Chapter 11
Game Programming II

Good games may require more tools than we’ve got so far, so let’s look at a bit more mouse control, making more interesting objects and working with sound.

11 More Mouse Interaction
So far we’ve looked at working with left mouse clicks and detecting mouse drags. We’ll begin this chapter by learning more about working with the mouse.

11.1 Detecting Left and Right Mouse Clicks
The MousePressed method detects only left clicks by default. We can also detect right clicks if we wish. The code below detects both left and right clicks by using the standard Java int values MouseEvent.BUTTON1 and MouseEvent.BUTTON3 respectively. Some mice have a middle button which can be detected with BUTTON2 but use of the middle button is somewhat rare in practice.

```java
public void mousePressed(MouseEvent e)
{
    if (e.getButton() == MouseEvent.BUTTON1) // left click?
    {
        leftClickCount++;
    }
    else if (e.getButton() == MouseEvent.BUTTON3) // right click?
    {
        rightClickCount++;
    }
} // mousePressed
```

You can use this technique to trigger different actions with left and right clicks. The example below allows you to drag the rectangle with only the left mouse button and drag the oval with only the right mouse button.

DragObjects2
// DragObjects2.java

```java
import java.awt.*;
import java.awt.event.*;
import acm.graphics.*;
import acm.program.*;
```
public class DragObjects2 extends GraphicsProgram
{
    GObject gobj;
    GPoint lastClick;
    GRect rect;
    GOval oval;
    int whichClick;

    public void init()
    {
        rect = new GRect(100, 100, 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)
    {
        lastClick = new GPoint(e.getPoint());
        gobj = getElementAt(lastClick);
        if (e.getButton() == MouseEvent.BUTTON1) // left click
        {
            whichClick = 1;
        }
        else // right click
        {
            whichClick = 3;
        }
    } // mousePressed

    public void mouseDragged(MouseEvent e)
    {
        if ((gobj == rect && whichClick == 1) || (gobj == oval && whichClick == 3))
        {
            gobj.move(e.getX() - lastClick.getX(), e.getY() - lastClick.getY());
            lastClick = new GPoint(e.getPoint());
        }
    } // mouseDragged

    // no run method needed!
}
A Note for Mac and Linux Users
Some Mac mice have only one button and so the program above doesn’t work. You can solve this problem two ways

- buy a 2-button USB mouse. All reasonably recent Macs support a two button mouse.
- configure your Mac to simulate a right click. Check the Setting Up Your Computer appendix for information.

Linux and its various user interfaces support a two button mouse, which is standard equipment for Linux users.

11.2 Time Slicing Object Behavior
A graphic object that never changes is a boring graphic object. Now we’ll take a look at changing the behavior of an object during game play. We’ll call this time slicing.

11.2.1 Game Loop Synchronized Behavior
The simplest time sliced behavior changes with every pass through the game loop. The example below, based on the UFO2 we created back in chapter 5, changes the color of the UFO body from the original color to black and back again.

UFO3

```java
//UFO3.java
import acm.graphics.*;
import java.awt.*;

public class UFO3 extends GCompound
{
    private GOval body, bubble, alien;
    private Color nextColor;

    public UFO3(Color bodyColor, Color alienColor)
    {
        body = new GOval(50, 25);
        body.setFilled(true);
        body.setColor(bodyColor);
        bubble = new GOval(25, 25);
        alien = new GOval(10, 10);
        alien.setFilled(true);
        alien.setColor(alienColor);
        add(bubble, 13, 0);
        add(alien, 20, 5);
        add(body, 0, 13);
        nextColor = Color.BLACK;
    }
}
```

©Daniel L. Schuster, 2009, all rights reserved
public void switchColor() {
    Color tempColor;
    if (body.getColor() == Color.BLACK) {
        body setColor(nextColor);
        nextColor = Color.BLACK;
    } else {
        tempColor = body.getColor();
        body setColor(Color.BLACK);
        nextColor = tempColor;
    }
} //switchColor

The program below puts a UFO3 in the window and moves it, changing colors with each iteration of the game loop.

