CFD Lattice Boltzmann Solver Final Report

Mateusz Cierniak

1. Introduction – the concept.

I have chosen for my 2nd project to create a Lattice Boltzmann Method solver in the C++ language. Initially I worked in Visual Studio 2012 on a Windows 7 OS, but certain compatibility and cross platform issues have made me switch to a UNIX based Linux OS (Debian stable) and it's standard C++ compiler.

The solver simulates a Vortex Karman Street, which is a pattern of vortices appearing when a fluid in motion flows around a blunt obstacle. By the standard DnQm classification scheme, it is a D2Q9 Lattice Boltzmann solver, which is a 2 dimensional lattice with 9 possible directions of motion (8 + stationary). The fluid is put in motion artificially, by inserting particles on the left edge with a right oriented horizontal velocity. The obstacle is a circle located half way to the top vertically and in 1/6 of total horizontal length. Its diameter is 1/2 of simulation height. Particles entering the right edge of the simulation are being moved left, therefore creating an artificial extension to infinity. At the same time with every simulation step new particles are being added on the left edge, guaranteeing mass conservation. This is a solution I looked up recently, since on the last exercise it was pointed out to me, that periodic boundary conditions on vertical edges would cause a disturbance in the system by adding vertical velocities to the beginning. Such a problem does not exist for horizontal edges, therefore plain periodic boundary conditions remain there.

Contrary to other CFD methods, the LB method does not compute particles as individual interacting data points, but rather, computes equilibrium particle probability distribution, for a discrete set of possible velocities. It was developed based on the Lattice Gas Automata, a group of CFD methods which describe the simulated medium as a discrete lattice, of which each point can take one of a number of states (associated with a set number of particles with set velocities), and the simulation proceeds in a two-step fashion – propagation along the velocity direction of each individual particle and collision of particles converging on the same lattice point (a good example of such a solution is the FHP model we previously developed). Similarly, the LB method computes probability distribution, using a Taylor expansion of the Maxwell-Boltzmann distribution function, propagates particles according to the associated velocities, and then computes the collisions by calculating the resulting vertical and horizontal velocities, the density and determining the new equilibrium distributions for those parameters.

Besides the solving method itself, I wanted my solver to visualize the ongoing simulation in real time. To do this I used a well known to me family of C++ libraries, OpenGL, with added open source GLEW and GLUT extensions. I was hoping for a solution which would be compatible with both Windows and Linux operating systems, however I was not entirely successful with that. There were some issues regarding window handling and OS environmental variables, which are simply named differently in each system. In order not to sacrifice too much time on the issue I have decided to write my code for a Linux operating system instead. The choice of OpenGL as a visual library was made mostly because I have covered it during my 1st

year Objective Programming course, and I was able to use my old projects as a guide to implementing the library's methods in practice. The most notable contribution was the color map data for the velocity plot, which I previously used for a FDTD solver for electromagnetic wave propagation. Another reason to use this library is its popularity and abundance of guides and forum discussions, which were very helpful, as they usually cover a wide variety of concepts and possible implementations beyond what a programming course or online guide may contain.

2. Implementation

In order to compute the model, I create 9 floating point number arrays, another 9 temporary arrays for rewriting purposes, an array of obstacles, color map arrays, appropriate scalar variables, and several handler functions which I will now go over in detail.

The pre-main section of my code is as follows:

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

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

#include <crtdbg.h>*/

using namespace std;

//Method for calculating 2D positions on a 1D vector representation
#define I2D(ni,i,j) (((ni)*(j))+i)
#define D2Ix(ni,i) (i%ni)
#define D2Iy(ni,i) (i/ni)

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

//Arrays
float
*f0,*f1,*f2,*f3,*f4,*f5,*f6,*f7,*f8;

float
*tempf0,*tempf1,*tempf2,*tempf3,*tempf4,*tempf5,*tempf6,*tempf7,*tempf8
;
float *cmap,*plotvar;
int *solid;
unsigned int *cmap_rgba, *plot_rgba;

