Creative Greenfoot

The worlds of computers and art are merging in new and complex ways. Greenfoot is designed to ease the transition into this immerging field, allowing for the easy development of two-dimensional graphical applications, such as simulations and interactive games.

This book provides you with the knowledge and processes necessary to create engaging interactive applications in Greenfoot. It systematically covers essential technologies and algorithms used in creative programming, such as collision detection, easing, and particle effects. You will not only learn to use best practices, but also learn about how these practices were developed and how to augment them to suit your individual needs. After learning the ins and outs of designing and implementing interactive applications, you will see how to interface gamepad controllers to provide an unparalleled immersive experience.

The book will take you through the advanced, creative, and engaging functions of Greenfoot while teaching object-oriented Java.

Who this book is written for

This book is for coding students and Java programmers of all levels interested in building engaging, interactive applications with Greenfoot. Familiarity with the very basics of Greenfoot is assumed.

What you will learn from this book

- Use various methods of animation to breathe life into all aspects of a game or simulation
- Evaluate the different methods of collision detection and choose the right method for the task at hand
- Design and implement scrolling and mapped environments
- Incorporate simple techniques to give the illusion of intelligence to actors in Greenfoot scenarios
- Create user-friendly and intuitive user interfaces
- Heighten the user’s experience by providing gamepad support for your Greenfoot scenarios

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Collision Detection'
- A synopsis of the book’s content
- More information on Creative Greenfoot

About the Author

Michael Haungs is a professor at California Polytechnic State University, San Luis Obispo, where he teaches and conducts research in game design, game development, web application development, and distributed systems. He received his bachelor's degree in science in industrial engineering and operations research from UC Berkeley, his master's degree in science in computer science from Clemson University, and his PhD from UC Davis. He is the author of PolyXpress (http://mhaungs.github.io/PolyXpress)—a system that allows the writing and sharing of location-based stories. Haungs is actively involved in curriculum development and undergraduate education. Through industry sponsorship, he has led several K-12 outreach programs to inform and inspire both students and teachers about opportunities in computer science. Haungs is also a co-director of the liberal arts and engineering studies (LAES) program. LAES is a new, multidisciplinary degree offered jointly by the College of Liberal Arts and the College of Engineering at Cal Poly and represents a unique focus on graduating creative engineers.
Creative Greenfoot

This book is designed to help you learn how to program games and other interactive applications quickly using a learn-by-doing approach. Unlike other texts, which start with a detailed description of all aspects of a language or development platform, we will only cover exactly what is needed for the task at hand. As you progress through the book, your programming skill and ability will grow as you learn topics such as animation, collision detection, artificial intelligence, and game design. Project-based learning is a proven approach and becoming prominent in primary, secondary, and higher education. It enhances the learning process and improves knowledge retention.

The topics presented in this book closely follow the ones I cover in my game design class. Through years of teaching this material, I have found that a project-based learning approach can quickly get students successfully programming and creating interesting games and applications. I hope that you too will be amazed with how much you can accomplish in a short amount of time.

We will code our games in Java. Java is one of the most popular and powerful programming languages in the world and is widely used in the finance industry, gaming companies, and research institutions. We will be doing our programming in Greenfoot (www.greenfoot.org)—an interactive Java development environment. This environment allows both novice and experienced programmers to quickly create visually appealing applications. It provides a safe environment for experimentation and allows you to share your work on a variety of platforms.

To get the most out of this book, you should:

- Open Greenfoot and code as you are reading the book
- Experiment with the code you have after completing a chapter
- Know that some details not covered in a chapter will be addressed in an upcoming chapter
- Be proud of your accomplishments and share them with friends, family, and the Greenfoot community

Learning is not a passive activity. Dig into each chapter and experiment, add your own unique twists, and then code something uniquely your own. I can't wait to see what you can do.
What This Book Covers

Chapter 1, Let's Dive Right in..., takes you through a complete tutorial for creating a simple game complete with an introduction screen, game over screen, a score, mouse input, and sound. This tutorial serves the purpose of introducing you to Greenfoot basics, Java basics, and good programming practices.

Chapter 2, Animation, discusses how to perform animation in Greenfoot. Animation requires appropriate and well-timed image swapping as well as realistic movement around the screen. After reading the given topic and seeing an example, you will apply learned animation techniques to the game you created in Chapter 1, Let's Dive Right in....

Chapter 3, Collision Detection, discusses why collision detection is necessary for most simulations and games. You will learn how to use Greenfoot's built-in collision detection mechanisms and then learn more accurate methods to do collision detection. You will use both border-based and hidden-sprite methods of collision detection to create a zombie invasion simulation.

Chapter 4, Projectiles, talks about how actors in creative Greenfoot often have movement that can best be described as being launched. A soccer ball, bullet, laser, light ray, baseball, and firework are examples of this type of object. You will learn how to implement this type of propelled movement. You will also learn how gravity, if present, affects it by working through the implementation of a comprehensive platform game.

