up with a bullet that is visible on both light and dark backgrounds:



Highly visible bullet

Motion and firing mechanics will either have to be tuned setting by setting, or rewritten from scratch. Implementing score system, powerups and improving enemy AI deserve to have chapters of their own. The rest can be taken care of right away.

Drawing Water Beyond Map Boundaries

We don't want to see darkness when we come to the edge of game world. Luckily, it is a trivial fix. In `Map#draw` we check if tile exists in map before drawing it. When tile does not exist, we can draw water instead. And we can always fallback to water tile in `Map#tile_at`:

```
class Map
  # ...
  def draw(viewport)
    viewport.map! { |p| p / TILE_SIZE }
    x0, x1, y0, y1 = viewport.map(&:to_i)
    (x0..x1).each do |x|
      (y0..y1).each do |y|
        row = @map[x]
        map_x = x * TILE_SIZE
        map_y = y * TILE_SIZE
        if row
          tile = @map[x][y]
          if tile
            tile.draw(map_x, map_y, 0)
          else
            @water.draw(map_x, map_y, 0)
```

```ruby
          end
        else
          @water.draw(map_x, map_y, 0)
        end
      end
    end
  end
  # ...
  private
  # ...
  def tile_at(x, y)
    t_x = ((x / TILE_SIZE) % TILE_SIZE).floor
    t_y = ((y / TILE_SIZE) % TILE_SIZE).floor
    row = @map[t_x]
    row ? row[t_y] : @water
  end
  # ...
end
```

Now the edge looks much better:

Map edge

Generating Tree Clusters

To make the map more fun to play at, we will generate some trees. Let's start with `Tree` class:

09-polishing/entities/tree.rb

```ruby
class Tree < GameObject
  attr_reader :x, :y, :health, :graphics

  def initialize(object_pool, x, y, seed)
    super(object_pool)
    @x, @y = x, y
    @graphics = TreeGraphics.new(self, seed)
    @health = Health.new(self, object_pool, 30, false)
    @angle = rand(-15..15)
  end

  def on_collision(object)
    @graphics.shake(object.direction)
  end

  def box
    [x, y]
  end
end
```

Nothing fancy here, we want it to shake on collision, and it has graphics and health. `seed` will used to generate clusters of similar trees. Let's take a look at `TreeGraphics`:

09-polishing/entities/components/tree_graphics.rb

```
class TreeGraphics < Component
  SHAKE_TIME = 100
  SHAKE_COOLDOWN = 200
  SHAKE_DISTANCE = [2, 1, 0, -1, -2, -1, 0, 1, 0, -1, 0]
  def initialize(object, seed)
    super(object)
    load_sprite(seed)
  end

  def shake(direction)
    now = Gosu.milliseconds
    return if @shake_start &&
      now - @shake_start < SHAKE_TIME + SHAKE_COOLDOWN
    @shake_start = now
    @shake_direction = direction
    @shaking = true
  end

  def adjust_shake(x, y, shaking_for)
    elapsed = [shaking_for, SHAKE_TIME].min / SHAKE_TIME.to_f
    frame = ((SHAKE_DISTANCE.length - 1) * elapsed).floor
    distance = SHAKE_DISTANCE[frame]
    Utils.point_at_distance(x, y, @shake_direction, distance)
  end

  def draw(viewport)
    if @shaking
      shaking_for = Gosu.milliseconds - @shake_start
      shaking_x, shaking_y = adjust_shake(
        center_x, center_y, shaking_for)
      @tree.draw(shaking_x, shaking_y, 5)
      if shaking_for >= SHAKE_TIME
        @shaking = false
      end
    else
      @tree.draw(center_x, center_y, 5)
    end
    Utils.mark_corners(object.box) if $debug
  end

  def height
    @tree.height
  end

  def width
    @tree.width
  end

  private

  def load_sprite(seed)
    frame_list = trees.frame_list
    frame = frame_list[(frame_list.size * seed).round]
    @tree = trees.frame(frame)
  end

  def center_x
    @center_x ||= x - @tree.width / 2
  end

  def center_y
    @center_y ||= y - @tree.height / 2
  end

  def trees
    @@trees ||= Gosu::TexturePacker.load_json($window,
      Utils.media_path('trees_packed.json'))
  end
end
```

Shaking is probably the most interesting part here. When `shake` is called, graphics will start drawing tree shifted in given direction by amount defined in `SHAKE_DISTANCE` array. `draw` will be stepping through `SHAKE_DISTANCE` depending on `SHAKE_TIME`, and it will not be shaken again for `SHAKE_COOLDOWN` period, to avoid infinite shaking while driving into it.

We also need some adjustments to `TankPhysics` and `Tank` to be able to hit trees. First, we want to create an empty `on_collision(object)` method in `GameObject` class, so all game objects will be able to collide.

Then, `TankPhysics` starts calling `Tank#on_collision` when collision is detected:

```
class Tank < GameObject
  # ...
  def on_collision(object)
    return unless object
    # Avoid recursion
    if object.class == Tank
      # Inform AI about hit
      object.input.on_collision(object)
    else
      # Call only on non-tanks to avoid recursion
      object.on_collision(self)
    end
    # Bullets should not slow Tanks down
    if object.class != Bullet
      @sounds.collide if @physics.speed > 1
    end
  end
  # ...
end
```

The final ingredient to our `Tree` is `Health`, which is extracted from `TankHealth` to reduce duplication. `TankHealth` now extends it:

09-polishing/entities/components/health.rb

```
class Health < Component
  attr_accessor :health

  def initialize(object, object_pool, health, explodes)
    super(object)
    @explodes = explodes
    @object_pool = object_pool
    @initial_health = @health = health
    @health_updated = true
  end

  def restore
    @health = @initial_health
    @health_updated = true
  end

  def dead?
    @health < 1
  end

  def update
    update_image
  end

  def inflict_damage(amount)
    if @health > 0
      @health_updated = true
      @health = [@health - amount.to_i, 0].max
      after_death if dead?
    end
  end

  def draw(viewport)
    return unless draw?
```

```ruby
    @image && @image.draw(
      x - @image.width / 2,
      y - object.graphics.height / 2 -
      @image.height, 100)
  end

  protected

  def draw?
    $debug
  end

  def update_image
    return unless draw?
    if @health_updated
      text = @health.to_s
      font_size = 18
      @image = Gosu::Image.from_text(
          $window, text,
          Gosu.default_font_name, font_size)
      @health_updated = false
    end
  end

  def after_death
    if @explodes
      if Thread.list.count < 8
        Thread.new do
          sleep(rand(0.1..0.3))
          Explosion.new(@object_pool, x, y)
          sleep 0.3
          object.mark_for_removal
        end
      else
        Explosion.new(@object_pool, x, y)
        object.mark_for_removal
      end
    else
      object.mark_for_removal
    end
  end
end
```

Yes, you can make tree explode when it's destroyed. And it causes cool chain reactions blowing up whole tree clusters. But let's not do that, because we will add something more appropriate for explosions.

Our `Tree` is ready to fill the landscape. We will do it in `Map` class, which will now need to know about `ObjectPool`, because trees will go there.

```ruby
class Map
  # ...
  def initialize(object_pool)
    load_tiles
    @object_pool = object_pool
    object_pool.map = self
    @map = generate_map
    generate_trees
  end
  # ...
  def generate_trees
    noises = Perlin::Noise.new(2)
    contrast = Perlin::Curve.contrast(
      Perlin::Curve::CUBIC, 2)
    trees = 0
    target_trees = rand(300..500)
    while trees < target_trees do
      x = rand(0..MAP_WIDTH * TILE_SIZE)
      y = rand(0..MAP_HEIGHT * TILE_SIZE)
      n = noises[x * 0.001, y * 0.001]
      n = contrast.call(n)
      if tile_at(x, y) == @grass && n > 0.5
        Tree.new(@object_pool, x, y, n * 2 - 1)
```

```ruby
        trees += 1
      end
    end
  end
  # ...
end
```

Perlin noise is used in similar fashion as it was when we generated map tiles. We allow creating trees only if noise level is above `0.5`, and use noise level as seed value - `n * 2 - 1` will be a number between 0 and 1 when `n` is in `0.5..1` range. And we only allow creating trees on grass tiles.

Now our map looks a little better:

Hiding among procedurally generated trees

Generating Random Objects

Trees are great, but we want more detail. Let's spice things up with explosive boxes and barrels. They will be using the same class with single sprite sheet, so while the sprite will be chosen randomly, behavior will be the same. This new class will be called `Box`:

09-polishing/entities/box.rb

```ruby
class Box < GameObject
  attr_reader :x, :y, :health, :graphics, :angle

  def initialize(object_pool, x, y)
    super(object_pool)
    @x, @y = x, y
    @graphics = BoxGraphics.new(self)
    @health = Health.new(self, object_pool, 10, true)
    @angle = rand(-15..15)
  end

  def on_collision(object)
    return unless object.physics.speed > 1.0
    @x, @y = Utils.point_at_distance(@x, @y, object.direction, 2)
    @box = nil
  end

  def box
    return @box if @box
    w = @graphics.width / 2
```

```ruby
    h = @graphics.height / 2
    # Bounding box adjusted to trim shadows
    @box = [x - w + 4,      y - h + 8,
            x + w , y - h + 8,
            x + w , y + h,
            x - w + 4,      y + h]
    @box = Utils.rotate(@angle, @x, @y, *@box)
  end
end
```

It will be generated with slight random angle, to preserve realistic shadows but give an impression of chaotic placement. Tanks will also be able to push boxes a little on collision, but only when going fast enough. `Health` component is the same one that `Tree` has, but initialized with less health and explosive flag is `true`, so the box will blow up after one hit and deal extra damage to the surroundings.

`BoxGraphics` is nothing fancy, it just loads random sprite upon initialization:

09-polishing/entities/components/box_graphics.rb

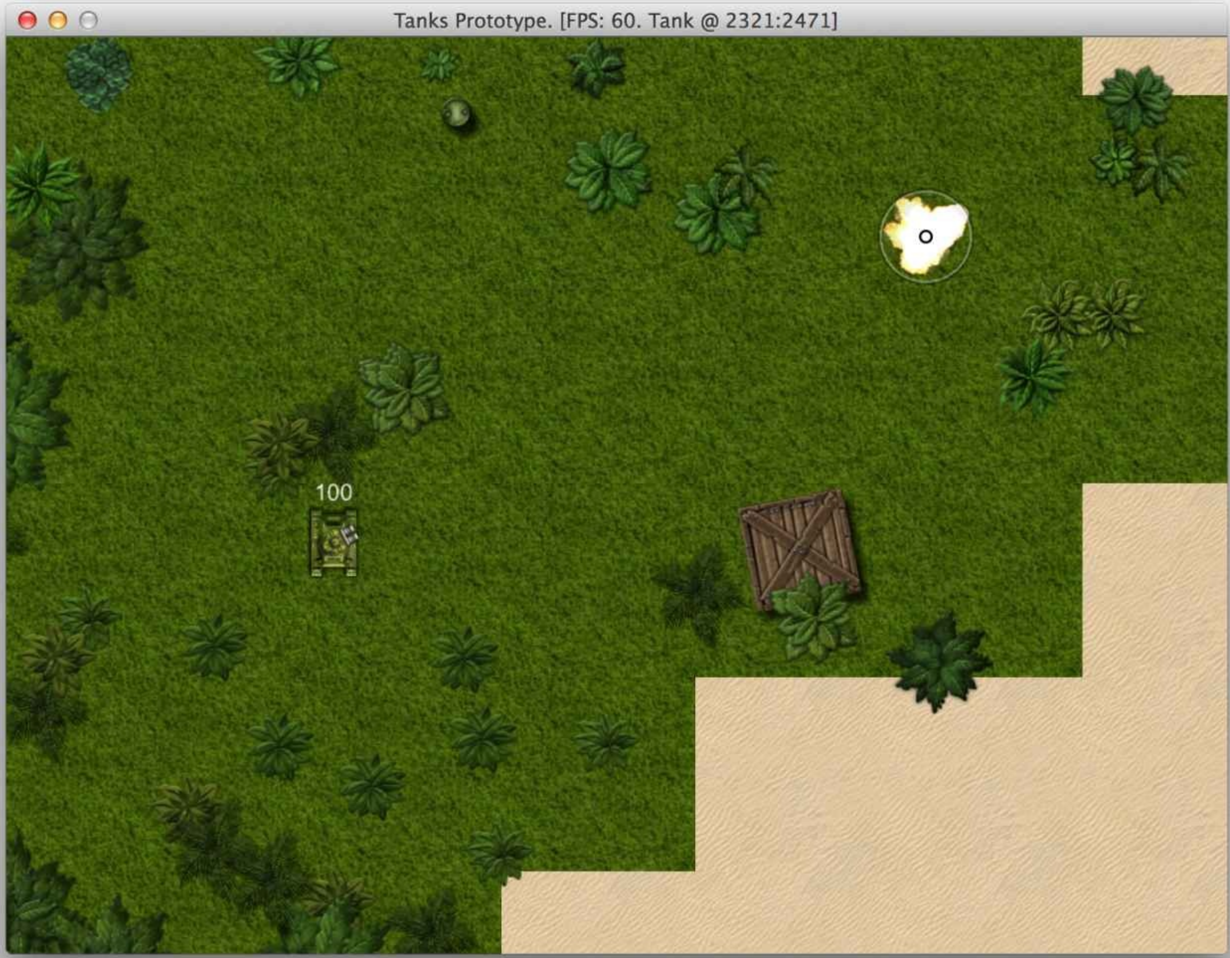
```ruby
class BoxGraphics < Component
  def initialize(object)
    super(object)
    load_sprite
  end

  def draw(viewport)
    @box.draw_rot(x, y, 0, object.angle)
    Utils.mark_corners(object.box) if $debug
  end

  def height
    @box.height
  end

  def width
    @box.width
  end

  private

  def load_sprite
    frame = boxes.frame_list.sample
    @box = boxes.frame(frame)
  end

  def center_x
    @center_x ||= x - width / 2
  end

  def center_y
    @center_y ||= y - height / 2
  end

  def boxes
    @@boxes ||= Gosu::TexturePacker.load_json($window,
      Utils.media_path('boxes_barrels.json'))
  end
end
```

Time to generate boxes in our `Map`. It will be similar to trees, but we won't need Perlin noise, since there will be way fewer boxes than trees, so we don't need to form patterns. All we need to do is to check if we're not generating box on water.

```ruby
class Map
  # ...
  def initialize(object_pool)
    # ...
    generate_boxes
```

```ruby
  end
  # ...
  def generate_boxes
    boxes = 0
    target_boxes = rand(10..30)
    while boxes < target_boxes do
      x = rand(0..MAP_WIDTH * TILE_SIZE)
      y = rand(0..MAP_HEIGHT * TILE_SIZE)
      if tile_at(x, y) != @water
        Box.new(@object_pool, x, y)
        boxes += 1
      end
    end
  end
  # ...
end
```

Now give it a go. Beautiful, isn’t it?



Boxes and barrels in the jungle

Implementing A Radar

With all the visual noise it is getting increasingly difficult to see enemy tanks. That's why we will implement a `Radar` to help ourselves.

