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

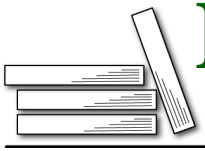
Learn Game Programming with Ruby

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*.

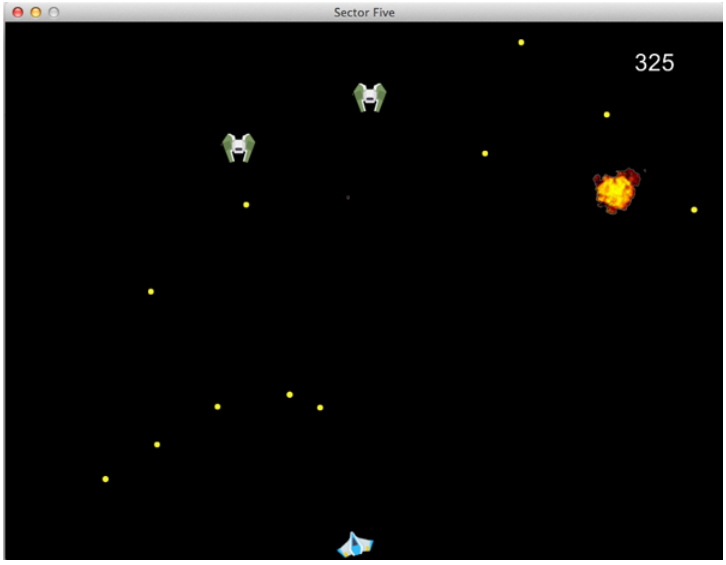
Printed in the United States of America.
ISBN-13: 978-1-68050-073-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2015

CHAPTER 4

Creating a Sprite-Based Game

Many video games—from old classics such as *Asteroids* and *Pac-Man*, to modern mobile games such as *Flappy Bird* and *Temple Run*—are made using *sprites*. Sprite is a term that dates back to 1970s video game systems like the Atari. It refers to a small image that moves around inside a scene. In a single game there can be many sprites; often one sprite is controlled by the player, and others are controlled by the program.

We're going to make our own sprite-based game called Sector Five, in which the player moves a spaceship around the screen by pressing keys. Waves of enemy ships descend from above, and the player needs to shoot them before they get to the bottom of the screen and destroy the player's base. When we're finished, it will look like this.



Sector Five has four different kinds of sprites. One is a spaceship, controlled by the player. Enemy ships are sprites that descend from the top of the screen. Bullet sprites appear when the player presses the spacebar and move in a straight line from the player ship. And animated explosion sprites appear when the bullets hit enemy ships. Each kind of sprite acts differently, and each has a Ruby *class* to describe its behavior. Once we've written the class that describes what a sprite can do, we'll create *instances* of that class, one for each object in our game.

Sector Five is a more complicated and ambitious game than Whack-A-Ruby, and we'll be working on this game for three chapters. In this chapter you'll learn to:

- Create classes to represent different kinds of sprites.
- Use those classes to create sprites in your window.
- Move a sprite by pressing the keys.
- Use constants to adjust the play of your game and make it more challenging.

In the next chapter, you'll learn how to add piles of enemy ships, all based on one class, and how to tell when sprites collide with each other. In the final chapter on Sector Five, you'll learn how to add sound effects to your game. When you're finished, you'll have all the tools you need to create your own sprite-based games.

Start by copying the folder called SectorFive_starter from the source folder you downloaded earlier into your Games folder. It has the images and sounds for all three chapters' worth of Sector Five. It also has a file, sector_five.rb, that creates a window, just like the one we started with in Whack-A-Ruby.

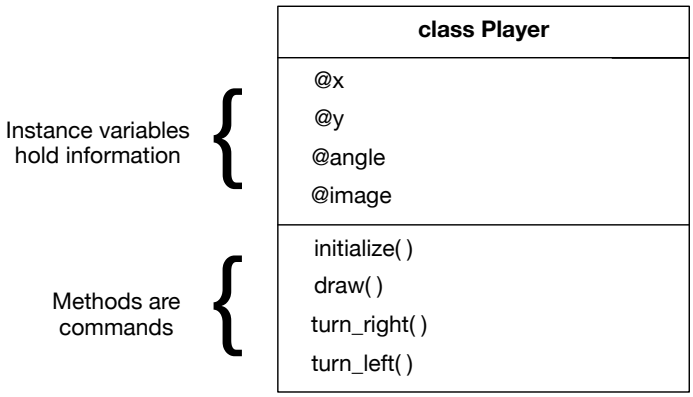
SectorFive/SectorFive_starter/sector_five.rb

```
require 'gosu'

class SectorFive < Gosu::Window
  def initialize
    super(800, 600)
    self.caption = 'Sector Five'
  end
end

window = SectorFive.new
window.show
```

