

Lattice Gas Cellular Automaton Model for Fluid Flow

```
// Lattice Gas Cellular Automaton Model for Fluid Flow

#include <cmath>
#include <cstdlib>
#include <iostream>
using namespace std;
#include <GL/glut.h>

const double pi = 4 * atan(1.0);
const double cos30 = cos(pi / 6);
typedef unsigned char byte;

double Lx = 200;           // x extent in units of lattice spacing
double Ly = 50;           // y extent in units of lattice spacing
int Nx, Ny;               // number of cells in x and y
byte **cell;              // cells
byte **cellNext;          // to help update synchronously
byte rule[256];           // collision rules

const byte
    east = 1,
    southeast = 2,
    southwest = 4,
    west = 8,
    northwest = 16,
    northeast = 32,
    zero = 64,
```

```
    obstacle = 128;

double xObstacle, yObstacle, rObstacle;

int injectInterval;          // interval for injecting particles from left

void allocate() {
    static int oldNx = 0;
    if (oldNx != Nx) {
        if (cell != NULL) {
            for (int i = 0; i < oldNx; i++) {
                delete [] cell[i];
                delete [] cellNext[i];
            }
            delete [] cell;
            delete [] cellNext;
        }
        oldNx = Nx;
        cell = new byte * [Nx];
        cellNext = new byte * [Nx];
        for (int i = 0; i < Nx; i++) {
            cell[i] = new byte [Ny];
            cellNext[i] = new byte [Ny];
        }
    }
}

bool insideObstacle(int i, int j) {
    double x = i;
```

```
double y = j * cos30;
double rSqd = (x - x0bstacle) * (x - x0bstacle) +
              (y - y0bstacle) * (y - y0bstacle);
return rSqd <= r0bstacle * r0bstacle;
}
```

```
double vx[64], vy[64], v[64], r[64];    // for plotting
```

```
void makePlotTables() {
```

```
    // compute all possible combinations of velocities
    double vmax = 0;
    for (int i = 0; i < 64; i++) {
        vx[i] = vy[i] = r[i] = 0;
        int n = i;
        double theta = 0;
        for (int j = 0; j < 6; j++) {
            if ((n & 1) == 1) {
                vx[i] += cos(theta);
                vy[i] += sin(theta);
                r[i] += 1;
            }
            n >>= 1;
            theta -= pi / 3;
        }
        v[i] = sqrt(vx[i] * vx[i] + vy[i] * vy[i]);
        r[i] = sqrt(r[i] / 12.0);
        if (v[i] > vmax)
            vmax = v[i];
    }
```

```
}

// normalize them for plotting
for (int i = 0; i < 63; i++) {
    if (v[i] > 0) {
        vx[i] *= 1 / vmax;
        vy[i] *= 1 / vmax * 2 / sqrt(3.0);
    }
}

}

void initialize() {
    Nx = int(Lx);
    Ny = int(ceil(Ly / cos30));
    allocate();
    xObstacle = Lx / 2;
    yObstacle = Ly / 2;
    rObstacle = Ly / 4;
    for (int i = 0; i < Nx; i++)
    for (int j = 0; j < Ny; j++) {
        cell[i][j] = cellNext[i][j] = 0;
        if (insideObstacle(i, j) || j == 0 || j == Ny - 1)
            cell[i][j] = cellNext[i][j] = obstacle;
        else
            cell[i][j] = cellNext[i][j] = 0;
    }

    // fill collision rule table
    for (int i = 0; i < 256; i++)
```

```
rule[i] = i;

// two-particle collisions
byte i = east | west;
byte j = southwest | northeast;
byte k = northwest | southeast;
rule[i] = j; rule[j] = k; rule[k] = i;

// three-particle collisions
i = east | southeast | west;
j = southeast | southwest | northeast;
rule[i] = j; rule[j] = i;

i = east | southwest | west;
j = southeast | southwest | northwest;
rule[i] = j; rule[j] = i;

i = east | southeast | northwest;
j = east | southwest | northeast;
rule[i] = j; rule[j] = i;

i = east | southwest | northwest;
j = southeast | west | northeast;
rule[i] = j; rule[j] = i;

i = east | west | northwest;
j = southwest | northwest | northeast;
rule[i] = j; rule[j] = i;
```

```
i = southeast | west | northwest;
j = southwest | west | northeast;
rule[i] = j; rule[j] = i;

i = east | west | northeast;
j = southeast | northwest | northeast;
rule[i] = j; rule[j] = i;

// four-particle collisions
i = east | southeast | west | northwest;
j = east | southwest | west | northeast;
