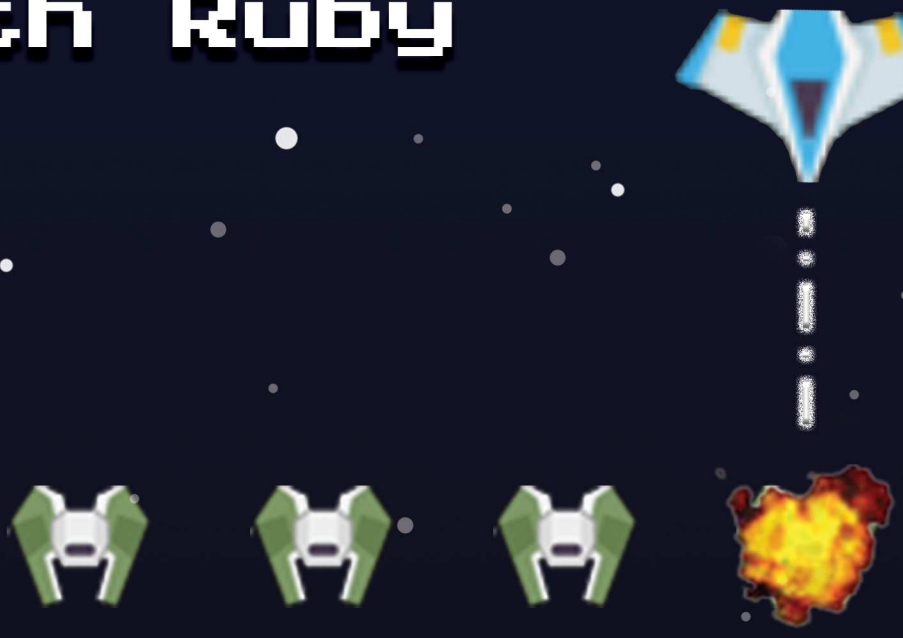
The Pragmatic Programmers

Learn Game Programming with Ruby



Bring Your Ideas to Life with Gosu

Credits

Mark Sobkowicz

foreword by Julian Rashke,
Lead Gosu developer

edited by Brian P. Hogan

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/msgpkids/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Dave & Andy

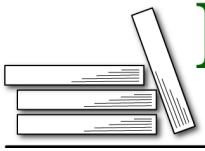
Learn Game Programming with Ruby

Bring Your Ideas to Life with Gosu

Mark Sobkowicz

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



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*.

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: B2.0—June 29, 2015

Contents

Change History

The book you're reading is in beta. This means that we update it frequently. This chapter lists the major changes that have been made at each beta release of the book, with the most recent change first.

B2.0: 29 June 2015

- The book has three new chapters and is now content-complete. The new chapters are:
  - Chapter 7, *Making a Platformer Game with Physics*, on page 131
  - Chapter 8, *Making a Side-Scrolling Game*, on page 163
  - Chapter 9, *Package and Share Your Game*, on page 181

  In addition to the new content, the book has been revised to work with Gosu 0.9.

B1.0: 15 April 2015

- Initial beta release.

Foreword

Unlike most web pages or 'normal' computer applications, 2D games do not follow a strict set of rules. Every game is different; one might be played from a top-down perspective, another might scroll from left to right, and another might be a turn-based puzzle game with square or hexagon-shaped tiles. Some games follow a story, others can be played with friends. As you can imagine, the code for each game will be just as unique as its gameplay. This lack of a common structure can be quite intimidating. How do you start building a city planning game, a virtual football match, or any other kind of game?

This book guides you through the process of writing four very different games. The exercises in each chapter will show you how to extend each kind of game. You can mix and match parts from different chapters, or try to mimick features from your favorite video games in Ruby. Try to structure the code differently every once in a while. Eventually, you will be able to build games that are unlike any example shown in this book.

While these games are different, some tasks like reacting to keyboard input, displaying image files, or playing sound effects are the same. This book uses a library called Gosu, which provides all of these basics on Windows, OS X and other operating systems. Gosu is a media library, not a complete game development kit. For example, there is no Map class or any collision detection logic in Gosu, because it is hard to design, much less use, a library that suits *all* kinds of games at the same time. This lack of reusable components is a good thing though. It means that this book is less about learning Gosu, and more about constructing games using universal programming constructs such as objects, methods, loops, and arrays.

The Ruby language is a great candidate for this task. Working with objects feels natural in Ruby, and games are fantastic for learning object-oriented programming. If you look at the screen, you can usually see which objects

and classes are involved in a game, making it straightforward to model them in code.

But Ruby is also very concise. Game design is about experimentation and refinement, not about having a master plan from the start. Building a game is often about deleting code and trying something else. In a verbose programming language, discarding code can be frustrating. Who wants to give screens full of painfully hand-written code? In Ruby, the same logic might fit into a few lines of code, and re-writing it is a breeze.

With these tools in your hands, enjoy inventing your very own games!

Julian Raschke
Gosu Developer

Acknowledgements

Before I thank anyone else, I want to thank my wife Michelle. Throughout the time I spent working on this book, she never said a word that was not supportive. This book is a testament to her patience and good nature, more than any other factors.

My students at Lincoln Sudbury Regional High School were my original inspiration to write this book, as many of them had questions that went beyond what we could cover in our short *Introduction to Programming* class. They showed me time and again that imagination and creativity are the secret sauce; that you don't have to be an expert to have a great time making games, and that beginners can make games that are exciting and fun to play. A shout out to Chris A. and Mia F. who asked some of the hardest questions.

This book would not be possible without the work of Julian Raschke, who wrote the Gosu gem for Ruby. In addition to this gift to the Ruby community, he was patient and generous with me personally, reviewing the book and helping me understand some of the finer points of the library he wrote.

Thanks to my technical reviewers for their insightful suggestions and for finding my mistakes, both in the code and its explanation. I am grateful to them for lending me their expertise and time to help make this book better. I gratefully acknowledge the contributions of Craig Castelaz, Douglas Gray, Scott Hofmann, Marianne K, Steve Morss, Rudy R, Darian Springer, Charlie Stran, and Stephen Wolff.

I'd like to acknowledge my colleagues at Lincoln Sudbury Regional High School for creating a place where intellectual curiosity is nurtured and where risks and experimentation are encouraged and expected. May Lincoln Sudbury forever remain a "different kind of place."

And a final thanks for the patience and perseverence of my development editor, Brian Hogan, managing editor Susannah Pfalzer, and all the fine people working at The Pragmatic Programmers. I brought my idea to them because

I love their books, and I thank them for the opportunity to work with them on this one.

Introduction

Hello and welcome. This is a book about making games with and for your computer. Games like some of the ones you love on your computer, your phone, or your game console. By working your way though this book, you'll make four games, each of a different type. You'll learn how to open a window on the screen of your computer and then fill it with moving pictures. You'll make those images interact with each other, and you'll control them with the mouse and keys of the computer.

The games we make will feel pretty familiar to you. They aren't copies of other games, but they do use familiar patterns and principles that you'll be able to use to make your *own* games. The techniques you learn making a spaceship fly around by pressing keys on the keyboard can also help you move a chicken across a road full of traffic. Each chapter in this book is based around a few different elements of game creation, and each element is applicable to a wide range of games.

The goal of this book is to help you bring your own ideas to life. When you've learned these elements of game development, you'll be able to take a game you've imagined, and create that game so that it runs on your computer - and also on your friends' computers.

Ruby and Gosu

Along the way, you'll level up your programming skills. Becoming a better programmer will help you make great games, and making games will help you improve your programming skills. To make the games in this book you'll use the Ruby programming language, along with a game library called Gosu. Ruby is a great language both for learning to program, and for making games. It has "an elegant syntax that is natural to read and easy to write."[1]. The Ruby language focuses on *objects* and this makes it a great fit for creating games, as you'll see as you work through this book.

1. https://www.ruby-lang.org/en/

The Gosu game library will provide the structure for your games, while leaving their design and content completely up to you. It gives you the tools you need to place images on the screen, move them around, and play sounds to spice up your game. At the same time it doesn't do anything you won't understand. You and your code will always be in control of what is happening in your game window. Gosu is also a great springboard to other platforms. In particular, working with SpriteKit, Apple's framework for making two dimensional games for iOS and OS X, feels like a natural step up from writing games with Gosu.

Ruby and Gosu are free, open source software that work well on both Windows and Mac OS X computers. Since you can just download everything you need, it's easy to get started yourself, and perhaps you find other people willing to learn game programming along with you in your school or town. There is an online community dedicated to game programming with Ruby and Gosu, with a showcase where people share their games and a forum where they ask and answer questions[2]. Many people in this community have shared both finished games and the *code* for those games, and these can be a great source for learning and inspiration.

What You'll Need

First, you'll need a computer. It can be Mac running OS X 10.9 or later, or it can be a Windows computer running Windows 7 or later. You need to be comfortable with the *file system* on your computer, so you can save files where you want them and organize them into folders.

To get the most out of this book, you need a little programming experience. If you have already used Ruby, you're ready to go. If you have experience with a different programming language, you might want to pick up a book like *Learn to Program [Pin09]* or *Programming Ruby 1.9 & 2.0 [FH13]* and learn a little Ruby syntax before you start on the games in this book.

The Road Ahead

As you go through this book, we'll be making some games together. The games follow a progression, and each chapter assumes that you have worked through the preceding chapters. Here is a summary of what you'll be learning in each chapter.

2. http://www.libgosu.org/

- In Chapter 1, *Get Ready*, on page 1, you'll set up your computer to use Ruby and Gosu to write games.
- In Chapter 2, *Creating Your First Game*, on page 13, you'll make a simple game. You'll learn how a Gosu game is organized and how to use Gosu to open a window on your computer, fill it with pictures, and move those pictures around.
- In Chapter 3, *Creating a Sprite-based Game*, on page 39, you'll begin a new game, Sector Five, in which a player controls a spaceship that shoots down enemy invaders. Each thing you draw on the screen will be a *sprite*, and you'll learn to create a class for each type of sprite in the game.
- In Chapter 4, *Managing Lots of Sprites*, on page 57, you'll learn how to manage many sprites, by organizing them with arrays. By iterating through these arrays, you'll be able to handle the movement and interactions of many sprites in the window at once.
- In Chapter 5, *Adding Scenes and Sounds*, on page 81, you'll break your game into multiple scenes, giving it a start scene with some instructions, and an end scene with credits. You'll learn how to add music and sound effects to finish Sector Five.
- In Chapter 6, *Creating a Puzzle Game*, on page 103, you'll create a puzzle game called Twelve. This game focuses on user interaction, and you'll learn how to write code to implement the rules of the game.
- In Chapter 7, *Making a Platformer Game with Physics*, on page 131, you'll use a physics engine to make objects move naturally. In Escape, a hero will jump between platforms, dodging boulders that fall, spin, and bounce.
- In Chapter 8, *Making a Side-Scrolling Game*, on page 163, you'll learn how to make our platformer game scroll, using a camera object to follow the motion of the hero. This will allow you to have a game field that is bigger than your screen.
- In Chapter 9, *Package and Share Your Game*, on page 181, you'll learn to package up a game into a single executable so you can share it with your friends.

When we're done, you'll have a better understanding of how games are put together, and some new programming tools in your toolbox. You'll be ready to take your own ideas, and turn them into games.

Bumps in the Road

Whether you're following a tutorial, or writing your own code, it can be frustrating when things don't work. This book tries to anticipate some of these situations, and give you some strategies to deal with them. At these places in the tutorial, you'll find a section with some ideas for how to solve particular problems.

What if it Doesn't Work?

We'll explore different sorts of problems that can occur, and look at ways to solve them. We'll learn how to interpret some common errors and look at ways to find answers on the internet.

Writing games is fun, and hopefully the rewards of making your programs work will outweigh the frustration you feel when they don't. Sometimes the answer will come to you only after you've walked away from the computer for a little while. Stick with it! Persistence is one of the most important assets a programmer can have.

What's Next?

Before you can actually start making games, you'll need to install a few things on your computer. The next chapter will explain what you need, and take you step by step through getting ready.

CHAPTER 1

Get Ready

Using the Ruby programming language and the Gosu game library, we'll create some amazing games with images, sounds, and Ruby code. Before you can start writing those games, you need to install a few things on your computer. To write programs we'll need to use programs! Some of those programs will be the kind you might be used to, that you launch by clicking an icon. Others do their work in the background; they are unseen libraries of code that your computer needs to run the programs you write. All together, the tools and programs you use to write your games are called your *development environment*. The environment you need to make games with Ruby and Gosu consists of the following things.

Ruby

Ruby is the programming language we'll use to create our games. If you've written Ruby code before, great! If you haven't, the tutorials in this book are complete, and you can learn some Ruby by following them. To get a deeper understanding of Ruby, you might want to take a look at a book like *Learn to Program [Pin09]*, or use an online tutorial, such as the one on Code Academy[1].

Developer Tools

In order to run Ruby programs that create windows on the screen and play sounds, we need to install some background libraries. These libraries let our Ruby programs access the system resources needed to run our games.

Gosu

Gosu is the library that we'll be using to write our games. Gosu is a collection of Ruby classes designed to make game writing simple and fun.

1. http://www.codecademy.com

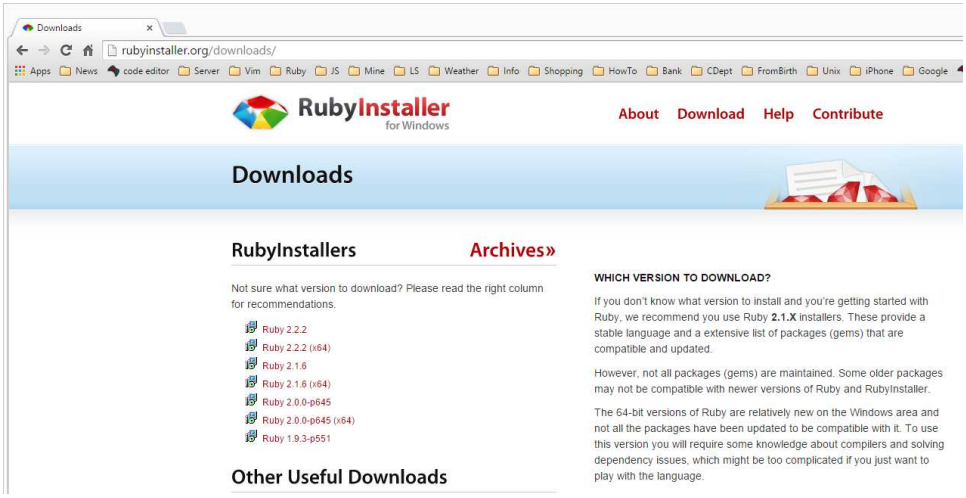
A Text Editor

We use a text editor to write our programs. A text editor creates plain text files, without any formatting. Microsoft Word and Apple Pages are not text editors, since the files they produce contain all sorts of information besides the text in the document. You can use any text editor to write code, but some have useful features, both for writing programs and for running them right from the editor.

How you get these things installed depends on which operating system you're using. If your computer runs Windows, read on. If you're using a Mac, skip ahead now to *Getting Ready with OS X*, on page 6.

Getting Ready with Windows

With Windows, you start by installing Ruby. Point your browser to http://rubyinstaller.org and click the big red button titled 'Downloads'. This takes you to a page with a list of things, as shown in the following image.



You install both Ruby and the Developer Tools from this page. From the list of Ruby installers, choose the one for Ruby 2.2.2 (or 2.2.x, where x is a number larger than 2.) Don't get the one labelled (x64).

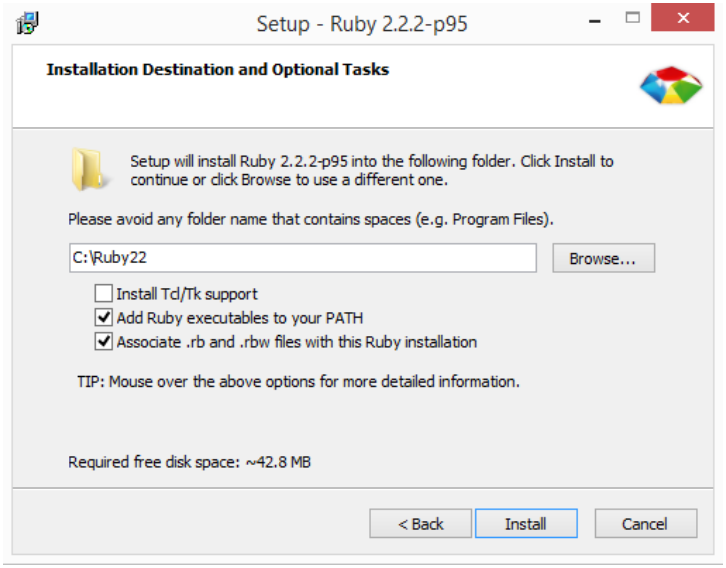
Ruby Versions on Windows

The goal of this book is to make installing Ruby and Gosu as simple as possible for a wide range of users and computers. When this was written, the newest version of Ruby available on Windows was Ruby 2.2.2. When you do the installation, that might no longer be true. Look for the version that starts with 2.2. If you already have a version of Ruby, or if you want to use the latest and greatest, by all means give it a

try. For a setup that will get you writing games as soon as possible, use these instruction for Ruby 2.2.

All the games in this book work with any Ruby 1.9.3 or later.

When you run the installer, you are first asked to accept the license. Then a dialog, shown in the following figure presents you with three checkboxes. Check the two shown and continue.



The installer creates a folder `Ruby22` in the `C:\` directory, and puts your Ruby in that folder. You can use the File Explorer to see that it is there.

From the same web page, a little further down, download the Development Kit that is “For use with Ruby 2.0 and above (32 bit version)” as shown in the following image.

DEVELOPMENT KIT

For use with Ruby 1.8.7 and 1.9.3:

DevKit-tdm-32-4.5.2-20111229-1559-sfx.exe

For use with Ruby 2.0 and above (32bits version only):

DevKit-mingw64-32-4.7.2-20130224-1151-sfx.exe

For use with Ruby 2.0 and above (x64 - 64bits only)

DevKit-mingw64-64-4.7.2-20130224-1432-sfx.exe

MD5 & SHA256 Checksums

For MD5 and SHA256 checksums of available downloads please check the corresponding **package/version** *files tab* or *release notes* at the RubyInstaller repository on Bintray.

Sorry the inconvenience.

SPEED AND COMPATIBILITY

RubyInstaller is compiled with MinGW which offers improved speed and better RubyGem compatibility, including support for many more native C-based extensions such as Ruby FFI, Nokogiri, FXRuby and many others.

CONVENIENCE

No additional software is needed if you want to use the executable versions of the RubyInstaller. If you would like to use the 7-Zip archived versions or the Ruby documentation, you will need to download 7-Zip from the 7-Zip website.

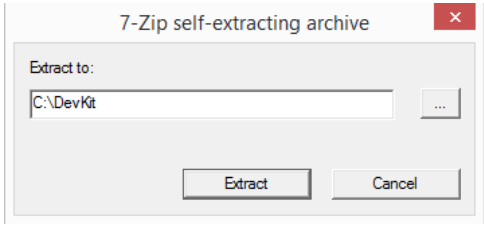
DOCUMENTATION

As an added convenience for Windows users, we've made available the Ruby Core and Standard Library documentation in Compiled HTML Help (CHM) format.

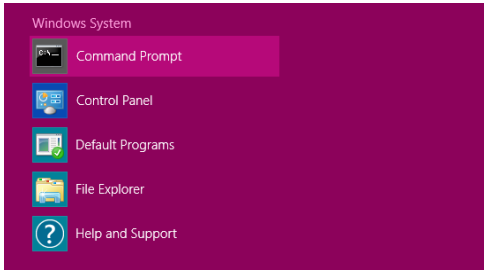
BUILD YOUR OWN NATIVE EXTENSIONS

The RubyInstaller Development Kit (DevKit) is a MSYS/MinGW based toolkit than enables you to build many of the native C/C++ extensions available for Ruby.

When this download is finished, run the installer. You need to tell the installer where to put the Development Kit. Put it in a folder of its own, called DevKit, alongside the Ruby22 Folder. To do this, enter C:\DevKit in the window as shown in the following image.



Even though you've run the Development Kit installer, you're not done installing these libraries. To finish, you have to dive a little deeper. You need to use the *command line* to do the rest of the installation. Take a look in your launch window, or start menu. The following image shows the launch window in Windows 8.



Find and open the "Command Prompt" application. This application opens a window on your screen with a prompt where you can type commands to the computer. When you've completed a command and hit the return key, your computer responds with the results of your command. When you open the

Command Prompt, your location in the file system is your home directory. To finish installing the Developer Kit, we move the location where commands are executed into the new DevKit folder. Type 'cd \DevKit' at the command line, and hit the return key.

```
C:\> cd \DevKit
C:\DevKit\>
```

The prompt in the Command Line application now indicates that you're ready to execute commands in the DevKit folder. The next command finds the Ruby you installed previously, and sets up the Development Kit to use it. Note that you're running a Ruby program called dk.rb with this command.

```
C:\DevKit> ruby dk.rb init
[INFO] found RubyInstaller v2.2.2 at C:/Ruby22

Initialization complete! Please review and modify the
auto-generated 'config.yml' file to ensure it contains
the root directories to all of the installed Rubies you
want enhanced by the DevKit.
```

This is good news, since the Development Kit is going to enhance Ruby 2.2 just as you want it to. You don't need to do anything to the config.yml file. Next, at the command line, run the following.

```
C:\DevKit> ruby dk.rb install
[INFO] Installing 'C:/Ruby22/lib/ruby/site_ruby/devkit.rb'
```

Your Ruby is ready. You can now execute Ruby programs, either from the command line, or directly from a text editor, which we'll look at soon. You still need to install Gosu, which you also do from the command line. Gosu is a Ruby Gem, meaning a package of code that extends what Ruby can do. The Gosu Gem includes code that opens a window on your computer screen, fills it with images that move, and plays sounds though your speakers. At the prompt, now type the following.

```
C:\DevKit> gem install gosu
Fetching: gosu-0.9.2-x86-mingw32.gem (100%)
Successfully installed gosu-0.9.2-x86-mingw32
1 gem installed
```

Gosu is now ready. The last game in the book uses one more gem, called Chipmunk, which is a physics library and will help handle some complicated math. Install Chipmunk now by typing the following at the command line.

```
C:\DevKit> gem install chipmunk
Fetching: chipmunk-6.1.3.4.gem (100%)
Temporarily enhancing PATH to include DevKit...
Building native extensions.  This could take awhile...
```

```
Successfully installed chipmunk-6.1.3.4
1 gem installed
```

You are now ready to write some games, and you should skip the section on installing for OS X and move on to *Install a Text Editor*, on page 10. If something went wrong, you should instead read *What if it Doesn't Work?*, on page 9.

Getting Ready with OS X

These instructions have been tested on OS X Mavericks (10.9.5), and on OS X Yosemite (10.10.3). There was a bug in OS X 10.9.2 and earlier that required a complicated workaround, so check the OS X version you are using.

To do this installation, you need to be logged into the computer with an account that has administrator privileges. If you're working on your own computer and yours is the only account, you are all set. If not, you can see what privileges an account has in the Users and Groups panel of System Preferences.

If you're logged in on an account that says "Standard" instead of "Admin", you should get some help from the owner of the computer, either to change your account to an Admin account, or to help you do the installation on their account. You won't need to be an administrator to write and run programs, just to install the software in this chapter.

All the steps of this installation are done using the *command line*. Find and run the application called Terminal. It's in a folder called Utilities in the Applications

folder. Terminal provides a prompt where we can type commands. The prompt is a $ symbol. When you're typing the commands in this chapter, don't type the $.

On OS X, Ruby is already installed. You can make sure of this by typing the following at the command line.

```
$ ruby --version
ruby 2.0.0p481 (2014-05-08 revision 45883) [universal.x86_64-darwin14]
```

The Ruby version that comes with OS X is version 2.0.0. The number following the 'p' is a build number. There are newer versions of Ruby, but this version is fine to get started writing some games. Your Ruby is ready.

Next, install the command line tools. Xcode is Apple's all-in-one programming environment, and the development tools we need are a small part of this large package. Apple provides a way to install just the parts we need, by typing the following at the command line prompt.

```
$ xcode-select --install
```

When you hit return after typing the command, a window opens asking if you'd like to install the tools now, as shown in the following figure.

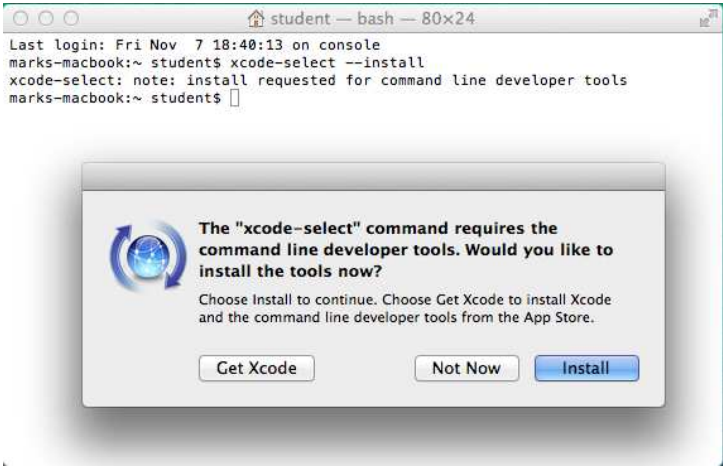


Figure 1—Installing the Xcode Command Line Tools

Choose "Install", and wait a few minutes as the tools are installed.

The next thing you install is called Homebrew. Homebrew is a package manager for the Mac, which means it is used to install and update other software. You're going to use it to install three libraries of code that Gosu needs, but

that do not come with the XCode Command Line Tools. To install Homebrew, type a single long line of code at the command line prompt and then hit return. You can copy and paste it from the Homebrew website, at http://brew.sh. The install instructions are near the bottom of the web page.

If you prefer you can type the following command *onto a single line in Terminal*.

```
$ ruby -e "$(curl -fsSL
   https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

This line of code first downloads the Homebrew installer, which is a Ruby program. It then runs the installer with the `ruby` command. The installer shows you exactly what it is installing as it goes. It asks for your password in order to run the installation.

```
WARNING: Improper use of the sudo command could lead to
data loss or the deletion of important system files.  Please
double-check your typing when using sudo.  Type "man sudo"
for more information.

To proceed, enter your password, or type Ctrl-C to abort.

Password:
```

This warning might sound scary, but behind the scenes its doing the same thing as any software installation that requires your password. When you use `sudo` yourself you should indeed be very careful about what commands you give. In this case Homebrew is using it for you, and it is quite safe.

Once Homebrew is installed, it recommends that you run `brew doctor` to see that it is installed properly.

```
$ brew doctor
```

Homebrew might have some recommendations for you, or it might just reply that everything is fine, with the note, “Your system is ready to brew.”

We now use Homebrew to install some libraries. The three libraries we’ll need are titled ‘sdl2’, ‘libogg’, and ‘libvorbis’. They enable Gosu to play a variety of

audio formats, and you won't be able to install Gosu without them. Install them using Homebrew by typing the following at the command line.

```
$ brew install sdl2 libogg libvorbis
```

Homebrew provides a series of messages as it downloads each item and installs, or 'pours' it.

Once these are installed, you're done with Homebrew, but not done with the command line. Next, you install Gosu itself. Gosu is a Ruby Gem, or package, and in order to install it you use the `sudo` command. Since you're typing it yourself this time, copy the following command very carefully at the command line prompt. You may need to provide your password again.

```
$ sudo gem install gosu
Fetching: gosu-0.9.2.gem (100%)
Building native extensions.  This could take a while...
Successfully installed gosu-0.9.2
```

The last game in the book uses one more gem, called Chipmunk, which is a physics library that handles some complicated math. Install Chipmunk now, at the command line.

```
$ sudo gem install chipmunk
```

The messages you get will be almost the same as the ones you got installing the Gosu gem. You're now ready to write some games with Gosu. To actually write them, you'll use a text editor, which we discuss next, in *Install a Text Editor*, on page 10.

What if it Doesn't Work?

Following directions like the ones in this chapter can be difficult. Typing commands at the command line is tough, since every character has to be exactly correct. Even if you follow every direction in this chapter something might not work. You might be using an older system, or a different Ruby version might already be installed on your computer. If the command line says there is an error, scroll back and see if you typed the command exactly as given. If you didn't, start again at that point. If you can't figure out where the install started to go wrong, start at the beginning. The installers we've used will not cause any problems if run more than once. If you are still having trouble getting Ruby and Gosu up and running, there are some more discussions of how to set up Gosu on the Gosu Wiki [2]. The Gosu website also has an issues page where Gosu users post problems they are having, including

2. https://github.com/gosu/gosu/wiki

installation problems [3]. In addition, you can use the forum at Pragmatic Programmers dedicated to this book to talk with other readers, as well as with the author.

Install a Text Editor

You can write Ruby programs in any text editor, such as NotePad on Windows or TextEdit on OS X. Your experience will be much better with a specialized programming text editor, which can help you by automatically indenting code, coloring your code based on its syntax, and more. If you have a favorite, and know how to use it, by all means use what you have. If you don't yet have a programming editor, using Sublime Text 3 is a great option for working with Ruby and Gosu.

Sublime Text 3 is a cross platform text editor with versions for Mac, Windows and Linux. You can download the most recent version of Sublime Text 3 at http://www.sublimetext.com/3. When you are editing a Ruby file in Sublime Text, you can run it with CTRL-B on Windows, or Command-B on OS X. This is a great convenience compared to running it each time from the command line. You can try a full featured version of Sublime Text 3 for free, and use it to do the tutorials in this book.

Organize Your Workspace

Each game you write consists of a number of computer files. Some are text files you create with your text editor. Others are image and sound files you make with other software or find on the internet. To keep these files organized, make a new folder on your `Desktop`, and name it `Games`. For each game you make, put a new folder inside this one. In addition, make some folders to store images and sounds you make or find that you might want to use in a future game.

3. https://github.com/gosu/gosu/issues

Then get the source code and other files for the projects in this book. They've been compressed and archived into a single file which you can download at the web page for this book, https://www.pragprog.com/book/msgpkids. Download that file and unarchive it. Move the resulting folder into your Games folder and take a look inside. Inside is a folder for each of the games we're going to make. Each game folder has several folders inside. The one with the name ending in _starter has just the image and sound files for the game. The others are versions of the game at different points along the path to completion. You can use them to check your code if you get stuck, or as starting points if you want to focus on a particular part of the book.

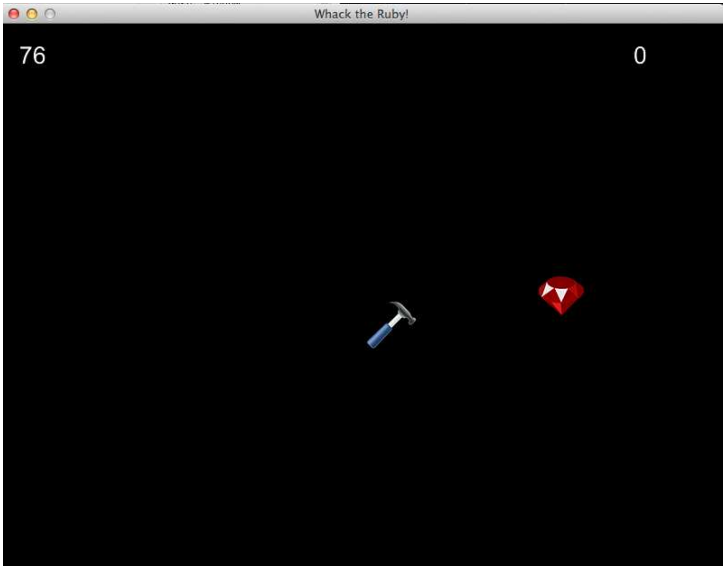
What's Next

You're ready to start! You have Ruby along with the Gosu and Chipmunk gems installed and ready to go. Next, you'll create your first game with some Ruby code, the Gosu library, and a few images.

CHAPTER 2

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 you play, 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 our 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 our games. When we're done, you'll be able to:

- Make a window appear on the screen of your computer.

- 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 we write starts by opening a window on our screen. That window is where we bring our games to life, drawing images and making them move. Gosu provides us 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 our games. Each time we write a game, we'll start by creating a *subclass* of Gosu::Window. For our 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 10. Using your text editor, make a new file and save it in that folder with the name whack_a_ruby.rb.

To use Gosu's classes in our project, we need to add it 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 happens yet. We need to add a *method* to our class to tell Gosu a few things about our window, and we need to create and run an *instance* of our game.

The first method we add to our new class is called initialize(). This method is run when we create an instance of our class, so in our game it will be run only once. In the initialize() method of WhackARuby, we tell Gosu what size window we want. Our 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 x 600 pixel window will fit on any modern computer screen with some room to spare, so its a good size to use if you'd like to share your games with friends. Inside our initialize() method we call the super() method and pass in the dimensions of our window. This sends our dimensions to the initialize() method of Gosu::Window.

We also set the window caption in the initialize() method. We give the player some simple instructions: “Whack the Ruby!”, at the top of the window.

After our class code, we create a single instance of our game, and call its show() method. We didn’t write the show() method - it’s part of Gosu. Our 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, either using 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 appears. 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. Its 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 we configure the size of our window, it’s still empty and black. So let’s look at how we draw images inside it.

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 way is that you left off an 'end' statement. Read the program output for hints.

Your game runs, but 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 if that helps you figure it out.

Your game runs, but 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 whats going wrong.

Lets look at some program output. The following output was generated by leaving the `end` off of 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 also 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 does not 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 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 189, 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 10. Here is what the images look like.


This website makes it clear that all 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 we draw in our empty window is the ruby. We have the ruby image file in our game folder, but our `WhackARuby` class doesn't know about it yet. Gosu supplies us with a class for handling images, named `Gosu::Image`. In our `initialize()` method, we create an instance of `Gosu::Image`, and load our ruby image into it.

We add a line of code at the end of the initialize method that loads the image file into our game. Make sure this 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. In order to create the variable in the initialize() method, and use it in another method, we need to make it an instance variable. As our games get more complex, we'll be making a lot of instance variables.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

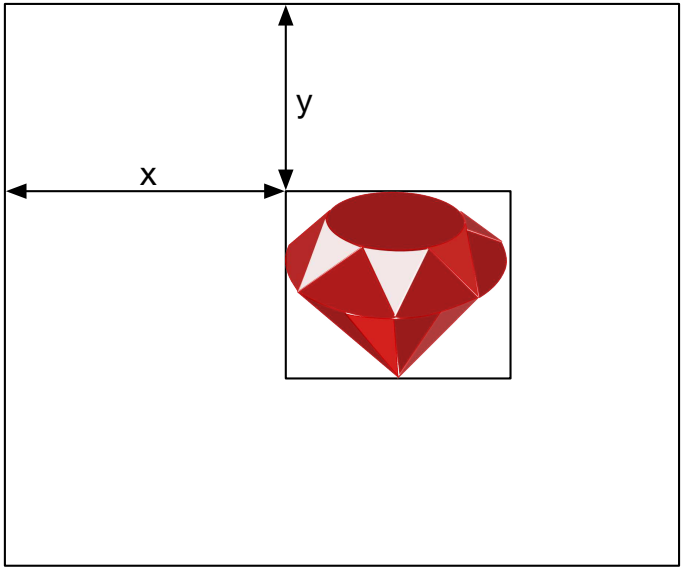
```
def initialize
  super(800, 600)
  self.caption = 'Whack the Ruby!'
➤ @image = Gosu::Image.new('ruby.png')
end
```

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

Next we add a new method to the WhackARuby class, called draw(). The draw() method is a special method in Gosu, that is run automatically when we give the final command window.show. In the draw() method of WhackARuby we use the draw() method of @image, the instance variable we 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 our game. The draw() method of @image is going to draw just the image of the ruby. Each image in our game belongs to separate instance of Gosu::Image which we use to draw that image.

When we use the draw() method of Gosu::Image, we need to specify *where* we want Gosu to draw the image, by providing three *arguments*. Two arguments give the location where we 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, we'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 we need to think about when we have more than one image. The following figure shows the placement of an image in the window.



As shown in the picture, the position we 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 picture, but does not appear in our game window. The computer treats all images as rectangular, even ones that are a different shape.

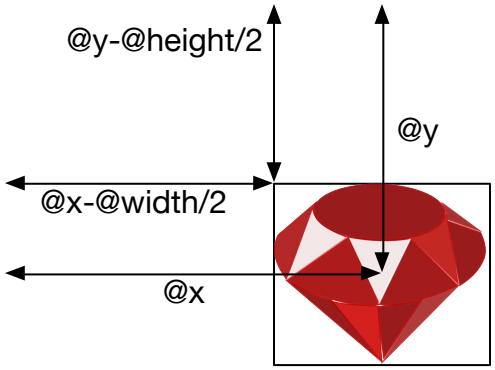
We store these positions, x and y, as instance variables in our game. We set their *initial* values, @x and @y, in the initialize() method after the line that creates the @image instance variable. The lines we add here to the initialize() method are highlighted.

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 we need to do in many games is to find the *distance* between objects to see if 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. We can do this by changing the values we send to the draw method. Instead of sending @x, we send @x - @width / 2, where @width is the width of our image.

Likewise, we send @y - @height / 2 for the value of y. By doing this, our 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 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.

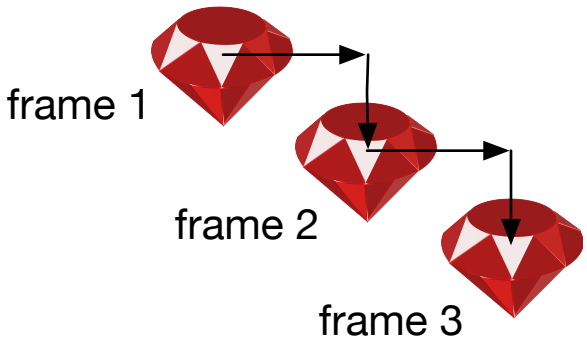
The draw() method of the WhackARuby class is *inherited* from the Gosu::Window class. It has a special role in Gosu games, which is discussed in the next section.

Move the Ruby

What do we mean, in a game like this one, when we say we want to move something? A video game is like a movie, in that it basically fools the user into seeing motion by showing a rapid sequence of pictures in which some things are in a slightly different position. We call these pictures *frames*. If an image on our screen changes position too far between one frame and the next it doesn't look like it is moving; it looks like it is jumping around.

Velocity

In order to give the ruby image the appearance of smooth motion, we change its position by the same amount every frame. This amount is called the *velocity*. To move the ruby in the horizontal direction, we change the value of @x. To move it in the vertical direction, we change the value of @y. We create two new variables, @velocity_x and @velocity_y to keep track of the velocity. The following image shows the position of the ruby over several frames of the game. Each frame the ruby moves @velocity_x to the right, and @velocity_y down.



This picture exaggerates the velocity to clearly show how we move the ruby. Our actual velocity is much smaller than the size of the ruby, so the ruby of each frame overlaps where the ruby was in the previous frame. We create the velocity variables in the initialize() method, after the lines that set the width and height of the ruby.

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
➤ @velocity_x = 5
➤ @velocity_y = 5
end
```

Our game has two methods so far, initialize() and draw(). A third Gosu method, update(), steps forward through the frames of our game, making our ruby move.

