Chapter 6
Game Programming I

To this point we've learned to create simple objects such as UFOs on screen and move them around in a controlled manner. Now we'll take a look at the components of basic game programming.

In addition to moving objects, various games may include many other techniques such as:

- detecting mouse clicks and the objects that were clicked on.
- dragging an object with the mouse or having an object automatically following the mouse.
- detecting keystrokes.
- displaying score.
- detecting collision between objects.

In this chapter we'll examine each of these.

6.1 Building Blocks for Game Programming

Game programs share some common building blocks, which we examine now.

6.1.1 the init Method

The init( ) method is part of every Java graphics game. It contains code that must be executed before game play actually begins, typically doing things like coloring the application window, putting objects in place and other setup essentials.

init( ) must be written in this form.

```java
public void init( )
{
    ...statements...
}
```

The init( ) method executes automatically before the run( ) method begins. It is part of standard Java but we must write the code to fill it in.

It is typically used like this.
public class AnyProgram extends GraphicsProgram
{
    GOval target;

    public void init()
    {
        target = new GOval(30, 30, 100, 100);
        target.setFilled(true);
        target.setColor(Color.GREEN);
    } //init
    ...more code ...
} //AnyProgram

Declaring an identifier such as target inside the public class that defines the entire program but outside the init( ) or any other method gives that identifier global scope. Global scope means we can use it in any method in the program, including for example the init( ) method and the run( ) method, without passing it as an argument. Variables with global scope should be used sparingly but are required in this case.

The init( ) method is used to set up the initial game playing conditions — create the targets, display the background, set up the scoring variables, etc. If something needs to be done before the play actually begins, it probably should be done in the init( ) method.

6.1.2 Paying Attention to the Mouse
By default an ACM Java graphics program ignores the mouse. We've already modified this somewhat by using the waitForClick( ) method, which causes the program to freeze until the mouse is clicked on the window.

But we need more. In particular, we often need to know where the mouse was clicked and what object was clicked on. To access all this information, we need to tell the program to pay full attention to the mouse. The standard Java method we need is:

addMouseListeners( );

This method call is usually included in the init( ) method. We might have

public void init()
{
    ...code to initialize variables...
    addMouseListeners( );
} //init
6.1.3 Working with the Mouse

When a mouse button is clicked a record of this event, called a mouse event, is created. It records what mouse activity happened and where it happened in the application window. Mouse events are part of standard Java, which provides the MouseEvent object to hold the necessary information.

Your program needs

```java
import java.awt.event.*;
```

to load the MouseEvent object. We must also write a mousePressed( ) method, which will contain code that automatically executes when the click occurs. The mousePressed( ) method must be written in this form:

```java
public void mousePressed(MouseEvent e)
{
    ...statements...
} //mousePressed
```

You don’t have to name the MouseEvent e. It's just a variable so you could name it anything meaningful. Also, note that the mousePressed( ) method is executed for a click of any mouse button—left, right or center.

6.1.4 An Example Program

This program simply counts mouse clicks, displaying the count of the clicks in the application window.

CountClicks

```java
//CountClicks.java
import acm.program.*;
import acm.graphics.*;
import java.awt.event.*;

public class CountClicks extends GraphicsProgram
{
    int clickCount;
    GLabel countLabel;

    public void init( )
    {
        clickCount = 0;
        countLabel = new GLabel("Click count: " + clickCount);
        countLabel.setFont("Serif-BOLD-24");
        add(countLabel, 5, 30);
    }

    public void mousePressed(MouseEvent e)
    {
        clickCount += 1;
        countLabel = new GLabel("Click count: " + clickCount);
        countLabel.setFont("Serif-BOLD-24");
        add(countLabel, 5, 30);
    }
}
```

initialize count and label for displaying the count
addMouseListeners();
waitForKey();
} //init

public void mousePressed(MouseEvent e)
{
    clickCount++;
    countLabel.setText("Click count: " + clickCount);
} //mousePressed

public void run()
{
    /*nothing to do here */
} //CountClicks

Java doesn’t care about the order of the methods in the source code but it is convenient to put the standard Java methods (init(), mousePressed(), ...) above the run(), and the methods unique to the particular program below the run().

6.1.5 Detecting the Object That Was Clicked On

The program above is not very interesting, partly because it treats all mouse clicks as the same, simply counting each one. Games usually require that we identify the objects we clicked on so that we can move things on screen, shoot the goblins, etc. To do this we need to look more carefully at MouseEvents and the mousePressed() method.

The location of a mouse click

Let us assume a MouseEvent e has already been created. We can then find the window coordinates of the click easily.

double x = e.getX();
double y = e.getY();

The getX() and getY() methods extract the coordinate information from the MouseEvent e, which is then assigned to variables x and y. This gives us the point (x, y) on the application window where the click occurred.

The object that was clicked on

The ACM Java library method getElementAt() returns the object that was clicked on. Using it is simple.

GRect theRectangle = getElemeentAt(x, y);

But in many situations the click could be on a GRect, a GOval or some other object class. Since we often can’t predict the particular class of object that will be clicked on, we usually store the object returned by getElmeentAt() in a GObject, and not in a more specific class such as GRect.
GObject theObject = getElementAt(x, y);

Getting the clicked on object usually happens in the automatically executed mousePressed( ) method.

