Anatomy of a Game Engine

By:

R.G. (Dick) Baldwin

Anatomy of a Game Engine

By:

R.G. (Dick) Baldwin

Online:
< http://cnx.org/content/col11489/1.13/ >

OpenStax-CNX

Collection structure revised: February 7, 2013
PDF generated: October 3, 2015
For copyright and attribution information for the modules contained in this collection, see p. 290.

Table of Contents

Chapter 1

Slick0100: Getting started with the Slick2D game library[1]

1.1 Table of Contents

1.2 Preface

Turning the crank

As a professor of Computer Information Technology at Austin Community College, I teach courses in game programming using both C++ and C#/XNA. I have long had a concern that students enter my courses expecting to simply *"turn the crank"* on a game engine such as Dark GDK or XNA and have great games emerge from the other end of the process. Unfortunately, it isn't quite that easy.

---

[1] This content is available online at <http://cnx.org/content/m45726/1.9/>.

Anatomy of a game engine

Given time limitations and other restrictions, it is not practical to teach those students much about the inner working of such game engines. Therefore, I have decided to publish a series of modules on the anatomy of a game engine that my students, *(and other interested parties)* can read to learn about those inner workings.

First in a collection

Therefore, this module is the first in a collection of modules designed to teach you about the anatomy of a typical game engine *(sometimes called a game framework)* **.**

The Slick2D library

I have chosen to concentrate on a free game library named Slick2D [2] *,(which is written in Java)* for several reasons including the following:

- Java is the language with which I am the most comfortable. Hence, I can probably do a better job of explaining the anatomy of a game engine that uses Slick2D than would be the case for a game engine written in C++, C#, Python, or some other programming language.
- Java has proven in recent years to be a commercially successful game programming language. For example, I cite the commercial game named Minecraft [3] , written in Java, for which apparently millions of copies have been sold. Also, Java is used for developing apps for Android.
- Slick2D is free and the source code for Slick2D is readily available.
- The overall structure of a basic Slick2D game engine is very similar to Dark GDK and XNA, and is probably similar to other game engines as well.
- Java is platform independent.

Applicable to other environments as well

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to other game engines written in different programming languages.

Purpose

The purpose of this module is to get you started, including showing you how to download and install Slick2D, and how to compile and execute your first Slick2D program. Future modules will start digging into and explaining the inner workings of a basic Slick2D game engine.

What you should know

This series of modules is not intended for beginning programmers. As a minimum, you should already know about fundamental programming concepts such as **if** statements, **for** loops, **while** loops, method or function calls, parameter passing, etc. Ideally, you will have some object-oriented programming knowledge in a modern programming language such as Java, C#, C++, or possibly Python or JavaScript.

You should also be relatively comfortable with the command-line interface, directory or folder trees, batch or script files, etc.

Finally, you should also be comfortable downloading and installing software on the machine and operating system of your choice.

What you will learn

In this module, you will learn how to download and install Slick2D on a Windows XP, Vista, or Windows 7 machine and how to compile and execute a very simple Slick2D program. *(If you are using a different operating system, you will need to translate this information to your system of choice. However, since Java is platform independent, the code details that I will discuss will apply to all or most platforms.)*

1.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

---

[2] http://slick.ninjacave.com/

[3] http://minecraft.net/

1.2.1.1 Figures

- Figure 1 (p. 9) . Output from Slick2D during program startup.
- Figure 2 (p. 13) . A default Slick2D game window.

1.2.1.2 Listings

- Listing 1 (p. 6) . Slick2D program named Slick0100a.java.
- Listing 2 (p. 7) . The file named CompileAndRun.bat.

1.3 Preview

Most of the Slick2D tutorials that you will find on the Internet will begin by telling you to download and install a high-level IDE such as Eclipse or NetBeans. I won't do that.

While high-level IDEs are great for improving productivity for experienced programmers, I consider them to be overkill for students just learning how to program. Not only are they overkill, they also hide many details that beginning programmers need to understand.

Therefore, I will show you how to install Slick2D in such a way that you can easily compile and execute Slick2D programs from the command line with no need for a high level IDE. All you will need is a text editor *(preferably color coded for Java syntax)* , the free Slick2D distribution, and the free Java Development Kit from Oracle.

1.4 Download the required software

1.4.1 Text editors

There are numerous free text editors available on the Internet, some with and some without Java syntax color coding. *(In a pinch, even Windows Notepad will suffice.)* Here are links to a few of them.

- JCreator [4]
- jGRASP [5]
- DrJava [6]
- Arachnophilia [7]

1.4.2 The Slick2D distribution

I will be using this material in some of the Java OOP programming courses that I teach. I expect that changes and improvements will be made to the Slick2D library over time. However, it can be very confusing when different students in the same programming course are using different versions of software, particularly if changes to the software are made that are not backward compatible.

Therefore, to ensure that my students all download and use the same version of the software in my courses, I will make a copy of a particular version of the Slick2D distribution available by clicking here [8] . *(Note: as of 06/06/15, this is a 64-bit version of the distribution. The older 32-bit distribution is available here [9] .)*

---

[4] http://www.jcreator.com/

[5] http://www.jgrasp.org/

[6] http://www.drjava.org/

[7] http://www.arachnoid.com/arachnophilia/index.php

[8] http://cnx.org/content/m45726/latest/slick-64.zip

[9] http://cnx.org/content/m45726/latest/slick-1.zip

If you are not one of my students, you may prefer to go to the Slick2D [10] main page and select the link to download the latest version of the distribution. Save that file because I will have more to say about it later.

1.4.3 The Java Development Kit

Go to Oracle [11] and download the latest release of Java SE *(standard edition)* that is compatible with your system. Then open the installation instructions [12] and select the link for your system. For example, there is *(or was)* a link on that page that reads:

> *JDK Installation for Microsoft Windows* [13] *- Describes how to install the JDK on 32-bit and 64-bit Microsoft Windows operating systems.*

Follow the link to the installation instructions for your system and follow those instructions to install the Java Development Kit. When doing the installation, pay attention to the link that reads Updating the PATH Environment Variable (Optional) [14] . This is where many of my students encounter installation difficulties.

> *(Note that over time, some of these links may change. However, the general concepts involved should continue to be relevant.)*

There are also issues dealing with something called the ***classpath*** , but I will explain how to deal with those issues later.

1.5 Install the required software

I am assuming that you can install the text editor and the JDK with no help from me. Therefore, I will concentrate on installing and configuring the Slick2D software. *(Note that these instructions apply to the 64-bit versions. Instructions for installing the 32-bit version were similar.)*

The Code folder tree

Begin by creating a folder somewhere on your disk named **Code** *(or some other similar name of your choosing)* .

Create three sub-folders under the **Code** folder having the following names:

- jars
- lwjglbin
- Slick0100a *(this will change from one program to the next)*

(See Download source code (p. 14) to download folders named ***jars*** *,* ***lwjglbin*** *, and* ***Slick0100a*** *already populated with the minimum required 64-bit Windows-compatible software.)*

Extract the contents of the zip file

Using whatever program you can find to open a zip file *(I use a program named WinZip)* , extract and save the following files from the lib folder in the 64-bit Slick2D distribution that you **downloaded earlier** .

- lwjgl.jar
- slick.jar
- natives-windows.jar

---

[10] http://slick.ninjacave.com/
[11] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[12] http://docs.oracle.com/javase/7/docs/webnotes/install/index.html
[13] http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html
[14] http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html#path

There are many other files in the Slick2D distribution file, but we don't need them just yet. If we need them in a future module, I will tell you.

These three files are needed to satisfy the ***classpath*** and ***java.library.path*** requirements that I will describe later.

The first two jar files

Copy the first two jar files from the above list (p. 4) into your new folder named **jars** .

As you will see later, this results in the need to execute the following command in order to set the classpath whenever you compile or execute a Slick2D program:

-cp .;../jars/slick.jar;../jars/lwjgl.jar

The third jar file

The third file in the above list (p. 4) applies to Windows only. If you are using a different system, you should find a similar file in the Slick2D distribution that applies to your system. *(For example, the distribution contains files named natives-linux.jar and natives-mac.jar.)*

Extract contents of the jar file

Using whatever program you can find to open a jar file *(I use a program named WinZip)* , extract the following files from the file named **natives-windows.jar** :

- jinput-dx8_64.dll
- jinput-raw_64.dll
- lwjgl64.dll
- OpenAL64.dll

Copy these four files into your new folder named **lwjglbin** .

As you will see later, this results in the need to execute the following command in order to set the ***java.library.path*** system property:

-Djava.library.path=../lwjgnbin

(These files can also be stored in the folder from which the program is being run and this will eliminate the requirement to set the java.library.path if you prefer that approach.)

1.6 Create, compile, and execute your first Slick2D program

1.6.1 Create a source-code file

Use your text editor to create a text file named **Slick0100a.java** and store it in the folder named **Slick0100a** .

(Be careful to ensure that the file has the correct extension, particularly if you create it with Windows Notepad. An extension of .txt won't work.)

Carefully copy the code from Listing 1 (p. 6) into the text file. This is the file that you will attempt to compile and run to confirm correct operation of your system.

*(See Download source code (p. 14) to download a folder named **Slick0100a** already populated with the code from Listing 1 (p. 6) .)*

Listing 1 . Slick2D program named Slick0100a.java.

```
/*Slick0100a.java
Copyright 2012, R.G.Baldwin

A simple program that shows the method definitions
required by the Slick framework.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.SlickException;

public class Slick0100a extends BasicGame{

  public Slick0100a(){
    //Call to superclass constructor is required.
    super("This title will be overridden later.");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    //Constructor for AppGameContainer requires parameter
    // of interface type Game. Hence, object of this class
    // must provide concrete definitions of the five
    // methods declared in the Game class. Two of those
    // methods are overridden in the BasicGame class. The
    // other three are not.
    AppGameContainer app =
                       new AppGameContainer(new Slick0100a());
    app.start();//this statement is required
  }//end main
  //----------------------------------------------------//

  @Override
  public void init(GameContainer gc)
                                   throws SlickException {
    //Concrete override required.
    //Do any required initialization here.
  }//end init
  //----------------------------------------------------//

  @Override
  public void update(GameContainer gc, int delta)
                                    throws SlickException{
    //Concrete override required.
    //Put the game logic here.
  }//end update
  //----------------------------------------------------//

  public void render(GameContainer gc, Graphics g)
```

Table 1.1

1.6.2 Compile and run the program

1.6.2.1 Create a batch file

Use your text editor to create a text file named **CompileAndRun.bat** and store it in the folder named **Slick0100a** .

(Once again, be careful to ensure that the file has the correct extension, particularly if you create it with Windows Notepad. An extension of .txt won't work.)

Carefully copy the contents Listing 2 (p. 6) into the text file. This is the file that you will use in your attempt to compile and run your source-code file named **Slick0100a.java** . *(Line breaks or wrapped lines are not allowed. Make certain that your batch file has only seven lines of text exclusive of blank lines.)*

Listing 2 . The file named CompileAndRun.bat.

```
echo off
del *.class

rem refer to jar files in the folder named jars
javac -cp .;../jars/slick.jar;../jars/lwjgl.jar Slick0100a.java

rem set the java.library.path and the classpath and run the program
java -Djava.library.path=../lwjglbin -cp .;../jars/slick.jar;../jars/lwjgl.jar Slick0100a

pause
```

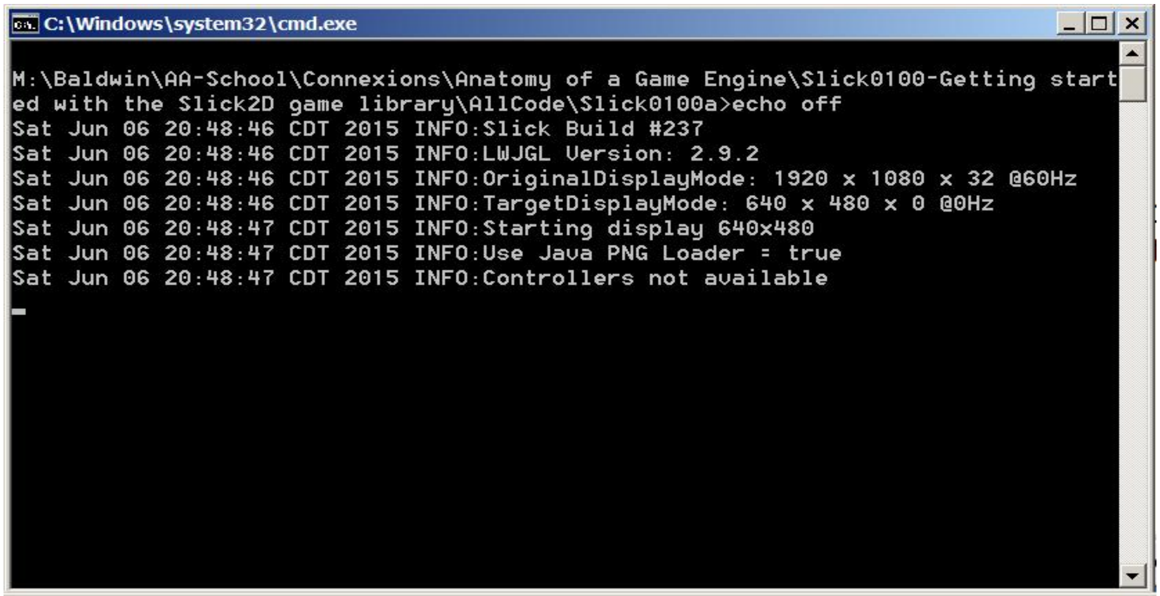
Table 1.2

1.6.2.2 Execute the batch file

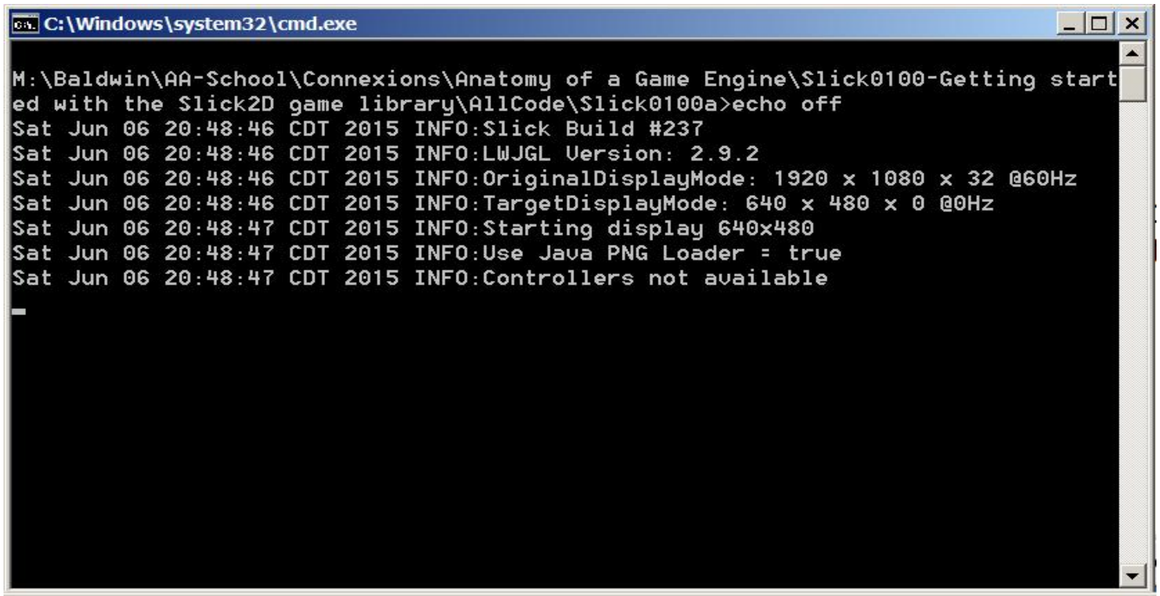
Double-click your new batch file named **CompileAndRun.bat** *(or execute it in whatever manner you prefer.)* This should cause two new windows to appear on your screen.

Slick2D output during startup

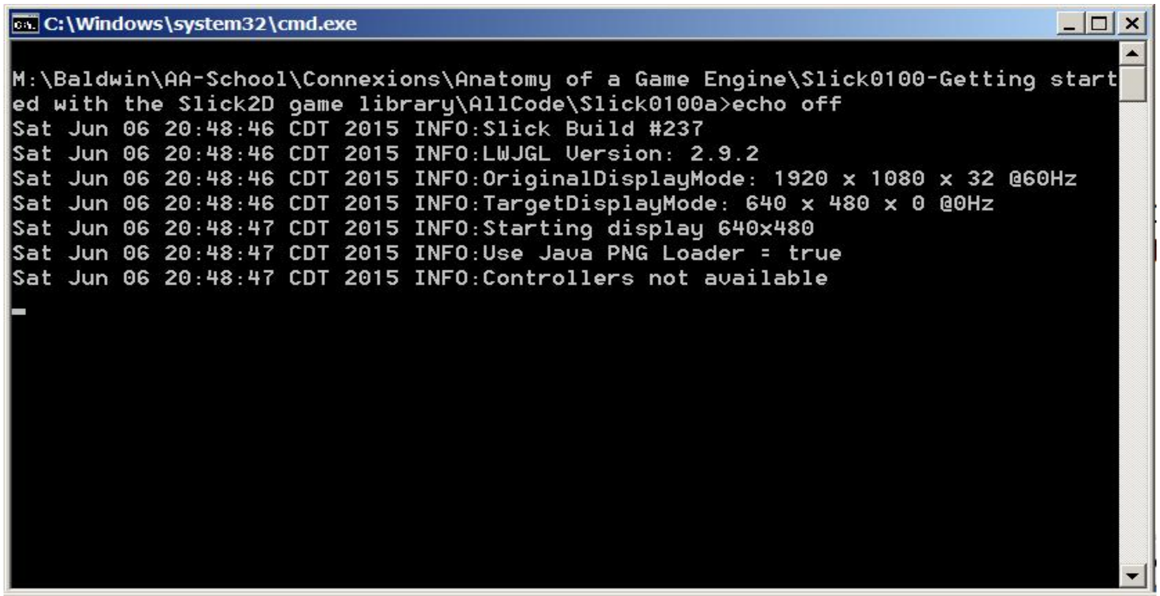
The first window to appear should look similar to Figure 1 (p. 9) .



```
M:\Baldwin\AA-School\Connexions\Anatomy of a Game Engine\Slick0100-Getting start
ed with the Slick2D game library\AllCode\Slick0100a>echo off
Sat Jun 06 20:48:46 CDT 2015 INFO:Slick Build #237
Sat Jun 06 20:48:46 CDT 2015 INFO:LWJGL Version: 2.9.2
Sat Jun 06 20:48:46 CDT 2015 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sat Jun 06 20:48:46 CDT 2015 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sat Jun 06 20:48:47 CDT 2015 INFO:Starting display 640x480
Sat Jun 06 20:48:47 CDT 2015 INFO:Use Java PNG Loader = true
Sat Jun 06 20:48:47 CDT 2015 INFO:Controllers not available
```

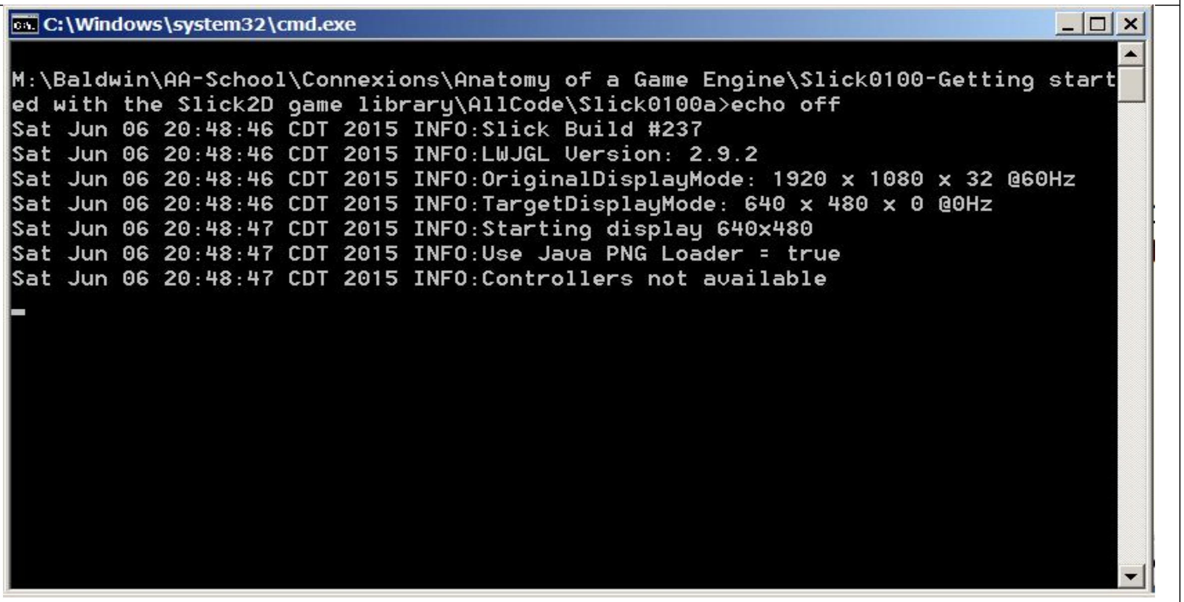


```
M:\Baldwin\AA-School\Connexions\Anatomy of a Game Engine\Slick0100-Getting start
ed with the Slick2D game library\AllCode\Slick0100a>echo off
Sat Jun 06 20:48:46 CDT 2015 INFO:Slick Build #237
Sat Jun 06 20:48:46 CDT 2015 INFO:LWJGL Version: 2.9.2
Sat Jun 06 20:48:46 CDT 2015 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sat Jun 06 20:48:46 CDT 2015 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sat Jun 06 20:48:47 CDT 2015 INFO:Starting display 640x480
Sat Jun 06 20:48:47 CDT 2015 INFO:Use Java PNG Loader = true
Sat Jun 06 20:48:47 CDT 2015 INFO:Controllers not available
```



```
M:\Baldwin\AA-School\Connexions\Anatomy of a Game Engine\Slick0100-Getting start
ed with the Slick2D game library\AllCode\Slick0100a>echo off
Sat Jun 06 20:48:46 CDT 2015 INFO:Slick Build #237
Sat Jun 06 20:48:46 CDT 2015 INFO:LWJGL Version: 2.9.2
Sat Jun 06 20:48:46 CDT 2015 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sat Jun 06 20:48:46 CDT 2015 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sat Jun 06 20:48:47 CDT 2015 INFO:Starting display 640x480
Sat Jun 06 20:48:47 CDT 2015 INFO:Use Java PNG Loader = true
Sat Jun 06 20:48:47 CDT 2015 INFO:Controllers not available
```

Figure 1 . Output from Slick2D during program startup.


C:\Windows\system32\cmd.exe

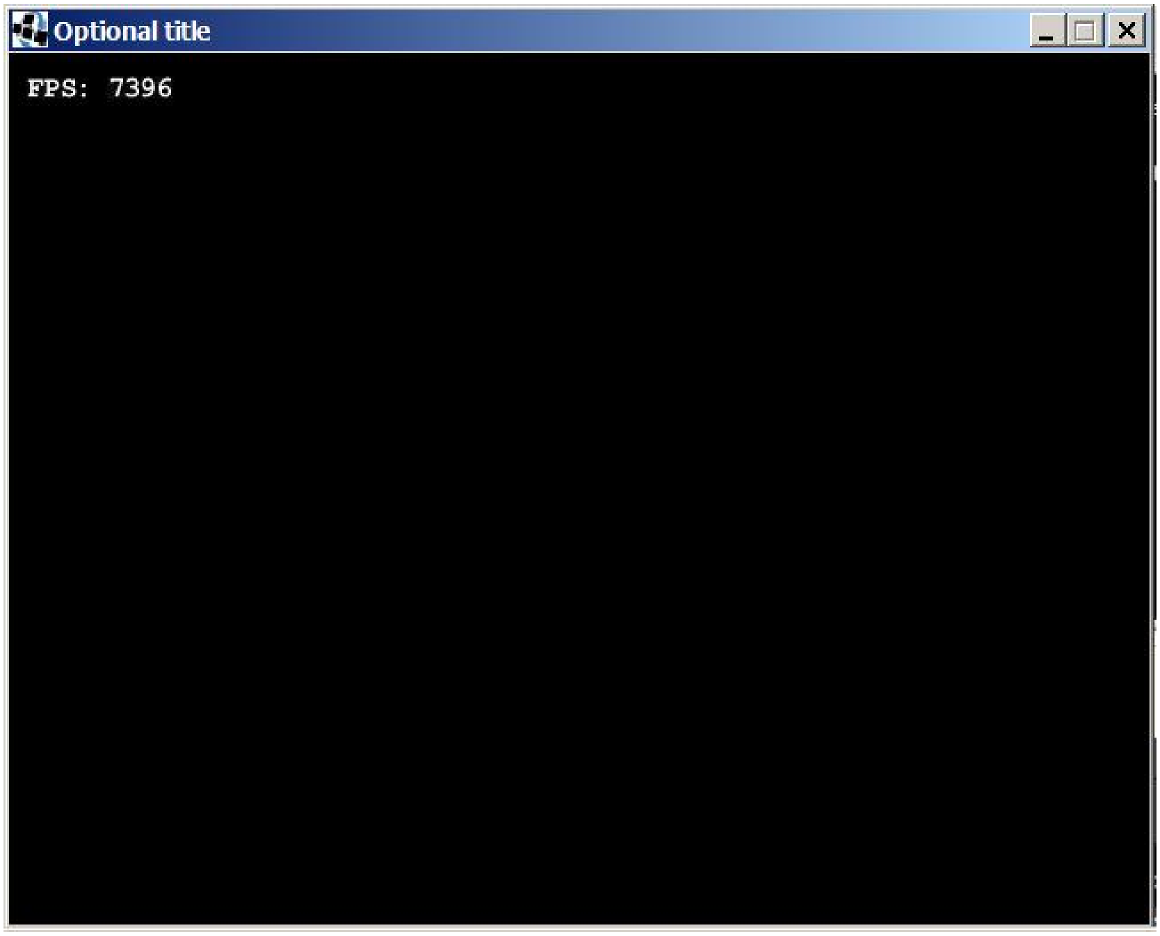
M:\Baldwin\AA-School\Connexions\Anatomy of a Game Engine\Slick0100-Getting start
ed with the Slick2D game library\AllCode\Slick0100a>echo off
Sat Jun 06 20:48:46 CDT 2015 INFO:Slick Build #237
Sat Jun 06 20:48:46 CDT 2015 INFO:LWJGL Version: 2.9.2
Sat Jun 06 20:48:46 CDT 2015 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sat Jun 06 20:48:46 CDT 2015 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sat Jun 06 20:48:47 CDT 2015 INFO:Starting display 640x480
Sat Jun 06 20:48:47 CDT 2015 INFO:Use Java PNG Loader = true
Sat Jun 06 20:48:47 CDT 2015 INFO:Controllers not available

Table 1.3

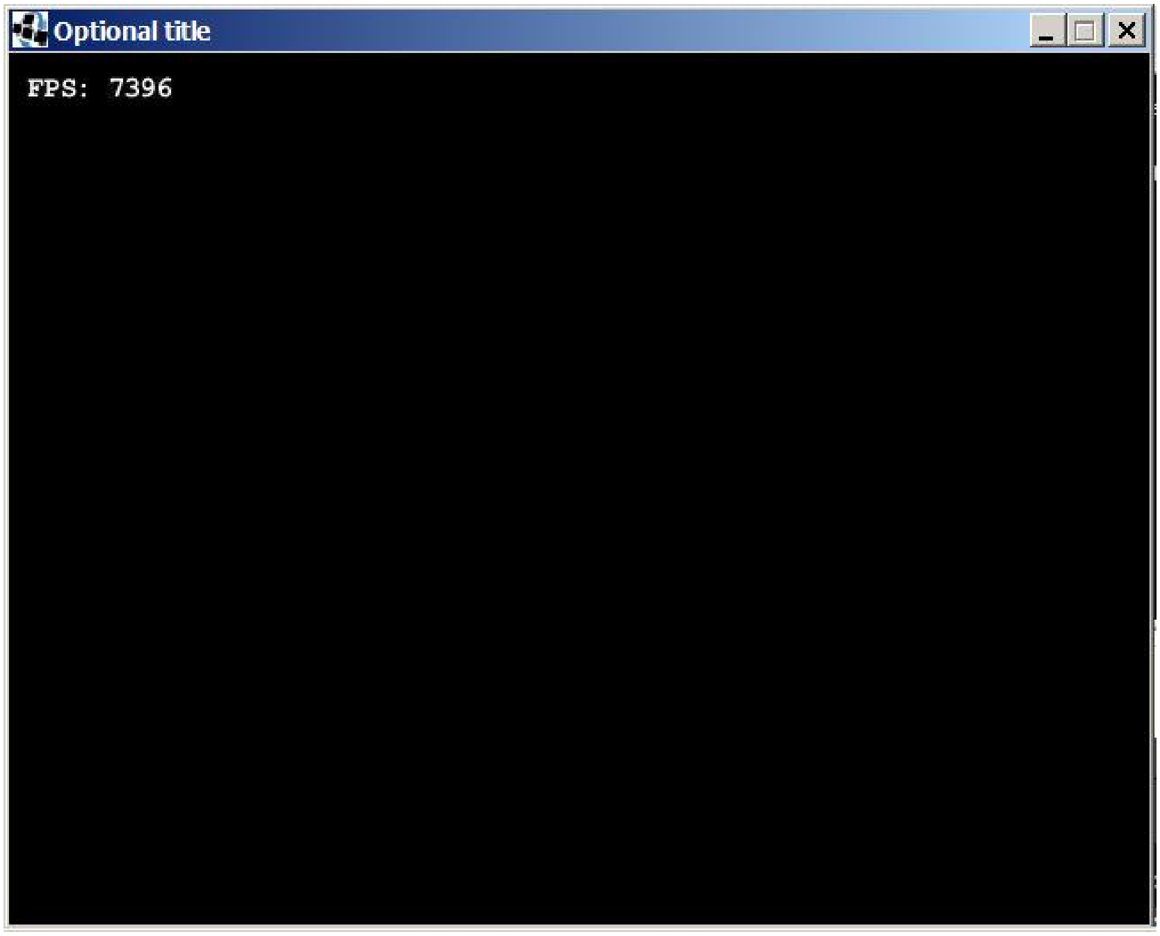
Figure 1 (p. 9) shows information produced by Slick2D when a compiled Slick2D program starts running.

A default Slick2D game window

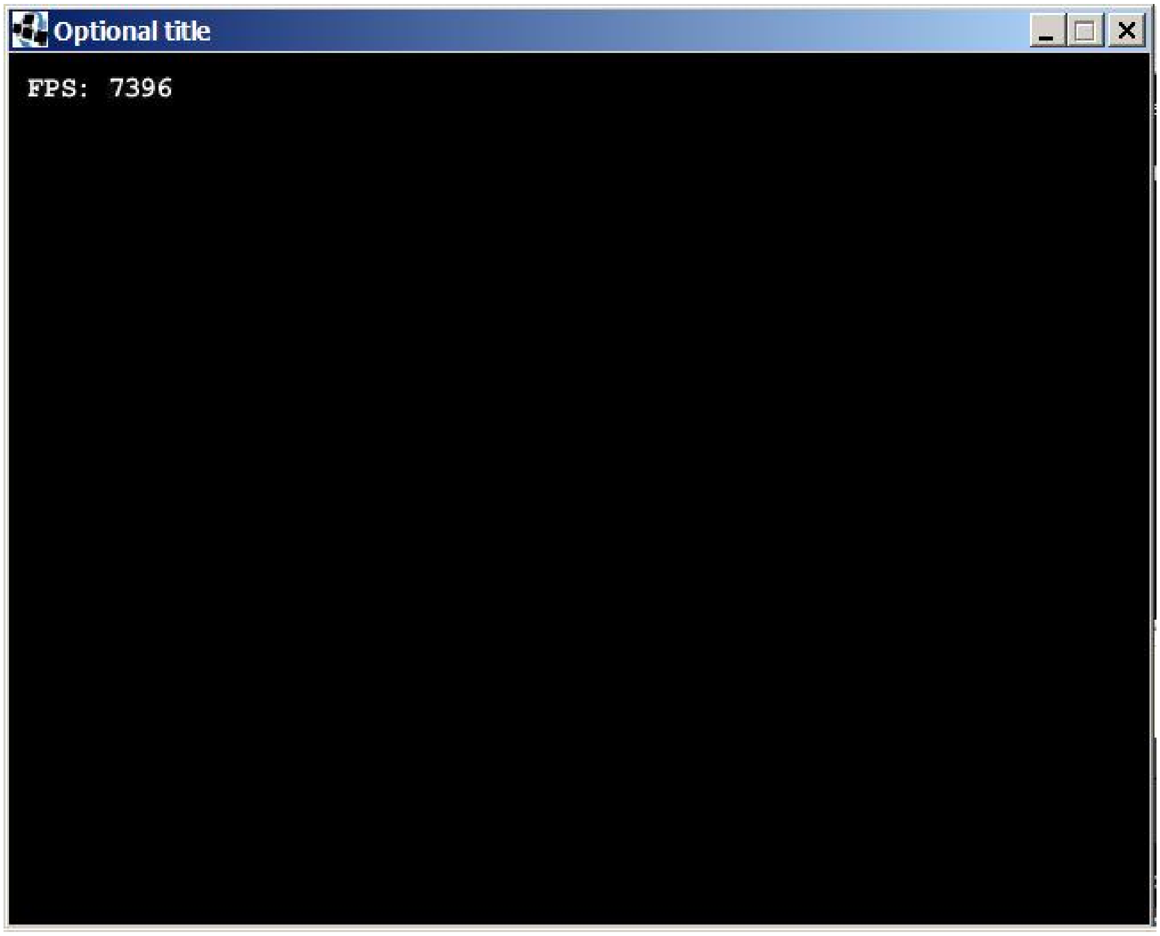
The second window to appear should look something like Figure 2 (p. 13) .



```
FPS: 7396
```



FPS: 7396



```
FPS: 7396
```

Figure 2 . A default Slick2D game window.

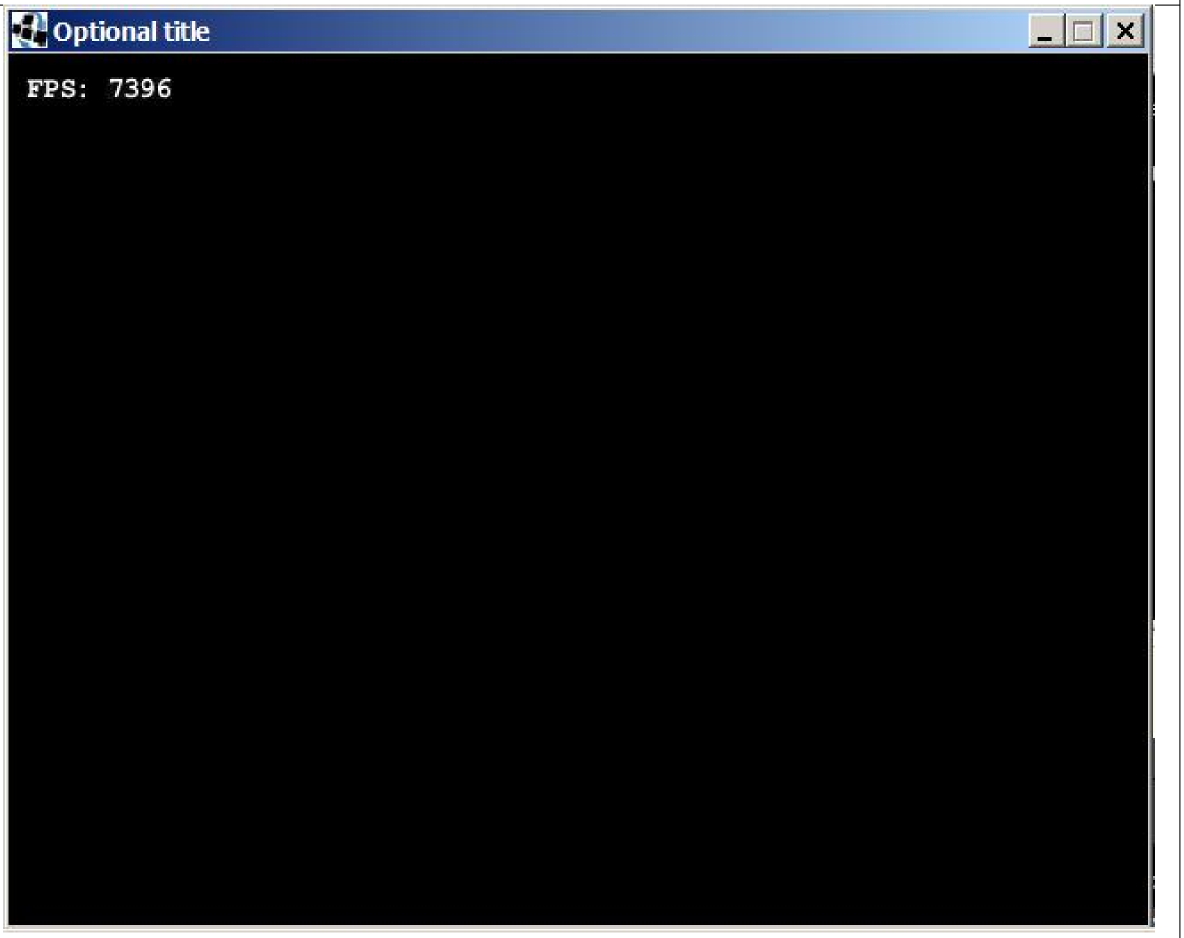


Table 1.4

Figure 2 (p. 13) is a default Slick2D game window. As you will learn in the next module, this particular Slick2D program has no interesting behavior. In effect, it is an "empty" game program. Therefore, the only thing showing in the game window is a counter in the top left corner that shows the execution rate in frames per second.

1.7 Run the program

I encourage you to copy the code from Listing 1 (p. 6) and Listing 2 (p. 7) . Install the necessary software on your computer as described above. Compile the code and execute it. If you don't see results similar to those shown above, go back and review the instructions very carefully.

1.8 Summary

I showed you how to install Slick2D in such a way that you can easily compile and execute Slick2D programs from the command line with no need for a high level IDE.

1.9 What's next?

In the next module, I will use the code from Listing 1 (p. 6) to begin explaining the anatomy of a basic Slick2D game engine.

1.10 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0100: Getting started with the Slick2D game library.
- File: Slick0100.htm
- Published: 02/03/13
- Revised: 06/06/15 for 64-bit

Disclaimers: Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

1.11 Download source code

Click here [15] to download a zip file containing the source code for all of the sample programs in this collection. The zip file also contains populated 64-bit versions of the folders named jars and lwjglbin (p. 4) to save you the trouble of downloading the distribution and populating those folders.

Extract the contents of the zip file into an empty folder. Each program should end up in a separate folder. Double-click the file named ***CompileAndRun.bat*** in each folder to compile and run the program contained in that folder.

-end-

---

[15] http://cnx.org/content/m45726/latest/AllCode.zip

Chapter 2

Slick0110: Overview[1]

2.1 Table of Contents

2.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages.

The purpose of this module is to teach you about some of the characteristics of game engines in general, and to teach you how Slick2D fits those characteristics.

---

[1] This content is available online at <http://cnx.org/content/m45728/1.3/>.

2.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

2.2.1.1 Figures

- Figure 1 (p. 16) . Steps for writing a Slick2D game program.
- Figure 2 (p. 22) . Output from Slick2D during program startup.
- Figure 3 (p. 26) . A default Slick2D game window.
- Figure 4 (p. 29) . Abstract methods declared in the Game interface.

2.2.1.2 Listings

- Listing 1 (p. 19) . Simplest Slick2D program.
- Listing 2 (p. 20) . The file named CompileAndRun.bat.
- Listing 3 (p. 28) . Beginning of the class named Slick0110a.

2.3 The bottom line at the top

In order to write a game program using the Slick2D game library that will run as a Java application, you must, as a minimum, perform the steps shown in Figure 1 (p. 16) .

Figure 1 . Steps for writing a Slick2D game program.

```
1. Define a class containing a main method that will run as an application.

2. Define and instantiate an object from a class that implements Slick2D's
   Game interface. (Can be combined with item 1 above.)

3. Instantiate an object of Slick2D's AppGameContainer class, passing the
   Game object's reference (from item 2  above) as a parameter to the
   AppGameContainer constructor.

4. Call the start method on the object of type AppGameContainer
  (from item 3 above).
```

Table 2.1

2.4 Preview

The purpose of this module is to teach you about some of the characteristics of game engines in general, and to teach you how Slick2D fits those characteristics.

What you have learned

In the previous module, you learned how to download Slick2D and how to install Slick2D in such a way that you can easily compile and execute Slick2D programs from the command line with no need for a high level IDE such as Eclipse or NetBeans.

What you will learn

To begin with, you will learn what we often mean when we speak of a *"game engine."* You will also learn how that terminology relates to something that we often refer to as a *"software framework."*

You will learn how to write a minimal Java application in conjunction with a set of Slick2D jar files to create your own Slick2D game engine. Using that program as an example, you will learn about the overall structure of the Slick2D game engine.

You will learn that game engines are typically *service provider* programs and you will learn about a common set of services that is provided by many game engines.

You will learn about the two cooperating objects that form the heart of the Slick2D game engine.

You will learn about the methods declared in the interface named **Game** .

2.5 What is a game engine?

The term "game engine" is jargon for something that is more properly called a *"software framework."*

2.5.1 A software framework

Here is part of what Wikipedia [2] has to say about a software framework:

A *software framework* , in computer programming, is an abstraction in which common code providing *generic functionality* can be selectively overridden or specialized by user code providing *specific functionality* .

Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries.

Software frameworks have these distinguishing features that separate them from libraries or **normal user applications** :

1. **inversion of control** - In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.
2. **default behavior** - A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-ops.
3. **extensibility** - A framework can be extended by the user by selective overriding of framework code in order to provide specific functionality
4. **non-modifiable framework code** - The framework code, in general should not normally be modified by the user. Users can extend the framework, but normally should not modify its code.

In short, a framework is a computer program that helps you to write computer programs.

Not a game engine

Under this definition, Slick2D in its raw form is not a game engine nor is it a framework. Instead, it is a library of Java classes that you can use to create a framework or game engine.

In particular, if you combine the contents of the files named **slick.jar** and **lwjgl.jar** with the minimal Java application shown in Listing 1 (p. 19) and described in Figure 1 (p. 16) , you will have created a game engine. That combination is what I will refer to hereafter as the *Slick2D game engine* .

Having done that, the framework description given above (p. 17) is a good match for the Slick2D game engine.

2.6 Background information

Listing 1 (p. 19) shows a very simple Java application program. This is possibly the simplest program that can be written using Slick2D that will run as a Java application.

---

[2] http://en.wikipedia.org/wiki/Software_framework

(A different approach is used to create a Slick2D program that will run as a Java applet, but I probably won't get into Java applets in this collection of modules.)

I will use this program to explain the overall structure of the Slick2D game engine in this module. I will explain the inner workings of the game engine in more detail in future modules.

Listing 1 . Simplest Slick2D program.

```
/*Slick0110a.java
Copyright 2012, R.G.Baldwin

Possibly the simplest game that can be coded to use the
Slick2D game engine and run as a Java application.

Compile and run the program by executing the file named
CompileAndRun.bat.

Tested using JDK 1.7 under WinXP and Win 7
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Game;

public class Slick0110a implements Game{

  public static void main(String[] args)
                                  throws SlickException{
    AppGameContainer app =
                   new AppGameContainer(new Slick0110a());
    app.start();//this statement is required
  }//end main
  //----------------------------------------------------//

  public void init(GameContainer gc)
                                  throws SlickException {
    //empty body
  }
  //----------------------------------------------------//

  public void update(GameContainer gc, int delta)
                                  throws SlickException{
    //empty body
  }
  //----------------------------------------------------//

  public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
    //empty body
  }
  //----------------------------------------------------//
  public String getTitle(){
    return "Optional title";
  }//end getTitle
  //----------------------------------------------------//

  public boolean closeRequested(){
    return false;
  }//end closeRequested
}//end class Slick0110a
```

Table 2.2

A batch file named CompileAndRun

Listing 2 (p. 20) shows the contents of a batch file that you can use to compile and execute the code in Listing 1 (p. 19) as was explained in an earlier module.

Listing 2 . The file named CompileAndRun.bat.

```
echo off
del *.class

rem refer to jar files in the folder named jars
javac -cp .;../jars/slick.jar;../jars/lwjgl.jar Slick0110a.java

rem set the java.library.path and the classpath and run the program
java -Djava.library.path=../lwjglbin -cp .;../jars/slick.jar;../jars/lwjgl.jar Slick0110a

pause
```

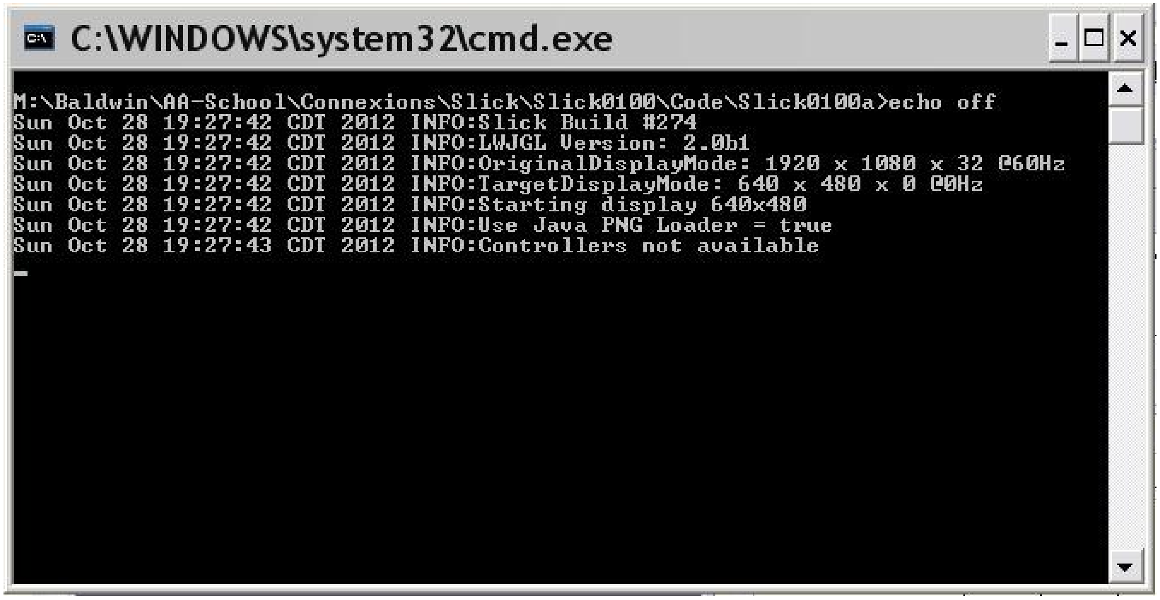
Table 2.3

Execute the batch file

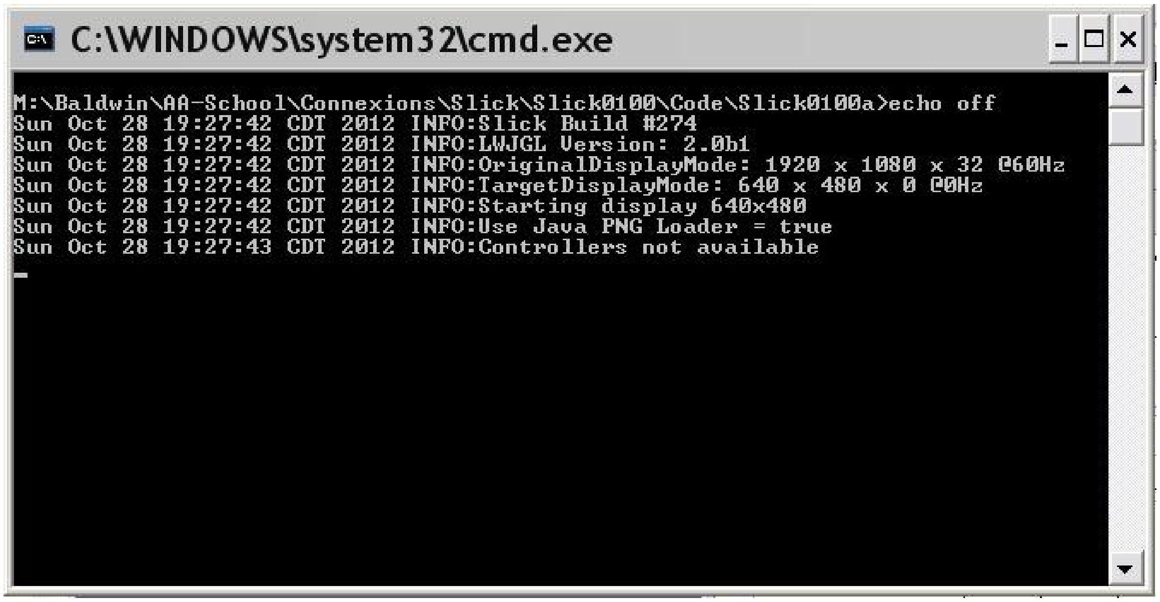
If you double-click the batch file named **CompileAndRun.bat** *(or execute it in whatever manner you prefer),* two new windows should appear on your computer screen.

Slick2D output during startup

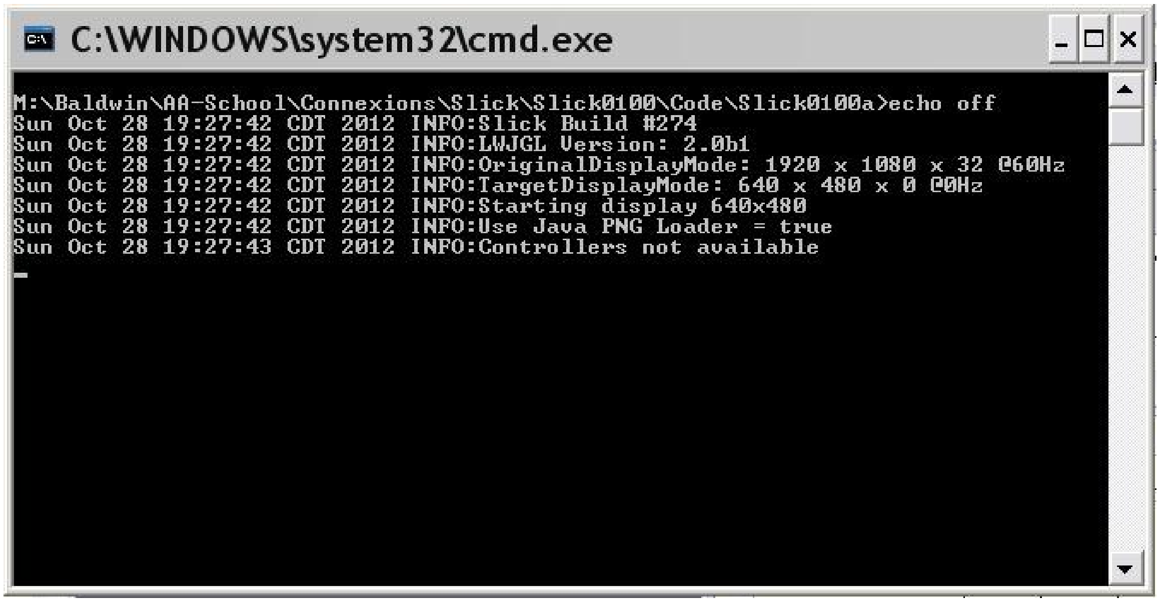
The first window to appear should look similar to Figure 2 (p. 22) .



```
M:\Baldwin\AA-School\Connexions\Slick\Slick0100\Code\Slick0100a>echo off
Sun Oct 28 19:27:42 CDT 2012 INFO:Slick Build #274
Sun Oct 28 19:27:42 CDT 2012 INFO:LWJGL Version: 2.0b1
Sun Oct 28 19:27:42 CDT 2012 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:Starting display 640x480
Sun Oct 28 19:27:42 CDT 2012 INFO:Use Java PNG Loader = true
Sun Oct 28 19:27:43 CDT 2012 INFO:Controllers not available
_
```

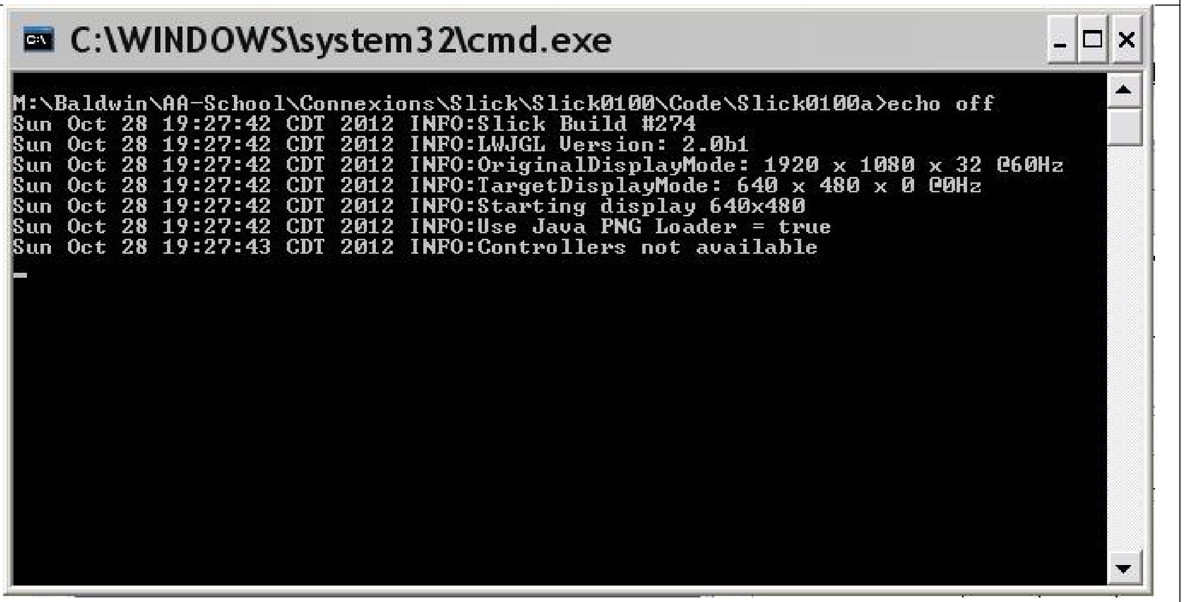


```
M:\Baldwin\AA-School\Connexions\Slick\Slick0100\Code\Slick0100a>echo off
Sun Oct 28 19:27:42 CDT 2012 INFO:Slick Build #274
Sun Oct 28 19:27:42 CDT 2012 INFO:LWJGL Version: 2.0b1
Sun Oct 28 19:27:42 CDT 2012 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:Starting display 640x480
Sun Oct 28 19:27:42 CDT 2012 INFO:Use Java PNG Loader = true
Sun Oct 28 19:27:43 CDT 2012 INFO:Controllers not available
```



```
M:\Baldwin\AA-School\Connexions\Slick\Slick0100\Code\Slick0100a>echo off
Sun Oct 28 19:27:42 CDT 2012 INFO:Slick Build #274
Sun Oct 28 19:27:42 CDT 2012 INFO:LWJGL Version: 2.0b1
Sun Oct 28 19:27:42 CDT 2012 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:Starting display 640x480
Sun Oct 28 19:27:42 CDT 2012 INFO:Use Java PNG Loader = true
Sun Oct 28 19:27:43 CDT 2012 INFO:Controllers not available
```

Figure 2 . Output from Slick2D during program startup.