Update Means Animate

The update() method is where we move our objects, and handle many user actions such as mouse clicks and key presses. When we run our Gosu project, the initialize() method runs once, then the update() and draw() methods run over and over, until our game is done. As long as the computer can keep up, these methods run sixty times per second.

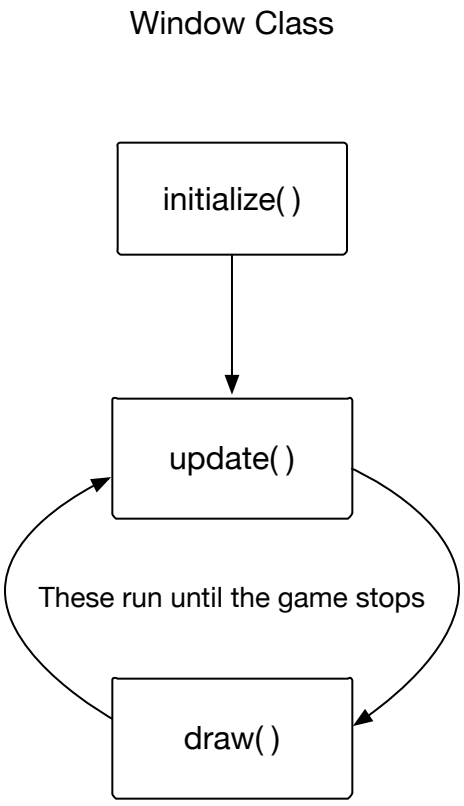


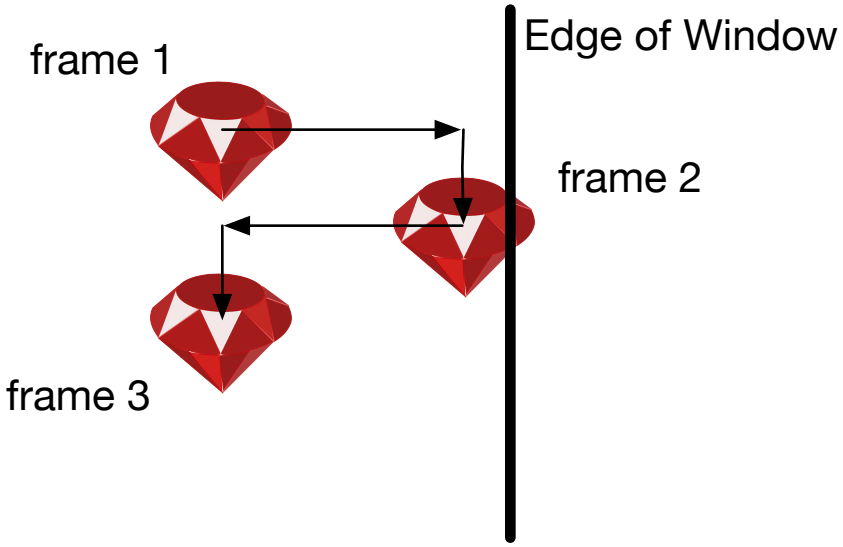
Figure 2—The Gosu Run Loop

In update(), we change the values of @x and @y. We add @velocity_x to @x, and @velocity_y to @y. Add the update method after the initialize() method and before the draw() method.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def update
  @x += @velocity_x
  @y += @velocity_y
end
```

When we run the game now, the ruby moves, but it moves right off the screen. We want the ruby to bounce off the edges of the screen, which means we have to check in our update() method to see when it gets to an edge. When it does, we reverse its velocity in that direction. If it hits the right edge, we want it to keep moving down, but stop moving to the right and start moving to the left, as shown in the following figure.



In the figure, the ruby overlaps the edge of the window in frame two. It then reverses direction. Between frame two and frame three, it moves to the left. Once again, the distance the ruby moves in each frame is exaggerated to make the figure clear. Since the ruby only moves a small number of pixels each frame, the ruby doesn't ever overlap the edge of the window very much and the player sees the ruby bounce off the edge of the window.

To make the ruby reverse directions at the right edge, we change @velocity_x from positive to negative. In the update() method, just after we move the ruby, we check to see if the ruby overlaps the window's edge. The ruby is over the edge of the window *if* the value of @y plus half the width of the ruby image is greater than the width of the window. When this happens, we multiply @velocity_x by -1. When the ruby overlaps the left edge, we change @velocity_x from negative to positive in the same way. We change @velocity_y when the ruby reaches the top or the bottom of the screen.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def update
  @x += @velocity_x
  @y += @velocity_y
➤ @velocity_x *= -1 if @x + @width / 2 > 800 || @x - @width / 2 < 0
➤ @velocity_y *= -1 if @y + @height / 2 > 600 || @y - @height / 2 < 0
end
```

When we run the game now, the ruby moves and bounces off the edges. We did it with the Gosu run loop, some instance variables, and a little math. Try

making a few changes. How would you make the ruby move more slowly or more quickly?

Make the Ruby Blink

We want our ruby to be invisible most of the time, and then pop onto our screen for a fraction of a second, during which time we try to whack it with the mouse. As is often the case, to add a feature we start by adding a new instance variable. We create the variable in the initialize() method, use it and change it in the update() method, and check it in the draw() method, since it affects whether or not we draw the ruby.

Our instance variable is called @visible, and is an integer. The ruby is visible and hittable when @visible is positive, and hidden when @visible is negative. When we make it visible, we set its value to 30. Then we decrease it by one every frame. This means that each time the ruby becomes visible, it will remain visible for 30 frames, and then become invisible again. After the ruby has been invisible for ten frames, there is a small chance that the ruby reappears each frame.

To do this in code, we make changes in all three methods. First, inside the initialize() method, create the @visible variable. Inside the method means that it goes after def initialize and before the end that ends the method. Generally, we add it after the other code that we've already put in the method, though often it won't really matter. When the method executes, the code is run in the order you write it.

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
  @velocity_x = 5
  @velocity_y = 5
➤ @visible = 0
end
```

In the update() method, we decrease the value of @visible. If visible is negative and has been for ten frames, there is a small chance (1%) that the ruby becomes visible for 30 frames. Add the highlighted code inside the update() method.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def update
```

```
    @x += @velocity_x
    @y += @velocity_y
    @velocity_x *= -1 if @x + @width / 2 > 800 || @x - @width / 2 < 0
    @velocity_y *= -1 if @y + @height / 2 > 600 || @y - @height / 2 < 0
➤   @visible -= 1
➤   @visible = 30 if @visible < -10 && rand < 0.01
  end
```

In the draw() method, we check to see if @visible is positive, and draw the ruby only if it is. Replace the draw() method with the following.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def draw
  if @visible > 0
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
  end
end
```

Now the ruby blinks in and out as it moves around the screen. Later on we can adjust some of the numerical values we've used in order to balance the game and make it more fun.

Add the Hammer

To whack our ruby, we use a hammer. We draw our hammer image in the window, and have it follow the player's mouse. The player tries to track the motion of the ruby by moving the hammer. Then when the ruby becomes visible ... bam! The player clicks the mouse, and we detect whether the position of the mouse click is close enough to the position of the ruby for a hit.

Drawing the hammer is similar to drawing the ruby. In the initialize() method we make an instance variable for the hammer image, and load the image file.

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
  @velocity_x = 5
  @velocity_y = 5
  @visible = 0
➤   @hammer_image = Gosu::Image.new('hammer.png')
end
```

Make sure you've copied the hammer.png from the book's source code into your WhackARuby folder. If the file is not there, Ruby will let you know with an error message.

Next we draw the hammer image at the position of the mouse. We get the position of the mouse by using two methods that Gosu gives us through the Gosu::Window class, mouse_x() and mouse_y(). These methods return the position of the mouse within the window. In order to draw the hammer roughly centered on the position of the mouse, we draw the image just as we did the ruby image, offset by half the width of the image in the x direction and half the height in the y direction. Put the following in the draw() method.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def draw
  if @visible > 0
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
  end
  @hammer_image.draw(mouse_x - 40, mouse_y - 10, 1)
end
```

When you run the program now, the hammer appears at the mouse location. If you move the mouse outside the window, the hammer doesn't get drawn. When you move the mouse back into the window, you see the hammer again.

Detecting Mouse Clicks

In the update() method, we check to see if the mouse click is *close enough* to the ruby. For now, we make close enough equal to 50 pixels, but we can change that later if it doesn't feel right. We use the built in class method Gosu.distance() to see if our click is less than 50 pixels from the center of the ruby. If it is, we've got a hit. When we hit the ruby, we want our screen to flash green, and when we miss, it should flash red. We use a new instance variable, @hit, to keep track. Normally, @hit is zero. But if we get a hit we change it to 1, and if we miss, we change it to -1. We set @hit in the update() method, and we check it in the draw() method. As with our other instance variables, we create @hit and give it an initial value in 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
  @velocity_x = 5
```

```
    @velocity_y = 5
    @visible = 0
    @hammer_image = Gosu::Image.new('hammer.png')
    @hit = 0
  end
```

To handle the mouse click, we use another special method of Gosu::Window. The button_down() method runs whenever we press any key or click the mouse. Gosu includes support for game pads, so we can also detect game pad button presses. In the button_down() method we first check to see if the button is the left mouse button. If it is we then check to see if the position of the mouse is within 50 pixels of the position of the ruby. If this is also true, we set @hit to 1. If not, we set @hit to -1.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def button_down(id)
  if (id == Gosu::MsLeft)
    if Gosu.distance(mouse_x, mouse_y, @x, @y) < 50 && @visible >= 0
      @hit = 1
    else
      @hit = -1
    end
  end
end
```

In the draw() method, we check the @hit variable. If it is -1 or 1 we set a color, and then fill the screen with that color using the draw_quad() method. The draw_quad() method requires twelve parameters and draws a quadrilateral - a four sided shape. For each of the four corners of the quadrilateral, we provide the x coordinate, the y coordinate, and our color.

The positions of the top left corner is x = 0 y = 0. The other corner are at x = 800 y = 0, x = 800 y = 600, and x = 0 y = 600.

Once we set the color c, we are ready to use draw_quad() to fill the screen with color. After we've drawn the rectangle of color, we set @hit back to zero so it switches back to black the next time the draw() method runs.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def draw

  if @visible > 0
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
  end
  @hammer_image.draw(mouse_x - 40, mouse_y - 10, 1)
  if @hit == 0
    c = Gosu::Color::NONE
  elsif @hit == 1
    c = Gosu::Color::GREEN
```

```
    elsif @hit == -1
      c = Gosu::Color::RED
    end
    draw_quad(0, 0, c, 800, 0, c, 800, 600, c, 0, 600, c)
    @hit = 0
  end
```

Run the game now and try to hit the ruby. When you get a hit, the screen flashes green, and when you miss, the screen flashes red. It's starting to look like a game.

What if it doesn't work?

The draw_quad() method takes a lot of parameters. If we leave just one of them off, or put them in the wrong order, we get an error. See if you can spot this one.

```
draw_quad(0, 0, c, 800, 0, c, 800, 600, c, 0, c)
```

Did you find the problem? One number is left off, near the end. When we run the game, we get an error.

```
/Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:102: in `draw_quad': wrong #
of arguments(11 for 12) (ArgumentError)
  from /Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:102 in `draw'
  from /Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:124 in `<main>'
[Finished in 1.5s with exit code 1]
```

This one is pretty informative. It tells us that the error is on either line 102 or 124, and it tells us the problem. We have the wrong number of arguments, eleven instead of twelve, when we call draw_quad(). We try to fix it by adding another number, at the end, so it looks like this.

```
draw_quad(0, 0, c, 800, 0, c, 800, 600, c, 0, c, 600)
```

Unfortunately, we have a different error now.

```
  /Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:102: in `draw_quad':
  Expected argument 11 of type double, but got
  Gosu::Color #<Gosu::Color:0x007fcf4202da78... (TypeError)
  in SWIG method 'drawQuad'
  from /Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:102: in `draw'
  from /Users/mark/Desktop/WhackARuby/whack_a_ruby.rb:124: in `<main>'
[Finished in 1.3s with exit code 1]
```

The error is still on the same line, but now argument eleven is the wrong type, Gosu::Color instead of double. We need to put the arguments in the correct order, two numbers and a color, in that order, for each of the four corners of the rectangle.

Keep Score

Keeping track of the player's score is one way to make Whack-A-Ruby a game. Players can try and beat their friends' scores, or their own previous score. They can brag about their high score, and post it on social media.

In Whack-A-Ruby, we start with zero points. Every time we whack the ruby, let's add five points, and every time we miss, take away one point.

We draw our score on the screen using another Gosu class, Gosu::Font. To create an instance of Gosu::Font we tell Gosu what size characters we want in our window. The size of the font is the height of the tallest character in pixels. In the initialize() method, we create our @font instance variable, along with another variable to keep track of the score.

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
  @velocity_x = 5
  @velocity_y = 5
  @visible = 0
  @hammer_image = Gosu::Image.new('hammer.png')
  @hit = 0
➤ @font = Gosu::Font.new(30)
➤ @score = 0
end
```

We already check to see whether we hit the ruby in the button_down(id). We can add and subtract from the @score variable here.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def button_down(id)
  if (id == Gosu::MsLeft)
    if Gosu.distance(mouse_x, mouse_y, @x, @y) < 50 && @visible >= 0
      @hit = 1
➤     @score += 5
    else
      @hit = -1
➤     @score -= 1
    end
  end
end
```

Then we draw the score, using the draw() method of Gosu::Font. We pass in the string we want to draw, which in our case is the score. Since the score is an integer, we convert it to a string. We also supply the position of the score on the screen. This goes in the draw() method.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def draw
  if @visible > 0
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
  end
  @hammer_image.draw(mouse_x - 40, mouse_y - 10, 1)
  if @hit == 0
    c = Gosu::Color::NONE
  elsif @hit == 1
    c = Gosu::Color::GREEN
  elsif @hit == -1
    c = Gosu::Color::RED
  end
  draw_quad(0, 0, c, 800, 0, c, 800, 600, c, 0, 600, c)
  @hit = 0
➤ @font.draw(@score.to_s, 700, 20, 2)
end
```

We're getting close to done. But if you play now, you can play forever, and eventually get any score you want.

Set a Time Limit

To make our game competitive, let's give the player 100 seconds to play. That way, a higher score indicates more skill, and not just more patience.

Gosu has a class method Gosu.milliseconds() that returns the number of milliseconds since the game started. We use this method to calculate how many seconds the player has been playing, and subtract that from 100. In the update() method, we figure out how many seconds remain and convert it to a string.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def update
  @x += @velocity_x
  @y += @velocity_y
  @velocity_x *= -1 if @x + @width / 2 > 800 || @x - @width / 2 < 0
  @velocity_y *= -1 if @y + @height / 2 > 600 || @y - @height / 2 < 0
  @visible -= 1
  @visible = 30 if @visible < -10 && rand < 0.01
➤ @time_left = (100 - (Gosu.milliseconds / 1000))
end
```

We can display this on the screen using our @font instance. As long as we want the same font and size, we don't need a new instance of Gosu::Font; we can reuse the one we already have. In the draw() method we draw the @time_left string.

WhackARuby/WhackARuby_1/whack_a_ruby.rb

```
def draw
  if @visible > 0
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
  end
  @hammer_image.draw(mouse_x - 40, mouse_y - 10, 1)
  if @hit == 0
    c = Gosu::Color::NONE
  elsif @hit == 1
    c = Gosu::Color::GREEN
  elsif @hit == -1
    c = Gosu::Color::RED
  end
  draw_quad(0, 0, c, 800, 0, c, 800, 600, c, 0, 600, c)
  @hit = 0
➤ @font.draw(@time_left.to_s, 20, 20, 2)
end
```

When you run the program now, the time counts down from 100. If we play long enough now, the time remaining becomes negative. We need to stop the countdown, and end the game.

Game Over!

For our first game, we want to end it very simply. When the clock reaches zero, let's stop all movement, show the ruby, and write "Game Over" in the center of the screen. We use a boolean instance variable, @playing, to keep track of when the game is over. Create this variable, and set its value to true in the initialize() method.

WhackARuby/WhackARuby_2/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
  @velocity_x = 5
  @velocity_y = 5
  @visible = 0
  @hammer_image = Gosu::Image.new('hammer.png')
  @hit = 0
```

```
    @score = 0
    @font = Gosu::Font.new(30)
➤   @playing = true
  end
```

In the update() method, we check the value of @playing. If its value is true, we do all the things we were doing before. If the value of @playing is false, we don't do anything at all. We also check to see how much time has elapsed using Gosu.milliseconds(). If 100 seconds have elapsed, we set @playing to false.

WhackARuby/WhackARuby_2/whack_a_ruby.rb

```
def update
➤   if @playing
      @x += @velocity_x
      @y += @velocity_y
      @visible -= 1
      @velocity_x *= -1 if @x + @width / 2 > 800 || @x - @width / 2 < 0
      @velocity_y *= -1 if @y + @height / 2 > 600 || @y - @height / 2 < 0
      @visible = 30 if @visible < -10 && rand < 0.01
      @time_left = (100 - ((Gosu.milliseconds - @start_time) / 1000))
➤     @playing = false if @time_left < 0
➤   end
  end
```

In the button_down() method we use the same pattern, and only check to see if we've clicked our mouse when @playing is true.

WhackARuby/WhackARuby_2/whack_a_ruby.rb

```
def button_down(id)
➤   if @playing
      if id == Gosu::MsLeft
        if Gosu.distance(mouse_x, mouse_y, @x, @y) < 50 && @visible >= 0
          @hit = 1
          @score += 5
        else
          @hit = -1
          @score -= 1
        end
      end
➤   end
  end
```

In the draw() method, we add the code that draws the "Game Over" message.

WhackARuby/WhackARuby_2/whack_a_ruby.rb

```
def draw
  if @visible > 0
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
  end
  @hammer_image.draw(mouse_x - 40, mouse_y - 10, 1)
  if @hit == 0
```

```
    c = Gosu::Color::NONE
  elsif @hit == 1
    c = Gosu::Color::GREEN
  elsif @hit == -1
    c = Gosu::Color::RED
  end
  draw_quad(0, 0, c, 800, 0, c, 800, 600, c, 0, 600, c)
  @hit = 0
  @font.draw(@time_left.to_s, 20, 20, 2)
  @font.draw(@score.to_s, 700, 20, 2)
➤ unless @playing
➤   @font.draw('Game Over', 300, 300, 3)
➤   @visible = 20
➤ end
end
```

We're almost done. When our game ends now, we have to quit the game to play again. Instead, let's allow the player to press a key to play again.

Play Again?

Let's add a "Press the Space Bar to Play Again" message, and set up the game to play again. We need to set the score back to zero and give the player a new 100 seconds to play. We add the new message to the screen in the `draw()` method when our game is over.

WhackARuby/WhackARuby_2/whack_a_ruby.rb

```
def draw
  if @visible > 0
    @image.draw(@x - @width / 2, @y - @height / 2, 1)
  end
  @hammer_image.draw(mouse_x - 40, mouse_y - 10, 1)
  if @hit == 0
    c = Gosu::Color::NONE
  elsif @hit == 1
    c = Gosu::Color::GREEN
  elsif @hit == -1
    c = Gosu::Color::RED
  end
  draw_quad(0, 0, c, 800, 0, c, 800, 600, c, 0, 600, c)
  @hit = 0
  @font.draw(@time_left.to_s, 20, 20, 2)
  @font.draw(@score.to_s, 700, 20, 2)
  unless @playing
    @font.draw('Game Over', 300, 300, 3)
➤   @font.draw('Press the Space Bar to Play Again', 175, 350, 3)
    @visible = 20
end
```

When we calculate the time, we've been using the Gosu.milliseconds() method assuming the start time of the game is zero. When we restart the game, Gosu.milliseconds() no longer tells us how long we've been playing. To adjust for this, we add a new instance variable called @start_time that keeps track of when the current game started. In the initialize() method, we create that variable and set it to zero.

WhackARuby/WhackARuby_2/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
  @velocity_x = 5
  @velocity_y = 5
  @visible = 0
  @hammer_image = Gosu::Image.new('hammer.png')
  @hit = 0
  @score = 0
  @font = Gosu::Font.new(30)
  @playing = true
➤ @start_time = 0

end
```

Then we change both references to Gosu.milliseconds() in the update() method to subtract the start time. That difference, Gosu.milliseconds - @start_time is the elapsed time of the current game.

WhackARuby/WhackARuby_3/whack_a_ruby.rb

```
def update
  if @playing
    @x += @velocity_x
    @y += @velocity_y
    @visible -= 1
➤   @time_left = (100 - ((Gosu.milliseconds - @start_time) / 1000))
➤   @playing = false if @time_left < 0
    @velocity_x *= -1 if @x + @width/2 > 800 || @x - @width / 2 < 0
    @velocity_y *= -1 if @y + @height/2 > 600 || @y - @height / 2 < 0
    @visible = 30 if @visible < -10 and rand < 0.01
  end
end
```

Run your program now and make sure that it behaves the same as it did before.

Next, we change the button_down() method so that it checks for the space bar, but only when @playing is false. When the user presses the space bar we reset the game, and set @start_time to the current value of Gosu.milliseconds.

WhackARuby/WhackARuby_3/whack_a_ruby.rb

```
def button_down(id)
  if @playing
    if (id == Gosu::MsLeft)
      if Gosu.distance(mouse_x, mouse_y, @x, @y) < 50 && @visible >= 0
        @hit = 1
        @score += 5
      else
        @hit = -1
        @score -= 1
      end
    end
  else
    if (id == Gosu::KbSpace)
      @playing = true
      @visible = -10
      @start_time = Gosu.milliseconds
      @score = 0
    end
  end
end
```

And our first game is complete. Now you should play for awhile. And show your friends the game you've made. While you're playing, think about what you find annoying, and what would make the game more challenging. One of the best things about the games you write yourself is that you can change everything about them. Our game is complete, but it may not be *done*.

Make it Your Own

While we made the Whack-A-Ruby game, we made many choices. How fast the Ruby moves, how long it stays visible, with what chance it reappears. Is 100 seconds too long, or too short? The game might be too hard, or too easy. Change the game until you're happy with it. Make it your game! Here is a list of some changes you could make, in no particular order.

Make the ruby appear for a shorter or longer time.

Figure out which number determines how many frames the ruby is visible, and change the number.

Make the ruby appear more or less often.

Figure out which number determines how often the ruby becomes visible, and change it.

Change the images to something else.

Add the images to the game folder. Look in the code for file names, and pay attention to image sizes.

Add another ruby bouncing around that you can click.

You need duplicates for a whole bunch of instance variables. Maybe name them `@x_2`, `@velocity_y_2`, etc. This one is challenging!

Add another thing that bounces around that you can't click. Maybe an emerald?

You need another image, and some new instance variables.

Make "Game Over" appear in a larger font.

Make another instance variable for a different sized font.

What's Next

In Whack-A-Ruby, our goal was to learn how Gosu works, and we made a working game. We put all our code in the `WhackARuby` class, and it started to get a little long, even for our simple game. For our next game, we'll separate our code out into more classes, one for each type of object in our game. This will prove to be a powerful tool, especially when we want to have lots of objects on the screen at once.

CHAPTER 3

Creating a Sprite-based Game

Many video games, from old classics like “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 1970’s video games systems like 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 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 done 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 space bar 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 we'll learn to:

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

In the next chapter, we'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 last, we'll learn how to add sound effects to our game. When we're done 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.

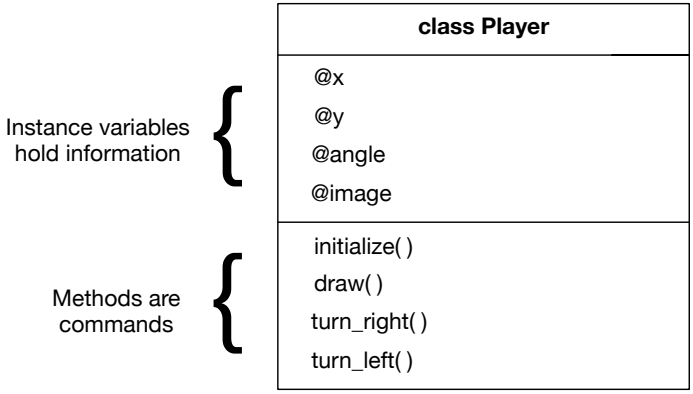
SectorFive/SectorFive_starter/sector_five.rb

```
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 we'll create, move, and draw our sprites. Run this program to make sure all is well. An empty window will open on your screen. The first sprite we'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*, that are commands we 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 to want it to follow. The player ship sprite stores an image and a position. Since 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 its included in the diagram because without it, there won't even be a ship.

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

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

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

In the initialize() method of the Player class, we create and set some instance variables. We set the position of the ship, just as we did for the ruby in Whack-A-Ruby. We 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 we'll use later to let the ship interact with edges of the window.

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, we 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 we send as the first two parameters.

Back in SectorFive, we can now use these methods to add and draw the player. First, we include our new code in the sector_five.rb file, using require_relative. This line goes just after require gosu and before our 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, we create the ship.

SectorFive/SectorFive_1/sector_five.rb

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

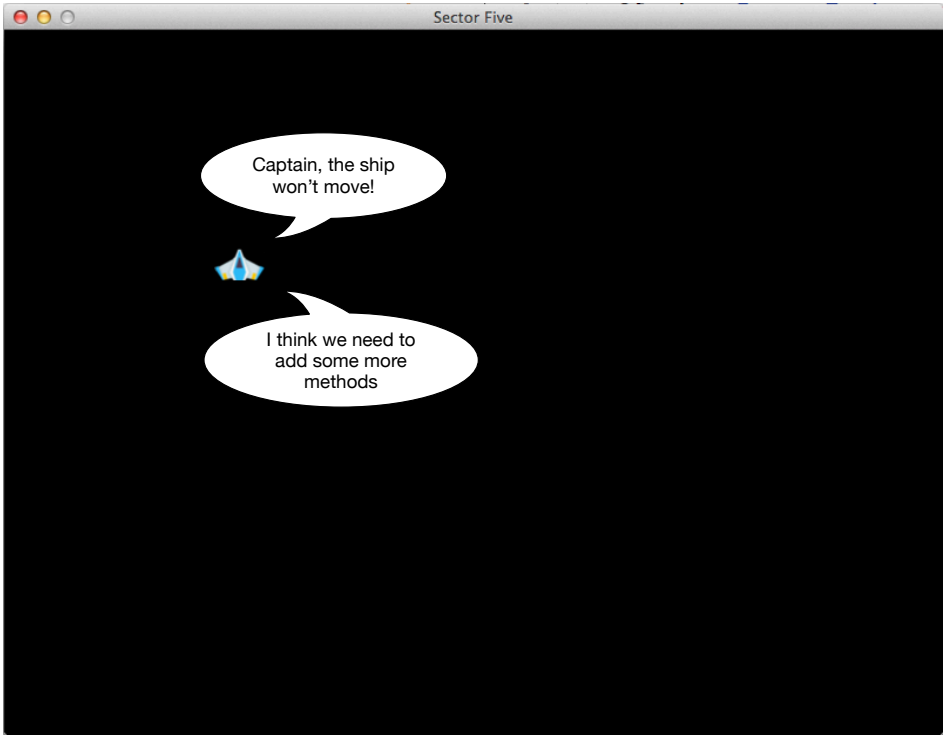
Notice that we send self as a parameter to the initialize() method. The initialize() method of Player takes the window as an argument. In the SectorFive class, the window is self, so that's what we pass to Player.new(). We'll do the same thing each time we create a new sprite.

The SectorFive class now gets a draw() method, where we'll eventually draw all the sprites in the game. For now, we 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, its likely you've got the player.rb file in the front tab or window of your editor. When we run the game, our ship appears, 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 by pressing keys on the keyboard.

What if it Doesn't Work?

Now that our program consists of more than one file, errors that we make can be in either file. Maybe while following the tutorial we accidentally put the code for Player.draw()

in sector_five.rb. Hopefully we'd would see that we had two methods named draw() in SectorFive, but what if we missed this? When we run the game this way, we 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 key presses exactly the same as mouse clicks, and calls all of them buttons. We use the left arrow, the right arrow, and the up arrow to move the ship. The ship moves forward by firing its engines; our ship doesn't have any way to fire the engines backward. If we stop firing the engines, the ship coasts gradually to a stop.

Should we Make it Realistic?

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 our 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 she wants. And that is fun!

Turn the Ship

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

SectorFive/SectorFive_1/player.rb

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

```
end
```

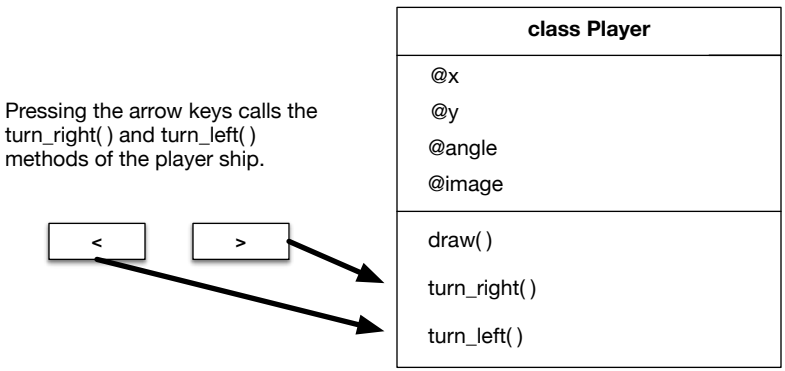
Next we change the update() method of the SectorFive class to call the turn_right() and turn_left() when we press the arrow keys. Gosu gives us access to the keys, just as it gives us access to the mouse. Many of them you might be able to figure out, like Gosu::KbLeft for the left arrow key, or Gosu::KbA 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.

We check to see if each key is pressed with a method called button_down?() that is part of the Gosu::Window class. What do you think will happen if both keys are pressed?

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
```

Look at what we just did to add a behavior to the player ship. In the Player class, we wrote two methods, turn_right() and turn_left(). Then in the update() method of SectorFive, we 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 its time to get our ship moving forward.

button_down vs. button_down?

In WhackARuby, we 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, we use the button_down?() method, which is a different method. The question mark is part of the method name. When should you use each?

button_down?()

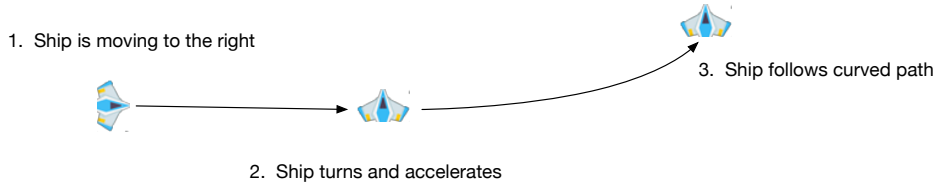
Use this method when holding the button down should do something over and over. Put it in the update() method, inside a *conditional* statement. We use this method to turn the ship; if we hold the arrow key down the ship keeps turning.

button_down(id)

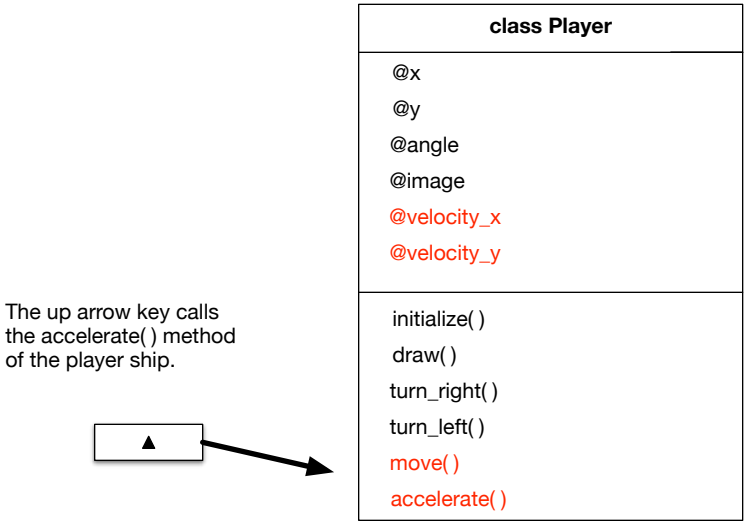
When you want the press to do something, and then not do it again until you release the button and press it again, use button_down(id). We used this method to whack the ruby, and we'll use it to fire bullets. Each key press will fire one bullet, and holding down the key won't do anything beyond the initial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

Make the Ship Accelerate

When we press the forward arrow, we want the ship to *accelerate*. Accelerate means to change the velocity. If our ship is sitting still and we press the forward arrow, it moves in the direction it is pointing, speeding up as it goes. If we 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, we need to add a few new variables, and a few new methods to the Player class. The variables will keep track of the velocity of the ship. One method of the SectorFive class, accelerate(), will be called when we hold down the up arrow key. Another one, move() will get called every frame, so the ship keeps moving even when we're not pressing a key.

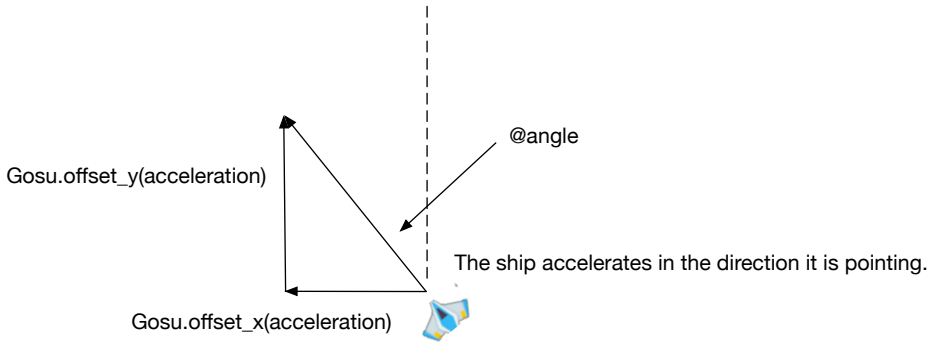


Add the two new variables, and set them to zero, in the initialize() method of the Player class.

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, we 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 us.



The offset_x() method takes the angle and an amount as arguments, and returns the amount in the x direction, either positive or negative. We could do this ourselves using a little trigonometry, but since games make use of these calculations so often, Gosu provides them for convenience. We 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
```

We 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 we also slow the ship down, 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 our move() and accelerate() methods are ready, we call them in the update() method of the SectorFive class. The ship moves every frame, and accelerates whenever we 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
```

Since we want the ship to accerate continuously when the arrow key is held down, we use the button_down?() method. Run the game now, and move the ship around. See if 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.

We've added the ship to the window, and we can move it around with the arrow keys. In the SectorFive class, we detect button presses, and those button presses call methods of the @player object, to tell it what to do. The following

diagram 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.