Chapter 5, Interactive Application Design and Theory, discusses creating engaging and immersive experiences in Greenfoot, which is far more involved than compiling a collection of programming effects into one application. In this chapter, you will learn how to engage your user by understanding the relationship between user choice and outcome, conditioning the user, and including the right level of complexity into your work. You will be shown a proven iterative development process that helps you put the theory into practice.

Chapter 6, Scrolling and Mapped Worlds, discusses how to create worlds that are much more extensive than the ones that can fit into the confines of a single screen. At the beginning of the chapter, you will code a scrolling exploration game and by the end of the chapter you will expand it into a large mapped game.

Chapter 7, Artificial Intelligence, talks about how AI, despite being a deep and complex topic, has some simple techniques you can learn to give the illusion of having intelligent, autonomous actors in your worlds. First, you will learn how to effectively use random behaviors. Next, you will implement simple heuristics to simulate intelligent behavior. Last, you will learn the A* search algorithm to allow game actors to intelligently bypass obstacles when moving between two locations on the screen.
Chapter 8, User Interfaces, discusses adding an interface to your Greenfoot scenarios. In this chapter, you will learn how to communicate with your user through buttons, labels, menus, and a heads-up display.

Chapter 9, Gamepads in Greenfoot, discusses the capabilities of a gamepad device and then teaches you how to set up Greenfoot to work with it. You will then add gamepad support to the game we created in Chapter 1, Let's Dive Right in..., and Chapter 2, Animation.

Chapter 10, What to Dive into Next..., gives you an opportunity to reflect on the skills you learned during the course of this book. I then go on to suggest projects you should attempt in order to continue your journey as a programmer and interactive application author.
Collision Detection

"Live as if you were to die tomorrow. Learn as if you were to live forever."

– Mahatma Gandhi

Often, you will need to determine whether two or more objects are touching in Greenfoot. This is known as collision detection and it is necessary for most simulations and games. Detection algorithms range from simple bounding-box methods to very complex pixel color analysis. Greenfoot provides a wide variety of simple methods to accomplish collision detection; you were introduced to some of them in Chapter 1, Let’s Dive Right in…, and Chapter 2, Animation. In this chapter, you will learn how to use Greenfoot’s other built-in collision detection mechanisms and then learn more accurate methods to use them to do collision detection. While pixel-perfect collision detection is beyond the scope of this book, the border-based and hidden-sprite methods of collision detection will be sufficient for most Greenfoot applications. The topics that will be covered in this chapter are:

• Greenfoot built-in methods
• Border-based methods
• Hidden-sprite methods

We will take a break from working on Avoider Game and use a simple zombie invasion simulation to illustrate our collision detection methods. Zombies seem apropos for this chapter. Judging from his quote above, I think Gandhi wanted you to learn as if you were a zombie.
ZombiInvasion interactive simulation

In Chapter 1, Let’s Dive Right in… and Chapter 2, Animation, we went step by step in building Avoider Game and ended up with playable versions of the game by the end of each chapter. In the zombie simulation, we will watch a horde of zombies break through a wall and make their way to the homes on the other side. The user will be able to interact with the simulation by placing explosions in the simulation, that will destroy both types of zombies and the wall. For our zombie simulation, I am going to supply most of the code in the beginning, and we will concentrate our efforts on implementing collision detection. All the code supplied uses concepts and techniques we covered in the last two chapters, and it should look very familiar. We will just provide an overview discussion of the code here. Figure 1 provides a picture of our scenario.

Let’s create a new scenario called ZombieInvasion and then incrementally add and discuss the World subclass and Actor subclasses. Alternatively, you can download the initial version of ZombieInvasion at: http://www.packtpub.com/support
Dynamically creating actors in ZombieInvasionWorld

This class has two main responsibilities: placing all the actors in the world and creating an explosion whenever the mouse is clicked. For the most part, the user will just observe the scenario and will only be able to interact with it by creating explosions. The ZombieInvasionWorld class is rather simple because we are creating an interactive simulation and not a game. Here's the code to accomplish this:

```java
import greenfoot.*;

public class ZombieInvasionWorld extends World {
    private static final int DELAY = 200;
    int bombDelayCounter = 0; // Controls the rate of bombs

    public ZombieInvasionWorld() {
        super(600, 400, 1);
        prepare();
    }

    public void act() {
        if( bombDelayCounter > 0 ) bombDelayCounter--;
        if( Greenfoot.mouseClicked(null) && (bombDelayCounter == 0) ) {
            MouseInfo mi = Greenfoot.getMouseInfo();
            Boom pow = new Boom();
            addObject(pow, mi.getX(), mi.getY());
            bombDelayCounter = DELAY;
        }
    }

    private void prepare() {
        int i,j;
        for( i=0; i<5; i++ ) {
            Wall w = new Wall();
            addObject(w, 270, w.getImage().getHeight() * i);
        }
        for( i=0; i<2; i++ ) {
            for( j=0; j<8; j++ ) {
                House h = new House();
                addObject(h, 400 + i*60, (12 +h.getImage().getHeight()) * j);
            }
        }
    }
}
```
When you right-click on the scenario screen and choose **Save the World** in the pop-up menu, Greenfoot will automatically create the `prepare()` method for you and will supply the appropriate code to add each `Actor` on the screen. This creates the initial state of your scenario (the one the user sees when they first run your scenario). In `ZombieInvasionWorld`, we are manually implementing the `prepare()` method and can do so in a more compact way than Greenfoot. We use for-loops to add our actors. Via this method, we add `Wall`, `House`, `Zombie1`, and `Zombie2`. We will implement these classes later in the chapter.