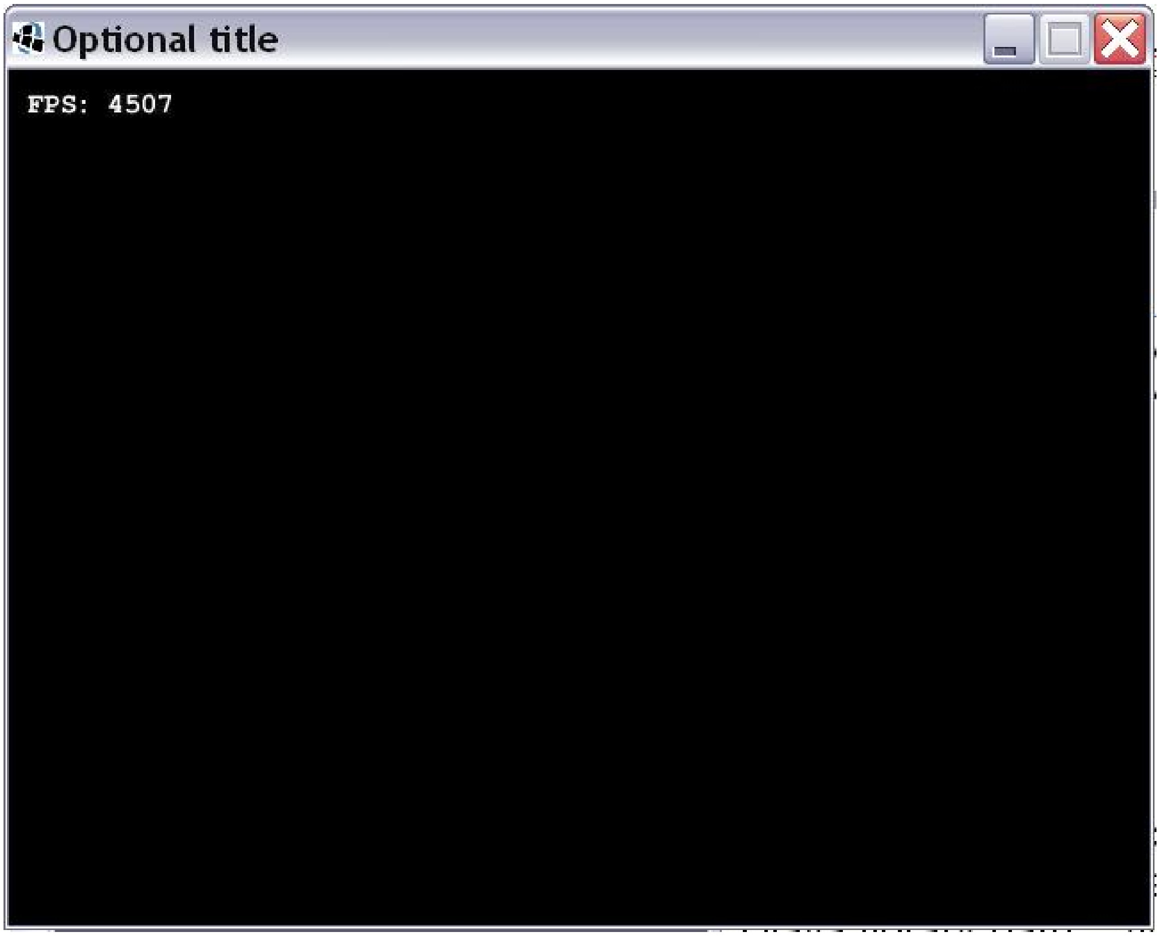
```
M:\Baldwin\AA-School\Connexions\Slick\Slick0100\Code\Slick0100a>echo off
Sun Oct 28 19:27:42 CDT 2012 INFO:Slick Build #274
Sun Oct 28 19:27:42 CDT 2012 INFO:LWJGL Version: 2.0b1
Sun Oct 28 19:27:42 CDT 2012 INFO:OriginalDisplayMode: 1920 x 1080 x 32 @60Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:TargetDisplayMode: 640 x 480 x 0 @0Hz
Sun Oct 28 19:27:42 CDT 2012 INFO:Starting display 640x480
Sun Oct 28 19:27:42 CDT 2012 INFO:Use Java PNG Loader = true
Sun Oct 28 19:27:43 CDT 2012 INFO:Controllers not available
_
```

Table 2.4

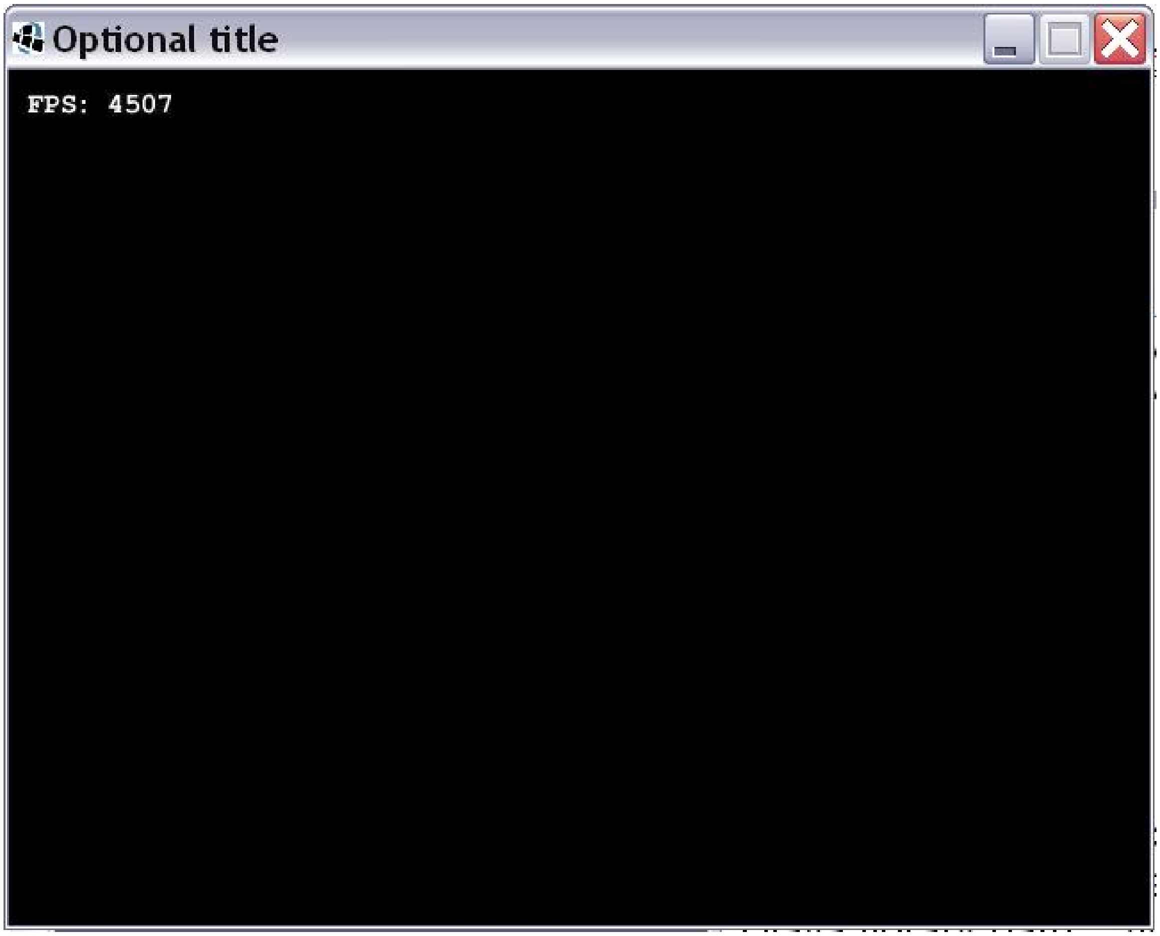
Figure 2 (p. 22) shows typical information produced by Slick2D when the compiled Slick2D program starts running.

A default Slick2D game window

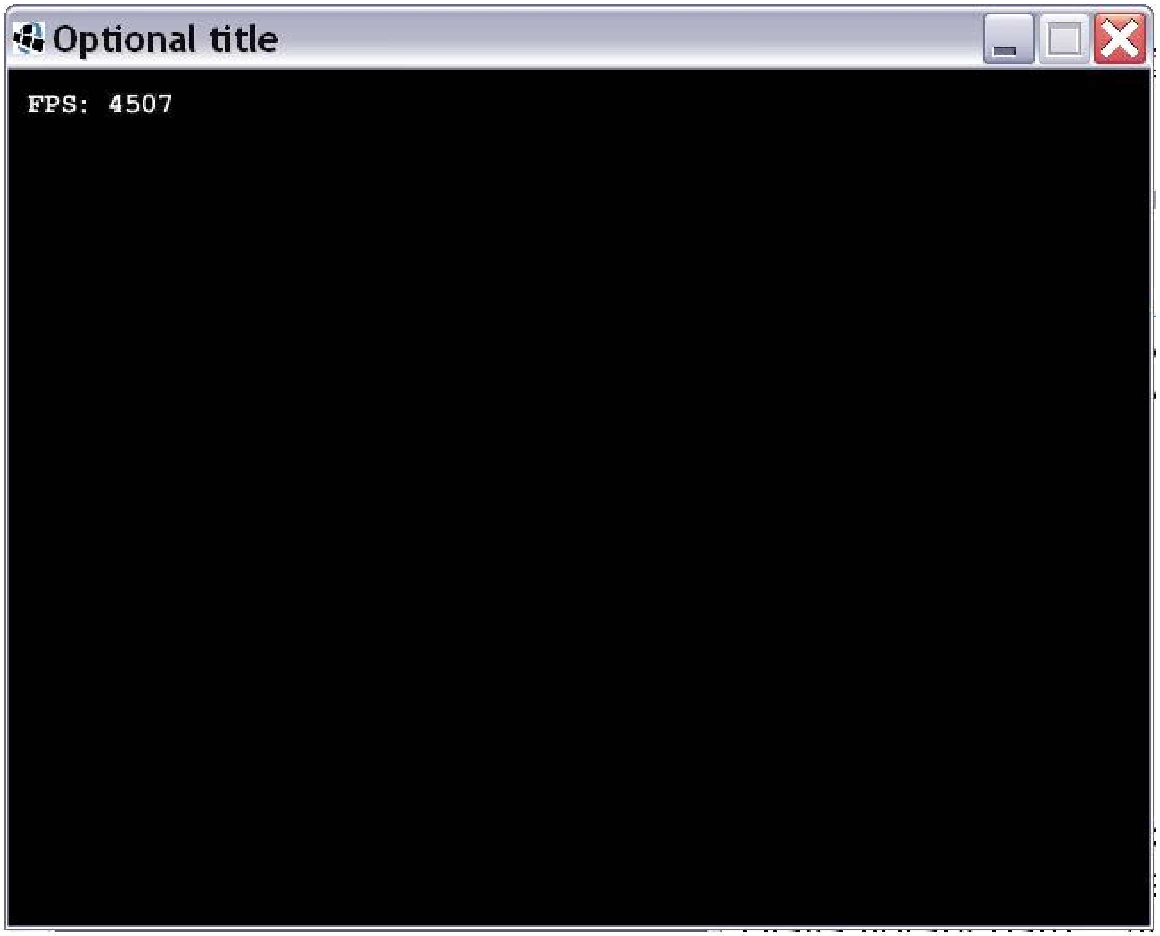
The second window to appear should look something like Figure 3 (p. 26) .



```
FPS: 4507
```

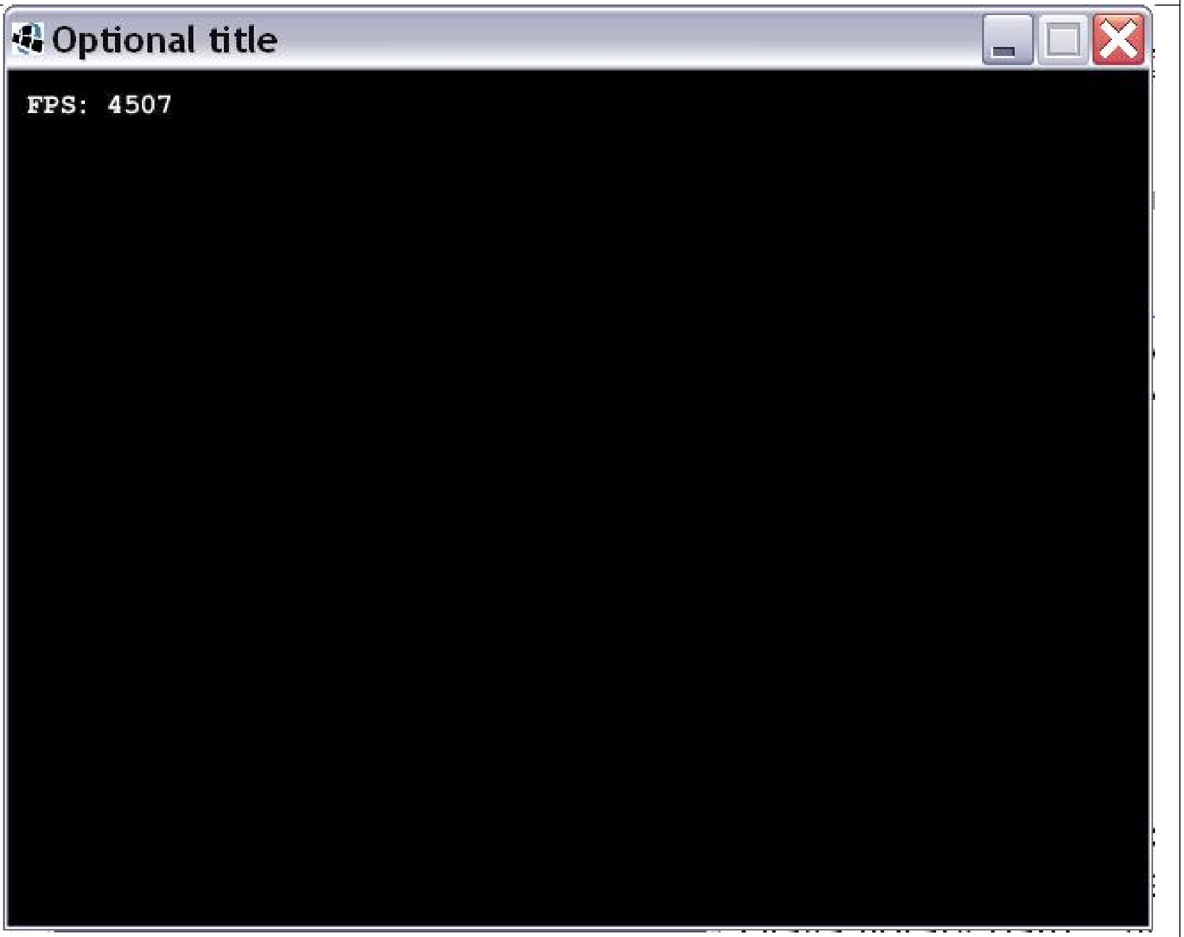


```
FPS: 4507
```



FPS: 4507

Figure 3 . A default Slick2D game window.



FPS: 4507

Table 2.5

Figure 3 (p. 26) is a default Slick2D game window. This Slick2D program *(see Listing 1 (p. 19))* has no interesting behavior. In effect, it is an "empty" game program. Therefore, the only thing showing in the game window is a counter in the top left corner that shows the execution rate in frames per second. *(I will show you how to control the execution rate in a future module.)*

The appearance of this empty game window matches the second item in the above list (p. 17) titled **default behavior** . In particular, the default behavior of the Slick2D game engine is to display an empty game window with an active FPS counter in the upper-left corner..

2.7 Discussion and sample code

2.7.1 A service provider program

Many game engines, *(and the Slick2D game engine is no exception)* , are *"service provider"* programs. They provide services that make it easier to write game programs than would otherwise be the case if you were to *"start from scratch"* to write a game program.

Different game engines provide different services. However, most game engines, including Slick2D, provide at least a minimum set of services, which includes the following:

- An opportunity to initialize the game state.
- Overall flow control that includes a game loop, which alternates in some fashion between
    - an *update* phase, in which the game state is updated, and
    - a *rendering* phase in which portions of the game state may be displayed for the benefit of the player.

These services are provided in different ways in different game engines. You will learn how they are provided by the Slick2D game engine in future modules.

2.7.2 Two primary objects

At a minimum, a Slick2D game that runs as an application *(not an applet)* consists of at least two **cooperating objects** :

1. An object instantiated from a subclass of the Slick2D class named **GameContainer** .
2. An object instantiated from a class that implements the Slick2D interface named **Game** .

(For the remainder of this and future modules, unless I specifically indicate that I am discussing a Slick2D game applet, you can assume that I am talking about a Slick2D game program that runs as an application.)

2.7.2.1 Behavior of an object of the AppGameContainer class

The **GameContainer** object *(item 1 above (p. 27))* , and the behavior of its methods, manages the game play after the game program starts running. For example, this is the object that manages the game loop.

For the program shown in Listing 1 (p. 19) , this object is an object of the class named **AppGameContainer** , which extends the class named **GameContainer** .

The **AppGameContainer** class provides many public methods by which you can manipulate the behavior of the container object. However, the authors of the Slick2D library did not intend for you to physically modify the source code in the **GameContainer** class or the **AppGameContainer** class.

2.7.2.2 Behavior of an object that implements the Game interface

The **Game** object *(item 2 above (p. 27))* , and the behavior of its methods is what distinguishes one Slick2D game from another Slick2D game. The authors of the Slick2D library did intend for you to physically modify the source code in the class that implements the **Game** interface. This is how you distinguish your game from games written by others.

*(Clarification: See a later discussion of a class named **BasicGame** , which implements the **Game** interface. The authors of the Slick2D library did not intend for you to modify the source code in the **BasicGame** class. Instead, they intended for you to subclass that class and to modify the behavior of the **Game** object by overriding inherited abstract methods.)*

2.7.3 Beginning of the class named Slick0110a

Consider the code fragment shown in Listing 3 (p. 28) . *(This code was extracted from Listing 1 (p. 19) to make it easier to discuss.)*

Listing 3 . Beginning of the class named Slick0110a.

```
public class Slick0110a implements Game{

  public static void main(String[] args)
                                  throws SlickException{
    AppGameContainer app =
                   new AppGameContainer(new Slick0110a());
    app.start();//this statement is required
  }//end main
```

Table 2.6

Listing 3 (p. 28) shows the beginning of the class named **Slick0110a** along with the entire **main** method.

An object of the interface type Game

To begin with, note that the **Slick0110a** class implements the interface named **Game** . Therefore, an object of this class satisfies the requirement for the second type of object identified as item 2 in the above list (p. 27) . In other words, an object of this class is an object of the interface type **Game** .

An object of the AppGameContainer class

Now note the first statement in the **main** method in Listing 3 (p. 28) that instantiates a new object of the class named **AppGameContainer** and saves that object's reference in the local variable named **app** . This object satisfies the requirement for the first type of primary object identified as item 1 in the above list (p. 27) . In other words, an object of this class is an object of the type **GameContainer** because the class named **AppGameContainer** is a subclass of **GameContainer** .

Tying the two objects together

The class named **AppGameContainer** provides two overloaded constructors, each of which requires an incoming parameter that is a reference to an object instantiated from a class that implements the interface named **Game** . The code in the **main** method in Listing 3 (p. 28) uses the simpler of the two overloaded constructors to instantiate a new object of the class named **AppGameContainer** and to save its reference in the local variable named **app** .

The code in Listing 3 (p. 28) also instantiates a new object of the class named **Slick0110a** and passes that object's reference to the constructor for the class named **AppGameContainer** . This is legal because the class named **Slick0110a** implements the interface named **Game** .

At this point, the two objects described in the above list (p. 27) exist and occupy memory. From this point forward, the *container* object knows about the *game* object and has access to its members.

Start the game program running

Finally, the last statement in the **main** method calls the **start** method on the new container object to cause the program to be initialized and to cause the game loop to start running.

2.7.4 The Game interface

One of the rules in Java programming is that whenever a new class definition inherits an abstract method declaration, either the new class definition must provide a concrete definition for the abstract method or the class itself must be declared abstract.

All of the methods declared in an interface are implicitly abstract. Therefore, whenever a new class definition implements an interface that declares methods, it inherits one or more abstract method declarations and the above rule applies.

What is a concrete method definition

Not much is required to provide a concrete method definition. All that is necessary to define concrete methods that return **void** is to replicate the signature of the abstract method and to provide an empty body delineated by a pair of empty curly brackets. If the return type for the abstract method is not void, the body of the concrete version must contain a **return** statement that matches the specified return type.

Five abstract methods

The interface named **Game** declares the five abstract methods shown in **Figure 4** :

Figure 4 . Abstract methods declared in the Game interface.

```
boolean closeRequested() - Notification that a game close has been requested
 Returns: True if the game should close

String getTitle() - Get the title of this game
 Returns: The title of the game

void init(GameContainer container) throws SlickException - Initialise the game.
 This can be used to load static resources. It's called before the game loop starts
 Parameters: container - The container holding the game

void render(GameContainer container, Graphics g) throws SlickException - Render the
 game's screen here.
 Parameters: container - The container holing this game,
             g - The graphics context that can be used to render. However,
             normal rendering routines can also be used.

void update(GameContainer container,int delta) throws SlickException - Update the
 game logic here. No rendering should take place in this method though it won't
 do any harm.
 Parameters: container - The container holding this game,
             delta - The amount of time that has passed since last update
             in milliseconds
```

Table 2.7

Concrete versions of the inherited abstract methods

As you can see in Listing 1 (p. 19) , the new class named **Slick0110a** is not declared abstract. Therefore, it must provide concrete versions of the inherited abstract methods shown in Figure 4 (p. 29) .

The **init** , **update** , and **render** methods in Listing 1 (p. 19) return void and are defined with empty bodies. The **getTitle** and **closeRequested** methods do not return void. Therefore each of these concrete versions contains a **return** statement of the required type.

Not much fun to play

As you learned earlier, the skeleton code for this Slick2D game program shown in Listing 1 (p. 19) can be compiled and executed. However, it isn't very much fun to play because it doesn't do anything other than to sit there and display the frames per second *(FPS)* rate in the upper-left corner of the game window. Make no mistake about it, however, the game loop is running meaning that the game is active.

Not the recommended form

While this is probably the simplest Slick2D game program that can be written to run as a Java application, it is not the recommended form for an empty Slick2D game skeleton. You will learn in the next module that instead of implementing the **Game** interface directly, it is better to extend a Slick2D helper class named

BasicGame that implements the **Game** interface and provides some additional services that may be useful to the game programmer. However, even when you do that, it is still necessary to write code to put some meat on the skeleton's bones to create a playable game.

2.8 Run the program

I encourage you to copy the code from Listing 1 (p. 20) and Listing 2 (p. 20) Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

2.9 Summary

The main purpose of this module is to teach you about some of the characteristics of game engines and frameworks in general, and to teach you how Slick2D fits those characteristics.

More specifically, you learned what we often mean when we speak of a *"game engine."* You learned how that terminology relates to something that we often refer to as a *"software framework."*

You learned how to write a minimal Java application in conjunction with a set of Slick2D jar files to create your own Slick2D game engine. Using that program as an example, you learned about the overall structure of the Slick2D game engine.

You learned that game engines are typically *service provider* programs and you learned about a common set of services that is provided by most game engines.

You learned about the two cooperating objects that form the heart of the Slick2D game engine.

You learned about the methods declared in the interface named **Game** .

You learned that in order to write a game program using the Slick2D game engine that will run as a Java application, you must, as a minimum, perform the steps shown in Figure 1 (p. 16) .

2.10 What's next?

In the next module, I will begin explaining the purpose of the methods that are inherited from the **Game** interface and will begin showing how you can override those methods to control the behavior of your Slick2D game program.

2.11 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0110: Overview
- File: Slick0110.htm
- Published: 02/03/13
- Revised 06/09/15 for 64-bit

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Chapter 3

Slick0120: Starting your program[1]

3.1 Table of Contents

3.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

3.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the listings while you are reading about them.

---

[1] This content is available online at <http://cnx.org/content/m45732/1.3/>.

3.2.1.1 Listings

- Listing 1 (p. 35) . The main method.
- Listing 2 (p. 37) . The start method of the AppGameContainer class.
- Listing 3 (p. 38) . Constructor for the AppGameContainer class that takes a single parameter.
- Listing 4 (p. 38) . Constructor for the AppGameContainer class that takes four parameters.
- Listing 5 (p. 39) . The getDelta method of the GameContainer class.
- Listing 6 (p. 42) . Source code for Slick0120a.java

3.3 Preview

The main purpose of this module is to analyze the behavior of the Slick2D game engine when you start a Slick2D game running.

What you have learned

In previous modules, you learned how to download Slick2D and how to install Slick2D in such a way that you can easily compile and execute Slick2D programs from the command line with no need for a high level IDE such as Eclipse or NetBeans.

You also learned what we often mean when we speak of a *"game engine"* and how that terminology relates to a *"software framework."*

You learned how to write a minimal Java application in conjunction with a set of Slick2D jar files to create your own Slick2D game engine. Using that program as an example, you learned about the overall structure of the Slick2D game engine.

You learned that game engines are typically *service provider* programs and you learned about a common set of services that is provided by most game engines.

You learned about the two cooperating objects that form the heart of the Slick2D game engine.

- An object instantiated from a subclass of the Slick2D class named **GameContainer** .
- An object instantiated from a class that implements the Slick2D interface named **Game** .

And last but not least, you learned about the five abstract methods declared in the interface named **Game** :

- boolean closeRequested()
- String getTitle()
- void init(GameContainer container)
- void render(GameContainer container, Graphics g)
- void update(GameContainer container, int delta)

What you will learn

You will learn how and why you should extend the **BasicGame** class instead of implementing the **Game** interface directly.

You will learn about the constructors for the **AppGameContainer** class.

You learned earlier that you need to call the **start** method on an object of the **AppGameContainer** class to cause your Slick2D game program to start running. You will learn that the **start** method calls the following three methods:

- setup
- getDelta
- gameLoop

You will learn about the behavior of the **setup** and **getDelta** methods in this module. An explanation of the **gameLoop** method will be deferred until the next module.

3.4 General background information

Listing 6 (p. 42) shows the skeleton code for a basic game class named **Slick0120a** . This code differs from the skeleton code presented in earlier modules in two important respects:

1. **The class named Slick0120a extends** the Slick2D class named **BasicGame** instead of extending **Object** and implementing the Slick2D interface named **Game** .
2. **The class named Slick0120a does not override** the methods named **getTitle** and **closeRequested** . *(They are overridden with default behavior in the* ***BasicGame*** *class.)* Instead, it overrides only the following methods that are declared in the Slick2D **Game** interface:
   a. init
   b. update
   c. render

The class named BasicGame

Regarding the first item (p. 35) in the above list, while it is technically possible to write a Slick2D game program by implementing the **Game** interface directly, the Slick2D helper class named **BasicGame** implements the **Game** interface and provides a number of useful methods as well. Therefore, by extending the **BasicGame** class, you not only implement the **Game** interface, you also get the benefit of the methods that are defined in the **BasicGame** class.

The methods named init, update, and render

Note, however that the **Basic** game class does not define concrete versions of the methods named **init** , **update** , and **render** . Therefore, you are still required to provide concrete versions of those methods in your class that extends the **BasicGame** class *(or some subclass of that class)* .

The class named **Slick0120a** does provide concrete versions of methods as indicated in the second item (p. 35) in the above list.

The methods named getTitle and closeRequested

Further regarding the second item (p. 35) in the above list, the class named **BasicGame** does provide concrete versions of the methods named **getTitle** and **closeRequested** . Therefore, unless you need to provide different behavior for those two methods, you don't need to override them in your new class that extends the **BasicGame** class.

3.5 Discussion and sample code

Listing 1 (p. 35) shows the **main** method for our new class named **Slick0120a** .

Listing 1 . Listing 1 . The main method.

```
public static void main(String[] args)
                                  throws SlickException{
  AppGameContainer app =
                 new AppGameContainer(new Slick0120a());
  app.start();//this statement is required
}//end main
```

Table 3.1

We will dissect this code to make certain that we understand what it means and why we need it.

3.5.1 Two primary objects

You learned in an earlier module that a Slick2D game that runs as an application *(not an applet)* consists of at least two cooperating objects:

1. **An object instantiated from a subclass of the Slick2D class** named **GameContainer** .
2. **An object instantiated from a class that implements the Slick2D interface** named **Game** .

3.5.1.1 Behavior of an object of the AppGameContainer class

The **GameContainer** object *(item 1 above (p. 36))* manages the program startup and the game play after the game program starts running. For example, this is the object that manages the game loop.

As shown in Listing 1 (p. 35) , for the program named **Slick0120a** , this object is an object of the class named **AppGameContainer** , which extends the class named **GameContainer** .

The **AppGameContainer** class provides many public methods *(including the method named **start** , which is called in Listing 1 (p. 35))* by which you can manipulate the behavior of the container object. The authors of the Slick2D library did not intend for you to physically modify the source code in the **GameContainer** class or the **AppGameContainer** class.

3.5.1.2 Behavior of an object that implements the Game interface

The behaviors of the methods of the **Game** object *(item 2 above (p. 36))* are what distinguishes one Slick2D game from another Slick2D game.

You need not implement the **Game** interface directly. The authors of the Slick2D library provided a helper class named **BasicGame** that implements the **Game** interface and provides a number of useful methods. They intended for you to extend the **BasicGame** class and to override at least three of the methods declared in the **Game** interface in order to provide the desired behavior for your game..

As mentioned earlier, the class named **Slick0120a** extends the **BasicGame** class, thereby implementing the **Game** interface and getting the benefit of methods defined in the **BasicGame** class.

The code in the **main** method in Listing 1 (p. 35) instantiates an object of the **Slick0120a** class and passes that object's reference to the constructor for the class named **AppGameContainer** . Therefore, Listing 1 (p. 35) instantiates both of the required objects and connects them in the manner intended by the authors of the Slick2D library.

3.5.2 Starting the game

The main purpose of this module is to analyze the behavior of the Slick2D game engine when you start a Slick2D game running.

Listing 1 (p. 35) calls the **start** method on a reference to the **AppGameContainer** object. The source code for the **start** method is shown in Listing 2 (p. 37) .

Listing 2 . The start method of the AppGameContainer class.

```
public void start() throws SlickException {
  try {
    setup();

    getDelta();
    while (running()) {
      gameLoop();
    }
  } finally {
    destroy();
  }

  if (forceExit) {
    System.exit(0);
  }
}//end start
```

Table 3.2

Copyright and license information

Copyright and license information: I was unable to find any copyright or license information in the zip file that I downloaded from http://slick.ninjacave.com/ [2] . I acknowledge that I am not the author of the code in that zip file and the copyrights for that material are held by someone other than myself.

Constructors and methods

This and the next few modules will explore and discuss the constructors for the **AppGameContainer** class *(see Listing 1 (p. 35))* along with salient aspects of the following methods that are called in Listing 2 (p. 37) :

- setup
- getDelta
- gameLoop

3.5.3 The constructors for the AppGameContainer class

Listing 1 (p. 35) instantiates a new object of the **AppGameContainer** class by calling a constructor that takes a single parameter of the Slick2D interface type **Game** .

The source code for that constructor is shown in Listing 3 (p. 38) .

---

[2] http://slick.ninjacave.com/

Listing 3 . Constructor for the AppGameContainer class that takes a single parameter.

```
public AppGameContainer(Game game) throws SlickException {
  this(game,640,480,false);
}//end constructor
```

Table 3.3

A constructor with four parameters

The code in Listing 3 (p. 38) simply calls another overloaded version of the constructor passing four default parameters that specify a game window of 640x480 pixels.

The constructor that takes four parameters is shown in Listing 4 (p. 38) .

Listing 4 . Constructor for the AppGameContainer class that takes four parameters.

```
public AppGameContainer(Game game,
                        int width,
                        int height,
                        boolean fullscreen)
                                 throws SlickException {
  super(game);

  originalDisplayMode = Display.getDisplayMode();

  setDisplayMode(width,height,fullscreen);
}//end constructor
```

Table 3.4

The first parameter is a reference to the game that is to be wrapped by the **GameContainer** object. The code in Listing 4 (p. 38) passes that reference to its superclass, **GameContainer** , where it is saved in a *protected* variable of type **Game** named **game** . As a *protected* variable, it is accessible to all of the methods of the **AppGameContainer** class for use later.

Then Listing 4 (p. 38) saves the current display mode in a variable named **originalDisplayMode** , presumably to be used later.

Finally, Listing 4 (p. 38) calls the method named **setDisplayMode** to set the display mode to match the incoming parameters.

(This is the constructor that you would use if you wanted to cause the size of the game window to be something other than the default of 640 by 480 pixels.)

3.5.4 The setup method of the AppGameContainer class

The **setup** method is fairly long and complicated. Most of the code in the method has to do with the creation and formatting of the game window. I will skip over that code and leave it as an exercise for interested students to analyze.

Initialization of the game

Finally a statement near the end of the **setup** method calls a method named **init** on a reference to the **Game** object, passing a reference to the object of type **AppGameContainer** as a parameter.

This is what we would refer to as a callback that uses the reference to the **Game** object that was passed to the constructor to call the method named **init** on the **Game** object.

The effect is to call the method named **init** belonging to the game program shown in Listing 6 (p. 42) . This causes the initialization code *(if any)* that you have written into the overridden **init** method to be executed. If the overridden version of the method has an empty body *(as in Listing 6* (p. 42) *)* , it simply returns without doing anything. This is how your game gets initialized.

3.5.5 The getDelta method of the GameContainer class

The **AppGameContainer** class inherits a *protected* method named **getDelta** from its superclass named **GameContainer** .

The **getDelta** method is called from the **start** method shown in Listing 2 (p. 37) .

What is delta?

An **int** parameter named **delta** is received by the **update** method shown in Listing 6 (p. 42) . *(The* ***update*** *method s a concrete version of the method having the same signature that is declared in the* ***Game*** *interface.)*

According to the documentation for the **update** method in the **Game** interface, delta is

"The amount of time that has passed since last update in milliseconds"

Having that time available can be valuable in some game programs. For example, you might like for one of the actors to light a fuse on a bomb and have that bomb detonate some given number of milliseconds later. In that case, the program would need real time information to know when to detonate the bomb.

Listing 5 (p. 39) shows the source code for the **getDelta** method.

Listing 5 . The getDelta method of the GameContainer class.

```
protected int getDelta() {
  long time = getTime();
  int delta = (int) (time - lastFrame);
  lastFrame = time;

  return delta;
}//end getDelta method
```

Table 3.5

Without getting into the details, the method named **getTime** that is called in Listing 5 (p. 39) returns the amount of time, *(with a resolution of one millisecond)* , that has elapsed since a historical point in time before the game started running.

The **GameContainer** class contains a *protected* instance variable of type **long** named **lastFrame** that is used to store a time value.

The code in Listing 5 (p. 39)

- subtracts the time value stored in **lastFrame** from the current time,
- converts the time difference to type **int** , saving it in **delta** , and
- stores the current time in **lastFrame** .

The difference between the two time values represents a time interval and that difference is returned as type **int** .

Various methods in the **AppGameContainer** and **GameContainer** classes call the **getDelta** method in such a way that the value of delta represents the time required to update and render one frame when the program is running. *(There are some other options as well that I may discuss in a future module.)*

When the method named **update** is called in Listing 6 (p. 42) , the incoming parameter named **delta** contains the number of milliseconds that have elapsed since the last time that the **update** method was called.

When the method named **getDelta** is called in Listing 2 (p. 37) , the return value is discarded. This suggests that the call to the **getDelta** method in Listing 2 (p. 37) is made simply to cause the variable named **lastFrame** to be initialized with time that the **start** method was called.

3.5.6 The gameLoop method of the AppGameContainer class

That leaves us with one more method call from Listing 2 (p. 37) that we need to examine – **gameLoop** . I anticipate that will be a fairly long discussion, so I am going to defer that discussion until the next module.

3.6 Run the program

As explained earlier, the skeleton code in Listing 6 (p. 42) is different from the skeleton code that I presented in earlier modules. Therefore, I encourage you to copy the code from Listing 6 (p. 42) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

3.7 Summary

The main purpose of this module was to analyze the behavior of the Slick2D game engine when you call the **start** method to cause a Slick2D game program to start running.

You learned how and why you should extend the **BasicGame** class instead of implementing the **Game** interface directly.

You learned about the behavior of the constructors for the **AppGameContainer** class.

You learned that the **start** method of the **AppGameContainer** class calls the following three methods:

- setup
- getDelta
- gameLoop

You learned about the behavior of the **setup** and **getDelta** methods in this module.

3.8 What's next?

I will provide an explanation of the **gameLoop** method in the next module.

3.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0120: Starting your program

- File: Slick0120.htm
- Published: 02/04/13
- Revised: 06/09/15

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

3.10 Complete program listing

Listing 6 (p. 42) provides a complete listing for the skeleton program named **Slick0120a** .

Listing 6 . Source code for Slick0120a.java.

```
/*Slick0120a.java
Copyright 2012, R.G.Baldwin

Skeleton code for a basic game.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.SlickException;

public class Slick0120a extends BasicGame{

  public Slick0120a(){
    //Call to superclass constructor is required.
    super("Slick0120a, Baldwin.");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    AppGameContainer app =
                  new AppGameContainer(new Slick0120a());
    app.start();//this statement is required
  }//end main
  //----------------------------------------------------//

  @Override
  public void init(GameContainer gc)
                                   throws SlickException {
    //No initialization needed for this program.
  }//end init
  //----------------------------------------------------//

  @Override
  public void update(GameContainer gc, int delta)
                                    throws SlickException{
    //Put game logic here
  }//end update
  //----------------------------------------------------//


  public void render(GameContainer gc, Graphics g)
                                    throws SlickException{
    //Put drawing code here.
  }//end render

}//end class Slick0120a
```

Table 3.6

-end-

Chapter 4

Slick0130: The game loop[1]

4.1 Table of Contents

4.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

---

[1] This content is available online at <http://cnx.org/content/m45733/1.4/>.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

4.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

4.2.1.1 Figures

- Figure 1 (p. 55) . Screen output from program named Slick0130a.

4.2.1.2 Listings

- Listing 1 (p. 48) . The start method of the AppGameContainer class.
- Listing 2 (p. 49) . The gameLoop method of the AppGameContainer class.
- Listing 3 (p. 50) . Beginning of the updateAndRender method of the GameContainer class.
- Listing 4 (p. 52) . The remainder of the updateAndRender method of the GameContainer class.
- Listing 5 (p. 56) . Beginning of the class named Slick0130a.
- Listing 6 (p. 57) . The main method.
- Listing 7 (p. 57) . The overridden init method.
- Listing 8 (p. 58) . The overridden update method.
- Listing 9 (p. 59) . The overridden render method.
- Listing 10 (p. 62) . Source code for the program named Slick0130a.

4.3 Preview

What you have learned

The main purpose of this and the previous module is to analyze the behavior of the Slick2D game engine when you call the **start** method to cause a Slick2D game program to start running.

In the previous module, you learned how and why you should extend the **BasicGame** class instead of implementing the **Game** interface directly.

You learned about the behavior of the constructors for the **AppGameContainer** class.

You learned that the **start** method of the **AppGameContainer** class *(see Listing 1* (p. 48)) calls the following three methods:

- setup
- getDelta
- gameLoop

You learned about the behavior of the **setup** and **getDelta** methods.

What you will learn

I will explain the overall behavior of the **gameLoop** method in this module.

In addition, you will learn

- about a property of the **GameContainer** class named **running** , and how it is used by the **start** method to keep the game loop running,
- about the salient features of the **gameLoop** method of the **AppGameContainer** class,
- about the **updateAndRender** method of the **GameContainer** class and how it decides when and if to call the **update** and **render** methods of the object of the **Game** class that is wrapped in the container,

- about the difference between normal delta and smoothed delta,
- about **minimumLogicInterval** and **maximumLogicInterval** and how the contents of those two variables are used to determine if, when, and how many times to call the **update** method during each iteration of the game loop,
- how the contents of **minimumLogicInterval** and **maximumLogicInterval** are used to determine the value that is passed as delta each time the **update** method is called,
- that the **render** method is normally called once and only once during each iteration of the game loop,
- how you can use the value of delta that is received by the **update** method to control the behavior of a game program,
- that you can set the size of the game window when you instantiate an object of the **AppGameContainer** class by passing dimension parameters to the constructor,
- that you can set the target frame rate by calling the **setTargetFrameRate** method on the **GameContainer** object, and
- how to display text in the game window.

4.4 General background information

As you learned in the previous module, the **start** method *(see Listing 1* (p. 48)) calls the **setup** method and then calls the **getDelta** method. Following that, it calls the **gameLoop** method as described below.

4.4.1 The property named running

The **GameContainer** class declares a protected **boolean** variable named **running** , which is inherited into the object of the **AppGameContainer** class. The descriptive comment reads *"True if we're currently running the game loop."*

The initial value of this variable is true and as near as I can tell, it only goes false

- when the **exit** method is called,
- when the **closeRequested** method is called and returns true, or
- when some code in the game throws a **SlickException** .

Calling the gameLoop method

The **start** method in Listing 1 (p. 48) shows a call to the **gameLoop** method inside a **while** loop with a call to the **running** method as the conditional clause.

Listing 1 . The start method of the AppGameContainer class.

```
public void start() throws SlickException {
  try {
    setup();

    getDelta();
    while (running()) {
      gameLoop();
    }
  } finally {
    destroy();
  }

  if (forceExit) {
    System.exit(0);
  }
}//end start
```

Table 4.1

As you can see in Listing 1 (p. 48) , the **gameLoop** method is called repeatedly while the variable named **running** is true. Each time it returns, it is called again.

I will refer to each call to the **gameLoop** method as one iteration of the game loop in the discussion that follows.

4.4.2 The gameLoop method

The **gameLoop** method of the **AppGameContainer** class is shown in Listing 2 (p. 49) .

Listing 2 . The gameLoop method of the AppGameContainer class.

```
protected void gameLoop() throws SlickException {
  int delta = getDelta();
  if (!Display.isVisible() && updateOnlyOnVisible) {
    try { Thread.sleep(100); } catch (Exception e) {}
  } else {
    try {
      updateAndRender(delta);
    } catch (SlickException e) {
      Log.error(e);
      running = false;
      return;
    }//end catch
  }//end else

  updateFPS();

  Display.update();

  if (Display.isCloseRequested()) {
    if (game.closeRequested()) {
      running = false;
    }//end if
  }//end if
}//end gameLoop method
```

Table 4.2

A verbal description

This is my verbal description of what happens each time the **start** method calls the **gameLoop** method.

First the **gameLoop** method gets the value for delta *(the elapsed time since the call to the gameLoop method during the previous iteration of the game loop)* .

If the display *(the game window)* is not visible and a property named **updateOnlyOnVisible** is true, the **gameLoop** method takes appropriate action. I will leave it as an exercise for interested students to analyze those actions. *(The program goes to sleep for 100 milliseconds.)*

If the display is visible, the **gameLoop** method calls the **updateAndRender** method of the **GameContainer** class passing delta as a parameter. Upon return, the **gameLoop** method performs some housekeeping tasks and terminates.

4.4.3 The updateAndRender method

Listing 3 (p. 50) shows the beginning of the **updateAndRender** method of the **GameContainer** class. This is the code that controls calls to the **update** method. The code that controls calls to the **render** method is shown in Listing 4 (p. 52) later.

Listing 3 . Beginning of the updateAndRender method of the GameContainer class.

```
protected void updateAndRender(int delta)
                                 throws SlickException {
  if (smoothDeltas) {
    if (getFPS() != 0) {
      delta = 1000 / getFPS();
    }//end if
  }//end if

  input.poll(width, height);

  Music.poll(delta);
  if (!paused) {
    storedDelta += delta;

    if (storedDelta >= minimumLogicInterval) {
      try {
        if (maximumLogicInterval != 0) {
          long cycles =
                     storedDelta / maximumLogicInterval;
          for (int i=0;i<cycles;i++) {
            game.update(this,(int)maximumLogicInterval);
          }//end for loop

          int remainder =
                    (int)(delta % maximumLogicInterval);
          if (remainder > minimumLogicInterval) {
            game.update(
              this,(int)(delta % maximumLogicInterval));
            storedDelta = 0;
          } else {
            storedDelta = remainder;
          }//end else
        } else {
          game.update(this, (int) storedDelta);
          storedDelta = 0;
        }//end else

      } catch (Throwable e) {
        Log.error(e);
        throw new SlickException(
        "Game.update() failure - check the game code.");
      }//end catch
    }//end if on minimumLogicInterval
  } else {
    game.update(this, 0);
  }//end else
```

Table 4.3

Another verbal description

This is my verbal description of what happens when the **gameLoop** method calls the **updateAndRender** method.

(I will not discuss those actions that represent communication with the hardware via the Lightweight Java Game Library (lwjgl) as well as a few other housekeeping actions) .

The **updateAndRender** method begins by selecting between normal delta and *"smooth deltas."*

(Smooth delta values essentially represent a moving average of individual delta values computed in a somewhat roundabout way.)

Calling the update and render methods

The **update** method of the **Game** object will be called none, one, or more times during each iteration of the game loop on the basis of the contents of two variables named **minimumLogicInterval** and **maximumLogicInterval** .

The default value for **minimumLogicInterval** is 1. The default value for **maximumLogicInterval** is 0. Methods are provided by which you can change the values of these two variables.

The **render** method of the **Game** object will be called only once during each iteration of the game loop following the call or calls to the **update** method.

4.4.3.1 Calls to the update method

If the **paused** property is true, the **update** method is called once passing a value of zero for delta. Otherwise, the value of delta is added to the contents of a variable named **storedDelta** for the purpose of accumulating individual delta values.

Then the method enters a somewhat complex logic process, which I will describe as follows:

- If **storedDelta** is less than **minimumLogicInterval** , don't call **update** during this iteration of the game loop.
- If **storedDelta** is greater than or equal to **minimumLogicInterval** and **maximumLogicInterval** has a value of 0, call the **update** method once passing **storedDelta** as a parameter. Then set **storedDelta** to zero to set the accumulated value back to 0.
- If **storedDelta** is greater than **minimumLogicInterval** and **maximumLogicInterval** is not equal to zero, call the **update** method several times in succession *(if needed)* during this iteration of the game loop, passing a value for delta during each call that is less than or equal to **storedDelta** . Continue this process until the sum of the delta values passed in the method calls equals **storedDelta** .

Possible outcomes

This algorithm results in the following possible outcomes regarding calls to the **update** method during each iteration of the game loop prior to calling the **render** method:

1. **No call at all** .
2. **One call with a delta value of zero** .
3. **One call with a non-zero value for delta** .
4. **Multiple calls, each with a non-zero value for delta** .

Analysis of the outcomes

Item 1 (p. 51) represents a situation where you don't want to execute **update** code for values of delta that are below a certain threshold and you prefer to execute **update** code less frequently using accumulated values of delta instead.

Item 2 (p. 51) represents a situation where the **paused** property has been set to true and no updates should be performed. *(This situation is indicated by a delta value of zero, which can be tested by code in the update method.)*

Item 3 (p. 51) represents a situation where you are willing to execute the code in the **update** method once during each iteration of the game loop using the incoming value of delta.

Item 4 (p. 51) represents a situation where you need to execute the code in the **update** method two or more times in succession during each iteration of the game loop with the total value of delta being divided into smaller values.

4.4.3.2 Calls to the render method

The situation regarding calls to the **render** method, as shown in Listing 4 (p. 52) , is much less complicated.

Listing 4 . The remainder of the updateAndRender method of the GameContainer class.

```
    if (hasFocus() || getAlwaysRender()) {
      if (clearEachFrame) {
        GL.glClear(SGL.GL_COLOR_BUFFER_BIT |
                             SGL.GL_DEPTH_BUFFER_BIT);
      }//end if

      GL.glLoadIdentity();

      graphics.resetFont();
      graphics.resetLineWidth();
      graphics.setAntiAlias(false);
      try {
        game.render(this, graphics);
      } catch (Throwable e) {
        Log.error(e);
        throw new SlickException(
          "Game.render() failure - check the game code.");
      }//end catch
      graphics.resetTransform();

      if (showFPS) {
        defaultFont.drawString(10,10,"FPS: "+recordedFPS);
      }//end if

      GL.flush();
    }//end if on hasFocus

    if (targetFPS != -1) {
      Display.sync(targetFPS);
    }//end if
  }//end method updateAndRender
```

Table 4.4

One call per iteration of the game loop

Although there is some tedious housekeeping code in Listing 4 (p. 52) , one call to the **render** method is made during each iteration of the game loop provided that the game window has the focus or a property named **alwaysRender** is true.

(The default value for **alwaysRender** *is false, but a public method is provided to set its value to true or false.)*

4.4.4 Overall structure of a game program

Although the Slick2D library can be used in a variety of ways to create game programs, the overall structure for **one approach** looks something like the following.

- Define a class with a **main** method.
- Cause the **main** method to instantiate an object of the **BasicGame** class.
- Cause the **main** method to instantiate an object of the **AppGameContainer** class, passing the **BasicGame** object's reference as a parameter to the constructor for **AppGameContainer** .
- Cause the **main** method to call the **start** method on the **AppGameContainer** object.
- Override the **init** method inherited from the **Basic** game class to initialize the state of your game. This method will be called once by default before the game loop begins.
- Override the **update** method to update the state of your game during each iteration of the game loop. Use the incoming value of delta for timing control. The **update** method will be called none, one, or more times during each iteration of the game loop as described earlier.
- Override the **render** method to draw the state of your game in the game window once during each iteration of the game loop.
- Optionally override the inherited **getTitle** and **closeRequested** methods if needed.
- Using the Slick2D javadocs [2] and the Java javadocs [3] *(or a later version)* as a guide, write code into your constructor, your **main** method, and your overridden methods to tailor the behavior of your game program to your liking.

4.5 Discussion and sample code

By now, you should have a pretty good understanding of the basics of writing a game program using Slick2D using the approach described above (p. 53) .

Previous modules have presented skeleton code for writing such a program. In this module, I will present and discuss a program that has a little more meat on that skeleton's bones to illustrate a few more concepts. Future modules will dig much more deeply into the capabilities provided by the Slick2D library.

4.5.1 The program named Slick0130a

Listing 10 (p. 62) provides a complete listing for the program named **Slick0130a** . I will explain the differences between this program and the skeleton programs presented in earlier modules. Before getting into the code details, however, I will show you the output produced by the program.

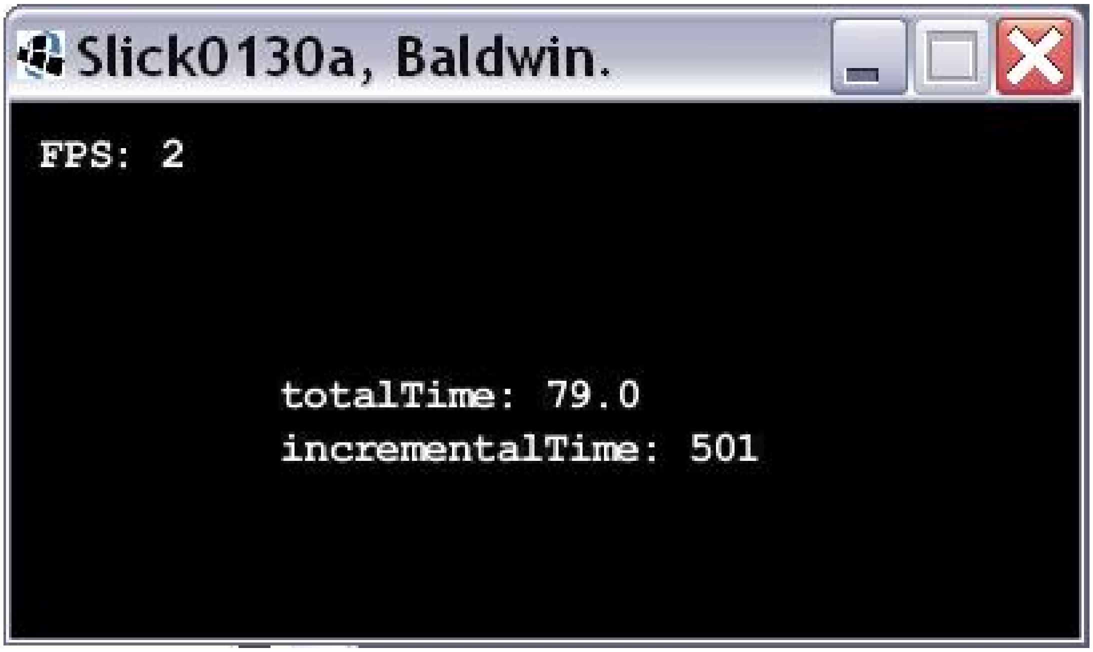
4.5.1.1 The screen output

Figure 1 (p. 55) shows a screen shot of the output in the game window while the program is running.