//Scalars
float tau,feq1,feq2,feq3;
float vxin,vyin,density;
float width,height;
int nx,ny;
int n_color;
```

```
int x_pos_old,y_pos_old;
int gnu_flag(0),key_press_flag(0);
int par(34);
char ppar((char)par);
string f_call;

//OpenGL functions
void display();
void resize(int,int);
void p_key_down(unsigned char,int,int);
void p_key_up(unsigned char,int,int);

//LB  functions
void stream();
void per_BC();
void solid_BC();
void in_BC();
void ex_BC();
void collide();
```

The pre-processor part inserts all the relevant libraries, standard and standard i-o are used for most generic C functions, and C console printf command, which is faster than the std cout. Math lib helps with floating point operations and functions, such as absolute value or square root, the GL libraries are essential OpenGL functions and variables. I also inserted iostream and string-stream libraries because of a problem I had with data acquisition from file. More on that later. Thanks to a hint in our lecture notes regarding expressing 2d lattices as 1d numerical vectors, I created the pre-processor macros for handling such expressions. Because of high repetition of these functions I decided to define them in the pre-processor rather than as regular code functions. Depending on operating system and processor cache management this might make a difference or it might not. Since my code uses a high number of arrays, I also insert a memory leak checking method, just in case. I use it only right after modifying the code somehow and leave it inactive otherwise. It was a point of principle during my previous programing courses to check this, as a memory leak could be a serious problem for such calculations.

```
int main(int argc, char** argv){
	f_call="gnuplot -e 'plot                                                ";
	f_call.insert(17,&ppar);
	f_call.insert(18,"vdata.dat");
	f_call.insert(27,&ppar);
	f_call.insert(29,"with vectors filled lc palette' -persist");
```

The beginning of my code was created by me at the end. The variable f_call is responsible for calling gnuplot and creating a vector plot of lattice velocities upon exiting the simulation. Because such a call requires the use of quotation marks as string characters I could not input the command directly. The command reads:

gnuplot -e 'plot "vdata.dat" with vectors filled lc palette' -persist"

The same command used on the .dat file after exiting the simulation should plot the obtained vector field.

```
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>>vxin>>vyin>>density>>tau;
        fclose(param);

        printf ("nx= %d\n",nx);
        printf ("ny= %d\n",ny);
        printf ("vxin= %f\n",vxin);
        printf ("vyin= %f\n",vyin);
        printf ("density= %f\n",density);
        printf ("tau= %f\n",tau);