GameLoopSynchronized

//GameLoopSynchronized.java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
public class GameLoopSynchronized extends GraphicsProgram {
    final int WAIT = 1000;
    UFO3 u1;

    public void init() {
        u1 = new UFO3(Color.RED, Color.BLUE);
        add(u1, 10, 100);
    } //init

    public void run() {
        while(true) {
            pause(WAIT);
            u1.move(1, 0);
            u1.switchColor();
        } //game loop
    } //run
The program works great, flipping the colors every second. But that one second pause means that movement is very slow. Now try it with a smaller \textit{WAIT} value, something more realistic like 20 milliseconds.

Not so great, right? The faster execution speed really messes up the color change effect, probably ruining it for most purposes.

Time slicing behavior that is synchronized with the game loop is typically not sufficient for sophisticated, attractive behaviors.

\textbf{11.2.2 Game Loop Dependent Behavior}

Consider the problem. Small \textit{WAIT} values are usually required to get reasonably smooth and quick motion. But the behavior change has to be slower. For this example what we want is to move a ufo every 20 milliseconds (1/50 second) but switch the color every 500 milliseconds (1/2 second). No problemo!

Recall the \% (modulos) operator from Chapter 1. $x \% y$ calculates the remainder when $x$ is divided by $y$. For example

- $20 \% 3 \rightarrow 2$
- $20 \% 4 \rightarrow 4$
- $20 \% 5 \rightarrow 0$
- $20 \% 30 \rightarrow 20$

In particular note that

- $500 \% 500 \rightarrow 0$
- $1000 \% 500 \rightarrow 0$
- $1500 \% 500 \rightarrow 0$

and so on. We can use the \% operator to determine if the current time is a multiple of a $\frac{1}{2}$ second and we can use that to change the color. Let’s modify the \textit{switchColor()} method so that color changes only happen every 500 milliseconds.

\textbf{UFO4}

\begin{verbatim}
//UFO4.java
import acm.graphics.*;
import acm.util.*;
import java.awt.*;
public class UFO4 extends GCompound {
    private GOval body, bubble, alien;
    private Color nextColor;

    public UFO4() {
        // Initialize ufo parts...
    }

    public void move() {
        // Move ufo...
    }

    public void switchColor() {
        // Check if it's time to switch color...
    }
}
\end{verbatim}
public UFO4(Color bodyColor, Color alienColor)
{
    body = new GOval(50, 25);
    body.setFilled(true);
    body.setColor(bodyColor);
    bubble = new GOval(25, 25);
    alien = new GOval(10, 10);
    alien.setFilled(true);
    alien.setColor(alienColor);
    add(bubble, 13, 0);
    add(alien, 20, 5);
    add(body, 0, 13);
    nextColor = Color.BLACK;
} //UFO4

public void switchColor(int ct)
{
    Color tempColor;
    if ((body.getColor() == Color.BLACK) && (ct % 500) == 0)
    {
        body.setColor(nextColor);
        nextColor = Color.BLACK;
    }
    else if (ct % 500 == 0)
    {
        tempColor = body.getColor();
        body.setColor(Color.BLACK);
        nextColor = tempColor;
    }
} //switchColor
} //UFO4

The program below tracks the current game time and passes that to switchColor( ) which determines if it’s time for a color change.

GameLoopDependent
//GameLoopDependent.java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class GameLoopDependent extends GraphicsProgram
{
    final int WAIT = 100;
    int currentTime;
UFO4 u1;

public void init( )
{
currentTime = 0;
u1 = new UFO4(Color.RED, Color.BLUE);
add(u1, 10, 100);
} //init

public void run( )
{
while(true)
{
  pause(WAIT);
currentTime = currentTime + WAIT;
u1.move(1, 0);
u1.switchColor(currentTime);
} //game loop
} //run
} //GameLoopDependent

11.2.3 Behavior that Executes for a Fixed Period of Time

Oh wait, we’ve already solved that problem when we raise shields for a UFO in the Objects II chapter, so we’ll refer you there for an example.

