

PHY 411-506 Computational Physics II
Chapter 12: Interdisciplinary Topics
Lecture 10

Monday April 28, 2008

Lecture Outline

Forest Fire Model	3
Initializing the forest	3
Local update rule	4
OpenGL Forest File Program	4
Results found by Bak and collaborators	12
 Lattice Gas Cellular Automaton Model for Fluid Flow	 14
OpenGL Lattice Gas Program	16
 Continuum Limit of Lattice Gas Cellular Automaton Model	 34
The equations of fluid dynamics	34
Microscopic lattice gas equations	37
Continuum limit of the lattice gas equations	40

Forest Fire Model

- The forest fire model of P. Bak, K. Chen, and C. Tang, *Phys. Lett. A* **147**, 297 (1990), is a simple *probabilistic* cellular automaton with complex behavior that mimics the spreading of fires in a forest.
- The forest is modeled as a square region of side L with a regular lattice of L^2 cells or sites which represent trees. A cell can have three states:
 - ◇ A living tree, which is colored green.
 - ◇ A tree on fire, which is colored yellow.
 - ◇ A dead tree, which is colored black.

Initializing the forest

- At time $t = 0$, the forest is populated using two parameters:
 - ◇ Each cell has a tree with probability p_t , i.e., generate a random number r in the unit interval $0 < r < 1$, and color the site green if $r < p_t$, or black otherwise.
 - ◇ If the cell has a tree (if $r < p_t$ in the previous step) then set the tree on fire with probability p_f , i.e., generate a uniform deviate r' and

color the tree yellow if $r' < p_f$.

Local update rule

- At time t , each cell in the forest will be in one of three states, green, yellow, or black. The forest at time $t + 1$ is determined by the following rules:
 - ◇ If a tree is green, check its four nearest neighbors. If any neighbor is yellow (on fire), then the tree catches fire and is colored yellow.
 - ◇ If a tree is yellow (on fire), then it dies and is colored black.
 - ◇ If a tree is dead (black), then a new tree is grown with probability p , i.e., generate a uniform deviate r and color the cell green if $r < p$.
- Note that rules 1 and 2 are *deterministic*, but rule 3 is *probabilistic*: this model is therefore a *probabilistic* cellular automaton.

OpenGL Forest File Program

- The following program implements the model using OpenGL graphics.