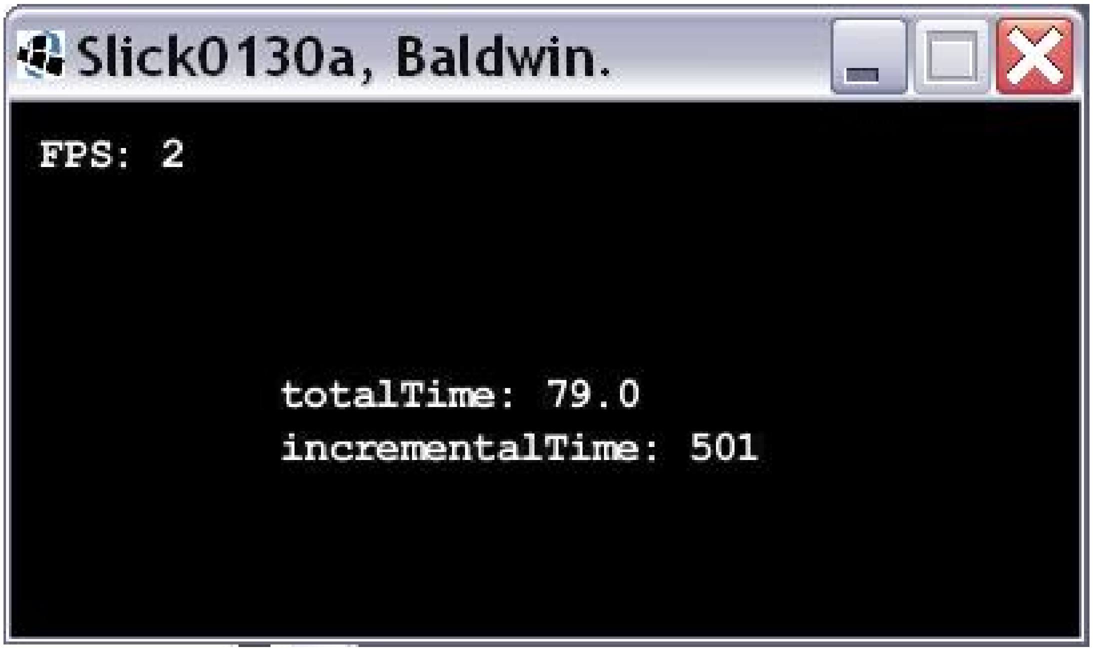
---

[2] http://slick.ninjacave.com/javadoc/

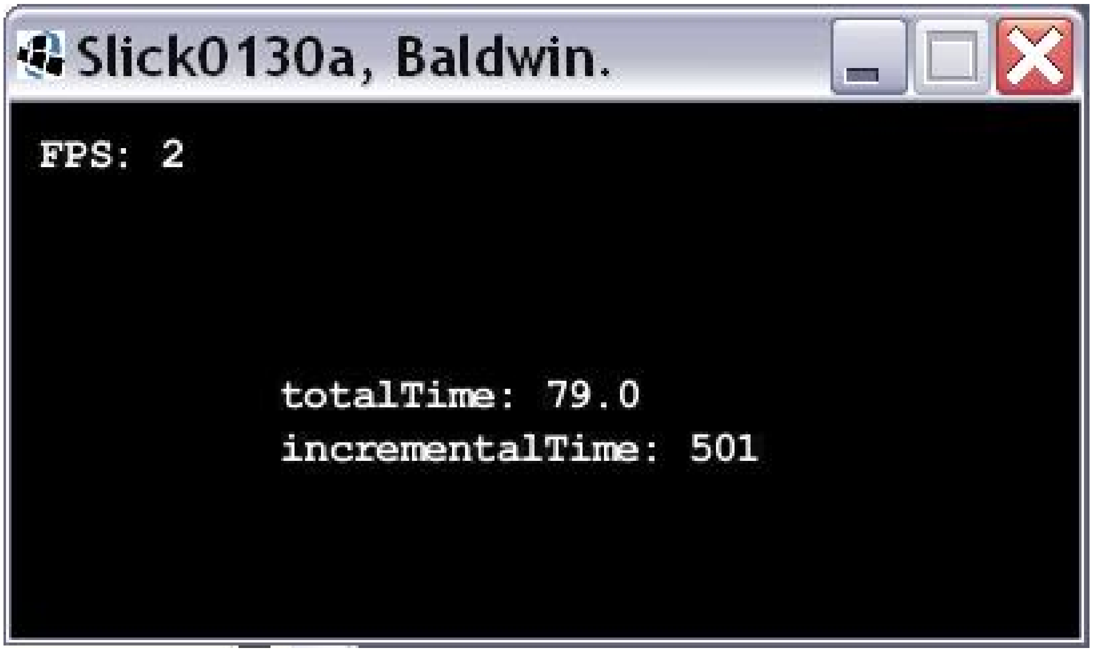
[3] http://docs.oracle.com/javase/7/docs/api/index.html



```
FPS: 2

totalTime: 79.0
incrementalTime: 501
```

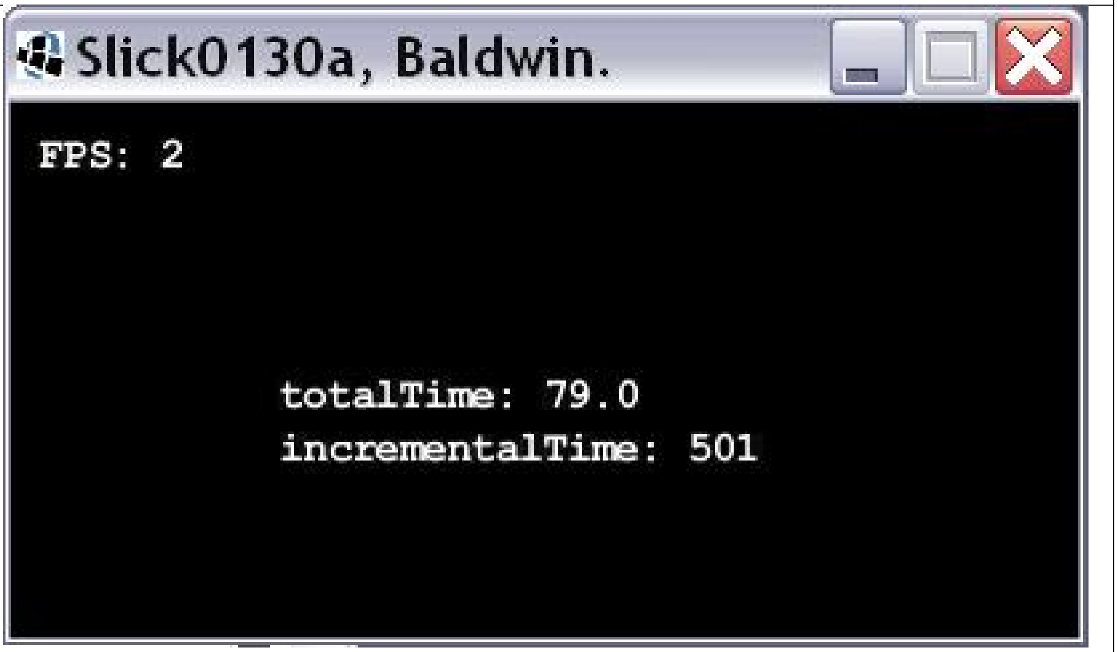


```
FPS: 2

totalTime: 79.0
incrementalTime: 501
```



```
FPS: 2

totalTime: 79.0
incrementalTime: 501
```

Figure 1 . Screen output from program named Slick0130a.



```
FPS: 2

totalTime: 79.0
incrementalTime: 501
```

Table 4.5

The frame rate

The text in the upper-left corner of Figure 1 (p. 55) is the rate in frames per second that the game loop is running. *(Two frames per second in this case.)* This value is placed there by default. The

GameContainer class provides a public method named **setShowFPS** that you can call whenever you have access to the **AppGameContainer** object to disable or enable the display of this information.

The reported value for FPS is always an integer. On my computer, it bounces back and forth between 2 and 3 frames per second when this program is running.

The total elapsed time

The first line of text near the center of Figure 1 (p. 55) shows the computed value of the total elapsed time in seconds since the game loop started running. As you will see later, this value is computed by accumulating successive values of the incoming **delta** parameter in the **update** method.

If you compile and run this program, you should see this value counting up in one-half second increments, which is consistent with a frame rate of two frames per second.

The value of delta

The second line of text near the center of Figure 1 (p. 55) shows the value of delta received by the most recent call to the **update** method. On my computer, this value seems to range between 499 and 501 milliseconds, which is consistent with a frame rate of two frames per second.

4.5.1.2 Beginning of the class named Slick0130a

Listing 5 (p. 56) shows the beginning of the class named **Slick0130a** including the declaration of some instance variables and the constructor.

Listing 5 . Beginning of the class named Slick0130a.

```
public class Slick0130a extends BasicGame{

  //Instance variables for use in computing and
  // displaying total time since program start and
  // time for each frame.
  double totalTime = 0;
  int incrementalTime = 0;

  public Slick0130a(){
    //Call to superclass constructor is required.
    super("Slick0130a, Baldwin.");
  }//end constructor
```

Table 4.6

The instance variables that are declared in Listing 5 (p. 56) are used to compute and display the values shown near the center of Figure 1 (p. 55) .

4.5.1.3 The main method

Listing 6 (p. 57) shows the **main** method for the program named **Slick0130a** .

Listing 6 . The main method.

```
public static void main(String[] args)
                                  throws SlickException{
  try{
    AppGameContainer app = (
                     new AppGameContainer(
                       new Slick0130a(),400,200,false));
    app.start();
  }catch(SlickException e){
    e.printStackTrace();
  }//end catch
}//end main
```

Table 4.7

A different constructor

Listing 6 (p. 57) calls a different overloaded constructor for the **AppGameContainer** class than I have used in earlier modules.

This version of the constructor allows for setting the width and height of the game window. In this case, the game window is set to a width of 400 pixels and a height of 200 pixels as shown in Figure 1 (p. 55) .

The last parameter to this constructor is described as a **boolean** parameter that allows for the selection of a full-screen game window. As of this writing, I have been unable to get this to work. When I set the third parameter to true, I get a compiler error. However, I haven't spent any time investigating what I might be doing wrong.

4.5.1.4 The overridden init method

The overridden **init** method is shown in Listing 7 (p. 57) .

Listing 7 . The overridden init method.

```
public void init(GameContainer gc)
                                 throws SlickException {
  //Set the frame rate in frames per second.
  gc.setTargetFrameRate(2);

}//end init
```

Table 4.8

Set the target frame rate

Listing 7 (p. 57) calls the **setTargetFrameRate** on the **GameContainer** object passing 2 as a parameter. The description of this method in the javadocs [4] is *"Set the target fps we're hoping to get,"*

---

[4] http://slick.ninjacave.com/javadoc/org/newdawn/slick/GameContainer.html#setTargetFrameRate%28int%29

4.5.1.5 The overridden update method

Listing 8 (p. 58) shows the overridden update method.

Listing 8 . The overridden update method.

```
public void update(GameContainer gc, int delta)
                                  throws SlickException{
  //Compute and save total time since start in seconds.
  totalTime += delta/1000.0;

  //Save delta for display in render method.
  incrementalTime = delta;

}//end update
```

Table 4.9

Compute total elapsed time

Listing 8 (p. 58) converts the incoming value of delta in milliseconds into seconds and adds it to the value stored in the instance variable named **totalTime** that is declared in Listing 5 (p. 56) . The **totalTime** value will be used to display the first line of text near the center of Figure 1 (p. 55) .

Save the value of delta

Listing 8 (p. 58) saves the incoming value of delta in the instance variable named i **ncrementalTime** that was also declared in Listing 5 (p. 56) . The value stored in **incrementalTime** will be used to display the second line of text near the center of Figure 1 (p. 55) .

4.5.1.6 The overridden render method

Listing 9 (p. 59) shows the overridden **render** method.

Listing 9 . The overridden render method.

```
public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
  //Truncate totalTime to one decimal digit and
  // display
  double time = (int)(totalTime*10)/10.0;
  g.drawString("totalTime: "+time,100.0f,100.0f);

  //Display incremental time.
  g.drawString("incrementalTime: " + incrementalTime,
             100.0f,120.0f);
}//end render
```

Table 4.10

Truncate and draw the total time

Listing 9 (p. 59) begins by truncating the value of **totalTime** to only two decimal digits and saving the truncated value in a local variable named **time** . I will leave it as an exercise for the student to analyze the code that I used to do that.

Then Listing 9 (p. 59) calls the **drawString** method on the graphics context received as an incoming parameter of type **Graphics** to display the value in the first line of text near the center of Figure 1 (p. 55) .

The **drawString** method takes three parameters. The first is the string that is to be drawn in the game window and the next two are the horizontal and vertical coordinates for the location in which the string is to be drawn.

Draw the saved value of delta

Then Listing 9 (p. 59) calls the **drawString** method again to draw the saved value of delta from the most recent call to the **update** method as the second line of text near the center of Figure 1 (p. 55) .

4.5.1.7 End of discussion

That concludes the discussion of the program named **Slick0130a** . Although this is a simple program, it should provide a little more insight into one approach (p. 53) to creating a game program using the Slick2D library.

4.6 Run the program

I encourage you to copy the code from Listing 10 (p. 62) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

4.7 Summary

You learned about a property of the **GameContainer** class named **running** , and how it is used by the **start** method to keep the game loop running.

You learned about the salient features of the **gameLoop** method of the **AppGameContainer** class.

You learned about the **updateAndRender** method of the **GameContainer** class and how it decides when and if to call the **update** and **render** methods of the object of the **Game** class that is wrapped by the container.

You touched on the difference between normal delta and smoothed delta.

You learned about **minimumLogicInterval** and **maximumLogicInterval** and how the contents of those two variables are used to determine if, when, and how many times to call the **update** method during each iteration of the game loop. You also learned how the contents of these two variables are used to determine the value that is passed as delta each time the update method is called.

You learned that the **render** method is normally called once and only once during each iteration of the game loop.

You saw a simple example of how you can use the value of delta that is received by the **update** method to control the behavior of a game program.

You learned that you can set the size of the game window when you instantiate an object of the **AppGameContainer** class by passing dimension parameters to the constructor.

You learned that you can set the target frame rate by calling the **setTargetFrameRate** method on the **GameContainer** object.

You learned how to display text in the game window.

4.8 What's next?

In the next module, we will take a look at displaying images with transparency.

4.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0130: The game loop
- File: Slick0130.htm
- Published: 02/04/13
- Revised: 06/09/15 for 64-bit

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

4.10 Complete program listing

Listing 10 (p. 62) provides a complete listing of the program named **Slick0130a** .

Listing 10 . Source code for the program named Slick0130a.

```
/*Slick0130a.java
Copyright 2012, R.G.Baldwin

A very skinny Slick program. Barely more than a skeleton.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.SlickException;

public class Slick0130a extends BasicGame{

  //Instance variables for use in computing and
  // displaying total time since program start and
  // time for each frame.
  double totalTime = 0;
  int incrementalTime = 0;

  public Slick0130a(){
    //Call to superclass constructor is required.
    super("Slick0130a, Baldwin.");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    try{
      AppGameContainer app = (
                      new AppGameContainer(
                        new Slick0130a(),400,200,false));
      app.start();
    }catch(SlickException e){
      e.printStackTrace();
    }//end catch
  }//end main
  //----------------------------------------------------//

  @Override
  public void init(GameContainer gc)
                                   throws SlickException {
    //Set the frame rate in frames per second.
    gc.setTargetFrameRate(2);

  }//end init
  //----------------------------------------------------//

  @Override
  public void update(GameContainer gc, int delta)
                                    throws SlickException{
    //Compute and save total time since start in seconds.
```

Table 4.11

-end-

Chapter 5

Slick0140: A first look at Slick2D bitmap graphics[1]

5.1 Table of Contents

5.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is to take a first look at bitmap graphics in Slick2D.

---

[1] This content is available online at <http://cnx.org/content/m45737/1.4/>.

5.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

5.2.1.1 Figures

- Figure 1 (p. 71) . Background image in Windows Paint.
- Figure 2 (p. 75) . Ladybug image in Windows Paint.
- Figure 3 (p. 77) . Ladybug image in Windows Picture and Fax Viewer.
- Figure 4 (p. 79) . Ladybug image in Gimp.
- Figure 5 (p. 83) . Output from program Slick0140a.

5.2.1.2 Listings

- Listing 1 (p. 84) . Beginning of the class named Slick0140a.
- Listing 2 (p. 85) . The main method.
- Listing 3 (p. 85) . The overridden init method.
- Listing 4 (p. 86) . The overridden render method.
- Listing 5 (p. 89) . Source code for the program named Slick0140a.

5.3 Preview

What you have learned

In the previous module, you learned about a property of the **GameContainer** class named **running** , and how it is used by the **start** method to keep the game loop running.

You learned about the salient features of the **gameLoop** method of the **AppGameContainer** class.

You learned about the **updateAndRender** method of the **GameContainer** class and how it decides when and if to call the **update** and **render** methods of the object of the **Game** class that is wrapped by the container.

You touched on the difference between normal **delta** and smoothed **delta** .

You learned about **minimumLogicInterval** and **maximumLogicInterval** and how the contents of those two variables are used to determine if, when, and how many times to call the **update** method during each iteration of the game loop. You also learned how the contents of these two variables are used to determine the value that is passed as **delta** each time the **update** method is called.

You learned that the **render** method is normally called once and only once during each iteration of the game loop.

You saw a simple example of how you can use the value of **delta** that is received by the **update** method to control the behavior of a game program.

You learned that you can set the size of the game window when you instantiate an object of the **AppGameContainer** class by passing dimension parameters to the constructor.

You learned that you can set the target frame rate by calling the **setTargetFrameRate** method on the **GameContainer** object.

You learned how to display text in the game window.

What you will learn

In this module, you will learn that while there are many classes, interfaces, and methods in the Slick2D library with names that match classes, interfaces, and methods in the standard edition Java library, they are not the same.

You will learn how to set the drawing mode so that bitmap images drawn in the game window will either honor or not honor transparent pixels.

You will learn how to draw bitmap images in the game window using both the **draw** methods of the **Image** class and the **drawImage** methods of the **Graphics** class.

5.4 General background information

Bitmaps and shapes

Many game programs communicate primarily with the player using interactive graphics. Sometimes those graphics involve drawing bitmap images. Both the Slick2D **Image** class and the Slick2D **Graphics** class provide methods for drawing bitmap images.

Sometimes game programs involve drawing shapes such as circles, rectangles, polygons, arcs, etc. The Slick2D **Graphics** class and other classes such as the Slick2D **Shape** class provide methods for creating and drawing such shapes and filling closed shapes with color.

And of course, some game programs involve a combination of the two.

This module concentrates on drawing bitmap images both honoring and not honoring transparent pixels.

Common class names

The Slick2D library contains numerous classes, interfaces, and methods with names that match the names in the Java standard edition library, such as **Color** , **Graphics** , **Image** , **Line** , **Rectangle** , **Shape** , **Transform** , **TextField** , etc.

You need to be aware, however, that even though the names are the same, and the behaviors may be similar, these are not standard Java classes. Their behaviors will often be different from standard Java classes. Therefore, ready access to the documentation at http://slick.ninjacave.com/javadoc/ [2] while you are programming in Slick2D is very important even if you are a seasoned Java programmer.

Illustrate major differences

The program that I will present in this module will illustrate some of the major differences between the two libraries insofar as graphics programming is concerned. For example, both libraries have a class named **Image** , which has generally the same purpose in both libraries. However, the **Image** class in the Slick2D library provides about ten different overloaded **draw** methods that you can call to draw images in the game window.

The **Image** class in the Java standard library doesn't have any draw methods. Instead, it is necessary to get a graphics context on the image and then call the **drawImage** method on that context to draw the image.

Draw bitmap images two different ways

The Slick2D **render** method receives an incoming parameter of type **Graphics** , which also provides a **drawImage** method that can be used to draw an image in the game window.

I will show you how to draw bitmap images using the **draw** methods of the **Image** class and also show you how to draw bitmap images using the **drawImage** methods of the **Graphics** class.

5.5 Discussion and sample code

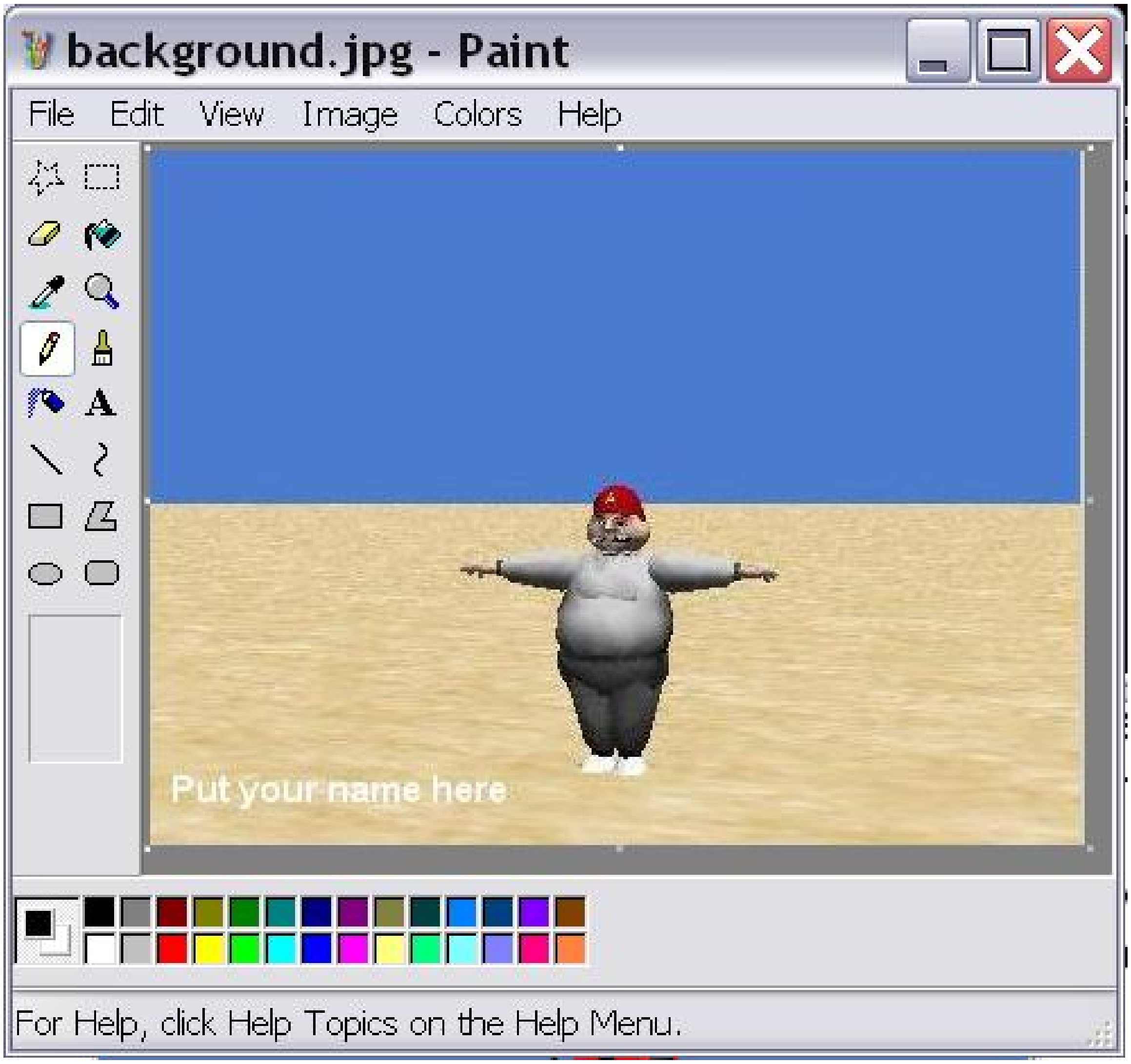
Listing 5 (p. 89) provides a complete listing of a Slick2D program named **Slick0140a** . Before getting into the details of the code, however, I will show you the input and output images.

Input images

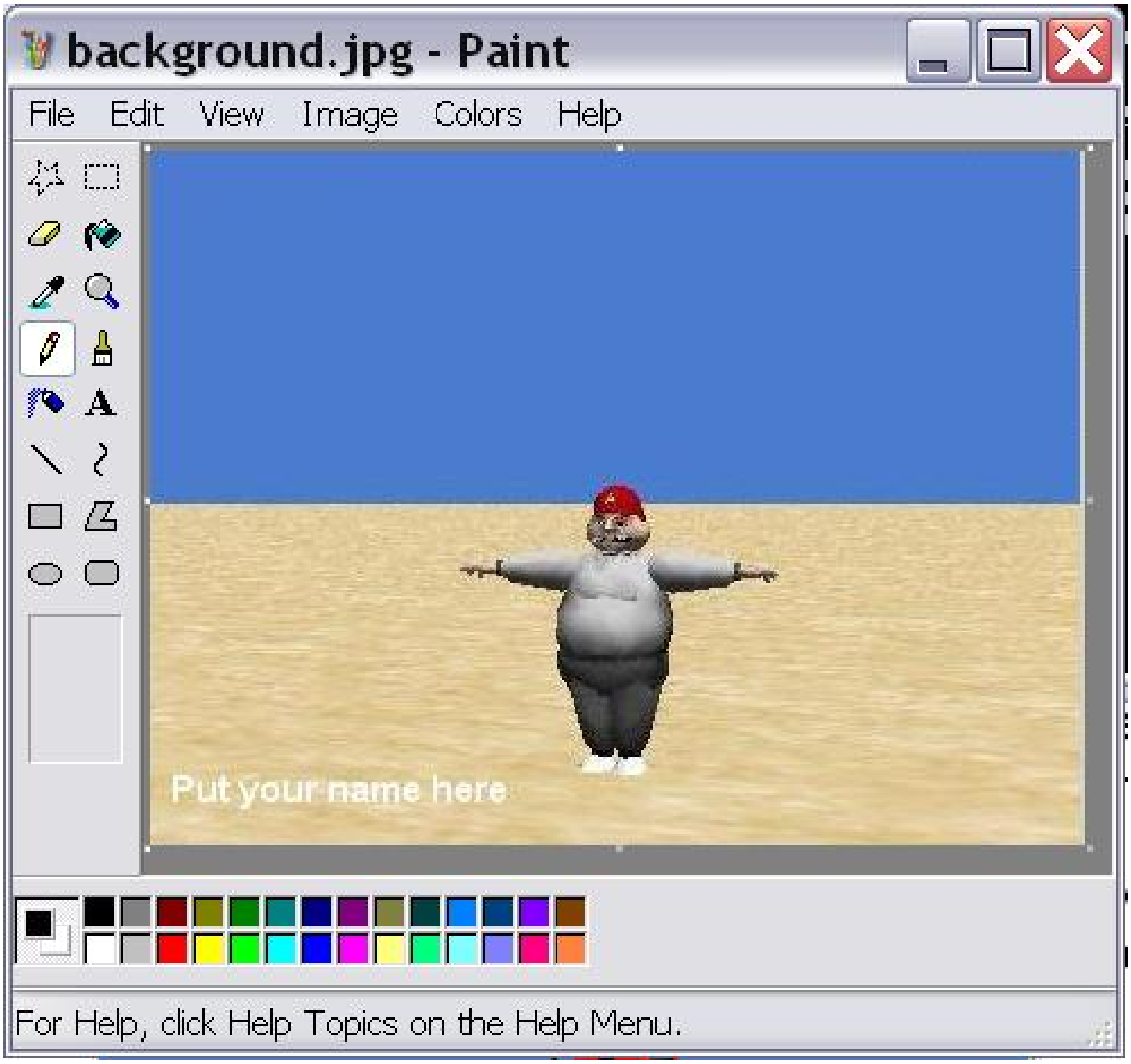
This program uses two input images. An image file named **background.jpg** is used to create a background in the game window. Figure 1 (p. 71) shows what that image looks like when viewed in the *Windows Paint* program.

---

[2] http://slick.ninjacave.com/javadoc/



For Help, click Help Topics on the Help Menu.


background.jpg - Paint
File
Edit
View
Image
Colors
Help
Put your name here
For Help, click Help Topics on the Help Menu.

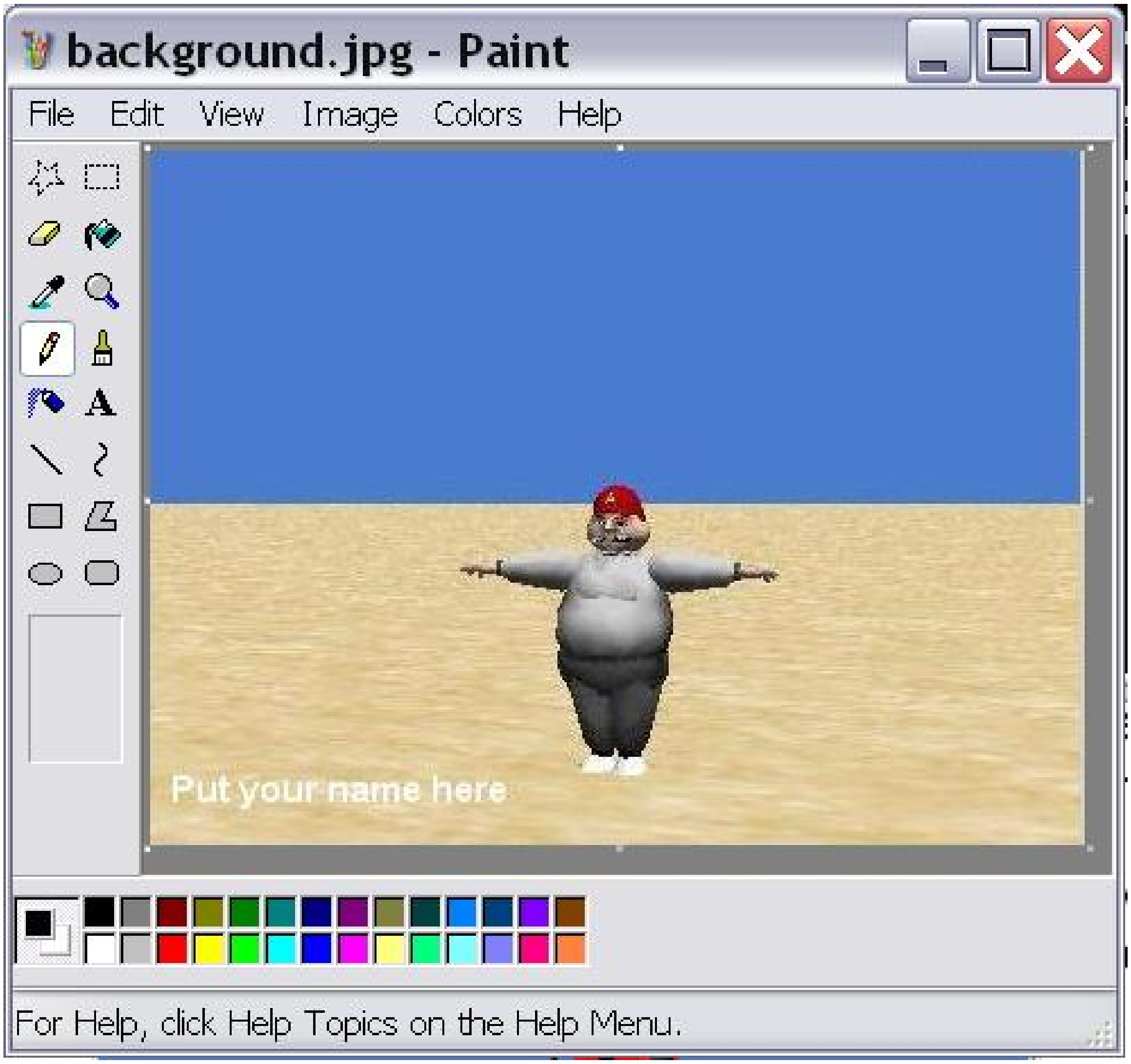
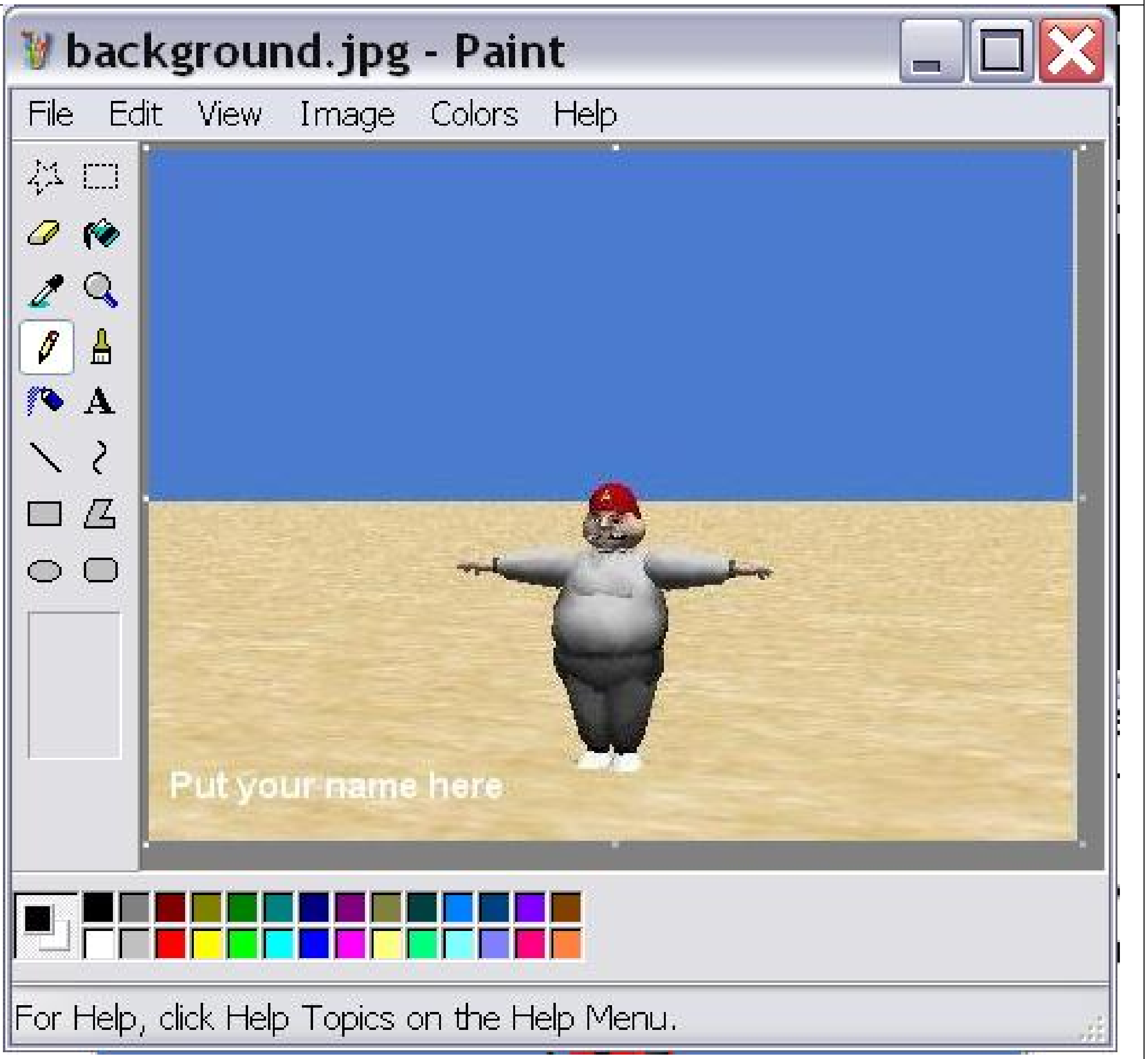

background.jpg - Paint
File
Edit
View
Image
Colors
Help
Put your name here
For Help, click Help Topics on the Help Menu.

Figure 1 . Background image in Windows Paint.



For Help, click Help Topics on the Help Menu.

Table 5.1

The ladybug image

The second input image is a file named **ladybug.png** , which is a picture of a red and black ladybug on a transparent black background. Figure 2 (p. 75) shows this image when viewed in the *Windows Paint* program.


ladybug.png...
File
Edit
View
Image
Colors
Help
For Help, click Help Topics on the Help


ladybug.png...
File
Edit
View
Image
Colors
Help
A
For Help, click Help Topics on the Help


ladybug.png...
File
Edit
View
Image
Colors
Help
A
For Help, click Help Topics on the Help

Figure 2 . Ladybug image in Windows Paint.

Table 5.2

Figure 3 (p. 77) shows the same ladybug image when viewed in the *Windows Picture and Fax Viewer* program. Note that the black background from Figure 2 (p. 75) is transparent in Figure 3 (p. 77) .


la...


la...

Figure 3 . Ladybug image in Windows Picture and Fax Viewer.


Table 5.3

Figure 4 (p. 79) shows the same ladybug image when viewed in the *Gimp* image editor program. This program provides even a different treatment for the transparent pixels.


File
Edit
Select
Vie
0
50
px

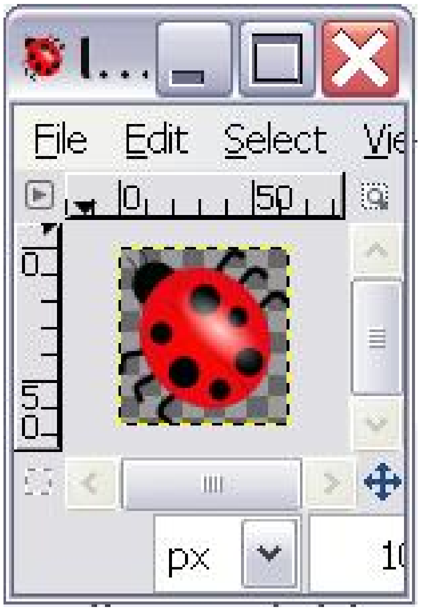

File
Edit
Select
Vie
0
50
px

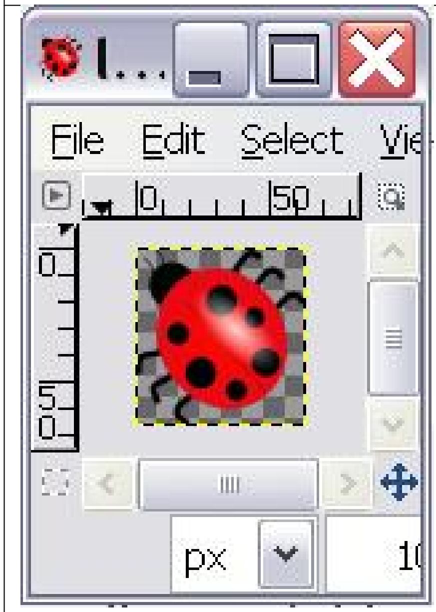
Figure 4 . Ladybug image in Gimp.


Table 5.4

The output image

Figure 5 (p. 83) shows a screen shot of the game window while the program is running


Slick0140a, baldwin
Put your name here


Slick0140a, baldwin
Put your name here


Slick0140a, baldwin
Put your name here

Figure 5 . Output from program Slick0140a.

Table 5.5

Different drawing parameters

The same ladybug image is drawn three times in Figure 5 (p. 83) with different drawing parameters.

The leftmost image of the ladybug in Figure 5 (p. 83) is drawn with a scale factor of 0.75 and a drawing mode that honors transparency: MODE_NORMAL.

The center image of the ladybug in Figure 5 (p. 83) is drawn using a different approach with a scale factor of 1.0 and a drawing mode that honors transparency: MODE_NORMAL.

The rightmost image of the ladybug in Figure 4 (p. 79) is drawn using the same approach as the leftmost image, a scale factor of 1.25, and a drawing mode that does not honor transparency: MODE_ALPHA_BLEND.

Are the mode names reversed?

As mentioned above, the two images of the ladybug with transparency were drawn using a Slick2D constant named **MODE_NORMAL** .

The image of the ladybug on the right without transparency was drawn using a Slick2D constant named **MODE_ALPHA_BLEND** .

These names seem to have been reversed. I would expect the constant with a name that includes the words *alpha* and *blend* to honor transparency but that doesn't seem to be the case.

5.5.1 Beginning of the class named Slick0140a

Listing 1 (p. 84) shows the beginning of the class named **Slick0140a** including some instance variable declarations and the constructor.

Listing 1 . Beginning of the class named Slick0140a.

```
  public class Slick0140a extends BasicGame{

Image ladybug = null;
Image background = null;

float leftX = 100;//leftmost position of ladybug
float leftY = 100;

float middleX = 200;//middle position of ladybug
float middleY = 50;

float rightX = 300;//rightmost position of ladybug
float rightY = 100;


float leftScale = 0.75f;//drawing scale factors
float rightScale = 1.25f;
//----------------------------------------------------//

public Slick0140a(){//constructor
  //Set the title
  super("Slick0140a, baldwin");
}//end constructor
```

Table 5.6

The instance variables shown in Listing 1 (p. 84) and the values that they contain will be used later to display the three ladybug images shown in Figure 5 (p. 83) .

There is nothing new in the constructor in Listing 1 (p. 84) .

5.5.2 The main method

There is also nothing new in the **main** method in Listing 2 (p. 85) .

Listing 2 . The main method.

continued on next page

```
    public static void main(String[] args)
                                  throws SlickException{
   AppGameContainer app = new AppGameContainer(
                        new Slick0140a(),414,307,false);
   app.start();
 }//end main
```

Table 5.7

5.5.3 The overridden init method

The overridden **init** method is shown in Listing 3 (p. 85) . There is quite a bit of new material in Listing 3 (p. 85) .

Listing 3 . The overridden init method.

```
    public void init(GameContainer gc)
                                 throws SlickException {
   ladybug = new Image("ladybug.png");
   background = new Image("background.jpg");

   gc.setShowFPS(false);//disable FPS display

   gc.setTargetFrameRate(60);//set frame rate
 }//end init
```

Table 5.8

Two new Image objects

Listing 3 (p. 85) begins by instantiating two new Slick2D **Image** objects from the image files discussed earlier and saving those object's references in two of the instance variables that were declared in Listing 1 (p. 84) .

(Note that in this case, the image files were located in the same folder as the source code for the program. Therefore, a path specification to the image files was not needed.)

I will remind you again that the Slick2D **Image** class is different from the **Image** class in the standard edition Java library.

Don't display FPS

You may have noticed that the FPS display is missing from the upper-left corner of Figure 5 (p. 83) . That is because it was disabled by the call to the **setShowFPS** method in Listing 3 (p. 85) , passing false as a parameter to the method.

Set the target frame rate

The last statement in Listing 3 (p. 85) sets the target frame rate to 60 frames per second.

5.5.4 An empty update method

The **update** method in Listing 5 (p. 89) is empty so there is no point in showing it here.

5.5.5 The overridden render method

The overridden **render** method is shown in Listing 4 (p. 86) .

Listing 4 . The overridden render method.

```
   public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
  //Note that the names of the drawMode constants seem
  // to be backwards.

  //Draw the background and two versions of the
  // ladybug by calling a draw method of the Image
  // class.
  g.setDrawMode(g.MODE_NORMAL);//honors transparency
  background.draw(0,0);
  ladybug.draw(leftX,leftY,leftScale);

  g.setDrawMode(g.MODE_ALPHA_BLEND);//no transparency
  ladybug.draw(rightX,rightY,rightScale);

  //Draw a third version of the ladybug by calling
  // a drawImage method of the Graphics class.
  g.setDrawMode(g.MODE_NORMAL);
  g.drawImage(ladybug,middleX,middleY);
}//end render
```

Table 5.9

Draw the background and the leftmost ladybug

Listing 4 (p. 86) begins by calling the **setDrawMode** method on the incoming **Graphics** parameter to set the drawing mode to MODE_NORMAL as described earlier. Then it calls one of the overloaded **draw** methods of the background **Image** object and the ladybug **Image** object to draw the background and the leftmost ladybug in Figure 5 (p. 83) .

Note that the drawing coordinates and the scale factor are passed to the **draw** method.

Also note that this drawing of the ladybug image honors transparent pixels.

Draw the rightmost ladybug

After that, Listing 4 (p. 86) calls the **setDrawMode** method on the incoming **Graphics** parameter to set the drawing mode to MODE_ALPHA_BLEND and calls the same **draw** method of the ladybug **Image** object to draw the rightmost ladybug in Figure 5 (p. 83) .

Note that this drawing of the ladybug image does not honor transparent pixels.

Call the drawImage method of the Graphics class

Finally, Listing 4 (p. 86) calls the **setDrawMode** method on the incoming **Graphics** parameter to set the drawing mode back to MODE_NORMAL and then calls the **drawImage** method on the incoming **Graphics** parameter to draw the middle ladybug in Figure 5 (p. 83) . *(This approach is similar to the approach that would be used to draw an image using the standard edition Java library.)*

Note that the reference to the ladybug **Image** object and the drawing coordinates are passed as parameters to the **drawImage** method. Some of the overloaded **drawImage** methods provide scaling. However, there is no scale parameter for this version of the **drawImage** method so the ladybug was drawn at actual size.

Many overloaded drawing methods

There are many overloaded versions of the **draw** and the **drawImage** methods.

That completes the discussion of the program named **Slick0140a** .

5.6 Run the program

I encourage you to copy the code from Listing 5 (p. 89) Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

5.7 Summary

You learned that while there are many classes, interfaces, and methods in the Slick2D library with names that match classes, interfaces, and methods in the standard edition Java library, they are not the same.

You learned that you can access the Slick2D documentation at http://slick.ninjacave.com/javadoc/ [3] . *(A copy of the documentation is also included in the distribution zip file.)*

You learned how to set the drawing mode so that bitmap images drawn in the game window will either honor or not honor transparent pixels.

You learned how to draw bitmap images in the game window using both the **draw** methods of the **Image** class and the **drawImage** methods of the **Graphics** class.

5.8 What's next?

In the next module, you will learn how to make sprites move at a constant speed in front of an image in the face of a widely varying frame rate. You will also learn about a rudimentary form of collision detection.

5.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0140: A first look at Slick2D bitmap graphics
- File: Slick0140.htm
- Published: 02/04/13
- Revised: 06/09/15 for 64-bit

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such

---

[3] http://slick.ninjacave.com/javadoc/

a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

5.10 Complete program listing

Listing 5 (p. 89) contains a complete listing of the program named **Slick0140a** .

Listing 5 . Source code for the program named Slick0140a.

```
/*Slick0140a.java
Copyright 2012, R.G.Baldwin

Illustrates drawing a sprite image with transparent
parts on a background image using two different
approaches.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;

public class Slick0140a extends BasicGame{

  Image ladybug = null;
  Image background = null;
  float leftX = 100;//leftmost position of ladybug
  float leftY = 100;

  float middleX = 200;//middle position of ladybug
  float middleY = 50;

  float rightX = 300;//rightmost position of ladybug
  float rightY = 100;


  float leftScale = 0.75f;//drawing scale factors
  float rightScale = 1.25f;
  //----------------------------------------------------//

  public Slick0140a(){//constructor
    //Set the title
    super("Slick0140a, baldwin");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    AppGameContainer app = new AppGameContainer(
                         new Slick0140a(),414,307,false);
    app.start();
  }//end main
  //----------------------------------------------------//

  @Override
  public void init(GameContainer gc)
                                   throws SlickException {
    ladybug = new Image("ladybug.png");
    background = new Image("background.jpg");
```

Table 5.10

-end-

Chapter 6

Slick0150: A first look at sprite motion, collision detection, and timing control[1]

6.1 Table of Contents

6.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

---

[1] This content is available online at <http://cnx.org/content/m45738/1.4/>.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is teach you how to make sprites move at a constant speed in front of an image in the face of a widely varying frame rate. You will also learn about a rudimentary form of collision detection.

6.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the Figures and Listings while you are reading about them.

6.2.1.1 Figures

6.2.1.2 Listings

6.3 Preview

What you have learned

In the previous module, you learned that while there are many classes and interfaces in the Slick2D library with names that match the names of classes and interfaces in the standard edition Java library, they are not the same.

You learned that you can access the Slick2D documentation at http://slick.ninjacave.com/javadoc/ [2] . *(A copy of the documentation is also included in the distribution zip file.)*

You learned how to set the drawing mode so that bitmap images drawn in the game window will either honor or not honor transparent pixels.

You learned how to draw bitmap images in the game window using both the **draw** methods of the **Image** class and the **drawImage** methods of the **Graphics** class.

What you will learn

In this module, you will learn how to make sprites move at a constant speed in front of an image in the face of widely varying frame rates. You will also learn about a rudimentary form of collision detection.

---

[2] http://slick.ninjacave.com/javadoc/

6.4 General background information

The update and render methods

Following initialization, the Slick2D game engine switches back and forth between an **update** method and a **render** method.

We write code to control the state of the game in the **update** method. We write code to display the state of the game in the **render** method.

A sprite

According to one definition, a sprite is *a computer graphic that may be moved on-screen and otherwise manipulated as a single entity.*

We will use the image of the ladybug shown in Figure 1 (p. 94) as a sprite in two different programs that I will explain in this module.

Figure 1 . Image of a ladybug.


Table 6.1

We will cause that sprite to move in front of the background image shown in Figure 2 (p. 98) .


Put your name here


Put your name here


Put your name here

Figure 2 . Background image.

Table 6.2

A bouncing sprite

In particular, we will cause the sprite to bounce around inside the game window like a pool ball on a pool table. Whenever it strikes an edge of the game window, it will bounce off in the opposite direction. This process will continue until the program is terminated.

The target frame rate

As you learned in an earlier module, a **GameContainer** method named **setTargetFrameRate** can be called in an attempt to cause the program to run at a constant frame rate.

This method can slow the frame rate down to the target value on fast computers. However, it cannot speed the frame rate up to the target value on slower computers.

Therefore, a call to the **setTargetFrameRate** method should actually be viewed as setting the maximum frame rate that the program will run.

An appearance of achieving the target frame rate

Sometimes it is important to cause the program to give the appearance of running at the target frame rate even if it is actually running slower.

One example is when a sprite is moving across the game window. It is often desirable to cause the sprite to move at the same overall speed regardless of the speed of the computer. For example, you probably wouldn't want a missile to take a long time to reach its target on a slow computer and a short time to reach its target on a fast computer.

Accuracy versus graphic quality

I will explain a program in this module that is designed to achieve such a result. The upside is that you can often achieve the appearance of the target frame rate in terms of the overall speed of the sprite. The downside is that the motion of the sprite may be less smooth than would be the case if the computer were actually running at the target frame rate.

The parameter named delta

Each time the **update** method is called, it receives an incoming parameter named delta whose value is the number of milliseconds that have elapsed since the most recent call to the **update** method. In the case of a highly varying frame rate, such as may occur when the **render** method is required to draw a complex and constantly changing scene, the value of delta may vary significantly from one call to the next of the **update** method.

Fortunately, the value of delta can often be used to give the appearance of running at the target frame rate even though the actual frame rate may be below the target. I will show you how to do that in this module.

6.5 Discussion and sample code

6.5.1 A program with a relatively constant frame rate - Slick0150a

I will begin by discussing a case with a relatively constant frame rate. The program for this case, **Slick0150a** , is shown in Listing 9 (p. 121) .

6.5.1.1 The screen output for Slick0150a

Before getting into the coding details, I will show you some output. Figure 3 (p. 103) shows a screen shot of the game window while the program is running.

Slick0150a, baldwin
FPS: 62 traversalTime: 3015
Put your name here

Slick0150a, baldwin

```
FPS: 62   traversalTime: 3015
```

Put your name here


Slick0150a, baldwin
FPS: 62 traversalTime: 3015
Put your name here

Figure 3 . Screen shot of the program named Slick0150a.

FPS: 62 traversalTime: 3015

Table 6.3

The screen shot in Figure 3 (p. 103) caught the ladybug in mid flight. As mentioned earlier, the next time it collides with one of the edges of the game window, it will bounce off and move in the opposite direction like a pool ball striking the cushion on the edge of a pool table.

The FPS output

As you learned in an earlier module, the text in the upper-left corner is the frame rate in frames per second computed and automatically displayed by the game engine. You will see later that a target frame rate of 60 frames per second was requested by calling the method named **setTargetFrameRate** and passing 60 as a parameter.

At 60 frames per second, a time interval of 16.6666 milliseconds would be required to complete each frame. It appears that the **setTargetFrameRate** method truncates this value to an integer value of 16 milliseconds, which represents a frame rate of 62.5 frames per second. It also appears that the code that displays the frame rate converts the actual frame rate to an integer for display. Hence you see an FPS value of 62 in Figure 3 (p. 103) .

The traversalTime output

The **traversalTime** output that you see in Figure 3 (p. 103) is computed and displayed by the program that we will examine shortly. This is the time required for the sprite to complete one round trip from the right edge to the left edge and back to the right edge.

If you compile and run this program, you will see that the **traversalTime** value is reasonably stable at around 3015 milliseconds.