**Working with null**

Suppose the click didn’t occur on any graphical object within the window. What does the getElementAt( ) method return then? Java uses the special value null to indicate “there’s nothing here” or more specifically “no object” and that’s what is returned by getElementAt( ) when a click occurs in an empty area of the window.

Assume a redBall object already exists. Here’s a basic mousePressed( ) method that counts clicks on this object, with another count for clicks off the object.

```java
public void mousePressed(MouseEvent e) {
    double x = e.getX();
    double y = e.getY();
    GObject theObject = getElementAt(x, y);
    if (theObject == redBall) {
        clicksOnBall++;
    } else if (theObject == null) {
        clicksOffBall++;
    }
} //mousePressed
```

This implementation of mousePressed( ) follows a common pattern.

1) get the coordinates where the click occurred.
2) find out which object was at those coordinates.
3) take appropriate action based on which object was clicked.

### 6.1.6 The Game Loop

Action within a game is usually repetitive—fly the plane until it lands, move the widget until it is the correct place, etc. Thus basic game activity is executed within a loop called the game loop. The game loop may be written as a while(true) break loop and often takes a form as shown below.

```java
while(true) {
    modify objects as needed
    check for interaction of objects and take appropriate action
    if some condition is met then the game is over so break
}
```
Actions that need to be taken frequently, such as moving an object or checking for collision of objects, execute inside the game loop.

### 6.1.7 A Simple Shoot-em-up Game

Clicking on a stationary target is not all that interesting, but if we add movement to the object then we've got a challenging target to shoot at and that's a real game. Not fancy, but we've got to start somewhere.

Below is a simple game that bounces a ball around the window for a fixed time, counting the successful 'shots' at the ball as it moves around the screen. It uses several techniques we've already explored—animation, bouncing, detecting click locations and updating a score.

**ShootTarget1**

```java
//ShootTarget1.java
import acm.program.*;
import acm.graphics.*;
import java.awt.event.*;
import java.awt.*;
public class ShootTarget1 extends GraphicsProgram {
    public static final int APPLICATION_WIDTH = 800;
    public static final int APPLICATION_HEIGHT = 500;
    final int WAIT = 5;
    final int DIAM = 50;
    final int MAX_TIME = 10000;  //10 seconds
    int clicksOnBall, time, redBallXJump, redBallYJump;
    GOval redBall;
    GLabel clicksOnBallLabel;
    public void init() {
        time = 0;
        redBallXJump = 1;
        redBallYJump = 1;
        redBall = new GOval(90, 40, DIAM, DIAM);
        redBall.setFilled(true);
        redBall.setColor(Color.RED);
        add(redBall);
        clicksOnBall = 0;
        clicksOnBallLabel = new GLabel("Ball click count: " + clicksOnBall);
    }
}
```

set up the movement for the target

set up the target

set up the counter

©Daniel L. Schuster, 2009, all rights reserved
clicksOnBallLabel.setFont("*-BOLD-16");
add(clicksOnBallLabel, 0, 16);

displayInstructions();
addMouseListener();
}
//init

public void mousePressed(MouseEvent e)
{
    double x = e.getX();
    double y = e.getY();
    GObject theObject = getElementAt(x, y);

    if (theObject == redBall)
    {
        clicksOnBall++;
        clicksOnBallLabel.setLabel("Ball click count: "+ clicksOnBall);
    }
}
//mousePressed

public void run()
{
    while(true) //game loop
    {
        pause(WAIT);
        time = time + WAIT;
        if (time > MAX_TIME)
        {
            break; //exit the game loop
        }
        if (redBall.getY() <= 0) //at top of window?
        {
            redBallYJump = 1;
        }
        else if (redBall.getY() + DIAM >= APPLICATION_HEIGHT) //at bottom?
        {
            redBallYJump = -1;
        }
        else if (redBall.getX() <= 0) //at left?
        {
            redBallXJump = 1;
        }
        else if (redBall.getX() + DIAM >= APPLICATION_WIDTH) //at right?
        {
            redBallXJump = -1;
        }
    }
}
redBall.move(redBallXJump, redBallYJump);

} //game loop

} //run

public void displayInstructions() {
    GLabel sign1 = new GLabel("Click on the moving ball");
    GLabel sign2 = new GLabel("-- click to start --");
    sign1.setFont("*BOLD-20");
    sign2.setFont("*BOLD-20");
    add(sign1, 320, 200);
    add(sign2, 340, 220);
    waitForClick();
    remove(sign1);
    remove(sign2);

} //displayInstructions

} //ShootTarget1

6.2 More Game Components
More sophisticated games require more tools, so let's take a look at some additional useful techniques.

6.2.1 Detecting Object Collision
Games that involve movement of objects often require that we detect collision between objects. Using the ACM library this is simple. We begin with a discuss of object collision and what it means.

Consider the filled rectangles below. Clearly they collide when any of their edges overlap.

![not a collision]
![definitely a collision]

But what about this rectangle and circle? What does it mean to say they have collided?

![not a collision]
definitely a collision

but what about this? Is this a collision?

The third example might not normally be considered a collision. After all, the objects don't actually bump into each other.

But consider not just the objects, but the rectangle around the object.

yes, that's a collision!

