THE UNIVERSITY OF TEESSIDE

Fluid simulation

BSc Visualisation

Paul Demeulenaere

01/04/2008

Supervisor: Eudes Diemoz

Second reader: Tyrone Davison

THE UNIVERSITY OF TEESSIDE

SCHOOL OF COMPUTING AND MATHEMATICS

MIDDLESBROUGH

Fluid Simulation

BSc Visualisation

April 2008

Paul Demeulenaere

Supervisor: Eudes Diemoz

Second reader: Tyrone Davison

Abstract

The implementation of a fluid simulation with physical interaction thanks to obstacles by Navier-Stokes set of equation is the main objective of this project. The study of rendering technique is also necessary; the vector field from theoretical resolution is not really visually interesting.

The development has been separated in a few phases for each milestone necessary to complete this project, the analysis to set theoretical aspect, the design to order the development and finally the implementation in C++.

The analysis was done and gives algorithm for the approximation of displacement inside a two dimensional fluid. Some rendering techniques have also been studied. The analysis was very important in this project because the fluid simulation is not a very simple concept.

A tool fully portable was expected, the design of the fluid is essential. These classes are considered to be easy to use. The most of rendering classes doesn't depend of the API, again, to improve the portability.

The main application is done with Qt; it is a tool to develop program portable with a professional user interface. The final display is realized thank to OpenGL and shaders in Cg has been used for special effects.

Finally, the main objectives of this project have been completed, now, the Navier-Stokes equations are studied and some applications for the demonstration are developed. Furthermore, it was a very rewarding project.

Acknowledgements

First, I would like to thank my project supervisor, Eudes Diemoz who has supported and followed me during all the development process. I would like to thank my second reader, Tyrone Davison who has also helped me thanks to real times graphics sessions.

Then, I would like to thank you all the staff of the practical project for the guidance given in methodology and report writing.

Contents

ABSTRACT 3

ACKNOWLEDGEMENTS 4

CONTENTS 5

CHAPTER 1 - INTRODUCTION 8

CHAPTER 2 - METHODOLOGY 9

1. DEVELOPMENT APPROACH 9
2. TESTING 9
3. BACKUP 10

CHAPTER 3 - FLUID ANALYSIS 11

1. HISTORY 11
2. FOREWORDS 12
A. SYMBOL AND WRITING 12
B. VECTOR FIELD 12
3. DEFINITION 13
4. DECOMPOSITION OF NAVIER-STOKES EQUATIONS 15
A. PROJECTION RESOLUTION 16
B. RECAPITULATIVE 17
5. RESOLUTION IN A FINITE SPACE 18
A. VISCOSITY 18
B. ADVECTION 22
C. PROJECTION 24
D. BOUNDARY CONDITIONS 26
E. FILLING OBSTACLES 28

CHAPTER 4 - RENDERING ANALYSIS 29

1. VELOCITY 29
2. VORTEX 29
3. PARTICLES 31
4. OTHERS TECHNIQUES 32
A. BUMP EFFECT 32
B. COLOUR REPRESENTATION 32

CHAPTER 5 - FLUID DESIGN 33

1. VECTOR 33
2. FIELD 34
3. OBSTACLE 35
4. FLUID 36

CHAPTER 6 - RENDERING DESIGN 37

1. COLOUR 37
2. DISPLAY 38

CHAPTER 7 - IMPLEMENTATION 39

3. GLOBAL IMPLEMENTATION BY QT 39
A. HISTORY 39
B. SPECIFICATIONS 40
C. USAGE 41
4. FLUID IMPLEMENTATION 42
A. GENERAL IMPLEMENTATION 42
B. DIFFICULTY 42
C. OPTIMIZATION 43
D. PARALLEL EXECUTION 44
5. RENDERING IMPLEMENTATION 47
A. OPENGL 48
B. SHADERS 49

CHAPTER 8 - IMPROVEMENTS 52

1. 3D FLUID 52
2. NON-REAL TIME RENDERING 53
3. CUDA 54

CHAPTER 9 - CONCLUSION 55

REFERENCES 56

LIST OF FIGURES 57

APPENDIX A – SCHEDULE 58

APPENDIX B – UML 59

Chapter 1 - Introduction

Fluid Simulation is used in meteorology, aerodynamics and hydrodynamics among others. It can be in real (e.g. a wind tunnel) or in digital simulations. The Navier-Stokes set of equations represent the fundamentals behind fluid simulation. It is very difficult to resolve fluid simulation analytically. These days, fluid simulation can be approximated thanks to computers. Video Games can also use fluid simulation on the GPU to simulate dynamic smoke or clouds. The movies use fluid simulation to make water of fire effect. One of the first applications is relatively recent; it is the movie "Antz" of DreamWorks in 1998, Jos Stam has wrote a paper the following years about the computation of fluid, today, it is the main reference of study about this subject.

The aim of this project is to develop a tool to simulate dynamic and fast fluid and create some sample applications in order to sell this product.

The mains objectives of this final year projects are:

- Implementation of a fluid simulation thanks to the CPU
- Physical interaction with obstacles
- Study of the implementation by the GPU
- Demonstration applications

This document will describe the process of development of this project, the work methodology will be explain first, then the theory about computation of fluid simulation, following by the design of the tool, and finally the implementation and possibility of ameliorations.

Chapter 2 - Methodology

1. Development approach

The final year project is not a project realized into a company but it needs a methodology to optimize the work and avoid a useless waste of time. The main model used in this project was a waterfall iterative method, or also called incremental method. This model consists in a division of the project in sequential phases. It is incremental because all analysis hasn't been done in the same time; it is possible to divide the project in some milestones: for example, the first can be the implementation of the fluid without thinking about rendering technique, then, when the fluid is computed and debugged, the techniques or rendering can be studied.

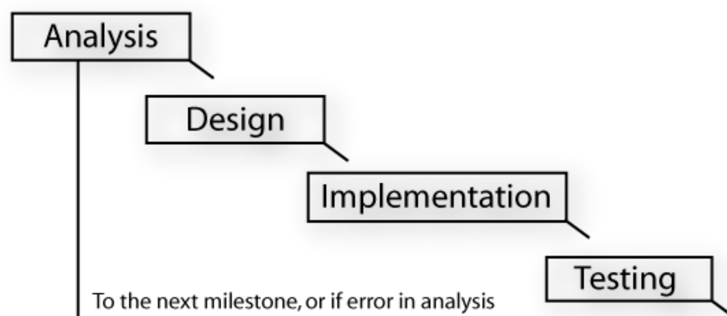


Figure 1: Development process

If the project is a company project with different members who have different skills, the method can be more complex to avoid that someone has to wait for the work of another person. But this project is a personal project, so, the "waterfall" method is probably the best one because it is difficult to work on different subjects in the same time.

2. Testing

The testing is an important part. Indeed, if the test phase is ignored, the development can be complicated and the debugging phase becomes very difficult. To simplify the test phase, it is necessary to divide the work: when a class is finished, it need to be debugged before do another module.

In C++, the debugger tools of Visual Studio are powerful: it is possible to put breakpoint before an instruction to check the result of this last one and follow computation looking for errors. Some tools are very useful, like conditional breakpoint to stop the program in particularly conditions or the call stack which permit to indentify the problematic function when an unexpected error appears such as a bad pointer.

When a graphical API is used or a Shading Language is compiled, it is required to be careful about the compatibility with most of the hardware. It is not possible to test an application on every configuration imaginable with all possible drivers, so, the easiest and the more judicious is to look at the documentation of specific functions to be aware of possible problems during the utilisation. In more important project using a lot of hardware capabilities, it is a very important phase to avoid compatibility problems.

3. Backup

The backup is essential in a project. It is easy to forget a lot of work due to a hard disk crashes. There is also a risk of a bad manipulation which can modify or overwrite an important file. Some software permits to backup a folder. For this project, "SyncBack" has been chose. This software can copy all a folder to another hard drive or upload it on a FTP server. This software is simple and free. It is possible to plan a regular backup compressed or not ignoring some file extension, like Intellisence (Visual Studio) files which are generated to speed up autocompletion but which can be very heavy.

There are other development tools which can be very powerful. The most famous is the SVN (Subversion) which is a "new version" of the CVS (Concurrent Version System). This system permit to several programmers to work on the same project, it is used on very big project and particularly open sources project. It works on a server and need a lot of space because all modifications and all adding are saved. There are a lot of tools to communicate with a SVN server as, for Windows, TortoiseSVN or eSvn, on the other operating systems, they permit to synchronise a folder with the latest files developed.

Chapter 3 - Fluid analysis

The aim of this analysis is to understand the simulation of a fluid as it is described by physician and as it can be interpreted by a computer. Mathematical concepts are sometimes complicated but the essential is to see the origins of these algorithms.

1. History

The equations which governed fluid displacement have been described for the first time by the French mathematician Claude Navier. Twenty years later, the Irish scientist George Stokes improved Navier work including a friction term. These equations are based on a famous Newton law: $\vec{F} = m\vec{a}$.

Analytically resolution is very difficult; Navier-Stokes is still an enigma. The *Clay Mathematics Institute* had promise (the 24th may 2000) to give one billion dollars to resolve it.

Figure 3 : Claude Navier

Figure 2 : George Stokes

2. Forewords

These sections going to explain how simulate a fluid by Navier-stokes equation in two dimensions. First, it is necessary to define some concept and writing conventions.

a. Symbol and writing

Symbol	Name	Definition
$\vec{\nabla} f(x,y)$	Gradient	$\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$
$div\ f(x,y)$	Divergence	$\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}$
Δf	Laplacian	$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$

b. Vector field

A vector field is a mapping of a vector-valued function onto a parameterized space, such as a Cartesian grid. Vector field are often used in physic to represent field of forces, in this section, vectors will represent a velocity.

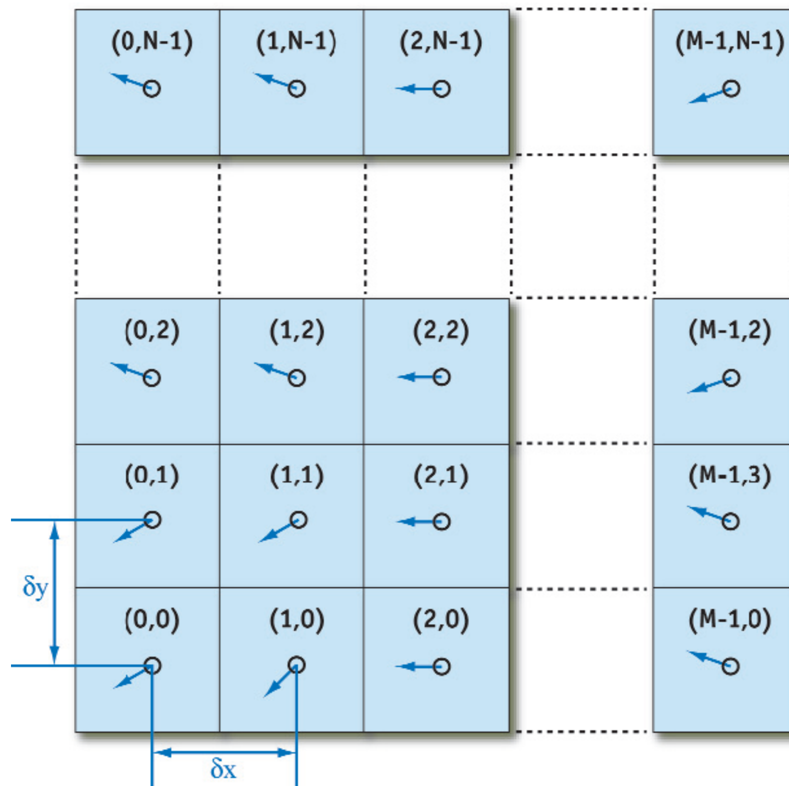


Figure 4 : Vector grid (from GPU gem)

3. Definition

An incompressible and homogeneous fluid can be represented by a velocity field $u(x, y, t)$. A fluid is incompressible when the volume is constant over the time, and this is homogeneous when its density is constant in space.

This incompressible and homogeneous fluid is defined by Navier-Stokes Equation:

$$\frac{\partial u}{\partial t} = -(u.\vec{\nabla})u - \frac{1}{\rho}\vec{\nabla}p + v\Delta u + F$$

v and ρ are constant representing the fluid (viscosity and pressure coefficients), and F represent external forces that act on the fluid (e.g.: gravity or wind). p is the scalar field for pressure.