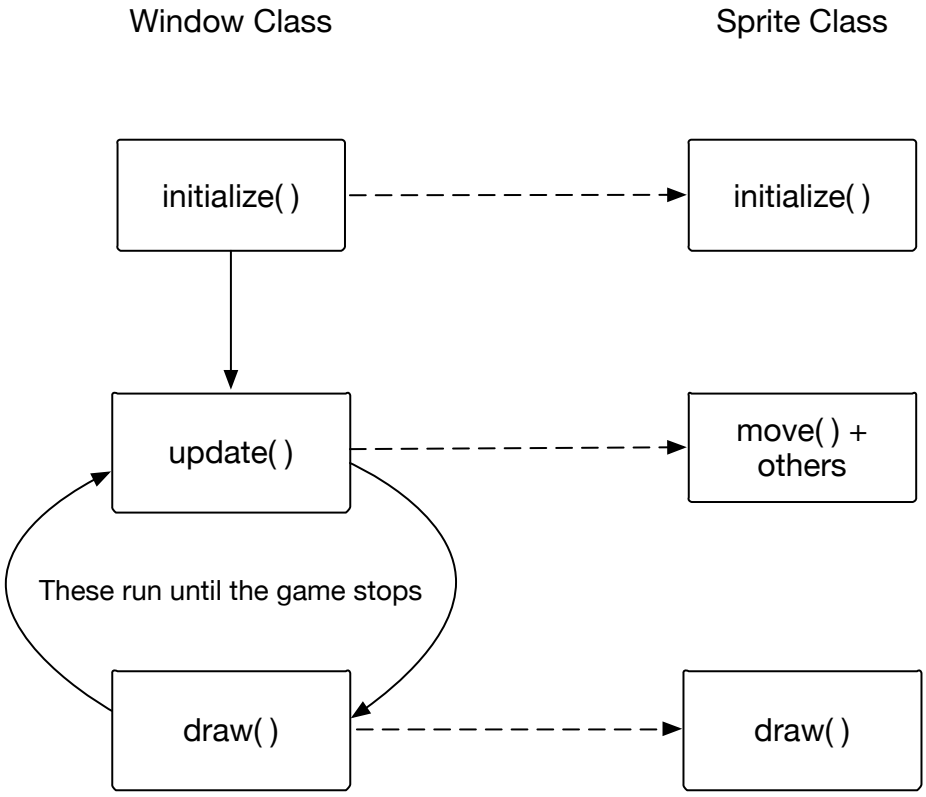


Figure 3—The Gosu Run Loop with a Sprite

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

Use Constants to Adjust your Game

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

and classes. Our three constants are named ROTATION_SPEED, ACCELERATION, and FRICTION.

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

SectorFive/SectorFive_2/player.rb

```
class Player
  ROTATION_SPEED = 3
```

Then, in the turn_right() and turn_left() methods, we replace the number 3 with the constant we'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 we can change the value of ROTATION_SPEED in one place to adjust our 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
```

We 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 the FRICTION replaces the number 0.9 in the move() of Player.

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, we 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

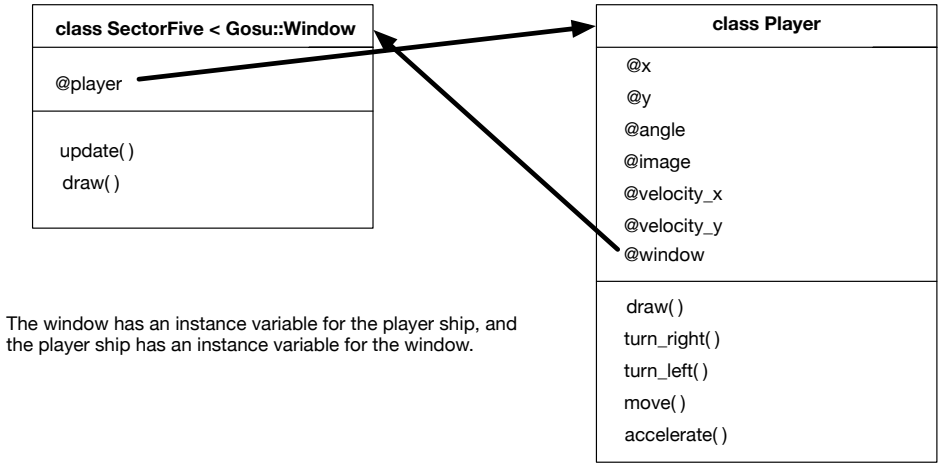
```
class SectorFive < Gosu::Window
  WIDTH = 800
  HEIGHT = 600
  def initialize
    super(WIDTH,HEIGHT)
    self.caption = 'Sector Five'
    @player = Player.new(self)
  end
```

By using constants, we get code that is a little easier to understand, and a little easier to adjust. Now, if we want to adjust the rotation speed, we only have to change it in one place, and 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 its 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 8, *Making a Side-Scrolling Game*, on page 163. 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 edges 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.

In order for the player ship to stop at the edges, it needs to know where the edges are. We'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 we have @player, when it is created, save the *reference* to the window object in an instance variable called @window.



The ship reaches the edge when its center gets within a distance of the edge equal to the radius of the ship. So we 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 us use our @window reference to *get* the width and height of the window. These methods are called width() and height(). We 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, we move it back to the edge and set its velocity in that direction to zero. The player ship can still go off the top of the window. In Chapter 5, *Adding Scenes and Sounds*, on page 81 we'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
```

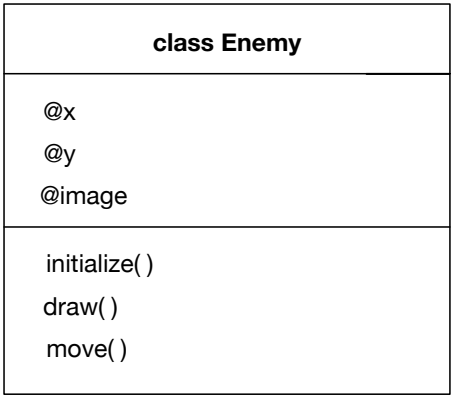
```
    @velocity_y *= FRICTION
➤   if @x > @window.width - @radius
➤     @velocity_x = 0
➤     @x = @window.width - @radius
➤   end
➤   if @x < @radius
➤     @velocity_x = 0
➤     @x = @radius
➤   end
➤   if @y > @window.height - @radius
➤     @velocity_y = 0
➤     @y = @window.height - @radius
➤  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

Run the game now, and the player ship stops at the left, right, and bottom borders of the window. Next we add enemy ships, falling from above.

Make an Enemy

The second kind of sprite we add to our game is the enemy ship. Enemy ships appear at the top of the screen, at a random horizonal position, and fall straight down. We will need enough of them so that it's a challenge to shoot them all before they reach the bottom, but to write and test our enemy ship class, we'll first make just one.

The Enemy class is simpler than the Player class. Enemy ships only move down, and are controlled by the computer. They don't respond directly to player actions. The Enemy class has initialize(), move(), and draw() methods as well as instance variables for its position and image. Here is the diagram for the Enemy class.



Make sure the enemy.png image is in your images folder. Then make a new file, enemy.rb, for the Enemy class in your SectorFive folder.

In the initialize() method of the Enemy class, we set @y to zero, which is the top of the window. We set @x to a random number. Since @x is going to be the center of our enemy ship, and we want the whole enemy to fit in the window, the horizontal position should be a random number with a minimum value equal to the radius of the enemy, and a maximum value equal to the width of the window minus the radius.

SectorFive/SectorFive_2/enemy.rb

```
class Enemy
  def initialize(window)
    @radius = 20
    @x = rand(window.width - 2 * @radius) + @radius
    @y = 0
    @image = Gosu::Image.new('images/enemy.png')
  end
end
```

We can make a random number between any two values by passing the difference between the minimum and maximum values to the rand() method and then adding the minimum value.

In the move() method, we increase @y by the speed of the ship, which we make a constant called SPEED. We add the SPEED declaration just after class Enemy.

SectorFive/SectorFive_2/enemy.rb

```
class Enemy
  SPEED = 4
  def initialize(window)
    @radius = 20
    @x = rand(window.width - 2 * @radius) + @radius
    @y = 0
    @image = Gosu::Image.new('images/enemy.png')
  end
```

Then we use the SPEED constant in the move() method. The enemy ship moves only in the y direction.

SectorFive/SectorFive_2/enemy.rb

```
def move
  @y += SPEED
end
```

In the draw() method of Enemy, we offset the image by @radius to center the image on @x and @y.

SectorFive/SectorFive_2/enemy.rb

```
def draw
```

```
  @image.draw(@x - @radius, @y - @radius, 1)
end
```

Once the Enemy class is set up, we can create, move and draw a single enemy with just a few lines of code in the SectorFive class. We need to include the code for the Enemy class in sector_five.rb just after we add the Player class.

SectorFive/SectorFive_2/sector_five.rb

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

Add one new line to the initialize() method that creates a single enemy and stores it in @enemy.

SectorFive/SectorFive_2/sector_five.rb

```
def initialize
  super(WIDTH,HEIGHT)
  self.caption = 'Sector Five'
  @player = Player.new(self)
➤ @enemy = Enemy.new(self)
end
```

We move the enemy in the update() method, with another line of code.

SectorFive/SectorFive_2/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
➤ @enemy.move
end
```

Finally, we draw the enemy ship in the draw() method of the SectorFive class.

SectorFive/SectorFive_2/sector_five.rb

```
def draw
  @player.draw
➤ @enemy.draw
end
```

When you run the game now, a single enemy falls from the top of the screen, and off the bottom.

Make it Your Own

Our game isn't done, but this is a good time to stop and do a few experiments. *Save a copy of your entire game folder before you make any big changes*. Then try one or more of the following exercises. Or make something up yourself.

Change the constants.

The contants in Player, especially ACCELERATION and FRICTION control how the player ship moves. Even small changes in these can make a big difference in how maneuvering the ship feels. Try changing FRICTION to 1. Try changing it to 0. What happens? Can you explain why, by looking at the code? Try adjusting those constants to make controlling the ship challenging and fun.

Change the enemy ship movement.

Right now the enemy ship moves straight down. Try having it move at an angle. What should it do if it reaches an edge? All the changes should be in the Enemy class.

Make the player ship bounce.

The player ship hits the edges and the bottom now and stops. But you could make the player ship bounce off the edges instead, by changing a few lines of code in the move() method of the Player class. When the ruby in Whack-A-Ruby hit the sides of the window, it bounced off. Go back and look at that code if you've forgotten how we did that.

Make several enemies.

In the next chapter we'll be making lots of enemies, but you could increase the number to two or three right now. Make more instances of the Enemy class to create more enemies.

What's Next

In this chapter, we learned to use sprite classes to make player and enemy ships appear on the screen. We used the keyboard to move the player ship, and made one enemy ship fall from above. But one enemy is not enough. In the next chapter, we'll learn how to turn one enemy into dozens, or even hundreds. We'll also make bullets, and detect collisions between bullets and enemy ships. And then we'll make the enemies explode in balls of fire.

CHAPTER 4

Managing Lots of Sprites

Our Sector Five game now has only one enemy ship. In this chapter we add more, so a constant stream of enemies falls from above. Then we let our player ship shoot them with bullets, and when bullets hit enemies they explode. This means we'll have dozens of enemy ships, bullets and explosions all in our window at once. We need a way to deal with tons of sprites in our game - adding them, deleting them, and managing all their interactions.

We'll learn how to create and handle all these sprites with one great tool, the array. An array holds a list of objects, which in our game are sprites. When we need a new sprite, such as a new enemy or bullet, we add it to the array, and when one is destroyed, we remove it. By *iterating* through our arrays, we move and draw all these sprites.

Sector Five starts without any enemies or bullets. Enemies appear at random times and fall from the top, while bullets are created by the player. Once both enemies and bullets are in our window we check for collisions between them. When a bullet hits an enemy, we remove both and add an explosion. Explosions are a little different from our other sprites. The explosion image is animated, like a little movie that plays within our window.

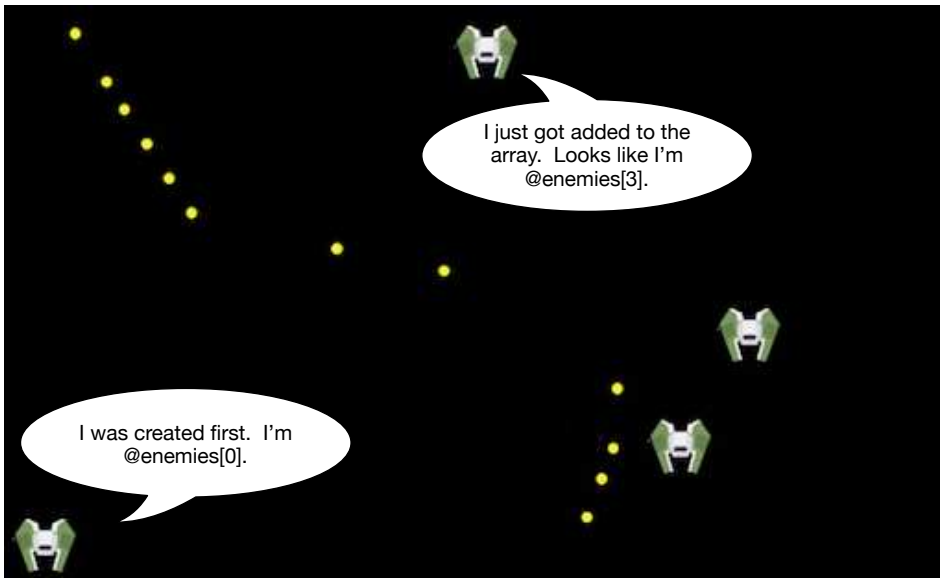
In this chapter we'll:

- Use arrays to hold many sprites of the same kind.
- Iterate through an array with the `each()` method to move and draw all the sprites.
- Detect collisions between sprites.
- Create explosion sprites with animated images.

When we're done our sprite game toolkit will be getting pretty full. We'll add some finishing touches in the next chapter.

Make more Enemies

Having a single enemy that falls from the top of the window to the bottom was a good way to write and test the `Enemy` class. Now let's replace that single enemy with a stream of enemies. We store all the enemies that are in the game using a single Ruby array called `@enemies`.



The Array class has lots and lots of methods[1], but we use just three of them in Sector Five.

push()

The push() method adds a new object to the array. When its time to add a new enemy, we write:

```
@enemies.push Enemy.new(self)
```

and one more enemy is added to the array. It's up to us to make sure we don't add anything to the array that's not an enemy.

delete()

When its time to remove an enemy, we use the delete() method. If the enemy we want to remove is stored in the variable enemy_to_remove, we say:

```
@enemies.delete enemy_to_remove
```

and poof! One less enemy.

each()

When we want to do something to *each* of the enemies in the array, such as move them or check to see if they are touching a bullet, we use the each() method to loop through the enemies.

1. You can see all the methods of the Array class by looking in the Ruby documentation at http://ruby-doc.org//core-2.0.0/Array.html.

```
@enemies.each do |enemy|
   #do something to enemy here
end
```

We use the each() method to move our sprites, draw them, and to test for collisions between them.

To use an array to hold our enemies, we don't change anything about the Enemy class. We just change how we create, move, and draw enemies in the SectorFive class.

Create an Empty Array

In the initialize() method of the SectorFive class, we wrote one line to create a single enemy.

SectorFive/SectorFive_2/sector_five.rb

```
def initialize
  super(WIDTH,HEIGHT)
  self.caption = 'Sector Five'
  @player = Player.new(self)
➤ @enemy = Enemy.new(self)
end
```

Now let's replace that line with one that creates an empty array. An array in Ruby can be written as a list of objects, separated by commas, and surrounded by square brackets. Square brackets with nothing inside denote an empty array.

SectorFive/SectorFive_3/sector_five.rb

```
def initialize
  super(WIDTH, HEIGHT)
  self.caption = 'Sector Five'
  @player = Player.new(self)
➤ @enemies = []
end
```

We add new enemies to the array in the update() method. The initialize() method only runs once, so if we wanted to create our enemies there we'd have to decide in advance how many to create.

Ruby Refresher: Arrays

Arrays are a key part of Sector Five, and you'll find them very helpful in almost any kind of game you want to write. If you're a little rusty with Ruby arrays, or learned arrays in another language and could use a little help with Ruby syntax, some of the resources mentioned in Chapter 1, *Get Ready*, on page 1 might give you some help.

In particular, chapter eight of Chris Pine's book *Learn to Program*, called *Arrays and Iterators*, is a great introduction.

Add A Stream of Enemies

Enemies appear from above, and fall down through the window. If we added an enemy every update, our window would be full of enemies. Instead, let's have a small chance of a new enemy appearing each frame. This number helps determine how difficult the game is and we might want to change it later, so we make it a constant of the SectorFive class, called ENEMY_FREQUENCY. We set it initially to 0.05.

SectorFive/SectorFive_3/sector_five.rb

```
class SectorFive < Gosu::Window
  WIDTH = 800
  HEIGHT = 600
➤   ENEMY_FREQUENCY = 0.05
```

In the update() method, we create a random number each frame. It that number is less than ENEMY_FREQUENCY, a new enemy is added to the array using the push() method. Put this code at the beginning of the update() method.

SectorFive/SectorFive_3/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
➤   if rand < ENEMY_FREQUENCY
➤     @enemies.push Enemy.new(self)
➤   end
end
```

When used without an argument, the rand() method returns a value between zero and one. If ENEMY_FREQUENCY is 0.05, we add an enemy about one frame in twenty. Since there are sixty frames per second, we average about three enemies per second, which should be enough to cause our player some serious trouble.

Move All the Enemies

When the update() method runs, our @enemies array might have one enemy, or zero enemies, or forty-seven enemies. What we want is for each of them to

move down the window. First, in the update() of SectorFive, we remove the single line of code that moved one enemy.

SectorFive/SectorFive_2/sector_five.rb

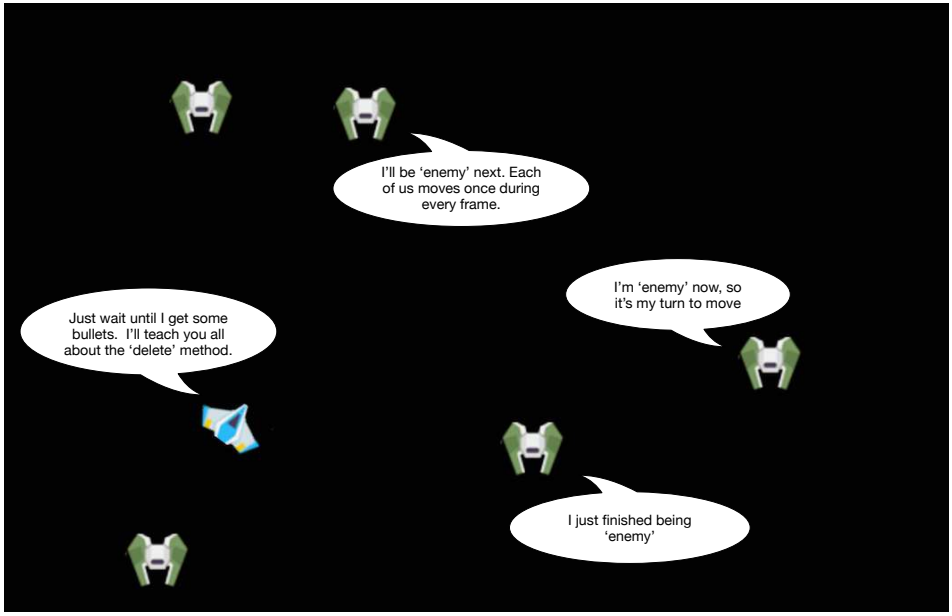
```
def update
  @player.turn_left if button_down?(Gosu::KbLeft)
  @player.turn_right if button_down?(Gosu::KbRight)
  @player.accelerate if button_down?(Gosu::KbUp)
  @player.move
➤ @enemy.move
end
```

We replace it by three lines of code that tells each member of the @enemies array to move. We do this using the each() method of the Array class. When we use the each() method we create a temporary variable, in this case called enemy.

SectorFive/SectorFive_3/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
  if rand < ENEMY_FREQUENCY
    @enemies.push Enemy.new(self)
  end
➤ @enemies.each do |enemy|
➤   enemy.move
➤ end
end
```

This is a code pattern we'll be using several times in this game. Using the each() method of the Array class , we iterate through the array. Each of the elements of the array gets a turn being enemy, and each has its move() method called.



If there are no enemies, nothing happens. If there are forty enemies, the `move()` method is called on each one.

Draw All the Enemies

To draw all the enemies, we do pretty much the same thing we did to move all the enemies. In the `draw()` method of the `SectorFive` class, remove the single line of code that draws an enemy.

SectorFive/SectorFive_2/sector_five.rb

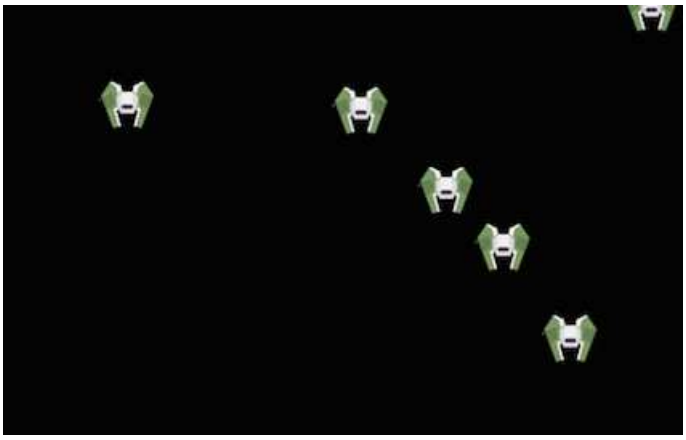
```
def draw
  @player.draw
➤   @enemy.draw
end
```

Replace that one line with three lines that use the `each()` method to draw all the enemies.

SectorFive/SectorFive_3/sector_five.rb

```
def draw
  @player.draw
➤   @enemies.each do |enemy|
➤     enemy.draw
➤   end
end
```

When you run the game now, enemies fall from the top of the window to the bottom.



Try adjusting the ENEMY_FREQUENCY constant, to see what happens. Make it equal to 1, and see what it looks like if we add an enemy every frame. By changing this number we can get anything from an occasional enemy to a constant stream. We can't hurt the enemies, so let's give the player some bullets, and let the mayhem begin.

What if it Doesn't Work?

The each() method makes arrays very useful to us in games, since we don't need to know in advance how many objects are in the array. Errors involving each() will crop up, if only because we use it so often.

The following output was created by replacing enemy.draw with @enemies.draw in the draw() method of the SectorFive class.

```
/Users/mark/Desktop/SectorFive_3/sector_five.rb:54: in `block in draw':
undefined method `draw' for #<Array:0x007fd022803e60> (NoMethodError)
  from /Users/mark/Desktop/SectorFive_3/sector_five.rb:53: in `each'
  from /Users/mark/Desktop/SectorFive_3/sector_five.rb:53: in `draw'
  from /Users/mark/Desktop/SectorFive_3/sector_five.rb:62: in `<main>'
[Finished in 1.5s with exit code 1]
```

The output reminds us that @enemies is an array, not an instance of the Enemy class, and so it doesn't have a draw() method. When we use the each() method to go through the array, the variable enemy becomes each of the elements of the @enemies array in turn, and those elements each do have a draw() method.

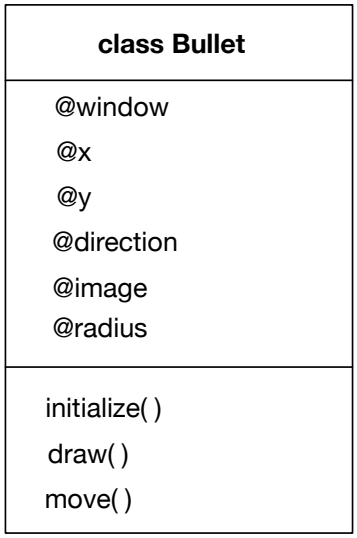
Fire Bullets

In order to destroy some enemies, the player shoots them with bullets. Bullets are another kind of sprite, so we make a class, Bullet, to describe them. When the player presses the space bar, a bullet is created. It travels in the direction the ship was pointing when it was fired, until it hits an enemy, or flies out of

the window. When a bullet hits an enemy ship, both are destroyed in a ball of flame.

The Bullet Class

We've already made two sprite classes, so you are probably getting the hang of it. When we make a bullet, we tell it where to start, and what direction to go. Once we've made a bullet, it should keep moving in that direction until it hits an enemy or flies out of the window. Here is a diagram for the Bullet class.



Start by creating a bullet.rb file, and saving it in the same folder as sector_five.rb.

When we create a bullet, it has the same the position and angle as the player ship at the time the space bar is pressed. So the initialize() method of the Bullet class takes these values as arguments, and then saves them in instance variables.

SectorFive/SectorFive_3/bullet.rb

```
class Bullet
  def initialize(window, x, y, angle)
    @x = x
    @y = y
    @direction = angle
    @image = Gosu::Image.new('images/bullet.png')
    @radius = 3
    @window = window
  end
end
```

The speed of the bullet is a constant, so at the top of the Bullet class, before the initialize() method let's set it to 5.

SectorFive/SectorFive_3/bullet.rb

```
class Bullet
➤ SPEED = 5
```

Once a bullet is fired, we move it each frame of the game. The move() method of the Bullet class uses the Gosu.offet_x() and Gosu.offset_y() methods, to change @x and @y.

SectorFive/SectorFive_3/bullet.rb

```
def move
    @x += Gosu.offset_x(@direction, SPEED)
    @y += Gosu.offset_y(@direction, SPEED)
end
```

In the draw() method of the Bullet class, we make sure the image is centered on @x, @y, so it appears in the middle of the ship when we shoot.

SectorFive/SectorFive_3/bullet.rb

```
def draw
  @image.draw(@x - @radius, @y - @radius, 1)
end
```

Run the project now. You won't see any bullets, since we haven't actually made any yet. But if you have errors in the bullet.rb file, Ruby will tell you, and you can fix them before we move on.

Add a Bullet at the Player Location

To use the Bullet class in our game, we require our new file in sector_five.rb, right after the files for player and enemy.

SectorFive/SectorFive_3/sector_five.rb

```
require 'gosu'
require_relative 'player'
require_relative 'enemy'
➤ require_relative 'bullet'
```

As we did for the enemies, make an empty array in the initialize() method of the SectorFive class.

SectorFive/SectorFive_3/sector_five.rb

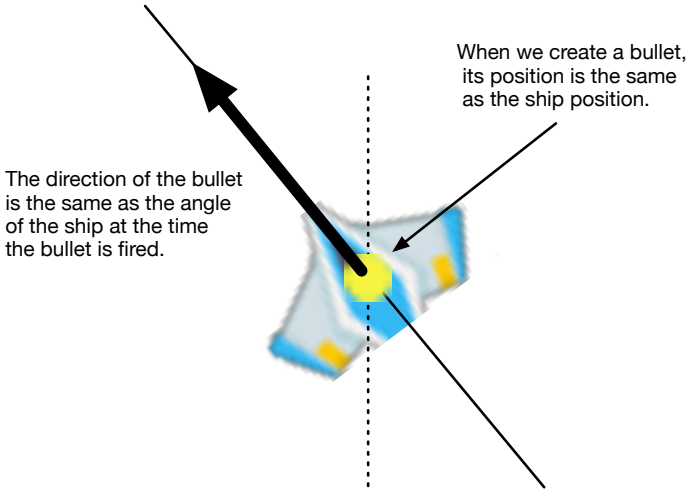
```
def initialize
  super(WIDTH, HEIGHT)
  self.caption = 'Sector Five'
  @player = Player.new(self)
  @enemies = []
➤ @bullets = []
end
```

Since we want one bullet per press of the space bar, we use the button_down() to create the bullets. We need to *get* the position and direction of the player ship in order to create a bullet. Ruby has a helper built in for getting instance variables, attr_reader. At the top of the Player class, add a single line of code.

SectorFive/SectorFive_3/player.rb

```
class Player
  ROTATION_SPEED = 3
  ACCELERATION = 2
  FRICTION = 0.9
  attr_reader :x, :y, :angle, :radius
```

This line is a shortcut that creates four methods x(), y(), angle(), and radius(). Each of these methods returns the value of the appropriate instance variable. We use them in the SectorFive class to create the bullets.



In the button_down(id) method, we create a bullet by passing in the position and angle of the player ship as arguments to the initialize() method. Using button_down(id)means our player gets one bullet per button press, and will have to release the space bar and press again to get another.

SectorFive/SectorFive_3/sector_five.rb

```
def button_down(id)
    if id == Gosu::KbSpace
      @bullets.push Bullet.new(self, @player.x, @player.y, @player.angle)
    end
end
```

The bullets are created right in the middle of the player ship.

Challenge: Move and Draw the Bullets

Before you go on, see if you can move and draw all the bullets. All the code you add should be in the SectorFive class, in the move() and draw() methods. You have enough tools in your toolbox to do it; you already move and draw all the enemies. When you're done, you should be able to run the game and shoot bullets. If you are successful you can skip the next section.

Move and Draw the Bullets

To move the bullets, we use the each() method of the array, sending the move() message to each bullet. Put this in the update() method.

SectorFive/SectorFive_3/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
  if rand < ENEMY_FREQUENCY
    @enemies.push Enemy.new(self)
  end
  @enemies.each do |enemy|
    enemy.move
  end
  @bullets.each do |bullet|
    bullet.move
  end
end
```

To draw the bullets, add the following to the draw() method of the SectorFive class.

SectorFive/SectorFive_3/sector_five.rb

```
def draw
  @player.draw
  @enemies.each do |enemy|
    enemy.draw
  end
  @bullets.each do |bullet|
    bullet.draw
  end
end
```

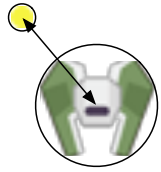
When we run the program now, we can now fly around and shoot bullets. The bullets fly in a straight line, right out of the window, passing right through the enemies. We need to fix that.

Handle Collisions

We have a bunch of enemies falling down, and bullets moving in all directions. Now we need to know when a bullet hits an enemy. This process, of figuring out when two object touch each other, is called *collision detection*.

In Sector Five we're going to use a very simple algorithm for detecting collisions. We pretend our objects are circular. Our bullets *are* circular, and while our enemies are not precisely circular they are pretty close. Even if its not perfect it will *look* like the enemies explode when our bullets hit them. And that is what we really want.

In order to figure out if two circular objects overlap, we find the distance between their centers. If that distance is small enough, there is a collision. The threshold distance is the radius of one, plus the radius of the other.



distance > enemy.radius + bullet.radius
no collision

distance < enemy.radius + bullet.radius
collision

To check for a collision between one bullet and and one enemy, we need to know the position and radius of each. We use the attr_reader helper for this; we add a line in the Bullet class, right after the class definition and before the first method.

SectorFive/SectorFive_3/bullet.rb

```
class Bullet
SPEED = 5
attr_reader :x, :y, :radius
```

Then we add the same attr_reader helper to the Enemy class.

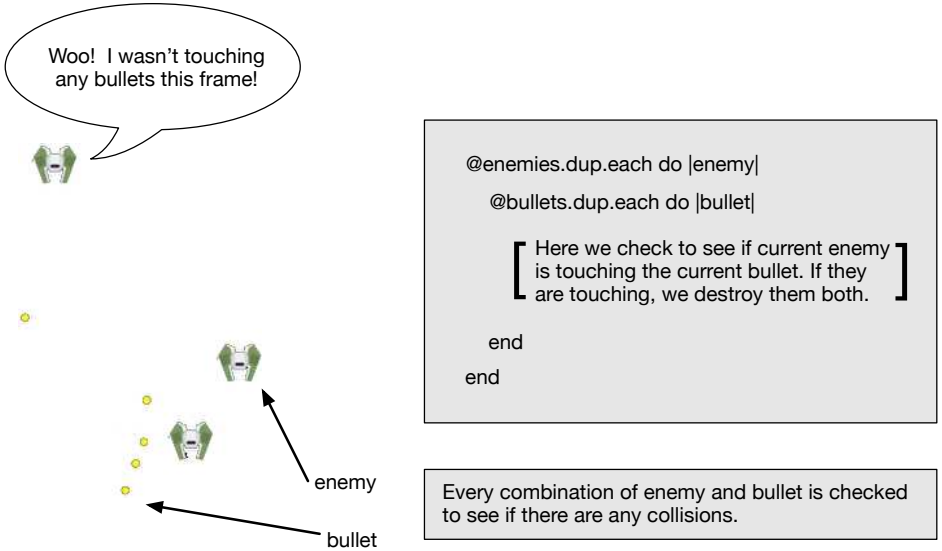
SectorFive/SectorFive_3/enemy.rb

```
class Enemy
  SPEED = 4
  attr_reader :x, :y, :radius
```

To tell if any bullets are touching any enemies, we need to check each enemy against each bullet. For instance if there were twelve bullets and ten enemies in the window, that would be 120 checks. But with the each() method we can make short work of this task.

This is a little different than using the each() method to move all the enemies or all the bullets. In this case we plan to *change* the arrays we are iterating through by removing objects from them. In order to do this, we create copies of the arrays with Ruby's dup() method, and then iterate though those. That ensures that each() checks every single bullet, and every single enemy.

We start by looping through the enemies. For each enemy, we loop through the bullets, again using the each() method. We check all these combinations to see if they are touching, and if they are, we remove both bullet and enemy.



We use the Gosu.distance() method here, just as we did in WhackARuby. It calculates the distance between any two points in the scene. When the two objects collide we delete them each from their array. We check for collisions in the update() method of the SectorFive class.

SectorFive/SectorFive_3/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
    if rand < ENEMY_FREQUENCY
      @enemies.push Enemy.new(self)
    end
    @enemies.each do |enemy|
      enemy.move
    end
    @bullets.each do |bullet|
      bullet.move
    end
➤   @enemies.dup.each do |enemy|
➤     @bullets.dup.each do |bullet|
➤       distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
➤       if distance < enemy.radius + bullet.radius
➤         @enemies.delete enemy
➤         @bullets.delete bullet
➤       end
➤     end
➤   end
  end
```

The delete() method of the Array class removes the argument of the method from the array. We don't need to erase sprites from the window ourselves. We just delete the enemies from the @enemies array. When Gosu calls the draw() method, any enemies we've deleted are no longer in the array, and so are no longer drawn in the window.

Run the game. When a bullet collides with an enemy, both the bullet and the enemy disappear from the window. But we don't just want the enemy to disappear; it should explode in a ball of fire.

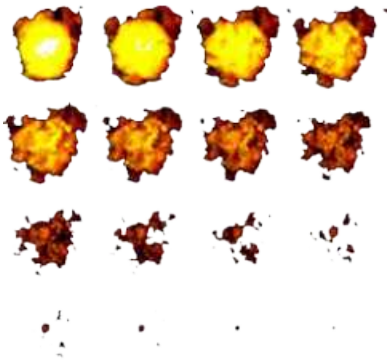
Make Animated Explosions

We could make an explosion with a single image, but it wouldn't look very convincing. Our explosion starts as a bright yellow ball of flame and then fades away, turning a darker orange and getting smaller. We do this by drawing sixteen images in the same location. When the explosion first appears, we draw the first in the sequence of images. Each subsequent time the game runs the draw() method, the next image in the sequence is drawn.

Use a Sprite Sheet

One way to load our images would be to copy sixteen individual image files into our project, and then load them into an array of images. This would work fine, though it might involve a lot of typing. Game artists often group related

images together by stitching them into a single image, called a *sprite sheet*. We use the following sprite sheet for our explosion animation.



The image at the top left of the sprite sheet is drawn first, and the image at the bottom right is drawn last. There are tools you can get to help you make a sprite sheet, and you can find sprite sheets online, in the same way you can find other game images. Now we set up our Explosion class to load its images from this sprite sheet.

In a new file called explosion.rb, create the empty Explosion class.

SectorFive/SectorFive_3/explosion.rb

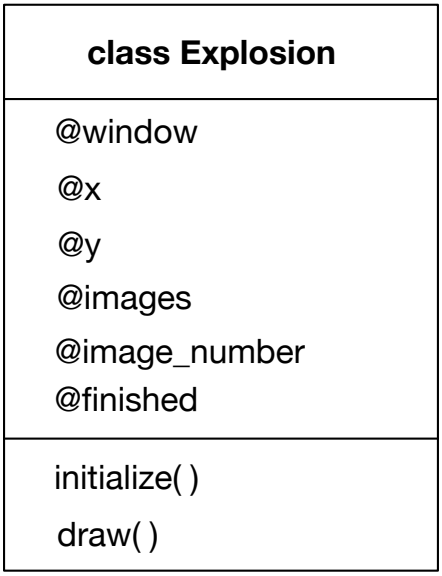
```
class Explosion
end
```

We also import explosion.rb into the sector_five.rb file.

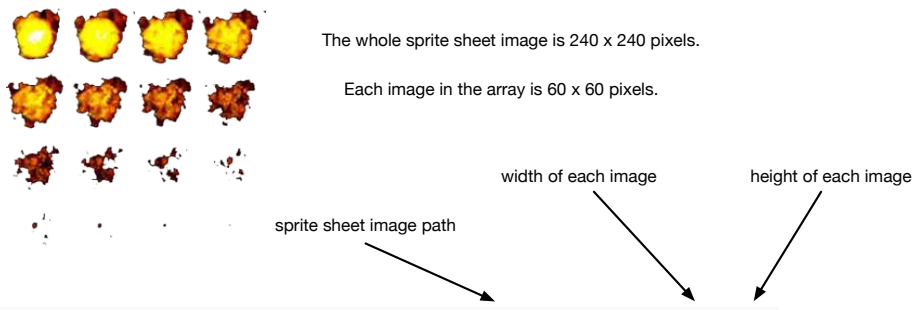
SectorFive/SectorFive_3/sector_five.rb

```
require_relative 'explosion'
```

Instead of a single image, the Explosion class uses an array of images. Other than that, its much like our other sprite classes. A variable called @image_index keeps track of the current image, while @finished is a boolean variable we set to true when our little animation is complete.



To make a new explosion, we pass in the location as arguments. The @images array gets created using a method of the Gosu::Image class called load_tiles(). This method chops up the sprite sheet into rectangles, and returns an array of images.




```
@images = Gosu::Image.load_tiles('images/explosions.png', 60, 60)
```

Here is the initialize() method of the Explosion class.

SectorFive/SectorFive_3/explosion.rb

```
def initialize(window, x, y)
  @x = x
  @y = y
  @radius = 30
  @images = Gosu::Image.load_tiles('images/explosions.png', 60, 60)
  @image_index = 0
  @finished = false
end
```

The Explosion class doesn't need a move() method, since our explosions stay in one place.

Animating the Explosion

Now that we've got sixteen explosion images in the @images array, we need to cycle through them, drawing a different one each frame for sixteen frames. We do it all in the draw() method of the Explosion class.