This window is where you'll create, move, and draw your sprites. Run this program to make sure all is well. An empty window will open on your screen. The first sprite you'll add is a spaceship for the player to fly around the screen.

The Player Class

Each sprite class in Sector Five manages one kind of sprite. A sprite class is a collection of *instance variables* that store information about the sprite, and *methods*, which are commands you give the sprite. When you're creating a new sprite class, make a list of the information the sprite needs to store and the commands you want it to follow. The player ship sprite stores an image and a position. Because it can rotate, it also stores the angle through which its image has turned. The commands the ship follows include "turn right," "turn left," and "draw." A class diagram shows the instance variables and methods for a class in a box. Here is one for the Player class:



The initialize() method is not really a command to the ship, but it's included in the diagram because without it, there won't even be a ship.

You'll create and test the Player class one piece at a time to learn how these variables and methods work. Later, when you've had more experience, you might write most of a sprite class before you test its methods.

Create a new file, player.rb, in the same folder as your game file, sector_five.rb.

First add just enough to player.rb so that you can create and draw the ship image. To get there, create the initialize() and draw() methods of the Player class. Then, in your SectorFive class, you'll use those methods. In the initialize() method, you just say "Make a new ship, based on class Player, and store it in the @player variable." Then, when it's time to draw the ship, you just tell @player to execute its draw() method.

In the initialize() method of the Player class, you create and set some instance variables. Set the position of the ship, just as you did for the ruby in Whack-A-Ruby. Create an image variable using the file ship.png in the images folder. The initialize() method takes one argument, a reference to the window, which you'll use later to let the ship interact with the window edges.

SectorFive/SectorFive_1/player.rb

```
class Player
  def initialize(window)
    @x = 200
    @y = 200
    @angle = 0
    @image = Gosu::Image.new('images/ship.png')
  end
end
```

In the draw() method of the Player class, use a new method of Gosu::Image, draw_rot(). This method draws the image rotated by any angle, measured in degrees. Put the draw() method after the initialize() method:

SectorFive/SectorFive_1/player.rb

```
def draw
  @image.draw_rot(@x, @y, 1, @angle)
end
```

Another useful thing about the draw_rot() method is that it centers the image on the x and y values you send as the first two parameters.

Back in SectorFive, you can now use these methods to add and draw the player. First, include your new code in the sector_five.rb file, using require_relative. This line goes just after require gosu and before your SectorFive class. When some of the code is highlighted with arrows, only the highlighted code is new—the rest is there to help you figure out where to put the new code.

SectorFive/SectorFive_1/sector_five.rb

```
require 'gosu'
➤ require_relative 'player'
```

In the initialize() method of SectorFive, create the ship:

SectorFive/SectorFive_1/sector_five.rb

```
def initialize
  super(800, 600)
  self.caption = "Sector Five"
➤ @player = Player.new(self)
end
```

Notice that you send self as a parameter to the initialize() method. The initialize() method of Player takes the window as an argument. In the SectorFive class, the

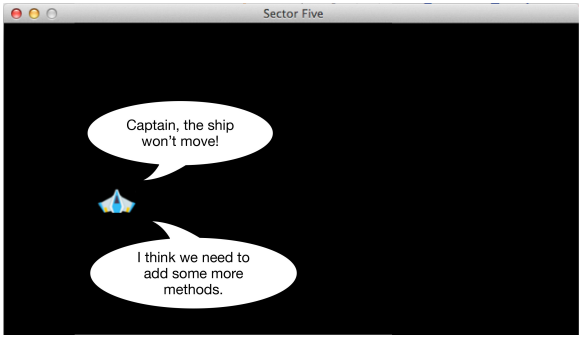
window is self, so that's what you pass to Player.new(). You'll do the same thing each time you create a new sprite.