`fire.cpp`

```
/*
   Two-dimensiona Forest Fire Model
*/

#include <cstdlib>
#include <iostream>
using namespace std;

#ifdef __APPLE__
# include <GLUT/glut.h>
#else
# include <GL/gl.h>
# include <GL/glu.h>
# include <GL/glut.h>
#endif

int L = 400;           // number of trees in x and y
double pt;            // initial probability to have a tree
double pf;            // probability this tree is on fire
double p;              // probability to grow a tree
enum {LIVE, FIRE, DEAD}; // 3 states of cells
int **tree;           // the trees
int **oldTree;        // previous for synchronous updating
GLubyte *image;       // for drawing forest image
bool running;         // boolean variable to start/stop simulation
```

```
double pMult = 2;                // factor to increase/decrease p using menu

void allocate() {
    static int oldL = 0;
    if (L != oldL) {
        if (tree != 0) {
            for (int i = 0; i < oldL; i++) {
                delete [] tree[i];
                delete [] oldTree[i];
            }
            delete [] tree;
            delete [] oldTree;
            delete [] image;
        }
        tree = new int* [L];
        oldTree = new int* [L];
        for (int i = 0; i < L; i++) {
            tree[i] = new int [L];
            oldTree[i] = new int [L];
        }
        image = new GLubyte [9 * L * L];
    }
}

void initialize() {
```

```
allocate();
for (int i = 0; i < L; i++)
for (int j = 0; j < L; j++) {
    if (rand() / double(RAND_MAX) < pt) {
        tree[i][j] = oldTree[i][j] = LIVE;
        if (rand() / double(RAND_MAX) < pf) {
            tree[i][j] = oldTree[i][j] = FIRE;
        }
    } else {
        tree[i][j] = oldTree[i][j] = DEAD;
    }
}
}
```

```
void takeStep() {

    // save state of the forest
    for (int i = 0; i < L; i++)
    for (int j = 0; j < L; j++)
        oldTree[i][j] = tree[i][j];

    // probabilistic automaton update rule
    for (int i = 0; i < L; i++)
    for (int j = 0; j < L; j++) {
        switch (oldTree[i][j]) {
```

```
    case LIVE:
        // use periodic boundary conditions
        if ( oldTree[i > 0 ? i - 1 : L - 1][j] == FIRE ||
            oldTree[i < L - 1 ? i + 1 : 0][j] == FIRE ||
            oldTree[i][j > 0 ? j - 1 : L - 1] == FIRE ||
            oldTree[i][j < L - 1 ? j + 1 : 0] == FIRE      )
            tree[i][j] = FIRE;
        break;
    case FIRE:
        tree[i][j] = DEAD;
        break;
    case DEAD:
        if (rand() / double(RAND_MAX) < p)
            tree[i][j] = LIVE;
        break;
    default:
        break;
}
}
glutPostRedisplay();
}

void reshape(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
    gluOrtho2D(0, w, 0, h);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

GLubyte colors[3][3] = { {0, 255, 0}, {255, 255, 0}, {0, 0, 0} };

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    for (int i = 0; i < L; i++)
        for (int j = 0; j < L; j++)
            for (int k = 0; k < 3; k++)
                image[3 * (L * i + j) + k] = colors[tree[i][j]][k];
    glRasterPos2i(0, 0);
    glDrawPixels(L, L, GL_RGB, GL_UNSIGNED_BYTE, image);
    glutSwapBuffers();
}

void mouse (int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
        if (running) {
            glutIdleFunc(NULL);
            running = false;
        } else {
```

```
        glutIdleFunc(takeStep);
        running = true;
    }
}

void reset(int menuItem) {
    switch (menuItem) {
    case 1:                // user selects "Increase p" from popup
        p *= pMult;
        if (p > 1)
            p = 1;
        break;
    case 2:                // user selects "Decrease p" from popup
        p /= pMult;
        break;
    default:
        break;
    }
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    if (argc > 1)
        L = atoi(argv[1]);
}
```

```
    cout << " Enter probabiltiy p_t: ";
    cin >> pt;
    cout << " Enter probabiltiy p_f: ";
    cin >> pf;
    cout << " Enter probabiltiy p: ";
    cin >> p;
    initialize();
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(L, L);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Two-dimensional forest fire automaton");
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glShadeModel(GL_FLAT);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(NULL);
    glutMouseFunc(mouse);
    glutCreateMenu(reset);           // create a popup menu
    glutAddMenuEntry("Increase p", 1);
    glutAddMenuEntry("Decrease p", 2);
    glutAttachMenu(GLUT_RIGHT_BUTTON); // attach it to right mouse button
    glutMainLoop();
}
```

Results found by Bak and collaborators

- Recall that the dynamics of the model depends on one parameter, namely the probability p for growing new trees from dead stumps. The system is more or less independent of the starting configuration. Bak et al. found the following results for this model:

- ◇ The system is characterized by a *correlation length*

$$\xi(p) \sim \frac{1}{p^\nu},$$

where $\nu \approx 1.0$ in this 2-dimensional model. (Bak et al. also studied the system in 3 dimensions.)

- If the correlation length is larger than the linear size of the system

$$\xi(p) > L,$$

then the fires die out in a time $\sim L$. The growth of new trees from dead stumps is not large enough to sustain the fires.

- If the correlation length is smaller than the linear size of the system

$$\xi(p) < L,$$

then the fires are sustained: the system reaches a critical steady state in which the growth of new trees feeds the fires with sufficient fuel to keep them burning for ever.

- ◇ The distribution of fires in the critical state is a *fractal*, i.e., an object with fractional dimension. In the 2-dimensional model, they found that the fractal dimension of the fire fronts $D \simeq 1.0$, i.e., the fronts are approximately linear; but in 3 dimensions, they found $D \simeq 2.5$, i.e., intermediate between a surface ($D = 2$) and a volume ($D = 3$) distribution.
- There are many variations on the simple forest fire cellular automaton model, which show interesting behavior.
- This Java Applet implements a model with 7 different states and 8 probabilities.

