Extracted from:

Learn Game Programming with Ruby

Bring Your Ideas to Life with Gosu

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Learn Gome Progromming with Ruby

Bring Your Ideas to Life with Gosu

Credits

Mark Sobkowicz

foreword by Julian Rashke,
Lead Gosu developer

edited by Brian P. Hogan

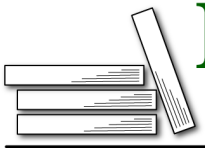
Learn Game Programming with Ruby

Bring Your Ideas to Life with Gosu

Mark Sobkowicz

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (index)
Cathleen Small; Liz Welch (copyedit)
Dave Thomas (layout)
Janet Furlow (producer)
Ellie Callahan (support)

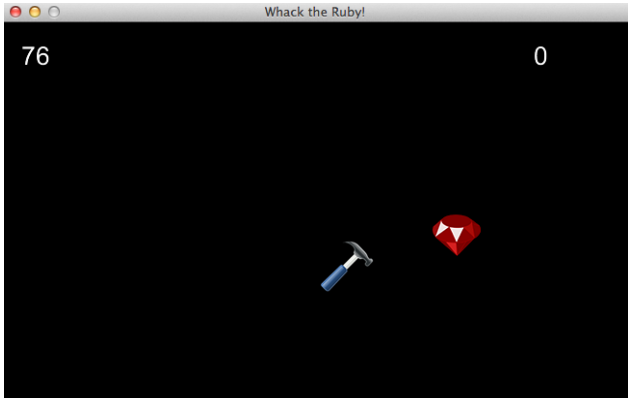
For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-073-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 3

Creating Your First Game

In this book, we're going to use Ruby and Gosu to make a variety of games. Our first one, Whack-A-Ruby, will be a pretty simple game in the spirit of Whack-A-Mole. When the game is started, a window opens on the screen, and an image of a ruby bounces around the window, blinking on and off. Players try to hit the ruby with a hammer while it's visible, scoring points when they succeed. The finished game will look like this.



We'll create this game step by step, typing code into the text editor and running it. Along the way, you'll become familiar with the most important classes and methods in the Gosu library, and you'll learn how they work together to provide the framework for your games. When you're finished, you'll be able to:

- Make a window appear on your computer screen.
- Draw an image in the window.
- Move the image around.
- Detect mouse clicks.
- Display text on the window.

We're ready to start. Fire up your text editor—it's time to write some code.

Make an Empty Window

Each game you'll write starts by opening a window on your screen. That window is where you'll bring your games to life, drawing images and making them move. Gosu provides you with a *class* for drawing that window, called Gosu::Window. This class does more than just draw the window; it also provides *methods* that give structure to your games. Each time you write a game, you'll start by creating a *subclass* of Gosu::Window. For your first game, that subclass will be called WhackARuby.

To create this class, make a new folder called WhackARuby, inside the Games folder you made in *Organize Your Workspace*, on page ?. Using your text editor, make a new file and save it in that folder with the name whack_a_ruby.rb.

To use its classes in your project, you'll need to include Gosu with require. The empty WhackARuby class looks like this.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
require 'gosu'

class WhackARuby < Gosu::Window
end
```

You can run this code, but nothing will happen yet. You need to add a *method* to your class to tell Gosu a few things about your window, and you need to create and run an *instance* of your game.

The first method you'll add to your new class is called initialize(). This method is run when you create an instance of your class, so in your game it will be run only once. In the initialize() method of WhackARuby, you'll tell Gosu what size window you want. Your window will be 800 pixels wide and 600 pixels tall. Pixels are the unit of measurement for everything in Gosu. You can make your windows bigger than this, but an 800×600-pixel window will fit on any modern computer screen with some room to spare, so it's a good size to use if you'd like to share your games with friends. Inside your initialize() method, you'll call the super() method and pass in the dimensions of your window. This sends your dimensions to the initialize() method of Gosu::Window.

You'll also set the window caption in the initialize() method. You can give the player some simple instructions—"Whack the Ruby!"—at the top of the window.

After your class code, you'll create a single instance of your game and call its show() method. You didn't write the show() method—it's part of Gosu. Your whole file now looks like this.

WhackARuby/WhackARuby_1/whack_a_ruby.rb
```
require 'gosu'

class WhackARuby < Gosu::Window
  def initialize
    super(800, 600)
    self.caption = 'Whack the Ruby!'
  end
end

window = WhackARuby.new
window.show
```

Run the program, using either the editor or the command line. If you're using Sublime Text, you can run your code with Ctrl-B on Windows or Command-B on OS X. If you're using the command line to run your program, navigate to your game folder and then use the `ruby` command.