The SectorFive class now gets a draw() method, where you'll eventually draw all the sprites in the game. For now, just draw the player ship:

SectorFive/SectorFive_1/sector_five.rb

```
def draw
  @player.draw
end
```

Before you run the game, if you're running right from your editor, make sure the sector_five.rb file is open in the front window or tab. If you run and nothing happens, you've likely got the player.rb file at the front. When you run the game, your ship will appear, as shown in the following picture:



We've gotten our ship to appear, but it just sits there. We want to let the player move it around, not with the mouse, but using the keyboard.

What If It Doesn't Work?

Now that your program consists of more than one file, errors you make can be in either file. Maybe while following the tutorial you accidentally put the code for Player.draw() in sector_five.rb. Hopefully you'd see that you had two methods named draw() in SectorFive, but what if you missed this? When you run the game this way, you get this error:

```
/Users/mark/Desktop/SectorFive_1/sector_five.rb:13: in `draw':
undefined method `draw_rot' for nil:Nil (NoMethodError) from
/Users/mark/Desktop/SectorFive_1/sector_five.rb:18: in `main'
```

The error is on the line that says @image.draw_rot(@x, @y, 1, @angle). The real clue is that Ruby sees that the method draw_rot() has been called on something that is nil. @image is nil, since it is not instantiated in SectorFive. The problem is that @image doesn't belong to SectorFive, but rather to Player. Don't give up reading the error messages! You'll get to understand them better and better if you keep at it.

Move the Ship

The player moves the ship by pressing three keys. Gosu treats keys exactly the same as mouse buttons and calls all of them buttons. You use the left arrow, the right arrow, and the up arrow to move the ship. The ship moves forward by firing its engines; the ship doesn't have any way to fire the engines backward. If you stop firing the engines, the ship coasts gradually to a stop.

Of course, this isn't actually realistic, but our primary goal is to make the game fun. Sometimes realistic is fun, and sometimes realistic is just frustrating. Our job as game designers is to strike the right balance. In this game, we want moving the spaceship around to be fun. We use some ideas from physics to make the ship feel like a real object that we are moving by pressing three keys. We want the player to be in control of the ship, but not in total control. The player has to develop a little skill to move the ship the way he or she wants. And that is fun!

Turn the Ship

To rotate the ship, create two methods in the Player class, turn_right() and turn_left(). These methods change the @angle variable, and so they change the way the ship is drawn.

SectorFive/SectorFive_1/player.rb

```
def turn_right
  @angle += 3
end
def turn_left
  @angle -= 3
end
```

Next, change the update() method of the SectorFive class to call turn_right() and turn_left() when you press the arrow keys. Gosu gives you access to the keys, just as it gives you access to the mouse. You might be able to figure out many of the key constants, like Gosu::KbLeft for the left arrow key or Gosu::KbA for the A key. There is a list of all the key constants in the documentation of the Gosu class, which you can find at http://www.libgosu.org/rdoc/Gosu.html.

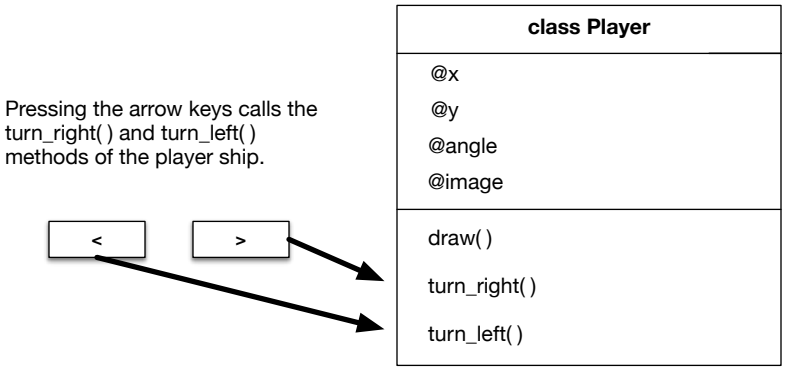
You use a method called button_down?() to see whether each key is pressed; this method is part of the Gosu::Window class.

SectorFive/SectorFive_1/sector_five.rb

```
def update
  @player.turn_left if button_down?(Gosu::KbLeft)
  @player.turn_right if button_down?(Gosu::KbRight)
end
```

(What do you think will happen if both keys are pressed?)