11.2.4 Behavior that Executes Randomly

This time let’s consider something that happens randomly. The method below switches the ufo color one time in five by using random numbers as we saw in Chapter 6.

public void switchColor( )
{
  Color tempColor;
  RandomGenerator rg = new RandomGenerator();
  int v = rg.nextInt(1, 5); //random from 1 to 5
  if (v == 1) && (body.getColor() == Color.BLACK))
  {
    body.setColor(nextColor);
    nextColor = Color.BLACK;
  }
  else if (v == 1)
  {
    tempColor = body.getColor();
    body.setColor(Color.BLACK);
    nextColor = tempColor;
  }
}
The `switchColor()` code above is a little awkward though due to the way the conditions are written. Below is a more expressive, natural version of `switchColor()` that begins the conditionals with a check for “should the colors be switched.”

```java
public void switchColor(int ct)
{
    Color tempColor;
    RandomGenerator rg = new RandomGenerator();
    int v = rg.nextInt(1, 5); //random from 1 to 5
    if (v == 1)
    {
        if (body.getColor() == Color.BLACK)
        {
            body.setColor(nextColor);
            nextColor = Color.BLACK;
        }
        else
        {
            tempColor = body.getColor();
            body.setColor(Color.BLACK);
            nextColor = tempColor;
        }
    } //v == 1
} //switchColor
```

We could make similar changes to the `switchColor()` method that changes color every ½ second in `UFO4`.

### 11.3 Working with Sound

Sound is an integral, if sometimes irritating, part of any modern game, so let's take a look at integrating some basic noise into our games.

#### 11.3.1 Getting Started

The ACM Java library provides sound support through the `SoundClip` object. Working with sound can get pretty ugly pretty quick so we'll focus on the basics most useful in a game.

**Audio Formats**

For our purposes digital sound information will be stored in a file which we will load and play. There are many different sound file formats. `SoundClips` support three of these:

- AU—an audio format from Sun
- WAV—the waveform format, common to the Windows environment. Not all wav files are supported by `SoundClips`
• **AIFF**—the *Audio Interchange File Format*, an audio format commonly used on Apple products

Your computer probably has many of these sound file already installed. The examples below use
• **sw-theme.wav**—a passage from Star Wars
• **ding.wav**—a short sound found on many Windows computer
• **laser_trill.au**—a short sound found on the web

**The SoundClip Object**
The *SoundClip* object is located within the *acm.util.* library, which we first **import**.

```
SoundClip x = new SoundClip("sw-theme.wav");
```

**Controlling Sound Play**
The volume must be set to a value in the range (0, 1], though other software or hardware controls may change the volume.

```
x.setVolume(.5)
```

There are two methods for playing a sound clip.

```
x.play();  \(\leftarrow\) plays the sound clip once and only once
x.loop();  \(\leftarrow\) plays the sound clip over and over indefinitely
```

and a *SoundClip* can be stopped with

```
x.stop();
```

and the clip can be 'rewound' to the beginning with

```
x.rewind();
```

so that the next play begins at the start of the clip.

**A Separate Thread**
Setting the color of a *GRect* or moving one in the window executes basically instantaneously. In contrast, playing a *SoundClip* takes time. There's a conflict there because game really requires that sound plays *while* the game executes.

To make this happen efficiently the main program executes in one **thread**, or path of program execution, and play of the *SoundClip* automatically executes in a separate thread. The use of a separate thread means the sound play can be effectively interlaced with game play.
The statements

\begin{verbatim}
x.loop();
pause(4000);
x.stop();
\end{verbatim}

execute as shown here.

Here's a small demo program.

**SoundDemo1**

//SoundDemo1.java
import acm.graphics.*;
import acm.program.*;
import acm.util.*;

public class SoundDemo1 extends GraphicsProgram {

©Daniel L. Schuster, 2009, all rights reserved
public void run()
{
    SoundClip x = new SoundClip("sw-theme.wav");
    x.setVolume(.8);
    x.play();
    pause(1000);
    x.stop();
    waitForClick();
    x.rewind();
    x.loop();
    pause(10000);
    x.stop();
    pause(3000);
    x.rewind();
    x.loop();
} //run
} //SoundDemo1

11.3.2 Working with Games

SoundClips play in the background, so you can effectively run the game action while sound plays.