Lattice Gas Cellular Automaton Model for Fluid Flow

- This surprisingly effective model of the 2-D Navier-Stokes equations was published by U. Frisch, B. Hasslacher and Y. Pomeau in *Phys. Rev. Lett.* **56**, 1505 (1986).
- Their model has the following properties:
 - ◇ They found that a lattice with *hexagonal symmetry* was required to include a sufficient degree of rotational symmetry necessary for the conservation of angular momentum in a continuous fluid:

```

X   X   X   X   X
  X   0   0   X   X
X   0   X   0   X
  X   0   0   X   X
X   X   X   X   X

```

- ◇ To simulate variable density, each cell can have up to six fluid particles of mass $m = 1$. Thus the fluid density ρ can take 7 different values.
- ◇ The *free-streaming* (Euler) properties of the model are implemented by the rules

- Fluid particles have one of 6 possible velocities. The velocities are such that the particle moves to one of six neighboring cells in one time step.
- The velocities of particles in a particular cell *must* be distinct.
- ◇ The *viscous* (Navier-Stokes) properties are implemented by a set of collision rules. If there are 2, 3 or 4 particles at a site after the free-streaming rule has been applied, the particles are made to collide and change their velocities according to rules which conserve momentum (an x represents a particle with velocity in that particular direction):

◇

```

  ○ ○
x   x   =>   ○ x   OR   x ○
  ○ ○         ○   ○         ○   ○
  ○ ○         x   ○         ○   x

```

◇

```

  x ○
○   x   =>   ○ x
  x ○         x   ○
  ○ ○         ○   x

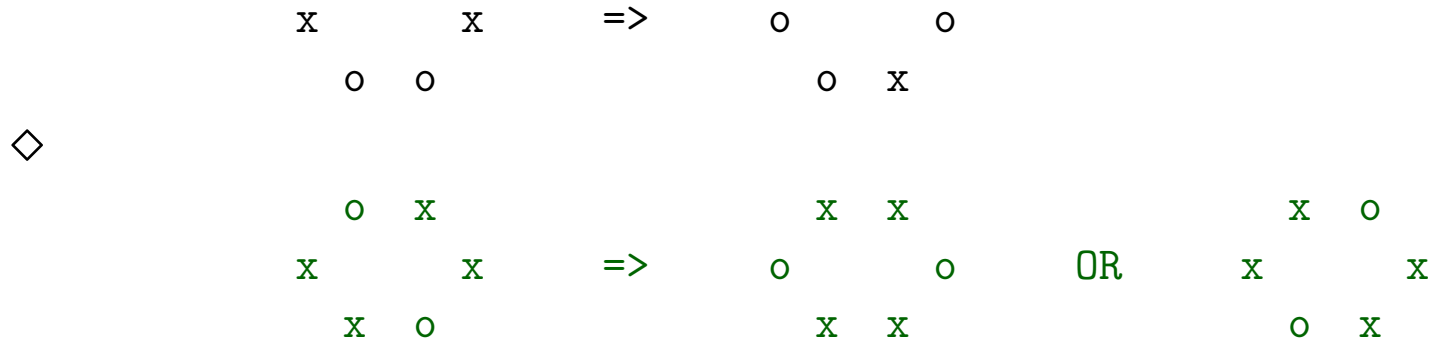
```

◇

```

  ○ x         x x

```



- In the first and fourth rules, there are two equivalent choices.
 - ◇ One possibility is to select them at random: however this would make the automata probabilistic or stochastic.
 - ◇ To keep the system deterministic the three choices in each case can be cycled in some order: choice of a particular order breaks *handedness* or *chiral* symmetry.
- The Lattice Gas Java Applet implements these rules.

OpenGL Lattice Gas Program

- The following program is a C++ version of the Java Applet.