The `act()` method is responsible for listening for mouse-click events. If the mouse is clicked, we add a `Boom` object at the current position of the mouse. Boom is an actor we create just to display the explosion, and we want it placed exactly where the mouse was clicked. We use a delay variable, `boomDelayCounter`, to prevent the user from rapidly creating too many explosions. Remember, we explained delay variables in detail in the previous chapter, *Chapter 2, Animation*. If you want the user to have the ability to rapidly create explosions, then simply remove the delay variable.

### Creating obstacles

We will create two obstacles for our zombie horde: houses and walls. In the simulation, the `House` object has no functionality. It is simply an obstacle to zombie actors:

```java
import greenfoot.*;

public class House extends Actor {
}
```

The code for the `House` class is extremely simple. Its sole purpose is just to add an image (`buildings/house-8.png`) of a house to `Actor`. It has no other functionality.
Walls are more complex than houses. As the zombies beat on the walls, they slowly crumble. The majority of the code for the Wall class implements this animation, as shown in the following code:

```java
import greenfoot.*;
import java.util.*;

public class Wall extends Actor {
    int wallStrength = 2000;
    int wallStage = 0;

    public void act() {
        crumble();
    }

    private void crumble() {
        // We will implement this in the next section...
    }
}
```

The implementation of the animation of the Wall class crumbling is very similar to that of the Avatar class taking damage that we looked at, in the previous chapter, Chapter 2, Animation. The interesting code is all contained in the crumble() method, which is called repeatedly from the act() method. Figure 1 shows the walls in various states of decay. We will implement and explain in detail the crumble() method in the Detecting a collision with multiple objects section.

**Creating our main actor framework**

The Zombie class contains all the code that describes the behavior for zombies in our simulation. Zombies continually lumber forward trying to get to the humans in the houses. They beat on and eventually destroy any walls in the way, as shown in the following code:

```java
import greenfoot.*;
import java.util.*;

public class Zombie extends Actor {
    int counter, stationaryX, amplitude;

    protected void addedToWorld(World w) {
        stationaryX = getX();
        amplitude = Greenfoot.getRandomNumber(6) + 2;
    }
```
The two important methods in this class are `shake()` and `canMarch()`. The `shake()` method implements the back-and-forth lumbering movement of the zombies. It calls `setLocation()` and leaves the `y` coordinate unchanged. It changes the `x` coordinate to have sinusoidal motion (back and forth). The distance it moves back and forth is defined by the `amplitude` variable. This type of motion was also used by one of the power-downs described in Chapter 2, *Animation* and is shown in Figure 2.

![Figure 2: This is an illustration of using a sine wave to produce back and forth motion in Zombie objects.](image)

We start with a standard sine wave (a), rotate it 90 degrees (b), and reduce the amount we move in the `y` direction until the desired effect is achieved (not moving in the `y`-direction). Callouts (c) and (d) show the effects of reducing movement in the `y` direction.
We will fully implement and explain `canMarch()` in the *Detecting a collision with multiple objects* section. The method `canMarch()` checks surrounding actors (houses, walls, or other zombies) to see whether any are in the way of the zombie moving forward. As a temporary measure, we insert the following line at the end of `canMarch()`:

```java
    return false;
```

This allows us to compile and test the code. By always returning `false`, the `Zombie` objects will never move forward. This is a simple placeholder, and we will implement the real response later in the chapter.

We have two subclasses of the `Zombie` class: `Zombie1` and `Zombie2`:

```java
    public class Zombie1 extends Zombie {
    
    }
    public class Zombie2 extends Zombie {
    
    }
```

This allows us to have two different-looking zombies but only write code for zombie behavior once. I chose to have a blue (*people/ppl1.png*) zombie and a yellow-orange (*people/ppl3.png*) zombie. If you have any artistic skill, you might want to create your own PNG images to use. Otherwise, you can continue to use the images provided with Greenfoot, as I have done.