The theoretical traversalTime

Although it isn't shown here, a separate output on the command-line window reported the width of the background image to be 414 pixels and the width of the sprite to be 48 pixels. The sprite is never allowed to go outside the boundaries of the game window, so the one-way distance from the left edge to the right edge is 366 pixels. *(This is the distance that the upper-left corner of the sprite travels during the one-way trip.)* The round-trip distance is twice that, or 732 pixels.

You will see later that the sprite is caused to move horizontally by four pixels during each frame. At 62 frames per second, this represents a horizontal speed for the sprite of 248 pixels per second. At that speed, the sprite should complete the round trip in 2952 milliseconds. That is close enough to the typical reported time of 3015 milliseconds to validate the theoretical considerations.

Relatively smooth motion

When I compile and run this program, I see the sprite moving with a relatively smooth motion. Unless your computer is very slow, you should probably see the same thing.

6.5.1.2 Beginning of the class named Slick0150a

Listing 1 (p. 105) shows the beginning of the class definition for the class named **Slick0150a** .

Listing 1 . Beginning of the class named Slick0150a.

```
public class Slick0150a extends BasicGame{

  Image bug = null;
  Image background = null;

  float backgroundWidth;
  float backgroundHeight;

  float bugX = 100;
  float bugY = 100;
  float bugWidth;
  float bugHeight;

  float bugXDirection = 1.0f;//initial direction to right
  float bugYDirection = 1.0f;//initial direction is down

  float xStep = 4.0f;//horizontal step size
  float yStep = 3.0f;//vertical step size

  float bugScale = 0.75f;//drawing scale factor

  //Used to compute and display the time required for the
  // bug to make each round trip across the game window
  // and back.
  long oldTime = 0;
  long traversalTime = 0;

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;
```

Table 6.4

Listing 1 (p. 105) consists entirely of instance variable declarations. The purpose of each of these variables should become clear as they are used later in the code. No explanation beyond the embedded comments should be needed at this point.

6.5.1.3 The constructor and the main method

There is nothing new in the constructor and the **main** method. You can view them both in Listing 9 (p. 121) .

6.5.1.4 The init method

The **init** method is shown in Listing 2 (p. 106) . I will explain the new material in this method.

Listing 2 . The init method for Slick0150a.

```
    public void init(GameContainer gc)
                                 throws SlickException {
  oldTime = gc.getTime();

  bug = new Image("ladybug.png");
  background = new Image("background.jpg");

  backgroundWidth = background.getWidth();
  backgroundHeight = background.getHeight();

  bugWidth = bug.getWidth()*bugScale;
  bugHeight = bug.getHeight()*bugScale;

  System.out.println(
                 "backgroundWidth: " + backgroundWidth);
  System.out.println(
               "backgroundHeight: " + backgroundHeight);
  System.out.println("bugWidth: " + bugWidth);
  System.out.println("bugHeight: " + bugHeight);

  gc.setTargetFrameRate(targetFPS);//set frame rate
}//end init
```

Table 6.5

Get the time

The **GameContainer** class provides a method named **getTime** , which is described simply as:

"Get the accurate system time."

I am interpreting this to mean that the method will return the system time good to one millisecond relative to a well-defined time origin.

(Standard Java uses January 1, 1970 as the origin or epoch of time but Slick2D may use a different origin. Since we will be working with changes in time and not absolute time, the time origin doesn't matter.)

The **init** method in Listing 2 (p. 106) calls the **getTime** method to get and save the time in an instance variable named **oldTime** . The values in this variable will be used later to compute the round-trip time required for the sprite to move across the game window and back to the starting point at the right edge of the window.

Create the images

Listing 2 (p. 106) creates the ladybug **Image** object and the background **Image** object using code that you have seen before, and stores those object's references in the instance variables named **bug** and **background** .

Get, save, and display the widths and heights of the images

Then Listing 2 (p. 106) calls accessor methods to get, save, and display the widths and the heights of the **bug** and **background** objects.

Set the target frame rate

Finally, Listing 2 (p. 106) calls the **setTargetFrameRate** method to set the target frame rate to 60 frames per second.

6.5.1.5 The update method

The overridden **update** method begins in Listing 3 (p. 107) .

The code in Listing 3 (p. 107) uses a very simple approach to cause the sprite to exhibit motion.

Listing 3 . Beginning of the update method for Slick0150a.

```
    public void update(GameContainer gc, int delta)
                              throws SlickException{
  //Compute new location for the sprite.
  bugX += bugXDirection*xStep;
  bugY += bugYDirection*yStep;
```

Table 6.6

Compute new sprite locations

Each time the **update** method is called, Listing 3 (p. 107) computes new location coordinate values for the sprite, which are either increased or decreased by the values stored in **xStep** and **yStep** .

Repetitive displays of the sprite in the new locations by the **render** method produces the impression that the sprite is moving.

Step values are independent of the frame rate

The step values are multiplied by the contents of direction variables in Listing 3 (p. 107) , each of which contains either +1 or -1, and the products are added to the current location coordinates.

As you will see shortly, the algebraic signs of the direction variables are changed each time the sprite collides with an edge of the game window.

This code assumes a constant frame rate and does not correct for variations in the frame rate. In other words, the size of the step taken during each frame is the same regardless of how long it takes to complete a frame. If the computer is running below the target frame rate, the sprite will appear to move more slowly than would be the case if the computer is running at the target frame rate.

Collision detection

The code in Listing 4 (p. 108) begins by detecting a collision of the right edge of the sprite with the right edge of the game window and reverses the sprite's direction of motion when a collision occurs.

Note that if the rightmost portion of the sprite actually tries to move beyond the right edge of the game window, it is stopped at the edge of the game window.

Listing 4 . Detection of collision with right edge.

```
if(bugX+bugWidth >= backgroundWidth){
  //A collision has occurred.
  bugXDirection = -1.0f;//reverse direction
  //Set the position to the right edge less the width
  // of the sprite.
  bugX = backgroundWidth - bugWidth;

  //Compute traversal time for the bug to make one
  // round trip across the game window and back.
  long currentTime = gc.getTime();
  traversalTime = currentTime - oldTime;
  oldTime = currentTime;
}//end if
```

Table 6.7

Compute and save the traversal time

Then the code in Listing 4 (p. 108) computes the elapsed time since the previous collision of the sprite with the right edge of the game window and saves that time interval in the variable named **traversalTime** . That **traversalTime** value will be displayed when the **render** method is called producing the output shown in Figure 3 (p. 103) .

Test for collisions on other three sides of game window

Listing 5 (p. 109) tests for collisions between the sprite and the other three sides of the game window and takes appropriate action when a collision occurs. The code in these tests is less complex than in Listing 4 (p. 108) because they don't need to compute the **traversalTime** .

Listing 5 . Test for collisions on other three sides of game window.

```
  //Continue testing for collisions with the edges.
  if(bugX <= 0){
    bugXDirection = 1.0f;
    bugX = 0;
  }//end if

  if(bugY+bugHeight >= backgroundHeight){
    bugYDirection = -1.0f;
    bugY = backgroundHeight - bugHeight;
  }//end if

  if(bugY <= 0){
    bugYDirection = 1.0f;
    bugY = 0;
  }//end if

}//end update
```

Table 6.8

6.5.1.6 The render method

The render method is shown in its entirety in Listing 6 (p. 109) .

Listing 6 . The render method for Slick150a.

```
    public void render(GameContainer gc, Graphics g)
                              throws SlickException{
  //set the drawing mode to honor transparent pixels
  g.setDrawMode(g.MODE_NORMAL);

  //Draw the background to erase the previous picture.
  background.draw(0,0);

  //Draw the bug in its new location.
  bug.draw(bugX,bugY,bugScale);

  //Display the traversal time computed in the update
  // method.
  g.drawString(
            "traversalTime: "+traversalTime,100f,10f);

}//end render
```

Table 6.9

There is really nothing new in Listing 6 (p. 109) . Therefore, it shouldn't require an explanation beyond the embedded comments.

Each time this method is called, the location of the sprite will have changed by a few pixels relative to its previous location. Displaying the sprite in a new location each time the picture is drawn produces the impression that the sprite is moving.

6.5.2 A program with a highly variable frame rate - Slick0150b

This program differs from the previous program in that it attempts to maintain a constant overall speed as the bug moves across the game window despite the fact that the instantaneous frame rate varies quite a bit from one frame to the next. To accomplish this, the step size is made to vary in proportion to the delta value received by the **update** method, or inversely with the instantaneous frame rate.

A complete listing of the program is provided in Listing 10 (p. 123) near the end of the module. Most of the code in this program is the same as code in the previous program, so I will explain only the code that differs between the two.

6.5.2.1 The screen output for Slick0150b

Figure 4 (p. 113) shows a screen shot of the game window while this program is running. I will have more to say about this output later after I explain some of differences between this program and the program named **Slick0150a** .

Slick0150b, baldwin
FPS: 47   traversalTime: 3065
Put your name here


Slick0150b, baldwin
FPS: 47   traversalTime: 3065
Put your name here

Figure 4 . Screen shot of the program named Slick0150b.

Table 6.10

6.5.2.2 The render method

The significant differences between the two programs occur in the **update** method and the **render** method. I will begin with an explanation of the **render** method, which purposely creates an issue that is resolved by code in the **update** method.

The **render** method is shown in its entirety in Listing 7 (p. 114) .

Listing 7 . The render method for Slick150b.

```
    public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
  //set the drawing mode to honor transparent pixels
  g.setDrawMode(g.MODE_NORMAL);//honors transparency

  //Draw the background to erase the previous picture.
  background.draw(0,0);

  //Draw the bug in its new location.
  bug.draw(bugX,bugY,bugScale);

  //Display the traversal time computed in the update
  // method.
  g.drawString(
            "traversalTime: "+traversalTime,100f,10f);

  //Purposely insert a time delay.
  int sleepTime = (((byte)random.nextInt()) + 128)/6;
  gc.sleep(sleepTime);

}//end render
```

Table 6.11

Purposely insert a time delay

Everything down to the last two lines of code in Listing 7 (p. 114) is the same as the program named **Slick0150a** . At that point I inserted code that will cause an additional random time delay ranging from 0 to 43 milliseconds before the **render** method returns. I did this to simulate a situation in which the rendering process is very complex and the time to render varies quite a lot from one frame to the next.

A new average frame rate

In this case, the average additional delay time should be about 21.5 msec. This makes it impossible to maintain the target frame rate of 60 frames per second or 16.666 milliseconds per frame.

This additional delay should result in an average frame rate of about 46 or 47 frames per second, which is consistent with the screen output shown in Figure 4 (p. 113) .

A wide variation in delta values

Not only does this code result in a reduction in the average frame rate, it also results in a wide variation in the values of delta received by the **update** method on a frame to frame basis.

As before, the **init** method calls the **setTargetFrameRate** method requesting a frame rate of 60 frames per second. This guarantees that the minimum delta that will be received by the **update** method will be in the neighborhood of 16 milliseconds. *(The game loop won't be allowed to run any faster than 60 frames per second.)*

The last two lines of code in Listing 7 (p. 114) will cause the value of delta to be as large as about 43 milliseconds.

Therefore, the incoming delta values in the **update** method will vary between about 16 milliseconds and about 43 milliseconds on a totally random basis from one frame to the next.

6.5.2.3 The update method

Listing 8 (p. 115) shows the code in the **update** method that is different from the code in the **update** method for the program named **Slick0150b** .

Listing 8 . Beginning of the update method for Slick0150b.

```
      public void update(GameContainer gc, int delta)
                                    throws SlickException{
    //Compute new location for the sprite.
    bugX += bugXDirection*xStep*delta*targetFPS/1000.0;
    bugY += bugYDirection*yStep*delta*targetFPS/1000.0;

    //The following code does not correct for variations
    // in delta. The step size is always the same
    // regardless of delta. Enable this code and disable
    // the two statements above to see the effect.
//    bugX += bugXDirection*xStep;
//    bugY += bugYDirection*yStep;
```

Table 6.12

Compute new location for the sprite

As before, the method begins by computing a new location for the sprite. However, the code in Listing 8 (p. 115) attempts to maintain a constant overall speed as the bug moves across the game window despite the fact that the value of delta varies quite a bit from one frame to the next.

Vary the step size

In order to accomplish this, the step size is caused to vary in proportion to delta or inversely with the instantaneous frame rate. Given the earlier estimate that the value of delta can vary from about 16 milliseconds to about 43 milliseconds, the step size can vary from about 4 pixels per frame to about 10 pixels per frame. When the value of delta is small, the step size will be small. When the value of delta is large, the step size will be large.

Maintaining a constant overall speed of motion

As a result of the long time delays introduced into the **render** method, the average frame rate has been slowed down to around 47 frames per second as shown in Figure 4 (p. 113) . However, as also shown in Figure 4 (p. 113) , the traversal time continues to be close to the target of around 3000 milliseconds. Therefore, the algorithm is deemed to be successful in maintaining a relatively constant overall speed of motion.

The visible output

This algorithm and the widely varying values of delta result in sprite motion that isn't as smooth as with the program named **Slick0150b** . However, the sprite gets to where it needs to be when it needs to be there despite widely varying values of delta, and that is the objective of the algorithm.

Results without correction for varying frame rate

The last two statements in Listing 8 (p. 115) show an alternative approach that does not attempt to correct for variations in the value of delta.

(This approach is essentially the same as was used in the program named ***Slick0150a*** *above.)*

When the first two statements in Listing 8 (p. 115) are disabled and the last two statements in Listing 8 (p. 115) are enabled, the output is as shown in Figure 5 (p. 119) .


Slick0150b, baldwin
FPS: 43   traversalTime: 4337
Put your name here

Slick0150b, baldwin

FPS: 43   traversalTime: 4337

Put your name here

FPS: 43   traversalTime: 4337

Figure 5 . Screen shot of the program named Slick0150b without correction for varying frame rate.

Slick0150b, baldwin

FPS: 43 traversalTime: 4337

Put your name here

Table 6.13

Increased traversal time

Figure 5 (p. 119) shows the output of a system where the value of delta varies widely but no correction is made for those variations. As you can see, the frame rate is reduced as in Figure 4 (p. 113) . As you can also see, the traversal time is increased significantly from around 3000 milliseconds to around 4300 milliseconds. As a result, the sprite does ***not*** get to where it needs to be when it needs to be there.

6.6 Run the programs

I encourage you to copy the code from Listing 9 (p. 121) and Listing 10 (p. 123) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

6.7 Summary

In this module, you learned how to make sprites move at a constant speed in front of an image in the face of a widely varying frame rate. You also learned about a rudimentary form of collision detection.

6.8 What's next?

In the next module, you will learn about using the **draw** , **drawCentered** , and **drawFlash** methods of the Image [3] class.

6.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0150: A first look at sprite motion, collision detection, and timing control

- File: Slick0150.htm
- Published: 02/04/13
- Revised: 06/09/15 for 64-bit

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

6.10 Complete program listings

Listing 9 (p. 121) and Listing 10 (p. 123) provide complete listings of the programs discussed in this module.

---

[3] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Image.html

Listing 9 . Source code for the program named Slick0150a.

```
/*Slick0150a.java
Copyright 2012, R.G.Baldwin

Cause a ladybug sprite to bounce around inside the game
window.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;

public class Slick0150a extends BasicGame{

  Image bug = null;
  Image background = null;

  float backgroundWidth;
  float backgroundHeight;

  float bugX = 100;
  float bugY = 100;
  float bugWidth;
  float bugHeight;

  float bugXDirection = 1.0f;//initial direction to right
  float bugYDirection = 1.0f;//initial direction is down

  float xStep = 4.0f;//horizontal step size
  float yStep = 3.0f;//vertical step size

  float bugScale = 0.75f;//drawing scale factor

  //Used to compute and display the time required for the
  // bug to make each round trip across the game window
  // and back.
  long oldTime = 0;
  long traversalTime = 0;

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;
  //----------------------------------------------------//

  public Slick0150a(){//constructor
    //Set the title
    super("Slick0150a, baldwin");
  }//end constructor
  //----------------------------------------------------//
```

Table 6.14

.

Listing 10 . Source code for the program named Slick0150b.

```
/*Slick0150b.java
Copyright 2012, R.G.Baldwin

Cause a ladybug sprite to bounce around inside the game
window.

A random time delay is inserted in the render method to
simulate a situation where the rendering process is very
complex and the time to render varies from one frame to
the next.

The program attempts to maintain a constant physical
speed as the bug moves across the game window despite
the fact that the delta varies quite a bit from one
frame to the next. The step size varies in proportion
to delta or inversely with frame rate.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import java.util.Random;

public class Slick0150b extends BasicGame{
  Random random = new Random();

  Image bug = null;
  Image background = null;

  float backgroundWidth;
  float backgroundHeight;

  float bugX = 100;//initial position of ladybug
  float bugY = 100;
  float bugWidth;
  float bugHeight;

  float bugXDirection = 1.0f;//initial direction to right
  float bugYDirection = 1.0f;//initial direction is down

  float xStep = 4.0f;//horizontal step size
  float yStep = 3.0f;//vertical step size

  float bugScale = 0.75f;//drawing scale factor

  //Used to compute and display the time required for the
  // bug to make each round trip across the game window
  // and back.
  long oldTime = 0;
```

Table 6.15

-end-

Chapter 7

Slick0160: Using the draw and drawFlash methods.[1]

7.1 Table of Contents

7.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

---

[1] This content is available online at <http://cnx.org/content/m45748/1.5/>.

An earlier module titled A first look at Slick2D bitmap graphics [2] introduced you to the use of bitmap graphics in Slick2D. The purpose of this module is dig a little deeper into the use of bitmap graphics

7.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

7.2.1.1 Figures

7.2.1.2 Listings

7.3 Preview

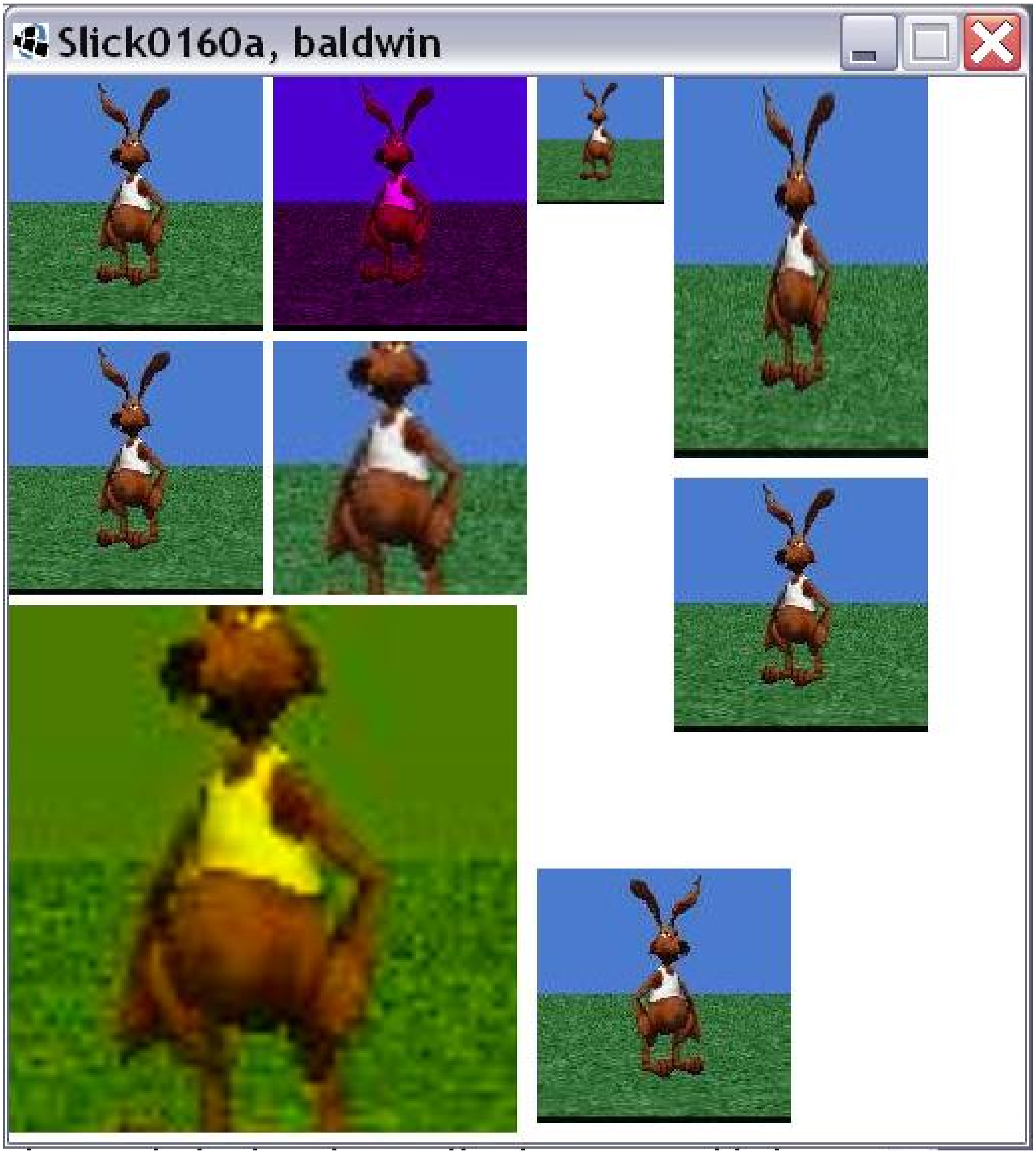
Bitmap graphics are used in a variety of ways in game and simulation programming. Therefore, I will present and explain two programs that dig more deeply into the use of bitmap graphics in Slick2D.

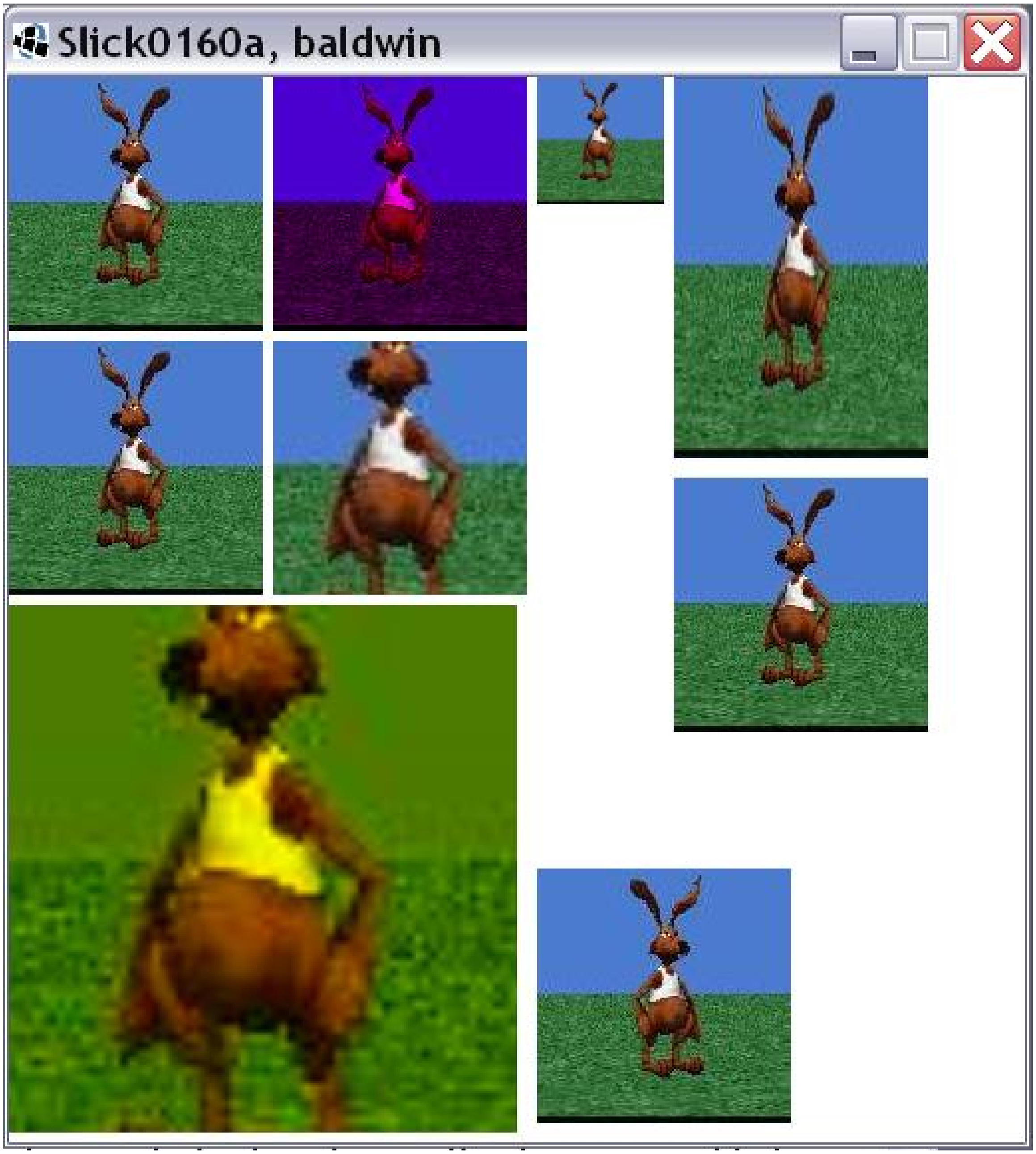
The program named Slick0160a

The first program named **Slick0160a** calls the **draw** method of the **Image** class several times in succession to illustrate some of the options available with the draw method. This program also illustrates flipping an image. The output from this program is shown in Figure 1 (p. 130) .

---

[2] http://cnx.org/contents/ec409a1f-e946-486a-a681-980d0effa996/Slick0140-A-first-look-at-Slic


Slick0160a, baldwin


Slick0160a, baldwin

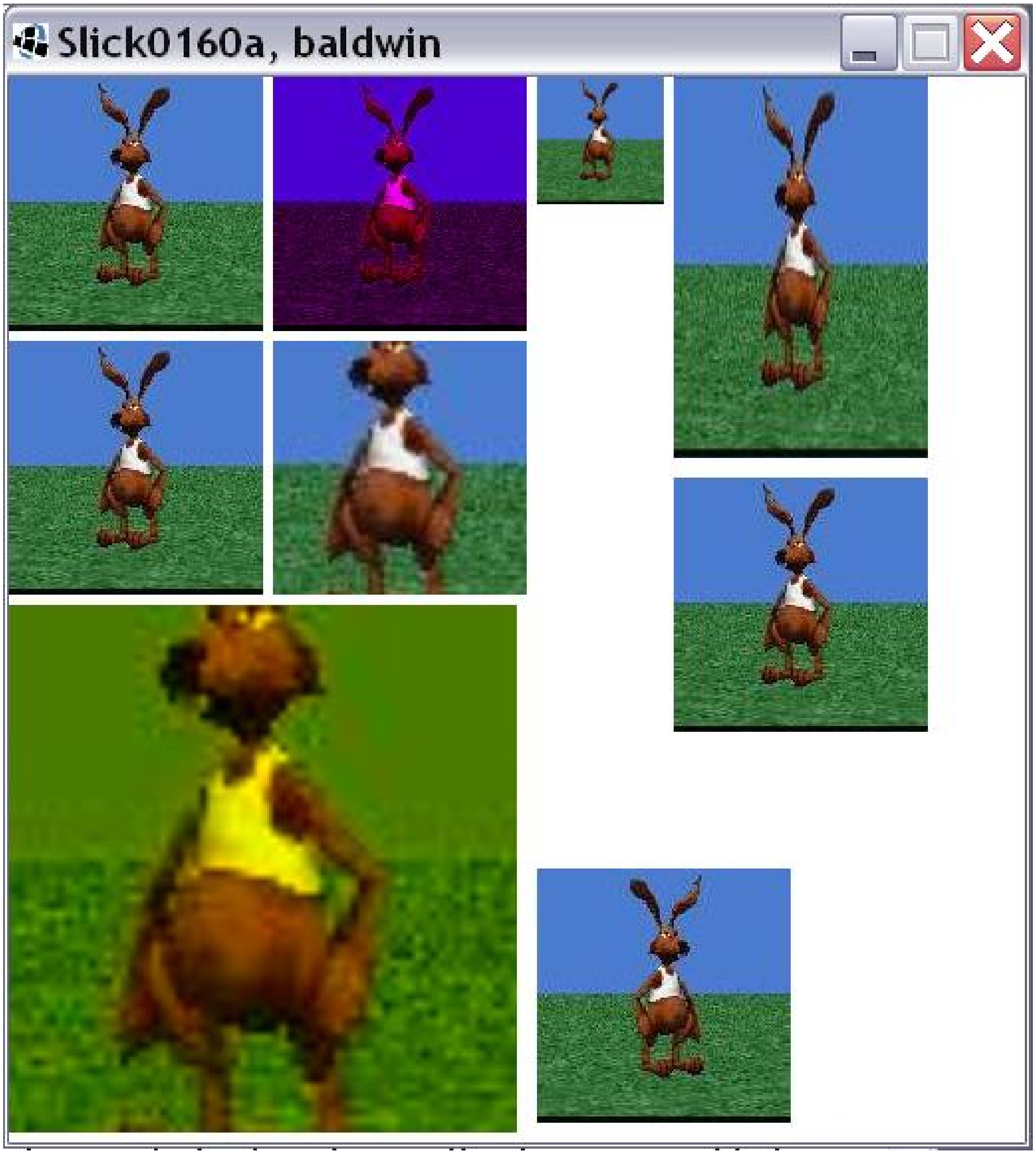

Slick0160a, baldwin

Figure 1 . Output from the program named Slick0160a.

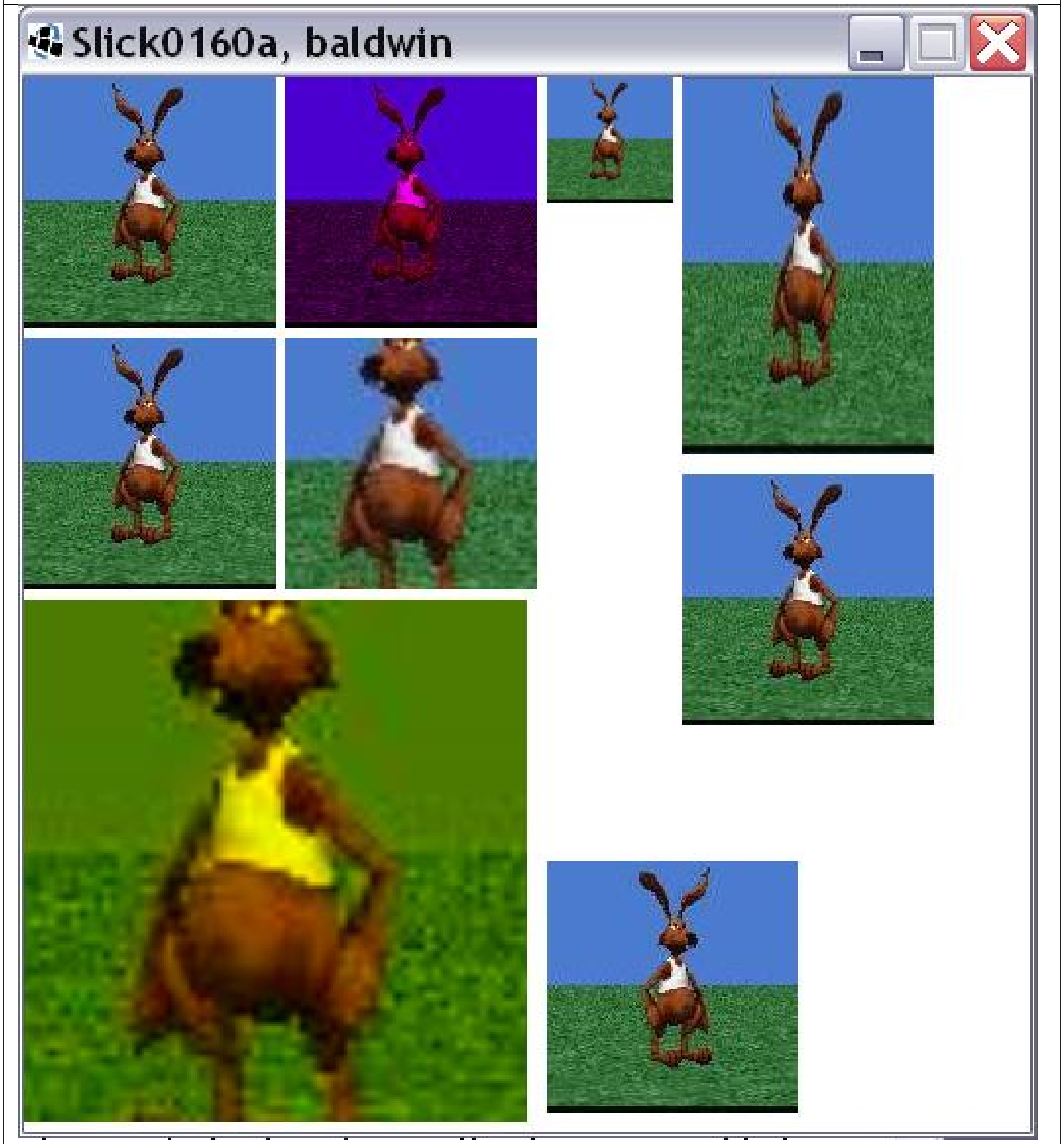
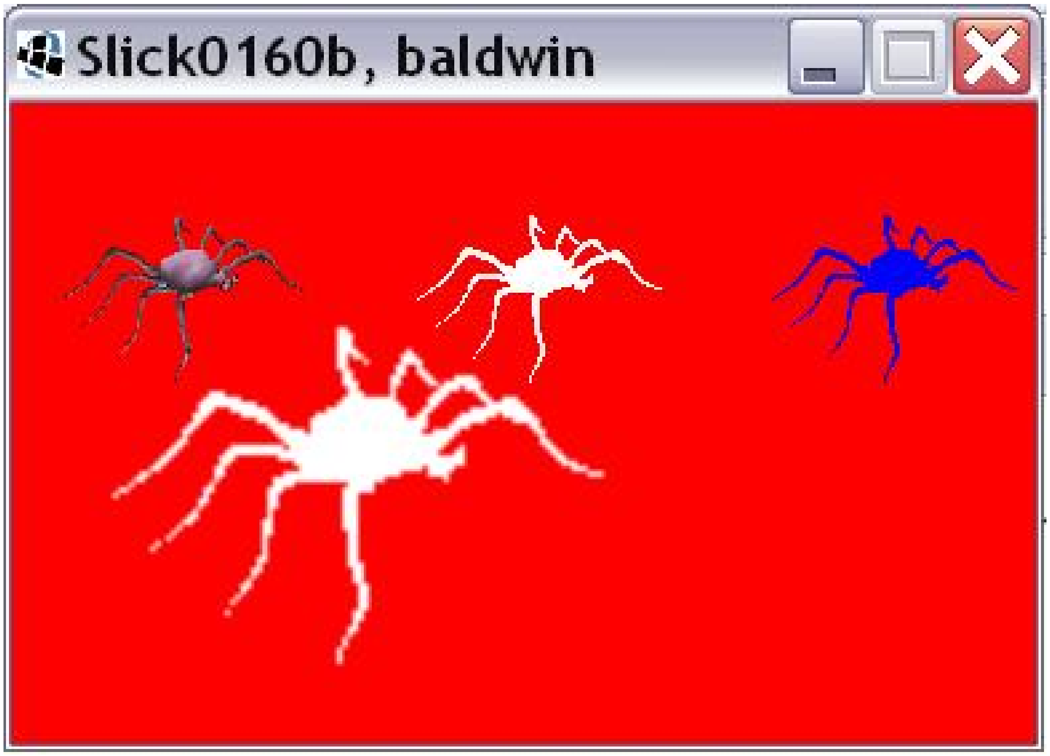
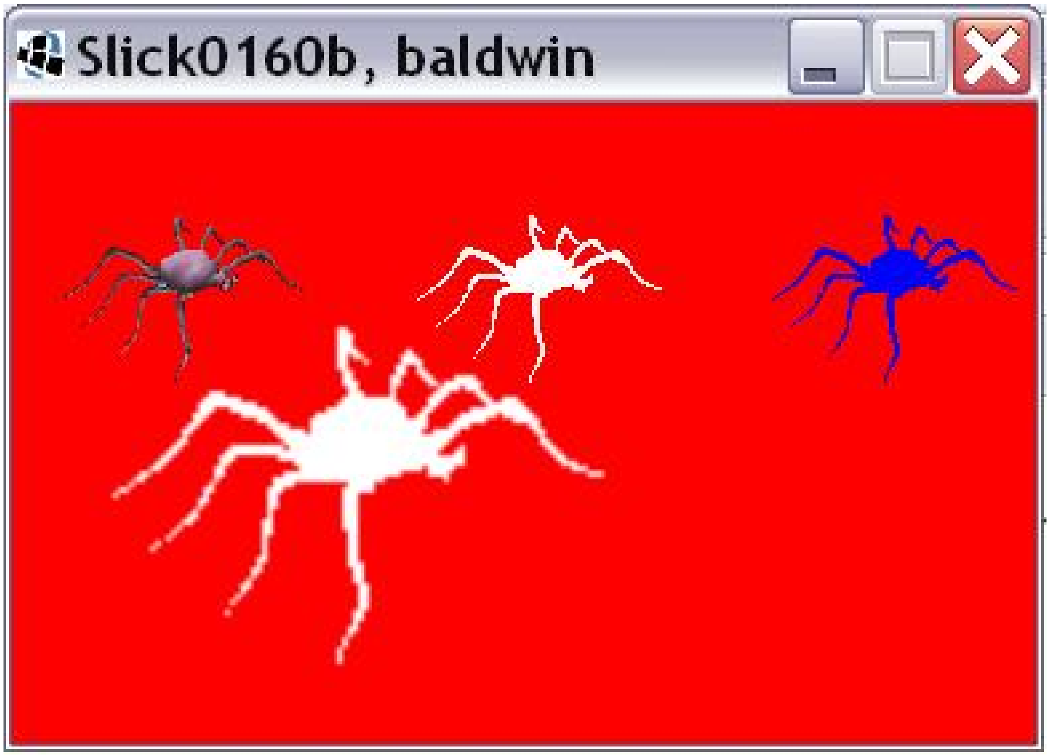


Table 7.1

The program named Slick0160b

The second program named **Slick0160b** illustrates the use of the **drawFlash** method to draw an image in silhouette and to cause the silhouette to switch back and forth between two or more colors. The program draws several spiders in silhouette. It causes a large spider to flash back and forth between a white silhouette and a blue silhouette. A screen shot of the output from the program while the large spider is in its white state is shown in Figure 2 (p. 133) .


Slick0160b, baldwin


Slick0160b, baldwin

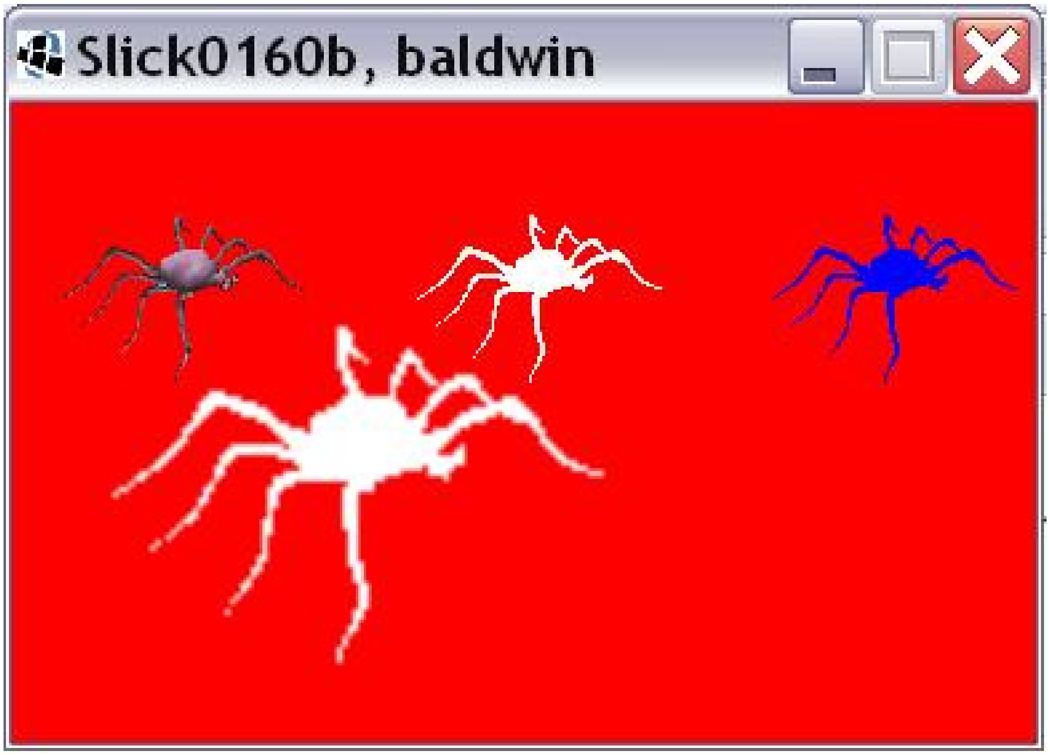

Slick0160b, baldwin

Figure 2 . One output from the program named Slick0160b.

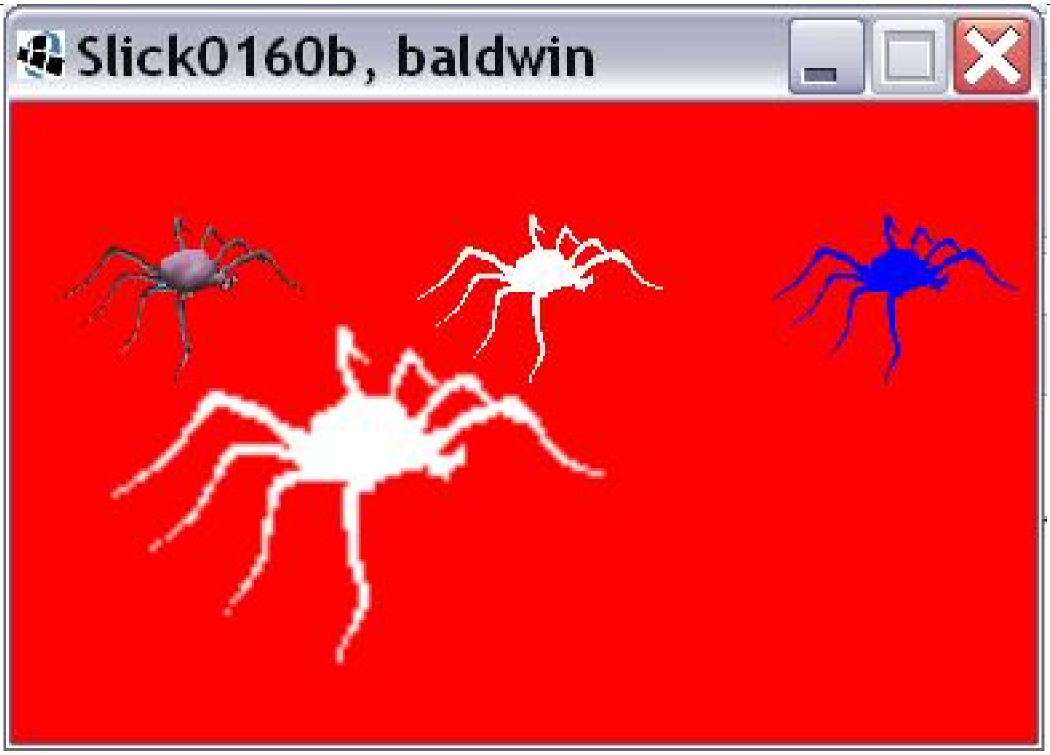
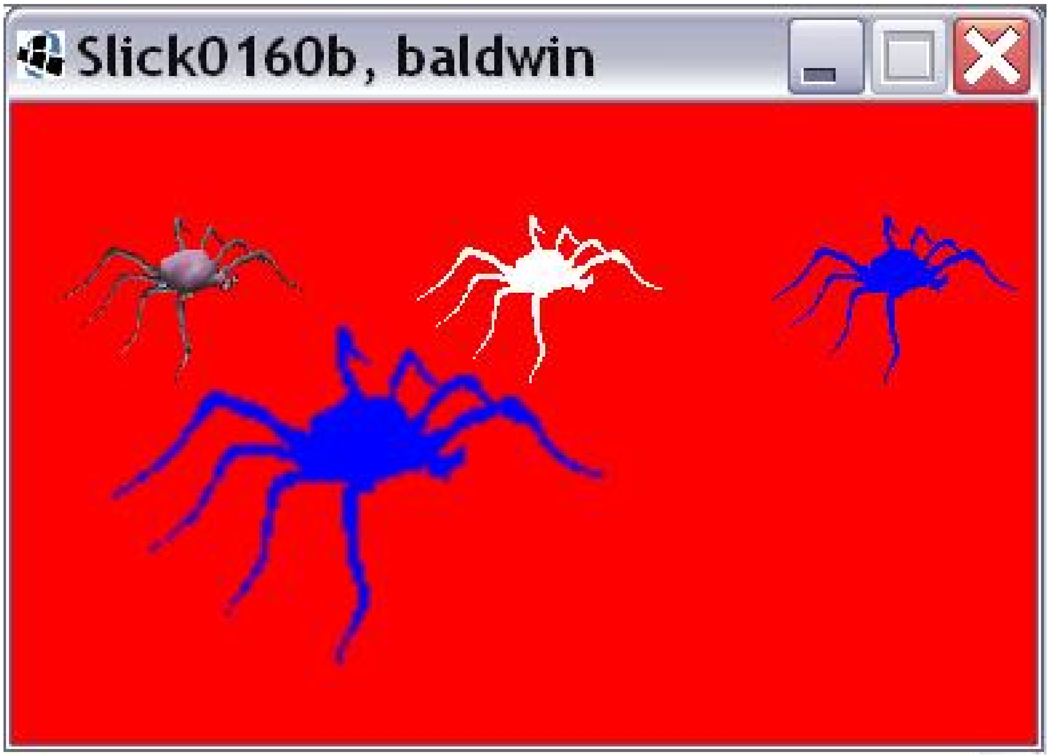
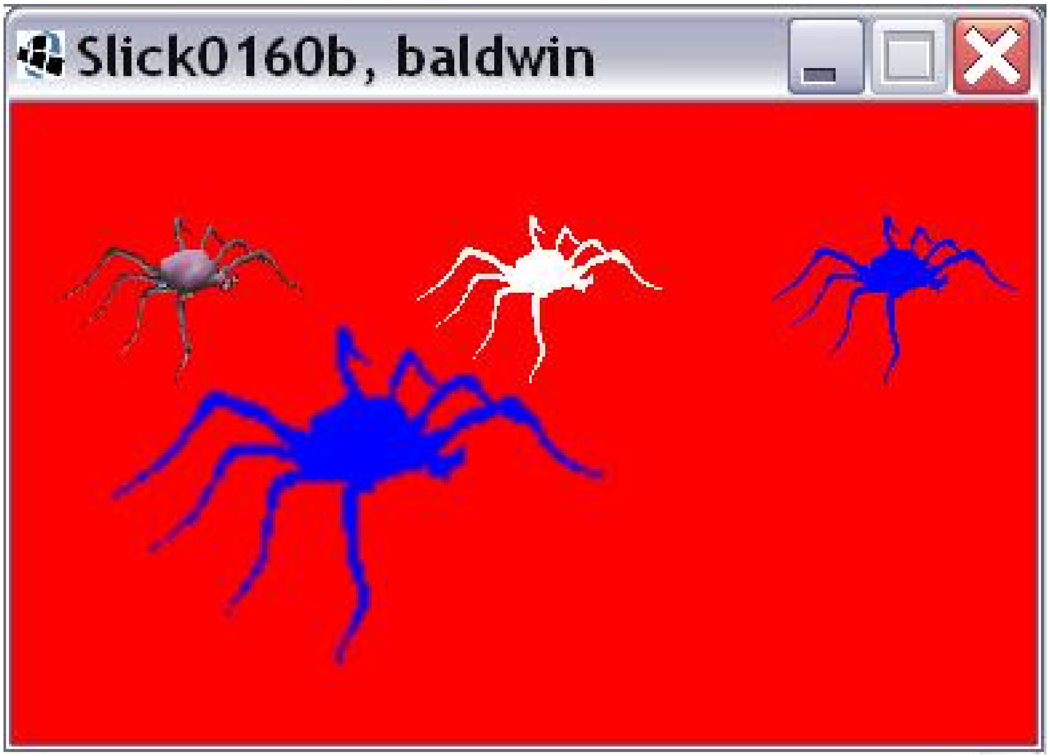


Table 7.2

A screen shot of the output from the program while the large spider is in its blue state is shown in Figure 3 (p. 136) .


Slick0160b, baldwin


Slick0160b, baldwin

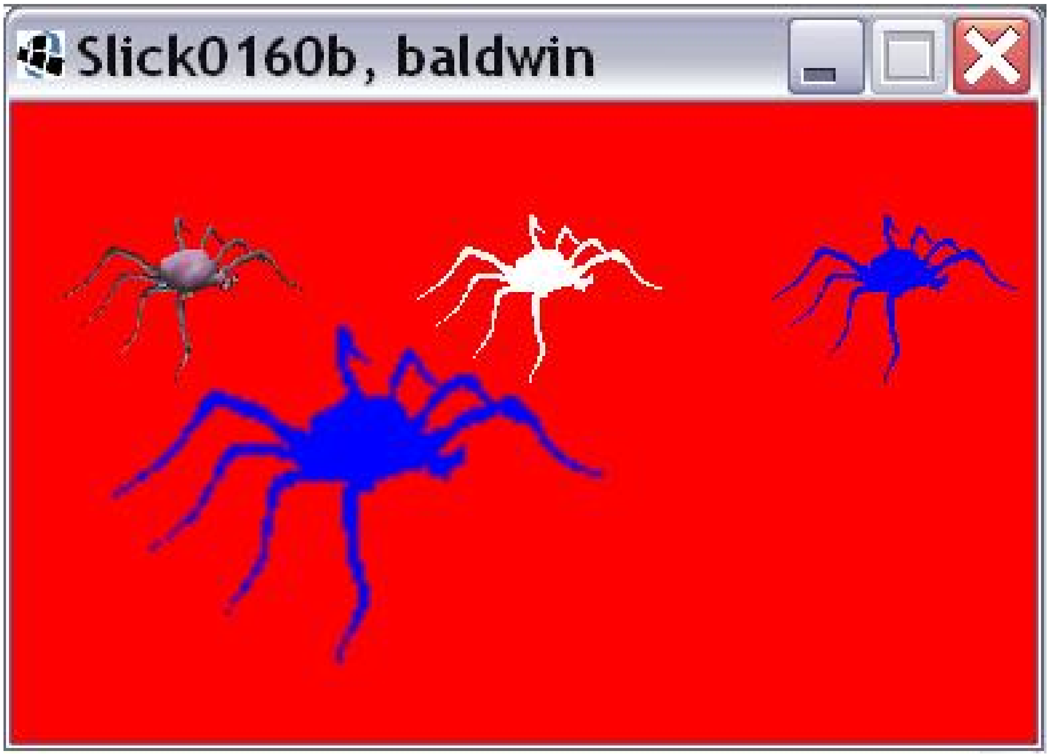

Slick0160b, baldwin

Figure 3 . Another output from the program named Slick0160b.

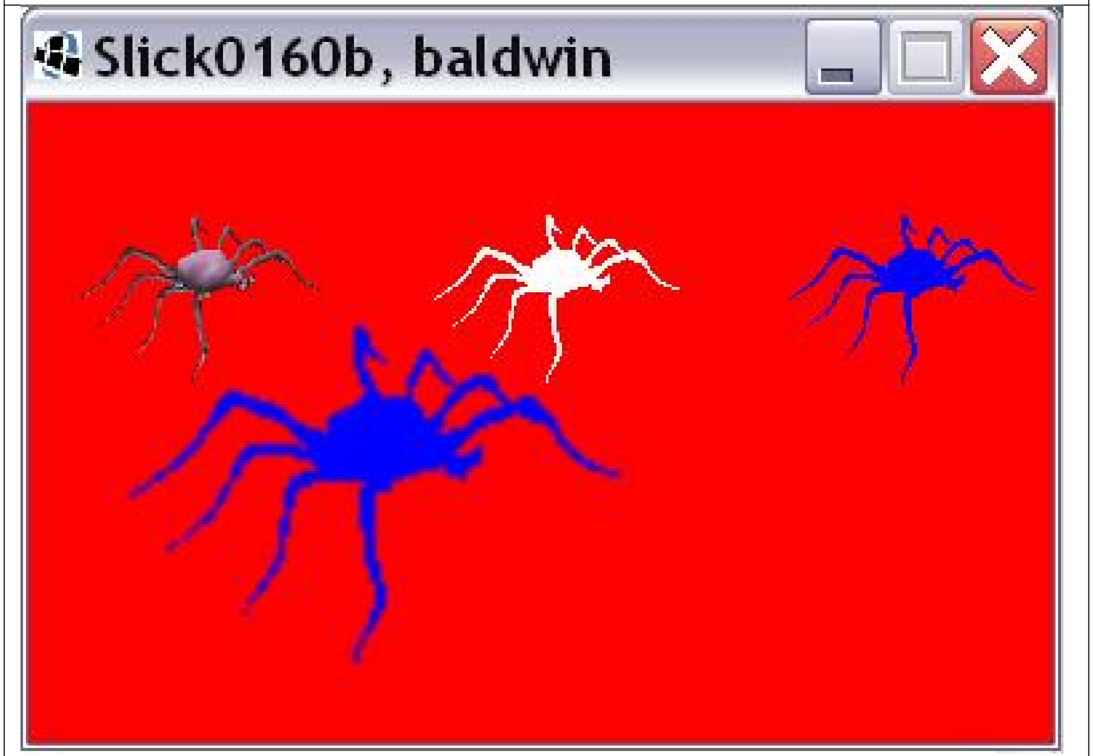


Table 7.3

What you have learned

In the previous module, you learned how to make sprites move at a constant speed in front of an image in the face of widely varying frame rates. You also learned about a rudimentary form of collision detection.