latgas.cpp

/*

```
Lattice Gas Cellular Automaton Model of Fluid Flow
*/

#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

#ifdef __APPLE__
# include <GLUT/glut.h>
#else
# include <GL/gl.h>
# include <GL/glu.h>
# include <GL/glut.h>
#endif

const double pi = 4 * atan(1.0);
const double cos30 = cos(pi / 6);

const int RI = 1, RD = 2, LD = 4, LE = 8, LU = 16, RU = 32, S = 64, B = 128;
const int mask[] = {RI, RD, LD, LE, LU, RU, S, B};
const double vDirection[][2] = {{1, 0}, {0.5, -cos30}, {-0.5, -cos30},
                                  {-1, 0}, {-0.5, cos30}, {0.5, cos30}};
const int nv[] = {3, 4, 5, 0, 1, 2};
```

```
int Nx = 20;           // number of cells in x direction
int Ny = 20;           // number of cells in y direction
int bx = 4;            // x extent of initial block
int by = 4;            // y extent of initial block
int injectDelay = 0;   // time steps to delay injection

int t;                 // current time step
bool running;          // is the animation running?
bool flow;             // particles injected from left?

int rule[256];
void ruletable () {
    // 6-bit saturated deterministic rule
    for (int i = 0; i < 256; i++)
        rule[i] = i;
    rule[RI|LD|LU] = RD|LE|RU;
    rule[RD|LE|RU] = RI|LD|LU;
    rule[RI|LE] = LD|RU;
    rule[RD|LU] = RI|LE;
    rule[LD|RU] = RD|LU;
    rule[RI|RD|LE|LU] = RI|LD|LE|LU;
    rule[RI|LD|LE|LU] = RD|LD|LU|RU;
    rule[RD|LD|LU|RU] = RI|RD|LE|LU;
    rule[RI|RD|LU] = RI|LD|RU;
```

```
rule[RI|LD|RU] = RI|RD|LU;
rule[RD|LU|RU] = RI|LE|RU;
rule[RI|LE|RU] = RD|LU|RU;
rule[RD|LD|LU] = RI|LD|LE;
rule[RI|LD|LE] = RD|LD|LU;
rule[RD|LE|LU] = LD|LE|RU;
rule[LD|LE|RU] = RD|LE|LU;
rule[RI|RD|LE] = RD|LD|RU;
rule[RD|LD|RU] = RI|RD|LE;
rule[RI|LE|RU] = LD|LU|RU;
rule[LD|LU|RU] = RI|LE|RU;
}

int *lat, *latn;
int **nn;

void nntable () {
    // nn[n][dir] gives the neighbor whose particle moving
    // in direction dir will move to site n
    for (int j = 0; j < Ny; j += 2) {
        for (int i = 0; i < Nx; i++) {
            int n = j * Nx + i;
            int ip = i + 1;
            if (ip > Nx - 1)
                ip = 0;
        }
    }
}
```

```
        int im = i - 1;
        if (im < 0)
            im = Nx - 1;
        int jp = j + 1;
        if (jp > Ny - 1)
            jp = 0;
        int jm = j - 1;
        if (jm < 0)
            jm = Ny - 1;
        nn[n][0] = j * Nx + im;
        nn[n][1] = jp * Nx + im;
        nn[n][2] = jp * Nx + i;
        nn[n][3] = j * Nx + ip;
        nn[n][4] = jm * Nx + i;
        nn[n][5] = jm * Nx + im;
    }
}
for (int j = 1; j < Ny; j += 2) {
    for (int i = 0; i < Nx; i++) {
        int n = j * Nx + i;
        int ip = i + 1;
        if (ip > Nx - 1)
            ip = 0;
        int im = i - 1;
        if (im < 0)
```

```
        im = Nx - 1;
    int jp = j + 1;
    if (jp > Ny - 1)
        jp = 0;
    int jm = j - 1;
    if (jm < 0)
        jm = Ny - 1;
    nn[n][0] = j * Nx + im;
    nn[n][1] = jp * Nx + i;
    nn[n][2] = jp * Nx + ip;
    nn[n][3] = j * Nx + ip;
    nn[n][4] = jm * Nx + ip;
    nn[n][5] = jm * Nx + i;
    }
}
}
```

```
void injectParticlesFromLeft() {
    for (int j = 0; j < Ny; j++) {
        int n = j * Nx;
        lat[n] = 35;
    }
}
```

```
void removeParticlesFromRight () {
```

```
    for (int j = 0; j < Ny; j++) {
        int n = j * Nx + Nx - 1;
        lat[n] &= 28;
    }
}

void initialize() {

    // allocate memory for arrays
    static int oldNx = 0, oldNy = 0;
    if (Nx != oldNx || Ny != oldNy) {
        if (lat != 0) {
            for (int i = 0; i < oldNx * oldNy; i++)
                delete [] nn[i];
            delete [] nn;
            delete [] lat;
            delete [] latn;
        }
        lat = new int [Nx * Ny];
        latn = new int [Nx * Ny];
        nn = new int* [Nx * Ny];
        for (int i = 0; i < Nx * Ny; i++)
            nn[i] = new int [6];
    }
}
```

```
// initialize tables
nntable();
rulerule();

// empty the lattice
for (int n = 0; n < Nx * Ny; n++)
    lat[n] = latn[n] = 0;

// fill the block in the center of the lattice with 6 particles per site
for (int j = -by; j <= by; j++) {
    for (int i = -bx; i <= bx; i++) {
        int n = (j + Ny / 2) * Nx + i + Nx / 2;
        if (flow)
            lat[n] = B;
        else
            lat[n] = RI|RD|LD|LE|LU|RU;
    }
}

if (flow)
    injectParticlesFromLeft();

t = 0;
}
```

```
double framesPerSec = 10;           // control animation rate if too fast

void takeStep() {
    ++t;
    clock_t start_time = clock();    // get CPU clock time
    if (flow)
        removeParticlesFromRight();

    // bounce back boundary conditions v goes to -v
    // move particles
    for (int j = 0; j < Ny; j++) {
        for (int i = 0; i < Nx; i++) {
            int n = j * Nx + i;
            for (int dir = 0; dir < 6; dir++) {
                if ((lat[nn[n][dir]] & mask[dir]) != 0) {
                    if ((lat[n] & mask[7]) != 0)
                        // reflection
                        latn[nn[n][dir]] |= mask[nv[dir]];
                    else
                        // particle moves from nearest neighbor
                        latn[n] |= mask[dir];
                }
            }
        }
    }
}
```

```
// collisions
for (int j = 0; j < Ny; j++) {
    for (int i = 0; i < Nx; i++) {
        int n = j * Nx + i;
        if ((lat[n] & mask[7]) == 0) {
            lat[n] = rule[latn[n]];
            latn[n] = 0;
        }
    }
}

if (flow) {
    if (injectDelay == 0 || t % injectDelay == 0)
        injectParticlesFromLeft();
}

// delay if animation is too fast
while ((double(clock()) - start_time)/CLOCKS_PER_SEC < 1.0 / framesPerSec)
    ;

glutPostRedisplay();
}

void drawDisk(double x, double y, double r, int slices=24) {
```

```

    glBegin(GL_TRIANGLE_FAN);
        glVertex2d(x, y);
        double phi = 2 * pi / double(slices);
        for (int j = 0; j < slices + 1; j++)
            glVertex2d(x + r * cos(phi * j), y + r * sin(phi * j));
    glEnd();
}