        //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;
        }
        fscanf(color_map,"%d",&n_color);
        cmap_rgba=new unsigned int[n_color*sizeof(unsigned int)];
        for (int i=0;i<n_color;i++){
               fscanf(color_map,"%f%f%f",&r_color,&g_color,&b_color);
               cmap_rgba[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 of my code is responsible for reading data from files param.dat and cmap.dat, the simulation parameters and color map respectively. The color map file consist of triplets of numbers, which represent saturation of red, green and blue colors respectively. After the retrieval they are inserted into the cmap array using bitwise multiplication, resulting in an 8 bit integer with 11 at the beginning and 2 bits assigned to each color saturation. It is a standard method I have been using in my previous simulations. For some reason, I could not use the same method for the simulation parameters. The program would interpret them as 0 all the time. Because of that, I decided to load them using string stream typecasting method used above. I found it on the www.cplusplus.com site.

```
//Initializing arrays
        f0 = new float[array_size];
        f1 = new float[array_size];
        f2 = new float[array_size];
        f3 = new float[array_size];
        f4 = new float[array_size];
        f5 = new float[array_size];
        f6 = new float[array_size];
```

```
        f7 = new float[array_size];
        f8 = new float[array_size];

        tempf0 = new float[array_size];
        tempf1 = new float[array_size];
        tempf2 = new float[array_size];
        tempf3 = new float[array_size];
        tempf4 = new float[array_size];
        tempf5 = new float[array_size];
        tempf6 = new float[array_size];
        tempf7 = new float[array_size];
        tempf8 = new float[array_size];

        plotvar = new float[array_size];

        plot_rgba = new unsigned int[nx*ny*sizeof(unsigned int)];

        solid = new int[nx*ny*sizeof(int)];

        for (int i=0;i<nx*ny;i++){
                f0[i]=feq1*density*(1.f-1.5f*vxin*vxin);
                f1[i]=feq2*density*(1.f+3.f*vxin+4.5f*vxin*vxin-
1.5f*vxin*vxin);
                f2[i]=feq2*density*(1.f-1.5f*vxin*vxin);
                f3[i]=feq2*density*(1.f-3.f*vxin+4.5f*vxin*vxin-
1.5f*vxin*vxin);
                f4[i]=feq2*density*(1.f-1.5f*vxin*vxin);
                f5[i]=feq3*density*(1.f+3.f*vxin+4.5f*vxin*vxin-
1.5f*vxin*vxin);
                f6[i]=feq3*density*(1.f-3.f*vxin+4.5f*vxin*vxin-
1.5f*vxin*vxin);
                f7[i]=feq3*density*(1.f-3.f*vxin+4.5f*vxin*vxin-
1.5f*vxin*vxin);
                f8[i]=feq3*density*(1.f+3.f*vxin+4.5f*vxin*vxin-
1.5f*vxin*vxin);
                plotvar[i]=vxin;
                if(((D2Ix(nx,i)-50)*(D2Ix(nx,i)-50))/(3*3)+((D2Iy(nx,i)-
50)*(D2Iy(nx,i)-50))/(3*3)>5*5){
                        solid[i]=1;
                }
                else{
                        solid[i]=0;
                }
        }
```

This part of the code initializes all arrays and fills them with densities resulting from initial velocities (I included only horizontal component velocities, because we are interested only in horizontal flow). For each new command there exists a delete command at the end of the main function to avoid memory leaks. Also in this part I define the obstacle as every point satisfying the ellipse inequality.

```
//Initializing window
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowSize(nx,ny);
        glutInitWindowPosition(50,50);
        glutCreateWindow("Karman Vortex Street");
```

```
        // 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 ")){
                fprintf(stderr,"ERROR: Support for necessary OpenGL
extensions missing.");
                fflush(stderr);
                system("pause");
                return 2;
        }

        glClearColor(0.0,0.0,0.0,0.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0,nx,0.0,ny,-200.0,200.0);

        // Generating 2D texture
        glEnable(GL_TEXTURE_2D);
        glGenTextures(1,&gl_Tex);
        glBindTexture(GL_TEXTURE_2D,gl_Tex);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA8,nx,ny,0,GL_RGBA,
GL_UNSIGNED_BYTE,NULL);

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

        //Main loop
        printf("Starting GLUT main loop...\n");
        glutDisplayFunc(display);
        glutReshapeFunc(resize);
        glutKeyboardFunc(p_key_down);
        glutKeyboardUpFunc(p_key_up);
        glutIdleFunc(display);
        glutMainLoop();
```

The last essential part of the main function are almost essentially OpenGL functions. Most of them come from my classes or a tutorial on the official OpenGL wiki webpage (www.opengl.org/wiki/). In reality the important ones are the first 5, which are responsible for opening the visualization window, and the last 6, which link my handle functions with different window behavior. The rest either the tutorial told me they need to be there, or I learn to put there during previous classes without much thought into their detailed functionality.

```
//GLUT handle function for LB steps
void display(){

        int ip1,jp1,i0,icol,i1,i2,i3,i4,isol;
        float minvar(-0.1),maxvar(0.1),frac;
```

```
        stream();
        per_BC();
        solid_BC();
        in_BC();
        ex_BC();
        collide();

        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;
                        isol=(int)solid[i0];
                        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

        glTexSubImage2D(GL_TEXTURE_2D,0,0,0,nx,ny,GL_RGBA,GL_UNSIGNED_BY
TE,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();
}
```