Because the fluid is incompressible, it is possible to add:

$$div\ u(x, y, t) = 0$$

There is a condition on boundaries. Indeed, this equation is defined in infinity. The limitation of the effect can be obtained applying this condition on boundaries.

$$u(x, y, t).n = 0$$

Where n is normal vector on the boundary.

(J.Harris, 2006)

It is interesting to identify different terms in this Navier-Stokes equation:

Advection	$(\boldsymbol{u}.\nabla)\boldsymbol{u}$
Viscosity	$v\Delta u$
Pressure	$\frac{1}{\rho}\vec{\nabla}$

Advection defines how the fluid transports densities, objects or itself.

When a force is applied to the fluid, it does not instantly propagate trough the entire volume, it make *pressure* in the fluid.

A fluid can be more “thicker” than another; *viscosity* coefficient depends of fluid type and its temperature:

Fluid	Viscosity coefficient
Water - 20° C (68° F)	10^{-3}
Air – 0°C	$17{,}1 \times 10^{-6}$
Honey	10
Vegetable oil	Between 81×10^{-3} and 100×10^{-3}

(Viscosity - Wikipedia, 2008)

4. Decomposition of Navier-Stokes Equations

The main difficulty in Navier-Stokes equations is the pressure, we only know this term by its gradient. Helmholtz Hodge theorem permit avoid this problem.

Helmholtz-Hodge Decomposition Theorem

A vector field w on Ω can be decomposed in the form:

$$w = u + \vec{\nabla} p$$

Where u has zero divergence

(J.Harris, 2006)

This theorem can define a projection operator $\mathbb{P}$ which projects a vector field w into:

$$\mathbb{P}(w) = u$$

In these conditions, there are some interesting properties:

First property: $\mathbb{P}\left(\vec{\nabla} p\right) = 0$

Second property: $\mathbb{P}(u) = u$

By Helmholtz-Hodge Decomposition and application of $\mathbb{P}$ on first fluid equation and thanks to the last two properties:

$$\mathbb{P}(w) = \mathbb{P}(u) + \mathbb{P}(\vec{\nabla} p)$$

$$\mathbb{P}\frac{\partial u}{\partial t} = \mathbb{P}(-(u.\nabla)u - \frac{1}{\rho}\vec{\nabla}(p) + v\Delta u + F)$$

$$\frac{\partial u}{\partial t} = \mathbb{P}(-(u.\nabla)u + v\Delta u + F)$$

a. Projection resolution

$$\frac{\partial \tilde{u}}{\partial x}(x, y, t) = -(u.\nabla)u + v\Delta u + F$$

$$\tilde{u}(x, y, t_0) = u(x, y, t_0)$$

With boundary conditions: $\tilde{u}.n = 0$

Applying projection, it gets:

$$\mathbb{P}\left(\frac{\partial \tilde{u}}{\partial t}\right) = \frac{\partial \mathbb{P}(\tilde{u})}{\partial t} = \mathbb{P}(-(u.\nabla)u + v\Delta u + F) = \frac{\partial u}{\partial t}$$

Yields:

$$\mathbb{P}(\tilde{u}) = u + k$$

Where k is a constant vector field on time with $div\ k = 0$, k is null because $\tilde{u}(x, y, t_0) = u(x, y, t_0)$, so there is:

$$\mathbb{P}(\tilde{u}) = u$$

Thanks to Helmholtz-Hodge theorem, it is possible to confirm:

$$\tilde{u} = u + \vec{\nabla}(\varphi)$$

$$div\ \tilde{u} = div\ u + div\left(\vec{\nabla}(\varphi)\right)$$

There is the condition $div\ u = 0$, so:

$$div\ \tilde{u} = div\left(\vec{\nabla}(\varphi)\right)$$

$$\mathrm{div}\ \tilde{u} = \Delta\varphi$$

This last equation is a Poisson equation; the resolution of this equation is simple with a Gauss-Seidel relaxation for example.

Knowing $\tilde{u}$ and φ, the result of u is simple, because:

$$u = \tilde{u} - \vec{\nabla}(\varphi)$$

Again, the fluid is closed, there is the boundary condition $\tilde{u}.n = 0$

b. Recapitulative

This last demonstration is not really easy; the main goal is to understand the unwinding for the next part.

First, compute $\tilde{u}$ with:

$$\frac{\partial \tilde{u}}{\partial x}(x, y, t) = -(u.\nabla)u + v\Delta u + F$$

$$\tilde{u}(x, y, t_0) = u(x, y, t_0)$$

With boundary conditions: $\tilde{u}.n = 0$

Next, research of φ with:

$$\operatorname{div} \tilde{u} = \Delta\varphi$$

Finally we have u because:

$$u = \tilde{u} - \vec{\nabla}(\varphi)$$

5. Resolution in a finite space

So, the main equation to resolve is:

$$\frac{\partial \tilde{u}}{\partial x}(x, y, t) = -(u.\nabla)u + v\Delta u + \ F$$

This equation can be divided in:

$$\frac{\partial \tilde{u}}{\partial x}(x, y, t) = -(u.\nabla)u$$

And:

$$\frac{\partial \tilde{u}}{\partial x}(x, y, t) = \ v\Delta u$$

It is possible to add easily externals forces later. First equation represent the advection, second is the viscosity.

a. Viscosity

$$\frac{\partial \tilde{u}}{\partial x}(x, y, t) = \ v\Delta u$$

It is possible to formulate an explicit, discrete form of this equation in:

$$\tilde{u}(x, y, t + \delta t) = \tilde{u}(x, y, t) + v\delta t\Delta u(x, y, t + \delta t)$$

With δt a little interval of time and $\tilde{u}(x, y, t_0) = u(x, y, t_0)$

This representation is not really interesting because this method is not stable along time. The implicit form of first equation is more correct and it is a Poisson Equation:

$$(I - v\delta t\Delta)\tilde{u}(x, y, t + \delta t) = u(x, y, t)$$

Poisson Equation is resolvable with a Jacobi method or Gauss-Seidel relaxation. In fact, these two methods are very close.

(Stam, Stable Fluids, 1999)

Solution of Poisson Equation by Gauss-Seidel or Jacobi method

(math-linux, 2008)

They are iterative method for solving a linear system such as $Ax = B$

We say $A = M - N$ where M is an invertible matrix, so we have:

$$Ax = B$$
$$Mx = Nx + B$$
$$x = M^{-1}Nx + M^{-1}B$$

The algorithm used is:

$$\begin{cases} x^{(0)}\ given \\ x^{(k+1)} = M^{-1}Nx + M^{-1}B \end{cases}$$

If x is a solution of $Ax = B$ then $x = M^{-1}Nx + M^{-1}B$

Programming, it is not necessary to check the result on each iteration, it will be very expensive: making about twenty compute is quicker and, most of the time, it is a correct approximation.

It is difficult to compute $x^{(k+1)} = M^{-1}Nx + M^{-1}B$, Jacobi and Gauss Seidel methods permit to avoid inversion matrices compute:

A is decomposable in this way:

$$A = D - E - F$$

Where:

- D is the diagonal
- E the strictly lower triangular part of A
- F the strictly upper triangular part of A

Jacobi method

With Jacobi method, we choose $M = D$ and $N = E + F$

$$x^{(k+1)} = D^{-1}(E + F)x^{k} + D^{-1}B$$

With the ith line of $D^{-1}(E + F)$ is $-\left(\frac{a_{i,1}}{a_{i,i}}, \dots, \frac{a_{i,i-1}}{a_{i,i}}, 0, \frac{a_{i,i+1}}{a_{i,i}}, \dots, \frac{a_{i,n}}{a_{i,i}}\right)$

Yields:

$$x_i^{k+1} = -\frac{1}{a_{ii}} \sum_{j=1, j\neq i}^{n} a_{ij}\, x_j^{(k)} + \frac{b_i}{a_{ii}}$$

Gauss Seidel method

With Gauss-Seidel method, we choose $M = D - E$ and $N = F$

$$x^{(k+1)} = (D - E)^{-1}Fx^{k} + (D - E)^{-1}B$$

Yields:

$$x_i^{k+1} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}\, x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}\, x_j^{(k)}}{a_{ii}}$$

Before have an algorithm for viscosity, it is important to have a finite form for the Laplacian operator Δ :

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

This operator can be approximate in a Cartesian space by:

$$\Delta f_{i,j} = \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{\delta x^2} + \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j+1}}{\delta y^2}$$

With δx and δy are grid spacing, we will admit $\delta x \cong \delta y \cong \left(\frac{x+y}{2}\right)^{-1}$ because most of the time: $\delta x = \delta y$

Algorithm for viscosity by a Jacobian iteration

```
Algorithm ComputeViscocity
        Input   : Vector Field « In », viscosity value « v », time
        interval δt, size δx (invert of (sizeX+sizeY)/2)
        Output : Vector Field « Out »

        Float alpha = (δx* δx) / ( v* δt)
        Float rBeta = 1/(4 + alpha)

        for k=0 to 20 do
          for i=1 to In.sizeX-1
            for j=1 to In.sizeY-1
                out[i][j] = (out[i+1][j] + out[i+1][j] + out[i][j+1] +
                out[i][j-1] + in[i][j]* alpha )/rBeta

        ApplyConditionsOnSide(out);
```

b. Advection

$$\frac{\partial \tilde{u}}{\partial x}(x, y, t) = -(u.\nabla)u$$

Again, it is possible to formulate an explicit, discrete form of this equation:

$$\tilde{u}(i, j, t + \delta t) = \tilde{u}(i, j, t) - \delta t(u.\nabla)u$$

But, like with viscosity, this method is not stable along time, It is more judicious to use the method describe in GPU gems: "The implicit advection step traces backward through the velocity field to determine how quantities are carried forward" (J.Harris, 2006):

$$\tilde{u}(i, j, t + \delta t) = u\big(x - u_x(x, t)\delta t, x - u_y(y, t)\delta t, t\big)$$

Or in discredited space:

$$\tilde{u}(i, j, t + \delta t) = u\left(i - u_x(i, t)\frac{\delta t}{\delta x}, j - u_y(j, t)\frac{\delta t}{\delta y}, t\right)$$

With this equation, advection is very simple but there are two problems with this implementation.

First, because the fluid is closed, values of $i - u_x(i, t)\frac{\delta t}{\delta x}$ or $j - u_y(j, t)\frac{\delta t}{\delta y}$ can overtake the fluid but it is possible only for high velocity and small exceedance on boundaries and it is already physically not really correct in this situation, so the method consist in simply put back this values inside the fluid.

Then, values of $(i, t)\frac{\delta t}{\delta x}$ or $u_y(j, t)\frac{\delta t}{\delta y}$ are not always integers. So, a measure the contribution of the four vectors near computed value will be required.

