Fluid Simulation Project Report

Author: Guangyu Hu

StudentId: 21014490

Abstract

This article shows the implementation of a functional extension to Professor Martin's Fluid Simulation project at Massey University, which includes fluid internal obstacles and external boundary effects. The internal obstacles include circular obstacles and obstacles in the shape of wing cross-sections. The project is based on the Android x86 platform and uses the Java, C++ and Halide languages to simulate fluids. The simulations are based on the Navier-Stokes equations.
This article starts with a general framework and flowchart of the project, followed by a theoretical algorithm based on internal obstacles and boundaries. Lastly, it shows the implementation and data comparison and analysed the phenomena and techniques associated with fluid simulation.

1. Introduction

Water running between riverbanks, smoke curling from a cigarette, steam streaming from a teapot, water vapour rising into clouds, and paint being mixed in a can are just a few examples of the fluids that surround us. The movement of fluids lies at the heart of all of them. People want to realistically represent each of these occurrences in interactive graphics apps.

1.1 Equations and Method

Fluid simulation is mainly based on the basic equations of fluid mechanics
the Naiver-Stokes equations:

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

$$\nabla \cdot \mathbf{u} = 0$$

The N-S equations describe the motion and mechanics of fluids, and are one of the fundamental equations in fluid mechanics. Different numerical simulation methods, such as Finite Difference Method (FDM) and Finite Volume Method (FVM) discretize the Navier-Stokes equations, converting the continuous partial differential equations into discrete difference equations, and then use iterative numerical methods to solve for the numerical solution of the flow field.
This project is based on the implementation described in “Real-Time Fluid Dynamics for Games” which is one of the FDM. As with any algorithm, the solution of the Navier-Stokes equations is divided into simple steps: diffuse(), advect() and project().
The code below is a part from Jos Stam’s paper[1].

```
void vel_step ( int N, float * u, float * v, float * u0, float * v0,
          float visc, float dt )
{
    add_source ( N, u, u0, dt ); add_source ( N, v, v0, dt );
    SWAP ( u0, u ); diffuse ( N, 1, u, u0, visc, dt );
    SWAP ( v0, v ); diffuse ( N, 2, v, v0, visc, dt );
    project ( N, u, v, u0, v0 );
    SWAP ( u0, u ); SWAP ( v0, v );
    advect ( N, 1, u, u0, u0, v0, dt ); advect ( N, 2, v, v0, u0, v0, dt );
    project ( N, u, v, u0, v0 );
}
```

Fig1. vel_step code example

In the diffuse method, the authors use the Gauss-Seidel relaxation method to keep the density values stable at large diffusion rates, without negative values. In advect, backwards rather than forwards grid cell values are used for linear interpolation. The project procedure makes sure that the velocity is mass-conserving. This is a crucial characteristic of actual fluids that needs to be upheld. It pushes the flow to have several vortices, which creates realistic-looking swirly flows. It will appear if a specific number of parameters can be met. The Von Karmen vortex is precisely the simulation of real nature after the project has been correctly implemented.

In the early 1990s, researchers developed a completely different approach to CFD, starting from physics at the molecular scale: Lattice Boltzmann Method (LBM). And it is another a good second order accurate solution to the weakly compressible (or even incompressible) N-S equation. It is essentially a Lagrangian perspective-based simulation method, and in a form somewhat similar to the Eulerian perspective lattice. It combines microscopic particle dynamics with macroscopic fluid laws, while providing good accuracy. Here is the Lattice Boltzmann fluid simulate equation:

$$f_i(x, t + \Delta t) = f_i(x - c_i \Delta t, t) - \frac{\Delta t}{\tau f}(f_i(x - c_i \Delta t, t) - f_i^{eq}(x - c_i \Delta t, t))$$

The core of LBM simulation is the solution of the velocity and pressure at various locations on the grid, and the numerical solution of the flow field is obtained by iterative calculations to simulate the motion of the fluid. This method is relatively common and can be used to simulate fluid problems in two or three dimensions, but is generally computationally intensive and not suitable for real-time application scenarios. So here we have not used LBM as the preferred option.

1.2 Halide Language

Another key to this project was the optimisation of the code using the Halide language[10]. Halide is a DSL (Domain Specific Language) for high performance image and signal processing. It was developed by researchers at MIT's Computer Science and Artificial Intelligence Laboratory to change the status quo in image processing programming. The Schedule section specifies when and where the algorithm will perform its calculations. That is outlines the pipeline mapping for a certain processing architecture. Race conditions cannot exist since function evaluations may always be

calculated in parallel. To support programming for mobile, desktop, and GPU devices, use the same method but a different schedule. Any intermediate buffers' sizes may always be determined by Halide, which will then iterate over them automatically.
The algorithm also specifies the inputs, outputs, and execution sequence of the pipeline. In this project we use the vectorize(), parallel(), compute_root(), compute_at(), compute_with(), tile() and other methods provided in Halide schedules to parallelize vel_step(), den_step(), fillBitmap(), which greatly improves the running efficiency.

2. Background

2.1 Boundary Conditions

The local project focuses on increasing the handling of internal and external boundaries in fluid simulations, currently more implemented is the handling of external boundaries, where u, v on the boundary are processed by taking the negative value of the adjacent cell, and div, p are set to the value of the adjacent grid.

