1. Introduction – the concept.

I have chosen for my 1st project to simulate a fluid laminar flow and to show the Hagen-Poiseuille law using an FHP lattice solver. The program was written in c++ and uses the same OpenGL visualization technique as my 2nd project (Lattice Boltzmann solver).

The Hagen-Poiseuille law describes the pressure drop in a fluid flowing through a pipe with a constant cross section. The assumptions are that the fluid is incompressible and Newtonian, and that the tube’s radius is significantly lower than its length and that it is of cylindrical shape. As a result of this law we observe a velocity gradient in the pipe with a single maximum in its center, which is what my model aims to show.

The FHP method was named after its inventors, Uriel Frisch, Brosl Hasslacher and Yves Pomeau. It is a type of a lattice gas cellular automata with a hexagonal lattice. The lattice points are described by a discrete set of states, which symbolize particles with different velocities. Since the lattice is hexagonal, each point may contain up to 7 particles, moving in each of the allowed directions and one stationary. In each time step, the particles are moved in their respective directions and then a collision effect at each point is being calculated. We distinguish 3 FHP models depending on possible mass and momentum conserving collisions. The simplest, FHP-I, only computes 2 particle symmetric collisions (CM frame) and does not include the existence of stationary particles. FHP-II includes 3 particle collisions and the most advanced, FHP-III, also includes 4 and 3 particle asymmetric interactions. The method of computing these collisions vary depending on used solution for lattice data handling. The most efficient, in my opinion, is to treat each lattice point as an integer value of range 0-127, and an existence of a particle with a specific velocity as a bit value within this integer, therefore all interactions can be described as bitwise operations on that value.

2. Implementation

The model itself, and thus my implementation, is very simple. In the pre-main part of my code I define a 2d lattice, a temporary lattice for modifications, vectors for collision rules and a variety of handler variables and functions which I will describe in detail.

```c++
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<GL/glew.h>
#include<GL/glut.h>
#include<stdio.h>
#include<string>
#include<iostream>
#include<sstream>
#include<time.h>

//Headers for Memory Leak check
```
/**
#define CRTDBG_MAP_ALLOC

#include <crtdbg.h>*/

using namespace std;

#define LAST(k,n) (((k)&((1<<(n))-1))
#define MID(k,m,n) LAST((k)>>((n)-(m))
#define I2D(ni,i,j) (((ni)*(j))+i)

//OpenGL pixel buffer and texture handle objects
GLuint gl_PBO, gl_Tex;

//Arrays
int **grid,**new_grid;
int *ruleset1,*ruleset2;
float *cmap,*plotvar;
float width,height;
double *v_profile;
unsigned int *cmap_rgba, *plot_rgba;

//Scalars
int nx,ny,t,t_curr(0),n_color,win_x(800),win_y(200);
bool FHP2(false);

//OpenGL functions
void display();
void resize(int,int);

//FHP functions
void insert(int,int);
void move();
int mass(int,int);
double momentum_x(int,int);
double mass_avrg(int,int,int,int);

This part also includes a memory leak check which I need for debugging, since I operate on a large number of globally defined arrays, which is commonly considered a bad programming practice. My experience however has taught me that using locally defined structures during a simulation has a significant negative impact on processing time, due to the amount of function-to-function copy mechanisms. The solution could be to define custom objects with efficient copying creators, but using global arrays is much simpler.

int main(int argc, char** argv){
    srand(time(NULL));
    float r_color,g_color,b_color;
    char scalars[100];

    FILE *color_map,*param;

    //Reading sim parameters from file
    param=fopen("param.dat","r");
    if(param==NULL){
        printf("Error: can't open param.dat \n");
    }
system("pause");
    return 1;
}
fgets(scalars,100,param);
fgets(scalars,100,param);
std::istringstream iss(scalars);
iss>>nx>>ny>>t;
fclose(param);

printf("nx= %d\n",nx);
printf("ny= %d\n",ny);
printf("t= %d\n",t);

//Reading color map from file
color_map=fopen("cmap.dat","r");
if (color_map==NULL){
    printf("Error: can't open cmap.dat \n");
    system("pause");
    return 1;
}
fsscanf(color_map,"%d",&n_color);
cmap_rgb=new unsigned int[n_color*sizeof(unsigned int)];
for (int i=0;i<n_color;i++){
    fsscanf(color_map,"%f%f%f",&r_color,&g_color,&b_color);
    cmap_rgb[i]=(int)(255.0f)<<24) | ((int)(b_color*255.0f)<<16) | ((int)(g_color*255.0f)<<8) | ((int)(r_color*255.0f)<<0);
}
fclose(color_map);

This part is almost identical to my previous LB project. These functions prepare the simulation, load external variables, color maps for pixel vectors and define the seed for the pseudorandom algorithm.