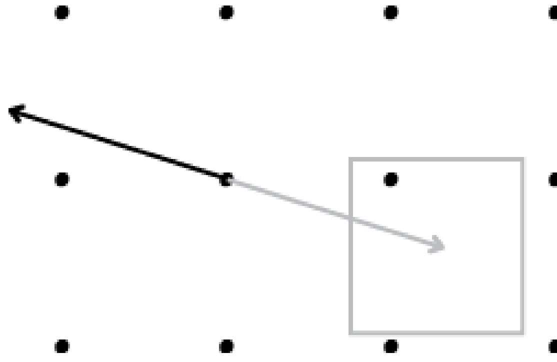


Figure 5: The geometrical representation of the advection term

Algorithm for advection with area compute

```
Algorithm ComputeAdvection
        Input  : Vector Field « In », time interval δt, size δx (invert
        of (sizeX+sizeY)/2)
        Output : Vector Field « Out »

        float alpha = δt / δx
        int dx = In.SizeX
        int dy = In.SizeY

        for i=1 to In.sizeX-1
            for j=1 to In.sizeY-1

                float res = Vector(i,j) - In[i][j]*alpha

                /* stay inside */
                if res.x<0.5 then
                   res.x = 0.5
                elseif res.x>0.5+dx
                   res.x = 0.5 + dx

                if res.y<0.5 then
                   res.y = 0.5
                elseif res.y>0.5+dy
                   res.y = 0.5 + dy

                /* research of integer values */
                int iresx = res.x;
                int iresy = res.y;

                /* Areas contribution */
                float Asw = (1.0+iresx-res.x) * (1.0+iresy-res.y);
                float Ase = (res.x - iresx)   * (1.0+iresy-res.y);
                float Anw = (1.0+iresx-res.x) * (res.y - iresy);
                float Ane = (res.x - iresx)   * (res.y - iresy);

                /* Interpolate the result */
                out[i][j] = in[iresx][ iresy]*Asw
                            + in[iresx+1][ iresy]*Ase
                            + in[iresx][ iresy+1]*Anw
                            + in[iresx+1][ iresy+]* Ane

        ApplyConditionOnSide(Out);
```

c. Projection

Now, after adding additional forces, the computation value of $\widetilde{u}(i, j, t + \delta t)$ is possible. Since decomposition of Navier-stokes equation, there are:

$$\operatorname{div} \tilde{u} = \Delta\varphi$$

And

$$u = \tilde{u} - \vec{\nabla}(\varphi)$$

Finite form for divergence will be necessary:

$$div\ f = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}$$

$$div\ f_{i,j} = \frac{f_{i+1,j} - f_{i-1,j}}{2\delta x} + \frac{f_{i,j+1} - f_{i,j-1}}{2\delta y}$$

Like with Laplacian operator, δx and δy are grid spacing and it will be admitted that $\delta x \cong \delta y \cong \left(\frac{x+y}{2}\right)^{-1}$, so:

$$div\ \tilde{u}_{i,j} = \frac{\tilde{u}_{i+1,j} - \tilde{u}_{i-1,j} + \tilde{u}_{i,j+1} - \tilde{u}_{i,j-1}}{2\delta x}$$

To avoid error by division:

$$2\delta x * \operatorname{div} \tilde{u} = 2\delta x * \Delta\varphi$$

Therefore, algorithm to compute $2\delta x * \operatorname{div} \tilde{u}$ is possible, this is a scalar field called β:

```
Algorithm DivSquare
        Input : Vector Field « In », size δx (invert of
        (sizeX+sizeY)/2)
        Output : Scalar Field « Beta »

        for i=1 to In.SizeX-1
            for j=1 to In.SizeY-1
                Beta[i][j] = 0.5*(In[i+1][j].x-In[i-1][j].x +
                In[i][j+1].y-In[i][j-1].y)


        ApplyConditionOnSide(Beta)
```

Now, the equation is $2\delta x * \Delta\varphi = \beta$, like with viscosity, it is a Poisson Equation resolvable thank to a Gauss-Seidel method.

```
Algorithm ComputePhi
        Input  : Scalar Field « Beta »
        Output : Scalar Field « phi »

        for k=0 to 20 do
          for i=1 to In.sizeX-1
            for j=1 to In.sizeY-1
                phi[i][j] = (-Beta[i][j] + phi[i-1][j] + phi[i+1][j] +
                phi[i][j+1] + phi[i][j-1])/4.0

        ApplyConditionOnSide(phi)
```

After this algorithm, the scalar field φ (in fact, $\varphi * 2\delta x$), is computed thanks to relation $u = \tilde{u} - \vec{\nabla}(\varphi)$, the value of u is computable.

The finite form of gradient operator is:

$$\vec{\nabla}\varphi_{i,j} = \left(\frac{\varphi_{i+1,j} - \varphi_{i-1,j}}{2\delta x}, \frac{\varphi_{i,j+1} - \varphi_{i,j-1}}{2\delta y}\right)$$

Again, $\delta x \cong \delta y \cong \left(\frac{x+y}{2}\right)^{-1}$

```
Algorithm ComputeU
Input  : Scalar field « phi », size δx (invert of (sizeX+sizeY)/2)
Output : Vector field « u »


        for i=1 to In.sizeX-1
           for j=1 to In.sizeY-1
                    u[i][j]          =          u[i][j]          +
                Vector(-((phi[i+1][j]      -      phi[i-1][j]))*0.5f),
                -((phi[i][j+1] - phi[i][j-1]))*0.5f)))

        ApplyConditionOnSide(u);
```

To conclude with computation of projection of $\tilde{u}$ into u, we it is required to make an algorithm using the last trhee:

```
Algorithm ComputeProjection
        Input  : Vector field « In »
        Output : Vector field « Out »
        ScalarField Phi
        ScalarField Beta
        DivSquare(In, δx,Beta)
        ComputePhi(Beta, δx,Phi)
        ComputeU(Phi, δx, Out)
```

d. Boundary conditions

Like is defined by the Navier-Stokes equation, we have to check $\tilde{u}.n = 0$ on the boundaries, but also, we have to keep the continuity of the fluid. n defined the normal of the boundary so, velocity vectors on the boundaries should have the same direction of the side (but not always the same way).

To keep the continuity and check $\tilde{u}.n = 0$, projection of the closest vector on the side is an acceptable method.

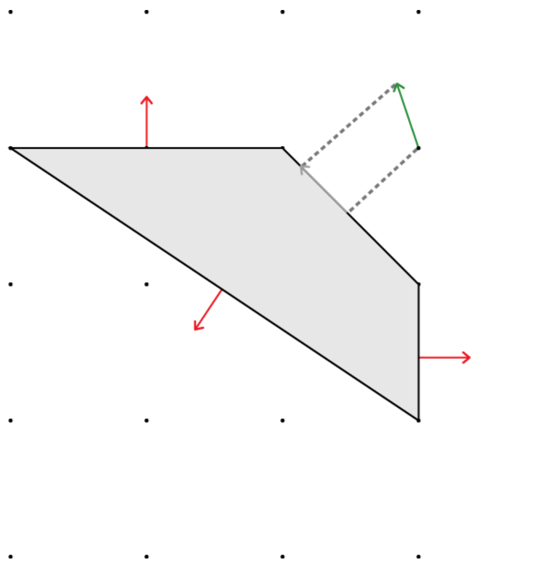


Figure 6 : Projection of a vector on a boundary

To project a vector on another, the simplest is using the scalar product.

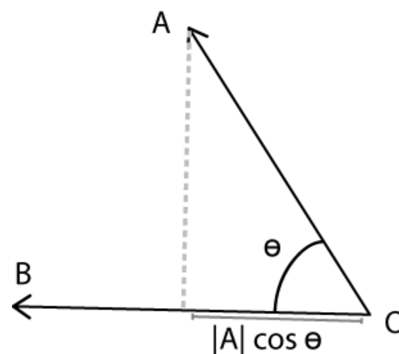


Figure 7 : The scalar Product

By definition, there is:

$$\overrightarrow{OA}.\overrightarrow{OB} = \left|\overrightarrow{OA}\right| \times \left|\overrightarrow{OB}\right| \times \cos\theta$$

So:

$$\left|\overrightarrow{OA}\right| \cos\theta = \frac{\overrightarrow{OA}.\overrightarrow{OB}}{\left|\overrightarrow{OB}\right|}$$

Or, in a 2D space:

$$\left|\overrightarrow{OA}\right| \cos\theta = \frac{x_a x_b + y_a y_b}{\sqrt{x_b^2 + y_b^2}}$$

The vector $\overrightarrow{OB}$ is pre-computed, so to optimize resolution of velocity vectors on the boundaries, it’s interesting to have a normalized vector tu avoid useless division. On horizontal and vertical boundaries, we have to notice that this computation is easier; it is just a copy of the x or the y value keeping the other component at null.

e. Filling obstacles

Then, it is essential to fill the obstacle with null velocity vector, we could convert the polygon in a list of triangle to apply the rasterisation algorithm, but in 2D, it is possible to scan the polygon from the bottom to the top looking for each vertex if we are inside or outside.

In fact, we check if a vertex is on the left or the right of each edge like is described in “Algorithmes pour la synthèse d’image et l’animation 3d (Algorithms for image synthesis and 3D animation)”, this method work with no-convex polygon, it is its main advantage.

This algorithm works with a particularly organization of data where an “array of edge” is defined for each coordinate on the y axis. This array of edge defined a potential begin of edges. When the polygon is scanned from the minimal value of y to the maximal value of y, an “active array of edges” which could be, in fact, a list of edges is used to store x values of the begin and the end of the polygon.

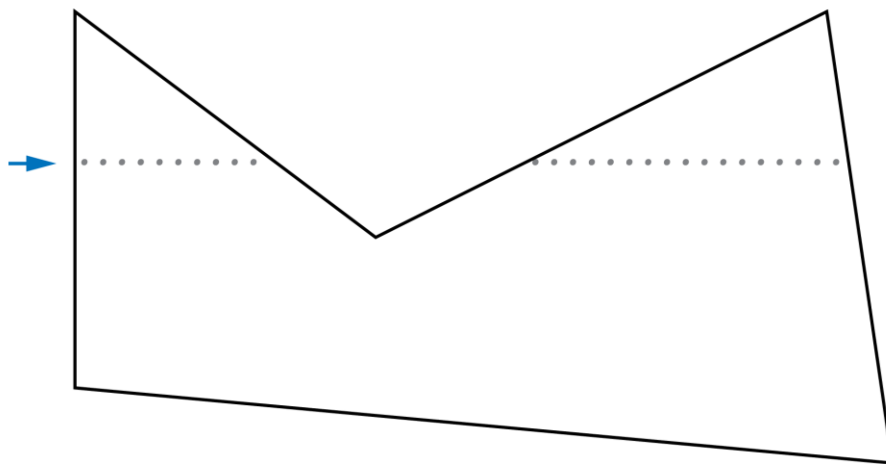


Figure 8 : The filling polygon technique used

Chapter 4 - Rendering analysis

Fluid simulation by Navier-Stokes equation gives a vector field, it could be interesting for an engineer to measure aerodynamic impact of car profile for example but in most of case, fluid simulation is used to create a visual effect.

1. Velocity

The first approach could be to render the fluid by value of the velocity, it is easy and powerful, the magnitude of each vector is computed and interpolated on a texture to render a fluid which look like a smoke.

2. Vortex

Also called curl, the vortex of a vector field is very interesting, it define how the fluid have the tendency to rotate about a centre like a twister turn around its eye.

$$curl\, u(x,y,z) = \begin{pmatrix} \frac{\partial u_z}{\partial y} - \frac{\partial u_y}{\partial z} \\ \frac{\partial u_x}{\partial z} - \frac{\partial u_z}{\partial x} \\ \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y} \end{pmatrix}$$

The fluid is not defined on z, so we can simplify by:

$$curl\, u(x,y,z) = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}$$

Again, it is possible to approximate value of curl in a discredited space by:

$$curl\, u(i,j) = \frac{u_x(i+1,j) - u_x(i-1,j)}{\delta x} - \frac{u_y(i,j+1) - u_y(i,j-1)}{\delta y}$$

We can see than sign of the curl define the way of rotation. For an engineer, this value is important, indeed, in aerodynamic, more a profile is optimized, less vortex it is generate. Then, generally, we can use this value to improve the display of the fluid defining a bump mapping for example.

Figure 9 : Vortex created by the passage of an aircraft wing (Wikipedia, 2008)

3. Particles

In fact, Navier-Stokes equation gives velocity of particles inside a fluid but in this resolution, particles doesn't move, so, it could be interesting to put elements which follow velocity around it.

There are a lot of rendering techniques of a particles field. One of most interesting is the Metaballs. Metaballs or "blobbies" are often used to represent organics shape or fluids. This concept has been invented by Jim Blinn in the 80's; he is also the creator of the bump mapping method.

It is feasible to understand this method as a representation of the gravity value between particles.

In physic, thanks to Newton laws:

$$F = G\frac{m}{d^2}$$

With *G* is a constant, *m*, mass and *d*, the distance from the object, to simplify, it is possible to declare $G = m = 1$. Therefore, this equation became:

$$F = \frac{1}{d^2}$$

If there is more than one particle, each contribution of each particle is added.