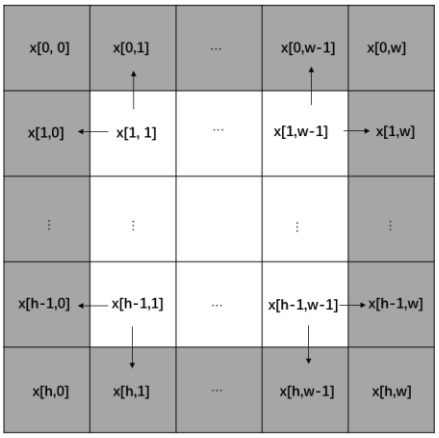


Fig2. Boundary value

In the FLUID SIMULATION Course Notes written by Robert Bridson and Matthias Muller-Fischer[3]: A solid wall boundary is where the fluid is in contact with, the fluid better not be flowing into the solid or out of it, thus the normal component of velocity has to be zero:

$$\vec{u} \cdot \hat{n} = 0$$

if the solid isn't already moving. The normal component of the solid's velocity must generally equal the normal component of the fluid's velocity:

$$\vec{u} \cdot \hat{n} = \vec{u}_{\text{solid}} \cdot \hat{n}$$

Some of the other more popular fluid simulations are based on the LBM algorithm. Dan Schroeder[5] used the Lattice Boltzmann algorithm in his experiments to simulate the effect of internal boundaries on circles and other shapes:

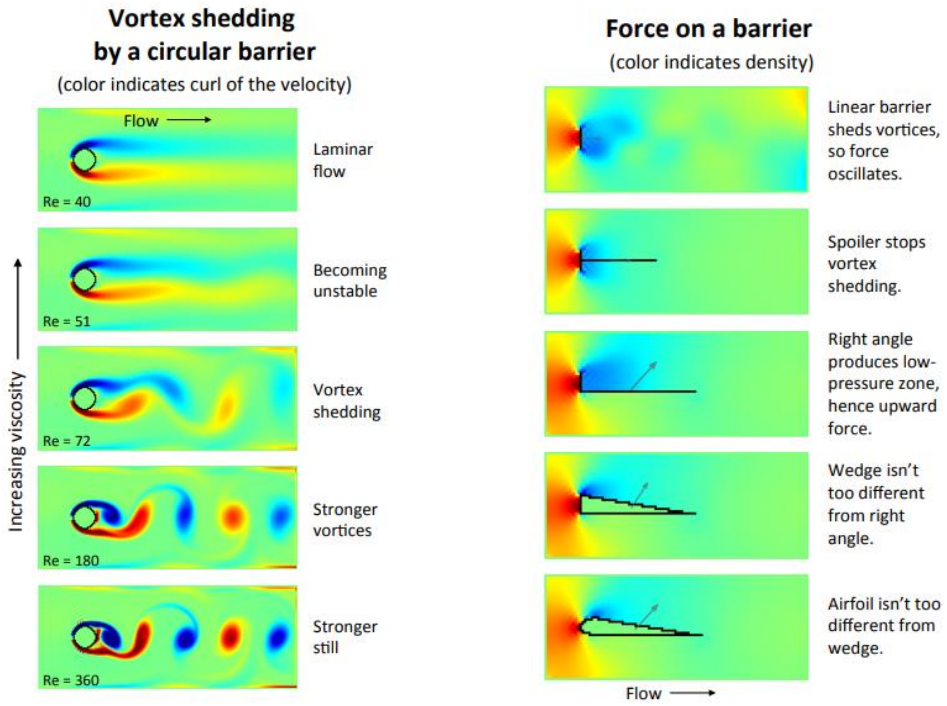


Fig3. LBM fluid Simulation

In Jos Stam's [2], boundaries are divided into five types:

1. SLIP 2. NOSLIP 3. INFLOW 4. OUTFLOW 5. PERIODIC

and NOSLIP is considered the type of boundary that is the closest to the natural fluid. We therefore use the NOSLIP type for all analyses here.

In this project we first added similar external boundaries and also added internal boundaries, our project started by implementing the fluid simulation as well as the internal and external boundaries using java and C++ and rewriting the code in the Halide language. The rewritten methods were then called in the project using android jni.

2.2 Halide generator

Halide Generator is a component of the Halide compiler. It can generate efficient code for multiple platforms, including x86, ARM, CUDA, and OpenCL. My work is based on a project by Professor Martin, who used Halide to rewrite advect(), diffuse(), project(). He used Halide schedules to optimise the runtime efficiency and to implement three generators to generate target files with different platform parameters, and then package and link them to generate a usable so file for the android project to call. The three generator functions:

HALIDE_REGISTER_GENERATOR (DensStepGenerator, halide_dens_step);
HALIDE_REGISTER_GENERATOR (VelStepGenerator, halide_vel_step);
HALIDE_REGISTER_GENERATOR (BitmapGenerator, halide_bitmap);

Shown below is a diagram of the file generated by the

. /halide_generator -g halide_vel_step -o **x86_64 target=x86-64-android** -e o,h command to generate the file.