09-polishing/entities/radar.rb

```
class Radar
  UPDATE_FREQUENCY = 1000
  WIDTH = 150
  HEIGHT = 100
  PADDING = 10
  # Black with 33% transparency
  BACKGROUND = Gosu::Color.new(255 * 0.33, 0, 0, 0)
  attr_accessor :target

  def initialize(object_pool, target)
    @object_pool = object_pool
    @target = target
    @last_update = 0
  end

  def update
    if Gosu.milliseconds - @last_update > UPDATE_FREQUENCY
      @nearby = nil
    end
    @nearby ||= @object_pool.nearby(@target, 2000).select do |o|
      o.class == Tank && !o.health.dead?
```

```ruby
    end
  end

  def draw
    x1, x2, y1, y2 = radar_coords
    $window.draw_quad(
      x1, y1, BACKGROUND,
      x2, y1, BACKGROUND,
      x2, y2, BACKGROUND,
      x1, y2, BACKGROUND,
      200)
    draw_tank(@target, Gosu::Color::GREEN)
    @nearby && @nearby.each do |t|
      draw_tank(t, Gosu::Color::RED)
    end
  end

  private

  def draw_tank(tank, color)
    x1, x2, y1, y2 = radar_coords
    tx = x1 + WIDTH / 2 + (tank.x - @target.x) / 20
    ty = y1 + HEIGHT / 2 +  (tank.y - @target.y) / 20
    if (x1..x2).include?(tx) && (y1..y2).include?(ty)
      $window.draw_quad(
        tx - 2, ty - 2, color,
        tx + 2, ty - 2, color,
        tx + 2, ty + 2, color,
        tx - 2, ty + 2, color,
        300)
    end
  end

  def radar_coords
    x1 = $window.width - WIDTH - PADDING
    x2 = $window.width - PADDING
    y1 = $window.height - HEIGHT - PADDING
    y2 = $window.height - PADDING
    [x1, x2, y1, y2]
  end
end
```

`Radar`, like `Camera`, also has a target. It uses `ObjectPool` to query nearby objects and filters out instances of alive `Tank`. Then it draws a transparent black background and small dots for each tank, green for target, red for the rest.

To avoid querying `ObjectPool` too often, `Radar` updates itself only once every second.

It is initialized, updated and drawn in `PlayState`, right after `Camera`:

```ruby
class PlayState < GameState
  # ...
  def initialize
    # ...
    @camera.target = @tank
    @radar = Radar.new(@object_pool, @tank)
    # ...
  end
  # ...
  def update
    # ...
    @camera.update
    @radar.update
    # ...
  end
  # ...
  def draw
    # ...
    @camera.draw_crosshair
    @radar.draw
  end
  # ...
end
```

Time to enjoy the results.

Radar in action

Dynamic Sound Volume And Panning

We have improved the visuals, but sound is still terrible. Like some superhero, you can hear everything that happens in the map, and it can drive you insane. We will fix that in a moment.

The idea is to make everything that happens further away from camera target sound less loud, until the sound fades away completely. To make player's experience more immersive, we will also take advantage of stereo speakers - sounds should appear to be coming from the right direction.

Unfortunately, `Gosu::Sample#play_pan` does not work as one would expect it to. If you play the sample with just a little panning, it completely cuts off the opposite channel, meaning that if you play a sample with pan level of `0.1` (10% to the right), you would expect to hear something in left speaker as well. The actual behavior is that sound plays through the right speaker pretty loudly, and if you increase pan level to, say, `0.7`, you will hear the sound through right speaker again, but it will be way more silent.

To implement realistic stereo sounds that come through both speakers when panned, we need to play two samples with opposite `pan` level. After some experimenting, I discovered that fiddling with pan level makes things sound weird, while playing with volume produces softer, more subtle effect. This is what I ended up having:

09-polishing/misc/stereo_sample.rb

```ruby
class StereoSample
  @@all_instances = []

  def self.register_instances(instances)
    @@all_instances << instances
  end

  def self.cleanup
    @@all_instances.each do |instances|
      instances.each do |key, instance|
        unless instance.playing? || instance.paused?
          instances.delete(key)
        end
      end
    end
  end

  def initialize(window, sound_l, sound_r = sound_l)
    @sound_l = Gosu::Sample.new(window, sound_l)
    # Use same sample in mono -> stereo
    if sound_l == sound_r
      @sound_r = @sound_l
    else
      @sound_r = Gosu::Sample.new(window, sound_r)
    end
    @instances = {}
    self.class.register_instances(@instances)
  end

  def paused?(id = :default)
    i = @instances["#{id}_l"]
    i && i.paused?
  end

  def playing?(id = :default)
    i = @instances["#{id}_l"]
    i && i.playing?
  end

  def stopped?(id = :default)
    @instances["#{id}_l"].nil?
  end

  def play(id = :default, pan = 0,
           volume = 1, speed = 1, looping = false)
    @instances["#{id}_l"] = @sound_l.play_pan(
      -0.2, 0, speed, looping)
    @instances["#{id}_r"] = @sound_r.play_pan(
      0.2, 0, speed, looping)
    volume_and_pan(id, volume, pan)
  end

  def pause(id = :default)
    @instances["#{id}_l"].pause
    @instances["#{id}_r"].pause
  end

  def resume(id = :default)
    @instances["#{id}_l"].resume
    @instances["#{id}_r"].resume
  end

  def stop
    @instances.delete("#{id}_l").stop
    @instances.delete("#{id}_r").stop
  end

  def volume_and_pan(id, volume, pan)
    if pan > 0
```

```ruby
      pan_l = 1 - pan * 2
      pan_r = 1
    else
      pan_l = 1
      pan_r = 1 + pan * 2
    end
    pan_l *= volume
    pan_r *= volume
    @instances["#{id}_l"].volume = [pan_l, 0.05].max
    @instances["#{id}_r"].volume = [pan_r, 0.05].max
  end
end
```

`StereoSample` manages stereo playback of sample instances, and to avoid memory leaks, it has `cleanup` that scans all sample instances and removes samples that have finished playing. For this removal to work, we need to place a call to `StereoSample.cleanup` inside `PlayState#update` method.

To determine correct pan and volume, we will create some helper methods in `Utils` module:

```ruby
module Utils
  HEARING_DISTANCE = 1000.0
  # ...
  def self.volume(object, camera)
    return 1 if object == camera.target
    distance = Utils.distance_between(
      camera.target.x, camera.target.y,
      object.x, object.y)
    distance = [(HEARING_DISTANCE - distance), 0].max
    distance / HEARING_DISTANCE
  end

  def self.pan(object, camera)
    return 0 if object == camera.target
    pan = object.x - camera.target.x
    sig = pan > 0 ? 1 : -1
    pan = (pan % HEARING_DISTANCE) / HEARING_DISTANCE
    if sig > 0
      pan
    else
      -1 + pan
    end
  end

  def self.volume_and_pan(object, camera)
    [volume(object, camera), pan(object, camera)]
  end
end
```

Apparently, having access to `Camera` is necessary for calculating sound volume and pan, so we will add `attr_accessor :camera` to `ObjectPool` class and assign it in `PlayState` constructor. You may wonder why we didn't use `Camera#target` right away. The answer is that camera can change it's target. E.g. when your tank dies, new instance will be generated when you respawn, so if all other objects would still have the reference to your old tank, guess what you would hear?

Remastered `TankSounds` component is probably the most elaborate example of how `StereoSample` should be used:

09-polishing/entities/components/tank_sounds.rb

```ruby
class TankSounds < Component
  def initialize(object, object_pool)
    super(object)
    @object_pool = object_pool
  end
```

```ruby

  def update
    id = object.object_id
    if object.physics.moving?
      move_volume = Utils.volume(
        object, @object_pool.camera)
      pan = Utils.pan(object, @object_pool.camera)
      if driving_sound.paused?(id)
        driving_sound.resume(id)
      elsif driving_sound.stopped?(id)
        driving_sound.play(id, pan, 0.5, 1, true)
      end
      driving_sound.volume_and_pan(id, move_volume * 0.5, pan)
    else
      if driving_sound.playing?(id)
        driving_sound.pause(id)
      end
    end
  end

  def collide
    vol, pan = Utils.volume_and_pan(
      object, @object_pool.camera)
    crash_sound.play(self.object_id, pan, vol, 1, false)
  end

  private

  def driving_sound
    @@driving_sound ||= StereoSample.new(
      $window, Utils.media_path('tank_driving.mp3'))
  end

  def crash_sound
    @@crash_sound ||= StereoSample.new(
      $window, Utils.media_path('metal_interaction2.wav'))
  end
end
```

And this is how static `ExplosionSounds` looks like:

09-polishing/entities/components/explosion_sounds.rb

```ruby
class ExplosionSounds
  class << self
    def play(object, camera)
      volume, pan = Utils.volume_and_pan(object, camera)
      sound.play(object.object_id, pan, volume)
    end

    private

    def sound
      @@sound ||= StereoSample.new(
        $window, Utils.media_path('explosion.mp3'))
    end
  end
end
```

After wiring everything so that sound components have access to `ObjectPool`, the rest is straightforward.

Giving Enemies Identity

Wouldn't it be great if you could tell yourself apart from the enemies. Moreover, enemies could have names, so you would know which one is more aggressive or have, you know, personal issues with someone.

To do that we need to ask the player to input a nickname, and choose some funny names for each enemy AI. Here is a nice list we will grab: http://www.paulandstorm.com/wha/clown-names/

We first compile everything into a text filed called `names.txt`, that looks like this:

media/names.txt

```
Strippy
Boffo
Buffo
Drips
...
```

Now we need a class to parse the list and give out random names from it. We also want to limit name length to something that displays nicely.

09-polishing/misc/names.rb

```ruby
class Names
  def initialize(file)
    @names = File.read(file).split("\n").reject do |n|
      n.size > 12
    end
  end

  def random
    name = @names.sample
    @names.delete(name)
    name
  end
end
```

Then we need to place those names somewhere. We could assign them to tanks, but think ahead - if our player and AI enemies will respawn, we should give names to inputs, because `Tank` is replaceable, driver is not. Well, it is, but let's not get too deep into it.

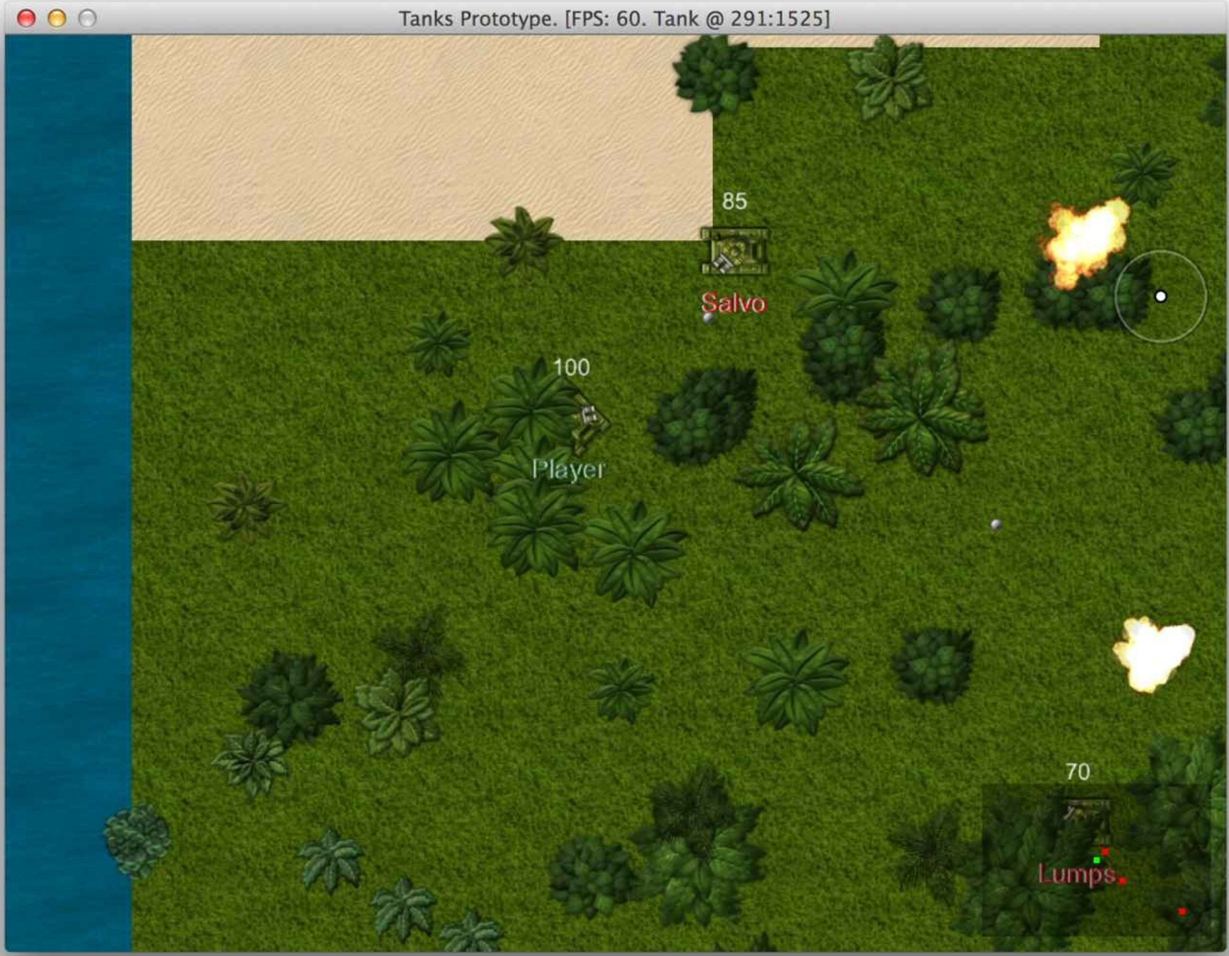
For now we just add name parameter to `PlayerInput` and `AiInput` initializers, save it in `@name` instance variable, and then add `draw(viewport)` method to make it render below the tank:

```ruby
# 09-polishing/entities/components/player_input.rb
class PlayerInput < Component
  # Dark green
  NAME_COLOR = Gosu::Color.argb(0xee084408)

  def initialize(name, camera)
    super(nil)
    @name = name
    @camera = camera
  end
  # ...
  def draw(viewport)
    @name_image ||= Gosu::Image.from_text(
      $window, @name, Gosu.default_font_name, 20)
    @name_image.draw(
      x - @name_image.width / 2 - 1,
      y + object.graphics.height / 2, 100,
      1, 1, Gosu::Color::WHITE)
    @name_image.draw(
      x - @name_image.width / 2,
      y + object.graphics.height / 2, 100,
      1, 1, NAME_COLOR)
  end
  # ...
end
```

```ruby
# 09-polishing/entities/components/ai_input.rb
class AiInput < Component
  # Dark red
  NAME_COLOR = Gosu::Color.argb(0xeeb10000)

  def initialize(name, object_pool)
    super(nil)
    @object_pool = object_pool
    @name = name
    @last_update = Gosu.milliseconds
  end
  # ...
  def draw(viewport)
    @motion.draw(viewport)
    @gun.draw(viewport)
    @name_image ||= Gosu::Image.from_text(
      $window, @name, Gosu.default_font_name, 20)
    @name_image.draw(
      x - @name_image.width / 2 - 1,
      y + object.graphics.height / 2, 100,
      1, 1, Gosu::Color::WHITE)
    @name_image.draw(
      x - @name_image.width / 2,
      y + object.graphics.height / 2, 100,
      1, 1, NAME_COLOR)
  end
  # ...
end
```

We can see that generic `Input` class can be easily extracted, but let's follow the Rule of three and not do premature refactoring.

Instead, run the game and enjoy dying from a bunch of mad clowns.



Identity makes a difference

Respawning Tanks And Removing Dead Ones

To implement respawning we could use `Map#find_spawn_point` every time we wanted to respawn, but it may get slow, because it brute forces the map for random spots that are not water. We don't want our game to start freezing when tanks are respawning, so we will change how tank spawning works. Instead of looking for a new respawn point all the time, we will pre-generate several of them for reuse.

```
class Map
  # ...
  def spawn_points(max)
    @spawn_points = (0..max).map do
      find_spawn_point
    end
    @spawn_points_pointer = 0
  end

  def spawn_point
    @spawn_points[(@spawn_points_pointer += 1) % @spawn_points.size]
  end
  # ...
end
```

Here we have `spawn_points` method that prepares a number of spawn points and stores them in `@spawn_points` instance variable, and `spawn_point` method that cycles through all

`@spawn_points` and returns them one by one. `find_spawn_point` can now become `private`.