```

```

void display() {
    static GLubyte colors[7][3] = {
        {(255*0)/6, (255*6)/6, 0}, {(255*1)/6, (255*5)/6, 0},
        {(255*2)/6, (255*4)/6, 0}, {(255*3)/6, (255*3)/6, 0},
        {(255*4)/6, (255*2)/6, 0}, {(255*5)/6, (255*1)/6, 0},
        {(255*6)/6, (255*0)/6, 0}
    };
    glClear(GL_COLOR_BUFFER_BIT);

    for (int i = 0; i < Nx; i++) {
        for (int j = 0; j < Ny; j++) {
            double dx = 0;
            if (j%2 != 0)
                dx = 0.5;
            int n = lat[j * Nx + i];
            if (n == B) {
                // draw barrier
            }
        }
    }
}

```

```
        glColor3ub(0, 0, 255);
        drawDisk(i + dx, j * cos30, 0.5);
    } else {
        int m = n;
        int k = 0;
        for (int bit = 0; bit < 6; bit++) {
            if ((m & 1) == 1)
                k += 1;
            m >>= 1;
        }
        if (k > 0) {
            // draw fluid particles
            glColor3ubv(colors[k]);
            drawDisk(i + dx, j * cos30, 0.5);
            // draw velocity vectors
            glColor3ub(0, 0, 0);
            for (int k = 0; k < 6; k++) {
                if ((n & mask[k]) == mask[k]) {
                    glBegin(GL_LINES);
                    glVertex2d(i + dx, j * cos30);
                    glVertex2d(i + dx + 0.5 * vDirection[k][0],
                               j * cos30 + 0.5 * vDirection[k][1]);
                    glEnd();
                }
            }
        }
    }
```

```

        } else {
            // draw lattice point
            glColor3ub(127, 127, 127);
            drawDisk(i + dx, j * cos30, 0.1);
        }
    }
}

glutSwapBuffers();
}

int VPx0 = 0;           // left pixel of viewport on window
int VPdx = 400;        // width of viewport on window in pixels
int VPy0 = 0;         // bottom pixel of viewport on window
int VPdy = 400;       // height of viewport on window in pixels

void reshape(int w, int h) {
    double aspect = w * ((Ny - 1) * cos30 + 1) / double(h * (Nx + 0.5));
    VPx0 = VPy0 = 0;
    VPdx = w;
    VPdy = h;
    if (aspect > 1) {
        VPx0 = int(w * (1 - 1 / aspect) / 2.0);
        VPdx = int(w / aspect);
    }
}

