Chapter Three: Game Needs

Section 1: Four Basics

A computer game needs to interact with the player. The game must communicate with the player by accepting player input and presenting the results of that input. Currently Greenfoot does not support mouse interactivity but it does allow keyboard input.

Most games need to detect when two game objects are touching each other. We could compute this by writing our own code, but Greenfoot provides a method to detect this situation.

In games, virtual opponents may be slain, buildings destroyed, or the player-'s avatar may be annihilated. When these events occur, the object in question is removed from the game. The Greenfoot system provides a mechanism for removing objects from a game.

Games often have a random element (e.g. rolling dice or drawing a card). While computers cannot create truly random numbers, they can generate numbers that are "close enough" to being random. Again Greenfoot provides this capability for us.

In this chapter we will discuss each of these four concepts and how they work in Greenfoot.

Section 2: Keyboard Control

Greenfoot allows us to determine if particular keys are currently pressed. The method is called isKeyDown and is part of the Greenfoot class. The method returns a boolean. It returns true if the key is currently pressed and false otherwise. It is invoked as follows:

```
Greenfoot.isKeyDown( keyName )
```

Where keyName is one of the keyboard keys or "up", "down", "left", or "right". Note that the key must be in quotes (e.g. "A"). The up arrow key is designated by "up", the down arrow key is designated by "down", "left" denotes the left arrow key, and "right" denotes the right arrow key.

Example

To make an actor move to the left when the left arrow key is pressed, the following code can be added to the actor's act() method:

```
if ( Greenfoot.isKeyDown( "left" ) )
{
```

```
    setLocation( getX() + 1, getY() );
}
```

Section 3: Collision Detection

Greenfoot can determine if two actors intersect. One method it provides is the getOneIntersectingObject method (which is a method of the Actor class). This method requires a parameter which is a *class*. It will return an actor object if an only if the actor object intersects and if it is an object of the specified class. Note, if there is more than one object that satisfies these two constraints then one of them is returned at random. If there are no intersecting objects that qualify then the method returns the null object. Because of this it is usual to first call getOneIntersectingObject and then use an if statement to determine if there is in fact an intersecting object or not. In a later chapter, we will show how to get a list of all of the intersecting objects and process.

Example

```
Actor theFish = getOneIntersectingObject( Fish.class );
if ( theFish == null )
{
   // There is no intersecting Fish object.
}
else
{
   // There is an intersecting Fish object.
}
```

Notice that the parameter is created using the .class member variable after the name of a class.

The other thing to be aware of is that the method returns an *Actor* object. This means that you cannot invoke methods specific to the Fish class. In order to do that you need to *cast* the returned Actor to a Fish object. A cast operation just converts one data type to another. It is accomplished by placing the class name inside ( ) as follows:

```
GreenFish theFish = (GreenFish) getOneIntersectingObject( GreenFish.class );
if ( theFish == null )
{
   // There is no intersecting GreenFish object.
}
else
{
   // There is an intersecting GreenFish object.
   // We can invoke GreenFish methods here because we cast the Actor to a GreenFish object.
}
```

How Greenfoot Determines Intersections

Determining if two arbitrary objects intersect is a complex operation (trust me). So, Greenfoot simplifies the problem. Greenfoot approximates the actual shape of the objects using rectangle bounding boxes (to be more technical, they are upright or axis aligned rectangular bounding boxes). It then calculates whether these rectangles intersects. If they do, then Greenfoot considers the objects to be intersecting (when in fact the actual shapes might not intersect).

Calculating whether two upright rectangles intersect is a lot easier than determining whether two arbitrary shapes intersect. Assume that there is Rectangle class which has a minimum x and y value and a maximum x and y value as member variables. Whether two rectangles intersect can be answered with the following Rectangle class method:

```
public boolean rectangleIntersect( Rectangle aRectangle )
{
  if ( maximumX >= aRectangle.minimumX )
  {
     if ( minimumX <= aRectangle.maximumX )
     {
        if ( maximumY >= aRectangle.minimumY )
        {
           if ( miminumY <= aRectangle.maximumY )
           {
              return true;
           }
        }
     }
  }
  return false;
}
```

Many game engines use this simplified idea of bounding rectangles for calculating intersections between complex game objects. If the bounding rectangles intersect then the game engine does more complex processing to determine if the objects really do intersect. If the bounding rectangles do not intersect then the more complicated calculation is avoided.

Section 3: Removing Objects

It is fairly easy to remove an object from a Greenfoot World. Just invoke the removeObject method of the World class and pass in the object to be removed as a parameter (in some sense removeObject is the opposite of the addObject method).

The following (when placed in a World subclass definition) will remove an object:

```
removeObject( object );
```

The code is slightly different when used inside the class definition of an Actor subclass because we have to gain access to the current World first:

```
getWorld().removeObject( object );
```

The getWorld method returns the World object to which this Actor has been added.

Section 4: Random Numbers

Often a game needs random numbers: A random direction for a sprite, a random number of flowers growing, sometimes the bee comes after you, sometimes not. Rolling a die is an example of random number generation where the random numbers are 1, 2, 3, 4, 5 or 6. Each of the numbers occurs with equal probability. Greenfoot provides a random number generation method which is used as follows:

```
int randNumber = Greenfoot.getRandomNumber( 6 );
```

This returns a random number 0, 1, 2, 3, 4, or 5 with equal probability. It is like rolling a 6 sided dice where the sides are numbers 0 to 5.

In general you get a random number from 0 to N-1, where N is the parameter passed to getRandomNumber.

Random Walk Example:
This example shows how to make an Actor to move randomly in one of the four cardinal directions:

```
act()
{
  int theDirection = Greenfoot.getRandomNumber( 4 );

  if ( theDirection == 0 )
  {
    moveUp();
  }
  if ( theDirection == 1 )
  {
    moveDown();
  }
  if ( theDirection == 2 )
  {
    moveLeft();
  }
  if ( theDirection == 3 )
  {
    moveRight();
  }
```

P4games.org, copyright 2008

}