We will use `Map#spawn_points` when initializing `PlayState` and pass `ObjectPool` to `PlayerInput` (`AiInput` already has it), so that we will be able to call `@object_pool.map.spawn_point` when needed.

```
class PlayState < GameState
  # ...
  def initialize
    # ...
    @map = Map.new(@object_pool)
    @map.spawn_points(15)
    @tank = Tank.new(@object_pool,
      PlayerInput.new('Player', @camera, @object_pool))
    # ...
    10.times do |i|
      Tank.new(@object_pool, AiInput.new(
        @names.random, @object_pool))
    end
  end
  # ...
end
```

When tank dies, we want it to stay dead for 5 seconds and then respawn in one of our predefined spawn points. We will achieve that by adding `respawn` method and calling it in `PlayerInput#update` and `AiInput#update` if tank is dead.

```
# 09-polishing/entities/components/player_input.rb
class PlayerInput < Component
  # ...
  def update
    return respawn if object.health.dead?
    # ...
  end
  # ...
  private

  def respawn
    if object.health.should_respawn?
      object.health.restore
      object.x, object.y = @object_pool.map.spawn_point
      @camera.x, @camera.y = x, y
      PlayerSounds.respawn(object, @camera)
    end
  end
  # ...
end

# 09-polishing/entities/components/ai_input.rb
class AiInput < Component
  # ...
  def update
    return respawn if object.health.dead?
    # ...
  end
  # ...
  private

  def respawn
    if object.health.should_respawn?
      object.health.restore
      object.x, object.y = @object_pool.map.spawn_point
      PlayerSounds.respawn(object, @object_pool.camera)
    end
  end
end
```

We need some changes in `TankHealth` class too:

```
class TankHealth < Health
  RESPAWN_DELAY = 5000
```

```
  # ...
  def should_respawn?
    Gosu.milliseconds - @death_time > RESPAWN_DELAY
  end
  # ...
  def after_death
    @death_time = Gosu.milliseconds
    # ...
  end
end

class Health < Component
  # ...
  def restore
    @health = @initial_health
    @health_updated = true
  end
  # ...
end
```

It shouldn't be hard to put everything together and enjoy the never ending gameplay.

You may have noticed that we also added a sound that will be played when player respawns. A nice "whoosh".

09-polishing/entities/components/player_sounds.rb

```
class PlayerSounds
  class << self
    def respawn(object, camera)
      volume, pan = Utils.volume_and_pan(object, camera)
      respawn_sound.play(object.object_id, pan, volume * 0.5)
    end

    private

    def respawn_sound
      @@respawn ||= StereoSample.new(
        $window, Utils.media_path('respawn.wav'))
    end
  end
end
```

Displaying Explosion Damage Trails

When something blows up, you expect it to leave a trail, right? In our case explosions disappear as if nothing has ever happened, and we just can't leave it like this. Let's introduce `Damage` game object that will be responsible for displaying explosion residue on sand and grass:

09-polishing/entities/damage.rb

```
class Damage < GameObject
  MAX_INSTANCES = 100
  attr_accessor :x, :y
  @@instances = []

  def initialize(object_pool, x, y)
    super(object_pool)
    DamageGraphics.new(self)
    @x, @y = x, y
    track(self)
  end

  def effect?
    true
  end

  private
```

```ruby

  def track(instance)
    if @@instances.size < MAX_INSTANCES
      @@instances << instance
    else
      out = @@instances.shift
      out.mark_for_removal
      @@instances << instance
    end
  end
end
```

`Damage` tracks it's instances and starts removing old ones when `MAX_INSTANCES` are reached. Without this optimization, the game would get increasingly slower every time somebody shoots.

We have also added a new game object trait - `effect?` returns true on `Damage` and `Explosion`, false on `Tank`, `Tree`, `Box` and `Bullet`. That way we can filter out effects when querying `ObjectPool#nearby` for collisions or enemies.

09-polishing/entities/object_pool.rb

```ruby
class ObjectPool
  attr_accessor :objects, :map, :camera

  def initialize
    @objects = []
  end

  def nearby(object, max_distance)
    non_effects.select do |obj|
      obj != object &&
        (obj.x - object.x).abs < max_distance &&
        (obj.y - object.y).abs < max_distance &&
        Utils.distance_between(
          obj.x, obj.y, object.x, object.y) < max_distance
    end
  end

  def non_effects
    @objects.reject(&:effect?)
  end
end
```

When it comes to rendering graphics, to make an impression of randomness, we will cycle through several different damage images and draw them rotated:

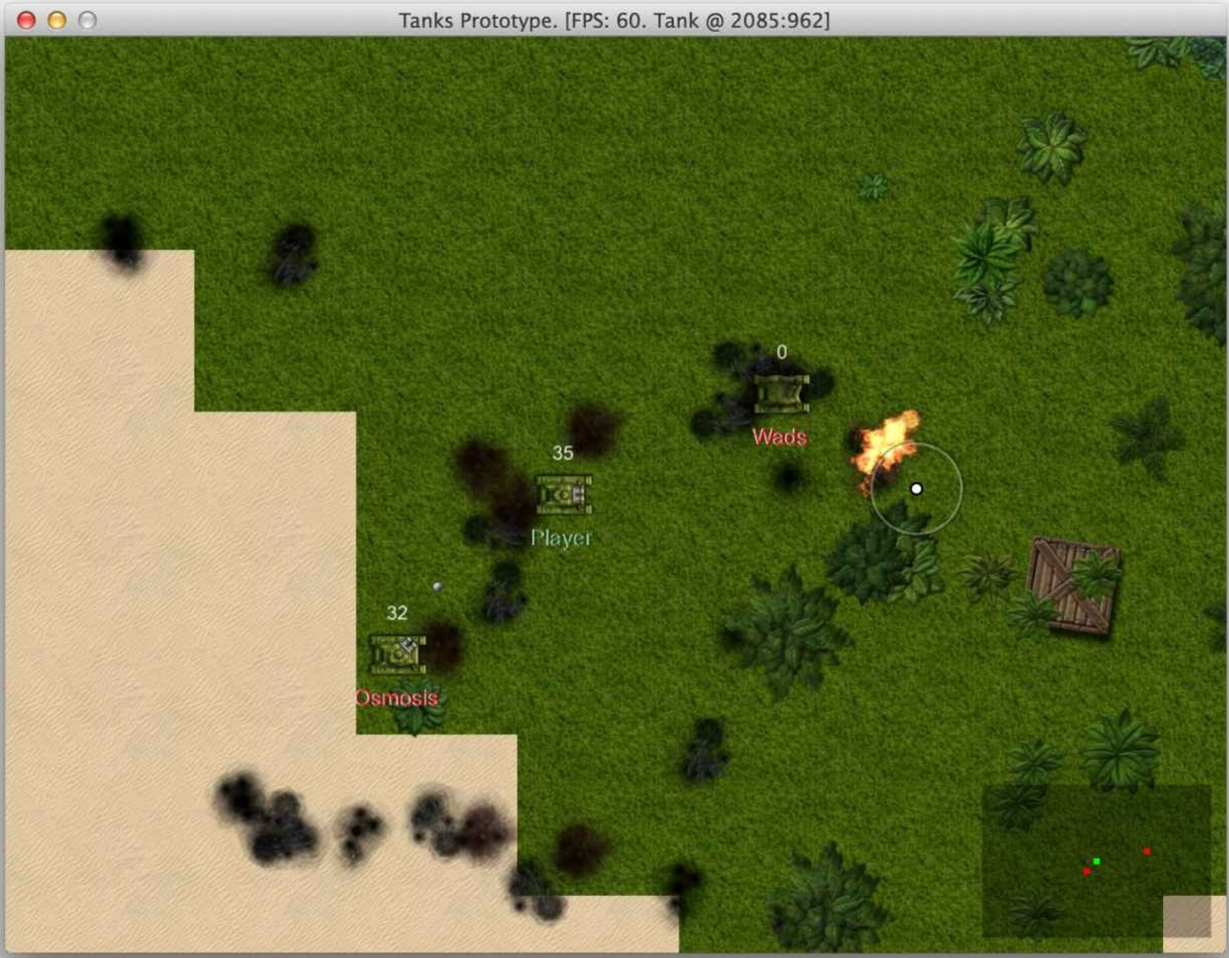
09-polishing/entities/components/damage_graphics.rb

```ruby
class DamageGraphics < Component
  def initialize(object_pool)
    super
    @image = images.sample
    @angle = rand(0..360)
  end

  def draw(viewport)
    @image.draw_rot(x, y, 0, @angle)
  end

  private

  def images
    @@images ||= (1..4).map do |i|
      Gosu::Image.new($window,
        Utils.media_path("damage#{i}.png"), false)
    end
  end
end
```

`Explosion` will be responsible for creating `Damage` instances on solid ground, just before explosion animation starts:

```
class Explosion < GameObject
  def initialize(object_pool, x, y)
    # ...
    if @object_pool.map.can_move_to?(x, y)
      Damage.new(@object_pool, @x, @y)
    end
    # ...
  end
  # ...
end
```

And this is how the result looks like:



Damaged battlefield

Debugging Bullet Physics

When playing the game, there is a feeling that bullets start out slow when fired and gain speed as time goes. Let's review `BulletPhysics#update` and think why this is happening:

```
class BulletPhysics < Component
  # ...
  def update
    fly_speed = Utils.adjust_speed(object.speed)
    fly_distance = (Gosu.milliseconds - object.fired_at) *
      0.001 * fly_speed / 2
    object.x, object.y = point_at_distance(fly_distance)
    check_hit
    object.explode if arrived?
  end
  # ...
end
```

Flaw here is very obvious. `Gosu.milliseconds - object.fired_at` will be increasingly bigger as time goes, thus increasing `fly_distance`. The fix is straightforward - we want to calculate `fly_distance` using time passed between calls to `BulletPhysics#update`, like this:

```
class BulletPhysics < Component
  # ...
  def update
    fly_speed = Utils.adjust_speed(object.speed)
```

```ruby
    now = Gosu.milliseconds
    @last_update ||= object.fired_at
    fly_distance = (now - @last_update) * 0.001 * fly_speed
    object.x, object.y = point_at_distance(fly_distance)
    @last_update = now
    check_hit
    object.explode if arrived?
  end
  # ...
end
```

But if you would run the game now, bullets would fly so slow, that you would feel like Neo in The Matrix. To fix that, we will have to tell our tank to fire bullets a little faster.

```ruby
class Tank < GameObject
  # ...
  def shoot(target_x, target_y)
    if can_shoot?
      @last_shot = Gosu.milliseconds
      Bullet.new(object_pool, @x, @y, target_x, target_y)
        .fire(self, 1500) # Old value was 100
    end
  end
  # ...
end
```

Now bullets fly like they are supposed to. I can only wonder why haven't I noticed this bug in the very beginning.

Making Camera Look Ahead

One of the most annoying things with current state of prototype is that `Camera` is dragging behind instead of showing what is in the direction you are moving. To fix the issue, we need to change the way how `Camera` moves around. First we need to know where `Camera` wants to be. We will use `Utils.point_at_distance` to choose a spot ahead of the `Tank`. Then, `Camera#update` needs to be rewritten, so `Camera` can dynamically adjust to it's desired spot. Here are the changes:

```ruby
class Camera
  # ...
  def desired_spot
    if @target.physics.moving?
      Utils.point_at_distance(
        @target.x, @target.y,
        @target.direction,
        @target.physics.speed.ceil * 25)
    else
      [@target.x, @target.y]
    end
  end
  # ...
  def update
    des_x, des_y = desired_spot
    shift = Utils.adjust_speed(
      @target.physics.speed).floor + 1
    if @x < des_x
      if des_x - @x < shift
        @x = des_x
      else
        @x += shift
      end
    elsif @x > des_x
      if @x - des_x < shift
        @x = des_x
      else
        @x -= shift
      end
    end
    if @y < des_y
```

```ruby
      if des_y - @y < shift
        @y = des_y
      else
        @y += shift
      end
    elsif @y > des_y
      if @y - des_y < shift
        @y = des_y
      else
        @y -= shift
      end
    end
    # ...
  end
  # ...
end
```

It wouldn't win code style awards, but it does the job. Game is now much more playable.

Reviewing The Changes

Let's get back to our list of improvements to see what we have done:

1. Enemy tanks do not respawn.
2. Random maps are boring and lack detail, could use more tiles or random environment objects.
3. Bullets are hard to see on green surface.
4. Hard to tell where enemies are coming from, radar would help.
5. Sounds play at full volume even when something happens across The whole map.
6. My tank should respawn after it's dead.
7. Map boundaries are visible when you come to the edge.
8. Both my tank and enemies don't have any identity. Sometimes hard to distinguish who is who.
9. Explosions don't leave a trace.
10. Dead tanks keep piling up and cluttering the map.
11. Camera should be scouting ahead of you when you move, not dragging behind.
12. Bullets seem to accelerate.

Not bad for a start. This is what we still need to cover in next couple of chapters:

1. Enemy tanks shoot at my current location, not at where I will be when bullet hits me.
2. Enemy tanks don't avoid collisions.
3. Enemy tank movement patterns need polishing and improvement.
4. No idea who has most kills. HUD with score and some state that displays score details would
5. Would be great to have random powerups like health, extra damage.
6. Motion and firing mechanics seem clumsy. help.
7. Tanks could leave trails.

I will add "Optimize ObjectPool performance", because game starts slowing down when too many objects are added to the pool, and profiling shows that `Array#select`, which is the heart of `ObjectPool#nearby`, is the main cause. Speed is one of most important features of any game, so let's not hesitate to improve it.

Dealing With Thousands Of Game Objects

Gosu is blazing fast when it comes to drawing, but there are more things going on. Namely, we use `ObjectPool#nearby` quite often to loop through thousands of objects 60 times per second to measure distances among them. This slows everything down when object pool grows.

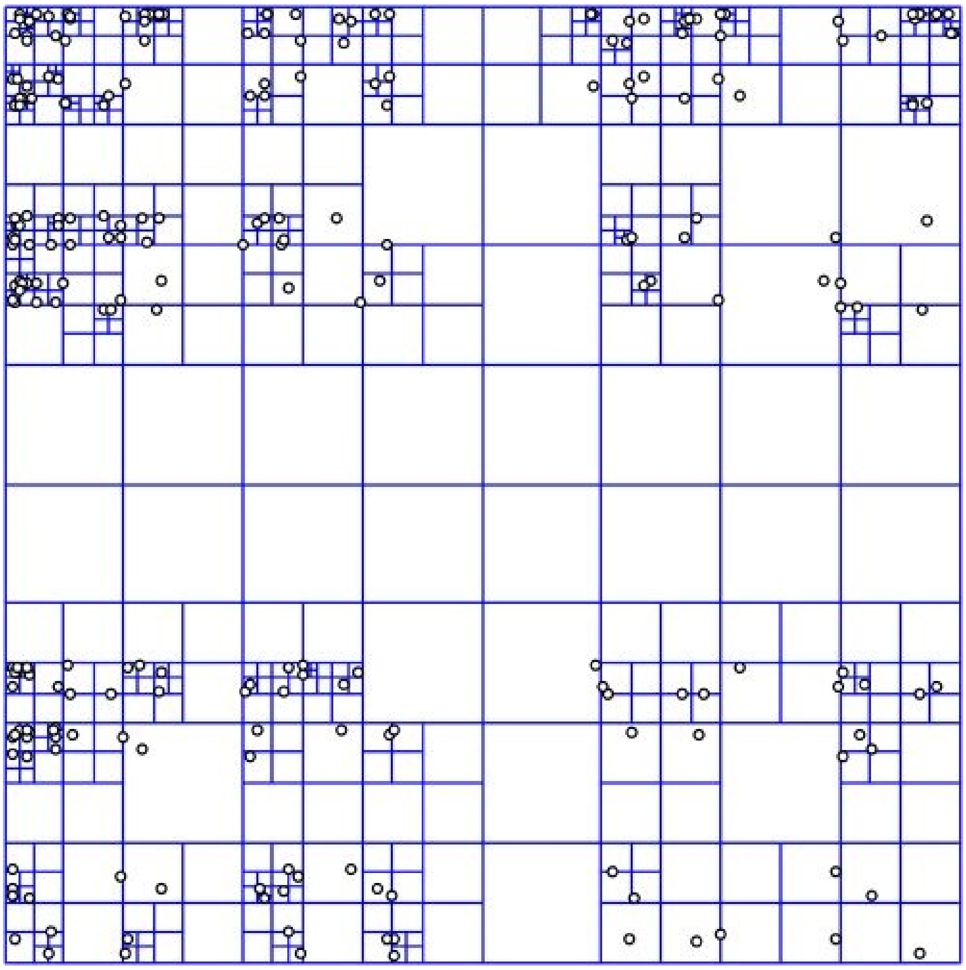
To demonstrate the effect, we will generate 1500 trees, 30 tanks, ~100 boxes and leave 1000 damage trails from explosions. It was enough to drop FPS below 30:

Running slow with thousands of game objects

Spatial Partitioning

There is a solution for this particular problem is "Spatial Partitioning", and the essence of it is that you have to use a tree-like data structure that divides space into regions, places objects there and lets you query itself in logarithmic time, omitting objects that fall out of query region. Spatial Partitioning is explained well in Game Programming Patterns.