This function is directly called by the GLUT main loop, as seen in the previous code section. It first calls all LB method functions and then calculates the color vector for every pixel of the GLUT window. I implemented it to have an adjustable value range, and automatically computes to which values assign what color by determining the range and number of colors available. The rest are, again, functions I have taken from various OpenGL wiki tutorials.

```
//Function for LB streaming step
void stream(){

        int im1,ip1,jm1,jp1,i0;

        for(int j=0;j<ny;j++){
                jm1=j-1;
                jp1=j+1;
                if(j==0){jm1=0;}
                if(j==(ny-1)){jp1=ny-1;}
                for(int i=1;i<nx;i++){
                        i0=I2D(nx,i,j);
                        im1=i-1;
```

```
                        ip1=i+1;
                        if(i==0){im1=0;}
                        if(i==(nx-1)){ip1=nx-1;}
                        tempf1[i0]=f1[I2D(nx,im1,j)];
                        tempf2[i0]=f2[I2D(nx,i,jm1)];
                        tempf3[i0]=f3[I2D(nx,ip1,j)];
                        tempf4[i0]=f4[I2D(nx,i,jp1)];
                        tempf5[i0]=f5[I2D(nx,im1,jm1)];
                        tempf6[i0]=f6[I2D(nx,ip1,jm1)];
                        tempf7[i0]=f7[I2D(nx,ip1,jp1)];
                        tempf8[i0]=f8[I2D(nx,im1,jp1)];
                }
        }

        for (int j=0;j<ny;j++){
                for (int i=1;i<nx;i++){
                        i0=I2D(nx,i,j);
                        f1[i0]=tempf1[i0];
                        f2[i0]=tempf2[i0];
                        f3[i0]=tempf3[i0];
                        f4[i0]=tempf4[i0];
                        f5[i0]=tempf5[i0];
                        f6[i0]=tempf6[i0];
                        f7[i0]=tempf7[i0];
                        f8[i0]=tempf8[i0];
                }
        }
}
```

This function computes the propagation step of the LB method. Similarly to my FHP solver, the code focuses on a point and writes everything that flows into it.

```
//Periodic boundry conditions for "floor" and "ceiling"
void per_BC(){

        int i0,i1;

        for(int i=0;i<nx;i++){
                i0=I2D(nx,i,0);
                i1=I2D(nx,i,ny-1);
                f2[i0]=f2[i1];
                f5[i0]=f5[i1];
                f6[i0]=f6[i1];
                f4[i1]=f4[i0];
                f7[i1]=f7[i0];
                f8[i1]=f8[i0];
        }
}

//Collision rules for obstacles
void solid_BC(){
        int i0;
        float f1old,f2old,f3old,f4old,f5old,f6old,f7old,f8old;

        for(int j=0;j<ny;j++){
                for(int i=0;i<nx;i++){
                        i0=I2D(nx,i,j);
                        if(solid[i0]==0){
                                f1old=f1[i0];
```

```
                                f2old=f2[i0];
                                f3old=f3[i0];
                                f4old=f4[i0];
                                f5old=f5[i0];
                                f6old=f6[i0];
                                f7old=f7[i0];
                                f8old=f8[i0];

                                f1[i0]=f3old;
                                f2[i0]=f4old;
                                f3[i0]=f1old;
                                f4[i0]=f2old;
                                f5[i0]=f7old;
                                f6[i0]=f8old;
                                f7[i0]=f5old;
                                f8[i0]=f6old;
                        }
                }
        }
}

//Inlet boundary condition -  all particles entering from the "left"
have a set x component velocity
void in_BC(){

        int i0;
        float f1new,f5new,f8new,vx_term;

        vx_term=1.f+3.f*vxin+3.f*vxin*vxin;
        f1new=density*feq2*vx_term;
        f5new=density*feq3*vx_term;
        f8new=f5new;

        for(int j=0;j<ny;j++){
                i0=I2D(nx,0,j);
                f1[i0]=f1new;
                f5[i0]=f5new;
                f8[i0]=f8new;
        }
}

//Exit boundry condition - all f pointing towards the right edge are
moved left
void ex_BC(){

        int i0, i1;

        for(int j=0;j<ny;j++){
                i0=I2D(nx,nx-1,j);
                i1=i0-1;
                f3[i0]=f3[i1];
                f6[i0]=f6[i1];
                f7[i0]=f7[i1];
        }
}
```