First we check to see if the @image_index variable is in the range of the array. Assuming it is, we draw the current image and increment @image_index, so that the next image in the array is drawn in the next frame of the game. If we've run out of images, we don't draw anything and we set the @finished variable to true. Later, we'll check this value and remove finished explosions from the array.

SectorFive/SectorFive_3/explosion.rb

```
def draw
  if @image_index < @images.count
    @images[@image_index].draw(@x - @radius, @y - @radius, 2)
    @image_index += 1
  else
    @finished = true
  end
end
```

Array numbering start at zero, so the highest number of the index of the array is one less than the count. If there are sixteen images in the array, @image_index goes from zero up to fifteen. If we tried to draw number sixteen, we'd get an error and crash our game.

Note that we cannot just use each() to draw all the explosion images. If we tried, Gosu would draw all the explosion images during a single run of the draw() method. Instead of an animation, all the images would appear at once, and only for a single frame. In order to spread out drawing the images over sixteen frames of the game, we need to manage the images ourselves, keeping track of which one we're drawing in each frame.

Make them Explode

We've written the Explosion class, and now we're ready to use it to make some enemies explode. First, we add a line of code in the initialize() method of the SectorFive class to create an empty array where we can put the explosions, just like we already do for bullets and enemies.

SectorFive/SectorFive_3/sector_five.rb

```
def initialize
```

```
    super(WIDTH, HEIGHT)
    self.caption = 'Sector Five'
    @player = Player.new(self)
    @enemies = []
    @bullets = []
➤   @explosions = []
  end
```

We're already checking for collisions between bullets and enemies, and removing the enemies. In that same loop we also create a new explosion and add it to the array.

SectorFive/SectorFive_3/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
  if rand < ENEMY_FREQUENCY
    @enemies.push Enemy.new(self)
  end
  @enemies.each do |enemy|
    enemy.move
  end
  @bullets.each do |bullet|
    bullet.move
  end
  @enemies.dup.each do |enemy|
    @bullets.dup.each do |bullet|
      distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
      if distance < enemy.radius + bullet.radius
        @enemies.delete enemy
        @bullets.delete bullet
➤       @explosions.push Explosion.new(self, enemy.x, enemy.y)
      end
    end
  end
end
```

Note that only the highlighted line of code is new. In the `draw()` method of the SectorFive class, we add a loop to draw the explosions.

SectorFive/SectorFive_3/sector_five.rb

```
def draw
  @player.draw
  @enemies.each do |enemy|
    enemy.draw
  end
  @bullets.each do |bullet|
    bullet.draw
  end
```

```
    @explosions.each do |explosion|
      explosion.draw
    end
  end
```

When you run the program now, and you shoot some enemies, they explode. So we're done, right?

Play for awhile. You might notice that the game starts to slow down. This might happen fairly quickly, or it might take awhile, depending on your computer. But eventually the game will *really* slow down. Our arrays are getting too big, and the computer is taking too long to do all the calculations for each frame.

Cleaning Up our Arrays

We've added a whole bunch of sprites to our arrays. Enemies, bullets, and explosions. We have deleted a few enemies and bullets that collided, but many enemies and bullets simply fly off the sides of our game and disappear. They may be gone from the window, but they are still in our arrays. And all the explosions are still hanging around, even though they aren't being drawn. So our arrays start to get really big, and we're still using the each() method to move and draw them. Ruby is doing a lot of calculations to move sprites we can't even see. We can fix this by deleting the sprites we're no longer using.

Delete the Explosions

Right now each explosion stops drawing itself after its sixteen frames of images, but it is still there, cluttering up our @explosions array. The solution is to delete the explosion when we're done with it. That's what the @finished variable is for. In the initialize() method of the Explosion class, we set @finished to false. In the draw() method, we set it to true when @image_index gets beyond the range of the @images array.

To remove the explosions that are finished, in the update() method we check all of our explosions, and remove any whose @finished has been set to true. So we can ask if an explosion is finished, we use the attr_reader helper in the Explosion class.

SectorFive/SectorFive_3/explosion.rb

```
class Explosion
  attr_reader :finished
```

Then we remove the expired explosions by adding to the update() method of the SectorFive class. We use the dup() method on our array, since we're changing the array as we iterate through it.

SectorFive/SectorFive_3/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
  if rand < ENEMY_FREQUENCY
    @enemies.push Enemy.new(self)
  end
  @enemies.each do |enemy|
    enemy.move
  end
  @bullets.each do |bullet|
    bullet.move
  end
  @enemies.dup.each do |enemy|
    @bullets.dup.each do |bullet|
      distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
      if distance < enemy.radius + bullet.radius
        @enemies.delete enemy
        @bullets.delete bullet
        @explosions.push Explosion.new(self, enemy.x, enemy.y)
      end
    end
  end
  @explosions.dup.each do |explosion|
    @explosions.delete explosion if explosion.finished
  end
end
```

Now only the explosions we can see are still in the @explosions array.

Delete Bullets and Enemies

We remove enemies and bullets from our arrays when they are no longer visible in the window. Enemies just fall off the bottom, while the bullets can exit from any side of the window. The test for enemies is simple enough that we'll put it right in the update(). The test to see if bullets are still in the window is more complicated, and we'll move the code to the Bullet class, so we can keep our update() method as simple as possible.

We give the Bullet class a new method, named onscreen?(). This method returns true if the position of the bullet is within the bounds of the window, and false if it is not.

The last line of onscreen?() is true only if the bullet is between the left and right edges, and also between the top and bottom.

SectorFive/SectorFive_3/bullet.rb

```
def onscreen?
  right = @window.width + @radius
  left = -@radius
  top = -@radius
  bottom = @window.height + @radius
  @x > left and @x < right and @y > top and @y < bottom
end
```

In the update() method we loop through the bullets, and remove the ones that are no longer in the window.

SectorFive/SectorFive_3/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
  if rand < ENEMY_FREQUENCY
    @enemies.push Enemy.new(self)
  end
  @enemies.each do |enemy|
    enemy.move
  end
  @bullets.each do |bullet|
    bullet.move
  end
  @enemies.dup.each do |enemy|
    @bullets.dup.each do |bullet|
      distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
      if distance < enemy.radius + bullet.radius
        @enemies.delete enemy
        @bullets.delete bullet
        @explosions.push Explosion.new(self, enemy.x, enemy.y)
      end
    end
  end
  @explosions.dup.each do |explosion|
    @explosions.delete explosion if explosion.finished
  end
  @enemies.dup.each do |enemy|
    if enemy.y > HEIGHT + enemy.radius
      @enemies.delete enemy
    end
  end
  @bullets.dup.each do |bullet|
    @bullets.delete bullet unless bullet.onscreen?
  end
end
```

Now you can play for a long time and the game won't slow down. The arrays only have the objects that we see in the window, so they never grow out of control.

Make it Your Own

At this point we have an almost completed sprite based game. This is a good point to stop and make some changes. Trying some experiments with the game will hone your skills and prepare you to bring some of your own game ideas to life. Here are a few ideas for changes you could make. Save a copy of your game before you start.

Make explosions drift.

Make the explosions drift. In a random direction, or down, or whatever you like. You need to give the `Explosion` class a `move()` method, and call that method on all the explosions.

Make explosions kill enemies.

Make the explosions collide with enemies, destroying the enemies if they are too close to the explosions. You need to test, just as we did with bullets and enemies, if any of the explosions are close enough to enemies to make them explode. See if you can get some explosion chain reactions going.

Make enemies fall at different speeds.

Right now the `Enemy` class has a constant `SPEED`, but you could change that to an instance variable. Then set that variable to a random number in `initialize()`.

Slow down the explosion animation.

Make it so each image in the animation lasts for two or three frames in the game. There are several ways to accomplish this. One way would be to make a separate `@frame_count` variable and then calculate `@image_count` from that.

Have the enemies shoot at the player.

This is a big change, and a significant challenge. Use the `Bullet` class if you can, rather than making a new class. Maybe add an instance variable that keeps track of whose bullet it is, and use a different image depending on whether its a player bullet or an enemy bullet. You could have the enemies fire randomly, or in the direction of the player.

What's Next?

Our sprite game toolkit is getting pretty full. We have only a few types of sprites, but we can make as many of each kind as we need. We can detect and handle key presses and collisions. Next, we'll finish our game by adding multiple *scenes*, so that we can give the player some instructions at the beginning, and a score at the end. We'll also add background music and sound effects, to add depth to our game.

CHAPTER 5

Adding Scenes and Sounds

Our game now has lots of sprites and is pretty fun to play. In its current state, its more the *core* of a game than it is an actual game. Imagine if we shared our game with some friends. They might have no idea what to do, which keys to press, or even that the falling ships are enemies that we can shoot. The game also never ends or has any kind of score.

In this chapter we change the game to show our players a little information about how to play, and let them start the game when they are ready.

When players press a key, the game itself begins, with enemies descending from the top of the window. The game ends when the player ship touches an enemy, flies off the top, or after one hundred enemy ships have been added. When the game ends we show players another screen that tells them how they did, with some credits and a choice to play again or quit. Different things

are shown in the window before the game begins, during the game, and after it ends. We call these phases of the game *scenes*.

Then we finish up our game by adding music and sounds. A different musical track plays in each scene, helping us establish the proper mood of anticipation, exciting action, or completion. We add some sound effects when enemies explode and when bullets are fired, to bring the player deeper into the game.

In this chapter we:

- Create distinct scenes for our game.
- Transition between our scenes.
- Draw scrolling credits on the screen.
- Add sound effects and background music to scenes.

When we're done our game will be complete, and we'll be ready to refine it, share it with our friends, and make more games.

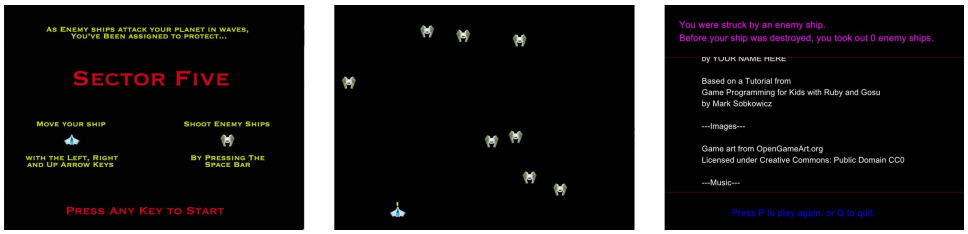
Start over with Scenes

Breaking a game into scenes is a flexible way to incorporate all kinds of mechanics into the game. Scenes could be game levels, short animations, or rooms in a dungeon. They help break a complex game into manageable chunks, like separate mini games that you can build one at a time.

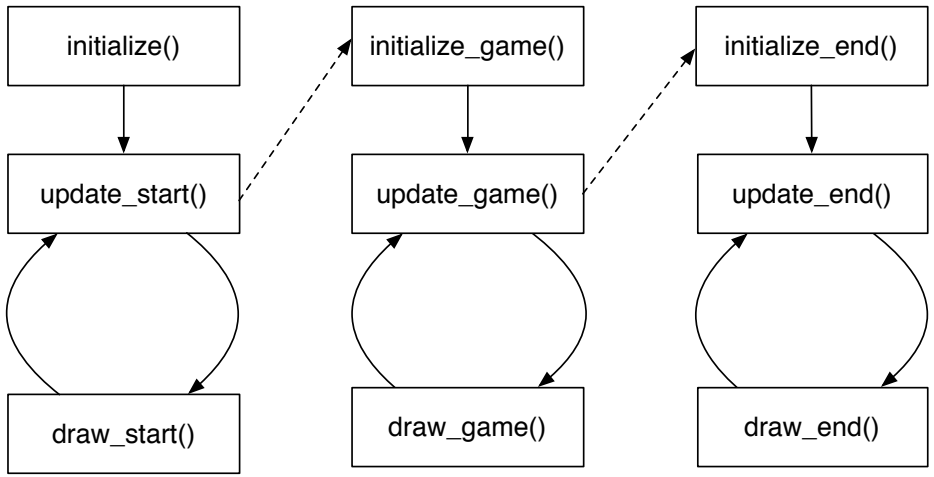
In order to organize our game into scenes, we write a completely new SectorFive class, but in it we reuse much of the code from our old one. We don't change our sprite classes at all. In the new SectorFive class we create separate update and draw methods for each scene. Most of the code we wrote in the last few chapters goes into the methods for the *game* scene.

Since we're starting over, we make a new file for our new SectorFive class. Create a file called sector_five_scenes.rb, and save it in the same folder as sector_five.rb. We are still using all our sprite classes and images, so its important to save the new file in the same folder as the old one. Don't delete sector_five.rb, since we'll be reusing much of its code in our new class.

In our game code we give our scenes names which we use in the names of some new methods. The game begins with a scene we call the *start* scene, and then transitions to the *game* scene when the player presses a key. The game can end in several ways, and when it does changes to the *end* scene.



Breaking the game into scenes changes where we put our code. Each scene gets its own update() and draw() methods. For the start scene we call these methods update_start() and draw_start(), and we stick to that naming strategy for our other scenes. When we transition to a new scene we call an initialize method for that scene. For instance, when we want to change to the game scene we call the initialize_game() method. The following figure shows the flow of our three scene game, with the dotted lines representing the transitions between scenes.



Our task is to achieve the flow in the preceding image with the tools Gosu gives us. Gosu runs the initialize() method, followed by draw() and update() in a loop, as shown in Figure 2, *The Gosu Run Loop*, on page 23. To achieve our new flow, we create a variable, @scene, to keep track of what scene we're currently in. Then in the update() and draw() methods of the SectorFive class, we check to see what scene we're in, and run the appropriate method. When we run the transition methods, such as the initialize_game() method, we set up instance variables for the new scene and change the value of @scene, so that the appropriate methods are called when Gosu runs its update() and draw() methods.

In the initialize() method the variable @scene is set to :start. Here is our new SectorFive class with its initialize() method.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
require 'gosu'
require_relative 'player'
require_relative 'enemy'
require_relative 'bullet'
require_relative 'explosion'
class SectorFive < Gosu::Window
  WIDTH = 800
  HEIGHT = 600
  ENEMY_FREQUENCY = 0.05
  def initialize
    super(WIDTH, HEIGHT)
    self.caption = "Sector Five"
    @background_image = Gosu::Image.new('images/start_screen.png')
    @scene = :start
  end
end
window = SectorFive.new
window.show
```

We don't need an initialize_start() method, since the initialize() method takes care of setting up the start scene.

Scene Methods

In the draw() method, we check the value of @scene and run the appropriate method.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def draw
  case @scene
  when :start
    draw_start
  when :game
    draw_game
  when :end
    draw_end
  end
end
```

The case control statement makes for readable code, and makes it easy to add new scenes later.

Each of the draw methods we've made, draw_start(), draw_game(), and draw_end(), is just like the draw() method before we introduced scenes. In draw_start(), we just draw the background image.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def draw_start
  @background_image.draw(0,0,0)
end
```

In the draw_game(), we can reuse the code that used to be in draw(). You can copy the code from sector_five.rb.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def draw_game
  @player.draw
  @enemies.each do |enemy|
    enemy.draw
  end
  @bullets.each do |bullet|
    bullet.draw
  end
  @explosions.each do |explosion|
    explosion.draw
  end
end
```

We'll add the draw_end() method later in the chapter, since what we draw depends on how the game ends.

We create our update methods in much the same way. Since we're just showing a static image in the start scene, we don't actually need an update_start() method. In the update() method, we check for the other two scenes, and if @scene is equal to :start, nothing happens.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def update
  case @scene
  when :game
    update_game
  when :end
    update_end
  end
end
```

When we run the game now, it shows the start scene image. Nothing happens now until the player presses a key.

Change Scenes

While we're in the start scene, we're waiting for the player to press a key. We detect the key press using the button_down() method. The button_down() method does different things depending on which scene we're in, and we don't want

the game to restart every time a key is pressed. Each scene gets a separate button_down() method, much like it gets its own draw() and update() methods.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def button_down(id)
  case @scene
  when :start
    button_down_start(id)
  when :game
    button_down_game(id)
  when :end
    button_down_end(id)
  end
end
```

When we are in the start scene, and press any key, we transition to the game scene. In button_down_start() we don't need to check which key was pressed, since we don't really care. We just run the initialize_game() method.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def button_down_start(id)
  initialize_game
end
```

The initialize_game() method does most of what the initialize() method did in the previous version of the game. You can copy most of it over. Don't run the super() method again, though. The superclass of SectorFive, Gosu::Window, doesn't have an initialize_game() method, so sending that message would cause an error. You also don't need to set the window caption again.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def initialize_game
  @player = Player.new(self)
  @enemies = []
  @bullets = []
  @explosions = []
  @scene = :game
end
```

In addition to the code we copied from our old initialize(), we've changed the @scene variable to :game. By doing this, we switch from running the update_start() and draw_start() methods in a loop, to running the update_game() and draw_game() methods.

Since the game scene is really just the game we wrote before, the update_game() method is what used to be the update() method. We can copy it over exactly, though later we'll add some code to end the game.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def update_game
  @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
  if rand < ENEMY_FREQUENCY
    @enemies.push Enemy.new(self)
  end
  @enemies.each do |enemy|
    enemy.move
  end
  @bullets.each do |bullet|
    bullet.move
  end
  @enemies.dup.each do |enemy|
    @bullets.dup.each do |bullet|
      distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
      if distance < enemy.radius + bullet.radius
        @enemies.delete enemy
        @bullets.delete bullet
        @explosions.push Explosion.new(self, enemy.x, enemy.y)
      end
    end
  end
  @explosions.dup.each do |explosion|
    @explosions.delete explosion if explosion.finished
  end
  @enemies.dup.each do |enemy|
    if enemy.y > HEIGHT + enemy.radius
      @enemies.delete enemy
    end
  end
  @bullets.dup.each do |bullet|
    @bullets.delete bullet unless bullet.onscreen?
  end
end
```

The code that was in the button_down() method now goes in the button_down_game() method.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def button_down_game(id)
  if id == Gosu::KbSpace
    @bullets.push Bullet.new(self,  @player.x, @player.y, @player.angle)
  end
end
```

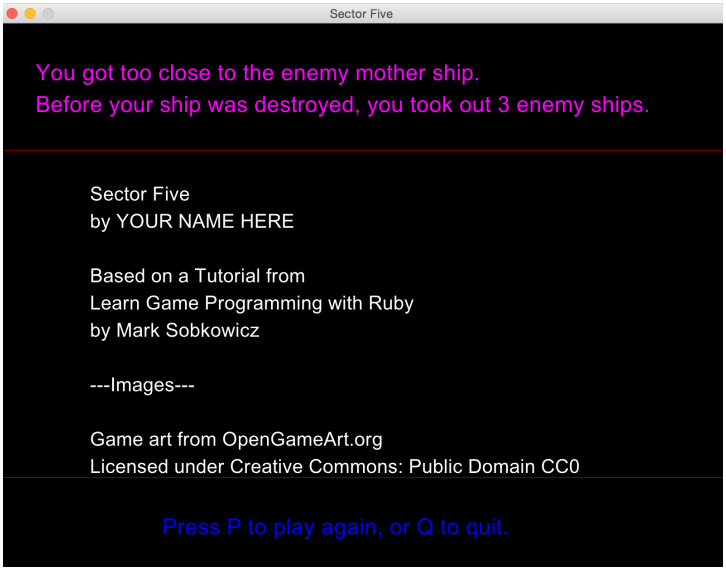
Run the game now, and press any key. You've got two scenes, and the game is the same as before. We have one more scene to add, for when the game is over. Before we add this scene, we need to discuss how the game ends.

End the Game

In some games, the point is to finish some task, or get to the highest level. In other games, there is score that the player can try and beat the next time they play. Our game includes both kinds of goals. The player is rewarded for surviving long enough for one hundred enemies to appear, and at the same time the game counts how many enemies the player destroys. Our game ends in one of three ways.

- One hundred enemy ships have appeared. Yay! The player has survived the wave of enemies.
- An enemy ship has hit the player ship. The player ship is destroyed, but hopefully blew up some enemies first.
- The player ship flew out of the top of the window. The enemy mother ship has destroyed the player ship.

However the game ends, it transitions to the end scene. We display a message telling the player how the game ended and how well she did. The end scene is an important part of the game. Its purpose is to let the player play again easily, and it should also provide a way to quit the game gracefully. It's a good place to give credit where credit is due, to the artists who made the graphics and sounds for the game, as well as to the programmer who wrote such an amazing game.



Credits scroll from the bottom to the top of a box inside the window and then repeat.

Keeping Score

Instead of a single score, we keep track of several things during our game. We count how many enemies have appeared and how many enemies the player has destroyed. Let's add some new instance variables for these in the initialize_game() method.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def initialize_game
  @player = Player.new(self)
  @enemies = []
  @bullets = []
  @explosions = []
  @scene = :game
➤   @enemies_appeared = 0
➤   @enemies_destroyed = 0
end
```

We increment each of these variables in a different place in our program. In each case, we've already written a conditional statement to check for the appropriate event, and we just need to add a statement to change the variable. For @enemies_appeared, we add one to its value in the same place we create enemies, in the update_game() method. Note that the whole update_game() method is not shown, only the code near where we make the addition.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
if rand < ENEMY_FREQUENCY
  @enemies.push Enemy.new(self)
  @enemies_appeared += 1
end
```

In the case of @enemies_destroyed, we have a statement in the update_game() method where we do our collision detection. When a bullet hits an enemy, we remove the bullet and the enemy, and add a new explosion. We also increment @enemies_destroyed here.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
@enemies.dup.each do |enemy|
  @bullets.dup.each do |bullet|
    distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
    if distance < enemy.radius + bullet.radius
      @enemies.delete enemy
      @bullets.delete bullet
      @explosions.push Explosion.new(self, enemy.x, enemy.y)
      @enemies_destroyed += 1
    end
  end
end
```

However the game ends, we call the initialize_end() method, and pass in a single argument that describes how the game ended. We'll write that method a little later; for now we just need to know that it takes that one argument.

One way the game can end is if enough enemies have appeared. We name the maximum number of enemies MAX_ENEMIES, and set it to 100 in the same place we create our other constants.

SectorFive/SectorFive_4/sector_five_scenes.rb
```
class SectorFive < Gosu::Window
  WIDTH = 800
  HEIGHT = 600
  ENEMY_FREQUENCY = 0.05
  MAX_ENEMIES = 100
```

In the update_game() method, we check each of our endgame conditions. If any is fulfilled, we call the initialize_end() method with the appropriate argument.

First we check if the number of enemies that has appeared exceeds the value of MAX_ENEMIES. If it is, we call the initialize_end() method with :count_reached as the argument.

To end the game when an enemy ship collides with the player, we loop through the enemy ships to see if an enemy hits the player ship. If one does, we call the initialize_end() method with the argument :hit_by_enemy.

This code is similar to what we wrote to check for collisions between bullets and enemies; the difference is that there is only one player instance, @player, so we only loop through the @enemies array, and check each one to see if it has collided with the player ship.

If the player ship flys off the top of the screen, it comes in range of the enemy mother ship, and is instantly destroyed. In this case we send the argument :off_top. Our update_game() method now looks like the following. The new code is highlighted at the bottom.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def update_game
  @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
  if rand < ENEMY_FREQUENCY
    @enemies.push Enemy.new(self)
    @enemies_appeared += 1
  end
  @enemies.each do |enemy|
    enemy.move
  end
  @bullets.each do |bullet|
    bullet.move
  end
  @enemies.dup.each do |enemy|
    @bullets.dup.each do |bullet|
      distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
      if distance < enemy.radius + bullet.radius
        @enemies.delete enemy
        @bullets.delete bullet
        @explosions.push Explosion.new(self, enemy.x, enemy.y)
        @enemies_destroyed += 1
      end
    end
  end
  @explosions.dup.each do |explosion|
    @explosions.delete explosion if explosion.finished
  end
  @enemies.dup.each do |enemy|
    if enemy.y > HEIGHT + enemy.radius
      @enemies.delete enemy
    end
  end
  @bullets.dup.each do |bullet|
    @bullets.delete bullet unless bullet.onscreen?
  end
➤   initialize_end(:count_reached) if @enemies_appeared > MAX_ENEMIES
```

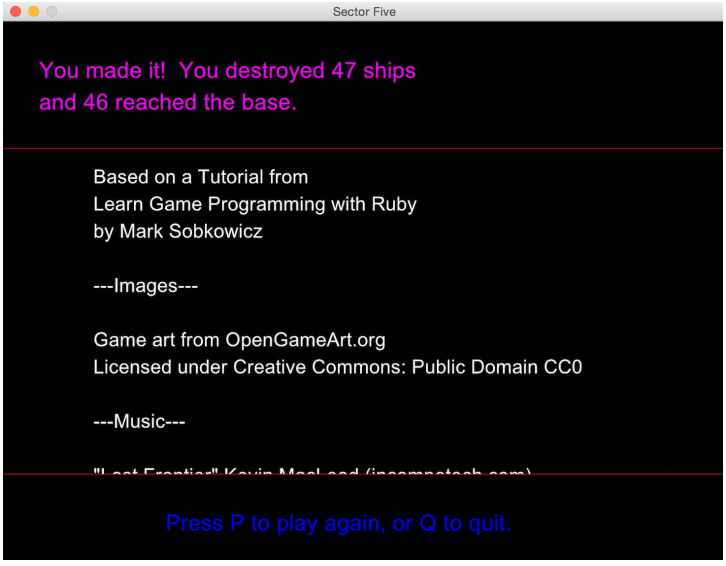
```
    @enemies.each do |enemy|
      distance = Gosu.distance(enemy.x, enemy.y, @player.x, @player.y)
      initialize_end(:hit_by_enemy) if distance < @player.radius + enemy.radius
    end
    initialize_end(:off_top) if @player.y < -@player.radius
end
```

However the game ends, we don't just want to quit, we want to tell the player how she did, and encourage her to try again.

Scrolling Credits

Making our credits scroll up the window lets us fit any amount of text into a finite space. Since we want to give credit to everyone who created the art and music for our game, not to mention the programmer, we end up with too much text to fit into the window at once. Plus, scrolling credits look both classic and cool.

Gosu makes it easy to write text on one line, but if we want multiple lines of text we need to take care of it ourselves, by drawing each line separately. To make our credits scroll up the window we use a strategy similar to the one we used to make enemies fall from the top. In our end scene the credits scroll up between horizontal red lines, while the text at the top and bottom of the window stays fixed.



Each line of text will be treated as a sprite, which we initially place below the bottom of the window. In the `update_end()` method we move each credit up in

the window. When the last line gets above the top of the window, we move all the lines back to their starting position, and the process starts over.

Start by creating a Credit class, in a new file, credit.rb. We make the speed of the credits a constant, equal to 1, and the initialize() method takes a string value that is the text we want to display on one line, along with x and y values for the initial position of the credit. The initial y position of the credit is stored in a separate variable, so we can reset our credits and run them again after they've finished.

SectorFive/SectorFive_4/credit.rb

```
class Credit
  SPEED = 1
  attr_reader :y

  def initialize(window, text, x, y)
    @x = x
    @y = @initial_y = y
    @text = text
    @font = Gosu::Font.new(24)
  end
end
```

The Credit class needs move() and draw() methods, much like those in the Enemies class. In addition, we make a reset() method, so we can start our credits over.

SectorFive/SectorFive_4/credit.rb

```
def move
  @y -= SPEED
end

def draw
  @font.draw(@text, @x, @y, 1)
end

def reset
  @y = @initial_y
end
```

To use the Credit class in our game, we require it along with our other classes.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
require 'gosu'
require_relative 'player'
require_relative 'enemy'
require_relative 'bullet'
require_relative 'explosion'
➤ require_relative 'credit'
```

The initialize_end() method sets up the end scene. In it, we create an instance of Gosu::Font to draw our end message on the screen, and we create and fill our array of credits.

We create three lines of messages that won't move up the screen. Two of these, @message and @message2, will be at the top, and depend on how the game ended. The third, @bottom_message, always says the same thing.

We could load up the credits by writing a line of code for each credit, but instead, we load them from a text file. Then, the credits can be changed anytime by editing the file. We start with the following text in the credits.txt file.

```
Sector Five
by YOUR NAME HERE

Based on a Tutorial from
Learn Game Programming with Ruby
by Mark Sobkowicz

---Images---

Game art from OpenGameArt.org
Licensed under Creative Commons: Public Domain CC0
```

We create one credit from each line of the text file. If we leave a blank line, a blank credit is created, which leaves a gap between the lines of text. We create the first credit at y = 700, near the bottom of the screen. Each one after has a y that is 30 more than the one before. This makes the credits appear in a list. Some of the credits are initially below the bottom of the screen, but thats fine. We scroll them up the screen in update_end().

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def initialize_end(fate)
  case fate
  when :count_reached
    @message = "You made it!  You destroyed #{@enemies_destroyed} ships"
    @message2= "and #{@enemies_escaped} reached the base."
  when :hit_by_enemy
    @message = "You were struck by an enemy ship."
    @message2 = "Before your ship was destroyed, "
    @message2 += "you took out #{@enemies_destroyed} enemy ships."
  when :off_top
    @message = "You got too close to the enemy mother ship."
    @message2 = "Before your ship was destroyed, "
    @message2 += "you took out #{@enemies_destroyed} enemy ships."
  end
  @bottom_message = "Press P to play again, or Q to quit."
  @message_font = Gosu::Font.new(28)
  @credits = []
```

```
  y = 700
  File.open('credits.txt').each do |line|
    @credits.push(Credit.new(self,line.chomp,100,y))
    y+=30
  end
  @scene = :end
end
```

When text is loaded line by line from a file, each line of text has a line break at the end. Gosu::Font throws an error if you try to draw text with a line break, so we use the chomp() method to remove the breaks.

In the draw_end() method, we want the credits to be drawn only within an invisible box on the screen, as if there were a window over part of the screen through which we could see the credits. To do this Gosu::Window has a method clip_to(), which takes four arguments. The first two are the x and y position of the top left corner of a rectangle, and the second two are the width and height of the rectangle. We also draw our messages, and red lines at the top and bottom of the scrolling credits region.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def draw_end
  clip_to(50,140,700,360) do
    @credits.each do |credit|
      credit.draw
    end
  end
  draw_line(0,140,Gosu::Color::RED,WIDTH,140,Gosu::Color::RED)
  @message_font.draw(@message,40,40,1,1,1,Gosu::Color::FUCHSIA)
  @message_font.draw(@message2,40,75,1,1,1,Gosu::Color::FUCHSIA)
  draw_line(0,500,Gosu::Color::RED,WIDTH,500,Gosu::Color::RED)
  @message_font.draw(@bottom_message,180,540,1,1,1,Gosu::Color::BLUE)
end
```

The top and bottom messages are drawn in color. The color is an optional argument to the Gosu::Font.draw() method, and to use it you need to include two more optional arguments before the color, which are x and y scaling factors. We use the color constants Gosu::Color::RED, Gosu::Color::BLUE, and Gosu::Color::FUCHSIA. Don't worry if you haven't heard of fuchsia; you can look up the color constants in the Gosu documentation[1].

In the update_end() method, we move the credits, and when they get high enough on the screen so that the bottom credit is no longer visible, we reset them to their original positions and watch them go by again. Since our clipping rectangle starts at y = 300, we reset them when the last credit reaches y = 250.

1. http://www.libgosu.org/rdoc/Gosu/Color.html

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def update_end
  @credits.each do |credit|
    credit.move
  end
  if @credits.last.y < 150
    @credits.each do |credit|
      credit.reset
    end
  end
end
```

There are many things you could adjust here, including placement and size of text, colors, and the speed of the credits. Feel free to change them to your liking.

Play Again

When we're in the end scene, we give the player the choice of quitting or playing again. If the player chooses to play again, we need to make sure all the correct variables are reset. To do this we can just run the initialize_game() method, which sets up a new game. We make the transition in button_down_end().

SectorFive/SectorFive_4/sector_five_scenes.rb
```
def button_down_end(id)
  if id == Gosu::KbP
    initialize_game
  elsif id == Gosu::KbQ
    close
  end
end
```

We're getting very close. The game is eerily quiet though, which we need to fix before our game is complete.

Add Music and Sounds

Music and sounds help set a mood, add another dimension to your game, and make it more polished. In this section we add a different background track to each scene, and add sound effects for shooting and explosions. Gosu makes a distinction between background music, which plays for a long time, and sound effects, which are short. Background music is streamed so the entire file does not need to be loaded into memory, and is not synchronized with the game. Sound effects, or *samples* are loaded into memory so that they can play at exactly the right moment in your game. Wouldn't do to have a lag between the enemy exploding and the explosion sound.

Finding Sounds

What you need are electronic files of sounds, in WAV or OGG format. If you have files in mp3 or some other format, they might work; you can try it. But WAV and OGG files should work across all platforms, so if you want to distribute your game to other people, you should find files in one of these formats, or convert the files you have into one of these formats.

Try, and Try Again

Playing sounds with Gosu can be troublesome and if you look through the Gosu forums, seems to be a pain point for many Gosu programmers. As you work with Gosu, you may have various sounds or songs either refuse to play, or play badly. The first thing to try is using a different sound format. OGG files seem to work the most consistently across all platforms, but WAV files may work better on your computer. To convert sounds, you can try the MediaHuman Audio Converter[a], which is free for OS X and Windows. The code included with this book includes sound files in OGG format.

a. http://www.mediahuman.com/audio-converter/

You can make these files yourself, or find them online, much like images. Perhaps you know someone that makes music, and would like one of their songs immortalized in your game. Or maybe you're a musician yourself. But if not, there are many sources of music online. Several are listed in *Images and Sounds*, on page 189.

Add Background Music

Two of the songs included in the tutorial are written by Kevin MacLeod and published on his website, incompetech.com. The third is by Tanner Helland and is published at tannerhelland.com. They are released under Creative Commons licenses which allow us to use them in any project, including a commercial project. We need to give credit to the creator, which we do by including the following text in the credits.txt file.

```
---Music---

"Lost Frontier" Kevin MacLeod (incompetech.com)
"Cephalopod" Kevin MacLeod (incompetech.com)
Licensed under Creative Commons: By Attribution 3.0
http://creativecommons.org/licenses/by/3.0/

"From Here" Tanner Helland (tannerhelland.com)
Licensed under Creative Commons: Attribution-ShareAlike 3.0
```

```
http://creativecommons.org/licenses/by-sa/3.0/
```

The three songs are for the start, game, and end scenes. For each song we create an instance of Gosu::Song, which takes only a file path as an argument. Then we play the song with the play() method. In the initialize() method of SectorFive, we add and play the first song.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def initialize
  super(WIDTH, HEIGHT)
  self.caption = "Sector Five"
  @background_image = Gosu::Image.new('images/start_screen.png')
  @scene = :start
➤ @start_music = Gosu::Song.new('sounds/Lost Frontier.ogg')
➤ @start_music.play(true)
end
```

Passing the optional value 'true' to the play() method plays the song in a loop, so that if the song reaches the end, it starts over. Some songs made for video games are designed to loop in such a way that you can't tell when it starts over. In the initialize_game() method, we add and play the next song. This one plays while the player shoots the ships, so it's fast paced and energetic.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def initialize_game
  @player = Player.new(self)
  @enemies = []
  @bullets = []
  @explosions = []
  @scene = :game
  @enemies_appeared = 0
  @enemies_destroyed = 0
➤ @game_music = Gosu::Song.new('sounds/Cephalopod.ogg')
➤ @game_music.play(true)
end
```

We don't need to end the first song. Gosu only plays one song at a time, so starting a new song stops whatever song was playing before.

In the initialize_end() method, we load and play the third song.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def initialize_end(fate)
  case fate
  when :count_reached
    @message = "You made it!  You destroyed #{@enemies_destroyed} ships"
    @message2= "and #{@enemies_escaped} reached the base."
  when :hit_by_enemy
    @message = "You were struck by an enemy ship."
    @message2 = "Before your ship was destroyed, "
```

```
      @message2 += "you took out #{@enemies_destroyed} enemy ships."
    when :off_top
      @message = "You got too close to the enemy mother ship."
      @message2 = "Before your ship was destroyed, "
      @message2 += "you took out #{@enemies_destroyed} enemy ships."
    end
    @bottom_message = "Press P to play again, or Q to quit."
    @message_font = Gosu::Font.new(28)
    @credits = []
    y = 700
    File.open('credits.txt').each do |line|
      @credits.push(Credit.new(self,line.chomp,100,y))
      y+=30
    end
    @scene = :end
➤   @end_music = Gosu::Song.new('sounds/FromHere.ogg')
➤   @end_music.play(true)
  end
```

When you play your game now, music plays throughout. Moody background music at the beginning and end, and thumping action music while the actual game is running. Next we need to hear those explosions.

Add Sound Effects

While music adds to the overall mood and experience of a game, sound effects provide feedback and improve the feeling of immersion in the world you've created. Animated fireballs are good, but animated fireballs accompanied by booming explosion sounds are better.