Probably the most appropriate data structure for our 2D game is quadtree. To quote Wikipedia, "quadtrees are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions." Here is how it looks like:



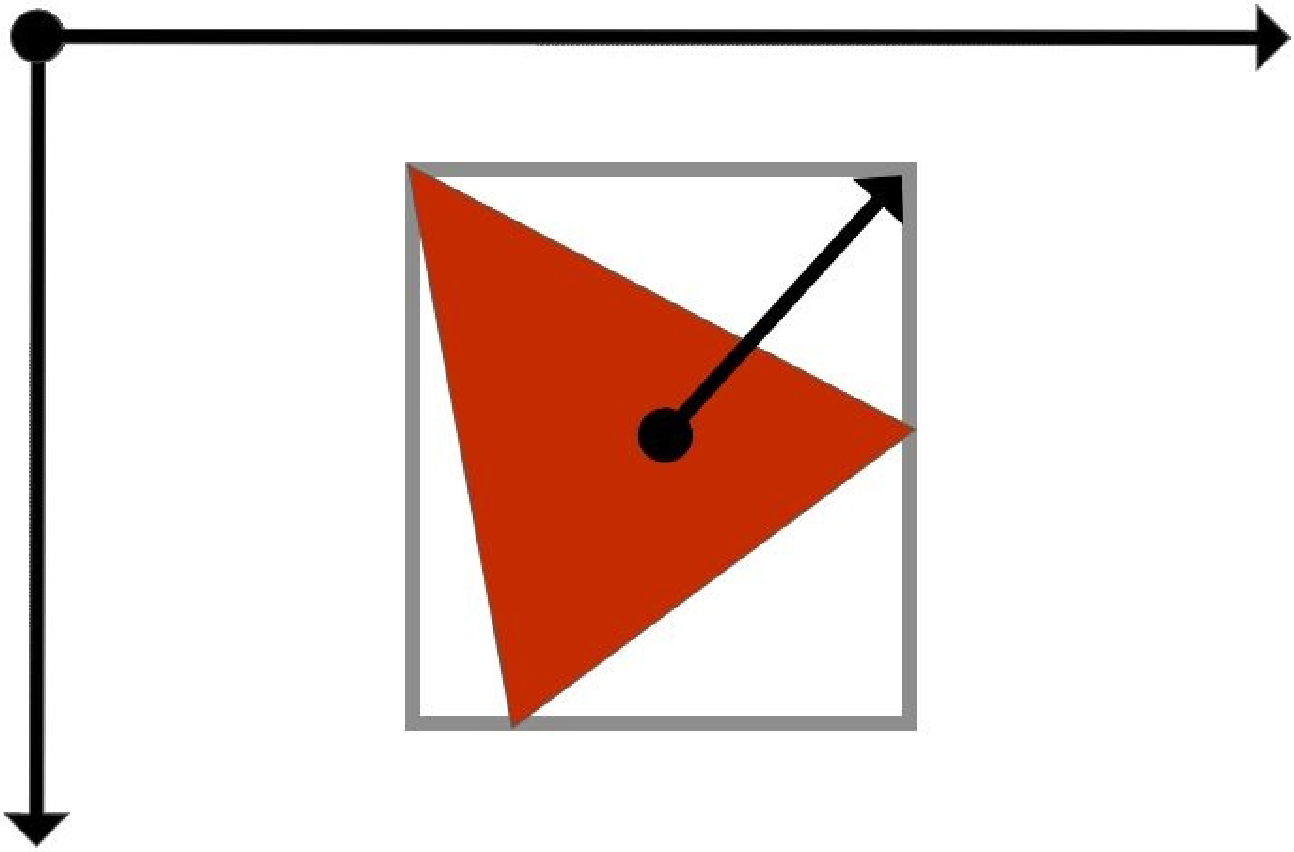
Visual representation of quadtree

Implementing A Quadtree

There are some implementations of quadtree available for Ruby - rquad, rubyquadtree and rubyquad, but it seems easy to implement, so we will build one tailored (read: closely coupled) to our game using the pseudo code from Wikipedia.

Axis Aligned Bounding Box

One of prerequisites of quadtree is Axis aligned bounding box, sometimes referred to as “AABB”. It is simply a box that surrounds the shape but has edges that are in parallel with the axes of underlying coordinate system. The advantage of this box is that it gives a rough estimate where the shape is and is very efficient when it comes to querying if a point is inside or outside it.



Axis aligned bounding box with center point and half dimension

To define axis aligned bounding box, we need it's center point and half dimension vector, which points from center point to one of the corners of the box, and two methods, one that tells if AABB contains a point, and one that tells if AABB intersects with another AABB. This is how our implementation looks like:

10-partitioning/misc/axis_aligned_bounding_box.rb

```ruby
class AxisAlignedBoundingBox
  attr_reader :center, :half_dimension
  def initialize(center, half_dimension)
    @center = center
    @half_dimension = half_dimension
    @dhx = (@half_dimension[0] - @center[0]).abs
    @dhy = (@half_dimension[1] - @center[1]).abs
  end

  def contains?(point)
    return false unless (@center[0] + @dhx) >= point[0]
    return false unless (@center[0] - @dhx) <= point[0]
    return false unless (@center[1] + @dhy) >= point[1]
    return false unless (@center[1] - @dhy) <= point[1]
    true
  end

  def intersects?(other)
    ocx, ocy = other.center
    ohx, ohy = other.half_dimension
    odhx = (ohx - ocx).abs
    return false unless (@center[0] + @dhx) >= (ocx - odhx)
    return false unless (@center[0] - @dhx) <= (ocx + odhx)
    odhy = (ohy - ocy).abs
    return false unless (@center[1] + @dhy) >= (ocy - odhy)
```

```ruby
    return false unless (@center[1] - @dhy) <= (ocy + odhy)
    true
  end

  def to_s
    "c: #{@center}, h: #{@half_dimension}"
  end
end
```

If you dig in `10-partitioning/specs`, you will find tests for this implementation too.

The math used in `AxisAlignedBoundingBox#contains?` and `AxisAlignedBoundingBox#intersects?` is fairly simple and hopefully very fast, because these methods will be called billions of times throughout the game.

QuadTree For Game Objects

To implement the glorious `QuadTree` itself, we need to initialize it with boundary, that is defined by an instance of `AxisAlignedBoundingBox` and provide methods for inserting, removing and querying the tree. Private `QuadTree#subdivide` method will be called when we try to `insert` an object into a tree that has more objects than it's `NODE_CAPACITY`.

10-partitioning/misc/quad_tree.rb

```ruby
class QuadTree
  NODE_CAPACITY = 12
  attr_accessor :ne, :nw, :se, :sw, :objects

  def initialize(boundary)
    @boundary = boundary
    @objects = []
  end

  def insert(game_object)
    return false unless @boundary.contains?(
      game_object.location)

    if @objects.size < NODE_CAPACITY
      @objects << game_object
      return true
    end

    subdivide unless @nw

    return true if @nw.insert(game_object)
    return true if @ne.insert(game_object)
    return true if @sw.insert(game_object)
    return true if @se.insert(game_object)

    # should never happen
    raise "Failed to insert #{game_object}"
  end

  def remove(game_object)
    return false unless @boundary.contains?(
      game_object.location)
    if @objects.delete(game_object)
      return true
    end
    return false unless @nw
    return true if @nw.remove(game_object)
    return true if @ne.remove(game_object)
    return true if @sw.remove(game_object)
    return true if @se.remove(game_object)
    false
  end

  def query_range(range)
    result = []
```

```ruby
    unless @boundary.intersects?(range)
      return result
    end

    @objects.each do |o|
      if range.contains?(o.location)
        result << o
      end
    end

    # Not subdivided
    return result unless @ne

    result += @nw.query_range(range)
    result += @ne.query_range(range)
    result += @sw.query_range(range)
    result += @se.query_range(range)

    result
  end

  private

  def subdivide
    cx, cy = @boundary.center
    hx, hy = @boundary.half_dimension
    hhx = (cx - hx).abs / 2.0
    hhy = (cy - hy).abs / 2.0
    @nw = QuadTree.new(
      AxisAlignedBoundingBox.new(
        [cx - hhx, cy - hhy],
        [cx, cy]))
    @ne = QuadTree.new(
      AxisAlignedBoundingBox.new(
        [cx + hhx, cy - hhy],
        [cx, cy]))
    @sw = QuadTree.new(
      AxisAlignedBoundingBox.new(
        [cx - hhx, cy + hhy],
        [cx, cy]))
    @se = QuadTree.new(
      AxisAlignedBoundingBox.new(
        [cx + hhx, cy + hhy],
        [cx, cy]))
  end
end
```

This is a vanilla quadtree that stores instances of `GameObject` and uses `GameObject#location` for indexing objects in space. It also has specs that you can find in code samples.

You can experiment with `QuadTree#NODE_CAPACITY`, but I found that values between 8 and 16 works best, so I settled with 12.

Integrating ObjectPool With QuadTree

We have implemented a `QuadTree`, but it is not yet incorporated into our game. To do that, we will hook it into `ObjectPool` and try to keep the old interface intact, so that `ObjectPool#nearby` will still work as usual, but will be able to cope with way more objects than before.

10-partitioning/entities/object_pool.rb

```ruby
class ObjectPool
  attr_accessor :map, :camera, :objects

  def size
    @objects.size
```

```
  end

  def initialize(box)
    @tree = QuadTree.new(box)
    @objects = []
  end

  def add(object)
    @objects << object
    @tree.insert(object)
  end

  def tree_remove(object)
    @tree.remove(object)
  end

  def tree_insert(object)
    @tree.insert(object)
  end

  def update_all
    @objects.map(&:update)
    @objects.reject! do |o|
      if o.removable?
        @tree.remove(o)
        true
      end
    end
  end

  def nearby(object, max_distance)
    cx, cy = object.location
    hx, hy = cx + max_distance, cy + max_distance
    # Fast, rough results
    results = @tree.query_range(
      AxisAlignedBoundingBox.new([cx, cy], [hx, hy]))
    # Sift through to select fine-grained results
    results.select do |o|
      o != object &&
        Utils.distance_between(
          o.x, o.y, object.x, object.y) <= max_distance
    end
  end

  def query_range(box)
    @tree.query_range(box)
  end
end
```

An old fashioned array of all objects is still used, because we still need to loop through everything and invoke `GameObject#update`. `ObjectPool#query_range` was introduced to quickly grab objects that have to be rendered on screen, and `ObjectPool#nearby` now queries tree and measures distances only on rough result set.

This is how we will render things from now on:

```
class PlayState < GameState
  # ...
  def draw
    cam_x = @camera.x
    cam_y = @camera.y
    off_x =  $window.width / 2 - cam_x
    off_y =  $window.height / 2 - cam_y
    viewport = @camera.viewport
    x1, x2, y1, y2 = viewport
    box = AxisAlignedBoundingBox.new(
      [x1 + (x2 - x1) / 2, y1 + (y2 - y1) / 2],
      [x1 - Map::TILE_SIZE, y1 - Map::TILE_SIZE])
    $window.translate(off_x, off_y) do
      zoom = @camera.zoom
      $window.scale(zoom, zoom, cam_x, cam_y) do
        @map.draw(viewport)
        @object_pool.query_range(box).map do |o|
```

```
          o.draw(viewport)
        end
      end
    end
    @camera.draw_crosshair
    @radar.draw
  end
  # ...
end
```

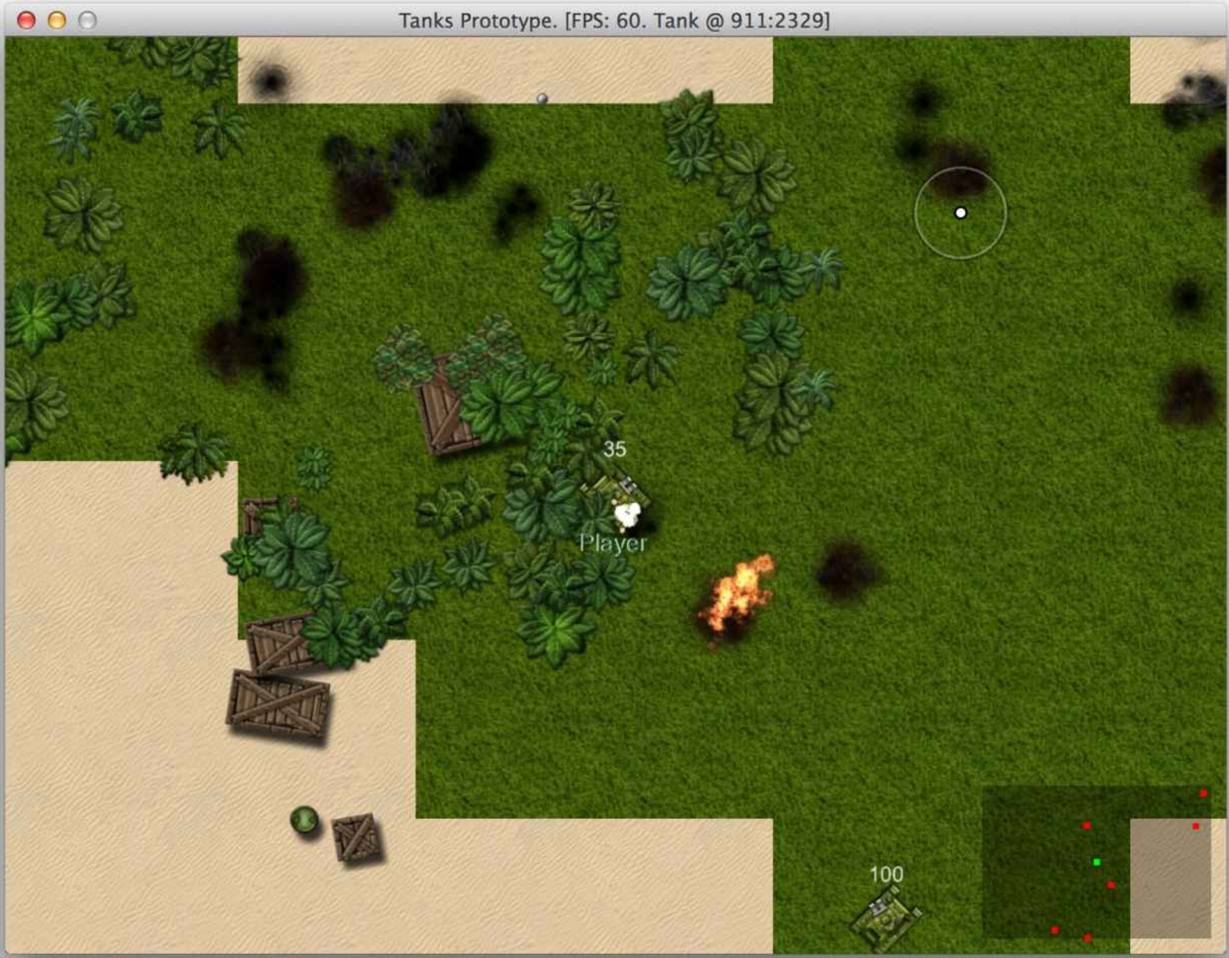
Moving Objects In QuadTree

There is one more errand we now have to take care of. Everything works fine when things are static, but when tanks and bullets move, we need to update them in our `QuadTree`. That's why `ObjectPool` has `tree_remove` and `tree_insert`, which are called from `GameObject#move`. From now on, the only way to change object's location will be by using `GameObject#move`:

```
class GameObject
  attr_reader :x, :y, :location, :components
  def initialize(object_pool, x, y)
    @x, @y = x, y
    @location = [x, y]
    @components = []
    @object_pool = object_pool
    @object_pool.add(self)
  end

  def move(new_x, new_y)
    return if new_x == @x && new_y == @y
    @object_pool.tree_remove(self)
    @x = new_x
    @y = new_y
    @location = [new_x, new_y]
    @object_pool.tree_insert(self)
  end
  # ...
end
```

At this point we have to go through all the game objects and change how they initialize their base class and update `x` and `y` coordinates, but we won't cover that here. If in doubt, refer to code at `10-partitioning`.