```

```

    } else {
        VPy0 = int(h * (1 - aspect) / 2.0);
        VPdy = int(h * aspect);
    }
    glViewport(VPx0, VPy0, VPdx, VPdy);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-0.5, Nx, -0.5, (Ny - 1) * cos30 + 0.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

bool addParticles;        // add 6 fluid particles at left mouse click position
bool addBarrier;         // add barrier at left mouse click position

void mouse(int button, int state, int x, int y) {
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    int j = int((VPdy - y + VPy0) / double(VPdy) * Ny);
    int i = int((x - VPx0) / double(VPdx) * (Nx+0.5) + (j%2 == 0 ? 0 : -0.5));
    int n;
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN) {
            if (i >= 0 && i < Nx && j >= 0 && j < Ny) {

```

```
        n = j * Nx + i;
        if (addParticles) {
            lat[n] = RI|RD|LD|LE|LU|RU;
        }
        if (addBarrier) {
            if (lat[n] == B)
                lat[n] = latn[n] = 0;
            else
                lat[n] = B;
        }
        glutPostRedisplay();
    }

}

break;
default:
    break;
}
}

void reset(int menuItem) {
    switch (menuItem) {
    case 1:
        if (running) {
            glutIdleFunc(NULL);
```

```
        running = false;
    } else {
        glutIdleFunc(takeStep);
        running = true;
    }
    break;
case 2:
    flow = !flow;
    break;
case 3:
    if (running) {
        glutIdleFunc(NULL);
        running = false;
    }
    addParticles = true;
    addBarrier = false;
    break;
case 4:
    if (running) {
        glutIdleFunc(NULL);
        running = false;
    }
    addBarrier = !addBarrier;
    addParticles = false;
    break;
```

```
    case 5:
        if (running) {
            glutIdleFunc(NULL);
            running = false;
        }
        initialize();
        glutPostRedisplay();
        break;
    default:
        break;
}
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    initialize();
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(VPdx, VPdy);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Lattice Gas Cellular Automaton Model of Fluid Flow");
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glShadeModel(GL_FLAT);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(NULL);
}
```

```
    glutMouseFunc(mouse);  
    glutCreateMenu(reset);           // create a popup menu  
    glutAddMenuEntry("Start/Stop", 1);  
    glutAddMenuEntry("Toggle Flow", 2);  
    glutAddMenuEntry("Add Particles", 3);  
    glutAddMenuEntry("Toggle Barrier", 4);  
    glutAddMenuEntry("Reset", 5);  
    glutAttachMenu(GLUT_RIGHT_BUTTON); // attach it to right mouse button  
    glutMainLoop();  
}
```

Continuum Limit of Lattice Gas Cellular Automaton Model

We have seen from running the program `latgas.cpp` that the 2D lattice gas on a hexagonal lattice approximates the behavior of a fluid.

It is interesting to see how the lattice gas automaton model approximates the equations of fluid dynamics. To do this, we need to

- form statistical averages over a large number of lattice gas systems, and
- examine the limit in which the *microscopic scale* of the model, i.e., the lattice spacing which determines the *mean free path* of the fluid particles, is much smaller than the *macroscopic scale* of variations in fluid density and velocity.

The equations of fluid dynamics

The equations of fluid dynamics follow from conservation of mass and momentum.

Consider a volume V inside the fluid. The mass of fluid in this volume is given by

$$\int \rho \, dV,$$

where ρ is the fluid density. The rate at which this mass decreases is determined by the rate at which fluid leaves the volume

$$\frac{d}{dt} \int \rho \, dV = - \int \rho \mathbf{u} \cdot d\mathbf{S} ,$$

where \mathbf{u} is the fluid velocity and the integral on the right is taken over the surface of the volume with $d\mathbf{S}$ being a surface element with direction along the *outward* normal. Using the *divergence theorem*

$$\int \rho \mathbf{u} \cdot d\mathbf{S} = \int \nabla \cdot \rho \mathbf{u} \, dV ,$$

we obtain the *continuity equation*

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \mathbf{u} = 0 .$$

Next, consider conservation of momentum

$$\rho \frac{d\mathbf{u}}{dt} = \mathbf{F} ,$$

which is just Newton's equation of motion for an element of fluid with unit mass. The total derivative on the left has two contributions

$$\frac{d\mathbf{u}}{dt} = \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} ,$$

the first term on the right represents the change in fluid velocity with t at a *fixed* point in space, and the second *advective* term represents the change in fluid velocity due to motion of fluid from neighboring points in space.