These are the boundary condition functions. The reason why I did this the way presented is because I planned to insert a handle function, to swap between different boundary conditions

on demand. I have not implemented this into my solution. In any case, the conditions are as follows: first the periodic boundary for horizontal edges, whatever flows above the top ends at the bottom and vice versa. Second are the collision rules for obstacles. Every particle is turned back the way it came from. Third is the artificial flow force, this function writes a line of particles with only horizontal velocity at the left edge, guaranteeing that all of them will move right until they hit the obstacle. The last one, as mentioned earlier, rewrites the second last line of particles on the right edge one line to the left. This method was suggested by A.J. Wagner in his publication "A practical introduction to the Lattice Boltzmann Method" (http://physics.ndsu.edu/fileadmin/physics.ndsu.edu/Wagner/LBbook.pdf).

```
//Function for LB collision step
void collide(){

        int i0;
        float ro,rovx,rovy,vx,vy,v_sq_term,dvx,dvy;
        float lf0eq,lf1eq,lf2eq,lf3eq,lf4eq,lf5eq,lf6eq,lf7eq,lf8eq;
        float rtau(1.f/tau),rtau1;

        FILE *vdata;

        rtau1=1.f-rtau;

        vdata=fopen("vdata.dat","w");
        for(int j=0;j<ny;j++){
                for (int i=0;i<nx;i++){

                        i0=I2D(nx,i,j);


        ro=f0[i0]+f1[i0]+f2[i0]+f3[i0]+f4[i0]+f5[i0]+f6[i0]+f7[i0]+f8[i0
];
                        rovx=f1[i0]-f3[i0]+f5[i0]-f6[i0]-f7[i0]+f8[i0];
                        rovy=f2[i0]-f4[i0]+f5[i0]+f6[i0]-f7[i0]-f8[i0];
                        vx=rovx/ro;
                        vy=rovy/ro;
                        //plotvar[i0]=sqrt(vx*vx+vy*vy);
                        plotvar[i0]=vx;

                        fprintf(vdata,"%d %d %f %f
%f\n",i,j,vx/0.05,vy/0.05,vx/0.1);

                        v_sq_term=1.5f*(vx*vx+vy*vy);

                        lf0eq=ro*feq1*(1.f-v_sq_term);
                        lf1eq=ro*feq2*(1.f+3.f*vx+4.5f*vx*vx-v_sq_term);
                        lf2eq=ro*feq2*(1.f+3.f*vy+4.5f*vy*vy-v_sq_term);
                        lf3eq=ro*feq2*(1.f-3.f*vx+4.5f*vx*vx-v_sq_term);
                        lf4eq=ro*feq2*(1.f-3.f*vy+4.5f*vy*vy-v_sq_term);

        lf5eq=ro*feq3*(1.f+3.f*(vx+vy)+4.5f*(vx+vy)*(vx+vy)-v_sq_term);
                        lf6eq=ro*feq3*(1.f+3.f*(-vx+vy)+4.5f*(-vx+vy)*(-
vx+vy)-v_sq_term);
                        lf7eq=ro*feq3*(1.f+3.f*(-vx-vy)+4.5f*(-vx-vy)*(-
vx-vy)-v_sq_term);
                        lf8eq=ro*feq3*(1.f+3.f*(vx-vy)+4.5f*(vx-vy)*(vx-
vy)-v_sq_term);
```

```
                        f0[i0]=rtau1*f0[i0]+rtau*lf0eq;
                        f1[i0]=rtau1*f1[i0]+rtau*lf1eq;
                        f2[i0]=rtau1*f2[i0]+rtau*lf2eq;
                        f3[i0]=rtau1*f3[i0]+rtau*lf3eq;
                        f4[i0]=rtau1*f4[i0]+rtau*lf4eq;
                        f5[i0]=rtau1*f5[i0]+rtau*lf5eq;
                        f6[i0]=rtau1*f6[i0]+rtau*lf6eq;
                        f7[i0]=rtau1*f7[i0]+rtau*lf7eq;
                        f8[i0]=rtau1*f8[i0]+rtau*lf8eq;
                }
        }
        fclose(vdata);
        if(gnu_flag==1){
                system("import -window root capture.png");
                //system("xwd -root | convert - capture.png");
alternative screenshot method
                system(f_call.c_str());
                gnu_flag=0;
                exit(0);
        }
}
```