//Initializing arrays
plotvar = new float[nx*ny*sizeof(float)];
for(int i=0;i<nx*ny;i++){plotvar[i]=0;}
plot_rgba = new unsigned int[nx*ny*sizeof(unsigned int)];
v_profile = new double[ny*sizeof(double)];
for(int i=0;i<ny;i++){
    v_profile[i]=0;
}
grid=new int*[nx*sizeof(int)];
new_grid=new int*[nx*sizeof(int)];
for(int i=0;i<nx;i++){
    grid[i]=new int[ny*sizeof(int)];
    new_grid[i]=new int[ny*sizeof(int)];
}
for(int i=0;i<ny;i++){
    for(int j=0;j<nx;j++){
        grid[j][i]=0;
        new_grid[j][i]=0;
    }
}
for(int i=0;i<ny;i++){
    for(int j=0;j<nx;j++){
        if(j==0 || j==ny-1){
            grid[j][i]=0;
            new_grid[j][i]=0;
        }
    }
}
grid[i][j]=128;
}
}

for(int i=0;i<(nx*ny*7)/10;i++){
    insert(rand()%nx,rand()%ny);
}

In this part all the relevant arrays and vectors are being defined and data is being pre allocated to them. All grid points are set to empty except for upper and lower horizontal edges, which are given a number (128). As I mentioned earlier, I describe lattice states using bit notation. The number 128 in binary is 10000000. The 8th bit in my notation represents an obstacle – an element which is not moving and anything colliding with it bounces back. Regarding my notation, the 7th bit represents stationary particles, the rest are velocity direction (in order of least significant bit up) upwards left, upwards right, right, downwards right, downwards left, left. This notation bears no significance, it was chosen by me at random and I am not sure if it was the right choice, as I was unable to devise a bitwise operation for collision handling and had to describe each possible collision manually. You may also notice that in this part the function “insert” is being called. It is a function for inserting particles into the grid with random velocities. It will be described later. The intention here is to fill the simulation area with approximately 10% maximum allowed density of particles. Later on the model will hold 30% density in a very narrow part of the grid to induce laminar flow.

ruleset1=new int[256];
ruleset2=new int[256];

//FHP-2 collision rules
for (int i=0;i<256;i++){
    ruleset1[i]=i;
    ruleset2[i]=i;
}

//2 particle symmetric collisions

//3 particle symmetric collisions
if(FHP2){
    //2 particle w/ stationary


    ruleset1[80]=40;ruleset1[40]=80;ruleset2[80]=40;ruleset2[40]=80;


These are my collision rules. The model distinguishes if I want it to run in FHP-I or FHP-II mode. Each undefined integer pair is set to be ignored. As I mentioned, I was unable to find a simpler way, and all of those numbers were calculated by me by hand, including collision rules, which inherently do not conserve momentum. However the model conserves mass, provided that the particles are surrounded by a layer of obstacles. I did not define any boundary conditions, just inserted obstacles where particles were not supposed to go.

//Initializing window
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize(win_x, win_y);
glutInitWindowPosition(50,50);
glutCreateWindow("FHP");

// Checking OpenGL extension support
printf("Loading extensions:
%s\n", glewGetErrorString(glewInit()));
if(glewIsSupported("GL_VERSION_2_0 "
"GL_ARB_pixel_buffer_object ""GL_EXT_framebuffer_object ")){
  printf(stderr,"ERROR: Support for necessary OpenGL
extensions missing.");
  fflush(stderr);
  system("pause");
  return 2;
}

gClearColor(0.0,0.0,0.0,0.0);
gMatrixMode(GL_PROJECTION);
LoadIdentity();
glOrtho(0,nx,0.0,ny,-200.0,200.0);

// Generating 2D texture
glEnable(GL_TEXTURE_2D);
genTextures(1,&gl_Tex);
BindTexture(GL_TEXTURE_2D,gl_Tex);
TexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
TexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
TexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
TexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
TexImage2D(GL_TEXTURE_2D,0,GL_RGBA8,nx,ny,0,GL_RGBA,
GL_UNSIGNED_BYTE,NULL);