These bounding rectangles intersect, and that's the way we will define collision—graphic objects collide if and only if their bounding rectangles overlap. It's not the most sophisticated definition, but it's convenient for us because it's easy to use and that's how collision is defined in the ACM library.

Note that the bounding rectangles are not GRects. They don't have any color, they can't be moved, they can't otherwise be characterized. We might say that bounding rectangles are purely theoretical rectangles with no characteristics beyond their corner coordinates. A bounding rectangle is known in the ACM library as a GRectangle.

Let's assume we have two graphical objects, r1 a GRect and o1 a GOval. The steps for detecting collision are:

1) get the bounding rectangle for r1 using the getBounds method
2) store that bounding rectangle in r1Box, which is a GRectangle (not a GRect)
3) repeat for o1
if (r1Box.intersects(o1Box) == true)  
{  
   ...steps to perform when intersection occurs...  
}

Now, where does this code go in the program? Well, objects collide when they move and typically movement is executed in the run( ) method. So a common application might look like this.

while(true)
{
   move object 1
   move object 2
   get bounding rectangle around object 1
   get bounding rectangle around object 2

   if the bounding rectangles intersect
   {
      do whatever needs to be done
   }
}

6.2.2 Dragging Objects with the Mouse

We can drag a graphical object by using the mousePressed( ) and mouseDragged( ) methods together. The key part of the code, shown below, requires some thought.

Let’s begin our discussion by assuming that the mouse button has been clicked and held down on an object. When the button is pressed, the mousePressed( ) method below runs automatically and gives value to three variables: lastX and lastY contain the click coordinates and gobj stores the object that has been clicked on. This mousePressed( ) method also moves the clicked object to the front to make it easy for us to see.

Assume lastX, lastY and gobj exist and have global scope. It is entirely possible that a mouse click will take place in an open area of the window, off of any objects. In this case, the getElementAt( ) method will return null. The null object doesn't have any properties and we can’t do anything with it, so we must write our code appropriately as shown below.

public void mousePressed(MouseEvent e)
{
   lastX = e.getX();
   lastY = e.getY();
   gobj = getElementAt(lastX, lastY);
   if (gobj != null)
      if the bounding rectangle around r1 intersects the bounding rectangle around o1, then perform appropriate steps
When the mouse, with the button held down, is moved, another MouseEvent is created and the mouseDragged method runs automatically. Below we use the lastX and lastY values set when mousePressed( ) executed.

```
public void mouseDragged(MouseEvent m)
{
    int xDisplacement = (int) (m.getX() – lastX);
    int yDisplacement = (int) (m.getY() – lastY);
    gobj.move(xDisplacement, yDisplacement);
    lastX = m.getX();
    lastY = m.getY();
}
```

record the amount of movement
move the object by the required amount.
and finally, record the new location so that it’s ready for the next mouse action.

The example below displays a rectangle and an oval, and allows the user to drag them around in the window.

```
import acm.graphics.*;
import acm.program.*;
import java.awt.*;
import java.awt.event.*;

public class DragObjects extends GraphicsProgram {
    GRect rect;
    GOval oval;
    GObject gobj;
    double lastX, lastY;

    public void init( )
    {
        rect = new GRect(1, 1, 150, 100);
        rect.setFilled(true);
        rect.setColor(Color.RED);
        add(rect);
        oval = new GOval(300, 115, 100, 70);
        oval.setFilled(true);
        oval.setColor(Color.GREEN);
        add(oval);
    }
}
```
addMouseListeners();
} //init

public void mousePressed(MouseEvent e)
{
    lastX = e.getX();
    lastY = e.getY();
    gobj = get_elementAt(lastX, lastY);
    if (gobj != null)
    {
        gobj.sendToFront();
    }
} //mousePressed

public void mouseDragged(MouseEvent m)
{
    if (gobj != null)
    {
        int xDisplacement = (int) (m.getX() - lastX);
        int yDisplacement = (int) m.getY() - lastY;

        gobj.move(xDisplacement, yDisplacement);
        lastX = m.getX();
        lastY = m.getY();
    }
} //mouseDragged

public void run()
{
    /*nothing to do here */
} //run

6.2.3 Automatically Following the Mouse
We can make a graphical object follow the mouse automatically by using the mouseMoved() method, which is executed every time the mouse is moved, whether or not there is a click. mouseMoved() is another method that is part of standard Java that we need to fill in.

Assume we have a ball of size DIAM. We can make the ball follow the mouse around, centered under the mouse with

public void mouseMoved(MouseEvent e)
{
    double x = e.getX();
    double y = e.getY();
    ball.setLocation(x-DIAM/2, y-DIAM/2); //offset to the center of the ball
} //mouseMoved
If we just want the upper left corner of the ball to follow the mouse, we would use  
\text{ball.setLocation}(x, y);

\textbf{6.2.4 Reading Keystrokes}

Sometimes the mouse doesn't do it and a game works better if it is controlled from the keyboard. Let's take a look at reading keystrokes. We begin with

\texttt{import java.awt.event.*;}

and

\texttt{addKeyListeners();}

which tells the program to pay full attention to the keyboard. This method is called in the \texttt{init()} method.