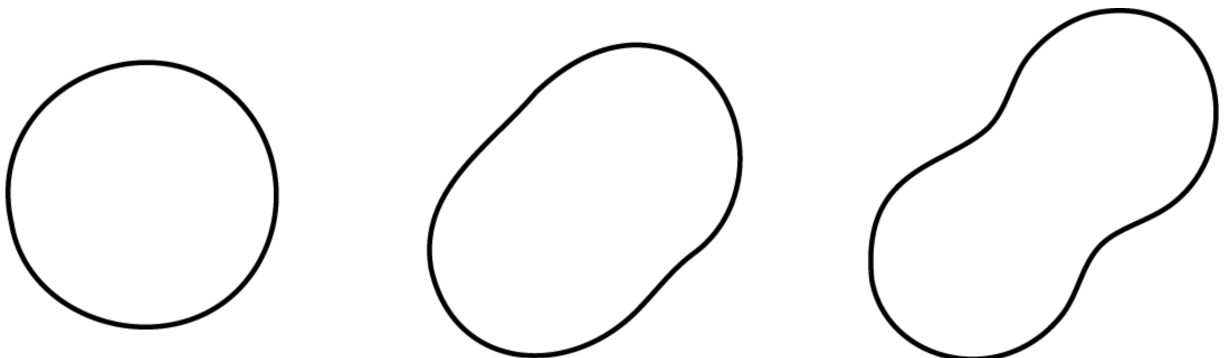


Figure 10 : two particles separating, setting a limit value to display edges

4. Others techniques

a. Bump effect

The render of a fluid can be very complex, particularly in three dimensional spaces, indeed, a volume rendering is necessary in this situation. This project has been designed for a 2D application but it is interesting to see that is possible to make a 3D surface effect of a fluid thank to a bump mapping or a mesh displacement, it is difficult to have a realistic effect using a 2D surface as a realist physic model because a fluid defined only on a plane doesn't exist. In fact, it is like the illumination in 3D: a light is often represented as a unique point but it is not possible in reality. A visual effect could be nice before realistic.

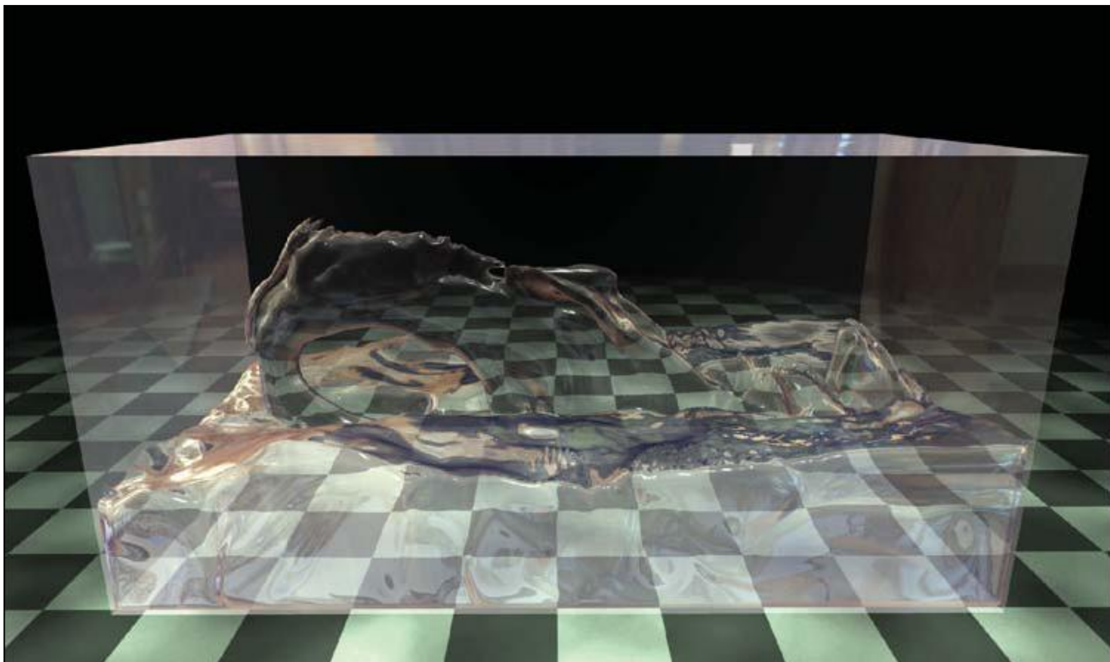


Figure 11 : 3D fluid from GPU gem 3

b. Colour representation

Colours in this project have been defined in HSL (Hue Saturation Lightness) instead of RGB (Red Green Blue) because with this representation of colour space, it is easier to implement a gradient from a colour to another, however screen and texture are in RGB, and thus, the conversion between these two representations had been studied.

Chapter 5 - Fluid design

The fundamental objective of this project is the development of a tool to compute a fluid, it is important to have a correct design for this part. Fluid classes must be independents and portable. The application has been developed in C++ with visual studio 2005, so, templates and classes are available. This part will explain the functioning of each part of this black box from the vector class to fluid class.

1. Vector

The vector class is very simple; this class have to be optimized because there are a lot of operations on velocity vectors. The main utility of this class is the simplification of operator like addition or dot product. It is more a structure than a class because there is not a private member. This structure contains simply two floating variables which represent value in x axis and y axis.

Vector
+x : float +y : float
+Vector() +~Vector() +Vector(in xi : float, in yi : float) +getNorme() : float +getNormalize() : Vector +getOrtho() : Vector +operator /(in v : const float &) : Vector +operator *(in v : const float &) : Vector +operator *(in v : const Vector &) : float +operator +(in v : const Vector &) : Vector +operator ^(in v : const Vector &) : Vector +operator -(in v : const Vector &) : Vector +Vector(in v : Vector &)

Figure 12 : Vector class

2. Field

The fluid is represented with a vector field, but a scalar field is also necessary (see “projection” in analysis part). An idea could be to make a mother class “field” and two child classes “scalar field” and “vector field” but in this situation, the templates are more interesting. Indeed, it permits to avoid a repetition in member function of child classes but vector class must have enough operators declared.

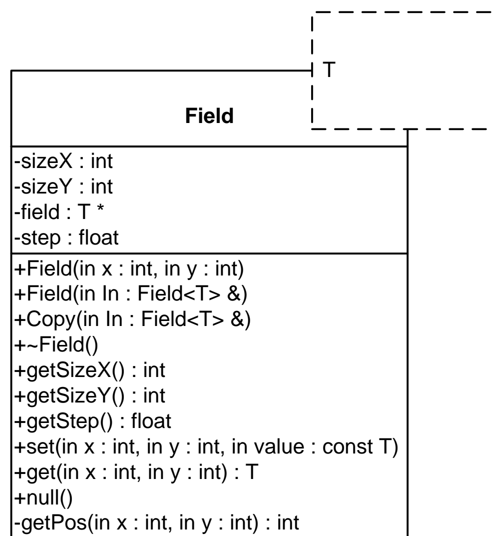


Figure 13 : Field template class

3. Obstacle

Obstacles computation could be very slow but it is often pre-computable, indeed, the vertices inside the polygon can be predicted, the points on the edges and their relation for example. When an obstacles is created, all computation possible have done, when the user, (the class fluid) want to put an obstacle on a fluid, he gives the scalar or vector field to modify values on this last one. The obstacle class uses two structures to organise the data, it is not used in another part of the program, so they could be stored in a namespace.

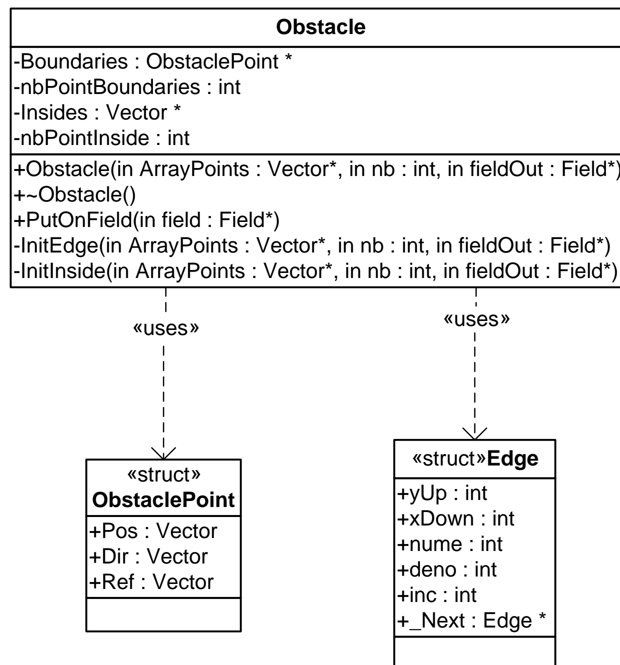


Figure 14 : the obstacles classes and tools

4. Fluid

The fluid class is probably the most important is this project, it create a vector field to represent the fluid as a velocity field. It is possible to modify value of viscosity, number of iteration for the resolution of Gauss-Seidel and Jacobi method (see “projection” or “viscosity”) and add obstacles. There is a lot of computation in this class, it is necessary to separate each computation to ease the development and debugging, that why there are three private functions “ComputeViscosity”, “ComputeAdvection” and “ComputeProjection”, when the user call “advance” function, the fluid call these last three. The fluid is readable thank to “GetVelocity” function.

Fluid
-velocity : Field * -obstacles : ObstacleList * -dt : float -nu : float -nb_Iteration : int -EnableViscosity : bool -VectorFieldTemp : Field * -phi : Field * -divS : Field *
+Fluid(in dx : int = 100, in dy : int = 100, in float time = 0,000000 : float = 0,000000, in float visco = 0,000000 : float = 0,000000, in nbIt : int = 20) +~Fluid() +AddVector(in x : int, in y : int, in In : Vector) +AddVector(in vertices : Vector*, in size : int, in In : Vector) +AddObstacle(in vertices : Vector*, in size : int) +Advance() +SetViscosity(in visco : float) +SetIteration(in nbi : int) +SetEnableViscosity(in enable : bool) +GetVelocity() : Field * -ComputeViscosity(in in : Field*, in out : Field*) -ComputeAdvection(in in : Field*, in out : Field*) -ComputeProjection(in in : Field*) -DivSquare(in in : Field*, in out : Field*) -ApplyConditionSide(in in : Field*)

Figure 15 : Fluid class

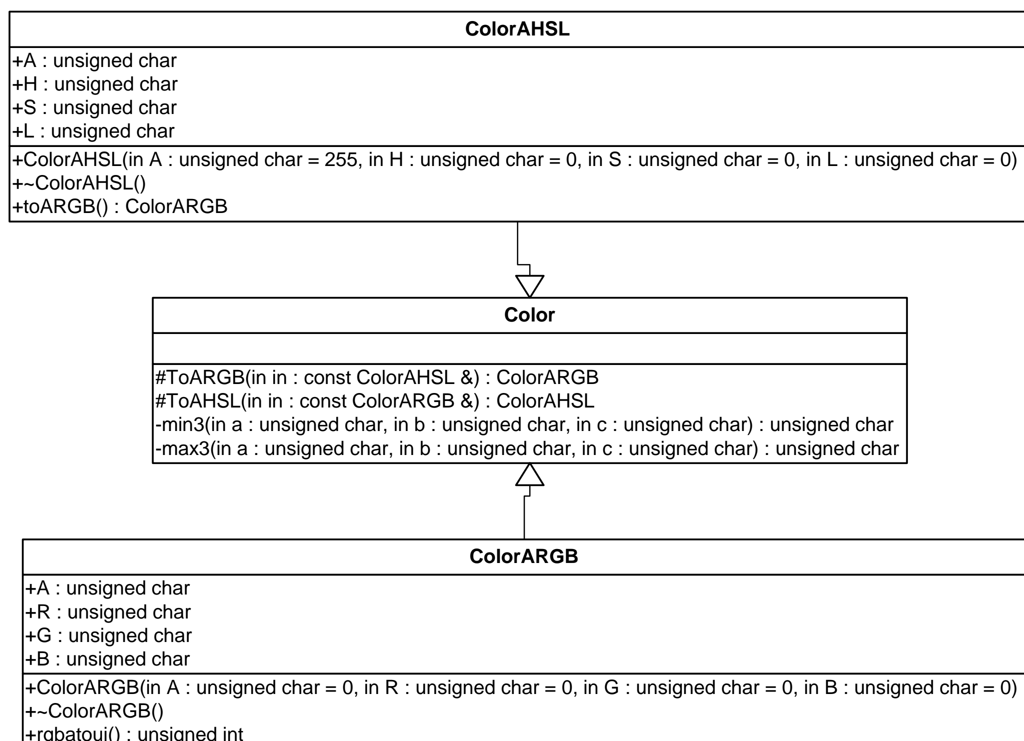
Chapter 6 - Rendering design

The renderer part can be seen has a tool to display the fluid. It is difficult to make a renderer which can make all effects possible, these classes are used to expose some samples of what is possible thank to fluid computation.