Gosu provides another class for short sounds that play when actions occur in the game. The Gosu::Sample class stores sound files in memory and plays them on demand. They play over the music, and over each other, so its best not to make them too long. The two sound effects included in this tutorial are each less than one second long. One, explosion.ogg, is for when enemies explode, and the other, shoot.ogg, is for each time we shoot a bullet. These sound files are from the freesound.org website, and have been placed in the public domain. Add this information to the credits file.

```
---Sound Effects---

Sound effects from freesound.org
Licensed under Creative Commons: Public Domain CC0
```

We load the sounds in the initialize_game() method. Then they'll be ready to play when we need them.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def initialize_game
  @player = Player.new(self)
  @enemies = []
  @bullets = []
  @explosions = []
  @scene = :game
  @enemies_appeared = 0
  @enemies_destroyed = 0
  @game_music = Gosu::Song.new('sounds/Cephalopod.ogg')
  @game_music.play(true)
➤ @explosion_sound = Gosu::Sample.new('sounds/explosion.ogg')
➤ @shooting_sound = Gosu::Sample.new('sounds/shoot.ogg')
end
```

Explosions are created in the update_game() method, and we play the appropriate sound whenever we create a new one. Only the part of the update_game() method where we test for collisions between bullets and enemies is shown, with the new line highlighted.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
@enemies.dup.each do |enemy|
  @bullets.dup.each do |bullet|
    distance = Gosu.distance(enemy.x, enemy.y, bullet.x, bullet.y)
    if distance < enemy.radius + bullet.radius
      @enemies.delete enemy
      @bullets.delete bullet
      @explosions.push Explosion.new(self, enemy.x, enemy.y)
      @enemies_destroyed += 1
➤     @explosion_sound.play
    end
  end
end
```

The shooting sound is a little loud and tends to get played quite a lot. We can tone it down by providing an optional volume argument to its play() method. Then we can still hear the background music.

SectorFive/SectorFive_4/sector_five_scenes.rb

```
def button_down_game(id)
  if id == Gosu::KbSpace
    @bullets.push Bullet.new(self,  @player.x, @player.y, @player.angle)
➤   @shooting_sound.play(0.3)
  end
end
```

Run the game and play it through. Make sure you die, fly off the top, and make it through, each at least once. Congratulations! You've finished the tutorial, and you've made a game, complete with tons of sprites, three scenes,

and sounds. You might feel ready to make a completely new game, or you might first want to try making some changes to this one.

Make it Your Own

Now that the game is done, you can take it anywhere you want. Here are some ideas to get you started. Feel free to do some or all of these, or to start from scratch with your own ideas.

Add another sound effect.

Add a sound effect for when enemies reach the bottom of the screen. Find a sound distinct from the exploding enemy sound, to alert the player that one got away.

Write the score on the screen.

Use an instance of `Gosu::Font` to write the number of enemies destroyed on the screen during the game scene. You'll need to decide how big to make it, what color, and where.

Make two waves of enemies.

Create a fourth scene that represents another wave of enemies, perhaps with different characteristics. If you made enemies that shoot at the player in a previous chapter, or enemies that move differently, you could add them in now.

Add a boss.

A boss is an enemy that is much harder to kill, often with some new abilities to test the player. Make a boss scene, with some appropriate music, and see if your players are up to the task.

Share your game.

In its current form as a bunch of Ruby, picture, and sound files, your game will only work on a computer whose owner has set up Ruby as we did in Chapter 1, *Get Ready*, on page 1. In Chapter 9, *Package and Share Your Game*, on page 181, you will learn how to package your game up for Mac or Windows, so others can use it just as they'd use any other application. If you'd like to share your game, you could skip ahead and do that now.

What's Next?

We've completed a sprite based game, in which the player controlled one sprite and others are managed by the program we wrote. Many games are of this type, and if you've worked through the tutorial to this point, you're ready to

make your own sprite based game. In the next chapter you'll learn how to make a puzzle game, where the player moves pieces on a board by clicking and dragging them. Gosu still provides the framework, but we'll use it somewhat differently, focusing on user actions rather than on the `update()` method.

CHAPTER 6

Creating a Puzzle Game

In this chapter, we'll learn to make a completely different kind of game from Sector Five. Our next game, Twelve, is a puzzle game, and is played on a grid of squares.

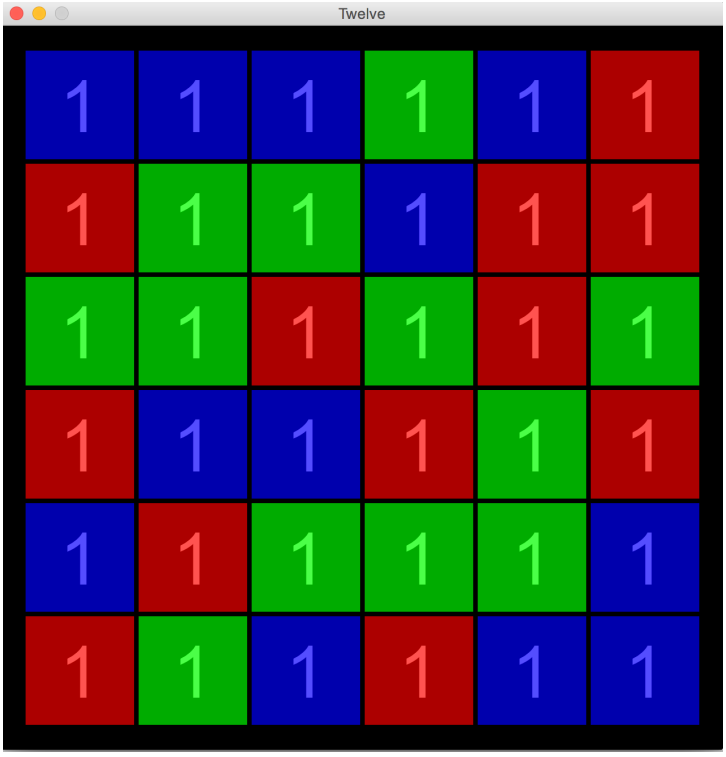


Figure 4—Twelve at the Start

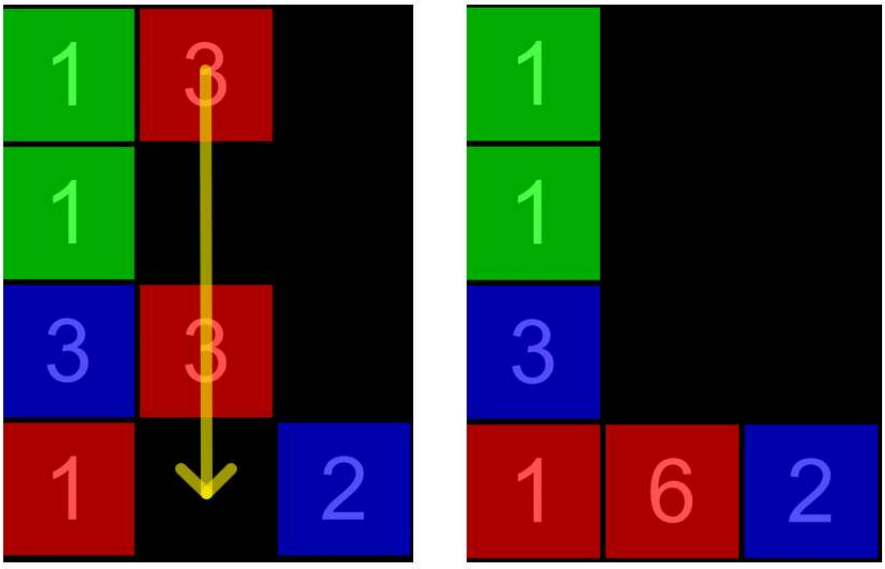
Players slide colored squares around a board, trying to combine as many like colored squares as they can. In this game most of the action happens when the player clicks and drags from one square to another. The objects in Twelve are the squares themelves, which stay in place. As successful moves occur, squares change their color and number. In order to figure out what happens when such a move occurs, we focus on the rules of our game, and learn how to translate those rules into code using a *flowchart*. Here is what we'll learn to do.

- Draw a layered image on the screen with colored squares and numbers.
- Handle mouse drags using Gosu's `button_down()` and `button_up()` methods.
- Create an flowchart to help figure out the result of a move.
- Translate the logic from our flowchart into code.

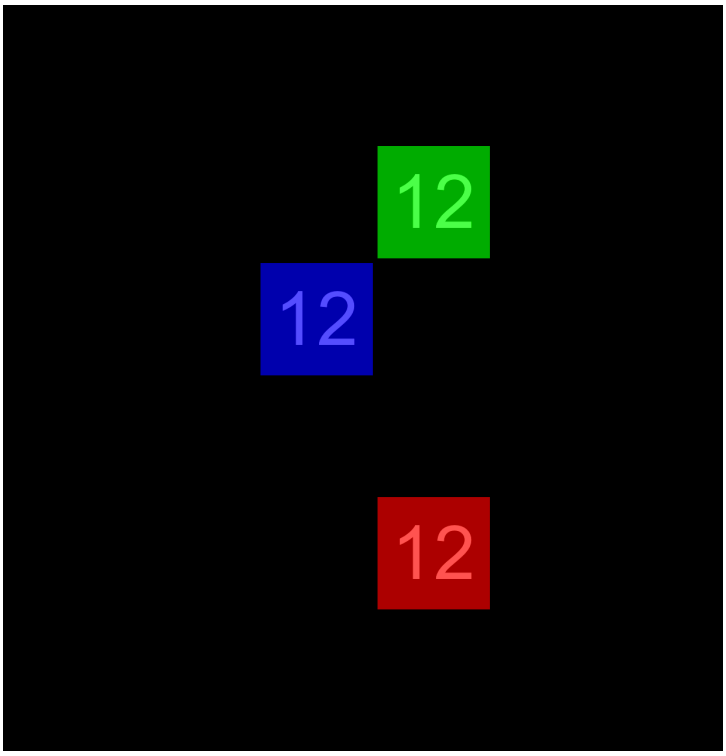
Puzzles are a fascinating class of games. Every few years, a puzzle game appears on the scene and you see it everywhere. In 2014, the game Threes! and its clones were on everyone's phones. A good puzzle game is simple to understand but difficult to master. Games of this sort are relatively easy to create and often have simple graphics, but making a great one is elusive. When yours goes viral, I hope you'll take a moment to write me about it (from your beachside mansion.)

When we start Twelve, the board looks something like the image in Figure 4, *Twelve at the Start*, on page 103, with twelve squares of each color randomly placed in a six by six grid. Since the colored squares are placed randomly, each game is a little different.

Players make moves by clicking on one square and dragging in a straight line onto a target square of the same color. When they release the mouse on the target square, the move is resolved. At the start of the game, the only possible moves are onto adjacent squares. As empty squares open up, moves can be made across them. If there are any empty squares on the other side of the target square, a move can continue into those. A region of the board is shown in the following picture, both before and after the player makes a move.



When the player completes the drag, and releases the mouse button, the new square has a number in it which is the sum of the numbers that were in the two joined squares. The name of the game, *Twelve* comes from the fact that the highest number you can get in a square is twelve. If you get three twelves, you've won the game.



Getting three twelves requires some strategic thinking, and maybe a little luck, as each starting board is different.

Great on a Tablet!

If you are working on a tablet or touchscreen laptop running Windows, Twelve is even more fun. You can touch and drag squares with your finger, without using a keyboard or mouse at all.

We build Twelve in stages. First, we draw the board, at the same time setting up much of the structure of the game. Then we focus on the user interaction and game logic. Some of the finishing touches are left up to you, as you learned how to add sounds and create a starting scene in Chapter 5, *Adding Scenes and Sounds*, on page 81.

Drawing The Board

Our game code is organized into three classes. The `Twelve` class is a subclass of `Gosu::Window`, and has the Gosu run loop methods: `initialize()`, `update()`, and `draw()`. It also has the user interaction methods `button_down()` and `button_up()`.

Roughly speaking, the Twelve class is responsible for creating and maintaining the window on the screen, and handling user input.

We create a separate class, Game, to handle the game logic. There is only one instance of the Game class at a time and it has methods that take user input, in the form of mouse down and mouse up locations, figure out if those represent legal moves in the game, and then resolve those moves.

A third class, Square, contains the state of a single square, and also draws that square. Each square has a color and a number, or is empty. Individual squares don't change position on the window. When a move is made, such as the move shown in the last picture, the numbers and colors of several squares are changed. Then the board is redrawn based on this new information the next time the draw() method of the Twelve class is executed.

Start by creating a folder called Twelve. In that folder, save three files called twelve.rb, game.rb, and square.rb, to hold our three classes. First, we write the Square class.

Making Squares

The main playing board of Twelve, as you can see in Figure 4, *Twelve at the Start*, on page 103, is a six by six grid of squares. We'll make each square 100 pixels on a side, and leave a 20 pixel border around the whole board, so our whole window will be 640 by 640 pixels. Squares will be identified by their row and column in the grid, each numbered from zero to five.

In square.rb, we write the Square class, and give it initialize() and draw() methods. That will be enough so that we can draw our board. The initialize() method takes row, column, and color values as arguments, as well as a reference to the window.

When we create the squares, each one has a @number value of 1. We use the symbols :red, :green, and :blue as values for the @color variable.

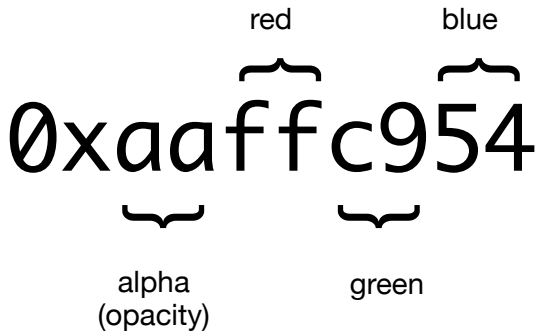
Since each square needs a reference to the same window, we'll use a class variable, @@window, to hold that reference. We'll also use class variables, @@font to draw our text, and @@colors to hold a hash that associates the symbols :red, :green, and :blue with three instances of the Gosu::Color class. We'll set those class variables when we create the first instance of Square.

Twelve/square.rb

```
require 'gosu'
class Square

  attr_reader :row, :column, :number, :color
```

```
def initialize(window, column, row, color)
  @@colors ||= {red: Gosu::Color.argb(0xaaff0000),
                green: Gosu::Color.argb(0xaa00ff00),
                blue: Gosu::Color.argb(0xaa0000ff)}
  @@font ||= Gosu::Font.new(72)
  @@window ||= window
  @row = row
  @column = column
  @color = color
  @number = 1
end
```

The Gosu::Color class has a method named argb() which takes as its argument an 8 digit hexidecimal number. Hexidecimal numbers in Ruby are written as "0x" followed by the eight digits. The first two digits are the *alpha* or opacity of the color. The remaining digits are two for the amount of red, two for the amount of green, and two for the amount of blue, as shown in the following figure.



Any color the screen can show, with any amount of transparency, can be represented by one of these numbers. Our three square colors are partially transparent, which will let the numbers show through from underneath.

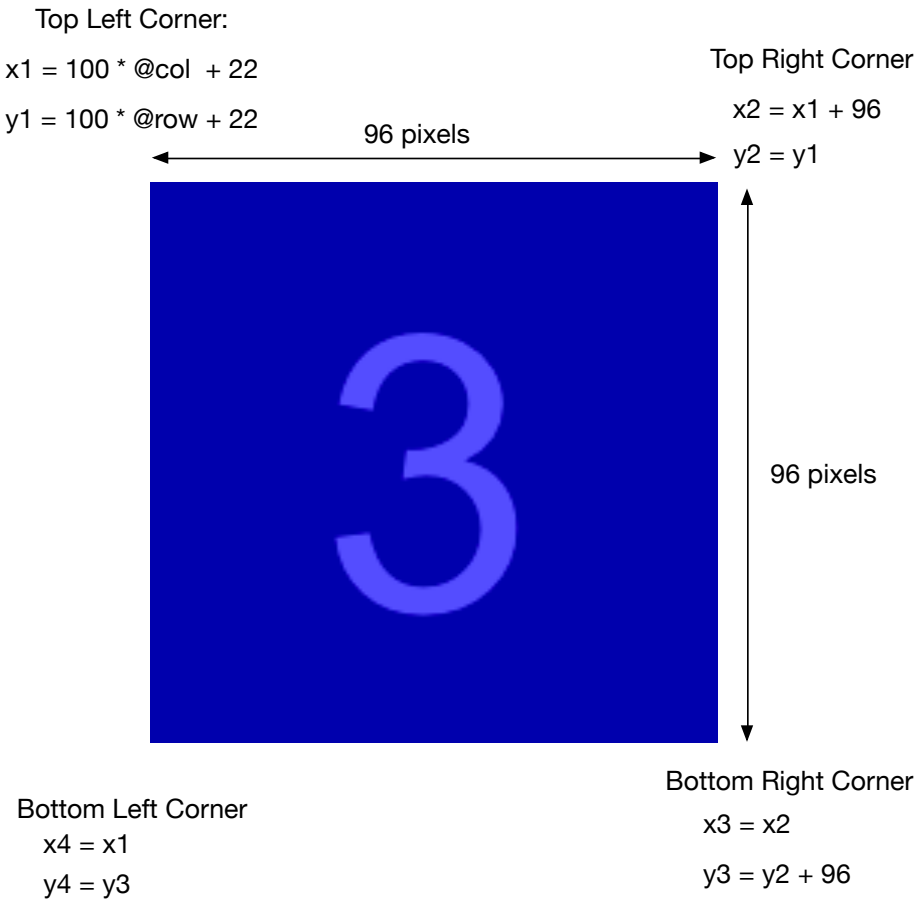
Hexidecimal Numbers

Hexidecimal numbers use the numerals 0 through 9, along with the letters A through F, to represent decimal numbers between 0 and 255 using only two characters. The letter A equals ten, the letter B eleven, and so on, up to F, which equals fifteen. To find the decimal equivalent of a hexidecimal number, multiply the first digit by sixteen and add the second digit. So the hexidecimal C9 corresponds to (12 x 16) + 9 = 201. Hexidecimal numbers are often used to denote colors in computer code.

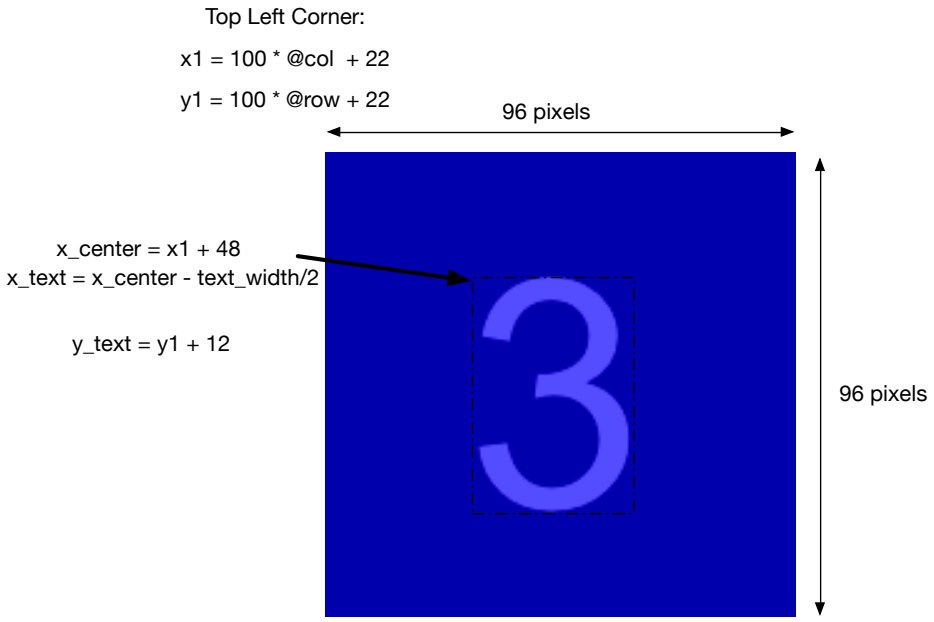
Drawing a Square

During the play of the game, some squares have numbers and colors, while other squares are empty. When a square becomes empty we don't delete it from the array, instead we set its `@number` variable to zero. An empty square might not remain empty, so our array will always have all thirty-six squares. When we draw a square we first check the value of `@number`. If `@number` isn't zero, we draw the square.

To draw a square, we need to figure out where on the window to put it, based on its `@row` and `@column` variables. Each square has a region 100 pixels on a side to occupy, and we leave a small, two pixel black border around around the edge of each square. This makes the colored part of the square 96 pixels on a side. We use the `draw_quad()` method to make the colored square, so we'll find the x and y values for all four corners, starting with the top left corner. The other corners positions are calculated from that one, as shown in the following image.



In order to draw the number exactly at the center of the square, we calculate where we should place the left edge of the number. We can find the width of any text using the text_width() method of the Gosu::Font class. By subtracting half the width of the text from the x position of the center of the square, we can get the x position of the left edge of the text. Finding the y position of the text is simpler, since the text is all the same height, 72 pixels.



In the draw() method of the Square class, we draw the squares over the numbers by giving the squares a larger z value than the numbers.

Twelve/square.rb

```
def draw
  if @number != 0
    x1 = 22 + @column * 100
    y1 = 22 + @row * 100
    x2 = x1 + 96
    y2 = y1
    x3 = x2
    y3 = y2 + 96
    x4 = x1
    y4 = y3
    c = @@colors[@color]
    @@window.draw_quad(x1, y1, c, x2, y2, c, x3, y3, c, x4, y4, c, 2)
    x_center = x1 + 48
    x_text = x_center - @@font.text_width("#{@number}") / 2
    y_text = y1 + 12
    @@font.draw("#{@number}", x_text, y_text, 1)
  end
end
```

Gosu doesn't really draw three dimensional pictures, but it does let you choose which things are drawn on top by giving them greater z values. Because of this you can think of the positive z direction as being up out of the screen.

When we draw the colored, semi-transparent square over the white number, we get numbers that contrast nicely with the colors of their squares.

Testing Some Squares

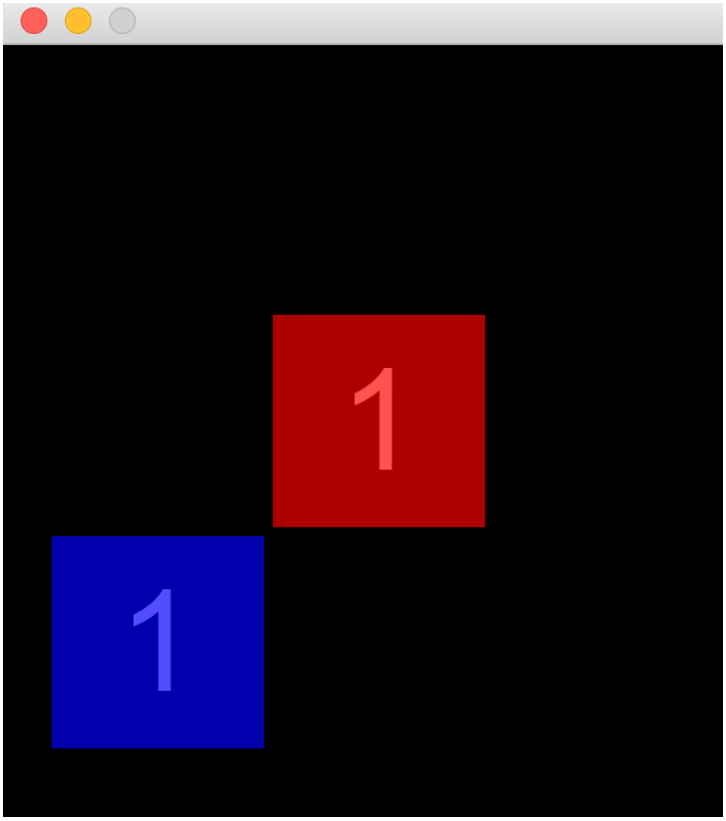
Before we move on to drawing the whole board, we test our Square class, to make sure things are working the way we want. When you're writing your own game this sort of testing can be very helpful.

Our simple test class creates two squares, and then draws them. This class isn't part of our finished game, but we leave it in the same folder as our game, in case we want to use it again. Name the class SquareTest, and put it in a file called square_test.rb.

Twelve/square_test.rb

```
require 'gosu'
require_relative 'square'

class SquareTest < Gosu::Window

  def initialize
    super(640, 640)
    self.caption = "Testing Squares"
    @square1 = Square.new(self, 0, 2, :blue)
    @square2 = Square.new(self, 1, 1, :red)
  end

  def draw
    @square1.draw
    @square2.draw
  end
end

window = SquareTest.new
window.show
```

When we run this small program, it draws two squares. The following image shows a region of the board containing these squares.



By changing the column and row values, we can draw the squares in different positions. Now that we can create and draw individual squares, we're ready to set up and draw the whole board.

A Grid of Squares

Our game has thirty-six squares, in six columns and six rows. We create them in the initialize() method of the Game class, and put them in an array called @squares. Once they are created, we draw them by iterating through the array and calling each square's draw() method.

When we create the squares, we can't just give each one a random color, since we want exactly twelve of each color. We handle this by creating an array, called color_list, with twelve elements each of :red, :green, and :blue. Then we randomize that array with the shuffle!() method of the Array class so that the colors are in random positions in the array, and there are still twelve of each. We then create the squares in six rows and six columns, each numbered zero to five. We calculate a value, index = row * 6 + column, which is different for each square, and use that element of the array to assign the color.

Twelve/game.rb
```
require_relative 'square'

class Game
  def initialize(window)
    @window = window
    @squares = []
    color_list = []
    [:red, :green, :blue].each do |color|
      12.times do
        color_list.push color
      end
    end
    color_list.shuffle!
    (0..5).each do |row|
      (0..5).each do |column|
        index = row * 6 + column
        @squares.push Square.new(@window, column, row, color_list[index])
      end
    end
    @font = Gosu::Font.new(36)
  end
end
```

Now the @squares array has thirty-six squares, with twelve of each color. We put these squares into the array in order. Any time we have the row and column of a square, we can calculate its array index with the same calculation, index = row * 6 + column, that we used to assign its color.

Now we give the Game class a draw() method so we can see them. The hard part of drawing the board is already done, since each square can draw itself. We just iterate through the array of squares and draw each one.

Twelve/game.rb
```
def draw
  @squares.each do |square|
    square.draw
  end
end
```

We'll come back to the Game class later, when its time to handle our player's interaction with the squares. Now let's set up the Twelve class so that it draws the board.

In twelve.rb, we put the Twelve class, and then use it to create our window. In the initialize() method, we create a single instance of Game. In draw(), we can just call the draw() method of @game.

Twelve/twelve.rb
```
require 'gosu'
```

```
require_relative 'game'

class Twelve < Gosu::Window

  def initialize
    super(640,640)
    self.caption = 'Twelve'
    @game = Game.new(self)
  end
  def draw
    @game.draw
  end


end

window = Twelve.new
window.show
```

When you run the game now, you see the game window, with thirty-six squares, as shown in Figure 4, *Twelve at the Start*, on page 103. Remember to make sure you run twelve.rb, and not game.rb or square.rb. Our board is ready. It's time to let the player make some moves.

Dragging a Square

Dragging from one square to another is at the heart of Twelve. A potential move is defined by which square the user mouses down on, along with the square where the user releases the mouse. We figure out if the move from the first square to the second is a valid move. If it is, we execute the move, changing the numbers and colors of the appropriate squares. If its not, we don't do anything at all, and the player can try again.

Before we start working with mouse clicks, you might notice that the mouse cursor does not appear when the mouse is in the window. For an arcade style game, this might be appropriate, but while playing Twelve, the user needs to see the mouse cursor. To make this change, add the following method to the Twelve class.

Twelve/twelve.rb

```
def needs_cursor?
  true
end
```

If you're going to use this game only on a touch screen laptop or tablet, you could leave this method out.

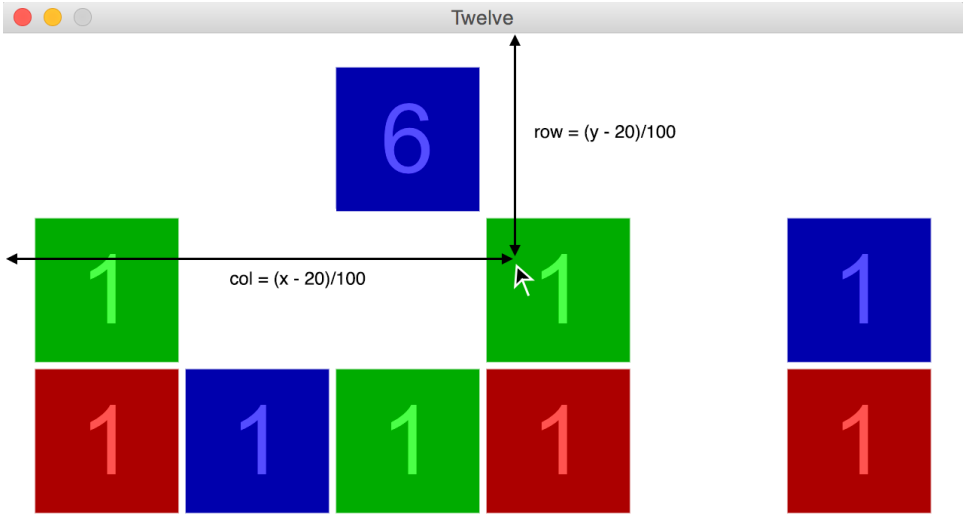
Mouse Down

When we mouse down in our window, we pass the location of the mouse click to a method of the Game class called handle_mouse_down().

Twelve/twelve.rb

```
def button_down(id)
  if id == Gosu::MsLeft
     @game.handle_mouse_down(mouse_x, mouse_y)
  end
end
```

In the handle_mouse_down() method, we use a little math to figure out which square the player clicked on. This is the reverse of the math we did to draw a square at a particular row and column. If, for instance, a mouse click has a x value of 350, it is in column 3, which is actually the fourth column from the left.



We do a similar calculation to find the row. Once we've figured out the row and column of the square, we get a reference to that square using a utility method, get_square(col,row) that we haven't yet written. The reference is saved in an instance variable, @start_square.

Twelve/game.rb

```
def handle_mouse_down(x,y)
  row = (y.to_i - 20)/100
  column = (x.to_i - 20)/100
  @start_square = get_square(column, row)
end
```

The get_square() method first checks to make sure we haven't clicked outside the board. If we have, there is no square there, so no square should be returned. Assuming the click is inside the board, the appropriate square is returned. Note the calculation of the index is the same as the one in the initialize() method.

Twelve/game.rb
```
def get_square(column, row)
  if column < 0 or column > 5 or row < 0 or row > 5
    return nil
  else
    return @squares[row * 6 + column]
  end
end
```

Now that we've got the starting square safely stored, the next step is to figure out what happens when the player releases the mouse button.

Mouse Up

A move in Twelve starts when the player clicks on a square. In the last section, we figured out which square was clicked on, and we saved a reference to that square in @start_square. The move ends when the player releases the mouse. At this point, if the move is a valid one, the contents of two squares will be combined, and put into the square where the mouse up occurred. In order for us to figure out if the move is valid, we first need to figure out which square the player's mouse cursor was in when the mouse button was released.

Just as Gosu::Window provides a button_down() method, it also provides a button_up() method, which fires when a key or mouse button is released. Our button_up() method will look much like our button_down() method.

Twelve/twelve.rb
```
def button_up(id)
  if id == Gosu::MsLeft
    @game.handle_mouse_up(mouse_x, mouse_y)
  end
end
```

The handle_mouse_up(x,y) method of the Game class does the same calculation that the handle_mouse_down(x,y) method does to find the row and column where the mouse up happened. Then the get_square() method is used to get a reference to that square, which is saved in @end_square. We check to make sure that the player actually dragged from one square to another, rather than into or from an area of our window with no squares. If both @start_square and @end_square

refer to squares, meaning that neither is nil, we call the move() method, and pass it both squares.

Twelve/game.rb
```
def handle_mouse_up(x, y)
  row = (y.to_i - 20) / 100
  column = (x.to_i - 20) / 100
  @end_square = get_square(column, row)
  if @start_square and @end_square
    move(@start_square, @end_square)
  end
  @start_square = nil
end
```

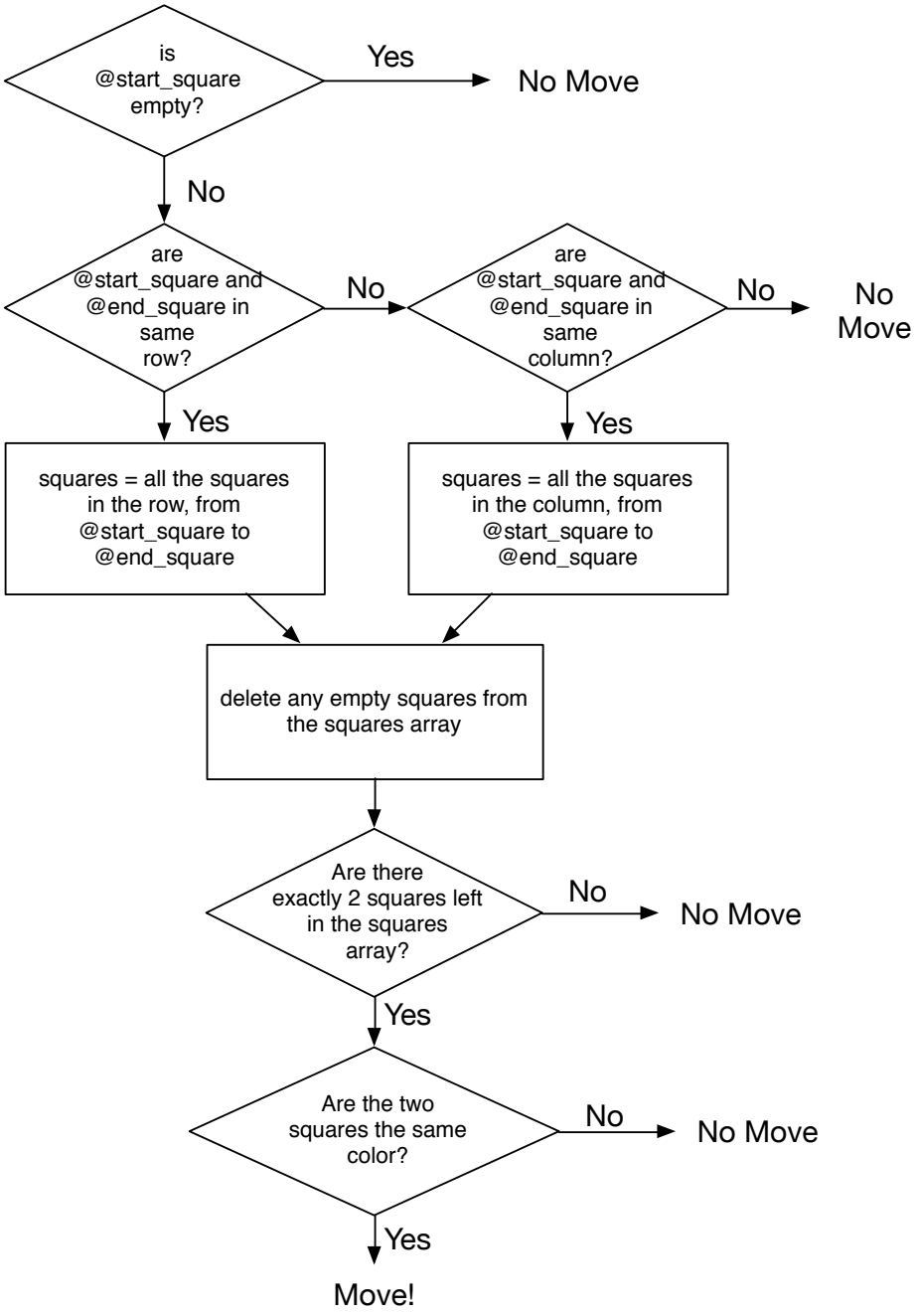
In the move() method, we now have to figure out if a move from @start_square to @end_square is a valid move. To do this let's take a closer look at our rules, and how we can express them in code.

Turn Rules into Code

The rules of our game seem simple enough. A drag is valid if it's within a row or a column, and if the two squares are the same color, and if there are no other squares between them. The drag can end on the second square, or it can continue onto empty squares beyond the second square. In this section we write the method, move(square1, square2) that makes these rules part of our game.

Make a Flowchart

To help us get from our rules to code, we create a flowchart. A flowchart is a diagram that represents a set of rules in an organized way. Just making a flowchart like this can help us clarify what we really mean when we state the rules of a game. In a flowchart, conditionals are represented with diamonds, and calculations are represented by rectangles. Each conditional has two arrows leading out of it, one for YES and one for NO, while a rectangle has only one path out. The rules of Twelve, that determine whether a particular move from @start_square to @end_square is valid, are represented by the following flowchart.