Look at what we just did to add a behavior to the player ship. In the Player class, you wrote two methods, turn_right() and turn_left(). Then in the update() method of SectorFive, you called those methods when the arrow keys were pressed.



You can run the game now and turn the ship with the arrow keys. After you spin the ship around a few times in each direction, you'll realize that it's time to get the ship moving forward.

button_down() vs. button_down?()

In Whack-A-Ruby, you used the button_down(id) method to detect mouse clicks and key presses. This method runs once for each time the button is clicked. To rotate the player ship, you use the button_down?() method, which is a different method. The question mark is part of the method name. When should you use each one?

button_down?()

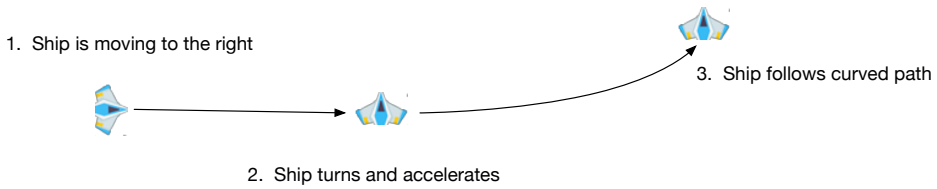
Use this method when holding the button down should do something over and over. Put it in the update() method, inside a *conditional* statement. You can use this method to turn the ship; if you hold the arrow key down, the ship keeps turning.

button_down(id)

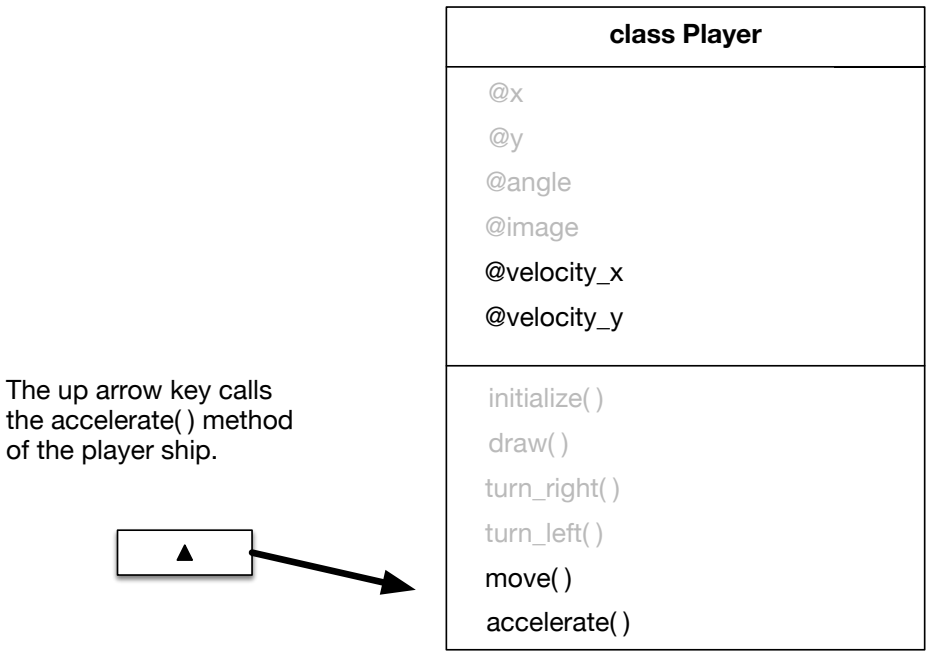
When you want the press to do something, and then not do it again until you release the button and press it again, use button_down(id). You used this method to whack the ruby, and you'll use it to fire bullets. Each key press will fire one bullet, and holding down the key won't do anything beyond the initial press. This method is separate from the update() and draw() methods and is not used inside them.

Make the Ship Accelerate

When you press the forward arrow, you want the ship to *accelerate*. Accelerate means to change the velocity. If your ship is sitting still and you press the forward arrow, it moves in the direction it is pointing, speeding up as it goes. If you turn the ship while it's moving and press the up arrow, the ship moves in a curved path, as shown in the following diagram.