What you will learn

In this module, you will learn about using the **draw** , **drawCentered** , and **drawFlash** methods of the Image [3] class.

7.4 General background information

The Slick2D Image [4] class defines about ten overloaded versions of the **draw** method. We will investigate several of them in this module.

The class also defines three overloaded versions of the **drawFlash** method along with a method named **drawCentered** . We will also investigate some of them.

7.5 Discussion and sample code

7.5.1 The program named Slick0160a

7.5.1.1 Beginning of the Slick0160a class

Will discuss in fragments

---

[3] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Image.html

[4] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Image.html

A complete listing of this program is provided in Listing 13 (p. 148) . As is my custom, I will break this program down and discuss it in fragments.

Listing 1 (p. 137) shows the beginning of the class down through the constructor.

Listing 1 . Beginning of the Slick0160a class.

```
import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

public class Slick0160a extends BasicGame{

  Image rabbit = null;

  float rabbitWidth;
  float rabbitHeight;

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;
  //----------------------------------------------------//

  public Slick0160a(){//constructor
    //Set the title
    super("Slick0160a, baldwin");
  }//end constructor
  //----------------------------------------------------//
```

Table 7.4

As usual, it is necessary to declare several import directives that point to classes in the Slick2D library. Also, as in several previous modules, the new class extends the Slick2D class named **BasicGame** .

Listing 1 (p. 137) declares several instance variables, initializing some of them.

The constructor simply sets the title on the game window.

7.5.1.2 The main method

The **main** method is shown in Listing 2 (p. 138) . There is nothing in Listing 2 (p. 138) that you haven't seen in several previous modules.

Listing 2 . The main method.

```
public static void main(String[] args)
                                  throws SlickException{
  AppGameContainer app = new AppGameContainer(
                      new Slick0160a(),512,537,false);
  app.start();
}//end main
```

Table 7.5

7.5.1.3 The init method

The **init** method is shown in Listing 3 (p. 138) .

There is nothing in Listing 3 (p. 138) that you haven't seen in previous modules.

Listing 3 . The init method.

```
@Override
public void init(GameContainer gc)
                                 throws SlickException {

  rabbit = new Image("rabbit.png");

  rabbitWidth = rabbit.getWidth();
  rabbitHeight = rabbit.getHeight();

  System.out.println(
                 "rabbitWidth: " + rabbitWidth);
  System.out.println(
               "rabbitHeight: " + rabbitHeight);

  gc.setShowFPS(false) ;
  gc.setTargetFrameRate(targetFPS);//set frame rate
}//end init
```

Table 7.6

7.5.1.4 The update method

The body of the **update** method is empty so it isn't shown here. You can view it in Listing 13 (p. 148) .

7.5.1.5 The render method

The interesting code in this program is in the **render** method, which begins in Listing 4 (p. 140) . There is nothing in Listing 4 (p. 140) that you haven't seen before.

Listing 4 . The render method.

```
public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
  //set the drawing mode to honor transparent pixels
  g.setDrawMode(g.MODE_NORMAL);
  g.setBackground(Color.white);
```

Table 7.7

Draw top four images

The code in Listing 5 (p. 140) calls four different overloaded versions of the **draw** method on the rabbit image to draw the top four images in Figure 1 (p. 130) .

Listing 5 . Draw the top four images.

```
  rabbit.draw(0f,0f);
  rabbit.draw(133f,0f,new Color(1.0f,0.0f,1.0f));
  rabbit.draw(266f,0f,0.5f);
  rabbit.draw(335f,0f,128f,192);
```

Table 7.8

Draw unchanged at the origin

The first call to the **draw** method in Listing 5 (p. 140) simply draws the image unchanged with its upper-left corner at the upper-left corner *(the origin)* of the game window.

Apply a color filter before drawing

The second call to the **draw** method in Listing 5 (p. 140) applies a color filter to the rabbit image and draws it with its upper-left corner at 133,0.

I haven't found an explanation as to exactly how the color filter is applied. It appears from experimentation that the pixel color values in the original image are multiplied by the red, green, and blue color values *(expressed in the range from 0 to 1.0)* in the color object that is passed as a parameter to the method. However, the **Image** class defines two constants named **FILTER_LINEAR** and **FILTER_NEAREST** that may have something to do with how color filtering is applied.

Apply a uniform scale factor before drawing

The third call to the **draw** method in Listing 5 (p. 140) applies a scale factor of 0.5 to both dimensions of the rabbit image and draws it with its upper-left corner at 266,0.

Change the dimensions before drawing

The fourth call to the **draw** method in Listing 5 (p. 140) resizes the rabbit image to a width of 128 pixels and a height of 192 pixels and draws the modified image with its upper-left corner at 335,0.

Draw three more images

The code in Listing 6 (p. 141) calls three different overloaded versions of the **draw** method on the rabbit image to draw the two images on the center left and the large image on the bottom left of Figure 1 (p. 130) .

Listing 6 . Draw three more images.

```
rabbit.draw(0f,133f);
rabbit.draw(133f,133f,32f,32f,96f,96f);
rabbit.draw(0f,266f,256f,532f,32f,32f,96f,96f,new Color(1.0f,1.0f,0.0f));
```

Table 7.9

Draw another unchanged version

The first call to the **draw** method in Listing 6 (p. 141) simply draws another unchanged version of the rabbit image in a new location, 0,133.

Extract and draw a rectangular section

The second call to the **draw** method in Listing 6 (p. 141) extracts a rectangular section from the original image and draws it the same size as the original image with its upper-left corner at 133,133.

The third and fourth parameters *(32,32)* specify the coordinates of the upper-left corner of the rectangle that is extracted. The fifth and sixth parameters *(96,96)* specify the coordinates of the lower-right corner of the rectangle that is extracted.

Extract and draw another rectangular section with color filtering

The third call to the **draw** method in Listing 6 (p. 141) extracts a rectangular section from the original image and draws it with a different size and also applies a color filter. I will leave it as an exercise for the student to go to the Slick2D documentation [5] for an explanation of the eight parameters of type **float** .

Draw image based on its center

The previous examples have drawn the image in a location based on its upper-left corner. The statement in Listing 7 (p. 141) draws the image on the center right in Figure 1 (p. 130) . However, instead of positioning the image based on its upper-left corner, the image is drawn with its center located at 399,266.

Listing 7 . Draw image based on its center.

```
rabbit.drawCentered(399f,266f);
```

Table 7.10

Draw a flipped copy

The code in Listing 8 (p. 142) makes a call to the **getFlippedCopy** method of the **Image** class, followed by a call to the **draw** method to draw the image in the bottom-right of Figure 1 (p. 130) . Note that the rabbit is facing the opposite direction in that image. The **boolean** parameters specify whether the image is to be flipped on the horizontal, vertical, or both axes. In this case, a value of true caused the image to be flipped on the horizontal axis only.

Listing 8 . Draw a flipped copy.

continued on next page

---

[5] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Image.html

```
    Image tempImage = rabbit.getFlippedCopy(true,false);
    tempImage.draw(266f,399f);

  }//end render

}//end class Slick0160a
```

Table 7.11

Listing 8 (p. 142) also signals the end of the **render** method and the end of the class named **Slick0160a** .

The Slick2D **Image** class provides many additional capabilities that are not illustrated in this program. I will leave it as an exercise for the student to explore them. However, there is one other set of three overloaded methods named **drawFlash** that I will illustrate in this module. That is the topic of the next section.

7.5.2 The program named Slick0160b

7.5.2.1 Beginning of the class named Slick0160b

A complete listing of this program is provided in Listing 14 (p. 150) . As before, I will break this program down and discuss it in fragments.

Listing 9 (p. 143) shows the beginning of the class named **Slick0160b** down through the **init** method.

Listing 9 . Listing 9, Beginning of the Slick0160b class.

```
import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

public class Slick0160b extends BasicGame{

  Image spider = null;

  float spiderWidth;
  float spiderHeight;

  Color silohetteColor = Color.white;
  long timeAccumulator = 0;
  long flashInterval = 128;

  //Target frame rate
  int targetFPS = 60;
  //----------------------------------------------------//

  public Slick0160b(){//constructor
    //Set the title
    super("Slick0160b, baldwin");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    AppGameContainer app = new AppGameContainer(
                          new Slick0160b(),384,240,false);
    app.start();
  }//end main
  //----------------------------------------------------//

  @Override
  public void init(GameContainer gc)
                                   throws SlickException {

    spider = new Image("spider.png");

    spiderWidth = spider.getWidth();
    spiderHeight = spider.getHeight();

    System.out.println("spiderWidth: " + spiderWidth);
    System.out.println("spiderHeight: " + spiderHeight);

    gc.setShowFPS(false);
    gc.setTargetFrameRate(targetFPS);//set frame rate
  }//end init
```

Table 7.12

There is nothing new or unusual about the code in Listing 9 (p. 143) . Therefore, no explanation beyond the embedded comments should be needed. You might want to note the instance variables at the beginning of the class. They will be used in the code that I will describe later.

7.5.2.2 The update method

Unlike the previous program, which simply displayed what appeared to be static images, this program causes one of the images to change as the program runs. These changes are programmed into the **update** method, which is shown in Listing 10 (p. 144) .

Listing 10 . The update method.

```
public void update(GameContainer gc, int delta)
                                  throws SlickException{
  timeAccumulator += delta;
  if(timeAccumulator >= flashInterval){
    //Reset accumulator and change color of spider
    // silhouette
    timeAccumulator = 0;
    if(silohetteColor.equals(Color.white)){
      silohetteColor = Color.blue;
    }else{
      silohetteColor = Color.white;
    }//end if
  }//end if
}//end update
```

Table 7.13

Switch color between white and blue

Recall that *(by default)* the **update** method is executed once during each iteration of the game loop. *(Other programming options are available regarding if, when, and how many times the* ***update*** *method is executed during one iteration of the game loop.)*

The purpose of the **update** method is to execute the program logic.

In this case, the program logic is simple; to switch the color of the large spider in Figure 2 (p. 133) between white and blue on a regular schedule.

As you will see in the **render** method later, the color of the large spider is determined by the value of the instance variable named **silohetteColor** , which is initialized to white in Listing 9 (p. 143) . Recall that the target frame rate is set to 60 frames per second in Listing 9 (p. 143) . A variable named **flashInterval** is initialized to 128 in Listing 9 (p. 143) .

Also recall that the value of the incoming parameter named **delta** is the number of **milliseconds** since the last time that the update method was called. This value was used in a significant way in the earlier module titled A first look at sprite motion, collision detection, and timing control [6] .

The program logic

Each time the **update** method is called, the incoming value of **delta** is added to the value in a variable named **timeAccumulator** . When the accumulated time meets or exceeds the value of **flashInterval**

---

[6] http://cnx.org/contents/c92f070e-494f-4eb1-a0b9-e36abe4359fd

, the color is switched from white to blue, or from blue to white, depending on its current value. Also the time accumulator is set to zero and a new white/blue cycle begins.

The color switch should occur approximately every 128 milliseconds, or about eight times per second.

7.5.2.3 The render method

Recall that the execution of program logic in the **update** method does not cause the player's view of the game to change. It is not until the **render** method is executed that the images on the screen change.

The **render** method begins in Listing 11 (p. 145) .

Listing 11 . Beginning of the render method.

```
public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
  //set the drawing mode to honor transparent pixels
  g.setDrawMode(g.MODE_NORMAL);
  g.setBackground(Color.red);

  //Draw the spider.
  spider.draw(0f,0f);

  //Draw a white silhouette of the spider.
  spider.drawFlash(133f,0f);

  //Draw a blue silhouette of the spider.
  spider.drawFlash(266f,0f,131f,128f,Color.blue);
```

Table 7.14

After setting the drawing mode to honor transparency and setting the background color to red, the code in Listing 11 (p. 145) causes the three images of the spider along the top of Figure 2 (p. 133) to be displayed each time the **render** method is called. *(Recall that the names of the drawing mode constants appear to be reversed between opaque and transparent.)*

The call to the **draw** method in Listing 11 (p. 145) displays the spider image in the upper-left corner of Figure 2 (p. 133) .

The first call to the **drawFlash** method calls one of three overloaded versions of the **drawFlash** method. This version of the method draws a white silhouette of the spider at a location specified by the parameters, 133,0.

The second call to the **drawFlash** method draws a silhouette of the spider at a location of 266,0, with a width of 131 pixels, a height of 128 pixels and a blue color.

Note that none of the code in Listing 11 (p. 145) depends on the logic that is executed in the **update** method. Therefore, the three images appear to be static despite the fact that they are being redrawn about 60 times per second.

The large flashing spider

The call to the **drawFlash** method in Listing 12 (p. 146) produces the large flashing spider at the bottom of Figure 2 (p. 133) and Figure 3 (p. 136) .

Listing 12 . The large flashing spider.

```
    //Cause an enlarged version of the spider to flash
    // between white and blue silhouette at a rate
    // of 1/flashInterval.
    spider.drawFlash(0f,0f,262f,256f,silohetteColor);

  }//end render

}//end class Slick0160b
```

Table 7.15

This is the same version of the **drawFlash** method that was called to produce the blue spider in the upper-right corner of Figure 2 (p. 133) . However, in this case, the third and fourth parameters specify that the spider should be drawn with a width of 262 pixels and a height of 256 pixels.

More importantly, rather than passing a constant color as the last parameter, Listing 12 (p. 146) passes the reference to the **Color** object stored in the variable named **silohetteColor** . Recall that the color represented by that object is periodically switched between white and blue in the **update** method of Listing 10 (p. 144) . This causes the color of the spider to switch between white and blue.

The end of the program

Listing 12 (p. 146) signals the end of the **render** method and the end of the class named **Slick0160b** .

7.6 Run the programs

I encourage you to copy the code from Listing 13 (p. 148) and Listing 14 (p. 150) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

7.7 Summary

In this module, you learned about using the **draw** , **drawCentered** , and **drawFlash** methods of the Image [7] class.

7.8 What's next?

In the next module, you will learn how to use the following methods of the Input [8] class to get user input:

- isKeyDown
- isMouseButtonDown
- getMouseX
- getMouseY

---

[7] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Image.html

[8] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Input.html

7.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0160: Using the draw and drawFlash methods
- File: Slick0160.htm
- Published: 02/05/13
- Revised: 09/03/15

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

7.10 Complete program listing

Complete listings of the programs discussed in this module are shown in Listing 13 (p. 148) and Listing 14 (p. 150) below.

Listing 13 . Source code for Slick0160a.java.

```
/*Slick0160a.java
Copyright 2012, R.G.Baldwin

Calls the draw method several times in succession to
illustrate the various options available with the draw
method. Also illustrates flipping an image.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

public class Slick0160a extends BasicGame{

  Image rabbit = null;

  float rabbitWidth;
  float rabbitHeight;

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;
  //--------------------------------------------------//

  public Slick0160a(){//constructor
    //Set the title
    super("Slick0160a, baldwin");
  }//end constructor
  //--------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    AppGameContainer app = new AppGameContainer(
                          new Slick0160a(),512,537,false);
    app.start();
  }//end main
  //--------------------------------------------------//

  @Override
  public void init(GameContainer gc)
                                   throws SlickException {

    rabbit = new Image("rabbit.png");
    rabbitWidth = rabbit.getWidth();
    rabbitHeight = rabbit.getHeight();

    System.out.println(
```

Table 7.16

.

Listing 14 . Source code for Slick0160b.java.

```
/*Slick0160b.java
Copyright 2012, R.G.Baldwin

Illustrates the drawFlash method to draw an image in
silhouette  format and to cause the silhouette  to switch
back and forth between two or more colors.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

public class Slick0160b extends BasicGame{

  Image spider = null;

  float spiderWidth;
  float spiderHeight;

  Color silohetteColor = Color.white;
  long timeAccumulator = 0;
  long flashInterval = 128;

  //Target frame rate
  int targetFPS = 60;
  //----------------------------------------------------//

  public Slick0160b(){//constructor
    //Set the title
    super("Slick0160b, baldwin");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    AppGameContainer app = new AppGameContainer(
                          new Slick0160b(),384,240,false);
    app.start();
  }//end main
  //----------------------------------------------------//

  @Override
  public void init(GameContainer gc)
                                   throws SlickException {

    spider = new Image("spider.png");

    spiderWidth = spider.getWidth();
```

Table 7.17

-end-

Chapter 8

Slick0170: Mouse and keyboard input[1]

8.1 Table of Contents

8.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is to explain some aspects of mouse and keyboard input.

8.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

8.2.1.1 Figures

- Figure 1 (p. 157) . Output from the program named Slick0170.java.

---

[1] This content is available online at <http://cnx.org/content/m45750/1.4/>.

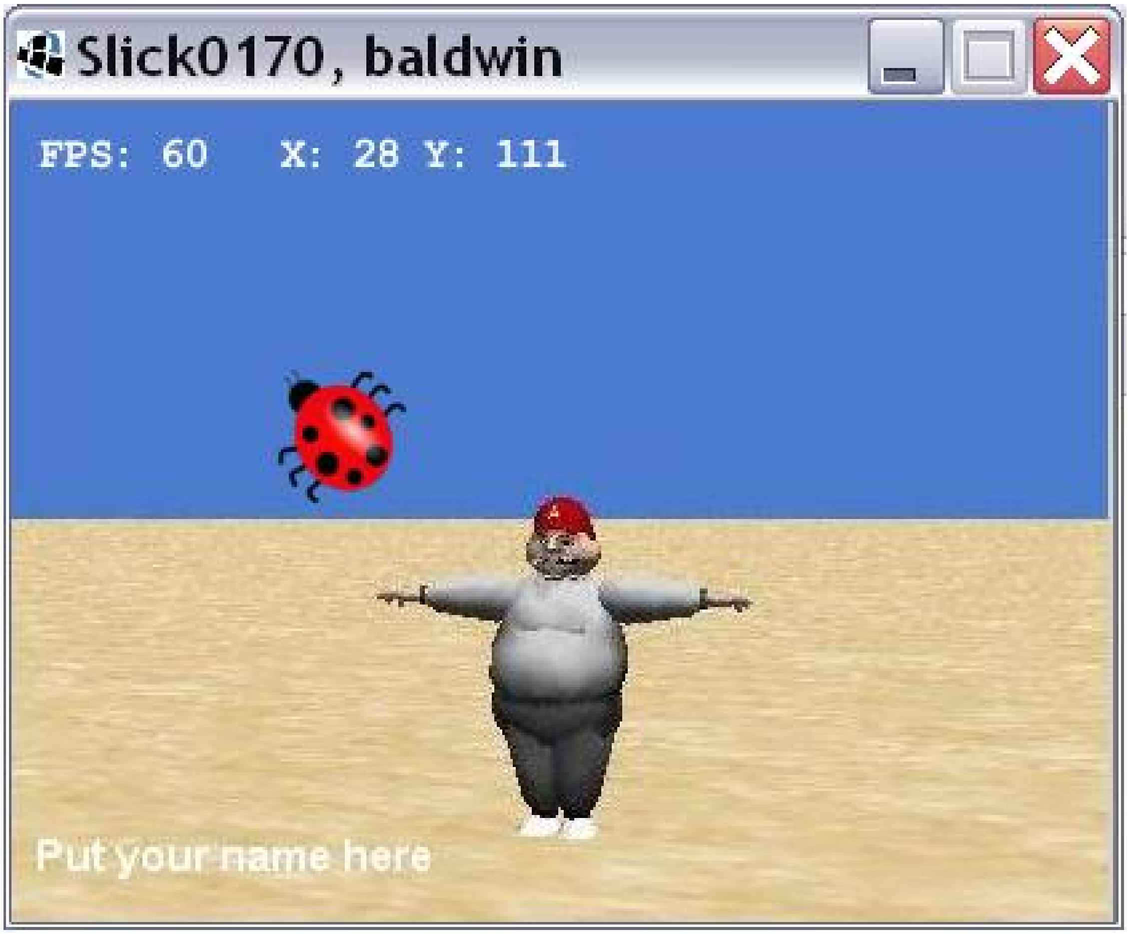
8.2.1.2 Listings

- Listing 1 (p. 159) . Beginning of the update method.
- Listing 2 (p. 160) . Test for up or down movement.
- Listing 3 (p. 161) . Test for collisions with the edges.
- Listing 4 (p. 161) . Get and save mouse coordinates.
- Listing 5 (p. 163) . Source code for the program named Slick0170.

8.3 Preview

Most games and many simulations are interactive. By that I mean that they require user input to perform according to their design.

I will present and explain a program in this module that allows the user to cause a ladybug sprite *(see Figure 1 (p. 157))* to move inside the game window by pressing the arrow keys on the keyboard or the left and right mouse buttons. *(The mouse pointer must be inside the game window for the mouse buttons to move the sprite.)*



Slick0170, baldwin
FPS: 60 X: 28 Y: 111
Put your name here

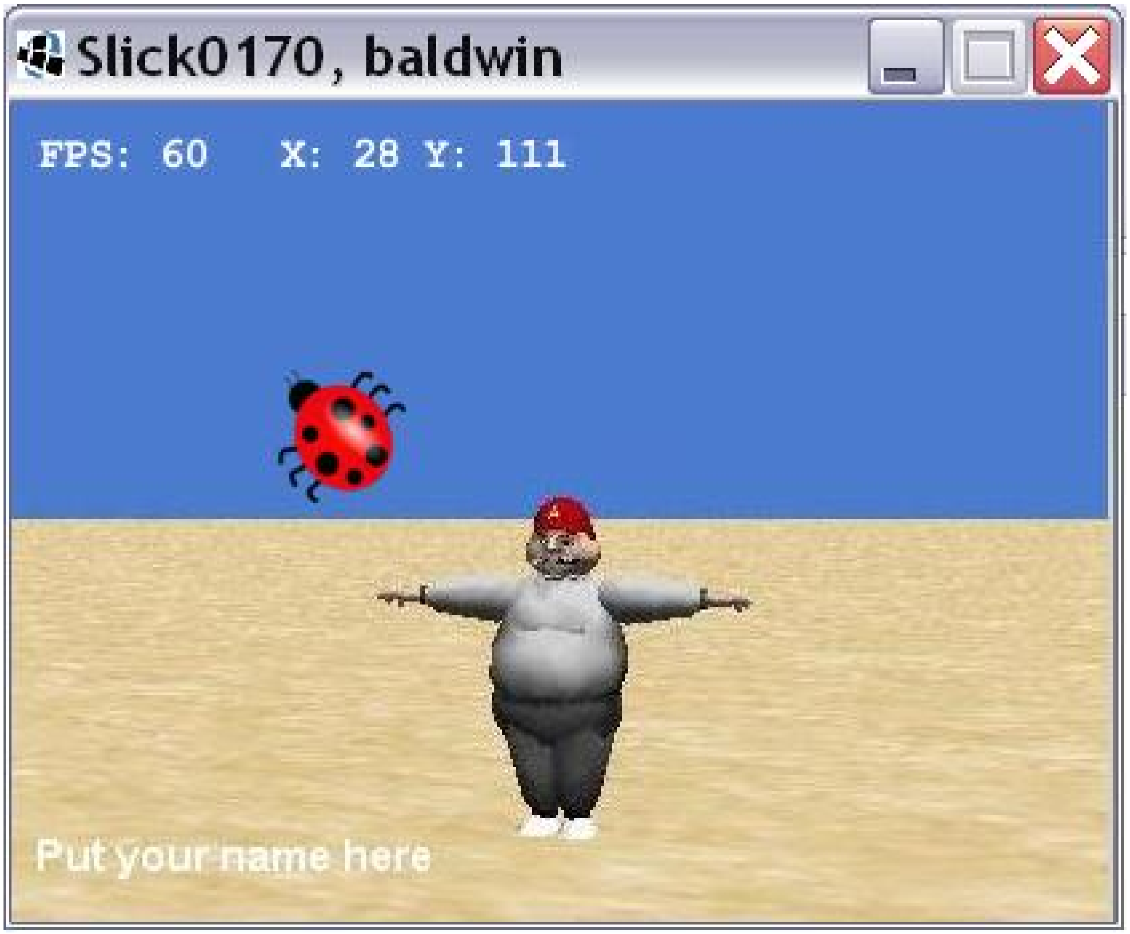

Slick0170, baldwin
FPS: 60 X: 28 Y: 111
Put your name here

Figure 1 . Output from the program named Slick0170.java.

Table 8.1

Operation

Pressing the right arrow key or the right mouse button causes the sprite to move to the right.

Pressing the left arrow key or the left mouse button causes the sprite to move to the left.

Pressing the up arrow key causes the sprite to move up, and pressing the down arrow key causes the sprite to move down.

The sprite cannot be caused to move up or down *(in this program)* by pressing mouse buttons.

What you have learned

In the previous module, you learned about using the **draw** , **drawCentered** , and **drawFlash** methods of the Image [2] class.

What you will learn

In this module, you will learn how to use the following methods of the Input [3] class to get input from the user:

- isKeyDown
- isMouseButtonDown

---

[2] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Image.html
[3] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Input.html

- getMouseX
- getMouseY

8.4 General background information

Modern computer programs fall generally in one or a combination of two categories:

- Event driven programs
- Polled programs

Analogy for an event-driven program

I like to think of event-driven programs as being somewhat analogous to the way that we normally drive our cars. When we come to a red stoplight, we remove our foot from the gas pedal, press the brake pedal to stop, and allow the motor to idle, thus consuming minimal fuel. *(If we don't have an automatic transmission, we will probably also disengage the clutch and move the gearshift leaver to the neutral position.)*

When we see that the light has turned green, we reengage the transmission if necessary, gently press the gas pedal, cause the motor to speed up, and drive through the intersection at a safe and reasonable speed.

Analogy for a polled program

I like to think of a polled program as being somewhat analogous to a car in which the gas pedal is strapped to the floor causing the motor to run at maximum rpm all the time.

In such a car, the only way to stop at a stop light would be to disengage the clutch and press the brake pedal. While the light is red, the motor would be consuming fuel at a high rate.

When the light turns green, we would reengage the clutch, speed through the intersection, and hope that we don't receive a traffic ticket.

Event-driven versus polled programs

Event-driven programs tend to idle when they have nothing to do, thus conserving computer resources. Polled programs run at full speed all of the time, thus consuming maximum computer resources.

Game and simulation programs, *(this one included)* , tend to be written as polled programs. Most other modern programs tend to be written as event-driven programs. However, you can probably write any program using either scenario, or perhaps a combination of the two.

Slick2D supports both scenarios

With regard to input, Slick2D supports both the polled and the event-driven scenarios.

Probably most user input in a Slick2D game or simulation program should be programmed using the polled scenario. However, if the user taps a key while the program is in the **render** method, the program might not recognize that the key has been tapped. If that tap is critical to the operation of the program, it might be wise to also employ the event-driven scenario to detect such critical events.

The program that I will discuss in this module is written using only the polled approach to user input. I have published numerous online tutorials that explain the use of the event-driven approach that you can find with a Google search.

Keyboard, mouse, and controller

The Slick2D Input [4] class supports input from the keyboard, the mouse, or from a game controller. I will discuss only keyboard and mouse input in this module.

8.5 Discussion and sample code

Much of this program is identical or very similar to the program named Slick0150a that I explained in the earlier module titled A first look at sprite motion, collision detection, and timing control [5] . I will explain only the code that is new and different in this module.

---

[4] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Input.html

[5] http://cnx.org/contents/c92f070e-494f-4eb1-a0b9-e36abe4359fd/

Will discuss in fragments

A complete listing of this program is provided in Listing 5 (p. 163) . Most of the code that is new and different is contained in the **update** method, which begins in Listing 1 (p. 159) .

Listing 1 . Beginning of the update method.

```
public void update(GameContainer gc, int delta)
                                  throws SlickException{

  //Get a reference to the Input object.
  Input input = gc.getInput();

  //Control horizontal bug position by pressing the
  // arrow keys or pressing the left and right mouse
  // buttons.
  if(input.isKeyDown(Input.KEY_RIGHT) ||
     input.isMouseButtonDown(Input.MOUSE_RIGHT_BUTTON)){
    bugX += xStep;
  }//end if

  if(input.isKeyDown(Input.KEY_LEFT) ||
      input.isMouseButtonDown(Input.MOUSE_LEFT_BUTTON)){
    bugX -= xStep;
  }//end if
```

Table 8.2

Get a reference to the Input object

Listing 1 (p. 159) begins by getting a saving a reference to the Input [6] object that is associated with the GameContainer [7] object. All user input can then be obtained by calling methods on the reference to the Input [8] object.

Test for right or left movement

Then Listing 1 (p. 159) uses a *logical inclusive or* operator to determine if either the *right arrow key* or the *right mouse button* *(or both)* is currently in the pressed *(or down)* state. In other words, is the user holding the *right arrow key* or the *right mouse button* down?

If the test returns true, the value of **bugX** is increased. This will cause the visual manifestation of the sprite to move to the right later when the **render** method is executed.

Then Listing 1 (p. 159) performs a similar test on the *left arrow key* and the *left mouse button* , decreasing the value of **bugX** if either test returns true.

Test for up or down movement

Listing 2 (p. 160) performs similar tests on the *up arrow key* and the *down arrow key* for the purpose of increasing or decreasing the value of **bugY** . If the value of **bugY** is changed, this will cause the sprite to move up or down later when the **render** method is executed.

---

[6] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Input.html

[7] http://slick.ninjacave.com/javadoc/org/newdawn/slick/GameContainer.html

[8] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Input.html

Listing 2 . Test for up or down movement.

```
//Control vertical bug position by pressing the arrow
// keys. Vertical bug position cannot be controlled by
// pressing mouse buttons.
if(input.isKeyDown(Input.KEY_UP)){
  bugY -= yStep;
}//end if

if(input.isKeyDown(Input.KEY_DOWN)){
  bugY += yStep;
}//end if
```

Table 8.3

No *up button* or *down button*

There is no *up button* and no *down button* on the mouse, so in this program it is not possible to move the sprite up or down by pressing mouse buttons. There are ways that such a thing could be accomplished *(such as holding down a keyboard key and pressing a mouse button)* , but they were not considered important for the purpose of this module.

Test for collisions with the edges

The code in Listing 3 (p. 161) is similar to, but simpler than the corresponding code in the earlier program named **Slick0150a** .

In this case, if the sprite collides with an edge, it simply stops moving instead of bouncing off the edge as was the case in the earlier program.

Listing 3 . Test for collisions with the edges.

```
    //Test for collisions with the sides of the game
    // window and stop moving the bug when a collision
    // occurs.
    if(bugX + bugWidth >= backgroundWidth){
      //Set the position to the right edge less the width
      // of the sprite.
      bugX = backgroundWidth - bugWidth;
    }//end if

    //Continue testing for collisions with the edges.
    if(bugX <= 0){
      bugX = 0;
    }//end if

    if(bugY + bugHeight >= backgroundHeight){
      bugY = backgroundHeight - bugHeight;
    }//end if

    if(bugY <= 0){
      bugY = 0;
    }//end if
```

Table 8.4

Get and save mouse coordinates

Listing 4 (p. 161) calls the **getMouseX** and **getMouseY** methods to get and save the coordinates of the mouse pointer when the mouse pointer is inside the game window. These values will be displayed later when the **render** method is executed as shown in Figure 1 (p. 157) .

Listing 4 . Get and save mouse coordinates.

```
    //Get and save the X and Y coordinates of the mouse
    // pointer.
    mouseX = input.getMouseX();
    mouseY = input.getMouseY();

  }//end update
```

Table 8.5

The **Input** class also provides two methods named **getAbsoluteMouseX** and **getAbsoluteMouseY** . I'm not certain how these two methods differ from the two methods that were called in Listing 4 (p. 161) , but I haven't spent any time investigating the difference.

The end of the update method

Listing 4 (p. 161) also signals the end of the **update** method and the end of this discussion.

8.6 Run the program

I encourage you to copy the code from Listing 5 (p. 163) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

8.7 Summary

In this module, you learned how to use the following methods of the Input [9] class to get user input:

- isKeyDown
- isMouseButtonDown
- getMouseX
- getMouseY

8.8 What's next?

In the next module, you will learn how to use objects of the **SpriteSheet** class and the **Animation** class to perform simple sprite sheet animation.

8.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0170: Mouse and keyboard input
- File: Slick0170.htm
- Published: 02/05/13
- Revised: 09/03/15

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

8.10 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 5 (p. 163) .

---

[9] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Input.html

Listing 5 . Source code for the program named Slick0170.

```
/*Slick0170java
Copyright 2013, R.G.Baldwin

Cause a ladybug sprite to move inside the game window by
pressing the arrow keys or the left and right mouse
buttons. The mouse pointer must be inside the game window
for the mouse buttons to move the sprite.

Right arrow or right mouse button: move right
Left arrow or left mouse button: move left
Up arrow: move up
Down arrow: move down

The program also gets and displays the X and Y
coordinates of the mouse pointer.

Much of this program is identical to the earlier program
named Slick0150a.java.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Input;

public class Slick0170 extends BasicGame{

  Image bug = null;
  Image background = null;

  float backgroundWidth;
  float backgroundHeight;

  float bugX = 100;
  float bugY = 100;
  float bugWidth;
  float bugHeight;

  float xStep = 4.0f;//horizontal step size
  float yStep = 3.0f;//vertical step size

  float bugScale = 0.75f;//drawing scale factor

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;

  //This is new code relative to Slick0150a.java
  int mouseX = 0;
```

Table 8.6

-end-

Chapter 9

Slick0180: Sprite sheet animation, part 1[1]

9.1 Table of Contents

9.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is to teach you how to use objects of the **SpriteSheet** class and the **Animation** class to perform simple sprite sheet animation.

---

[1] This content is available online at <http://cnx.org/content/m45751/1.3/>.

9.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

9.2.1.1 Figures

- Figure 1 (p. 167) . The sprite sheet.
- Figure 2 (p. 169) . Random screen shot of the animation in action.
- Figure 3 (p. 170) . Random screen shot of the animation in action.
- Figure 4 (p. 171) . Random screen shot of the animation in action.

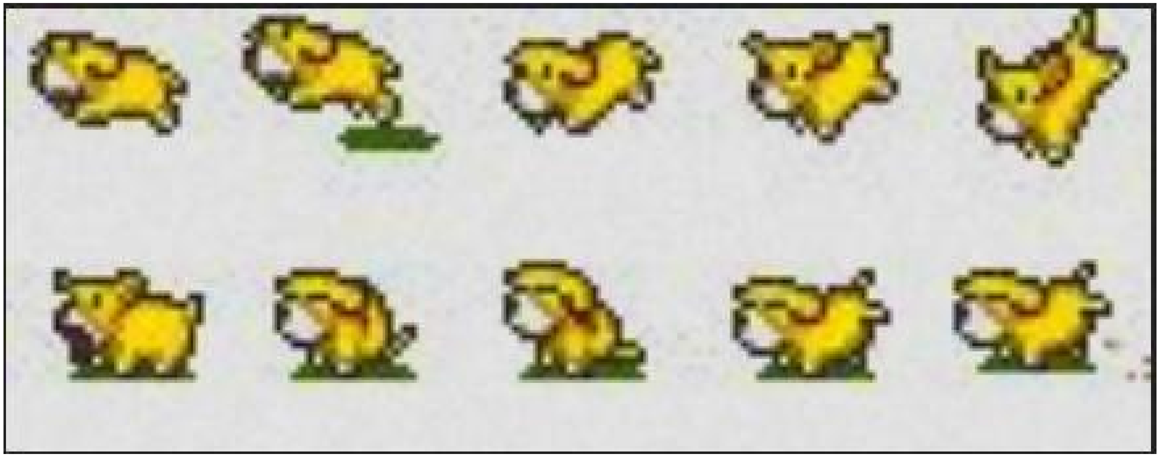
9.2.1.2 Listings

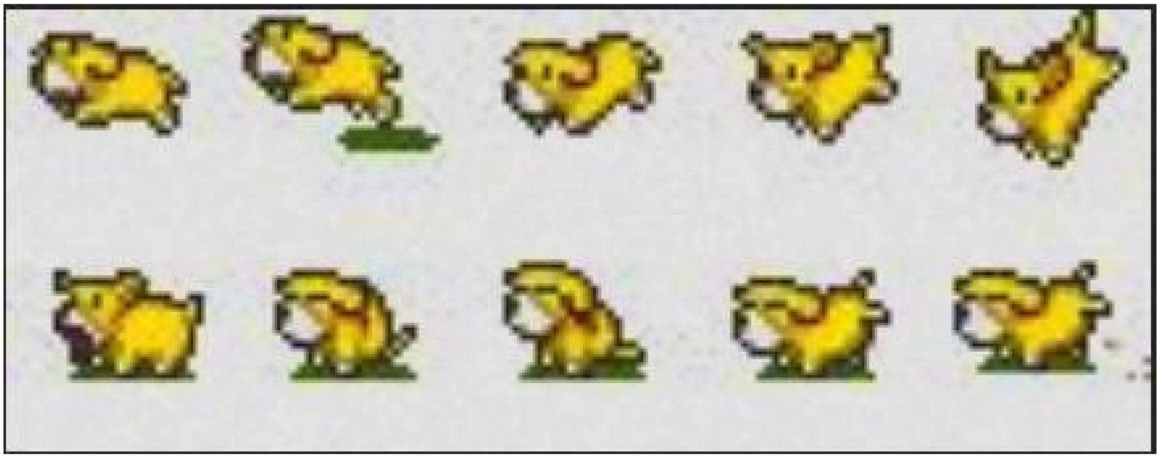
- Listing 1 (p. 174) . Beginning of the class named Slick0180.
- Listing 2 (p. 175) . Beginning of the init method.
- Listing 3 (p. 175) . Create a SpriteSheet object.
- Listing 4 (p. 176) . Create a new Animation object.
- Listing 5 (p. 176) . Set frame rate and display location.
- Listing 6 (p. 177) . The update method.
- Listing 7 (p. 177) . The render method.
- Listing 8 (p. 180) . Source code for Slick0180 .

9.3 Preview

I will present a program that uses the top row of sprites from the sprite sheet shown in Figure 1 (p. 167) along with a **SpriteSheet** object and an **Animation** object to produce an animation of a dog playing.

(Note that the overall sprite sheet image is quite small, and the image shown in Figure 1 (p. 167) was enlarged for this presentation.)





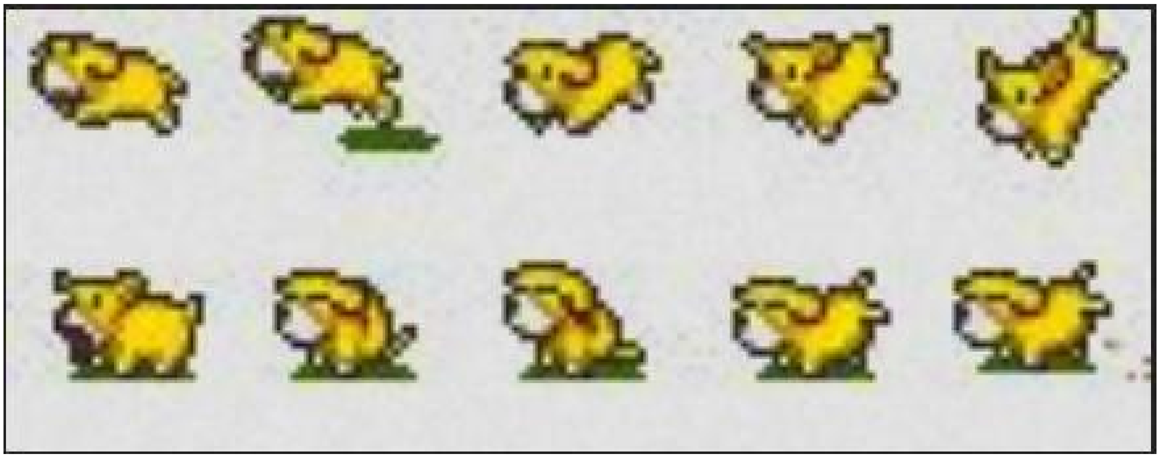


Figure 1 . The sprite sheet.

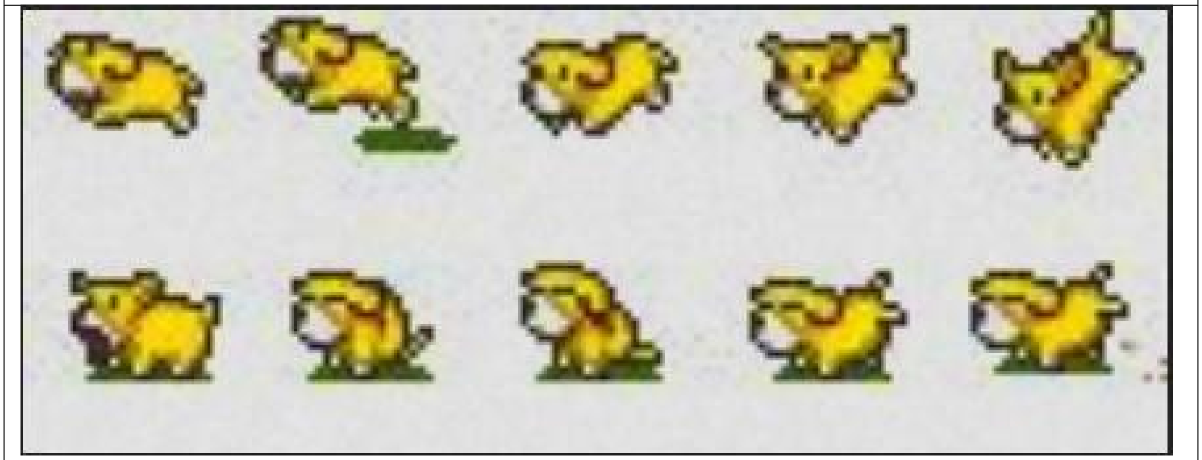


Table 9.1

Figure 2 (p. 169) , Figure 3 (p. 170) , and Figure 4 (p. 171) show random screen shots taken while the animation was running.

```
FPS: 62
totalTime = 34
```

```
FPS: 62
totalTime = 34
```

```
FPS: 62
totalTime = 34
```

Figure 2 . Random screen shot of the animation in action.

```
FPS: 62
totalTime = 34
```

Table 9.2

.

```
FPS: 62
totalTime = 77
```

```
FPS: 62
totalTime = 77
```

FPS: 62

totalTime = 77

Figure 3 . Random screen shot of the animation in action.

FPS: 62

totalTime = 77

Table 9.3

.

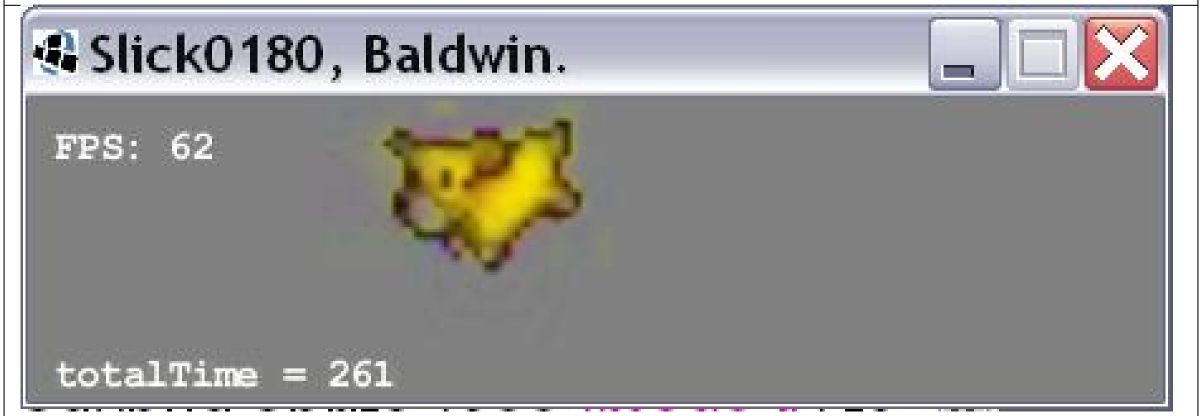
FPS: 62

totalTime = 261

```
FPS: 62

totalTime = 261
```

```
FPS: 62

totalTime = 261
```

Figure 4 . Random screen shot of the animation in action.



```
FPS: 62

totalTime = 261
```

Table 9.4

Operating characteristics

The program uses only the five sprites in the top row of Figure 1 (p. 167) . The five sprites in the bottom row are ignored. *(A program that uses all ten sprites in both rows will be presented in the next module.)*

By default, the program displays one cycle of five sprites each second. *(Each sprite is displayed for 200 milliseconds, or 0.2 seconds.)*

Clock time

As you can see in Figure 4 (p. 171) , clock time in seconds is displayed below the animation. That makes it easy to visually correlate the repetition rate with the clock.

Repetition rate is independent of the frame rate

The time that each image of the dog is displayed is independent of the frame rate. This can be demonstrated by changing the value of a variable named **targetDelta** and observing the relationship between the repetition rate and the clock. However, best results are achieved by keeping **targetDelta** less than the display time for each sprite *(* ***duration*** *)* .

What you have learned

In the previous module, you learned how to use the following methods of the Input [2] class to get user input:

- isKeyDown
- isMouseButtonDown
- getMouseX
- getMouseY

What you will learn

In this module, you will learn how to use objects of the **SpriteSheet** class and the **Animation** class to perform simple sprite sheet animation. In the next module, you will learn how to perform more complex animation.

9.4 General background information

9.4.1 The SpriteSheet class

Sprite sheets are individual sprites *(or images)* combined into a single image as shown in Figure 1 (p. 167) . Slick2D provides the **SpriteSheet** class that makes it relatively easy for you to access each of the sub-images of the sheet as separate images in your program.

The **SpriteSheet** class assumes that all the images are evenly spaced. It splits the source image into an even grid of cells and allows you to access the image in each cell as a separate image.

(Slick2D also provides the capability to work with packed sprite sheets with fewer restrictions on the organization of the sprite sheet.)

9.4.2 The Animation class

A series of images

Since well before the first Disney movies, animations have been created by displaying a series of images one after the other.

Each image *(or frame)* is typically displayed for the same amount of time, but that is not always the case, as will be demonstrated by the program in the next module.

Slick2D provides a class named **Animation** that does most of the heavy lifting in the display of an animation.

Create, populate, and configure the object

There are several different ways to create, populate, and configure an **Animation** object containing a series of images, with the same or different display durations for the images.

Displaying the images

---

[2] http://slick.ninjacave.com/javadoc/org/newdawn/slick/Input.html

By default, calling one of several overloaded **draw** methods on the **Animation** object causes it to display the sequence of images and to start over when the last image has been displayed. However, that behavior can be overridden in order to provide more customized behavior.

(It is actually more complicated that that, as you will see later in the discussion of the ***render*** *method.)*

Animations can be stopped, started and restarted *(returning to the first frame of the animation)* . The capabilities of the **Animation** class go far beyond those illustrated in this module and the next.

9.5 Discussion and sample code

9.5.1 The class named Slick0180

Will discuss in fragments

A complete listing of the program named **Slick0180** is provided in Listing 8 (p. 180) . I will break the program down and discuss it in fragments.

Listing 1 (p. 174) shows the beginning of the class named **Slick0180** down through the **main** method.

Listing 1 . Beginning of the class named Slick0180.

```
public class Slick0180 extends BasicGame{
  Image spriteSheetImage = null;

  float spriteSheetWidth;
  float spriteSheetHeight;
  int spritesPerRow = 5;
  int spritesPerColumn = 2;

  int targetDelta = 16;//msec
  int duration = 200;//time to display each sprite
  long totalTime = 0;//accumulate total time for display

  SpriteSheet spriteSheet;
  Animation animation;

  int spriteWidth;
  int spriteHeight;

  float spriteX = 0;//sprite drawing location
  float spriteY = 0;
  //----------------------------------------------------//
  public Slick0180(){
    //Call to superclass constructor is required.
    super("Slick0180, Baldwin.");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                  throws SlickException{
    AppGameContainer app = new AppGameContainer(
                      new Slick0180(),450,120,false);
    app.start();//this statement is required
  }//end main
```

Table 9.5

Instance variables

Listing 1 (p. 174) declares a number of instance variables. The purpose of these variables should become clear based on their names and their usage that I will discuss later.

The constructor and the main method

There is nothing new in the constructor and the **main** method in Listing 1 (p. 174) .

9.5.2 The init method

The **init** method begins in Listing 2 (p. 175) . The embedded comments should provide a sufficient explanation of the code in Listing 2 (p. 175) .

Listing 2 . Beginning of the init method.

```
  public void init(GameContainer gc)
                                   throws SlickException {
    spriteSheetImage = new Image("Slick0180a1.png");
    //Enlarge the sprite sheet.
    Image temp = spriteSheetImage.getScaledCopy(580,224);
    spriteSheetImage = temp;

    //Get, save, and display the width and the height
    // of the sprite sheet.
    spriteSheetWidth = spriteSheetImage.getWidth();
    spriteSheetHeight = spriteSheetImage.getHeight();

    System.out.println(
               "spriteSheetWidth: " + spriteSheetWidth);
    System.out.println(
               "spriteSheetHeight: " + spriteSheetHeight);

//Compute the width and height of the individual
// sprite images.
    spriteWidth = (int)(spriteSheetWidth/spritesPerRow);
    spriteHeight =
                (int)(spriteSheetHeight/spritesPerColumn);
```

Table 9.6

Create a SpriteSheet object

Listing 3 (p. 175) creates a new **SpriteSheet** object based on the sprite sheet image along with the width and height of the individual sprites.

Listing 3 . Create a SpriteSheet object.

```
    //Instantiate a new Spritesheet object based on the
    // width and height of the tiles.
    spriteSheet = new SpriteSheet(spriteSheetImage,
                                  spriteWidth,
                                  spriteHeight);
```

Table 9.7

Create a new Animation object

Listing 4 (p. 176) creates a new **Animation** object that will process the **SpriteSheet** object instantiated in Listing 3 (p. 175) .

Listing 4 . Create a new Animation object.

```
    //Create a new animation based on a selection of
    // sprites from the sprite sheet.
    animation = new Animation(spriteSheet,
                              0,//first column
                              0,//first row
                              4,//last column
                              0,//last row
                              true,//horizontal
                              duration,//display time
                              true//autoupdate
                              );
```

Table 9.8

Constructor parameters

Obviously, the first parameter to the constructor for the **Animation** class specifies the **SpriteSheet** object.

The second and third parameters specify that the first image in the sequence should be the top-left image in Figure 1 (p. 167) .

The fourth and fifth parameters specify that the last image in the sequence should be the top-right image in Figure 1 (p. 167) .

The *true* value for the sixth parameter specifies that the images should be scanned horizontally.

The *duration* value in the seventh parameter specifies that each image should be displayed for 200 milliseconds.

The *true* value for the last parameter specifies that the display should continue cycling through the images until the animation is stopped.

Set frame rate and display location

The code is Listing 5 (p. 176) sets the frame rate and specifies the drawing location. The drawing location is the location within the game window where the sprite will be displayed.

Listing 5 . Set frame rate and display location.

```
    gc.setShowFPS(true);//show FPS
    ////set frame rate
    gc.setTargetFrameRate((int)(1000/targetDelta));

    //Set drawing location. This is the location within
    // the game window where the sprite will be displayed.
    spriteX = spriteWidth;
    spriteY = 0;
  }//end init
```

Table 9.9

9.5.3 The update method

The **update** method is shown in Listing 6 (p. 177) . As indicated in the comments, the computation of **totalTime** in the method has nothing to do with the animation. Instead, it is used to display the clock time as shown in Figure 2 (p. 169) .

Listing 6 . The update method.

```
  public void update(GameContainer gc, int delta)
                                  throws SlickException{
    //Note that the following is for clock time display
    // only. It does not effect the animation.
    totalTime += delta;//update total time accumulator
  }//end update
```

Table 9.10

9.5.4 The render method

The **render** method is shown in Listing 7 (p. 177) . The only thing that is new here is the call to the **draw** method on the **Animation** object.

Listing 7 . The render method.

```
  public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
    g.setDrawMode(g.MODE_NORMAL);
    g.setBackground(Color.gray);

    //Draw the currently selected animation image at the
    // specified location
    animation.draw(spriteX,spriteY);

    g.drawString("totalTime = "+totalTime/1000,10f,100f);
  }//end render

}//end class Slick0180
```

Table 9.11

Powerful behavior

The behavior of the **Animation** object and its **draw** method is very powerful.

- The **Animation** object keeps track of the scheduling requirements of the animation, such as which image should be displayed at the current time.

- Calling the **draw** method on the **Animation** object causes that image to actually be displayed.

The image display schedule being managed by the **Animation** object is independent of the frame rate.

The **Animation** object does its thing, and the **render** method does its thing virtually independent of one another. When the **render** method decides that it is time to display an animation image, it calls the **draw** method on the **Animation** object.

The **Animation** object delivers the image that is scheduled for display at that point in time according to the predetermined animation schedule and the **draw** method causes the image to be displayed.

Overloaded draw methods

There are several overloaded versions of the **draw** method including versions to filter the colors and to change the width and height of the displayed image.

Best results

Now you know why, as mentioned earlier (p. 172) , best results are achieved by keeping **targetDelta** less than the display time (***duration***) for each sprite. If **targetDelta** is greater than the **duration** , some images will be skipped and not displayed in the proper sequence.

For example, if the **Animation** object is switching from one image to the next every 0.10 second, but the **draw** method is only being called every 0.13 seconds, some of the images in the sequence won't be displayed and the quality of the animation will probably be poor.

However, this is also dependent on the amount of change from one image to the next. If the change from one image to the next is small, then skipping an occasional image might not matter that much.

9.6 Run the program

I encourage you to copy the code from Listing 8 (p. 180) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

9.7 Summary

In this module, you learned how to use objects of the **SpriteSheet** class and the **Animation** class to perform simple sprite sheet animation.