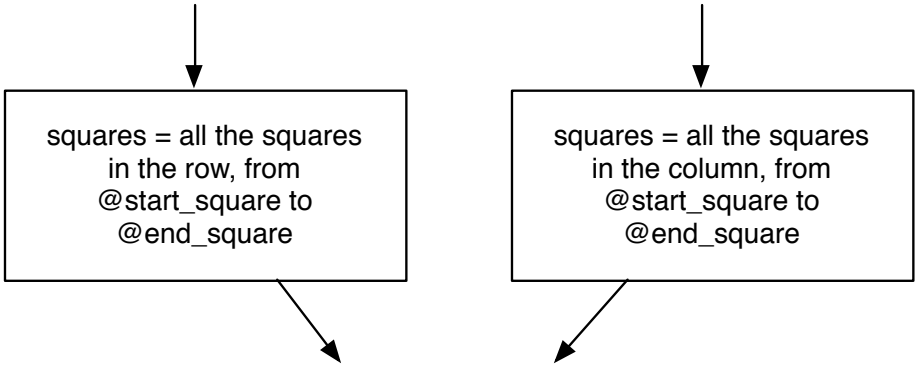


The strategy followed by this flowchart, to determine if the move is valid, is to go through all the things that would make a move *invalid* and test for those,

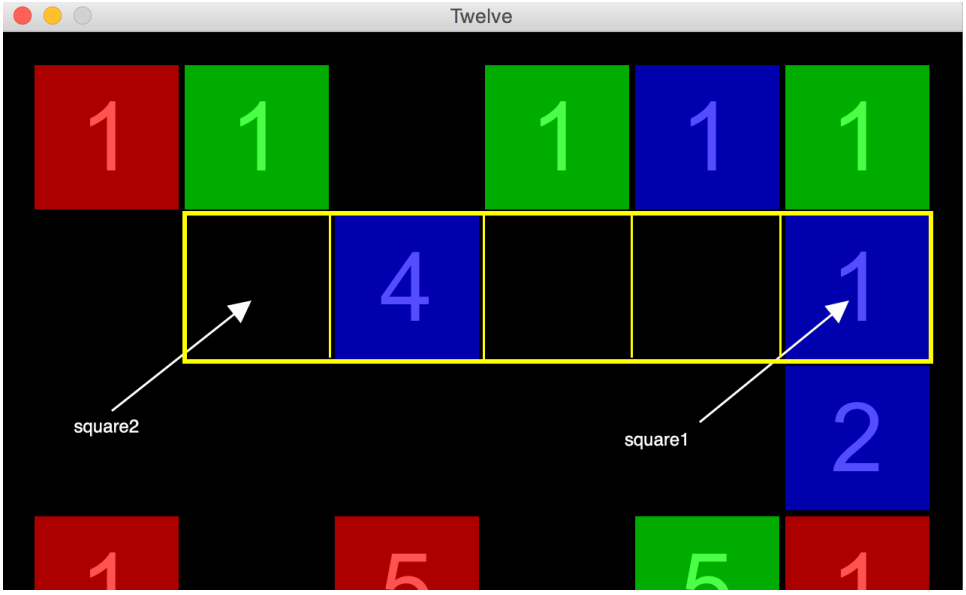
one at a time. Any move that isn't invalid for any of these reasons must be valid.

Make Code from the Flowchart

In order to turn this flowchart into code, we'll first write a few supporting methods. In particular, the calculations required by the top rectangles in the flowchart are complicated enough to make into their own methods.



One method, `squares_between_in_row(square1, square2)` takes two squares and returns an array of all the squares between those squares in a row. In the drag shown by the following picture, the method returns an array of five squares.



A second method, squares_between_in_column(square1, square2), does the same thing for squares in a column.

Twelve/game.rb
```
def squares_between_in_row(square1, square2)
  the_squares = []
  if square1.column < square2.column
    column_start, column_end = square1.column, square2.column
  else
    column_start, column_end = square2.column, square1.column
  end
  (column_start .. column_end).each do |column|
    the_squares.push get_square(column, square1.row)
  end
  return the_squares
end

def squares_between_in_column(square1, square2)
  the_squares = []
  if square1.row < square2.row
    row_start, row_end = square1.row, square2.row
  else
    row_start, row_end = square2.row, square1.row
  end
  (row_start .. row_end).each do |row|
    the_squares.push get_square(square1.column, row)
  end
  return the_squares
end
```

Now we can write our move(square1, square2) method. Each of the diamonds in our flowchart represents a conditional statement. If one of the branches of the conditional leads to a "No Move" result, we'll simply return from our method. This causes nothing to happen, which is what we want.

Twelve/game.rb
```
def move(square1, square2)
  return if square1.number == 0
  if square1.row == square2.row
    squares = squares_between_in_row(square1, square2)
  elsif square1.column == square2.column
    squares = squares_between_in_column(square1, square2)
  else
    return
  end
  squares.reject!{|square| square.number == 0}
  return if squares.count != 2
  return if squares[0].color != squares[1].color
  #valid move
end
```

If none of the four return statements are triggered, we have a valid move. We clear the two squares that were not empty, and fill the square our move ended on. The square we ended on might be one of the ones that had a value, or it might not. Our strategy works either way.

To change the state of the squares let's make some new methods in the Square class. One will clear a square, setting @number to zero. Another takes a number and a color as arguments, and sets the square's @number and @color variables to those values.

Twelve/square.rb

```
def clear
  @number = 0
end

def set(color, number)
  @color = color
  @number = number
end
```

Then at the end of the move(square1, square2) method of Game we can clear the two squares left in the squares array, and set the color and number of square2, which is where the move ended.

Twelve/game.rb

```
color = squares[0].color
number = squares[0].number + squares[1].number
squares[0].clear
squares[1].clear
square2.set(color, number)
```

Now our game works. Play a few games, and see if you can get to three twelves. One final annoyance is that we need to restart the game to play again. Let's make it so we can play again more easily.

Play Again?

We let people play again by pressing 'CTRL-R'. We could more easily make it just 'R', but then players might restart accidentally when they've almost won a game. Since the button_down() method responds to each key press, rather than to combinations of keys, we'll check to see if the 'R' key is pressed, and then use the button_down?() method to see if the 'CTRL' key is also being pressed at that moment. To restart the game, all we need to do is call the initialize() method of the Game class and replace @game with a new Game instance. Add this to the button_down() method of the Twelve class.

Twelve/twelve.rb

```
if id == Gosu::KbR && button_down?(Gosu::KbLeftControl)
```

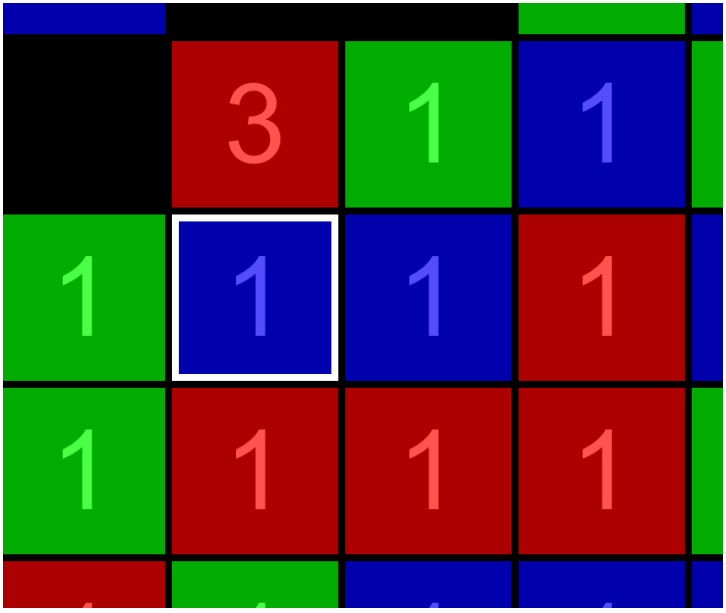
```
    @game = Game.new(self)
end
```

Thats a nice benefit we get from separating out the Game class. Since the @game object represents a single game, we can replace it any time with a new game. Now we can play over and over.

Add Visual Feedback

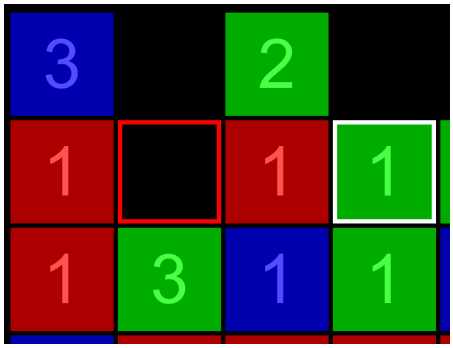
When you play Twelve now, all the action happens when the mouse button is released. When you click on a square and slide the mouse around, nothing appears to happen. In this section, we add highlights to the squares to give the user some visual feedback about the move they are making.

First, as the player clicks on a square to begin a move, we add a white square highlight to the starting square, as shown in the following image.

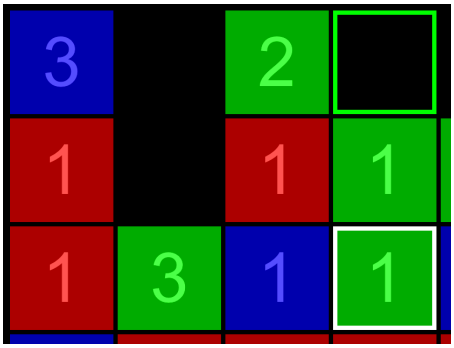


This square remains highlighted until the player releases the mouse, reminding the player which square is making the move.

As the player moves the mouse around to different squares, the square that the mouse cursor is in is also highlighted. If the move is a legal one, the highlight is green; if not, it is red.

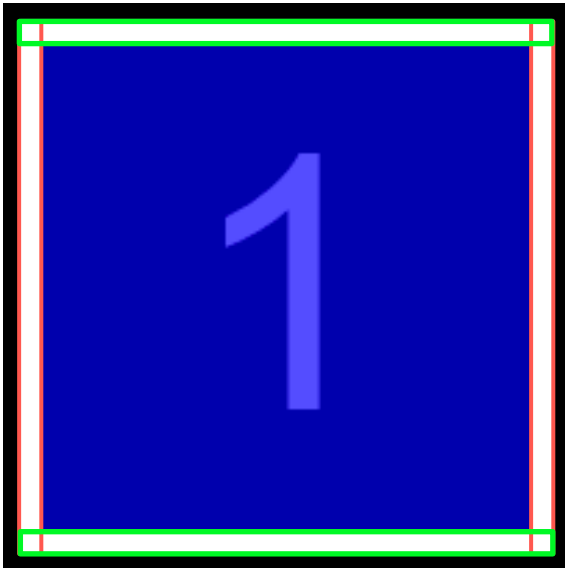


Highlighted Illegal Move



Highlighted Legal Move

Each of these highlights is actually four rectangles, each four pixels in one dimension and 96 in the other. Two of these rectangles, on the top and bottom of the square, are oriented horizontally, while the other two are oriented vertically. These rectangles are shown outlined in red and green in the following image.



The highlight is made up of two horizontal rectangles, and two vertical rectangles.

The highlight rectangles overlap at the corners, but since all four are one color, they appear as the outline of a single square.

To make these highlights, we first make a new method in the Square class called highlight(state). The state argument is either :start, :legal, or :illegal, and determines which color we draw the highlight rectangles. Two more methods, draw_horizontal_highlight(x1, y1, c) and draw_vertical_highlight(x1, y1, c) create the actual rectangles. The arguments x1 and y1 are the position of the top left corner of the highlight rectangle.

Twelve/square.rb

```
def highlight(state)
  case state
  when :start
    c = Gosu::Color::WHITE
  when :illegal
    c = Gosu::Color::RED
  when :legal
    c = Gosu::Color::GREEN
  end
  x1 = 22 + @column * 100
  y1 = 22 + @row * 100
  draw_horizontal_highlight(x1, y1, c)
  draw_horizontal_highlight(x1, y1 + 92, c)
  draw_vertical_highlight(x1, y1, c)
  draw_vertical_highlight(x1 + 92, y1, c)
end

def draw_horizontal_highlight(x1, y1, c)
  x2 = x1 + 96
  y2 = y1
  x3 = x1 + 96
  y3 = y1 + 4
  x4 = x1
  y4 = y1 + 4
  @@window.draw_quad(x1, y1, c, x2, y2, c, x3, y3, c, x4, y4, c, 3)
end

def draw_vertical_highlight(x1,y1,c)
  x2 = x1 + 4
  y2 = y1
  x3 = x1 + 4
  y3 = y1 + 92
  x4 = x1
  y4 = y1 + 92
  @@window.draw_quad(x1, y1, c, x2, y2, c, x3, y3, c, x4, y4, c, 3)
end
```

In the Twelve class, we implement the update() method, which we haven't needed until now. We want to know where the mouse cursor is all the time, so we can highlight the current square. In the update() we send the mouse position to a new method of the Game class called handle_mouse_move(x, y). There

is a nice symmetry to this, since we already have methods called handle_mouse_down(x, y) and handle_mouse_up(x, y).

Twelve/twelve.rb
```
def update
  @game.handle_mouse_move(mouse_x, mouse_y)
end
```

In the Game class, we implement the handle_mouse_move method. This method figures out which square the mouse is on, and sets the value of @current_square.

Twelve/game.rb
```
def handle_mouse_move(x, y)
  row = (y.to_i - 20) / 100
  column = (x.to_i - 20) / 100
  @current_square = get_square(column, row)
end
```

We write a new method, move_is_legal?(square1, square2), to help figure out what highlight to apply to @current_square. This method is very similar to the move(square1, square2) method, but it simply returns true or false and does not execute the move.

Twelve/game.rb
```
def move_is_legal?(square1, square2)
  return false if square1.number == 0
  if square1.row == square2.row
    squares = squares_between_in_row(square1, square2)
  elsif square1.column == square2.column
    squares = squares_between_in_column(square1, square2)
  else
    return false
  end
  squares.reject!{|square| square.number == 0}
  return false if squares.count != 2
  return false if squares[0].color != squares[1].color
  return true
end
```

Now we're ready to highlight the squares in the draw() method of the Game class. We don't want any highlighting between moves, when the value of @start_square is nil. We also make sure @current_square is different from @start_square before we give it a highlight.

Twelve/game.rb
```
def draw
  @squares.each do |square|
    square.draw
  end
➤ return unless @start_square
```

```
    @start_square.highlight(:start)
    return unless @current_square && @current_square != @start_square
    if move_is_legal?(@start_square, @current_square)
      @current_square.highlight(:legal)
    else
      @current_square.highlight(:illegal)
    end
  end
```

Now when you play you get some more feedback, and the game is more fun.

Up until now, it has been up to the user to determine when there are no more moves to make. We alreay have a method move_is_legal(square1, square2) that can tell us if a single move is legal. In the next section, we'll use that same method to test *every possible move* to find out if there are *any* legal moves left.

Check All the Moves

The game is over when there are no more moves for the player to make. Sometimes this is hard for players to figure out, and it's a nice touch to do it for them. Our strategy is to check every possible move until we find a legal one. If we don't find one, the game is over. We create two new methods in the Game class to help us figure this out. The first one, legal_move_for?(start_square) will return true if start_square has any possible legal moves, and false if it does not. First it checks to see if start_square is empty, and returns false in this case. Then it checks the moves from start_square to every square on the board. If it finds a legal move, it returns true. If it does not, it returns false.

Twelve/game.rb

```
def legal_move_for?(start_square)
  return false if start_square.number == 0
  @squares.each do |end_square|
    if move_is_legal?(start_square, end_square)
      return true
    end
  end
  return false
end
```

The game_over?() checks each square. If it finds one with a legal move it returns false, since in this case the game is not over. If no square has a legal move, it returns true.

Twelve/game.rb

```
def game_over?
  @squares.each do |square|
    return false if legal_move_for?(square)
  end
```

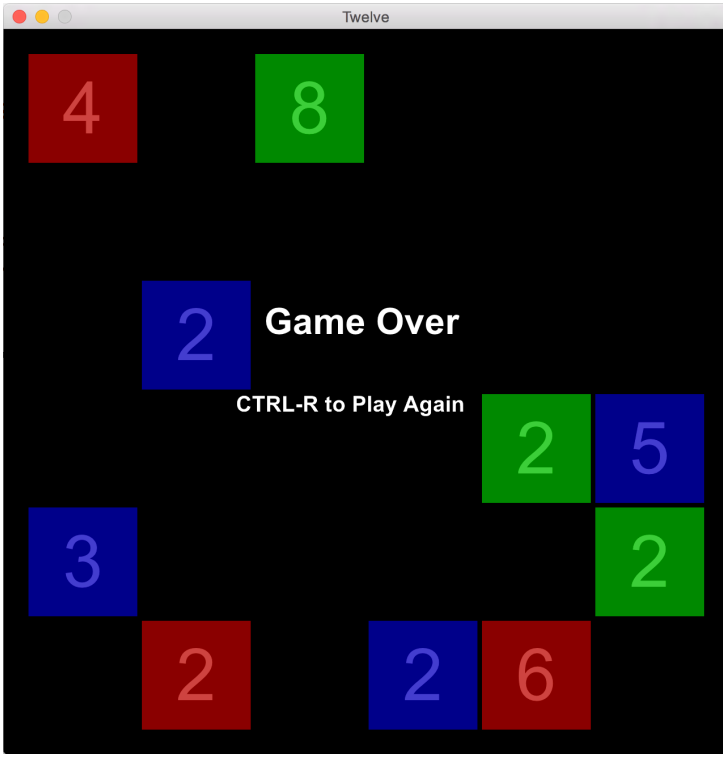
```
    return true
  end
```

When the game ends, we alert the player by drawing a translucent square over the whole board, and writing “Game Over” over that. We also tell the player how to play again, and we stop highlighting our squares.

Twelve/game.rb

```
  def draw
    @squares.each do |square|
      square.draw
    end
    if game_over?
      c = Gosu::Color.argb(0x33000000)
      @window.draw_quad(0, 0, c, 640, 0, c, 640, 640, c, 0, 640, c, 4)
      @font.draw('Game Over', 230, 240, 5)
      @font.draw('CTRL-R to Play Again', 205, 320, 5, 0.6, 0.6)
      return
    end
    return unless @start_square
    @start_square.highlight(:start)
    return unless @current_square && @current_square != @start_square
    if move_is_legal?(@start_square, @current_square)
      @current_square.highlight(:legal)
    else
      @current_square.highlight(:illegal)
    end
  end
```

When the game ends, the player can still see the board, though it is visibly darker. Clicking and moving the mouse no longer has any visible effect, since we return from the draw() method before the part that draws highlights.



Twelve now helps players in two ways. Players can see whether the move they are currently making is legal, and Twelve lets them know when the game is over and there are no more moves to make. You should play a few times and think of ways you could make it even better.

Make it Your Own

Twelve is complete now, and fun, but not really finished. Improving it could help you reinforce skills you learned earlier in this book, or could take you beyond what we've already done. Here are some ideas to get you started.

Add a start screen.

Break Twelve into two or more scenes, one that gives the user some instructions, and one for the game. A really amazing start screen could even play an animation of a move.

Add some sounds.

Give Twelve some sound effects, maybe a happy 'snick' for a successful move and something sadder for an unsuccessful one.

Add different visual cues.

The highlighted squares in *Add Visual Feedback*, on page 123 are just one possible way to give the player some feedback. Perhaps you can think of one you like better. What if the actual result of legal moves was shown, perhaps in a somewhat different color.

Refactor some methods.

There is quite a bit of repeated code in Twelve, especially in some methods in the `Game` class. See if you can tighten up the code, perhaps by rewriting the `move_is_legal?(square1, square2)` method and the `move(square1, square2)` methods so that one can be used by the other.

Let the player undo a move.

Make 'CTRL-U' or some other keypress undo the last move. To do this, before you execute a move, you could create and save a copy of the whole `@game` object.

Make a better game.

Try changing the rules of Twelve to make it easier, harder, or just more fun. What happens if there are four colors? Diagonal moves?

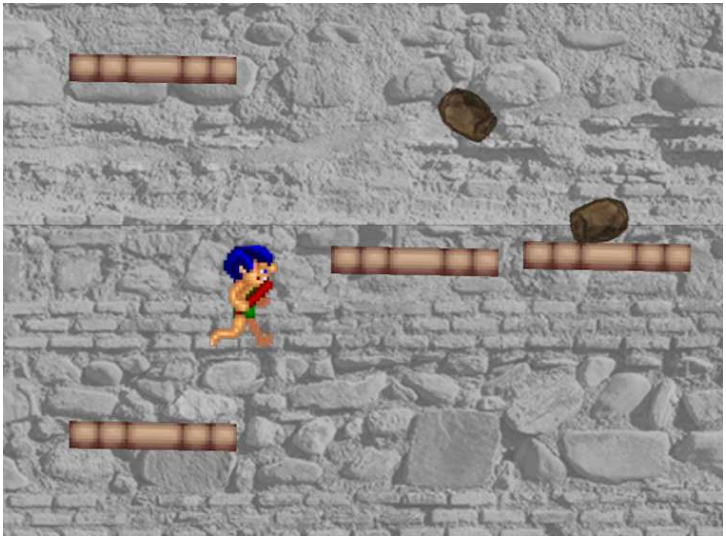
What's Next

We've now tackled a sprite-based game, and a puzzle game. Maybe you've come up with a great idea for a puzzle game of your own, and you want to get started. When you're ready, you can move on to the final game in this book. Our last game will be a scrolling plaformer, with a hero jumping between platforms and dodging boulders to escape a pit. We'll learn to use a physics engine to help us move objects more realistically, and a camera object so our entire scene can be bigger than what our window shows.

CHAPTER 7

Making a Platformer Game with Physics

Our next game, Escape, is a *platformer* where the player controls a character that is trying escape a pit. The character, named Chip, runs and jumps between platforms while avoiding boulders that fall, spinning and bouncing, from above.



When boulders hit Chip, he doesn't fall down but may be knocked to a lower platform, or even all the way to the bottom of the pit. Chip's ascent out of the pit is timed; a lower time is a better score.

Escape is a sprite game much like Sector Five. We'll make classes for each kind of sprite in the game: boulders, platforms, and Chip. The difference is that to make Chip jump and the boulders fall, bounce, and spin, we'll use a *physics engine*. As we add objects to our game, we describe their *properties* and then hand them over to the physics engine. The engine then takes over

moving the objects. For instance, in the Boulder class we provide some information: "Boulders are pretty heavy. They are affected by gravity and can rotate. They bounce off other objects like walls and platforms and have a certain amount of friction when they come in contact with other things." When we add boulders to our game we don't have to move them ourselves. Instead, the physics engine takes care of it. The boulders behave the way we said boulders should behave and when we draw them we ask the physics engine where they are.

When its time to make Chip run and jump by pressing keys on the keyboard, we don't move him directly. Instead we exert a *force* on him, and the physics engine does the calculations needed to figure out where he goes.

In this chapter, we:

- Use a physics engine to manage the motion of our objects.
- Add immovable objects to the game.
- Move objects in the game by applying forces.

We build Escape step by step. We start with the boulders, which fall, tumble, and bounce according to the laws of physics. We add platforms, walls and a floor which don't move, but which make the boulders bounce. Then we add Chip, who is our most complicated object. He is affected by gravity and can run and jump at our command. Finally, we add moving platforms that make the game more fun and challenging. In the next chapter, Chapter 8, *Making a Side-Scrolling Game*, on page 163, we'll improve Escape by making the pit deeper and adding a moving camera that follows the character.

Use a Physics Engine

Our physics engine is called Chipmunk, which we include in our project using the Ruby gem of the same name. The Chipmunk gem provides us with several classes which we can use to create objects. To understand how Chipmunk handles motion and interactions for us, let's look at some of these classes.

CP::Space

Our game will have a single instance of the CP::Space class. This space is the object that holds all the objects the the physics engine manages. When we create a new object, like a boulder or a platform, we add it to the space. The main method of the space is step() which moves the game move forward in time. We call the step() method of the space in the update() method of our game. The space also has some properties that affect all the objects we add to the space: gravity and damping.

CP::Body

Each object we add to the game has a body. The body object holds information about the position, velocity, mass, and rotational inertia of the object. For each object, we create the body, and then add the body to the space.

CP::Shape

Each object also needs a shape. The shape of an object determines how it interacts with all the other objects. The shapes of all the objects in our game will be *polygons*. In addition to the actual boundaries of the object, the shape keeps track of information about how the object will interact with other objects. *Friction* and *elasticity* are two properties of the shape that help determine what happens when two objects come in contact.

CP::Vec2

Many of the quantities that Chipmunk uses are *vectors*, with a horizontal part and a vertical part. This class allows us to create a single object with two components, x and y, to hold such a value.

The Laws of Physics

There are lots of laws of physics. There are laws about electricity and magnetism, laws about heat and temperature, and laws about the bending of light. The ones we're talking about in this chapter are Newton's three laws of motion. These laws are pretty famous, and you may have heard of them.

The first law states that an object in motion will remain in motion. Objects have inertia, and the more *mass* they have, the harder it is to stop them, or to get them moving.

Newton's second law states that if we want to get an object moving, or to change the motion of an object, we have to exert a *force* on it. An equation `F = ma`, governs this relationship. The `F` stands for force, the `m` for mass, and the `a` is acceleration. Understanding this equation helps us figure out how large to make the masses and forces in our game.

The third law concerns the interactions between objects. When one object pushes on another, it is pushed back in return with the same force. When a boulder knocks Chip off a platform, it will be pushed in the other direction.

Chipmunk works by applying these laws to all the objects which have been added to the physics space. By giving objects different masses, and applying forces to the objects, we can control what happens in the game.

Find the folder called `Escape_Starter` in the book files you downloaded earlier. Make a copy of that folder named `Escape`, and take a look through its contents.

There are a bunch of images we need to build Escape, along with a Ruby file for the game, escape.rb. Our game is still based on the Gosu::Window class, and so our initial setup is not very different from the games we've made already. The only change so far is that we require the Chipmunk gem in addition to Gosu.

We set up the physics engine in the initialize() of the Escape class by adding the space object that will hold and update all our physics objects. We also set a background image here for our game. Since we will be drawing it more than once to fill the space, we set its *tileability* option to true. The background image won't interact with any of the objects so we don't add it to the physics space.

Escape/Escape_1/escape.rb

```
require 'gosu'
require 'chipmunk'
class Escape < Gosu::Window
  attr_reader :space
  def initialize
    super(800,800)
    self.caption = 'Escape'
➤   @game_over = false
➤   @space = CP::Space.new
➤   @background = Gosu::Image.new('images/background.png', tileable: true)
  end
end

window = Escape.new
window.show
```

There are two properties of the space that we configure, damping and gravity. Damping is a number, which determines how much things will slow down on their own. A damping of 1.0 would mean objects in the space would never slow down. A damping of 0.9 means that each object in the space will lose ten percent of its velocity each second, if there are no other forces on it. We use a constant for this value, to make it easy to adjust. We did something like this on our own with the spaceship in *Move the Ship*, on page 44.

Gravity exerts a force on everything in the space. Chipmunk lets us set a value for gravity that points in any direction; it is represented by a Cp::Vec2 object that has a horizontal, or x, part and a vertical, or y, part. We want our gravity to point straight down, with only a vertical part but we may want to adjust its strength. To represent gravity we create a single constant, GRAVITY and set its value to 400.0. In the initialize() method of the Escape class, we set the value of @space.gravity to a vector which we create using the GRAVITY constant.

Escape/Escape_1/escape.rb

```
class Escape < Gosu::Window
```

```
➤   DAMPING = 0.90
➤   GRAVITY = 400.0
    attr_reader :space
    def initialize
      super(800,800)
      self.caption = 'Escape'
      @game_over = false
      @space = CP::Space.new
      @background = Gosu::Image.new('images/background.png', tileable: true)
➤     @space.damping = DAMPING
➤     @space.gravity = CP::Vec2.new(0.0, GRAVITY)
    end
```

Now any objects we add to our space will be pulled toward the bottom of the window. As soon as we add boulders to the space, they will start to fall.

In the update() method of our game, we tell our physics engine to *step* forward in time. While the update() method runs about sixty times per second, we want our physics to step forward in even smaller increments. When objects are moving quickly, they might overlap by a large amount in one sixtieth of a second. By having our physics engine update ten times every update(), or six hundred times per second, our physics will be more realistic. If our computer can't handle this rate, we can back off a bit, but Chipmunk has been written to be very quick and efficient. Here is the update() method of our game.

Escape/Escape_1/escape.rb

```
def update
  unless @game_over
    10.times do
      @space.step(1.0/600)
    end
  end
end
```

Our game is set up, with a space ready for us to add objects. Now let's add some boulders that tumble down from above.

Make Boulders Fall

In Chapter 3, *Creating a Sprite-based Game*, on page 39, you learned to make classes for each type of sprite in our game. You made classes for the player ship, for the enemies, for bullets and explosions. When we create sprites that will be moved by the physics engine, we still make classes, but those classes do things a little differently.

Bodies, Shapes, and Images

When we add a boulder to the game, we describe it by giving it some physical properties and a shape, and then we add it to the space. We also give it an image that matches its shape. If the image we choose and the shape we define match up reasonably well, the illusion of bouncing, spinning boulders is convincing. If they do not, the boulders look strange as they bounce off each other and off the platforms.

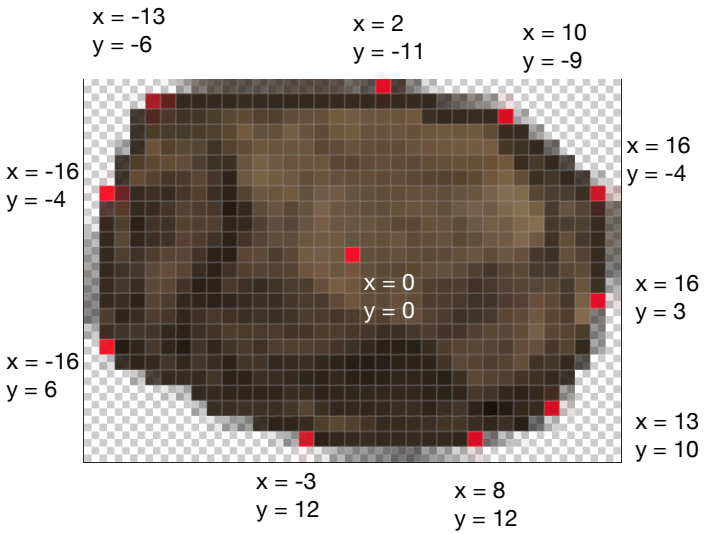
Make a new file, boulder.rb, to hold the Boulder class. In the initialize() method of the class, we first create the body. Bodies are initialized with two arguments, *mass*, and *rotational inertia*. Mass is a measure of resistance to changes in the velocity of the body, and rotational inertia is a measure of resistance to changes in rotation. We also set the position of the body, and its maximum speed, called v_limit.

Escape/Escape_1/boulder.rb

```
class Boulder
  SPEED_LIMIT = 500
  attr_reader :body, :width, :height
  def initialize(window, x, y)
    @body = CP::Body.new(400, 4000)
    @body.p = CP::Vec2.new(x, y)
    @body.v_limit = SPEED_LIMIT
  end
end
```

The maximum speed, or v_limit that we give each sprite in our game has a large effect on the play of our game. If you're looking to adjust the game to make gameplay easier or harder, changing the v_limit of some of your sprites is a good place to start.

Next we create the shape. In order to create a shape that matches our image, we take a very, very close look at the image, as shown in the following image. In this picture you can see the individual pixels. The grey and white checkerboard pattern is not part of the image but rather how many image editors indicate transparent pixels. The red dots aren't part of the image either; one marks the center, and the others are the vertices, or corners, of a *polygon* that will be the shape of the boulder object in the physics space.



We can find the position of each of the vertices, relative to the center, by counting pixels, remembering that the positive y direction is down. Then we put the ten vertices of the polygon, each a CP::Vec2 object, in an array called bounds. The vertices must be specified in a counter clockwise direction around the polygon and the polygon must be *convex*, meaning that it doesn't have any indentations.[1] If you break any of these rules you'll get an error like this one.

```
/Users/mark/Desktop/Escape/boulder.rb:24 in `initialize': The verts array
does not form a valid polygon! (ArgumentError)
```

Using the bounds array, we create the shape, which is attached to the body. We set two more properties of the shape, its *coefficient of friction*, named u, and its *elasticity*, named e. We also set @width and @height variables, which aren't used by the physics engine but which we use later to test whether Chip is standing on a boulder.

Then we finish by adding both the body and the shape to the space, and creating the @image variable. Add the following to the initialize() of the Boulder class.

Escape/Escape_1/boulder.rb

```
class Boulder
  FRICTION = 0.7
  ELASTICITY = 0.95
  SPEED_LIMIT = 500
```

1. http://en.wikipedia.org/wiki/Convex_and_concave_polygons

```
  attr_reader :body, :width, :height
  def initialize(window, x, y)
    @body = CP::Body.new(400, 4000)
    @body.p = CP::Vec2.new(x, y)
    @body.v_limit = SPEED_LIMIT
➤   bounds = [CP::Vec2.new(-13, -6),
➤             CP::Vec2.new(-16, -4),
➤             CP::Vec2.new(-16, 6),
➤             CP::Vec2.new(-3, 12),
➤             CP::Vec2.new(8, 12),
➤             CP::Vec2.new(13, 10),
➤             CP::Vec2.new(16, 3),
➤             CP::Vec2.new(16, -4),
➤             CP::Vec2.new(10, -9),
➤             CP::Vec2.new(2, -11)]
➤   shape = CP::Shape::Poly.new(@body, bounds, CP::Vec2.new(0, 0))
➤   shape.u = FRICTION
➤   shape.e = ELASTICITY
➤   @width = 32
➤   @height = 32
➤   window.space.add_body(@body)
➤   window.space.add_shape(shape)
➤   @image = Gosu::Image.new('images/boulder.png')
  end
```

In order to draw a boulder, we get the position and angle from the `@body` object. Since Chipmunk measures angles in radians, and Gosu measures angles in degrees, we need to convert the angle before we use it.

Escape/Escape_1/boulder.rb

```
def draw
  @image.draw_rot(@body.p.x, @body.p.y, 1, @body.angle * (180.0 / Math::PI))
end
```

The `initialize()` and `draw()` methods are the only methods we write for the `Boulder` class. There is no `move()` method, since the physics engine moves the boulders for us. Our boulders are now ready to be shaken loose and fall into the pit.

Adding Boulders

We want our boulders to fall from above, at random times and in random directions, to create a sense of crazy chaos. To do this we create boulders just off the top of the screen, at a random horizontal position. We also give each one a shove in a random direction. Start by adding the `boulder.rb` file in our project.

Escape/Escape_1/escape.rb

```
require 'gosu'
require 'chipmunk'
```

```
require_relative 'boulder'
```

Then create an empty array to hold the boulders in the initialize() method of the Escape class.

Escape/Escape_1/escape.rb

```
def initialize
  super(800,800)
  self.caption = 'Escape'
  @game_over = false
  @space = CP::Space.new
  @background = Gosu::Image.new('images/background.png', tileable: true)
  @space.damping = DAMPING
  @space.gravity = CP::Vec2.new(0.0, GRAVITY)
  @boulders = []
end
```

We add boulders at random times, about one frame in one hundred. This is similar to how we added enemies to Sector Five in Chapter 4, *Managing Lots of Sprites*, on page 57. Since we will likely want to adjust this frequency, we first create a constant, called BOULDER_FREQUENCY.

Escape/Escape_1/escape.rb

```
class Escape < Gosu::Window
  DAMPING = 0.90
  GRAVITY = 400.0
  BOULDER_FREQUENCY = 0.01
```

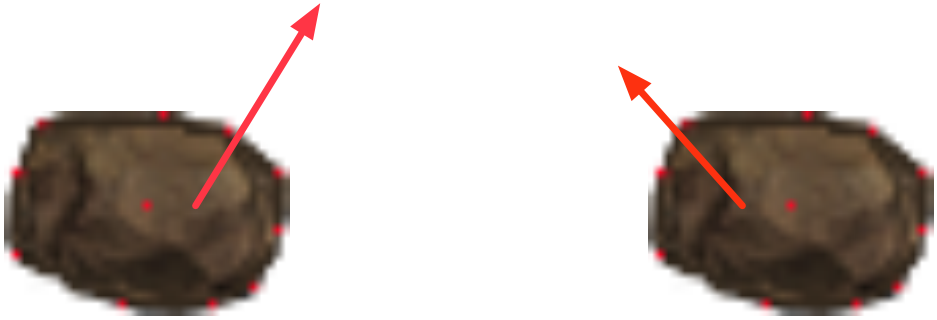
In the update() method of the Escape class, we add the boulders to the game. Each is just off the top of the window, at a random horizontal position between one quarter and three quarters of the window width.

Escape/Escape_1/escape.rb

```
def update
  unless @game_over
    10.times do
      @space.step(1.0/600)
    end
    if rand < BOULDER_FREQUENCY
      @boulders.push Boulder.new(self, 200 + rand(400), -20)
    end
  end
end
```

We give each boulder a push to get it started. Chipmunk gives us a method in the Cp::Body class, apply_impulse(), to apply an *instantaneous* force to an object. A force applied this way will get the object moving, but then lets it move on its own. When you kick a ball into the air, the kick gets it moving, but then gravity takes over. Your kick supplies an *impulse*. The apply_impulse() method

takes two arguments, both of type CP::Vec2. The first argument is *how much* force to apply. The second is *where* to apply the force, relative to the center of the object, as shown in the following image.



Each boulder gets a random push that makes it move and spin.

Applying the force away from the center makes the boulder spin. By making both arguments random, the player doesn't know what will happen next, and must keep a sharp eye out for falling boulders. Put the highlighted code at the end of the initialize() method of the Boulder class.

Escape/Escape_1/boulder.rb

```
def initialize(window, x, y)
  @body = CP::Body.new(400, 4000)
  @body.p = CP::Vec2.new(x, y)
  @body.v_limit = SPEED_LIMIT
  bounds = [CP::Vec2.new(-13, -6),
            CP::Vec2.new(-16, -4),
            CP::Vec2.new(-16, 6),
            CP::Vec2.new(-3, 12),
            CP::Vec2.new(8, 12),
            CP::Vec2.new(13, 10),
            CP::Vec2.new(16, 3),
            CP::Vec2.new(16, -4),
            CP::Vec2.new(10, -9),
            CP::Vec2.new(2, -11)]
  shape = CP::Shape::Poly.new(@body, bounds, CP::Vec2.new(0, 0))
  shape.u = FRICTION
  shape.e = ELASTICITY
  @width = 32
  @height = 32
  window.space.add_body(@body)
  window.space.add_shape(shape)
  @image = Gosu::Image.new('images/boulder.png')
➤ @body.apply_impulse(CP::Vec2.new(rand(100000) - 50000, 100000),
```

```
➤                                CP::Vec2.new(rand * 0.8 - 0.4, 0))
end
```

To draw our background and boulders, we add a `draw()` method to the `Escape` class. The background image doesn't fill the whole window, so we draw it twice.