Below is a lightly modified version of the NewUFO2. One of the constructors has been removed for brevity and sound capability has been added. When the ufo is moving the 'motor' will run, making sound. Callouts draw attention to every change to the class.

NewUFO2WithSound
//NewUFO2WithSound.java
import acm.graphics.*;
import acm.util.*;
import java.awt.*;
public class NewUFO2WithSound extends GCompound
{
    //*** private variables
    private GOval body, bubble, alien;
    private int fuel, shieldsDownTime;
    private boolean shieldsUp = false;
    private SoundClip motor;

    //*** private constants
    private static final int FUEL_MAX = 1000;
    private static final int SHIELD_TIME = 500;

    // load the SoundClip object
}
private static final int MAX_SHIELDS_USES = 8;

//*** private static variables
private static int shieldUsedCnt = 0;

//*** constructors
public NewUFO2WithSound() {
    body = new GOval(50, 25);
    body.setFill(true);
    body.setColor(Color.RED);
    bubble = new GOval(25, 25);
    alien = new GOval(10, 10);
    alien.setFill(true);
    alien.setColor(Color.GREEN);
    add(bubble, 13, 0);
    add(alien, 20, 5);
    add(body, 0, 13);
    fuel = FUEL_MAX;

    motor = new SoundClip("pacman.au");
    motor.setVolume(.5);
} //default constructor

//*** fuel methods
public int getFuel() {
    return fuel;
}

public void refuel() {
    fuel = FUEL_MAX;
}

//*** shields methods
public void raiseShields(int currentTime) {
    if (shieldUsedCnt < MAX_SHIELDS_USES && shieldsUp == false) {
        shieldUsedCnt++;
        shieldsUp = true;
        shieldsDownTime = currentTime + SHIELD_TIME;
    }
} //raiseShields

public boolean shieldsAreUp() {
    return shieldsUp;
}

private void lowerShields(int currentTime) {

if ((currentTime == ShieldsDownTime) && (shieldsUp == true))
    shieldsUp = false;
} //lowerShields

public int getShieldUses()
{
    return shieldUsedCnt;
}

/*** move methods
public void move(int x, int y, int currentTime)
{
    if (fuel > 0)
    {
        move(x, y);
        fuel = fuel - Math.abs(x) - Math.abs(y);
        if (x > 0 || y > 0) motor.play();
    }
} lowerShields(currentTime);
} //move

} //NewUFO2WithSound

Below is a modified version of TestUFO2. This time background music plays (the Star Wars theme), the ufo makes noise whenever it is moving and there are different sounds for picking up fuel, bouncing off the bomb and crashing into it. It's actually a simple conversion from the previous version.

TestNewUFO2WithSound

//TestNewUFO2WithSound.java
import acm.program.*;
import acm.graphics.*;
import acm.util.*;
import java.awt.event.*;
import java.awt.*;
public class TestNewUFO2WithSound extends GraphicsProgram
{
    final int PAUSE = 50;
    GRect bomb;
    GOval fuelCell;
    GLabel fuelSign1, shieldUses;
    NewUFO2WithSound u1;
    int u1XVel, u1YVel, time;
    boolean exit;
    SoundClip bkgnd, bump, crash, fuelUp;
    final int SOUND_LEVEL = 1;
public void init()
{
    u1 = new NewUFO2WithSound();
    add(u1, 0, 200);

    bomb = new GRect(100, 100);
    bomb.setColor(Color.RED);
    bomb.setFilled(true);
    add(bomb, 300, 200);

    fuelCell = new GOval(30, 30);
    fuelCell.setColor(Color.GREEN);
    fuelCell.setFilled(true);
    add(fuelCell, 400, 300);