### Creating an explosion

Here is the implementation of the `Boom` class we discussed previously in the description of the `ZombieInvasionWorld` class. The `Boom` class will immediately draw an explosion that will wipe out everything contained in the blast and then linger for a short time before disappearing. We create an explosion using the following code:

```java
import greenfoot.*;
import java.awt.Color;
import java.util.List;

public class Boom extends Actor {
    private static final int BOOMLIFE = 50;
    private static final int BOOMRADIUS = 50;
    int boomCounter = BOOMLIFE;

    public Boom() {
        GreenfootImage me = new GreenfootImage
            (BOOMRADIUS*2,BOOMRADIUS*2);

```
Collision Detection

```java
me.setColor(Color.RED);
me.setTransparency(125);
me.fillOval(0, 0, BOOMRADIUS * 2, BOOMRADIUS*2);
setImage(me);
}

public void act() {
    if (boomCounter == BOOMLIFE)
        destroyEverything(BOOMRADIUS);
    if (boomCounter-- == 0 ) {
        World w = getWorld();
        w.removeObject(this);
    }
}

private void destroyEverything(int x) {
    // We will implement this in the next section...
}
```

Let's discuss the constructor (Boom()) and act() methods. The Boom() method creates an image manually using the drawing methods of GreenfootImage. We used these drawing methods in this way to draw the stars and eyes in AvoiderGame, which we presented over the last two chapters, Chapter 1, Let's Dive Right in..., and Chapter 2, Animation. The constructor concludes by setting this new image to be the image of the actor using setImage().

The act() method has an interesting use of a delay variable. Instead of waiting for a certain amount of time (in terms of the number of calls to the act() method) before allowing an event to occur, the boomCounter delay variable is used to control how long this Boom object lives. After a short delay, the object is removed from the scenario.

We will discuss the implementation of the destroyEverything() method in a later section.

Test it out

You should now have a nearly complete zombie invasion simulation. Let's compile our scenario and make sure we eliminate any typos or mistakes introduced while adding the code. The scenario will not do much. The zombies will lumber back and forth but not make any forward progress. You can click anywhere in the running scenario and see the Boom explosion; however, it won't destroy anything yet.
Let's make this scenario a bit more interesting, using Greenfoot's collision detection methods.

**Built-in collision detection methods**

We are going to go through all the methods provided by Greenfoot to do collision detection. First, we will go over some methods and discuss their intended use. Then, we'll discuss the remaining methods in the context of more advanced collision detection methods (border-based and hidden-sprite). We have already used a few collision detection methods in the implementation of Avoider Game. We will only briefly describe those particular methods here. Finally, we will not discuss `getNeighbors()` and `intersects()`, as those methods are only useful for Greenfoot scenarios that contain worlds that are created with a cell size greater than one.

**Cell size and Greenfoot worlds**

Until now, we have only created worlds (AvoiderWorld and ZombieInvasionWorld) that have set the `cellSize` parameter of the World constructor to 1. The following is an excerpt from Greenfoot's documentation on the World class:

```java
public World(int worldWidth, int worldHeight, int cellSize)

Construct a new world. The size of the world (in number of cells) and the size of each cell (in pixels) must be specified.
```

Parameters:
- `worldWidth` - The width of the world (in cells).
- `worldHeight` - The height of the world (in cells).
- `cellSize` - Size of a cell in pixels.

The simple tutorials provided on Greenfoot's website mainly use large cell sizes. This makes game movement, trajectories, and collision detection very simple. We, on the other hand, want to create more flexible games that allow for smooth motion and more realistic animation. Therefore, we define our game cells to be 1 x 1 pixels (one pixel) and, correspondingly, will not discuss methods that target worlds with large cell sizes, such as `getNeighbors()` and `intersects()`.

As we go through our discussion, remember that we will, at times, add code to our ZombieInvasion scenario.
Collision Detection

Detecting a collision with a single object

The method `getOneIntersectingObject()` is great for simple collision detection and often used to see whether a bullet, or other type of enemy, hit the main protagonist of the game in order to subtract health, subtract life, or end the game. This is the method we used and explained in Chapter 1, Let’s Dive Right in..., to build our first working version of Avoider Game. We will not discuss it again here and only mention it in the next section as a means to illustrate the use of `isTouching()` and `removeTouching()`.

`isTouching()` and `removeTouching()`

The following is a common pattern for using `getOneIntersectingObject()`:

```java
private void checkForCollisions() {
    Actor enemy = getOneIntersectingObject(Enemy.class);
    if( enemy != null ) { // If not empty, we hit an Enemy
        AvoiderWorld world = (AvoiderWorld) getWorld();
        world.removeObject(this);
    }
}
```

We used this basic pattern in Avoider Game several times. The `isTouching()` and `removeTouching()` methods provide a more compact way to implement the preceding pattern. Here is an equivalent function using `isTouching()` and `removeTouching()` instead of `getOneIntersectingObject()`:

```java
private void checkForCollisions() {
    if( isTouching(Enemy.class) ) {
        removeTouching(Enemy.class);
    }
}
```

If all you’re doing is removing an object that the object intersects with, all you need is the `isTouching()` and `removeTouching()` methods. However, if you want to do something with the object that you’re intersecting with, which requires calling methods of the object’s class, then you need to store the intersected object in a named variable, which requires using the `getOneIntersectingObject()` method.