When a key is pressed, a \texttt{KeyEvent} object is created and holds a numeric key code of the pressed key. Key codes are named constants in the \texttt{KeyEvent} class. Here's a few of them.

\begin{itemize}
  \item \texttt{KeyEvent.VK_UP} - up arrow
  \item \texttt{KeyEvent.VK_RIGHT} - right arrow
  \item \texttt{KeyEvent.VK_DOWN} - down arrow
  \item \texttt{KeyEvent.VK_LEFT} - left arrow
  \item \texttt{KeyEvent.VK_SPACE} - space bar
  \item \texttt{KeyEvent.VK_A} through \texttt{KeyEvent.VK_Z} - A \rightarrow Z, a \rightarrow z
  \item \texttt{KeyEvent.VK_0} through \texttt{KeyEvent.VK_9} - 0 \rightarrow 9 (not on num pad though!)
\end{itemize}

VK_A through VK_Z are ordered, as are VK_0 through VK_9 and we may use that fact in our comparisons as shown in the example below.

The program below demonstrate basic reading of the keys. Reading keystrokes can be much more subtle than this, but we'll stay with this blunt instrument since what we really want to do is to control an arcade game. Dig farther into Java if finer control is needed.

\begin{verbatim}
public class ReadKeys extends GraphicsProgram {
    GLabel keyLabel;

    //ReadKeys.java
    import acm.graphics.*;
    import acm.program.*;
    import java.awt.event.*; // need for KeyEvent

    public class ReadKeys extends GraphicsProgram {
        GLabel keyLabel;
    }
\end{verbatim}
public void init() {
    keyLabel = new GLabel("none");
    keyLabel.setFont("*BOLD-50");
    add(keyLabel, 310, 215);
    addKeyListeners();
} //init

public void keyPressed(KeyEvent k) {
    //read arrows keys and the space bar, ignore everything else
    int key = k.getKeyCode();
    if (key == KeyEvent.VK_UP) {
        keyLabel.setLabel("UP");
    } else if (key == KeyEvent.VK_RIGHT) {
        keyLabel.setLabel("RIGHT");
    } else if (key == KeyEvent.VK_DOWN) {
        keyLabel.setLabel("DOWN");
    } else if (key == KeyEvent.VK_LEFT) {
        keyLabel.setLabel("LEFT");
    } else if (key == KeyEvent.VK_SPACE) {
        keyLabel.setLabel("SPACE");
    } else if (key == KeyEvent.VK_A) {
        keyLabel.setLabel("A");
    } else if (KeyEvent.VK_B <= key && key <= KeyEvent.VK_Z) {
        keyLabel.setLabel("B->Z");
    } else if (KeyEvent.VK_0 <= key && key <= KeyEvent.VK_9) {
        keyLabel.setLabel("0->9");
    }
} //keyPressed

public void run() {
    /*nothing to do here */
} //ReadKeys

Our next example program demonstrate moving an object under the control of the arrow keys.

In the example below the keyPressed( ) method is used to set the direction of movement, but the actual movement takes place within the game loop. This is typical of the keyPressed( ) and MousePressed( ) methods. They typically should be very brief and used only to set values that will be used in other methods.

KeyMovementControl1
//KeyMovementControl1.java
import acm.graphics.*;
import acm.program.*;
import java.awt.event.*;
import java.awt.*;
public class KeyMovementControl1 extends GraphicsProgram
{
    final int WAIT = 20;
    final int MV_AMT = 5;
    int xMove, yMove;

    public void init() {
        xMove = yMove = 0;
        addKeyListeners();
    } //init

    public void keyPressed(KeyEvent e) {
        int key = e.getKeyCode();
        if (key == KeyEvent.VK_UP) {
            yMove = -MV_AMT;
        } else if (key == KeyEvent.VK_RIGHT) {
            xMove = MV_AMT;
        } else if (key == KeyEvent.VK_DOWN) {
            yMove = MV_AMT;
        } else if (key == KeyEvent.VK_LEFT) {
            xMove = -MV_AMT;
        }
    } //keyPressed

    public void run() {
        GOval o = new GOval(40, 40);
        o.setColor(Color.RED);
        o.setFilled(true);
        add(o, 100, 100);

        while(true) //game loop
        {
            pause(WAIT);
            o.move(xMove, yMove);
            xMove = yMove = 0;
        } //game loop
    } //run
} //KeyMovementControl1

If you run this program you will note that movement stops as soon as a key is released.

Here's another version. This time the object continues to move after the key is released, and the space bar is used to stop movement.
KeyMovementControl2

//KeyMovementControl2.java
import acm.graphics.*;
import acm.program.*;
import java.awt.event.*;
import java.awt.*;
public class KeyMovementControl2 extends GraphicsProgram {

    final int WAIT = 20;
    final int MV_AMT = 5;
    int xMove, yMove;

    public void init() {
        xMove = yMove = 0;
        addKeyListeners();
    } // init

    public void keyPressed(KeyEvent e) {
        int key = e.getKeyCode();
        if (key == KeyEvent.VK_UP) {
            yMove = -MV_AMT;
            xMove = 0;
        } else if (key == KeyEvent.VK_RIGHT) {
            xMove = MV_AMT;
            yMove = 0;
        } else if (key == KeyEvent.VK_DOWN) {
            yMove = MV_AMT;
            xMove = 0;
        } else if (key == KeyEvent.VK_LEFT) {
            xMove = -MV_AMT;
            yMove = 0;
        } else if (key == KeyEvent.VK_SPACE) {
            xMove = 0;
            yMove = 0;
        }
    }
}
6.2.5 Randomizing the Action

Random numbers and other random values are extremely useful for simulations and game software, allowing us to create a game that varies from one play to another. The ACM library provides a simple mechanism for generating several kinds of random values.