Finally, FPS is back to stable 60 and we can focus on gameplay again.



Spatial partitioning saves the day

Implementing Powerups

Game would become more strategic if there were ways to repair your damaged tank, boost it's speed or increase rate of fire by picking up various powerups. This should not be too difficult to implement. We will use some of [these images](#):

Powerups

For now, there will be four kinds of powerups:

1. **Repair damage.** Wrench badge will restore damaged tank's health back to 100 when picked up.
2. **Health boost.** Green +1 badge will add 25 health, up to 200 total, if you keep picking them up.
3. **Fire boost.** Double bullet badge will increase reload speed by 25%, up to 200% if you keep picking them up.
4. **Speed boost.** Airplane badge will increase movement speed by 10%, up to 150% if you keep picking them up

These powerups will be placed randomly around the map, and will automatically respawn 30 seconds after pickup.

Implementing Base Powerup

Before rushing forward to implement this, we have to do some research and think how to elegantly integrate this into the whole game. First, let's agree that `Powerup` is a `GameObject`. It will have graphics, sounds and it's coordinates. Effects can by applied by harnessing `GameObject#on_collision` - when `Tank` collides with `Powerup`, it gets it.

11-powerups/entities/powerups/powerup.rb

```
class Powerup < GameObject
  def initialize(object_pool, x, y)
    super
    PowerupGraphics.new(self, graphics)
  end

  def box
    [x - 8, y - 8,
     x + 8, y - 8,
     x + 8, y + 8,
     x - 8, y + 8]
  end

  def on_collision(object)
```

```ruby
    if pickup(object)
      PowerupSounds.play(object, object_pool.camera)
      remove
    end
  end

  def pickup(object)
    # override and implement application
  end

  def remove
    object_pool.powerup_respawn_queue.enqueue(
      respawn_delay,
      self.class, x, y)
    mark_for_removal
  end

  def respawn_delay
    30
  end
end
```

Ignore `Powerup#remove`, we will get to it when implementing `PowerupRespawnQueue`. The rest should be straightforward.

Implementing Powerup Graphics

All powerups will use the same sprite sheet, so there could be a single `PowerupGraphics` class that will be rendering given sprite type. We will use `gosu-texture-packer` gem, since sprite sheet is conveniently packed already.

11-powerups/entities/components/powerup_graphics.rb

```ruby
class PowerupGraphics < Component
  def initialize(object, type)
    super(object)
    @type = type
  end

  def draw(viewport)
    image.draw(x - 12, y - 12, 1)
    Utils.mark_corners(object.box) if $debug
  end

  private

  def image
    @image ||= images.frame("#{@type}.png")
  end

  def images
    @@images ||= Gosu::TexturePacker.load_json(
      $window, Utils.media_path('pickups.json'))
  end
end
```

Implementing Powerup Sounds

It's even simpler with sounds. All powerups will emit a mellow "bleep" when picked up, so `PowerupSounds` can be completely static, like `ExplosionSounds` or `BulletSounds`:

11-powerups/entities/components/powerup_sounds.rb

```ruby
class PowerupSounds
  class << self
    def play(object, camera)
      volume, pan = Utils.volume_and_pan(object, camera)
```

```
      sound.play(object.object_id, pan, volume)
    end

    private

    def sound
      @@sound ||= StereoSample.new(
        $window, Utils.media_path('powerup.mp3'))
    end
  end
end
```

Implementing Repair Damage Powerup

Repairing broken tank is probably the most important powerup of them all, so let's implement it first:

11-powerups/entities/powerups/repair_powerup.rb

```
class RepairPowerup < Powerup
  def pickup(object)
    if object.class == Tank
      if object.health.health < 100
        object.health.restore
      end
      true
    end
  end

  def graphics
    :repair
  end
end
```

This was incredibly simple. `Health#restore` already existed since we had to respawn our tanks. We can only hope other powerups are as simple to implement as this one.

Implementing Health Boost

Repairing damage is great, but how about boosting some extra health for upcoming battles? Health boost to the rescue:

11-powerups/entities/powerups/health_powerup.rb

```
class HealthPowerup < Powerup
  def pickup(object)
    if object.class == Tank
      object.health.increase(25)
      true
    end
  end

  def graphics
    :life_up
  end
end
```

This time we have to implement `Health#increase`, but it is pretty simple:

```
class Health < Component
  # ...
  def increase(amount)
    @health = [@health + 25, @initial_health * 2].min
    @health_updated = true
  end
  # ...
end
```

Since `Tank` has `@initial_health` equal to 100, increasing health won't go over 200, which is exactly what we want.

Implementing Fire Rate Boost

How about boosting tank's fire rate?

11-powerups/entities/powerups/fire_rate_powerup.rb

```ruby
class FireRatePowerup < Powerup
  def pickup(object)
    if object.class == Tank
      if object.fire_rate_modifier < 2
        object.fire_rate_modifier += 0.25
      end
      true
    end
  end

  def graphics
    :straight_gun
  end
end
```

We need to introduce `@fire_rate_modifier` in `Tank` class and use it when calling `Tank#can_shoot?`:

```ruby
class Tank < GameObject
  # ...
  attr_accessor :fire_rate_modifier
  # ...
  def can_shoot?
    Gosu.milliseconds - (@last_shot || 0) >
      (SHOOT_DELAY / @fire_rate_modifier)
  end
  # ...
  def reset_modifiers
    @fire_rate_modifier = 1
  end
  # ...
end
```

`Tank#reset_modifier` should be called when respawning, since we want tanks to lose their powerups when they die. It can be done in `TankHealth#after_death`:

```ruby
class TankHealth < Health
  # ...
  def after_death
    object.reset_modifiers
    # ...
  end
end
```

Implementing Tank Speed Boost

Tank speed boost is very similar to fire rate powerup:

11-powerups/entities/powerups/tank_speed_powerup.rb

```ruby
class TankSpeedPowerup < Powerup
  def pickup(object)
    if object.class == Tank
      if object.speed_modifier < 1.5
        object.speed_modifier += 0.10
      end
      true
    end
  end
```

```ruby

  def graphics
    :wingman
  end
end
```

We have to add `@speed_modifier` to `Tank` class and use it in `TankPhysics#update` when calculating movement distance.

```ruby
# 11-powerups/entities/tank.rb
class Tank < GameObject
  # ...
  attr_accessor :speed_modifier
  # ...
  def reset_modifiers
    # ...
    @speed_modifier = 1
  end
  # ...
end

# 11-powerups/entities/components/tank_physics.rb
class TankPhysics < Component
  # ...
  def update
    # ...
      new_x, new_y = x, y
      speed = apply_movement_penalty(@speed)
      shift = Utils.adjust_speed(speed) * object.speed_modifier
    # ...
  end
  # ...
end
```

`Camera#update` has also refer to `Tank#speed_modifier`, otherwise the operator will fail to catch up and camera will be lagging behind.

```ruby
class Camera
  # ...
  def update
    # ...
    shift = Utils.adjust_speed(
      @target.physics.speed).floor *
      @target.speed_modifier + 1
    # ...
  end
  # ...
end
```

Spawning Powerups On Map

Powerups are implemented, but not yet spawned. We will spawn 20 - 30 random powerups when generating the map:

```ruby
class Map
  # ...
  def initialize(object_pool)
    # ...
    generate_powerups
  end
  # ...
  def generate_powerups
    pups = 0
    target_pups = rand(20..30)
    while pups < target_pups do
      x = rand(0..MAP_WIDTH * TILE_SIZE)
      y = rand(0..MAP_HEIGHT * TILE_SIZE)
      if tile_at(x, y) != @water
        random_powerup.new(@object_pool, x, y)
        pups += 1
      end
    end
```

```
  end

  def random_powerup
    [HealthPowerup,
     RepairPowerup,
     FireRatePowerup,
     TankSpeedPowerup].sample
  end
  # ...
end
```

The code is very similar to generating boxes. It’s probably not the best way to distribute powerups on map, but it will have to do for now.

Respawning Powerups After Pickup

When we pick up a powerup, we want it to reappear in same spot 30 seconds later. A thought “we can start a new `Thread` with `sleep` and initialize the same powerup there” sounds very bad, but I had it for a few seconds. Then `PowerupRespawnQueue` was born.

First, let’s recall how `Powerup#remove` method looks like:

```
class Powerup < GameObject
  # ...
  def remove
    object_pool.powerup_respawn_queue.enqueue(
      respawn_delay,
      self.class, x, y)
    mark_for_removal
  end
  # ...
end
```

Powerup enqueues itself for respawn when picked up, providing it’s class and coordinates. `PowerupRespawnQueue` holds this data and respawns powerups at right time with help of `ObjectPool`:

11-powerups/entities/powerups/powerup_respawn_queue.rb

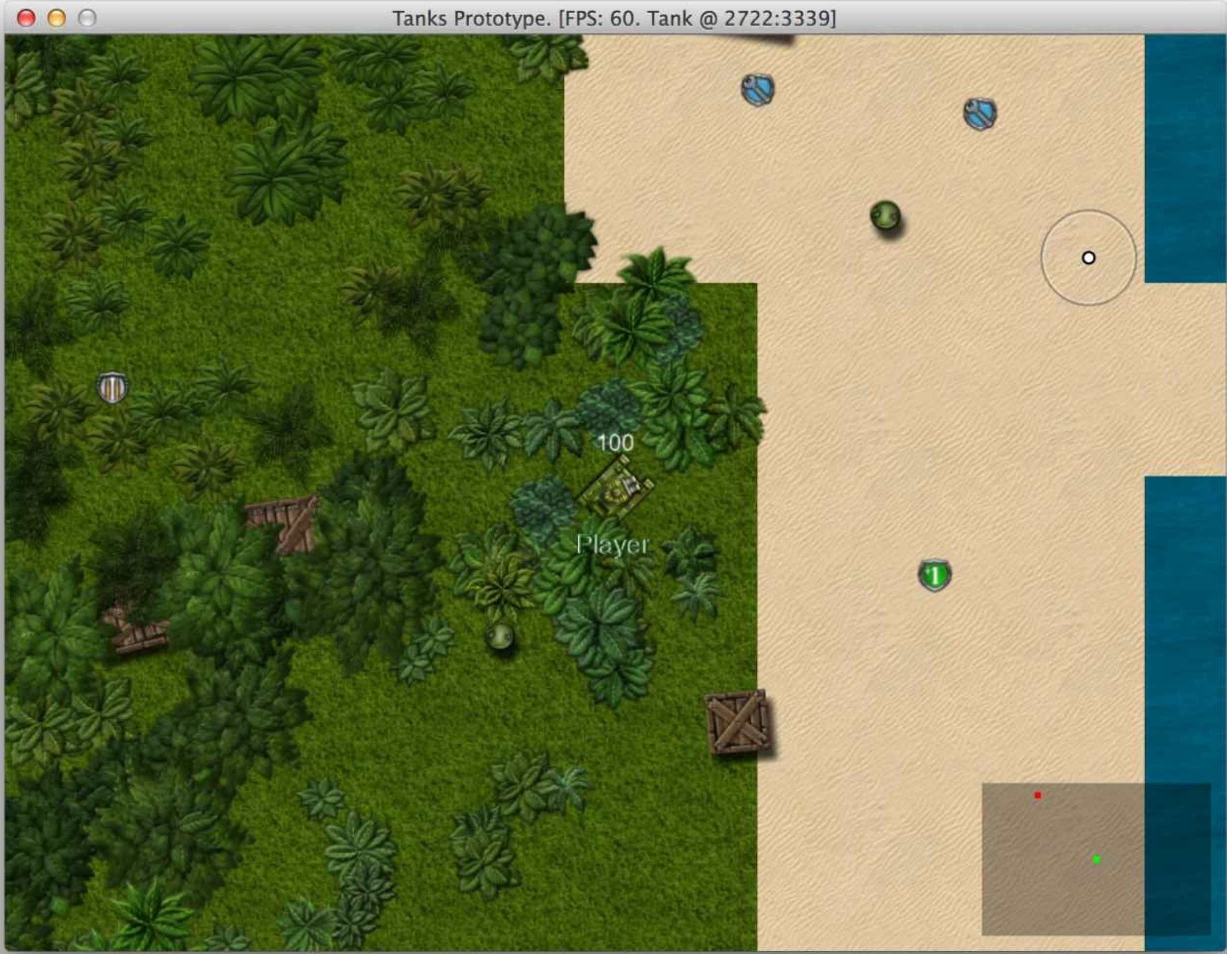
```
class PowerupRespawnQueue
  RESPAWN_DELAY = 1000
  def initialize
    @respawn_queue = {}
    @last_respawn = Gosu.milliseconds
  end

  def enqueue(delay_seconds, type, x, y)
    respawn_at = Gosu.milliseconds + delay_seconds * 1000
    @respawn_queue[respawn_at.to_i] = [type, x, y]
  end

  def respawn(object_pool)
    now = Gosu.milliseconds
    return if now - @last_respawn < RESPAWN_DELAY
    @respawn_queue.keys.each do |k|
      next if k > now # not yet
      type, x, y = @respawn_queue.delete(k)
      type.new(object_pool, x, y)
    end
    @last_respawn = now
  end
end
```

`PowerupRespawnQeueue#respawn` is called from `ObjectPool#update_all`, but is throttled to run once per second for better performance.

```
class ObjectPool
  # ...
```

```ruby
  attr_accessor :powerup_respawn_queue
  # ...
  def update_all
    # ...
    @powerup_respawn_queue.respawn(self)
  end
  # ...
end
```

This is it, the game should now contain randomly placed powerups that respawn 30 seconds after picked up. Time to enjoy the result.



Playing with powerups

We haven’t done any changes to AI though, that means enemies will only be picking those powerups by accident, so now you have a significant advantage and the game has suddenly became too easy to play. Don’t worry, we will be fixing that when overhauling the AI.