    fuelSign1 = new GLabel("u1 fuel: "+u1.getFuel( ));
    add(fuelSign1, 10, 20);
    shieldUses = new GLabel("shield uses: "+u1.getShieldUses( ));
    add(shieldUses, 650, 20);

    u1XVel = u1YVel = time = 0;
    exit = false;

    addKeyListeners();
    waitForClick();

    bump = new SoundClip("gun.au");
    bump.setVolume(SOUND_LEVEL*.8);
    crash = new SoundClip("laser_trill.au");
    crash.setVolume(SOUND_LEVEL*1);
    fuelUp = new SoundClip("miss.au");
    fuelUp.setVolume(SOUND_LEVEL*.6);
    bkgnd = new SoundClip("sw-theme.wav");
    bkgnd.setVolume(SOUND_LEVEL*.1);
    bkgnd.loop();
} //init

public void keyPressed(KeyEvent e)
{
    int key = e.getKeyCode();
    //u1 direction changes
    if ( key == KeyEvent.VK_UP) u1YVel = u1YVel - 1;
    else if( key == KeyEvent.VK_DOWN) u1YVel = u1YVel + 1;
    else if (key == KeyEvent.VK_RIGHT) u1XVel = u1XVel + 1;
    else if (key == KeyEvent.VK_LEFT) u1XVel = u1XVel - 1;
    //raise the shields
}

load the sound files
set the volumes relative to SOUND_LEVEL. a change of SOUND_LEVEL will change all sound levels.

start the background music
else if (key == KeyEvent.VK_SPACE)
{
    u1.raiseShields(time);
    shieldUses.setLabel("shield uses: " + u1.getShieldUses());
}
} //keyPressed

public void run()
{
    while(!exit) //game loop
    {
        time = time + PAUSE;
        pause(PAUSE);
        if (collides(u1, bomb) == true)
        {
            if (u1.shieldsAreUp() == true)
            {
                bump.play();
                u1XVel = -u1XVel;
                u1YVel = -u1YVel;
            }
            else
            {
                crash.play();
                exit = true;
                println("Collision without shields");
            }
        }
        else if ((fuelCell != null) && (collides(u1, fuelCell) == true))
        {
            u1.refuel();
            fuelUp.play();
            remove(fuelCell);
            fuelCell = null;
            println("refueled");
        }
        u1.move(u1XVel, u1YVel, time);
        fuelSign1.setLabel("u1 fuel: "+u1.getFuel());
        if (u1.shieldsAreUp())
            println("u1 shields up");
        else
            println("u1 shields down");
    } //game loop
println(u1);
} //run

public boolean collides(GObject ufo, GObject x)
{
    GRectangle ufoBox = ufo.getBounds();
    GRectangle xBox = x.getBounds();
    if (ufoBox.intersects(xBox) == true) return true;
    else return false;
} //collides
} //TestNewUFO2WithSound

One thing is worth a special note. The sound clip played when the ufo moves is about eight seconds long. Because of this, the ufo may stop moving in the window while the motor sound continues to play for several seconds.

One last thing—please use some sense and good taste. The example above has so much sound that it's more of a distraction than anything else. Try removing the background music and run the program again. You'll see that it's a much better program or at least it's less irritating.

Problem Set

FlyUFO2 Create a UFO to fly across the screen. The UFO has lights that blink every second and an exhaust that flickers every ½ second.

ChristmasTree Using time slicing, create a Christmas tree with three strings of lights. The first string should blink on and off every 3 seconds, the second every 1 second but out of synch with the first. The third string lights randomly. Use an an array or ArrayList to hold the three strings of lights.

YouChooseWithSound Modify your YouChoose program, including appropriate sound.

MusicPlayer Find several complete music selections in suitable format. Create a mouse controlled music player. It should, at a minimum, allow to you to play music, and pause and resume playing music, adjust the sound level of the music and switch from one music file to another.

PianoPlayer1 Find software for creating sound files of piano notes in a compatible file. Use it to create a virtual piano, using your keyboard for the piano keys. Try playing a tune.

PianoPlayer2 Modify PianoPlayer1 to allow recording by storing the notes in an array. Try your hand at composing music.