This is the last LB method function, collision computing. The function takes all the probabilities of a point (after propagation), determines the overall density and velocity components. Then, using those components, calculates new equilibrium distributions. Because this is the last step in the main loop, this function also updates the velocity matrix for visualization and writes the vector field to a file. If the p key is pressed it calls gnuplot using the previously defined string variable and ends the simulation. I initially wanted the key press to only plot a vector field without ending the simulation, but there were issues regarding file access and the fact, that replotting this field could result in a write error or plotting of an unfinished dataset. Also, upon exiting, the program will take a screenshot of the visualization.

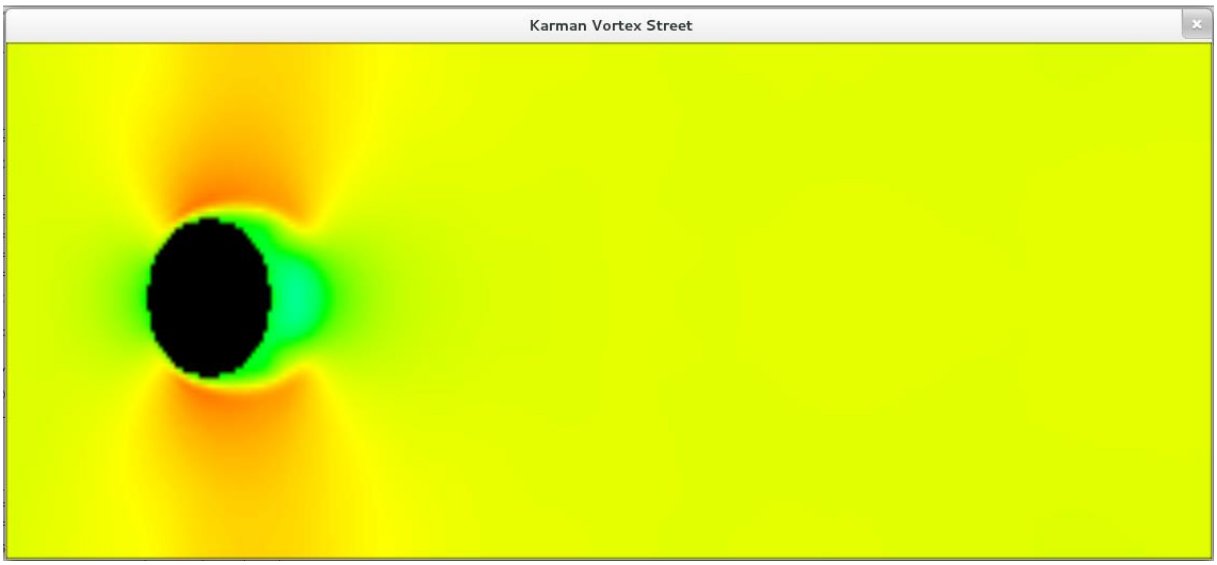
```
//Window resize callback
void resize(int w,int h){
        width=w;
        height=h;
        glViewport(0,0,w,h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.,nx,0.,ny,-200.,200.);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
}

void p_key_down(unsigned char key,int x,int y){
        if(key=='p'){
                if(key_press_flag==0){
                        gnu_flag=1;
                }
                key_press_flag=1;
        }
}

void p_key_up(unsigned char key,int x,int y){
        if(key=='p'){
```