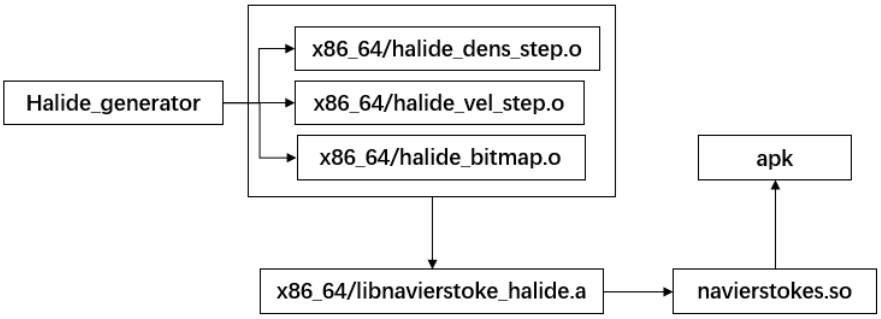


Fig4. Halide library generate process

This generator can produce object modules for any supported architecture and OS by running it with different parameters:
./halide_generator -g halide_bitmap -o **arm64-v8a target=arm-64-android** -e o,h
./halide_generator -g halide_vel_step -o **x86_64_gl target=x86-64-android-openglcompute** -e o,h

3. Methodology

Incorporating arbitrary boundaries requires applying the boundary conditions at arbitrary locations. This means that at each cell, we must determine in which direction the boundaries lie in order to compute the correct boundary values. This simulation requires more decisions to be made at each cell, leading to a slower and more complicated simulation. However, many interesting effects can be created this way, such as smoke flowing around obstacles.

3.1 Obstacle and Velocity

A simple way of implementing internal boundaries is to allocate a Boolean grid which indicates which cells are occupied by an object or not, so usually we need to use a two-dimensional array obstacle[i][j] to represent it. Here we use 1 for being in the obstacle and 0 for being outside the obstacle.

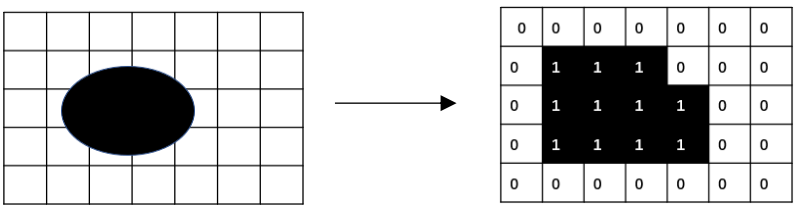


Fig5. internal boundary arrays

The boundary conditions include Dirichlet, Neumann, and mixed (coupled value and derivative), see equation below:

$$a\beta + b\frac{\partial \beta}{\partial \mathbf{n}} = c$$

Where βstands for velocity, div, or pressure etc., a, b and c are the coefficients.
For the pressure P, div, we can first discretize the boundary to facilitate processing for different boundary cases. This is shown in the following figure, The graphics may differ from the real implementation, but it can be confirmed that the type of boundary we need to analyse has not been changed.

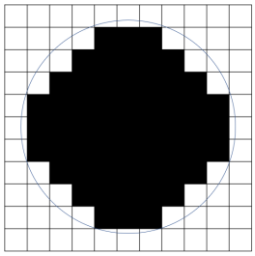


Fig6. Internal boundary for circle

For boundary points, Because the calculation is only for 4 points (top, bottom, left and right). We have divided all possible boundary patterns into 2 main categories of 8 types, the first being the case where only one adjacent point is outside the boundary, the second being the case where two points are outside the boundary, and the case where the upper right and lower right or the upper left and lower left are also included. The case where three neighbouring points are outside the boundary because the thickness required by the algorithm is not reached, so it is not in the scope of discussion. The centre grid x(i,j) will take different values depending on the location of different adjacent cell:

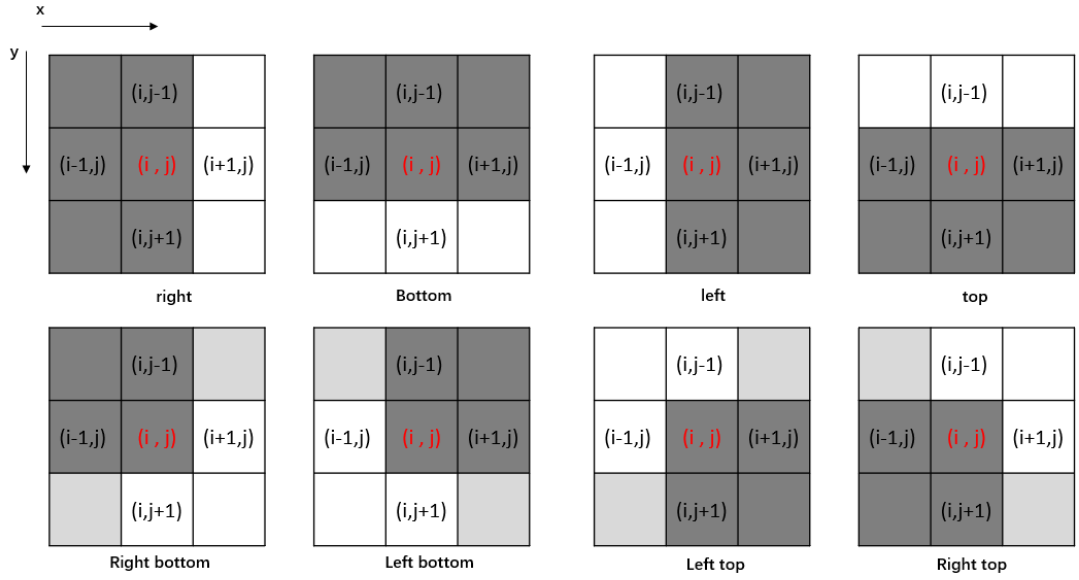


Fig7. Obstacle grid types

Or we can extend this classification. For example, by adding 5 constants a_x, which are added to the product of the obstacle arrays, then the different boundary types can be distinguished by the value of Type:

$$Type(i,j) = obstacle(i,j) * a1 + obstacle(i-1,j) * a2 + obstacle(i+1,j) * a3 + obstacle(i,j-1) * a4 + obstacle(i,j+1) * a5$$