```
$ ruby whack_a_ruby.rb
```

However you run the program, an empty window will appear. Try changing the window to 1000 pixels wide. Try changing the caption. Each time, run the program and see the results. Then change it back, so you can keep following these instructions. You should get used to running your program often. It's much easier to find any mistakes you might have made after typing a few small changes than after making a whole bunch of changes in different places in the file.

Regardless of how you configure the size of your window, it's still empty and black. So let's look at how you draw images inside it.

But first, this is probably a good time to look at the programmer's perennial problem: my code doesn't work.

What If It Doesn't Work?

You've followed the tutorial, typed a bunch of code, and then run the game. You expect the window to appear, but it doesn't. What happened? How can you fix it as quickly as possible and get back on track?

When you run your game, either with a text editor or with the command line, the program generates *output*. This program output can be very informative when your program isn't working properly. You won't see the output while the game is running. You'll only see it when the game is over or when it quits unexpectedly.

There are three ways a Ruby program can fail to work. We'll explore all three in more depth at various points in this book. The way the program fails tells you something about the cause, and knowing something about the cause can help you fix the problem and get back to making your game.

Your game doesn't run at all. Ruby can't understand your code, and you have a *syntax error*. It is structured incorrectly in some way. One common problem is that you left off an end statement. Read the program output for hints.

Your game runs, but it crashes at some point. You might see a window for just a fraction of a second, or the game might crash at some point while you're playing. One cause of this is that you spelled a method name incorrectly or spelled a variable name differently in two places. In this case, you can also read the program output to see whether that helps you figure out the problem.

Your game runs, but it doesn't behave as you expected. Problems like this can be both frustrating and fun to solve. They are like puzzles, and later in the book we'll explore some ways to dive in and see what's going wrong.

Let's look at some program output. The following output was generated by leaving the `end` off the `initialize()` method.

```
/Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:10: syntax error,
unexpected end-of-input, expecting keyword_end
window.show
           ^
[Finished in 0.1s with exit code 1]
```

In this case, Ruby tells us that the error is a syntax error and that Ruby reached the end of the file but was expecting an `end` statement. The error isn't on line 10, though, and putting the `end` after `window.show` doesn't fix the problem. With a missing `end` statement, Ruby tells you what the problem is, but you have to find the spot yourself.

Here is an example of a misspelled method. In this case, `window.show` is replaced with `window.shoe`.

```
/Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:10: in `<main>':
undefined method `shoe' for #<WhackARuby:0x007fc7f1049780@__swigtype__="
_p_Gosu__Window"> (NoMethodError)
[Finished in 1.3s with exit code 1]
```

The error message has some confusing parts, but the meaning is clear. We have an "undefined method shoe."

Whether you're following a tutorial or creating your own game, *run your program as often as possible*. This is the best thing you can do to make finding errors easier, since the error is likely in the code you just wrote.

Getting Images for Your Games

When you think about your favorite video games, what do you see in your mind? Whether you're thinking of *Angry Birds*, *Mario Kart*, or *Pac-Man*, you're probably thinking of the memorable art in the game. Your games will need art, and so do the games in this book.

Maybe you're an artist, or you know one who wants to make art for your games. If so, great! But if not, don't despair. There is plenty of art online, and much of it is free for you to use in your games. There is a list of some excellent sources in *Images and Sounds*, on page ?, and you'll find much more if you search the Internet.

The art for Whack-A-Ruby comes from the website http://www.openclipart.org. The images of a ruby and a hammer are in PNG format, which works well with Gosu on both Windows and OS X. You can find these files in the source code you downloaded in *Organize Your Workspace*, on page ?. Here is what the images look like:


The website makes it clear that all of their art is in the public domain and may be used for "unlimited commercial use." I encourage you to pay attention to the licenses under which art is released. Not everything on the Internet is free for you to use in your games, but plenty is. Some artists allow use of their art but require that you give them credit, also called *attribution*. Others require that if you use their art in your game, you have to give your game away under the same license they used to release their art. And others let you use their art with no strings attached. If you want to make your own art, go for it! Export it from your drawing program in PNG, GIF, or JPEG format, and it will be ready to use in your games.

Draw the Ruby

The first thing you'll draw in your empty window is the ruby. You have the ruby image file in your game folder, but your `WhackARuby` class doesn't know

about it yet. Gosu supplies you with a class for handling images, named Gosu::Image. In your initialize() method, create an instance of Gosu::Image and load your ruby image into it.

Add a line of code at the end of the initialize() method that loads the image file into your game. Make sure this line is inside the initialize() method, right after the line where the caption is set, and before the end that ends the initialize() method.

Instance variable names always start with an @ symbol and are variables that are accessible from all the methods in a class. To create the variable in the initialize() method and use it in another method, you need to make it an instance variable. As your games get more complex, you'll be making a lot of instance variables.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def initialize
  super(800, 600)
  self.caption = 'Whack the Ruby!'
➤ @image = Gosu::Image.new('ruby.png')
end
```

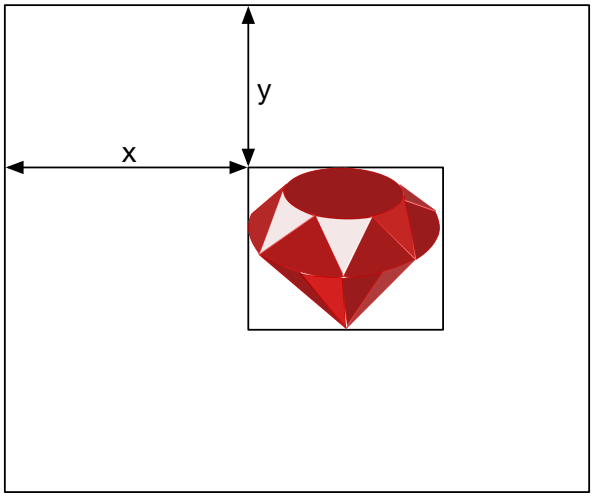
The new line of code is shown *highlighted*, with an arrow pointing to it. The rest of the code you've already written; it is shown here so you can see where to put the new line.