1. Colour

Colours for displaying has declared in an HSL (Hue Saturation Lightness) space but a computer can only render a colour with the three components: red, green and blue. The conversion method is needed.

It is possible to made two independent classes and conversion method separated but to simplify the development, another method has been chosen: A mother abstract class “color” with the two space conversions: ARGB to AHSL and AHSL to ARGB methods and two child classes “colorARGB” and “colorAHSL” which call methods from the mother class.



2. Display

In render analysis, some techniques of rendering are explained, the display class permit the drawing of the field as a velocity value, a vortex or simply, vectors. Even if the displaying has only been done on with an OpenGL application, the display class should be portable from an API to another.

There is an abstract "DisplayField" class which is totally separate from the API, there is not any draw function depending of the API, and the aim of this class is to create generic texture with an unsigned integer array for child class which really draw it. In this application, the child class is "DisplayOGL".

The user can chose the render technique thank to an enumeration with the shortcut VECTOR, VELOCITY and VORTEX, the draw function is always "DrawCurrent".

The DisplayField class is not directly linked to the fluid, the user send the field which is interpreted by this class thanks to an update function.

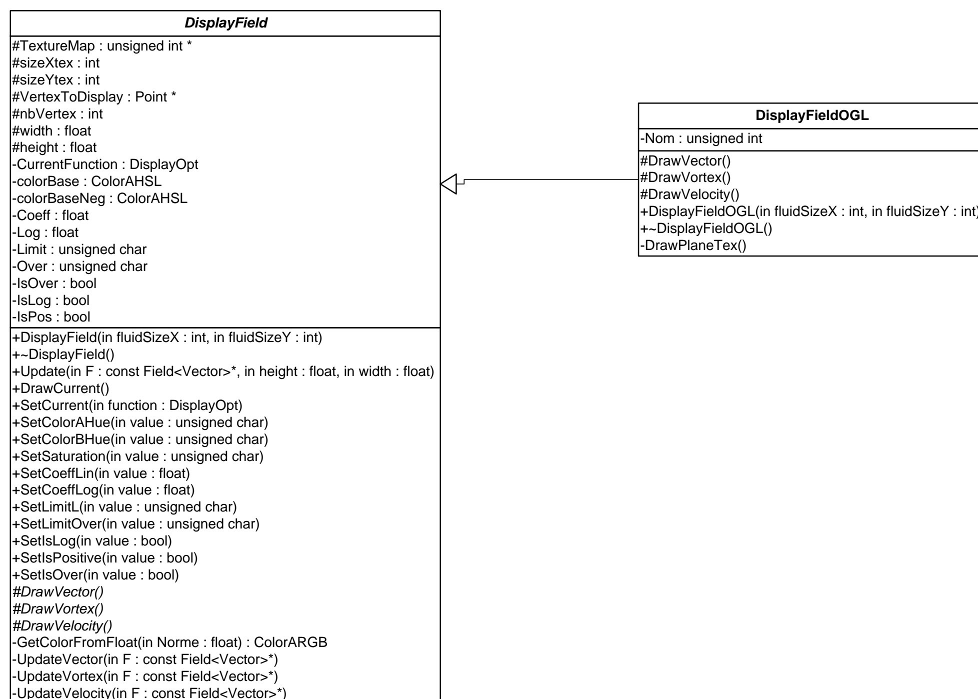


Figure 16 : Display classes

Chapter 7 - Implementation

3. Global Implementation by QT

The main application has been coded in C++ with the QT (pronounced “cute”) libraries which permit to create professional and portable application with a Graphical user Interface (GUI).

a. History

Eirik Chambe-Eng and Haavard Nord had begun the development of a multi-platform graphical API in 1991 at Trondheim in Norway for the “Norwegian Institute of Technology”. It is in March 1994 that Trolltech is created. The first version of QT is released in 1995, first versions was only on Unix and Windows environment, the Mac OS compatibility came in 2001 with the third version.

The last version release is Qt 4.3; the next update is actually in beta testing. Trolltech had also developed a user interface for Linux-based mobile called “Qtopia”. Recently, the 28th January 2008, Trolltech has been bought by another Norway company, Nokia.

Figure 17 : Trolltech logo

Figure 18 : QT logo

b. Specifications

Qt adds functionalities which don't natively exist in C++ thanks to a module called the "moc" (for Meta Object Compiler). The moc is run before the compiler is executed to interpret additional features such as the signal and slot system or the introspection. Introspection is the capability of an object to know information about itself as the list of its properties or its parents.

The signal and slot system is useful in a QT application; the main idea of this system is that it is possible to connect a "slot" which can be a member function of a class to a "signal" which is an event such as click on a button or the end of a timer. That is on this system that all the QT components communicate. Signals are also available with Boost Library but thank to QT, they can be threaded for asynchronous execution, it is possible to imagine a design where computation will be done in a particularly thread which communicate with the rest of the application with these signals.

Qt gives also a lot of possibilities; the best sample of application is KDE (K Desktop Environment) an open source environment of some Linux Operating Systems which is mainly based on Qt.

There are modules to establish communication by a Network or with a SQL database. It is possible to read and interpret XML. Qt understands a lot of type of data for images from the bmp to the SVG (Scalar Vector Graphics). It is possible to execute scripting language such as the JavaScript with QT. Especially, there is a module for OpenGL displaying like it is possible with the glut (OpenGL Utility Toolkit) to integrate an OpenGL device in the user interface.

It is also interesting to notice that a Qt application is fully portable on a Linux, Mac OS and Windows environment, the code and user interface files are the same.

c. Usage

About the conditions of utilisation, QT was a property technologies but it was incompatible with GPL (GNU General Public License) of KDE, since the third version, Trolltech have decided to open its product and apply a GPL licence for non-commercial application keeping a commercial licence for the other companies like Adobe Systems, Google or the NASA which can made benefit with this tool.

In this project, Qt has been including to make a user interface with a lot of components such as sliders and check boxes. The OpenGL integration is done thanks to QtOpenGL. The event handling such as mouse movement on the fluid to add artificial velocities is also completed thanks to QT. The UML of Qt part is described in the appendices.

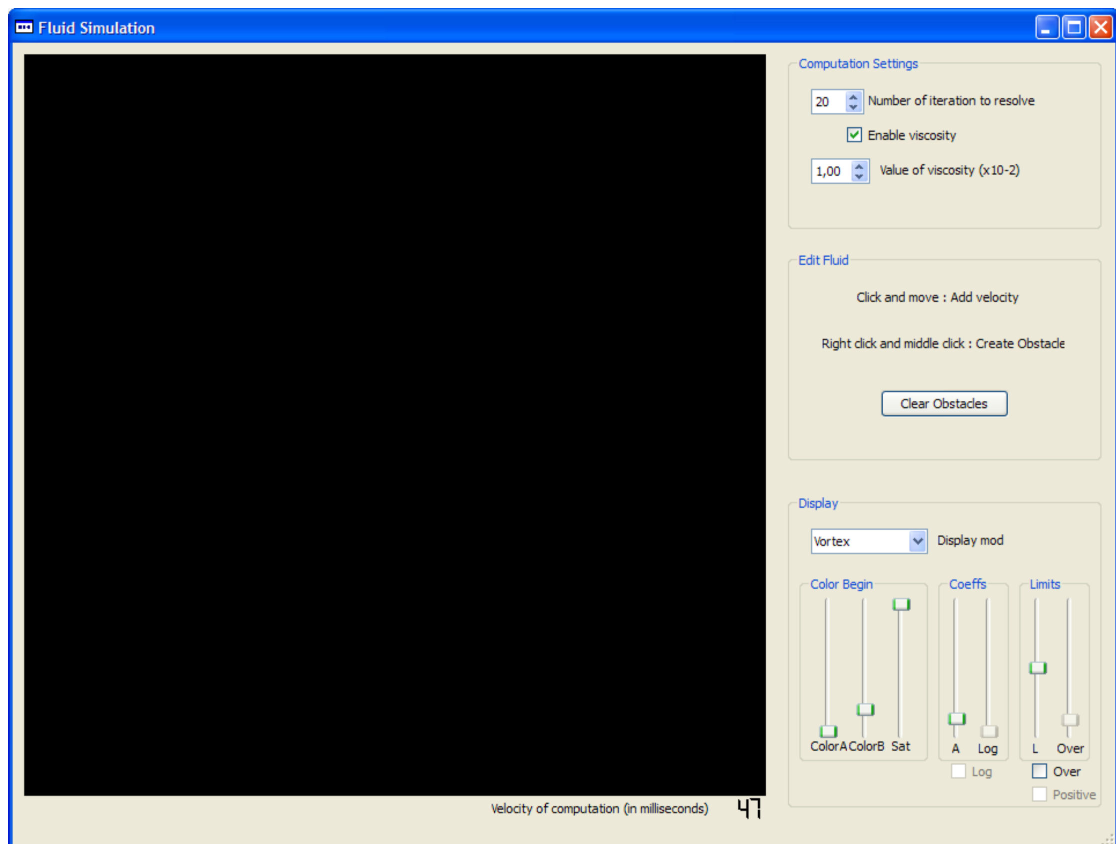


Figure 19: the user interface created

4. Fluid Implementation

The implementation of a fluid by Navier-Stokes equations resolution is the main objective of this project. This implementation must be portable and optimized.

a. General Implementation

The design of classes has been done to simplify the development of the fluid. Indeed, the computations of different parts are separated. The first objective is to implement the fluid the simplest that is possible, obstacles and rendering computation are secondary objectives.

Even if Qt is fully portable, it is important to avoid all externals dependencies of fluid classes. There is two external classes include into fluid computation. The first is “string.h” for memset and memcpy functions. These functions permit to modify big block of memory very quickly. It is possible to implement manually a setting of memory by an optimized way but this library is available on most of systems. The second is “math.h” for the square root (sqrt) function. This function is used to compute the length of a vector, like memory modification functions, it is possible to implement this function manually; there are a lot of algorithms to approximate a square root.

b. Difficulty

The main difficulty in the development of a fluid simulation is in the testing, the Navier-Stokes equations are very difficult to resolve analytically. It is necessary to approximate an idea of the result expected to check the product given by algorithm computed. The analysis must be close to the implementation in this part. It is only when the analysis is implemented that the errors are visible for a fluid simulation.

That is why some difficulties have appeared in this project, it is not easy to identify the source of the problem when the result excepted is not clear. The analysis has to be check a lot of time before the implementation of a correct result.

c. Optimization

The optimisation must be done when an application is fully tested, a code optimised can be very difficult to debug. The real optimisation of a code should depend of the hardware architecture, but there are some aspects which are universal in an optimisation process.

The first thing to do before really optimised is the profiling. The profiling is an analysis of performances. The main objective of a profiling is to identify the bottlenecks in the execution of an algorithm. It is possible to develop a profiler which analyses the execution of the application and gives information about the average call time of a set of particularly functions. In this project, a profiler hasn't been implemented; the measure of performances has been simply done disabling specific functions. The computation of viscosity and projection are now the two main bottlenecks, indeed, there is an iterative Poisson equation approximation which can be very slow in these two computations.

First, when bottlenecks are indentified, it is possible to look for values which can be pre-computed. Even if, this method can consume a lot of space memory, it can be interesting to speed up the performances of the application. The main algorithm which is massively precomputed in the fluid simulation is the filling of obstacles. The list of vertices inside and relation between vertices are saved when the obstacle is constructed. The pre-computation can slows the application, for example, if the access to the memory is slower than the real computation.

Finally, there is an easy aspect to forget in an optimisation process which can really increase the performances. It is necessary to choose judiciously type of data. For the testing phase, it is interesting to have accurate values to avoid errors of computation due to a too small precision, but later, when the algorithms are totally debugged, it is interesting to look at the optimum size of data used. For example, first, the class vector was implemented with floating point as double, but, the accurate of a simple float

is sufficient. On the most of hardware configurations, an operation of a double is slower than an operation on a float.