The pure Neumann boundary condition $\partial\beta/\partial\mathbf{n} = 0$ is used to calculate the pressure values and the div. The values on the border are made to equal the values of its neighbouring nodes. For the border node (i +1, j), for instance, we change its node type to East and then set $p(i+1,j) = p(i+2,j)$ in accordance with $\partial p/\partial \boldsymbol{n} = 0$. By analogy, the remaining nodes can be determined. We set the tangent component of velocity on the boundary to the negative value of the velocity of its neighbouring fluid nodes for velocity on the non-slip boundary, which indicates that the barrier pulls the fluid at the surface. We set the value for the obstacle's internal pressure or any other value to zero;
All these calculations are performed in set_inter_bnd method which is called by the set_bnd method, the following diagram shows where it is called in the project:

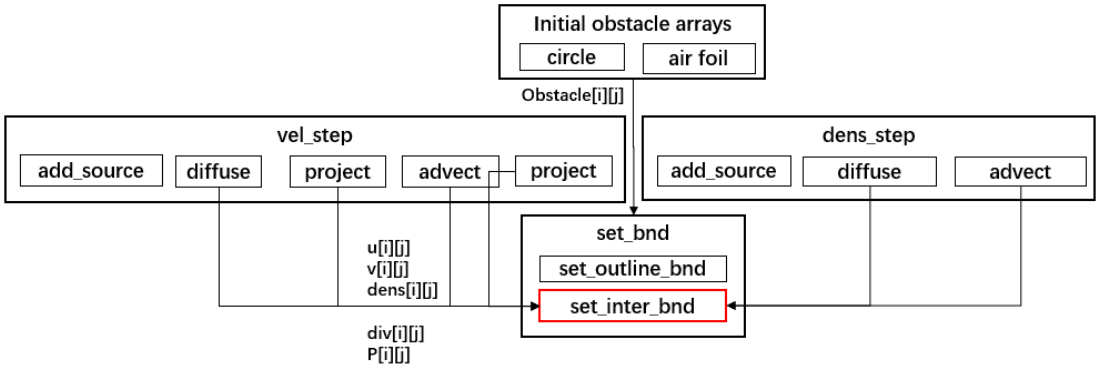


Fig8. set_inter_bnd calls

3.2 Density of obstacle

For the density on the inner boundary, we obtain the value by calculating the density of the four adjacent cells that belong to the external cell and averaging them, as shown in the figure below:

0.0	0.0	0.5	0.1	0.2	0.1	0.3
0.4	0.2	0.5	0.2	0.3	0.2	0.5
0.6	0.6	0.0	0.0	0.4	0.5	0.6
0.5	0.3	0.2	0.1	0.5	0.8	0.7
0.5	0.1	0.2	0.1	0.2	0.7	0.9

Fig9. obstacle for density

For a point on the boundary (obstacle(i,j)=1) can be seen simply as the following equation:

$$dens(i,j) = (dens(i-1,j) + dens(i+1,j) + dens(i,j+1) + dens(i,j-1))/4$$

In particular, if all 4 adjacent points of an obstacle boundary are also obstacles, this point belongs to the inner cell of the obstacle and we set its density to 0. If expressed in the formula:

$$\begin{pmatrix} obstacle(i-1,j) + obstacle(i+1,j) + \\ obstacle(i,j+1) + obstacle(i,j-1) \end{pmatrix} = 4$$
$$dens(i,j) = 0$$

4. Implementation

Professor Martin has implemented the fluid simulation using java c++ and Halide respectively, and we have also implemented the obstacles in the fluid simulation using each of these three languages. It is not usually necessary to list all three sets of code for the same algorithm, except when a comparison with the Halide code is required. We begin this section by giving the overall project stack, followed by a detailed explanation of the obstacle simulation code, including the initialisation of the obstacle, the calculation of velocity, density, pressure and div at the boundary points, and a comparison of the implementation of the Halide code with C++. This is followed by a list of problems encountered during the writing of the code and their solutions. Finally, a comparison of the data is given.

4.1 Fluid Simulation Project Stack

The following is a pile diagram of Professor Martin's project. The view is implemented through the four interfaces provided by simulation to call java code methods and Native code methods, and in the jni simulation the Halide method is called by judging the halide parameter.