In general, always use `getOneIntersectingObject()` instead of `isTouching()` and `removeTouching()`. It is more flexible and provides code that is easier to extend in the future.
Detecting a collision with multiple objects

The collision detection method `getIntersectingObjects()` returns a list of all the actors in a given class that the calling actor is touching. This method is needed when you need to take an action on every object that is touching a specific actor, or you need to change the state of an actor based on the number of objects touching it. When using `getOneIntersectingObject()`, you are only concerned about being touched by at least one object of a given type. For example, in the game PacMan, you lose a life anytime you touch a ghost. It wouldn't matter if you ran into one, two, or three; the end result would be the same—you'd lose a life. However, in our zombie simulation, the `Wall` actors take damage based on how many zombies are presently beating on it. This is a perfect use for `getIntersectingObjects()`.

In the `Wall` code presented above, we left out the implementation of the `crumble()` method. Here is that code:

```java
private void crumble() {
    List<Zombie> army = getIntersectingObjects(Zombie.class);
    wallStrength = wallStrength - army.size();
    if( wallStrength < 0 ) {
        wallStage++;
        if( wallStage > 4 ) {
            World w = getWorld();
            w.removeObject(this);
        } else {
            changeImage();
            wallStrength = 2000;
        }
    }
}

private void changeImage() {
    setImage("brick"+wallStage+".png");
}
```

Let's quickly go over the things we saw before. In the Hurting the avatar section of Chapter 2, Animation, we changed the image of the avatar to look damaged every time it was touched by an enemy. We are using the same animation technique here to make it look like walls are taking damage. However, in this code, we have given walls a durability property that is defined by the `wallStrength` variable. The value of `wallStrength` determines how many times a wall can be hit by a zombie before it visibly looks more crumbled and cracked.
Collision Detection

The `wallStrength` variable is actually just an example of a delay variable that we discussed in the previous chapter, *Chapter 2, Animation*. Instead of this variable delaying a certain amount of time, it is delaying a certain number of zombie hits. When `wallStrength` is less than 0, we change the image using the method `changeImage()` unless this is the fourth time we have crumbled, which will cause us to remove the wall altogether. *Figure 3* shows the wall images I created and used for this animation.

![Figure 3: These are the four images used to animate the walls crumbling](image)

Now, let's discuss the collision detection method `getIntersectingObjects()`. When called, this method will return all objects of a given class that intersect with the calling object. You specify the class of objects you are interested in by providing it as the argument to this method. In our code, I provided the argument `Zombie.class`, so the method would only return all the zombies that are touching the wall. Because of inheritance, we will get all of the `Zombie1` objects and all of the `Zombie2` objects that intersect. You can access, manipulate, or iterate through the objects returned using the methods defined in the `List` interface. For our purposes, we only wanted to count how many zombies we collided with. We get this number by calling the `size()` method on the `List` object returned from `getIntersectingObjects()`.

**Java interfaces and List**

The collision detection method `getIntersectingObjects()` introduces us for the first time to the `List` interface. In Java, interfaces are used to define a certain set of methods that two or more classes will have in common. When Java classes implement an interface, that class is promising that it implements all of the methods defined in that interface. So, the collection of `Actor` objects returned by `getIntersectingObjects()` could be stored in an array, linked list, queue, tree, or any other data structure. Whatever the data structure used for storing these objects, we know that we can access those objects via the methods defined in the `List` interface, such as `get()` or `size()`.

For more information, refer to the following link: [http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html](http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html).
In our ZombieInvasion simulation, we need to use getIntersectingObjects() one more time. Earlier, we left the implementation of canMarch() incomplete when we looked at the code for the Zombie class. Let's implement that method now using getIntersectingObjects(). Here is the code:

```java
private boolean canMarch() {
    List<Actor> things = getIntersectingObjects(Actor.class);
    for( int i = 0; i < things.size(); i++ ) {
        if( things.get(i).getX() > getX() + 20 ) {
            return false;
        }
    }
    return true;
}
```

This method checks whether or not there are any actors in the way of this object moving forward. It accomplishes this by first getting all objects of the Actor class that are touching it and then checking each one to see if it is in front of this object. We do not care if Actor is touching the calling object at the top, bottom, or back as these actors will not prevent this object from moving forward. This line of code in canMarch() gives us the list of all intersecting actors:

```java
    List<Actor> things = getIntersectingObjects(Actor.class);
```

We then iterate through the list of actors using a for loop. To access an item in a list, you use the get() method. The get() method has one formal parameter that specifies the index of the object in the list that you want. For each actor in the list, we check to see if the x coordinate is in front of us. If it is, we return false (we can't move); otherwise, we return true (we can move).