The force density \mathbf{F} has three contributions:

- external or *body forces* acting on the fluid, for example the force of gravity

$$\mathbf{F}_{\text{gravity}} = \rho \mathbf{g} ,$$

where \mathbf{g} is the acceleration due to gravity

- *pressure forces* due to neighboring fluid elements

$$- \int p \, d\mathbf{S} = - \int \nabla p \, dV ,$$

$$\mathbf{F}_{\text{pressure}} = -\nabla p ,$$

where p is the fluid pressure and the integrals are taken over the surface and volume of the element, respectively

- *viscous forces* due to internal friction or shearing stresses in the moving fluid

$$\mathbf{F}_{\text{viscous}} = \mu \nabla^2 \mathbf{u} + (\mu + \xi) \nabla (\nabla \cdot \mathbf{u}) ,$$

where μ is the *dynamic viscosity coefficient* and ξ is the *bulk viscosity coefficient* of the fluid.

A special case that is interesting for many applications is that of *incompressible flow*

$$\rho = \text{constant} , \quad \nabla \cdot \mathbf{u} = 0 .$$

Taking these forces into account results in the *Navier-Stokes equations* for incompressible viscous flow:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = \mathbf{g} - \frac{1}{\rho} \nabla p + \nu \nabla^2 \mathbf{u} ,$$

where $\nu = \mu/\rho$ is the *kinematic viscosity*.

Microscopic lattice gas equations

The lattice gas on a 2D hexagonal lattice can be represented by a vector $\mathbf{n}(\mathbf{x}) = (n_0, n_1, n_2, n_3, n_4, n_5)$ of *Boolean* variables that indicate the presence ($n_i = 1$) or absence ($n_i = 0$) of particles at cell \mathbf{x} and moving with velocities

$$\mathbf{c}_i = \left[\cos \left(\frac{\pi i}{3} \right) , -\sin \left(\frac{\pi i}{3} \right) \right] , \quad i = 0, 1, 2, 3, 4, 5,$$

measured in units of lattice spacing per time step.

The synchronous dynamics of the system is described by the equation

$$n_i(\mathbf{x} + \mathbf{c}_i, t + 1) = n_i(\mathbf{x}, t) + \Delta_i[\mathbf{n}(\mathbf{x}, t)] ,$$

where the first term on the right hand side gives the result of free streaming of particle i from cell \mathbf{x} to cell $\mathbf{x} + \mathbf{c}_i$, and Δ_i is the *collision term*. In this equation, $n_i(\mathbf{x}, t)$ is the result of free streaming from time $t - 1$. The collision term then alters the particle velocities according to the collision rules of the model.

To obtain an explicit formula for Δ_i we define the *negation* of n_i to be $\bar{n}_i = 1 - n_i$: if $n_i = 1$ then $\bar{n}_i = 0$ and vice versa. Consider for example a 3-particle collision with zero total momentum for which there are 2 states

$$\text{east} \mid \text{southwest} \mid \text{northwest} \Leftrightarrow \text{southeast} \mid \text{west} \mid \text{northeast}$$

which get exchanged when the particles collide. It is easy to see that this rule can be written

$$\Delta_i^{(3)} = n_{i+1}n_{i+3}n_{i+5}\bar{n}_i\bar{n}_{i+2}\bar{n}_{i+4} - n_in_{i+2}n_{i+4}\bar{n}_{i+1}\bar{n}_{i+3}\bar{n}_{i+5} ,$$

where the sums in the subscripts are taken modulo 6.

As another example, the cyclic 2-body collision rule

east | west \Rightarrow southwest | northeast \Rightarrow northwest | southeast

is represented by the collision term

$$\Delta_i^{(2)} = n_{i+2}n_{i+5}\bar{n}_i\bar{n}_{i+1}\bar{n}_{i+3}\bar{n}_{i+4} - n_in_{i+3}\bar{n}_{i+1}\bar{n}_{i+2}\bar{n}_{i+4}\bar{n}_{i+5} .$$

Similar expressions can be written for the other 3-particle collisions, and the cyclic 4-particle collisions. Then Δ_i is the *sum* of all of these mutually exclusive collision terms.