Furthermore, some types of data optimised are specific of a language or hardware, in cg, there is the half which represent an half float or in C++, it is possible to implement fixed point methods, this method consist in the storing a floating value into a integer to speed up the computation, it is a possibility but this techniques hasn't been implemented.

d. Parallel execution

Why does computations executable in parallelism instead of optimise a code to be executed linearly? It is simply to follow the actual evolution of hardware. Indeed, the speeds of components are slowing its progression to privilege parallel and multithread execution. Hardware's constructor are separating elements to have dedicated component for each computation such as the graphic cards, sound cards or, more recently physic cards. These components are often designed for parallel execution.

The CPU which is not really dedicated is also following this evolution. The multiprocessor or dual core technologies are not really recent but it is today that their success become. Instead of increase frequencies of processors, constructors had chose to multiply the numbers of cores. For example, Intel had developed the Pentium 4, it was the last generation of "simple" processor, this generation was made to reach 10ghtz frequencies, but there were several physical limitations. That why the actual generation of Intel Processor is different and permit numerous core on the same chip to run in the same time, "Duo Core" and "Quad Core". It is cheaper to build and can be powerful if applications considers this architecture.

A simple computation can be difficult to execute in parallel such as an average function of an array. Most of the computation in the fluid can be executed in parallel, on a fluid computation, it is often necessary to scan all a vector or scalar field to make a simple operation of each element. That is these computations which are easily parallelisable because the next result doesn't depend of the previous result and a shared memory is only necessary for the reading.

Actually, the most powerful parallel hardware which can be found on a computer is almost the graphic card. Indeed, the GPU (graphic process unit) is design to received information of a tree dimensional environment to give a display on a two dimensional screen. This goal involves a lot of computation on different elements such as the illumination computing on each pixel of the screen. That is why graphics card are designed to be massively parallel.

This is thanks to the shaders that computation of dynamic fluid simulation can be possible on the GPU. A shader is a program executed on the graphic card, there is mainly two kinds of shaders, the vertex shader which process position of vertices for the rasterizer, and the pixel or fragment shader which compute the colour of the pixel displayed. There is also a third kind of shader: the geometry shader is very recent and appears with version 3.0.

There is a few shading high level programming language, it is possible to advert the GLSL (OpenGL Shading Language) or the HLSL (DirectX High-Level Shader langage) but the shading language used in this project in the Cg from Nvidia, the particularity of this language is its mobility from an API to another. It is also interesting to see that the Cg is the shading language of the RSX, the Playstation 3's graphics card.

The API chooses to compute fluid on the GPU is DirectX, it is not really a fully portable solution but the rendering to texture (RTT) is really simplified.

The GPU is not flexible like the CPU, as it is described in the first “GPU gem” of Nvidia, it is necessary to define some concepts to compute a fluid by the GPU.

The first aspect is about how the data are stored. By the CPU, it is a field, in other word, an array of object “vector”, by the GPU; it will be in a texture which information on velocity information stored in different channels. Then we draw a plane where this texture is applied. The fragment shader is executed and the result of computation is readable into another texture used for the render target.

Again, the debugging is not simple on a fluid; it became very difficult when is done by the GPU, it is not possible to put a breakpoint to execute the program step by step with shader. Another problem with GPU fluid is the accuracy. Indeed, velocity vector is stored in a texture, so, for each pixel there is 8 bits per channel (red, green, blue or alpha) therefore there is only 255 values possible, it is possible to store positive value of velocity and negative value separately. If by the CPU there is a float value with 32bit, on the GPU, it is just 8 bit. This limit of accuracy can be problematic when very small values are expected and are finally resulted by a zero.

The fluid has been well implemented by the CPU. A first implementation has been done by the GPU but it needs a lot of debug to have something stable. I will be interesting to implement a tool to compare the result from the GPU and from the CPU to identify errors.

5. Rendering Implementation

The first application to compute a fluid is very simple; vectors are drawing on a frame thanks to the QtPainter. It is not a fast display but to debug fluid classes, it is sufficient.

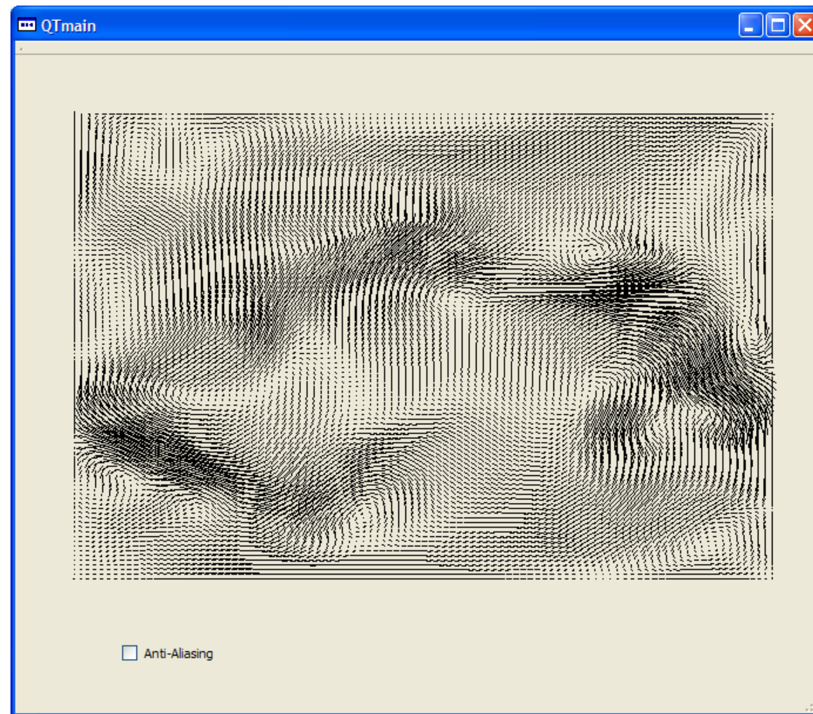


Figure 20 : First application

The raw vector field is not really visually interesting. As it is explain in the analysis part, there are a lot of techniques to render a vector field, it depends of the desired effect. In this project, tree main application has been done, the first one use only the OpenGL and displays the vortex, velocities or vectors or vector of the fluid with several parameters. The two other ones are more specific sample applications of a fluid simulation with some shader's effects.

a. OpenGL

To be logical with Qt portability, OpenGL is used for the display of the fluid. The result is draw on a texture update each frame, it could be fastest if the texture is created only when the fluid is modified. The texture is rendered with the filter linear, it is very important, because the fluid is often small and an interpolated result is expected.

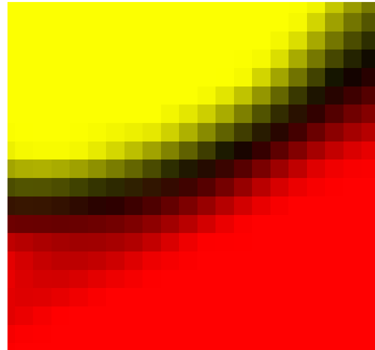


Figure 22: Nearest filter

Figure 21: Linear filter

The curl of the fluid is represented with two colours, one for positive vortex values and another for negative values. It is possible for the user of the application to modify colours in real time. The render of raw vectors is kept because it could be interesting to explain how the fluid is computed.

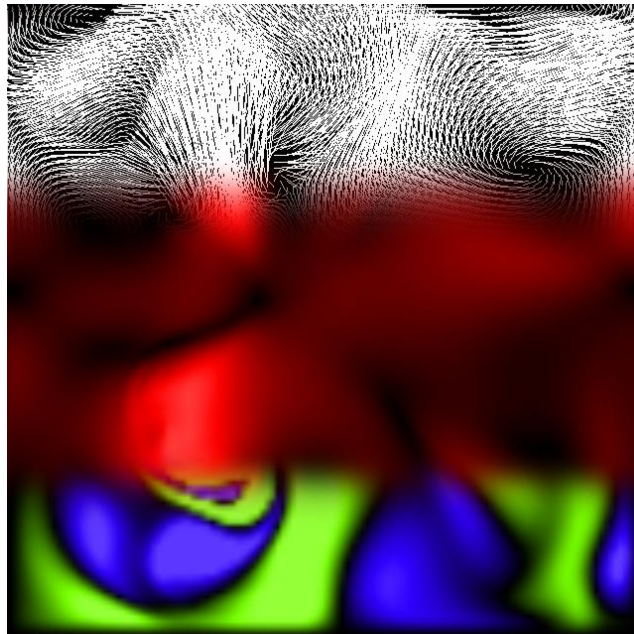


Figure 23: Mix of three renders

b. Shaders

Shaders are very useful in effects rendering, even if it is possible to compute a lot of algorithms with shader's language, the main goal of these programs is the rendering. The Cg can also be used with OpenGL, the Cg Toolkit of Nvidia facilitate the integration of shaders.

The fluid is sent to the shader in a texture but is already computed by the CPU, the red and green channels give information about x velocity and the two other, blue and alpha, the y velocity.

The Metaballs has been computed in the shaders, the positions of particles to represent the blob are stored, modified by the CPU and the movement is defined by the closest vector in the fluid. The position is given by two values for x and y between 0 and 1 because the actual position of the fragment computed is given thanks to texture coordinate defined on vertices which are defined between 0 and 1.

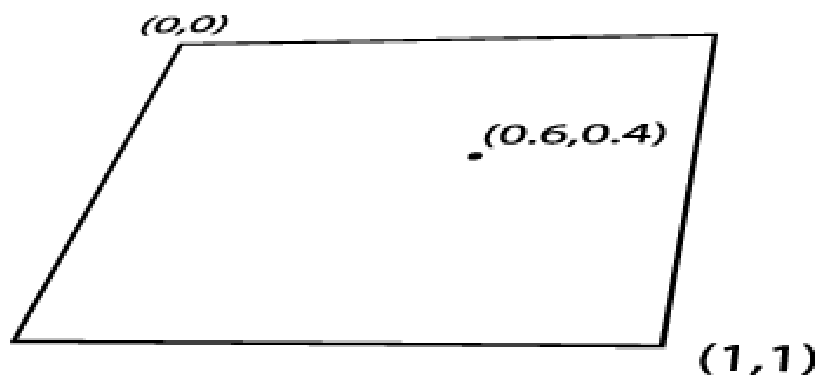


Figure 24: An idea of texture coordinate representation on a 2D plane

The simulation of a candle use the metaballs, in this effect, the mass of the particles, in other word, the value which define the size of influence, is decremented along the time. There is also attenuation on the side of the blobs to have a nicer effect. The information about particles is sent in a float array to the shader as a uniform value. The smoke generated by the candle burning is simulated thanks to the magnitude of the velocity.

Figure 25: Candle effect (with Penitent of Magdalen, LA County Art Museum)

Another effect applied on the fluid by shader's language is the bump mapping. The idea is not physically right. A displacement on the height of the fluid is defined thanks to his vortex which can be positive or negative. The simulation of the surface of a cup of coffee with a spoon mixing the liquid is the effect expected.

First, a cup of coffee is often circular, the implementation of obstacles become useful. Four obstacles close the cup. The area outside the circle is simply display with an alpha null checking the distance of the current fragment to the centre of the plane.

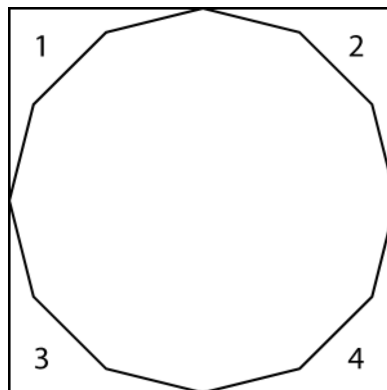


Figure 26 : The four obstacles to represent the cup of coffee

The foam of the coffee is represented by blobs. And finally, a bump mapping is added. This effect is not really finished; there are still some problems in the computation of vortex in the shader to resolve but the main goal of this computation is to give an idea of what is possible with fluid to simulate the surface of a liquid.