We have added the implementation of the crumble() method to the Wall class (don't forget to add changeImage() too) and the implementation of canMarch() to the Zombie class. Let's compile our scenario and observe what happens. Our simulation is almost complete. The only thing missing is the implementation of the destroyEverything() method in the Boom class. We will look at that implementation next.
Detecting multiple objects in range

The last method we need to implement to complete our simulation is destroyEverything(). In this method, we will use the Greenfoot collision detection method getObjectsInRange(). This method takes two parameters. We have seen the second parameter in all of the rest of the collision detection methods, and it specifies the class of actors we are testing for collision. The first parameter provides the radius of a circle drawn around the actor that defines where to search for collision. Figure 4 shows the relationship between the radius parameter and the search area. Unlike getIntersectingObjects(), getObjectsInRange() returns a list of actors that are within the range specified by the calling object.

Now that we know about the method getObjectsInRange(), let’s look at the implementation of destroyEverything():

```
private void destroyEverything(int x) {
    List<Actor> objs = getObjectsInRange(x, Actor.class);
    World w = getWorld();
    w.removeObjects(objs);
}
```

This method is short, yet powerful. It calls getObjectsInRange() with a radius x, the value that was passed to destroyEverything() when called, and Actor.class, which in Greenfoot terms means everything. All objects within the circle defined by the radius will be returned by getObjectsInRange() and stored in the objs variable. Now, we could iterate through all the objects contained in objs and remove them one at a time. Luckily, Greenfoot provides a function that can remove a list of objects with one call. Here’s its definition in Greenfoot’s documentation:

```
public void removeObjects(java.util.Collection objects)
Remove a list of objects from the world.

Parameters:
objects - A list of Actors to remove.
```
Time to test it out

The simulation is complete. Compile and run it and make sure everything works as anticipated. Remember, you can click anywhere to blow up buildings, walls, and zombies. Reset the scenario and move things around. Add walls and zombies and see what happens. Nice work!

Border-based collision detection methods

Border-based collision detection involves incrementally searching outward from \texttt{Actor} until either a collision is detected, or it is determined there are no obstacles in the way. The method finds the edge (or border) of the item collided with. This method is especially useful when objects need to bounce off each other or one object is landing on another and needs to remain on that object for a certain amount of time, for example, when a user-controlled \texttt{Actor} is jumping on a platform. We will introduce this method of collision detection in this chapter, as well as use it in upcoming chapters.

Detecting single-object collisions at an offset

The \texttt{at offset} versions of Greenfoot's collision detection methods are well suited to border-based collision detection. They allow us to check for a collision at a certain distance, or offset, from the center of the calling \texttt{Actor}. To demonstrate the use of this method, we will change the implementation of the \texttt{canMarch()} method in the \texttt{Zombie} class. Here is our revised version:

```java
private boolean canMarch() {
    int i=0;
    while(i<=step) {
        int front = getImage().getWidth()/2;
        Actor a = getOneObjectAtOffset(i+front, 0, Actor.class);
        if( a != null ) {
            return false;
        }
        i++;
    }
    return true;
}
```
Collision Detection

Typically, when an actor moves, it will change its position by a certain number of pixels. In the Zombie class, how far zombies will move, if they can, is stored in the step variable. We need to declare and initialize this instance variable by inserting the following line of code at the top of the Zombie class, as follows:

```
private int step = 4;
```

Using a step variable to store the length of movement for an actor is common practice. In the implementation of canMarch() above, we check each pixel in front of a zombie up to and including taking a full step. This is handled by the while loop. We increment the variable i from 0 to step, checking for a collision each time at the location i + front. Since the origin location of an object is its center, we set front to be half of the width of the image representing this actor. Figure 5 illustrates this search.

![Figure 5: Using border-based detection, an object searches for a collision one pixel at a time. It starts from its front and then searches for an object starting at front + 0 all the way to front + step.](image)

If we detect a collision any time in our while loop, we return false, indicating the actor cannot move forward; otherwise, we return true. Test out this new version of canMarch().