Creating a random generator
Before generating any type of random, we must create a mechanism for generating random values. We begin with

```java
import acm.util.*;

To create a random generating mechanism, we use

RandomGenerator rg = new RandomGenerator();
```

where `rg` is the name of our `RandomGenerator`.

Creating random ints
Once a `RandomGenerator` has been created you may use it to generate a random `int` in the range \( \text{minimum value} \leq \text{random int} \leq \text{maximum value} \).

These example generate random integers.

```java
int diceValue = rg.nextInt(1, 6); \leftarrow \text{random numbers from 1 to 6 (inclusive), useful for simulating a throw of a die.}
```
int \textit{jumpValue} = \texttt{rg.nextInt(-5, 5)}; \leftarrow \text{random number from -5 to 5 (inclusive), perhaps using the value to control an object's movement left or right 5 pixels.}

Now let's look at a console program that generates and displays random integers.

\begin{verbatim}
RandomInts
//RandomInts.java
import acm.program.*;
import acm.util.*;

public class RandomInts extends ConsoleProgram
{
    public void run( )
    {
        RandomGenerator rg = new RandomGenerator( );
        for(int i=0; i< 5; i++)
        {
            print(rg.nextInt(-5, 5) + " ");
        } //display 5 random integers
    } //run
} //RandomInts
\end{verbatim}

A sample run might give us

\begin{verbatim}
-2 -2 -4 -5 -3
\end{verbatim}

and another might give us

\begin{verbatim}
4 -1 0 -3 -5
\end{verbatim}

\textbf{Using randoms with graphic objects}
If \textit{ufo1} is a \textit{UFO} as we've created before and \textit{x} and \textit{y} are random integers, then

\texttt{ufo1.move(x, y);}

moves \textit{ufo1} by a random jump from the current location.

\textbf{Setting the random seed}
A random number is not truly random. A sequence of randoms is automatically initialized with a value taken from the system clock, and the random numbers are actually generated by a formula in Java.
The beginning value is called a seed. Since the system clock continuously changes, the seed continuously changes and thus a generated sequence of randoms changes from one run to the next. This means that for a program using randoms one run of the program will differ from the next. Great for playing a card game, lousy for finding problems in your program.

For debugging it is very useful to be able to recreate the same sequence of random numbers and this requires that we supply the seed for the random number generator, instead of it being taken from the system clock. Fortunately this is very simple. Below are two examples.

```
rg.setSeed(0);
rg.setSeed(99);
```

The code segment below will display the same set of random values repeatedly each time it is executed.

```
RandomGenerator ranGen = new RandomGenerator();
ranGen.setSeed(0);

for(int i=0; i<5; i++)
{
    println(ranGen.nextInt(1,10));
}
```

The particular value used to set the seed is unimportant. What is important is that you set the seed when you need consistent random values. Remember that you usually want to delete or comment out the setSeed( ) statement when bugs are squashed.

In the next chapter you'll see how to create other random values—doubles, booleans, Colors and other types of random values.

### 6.2.6 Keeping Time and Score

Many games require keeping time (how long did it take to find the Wumpus?) and keeping score, and that's what we'll look at next.

**Set up for keeping time and score**

Set up for timing and score keeping requires declaration and initialization of some global variables as shown in the code segment below.

```
final int WAIT = 50;

int time, score;
GLabel scoreLbl;
```
public void init( )
{
  time = 0; //game loop hasn't begun
  score = 0; //no points have been scored
  scoreLbl.setLabel("Score: " + score);
  scoreLbl.setFont("SanSerif-BOLD-30");
  add(scoreLbl, 2, 26); //position at upper left corner of window
  ...remainder of initialization...
}

Updating the time
Basic game execution occurs inside the game loop. This loop may contain many statements, but most of them run so quickly that for our purposes we can consider them as executing instantaneously. The pause( ) statement is different however. In it we use a WAIT value to slow execution to a speed we can play and this adds measurably to the current time. Thus we need to add the WAIT time to the current time value with every loop.

while(true) //game loop
{
  pause(WAIT);
  time = time + WAIT;
  ...remainder of game playing code...
}

Updating the score
Changing the score value can occur in many places. We consider these in turn.

- if scoring takes place when the mouse is clicked, then we must increment the score in the mousePressed( ) method.

  if (the object clicked on was the target)
  increment the score and display it

- if scoring takes place when objects collide, we must increment the score where collision is detected, normally in the game loop.

  while(true)
  {
    ...game code...
    if(o1Box intersects o2Box)
      increment the score and display it
    ...more game code...
  }

- if scoring takes place when a goal is reached, we increment scoring when we have confirmed that the goal has been reached, typically within the game loop.
while(true)
{
    ...game code...
    if (zombies are all dead and hero is still alive)
    increment the score and display it
    ...more game code...
}

Displaying the updated score is often done at the time of the score change, but can also be put off to any convenient point. For example, though play in the game loop and the resultant mouse clicks might change a score several times in one pass through the loop, it isn't necessary to display the changed score each time. It would probably change too fast for the player to see the changes anyway. Instead just use a single statement at the end of the game loop to display the net result for that pass through the loop.

while(true)
{
    ...game code including statements that change the value of the score...
    scoreLbl.setLabel(new value of the score);
}

6.3 An Example Program

Our goal is to write a game that drops balls that we attempt to block. A point is scored each time a ball is blocked.