Implementing Heads Up Display

In order to know what's happening, we need some sort of HUD. We already have crosshair and radar, but they are scattered around in code. Now we want to display active powerup modifiers, so you would know what is your fire rate and speed, and if it's worth getting one more powerup before going into the next fight.

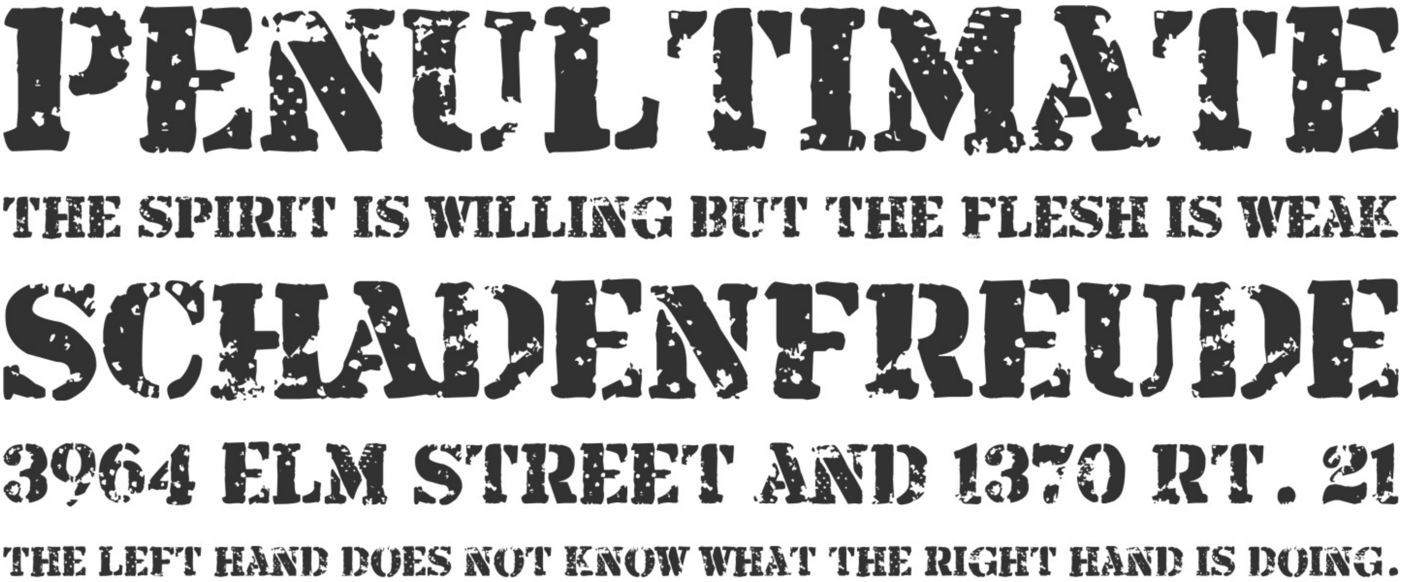
Design Considerations

While creating our HUD class, we will have to start building game stats, because we want to display number of kills our tank made. Stats topic will be covered in depth later, but for now let's assume that `@tank.input.stats.kills` gives us the kill count, which we want to draw in top-left corner of the screen, along with player health and modifier values.

HUD will also be responsible for drawing crosshair and radar.

Rendering Text With Custom Font

Previously, all text were rendered with `Gosu.default_font_name`, and we want something more fancy and more thematic, probably a dirty stencil based font like [this one]:



Armalite Rifle font

And one more fancy font will make our game title look good. Too bad we don't have a title yet, but "Tanks Prototype" writen in a thematic way still looks pretty good.

To have convenient access to these fonts, we will add a helper methods in `Utils`:

```
module Utils
  # ...
  def self.title_font
    media_path('top_secret.ttf')
  end

  def self.main_font
    media_path('armalite_rifle.ttf')
```

```ruby
  end
  # ...
end
```

Use it instead of `Gosu.default_font_name`:

```ruby
size = 20
Gosu::Image.from_text($window, "Your text", Utils.main_font, size)
```

Implementing HUD Class

After we have put everything together, we will get `HUD` class:

12-stats/entities/hud.rb

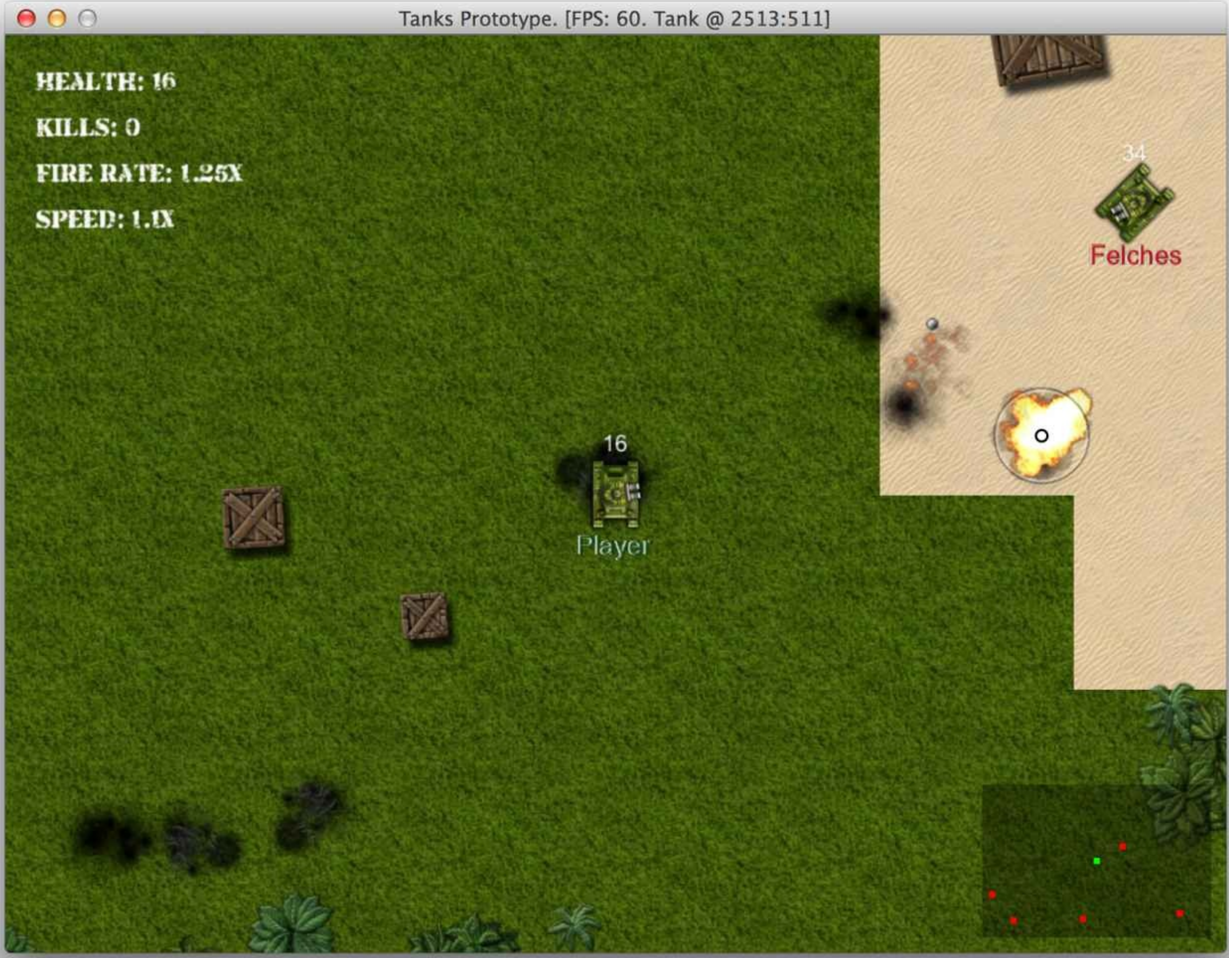
```ruby
class HUD
  attr_accessor :active
  def initialize(object_pool, tank)
    @object_pool = object_pool
    @tank = tank
    @radar = Radar.new(@object_pool, tank)
  end

  def player=(tank)
    @tank = tank
    @radar.target = tank
  end

  def update
    @radar.update
  end

  def health_image
    if @health.nil? || @tank.health.health != @health
      @health = @tank.health.health
      @health_image = Gosu::Image.from_text(
        $window, "Health: #{@health}", Utils.main_font, 20)
    end
    @health_image
  end

  def stats_image
    stats = @tank.input.stats
    if @stats_image.nil? || stats.changed_at <= Gosu.milliseconds
      @stats_image = Gosu::Image.from_text(
        $window, "Kills: #{stats.kills}", Utils.main_font, 20)
    end
    @stats_image
  end

  def fire_rate_image
    if @tank.fire_rate_modifier > 1
      if @fire_rate != @tank.fire_rate_modifier
        @fire_rate = @tank.fire_rate_modifier
        @fire_rate_image = Gosu::Image.from_text(
          $window, "Fire rate: #{@fire_rate.round(2)}X",
          Utils.main_font, 20)
      end
    else
      @fire_rate_image = nil
    end
    @fire_rate_image
  end

  def speed_image
    if @tank.speed_modifier > 1
      if @speed != @tank.speed_modifier
        @speed = @tank.speed_modifier
        @speed_image = Gosu::Image.from_text(
          $window, "Speed: #{@speed.round(2)}X",
          Utils.main_font, 20)
      end
    else
      @speed_image = nil
```

```
    end
    @speed_image
  end

  def draw
    if @active
      @object_pool.camera.draw_crosshair
    end
    @radar.draw
    offset = 20
    health_image.draw(20, offset, 1000)
    stats_image.draw(20, offset += 30, 1000)
    if fire_rate_image
      fire_rate_image.draw(20, offset += 30, 1000)
    end
    if speed_image
      speed_image.draw(20, offset += 30, 1000)
    end
  end
end
```

To use it, we need to hook into `PlayState`:

```
class PlayState < GameState
  # ...
  def initialize
    # ...
    @hud = HUD.new(@object_pool, @tank)
  end

  def update
    # ...
    @hud.update
  end

  def draw
    # ...
    @hud.draw
  end
  # ...
end
```

Assuming you have mocked `@tank.input.stats.kills` in `HUD`, you should get a neat view showing interesting things in top-left corner of the screen:



Shiny new HUD

Implementing Game Statistics

Games like one we are building are all about competition, and you cannot compete if you don't know the score. Let us introduce a class that will be responsible for keeping tabs on various statistics of every tank.

12-stats/misc/stats.rb

```ruby
class Stats
  attr_reader :name, :kills, :deaths, :shots, :changed_at
  def initialize(name)
    @name = name
    @kills = @deaths = @shots = @damage = @damage_dealt = 0
    changed
  end

  def add_kill(amount = 1)
    @kills += amount
    changed
  end

  def add_death
    @deaths += 1
    changed
  end

  def add_shot
    @shots += 1
    changed
  end

  def add_damage(amount)
    @damage += amount
    changed
  end

  def damage
    @damage.round
  end

  def add_damage_dealt(amount)
    @damage_dealt += amount
    changed
  end

  def damage_dealt
    @damage_dealt.round
  end

  def to_s
    "[kills: #{@kills}, " \
      "deaths: #{@deaths}, " \
      "shots: #{@shots}, " \
      "damage: #{damage}, " \
      "damage_dealt: #{damage_dealt}]"
  end

  private

  def changed
    @changed_at = Gosu.milliseconds
  end
end
```

While building the HUD, we established that `Stats` should belong to `Tank#input`, because it defines who is controlling the tank. So, every instance of `PlayerInput` and `AiInput` has to have it's own `Stats`:

```
# 12-stats/entities/components/player_input.rb
class PlayerInput < Component
  # ...
  attr_reader :stats

  def initialize(name, camera, object_pool)
    # ...
    @stats = Stats.new(name)
  end
  # ...
  def on_damage(amount)
    @stats.add_damage(amount)
  end
  # ...
end
```

```
# 12-stats/entities/components/ai_input.rb
class AiInput < Component
  # ...
  attr_reader :stats

  def initialize(name, object_pool)
    # ...
    @stats = Stats.new(name)
  end

  def on_damage(amount)
    # ...
    @stats.add_damage(amount)
  end
end
```

That itch to extract a base class from `PlayerInput` and `AiInput` is getting stronger, but we will have to resist the urge, for now.

Tracking Kills, Deaths and Damage

To begin tracking kills, we need to know whom does every bullet belong to. `Bullet` already has `source` attribute, which contains the tank that fired it, there will be no trouble to find out who was the shooter when bullet gets a direct hit. But how about explosions? Bullets that hit the ground nearby a tank deals indirect damage from the explosion.

Solution is simple, we need to pass the `source` of the `Bullet` to the `Explosion` when it's being initialized.

```
class Bullet < GameObject
  # ...
  def explode
    Explosion.new(object_pool, @x, @y, @source)
    # ...
  end
  # ...
end
```

Making Damage Personal

Now that we have the source of every `Bullet` and `Explosion` they trigger, we can start passing the cause of damage to `Health#inflict_damage` and incrementing the appropriate stats.

```
# 12-stats/entities/components/health.rb
class Health < Component
```

```ruby
  # ...
  def inflict_damage(amount, cause)
    if @health > 0
      @health_updated = true
      if object.respond_to?(:input)
        object.input.stats.add_damage(amount)
        # Don't count damage to trees and boxes
        if cause.respond_to?(:input) && cause != object
          cause.input.stats.add_damage_dealt(amount)
        end
      end
      @health = [@health - amount.to_i, 0].max
      after_death(cause) if dead?
    end
  end
  # ...
end
```

```ruby
# 12-stats/entities/components/tank_health.rb
class TankHealth < Health
  # ...
  def after_death(cause)
    # ...
    object.input.stats.add_death
    kill = object != cause ? 1 : -1
    cause.input.stats.add_kill(kill)
    # ...
  end
# ...
end
```

Tracking Damage From Chain Reactions

There is one more way to cause damage. When you shoot a tree, box or barrel, it explodes, probably triggering a chain reaction of explosions around it. If those explosions kill somebody, it would only be fair to account that kill for the tank that triggered this chain reaction.

To solve this, simply pass the cause of death to the `Explosion` that gets triggered afterwards.

```ruby
# 12-stats/entities/components/health.rb
class Health < Component
  # ...
  def after_death(cause)
    if @explodes
      Thread.new do
        # ...
        Explosion.new(@object_pool, x, y, cause)
        # ...
      end
      # ...
    end
  end
end
```

```ruby
# 12-stats/entities/components/tank_health.rb
class TankHealth < Health
  # ...
  def after_death(cause)
    # ...
    Thread.new do
      # ...
      Explosion.new(@object_pool, x, y, cause)
    end
  end
end
```

Now every bit of damage gets accounted for.

Displaying Game Score

Having all the data is useless unless we display it somehow. For this, let's rethink our game states. Now we have `MenuState` and `PlayState`. Both of them can switch one into another. What if we introduced a `PauseState`, which would freeze the game and display the list of all tanks along with their kills. Then `MenuState` would switch to `PlayState`, and from `PlayState` you would be able to get to `PauseState`.

Let's begin by implementing `ScoreDisplay`, that would print a sorted list of tank kills along with their names.

12-stats/entities/score_display.rb

```ruby
class ScoreDisplay
  def initialize(object_pool)
    tanks = object_pool.objects.select do |o|
      o.class == Tank
    end
    stats = tanks.map(&:input).map(&:stats)
    stats.sort! do |stat1, stat2|
      stat2.kills <=> stat1.kills
    end
    create_stats_image(stats)
  end

  def create_stats_image(stats)
    text = stats.map do |stat|
      "#{stat.kills}: #{stat.name} "
    end.join("\n")
    @stats_image = Gosu::Image.from_text(
      $window, text, Utils.main_font, 30)
  end

  def draw
    @stats_image.draw(
      $window.width / 2 - @stats_image.width / 2,
      $window.height / 4 + 30,
      1000)
  end
end
```

We will have to initialize `ScoreDisplay` every time when we want to show the updated score. Time to create the `PauseState` that would show the score.