9.8 What's next?

In the next module, you will learn how to use objects of the **SpriteSheet** class and the **Animation** class to perform more complex sprite sheet animations than was the case in this module.

9.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0180: Sprite sheet animation, part 1
- File: Slick0180.htm
- Published: 02/05/13
- Revised: 06/08/15 for 64-bit

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

9.10 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 8 (p. 180) .

Listing 8 . Source code for Slick0180 .

```
/*Slick0180.java
Copyright 2013, R.G.Baldwin

Uses one row of sprites from a sprite sheet along with
an Animation object to draw an animation of a dog playing.

By default, the program displays one cycle of five
sprites per second. Clock time is displayed below the
animation.

The time that each image of the dog is displayed is
independent of the frame rate. Demonstrate this by
changing the value of targetDelta and observing the
relationship between the animation times and the clock.

For best results, keep the targetDelta less than the
display time for each sprite (duration).

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.SpriteSheet;
import org.newdawn.slick.Animation;
import org.newdawn.slick.Color;

public class Slick0180 extends BasicGame{
  Image spriteSheetImage = null;

  float spriteSheetWidth;
  float spriteSheetHeight;
  int spritesPerRow = 5;
  int spritesPerColumn = 2;

  int targetDelta = 16;//msec
  int duration = 200;//time to display each sprite
  long totalTime = 0;//accumulate total time for display

  SpriteSheet spriteSheet;
  Animation animation;

  int spriteWidth;
  int spriteHeight;

  float spriteX = 0;//sprite drawing location
  float spriteY = 0;
  //----------------------------------------------------//
  public Slick0180(){
    //Call to superclass constructor is required.
```

Table 9.12

-end-

Chapter 10

Slick0190: Sprite sheet animation, part 2[1]

10.1 Table of Contents

10.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is to teach you how to use objects of the **SpriteSheet** class and the **Animation** class to perform more complex sprite sheet animations than was the case in the earlier module titled Slick0180: Sprite sheet animation, part 1 [2] .

---

[1] This content is available online at <http://cnx.org/content/m45753/1.3/>.

[2] http://cnx.org/contents/0fc6c084-4f60-4497-a2d7-802a396bc985

10.2.1 Viewing tip

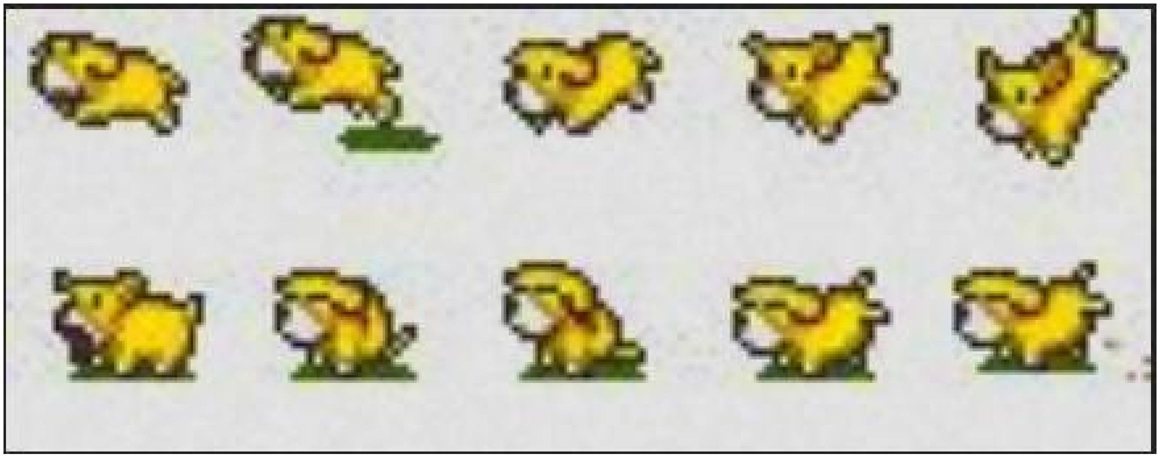
I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

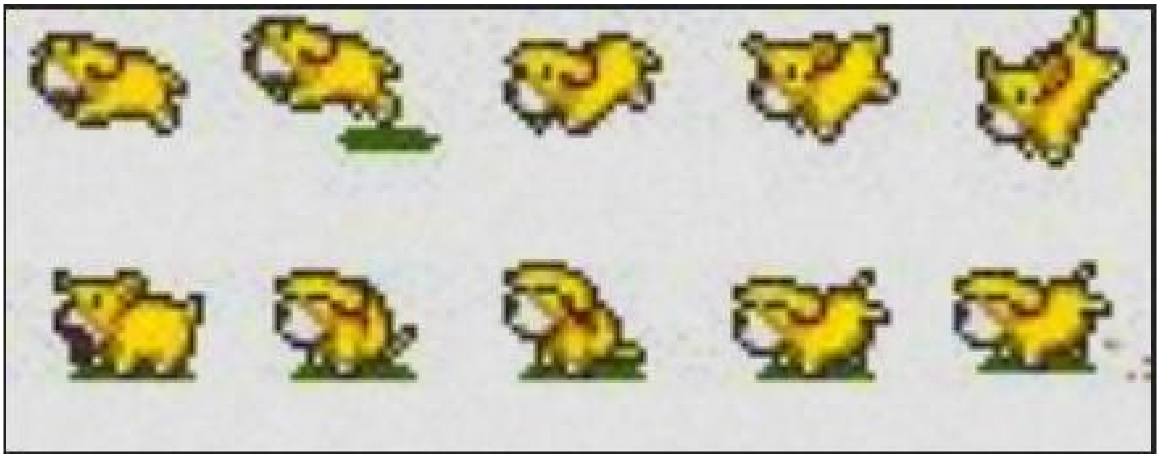
10.2.1.1 Figures

10.2.1.2 Listings

10.3 Preview

I will present and explain a program that uses both rows of sprites from the sprite sheet shown in Figure 1 (p. 185) . The program uses a **SpriteSheet** object and an **Animation** object to produce an animation of a dog playing and answering nature's call. *(Note that the overall sprite sheet image is quite small, and the image shown in Figure 1 (p. 185) was enlarged for this presentation.)*





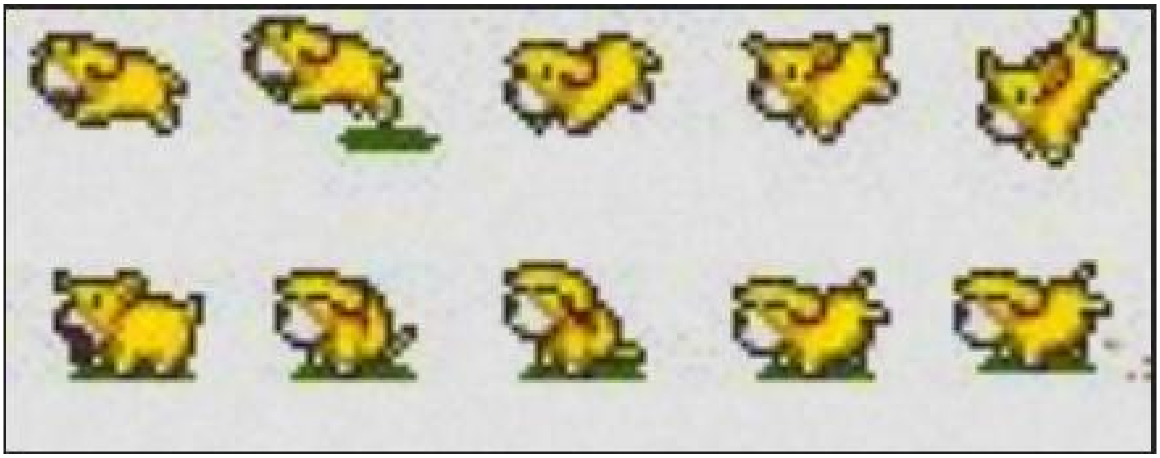


Figure 1 . The sprite sheet.

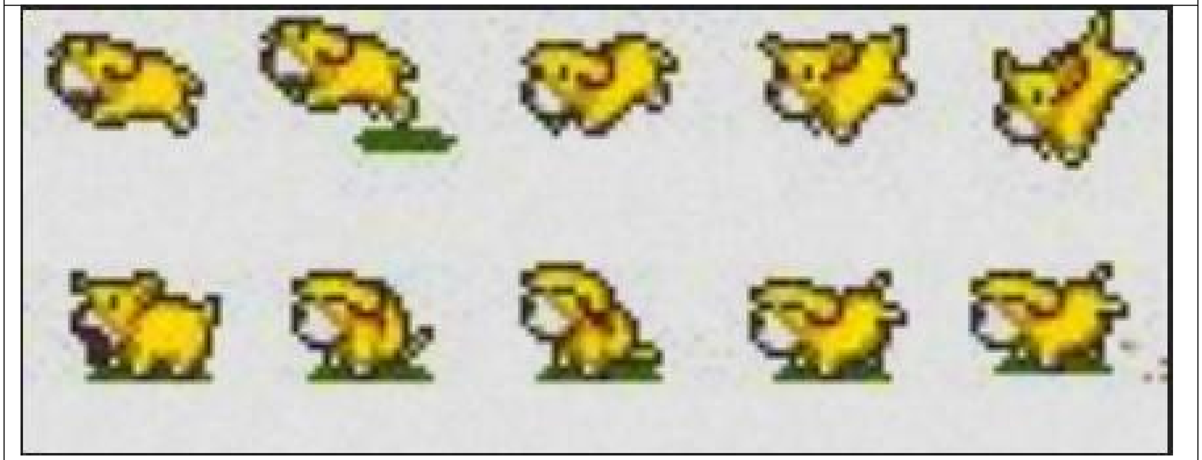


Table 10.1

Description of the animation

This animation begins with the sprite running from left to right across the game window. Then the sprite stops on the right side of the game window and answers nature's call. Although it isn't shown here, the sprite turns and faces left during that process. Then the sprite runs from right to left across the game window. This pattern repeats for as long as the program runs, and is illustrated by the three screen shots that follow.

Sprite running to the right

Figure 2 (p. 187) shows the sprite running from left to right. This is a flipped version of one of the images in the top row of Figure 1 (p. 185) .

Figure 2 . Sprite running to the right.

Table 10.2

Sprite answering nature's call

Figure 3 (p. 188) shows the sprite answering nature's call. This is a flipped version of one of images from the bottom row of Figure 1 (p. 185) .

Figure 3 . Sprite answering nature's call.


Table 10.3

Sprite running to the left

Figure 4 (p. 189) shows the sprite running from right to left. This is one of the images from the top row of Figure 1 (p. 185) .

FPS: 63

FPS: 63

Figure 4 . Sprite running to the left.
   FPS: 63

Table 10.4

What you have learned

In the previous module [3], you learned how to use objects of the **SpriteSheet** class and the **Animation** class to perform simple sprite sheet animation.

What you will learn

In this module, you will learn how to use objects of the **SpriteSheet** class and the **Animation** class to perform more complex sprite sheet animations than was in the previous module.

---

[3] http://cnx.org/contents/0fc6c084-4f60-4497-a2d7-802a396bc985

10.4 General background information

10.4.1 The SpriteSheet class

There isn't much that's new in this module regarding the **SpriteSheet** class. The program instantiates and populates a **SpriteSheet** object and used the images stored in that object to populate an **Animation** object.

The **SpriteSheet** object is used in a different way than was the case in the previous module, but that will be explained in conjunction with populating the **Animation** object.

10.4.2 The Animation class

An **Animation** object is populated in a significantly different way in this module than in the previous module.

In the previous module, a **SpriteSheet** object's reference was passed to the **Animation** constructor along with a specification of the images to be extracted from the sprite sheet and the amount of time that each image should be displayed. The **Animation** constructor extracted the images from the sprite sheet and populated the new **Animation** object automatically. Among other restrictions, it was necessary that each image be displayed for the same amount of time.

One image at a time

In this module, an empty **Animation** object is instantiated and then populated one image at a time. *(Of course, loops are used to make that process easier.)* Among other things, this makes it possible to:

- Use multiple copies of the individual images on the sprite sheet
- Use flipped versions of the images on the sprite sheet
- Specify different display times for the different images on the sprite sheet

Different animation rates

For example, the display times for the images from the bottom row of Figure 1 (p. 185) are four times greater than the display times for the images from the top row. Thus, the animation slows down when the sprite stops to answer nature's call on the right side of the game window then speeds up again when the sprite starts running from right to left.

10.5 Discussion and sample code

10.5.1 The class named Slick0190

Will discuss in fragments

A complete listing of the program named **Slick0190** is provided in Listing 11 (p. 199) . I will break the program down and discuss it in fragments.

Listing 1 (p. 191) shows the beginning of the class named **Slick0190** down through the **main** method. There is nothing in Listing 1 (p. 191) that should require an explanation beyond the embedded comments. However, it is worth noting that unlike the previous module, Listing 1 (p. 191) instantiates a new empty object of the class **Animation** and saves its reference in the instance variable named **animation** . This object will be populated with images by the **init** method later.

Listing 1 . Beginning of the class named Slick0190.

```
public class Slick0190 extends BasicGame{
  Image spriteSheetImage = null;

  float spriteSheetWidth;
  float spriteSheetHeight;
  int spritesPerRow = 5;
  int spritesPerColumn = 2;
  int spriteWidth;
  int spriteHeight;

  int targetDelta = 16;//msec
  SpriteSheet spriteSheet;
  Animation animation = new Animation();

  //Horizontal and vertical drawing coordinates.
  float spriteX = 0;
  float spriteY = 0;

  //----------------------------------------------------//
  public Slick0190(){
    //Call to superclass constructor is required.
    super("Slick0190, Baldwin.");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                  throws SlickException{
    AppGameContainer app = new AppGameContainer(
                        new Slick0190(),450,120,false);
    app.start();//this statement is required
  }//end main
```

Table 10.5

10.5.2 The init method

Most of the new code in this program is contained in the **init** method, which begins in Listing 2 (p. 192) . However, the code in Listing 2 (p. 192) is not new and should not require an explanation beyond the embedded comments.

Listing 2 . Beginning of the init method.

continued on next page

```
public void init(GameContainer gc)
                                throws SlickException {
  //Create a SpriteSheet object
  spriteSheetImage = new Image("Slick0190a1.png");
  //Enlarge the sprite sheet.
  Image temp = spriteSheetImage.getScaledCopy(580,224);
  spriteSheetImage = temp;

  spriteSheetWidth = spriteSheetImage.getWidth();
  spriteSheetHeight = spriteSheetImage.getHeight();
  spriteWidth = (int)(spriteSheetWidth/spritesPerRow);
  spriteHeight =
              (int)(spriteSheetHeight/spritesPerColumn);

  //Instantiate a new spriteSheet object based on the
  // width and height of the individual tiles on the
  // sheet.
  spriteSheet = new SpriteSheet(spriteSheetImage,
                               spriteWidth,
                               spriteHeight);
```

Table 10.6

Begin populating the Animation object.

The code in Listing 3 (p. 192) begins the process of populating the Animation object using images extracted from the sprite sheet shown in Figure 1 (p. 185) .

Listing 3 . Begin populating the Animation object.

```
  //Populate the Animation object
  //Begin by adding four sets of five sprites from the
  // top row with the images flipped to face right.
  for(int cntr = 0;cntr < 4;cntr++){
    for(int cnt = 0;cnt < 5;cnt++){
      animation.addFrame(
            spriteSheet.getSprite(cnt,0).getFlippedCopy(
                                  true,false),100);
    }//end inner loop
  }//end outer loop
```

Table 10.7

The inner loop in Listing 3 (p. 192) calls the **getSprite** method of the **SpriteSheet** five times in succession to extract each of the images in the top row in Figure 1 (p. 185) .

Flip the images

The sprites represented by those five images are facing the wrong direction. Therefore, Listing 3 (p. 192) calls the **getFlippedCopy** method of the **Image** class to flip the images horizontally before adding them to the contents of the **Animation** object.

The addFrame method

The version of the **addFrame** method used in Listing 3 (p. 192) requires two parameters:

- A reference to an **Image** object
- The time duration in milliseconds that the image should be displayed in the ongoing animation process

The display time duration

The time duration was set to 100 milliseconds for each of the images from the top row of Figure 1 (p. 185) . However, that is not a requirement. You can set a different time duration for every image that you add to an **Animation** object if that is required to meet your needs.

Repeat the process

The outer loop in Listing 3 (p. 192) causes the process to be repeated four times. Therefore, when the code in Listing 3 (p. 192) finishes executing, the **Animation** object contains 20 images made up of four set of the five images in the top row of Figure 1 (p. 185) . These 20 images will be used to cause the sprite to run and jump from left to right across the game window.

Add images from the bottom row of Figure 1

Listing 4 (p. 193) uses essentially the same logic *(broken into two nested loops)* to add four sets of images from the bottom row of Figure 1 (p. 185) .

Listing 4 . Add images from the bottom row of Figure 1.

```
//Add two sets of five sprites from the bottom row
// with the images flipped to face right.
for(int cntr = 0;cntr < 2;cntr++){
  for(int cnt = 0;cnt < 5;cnt++){
    animation.addFrame(
          spriteSheet.getSprite(cnt,1).getFlippedCopy(
                                     true,false),400);
  }//end inner loop
}//end outer loop

//Add two sets of five sprites from the bottom row
// with the images facing left.
for(int cntr = 0;cntr < 2;cntr++){
  for(int cnt = 0;cnt < 5;cnt++){
    animation.addFrame(
                    spriteSheet.getSprite(cnt,1),400);
  }//end inner loop
}//end outer loop
```

Table 10.8

Flip to face the right

The first two set of images are flipped to face to the right. The last two sets of images are not flipped. This is the reason for breaking this process into a pair of nested loops instead of using a single nested loop.

If you watch the animation carefully, you will see that the sprite begins answering nature's call facing to the right. Half way through answering nature's call, the sprite spins around and faces to the left. After that, it runs across the screen from right to left.

The display time duration

Note that the specified time duration for these twenty images is 400 milliseconds. Two major changes occur during this part of the animation *(relative to the previous part)* :

- The sprite does not move horizontally while these 20 images are being displayed
- Each of the 20 images is displayed four times as long as when the sprite is running.

Finish populating the animation object

Listing 5 (p. 194) finishes populating the Animation object by adding four more sets of the five images in the top row of Figure 1 (p. 185) . In this case, however, the images are not flipped. Therefore, they are used to cause the sprite to run from right to left across the game window.

Listing 5 . Finish populating the animation object.

```
    //Add four sets of five sprites from the top row with
    // the images facing left
    for(int cntr = 0;cntr < 4;cntr++){
      for(int cnt = 0;cnt < 5;cnt++){
        animation.addFrame(
                        spriteSheet.getSprite(cnt,0),100);
      }//end for loop
    }//end for loop

    gc.setShowFPS(true);//display FPS
    //Set frame rate
    gc.setTargetFrameRate((int)(1000/targetDelta));

  }//end init
```

Table 10.9

Listing 5 (p. 194) also takes care of some common administrative details at the end, signaling the end of the **init** method.

10.5.3 The update method

The **update** method begins in Listing 6 (p. 195) . The primary purpose of **update** method in this program is to control the physical placement of each sprite when it is displayed.

Listing 6 . Beginning of the update method.

continued on next page

```
public void update(GameContainer gc, int delta)
                                  throws SlickException{

  int stepSize = 15;//Distance the sprite moves
  int frame = animation.getFrame();//animation frame
  int oneThird = animation.getFrameCount()/3;
```

Table 10.10

The getFrameCount method

On the basis of the previous discussion, we already know that the **Animation** object contains a sequence of 60 images or frames. Rather than to rely on that knowledge, however, Listing 6 (p. 195) calls the **getFrameCount** method on the **Animation** object to determine the number frames in the object.

Three groups of sprites

That value is divided by 3 and saved in the variable named **oneThird** . The logic that follows is based on dividing the sprites into three equal size groups and processing them differently depending on whether they fall in the first, second, or third group.

Get the current frame number

Listing 6 (p. 195) calls the **getFrame** method on the **Animation** object to determine which frame should be displayed by this iteration of the game loop. That value is saved in the variable named **frame** .

Compute the display location

Having determined which image is to be displayed, we must then compute the horizontal position within the game window at which to display the image.

The display logic

If the current frame is within the first third, the display position assigned to the frame should make it appear that the sprite is running from left to right across the game window. Therefore, the horizontal display coordinate values for the sprites in this group should be proportional to the frame number from 0 through 19.

If the current frame is within the second third, the horizontal display coordinate should not change. *(The sprite should be stationary.)* The sprites in this group are all intended to be displayed in the same location.

If the current frame is within the third group, things are a little more complicated. The horizontal display coordinate values assigned to the sprites should make it appear that the sprite is running from right to left across the game window. Therefore, the coordinate value should be equal to the rightmost excursion less a value that is proportional to the frame number, after adjusting the frame number to account for the 20 frames during which the sprites were stationary.

This process begins in Listing 7 (p. 195) .

Listing 7 . Compute display locations for first 20 frames.

```
  if(frame < oneThird){
    //Sprite is moving to the right. Compute the new
    // position.
    spriteX = frame*stepSize;
```

Table 10.11

Figures in the first group

Listing 7 (p. 195) tests to determine if the current frame is in the first third. If so, it computes a horizontal position coordinate value as the product of the frame number and the **stepSize** in pixels, which was defined in Listing 6 (p. 195) .

Figures in the middle group

The process of computing the horizontal position coordinate value continues in Listing 8 (p. 196) .

Listing 8 . Don't change position for middle group of sprite images.

```
  }else if(frame < 2*oneThird){
    //Sprite is stationary. Don't change position
```

Table 10.12

Listing 8 (p. 196) tests to determine if the current frame is in the middle group. If so, it causes the **update** method to return without changing the horizontal position coordinate value, thus allowing the sprite to remain in a stationary position.

Figures in the third group

The horizontal position values computed in Listing 9 (p. 196) make it appear that the sprite is running from right to left across the game window.

This is one of those opportune times when it is appropriate to say that I will leave it as an exercise for the student to dust off their high-school algebra books and figure out how the code in Listing 9 (p. 196) achieves the desired result.

Listing 9 . Run from right to left.

```
  }else if(frame < 3*oneThird){
    //Cause the sprite to turn around and start
    // moving to the left toward the starting point.
    //Reduce frame count by number of frames during
    // which the sprite wasn't moving.
    frame -= oneThird;
    //Compute the new position.
    spriteX = (2*oneThird - frame)*stepSize;
  }//end else if

}//end update
```

Table 10.13

Listing 9 (p. 196) also signals the end of the **update** method.

10.5.4 The render method

Now it's time to realize the benefit of all of the hard work that went into planning for and writing the **code** in the **init** and **update** methods. The code in the **render** method, which is shown in Listing 10 (p. 197) , is almost trivial.

Listing 10 . The render method.

```
  public void render(GameContainer gc, Graphics g)
                                  throws SlickException{
    g.setDrawMode(g.MODE_NORMAL);
    g.setBackground(Color.gray);
    animation.draw(spriteX,spriteY);

  }//end render

}//end class Slick0190
```

Table 10.14

There is nothing left for the **render** method to do other than to send a message to the **Animation** object once during each iteration of the game loop asking it to draw the current frame.

Listing 10 (p. 197) signals the end of the **render** method, the end of the **Slick0190** class, and the end of the program.

10.6 Run the program

I encourage you to copy the code from Listing 11 (p. 199) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

10.7 Summary

In this module, you learned how to use objects of the **SpriteSheet** class and the **Animation** class to perform more complex sprite sheet animations than was the case in the previous module.

10.8 What's next?

In the next module, you will learn how to develop a sprite class from which you can instantiate and animate swarms of sprite objects.

10.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0190-Sprite sheet animation, part 2
- File: Slick0190.htm
- Published: 02/06/13
- Revised: 09/03/15

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

10.10 Complete program listing

A complete listing of the program discussed in this module is provided in Listing 11 (p. 199) .

Listing 11 . Source code for Slick0190.

```
/*Slick0190.java
Copyright 2013, R.G.Baldwin

Fairly complex animation using a sprite sheet.

Sprite moves to right during first third of the
animation. Sprite remains stationary during second third
of the animation. Sprite moves to the left back to the
starting point during the last third of the animation.

Much more complicated than Slick0180 for several reasons
including the following:

The sprite is moved horizontally during a portion but not
all of the animation. Movement must be synchronized with
the animation frame counter.

The sprite sheet contains only images of the dog facing
to the left. However, images of the dog facing to the
right are also required. This requires that each image
on the sprite sheet be extracted and flipped horizontally
before being fed to the Animation object for half of
the animation sequence.

The display duration for images from the first row is
shorter than for images from the second row.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.SpriteSheet;
import org.newdawn.slick.Animation;
import org.newdawn.slick.Color;

public class Slick0190 extends BasicGame{
  Image spriteSheetImage = null;

  float spriteSheetWidth;
  float spriteSheetHeight;
  int spritesPerRow = 5;
  int spritesPerColumn = 2;
  int spriteWidth;
  int spriteHeight;
```

```
  int targetDelta = 16;//msec
  SpriteSheet spriteSheet;
  Animation animation = new Animation();
```

Table 10.15

-end-

Chapter 11

Slick0200: Developing a sprite class[1]

11.1 Table of Contents

11.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is to teach you how to develop a sprite class *(see Sprite01 (p. 223))* from which you can instantiate and animate swarms of sprite objects.

11.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

---

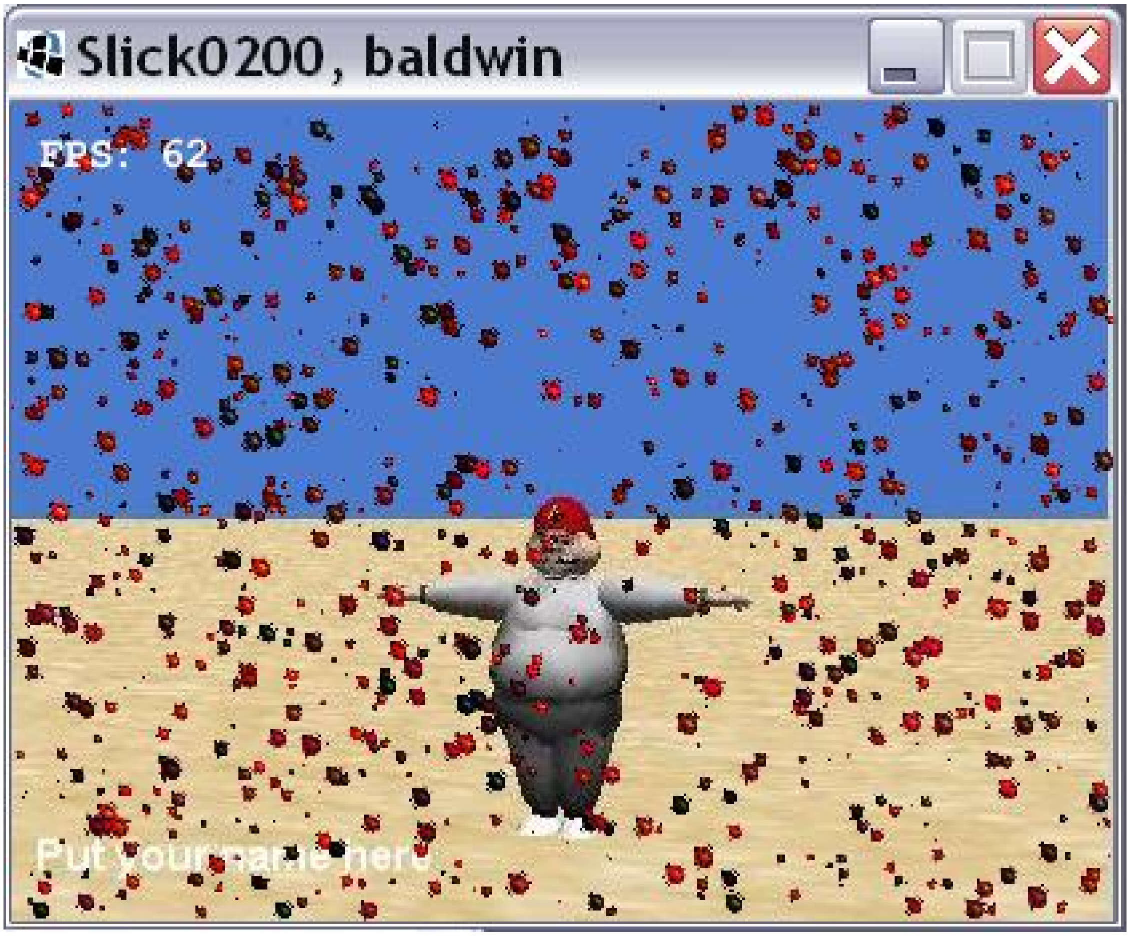
[1] This content is available online at <http://cnx.org/content/m45754/1.3/>.

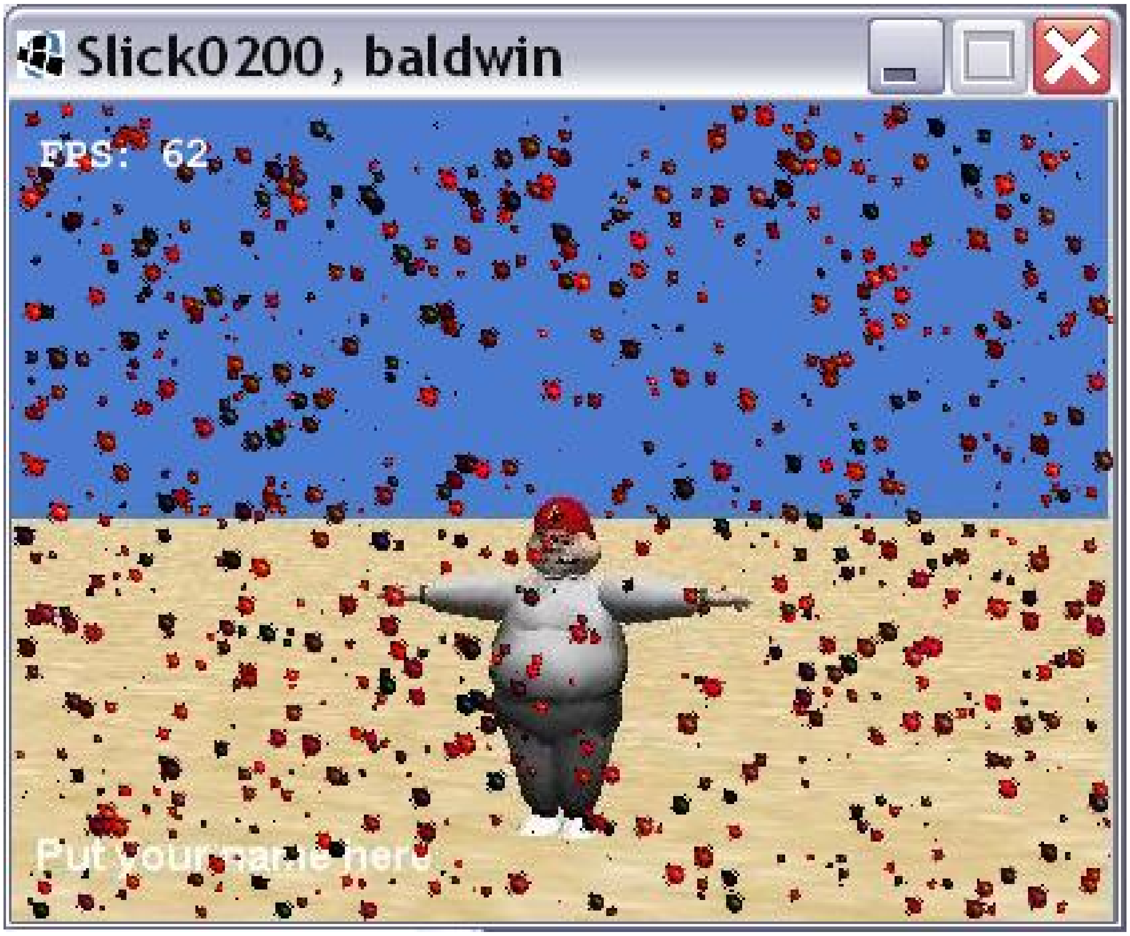
11.2.1.1 Figures

11.2.1.2 Listings

11.3 Preview

I will present and explain a program that uses a class named **Sprite01** *(see Listing 11 (p. 223))* to produce an animation of 1000 ladybug sprite objects flying around inside the game window as shown in Figure 1 (p. 206) .


Slick0200, baldwin
FPS: 62
Put your name here


Slick0200, baldwin
FPS: 62
Put your name here

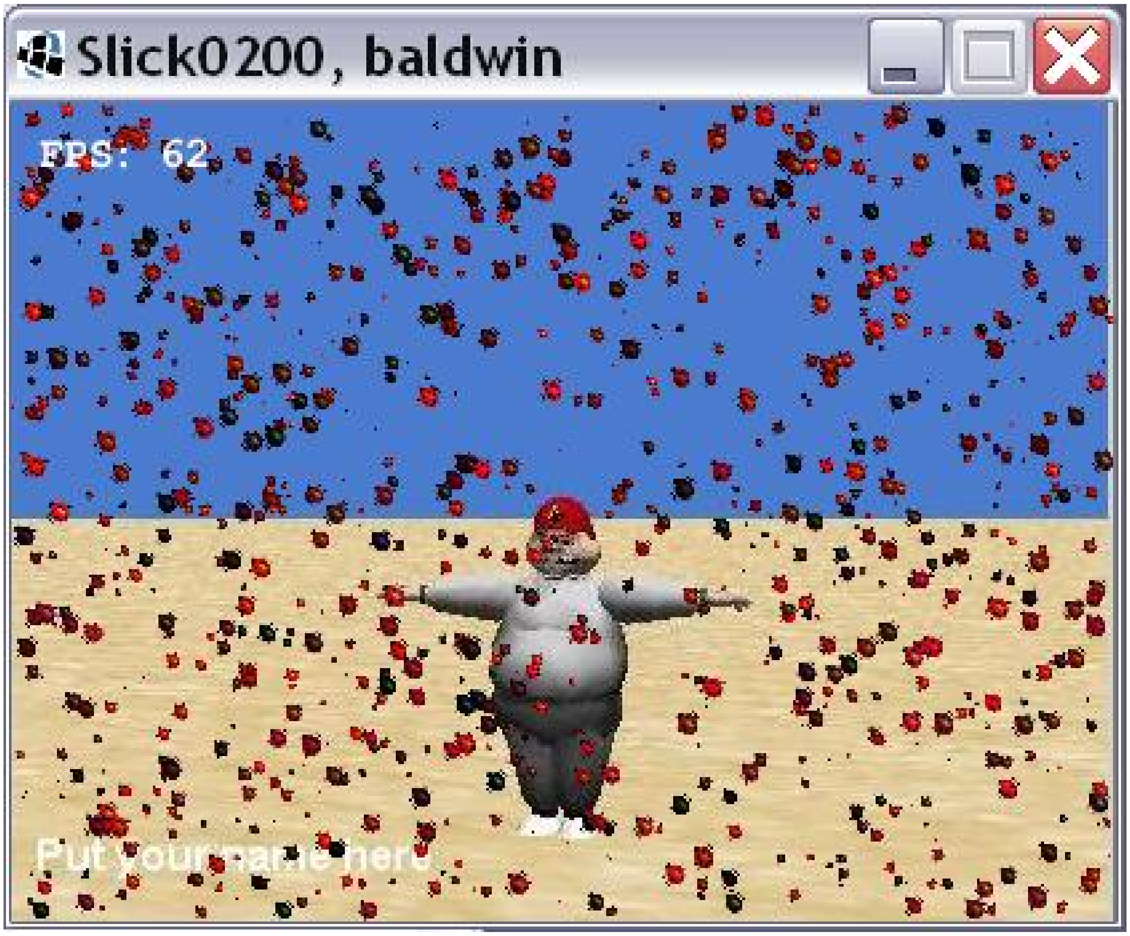

Slick0200, baldwin
FPS: 62
Put your name here

Figure 1 . Graphic output from program named Slick0200.

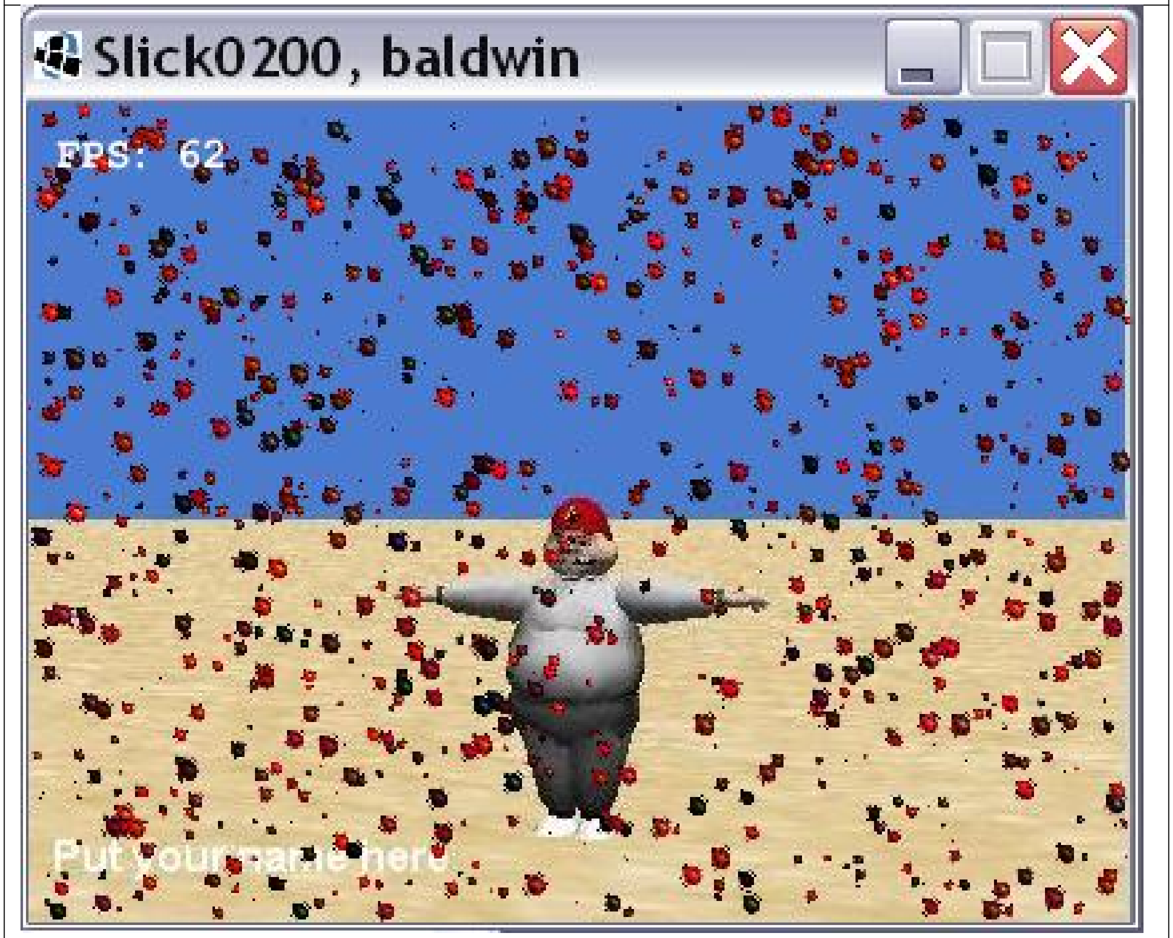


Table 11.1

The frame rate

As you can see from the text in the upper-left corner of Figure 1 (p. 206) , the program is running at 62 frames per second. My rather old desktop computer can maintain this frame rate up to about 7000 sprite objects. Beyond that, it can no longer handle the computing load and the frame rate begins to decrease.

What you have learned

In the previous module, you learned how to use objects of the **SpriteSheet** class and the **Animation** class to perform relatively complex sprite sheet animations.

What you will learn

In this module, you will learn how to develop a sprite class from which you can instantiate and animate swarms of sprite objects. In the next two modules, you will learn how to put that class to work.

11.4 General background information

While the Slick2D library provides many useful classes, there is nothing to stop you from developing your own classes to work in combination with the Slick2D library classes. That is the thrust of this module.

In an **earlier module** titled Slick0150: A first look at sprite motion, collision detection, and timing control [2] , you learned how to cause a single sprite to bounce around inside the game window as shown in Figure 2 (p. 210) .

---

[2] http://cnx.org/contents/c92f070e-494f-4eb1-a0b9-e36abe4359fd


Slick0150a, baldwin
FPS: 62 traversalTime: 3015
Put your name here

Slick0150a, baldwin
FPS: 62   traversalTime: 3015
Put your name here

Figure 2 . Graphic output from the earlier program.

Slick0150a, baldwin

FPS: 62   traversalTime: 3015

Put your name here

Table 11.2

Adding many more sprites would have been difficult

While it would have been possible to add more sprites to the animation by expanding the code used in that program, the code would have quickly gotten out of hand without the use of a sprite class and sprite objects. *(To use the common jargon, that program architecture was not very scalable.)*

Encapsulate complexity in a class

Basically, this program solves that problem by encapsulating many of the properties and methods that are useful for manipulating sprites into a class from which sprite objects can be instantiated. Most of the complexity is encapsulated in the class and thereby removed from the program that uses objects of the class.

The scenario

This program shows a baseball coach (*Figure 1* (p. 206)) being attacked by a swarm of vicious ladybug sprites. *(Don't worry, we will find a way to save the coach in the next module.)*

This program uses the class named **Sprite01** to populate the game window with 1000 ladybug sprites in different colors with different sizes that fly around the game window in different directions with different speeds as shown in Figure 1 (p. 206) .

11.5 Discussion and sample code

11.5.1 The class named Sprite01

A complete listing of this class is provided in Listing 11 (p. 223) . I will not explain the entire class in detail in this module. Instead, I will provide an overview of the class and then explain various parts of the class as I use them in this and the next two modules.

Beginning of the class named Sprite01

The beginning of the class named **Sprite01** down through the constructor is shown in Listing 1 (p. 212) .

Listing 1 . Beginning of the class named Sprite01.

```
public class Sprite01{
  Image image = null;//The sprite wears this image
  float X = 0f;//X-Position of the sprite
  float Y = 0f;//Y-Position of the sprite
  float width = 0f;//Width of the sprite
  float height = 0f;//Height of the sprite
  float xStep = 1f;//Incremental step size in pixels - X
  float yStep = 1f;//Incremental step size in pixels - Y
  float scale = 1f;//Scale factor for draw method
  Color colorFilter = null;//Color filter for draw method

  float xDirection = 1.0f;//Move to right for positive
  float yDirection = 1.0f;//Move down for positive

  int life = 1;//Used to control life or death of sprite

  boolean exposed = false;//Used in the contagion program

  //Constructor
  public Sprite01(Image image,//Sprite wears this image
                  float X,//Initial position
                  float Y,//Initial position
                  float xDirection,//Initial direction
                  float yDirection,//Initial direction
                  float xStep,//Initial step size
                  float yStep,//Initial step size
                  float scale,//Scale factor for drawing
                  Color colorFilter)
                    throws SlickException {

      //Save incoming parameter values
      this.image = image;
      this.X = X;
      this.Y = Y;
      this.xDirection = xDirection;
      this.yDirection = yDirection;
      this.xStep = xStep;
      this.yStep = yStep;
      this.scale = scale;
      this.colorFilter = colorFilter;

      //Compute and save width and height of image
      width = image.getWidth();
      height = image.getHeight();

  }//end constructor
```

Table 11.3

Straightforward code

The code in Listing 1 (p. 212) is straightforward. It simply declares a number of instance variables, most of which become properties of the object. Listing 1 (p. 212) also defines a constructor that receives and saves values for many of those properties.

The remaining code in Sprite01

If you examine the remaining code in Listing 11 (p. 223) , you will see that it consists of simple property accessor methods along with some methods that control the behavior of an object of the class. I will explain those behavioral methods when I use them later in this and the next two modules.

11.5.2 The class named Slick0200

Will discuss in fragments

A complete listing of the program named **Slick0200** is provided in Listing 10 (p. 221) . I will break the program down and explain it in fragments.

Beginning of the class named Slick0200

The class named Slick0200, down through the **main** method is shown in Listing 2 (p. 214) .

Listing 2 . Beginning of the class named Slick0200.

```
public class Slick0200 extends BasicGame{

  //Store references to Sprite01 objects here.
  Sprite01[] sprites = new Sprite01[1000];

  //Populate this with a ladybug image later.
  Image image = null;

  //Populate these variables with the background
  // image along with the width and height of the
  // image later.
  Image background = null;
  float backgroundWidth;
  float backgroundHeight;

  //This object produces random float values for a
  // variety of purposes.
  Random random = new Random();

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;
  //----------------------------------------------------//

  public Slick0200(){//constructor
    //Set the title
    super("Slick0200, baldwin");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                  throws SlickException{
    AppGameContainer app = new AppGameContainer(
                        new Slick0200(),414,307,false);
    app.start();
  }//end main
```

Table 11.4

Everything in Listing 2 (p. 214) is completely straightforward and should not require an explanation beyond the embedded comments.

11.5.2.1 The init method

The **init** method begins in Listing 3 (p. 215) .

Listing 3 . Beginning of the init method.

```
public void init(GameContainer gc)
                                throws SlickException {

  //Create and save the background image object. Also
  // compute and save the width and height of the image.
  background = new Image("background.jpg");
  backgroundWidth = background.getWidth();
  backgroundHeight = background.getHeight();

  //Create and save an Image object of a ladybug. The
  // sprites will wear this image
  image = new Image("ladybug.png");
```

Table 11.5

There is also nothing new in Listing 3 (p. 215) . Therefore, the embedded comments should suffice to explain the code.

Populate the array

Listing 4 (p. 215) uses a **for** loop to populate the array object referred to by the variable named **sprites** that was declared in Listing 2 (p. 214) . The array object is populated with references to objects of the class **Sprite01** .

Listing 4 . Populate the array.

```
  //Populate the array with references to objects of
  // the Sprite01 class.
  for(int cnt = 0;cnt < sprites.length;cnt++){
    sprites[cnt] = new Sprite01(
       image,//ladybug image
       backgroundWidth/2.0f,//initial position
       backgroundHeight/2.0f,//initial position
       (random.nextFloat() > 0.5) ? 1f : -1f,//direction
       (random.nextFloat() > 0.5) ? 1f : -1f,//direction
       0.1f+random.nextFloat()*2.0f,//step size
       0.1f+random.nextFloat()*2.0f,//step size
       random.nextFloat()*0.15f,//scale
       new Color(random.nextFloat(),//color filter
                 random.nextFloat(),
                 random.nextFloat()));
  }//end for loop

  gc.setTargetFrameRate(targetFPS);//set frame rate

}//end init
```

Table 11.6

Random values

Note that several of the properties of each **Sprite01** objects is initialized with random values.

The use of the **nextFloat** method of the object of the **Random** class may be new to you. If so, this method simply returns a random value between 0.0f and 1.0f each time it is called.

The conditional operator

If the use of the *conditional operator* involving the ? character and the : character is new to you, you will probably need to do some online research in order to understand the use of this operator.

Otherwise, the code in Listing 4 (p. 215) is straightforward and shouldn't require an explanation beyond the embedded comments.

Listing 4 (p. 215) signals the end of the **init** method.

11.5.2.2 The update method

The update method is shown in its entirety in Listing 5 (p. 216) .

Listing 5 . The update method.

```
public void update(GameContainer gc, int delta)
                                  throws SlickException{

  //Do the following for every sprite in the array
  for(int cnt = 0;cnt < sprites.length;cnt++){
    //Ask each sprite to move.
    sprites[cnt].move();

    //Ask each sprite to bounce off the edge if
    // necessary.
    sprites[cnt].edgeBounce(
                      backgroundWidth,backgroundHeight);
  }//end for loop

}//end update
```

Table 11.7

Move and bounce

Listing 5 (p. 216) uses a **for** loop to access each of the sprite objects, asking each object to *move* and to *bounce* off the edge of the game window if necessary.

Listing 5 (p. 216) could hardly be simpler. That is because the necessary complexity has been encapsulated in each object of the **Sprite01** class.

The move method of the Sprite01 class

Listing 6 (p. 217) shows the **move** method from the class named **Sprite01** .

Listing 6 . The move method of the Sprite01 class.

```
public void move(){
  X += xDirection*xStep;
  Y += yDirection*yStep;
}//end move
```

Table 11.8

The code in Listing 6 (p. 217) is also simple. However, in this case, the simplicity is somewhat deceiving. The apparent simplicity derives from the fact that the four required property values are routinely maintained by the object and are readily available to the two statements in the **move** method when needed.

The edgeBounce method of the Sprite01 class

The **edgeBounce** method of the **Sprite01** class is shown in Listing 7 (p. 217) . It is not simple.

Listing 7 . The edgeBounce method of the Sprite01 class.

```
public void edgeBounce(float winWidth,float winHeight){
  //Test for a collision with one of the edges and
  // cause to sprite to bounce off the edge if a
  // collision has occurred.
  if(X + width*scale >= winWidth){
    //A collision has occurred.
    xDirection = -1.0f;//reverse direction
    //Set the position to the right edge less the
    // width of the sprite.
    X = winWidth - width*scale;
  }//end if

  //Continue testing for collisions with the edges
  // and take appropriate action.
  if(X <= 0){
    xDirection = 1.0f;
    X = 0;
  }//end if

  if(Y + height*scale >= winHeight){
    yDirection = -1.0f;
    Y = winHeight - height*scale;
  }//end if

  if(Y <= 0){
    yDirection = 1.0f;
    Y = 0;
  }//end if
}//end edgeBounce
```

Table 11.9

Code that you have seen before

Listing 7 (p. 217) contains essentially the same code that was written into the **update** method of the earlier module mentioned above (p. 207) . In this case, however, all of the complexity has been encapsulated into the **Sprite01** class and replaced by a single call to the **edgeBounce** method in the **update** method of the program named **Slick0200** . Thus, the program's **update** method is now much simpler.

That concludes the discussion of the **update** method for this program.

11.5.2.3 The render method

Listing 8 (p. 218) shows the **render** method for this program.

Listing 8 . The render method.

```
  public void render(GameContainer gc, Graphics g)
                                   throws SlickException{

    //set the drawing mode to honor transparent pixels
    g.setDrawMode(g.MODE_NORMAL);

    //Draw the background to erase the previous picture.
    background.draw(0,0);

    //Draw every sprite in the array.
    for(int cnt = 0;cnt < sprites.length;cnt++){
      //Ask the sprite to draw itself.
      sprites[cnt].draw();
    }//end for loop
  }//end render

}//end class Slick0200
```

Table 11.10

The draw method of the Sprite01 class

The thing that is new about the code in Listing 8 (p. 218) is the call to the **draw** method of the **Sprite01** class. That **draw** method is shown in Listing 9 (p. 218) .

Listing 9 . The draw method of the Sprite01 class.

```
  //This method causes the sprite to be drawn each time
  // it is called.
  public void draw(){
    image.draw(X,Y,scale,colorFilter);
  }//end draw
```

Table 11.11

Little reduction in complexity

In this case, moving the call to the **draw** method of the **Image** class from the **render** method to the **Sprite01** class didn't do much to reduce the complexity of the program. However, that is because I kept the **draw** method in the **Sprite01** class very simple.

I could have made it much more capable and more complex by including additional functionality. For example, I could have caused the **draw** method to call the **drawFlash** method *(see Slick0160: Using the draw and drawFlash methods* [3]) of the **Image** class when the **life** property value goes to zero. In that case, only a silhouette of the dead sprite would be drawn in place of the actual image of the sprite.

That concludes the discussion of the **render** method.

11.6 Run the program

I encourage you to copy the code from Listing 10 (p. 221) and Listing 11 (p. 223) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

11.7 Summary

In this module, you learned how to develop a sprite class from which you can instantiate and animate swarms of sprite objects.

11.8 What's next?

In the next module, you will learn how to use the **Sprite01** class from this module to write a predator/prey simulation program involving thousands of sprites, collision detection, and sound effects.

11.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0200: Developing a sprite class
- File: Slick0200.htm
- Published: 02/06/13
- Revised: 10/03/15

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive

---

[3] http://cnx.org/contents/c8d91157-8952-40b7-8cca-d15ae4f65214

compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

11.10 Complete program listings

Complete listings of the code discussed in this module are provided in Listing 10 (p. 221) and Listing 11 (p. 223) .

Listing 10 . Source code for the program named Slick0200.

```
/*Slick0200.java
Copyright 2013, R.G.Baldwin

This program shows a baseball coach being attacked by a
swarm of vicious ladybugs.

This program uses the class named Sprite01 to populate
the game window with 1000 ladybug sprites in different
colors with different sizes that fly around the game
window in different directions with different speeds.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

import java.util.Random;

public class Slick0200 extends BasicGame{

  //Store references to Sprite01 objects here.
  Sprite01[] sprites = new Sprite01[1000];

  //Populate this with a ladybug image later.
  Image image = null;

  //Populate these variables with the background
  // image along with the width and height of the
  // image later.
  Image background = null;
  float backgroundWidth;
  float backgroundHeight;

  //This object produces random float values for a
  // variety of purposes.
  Random random = new Random();

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;
  //----------------------------------------------------//

  public Slick0200(){//constructor
    //Set the title
    super("Slick0200, baldwin");
  }//end constructor
  //----------------------------------------------------//
```

Table 11.12

.