Game objects
A ball is simply a colored circle and a blocker is a colored line, so they're easy to code. In fact, we probably don't really need to create these as separate objects. But if we do then later we can very easily could go back and change these objects to make them more interesting.

Cannonball

```java
//CannonBall.java
import acm.graphics.*;
import java.awt.*;

public class Cannonball extends GCompound
{
    final int SIZE = 30;
    final Color DEFAULT_COLOR = Color.BLACK;
    private GOval ring1;
    public Cannonball()
    {
        ring1 = new GOval(SIZE, SIZE);
        ring1.setFilled(true);
    }
}
```
ring1.setColor(DEFAULT_COLOR);
        add(ring1);
    } //Cannonball

    public Cannonball(Color cv)
    {
        ring1 = new GOval(SIZE, SIZE);
        ring1.setFilled(true);
        ring1.setColor(cv);
        add(ring1);
    } //Cannonball

The game itself is more complex. We'll develop it in two stages, the first implementing some basic functionality and the second adding more objects and features.

**Stage one development**

A cannonball is dropped from the top of the window and falls downward. The player moves the blocker by moving the mouse. Play continues with the cannonball reversing direction when it hits the blocker and again reversing direction when it hits the top of the window. Eventually the player misses a block and the cannonball hits the bottom of the window which ends the game.
Cannonade1

//Cannonade1.java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
import java.awt.event.*;
public class Cannonade1 extends GraphicsProgram
{
    public static final int APPLICATION_WIDTH = 400;
    public static final int APPLICATION_HEIGHT = 500;

    final int AW = APPLICATION_WIDTH;
    final int AH = APPLICATION_HEIGHT;
    final int WAIT = 5;
    final int CB_SIZE = 30;

    Cannonball cb1;
    Blocker blocker;
    int time;

    public void init() {
        setBackground(Color.BLUE);

        time = 0;
        cb1 = new Cannonball();
        add(cb1, AW/2, 0);
        blocker = new Blocker();
        add(blocker, AW/2, AH/2);

        addMouseListeners();
        waitForClick();
    } //init

    public void mouseMoved(MouseEvent e) {
        double x = e.getX();
        double y = e.getY();
        blocker.setLocation(x-40, y); //center the blocker under mouse
    } //mouseMoved

    public void run() {
        GRectangle cb1Box, blockerBox;
    }
}
This version implements the basic functionality of the game—fire a cannonball, block it when appropriate, bounce it off the blocker and the top of the window, end the game when it hits the bottom. But it's hardly a complete game.

**Stage two development**
Our next version adds quite a bit to the game—three cannonballs at random locations that fall with different speeds bouncing off the blocker in a random direction, bouncing of the sides of the window and scoring. The cannonball and blocker objects are the same and we'll focus only on the interesting changes within the game program.

**Cannonade2**

```java
//Cannonade2.java
import acm.program.*;
import acm.graphics.*;
import acm.util.*;
import java.awt.*;
import java.awt.event.*;

public class Cannonade2 extends GraphicsProgram
```
public static final int APPLICATION_WIDTH = 400;
public static final int APPLICATION_HEIGHT = 500;

final int AW = APPLICATION_WIDTH;
final int AH = APPLICATION_HEIGHT;
final int WAIT = 10;
final int CB_SIZE = 30;

Cannonball cb1, cb2, cb3;
Blocker blocker;
int time, score;
GLabel scoreLbl;

public void init()
{
    setBackground(Color.BLUE);

    time = 0;
    score = 0;
    scoreLbl = new GLabel(""+score);
    scoreLbl.setFont("SanSerif-BOLD-30");
    scoreLbl.setColor(Color.RED);
    add(scoreLbl, 2, 26);

    cb1 = new Cannonball();
    cb2 = new Cannonball(Color.GREEN);
    cb3 = new Cannonball(Color.YELLOW);

    RandomGenerator rg = new RandomGenerator();
    int rand1 = rg.nextInt(0, AW-50);
    int rand2 = rg.nextInt(0, AW-50);
    int rand3 = rg.nextInt(0, AW-50);

    add(cb1, rand1, 0);
    add(cb2, rand2, 0);
    add(cb3, rand3, 0);
    blocker = new Blocker();
    add(blocker, AW/2, AH/2);

    addMouseListeners();
    waitForClick();
} //init

public void mouseMoved(MouseEvent e)
double x = e.getX();
double y = e.getY();
blocker setLocation(x-40, y); //center blocker under mouse

public void run()
{
    RandomGenerator rg = new RandomGenerator();
    GRectangle cb1Box, cb2Box, cb3Box, blockerBox;

    int xMove1 = 0, xMove2 = 0, xMove3 = 0;
    int yMove1 = 3, yMove2 = 4, yMove3 = 5;

    boolean cb1Done = false, cb2Done = false, cb3Done = false;

    while(true) //game loop
    {
        pause(WAIT);
        time = time + WAIT;
        cb1.move(xMove1, yMove1);
        cb2.move(xMove2, yMove2);
        cb3.move(xMove3, yMove3);

        cb1Box = cb1.getBounds();
        cb2Box = cb2.getBounds();
        cb3Box = cb3.getBounds();
        blockerBox = blocker.getBounds();