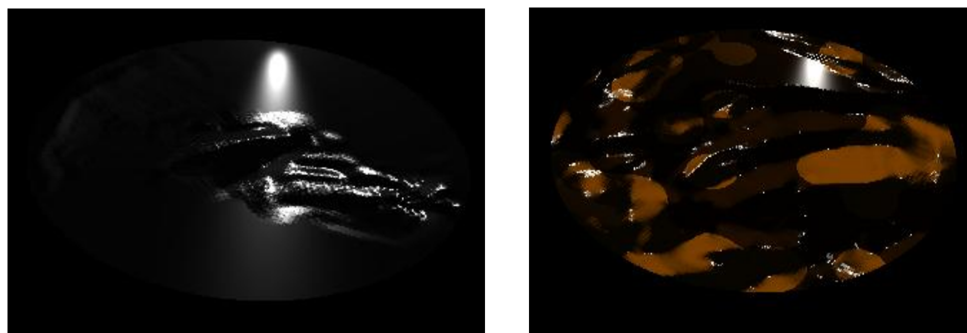


Figure 27: Surface with bump mapping from the vortex

Chapter 8 - Improvements

Even if, there is some leak this project is done but there is a lot possibility of improvements, perspective of development. This part is going to present a few ideas of how the development could become.

1. 3D Fluid

If a really correct and realistic or fluid simulation is expected, the fluid into a three dimensional space should be implemented. In this situation, the analysis could be worked again but Navier-Stokes equations are also correct in a 3D space.

Another aspect leads from 3D space in fluid simulation is the volume rendering. Indeed, if the fluid has a depth, the display is different; an accumulation of vertex beyond the fragment is necessary. The computation of the metaballs is not easy like in two dimensional spaces, it is possible to use marching cube but, it is often effective using the hardware (with shaders for the graphics card for example). The simulation of realistic fluid is also possible as it is explain in the third GPU gems.

In fact, the main problem in the computation of 3D fluid simulation is the real time, indeed, if a fluid with sixteen thousand vertices (128x128) is still relatively fast to compute, an hundred of these fluids (128x128x128) could be very slow on a computation on the CPU. The optimization specific of the hardware is more than necessary to have a real time and dynamic fluid simulation.

2. Non-Real time rendering

Another option of development could be to ignore the real time goal in Navier-Stokes equation. It could be very interesting to develop a plug-in for render and modeling software such as Maya or 3D studio max. A plug-in is an additional program of a main program to add functionalities. 3D studio max and Maya offer an Application Programming Interface (API) for the development of plug-in.

If the run time of computation of algorithm is not a problem anymore, it is possible to render fluid with a high resolution. Fluid in a three dimensional space computed by the CPU became possible. In fact, the main problem will be the memory consumption. Some plug-in to simulate effect like smoke or fire exist on 3D Studio Max, one of them is “FumeFx” from the society Stinisati. The render of the effect is precomputed and save in specific files but this plug-in use a lot of memory during its computation and it is problematic on limited configuration (can crash on complex effects).

Figure 28: Sample of render with FumeFx

3. CUDA

Finally, it will be interesting to present a hopeful technology. CUDA (for Compute Unified Device Architecture) is a technology from Nvidia which permit developers to use the C programming language to create program destined to be executed by the GPU. Actually, it is a technology essentially used by researchers; this technology is available on lastest Nvidia graphics card, the G8X series and most of Quadro series, these cards support 32bit floating point values. This technology is an anticipation of future requirements of gaming industry, the CUDA could speed up physical computation such as the fluid simulation.

The CUDA permit to code fast parallel algorithm using the graphic card, the access to data is considerably speed up and it doesn't need any API such as DirectX or OpenGL but it is possible to use one of them without interoperate with CUDA. This technology is compatible with multi-GPU system. Some mathematical libraries are included; the Fourier transform is implemented for example.

Furthermore, Nvidia is actually working on a debugger, it is very interesting because, today, the GPU development is not very easy to debug. There is another aspect very important with CUDA; this technology can be used for a CPU execution with the consideration of multi-core and multi-processors configurations. Intel also gives a particular compiler for its multi-core processors to optimise, but CUDA could be fully portable. It will be very interesting to develop an application and choose the hardware to execute it.

This technology is the most interesting perspective of development for this fluid simulation project. CUDA could be use for 3D fluid simulation and high definition fluid rendering; this technology has a promising future.

Chapter 9 - Conclusion

The fluid simulation is a complex subject. There are a lot of studies about it; I haven't look at all possibilities. I haven't presented all of these possibilities. The computing of the fluid simulation is still a research subject, some study try to explore the utilisation of the Fast Fourier Transform to speed up the computation of Navier-Stokes equations for example.

The main objectives have been successfully completed, the fluid has been implemented and it is fully portable. I can present and sell my work thanks to some demonstration application. I have especially acquired a lot of computing skills. I improve my knowledge about the organisation of an application and the ideal development process. I have acquired a lot of mathematical concept which are very common in general simulation like the resolution of a complex Poisson's equation with Jacobian or Gauss-Seidel method. Even if it wasn't the first time that I use QT, I have discovered several possibilities of this tool. Especially, I am now conversant with the computation of fluid simulation; the tool developed will permit to quickly implement a fluid simulation in another future project. One of my goals could be to design and set the fluid simulation to run on a really limited hardware such as a portable console, it will be an interesting challenge.

This project is not perfect and some parts are very draft, but, if I have to do again this kind of project, it will be really better, my experience has been improve. This final year project is rewarding, I am really satisfied of my work

.

References

J.Harris, M. (2006). *GPU gems.* NVidia.

JR, J. D. (1995). *Computational Fluid dynamic.*

Malgouyres, R. (2002). *Algorithmes pour la synthèse d'images et l'animation 3D.* Clermont-Ferrand: Dunod.

Mason, W., Jackie, N., Tom, D., & Dave, S. (2004). *OpenGL 1.4.* CampusPress.

math-linux. (2008). *math-linux*. Retrieved from math-linux: http://www.math-linux.com/

Stam, J. (2001). *A Simple Fluid Solver based on the FFT.* Seattle: Alias wavefront.

Stam, J. (2003). *Real-Time Fluid Dynamic for Games.* Toronto: Alias | wavefront.

Stam, J. (1999). *Stable Fluids.* SIGGRAPH.

Viscosity - Wikipedia. (2008, March 10). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Viscosity

Wikipedia. (2008, march 26). *Vortex*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Vortex

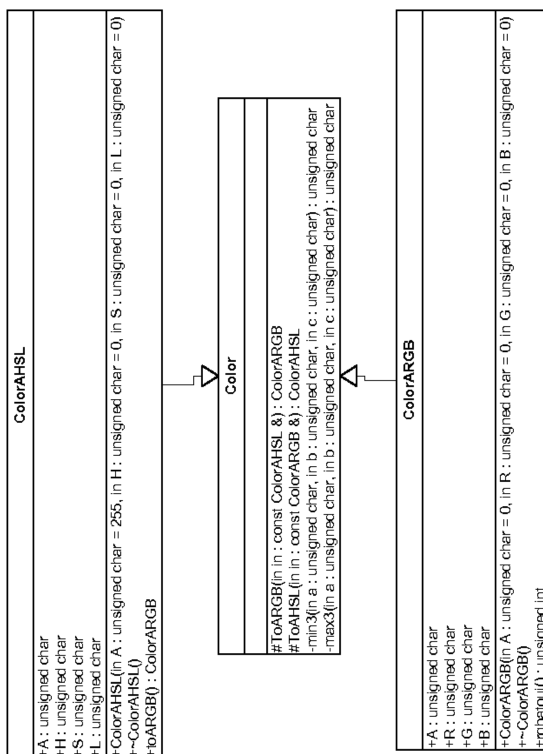
List of figures

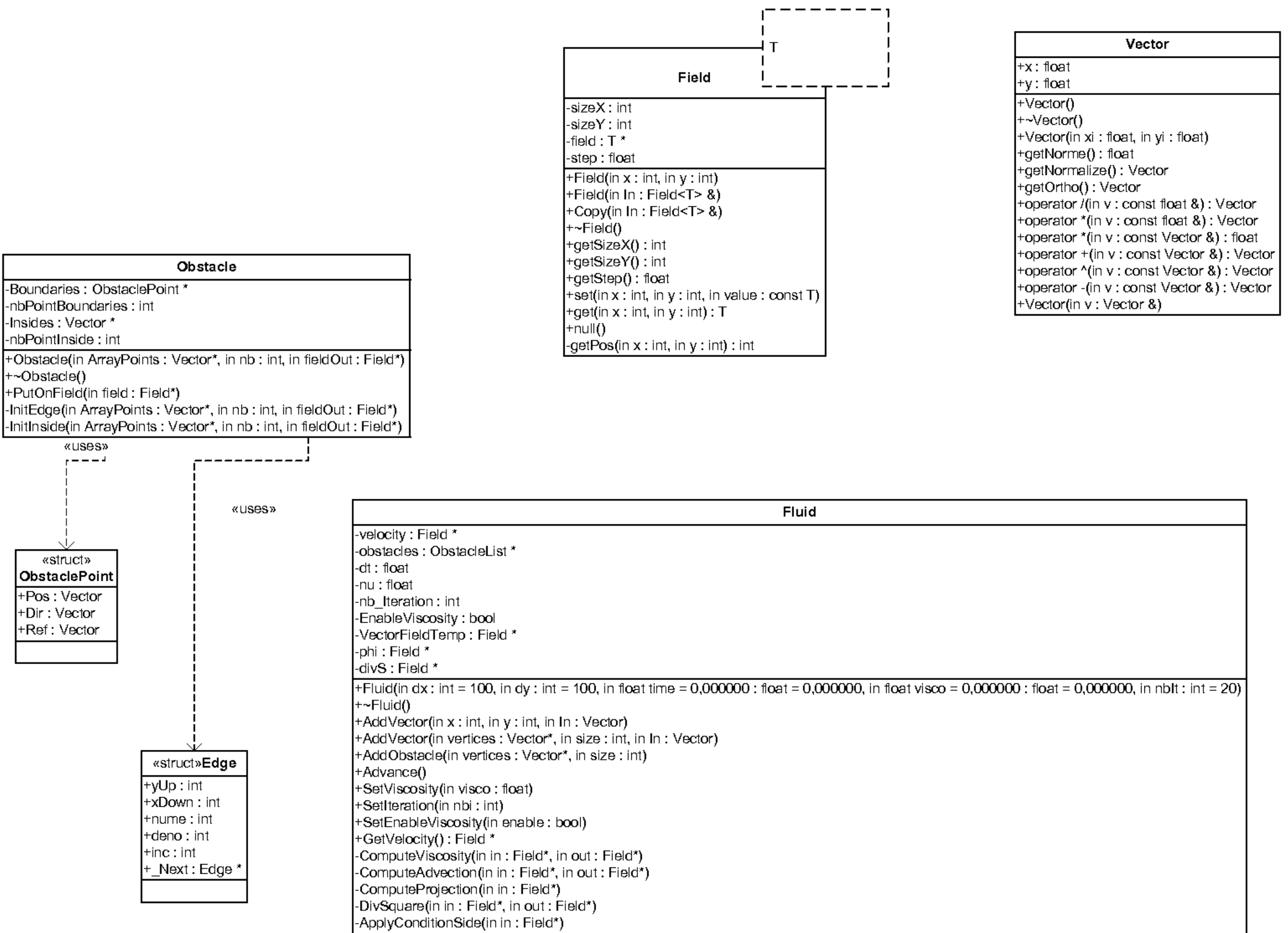
Figure	Page
Figure 1: Development process	9
Figure 2 : George Stokes	11
Figure 3 : Claude Navier	11
Figure 4 : Vector grid (from GPU gem)	12
Figure 5: The geometrical representation of the advection term	23
Figure 6 : Projection of a vector on a boundary	26
Figure 7 : The scalar Product	26
Figure 8 : The filling polygon technique used	28
Figure 9 : Vortex created by the passage of an aircraft wing (Wikipedia, 2008)	30
Figure 10 : two particles separating, setting a limit value to display edges	31
Figure 11 : 3D fluid from GPU gem 3	32
Figure 12 : Vector class	33
Figure 13 : Field template class	34
Figure 14 : Obstacle class and tools	35
Figure 15 : Fluid class	36
Figure 16 : Display classes	38
Figure 17 : Trolltech logo	39
Figure 18 : QT logo	39
Figure 19: the user interface created	41
Figure 20 : First application	47
Figure 21: Linear filter	48
Figure 22: Nearest filter	48
Figure 23: Mix of three renders	48
Figure 24: An idea of texture coordinate representation on a 2D plane	49
Figure 25: Candle effect (with Penitent Magdalen, LA County Art Museum)	50
Figure 26 : The four obstacles to represent the cup of coffee	51
Figure 27: Surface with bump mapping from the vortex	51
Figure 28: Sample of render with FumeFx	53

