a different update rule than the original lattice. The coarse grained lattice should be updated after every three updates of the original lattice. Draw the coarse grained lattice as a space-time diagram similar to what we have done for the original lattice, such that each cell in the coarse grained lattice is three times the size of a cell on the original lattice in both the space and time directions. Use rule 146 (10010010) for the original lattice and rule 128 (10000000) for the coarse grained lattice. Choose a lattice size L that is a multiple of 3 and run for a time that is a multiple of 3. You should see similar patterns in the two lattices, although the original lattice contains some details that are washed out by the coarse grained lattice. If you coarse grain the original lattice cells at each time step, you will obtain the same pattern as the coarse grained lattice.

(b) Modify your program such that each pair of cells is coarse grained to 1 if two original cells are both 0 or both 1 and coarse grained to 0 otherwise. Use rule 105 (01101001) on the original cells with $L = 120$ for 60 iterations and run the coarse grained system using rule 150 (10100110). You should obtain results similar to those found in part (a). ■

Traffic models. Physicists have been at the forefront of the development of a more systematic approach to the characterization and control of traffic. Much of this work was initiated at General Motors by Robert Herman in the late 1950s. The car-following theory of traffic flow that he and Elliott Montroll and others developed during this time is still used today. What has changed is the way we can implement these theories. The continuum approach used by Herman and Montroll is based on partial differential equations. An alternative that is more flexible and easier to understand is based on cellular automata.

We first consider a simple one lane highway where cars enter at one end and exit at the other end. To implement the Nagel–Schreckenberg cellular automaton model, we use integer arrays for the position x_i and velocity v_i, where i indexes a car and not a lattice site. The important input parameters of the simulation are the maximum velocity $v_{\max}$, the density of cars ρ, and the probability p of a car slowing down. This probability adds some randomization to the drivers. The algorithm implemented in class `Freeway` for the motion of each car at each iteration is as follows:

1. If $v_i < v_{\max}$, increase the velocity v_i of car i by one unit, that is, $v_i \to v_i + 1$. This change models the process of acceleration to the maximum velocity.
2. Compute the distance to the next car d. If $v_i \geq d$, then reduce the velocity to $v_i = d - 1$ to prevent crashes.
3. With probability p, reduce the velocity of a moving car by one unit. Thus, $v_i \to v_i - 1$.
4. Update the position x_i of car i so that $x_i(t+1) = x_i(t) + v_i$.

This ordering of the steps ensures that cars do not overlap.

Listing 14.3 One lane freeway class.

```
package org.opensourcephysics.sip.ch14.traffic;
import java.awt.Graphics;
import org.opensourcephysics.display.*;
import org.opensourcephysics.frames.*;
```



```
import org.opensourcephysics.display2d.*;
import org.opensourcephysics.controls.*;

public class Freeway implements Drawable {
   public int[] v, x, xtemp;
   public LatticeFrame spaceTime;
   public double[] distribution;
   public int roadLength;
   public int numberOfCars;
   public int maximumVelocity;
   public double p;              // probability of reducing velocity
   private CellLattice road;
   public double flow;
   public int steps, t;
   // number of iterations before scrolling space-time diagram
   public int scrollTime = 100;

   public void initialize(LatticeFrame spaceTime) {
      this.spaceTime = spaceTime;
      x = new int[numberOfCars];
      xtemp = new int[numberOfCars]; // used to allow parallel updating
      v = new int[numberOfCars];
      spaceTime.resizeLattice(roadLength, 100);
      road = new CellLattice(roadLength, 1);
      road.setIndexedColor(0, java.awt.Color.RED);
      road.setIndexedColor(1, java.awt.Color.GREEN);
      spaceTime.setIndexedColor(0, java.awt.Color.RED);
      spaceTime.setIndexedColor(1, java.awt.Color.GREEN);
      int d = roadLength/numberOfCars;
      x[0] = 0;
      v[0] = maximumVelocity;
      for(int i = 1;i<numberOfCars;i++) {
         x[i] = x[i-1]+d;
         if(Math.random()<0.5) {
            v[i] = 0;
         } else {
            v[i] = 1;
         }
      }
      flow = 0;
      steps = 0;
      t = 0;
   }

   public void step() {
      for(int i = 0;i<numberOfCars;i++) {
         xtemp[i] = x[i];
      }
      for(int i = 0;i<numberOfCars;i++) {
         if(v[i]<maximumVelocity) {
            v[i]++;                  // acceleration
         }
         // distance between cars
         int d = xtemp[(i+1)%numberOfCars]-xtemp[i];
         // periodic boundary conditions, d = 0 correctly treats one
         // car on road
```