        //blocking
        if (cb1Box.intersects(blockerBox) == true)
        {
            yMove1 = -yMove1;
            xMove1 = rg.nextInt(-1, 1);
            score++;
        }

        if (cb2Box.intersects(blockerBox) == true)
        {
            yMove2 = -yMove2;
            xMove2 = rg.nextInt(-1, 1);
            score = score + 2;
        }

        if (cb3Box.intersects(blockerBox) == true)
        {
            yMove3 = -yMove3;
            xMove3 = rg.nextInt(-1, 1);
        }
    }
}
```java
score = score + 3;
}

// at top of window?
if (cb1.getY() == 0)
{
    yMove1 = -yMove1;
}
if (cb2.getY() == 0)
{
    yMove2 = -yMove2;
}
if (cb3.getY() == 0)
{
    yMove3 = -yMove3;
}

// at left or right side?
if (((cb1.getX() <= 0) || (cb1.getX() + CB_SIZE >= AW))
{
    xMove1 = -xMove1;
}
if (((cb2.getX() <= 0) || (cb2.getX() + CB_SIZE >= AW))
{
    xMove2 = -xMove2;
}
if (((cb3.getX() <= 0) || (cb3.getX() + CB_SIZE >= AW))
{
    xMove3 = -xMove3;
}

// at bottom of window?
if ((cb1Done == false) && (cb1.getY() + CB_SIZE >= AH))
{
    score--;
    cb1Done = true;
}
if ((cb2Done == false) && (cb2.getY() + CB_SIZE >= AH))
{
    score--;
    cb2Done = true;
}
if ((cb3Done == false) && (cb3.getY() + CB_SIZE >= AH))
{
    score--;
    cb3Done = true;
}
```
You may notice one potential problem. Working with only three cannonballs requires a lot of repetitive code to do the same thing to each cannonball, and this would become very unwieldy if we had many more cannonballs. There’s got to be a better way, and there is. Arrays and ArrayLists will be covered in a later chapter and they make it possible to handle large numbers of the objects easily.

### 6.4 Better Games

Now let’s look at some of the things that make games that look and play better.

**Adding levels to the game**

If you think about the games we’ve written so far you’ll see the same general algorithm every time.

```java
initialize variables
while(true)
{
   //game playing code...
   if (some condition)
      break
}

A game with a second level of play adds another section of code afterwards.

**reinitialize variables as needed**
while(true)
{
   //game playing code...
   if (some condition)
      break
}"

©Daniel L. Schuster, 2009, all rights reserved
The only difficulty is that we must be careful about what variables we reinitialize.
- a score probably does not get reinitialized, allowing us to add points by playing the second level.
- objects that have moved may need to be put back in place.
- values that control speed may change in play from one level to the next, giving the player a faster playing environment.

Multiple levels can also be created with nested loops.

```java
while(true)
{
    initialize variables
    while(true)
    {
        ...game playing code...
        if (some condition)
            break out of inner game loop
    }
    if (some condition)
        break out of outer game loop
}
```

However, the actual variable initialization often must change from one level to another.

**Bar Meters**
A digital display might count down 3, 2, 1, 0 with 0 representing some important condition. An alternative display that doesn't require reading and is thus probably quicker to understand uses a bar meter to indicate object condition. Using the `setSize()` method to shrink a `GRect` we might have

<table>
<thead>
<tr>
<th>Value Represented</th>
<th>100%</th>
<th>75%</th>
<th>50%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green Bar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Green Bar</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Green Bar</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Even better would be to use `setSize()` and `setColor()`.

©Daniel L. Schuster, 2009, all rights reserved
Of course we can use meters like this for many purposes. In a lunar lander game, we might use a digital display for the altitude of the lander and bar meters for remaining fuel and condition of the shields. Here's an example dashboard.

More Sophisticated Movement
When running some of these programs you may have noticed some unusual behavior. Under some circumstances a moving object might 'hang up' on another object it is supposed to bounce off. Let's figure out why that happens and then discuss a fix for it.

Recall the Cannonade games above and consider a cannonball that moves 5 pixels per game loop and the blocker, with a distance of 7 pixels between them as shown.

The next iteration of the game loop might put them within 2 pixels of each other.

and the next pass through the loop might give us this situation.

At this point we detect collision. You may have noticed this overlap.

We bounce the cannonball up by the same amount as the last down movement, and again we have.
Everything looks and behaves as we expect. Nothing unusual is happening here.

Things change if, just after the collision, we move the blocker upward, perhaps by making it follow the mouse. The bounce moves the cannonball up but the blocker moves up also. We might have

which is detected as another collision. The direction changes and now the cannonball moves down. We might have

which is another collision. And so on—the cannonball is 'hung up' on the blocker.

Now that we've identified the problem, what causes it and what can we do about it?

The minimum size of the movement of our hypothetical cannonball, known as the movement granularity, is 5 pixels. But if the cannonball is only 1 pixel away from the blocker, the side of the window, or some other object in the game, then after the next movement the edge of the cannonball overlaps the other object, and that's just not physically possible. Our physics model is inaccurate.

For many simple games this is not really a problem. We just accept that sometimes the objects on screen behave differently than behavior in the real world. After all, it's just a game.

On the other hand, if we want our game to be the best, or if it's more than a game (perhaps it's a simulation used for pilot training), then we'll need to make our physics model more accurate.

The correct solution is to move only 1 pixel at a time, even for an object that moves 5 pixels per game loop, and to check for collision with each single pixel movement. This more accurately reflects reality and will thus give us more realistic behavior. We would have something like this.