To make the ship move like this, you need to add a few new variables and a few new methods to the Player class. The variables will keep track of the ship's velocity. One method of the SectorFive class, accelerate(), will be called when you hold down the up arrow key. Another one, move(), will get called every frame, so the ship keeps moving even when you're not pressing a key.

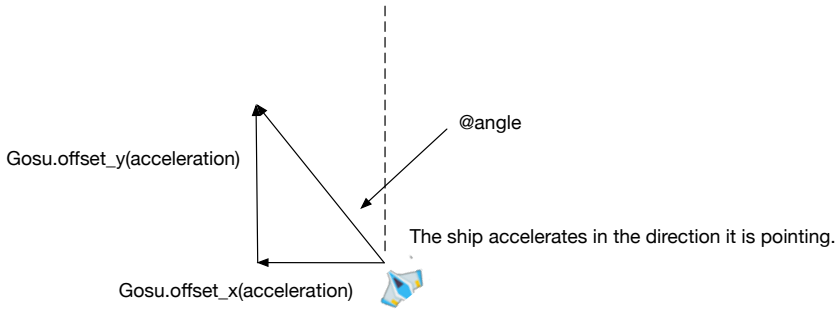


Add the two new variables and set them to 0 in the initialize() method of the Player class.

SectorFive/SectorFive_1/player.rb

```
def initialize(window)
  @x = 200
  @y = 200
  @angle = 0
  @image = Gosu::Image.new('images/ship.png')
  @velocity_x = 0
  @velocity_y = 0
end
```

In the accelerate() method, change the velocity of the ship in the direction that the ship is currently pointing. Gosu has some helper methods, offset_x() and offset_y(), that do some of the math for you.



The offset_x() method takes the angle and an amount as arguments and returns the amount in the x direction, either positive or negative. You could do this yourself using a little trigonometry, but since games make use of these calculations so often, Gosu provides them for convenience. Use these methods to change the velocity in the accelerate() method of the Player class.

SectorFive/SectorFive_1/player.rb

```
def accelerate
  @velocity_x += Gosu.offset_x(@angle, 2)
  @velocity_y += Gosu.offset_y(@angle, 2)
end
```

You change the position of the ship in the move() method of the Player class. This method is called every update, so that the ship moves even when no key is being pressed.

SectorFive/SectorFive_1/player.rb

```
def move
  @x += @velocity_x
  @y += @velocity_y
  @velocity_x *= 0.9
  @velocity_y *= 0.9
end
```

In the move() method of the Player class, you also slow down the ship by multiplying the velocities by 0.9 each update. This acts like a sort of friction and makes controlling the motion of the ship a little easier.

Now that your move() and accelerate() methods are ready, you call them in the update() method of the SectorFive class. The ship moves every frame and accelerates whenever you press the up arrow.

SectorFive/SectorFive_1/sector_five.rb

```
def update
  @player.turn_left if button_down?(Gosu::KbLeft)
  @player.turn_right if button_down?(Gosu::KbRight)
➤   @player.accelerate if button_down?(Gosu::KbUp)
➤   @player.move
end
```

Because you want the ship to accelerate continuously when the arrow key is held down, you use the button_down?() method. Run the game now and move the ship around. See whether you can fly in circles. Be careful! For now, you can fly your ship right out of the window. If you do, it can be tough to get it back in.

You've added the ship to the window, and you can move it around with the arrow keys. In the SectorFive class, you detect button presses, and those button presses call methods of the @player object to tell it what to do. The diagram on page 15 shows how the Gosu methods in SectorFive work together with the methods in the sprite class to let us create, move, and draw the player ship:

Before we add more sprites to the game, we'll spend a little more time with the ship, and you'll learn how you can adjust the way it moves to suit your players.

Use Constants to Adjust Your Game

You've put several numbers into the code that determine how the motion of the ship responds to key presses. In the turn_right() and turn_left() methods, you adjust the angle by 3. In the accelerate() method, you change @velocity_x and @velocity_y, and in move() you slow the ship down. We'll use constants to gather these numbers into one place. These constants are named with all capital letters, so you can keep them separate from variables and classes. The three constants are named ROTATION_SPEED, ACCELERATION, and FRICTION.