Detecting multiple-object collisions at an offset

The collision detection method getObjectsAtOffset() is very similar to getOneObjectAtOffset(). It just, as the name implies, returns all actors that collide at the given offset. To demonstrate its use, we are going to re-implement canMarch() as we did for getOneObjectAtOffset(). To take advantage of getting a list of actors that collide, we are going to add some additional functionality to canMarch(). For each actor blocking the forward movement of the zombie, we are going to shove them a little.
Here's the implementation of `canMarch()`:

```java
private boolean canMarch() {
    int front = getImage().getWidth()/2;
    int i = 1;
    while(i<=step) {
        List<Actor> a = getObjectsAtOffset(front+i,0,Actor.class);
        if( a.size() > 0 ) {
            for(int j=0;j<a.size()&&a.get(j) instanceof Zombie;j++){
                int toss = Greenfoot.getRandomNumber(100)<50 ? 1 : -1;
                Zombie z = (Zombie) a.get(j);
                z.setLocation(z.getX(),z.getY()+toss);
            }
            return false;
        }
        i++;
    }
    return true;
}
```

In this version, we use a `while` loop and `step` variable in much the same way we did previously for the `getOneObjectAtOffset()` version of `canMarch()`. Inside the `while` loop is where we added the new "shoving" functionality. When we detect that there is at least one `Actor` in the list, we iterate through the list using a `for` loop to slightly push each actor we collided with. The first thing we do in the `for` loop is check whether or not the `Actor` class is a `Zombie` class using the `instanceof` operator. If it isn't, we skip over it. We don't want the ability to shove `Wall` or `House`. For each zombie we collided with, we set the `toss` variable to `1` or `-1` with equal probability. We then move that zombie with `setLocation()`. The effect is interesting and gives the illusion that the zombies are trying to push and shove their way to the front. Compile and run the scenario with the changes to `canMarch()` and see for yourself. Figure 6 shows how the zombies bunch up with the preceding changes.
Collision Detection

The instanceof Operator

Java's instanceof operator checks whether the left-hand side argument is an object created from the class (or any of its subclasses) specified on the right-hand side. It will return true if it is and false otherwise. It will also return true if the left-hand side object implements the interface specified on the right-hand side.

Figure 6: Here's a view of the zombies pushing and shoving to get to the humans in the houses first

Hidden-sprite collision detection methods

One flaw with the getOneObjectAtOffsets() and getObjectsAtOffset() methods is that they only check the granularity of a single pixel. If an object of interest is one pixel above or below the offset provided to these methods, then no collision will be detected. In fact, in this implementation, if you allow the simulation to run until the zombies reach the houses, you'll notice that some zombies can move past the houses. This is because the pixel-only check fails between houses. One way to handle this deficiency is to use hidden-sprite collision detection. Figure 7 illustrates this method.
In the hidden-sprite method, you use another Actor class to test for collisions. *Figure 7* shows a Zombie object using a smaller, auxiliary Actor class to determine if a collision occurred with the flower. While the hidden sprite is shown as a translucent red rectangle, in practice, we would set the transparency (using `setTransparency()`) to 0, so that it would not be visible. The hidden-sprite method is very flexible because you can create any shape or size for your hidden sprite, and it does not have the problem of only looking at a single pixel that the two previous collision detection methods had. Next, we will once again change the `canMarch()` method in the Zombie class, this time using hidden-sprite collision detection.

The first thing we need to do, is create a new Actor that will serve as the hidden sprite. Because we are going to use this hidden sprite for zombies, let's call it `ZombieHitBox`. Create this subclass of Actor now and do not associate an image with it. We will draw the image in the constructor. Here is the implementation of `ZombieHitBox`:

```java
import greenfoot.*;
import java.awt.Color;
import java.util.*;

public class ZombieHitBox extends Actor {
    GreenfootImage body;
    int offsetX;
    int offsetY;
    Actor host;

    public ZombieHitBox(Actor a, int w, int h, int dx, int dy, boolean visible) {
        host = a;
        offsetX = dx;
        offsetY = dy;
        body = new GreenfootImage(w, h);
        if (visible) {
            body.setColor(Color.red);
        }
    }

    // Other methods...
}
```
Collision Detection

```
// Transparency values range from 0 (invisible)
// to 255 (opaque)
body.setTransparency(100);
body.fill();
}
setImage(body);
}

class ZombieHitBox {
    ZombieHitBox(Actor a, int w, int h, int dx, int dy, boolean visible) {
        GreenfootImage body = new GreenfootImage(w, h);
        body.setColor(Color.BLACK);
        body.setTransparency(100);
        body.fill();
        setImage(body);
    }

    public void act() {
        if( host.getWorld() != null ) {
            setLocation(host.getX()+offsetX, host.getY()+offsetY);
        } else {
            getWorld().removeObject(this);
        }
    }

    public List getHitBoxIntersections() {
        return getIntersectingObjects(Actor.class);
    }
}
```

The constructor for `ZombieHitBox` takes six parameters. The reason it takes so many parameters is that we need to provide the `Actor` class to which it is attached (the `a` parameter), define the size of the rectangle to draw (the `w` and `h` parameters), provide the offset of the rectangle from the provided `Actor` (the `dx` and `dy` parameters), and check whether the hidden sprite is visible (the `visible` parameter). In the constructor, we use `GreenfootImage()`, `setColor()`, `setTransparency()`, `fill()`, and `setImage()` to draw the hidden sprite. We went over these methods previously in Chapter 2, Animation.