12-stats/game_states/pause_state.rb

```ruby
require 'singleton'
class PauseState < GameState
  include Singleton
  attr_accessor :play_state

  def initialize
    @message = Gosu::Image.from_text(
      $window, "Game Paused",
      Utils.title_font, 60)
  end

  def enter
    music.play(true)
    music.volume = 1
    @score_display = ScoreDisplay.new(@play_state.object_pool)
    @mouse_coords = [$window.mouse_x, $window.mouse_y]
  end

  def leave
    music.volume = 0
    music.stop
    $window.mouse_x, $window.mouse_y = @mouse_coords
  end
```

```

  def music
    @@music ||= Gosu::Song.new(
      $window, Utils.media_path('menu_music.mp3'))
  end

  def draw
    @play_state.draw
    @message.draw(
      $window.width / 2 - @message.width / 2,
      $window.height / 4 - @message.height,
      1000)
    @score_display.draw
  end

  def button_down(id)
    $window.close if id == Gosu::KbQ
    if id == Gosu::KbC && @play_state
      GameState.switch(@play_state)
    end
    if id == Gosu::KbEscape
      GameState.switch(@play_state)
    end
  end
end
```

You will notice that `PauseState` invokes `PlayState#draw`, but without `PlayState#update` this will be a still image. We make sure we hide the crosshair and restore previous mouse location when resuming play state. That way player would not be able to cheat by pausing the game, targeting the tank while nothing moves and then unpausing ready to deal damage. Our `HUD` had `attr_accessor :active` exactly for this reason, but we need to switch it on and off in `PlayState#enter` and `PlayState#leave`.

```
class PlayState < GameState
  # ...
  def button_down(id)
    # ...
    if id == Gosu::KbEscape
      pause = PauseState.instance
      pause.play_state = self
      GameState.switch(pause)
    end
    # ...
  end
  # ...
  def leave
    StereoSample.stop_all
    @hud.active = false
  end

  def enter
    @hud.active = true
  end
  # ...
end
```

Time for a test drive.

Pausing the game to see the score

For now, scoring most kills is relatively simple. This should change when we will tell enemy AI to collect powerups when appropriate.

Building Advanced AI

The AI we have right now can kick some ass, but it is too dumb for any seasoned gamer to compete with. This is the list of current flaws:

1. It does not navigate well, gets stuck among trees or somewhere near water.
2. It is not aware of powerups.
3. It could do better job at shooting.
4. It's field of vision is too small, compared to player's, who is equipped with radar.

We will tackle these issues in current chapter.

Improving Tank Navigation

Tanks shouldn't behave like Roombas, randomly driving around and bumping into things. They could be navigating like this:

1. Consult with current AI state and find or update destination point.
2. If destination has changed, calculate shortest path to destination.
3. Move along the calculated path.
4. Repeat.

If this looks easy, let me assure you, it would probably require rewriting the majority of `AI` and `Map` code we have at this point, and it is pretty tricky to implement with procedurally generated maps, because normally you would use a map editor to set up waypoints, navigation mesh or other hints for AI so it doesn't get stuck. Sometimes it is better to have something working imperfectly over a perfect solution that never happens, thus we will use simple things that will make as much impact as possible without rewriting half of the code.

Generating Friendlier Maps

One of main reasons why tanks get stuck is bad placement of spawn points. They don't take trees and boxes into account, so enemy tank can spawn in the middle of a forest, with no chance of getting out without blowing things up. A simple fix would be to consult with `ObjectPool` before placing a spawn point only where there are no other game objects around in, say, 150 pixel radius:

```
class Map
  # ...
  def find_spawn_point
    while true
      x = rand(0..MAP_WIDTH * TILE_SIZE)
      y = rand(0..MAP_HEIGHT * TILE_SIZE)
      if can_move_to?(x, y) &&
          @object_pool.nearby_point(x, y, 150).empty?
        return [x, y]
      end
    end
```

```
  end
  # ...
end
```

How about powerups? They can also spawn in the middle of a forest, and while tanks are not seeking them yet, we will be implementing this behavior, and leading tanks into wilderness of trees is not the best idea ever. Let's fix it too:

```
class Map
  # ...
  def generate_powerups
    pups = 0
    target_pups = rand(20..30)
    while pups < target_pups do
      x = rand(0..MAP_WIDTH * TILE_SIZE)
      y = rand(0..MAP_HEIGHT * TILE_SIZE)
      if tile_at(x, y) != @water &&
          @object_pool.nearby_point(x, y, 150).empty?
        random_powerup.new(@object_pool, x, y)
        pups += 1
      end
    end
  end
  # ...
end
```

We could also reduce tree count, but that would make the map look worse, so we are going to keep this in our pocket as a mean of last resort.

Implementing Demo State To Observe AI

Probably the best way to figure out if our AI is any good is to target one of AI tanks with our game camera and see how it plays. It will give us a great visual testing tool that will allow tweaking AI settings and seeing if they perform better or worse. For that we will introduce `DemoState` where only AI tanks will be present in the map, and we will be able to switch camera from one tank to another.

`DemoState` is very similar to `PlayState`, the main difference is that there is no player. We will extract `create_tanks` method that will be overridden in `DemoState`.

```
class PlayState < GameState
  attr_accessor :update_interval, :object_pool, :tank

  def initialize
    # ...
    @camera = Camera.new
    @object_pool.camera = @camera
    create_tanks(4)
  end
  # ...
  private

  def create_tanks(amount)
    @map.spawn_points(amount * 3)
    @tank = Tank.new(@object_pool,
      PlayerInput.new('Player', @camera, @object_pool))
    amount.times do |i|
      Tank.new(@object_pool, AiInput.new(
        @names.random, @object_pool))
    end
    @camera.target = @tank
    @hud = HUD.new(@object_pool, @tank)
  end
  # ...
end
```

We will also want to display a smaller version of score in top-right corner of the screen, so let’s add some adjustments to `ScoreDisplay`:

```ruby
class ScoreDisplay
  def initialize(object_pool, font_size=30)
    @font_size = font_size
    # ...
  end

  def create_stats_image(stats)
    # ...
    @stats_image = Gosu::Image.from_text(
      $window, text, Utils.main_font, @font_size)
  end
  # ...
  def draw_top_right
    @stats_image.draw(
      $window.width - @stats_image.width - 20,
      20,
      1000)
  end
end
```

And here is the extended `DemoState`:

13-advanced-ai/game_states/demo_state.rb

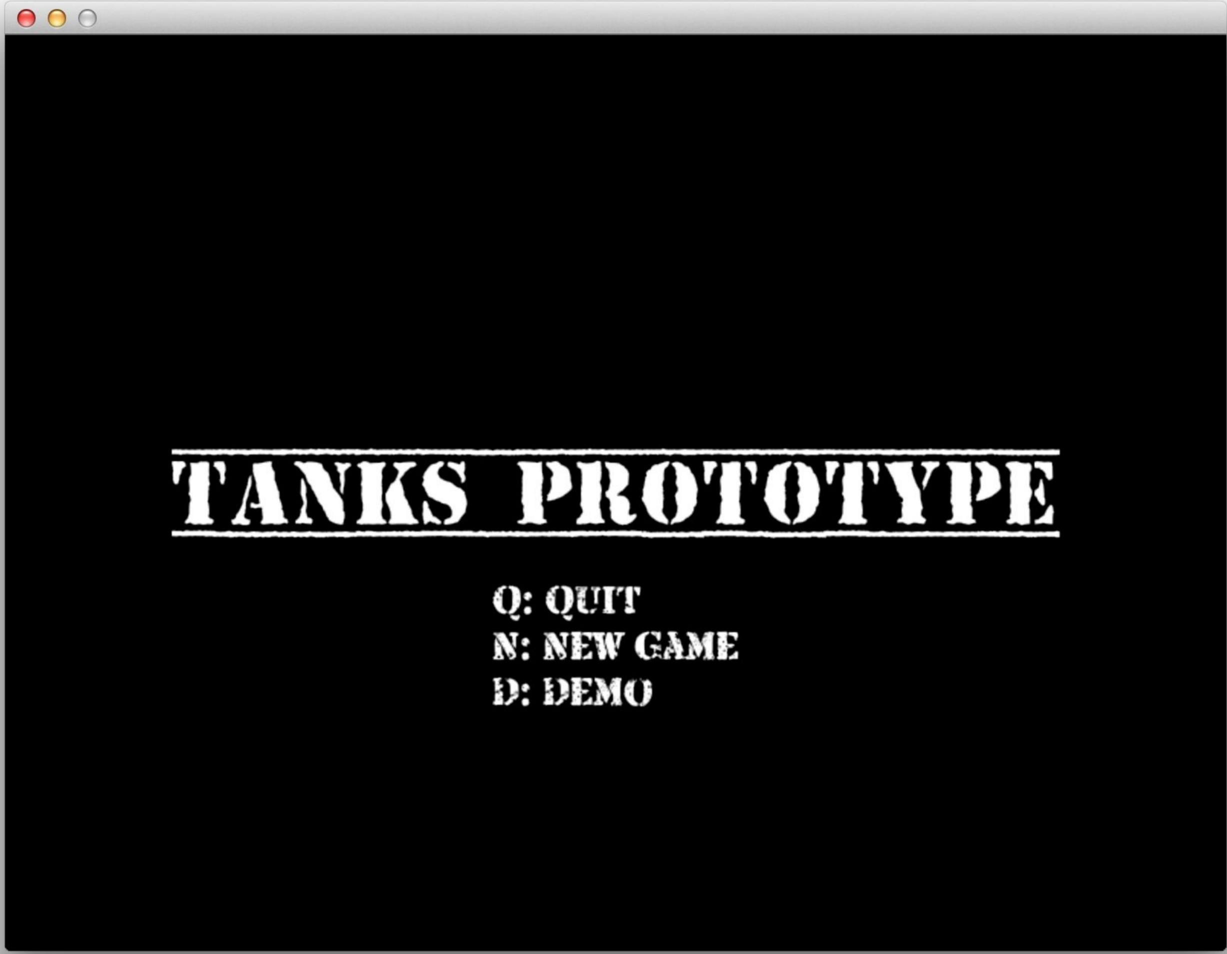
```ruby
class DemoState < PlayState
  attr_accessor :tank

  def enter
    # Prevent reactivating HUD
  end

  def update
    super
    @score_display = ScoreDisplay.new(
      object_pool, 20)
  end

  def draw
    super
    @score_display.draw_top_right
  end

  def button_down(id)
    super
    if id == Gosu::KbSpace
      target_tank = @tanks.reject do |t|
        t == @camera.target
      end.sample
      switch_to_tank(target_tank)
    end
  end

  private

  def create_tanks(amount)
    @map.spawn_points(amount * 3)
    @tanks = []
    amount.times do |i|
      @tanks << Tank.new(@object_pool, AiInput.new(
        @names.random, @object_pool))
    end
    target_tank = @tanks.sample
    @hud = HUD.new(@object_pool, target_tank)
    @hud.active = false
    switch_to_tank(target_tank)
  end

  def switch_to_tank(tank)
    @camera.target = tank
    @hud.player = tank
    self.tank = tank
```

```
  end
end
```

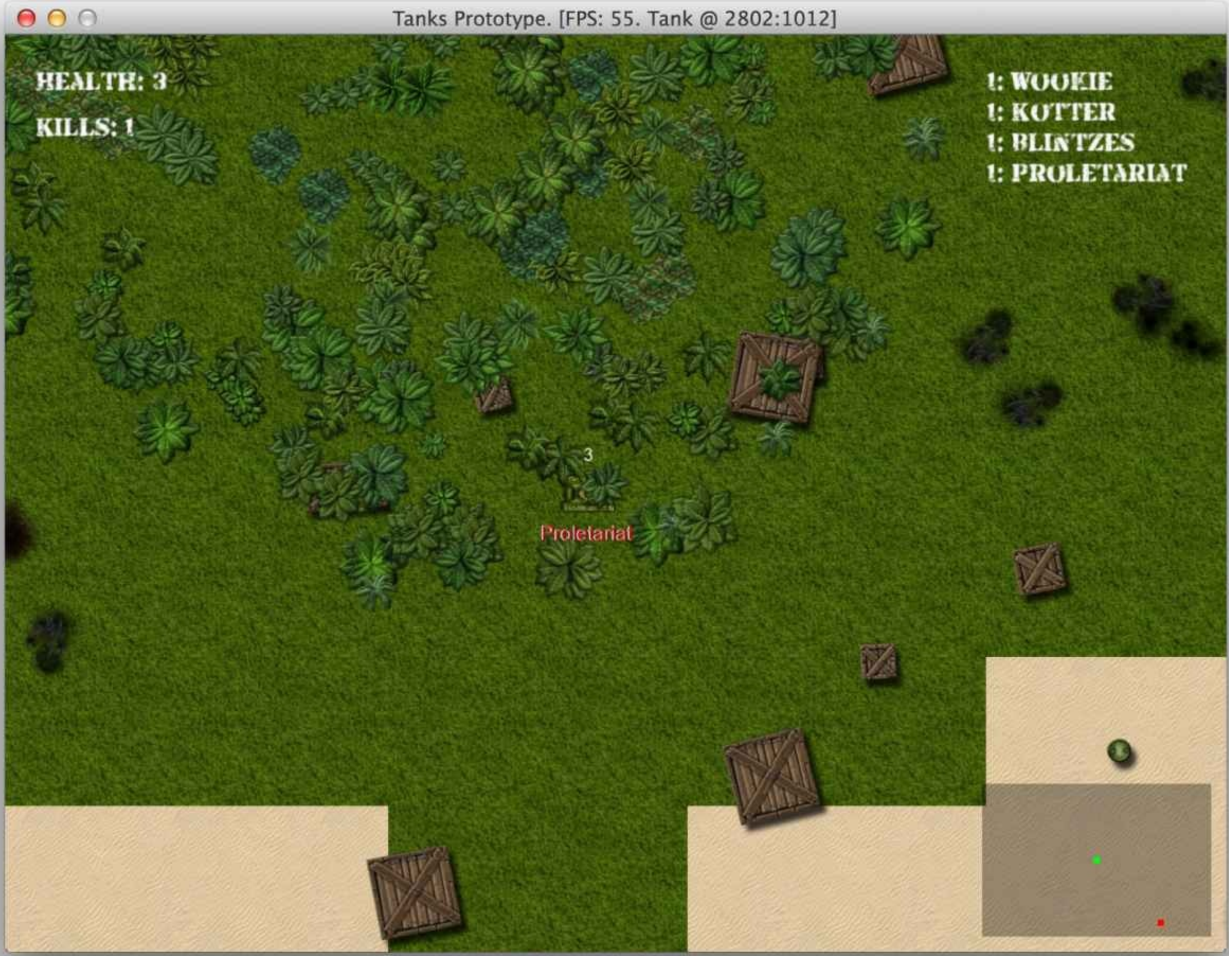
To have a possibility to enter `DemoState`, we need to change `MenuState` a little:

```
class MenuState < GameState
  # ...
  def update
    text = "Q: Quit\nN: New Game\nD: Demo"
    # ...
  end
  # ...
  def button_down(id)
    # ...
    if id == Gosu::KbD
      @play_state = DemoState.new
      GameState.switch(@play_state)
    end
  end
end
```

Now, main menu has the option to enter demo state:



Overhauled main menu



Observing AI in demo state

Visual AI Debugging

After watching AI behavior in demo mode for a while, I was terrified. When playing game normally, you usually see tanks in "fighting" state, which works pretty well, but when tanks go roaming, it's a complete disaster. They get stuck easily, they don't go too far from the original location, they wait too much.