To create the ROTATION_SPEED constant, add a line to the Player class, just after class Player and before the initialize() method:

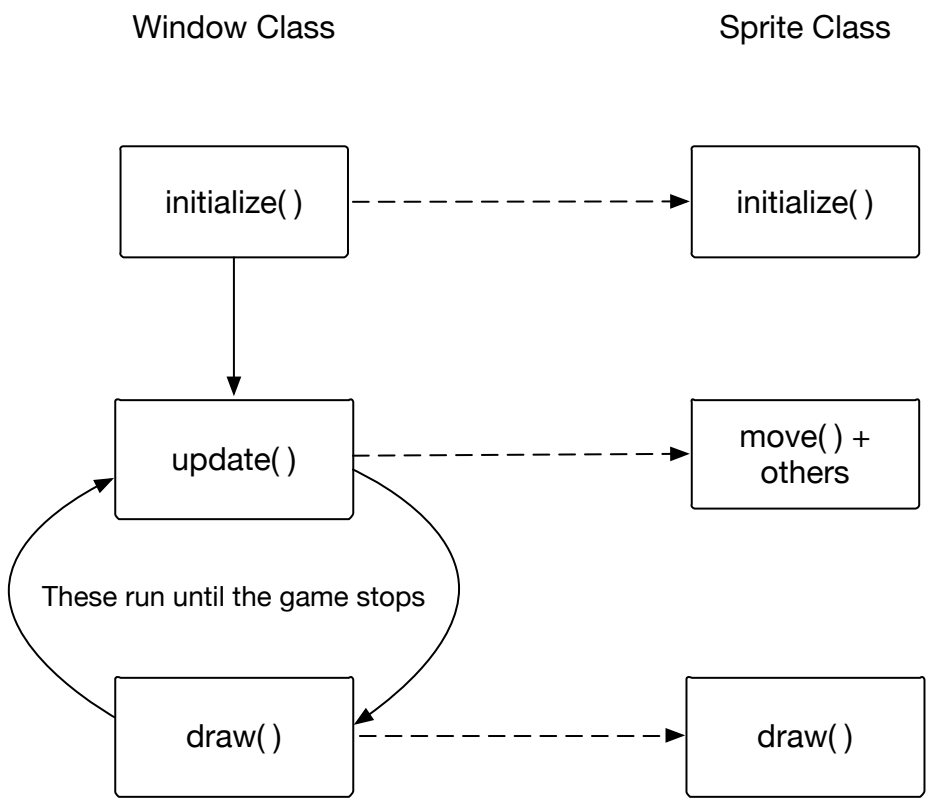


Figure 1—The Gosu run loop with a sprite

SectorFive/SectorFive_2/player.rb

```
class Player
  ROTATION_SPEED = 3
```

Then, in the turn_right() and turn_left() methods, replace the number 3 with the constant you've created:

SectorFive/SectorFive_2/player.rb

```
def turn_right
  @angle += ROTATION_SPEED
end

def turn_left
  @angle -= ROTATION_SPEED
end
```

If you run the game now, nothing has changed. But now you can change the value of ROTATION_SPEED in one place to adjust your game. Now create two more constants in the Player class, ACCELERATION and FRICTION. Put these right after the ROTATION_SPEED declaration:

SectorFive/SectorFive_2/player.rb

```
class Player
  ROTATION_SPEED = 3
  ACCELERATION = 2
  FRICTION = 0.9
```

Replace the number 2 in the accelerate() method with ACCELERATION:

SectorFive/SectorFive_2/player.rb

```
def accelerate
  @velocity_x += Gosu.offset_x(@angle, ACCELERATION)
  @velocity_y += Gosu.offset_y(@angle, ACCELERATION)
end
```

Then FRICTION replaces the number 0.9 in the move() method of the Player class:

SectorFive/SectorFive_2/player.rb

```
def move
  @x += @velocity_x
  @y += @velocity_y
  @velocity_x *= FRICTION
  @velocity_y *= FRICTION
end
```

After any or each of these replacements, you can run the game and everything should be the same.