```
while(true)
{
    ...game code...
    for(the number of pixels of movement per game loop)
    {
        move the cannonball 1 pixel in the current direction
        if (there is collision)
        {
            change direction to allow for bounce
            exit the loop
        }
    }
```

©Daniel L. Schuster, 2009, all rights reserved
Graphics Problem set

For all programs below it is assumed that you will create separate objects for elements of the game and use methods where appropriate.

1) **ShootTwo** modify the ShootTarget program above so that a smaller, slightly faster blue rectangle also bounces around in the application window. Clicks on the red ball are worth 1 point, clicks on the blue rectangle are worth 3 points. The score displays only when the game ends.

2) **ShootThree** Modify ShootTwo to include a second level, where the speeds of the balls increase, and the updated score is displayed continuously.

3) **ShootFour** write a program that displays four targets to shoot at by clicking on them. The targets are:
   a. a red ball which moves as it did in ShootTarget above, and is worth 1 point
   b. a blue ball which moves like the red ball but in another direction. It is slightly faster, slightly smaller and is worth 2 points
   c. a green ball is twice the size of a red ball and move similarly EXCEPT that they are usually invisible. They are worth 10 points. Use the modulus (%) operator and the setVisible( ) method to make the visibility change in the game loop
   d. a yellow ball is very fast, ½ the size of a red ball and moves only back and forth (horizontally). It is worth 20 points
   e. update the displayed score every game loop

4) **Survivor1** write a program that puts a red ball and a green ball on screen. The red ball bounces around the screen. The player drags the green ball around, avoiding the red ball as long as possible. When collision finally occurs, display the survival time.

5) **Survivor2** same as above, except that the speed of the red ball doubles after 10 seconds, then doubles again after another 10 seconds.

6) **Pong1** find a simple Pong game online and implement your own version, moving the player's paddle by moving it with the mouse. The computer's paddle should move automatically. The player's stays near the left end of the table, the computer’s paddle stays near the right end of the table. The computer never misses!

7) **Pong2** now modify your Pong1 games as follows developing the program in the suggested stages:
a. overall play – the player plays games and matches against the computer. But since the computer never misses, the player is really playing against his or her own scores.
b. playing a single game: play continues until the player has missed 3 times. During play each successful return by the player is worth 1 pt. Returns by the computer are not scored. The score is clearly displayed as a number in the bottom right corner of the window. Each game is played at one one playing speed.
c. playing a single match: a match consists of 3 games. When play moves from one game to another the playing speed increases
d. At the end of a match, the players score is prominently displayed. Additionally, a random comment such as “Great match” or “Uhh, ... you’d better keep your day job” is displayed. The player has the opportunity to play another match with a click on an appropriately labeled box on the screen.

8) **HelicopterInACave** create a helicopter in a cave scroller game. The helicopter maneuvers to avoid stalactites and stalagmites. Control the movement with the mouse (clicks for more power) or with the up arrow key (presses for more power). The helicopter falls under the influence of gravity. Points are calculated based on distance traveled before crashing. Provide two levels of play, the first more difficult than the second.

![HelicopterInACave](image)

9) **BugBot** create a playing field with various objects, as shown here. Don't use too many object as you will need to check collision with each one.
Add a BugBot to an open area of the playing field. Your's may certainly be more elaborate than the simple black circle shown below. The BugBot moves around the playing field. When it hits an object (one of the circles or rectangles) or a wall it 1) back up a few pixels 2) turns either left or right randomly 3) continues forward in the new direction.

The robot continues to move until it hits the final goal, the red circle marked with a white X, or 30 seconds have elapsed. Above is one possible path from the start to finish. However, it's not a very likely path, since you are turning randomly. Note that your program will have only one BugBot. Many of them are shown above to illustrate the path.

10) **UFOEscape** Create a UFO object similar to the UFOs we’ve created before. Fly the UFO around the window by making it follow the mouse. Asteroids appear at random locations on the edges of the screen and move across the screen. They cause damage to the UFO. Black holes appear randomly but don’t move. They kill the UFO. Dilithium crystals appear randomly and don’t move. They recharge the UFO’s fuel. Asteroids, black holes and
dilithium crystals should also be objects you design. The game ends when the UFO has been damaged beyond repair. Display the status of fuel, shields and survival time (in 1 second increments). Add levels to the game. Use only a few asteroids, black holes and dilithium crystals.

11) **YouChoose1** Create an arcade game of your own design or find one online to imitate. Your program must include

   a. moving object under player control
   b. scoring
   c. levels
   d. use of programmer defined objects

Your instructor's approval of your project is required *before* you begin. This is to protect you from taking on something that's too hard.