//Creating GPU pixel buffer
GenBuffers(1,&gl_PBO);
BindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, gl_PBO);
printf("Buffer created.\n");

//Main loop
printf("Starting GLUT main loop...\n");
displayFunc(display);
reshapeFunc(resize);
displayFunc(display);
MainLoop();

for(int i=0;i<nx;i++){
  delete grid[i];
  delete new_grid[i];
}
delete[] grid;
delete[] new_grid;
delete ruleset1;
delete ruleset2;
delete plotvar;
delete plot_rgba;
delete cmap_rgba;
The rest of the main loop is pretty straightforward. The glut functions initialize the window. I do not go over them in great detail as they were copied of OpenGL documentation files which include a “how to” part. The advantage of this library is that you always initialize the window in the same way, so all of the above can just be copied between projects without limiting intended functionality. The rest of the code is meant to release allocated memory and execute a clean exit.

void display()
{
    if (tCurr <= t)
    {
        int ip1, jp1, i0, icol, i1, i2, i3, i4, isol;
        float minVar(-3.0), maxVar(3.0), frac;

        move();

        while (mass_avrg((nx/3)-10, nx/3, 1, ny-1) < 0.3) {
            insert((nx/3)-rand()%10, (ny-1)-rand()%10); 
        }

        for (int j = 0; j < ny; j++){
            for (int i = 0; i < nx; i++){
                i0 = I2D(nx, i, j);
                frac = (plotvar[i0] - minVar)/(maxVar - minVar);
                icol = frac*n_color;
                if (MID(grid[i][j], 7, 8) == 1){isol = 0;}
                else {isol = 1;}
                plot_rgba[i0] = isol*cmap_rgba[icol];
            }
        }

        // Filling the pixel buffer with the plot_rgba array
        glBufferData(GL_PIXEL_UNPACK_BUFFER_ARB, nx*ny*sizeof(unsigned int), (void **)plot_rgba, GL_STREAM_COPY);
        // Copying the pixel buffer to the texture
        glTexImage2D(GL_TEXTURE_2D, 0, 0, nx, ny, GL_RGBA, GL_UNSIGNED_BYTE, 0);
        //Rendering
        glClear(GL_COLOR_BUFFER_BIT);
        glBegin(GL_QUADS);
        glTexCoord2f(0.0,0.0);
        glVertex3f(0.0,0.0,0.0);
        glTexCoord2f(1.0,0.0);
        glVertex3f(nx,0.0,0.0);
        glTexCoord2f(1.0,1.0);
        glVertex3f(nx,ny,0.0);
        glTexCoord2f(0.0,1.0);
        glVertex3f(0.0,ny,0.0);
        glEnd();
        glutSwapBuffers();
    }
}
t_curr++;
}
else{
    FILE *vdata;
    vdata=fopen("vdata.dat","w");
    for(int i=0;i<ny;i++)
        fprintf(vdata,"%d %f
",i,v_profile[i]/t);
    fclose(vdata);
    exit(0);
}

This is the GLUT window handle function. As you may have noticed before it is called both by the GLUT display function and GLUT idle state, which allows it to be repeated regardless of what is happening to the visualization window itself. The function calls another function – “move”, which is responsible for computing the movement and collision of particles in one time step. After that the function controls a narrow slit of the simulation area and makes sure that is has an approximate density of 30%.

//Window resize callback
void resize(int w,int h){
    width=w;
    height=h;
    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.,nx,0.,ny,-200.,200.);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

This small function allows the window to be resized by hand.

void insert(int _x,int _y){
    if(grid[_x][_y]==0){
        grid[_x][_y]=pow(2,rand()%7);
    }
    else if(grid[_x][_y]<127){
        int bity[8];
        for(int i=0;i<8;i++)
            bity[i]=MID(grid[_x][_y],i,i+1);
        int j=rand()%7;
        if(bity[j]!=1){
            bity[j]=1;
            grid[_x][_y]=0;
            for(int i=0;i<8;i++)
                grid[_x][_y]+=bity[i]*pow(2,i);
        }
    }
}