k = southeast | southwest | northwest | northeast;
rule[i] = j; rule[j] = k; rule[k] = i;

makePlotTables();
}

inline bool contains(const byte c, const byte d) {
    return (c & d) != 0;
}

inline void setNext(byte& cNext, const byte c, byte direction) {
    if (contains(c, direction)) {
        if (contains(cNext, obstacle)) {
            if (direction == east) cNext |= west;
            else if (direction == southeast) cNext |= northwest;
            else if (direction == southwest) cNext |= northeast;
            else if (direction == west) cNext |= east;
            else if (direction == northwest) cNext |= southeast;
```

```
        else if (direction == northeast) cNext |= southwest;
    } else {
        cNext |= direction;
    }
}

int step;

void takeStep() {

    // move particles to neighboring cells
    for (int i = 0; i < Nx; i++)
    for (int j = 0; j < Ny; j++) {

        // east
        int iN = i + 1, jN = j;
        if (iN < Nx)
            setNext(cellNext[iN][jN], cell[i][j], east);

        // south east
        iN = i + j % 2;
        jN = j - 1;
        if (jN >= 0 && iN >= 0 && iN < Nx)
            setNext(cellNext[iN][jN], cell[i][j], southeast);

        // south west
        iN = i - (j + 1) % 2;
        if (jN >= 0 && iN >= 0 && iN < Nx)
```

```
        setNext(cellNext[iN][jN], cell[i][j], southwest);

// west
iN = i - 1;
jN = j;
if (iN >= 0)
    setNext(cellNext[iN][jN], cell[i][j], west);

// north west
iN = i - (j + 1) % 2;
jN = j + 1;
if (jN < Ny && iN >= 0 && iN < Nx)
    setNext(cellNext[iN][jN], cell[i][j], northwest);

// north east
iN = i + j % 2;
if (jN < Ny && iN >= 0 && iN < Nx)
    setNext(cellNext[iN][jN], cell[i][j], northeast);
}

// apply collision rules
for (int i = 0; i < Nx; i++)
for (int j = 0; j < Ny; j++) {
    if (contains(cell[i][j], obstacle))
        cell[i][j] = cellNext[i][j];
    else {
        cell[i][j] = rule[cellNext[i][j]];
        cellNext[i][j] = 0;
    }
}
```

```
}

// inject particles at left end
if (injectInterval == 0 || step % injectInterval == 0) {
    for (int j = 1; j < Ny - 1; j++)
        cell[0][j] = east | northeast | southeast;
}

glutPostRedisplay();
++step;
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(0, 0, 255);
    for (int i = 0; i < Nx; i++)
        for (int j = 0; j < Ny; j++) {
            double dx = j % 2 == 0 ? 0 : 0.5;
            if (contains(cell[i][j], obstacle)) {
                glColor3ub(0, 0, 0);
                glRectd(i + dx, j * cos30, i + dx + 0.8, j * cos30 + 0.8);
            } else {
                glColor3ub(0, 0, 255);
                glBegin(GL_LINES);
                glVertex2d(i + dx, j * cos30);
                glVertex2d(i + dx + vx[cell[i][j]], j * cos30 + vy[cell[i][j]]);
                glEnd();
            }
        }
}
```

```
        glutSwapBuffers();
    }

void reshape(int w, int h) {
    int x0 = 0, y0 = 0, dx = w, dy = h;
    double aspect = (w * Ly) / (h * Lx);
    if (aspect > 1) {
        dx = int(dx / aspect);
        x0 = (w - dx) / 2;
    } else {
        dy = int(dy * aspect);
        y0 = (h - dy) / 2;
    }
    glViewport(x0, y0, dx, dy);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, Lx, 0, Ly);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

bool running;

void mouse (int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN) {
            if (running) {
```

```
        glutIdleFunc(NULL);
        running = false;
    } else {
        glutIdleFunc(takeStep);
        running = true;
    }
}
break;
default:
    break;
}
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    if (argc > 1)
        injectInterval = atoi(argv[1]);
    cout << injectInterval << endl;
    initialize();
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 200);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Lattice Gas Cellular Automaton Model");
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glShadeModel(GL_FLAT);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(takeStep);
    glutMouseFunc(mouse);
}
```

```
    glutMainLoop();  
}
```