Note that each collision term conserves mass and momentum

$$\sum_i \Delta_i(\mathbf{n}) = 0 , \quad \sum_i \mathbf{c}_i \Delta_i(\mathbf{n}) = 0 .$$

Using these relations, we can write the lattice analog of the continuity equation as

$$\sum_i n_i(\mathbf{x} + \mathbf{c}_i, t + 1) = \sum_i n_i(\mathbf{x}, t) ,$$

and the lattice analog of momentum conservation as

$$\sum_i \mathbf{c}_i n_i(\mathbf{x} + \mathbf{c}_i, t + 1) = \sum_i \mathbf{c}_i n_i(\mathbf{x}, t) .$$

Continuum limit of the lattice gas equations

Frish, Hasslacher, and Pomeau, *Phys. Rev. Lett.* **56**, 1505 (1986) showed that the lattice gas equations reduced to the Navier-Stokes equations in the continuum limit for small enough fluid velocities.

The lattice gas variables are the boolean occupation numbers n_i at each site. Let $N_i = \langle n_i \rangle$ be statistical average values of these variables, i.e., averages over many time steps after the system has come to a steady state. The fluid density ρ and velocity \mathbf{u} are then defined as

$$\rho = \sum_i N_i, \quad \rho \mathbf{u} = \sum_i N_i \mathbf{c}_i.$$

By assuming that ρ and \mathbf{u} vary slowly in space (measured in units of the lattice spacing) and time (measured in automaton time steps), and making Taylor series expansions in \mathbf{u} , Frish et al., were able to derive the following equations:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0,$$

$$\begin{aligned} \frac{\partial}{\partial t}(\rho u_\alpha) + \sum_{\beta} \frac{\partial}{\partial x_\beta} \left[g(\rho) \rho u_\alpha u_\beta + \mathcal{O}(u^4) \right] \\ = -\frac{\partial}{\partial x_\alpha} p + \eta_1(\rho) \nabla^2 u_\alpha + \eta_2(\rho) \frac{\partial}{\partial x_\alpha} \nabla \cdot \mathbf{u} , \end{aligned}$$

where $g(\rho) = (\rho - 3)/(\rho - 6)$, $p = (\rho/2)[1 - g(\rho)u^2]$, and $\eta_{1,2}$ are viscosity coefficients which depend on ρ .

Notice that these equations are *rotationally invariant*, i.e., the underlying hexagonal symmetry of the lattice gas has disappeared in the continuum limit! The second equation looks very similar to the Navier-Stokes equation, but there are some differences:

- The factor $g(\rho)$ on the left hand side is not present in the Navier-Stokes equations. This term breaks Galilean invariance, i.e., the equation is not invariant under a transformation to a moving reference frame.
- The pressure p is velocity dependent and does not give the correct value for the speed of sound $c_s = \sqrt{dp/d\rho}$ in the fluid.

In addition to these difficulties, it has been found that lattice gas simulations suffer from the following problems:

- The fluctuations in the fluid flow are too large: the flow is too noisy.
- Only low Reynolds numbers, i.e., very viscous fluid flows, can be simulated correctly.
- There are unphysical conserved quantities in addition to the physical conserved quantities (mass, momentum, energy, angular momentum).
- The collision rules become exponentially complex in many models, including the simplest realistic 3-D model.

The last problem is interesting. In the 2-D model, each cell can have up to 6 particles, each with a different velocity. The number of possible states is $2^6 = 64$ because each particle type can either be present or absent. In a collision, each state is transformed to a definite other state. One way of implementing collisions is with a 64×64 lookup table.

In 2-D, Frisch, Hasslacher and Pomeau found that a hexagonal lattice of cells was sufficient to recover isotropy (rotational invariance) in the Navier-Stokes equations: an earlier model with a square lattice did not have this property and did not give realistic results.

In 3-D it can be shown that there there is *no* regular crystal lattice that

leads to isotropic Navier-Stokes flows! A clever trick was found to solve this problem: use a 4-D face-centered hyper-cubic lattice. The velocities in 4-D are taken to be $c_1 = [\pm 1, \pm 1, 0, 0]$ and 23 other permutations. These velocities are projected onto a cube in 3-D to give

- 12 velocities to the centers of the 12 cube edges. These velocities have zero fourth-component. A cell in 3-D is allowed to have at most one of these particles.
- 6 velocities to the centers of the 6 cube faces. These velocities have fourth-component ± 1 , so the cell in 3-D is allowed to have at most *two* of these particles.

With 24 different velocities possible at each cell, the number of states is of order $2^{24} = 16,777,216$. It is not easy to specify appropriate collision rules for such a huge number of states. Once the rules are specified, it is very difficult to code them. The earliest calculations with this model were done using a 48 MByte lookup table which took about 2 hours of CRAY-2 CPU time to initialize properly!