Listing 11 . Source code for the sprite class named Sprite01.

```
/*Sprite01.java
Copyright 2013, R.G.Baldwin

An object of this class can be manipulated as a sprite
in a Slick2D program.

Tested using JDK 1.7 under WinXP
*********************************************************/
import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

public class Sprite01{
  Image image = null;//The sprite wears this image
  float X = 0f;//X-Position of the sprite
  float Y = 0f;//Y-Position of the sprite
  float width = 0f;//Width of the sprite
  float height = 0f;//Height of the sprite
  float xStep = 1f;//Incremental step size in pixels - X
  float yStep = 1f;//Incremental step size in pixels - Y
  float scale = 1f;//Scale factor for draw method
  Color colorFilter = null;//Color filter for draw method

  float xDirection = 1.0f;//Move to right for positive
  float yDirection = 1.0f;//Move down for positive

  int life = 1;//Used to control life or death of sprite

  boolean exposed = false;//Used in the contagion program

  //Constructor
  public Sprite01(Image image,//Sprite wears this image
                  float X,//Initial position
                  float Y,//Initial position
                  float xDirection,//Initial direction
                  float yDirection,//Initial direction
                  float xStep,//Initial step size
                  float yStep,//Initial step size
                  float scale,//Scale factor for drawing
                  Color colorFilter)
                    throws SlickException {

      //Save incoming parameter values
      this.image = image;
      this.X = X;
      this.Y = Y;
      this.xDirection = xDirection;
      this.yDirection = yDirection;
      this.xStep = xStep;
```

Table 11.13

-end-

Chapter 12

Slick0210: Collision detection and sound[1]

12.1 Table of Contents

12.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is to teach you how to use the **Sprite01** class developed in an earlier module titled Slick0200: Developing a sprite class [2] to write a predator/prey simulation program involving thousands of sprites along with collision detection and sound effects.

---

[1] This content is available online at <http://cnx.org/content/m45755/1.5/>.

[2] http://cnx.org/contents/84c02677-f58d-45b7-811d-b91571187235

12.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

12.2.1.1 Figures

- Figure 1 (p. 230) . Graphic output near the beginning of the simulation.
- Figure 2 (p. 234) . Graphic output near the middle of the simulation.
- Figure 3 (p. 238) . Graphic output near the end of the simulation.
- Figure 4 (p. 242) . Output for the harmless blue sprite scenario.

12.2.1.2 Listings

- Listing 1 (p. 244) . Beginning of the class named Slick0210.
- Listing 2 (p. 245) . Beginning of the init method.
- Listing 3 (p. 247) . Add a red sprite to the ArrayList object.
- Listing 4 (p. 247) . Populate the ArrayList object.
- Listing 5 (p. 248) . Beginning of the update method.
- Listing 6 (p. 248) . Test for a collision.
- Listing 7 (p. 249) . The isCollision method of the Sprite01 class.
- Listing 8 (p. 250) . Process a collision
- Listing 9 (p. 251) . Remove dead objects from the ArrayList object.
- Listing 10 (p. 252) . The render method.
- Listing 11 (p. 254) . Source code for the program named Slick0210.
- Listing 12 (p. 256) . Source code for the class named Sprite01.

12.3 Preview

In an **earlier module** titled Slick0200: Developing a sprite class [3] , we encountered a baseball coach that had been attacked by a swarm of vicious ladybug sprites. I promised you that we would later find a way to save the coach. That time has come.

In this module, I will explain a program that uses the **Sprite01** class from the earlier module to produce a simulation program with the output shown in Figure 1 (p. 230) , Figure 2 (p. 234) , and Figure 3 (p. 238) .

A swarm of insects

Once again, the coach has been attacked by a swarm of 1000 insects. However, in this case, the ladybug sprites have been replaced by vicious green beetle sprites.

A red predator beetle

Fortunately for the coach, a red predator beetle sprite with a taste for green beetles has come along and is gobbling up green beetles as fast as he can collide with them. *(According to the text at the top of Figure 1 (p. 230) , 152 of the 1000 beetles had been consumed by the time the screen shot in Figure 1 (p. 230) was taken.)*

---

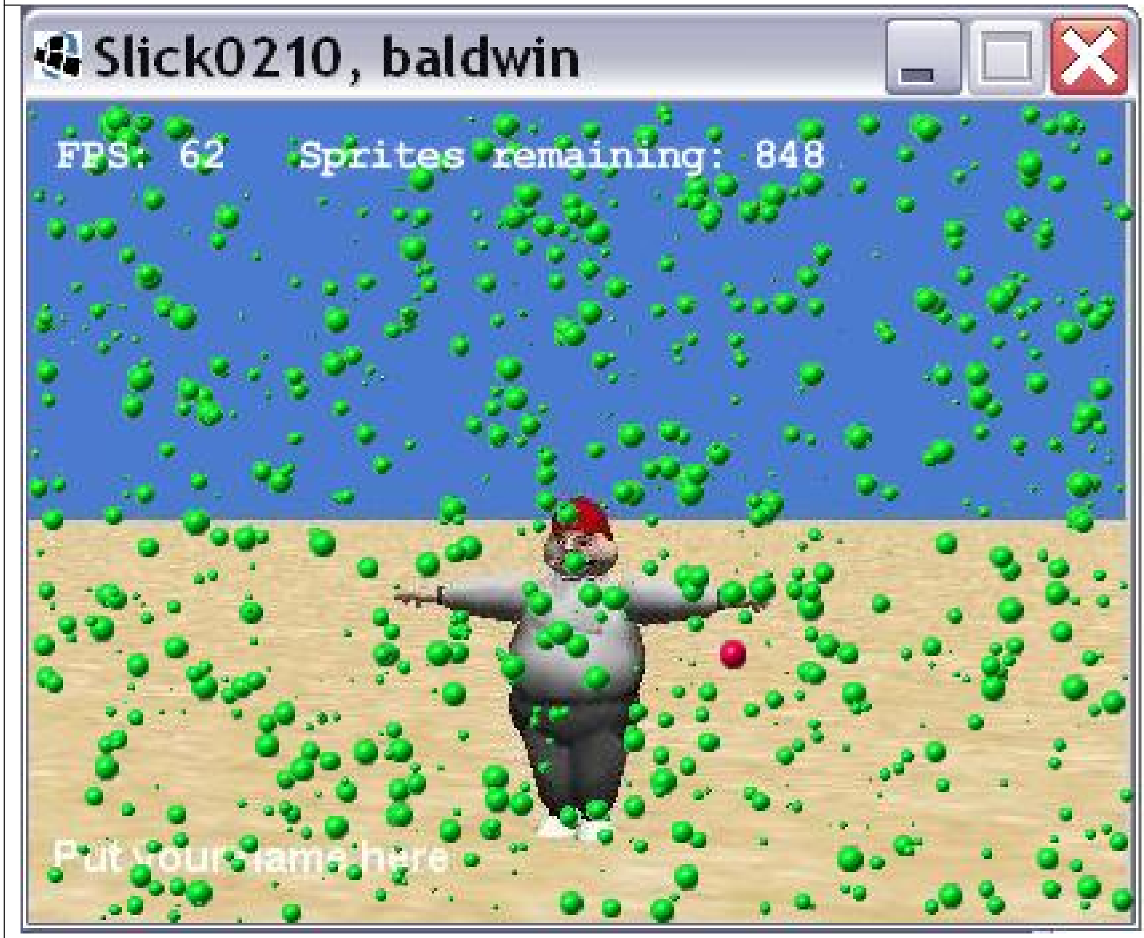
[3] http://cnx.org/contents/84c02677-f58d-45b7-811d-b91571187235


Slick0210, baldwin
FPS: 62
Sprites remaining: 848
Put your name here


Slick0210, baldwin
FPS: 62 Sprites remaining: 848
Put your name here

Slick0210, baldwin
FPS: 62 Sprites remaining: 848
Put your name here

Figure 1 . Graphic output near the beginning of the simulation.



FPS: 62   Sprites remaining: 848

Put your name here

Table 12.1

A fat and happy predator beetle

Figure 2 (p. 234) shows the situation some time later when all but 173 of the green beetles had been eaten. Note that the process of eating those nutritious beetles has caused the red beetle to gain some weight in Figure 2 (p. 234) .


Slick0210, baldwin
FPS: 62   Sprites remaining: 173
Put your name here

Slick0210, baldwin
FPS: 62 Sprites remaining: 173
Put your name here


Slick0210, baldwin
FPS: 62   Sprites remaining: 173
Put your name here

Figure 2 . Graphic output near the middle of the simulation.

Table 12.2

Cleaning up the scraps

Figure 3 (p. 238) shows the situation with only 36 green beetles remaining. Collisions between the beetles is rare at this point, so quite a bit more time will probably be required before the red beetle can collide with and eat the remaining green beetles.


Slick0210, baldwin
FPS: 62   Sprites remaining: 36
Put your name here

Slick0210, baldwin
FPS: 62 Sprites remaining: 36
Put your name here


Slick0210, baldwin
FPS: 62   Sprites remaining: 36
Put your name here

Figure 3 . Graphic output near the end of the simulation.

Table 12.3

What you have learned

In the previous module, you learned how to develop a sprite class from which you can instantiate and animate swarms of sprite objects.

What you will learn

In this module, you will learn how to use the **Sprite01** class developed in the earlier module (p. 226) to write a predator/prey simulation program involving thousands of sprites, collision detection, and sound effects.

12.4 General background information

Actually, it may have been more appropriate to describe this program in terms of jellyfish, *(which eat on the basis of opportunity)* instead of beetles, *(which are more deliberate in their actions)* .

In this program, the red sprite consumes a green sprite only when the two happen to collide by chance. The sprites are not attracted to one another. *(That would be a good exercise for a student project - attraction plus collision.)*

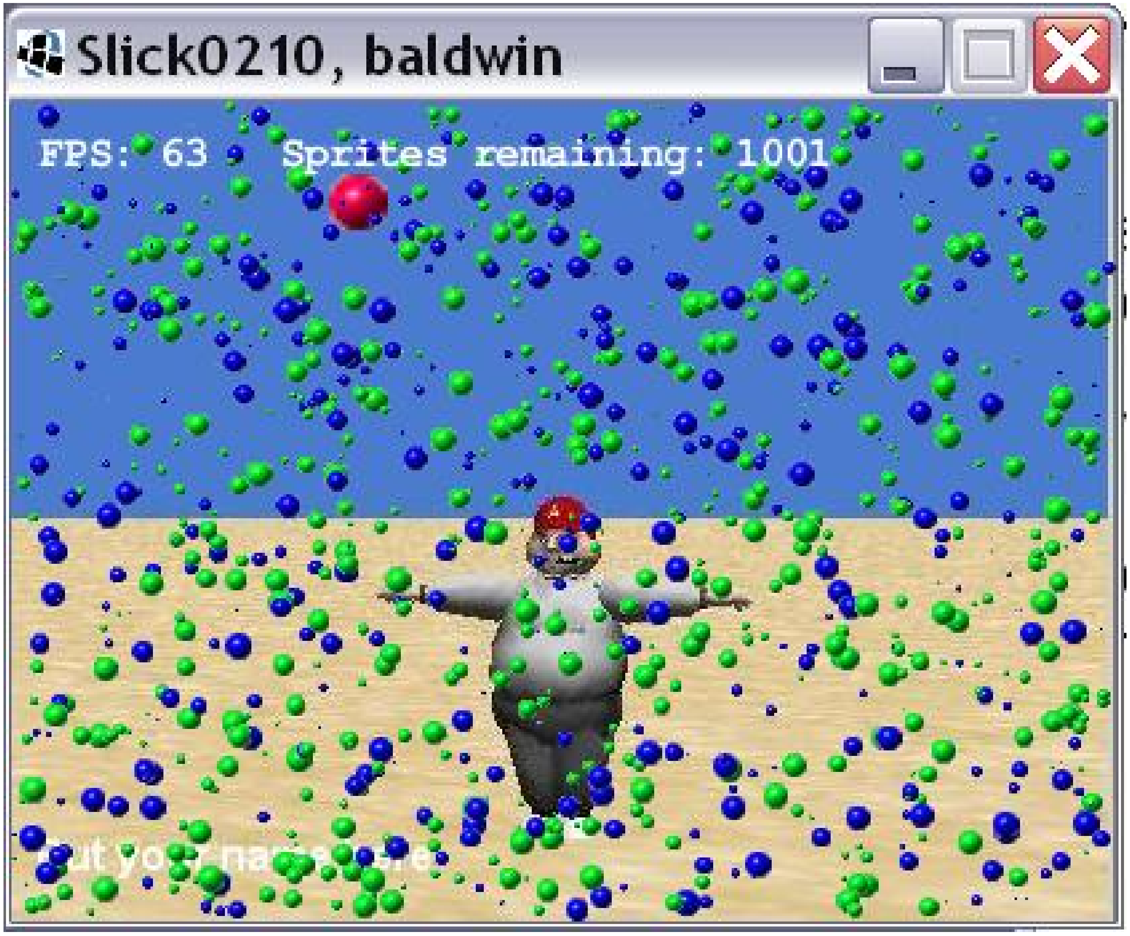
Two scenarios

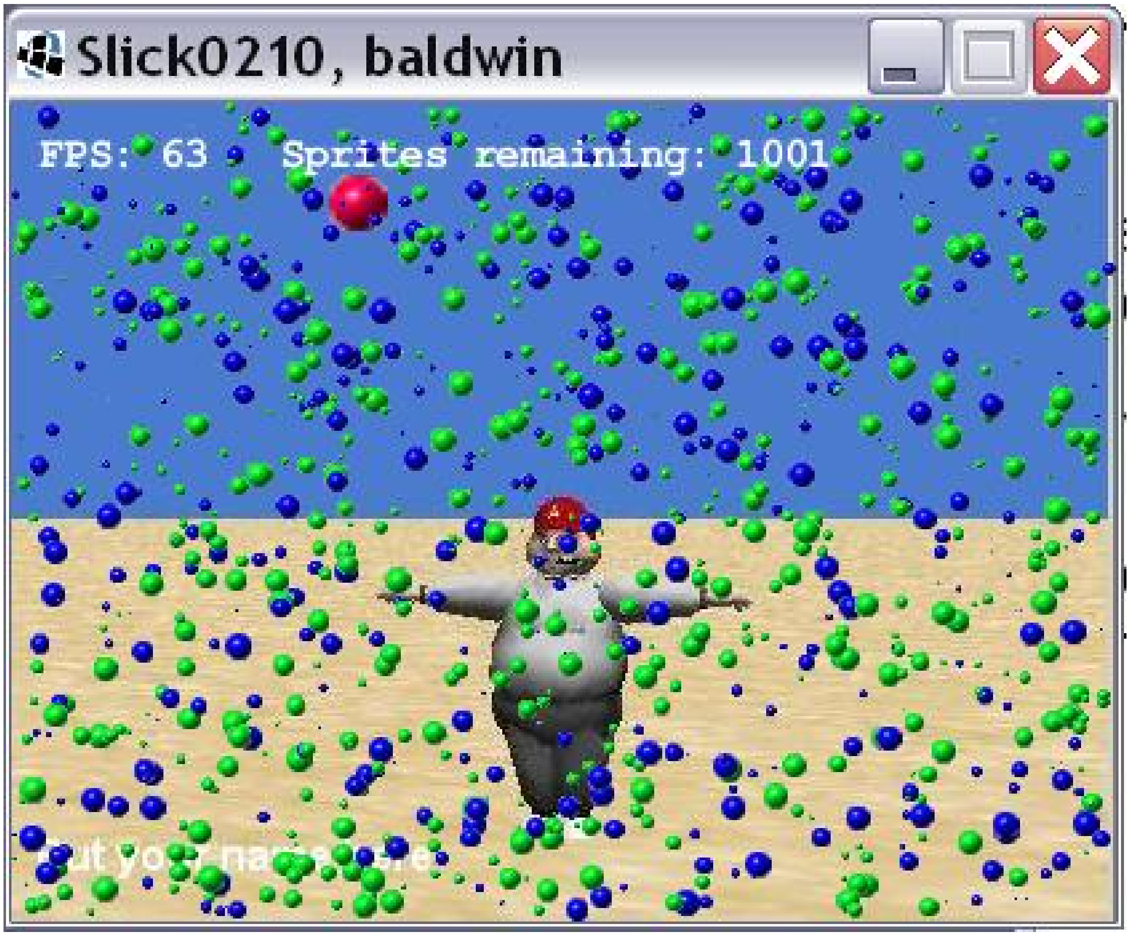
A baseball coach is attacked by a swarm of fierce green flying sprites. Fortunately, a red predator sprite comes along and attacks the green sprites just in time to save the coach.

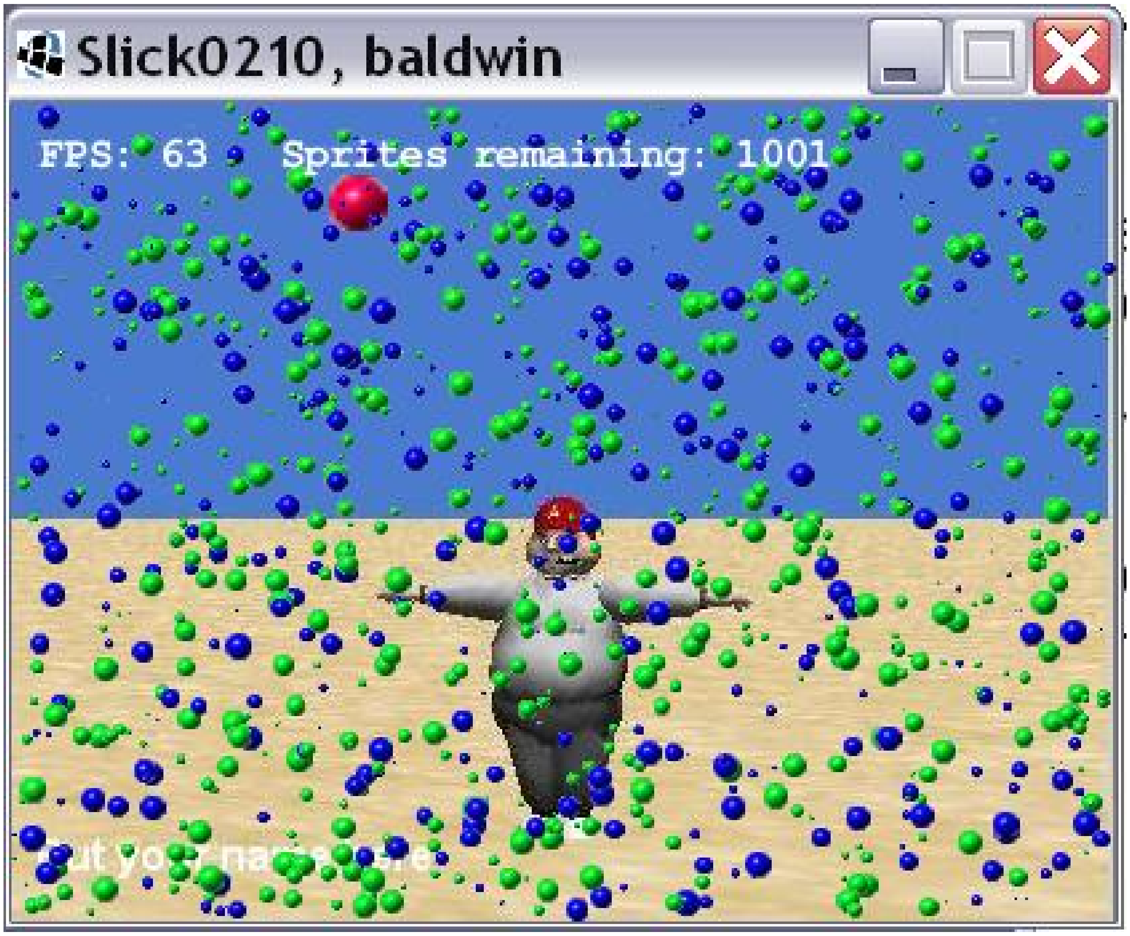
There are two scenarios that can be simulated by setting the variable named **dieOnCollision** *(see Listing 1 (p. 244))* to either *true* or *false* .

Harmless blue sprites

In one scenario *(* ***dieOnCollision*** *= false)* , the vicious green sprites become harmless blue sprites when they collide with the red sprite. A screen shot of this scenario is shown in Figure 4 (p. 242) .




Slick0210, baldwin
FPS: 63
Sprites remaining: 1001
Put your name here



Slick0210, baldwin
FPS: 63
Sprites remaining: 1001
Put your name here

Figure 4 . Output for the harmless blue sprite scenario.

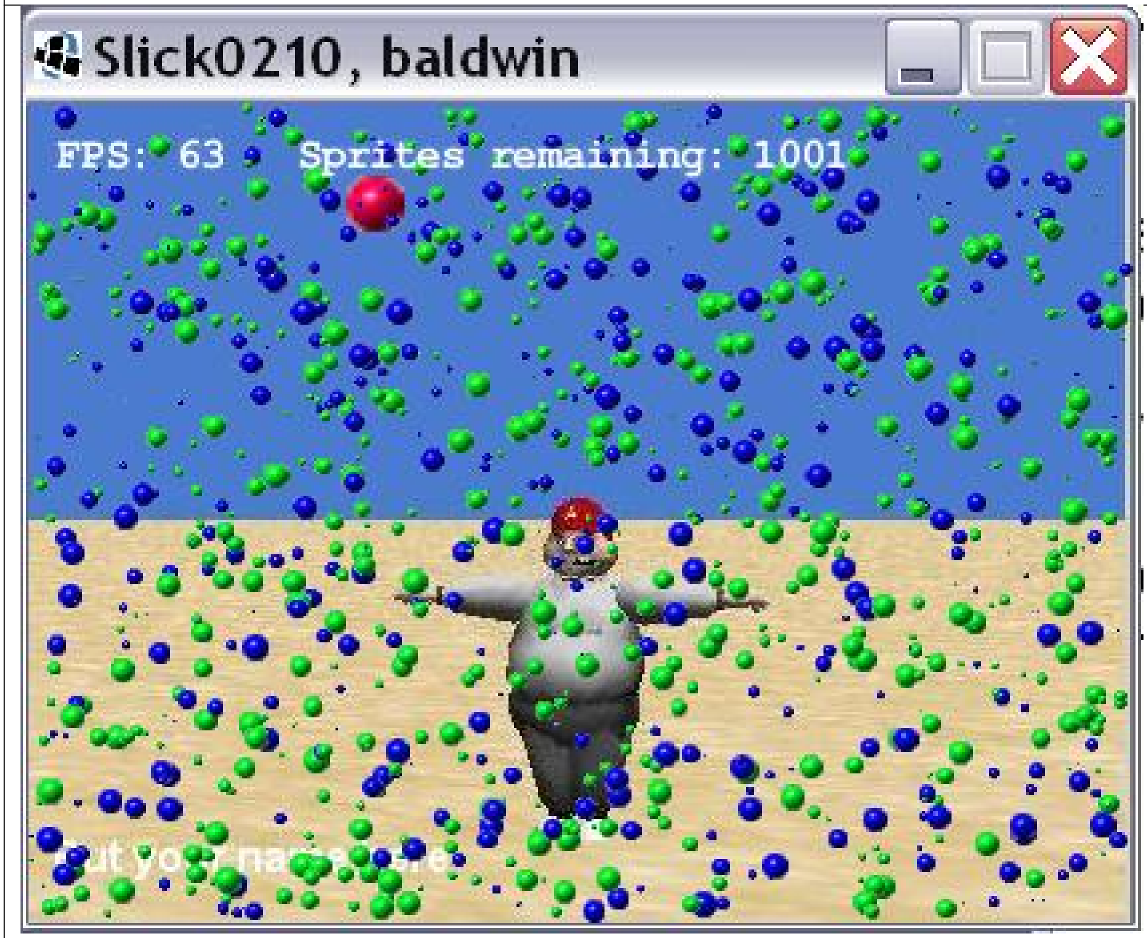


Table 12.4

Green sprites get consumed

In the other scenario (***dieOnCollision*** *= true)* , the green sprites are consumed by the red sprite upon contact and are removed from the population. This is the scenario shown in Figure 3 (p. 238) .

Get fat and happy

In both scenarios, contact between a green sprite and the red sprite causes the red sprite to increase in size.

If you allow the program to run long enough, the probability is high that all of the green sprites will have collided with the red sprite and will either have turned blue or will have been consumed.

12.5 Discussion and sample code

12.5.1 The class named Sprite01

The class named **Sprite01** is shown in Listing 12 (p. 256) . I will explain only those portions of that class that I use in this program that weren't explained in the earlier module.

12.5.2 The class named Slick0210

Will explain in fragments

A complete listing of the class named **Slick0210** is provided in Listing 11 (p. 254) . I will break the code down and explain it in fragments.

Beginning of the class named Slick0210.

The beginning of the class named **Slick0210** , down through the **main** method is shown in Listing 1 (p. 244) .

Listing 1 . Beginning of the class named Slick0210.

```
public class Slick0210 extends BasicGame{

  //Set the value of this variable to true to cause the
  // sprites to die on collision and to be removed from
  // the population.
  boolean dieOnCollision = true;

  //Store references to Sprite01 objects here.
  ArrayList <Sprite01> sprites =
                                new ArrayList<Sprite01>();

  //Change this value and recompile to change the number
  // of sprites.
  int numberSprites = 1000;

  //Populate these variables with references to Image
  // objects later.
  Image redBallImage;
  Image greenBallImage;
  Image blueBallImage;

  //Populate this variable with a reference to a Sound
  // object later.
  Sound blaster;

  //Populate these variables with information about the
  // background image later.
  Image background = null;
  float backgroundWidth;
  float backgroundHeight;

  //This object is used to produce values for a variety
  // of purposes.
  Random random = new Random();

  //Frame rate we would like to see and maximum frame
  // rate we will allow.
  int targetFPS = 60;
  //----------------------------------------------------//

  public Slick0210(){//constructor
    //Set the title
    super("Slick0210, baldwin");
  }//end constructor
  //----------------------------------------------------//

  public static void main(String[] args)
                                    throws SlickException{
    AppGameContainer app = new AppGameContainer(
                          new Slick0210(),414,307,false);
    app.start();
  }//end main
```

Table 12.5

ArrayList

There are two things that are new in Listing 1 (p. 244) . First there is the instantiation of an **ArrayList** object in place of the array object used in the program in the earlier module.

The use of an **ArrayList** instead of an array provides more flexibility in managing a collection of **Sprite01** objects. If you are unfamiliar with the use of **ArrayList** objects, just Google the keywords *baldwin java ArrayList generics* and I'm confident you will find explanatory material that I have published on that topic.

Sound

The second new item in Listing 1 (p. 244) is the declaration of a reference variable of the Slick2D **Sound** class. That variable will be used to hold a reference to a **Sound** object, that will be *played* each time the red sprite collides with a green sprite.

Otherwise, the code in Listing 1 (p. 244) is straightforward and shouldn't require further explanation.

12.5.2.1 The init method

The **init** method begins in Listing 2 (p. 245) .

Listing 2 . Beginning of the init method.

```
public void init(GameContainer gc)
                                 throws SlickException {

  //Create Image objects that will be used to visually
  // represent the sprites.
  redBallImage = new Image("redball.png");
  greenBallImage = new Image("greenball.png");
  blueBallImage = new Image("blueball.png");

  //Create a Sound object.
  blaster = new Sound("blaster.wav");

  //Create a background image and save information
  // about it.
  background = new Image("background.jpg");
  backgroundWidth = background.getWidth();
  backgroundHeight = background.getHeight();
```

Table 12.6

An object of the Sound class

The only thing new in Listing 2 (p. 245) is the instantiation of the object of type **Sound** . As you can see, the syntax for instantiation of a **Sound** object is essentially the same as for instantiating an **Image** object.

Add a red sprite

Listing 3 (p. 247) calls the **add** method of the **ArrayList** class to add a red sprite to the beginning of the **ArrayList** object. *(Actually it add a reference to that object and not the object itself.)*

You are already familiar with the constructor parameters *(shown in Listing 3* (p. 247)) for a **Sprite01** object.

Listing 3 . Add a red sprite to the ArrayList object.

```
    sprites.add(new Sprite01(
       redBallImage,//image
       backgroundWidth/2.0f,//initial position
       backgroundHeight/2.0f,//initial position
       (random.nextFloat() > 0.5) ? 1f : -1f,//direction
       (random.nextFloat() > 0.5) ? 1f : -1f,//direction
       0.1f+random.nextFloat()*2.0f,//step size
       0.1f+random.nextFloat()*2.0f,//step size
       0.5f+random.nextFloat()*1.5f,//scale
       new Color(1.0f,1.0f,1.0f)));//color filter
```

Table 12.7

Populate the ArrayList object

Listing 4 (p. 247) uses a **for** loop and the value of the variable named **numberSprites** *(see Listing 1 (p. 244))* to add 1000 green **Sprite01** object references to the **ArrayList** object.

Listing 4 . Populate the ArrayList object.

```
    for(int cnt = 0;cnt < numberSprites;cnt++){
      sprites.add(new Sprite01(
         greenBallImage,//image
         backgroundWidth*random.nextFloat(),//position
         backgroundHeight*random.nextFloat(),//position
         (random.nextFloat() > 0.5) ? 1f : -1f,//direction
         (random.nextFloat() > 0.5) ? 1f : -1f,//direction
         0.1f+random.nextFloat()*2.0f,//step size
         0.1f+random.nextFloat()*2.0f,//step size
         random.nextFloat()*1.0f,//scale
         new Color(1.0f,1.0f,1.0f)));//color filter
    }//end for loop

    gc.setTargetFrameRate(targetFPS);//set frame rate

  }//end init
```

Table 12.8

Listing 4 (p. 247) also sets the target frame rate and signals the end of the **init** method.

12.5.2.2 The update method

The overall behavior of the **update** method is to use a **for** loop to process the red sprite against each of the green sprites and to take appropriate actions when a collision between the red sprite and a green sprite occurs.

The **update** method begins in Listing 5 (p. 248) .

Listing 5 . Beginning of the update method.

```
public void update(GameContainer gc, int delta)
                                  throws SlickException{

  //Access to the first sprite in the ArrayList object.
  Sprite01 redBallSprite = sprites.get(0);

  //Do the following for every sprite in the ArrayList
  // object
  for(int cnt = 0;cnt < sprites.size();cnt++){
    //Get a reference to the Sprite01 object.
    Sprite01 thisSprite = sprites.get(cnt);

    //Ask the sprite to move according to its properties
    thisSprite.move();

    //Ask the sprite to bounce off the edge if it is at
    // an edge.
    thisSprite.edgeBounce(
                    backgroundWidth,backgroundHeight);
```

Table 12.9

Mostly same as before

The code in Listing 5 (p. 248) is mostly the same as code that you have seen before, so further explanation should not be necessary.

Test for a collision

The code in Listing 6 (p. 248) is new to this module. This code calls the **isCollision** method of the **Sprite01** class to test for a collision between the current green sprite and the red sprite.

Listing 6 . Test for a collision.

```
    boolean collision =
                 thisSprite.isCollision(redBallSprite);
```

Table 12.10

What is a collision?

There are many ways to define and implement collision detection in game and simulation programming. In this program, a collision is deemed to have occurred if any portion of the rectangular **redBallImage** overlaps any portion of the rectangular **greenBallImage** . *(Even though these images appear to be round, they are drawn on a transparent rectangular background.)*

12.5.2.3 The isCollision method of the Sprite01 class

The **isCollision** method of the Sprite01 class is shown in Listing 7 (p. 249) .

Listing 7 . The isCollision method of the Sprite01 class.

```
public boolean isCollision(Sprite01 other){
  //Create variable with meaningful names make the
  // algorithm easier to understand. Can be eliminated
  // to make the algorithm more efficient.
  float thisTop = Y;
  float thisBottom = thisTop + height*scale;
  float thisLeft = X;
  float thisRight = thisLeft + width*scale;

  float otherTop = other.getY();
  float otherBottom = otherTop + other.getHeight()*other.getScale();
  float otherLeft = other.getX();
  float otherRight = otherLeft + other.getWidth()*other.getScale();

  if (thisBottom < otherTop) return(false);
  if (thisTop > otherBottom) return(false);

  if (thisRight < otherLeft) return(false);
  if (thisLeft > otherRight) return(false);

  return(true);

}//end isCollision
```

Table 12.11

Methodology

This method detects a collision between the rectangular sprite object on which the method is called and another rectangular sprite object.

The methodology is to test four cases where a collision could not possibly have occurred and to assume that a collision has occurred if none of those cases are true.

Given that as background, you should be able to use a pencil and paper along with the code in Listing 7 (p. 249) to draw some rectangles and understand how the code in Listing 7 (p. 249) works.

Although I can't guarantee that the method won't call a collision when no collision actually occurred, I am pretty sure that it won't miss any collisions that do occur.

Process a collision

The code in Listing 8 (p. 250) is executed when the call to the **isCollision** method returns true. Therefore, this code processes a collision only when one has occurred.

The code excludes collisions between the red sprite and itself, *(which is an artifact of the algorithm)* . It also excludes collisions between the red sprite and blue sprites *(if they exist)* .

Listing 8 . Process a collision

```
     if((collision == true)&&
        (! thisSprite.getImage().equals(redBallImage)) &&
        (! thisSprite.getImage().equals(blueBallImage))){

       //A collision has occurred, change the color of
       // this sprite to blue and maybe cause it to
       // die and be removed from the population.
       thisSprite.setImage(blueBallImage);
       if(dieOnCollision){
         thisSprite.setLife(0);
       }//end if

       //Cause the redBallSprite to change direction on
       // a random basis.
       redBallSprite.setXDirection(
                  (random.nextFloat() > 0.5) ? 1f : -1f);
       redBallSprite.setYDirection(
                  (random.nextFloat() > 0.5) ? 1f : -1f);

       //Cause the redBallSprite to change stepsize on a
       // random basis.
       redBallSprite.xStep =
                       0.1f+random.nextFloat()*2.0f;
       redBallSprite.yStep =
                       0.1f+random.nextFloat()*2.0f;

       //Cause the redBallSprite to grow larger
       redBallSprite.setScale(redBallSprite.getScale() +
                    (redBallSprite.getScale()) * 0.001f);

       //Play a sound to indicate that a collision has
       // occurred.
       blaster.play();
     }//end if

   }//end for loop
```

Table 12.12

Not complicated code

Although the code in Listing 8 (p. 250) is long and tedious, it isn't particularly complicated. It consists mainly of calls to the accessor methods of the two sprite objects involved in the collision to modify their property values in some way.

Turn a green sprite into a blue sprite

For example, near the top of Listing 8 (p. 250) , there is a call to the **setImage** method of the green sprite to change it to a blue sprite.

Kill the green sprite

This is followed by a call to the **setLife** method to set the life of the *(now blue)* sprite object to 0, but only if the **dieOnCollision** variable belonging to the object is true. Later on, all sprite objects with a **life** property value of 0 will be removed from the population.

And so forth

I could continue down the page describing the calls to various other accessor methods, but that shouldn't be necessary. The embedded comments should suffice for the explanation.

Play a sound

Finally near the end of the code in Listing 8 (p. 250) , there is a call to the **play** method belonging to **Sound** object referred to as **blaster** .

Each time there is a collision between the red sprite and a green sprite, the sound loaded earlier from the file named **"blaster.wav"** is played.

The end of the for loop

Listing 8 (p. 250) signals the end of the **for** loop, but does not signal the end of the **update** method. There is one more task to complete before the **update** method terminates.

Remove dead objects from the ArrayList object

The code in Listing 9 (p. 251) uses an **Iterator** to remove all objects having a **life** property value that is less than or equal to zero from the **ArrayList** object.

Listing 9 . Remove dead objects from the ArrayList object.

```
    //Remove dead objects from the ArrayList object
    Iterator <Sprite01> iter = sprites.iterator();

    while(iter.hasNext()){
      Sprite01 theSprite = iter.next();
      if(theSprite.getLife() <= 0){
        iter.remove();
      }//end if
    }//end while loop

  }//end update
```

Table 12.13

Explanation of an Iterator

The explanation of an Iterator is beyond the scope of this module. However, if you Google the keywords *baldwin java iterator* , you will find several tutorials that I have published on this and related topics.

Listing 9 (p. 251) signals the end of the **update** method.

12.5.2.4 The render method

The render method is shown in Listing 10 (p. 252) . There is nothing new in this code.

Listing 10 . The render method.

```
  public void render(GameContainer gc, Graphics g)
                                  throws SlickException{

    //set the drawing mode to honor transparent pixels
    g.setDrawMode(g.MODE_NORMAL);

    //Draw the background to erase the previous picture.
    background.draw(0,0);

    //Draw every sprite in the ArrayList object.
    for(int cnt = 0;cnt < sprites.size();cnt++){
      sprites.get(cnt).draw();
    }//end for loop

    //Display the remaining number of sprites.
    g.drawString(
       "Sprites remaining: " + (sprites.size()),100f,10f);
    //Signal when all sprites have been eaten.
    if(sprites.size() == 1){
      g.drawString("Burp!",100f,25f);
    }//end if
  }//end render

}//end class Slick0210
//======================================================//
```

Table 12.14

The end of the class

Listing 10 (p. 252) also signals the end of the **Slick0210** class and the end of the program.

12.6 Run the program

I encourage you to copy the code from Listing 11 (p. 254) and Listing 12 (p. 256) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

12.7 Summary

In this module, you learned how to use the **Sprite01** class from an earlier module to write a predator/prey simulation program involving thousands of sprites, collision detection, and sound effects.

12.8 What's next?

In the next module, you will learn how to write a program that simulates the spread of a fatal communicable disease within a population.

12.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0210: Collision detection and sound
- File: Slick0210.htm
- Published: 02/06/13
- Revised: 10/03/15

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

12.10 Complete program listings

Complete listings of the code discussed in this module are provided in Listing 11 (p. 254) and Listing 12 (p. 256) .

Listing 11 . Source code for the program named Slick0210.

```
/*Slick0210.java
Copyright 2013, R.G.Baldwin

A baseball coach is attacked by a swarm of fierce green
flying insects. Fortunately, a red predator insect comes
along and attacks the green insects just in time to save
the coach.

There are two scenarios that can be exercised by setting
dieOnCollision to true or false. In one scenario,
the green insects become harmless blue insects when they
collide with the red insect. In the other case, they are
consumed by the red insect upon contact and removed from
the population.

In both scenarios, contact between a green insect and the
red insect causes the red insect to increase in size.

If you allow the program to run long enough, the
probability is high that all of the green insects will
have collided with the red insect and will either have
turned blue or have been consumed.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;
import org.newdawn.slick.Sound;

import java.util.Random;
import java.util.ArrayList;
import java.util.Iterator;

public class Slick0210 extends BasicGame{

  //Set the value of this variable to true to cause the
  // sprites to die on collision and to be removed from
  // the population.
  boolean dieOnCollision = true;

  //Store references to Sprite01 objects here.
  ArrayList <Sprite01> sprites =
                          new ArrayList<Sprite01>();

  //Change this value and recompile to change the number
  // of sprites.
  int numberSprites = 1000;
```

Table 12.15

.

Listing 12 . Source code for the class named Sprite01.

```
    /*Sprite01.java
Copyright 2013, R.G.Baldwin

An object of this class can be manipulated as a sprite
in a Slick2D program.

Tested using JDK 1.7 under WinXP
*********************************************************/
import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

public class Sprite01{
  Image image = null;//The sprite wears this image
  float X = 0f;//X-Position of the sprite
  float Y = 0f;//Y-Position of the sprite
  float width = 0f;//Width of the sprite
  float height = 0f;//Height of the sprite
  float xStep = 1f;//Incremental step size in pixels - X
  float yStep = 1f;//Incremental step size in pixels - Y
  float scale = 1f;//Scale factor for draw method
  Color colorFilter = null;//Color filter for draw method

  float xDirection = 1.0f;//Move to right for positive
  float yDirection = 1.0f;//Move down for positive

  int life = 1;//Used to control life or death of sprite

  boolean exposed = false;//Used in the contagion program

  //Constructor
  public Sprite01(Image image,//Sprite wears this image
                  float X,//Initial position
                  float Y,//Initial position
                  float xDirection,//Initial direction
                  float yDirection,//Initial direction
                  float xStep,//Initial step size
                  float yStep,//Initial step size
                  float scale,//Scale factor for drawing
                  Color colorFilter)
                    throws SlickException {

      //Save incoming parameter values
      this.image = image;
      this.X = X;
      this.Y = Y;
      this.xDirection = xDirection;
      this.yDirection = yDirection;
      this.xStep = xStep;
      this.yStep = yStep;
```

Table 12.16

-end-

Chapter 13

Slick0220: Simulating a pandemic[1]

13.1 Table of Contents

13.2 Preface

This module is one in a collection of modules designed to teach you about the anatomy of a game engine.

Although the modules in this collection will concentrate on the Java game library named Slick2D, the concepts involved and the knowledge that you will gain is applicable to different game engines written in different programming languages as well.

The purpose of this module is to teach you how to write a program that simulates the spread of a fatal communicable disease within a population *(a pandemic)* .

13.2.1 Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

---

[1] This content is available online at <http://cnx.org/content/m45757/1.6/>.

13.2.1.1 Figures

13.2.1.2 Listings

13.3 Preview

In an **earlier module** titled Slick0210: Collision detection and sound [2] , you learned how to write a non-trivial program involving thousands of sprites, collision detection, and sound. We will take that concept considerably further in this module by writing a program that simulates the spread of a fatal communicable disease within a population *(a pandemic)* and displays the results in animated graphic form.

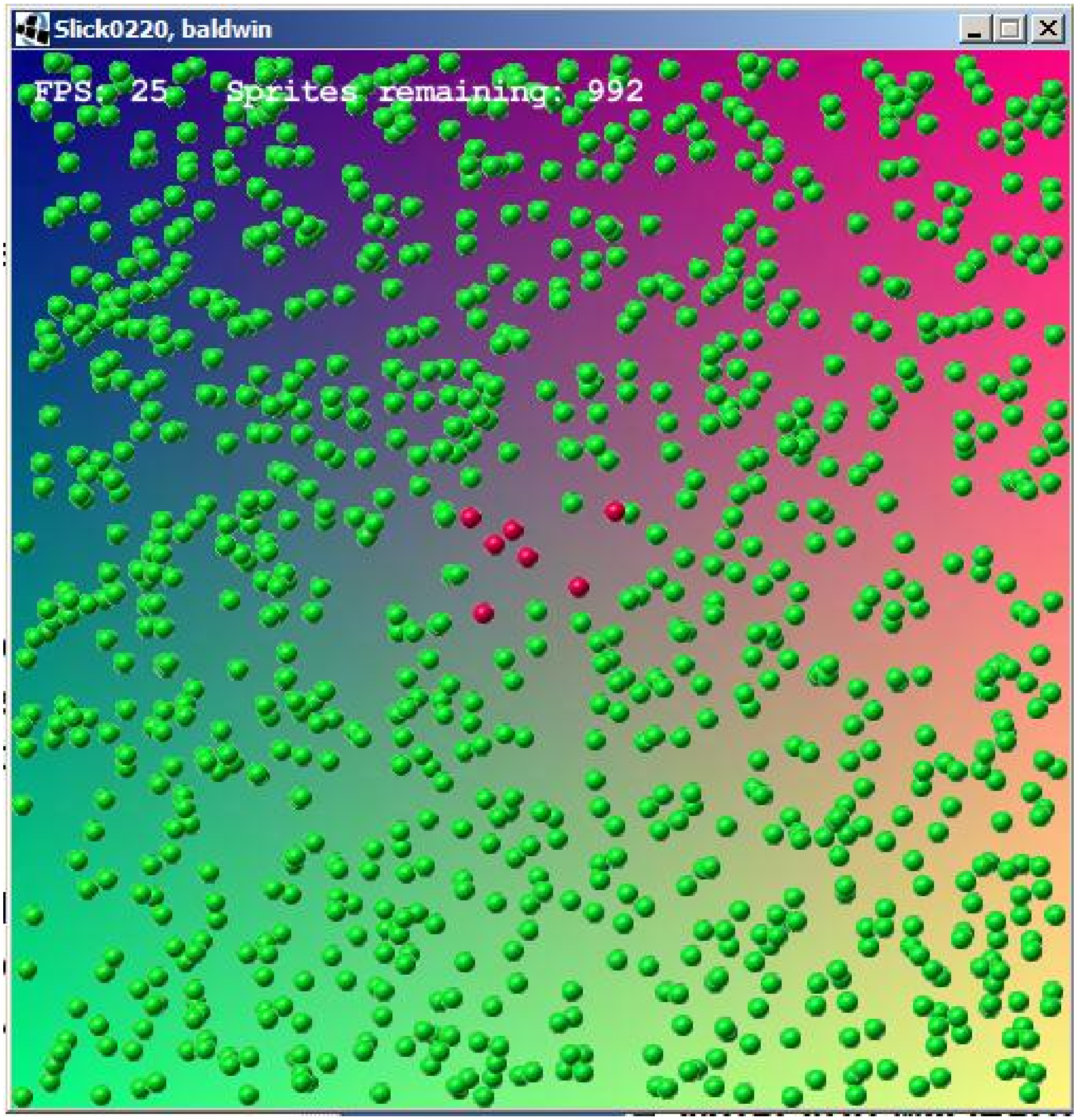
The disease has gained a foothold

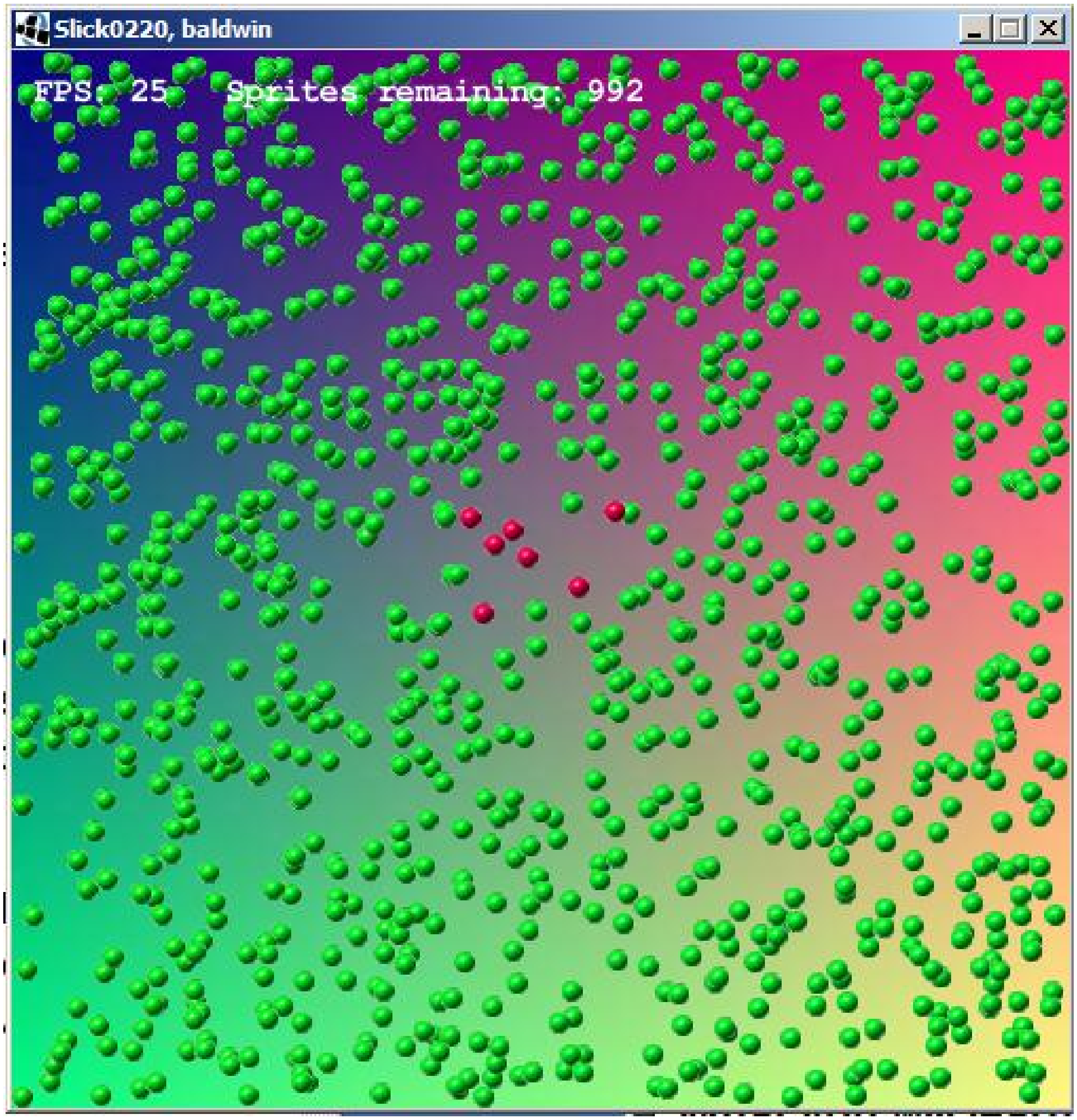
Figure 1 (p. 264) shows the result of inserting a single infected sprite into a population of healthy sprites. Healthy sprites are colored green and infected sprites are colored red.

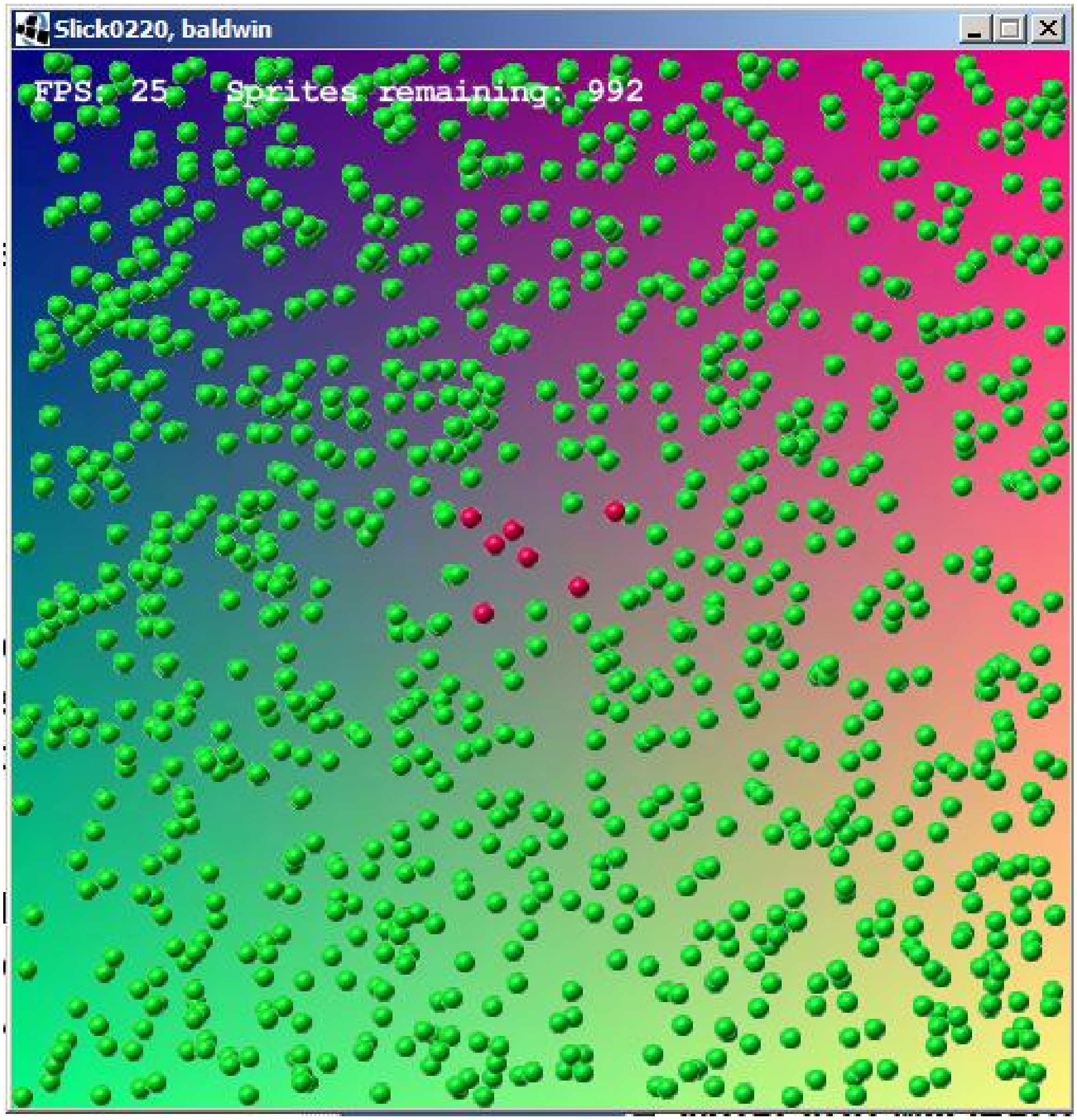
By the time the screen shot in Figure 1 (p. 264) was taken, the disease had gained a foothold, several other sprites had become infected, and eight of the original 1000 sprites had died, leaving only 992 live sprites including the seven that are infected.

---

[2] http://cnx.org/contents/7fd862dc-efad-463f-beff-1a1276a4b698


Slick0220, baldwin
FPS: 25   Sprites remaining: 992


Slick0220, baldwin
FPS: 25 Sprites remaining: 992



FPS: 25 Sprites remaining: 992

Figure 1 . The disease has gained a foothold.