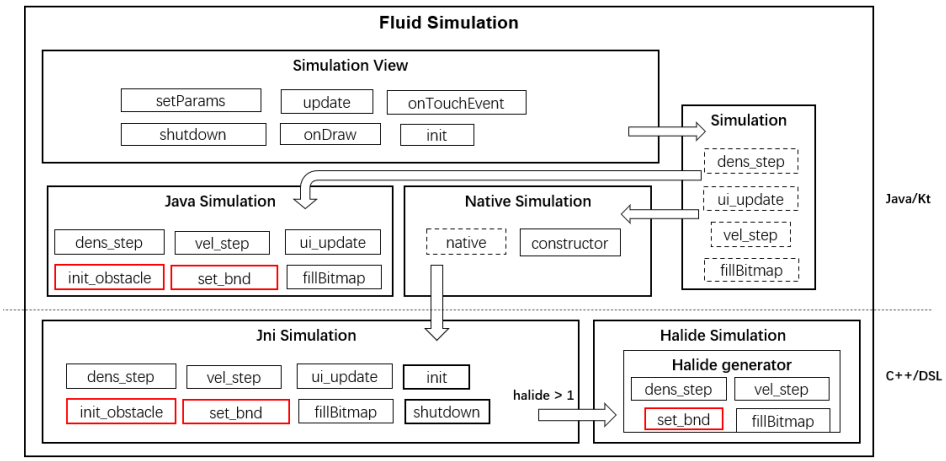


Fig10. Fluid Simulation Stack

4.2 Code Implement

4.2.1 Obstacle

The first step in simulating a fluid through an obstacle is to initialize the obstacle array, a process that takes place in the init_obstacle method. The new simulation adds two obstacles: a circle and an approximation of a wing cross-section. The wing cross-section is approximated by a rotated ellipse. The size of the obstacles is set in proportion to the aspect ratio of the mobile phone interface.

The initialization process takes place in the init method, which ensures that it is not empty in use, with the following code:

```
void init_obstacle(int NW, int NH,int type) {
    int i, j;
    switch (type) {
        case OBSTACLE_NONE:
            break;
        case OBSTACLE_CIRCLE: {
            float r = 0.065f * width;
            FOR_EACH_CELL
                    obtacle[IX(i, j)] = OUTSIDE_BOUNDARY;
                    float dy = (float) ((i + 0.5) - centerY);
                    float dx = (float) ((j + 0.5) - centerX);

                    if (dx * dx + dy * dy < r * r) {
                        obtacle[IX(i, j)] = INSIDE_BOUNDARY;
                    }
            END_FOR
            break;
        }
        case OBSTACLE_AIR_FOIL: {
            float offset = 0.05f * width;
            double a = 0.1 * height;  // long axis
            double b = 0.1 * width;   // short axis
            double x = centerX + offset;
            double y = centerY;
            float rotation = 2.3f;  //2 -> 90 / 4-> 45
            FOR_EACH_CELL
                    obtacle[IX(i, j)] = OUTSIDE_BOUNDARY;
                    // oval
                    double tempX = (j - x) * cos(M_PI / rotation) - (i - y) * sin(M_PI / rotation);
                    double tempY = (j - x) * sin(M_PI / rotation) + (i - y) * cos(M_PI / rotation);
                    double value = pow(tempX, 2) / pow(a, 2) +
                            pow(tempY, 2) / pow(b, 2);
                    if (value < 1 && (j < x)) {
                        obtacle[IX(i, j)] = INSIDE_BOUNDARY;
                    }
            END_FOR
            break;
        }
```

In the Halide code, we do not duplicate the initialization of the barrier, but pass it into the generator as an int-type 2-dimensional Buffer, which simplifies the code and keeps the data accurate, the code as below:

```
class VelStepGenerator : public Halide::Generator<VelStepGenerator> {
public:
    Input <Buffer<float>> u{ "u", 2 };
    Input <Buffer<float>> v{ "v", 2 };
    Input <Buffer<float>> u0{ "u0", 2 };
    Input <Buffer<float>> v0{ "v0", 2 };
    Input <Buffer<int>> obstacle{ "obstacle", 2 };
    Input<float> visc{ "visc" };
    Input<float> dt{ "dt" };
    Output<Buffer<float>> outputu{ "outputu", 2 };
    Output<Buffer<float>> outputv{ "outputv", 2 };
```

In addition, in the SettingsActivity where the simulation parameters are configured, we have added the option Obstacle Preference:

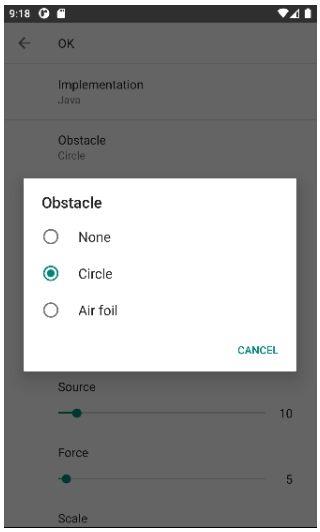


Fig11. Obstacle Preference

4.2.2 Set boundary

Inside boundary

To differentiate the fluid parameters density, velocity, div, p in set_bnd method processing. The internal boundary treatment is divided into four cases:

- TYPE_DENSITY
- TYPE_VELOCITY
- TYPE_DIV
- TYPE_P

For TYPE_DENSITY we set the velocity of obstacle boundary cell of its neighbouring fluid nodes, for TYPE_VELOCITY , we set the tangent component of velocity on the boundary to the negative value of velocity of its neighbor fluid nodes, for TYPE_DIV, TYPE_P we take a value of the velocity of its neighbouring fluid nodes according to the classification in Fig7.
In the Halide code we can see that the range is determined by bounds inference and iterate over them automatically. In the generate part, we calculate the mean of the neighbouring nodes by using the Expr of the boundaryAvg and use select instead of the if statement. At last, we use a Func to represent the final output. In the schedules section, because the calculation involves 3*3 grids, so we make the tiles 3,3 and parallelise the calculation in the y-direction . This is where Halide's strengths come into play. The specific code is as follows:

```
FuncRef set_insideBoundary(Func obtacle, Func xx, int paramType = 0) {
    //inside boundaries
    Func insideBoundaryFunc("insideBoundaryFunc");
    Expr middle = obtacle(x, y) == OUTSIDE_BOUNDARY;
    Expr left = obtacle(x - 1, y) == OUTSIDE_BOUNDARY;
    Expr right = obtacle(x + 1, y) == OUTSIDE_BOUNDARY;
    Expr top = obtacle(x, y - 1) == OUTSIDE_BOUNDARY;
    Expr bottom = obtacle(x, y + 1) == OUTSIDE_BOUNDARY;

    Expr left_int = cast<int>(left);
    Expr right_int = cast<int>(right);
    Expr top_int = cast<int>(top);
    Expr bottom_int = cast<int>(bottom);

    Expr neighborCount = left_int + right_int + top_int + bottom_int;

    Expr boundaryAvg = (select(left, xx(x - 1, y), 0.0f)
        + select(right, xx(x + 1, y), 0.0f)
        + select(top, xx(x, y - 1), 0.0f)
        + select(bottom, xx(x, y + 1), 0.0f))/ select(neighborCount > 0,
        cast<float>(neighborCount), 1.0f);

    insideBoundaryFunc(x, y) = select(middle, xx(x,y),
        select(paramType == TYPE_VELOCITY, -boundaryAvg,
        select(paramType == TYPE_DENSITY,boundaryAvg,
        getValueFromAdjcent(left,right,top,bottom,xx),xx(x,y))));

    if (!auto_sch) {
        if (gpu) {
            good_schedule({ insideBoundaryFunc });
        }
        else {
            insideBoundaryFunc.tile(x, y, xi, yi, 3, 3);
            insideBoundaryFunc.compute_root().parallel(y).vectorize(xi, 3);
        }
    }
    return insideBoundaryFunc(x, y);
}
```

As you can see, thanks to Halide's bounds inference and scheduling, image processing can be automatically parallelised and iterated over, which is very convenient and the Halide code is also much cleaner than the C++ code.

External boundary

For the external boundaries, we refer to the code in Jos Stam’s paper and rewrite it with Halide. Since the iteration is limited to 1 dimension, we use RDom (Reduction Domain) to calculate the reduction domain, ensuring that the upper and lower bounds and the left and right bounds are processed correctly. Because a single row of data is being processed, the Func use comput_root() which are not processed in parallel. Note that setting the outer boundary condition will have some effect on the source which has an offset in the y-axis.

```
FuncRef set_bnd(int b, Expr width, Expr height, Func xx, Func obtacle) {
    Func fin = BoundaryConditions::constant_exterior(obtacle, 0, 1, width, 1, height);
    Func fin1 = BoundaryConditions::constant_exterior(xx, 0, 1, width, 1, height);
    //outline boundaries
    RDom lr_boundary(1, width);
    fin1(lr_boundary,0) = select(b == 1, -fin1(lr_boundary,1), fin1(lr_boundary,1));
    fin1(lr_boundary,height + 1) = select(b == 1, -fin1(lr_boundary,height), fin1(lr_boundary,height));

    RDom tb_boundary(1, height);
    fin1(0,tb_boundary) = select(b == 2, -fin1(1,tb_boundary), fin1(1,tb_boundary));
    fin1(width + 1,tb_boundary) = select(b == 2, -fin1(width,tb_boundary), fin1(width,tb_boundary));

    ////// Corner boundaries
    fin1(0, 0) = 0.5f * (fin1(1, 0) + fin1(0, 1));
    fin1(0, width + 1) = 0.5f * (fin1(1, width + 1) + fin1(0, width));
    fin1(height + 1, 0) = 0.5f * (fin1(height, 0) + fin1(height + 1, 1));
    fin1(height + 1, width + 1) = 0.5f * (fin1(height, width + 1) + fin1(height + 1, width));

    if (!auto_sch) {
        if (gpu) {
            good_schedule({ fin1 });
        }
        else {
            fin1.compute_root();
        }
    }
    return  set_insideBoundary(fin, fin1);
}
```

Overall adaptation

In C++, we only need to add the corresponding processing code to the set_bnd method, whereas in the Halide code we need to add a Func that handles bounds for each Func output that involves bounds processing, The following code is only one modification in the project method, the other parts are covered in the advect and diffuse methods:

```
void project(Func u, Func v, Func uu, Func vv, Func obtacle,Expr w, Expr h) {
    Func div("div"), p{ "p" };
    Func boundaried_div{ "boundaried_div" };
    Func boundaried_p{ "boundaried_p" };
    Func boundaried_pre_v{ "boundaried_pre_v" };
    Func boundaried_pre_u{ "boundaried_pre_v" };

    Expr m = -1.0f / h;

    div(x, y) = m * (v(x + 1, y) - v(x - 1, y) + u(x, y + 1) - u(x, y - 1));
    boundaried_div(x, y) = set_bnd(0, w, h, div, obtacle);

    p(x, y) = convolve(boundaried_div, w, h);
    boundaried_p(x, y) = set_bnd(0, w, h, p, obtacle);

    boundaried_pre_v(x,y) = v(x, y) - 0.5f * w * (boundaried_p(x + 1, y) - boundaried_p(x - 1, y));
    vv(x, y) = set_bnd(1, w, h, boundaried_pre_v, obtacle);

    boundaried_pre_u(x, y) = u(x, y) - 0.5f * h * (boundaried_p(x, y + 1) - boundaried_p(x, y - 1));
    uu(x, y) = set_bnd(2, w, h, boundaried_pre_u, obtacle);

    good_schedule({ boundaried_div });
}
```