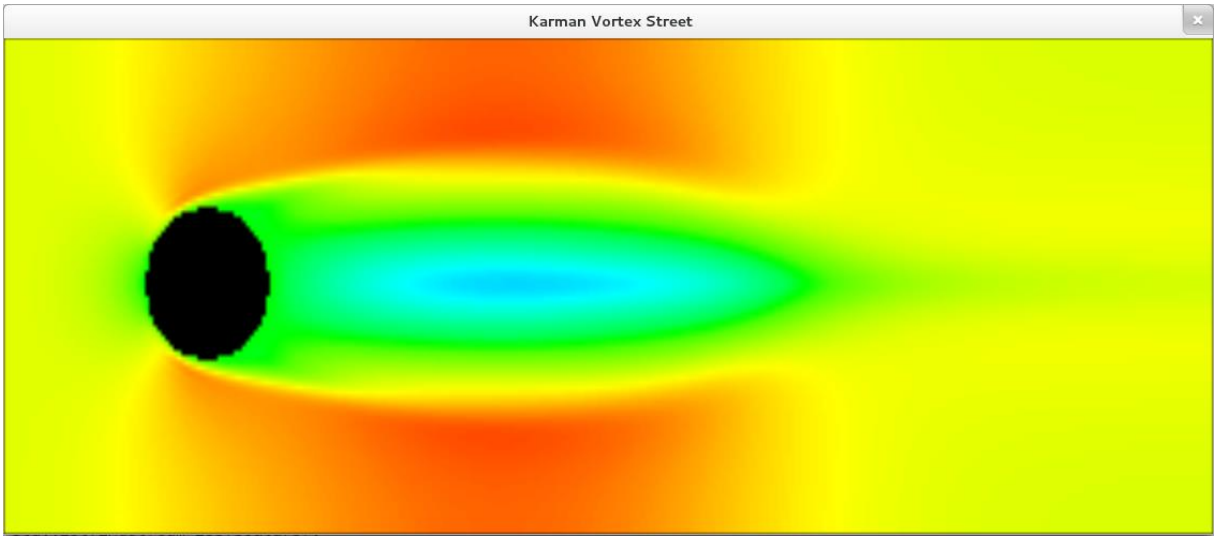
```
                key_press_flag=0;
        }
}
```

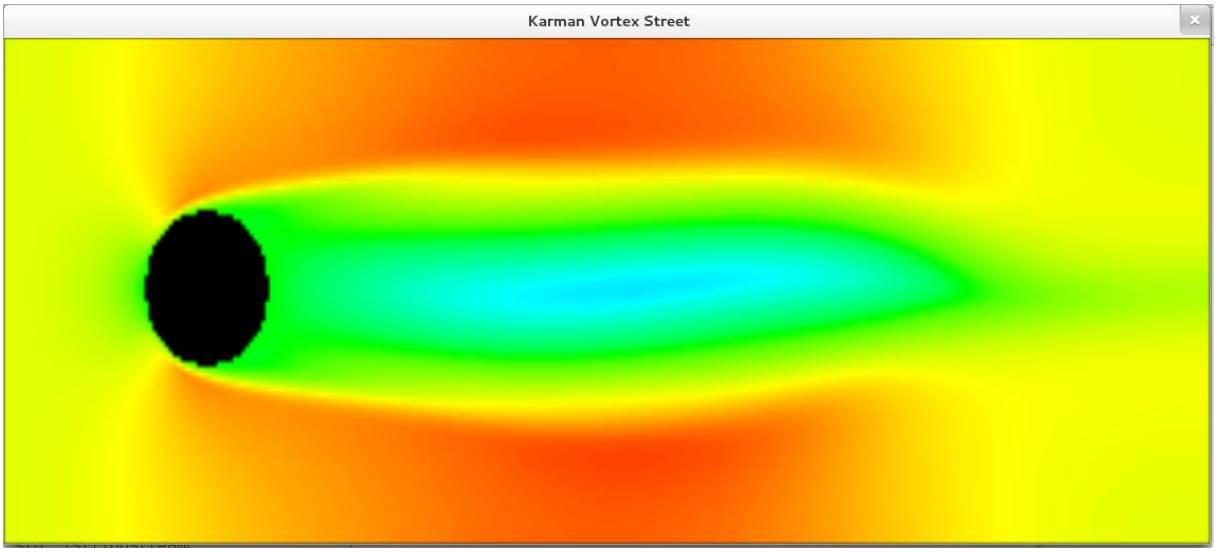
The last functions are purely OpenGL handles, used to allow the window to be resized for screenshot purposes and the key press handlers, for vector field plotting.