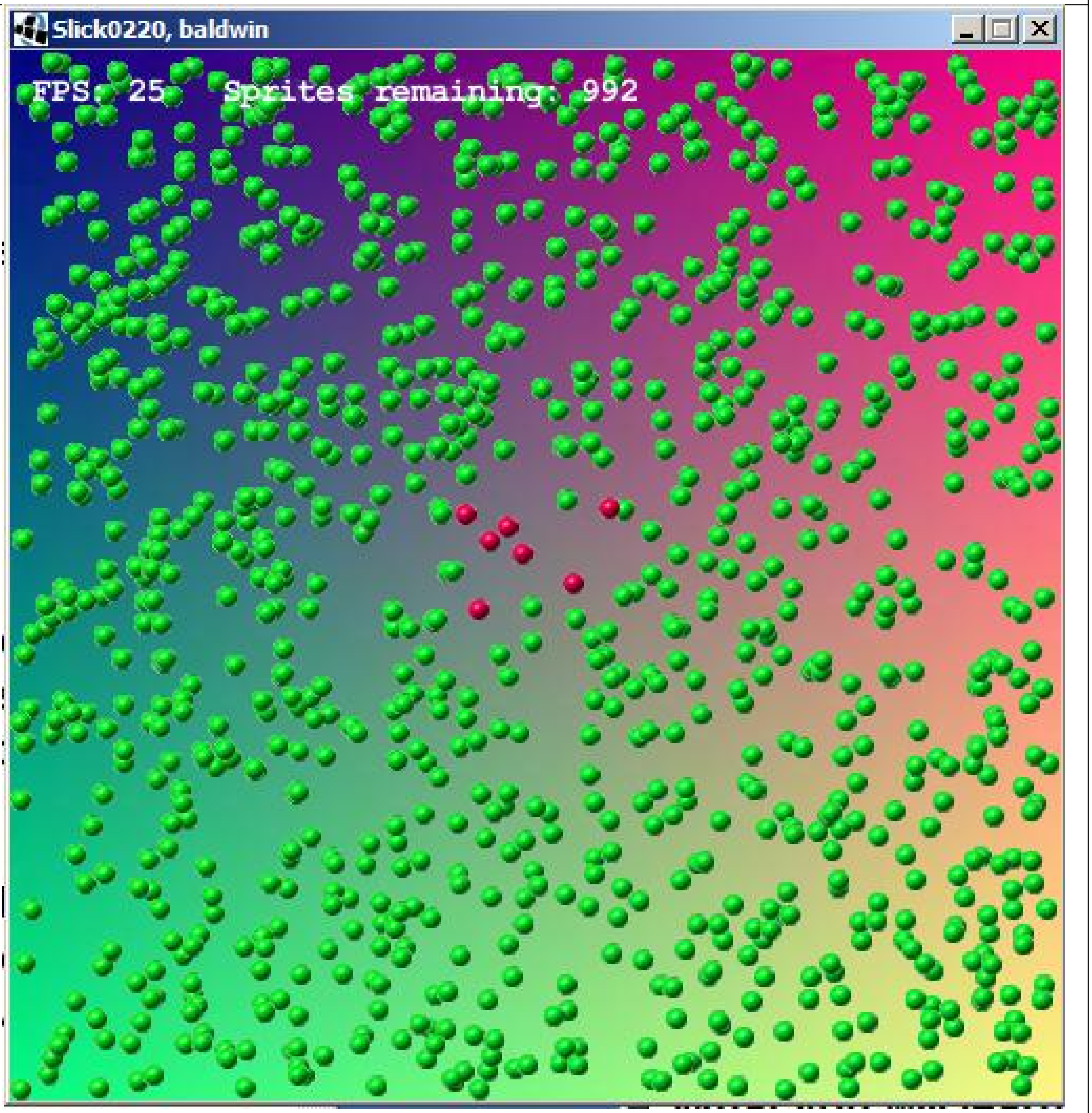
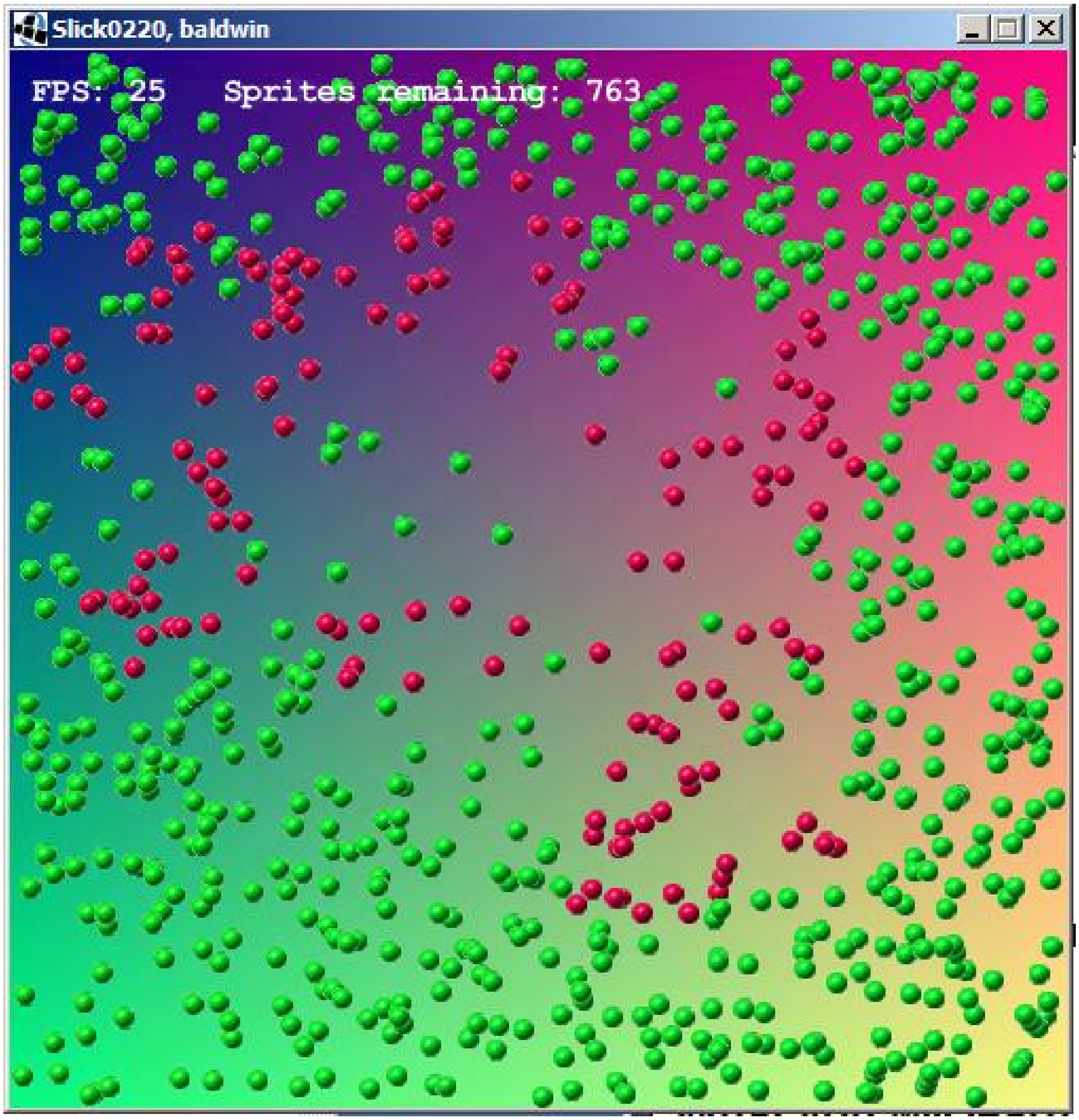
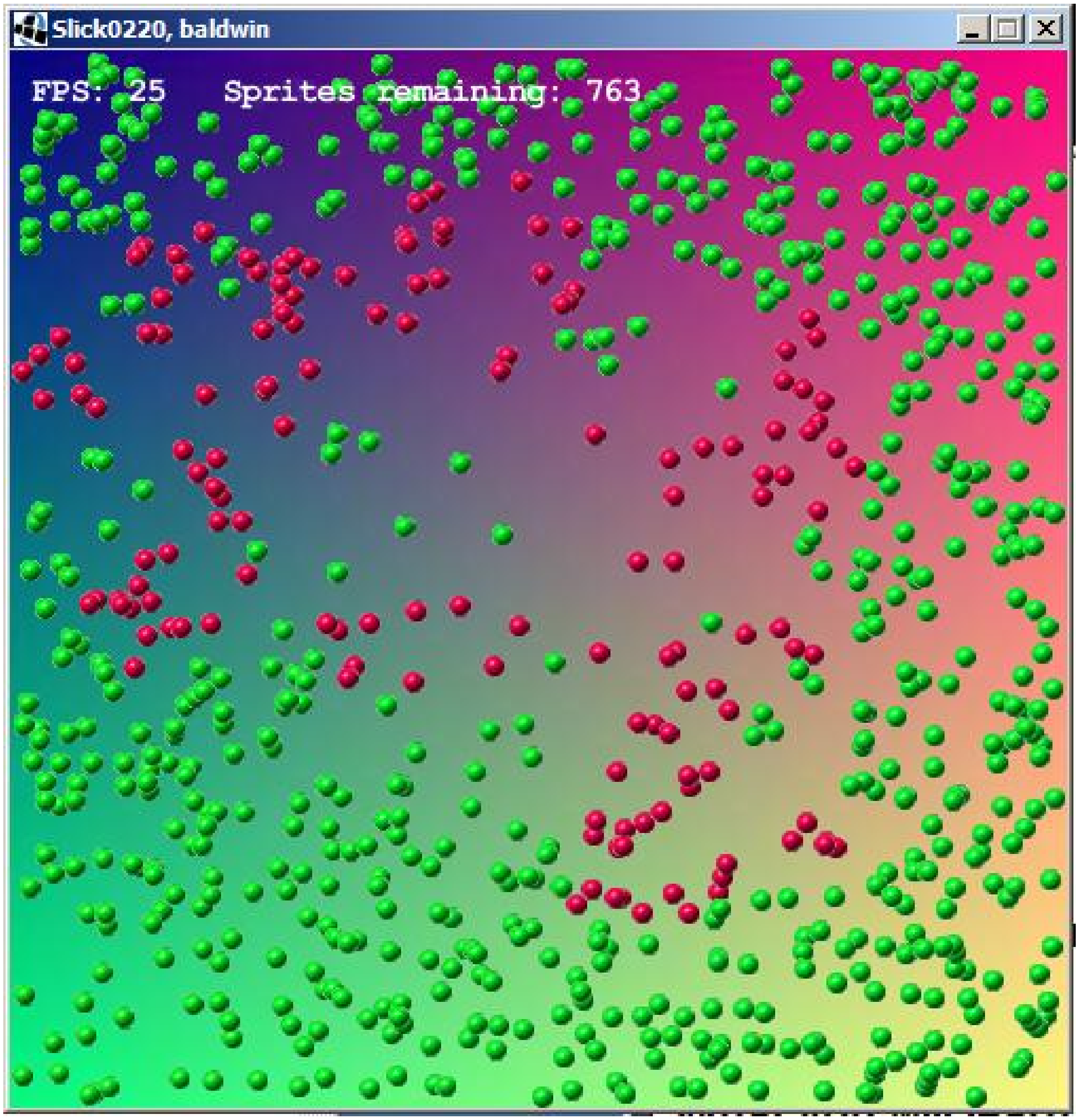


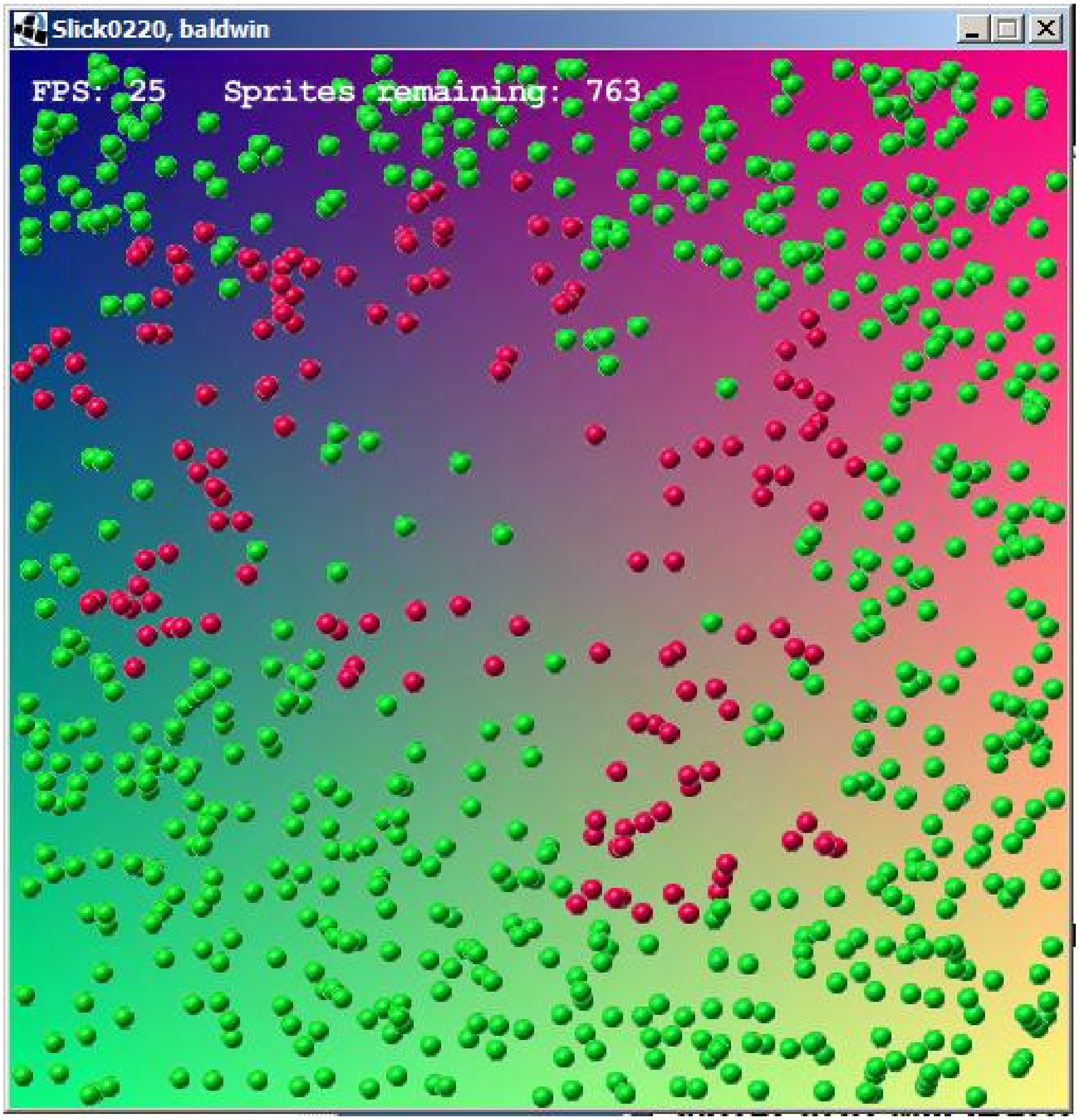
Table 13.1

The disease has spread into the population

Figure 2 (p. 268) shows the situation some time later when the disease has spread considerably. By this point, many sprites have become infected *(and are infecting others)* and only 763 of the original 1000 sprites are still alive including those that are infected.


Slick0220, baldwin
FPS: 25 Sprites remaining: 763


Slick0220, baldwin
FPS: 25 Sprites remaining: 763



FPS: 25   Sprites remaining: 763

Figure 2 . The disease has spread into the population.

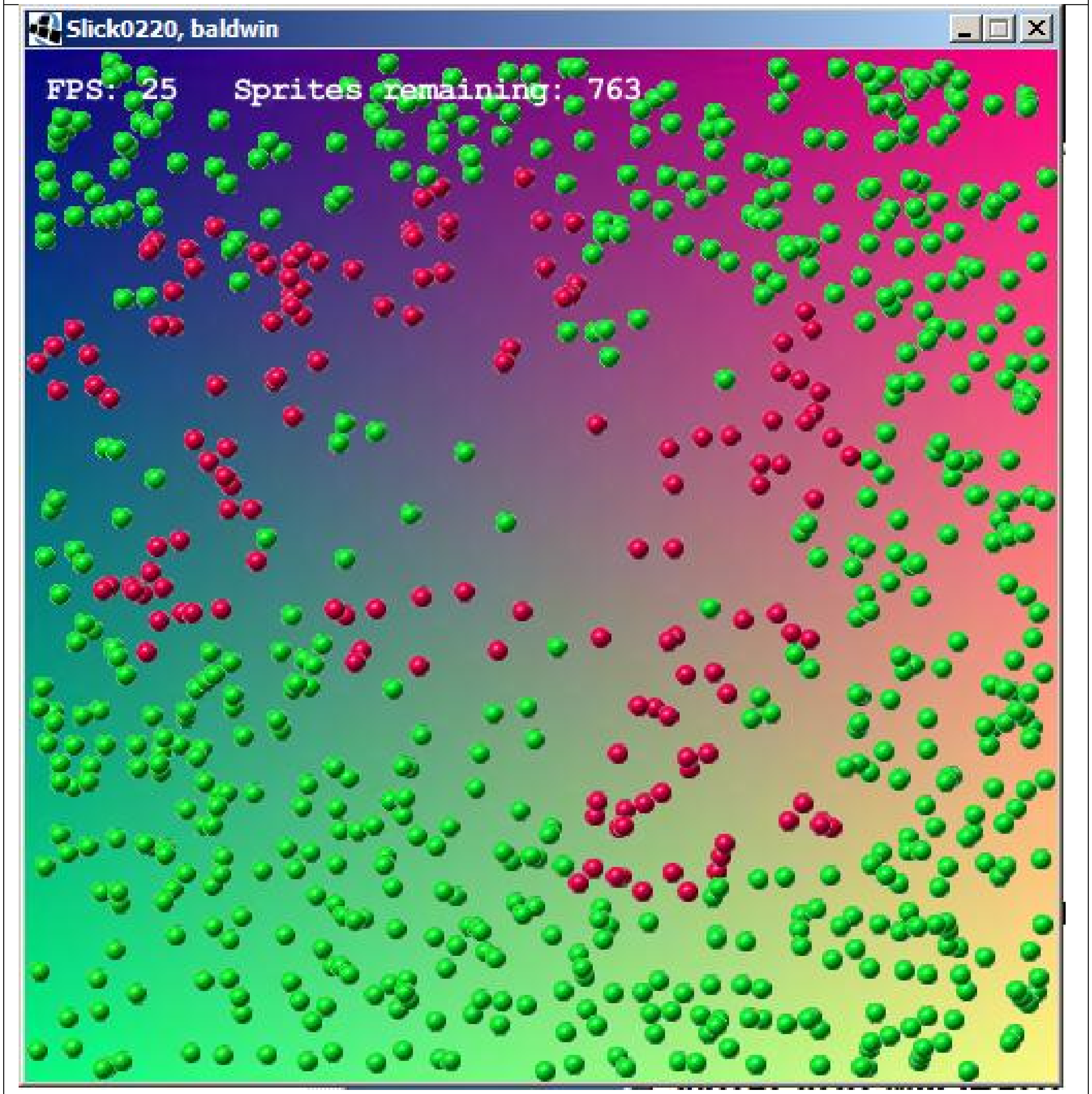
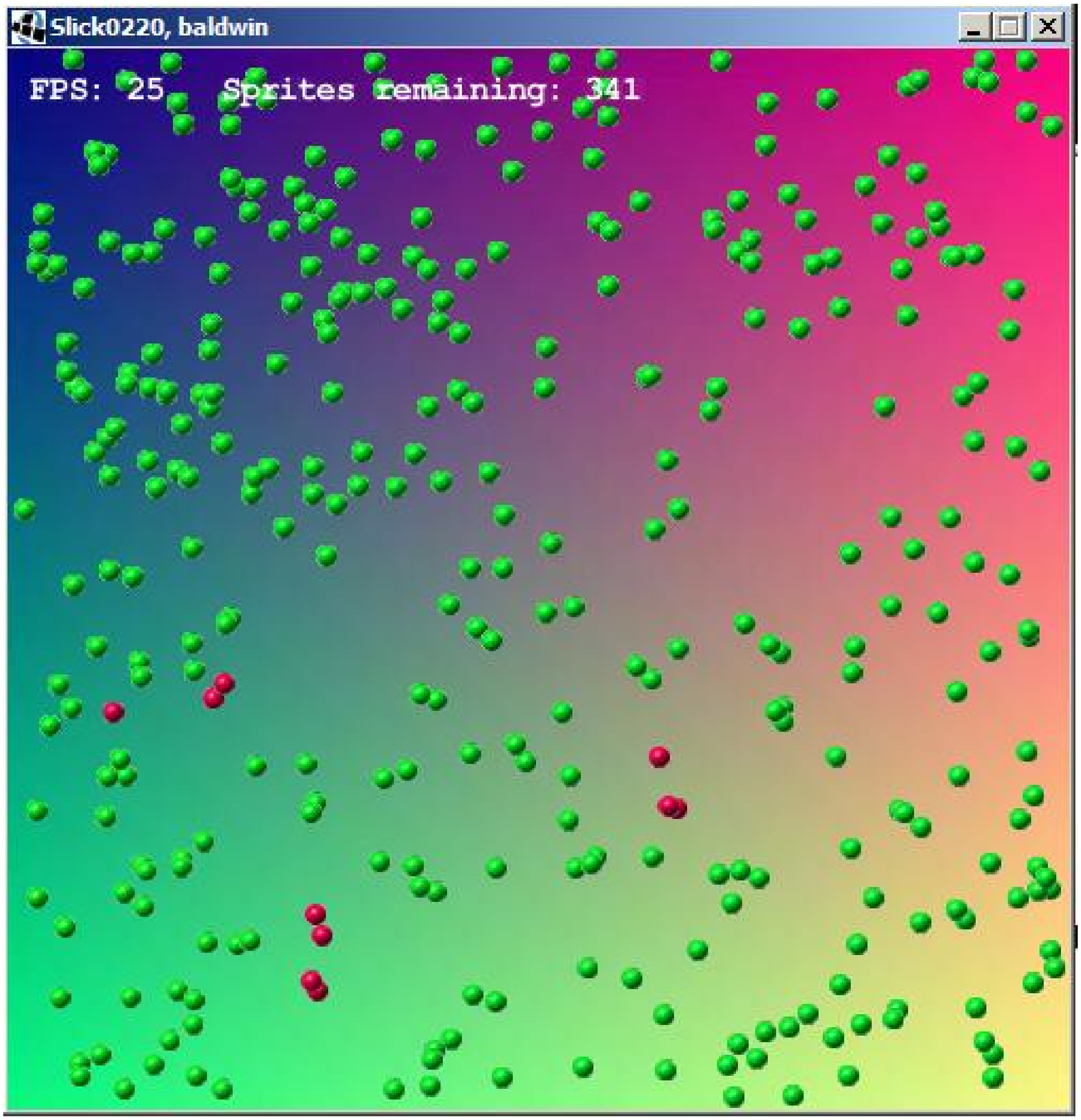
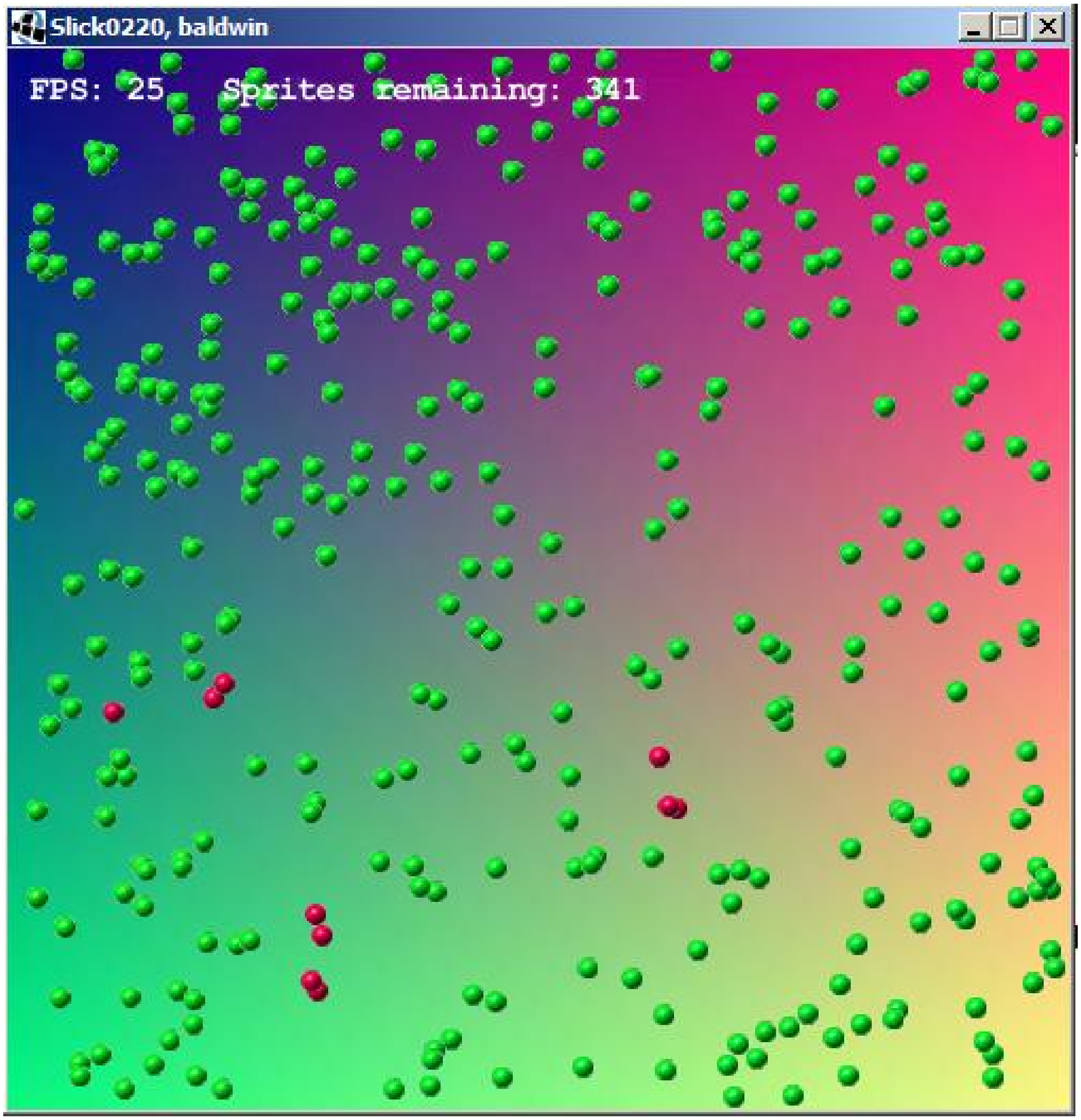


Table 13.2

The disease has receded

Figure 3 (p. 272) shows the situation much later. For the set of properties used to run this simulation, the pandemic appears to be receding with 341 of the 1000 original sprites still alive.


Slick0220, baldwin
FPS: 25 Sprites remaining: 341


Slick0220, baldwin
FPS: 25 Sprites remaining: 341

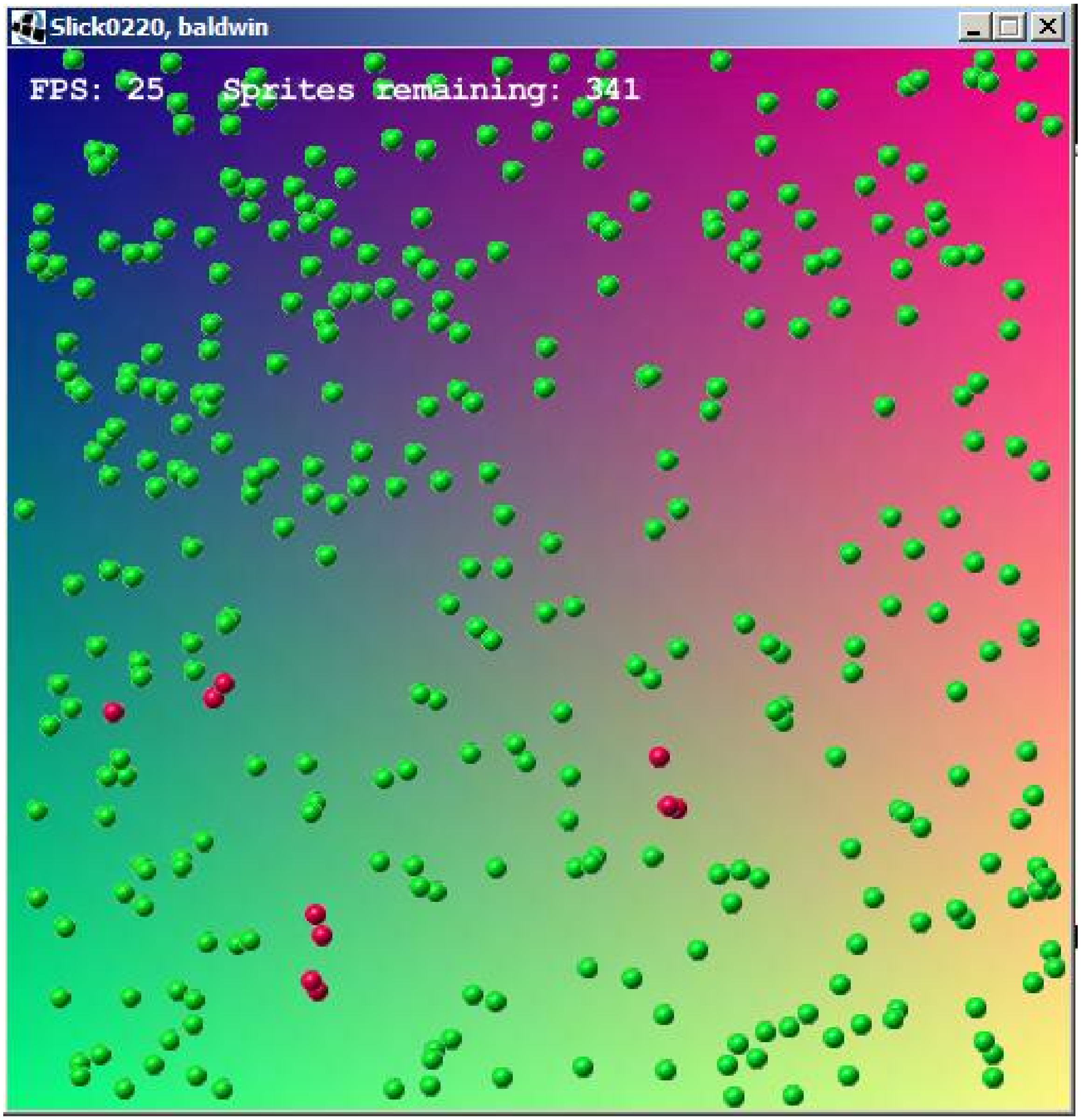

Slick0220, baldwin
FPS: 25   Sprites remaining: 341

Figure 3 . The disease has receded after killing many in the population.

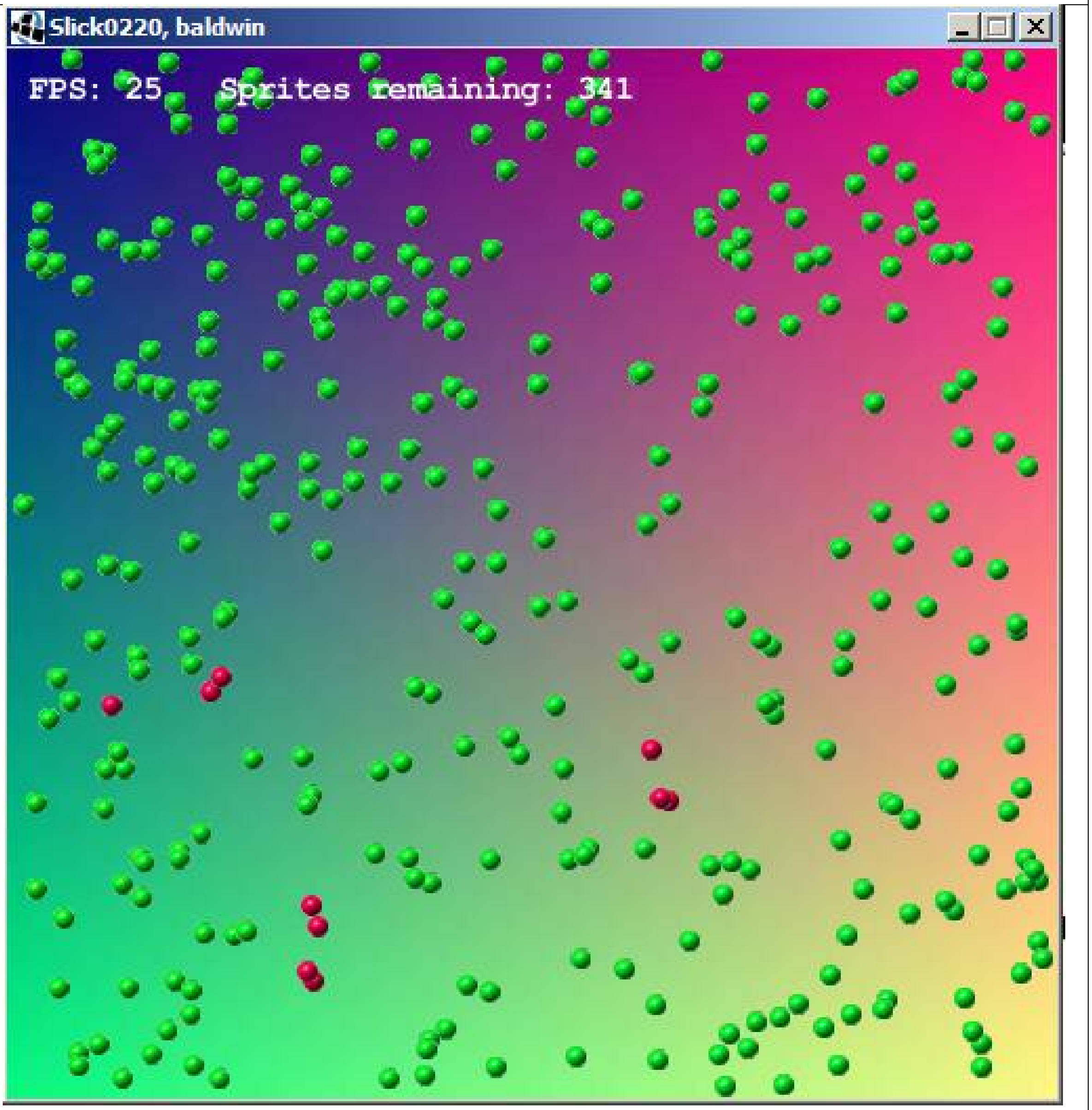


Table 13.3

Properties that control the spread

Later on, I will explain the properties that control the spread of the disease. Some sets of property values produce results similar to those shown above where the disease gains a foothold, spreads for awhile killing many sprites, and then recedes without killing the entire population.

Other sets of property values end up with all of the sprites having died.

Still other sets of property values end up with the disease being unable to gain a foothold and spread beyond just a few individual sprites.

What you have learned

You have learned how to use a basic Slick2D game engine to create simulations involving thousands of sprites, collision detection, and sound.

What you will learn

In this module, you will learn how to use what you have previously learned to write a relatively complex *(but somewhat simplified)* simulation of a real-world pandemic.

If you were to study the characteristic of pandemics, you could probably upgrade this program to produce a better model of a pandemic. For example, an interesting student project would be to allow healthy sprites to reproduce when they come in contact based on a random probability function. This would allow the population to be growing at the same time that it is dying off due to the disease. Of course, it may then be necessary to deal with the effects of a population explosion.

13.4 General background information

This program simulates the spread of a fatal communicable disease within a population.

A single infected sprite is introduced into a population of sprites. The disease is spread by physical contact between a healthy sprite and an infected sprite.

You can watch as the disease either spreads and kills the entire population or spreads for awhile, then recedes and dies out.

Infected sprites are colored red. Healthy sprites are colored green. A sound is emitted *(simply to demonstrate how to emit sounds)* each time there is contact between an infected sprite and a healthy sprite.

The final outcome

The final outcome is determined both by chance and by several factors including:

- The life expectancy of an infected sprite.
- The probability of infection due to contact with an infected sprite.
- The degree of mobility of both infected and healthy sprites.
- The population density of sprites.

The actual values for the first three factors for each individual are determined by a maximum value multiplied by a random number between 0 and 1.0.

Experimentation

Instance variables are provided for all four of these factors. You can modify the values and recompile the program to experiment with different combinations of the factors.

A good exercise for a student would be to create a GUI that allows the factors to be entered more easily without having to recompile the program for purposes of experimentation.

13.5 Discussion and sample code

13.5.1 The class named Sprite01

The class named **Sprite01** is shown in Listing 9 (p. 287) . There is nothing new in Listing 9 (p. 287) that I haven't explained in earlier modules.

13.5.2 The class named Slick0220

Will explain in fragments

A complete listing of the class named **Slick0220** is provided in Listing 8 (p. 285) . I will break the code down and explain it in fragments.

Beginning of the class named Slick0220.

The beginning of the class named **Slick0220** , down through the **main** method is shown in Listing 1 (p. 274) .

Listing 1 . Beginning of the class named Slick0220.

```
public class Slick0220 extends BasicGame{

  //The values of the following variables can be changed
  // to effect the spread of the disease.

  //Set the life expectancy of an infected sprite
  // in frames.
  int infectedSpriteLife = 96;

  //Set the maximum fraction of exposed sprites that will
  // become infected.
  float probabilityOfInfection = 0.5f;

  //Set the maximum step size that a sprite will move in
  // one frame.
  float maxStepSize = 1;

  //Set the initial number of sprites in the population.
  int numberSprites = 1000;

  //References to Sprite01 objects are stored here.
  ArrayList <Sprite01> sprites =
                              new ArrayList<Sprite01>();

  //These variables are populated with references to Image
  // objects later.
  Image redBallImage;
  Image greenBallImage;

  //This variable is populated with a reference to a Sound
  // object later.
  Sound blaster;

  //These variables are populated with information about
  // the background image later.
  Image background = null;
  float backgroundWidth;
  float backgroundHeight;

  //This object is used to produce random values for a
  // variety of purposes.
  Random random = new Random();

  //This is the frame rate we would like to see and
  // the maximum frame rate we will allow.
  int targetFPS = 24;
  //----------------------------------------------------//

  public Slick0220(){//constructor
    //Set the title
    super("Slick0220, baldwin");
  }//end constructor
  //----------------------------------------------------//
```

Table 13.4

There is nothing new in Listing 1 (p. 274) , so there should be no need for an explanation beyond the embedded comments.

13.5.2.1 The init method

The **init** method begins in Listing 2 (p. 276) .

Listing 2 . Beginning of the init method.

```
public void init(GameContainer gc)
                                 throws SlickException {

  //Create Image objects that will be used to visually
  // represent the sprites.
  redBallImage = new Image("redball.png");
  greenBallImage = new Image("greenball.png");

  //Create a Sound object.
  blaster = new Sound("blaster.wav");

  //Create a background image and save information
  // about it.
  background = new Image("background01.jpg");
  backgroundWidth = background.getWidth();
  backgroundHeight = background.getHeight();

  //Add a red sprite as the first element in the
  // ArrayList object. This sprite carries the disease
  // into the population.
  //Put it in the center of the game window. Make the
  // direction of motion random. Make the speed of
  // motion (step size)random. Make the size random.
  // Specify a white (do nothing)color filter.
  sprites.add(new Sprite01(
     redBallImage,//image
     backgroundWidth/2.0f,//initial position
     backgroundHeight/2.0f,//initial position
     (random.nextFloat() > 0.5) ? 1f : -1f,//direction
     (random.nextFloat() > 0.5) ? 1f : -1f,//direction
     0.1f+random.nextFloat()*2.0f,//step size
     0.1f+random.nextFloat()*2.0f,//step size
     0.5f+random.nextFloat()*0.5f,//scale
     new Color(1.0f,1.0f,1.0f)));//color filter

  //This is an infected object. Set its life
  // expectancy.
  sprites.get(0).setLife(
          (int)(random.nextFloat()*infectedSpriteLife));
```

Table 13.5

Sick but not yet dead

The only new code in Listing 2 (p. 276) is the call to the **setLife** method at the end. In the earlier module titled Slick0210: Collision detection and sound [3], the **life** property of a sprite was always either 0 or 1. A sprite with a **life** property value of 0 was dead. A sprite with a **life** property value of 1 was alive.

---

[3] http://cnx.org/contents/7fd862dc-efad-463f-beff-1a1276a4b698

This program is more nuanced and uses values other than 0 and 1 for the infected red sprites. A value of 0 still means that a sprite is dead. Any other positive value means that the sprite is sick and dying but not yet dead.

The value assigned to the **life** property for this sprite is a random value between 0 and **infectedSpriteLife** . This is one of the property values that has an impact on the extent to which the disease spreads through the population. The longer an infected sprite lives after becoming infected, the more healthy sprites it will infect and the more aggressive will be the disease.

You can modify this value *(see Listing 1 (p. 274))* and recompile the program to experiment with different values.

Remainder of the init method

The remainder of the **init** method is shown in Listing 3 (p. 277) .

Listing 3 . Remainder of the init method.

```
    //Populate the ArrayList object with green sprites.
    // Make the initial position random. Make the initial
    // direction of motion random. Make the speed
    // (step size) random. Make the size (scale) random.
    // Make the color filter white (do nothing).
    for(int cnt = 0;cnt < numberSprites;cnt++){
      sprites.add(new Sprite01(
        greenBallImage,//image
        backgroundWidth*random.nextFloat(),//position
        backgroundHeight*random.nextFloat(),//position
        (random.nextFloat() > 0.5) ? 1f : -1f,//direction
        (random.nextFloat() > 0.5) ? 1f : -1f,//direction
        random.nextFloat()*maxStepSize,//step size
        random.nextFloat()*maxStepSize,//step size
        1.0f,//scale
        new Color(1.0f,1.0f,1.0f)));//color filter
    }//end for loop

    gc.setTargetFrameRate(targetFPS);//set frame rate

  }//end init
```

Table 13.6

A population of healthy sprites

Listing 3 (p. 277) uses a **for** loop to add **numberSprites** *(see Listing 1 (p. 274))* healthy sprites to the population. This is another property that has an impact on the spread of the disease. Everything else being equal, the more sparse the population, the more difficult it is for the disease to get a foothold in the first place and the more difficult it is for the disease to spread if it does get a foothold.

The frame rate

Listing 3 (p. 277) also sets the frame rate to the value of **targetFPS** *(see Listing 1 (p. 274))* . Note that I slowed this program down to the standard movie frame rate of 24 fps *(as opposed to the typical 60 fps)* mainly because I wanted to run the simulation more slowly. In other words, I wanted it to be possible to see the disease spread through the population. Also, it is a fairly demanding program so it may not run at 60 fps on some machines.

End of the init method

Listing 3 (p. 277) also signals the end of the **init** method.

13.5.2.2 The update method

The **update** method begins in Listing 4 (p. 278) . This is the method where most of the added complexity in this program resides.

Listing 4 . Beginning of the update method.

```
public void update(GameContainer gc, int delta)
                                  throws SlickException{

  //Move all the sprites to their new positions.
  for(int cnt = 0;cnt < sprites.size();cnt++){
    //Get a reference to the current Sprite01 object.
    Sprite01 thisSprite = sprites.get(cnt);
    //Ask the sprite to move according to its
    // properties
    thisSprite.move();

    //Ask the sprite to bounce off the edge if necessary
    thisSprite.edgeBounce(
                      backgroundWidth,backgroundHeight);
  }//end for loop
```

Table 13.7

Nothing new in this code fragment

The is nothing new in the code fragment shown in Listing 4 (p. 278) . The new code begins in Listing 5 (p. 280) .

An overview of the code

Before getting down into the details of the code, I will give you a descriptive overview.

In the outer-most layer, the program uses a **for** loop to examine every sprite in the population looking for red or infected sprites.

When it finds an infected sprite, it decreases the value of its life expectancy. Then it uses an inner **for** loop to test that sprite against every sprite in the population looking for collisions.

Ignore collision with an infected red sprite

If the infected sprite collides with another infected sprite, it ignores the collision and keeps searching the population, looking for collisions with healthy sprites.

Collision with a healthy green sprite

If the infected sprite collides with a healthy *(green)* sprite, it causes that sprite to become *exposed* to the disease and plays a sound effect. *(As you will see later, sprites that are exposed to the disease don't always contract the disease.)*

The state of the population

When the infected sprite has been tested for a collision with every healthy sprite, four kinds of sprites exist in the population:

1. Healthy sprites that have not been exposed to the disease.
2. Healthy sprites that have been exposed to the disease.
3. Infected sprites that still have some remaining life.
4. Infected sprites whose **life** property is less than or equal to zero, meaning that they are dead.

A cleanup pass

An **Iterator** is used to make a cleanup pass through the population.

Exposed sprites are either converted to infected sprites or cleared of the exposure on the basis of a random value that has a maximum value of **probabilityOfInfection** *(see Listing 1 (p. 274))* .

Dead sprites are removed from the population.

The code to accomplish all of this begins with the **for** loop in Listing 5 (p. 280) .

Listing 5 . Process collisions.

```
  //Search for and process collisions between
  // infected (red) sprites and healthy (green)
  // sprites. Declare the green sprite to be exposed to
  // the disease when a collision occurs.
  for(int ctr = 0;ctr < sprites.size();ctr++){
    //Get a reference to the Sprite01 object.
    Sprite01 testSprite = sprites.get(ctr);

    if(testSprite.getImage().equals(redBallImage)){
      //This is an infected sprite. Reduce its life.
      testSprite.setLife(testSprite.getLife() - 1);

      // Do the following for every sprite in the
      // ArrayList object.
      for(int cnt = 0;cnt < sprites.size();cnt++){
        //Get a reference to the Sprite01 object.
        Sprite01 thisSprite = sprites.get(cnt);

        //Test for a collision between this sprite and
        // the infected test sprite.
        boolean collision =
                     thisSprite.isCollision(testSprite);

        //Process a collision if it has occurred.
        // Exclude collisions between the testSprite
        // and itself and with other infected sprites.

        if((collision == true)&&(!thisSprite.getImage().
                                 equals(redBallImage))){

          //A collision has occurred, set exposed to true
          thisSprite.setExposed(true);

          //Play a sound to indicate that a collision
          // has occurred.
          blaster.play();
        }//end if

      }//end for loop
    }//end if on redBallImage
```

Table 13.8

You should have no difficulty matching up the code in Listing 5 (p. 280) with the verbal description given above.

Make a cleanup pass

The code in Listing 6 (p. 282) uses an **Iterator** to make the cleanup pass described above.

Listing 6 . Make a cleanup pass.

```
    //Make a cleanup pass through the ArrayList object
    Iterator <Sprite01> iterB = sprites.iterator();

    while(iterB.hasNext()){
      Sprite01 theSprite = iterB.next();

      //Cause a percentage of the exposed objects to
      // contract the disease. Clear the others.
      if((theSprite.getExposed() == true) &&
         (random.nextFloat() < probabilityOfInfection)){
        theSprite.setImage(redBallImage);
        theSprite.setLife((int)(
                random.nextFloat()*infectedSpriteLife));
        theSprite.setExposed(false);
      }else{
        //Eliminate the effects of the exposure
        theSprite.setExposed(false);
      }//end else

      //Remove dead sprites
      if(theSprite.getLife() <= 0){
        iterB.remove();
      }//end if
    }//end while loop

  }//end outer for loop

}//end update
```

Table 13.9

Once again, you should have no difficulty matching up the code in Listing 6 (p. 282) with the verbal description given above.

13.5.2.3 The render method

The render method is shown in Listing 7 (p. 283) . There is nothing new in Listing 7 (p. 283) .

Listing 7 . The render method.

continued on next page

```
  public void render(GameContainer gc, Graphics g)
                                  throws SlickException{

    //set the drawing mode to honor transparent pixels
    g.setDrawMode(g.MODE_NORMAL);

    //Draw the background to erase the previous picture.
    background.draw(0,0);

    //Draw every sprite in the ArrayList object.
    for(int cnt = 0;cnt < sprites.size();cnt++){
      sprites.get(cnt).draw();
    }//end for loop

    //Display the number of sprites remaining.
    g.drawString(
       "Sprites remaining: " + (sprites.size()),100f,10f);
  }//end render

}//end class Slick0220
```

Table 13.10

13.6 Run the program

I encourage you to copy the code from Listing 8 (p. 285) and Listing 9 (p. 287) . Compile the code and execute it, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

13.7 Summary

In this module, you learned how to write a program that simulates the spread of a fatal communicable disease within a population.

13.8 Conclusion

Although I may come back and add more modules later, for now, this will be the final module in this collection.

The objective of the collection was to explain the anatomy of a game engine. I believe I have accomplished that objective and have also provided sample programs to illustrate the use of the game engine.

It is worth pointing out that **BasicGame** is not the only game engine architecture available with Slick2D. The Slick2D Wiki [4] refers to **BasicGame** as a game container and indicates that several others are available including:

- Applet Game Container
- ApplicationGDXContainer/AndroidGDXContainer

---

[4] http://slick.ninjacave.com/wiki/index.php?title=Main_Page

The documentation also describes a class named **StateBasedGame** , which provides a different anatomy than **BasicGame** . Bucky Roberts provides a series of video tutorials on state based games using Slick2D at http://www.youtube.com/watch?v=AXNDBQfCd08 [5]

13.9 Miscellaneous

This section contains a variety of miscellaneous information.

Housekeeping material

- Module name: Slick0220: Simulating a pandemic
- File: Slick0220.htm
- Published: 02/07/13
- Revised: 06/11/15 for 64-bit
- Revised: 10/03/15

Disclaimers: **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

13.10 Complete program listing

Complete listings of the code discussed in this module are provided in Listing 8 (p. 285) and Listing 9 (p. 287) .

---

[5] http://www.youtube.com/watch?v=AXNDBQfCd08

Listing 8 . Source code for Slick0220.

```
/*Slick0220.java
Copyright 2013, R.G.Baldwin

This program simulates the propagation of a fatal
communicable disease within a population.

A single infected sprite is introduced into a large
population of sprites. The disease is spread by physical
contact with an infected sprite.

You can watch as the disease either spreads and kills the
entire population or spreads for awhile, then recedes and
dies out. Infected sprites are colored red. Healthy
sprites are colored green. A sound is emitted (for drama)
each time there is contact between an infected sprite and
a healthy sprite.

The final outcome is determined both by chance and by
several factors including:

-The maximum life expectancy of an infected sprite
-The maximum probability of infection due to contact with
an infected sprite
-The maximum degree of mobility of both infected and
healthy sprites
-The population density of sprites.

The actual values for the first three factors for each
individual are determined by the maximum value multiplied
by a random number between 0 and 1.0.

Instance variables are provided for each of these factors.
You can modify the values and recompile the program to
experiment with different combinations of the factors.

A good exercise for a student would be to create a GUI
that allows the factors to be entered more easily for
purposes of experimentation.

Tested using JDK 1.7 under WinXP
*********************************************************/

import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;
import org.newdawn.slick.Sound;

import java.util.Random;
import java.util.ArrayList;
```

Table 13.11

.

Listing 9 . Source code for Sprite01.

```
/*Sprite01.java
Copyright 2013, R.G.Baldwin

An object of this class can be manipulated as a sprite
in a Slick2D program.

Tested using JDK 1.7 under WinXP
*********************************************************/
import org.newdawn.slick.AppGameContainer;
import org.newdawn.slick.BasicGame;
import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Image;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.Color;

public class Sprite01{
  Image image = null;//The sprite wears this image
  float X = 0f;//X-Position of the sprite
  float Y = 0f;//Y-Position of the sprite
  float width = 0f;//Width of the sprite
  float height = 0f;//Height of the sprite
  float xStep = 1f;//Incremental step size in pixels - X
  float yStep = 1f;//Incremental step size in pixels - Y
  float scale = 1f;//Scale factor for draw method
  Color colorFilter = null;//Color filter for draw method

  float xDirection = 1.0f;//Move to right for positive
  float yDirection = 1.0f;//Move down for positive

  int life = 1;//Used to control life or death of sprite

  boolean exposed = false;//Used in the contagion program

  //Constructor
  public Sprite01(Image image,//Sprite wears this image
                  float X,//Initial position
                  float Y,//Initial position
                  float xDirection,//Initial direction
                  float yDirection,//Initial direction
                  float xStep,//Initial step size
                  float yStep,//Initial step size
                  float scale,//Scale factor for drawing
                  Color colorFilter)
                    throws SlickException {

      //Save incoming parameter values
      this.image = image;
      this.X = X;
      this.Y = Y;
      this.xDirection = xDirection;
      this.yDirection = yDirection;
      this.xStep = xStep;
```

Table 13.12

-end-

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *Anatomy of a Game Engine*
Edited by: R.G. (Dick) Baldwin
URL: http://cnx.org/content/col11489/1.13/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Slick0100: Getting started with the Slick2D game library"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45726/1.9/
Pages: 1-14
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Slick0110: Overview"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45728/1.3/
Pages: 15-31
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Slick0120: Starting your program"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45732/1.3/
Pages: 33-43
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Slick0130: The game loop"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45733/1.4/
Pages: 45-63
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Slick0140: A first look at Slick2D bitmap graphics"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45737/1.4/
Pages: 65-90
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Slick0150: A first look at sprite motion, collision detection, and timing control"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45738/1.4/
Pages: 91-124
Copyright: R.G. (Dick) Baldwin
License: http://creativecommons.org/licenses/by/4.0/

Module: "Slick0160: Using the draw and drawFlash methods."
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45748/1.5/
Pages: 125-151

Module: "Slick0170: Mouse and keyboard input"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45750/1.4/
Pages: 153-164

Module: "Slick0180: Sprite sheet animation, part 1"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45751/1.3/
Pages: 165-181

Module: "Slick0190: Sprite sheet animation, part 2"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45753/1.3/
Pages: 183-200

Module: "Slick0200: Developing a sprite class"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45754/1.3/
Pages: 201-224

Module: "Slick0210: Collision detection and sound"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45755/1.5/
Pages: 225-257

Module: "Slick0220: Simulating a pandemic"
By: R.G. (Dick) Baldwin
URL: http://cnx.org/content/m45757/1.6/
Pages: 259-288

Anatomy of a Game Engine
Learn about the anatomy of a basic game engine written around the Slick2D Java game library.

About OpenStax-CNX