Extensions

For demonstration purposes, I have modified the original point source to a line source and added the ability to change the position of internal obstacles which is make it easier for simulating different situations. Also as mentioned above we have added a preference for selecting obstacle graphics.
See more code at: https://github.com/EricHuGuangyu/FluidSimulation_obstacle

5. Interactive exploration

Simulations are done on Android Emulator on window due to hardware limitations.
Hardware: AMD Ryzen 4750U 4.5G Hz
The simulation shows that at the bottom of the circular obstacle the fluid is split and at the top part the fluids converge and there are vortices in the wake. Also under the wing section shaped obstacle, there are vortices forming in the wake.

5.1 Numerical experiments

We have experimented with java, c++ and Halide codes to see how they behave under a circular. As you can see from the diagram below, the Halide code has a more detailed and smoother view.

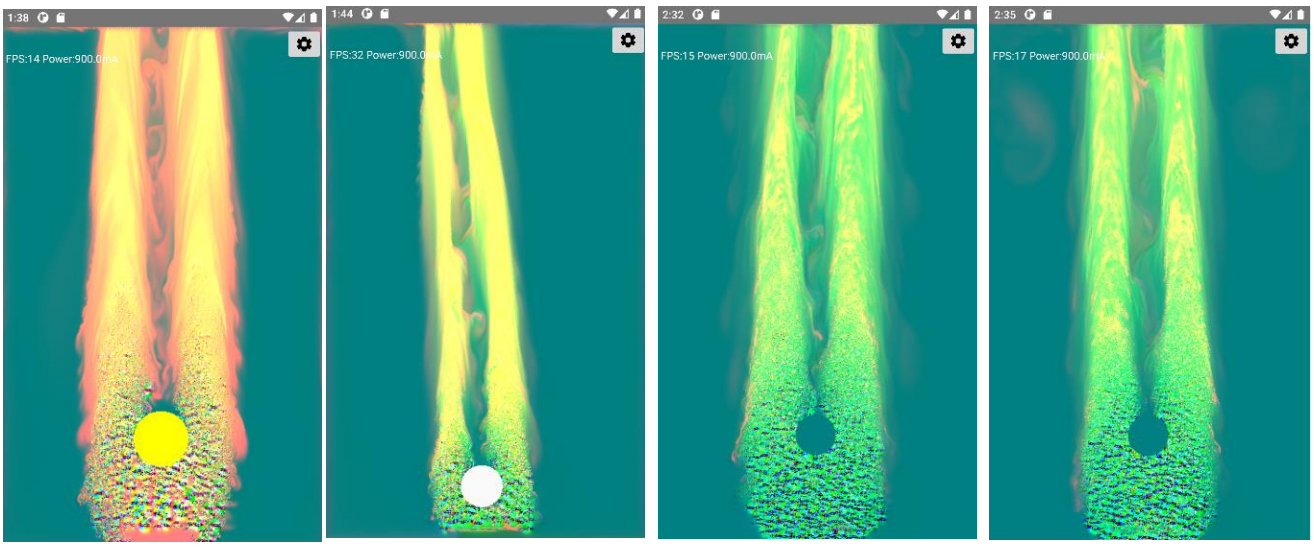


Fig12. Circle obstacle in Java /Native C/ Halide CPU /Halide OpenCL

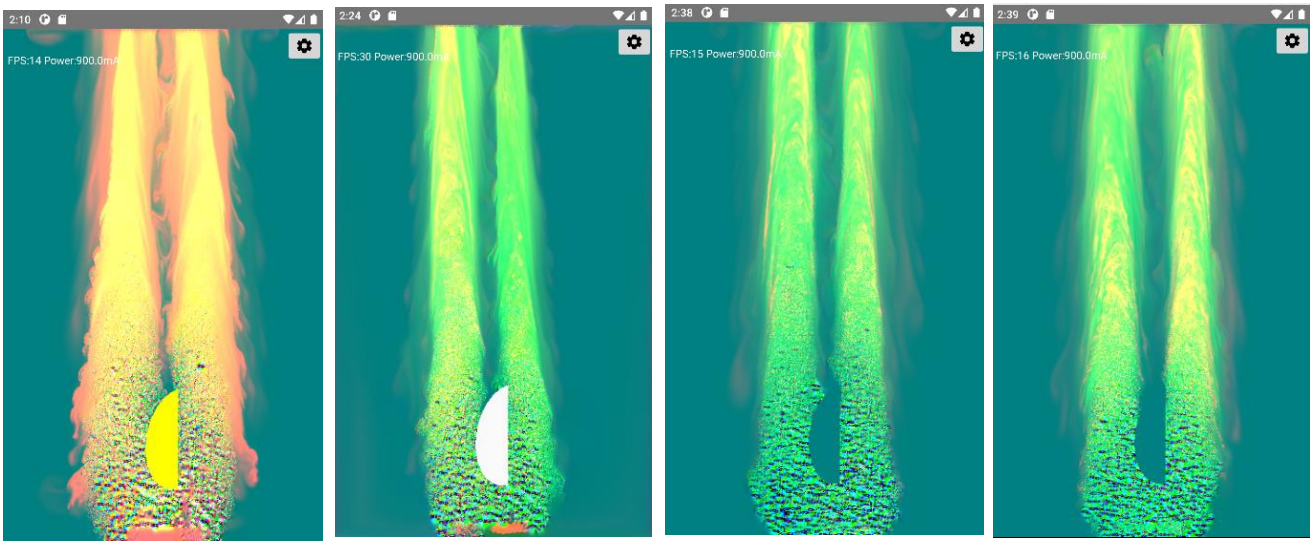
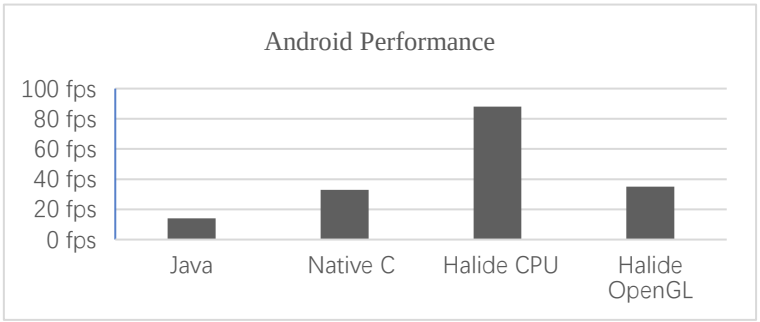