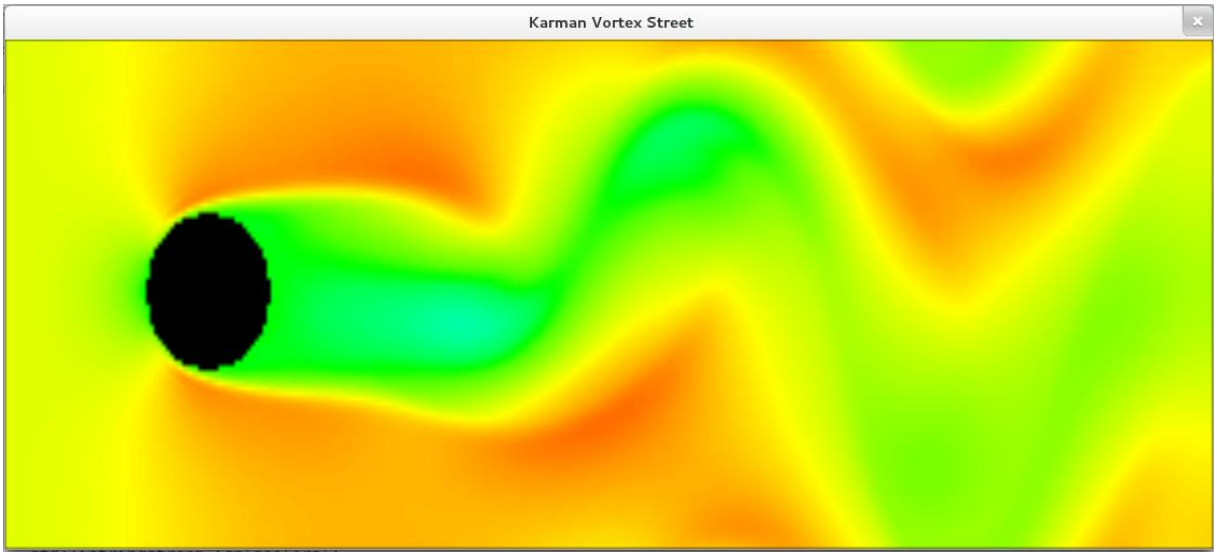
3. Results and conclusion

Here are some visualization screenshots from different points of the simulation. We can clearly see the Vector Street forming over time which allows me to conclude that the solver works correctly. A surprising observation was that no numerical artifacts could be observed in the simulation and that it was incredibly efficient in its use of processing power, clearly proving that LB is a method with a bright future in CFD. On top of those screenshots, I also include related vector field data. Because of their size I do not include graphics in the report. The plots can be obtained using the gnuplot formulae mentioned above.









4. Literature and sources

- Yuanxun Bill Bao, Justin Meskas “Lattice Boltzmann Method for Fluid Simulations”

http://www.cims.nyu.edu/~billbao/report930.pdf

- Alexander J. Wagner “A Practical Introduction to the Lattice Boltzmann Method”

http://physics.ndsu.edu/fileadmin/physics.ndsu.edu/Wagner/LBbook.pdf

- OpenGL documentation

https://www.opengl.org/

- Programmers section of stackexchange forums, opengl tag.

http://programmers.stackexchange.com/