Appendix A – Schedule

ID	Task Name	Start	Finish	Duration
1	Research on Navier Stokes Equation	08/10/2007	30/11/2007	40d
2	Design Application for CPU	02/11/2007	29/11/2007	20d
3	Implement Vector Field for CPU	12/11/2007	17/12/2007	26d
4	Implement Diffusion Field for CPU	03/12/2007	28/12/2007	20d
5	CPU collision Algorithms and Design	19/12/2007	29/01/2008	30d
6	Implementation of GPU shader	31/12/2007	11/01/2008	10d
7	GPU implementation Algorithms and Design	16/01/2008	29/02/2008	33d
8	Fluid GPU collisions	04/02/2008	11/03/2008	27d
9	Report	07/02/2008	28/03/2008	37d

Appendix B – UML



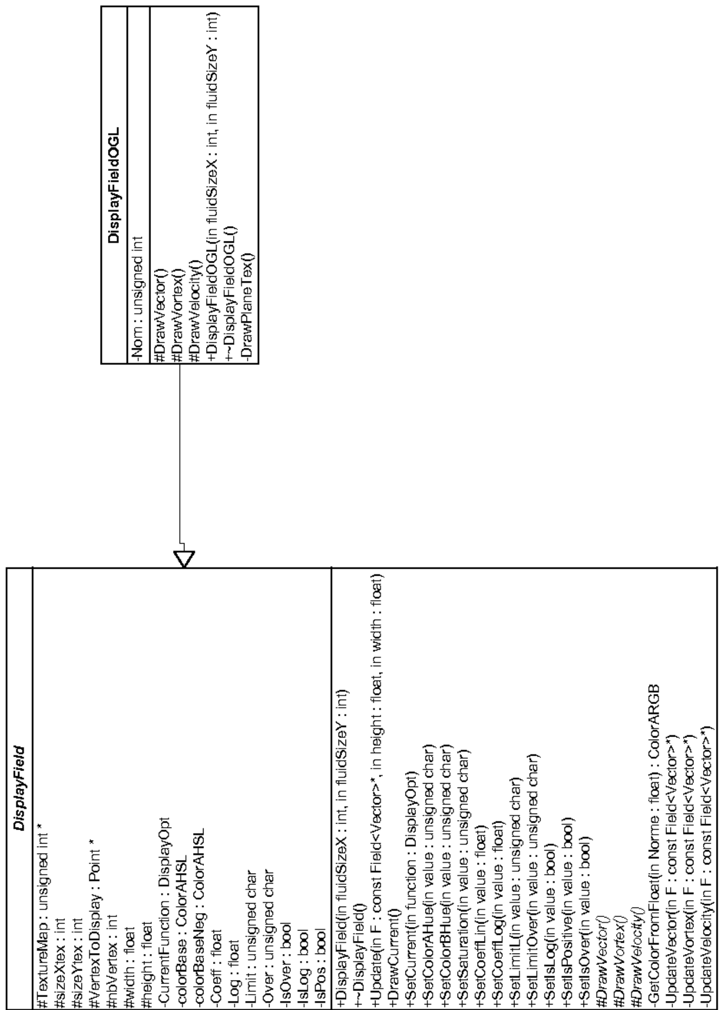


Obstacle
-Boundaries : ObstaclePoint *
-nbPointBoundaries : int
-Insides : Vector *
-nbPointInside : int
+Obstacle(in ArrayPoints : Vector*, in nb : int, in fieldOut : Field*)
+~Obstacle()
+PutOnField(in field : Field*)
-InitEdge(in ArrayPoints : Vector*, in nb : int, in fieldOut : Field*)
-InitInside(in ArrayPoints : Vector*, in nb : int, in fieldOut : Field*)
«uses»
«uses»
«struct»
ObstaclePoint
+Pos : Vector
+Dir : Vector
+Ref : Vector
«struct»Edge
+yUp : int
+xDown : int
+nume : int
+deno : int
+inc : int
+_Next : Edge *
Field
T
-sizeX : int
-sizeY : int
-field : T *
-step : float
+Field(in x : int, in y : int)
+Field(in In : Field<T> &)
+Copy(in In : Field<T> &)
+~Field()
+getSizeX() : int
+getSizeY() : int
+getStep() : float
+set(in x : int, in y : int, in value : const T)
+get(in x : int, in y : int) : T
+null()
-getPos(in x : int, in y : int) : int
Vector
+x : float
+y : float
+Vector()
+~Vector()
+Vector(in xi : float, in yi : float)
+getNorme() : float
+getNormalize() : Vector
+getOrtho() : Vector
+operator /(in v : const float &) : Vector
+operator *(in v : const float &) : Vector
+operator *(in v : const Vector &) : float
+operator +(in v : const Vector &) : Vector
+operator ^(in v : const Vector &) : Vector
+operator -(in v : const Vector &) : Vector
+Vector(in v : Vector &)
Fluid
-velocity : Field *
-obstacles : ObstacleList *
-dt : float
-nu : float
-nb_Iteration : int
-EnableViscosity : bool
-VectorFieldTemp : Field *
-phi : Field *
-divS : Field *
+Fluid(in dx : int = 100, in dy : int = 100, in float time = 0,000000 : float = 0,000000, in float visco = 0,000000 : float = 0,000000, in nbIt : int = 20)
+~Fluid()
+AddVector(in x : int, in y : int, in In : Vector)
+AddVector(in vertices : Vector*, in size : int, in In : Vector)
+AddObstacle(in vertices : Vector*, in size : int)
+Advance()
+SetViscosity(in visco : float)
+SetIteration(in nbi : int)
+SetEnableViscosity(in enable : bool)
+GetVelocity() : Field *
-ComputeViscosity(in in : Field*, in out : Field*)
-ComputeAdvection(in in : Field*, in out : Field*)
-ComputeProjection(in in : Field*)
-DivSquare(in in : Field*, in out : Field*)
-ApplyConditionSide(in in : Field*)

```
GLWidget
-fluid : Fluid *
-OldPos : QPoint
-timer : QTimer *
-timeElapsed : QTime *
-ComputeTimeUpdate : short
-lcdComputeTime : QLCDNumber *
-DisplayGL : DisplayFieldOGL *
+staticMetaObject : QMetaObject
+GLWidget(in parent : QWidget* = 0, in lcd : QLCDNumber* = 0)
+~GLWidget()
+SetFluidViscosity(in value : float)
+SetFluidEnableViscosity(in value : bool)
+SetFluidIteration(in value : int)
+SetCurrentDisplay(in value : DisplayOpt)
+SetColorADisplay(in value : unsigned char)
+SetColorBDisplay(in value : unsigned char)
+SetSaturationDisplay(in value : unsigned char)
+SetCoeffLinDisplay(in value : float)
+SetCoeffLogDisplay(in value : float)
+SetLimitLDisplay(in value : unsigned char)
+SetLimitOverDisplay(in value : unsigned char)
+SetIsLogDisplay(in value : bool)
+SetIsOver(in value : bool)
+SetIsPositive(in value : bool)
-Advance()
#initializeGL()
#paintGL()
#resizeGL(in width : int, in height : int)
#mouseMoveEvent(in event : QMouseEvent*)
#mouseReleaseEvent(in event : QMouseEvent*)
+qt_metacall(in _c : Call, in _id : int, in _a : void**) : int
+qt_metacast(in _clname : const char*) : void *
+metaObject() : const QMetaObject *
```

```
QTMain
-ui : QTMainClass
-glWidget : GLWidget *
+staticMetaObject : QMetaObject
+QTMain(in parent : QWidget* = 0, in flags : WFlags = 0)
+~QTMain()
-on_IsPositive_stateChanged(in Parameter1 : int)
-on_IsOver_stateChanged(in Parameter1 : int)
-on_IsLog_stateChanged(in Parameter1 : int)
-on_LimitOver_valueChanged(in Parameter1 : int)
-on_LimitLight_valueChanged(in Parameter1 : int)
-on_CoeffLog_valueChanged(in Parameter1 : int)
-on_CoeffLin_valueChanged(in Parameter1 : int)
-on_ColorSaturation_valueChanged(in Parameter1 : int)
-on_HueColorB_valueChanged(in Parameter1 : int)
-on_ComboBoxDisplay_currentIndexChanged(in Parameter1 : int)
-on_HueColorA_valueChanged(in Parameter1 : int)
-on_NumberIteration_valueChanged(in Parameter1 : int)
-on_ViscosityValue_valueChanged(in Parameter1 : double)
-on_viscosity_stateChanged(in Parameter1 : int)
+qt_metacall(in _c : Call, in _id : int, in _a : void**) : int
+qt_metacast(in _clname : const char*) : void *
+metaObject() : const QMetaObject *
```

```
Ui_QTMainClass
+centralWidget : QWidget *
+ComputeTime : QLCDNumber *
+StaticTextVelocity : QLabel *
+horizontalLayout : QWidget *
+hboxLayout : QHBoxLayout *
+ComputationSettingsBox : QGroupBox *
+viscosity : QCheckBox *
+StaticTextNumberIteration : QLabel *
+statictextViscosity : QLabel *
+ViscosityValue : QDoubleSpinBox *
+NumberIteration : QSpinBox *
+display : QGroupBox *
+labelModDisplay : QLabel *
+Coeffbox : QGroupBox *
+CoeffATxt : QLabel *
+LogTxt : QLabel *
+CoeffLog : QSlider *
+CoeffLin : QSlider *
+ComboBoxDisplay : QComboBox *
+ColorBox : QGroupBox *
+ColorATxt : QLabel *
+ColorATxt_2 : QLabel *
+label : QLabel *
+HueColorA : QSlider *
+HueColorB : QSlider *
+ColorSaturation : QSlider *
+IsLog : QCheckBox *
+IsOver : QCheckBox *
+IsPositive : QCheckBox *
+LimitsBox : QGroupBox *
+CoeffATxt_3 : QLabel *
+OverTxt : QLabel *
+LimitLight : QSlider *
+LimitOver : QSlider *
+ModEdit : QGroupBox *
+Temporary : QLabel *
+statusBar : QStatusBar *
+setupUi(in QTMainClass : QMainWindow*)
+retranslateUi(in QTMainClass : QMainWindow*)
```



DisplayFieldOGL
-Nom : unsigned int
#DrawVector()
#DrawVortex()
#DrawVelocity()
+DisplayFieldOGL(in fluidSizeX : int, in fluidSizeY : int)
+~DisplayFieldOGL()
-DrawPlaneTex()
DisplayField
#TextureMap : unsigned int *
#sizeXtex : int
#sizeYtex : int
#VertexToDisplay : Point *
#nbVertex : int
#width : float
#height : float
-CurrentFunction : DisplayOpt
-colorBase : ColorAHSL
-colorBaseNeg : ColorAHSL
-Coeff : float
-Log : float
-Limit : unsigned char
-Over : unsigned char
-IsOver : bool
-IsLog : bool
-IsPos : bool
+DisplayField(in fluidSizeX : int, in fluidSizeY : int)
+~DisplayField()
+Update(in F : const Field<Vector>*, in height : float, in width : float)
+DrawCurrent()
+SetCurrent(in function : DisplayOpt)
+SetColorAHue(in value : unsigned char)
+SetColorBHue(in value : unsigned char)
+SetSaturation(in value : unsigned char)
+SetCoeffLin(in value : float)
+SetCoeffLog(in value : float)
+SetLimitL(in value : unsigned char)
+SetLimitOver(in value : unsigned char)
+SetIsLog(in value : bool)
+SetIsPositive(in value : bool)
+SetIsOver(in value : bool)
#DrawVector()
#DrawVortex()
#DrawVelocity()
-GetColorFromFloat(in Norme : float) : ColorARGB
-UpdateVector(in F : const Field<Vector>*)
-UpdateVortex(in F : const Field<Vector>*)
-UpdateVelocity(in F : const Field<Vector>*)