Fig13. Air foil obstacle in Java /Native C/ Halide CPU /Halide OpenCL

The demo results show that Halide has a higher frame rate than C++ .



Java	Native C	Halide CPU	Halide OpenGL
14fps	33fps	88fps	35fps

Fig14 Android performance

5.2 About Karman vortex street

The appearance of vortexes is related to the size of the radius of the circular obstacle, the viscosity and especially to the Reynolds number, as discussed by Griebel et al. in their book "Numerical Simulation in Fluid Dynamics " discusses the conditions for the appearance of vortex streets. For highly viscous fluids (Re < 4), the flow splits in front of the obstruction before quickly coming together behind it. The friction along the obstacle surface is no longer sufficient to immediately re-join the two flow segments at lower viscosities (4< Re < 40).When the Reynolds number rises above 40, the flow becomes asymmetric and unstable, and eddies alternately shed in a time-periodic fashion from the upper and lower edges of the disc. The Karman vortex roadway is the name given to this wake of vortices.

5.3 Reflection

Some problems have been encountered in the coding process, for information only:

- Halide compilation error: Unhandled exception: Error: Func f1 cannot be given a new update definition, because it has already been realized or used in the definition of another Func
  Since the Func input parameter u(x,y) is duplicated by the output parameter, the new Func cannot be used when processing the input u(x,y) in set_bnd, and if it is used there will be a compilation error. I solved the problem after adding an intermediate temp Func.
- Add android log in JNI: #include <android/log.h> #define LOGI(...) __android_log_print(ANDROID_LOG_INFO, LOG_TAG, __VA_ARGS__)
  However, printf and cout printing function in Halide is not suitable for printing in android, I tried to support android_log printing by linking liblog.so, but it doesn't seem to work.
- JNI Runtime error:
  04-22 10:51:48.972 3627 657 F libc : Fatal signal 11 (SIGSEGV), code 2 (SEGV_ACCERR), fault addr 0xb822eb08 in tid 3657 (UpdateThread), pid 3627 (rtin. simulation)
  The problem is solved by adding BoundaryConditions::constant_exterior to handle the input

Conclusion

We adopt the obstacle in a Fluid simulation project which is using semi-Lagrangian method to solve NSEs. We use java, C++ and Halide code for implement. Then we compared obstacle effects and data in the real-time fluid. The huge advantages of Halide for graphic processing code are obvious: bounds inference, automatic parallelization and multiple architectures. Halide CPU gives best mobile

performance, Halide with OpenGL is more balanced in Performance and Power Consumption.

In the future, further graphics can be added, including simulation of fluid motion in the 3D case under different boundary algorithm conditions. Boundary algorithms can be optimised to add more boundary cell types. Simulations can also be performed on a wider range of platforms and hardware conditions to enrich the comparison data.

Reference

[1] Jos Stam, "Real-Time Fluid Dynamics for Games". Proceedings of the Game Developer Conference, March 2003

[2] Jos Stam Real-Time Fluid Dynamics for Games The art of fluid Animation

[3] Robert Bridson , Matthias Muller Eulerian Fluid Simulator

[4] Michael Griebel , Thomas Dornsheifer, Tilman Neunhoeffer Numerical Simulation in Fluid Dynamics-A Practical Introduction

[5] Dan Schroeder Lattice-Boltzmann Fluid Dynamics

[6] Youquan Liu An improved study of real-time fluid simulation on GPU

[7] Stam J, Fiume E. Turbulent wind fields for gaseous phenomena. In Proceedings of SIGGRAPH, 1993; 369–376.

[8] 20. Stam J. Stable fluids. In Proceedings of SIGGRAPH, 1999; 121–128.

[9] Fedkiw R, Stam J, Jensen HW. Visual simulation of smoke. In Proceedings of SIGGRAPH, 2001; 15–22.

[10] Halide website. http://halide-lang.org

[11] Youquan Liu; Xuehui Liu; Enhua Wu Real-time 3D fluid simulation on GPU with complex obstacles

[12] Mark J. Harris Fast Fluid Dynamics Simulation on the GPU