We use the `act()` method to ensure that this hidden sprite moves along with the `Actor` class it is attached to (we will call this the `host` actor). To do this, we simply call `setLocation()`, provide the current `x` and `y` position of the `host` actor and shift a little according to the offset values provided in the constructor. Before doing this, however, we check whether the `host` has not been deleted. If it has, we delete the hit box, as it only has meaning in relation to `host`. This handles the case where an explosion destroys `host`, but did not quite reach the hit box.

Finally, we provide one public method that the `host` actor will use to get all the actors that are colliding with the hidden sprite. We named this method as `getHitBoxIntersections()`.
Next, we need to augment the Zombie class to use this new hidden sprite. We need a handle on this hidden sprite, so we need to add a new property to the Zombie class. Insert this line of code under the declaration of the step variable:

private ZombieHitBox zbh;

Next, we need to augment the addedToWorld() method to create and connect ZombieHitBox to Zombie. Here is the implementation of that method:

protected void addedToWorld(World w) {
    stationaryX = getX();
    amplitude = Greenfoot.getRandomNumber(6) + 2;
    zbh = new ZombieHitBox(this, 10, 25, 10, 5, true);
    getWorld().addObject(zbh, getX(), getY());
}

We create a 10 x 25 rectangle for our hidden sprite and initially make it visible, so that we can test it in our scenario. Once you are satisfied with the placement and size of your hidden sprite, you should change the visible parameter of ZombieHitBox from true to false.

Now that we have created, initialized, and placed ZombieHitBox, we can make our changes to canMarch() to demonstrate the use of the hidden-sprite method:

private boolean canMarch() {
    if( zbh.getWorld() != null ) {
        List<Actor> things = zbh.getHitBoxIntersections();
        if( things.size() > 1 ) {
            int infront = 0;
            for(int i=0; i < things.size(); i++ ) {
                Actor a = things.get(i);
                if( a == this || a instanceof ZombieHitBox)
                    continue;
                if( a instanceof Zombie) {
                    int toss =
                    Greenfoot.getRandomNumber(100)<50 ? 1:-1;
                    infront += (a.getX() > getX()) ? 1 : 0;
                    if( a.getX() >= getX() )
                        a.setLocation(a.getX(),a.getY()+toss);
                    } else {
                        return false;
                    }
                }
            if( infront > 0 ) {
                return false;
            } else {
                return true;
            }
        }
    } else {
        return false;
    }
}
Collision Detection

```java
    return true;
}
else {
    getWorld().removeObject(this);
}
return false;
```

Unlike previous implementations of `canMarch()`, we need to first ask the hidden sprite for a list of actors colliding with this zombie. Once we get that list, we check that it has a size greater than one. The reason why it needs to be greater than one, is that `ZombieHitBox` will include the zombie it is attached to. If we are not colliding with any other zombies or actors, we return `true`. If we are colliding with a number of actors, then we iterate through them all and make some decisions based on the type of `Actor`. If `Actor` is this zombie or an instance of `ZombieHitBox`, we skip it and don't take any action. The next check is whether or not `Actor` is an instance of the `Zombie` class. If it isn't, then it is some other object, such as `House` or `Wall`, and we return `false`, so that we will not move forward. If it is an instance of the `Zombie` class, we check whether or not it is in front of this zombie. If it is, we shove it a little (just as we did in the previous implementation of `canMarch()`) and increment the `infront` variable. At the end of iterating through the list of actors, we check the `infront` variable. If there were any zombies in front of this zombie, we return `false` to prevent it from moving forward. Otherwise, we return `true`. The outermost `if` statement simply checks that the hitbox (`zbh`) associated with this object has not been previously destroyed by a `Boom` object. If it has, then we need to remove this object too.

Compile and run this version of the scenario. You should observe that the zombies bunch up nicely, push and shove each other, yet they are not able to move past the houses. Using the hidden-sprite method of collision detection is a bit more complex than the rest, but gives us good accuracy.

Challenge

Okay, we have implemented several forms of collision detection in our zombie simulation. Which method of collision detection do you prefer for this simulation?

For a challenge, create an `Actor` ball that occasionally rolls in from the left and knocks zombies out of the way. If the ball hits `wall`, have it do 1,000 damage to it. Which form of collision detection will you use to detect collisions between the ball and zombies and between the ball and a wall?
Summary
Collision detection is a crucial component of any game, simulation, or interactive application. Greenfoot provides built-in methods of detecting collisions. In this chapter, we carefully explained each of these methods and then demonstrated how you could use them to do more advanced collision detection. Specifically, we discussed border-based and hidden-sprite techniques. Moving forward, we will use collision detection often and will choose a method appropriate for our example. In the next chapter, we will look at projectiles and will have ample opportunity to put into practice what you have learned in this chapter.
Where to buy this book

You can buy Creative Greenfoot from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.