Next, add a new method to the WhackARuby class, called draw(). The draw() method is a special method in Gosu that is run automatically when you give the final command window.show. In the draw() method of WhackARuby, you use the draw() method of @image, the instance variable you created for the ruby.

It can be confusing at first to have two methods named draw(). The draw() method of WhackARuby is going to draw all the things in your game. The draw() method of @image is going to draw just the image of the ruby. Each image in your game belongs to a separate instance of Gosu::Image, which you use to draw that image.

When you use the draw() method of Gosu::Image, you need to specify *where* you want Gosu to draw the image by providing three *arguments*. Two arguments give the location where you want the image—the first is how many pixels horizontally from the left edge of the window, and the second is how many pixels vertically from the top of the window. From now on, you'll call these numbers *x* and *y*, like the position of a point on a graph. They are a little different from the coordinates you might be used to from math class, since the y value is measured *down from the top*, rather than up from the bottom. The third number tells Gosu how to layer images on top of each other, which

you need to think about when you have more than one image. The following figure shows the placement of an image in the window.



As shown in the figure, the position you give Gosu is where Gosu places the top-left corner of the image. For an image such as ruby.png that doesn't look rectangular, the position indicates the top-left corner of a rectangle that holds the image. This bounding rectangle is shown in the previous image but does not appear in your game window. The computer treats all images as rectangular, even ones that are a different shape.

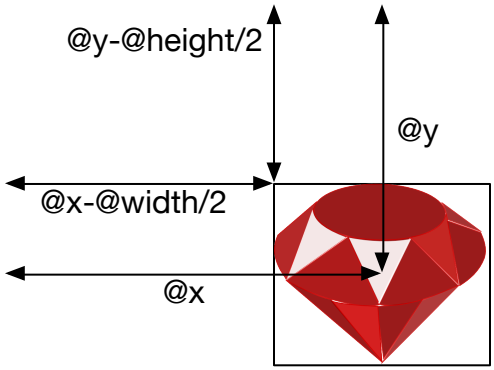
You store these positions, x and y, as instance variables in your game. You set their *initial* values, @x and @y, in the initialize() method after the line that creates the @image instance variable. The lines you add here to the initialize() method are highlighted.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def initialize
  super(800, 600)
  self.caption = 'Whack the Ruby!'
  @image = Gosu::Image.new('ruby.png')
  @x = 200
  @y = 200
end
```

One thing you need to do in many games is to find the *distance* between objects to see whether they overlap. This will be much easier if the variables @x and @y represent the position of the center of the ruby, rather than its top-left corner. You can do this by changing the values you send to the draw() method. Instead of sending @x, you send @x - @width / 2, where @width is the

width of your image. Likewise, you send @y - @height / 2 for the value of y. By doing this, your image will be centered on @x, @y, as shown in the following picture.



Set the initial values of @width and @height in the initialize() method. These values are the width and height of the ruby image, measured in pixels. Add these lines just after the lines that create the position variables and before the end of the initialize() method.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def initialize
  super(800, 600)
  self.caption = 'Whack the Ruby!'
  @image = Gosu::Image.new('ruby.png')
  @x = 200
  @y = 200
➤ @width = 50
➤ @height = 43
end
```

Then add the draw() method to WhackARuby. Put def draw just after the end of the initialize() method. Make sure the end of the WhackARuby class is after the end of the draw() method.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def draw
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
end
```

When you run the program now, you can see the ruby. It just sits there sparkling, inviting us to hit it! Before we do, we'll make it move and blink, so it's harder to hit.

Ruby Refresher: Methods Are Like Functions

If you're coming to Ruby from another language, such as Java or JavaScript, you might be used to referring to named blocks of code as functions. In Ruby, methods fill the same role, and they can have parameters and return values. We'll be writing a lot of methods, but if you've written functions in some other language, you'll find that methods are very similar.

One thing you might notice is that two different methods can have the same name. In our game, the `WhackARuby` class has a method called `draw()`, and the `Gosu::Image` class has a method called `draw()`. In the line of code `@image.draw`, Ruby knows to use the `draw()` inside `Gosu::Image`, since `@image` is of type `Gosu::Image`. We sometimes say `@image` *is* a `Gosu::Image`.

The `draw()` method of the `WhackARuby` class is *inherited* from the `Gosu::Window` class. It has a special role in Gosu games, which is discussed in the next section.