Some things could be improved just by changing `wait_time`, `turn_time` and `drive_time` to different values, but we certainly have to do bigger changes than that.

On the other hand, "observe AI in action, tweak, repeat" cycle proved to be very effective, I will definitely use this technique in all my future games.

To make visual debugging easier, build yourself some tooling. One way to do it is to have global `$debug` variable which you can toggle by pressing some button:

```
class PlayState < GameState
  # ...
  def button_down(id)
    # ...
    if id == Gosu::KbF1
      $debug = !$debug
    end
    # ...
  end
```

```
  # ...
end
```

Then add extra drawing instructions to your objects and their components. For example, this will make `Tank` display it's current `TankMotionState` implementation class beneath it:

```
class TankMotionFSM
  # ...
  def set_state(state)
    # ...
    if $debug
      @image = Gosu::Image.from_text(
          $window, state.class.to_s,
          Gosu.default_font_name, 18)
    end
  end
  # ...
  def draw(viewport)
    if $debug
      @image && @image.draw(
        @object.x - @image.width / 2,
        @object.y + @object.graphics.height / 2 -
        @image.height, 100)
    end
  end
  # ...
end
```

To mark tank's desired gun angle as blue line and actual gun angle as red line, you can do this:

```
class AiGun
  # ...
  def draw(viewport)
    if $debug
      color = Gosu::Color::BLUE
      x, y = @object.x, @object.y
      t_x, t_y = Utils.point_at_distance(x, y, @desired_gun_angle,
                                         BulletPhysics::MAX_DIST)
      $window.draw_line(x, y, color, t_x, t_y, color, 1001)
      color = Gosu::Color::RED
      t_x, t_y = Utils.point_at_distance(x, y, @object.gun_angle,
                                         BulletPhysics::MAX_DIST)
      $window.draw_line(x, y, color, t_x, t_y, color, 1000)
    end
  end
  # ...
end
```

Finally, you can automatically mark collision box corners on your graphics components. Let's take `BoxGraphics` for example:

```
# 13-advanced-ai/misc/utils.rb
module Utils
  # ...
  def self.mark_corners(box)
    i = 0
    box.each_slice(2) do |x, y|
      color = DEBUG_COLORS[i]
      $window.draw_triangle(
        x - 3, y - 3, color,
        x,     y,     color,
        x + 3, y - 3, color,
        100)
      i = (i + 1) % 4
    end
  end
  # ...
end

# 13-advanced-ai/entities/components/box_graphics.rb
class BoxGraphics < Component
  # ..
  def draw(viewport)
```

```ruby
    @box.draw_rot(x, y, 0, object.angle)
    Utils.mark_corners(object.box) if $debug
  end
  # ...
end
```

As a developer, you can make yourself see nearly everything you want, make use of it.

Visual debugging of AI behavior

Although it hurts the framerate a little, it is very useful when building not only AI, but the rest of the game too. Using this visual debugging together with Demo mode, you can tweak all the AI values to make it shoot more often, fight better, and be more agile. We won't go through this minor tuning, but you can find the changes by [viewing changes introduced in `13-advanced-ai`].

Making AI Collect Powerups

To even out the odds, we have to make AI seek powerups when they are required. The logic behind it can be implemented using a couple of simple steps:

1. AI would know what powerups are currently needed. This may vary from state to state, i.e. speed and fire rate powerups are nice to have when roaming, but not that important when fleeing after taking heavy damage. And we don't want AI to waste time and collect speed powerups when speed modifier is already maxed out.
2. `AiVision` would return closest visible powerup, filtered by acceptable powerup types.
3. Some `TankMotionState` implementation would adjust tank direction towards closest visible powerup in `change_direction` method.

Finding Powerups In Sight

To implement changes in `AiVision`, we will introduce `closest_powerup` method. It will query objects in sight and filter them out by their class and distance.

```
class AiVision
  # ...
  POWERUP_CACHE_TIMEOUT = 50
  # ...
  def closest_powerup(*suitable)
    now = Gosu.milliseconds
    @closest_powerup = nil
    if now - (@powerup_cache_updated_at ||= 0) > POWERUP_CACHE_TIMEOUT
      @closest_powerup = nil
      @powerup_cache_updated_at = now
    end
    @closest_powerup ||= find_closest_powerup(*suitable)
  end

  private

  def find_closest_powerup(*suitable)
    if suitable.empty?
      suitable = [FireRatePowerup,
                  HealthPowerup,
                  RepairPowerup,
                  TankSpeedPowerup]
    end
    @in_sight.select do |o|
      suitable.include?(o.class)
    end.sort do |a, b|
      x, y = @viewer.x, @viewer.y
      d1 = Utils.distance_between(x, y, a.x, a.y)
      d2 = Utils.distance_between(x, y, b.x, b.y)
      d1 <=> d2
    end.first
  end
  # ...
end
```

It is very similar to `AiVision#closest_tank`, and parts should probably be extracted to keep the code dry, but we will not bother.

Seeking Powerups While Roaming

Roaming is when most picking should happen, because Tank sees no enemies in sight and needs to prepare for upcoming battles. Let's see how can we implement this behavior while leveraging the newly made `AiVision#closest_powerup`:

```
class TankRoamingState < TankMotionState
  # ...
  def required_powerups
    required = []
    health = @object.health.health
    if @object.fire_rate_modifier < 2 && health > 50
      required << FireRatePowerup
    end
    if @object.speed_modifier < 1.5 && health > 50
      required << TankSpeedPowerup
    end
    if health < 100
      required << RepairPowerup
    end
    if health < 190
      required << HealthPowerup
    end
    required
  end

  def change_direction
    closest_powerup = @vision.closest_powerup(
      *required_powerups)
    if closest_powerup
      @seeking_powerup = true
```

```
      angle = Utils.angle_between(
        @object.x, @object.y,
        closest_powerup.x, closest_powerup.y)
      @object.physics.change_direction(
        angle - angle % 45)
    else
      @seeking_powerup = false
      # ... choose random direction
    end
    @changed_direction_at = Gosu.milliseconds
    @will_keep_direction_for = turn_time
  end
  # ...
  def turn_time
    if @seeking_powerup
      rand(100..300)
    else
      rand(1000..3000)
    end
  end
end
```

It is simple as that, and our AI tanks are now getting buffed on their spare time.

Seeking Health Powerups After Heavy Damage

To seek health when damaged, we need to change `TankFleeingState#change_direction`:

```
class TankFleeingState < TankMotionState
  # ...
  def change_direction
    closest_powerup = @vision.closest_powerup(
      RepairPowerup, HealthPowerup)
    if closest_powerup
      angle = Utils.angle_between(
        @object.x, @object.y,
        closest_powerup.x, closest_powerup.y)
      @object.physics.change_direction(
        angle - angle % 45)
    else
      # ... reverse from enemy
    end
    @changed_direction_at = Gosu.milliseconds
    @will_keep_direction_for = turn_time
  end
  # ...
end
```

This small change tells AI to pick up health while fleeing. The interesting part is that when tank picks up `RepairPowerup`, it's health gets fully restored and AI should switch back to `TankFightingState`. This simple thing is a major improvement in AI behavior.

Evading Collisions And Getting Unstuck

While observing AI navigation, it was noticeable that tanks often got stuck, even in simple situations, like driving into a tree and hitting it repeatedly for a dozen of seconds. To reduce the number of such occasions, we will introduce `TankNavigatingState`, which would help avoid collisions, and `TankStuckState`, which would be responsible for driving out of dead ends as quickly as possible.

To implement these states, we need to have a way to tell if tank can go forward and a way of getting a direction which is not blocked by other objects. Let's add a couple of methods to `AiVision`:

```
class AiVision
  # ...
  def can_go_forward?
```

```
    in_front = Utils.point_at_distance(
      *@viewer.location, @viewer.direction, 40)
    @object_pool.map.can_move_to?(*in_front) &&
      @object_pool.nearby_point(*in_front, 40, @viewer)
        .reject { |o| o.is_a? Powerup }.empty?
  end

  def closest_free_path(away_from = nil)
    paths = []
    5.times do |i|
      if paths.any?
        return farthest_from(paths, away_from)
      end
      radius = 55 - i * 5
      range_x = range_y = [-radius, 0, radius]
      range_x.shuffle.each do |x|
        range_y.shuffle.each do |y|
          x = @viewer.x + x
          y = @viewer.y + y
          if @object_pool.map.can_move_to?(x, y) &&
              @object_pool.nearby_point(x, y, radius, @viewer)
                .reject { |o| o.is_a? Powerup }.empty?
            if away_from
              paths << [x, y]
            else
              return [x, y]
            end
          end
        end
      end
    end
    false
  end

  alias :closest_free_path_away_from :closest_free_path
  # ...
  private

  def farthest_from(paths, away_from)
    paths.sort do |p1, p2|
      Utils.distance_between(*p1, *away_from) <=>
        Utils.distance_between(*p2, *away_from)
    end.first
  end
  # ...
end
```

`AiVision#can_go_forward?` tells if tank can move ahead, and `AiVision#closest_free_path` finds a point where tank can move without obstacles. You can also call `AiVision#closest_free_path_away_from` and provide coordinates you are trying to get away from.

We will use `closest_free_path` methods in newly implemented tank motion states, and `can_go_forward?` in `TankMotionFSM`, to make a decision when to jump into navigating or stuck state.

Those new states are nothing fancy:

13-advanced-ai/entities/components/ai/tank_navigating_state.rb

```
class TankNavigatingState < TankMotionState
  def initialize(object, vision)
    @object = object
    @vision = vision
  end

  def update
    change_direction if should_change_direction?
    drive
  end

```

```ruby
  def change_direction
    closest_free_path = @vision.closest_free_path
    if closest_free_path
      @object.physics.change_direction(
        Utils.angle_between(
          @object.x, @object.y, *closest_free_path))
    end
    @changed_direction_at = Gosu.milliseconds
    @will_keep_direction_for = turn_time
  end

  def wait_time
    rand(10..100)
  end

  def drive_time
    rand(1000..2000)
  end

  def turn_time
    rand(300..1000)
  end
end
```

`TankNavigatingState` simply chooses a random free path, changes direction to it and keeps driving.

13-advanced-ai/entities/components/ai/tank_stuck_state.rb

```ruby
class TankNavigatingState < TankMotionState
  def initialize(object, vision)
    @object = object
    @vision = vision
  end

  def update
    change_direction if should_change_direction?
    drive
  end

  def change_direction
    closest_free_path = @vision.closest_free_path
    if closest_free_path
      @object.physics.change_direction(
        Utils.angle_between(
          @object.x, @object.y, *closest_free_path))
    end
    @changed_direction_at = Gosu.milliseconds
    @will_keep_direction_for = turn_time
  end

  def wait_time
    rand(10..100)
  end

  def drive_time
    rand(1000..2000)
  end

  def turn_time
    rand(300..1000)
  end
end
```

`TankStuckState` is nearly the same, but it keeps driving away from `@stuck_at` point, which is set by `TankMotionFSM` upon transition to this state.

```ruby
class TankMotionFSM
  STATE_CHANGE_DELAY = 500
  LOCATION_CHECK_DELAY = 5000

  def initialize(object, vision, gun)
```

```ruby
    # ...
    @stuck_state = TankStuckState.new(object, vision, gun)
    @navigating_state = TankNavigatingState.new(object, vision)
    set_state(@roaming_state)
  end
  # ...
  def choose_state
    unless @vision.can_go_forward?
      unless @current_state == @stuck_state
        set_state(@navigating_state)
      end
    end
    # Keep unstucking itself for a while
    change_delay = STATE_CHANGE_DELAY
    if @current_state == @stuck_state
      change_delay *= 5
    end
    now = Gosu.milliseconds
    return unless now - @last_state_change > change_delay
    if @last_location_update.nil?
      @last_location_update = now
      @last_location = @object.location
    end
    if now - @last_location_update > LOCATION_CHECK_DELAY
      puts "checkin location"
      unless @last_location.nil? || @current_state.waiting?
        if Utils.distance_between(*@last_location, *@object.location) < 20
          set_state(@stuck_state)
          @stuck_state.stuck_at = @object.location
          return
        end
      end
      @last_location_update = now
      @last_location = @object.location
    end
    # ...
  end
  # ...
end
```

What this does is automatically change state to navigating when tank is about to hit an obstacle. It also tracks tank location, and if tank hasn't moved 20 pixels away from it's original direction for 5 seconds, it enters `TankStuckState`, which deliberately tries to navigate away from the `stock_at` spot.

AI navigation has just got significantly better, and it didn't take that many changes.

Wrapping It Up

Our journey into the world of game development has come to an end. We have learned enough to produced a playable game, yet only scratched the surface. Writing this book was a very enlightening experience, and hopefully reading it inspired or helped someone to get a start.

Lessons Learned

Building this small tanks game and learning about game development with Ruby certainly had some nasty bumps along the way, some of them made my head hit the ceiling.

Ruby Is Slow

This shouldn't be a shocker, because Ruby is a dynamic, interpreted language, but how exactly slow it is at some points was a staggering discovery. Probably the best evidence is that drawing map tiles off screen using native extensions was actually faster than doing `Camera#can_view?` checks that involve simple integer arithmetic and range checks.

If your game is going to deal with large number of entities, Ruby will start letting you down. Dreaming about going pro? Go for C++, you won't make a mistake here.

Knowing this, keep in mind that Ruby is a wonderful language, that has it's own strengths. It's great for prototyping and dynamic things. Some 5-10 lines of Ruby could translate into 50-100 lines of C++. Also, knowing multiple languages makes you a better developer.

Packaging Ruby Games Sucks

Unless you are releasing your game for tech savvy guys who can `gem install` it, get ready to go through hell. There is no nice and easy way to create a standalone executable application from Ruby code that involves native extensions. And you will go through hell once for every operating system you want to publish your game for.

That's not everything. Want to use the latest Ruby version? Check if you can make a package for it in your target OS before you start coding. Thinking of using something that relies on ImageMagick? Too bad, you probably won't be able to package the game into a native standalone app, at least on OSX. If you are planning on releasing the game, package early and package often, for every OS, and check if there will be no problems with native extensions.

Plan Networked Multiplayer Early

If you are going to build a game, don't make a mistake of thinking "I'll just make it multiplayer later", start at the very beginning. This was a lesson I learned the hard way. There had to be a chapter in this book about turning Tanks into multiplayer, but it didn't happen, because it would require a major rewrite of the code.

Creating A Well Polished Game Requires Extraordinary Effort

Hacking up a rough prototype is extremely fun. You get to build an engine, wire everything together. It definitely gives a sense of achievement. Turning it into a great game, however, is a different story. You can spend hours or even days tweaking how game controls work and still remain unsatisfied. Every tiny detail can be pushed further. Prefer quality over quantity, and remember that you probably cannot afford both and actually finish it within next couple of years.

Start Small, Take Baby Steps

Your first few games should be small experiments, prototypes or demos. Don’t attempt to build a game you wanted to build forever with your first shot. Try reimplementing Tetris, Pacman or Bejeweled instead. You will find it to be challenging enough, and when you will feel you have the skills to do something bigger, practice just a little more.

Don’t Reinvent The Wheel

Before doing anything, research. You will probably not get point in poly collision detection better than W. Randolph Franklin did it in his research. Even if you think you can do it on your own, learn what others discovered before you. Learn from other’s mistakes, not your own.

Special Thanks

I would like to thank Julian Raschke for creating and maintaining Gosu and for all the help on IRC, Gosu forums and GitHub. This book would not exist without your enormous contribution to Ruby game development scene.

Shout out goes to Shawn Anderson, creator of Gamebox. Thank you for moral support and encouragement. Studying Gamebox source code taught me many things about Gosu and game development.

You can find Julian, Shawn and more game development enthusiasts in #gosu on FreeNode.

And most importantly, thank you for reading this book!