Also create constants in the SectorFive class for the width and height of the window:

SectorFive/SectorFive_2/sector_five.rb

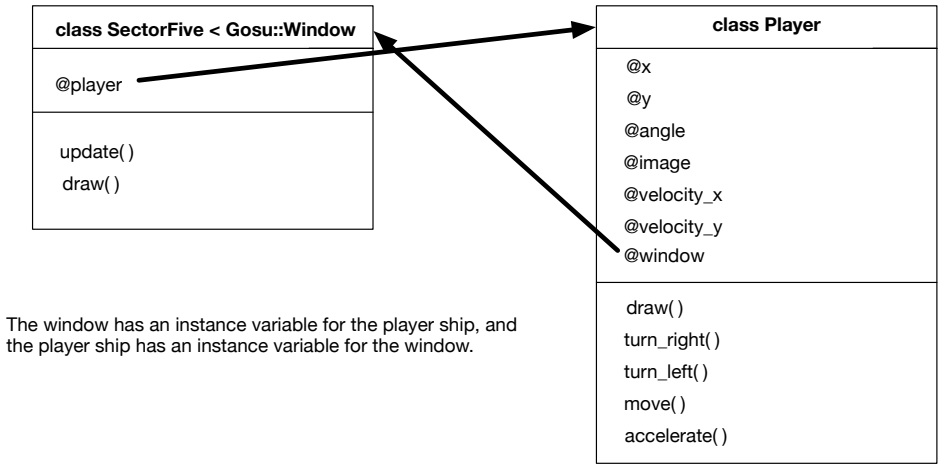
```
class SectorFive < Gosu::Window
  WIDTH = 800
  HEIGHT = 600
  def initialize
    super(WIDTH,HEIGHT)
    self.caption = 'Sector Five'
    @player = Player.new(self)
  end
```

By using constants, you get code that is a little easier to understand and a little easier to adjust. Now, if you want to adjust the rotation speed, you only have to change it in one place, not two. Naming things is often better than putting numbers right in your code.

Hitting the Edges

While flying the ship around, you probably flew the ship right out of the window at one time or another. This can be pretty frustrating, since when the ship is out of the window you can't see which way it's pointed, and it's very tough to maneuver it back into view. Think about how different games solve this problem. Some games *scroll*, so that the window actually follows the player. We'll explore this solution later in the book, in Chapter 9, *Making a Side-Scrolling Game*, on page ?. Some games *wrap*, so that if the player sprite moves off the left edge, it reappears at the right edge of the window. In Sector Five, we add bounds to our window, so that if the player ship gets to the right, left, or bottom edge of the window, the ship is stopped by the sector force fields. If the player ship ever goes off the top of the window, it is destroyed by the enemy mother ship.

For the player ship to stop at the edges, it needs to know where the edges are. You'll encounter this problem again and again, where one object—in this case, `@player`—needs to know some information about another object—in this case, the window. To solve this, you have `@player`, when it is created, save the *reference* to the window object in an instance variable called `@window`.



The window has an instance variable for the player ship, and the player ship has an instance variable for the window.

The ship reaches the edge when its center gets within a distance of the edge equal to the radius of the ship. So you also create an instance variable `@radius` in the `Player` class. Add these two variables in the `initialize()` method of the `Player` class:

SectorFive/SectorFive_2/player.rb

```
def initialize(window)
  @x = 200
  @y = 200
  @angle = 0
  @image = Gosu::Image.new('images/ship.png')
  @velocity_x = 0
  @velocity_y = 0
➤ @radius = 20
➤ @window = window
end
```

The Gosu::Window class has methods that let you use your @window reference to *get* the width and height of the window. These methods are called width() and height(). You handle the ship reaching the force fields by adding to the move() method of the Player class. When the ship reaches or overshoots the edge, you move it back to the edge and set its velocity in that direction to 0. The player ship can still go off the top of the window. In Chapter 6, *Adding Scenes and Sounds*, on page ?, you'll destroy the player ship and end the game when that happens.

SectorFive/SectorFive_2/player.rb

```
def move
  @x += @velocity_x
  @y += @velocity_y
  @velocity_x *= FRICTION
  @velocity_y *= FRICTION
➤ if @x > @window.width - @radius
➤   @velocity_x = 0
➤   @x = @window.width - @radius
➤ end
➤ if @x < @radius
➤   @velocity_x = 0
➤   @x = @radius
➤ end
➤ if @y > @window.height - @radius
➤   @velocity_y = 0
➤   @y = @window.height - @radius
➤ end

end
```

Run the game now, and the player ship will stop at the left, right, and bottom borders of the window. Next you'll add enemy ships, falling from above.