Escape/Escape_1/escape.rb

```
def draw
  @background.draw(0,0,0)
  @background.draw(0,529,0)
  @boulders.each do |boulder|
    boulder.draw
  end
end
```

Run the game now, and boulders will fall from above, at random intervals, and in random directions.

Gravity makes the boulders accelerate toward the ground, which means they will go faster and faster until they reach the speed limit we set for them. The force of gravity on a boulder will be the value we set for gravity, `400`, multiplied by the mass of the boulder, which is also `400`. This force of `160,000` is divided by the mass of the boulder to find it acceleration, which is `400`. So each second the boulder will change its velocity by `400` in the downward direction. Two things act to slow down the boulders. First, the space has a `damping` that slows

everything down a bit. Also, the speed of our boulders is capped at 500 pixels per second, so they don't fall too fast for Chip to avoid them.

We now have boulders that fall from above, and down past the bottom of the window. The next objects we add are some platforms for our hero to climb, and for the boulders to bounce off.

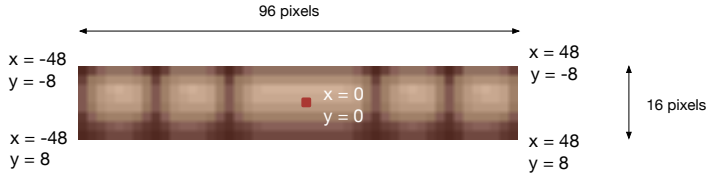
Make Stationary Walls and Platforms

Many of the objects in our game are *stationary*. When a boulder hits a platform it will bounce and tumble. But the platform won't move at all. We also add walls to keep things from falling off the edges of our game and a floor at the bottom. Chipmunk provides a way to add objects to the game that interact with all the other objects in the space but do not move themselves.

Static Bodies

When we made our boulders, we started by giving them a *body* that had a mass and a rotational inertia. Our platforms and walls don't have those properties; they simply don't move no matter what hits them. Chipmunk provides a special initializer for CP::Body that creates just such a *static* body, new_static(). Its possible to move an object with a static body by directly setting the values of its position, but the physics engine won't move it for you.

Our platforms are all the same. Their image is 96 pixels wide and 16 pixels tall, so their shape will be a rectangle - a polygon with four vertices.



When we add static objects to the physics space, we add their shape but not their body. Since the object won't move anyway, the physics space has no use for the body object. Here is the code for the platform class.

Escape/Escape_1/platform.rb

```
class Platform
  FRICTION = 0.7
  ELASTICITY = 0.8
  attr_reader :body, :width, :height
  def initialize(window, x, y)
    space = window.space
    @width = 96
    @height = 16
```

```
    @body = CP::Body.new_static
    @body.p = CP::Vec2.new(x,y)
    bounds = [CP::Vec2.new(-48, -8),
              CP::Vec2.new(-48, 8),
              CP::Vec2.new(48, 8),
              CP::Vec2.new(48, -8)]
    shape = CP::Shape::Poly.new(@body, bounds, CP::Vec2.new(0, 0))
    shape.u = FRICTION
    shape.e = ELASTICITY
    space.add_shape(shape)
    @image = Gosu::Image.new('images/platform.png')
  end

  def draw
    @image.draw_rot(@body.p.x, @body.p.y, 1, 0)
  end
end
```

We add a few platforms near the bottom to see how the boulders interact with the platforms. First, we require the platform class in escape.rb.

Escape/Escape_1/escape.rb

```
require 'gosu'
require 'chipmunk'
require_relative 'boulder'
require_relative 'platform'
```

We'll be adding a bunch of platforms to our game, so we add a new method, make_platforms() that creates all the platforms and returns an array of platforms. For now we add just four.

Escape/Escape_1/escape.rb

```
def make_platforms
  platforms = []
  platforms.push Platform.new(self,150,700)
  platforms.push Platform.new(self,320,650)
  platforms.push Platform.new(self,150,500)
  platforms.push Platform.new(self,470,550)
  return platforms
end
```

We call this method in the initialize() method of the Escape class, where we create a new instance variable, @platforms, to hold the array of platforms.

Escape/Escape_1/escape.rb

```
def initialize
  super(800,800)
  self.caption = 'Escape'
  @game_over = false
  @space = CP::Space.new
  @background = Gosu::Image.new('images/background.png', tileable: true)
```

```
    @space.damping = DAMPING
    @space.gravity = CP::Vec2.new(0.0, GRAVITY)
    @boulders = []
➤   @platforms = make_platforms
  end
```

In the draw() method of the Escape class, we draw the platforms.

Escape/Escape_1/escape.rb

```
def draw
  @background.draw(0,0,0)
  @background.draw(0,529,0)
  @boulders.each do |boulder|
    boulder.draw
  end
➤   @platforms.each do |platform|
➤     platform.draw
➤   end
end
```

When you run the game now, the boulders fall and hit the platforms. They bounce and spin. When they hit a corner they spin even more. This is the physics engine at work - we'd have had a hard time making the boulders behave so convincingly without it. In fact, we'd have had to write a physics engine!

Walls and a Floor

By adding walls and a floor, we keep our game objects in the window. All the boulders that fall stick around, and the player stops at the walls. By making the right wall a little shorter than the left, Chip can run out of the window at the top right, ending the game. We put the walls and floor just off the screen, so we don't need to draw them at all. We give them static bodies and shapes, and the physics engine takes care of the rest.

We use one class, called Wall, for both the floor and the walls. To set the shapes, we choose a rectangle that extends exactly to the edge of the screen. By making the position and size of the wall arguments of the initialize() method, we can create rectangular walls of any size.

Escape/Escape_1/wall.rb

```
class Wall
  FRICTION = 0.7
  ELASTICITY = 0.2
  def initialize(window, x, y, width, height)
    space = window.space
    @x = x
    @y = y
    @width = width
```

```
    @height = height
    @body = CP::Body.new_static()
    @body.p = CP::Vec2.new(x,y)
    @bounds = [CP::Vec2.new(-width / 2, -height / 2),
               CP::Vec2.new(-width / 2, height / 2),
               CP::Vec2.new(width / 2, height / 2),
               CP::Vec2.new(width / 2, -height / 2)]
    @shape = CP::Shape::Poly.new(@body, @bounds, CP::Vec2.new(0, 0))
    @shape.u = FRICTION
    @shape.e = ELASTICITY
    space.add_shape(@shape)
  end
end
```

The initialize() method is the only method of the Wall class. Once we create the walls and add them to the space, the physics engine takes care of all their interactions for us.

In the escape.rb file, we import the Wall class.

Escape/Escape_1/escape.rb

```
require 'gosu'
require 'chipmunk'
require_relative 'boulder'
require_relative 'platform'
require_relative 'wall'
```

Now we can add two walls and a floor to our game. Each makes a rectangle that comes just to the edge of the window and prevents boulders from bouncing or falling out of view. The left wall is 800 pixels tall, blocking the whole left side. The right wall is only 660 pixels tall, so that Chip can escape when he reaches the top of the pit. Add the following just after the code that adds the platforms, in the initialize() method of the Escape class.

Escape/Escape_1/escape.rb

```
@floor = Wall.new(self, 400,810,800,20)
@left_wall = Wall.new(self, -10, 400, 20,800)
@right_wall = Wall.new(self, 810,470,20,660)
```

When you run your game now, the boulders bounce off the walls as well as the platforms. When the boulders hit the ground they bounce a bit and come to rest. We've set the scene; now it's time for Chip to make his appearance.

Move a Character with Physics

Our hero, Chip, is an object in the physics space as well. He can be knocked off course by boulders and will be stopped by walls, platforms, and the floor.

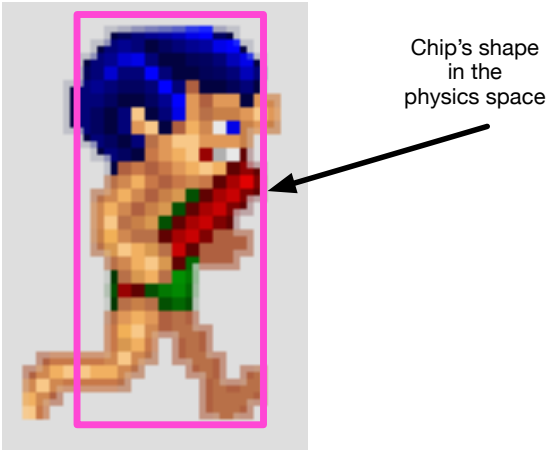
Chip will also run and jump when we press the arrow keys and space bar. We move him by exerting forces on him while the keys are pressed.

Create a new file to hold the Chip class, chip.rb. In initialize(), we first create Chip's physics properties, @body and @shape. Chip's mass will be 50, which is one eighth the mass of a boulder.

When our hero runs, we want him to *look* like he is running. To achieve this, we use a sprite sheet for his image, much like the one we used to animate the explosions in *Make Animated Explosions*, on page 71. This sprite sheet has eight images, and is shown in the following picture.

Chip stands tall throughout the game. Boulders can push him back, and even send him back to the ground, but they can't knock him down. To make sure Chip doesn't rotate, we set his rotational inertia to a seemingly strange thing, 100.0/0. Your math teacher probably told you you aren't allowed to divide by zero, but Chipmunk is ok with it. When Chipmunk sees a value like this it treats it like *infinity*. If Chip's rotational inertia is infinity, it means he won't rotate, no matter how much he is pushed.

Chip's shape, to keep things simple, is a rectangle. The height of the rectangle is the same as the height of the image, 64 pixels. The width of the rectangle is 20 pixels, which is an approximation of Chip's width. You can see a rectangle of this size superimposed on Chip in the following image.



After we add Chip to the physics space, we create a few more variables. The first, @action, keeps track of what Chip is doing so the appropriate images can be drawn. We draw Chip differently depending on whether he is running, jumping, or standing. We use the @image_index variable to show a sequence of images from the @images array when Chip is running. The @off_ground variable keeps track of whether Chip is on solid ground or not. If he's in the air he can still control his motion *some*, but much less than when he is on firm ground. We make four constants for the impulses we exert on Chip when he moves. RUN_IMPULSE and JUMP_IMPULSE are for when Chip is on the ground, and FLY_IMPULSE and (for lack of a better term) AIR_JUMP_IMPULSE are for when Chip's feet are off the ground. The jump impulses are much larger than the move impulses. This is due to the fact that we use the once-per-press button_down(key) method for jumping, and the once-per-frame button_down?(key) method for moving left and right.

Escape/Escape_1/chip.rb

```
class Chip
  RUN_IMPULSE = 600
  FLY_IMPULSE = 60
  JUMP_IMPULSE = 36000
  AIR_JUMP_IMPULSE = 1200
  SPEED_LIMIT = 400
  FRICTION = 0.7
  ELASTICITY = 0.2
  attr_accessor :off_ground
  def initialize(window, x, y)
    @window = window
    space = window.space
    @images = Gosu::Image.load_tiles('images/chip.png', 40, 65)
    @body = CP::Body.new(50, 100 / 0.0)
    @body.p = CP::Vec2.new(x, y)
```

```
    @body.v_limit = SPEED_LIMIT
    bounds = [CP::Vec2.new(-10, -32),
              CP::Vec2.new(-10, 32),
              CP::Vec2.new(10, 32),
              CP::Vec2.new(10, -32)]
    shape = CP::Shape::Poly.new(@body, bounds, CP::Vec2.new(0, 0))
    shape.u = FRICTION
    shape.e = ELASTICITY
    space.add_body(@body)
    space.add_shape(shape)
    @action = :stand
    @image_index = 0
    @off_ground = true
  end
end
```

Require the file in escape.rb.

Escape/Escape_1/escape.rb

```
require 'gosu'
require 'chipmunk'
require_relative 'boulder'
require_relative 'platform'
require_relative 'wall'
require_relative 'chip'
```

Then at the end of the initialize() method of the Escape class, we add Chip and place him near the bottom left corner of the window.

Escape/Escape_1/escape.rb

```
def initialize
  super(800,800)
  self.caption = 'Escape'
  @game_over = false
  @space = CP::Space.new
  @background = Gosu::Image.new('images/background.png', tileable: true)
  @space.damping = DAMPING
  @space.gravity = CP::Vec2.new(0.0, GRAVITY)
  @boulders = []
  @platforms = make_platforms
  @floor = Wall.new(self, 400,810,800,20)
  @left_wall = Wall.new(self, -10, 400, 20,800)
  @right_wall = Wall.new(self, 810,470,20,660)
  @player = Chip.new(self,70,700)
end
```

Now that our hero is in the scene, and has been added to the physics space, gravity will pull him down and boulders can knock him around. But we can't see Chip until we draw him.

Draw the Character

The image we draw for Chip each frame will depend on what he is doing. We're using the @action variable to keep track of this. We change the value of @action in the update() method in the next section; for now we assume @action has one of the following values: :stand, :run_right, :run_left, :jump_right, or :jump_left. In the draw() method of the Chip class, we check to see what our hero is doing, and draw the appropriate image.

When Chip is running, we draw a sequence of images. If we were to change the image every frame, or sixtieth of a second, Chip's feet would just be a blur. By adding 0.2 to the @image_index each frame, we draw the same image for five frames, then switch images. After the seventh image, we rotate back to the first using the modulo (%) operator. If Chip is jumping or standing we draw just the first image in the sprite sheet.

You may have noticed that in all the images in Chip's sprite sheet he is facing to the right. The draw_rot() method provides a way for us to flip those images horizontally when we draw them, by setting an optional parameter called scale_x to -1. The draw() method of the Chip class looks like the following.

Escape/Escape_1/chip.rb

```
def draw
  case @action
  when :run_right
    @images[@image_index].draw_rot(@body.p.x, @body.p.y, 2, 0)
    @image_index = (@image_index + 0.2) % 7
  when :stand, :jump_right
    @images[0].draw_rot(@body.p.x, @body.p.y, 2, 0)
  when :run_left
    @images[@image_index].draw_rot(@body.p.x, @body.p.y, 2, 0, 0.5, 0.5, -1, 1)
    @image_index = (@image_index + 0.2) % 7
  when :jump_left
    @images[0].draw_rot(@body.p.x, @body.p.y, 2, 0, 0.5, 0.5, -1, 1)
  else
    @images[0].draw_rot(@body.p.x, @body.p.y, 2, 0)
  end
end
```

In the draw() method of the Escape class, we draw Chip.

Escape/Escape_1/escape.rb

```
def draw
  @background.draw(0,0,0)
  @background.draw(0,529,0)
  @boulders.each do |boulder|
    boulder.draw
  end
```

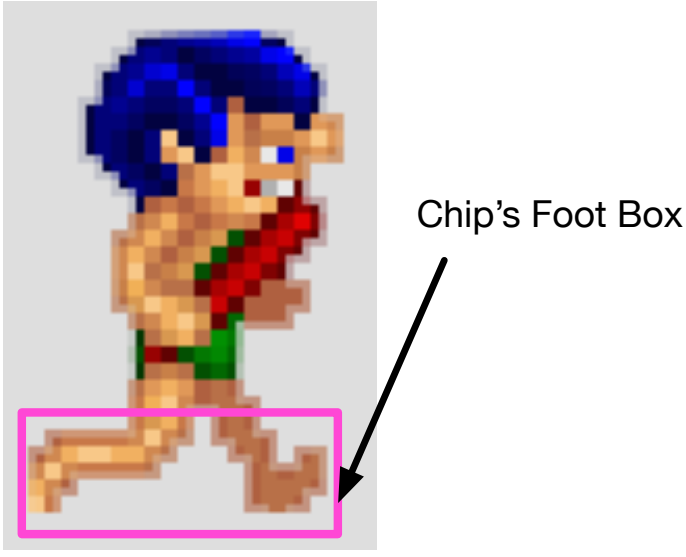
```
    @platforms.each do |platform|
      platform.draw
    end
➤   @player.draw
end
```

When you run your game now, Chip appears just above the floor, and gravity pulls him down. If you wait awhile, a boulder will knock Chip aside, one way or another. If you wait a long time, boulders will bury Chip. Let's get Chip moving so that he can escape the pit before he is buried.

Exert Forces on the Character

In Escape, we press the right or left arrow keys to make Chip run, and the space bar to make him jump. He jumps and runs *if* his feet are touching something solid. If he is airborne, we allow him to adjust his motion *a little*. He is a hero, after all. In update(), we first check if he is grounded, and set his @off_ground instance variable.

Create a new method in Chip, called touching?(footing). We're going to pass various objects as the footing parameter. Sometimes footing will be a boulder, but other times it will be a platform. In each case, we'll treat the object as a rectangle with a position given by x and y, a width, and a height. We'll also treat Chip's feet as a rectangle. We make the rectangle for Chip's feet extend a little beyond the rectangle that makes up his shape, as shown in the following picture.



We do this just to make sure that if the physics engine detects a boulder or platform under Chip, our method also detects the object and allows him to jump. If we cut it too close, the physics engine might stop him from moving while our touching?() method did not detect any overlap. This would make our game break, since Chip couldn't move. Here is the touching?(footing) method, that detects whether a given object is in contact with Chip's feet. It tests whether the rectangle of the footing object overlaps Chip's foot box.

Escape/Escape_1/chip.rb

```
def touching?(footing)
  x_diff = (@body.p.x - footing.body.p.x).abs
  y_diff = (@body.p.y + 30 - footing.body.p.y).abs
  x_diff < 12 + footing.width/2 and y_diff < 5 + footing.height / 2
end
```

Then we give Chip another method, check_footing(things). This method will set a value for @off_ground. This method takes as an argument an array of objects to check, which includes all the boulders and platforms. It also checks to see if Chip is touching the floor. If Chip's feet are touching any of these things, he can run and jump.

Escape/Escape_1/chip.rb

```
def check_footing(things)
  @off_ground = true
  things.each do |thing|
    @off_ground = false if touching?(thing)
  end
  if @body.p.y > 765
    @off_ground = false
  end
end
```

To make Chip run left and right, we don't just move him, like we did the spaceship. Instead we give him a push. We push him as long as the arrow keys are being pressed. In the update() method of Escape, add the following code.

Escape/Escape_1/escape.rb

```
if button_down?(Gosu::KbRight)
    @player.move_right
elsif button_down?(Gosu::KbLeft)
  @player.move_left
else
  @player.stand
end
```

We use the button_down() method to let chip jump, checking for the space bar. In this method we also let the player quit the game with the 'Q' key.

Escape/Escape_1/escape.rb
```
def button_down(id)
  if id == Gosu::KbSpace
    @player.jump
  end
  if id == Gosu::KbQ
    close
  end
end
```

This gives us some methods to implement in the Chip class. In each of the methods move_right() and move_left() we do two things. The first is that we set the variable @action, so that Chip is drawn correctly. The second is that we push Chip in the appropriate direction with the apply_impulse() method. We apply different impulses to Chip depending on whether his feet are touching the ground.

Escape/Escape_1/chip.rb
```
def move_right
  if @off_ground
    @action = :jump_right
    @body.apply_impulse(CP::Vec2.new(FLY_IMPULSE, 0), CP::Vec2.new(0, 0))
  else
    @action = :run_right
    @body.apply_impulse(CP::Vec2.new(RUN_IMPULSE,0 ), CP::Vec2.new(0, 0))
  end
end

def move_left
  if @off_ground
    @action = :jump_left
    @body.apply_impulse(CP::Vec2.new(-FLY_IMPULSE, 0), CP::Vec2.new(0, 0))
  else
    @action = :run_left
    @body.apply_impulse(CP::Vec2.new(-RUN_IMPULSE, 0), CP::Vec2.new(0, 0))
  end
end
```

In the jump() method the impulse we apply is much larger, since it will only be applied once when the space bar is pressed, and it has to get Chip moving up to the next platform.

Escape/Escape_1/chip.rb
```
def jump
  if @off_ground
    @body.apply_impulse(CP::Vec2.new(0, -AIR_JUMP_IMPULSE), CP::Vec2.new(0, 0))
  else
    @body.apply_impulse(CP::Vec2.new(0, -JUMP_IMPULSE), CP::Vec2.new(0, 0))
    if @action == :left
      @action = :jump_left
```

```
    else
      @action = :jump_right
    end
  end
end
```

The stand() method doesn't do anything to Chip but set his @action variable.

Escape/Escape_1/chip.rb

```
def stand
  @action = :stand unless off_ground
end
```

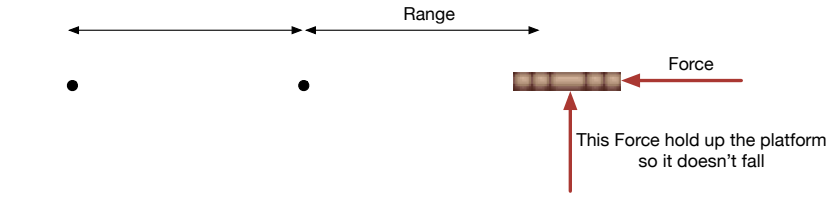
Run the game. Chip can run and jump, and looks like he is running and jumping. To complete the game we add a mix of moving and stationary platforms so that there are a few paths Chip can follow to escape the pit.

Add Moving Platforms

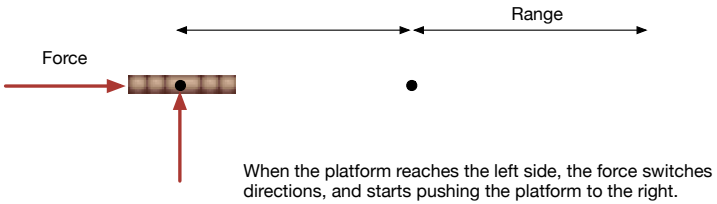
Moving platforms make our game more interesting and challenging. They require the player to think ahead, and to try to predict where the platforms will be when Chip reaches them. Some of our platforms will move side to side, and some will move up and down.

Adding moving platforms to our physics game requires a little creativity. Platforms that move like these don't really occur in nature. We could just *move* them ourselves by changing their position, but the result wouldn't be good. They wouldn't interact correctly with the player and the boulders. Instead, we place them in the space, and figure out what *forces* are needed to hold them up and move them back and forth. Then we cheat the physics engine, just a little, to keep them where we want them.

Our moving platforms will be represented by one class, MovingPlatform. A variable @direction will keep track of whether a particular platform moves up and down or side to side. Each platform will have a center position and a *range*. When the platform reaches a distance equal to the range away from the center, we start pushing it back toward the center.



The horizontal force will push the platform to the left until it reaches the dot on the left.



We push the platform all the way back through the center position until it reaches a distance from the center equal to the range on the other side. And then we repeat. The platform moves back and forth, turning around at a point a little past the declared range. By adjusting constants like the range, the force, and the maximum platform velocity, we can make the platforms move the way we want.

Start by making a new file, moving_platform.rb, for the new class. Include the file in the game.

Escape/Escape_1/escape.rb

```
require 'gosu'
require 'chipmunk'
require_relative 'boulder'
require_relative 'platform'
require_relative 'wall'
require_relative 'chip'
require_relative 'moving_platform'
```

The initialize() method of the MovingPlatform class takes a position given by x and y, along with the direction of travel and a range as parameters. We save the x and y arguments as @x_center and @y_center. The moving platform is very heavy compared to a boulder or Chip, and is unable to rotate. Since Chip has to jump off these platforms, we give the platform @width and @height variables.

Escape/Escape_1/moving_platform.rb

```
class MovingPlatform
  FRICTION = 0.7
  ELASTICITY = 0.8
```

```
  SPEED_LIMIT = 40
  attr_reader :body,:width,:height
  def initialize(window, x, y, range, direction)
    space = window.space
    @window = window
    @x_center = x
    @y_center = y
    @direction = direction
    @range = range
    @body = CP::Body.new(50000, 100.0 / 0)
    @width = 96
    @height = 16
    @body.v_limit = SPEED_LIMIT
  end
end
```

The initial position of the platform depends on whether the platform moves vertically or horizontally. By placing the platform initially at a position a little beyond its range, we ensure that it will start moving toward the center at the beginning of the game. We also create an instance variable, `@move`, that keeps track of which way the platform is moving.

Escape/Escape_1/moving_platform.rb

```
def initialize(window, x, y, range, direction)
  space = window.space
  @window = window
  @x_center = x
  @y_center = y
  @direction = direction
  @range = range
  @body = CP::Body.new(50000, 100.0 / 0)
  @width = 96
  @height = 16
  @body.v_limit = SPEED_LIMIT
➤ if @direction == :horizontal
➤   @body.p = CP::Vec2.new(x + range + 100, y)
➤   @move = :right
➤ else
➤   @body.p = CP::Vec2.new(x, y + range + 100)
➤   @move = :down
➤ end
end
```

To finish the `initialize()` method of the `MovingPlatform` class, we create the shape, which is the same as the shape of our stationary platforms. We add the body and shape to the physics space, and apply an upward force to keep our moving platforms from falling to the ground.

Escape/Escape_1/moving_platform.rb

```
def initialize(window, x, y, range, direction)
```

```
    space = window.space
    @window = window
    @x_center = x
    @y_center = y
    @direction = direction
    @range = range
    @body = CP::Body.new(50000, 100.0 / 0)
    @width = 96
    @height = 16
    @body.v_limit = SPEED_LIMIT
    if @direction == :horizontal
      @body.p = CP::Vec2.new(x + range + 100, y)
      @move = :right
    else
      @body.p = CP::Vec2.new(x, y + range + 100)
      @move = :down
    end
➤   bounds = [CP::Vec2.new(-48, -8),
➤             CP::Vec2.new(-48, 8),
➤             CP::Vec2.new(48, 8),
➤             CP::Vec2.new(48, -8)]
➤   shape = CP::Shape::Poly.new(@body, bounds, CP::Vec2.new(0, 0))
➤   shape.u = FRICTION
➤   shape.e = ELASTICITY
➤   space.add_body(@body)
➤   space.add_shape(shape)
➤   @image = Gosu::Image.new('images/platform.png')
➤   @body.apply_force(CP::Vec2.new(0, -20000000), CP::Vec2.new(0, 0))
  end
```

The previous image shows the strategy we use to move the platforms. It involves pushing them in one direction with a force, until the platform gets a certain distance, the range, away from its center position. Then the force switches and it gets pushed the other way. Now we implement that plan as code.

There are four possibilities for the way our platform is moving. It could be moving horizontally or vertically. If it's moving horizontally it can be moving right or left, and if its moving vertically it can be moving up or down. Once we create and add forces to the platform, the physics engine takes care of moving it. We need to readjust the forces each time the platform moves beyond its range.

We look first at a platform moving horizontally to the right. If it has moved far enough, we change its `@move` variable to `:left`. We also call the `reset_forces()` method, which removes all the forces on the platform. Then we apply two new forces. We have to reapply the upward force that balances gravity and we

apply a new force to the left. This force stops the platform over a short distance and then pushes it back to the left until it reaches the other end of the range.

Escape/Escape_1/moving_platform.rb

```
def move
  case @direction
  when :horizontal
    if @body.p.x > @x_center + @range && @move == :right
      @body.reset_forces
      @body.apply_force(CP::Vec2.new(0, -20000000), CP::Vec2.new(0, 0))
      @body.apply_force(CP::Vec2.new(-20000000, 0), CP::Vec2.new(0, 0))
      @move = :left
    end
end
```

Once it is moving left, we check to see if it has reached the other end of the range, then adjust the forces so that it starts moving to the right. We also add one line of code to cheat the physics engine, just a little. See if you can spot that line of code.

Escape/Escape_1/moving_platform.rb

```
def move
  case @direction
  when :horizontal
    if @body.p.x > @x_center + @range && @move == :right
      @body.reset_forces
      @body.apply_force(CP::Vec2.new(0, -20000000), CP::Vec2.new(0, 0))
      @body.apply_force(CP::Vec2.new(-20000000, 0), CP::Vec2.new(0, 0))
      @move = :left
    elsif @body.p.x < @x_center - @range && @move == :left
      @body.reset_forces
      @body.apply_force(CP::Vec2.new(0, -20000000), CP::Vec2.new(0, 0))
      @body.apply_force(CP::Vec2.new(20000000, 0), CP::Vec2.new(0, 0))
      @move = :right
    end
    @body.p.y = @y_center
end
```

For an object moving left and right we added the single line of code: `@body.p.y = @y_center`. This line ensures that when a boulder pushes the platform down a tiny bit, we move it back up. Without that adjustment, even though we apply a force that counteracts gravity, the platform would eventually be pushed down by the unceasing rain of boulders.

For a vertically moving platform, we do a very similar thing. When it is moving down, we check to see if it has moved down far enough, and start pushing it up. Only one force is applied, which is larger than the force of gravity.

Escape/Escape_1/moving_platform.rb

```
def move
  case @direction
  when :horizontal
    if @body.p.x > @x_center + @range && @move == :right
      @body.reset_forces
      @body.apply_force(CP::Vec2.new(0, -20000000), CP::Vec2.new(0, 0))
      @body.apply_force(CP::Vec2.new(-20000000, 0), CP::Vec2.new(0, 0))
      @move = :left
    elsif @body.p.x < @x_center - @range && @move == :left
      @body.reset_forces
      @body.apply_force(CP::Vec2.new(0, -20000000), CP::Vec2.new(0, 0))
      @body.apply_force(CP::Vec2.new(20000000, 0), CP::Vec2.new(0, 0))
      @move = :right
    end
    @body.p.y = @y_center
  when :vertical
    if @body.p.y > @y_center + @range && @move == :down
      @body.reset_forces
      @body.apply_force(CP::Vec2.new(0, -25000000), CP::Vec2.new(0, 0))
      @move = :up
  end
end
```

If the platform is moving up, we check to see when it has gone far enough, and change the force so that it is smaller than the force of gravity, allowing it to start moving downward. We add a line of code to keep it horizontally in place.

Escape/Escape_1/moving_platform.rb

```
when :vertical
  if @body.p.y > @y_center + @range && @move == :down
    @body.reset_forces
    @body.apply_force(CP::Vec2.new(0, -25000000), CP::Vec2.new(0, 0))
    @move = :up
  elsif @body.p.y < @y_center - @range && @move == :up
    @body.reset_forces
    @body.apply_force(CP::Vec2.new(0, -15000000), CP::Vec2.new(0, 0))
    @move = :down
  end
  @body.p.x = @x_center
end
```

The moving platforms need a draw method. We get the location of the platform from the @body variable.

Escape/Escape_1/moving_platform.rb

```
def draw
  @image.draw_rot(@body.p.x, @body.p.y , 0, 1)
end
```

To give Chip a path to the top of the pit, we add a mix of stationary and moving platforms. Feel free to adjust them to suit you, the ones here are just an example. In the next chapter, Chapter 8, *Making a Side-Scrolling Game*, on page 163, we'll tackle random placement of the platforms. The platforms are added to the @platforms array in the make_platforms() method, just after the four we added earlier.

Escape/Escape_1/escape.rb

```
def make_platforms
  platforms = []
  platforms.push Platform.new(self,150,700)
  platforms.push Platform.new(self,320,650)
  platforms.push Platform.new(self,150,500)
  platforms.push Platform.new(self,470,550)
➤ platforms.push MovingPlatform.new(self,580,600,70,:vertical)
➤ platforms.push Platform.new(self,320,440)
➤ platforms.push Platform.new(self,600,150)
➤ platforms.push Platform.new(self,700,450)
➤ platforms.push Platform.new(self,580,300)
➤ platforms.push MovingPlatform.new(self,190,330,50,:vertical)
➤ platforms.push MovingPlatform.new(self,450,230,70,:horizontal)
➤ platforms.push Platform.new(self,750,140)
➤ platforms.push Platform.new(self,700,700)
  return platforms
end
```

Each time the update() method runs we want to call the move() method on each of the moving platforms. Our @platforms array has both moving and stationary platforms, and stationary platforms don't have a move() method. In the update() we check to see if each platform has a move() method, and move those that do.

Escape/Escape_1/escape.rb

```
@platforms.each do |platform|
  platform.move if platform.respond_to?(:move)
end
```

Run the game now. We've got some platforms moving around, and some standing still. Chip can jump between them, until he gets hit by the boulders. Once we time his climb out of the pit, we'll have a game.

Time the Climb

Players will make chip run and jump between the platforms, and run off the right edge of the window near the top. We time how long it takes them, using the Gosu.milliseconds() method we first used way back in Whack-A-Ruby.

We write the score on the screen using an instance of Gosu::Font. In the initialize() method of the Escape class, we create this instance. We'll also add an image that shows the player where Chip can exit the window. The completed initialize() method, with the new code highlighted, looks like this.

Escape/Escape_1/escape.rb

```
def initialize
  super(800,800)
  self.caption = 'Escape'
  @game_over = false
  @space = CP::Space.new
  @background = Gosu::Image.new('images/background.png', tileable: true)
  @space.damping = DAMPING
  @space.gravity = CP::Vec2.new(0.0, GRAVITY)
  @boulders = []
  @platforms = make_platforms
  @floor = Wall.new(self, 400,810,800,20)
  @left_wall = Wall.new(self, -10, 400, 20,800)
  @right_wall = Wall.new(self, 810,470,20,660)
  @player = Chip.new(self,70,700)
➤ @sign = Gosu::Image.new('images/exit.png')
➤ @font = Gosu::Font.new(40)
end
```

The game ends when Chip leaves the window. Since he can only leave above the top of the right hand wall, we just have to check his x position. In the update() method of the Escape class, we check to see if the game is over.

Escape/Escape_1/escape.rb

```
def update
  unless @game_over
    10.times do
      @space.step(1.0/600)
    end
    if rand < BOULDER_FREQUENCY
      @boulders.push Boulder.new(self, 200 + rand(400), -20)
    end
    @player.check_footing(@platforms + @boulders)
    @platforms.each do |platform|
      platform.move if platform.respond_to?(:move)
    end
    if button_down?(Gosu::KbRight)
        @player.move_right
    elsif button_down?(Gosu::KbLeft)
      @player.move_left
    else
      @player.stand
    end
➤   if @player.x > 820
➤     @game_over = true
```

```
      @win_time = Gosu.milliseconds
    end
  end
end
```

Then in the draw() method we draw the exit sign. We draw the timer if the game is not over, and the final time if the game is over. Plus, we add the words "Game Over" in the middle of the screen. All in a lovely green font.

Escape/Escape_1/escape.rb

```
def draw
  @background.draw(0,0,0)
  @background.draw(0,529,0)
  @boulders.each do |boulder|
    boulder.draw
  end
  @platforms.each do |platform|
    platform.draw
  end
  @player.draw
  @sign.draw(650,30,1)
  if @game_over == false
    @seconds = (Gosu.milliseconds / 1000).to_i
    @font.draw("#{@seconds}", 10,20,3,1,1,0xff00ff00)
  else
    @font.draw("#{@win_time/1000}", 10,20,3,1,1,0xff00ff00)
    @font.draw("Game Over",200, 300, 3,2,2,0xff00ff00)
  end
end
```

Time to see if you can help Chip escape the pit. If its too easy or too hard, you can make some changes.

Make it Your Own

In Escape, we use a physics engine to move our objects. To do this, we specify a large number of properties for each object we added to the space. Each object has a mass, rotational inertia, friction, elasticity, and maximum velocity. The physics space itself has gravity and damping properties. If you want to understand the way these properties affect the game, the best way to learn is to change them and see what happens. Make a copy of your game before you start.

Change some Physics Properties

Play with some of the physics properties. See what happens if all friction is removed, or if all the elasticities are zero. See what happens when all the v_limit values are removed. When you have a feel for what they all do, try setting them to make the game as fun and challenging as you can.

Add a Treasure

Try adding a treasure, maybe on one of the platforms, that the Chip has to retrieve before he can get out of the pit. You can add a third wall blocking his exit, then move it out of the way when he gets the treasure.

Add a Power-up

A power-up is an object that gives Chip a power for some amount of time. Maybe it makes him jump higher, or blow past boulders. If you give Chip a huge mass boulders will just bounce off him. You can change the mass of a body, by writing something like the following.

```
@body.m = 500
```

If you increase Chip's mass by a factor of ten you need to increase any forces you exert on him by the same factor, or he'll jump like a lead brick.

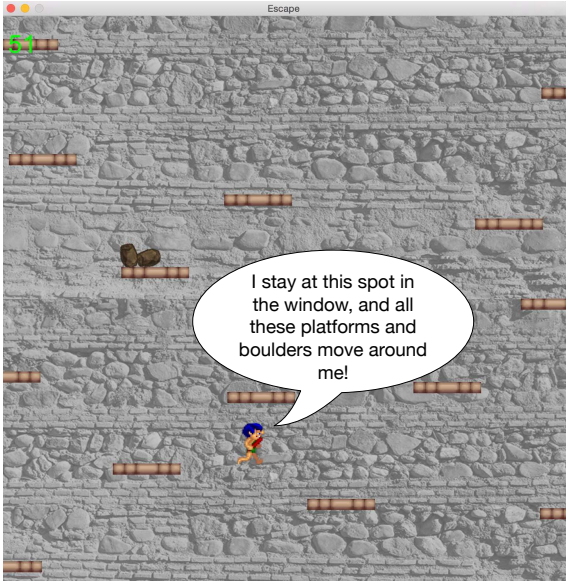
What's Next

We're not done with Chip yet. In the next chapter, we'll make his pit deeper. Then it won't fit in our window, so we'll make the window scroll to follow Chip's movement. We'll randomize the placement of the platforms, so every game is different. And we'll add quakes, to shake things up and keep those boulders moving.

CHAPTER 8

Making a Side-Scrolling Game

In our initial version of Escape, a hero named Chip jumps from platform to platform to escape a pit, while boulders fall around him. It's fun, but the pit is a little small, and the game too short. We could try making all the objects smaller so that more things fit in the window, but then it would be hard to see the action. Instead, we make the pit larger than the window, and we show just part of the pit in our window at any one time.