}
The function for inserting a particle with random velocity. For a given lattice position, it checks whether a particle already exists there, if not inserts a particle with random velocity. If the lattice point has particles it makes sure that the one inserted has an unused velocity or exits without inserting a particle at all.

```c
void move()
{
    int direction[8]={0,0,0,0,0,0,0,0};
    for(int i=0;i<ny;i++)
    {
        for(int j=0;j<nx;j++)
        {
            if(i%2==0)
            {
                if(j>0 && i>0)
                    direction[1]=MID(grid[j-1][i-1],3,4);
                if(i>0)
                    direction[2]=MID(grid[j][i-1],4,5);
                if(i<ny-1)
                    direction[4]=MID(grid[j][i+1],0,1);
                if(j>0 && i<ny-1)
                    direction[5]=MID(grid[j-1][i+1],1,2);
            }
            else
            {
                if(i>0)
                    direction[1]=MID(grid[j][i-1],3,4);
                if(j<nx-1 && i>0)
                    direction[2]=MID(grid[j+1][i-1],4,5);
                if(j<nx-1 && i<ny-1)
                    direction[4]=MID(grid[j+1][i+1],0,1);
                if(i<ny-1)
                    direction[5]=MID(grid[j][i+1],1,2);
            }
            if(j>0)
            {
                direction[0]=MID(grid[j-1][i],2,3);
            }
            if(j<nx-1)
            {
                direction[3]=MID(grid[j+1][i],5,6);
            }
        }
    }
    new_grid[j][i]=(direction[0]*pow(2,2))+(direction[1]*pow(2,3))+(direction[2]*pow(2,4))+(direction[3]*pow(2,5))+direction[4]+(direction[5]*2)+(direction[6]*pow(2,6))+(direction[7]*pow(2,7));
    for (int k=0;k<8;k++)
    {
        direction[k]=0;
    }
}
```
for(int i=0;i<ny;i++){
    for(int j=0;j<nx;j++){
        if(rand()%100<=50){
            grid[j][i]=ruleset1[new_grid[j][i]];
        } else{
            grid[j][i]=ruleset2[new_grid[j][i]];
        }
        new_grid[j][i]=0;
    }
}
double sr_v(0);
for(int j=0;j<ny;j++){
    sr_v=0;
    for(int i=0;i<nx;i++){
        if(i>(nx/3)+10){
            sr_v+=momentum_x(i,j);
        }
        plotvar[I2D(nx,i,j)]=momentum_x(i,j);
        //plotvar[I2D(nx,i,j)]=mass(i,j);
    }
    sr_v=sr_v/(nx-(nx/3+10));
    v_profile[j]+=sr_v;
}

This is the main FHP solver function. First it loads all velocity bits flowing into the targeted point, then computes collisions using ruleset vectors, and finally updates the visualization vector with either velocity (momentum) or mass at any given point of the lattice. It also computes output velocity data and averages it over horizontal area (from the first point without a set density to the end of the lattice).

int mass(int _x,int _y){
    int nr_p(0);
    for(int k=0;k<7;k++){
        nr_p+=MID(grid[_x][_y],k,k+1);
    }
    return nr_p;
}

double mass_avrg(int _x0,int _xmax,int _y0,int _ymax){
    double sr_mass(0.0);
    for(int i=_x0;i<_xmax;i++){
        for(int j=_y0;j<_ymax;j++){
            sr_mass+=mass(i,j);
        }
    }
    sr_mass=sr_mass/((_xmax-_x0)*(_ymax-_y0)*7);
    return sr_mass;
}

double momentum_x(int _x,int _y){
    int bity[8];
    double p_x(0);
    for(int k=0;k<8;k++){
        bity[k]=MID(grid[_x][_y],k,k+1);
    }
    if(bity[0]==1){
        p_x-=pow(1.25,0.5)/2;
These functions, called in previous sections of the code, are used for plotting the momenta, masses at a given point, or to output the average mass of a given area.

3. Results and conclusion

Here are some of the results obtained with my solver. Each set contains a best fit analytical solution to the Hagen-Poiseuille law.
As you can see, the FHP-II model is more accurate than FHP-I, as is to be expected, however when the pipe height reaches a point when it no longer can be considered narrow, FHP-I model “behaves better”. In reality this is a sign that FHP-I is less accurate, as for these heights the effects of turbulent flow should distort the data from expected values as shown in the FHP-II case.

4. Literature and sources

- Hagen-Poiseuille derivation from:
  https://en.wikipedia.org/wiki/Hagen%E2%80%93Poiseuille_equation#Liquid_flow_through_a_pipe

- Lattice Gas Automaton
  https://en.wikipedia.org/wiki/Lattice_gas_automaton

- Supplementary lecture notes on FHP models, Computation Fluid Dynamics course by Dr Matyka, Institute of Theoretical Physics, Department of Physics and Astronomy, University of Wroclaw.
  http://ift.uni.wroc.pl/~maq/zajecia/cfd2015/