This type of game is called *side-scrolling*, since we're watching from the side as the window scrolls to follow Chip's movements. We keep the game focused on Chip and his actions, by adjusting what part of the pit the player can see with an object we call the *camera*.

Many games scroll the player's view, in either one or both directions. Its a common technique that makes the player feel as if they are moving a character or vehicle within a large world of which they see only a local piece.

In our first version of Escape, the platforms were in the same place for every game. In this version, we create a more random arrangement of platforms that is different each time we play.

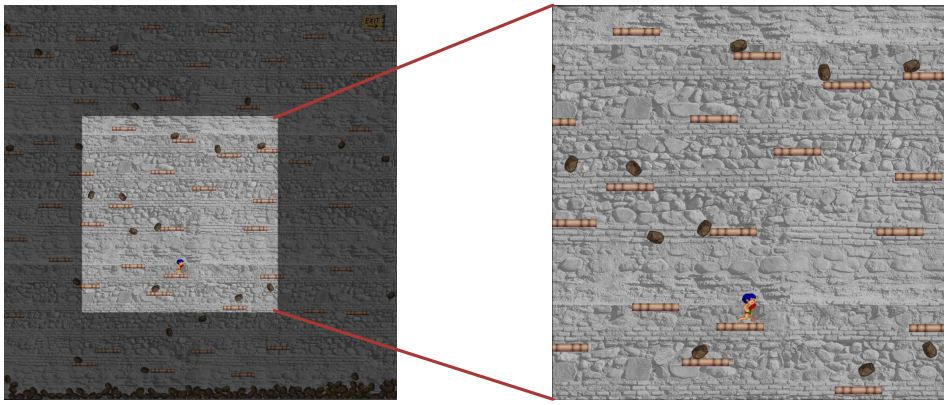
We also have a little fun with the camera, and introduce quakes into the game that shake the whole scene and make a bunch of boulders come down all at once. Quakes also get any boulders that have come to rest on the platforms moving again. In this chapter you learn to:

- Create a side-scrolling game where the game space is larger than the window.
- Generate random arrangements of platforms that are both challenging and possible for the player to beat.
- Use the camera object to add another feature, quakes, into our game.

When we're done, our platformer game will more challenging and fun. You'll have learned how to make a side-scrolling game, which can help make your games bigger and better.

Use a Camera

In this new version of Escape, the pit is much bigger than our game window. We make the pit a square 1600 pixels on a side, while the window is 800 pixels square, so we only see one quarter of the game inside the window at any one time. The part of the game space we show will depend on Chip's location.



All the game objects are in the physics space.

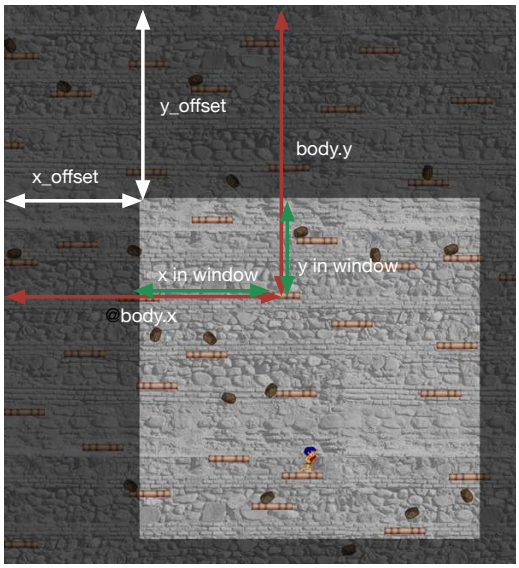
The camera draws only some of the game window, based on Chip's location.

It's as if a camera is following Chip around as he climbs out of the pit. When Chip gets close to an edge of the pit, or to the top or bottom, the camera doesn't follow him in the same way. Instead, the camera stops so that the whole camera view remains inside the pit.

The Camera Class

Start by making a copy of the `Escape` folder you made in the last chapter. If you don't have that you can start with the finished project in the `Escape_1` folder in the downloads for this book. We make only small changes to the sprite classes in the project, so we keep those as part of the new game. Create a new file for our camera class called `camera.rb`.

We use the camera to help us draw all the objects in the pit. The camera calculates two numbers, that *shift* the positions of the objects as we draw them in our window. These numbers are called `x_offset` and `y_offset`.



Calculating the position of an object in the window.

x = body.x - x_offset

y = body.y - y_offset

The physics space tracks the location of each object in the space, and we use that information along with the camera offsets to draw the objects in the window. The camera needs to know the sizes for the physics space and the window. In the initialize() of Camera, we store these values, and also calculate the maximum offsets. We use these maximum offsets to keep the area our camera shows within the bounds of the pit when Chip is near an edge.

Escape/Escape_2/camera.rb

```
class Camera
  attr_reader :x_offset,:y_offset

  def initialize(window, space_height, space_width)
    @window = window
    @space_height = space_height
    @window_height = window.height
    @space_width = space_width
    @window_width = window.width
    @x_offset_max = space_width - @window_width
    @y_offset_max = space_height - @window_height
  end
end
```

Our camera also needs a method that calculates the offsets based on the location of a sprite. We make that method as general as possible, since we might want to reuse our Camera class in another project. The center_on() method takes three arguments. One is a sprite, which needs to have getter methods for x and y. The others are two numbers that describe where in the window the sprite should be.

The second argument, right_margin, is the distance the sprite should be from the right edge of the window. The third, bottom_margin, is its ideal distance from the bottom. First we calculate the offsets as shown in the preceding image. Then we make sure the camera stays within the bounds of the pit.

Escape/Escape_2/camera.rb

```
def center_on(sprite, right_margin, bottom_margin)
  @x_offset = sprite.x - @window_width + right_margin
  @y_offset = sprite.y - @window_height + bottom_margin
  @x_offset = @x_offset_max if @x_offset > @x_offset_max
  @x_offset = 0 if @x_offset < 0
  @y_offset = @y_offset_max if @y_offset > @y_offset_max
  @y_offset = 0 if @y_offset < 0
end
```

In order to draw the images in the camera's view we use a class method of the Gosu::Window class, called translate(), that takes our offsets as parameters and then applies the offsets to any drawing operations in a *block* of code. This ability to write a method that wraps a block of code is a great feature of the Ruby language, and we take advantage of it here. We add an instance method called view() to our Camera class that uses the translate() method, along with our calculated offsets.

Escape/Escape_2/camera.rb

```
def view
  @window.translate(-@x_offset, -@y_offset) do
    yield
  end
end
```

The yield statement indicates that the @window.translate() method, and so also our view() method, let us pass in a block of code. Each line in the block we supply will be executed in the *context* of the @window.translate() method. What this means is that any draw statements we put in the block will be translated by the camera. Which is just what we need.

Since the translate() method *adds* rather than subtracts the offsets to the position of any object drawn in its block, we send it -@x_offset and -@y_offset as parameters.

Drawing with the Camera

We draw all the objects in the pit using the camera. This means that we put their draw commands in the block of the @camera.view method. These objects include the background images, platforms, boulders, and Chip. The score is not part of our game, and should stay in place in our window as Chip moves.

The initialize() method of our new Escape class is much the same as the one from our previous game. The first difference is that the walls and floor box in a space that is 1600 pixels square, which defines the size of our pit. Chip needs to be placed near the bottom of the pit, so his initial vertical position is different. We add the camera object, and center the camera on Chip. We also add music, so Chip has an upbeat song to help him jump. You can copy this method from the previous game, and change only the highlighted lines.

Escape/Escape_2/escape2.rb

```
require 'gosu'
require 'chipmunk'
require_relative 'camera'
require_relative 'boulder'
require_relative 'platform'
require_relative 'wall'
require_relative 'chip'
require_relative 'moving_platform'

class Escape < Gosu::Window
  DAMPING = 0.90
  GRAVITY = 400.0
  BOULDER_FREQUENCY = 0.01
  attr_reader :space
  def initialize
    super(800,800)
    self.caption = 'Escape'
    @space = CP::Space.new
➤   @player = Chip.new(self, 70, 1500)
➤   @camera = Camera.new(self, 1600, 1600)
➤   @camera.center_on(@player, 400, 200)
    @game_over = false
    @background = Gosu::Image.new('images/background.png', tileable: true)
    @space.damping = DAMPING
    @space.gravity = CP::Vec2.new(0.0, GRAVITY)
    @boulders = []
    @platforms = make_platforms
➤   @floor = Wall.new(self, 800, 1610, 1600, 20)
➤   @left = Wall.new(self,-10, 800, 20, 1600)
➤   @right = Wall.new(self, 1610, 870, 20, 1460)
➤   #END_HIGHLGIHT
➤   @sign = Gosu::Image.new('images/exit.png')
➤   @font = Gosu::Font.new(40)
➤   @font_small = Gosu::Font.new(18)
➤   @music = Gosu::Song.new('sounds/zanzibar.ogg')
➤   @music.play(true)
  end
end

window = Escape.new
```

```
window.show
```

In the draw() method of the Escape class, we use our camera to help draw our game. To do this we separate our objects into two groups. The first group consists of all the objects viewed by the camera. This includes Chip, the platforms, the boulders, and the exit sign. Moving the background image is an important part of our illusion, so we draw it with the camera also. In order to fill the whole pit, we draw the background image eight times, tiling it to fill the space. We include all these drawing statements in the block we pass into @camera.view.

Escape/Escape_2/escape2.rb

```
def draw
  @camera.view do
    (0..3).each do |row|
      (0..1).each do |column|
        @background.draw(799 * column, 529 * row, 0)
      end
    end
    @sign.draw(1450, 30, 2)
    @player.draw
    @boulders.each do |boulder|
      boulder.draw
    end
    @platforms.each do |platform|
      platform.draw
    end
  end
end
```

We can also draw objects outside this method. Our timer and ending messages should not move with the camera, so their draw statements are outside the @camera.view block. When the game is over we add some credits to the screen along with the “Game Over” message, using a separate method, draw_credits.

Escape/Escape_2/escape2.rb

```
def draw
  @camera.view do
    (0..3).each do |row|
      (0..1).each do |column|
        @background.draw(799 * column, 529 * row, 0)
      end
    end
    @sign.draw(1450, 30, 2)
    @player.draw
    @boulders.each do |boulder|
      boulder.draw
    end
    @platforms.each do |platform|
```

```
        platform.draw
      end
    end
    if @game_over == false
      @font.draw("#{@seconds}", 10, 20, 3, 1, 1, 0xff00ff00)
    else
      @font.draw("#{@win_time/1000}", 10, 20, 3, 1, 1, 0xff00ff00)
      draw_credits
    end
  end
```

The draw_credits() method writes our credits, one line at a time, onto the screen.

Escape/Escape_2/escape2.rb

```
def draw_credits
  color = 0xff00ff00
  @font.draw('Game Over',240, 150, 3, 2, 2, color)
  @font_small.draw('Images from the SpriteLib Collection',
                   100, 300, 3, 2, 2, color)
  @font_small.draw('by WidgetWorx under the terms of the',
                   100, 350, 3, 2, 2, color)
  @font_small.draw('Common Public License.',
                   100, 400, 3, 2, 2, color)
  @font_small.draw('Music:  Zanzibar, by Kevin MacLeod',
                   100, 500, 3, 2, 2, color)
  @font_small.draw('(incompetech.com)',
                   100, 550, 3, 2, 2, color)
  @font_small.draw('Licensed under',
                   100, 600, 3, 2, 2, color)
  @font_small.draw('Creative Commons: By Attribution 3.0',
                   100, 650, 3, 2, 2, color)
  @font_small.draw('http://creativecommons.org/licenses/by/3.0/',
                   100, 700, 3, 2, 2, color)
end
```

In the update() method of the Escape class, we center the camera on the player. We add boulders, just as we did in the first Escape game, but we have to spread them out over the 1600 pixels of our pit. We tell our physics space to update everything ten times. The lines of code that are added or changed from the update() method you wrote in the preceding chapter are highlighted in the following code.

Escape/Escape_2/escape2.rb

```
def update
  @camera.center_on(@player, 400, 200)
  if @game_over == false
    @seconds = (Gosu.milliseconds / 1000).to_i
    10.times do
      @space.step(1.0/600)
    end
```

```
    if rand < BOULDER_FREQUENCY
➤     @boulders.push Boulder.new(self, 200 + rand(1200), -20)
    end
➤   if @player.x > 1620
      @game_over = true
      @win_time = Gosu.milliseconds
    end
    @player.check_footing(@platforms + @boulders)
    if button_down?(Gosu::KbRight)
        @player.move_right
    elsif button_down?(Gosu::KbLeft)
      @player.move_left
    else
      @player.stand
    end
    @platforms.each do |platform|
      platform.move if platform.respond_to?(:move)
    end
  end
end
```

Our button_down(id) method doesn't change at all from the version you made in the last chapter.

Escape/Escape_2/escape2.rb

```
def button_down(id)
  if id == Gosu::KbSpace
    @player.jump
  end
  if id == Gosu::KbQ
    close
  end
end
```

Before the game will run, we need to add a make_platforms() method that returns an array of platforms. In the next section, we'll focus on this method, creating a random arrangment of moving and stationary platforms so that each game Chip needs to follow a different path out of the pit. For now, so you can test what you've written so far, you can just add a method that returns an empty array.

Escape/Escape_2/escape2.rb

```
def make_platforms
  platforms = []
  return platforms
end
```

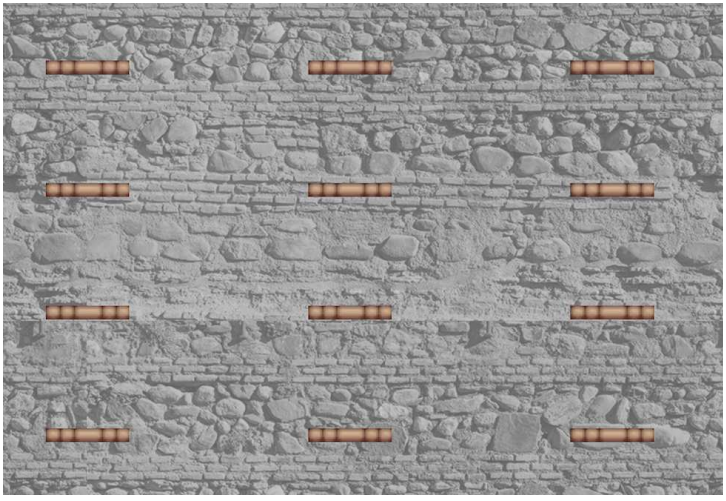
When you run the game now, boulders fall from above, and Chip can run across the floor. When he runs far enough to the right, the camera starts to

follow him. When he approaches the right hand side of the pit, the camera reaches its maximum offset_x and stops moving. Chip can jump, but without platforms he won't get very far.

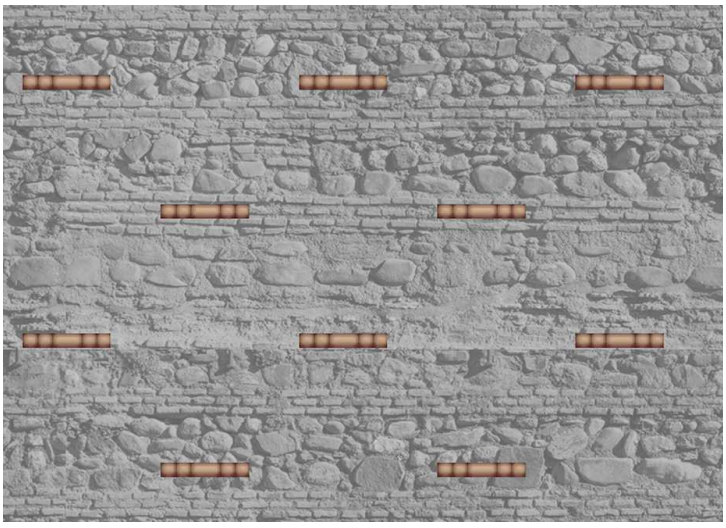
Place Platforms Randomly

In the first version of Escape, we placed the platforms ourselves, by specifying the position of each one. This meant every time someone played the game, the platforms were in the same spots. This is not necessarily a bad thing, as you can design the pit to be just as difficult as you like. In this version, we try another strategy, and place the platforms randomly. The three pictures that follow do not show the actual placement of platforms in our game. They help explain what we are doing in the calculations that actually place the platforms.

To place the platforms randomly, while still making the game playable each time, we start by placing the platforms in a rectangular grid. The following picture shows a region of the pit with the platforms placed this way.



Then every other row is shifted left, so that the platforms are in a pattern that is much easier for Chip to navigate.



Each platform is moved by a random amount in the horizontal direction, and a random amount in the vertical direction. This jumbles them up nicely, and makes certain paths much easier than others.

Finally, some of the platforms are made into moving platforms, and some are taken away entirely, leaving empty spaces that Chip will have to work around. The distribution of platforms is as follows.

- 50% of the platforms are stationary.
- 20% of the platforms move vertically.

- 20% of the platforms move horizontally.
- 10% of the platforms are removed.

These numbers are all easy to adjust, if we find the game too easy or too hard. We also add one nonrandom platform near the escape sign to help Chip run out of the pit.

In our make_platforms() method, we implement this strategy. We make a double loop with eleven rows and five olumns. The position of each platform in the rectangular grid is calculated based on its row and column. If its row is even we slide it to the left by 150 pixels. Then its position is changed by a random number. Finally, we choose a random number to decide if the platform is stationary or moving. The moving ones have a 50% chance each of moving horizontally or vertically. If the random number num is greater than 0.9, no platform is added at all.

Escape/Escape_2/escape2.rb

```
def make_platforms
  platforms = []
  (0..10).each do |row|
    (0..4).each do |column|
      x = column * 300 + 200
      y = row * 140 + 100
      if row % 2 == 0
        x -= 150
      end
      x += rand(100) - 50
      y += rand(100) - 50
      num = rand
      if num < 0.40
         direction = rand < 0.5 ? :vertical : :horizontal
         range = 30 + rand(40)
         platforms.push MovingPlatform.new(self, x,y,range, direction)
      elsif num < 0.90
         platforms.push Platform.new(self,x,y)
      end
    end
  end
  platforms.push Platform.new(self,1550,140)
  return platforms
end
```

When you play now, each arrangement of platforms is different. Try getting to the top. Its pretty challenging, especially since you can't see the platforms that are outside of the window. We could stop now, but before we do, let's use our camera object to add one more feature to our game.

Shake your Camera

We added the camera to our game so that we can follow Chip as he leaps his way out of the pit. By moving the camera, we show a region of the pit in Chip's vicinity. In this section, we take advantage of the camera and add a new feature to our game. Quakes happen randomly. When a quake occurs, the pit appears to shake, and a bunch of boulders rain down all at once. If Chip is near the top during a quake, he'd better move quickly to avoid them. We don't actually move Chip, the pit, or the platforms during a quake. We just shake the camera, and it looks like everything is shaking. A the same time, we give each of the boulders already in the game a random shove, adding to the illusion.

Each quake lasts 30 frames, which is half a second, and each frame of the quake we offset the camera a little bit both vertically and horizontally. We use a strategy similar to the one we used to make the ruby image appear for a short time in Whack-A-Ruby. We create a variable `@quake_time`, that is the number of frames left in a quake. In the `initialize()` method of the `Escape` class, we set that variable to zero and also add a sound effect for our quakes.

Escape/Escape_2/escape2.rb

```
def initialize
  super(800,800)
  self.caption = 'Escape'
  @space = CP::Space.new
➤ @player = Chip.new(self, 70, 1500)
```

```
    @camera = Camera.new(self, 1600, 1600)
    @camera.center_on(@player, 400, 200)
    @game_over = false
    @background = Gosu::Image.new('images/background.png', tileable: true)
    @space.damping = DAMPING
    @space.gravity = CP::Vec2.new(0.0, GRAVITY)
    @boulders = []
    @platforms = make_platforms
    @floor = Wall.new(self, 800, 1610, 1600, 20)
    @left = Wall.new(self,-10, 800, 20, 1600)
    @right = Wall.new(self, 1610, 870, 20, 1460)
    #END_HIGHLGIHT
    @sign = Gosu::Image.new('images/exit.png')
    @font = Gosu::Font.new(40)
    @font_small = Gosu::Font.new(18)
    @music = Gosu::Song.new('sounds/zanzibar.ogg')
    @music.play(true)
    @quake_time = 0
    @quake_sound = Gosu::Sample.new('sounds/quake.ogg')
  end
```

We make a method called quake() in the Escape class that starts a quake. This method sets the quake_time variable and plays the quake sound. It also sends each boulder a push.

Escape/Escape_2/escape2.rb

```
def quake
  @quake_time = 30
  @quake_sound.play
  @boulders.each do |boulder|
    boulder.quake
  end
end
```

Then we write the quake() method for the Boulder class. In it, we give each boulder the same shove that we gave it when it was created. It's enough to get many of the boulders that have come to rest on platforms moving again.

Escape/Escape_2/boulder.rb

```
def quake
      @body.apply_impulse(CP::Vec2.new(rand(100000) - 50000, 100000),
                                  CP::Vec2.new(rand*0.8 - 0.4, 0))
end
```

In the update() method of the Escape class, we decrement the @quake_time variable. If @quake_time is still greater than zero, we're still quaking, so we shake the camera. We also create a new boulder with a *much* higher chance than normal. Since a quake lasts 30 frames, and we have a one fifth chance per frame of creating a boulder, each quake creates an average of six boulders.

Escape/Escape_2/escape2.rb

```
def update
  @camera.center_on(@player, 400, 200)
  if @game_over == false
    @seconds = (Gosu.milliseconds / 1000).to_i
    10.times do
      @space.step(1.0/600)
    end
    if rand < BOULDER_FREQUENCY
      @boulders.push Boulder.new(self, 200 + rand(1200), -20)
    end
    if @player.x > 1620
      @game_over = true
      @win_time = Gosu.milliseconds
    end
    @player.check_footing(@platforms + @boulders)
    if button_down?(Gosu::KbRight)
        @player.move_right
    elsif button_down?(Gosu::KbLeft)
      @player.move_left
    else
      @player.stand
    end
    @platforms.each do |platform|
      platform.move if platform.respond_to?(:move)
    end
    if rand < 0.001
      quake
    end
    @quake_time -= 1
    if @quake_time > 0
      @camera.shake
      if rand < 0.2
        @boulders.push Boulder.new(self, 200 + rand(1200), -20)
      end
    end
  end
end
```

Finally, we write the shake() method of the Camera class. Note that we readjust the camera to Chip's position each frame, so we don't need to *unshake* the camera; that already happens automatically.

Escape/Escape_2/camera.rb

```
def shake
  @x_offset += rand(9) - 4
  @y_offset += rand(9) - 4
end
```

Now we've got quakes! Play the game, and see how fast you can reach the exit.

Make it Your Own

Making Escape added a few tools to your kit. You learned to make a game using the Chipmunk physics engine, and you learned how to make a sidescrolling platformer. As in our other games, there are plenty of things for you to change. Anywhere we added a random chance, distance, or time is an opportunity for you to experiment and tweak the game to your liking. You can make bigger changes too, by adding new features to the game. Make a copy before you begin.

Have fun with the camera.

What else can you do with the camera object? The rules for our camera are pretty simple. Try having the camera adjust to give the player a longer view in the direction Chip is moving. Give the camera a maximum velocity so that it can't move as fast as Chip.

Add more objects.

Put stars on each platform that you have to gather before the gate opens. Add exploding boulders that give Chip a huge shove when they explode. Add a creature that blinks between platforms that you can't pass. Add platforms that move in circles. Add a glowing arrow that points toward the location of the exit.

Learn more of Chipmunk.

In making Escape, we only explored part of what the Chipmunk physics engine can do. Reading the Chipmunk documentation[1] may give you some ideas. There are many aspects of Chipmunk we didn't touch on in Escape. Constraints in particular are a rich topic to explore. Maybe you'll get a great idea for your next game just by learning more of Chipmunk's capabilities.

What's Next

If you've worked through the whole book to this point, you've created four games using Ruby and Gosu. You've adjusted them to suit yourself, and hopefully completed some challenges to make them better. To let your friends play these games on their own computers, you'll need to send them a copy of your game they can run. In the next and final chapter of this book, you'll

1. http://beoran.github.io/chipmunk/

learn to *package* your games, making Windows or OS X executables that can be easily shared.

CHAPTER 9

Package and Share Your Game

You've finished your game. Its fun, and just the right difficulty. At least, *you* think it is. Now its time to share it with your friends and see what they think. Its also time to see if they can beat your high score. Feedback from other players is the best way to get better. Your friends can tell you what works and what doesn't; whats fun and what's not so great.

You can already play the game, but its a folder full of files, and you can only run it becuase you set up a development environment with Ruby and Gosu, back in Chapter 1, *Get Ready*, on page 1. You don't want to make your friends set up their computers the same way - you just want to send them the game and let them play. Next, we learn how to turn that folder full of files into an application that anyone can use.

In this chapter, as an example, we package up the Sector Five game we finished in Chapter 5, *Adding Scenes and Sounds*, on page 81, but the procedure we follow will work for any game you make. This process of *packaging* a game is completely different for Windows and Mac OS X applications. If you're packaging for Windows, read on. If you're packaging for OS X, skip ahead to *Packaging for OS X*, on page 183.

Packaging for Windows

We package our game using the same command line tool we used in Chapter 1, *Get Ready*, on page 1. We get a new Ruby gem, called `ocra`, which we use to create our executable.

Run the "Command Line Prompt with Ruby" application. At the prompt, type the following.

```
C:\ gem install ocra
Fetching: ocra-1.3.5.gem (100%)
Successfully installed ocra-1.3.5
```

To use ocra, we need to first use cd to move into the directory where our game files are. In this tutorial, the SectorFive folder is on the desktop, so at the command line we can type the following.

```
C:\Users\mark> cd Desktop/SectorFive
C:\Users\mark\Desktop\SectorFive>
```

The ocra command needs to be followed with the name of the main Ruby file in our game, and then with a list of the resources we want included in the packaged application. The --chdir-first option tells ocra where to look for the image and other resources. We list all the pictures, sounds, and any other files we've used, or if they are in folders, simply list the folders. It's easy to include extra images and sounds that we're not using this way, so move out any files you don't need before you start. Our gems, like Gosu and Chipmunk, and any Ruby files we've required will be included automatically. We can choose the name of our executable with the --output flag. To package SectorFive, type the following. The game will actually run, as ocra figures out all the files it needs to include. You can play, or just quit, and ocra will finish.

```
C:\SectorFive>ocra --chdir-first --output SectorFive.exe sector_five_scenes.rb
              images sounds credits.txt
=== Loading script to check dependencies
=== Detected gem ocra-1.3.5 (loaded, files)
===   6 files, 19031 bytes
=== Detected gem gosu-0.9.2-x86-mingw32 (loaded, files)
===   38 files, 9283787 bytes
=== Including 52 encoding support files (2836480 bytes, use --no-enc to exclude)
=== Building SectorFive
=== Adding user-supplied source files
=== Adding ruby executable ruby.exe

... more ...

=== Adding library files
=== Compressing  52191268 bytes
=== Finished building SectorFive (33786909 bytes)
```

Running this script will take awhile. When its done, a new file will be created, SectorFive.exe. This is the executable file that you can share with your friends. It contains your game, along with Ruby, Gosu, and all the media files included in your game. You might notice that SectorFive.exe is a 34MB file, which is pretty big! We'll discuss how you can share it in *Share your Game*, on page 187

Packaging for OS X

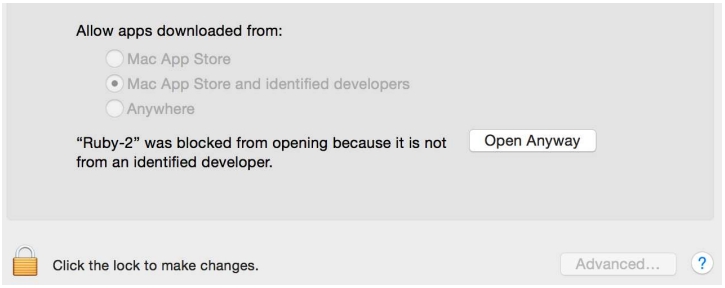
To package our game, we'll download a *wrapper* from the Gosu website. This wrapper is a complete application, written with Ruby and Gosu, that doesn't do much, but has the whole environment needed to run our game. Then we'll replace the application inside the wrapper with our game. Finally we'll make a few changes to a file called info.plist. Then we can share it with our friends.

Get the Wrapper Application

The authors of Gosu have provided an application wrapper so you can make an OS X app from your Ruby/Gosu game. Using your browser, go to https://github.com/gosu/ruby_app/releases/.

From that site, download the latest Ruby.app.zip. Find it in your downloads, unzip the file, and move it to your desktop. When you run the app, you'll probably see a message like this one.

This message appears because Apple is protecting you from malware. In System Preferences, open panel called “Security and Privacy”.

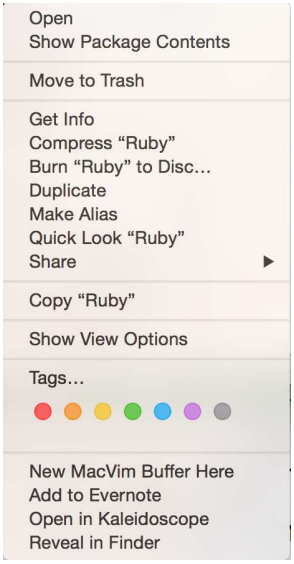


The default settings are to only run applications from the “Mac App Store and identified developers.” You could change your settings to run applications from “Anywhere”, but a safer choice is to choose “Open Anyway” to allow just this one to run, while leaving the default protection in place. *When you distribute your game, your users will likely get the same message, and will need to take the same steps to fix it.*

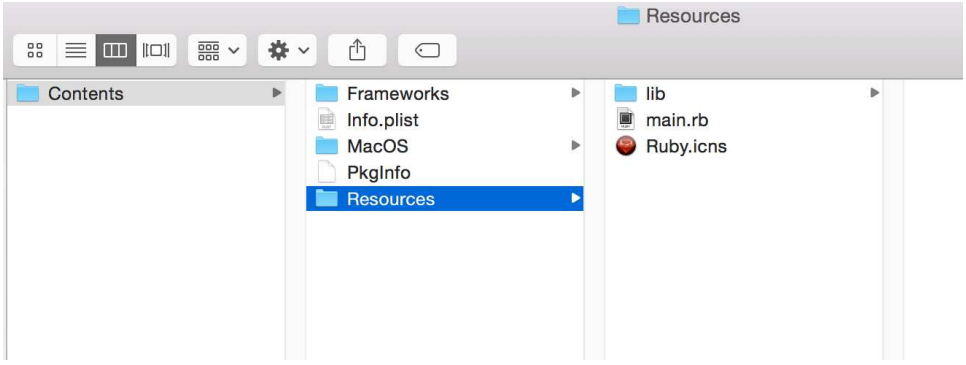
When you finally run the `Ruby.app` application, a big ruby rocks gently back and forth in a window on your screen. The next step is to replace that ruby with your game.

Open the Wrapper

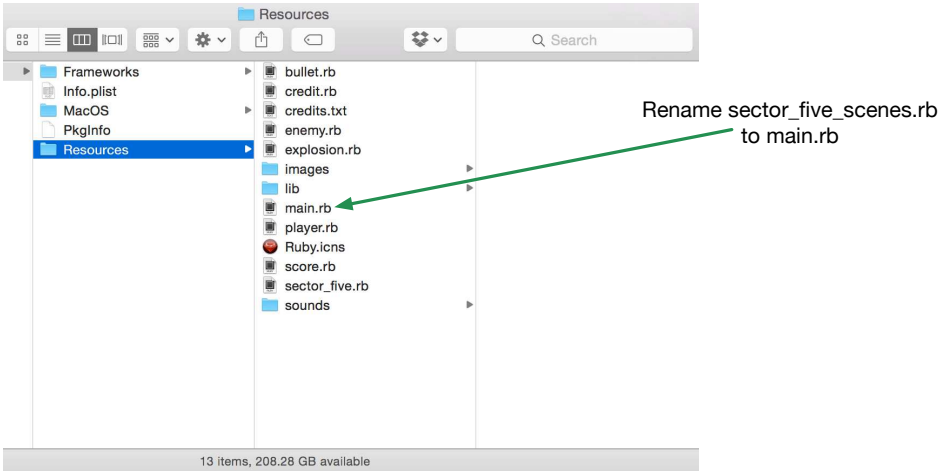
An application on OS X is really a folder. Double clicking a normal folder opens the folder and lets you see what is inside, but double clicking `Ruby.app` runs the application. To see inside it, right click (or hold down the control key while you click) on the `Ruby.app` icon in the Finder.



From the contextual menu, choose “Show Package Contents”. Here is a column view from the Finder, showing what is inside the Ruby.app before we make our changes.



The application package always runs the file named main.rb. To replace the sample app with our game, we first remove main.rb, and move our whole game, ruby files, pictures, sounds, and all, into its place. Make sure you *don’t* remove the lib folder - that’s where Gosu and a whole bunch of other files are. Then, we rename our game file, in this case sector_five_scenes.rb, to main.rb. Here is the same file structure with our game in place.



Then we make some changes to a file called info.plist. This file, according to Apple's developer website is "a structured text file that contains essential configuration information for a bundled executable." And since we're turning our game into a bundled executable, we put some information about our game into the info.plist file. Start by opening info.plist in your text editor.

The Info.plist file is a list of *keys* and values. For instance, in the following section, the key CFBundleIdentifier is associated with the value com.example.Ruby.

```
<key>CFBundleIdentifier</key>
<string>com.example.Ruby</string>
```

Change the value to com.example.SectorFive, or replace 'example' with your company's name, real or imagined. To change the name of the app, you can include a new key and value

```
<key>CFBundleDisplayName</key>
<string>Sector Five</string>
```

And also change the value associated with the key CFName.

```
<key>CFBundleName</key>
<string>Sector Five</string>
```

After you make these changes to Info.plist, you can change the name of your application package directory to Sector Five, in the Finder. If you change the name without changing Info.plist first, your application becomes broken and you won't be able to run it.

Before you share your OS X game, you should compress it into a ZIP file. Right click (or CTRL-click) on the file and choose "Compress SectorFive" from

the contextual menu. You can share the resulting ZIP file over email or a file sharing service.

Share your Game

Now you have a game that you're ready to share. The applications we've created are pretty large files. For SectorFive, the Windows executable is about 40MB, and the compressed OS X app bundle is 32MB. This is because each package includes a complete Ruby and Gosu installation, as well as any system resources that might be needed. In order to make sure the game runs on a wide range of computers, a lot of things are included. For this reason, you might want to share your game with a file sharing service such as Dropbox or Google Drive, rather than using email. Some email services, including Gmail, have file size limits for attachments.

What's Next

We've made some games together. You've tried some challenges, and made some changes of your own to the games you've made. It's time for you to make your own games. Maybe one of the games in this book gave you an idea for a new game, or maybe a game you played on your phone will get your juices flowing. Make some games, and share them. If you make one you're proud of, I hope you'll let me know, and send me a copy.

One next step you can take, after sharing your games, is sharing your code. Get some friends interested, and swap games and ideas. Work on a game together. If you're in school, start a club or other organization to promote game programming.

Many metropolitan areas have Ruby user groups that meet periodically. The people at those groups might be working on things other than games, but they are all interested in Ruby, and many will likely be interested in helping Ruby programmers grow their skills. And many programmers got started, just like you, by writing games, and they still may find game writing hard to resist.

Thanks again, for reading this book. I trust you've learned something about writing games, and that your programming skills have grown. It's time to say goodbye, and leave you to write some games yourself. Good luck, and happy game writing!

APPENDIX 1

Resources

Documentation

The Gosu gem has a great Ruby documentation site, with a list of every class and method with many examples. You can find it at http://www.libgosu.org/rdoc/.

The Chipmunk gem has some great documentation as well. You can find it at http://beoran.github.io/chipmunk/.

Images and Sounds

There are some great resources online that provide free graphics to aspiring game writers. A few are listed here, to get you started, and you can find many more by searching. Pay attention to the licenses under which art and music is released, and give credit to the artists as appropriate.

OpenGameArt.org

This is has a tremendous amount of free art, and has some good search tools as well. You can search by keyword, by type of art, or by license. Some sound effects are on this site as well. http://www.opengameart.org

Incompetech.com

Incompetech is a website where Kevin MacLeod shares and sells his musical creations. He has created a wide range of music that you can search by type and mood. Some of the background music in Sector Five came from this site.http://www.incompetech.com

Wikimedia Commons

You can find this site at http://commons.wikimedia.org. It is a collection of over 24 million freely usable photographs. The background image in Escape is from this site.

OpenClipArt

Lots of free clip art, at http://www.openclipart.org. The images of the ruby and hammer in WhackARuby are from this site.

Clker.com

A site of free clip art, at http://www.clker.com. The exit sign in Escape is from this site.

Widgetworx.com

Widgetworx is the creation of Ari Feldman, and features SpriteLib, a collection of free sprites. The sprites in Escape including the hero, Chip, are from this collection. http://www.widgetworx.com

Bibliography

[FH13] Dave Thomas, with Chad Fowler and Andy Hunt. *Programming Ruby 1.9 & 2.0 (4th edition)*. The Pragmatic Bookshelf, Raleigh, NC, and Dallas, TX, 4th, 2013.

[Pin09] Chris Pine. *Learn to Program (2nd edition)*. The Pragmatic Bookshelf, Raleigh, NC, and Dallas, TX, 2nd, 2009.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/msgpkids
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/msgpkids*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764