TECHNISCHE UNIVERSITÄT MÜNCHEN

An Interactive Fluid Dynamics Game on the iPhone

Master's Thesis

by

Hosam Hassan

1st examiner: Jun.-Prof. Dr Michael Bader
2nd examiner: Prof. Dr Bernd Brügge

Computational Science and Engineering (Int. Master's Program)

Informatics Department

Thesis handed on: 18th of December, 2009

Declaration of Authorship

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Signed: ____

Date: ____

“Every once in a while a revolutionary product comes along that changes everything. It’s very fortunate if you can work on just one of these in your career.”

Steve Jobs

TECHNISCHE UNIVERSITÄT MÜNCHEN

Abstract

Informatics Department

Master of Science

by Hosam Hassan

In this thesis work we will explain how to solve a computational problem on a small scale with real world limitations. The paper will describe the necessary steps to implement a computational fluid dynamics simulation engine on the iPhone for simulating incompressible fluids. The numerical model behind the fluid engine will be explained briefly and the software architecture for the engine will be discussed in details. A fluid visualization engine will be developed from the scratch and could be easily extended to describe different fluids. The performance optimization process is a corner stone of any real time fluid simulation, specially when it is done on a handheld device with limited processing power and memory. The CFD engine and the visualization engine performance will be analyzed using a profiling tool and the computing extensive parts of the code will be optimized. The performance optimization steps and techniques will be explained in details with code examples. In the end of the paper will see our engine being used in different fluid simulation scenarios and the limitations of the engine will be explained. Important issues in Software Engineering and in Computational Fluid Dynamics will be discussed side by side to help enforce the bridge between both fields and emphasis on the importance of Software Engineering in the field Scientific Computing.

Acknowledgements

I would like to thank Prof. Michael Bader for his extensive help and supervision throughout the thesis and Prof. Bernd Brügge for agreeing and encouraging me to implement my idea. I would like express my gratitude to my brother Asem Hassan for encouraging me to learn about the iPhone development and for buying me my first ever Mac Machine. I would like to thank my parents and girlfriend for their continuous support and encouragement throughout my thesis work.

Contents

Declaration of Authorship **i**

Abstract **iii**

Acknowledgements **iv**

List of Figures **vii**

Abbreviations **viii**

1 Introduction **1**
1.1 Problem Definition . . . 1
1.2 Overview . . . 2
1.3 Challenges . . . 6

2 Background and Previous work **7**
2.1 Overview . . . 7
2.2 Mathematical Model . . . 7
2.3 Numerical solution . . . 9
2.3.1 Obstacles Handling . . . 10

3 Software Architecture and Implementation **13**
3.1 Overview . . . 13
3.2 Software Architecture . . . 13
3.3 Model . . . 15
3.3.1 Model Design . . . 15
3.3.1.1 Core CFD . . . 17
3.3.1.2 Core CFD Manager . . . 17
3.3.1.3 Simulation Result . . . 19
3.3.1.4 Simulation Engine . . . 20
3.3.1.5 Obstacle Handler . . . 20
3.3.1.6 Obstacle . . . 21
3.4 View . . . 21
3.4.1 View Design . . . 21
3.4.1.1 Geometry Mapper . . . 23
3.4.1.2 Color Mapper . . . 23

3.4.1.3 Renderer . . . 23
3.4.1.4 EAGLView . . . 24
3.5 Controller . . . 24

4 Visualization 26
4.1 Definition . . . 26
4.2 Mapping . . . 28
4.2.1 Geometry Mapping . . . 29
4.2.2 Color Mapping . . . 31
4.2.2.1 Velocity to RGB Color Mapping . . . 31
4.2.2.2 Rendering . . . 33

5 Performance 37
5.1 Overview . . . 37
5.2 Hardware . . . 38
5.3 Code Profiling . . . 39
5.4 Memory Allocation and Memory Leaks . . . 40
5.5 CFD Core Engine optimization . . . 40
5.6 Performance optimization gain . . . 43

6 Test Cases 44
6.1 Non-interactive Simulations . . . 44
6.1.1 Flow over a step . . . 45
6.1.2 Karman Vortex Street . . . 46
6.2 Interactive Simulations . . . 47
6.2.1 Insert an Obstacle on touch . . . 47
6.2.2 Add Velocity on touch . . . 51

List of Figures

1.1 The simulation pipeline . . . 3
1.2 Computational steering problem . . . 4

2.1 The cell flag field . . . 11
2.2 The velocity and pressure values in the obstacle cell . . . 12

3.1 Model View Controller . . . 14
3.2 Model architecture overview . . . 16
3.3 View architecture overview . . . 22

4.1 The visualization process . . . 27
4.2 Mapping of the fluid grid to the screen . . . 28
4.3 Screen axis translation . . . 29
4.4 Screen axis rotation . . . 30
4.5 Screen axis scaling . . . 30
4.6 Color mapping . . . 31
4.7 Color Ramp . . . 32
4.8 The needed triangles to cover up the whole fluid domain . . . 33
4.9 One triangle strip . . . 34
4.10 Two triangle strips . . . 35
4.11 The rendering process . . . 36

5.1 Instruments screenshot . . . 39

6.1 Simulation running for the step in a fluid scenario on the iPhone . . . 45
6.2 The Karman Vortex Street simulation running on the iPhone . . . 46
6.3 Circle shaped obstacle . . . 48
6.4 Three finger touch . . . 49
6.5 Two finger swiping . . . 49

Abbreviations

CFD	**C**omputational **F**luid **D**ynamics
SOR	**S**uccessive **O**ver **R**elaxation
VTK	The **V**isualization **T**ool**K**it
MVC	**M**odel **V**iew **C**ontroller
App	**App**lication
RGB	**R**ed **G**reen **B**lue
API	**A**pplication **P**rogramming **I**nterface

Chapter 1

Introduction

1.1 Problem Definition

The paper explains how to implement and design a high performance real-time interactive computational fluid dynamics engine that will be able to run on a handheld device with limited computing power and memory. The engine could be used in games to model any type of incompressible fluid flow. An API will be carefully designed to construct an abstract solid and easy to use representation for the simulation engine. Game developers with basic CFD knowledge will be able to customize and integrate the engine easily in their games. The simulation engine will be designed following a strict software engineering practices to enforce usability, maintainability, reusability and extendability. The CFD engine will be built above a code previously written by me during the Computational Fluid Dynamics Practical course [1] [2][3]. The previous code lacked a proper design and performance optimization was ignored. No object oriented practices were enforced making it very complex act as a standalone fluid simulation engine. Further more, the code did not provide any possible way to implement an interactive computational steering scenarios, where the user is allowed to interfere and change the problem parameters during the execution of the simulation.

The second important part of the problem is how to visualize the simulation results once the simulation engine returns the computed values of the fluid. Different algorithms are applied on the computed values to convert them into meaningful colors for the user,

making the fluid flow easier to be interpreted and understood. The visualization engine is done from the scratch and without relying on any kind of previous code.

The last part of the problem is performance optimization for the simulation engine and the visualization engine since the code will have to run and respond in real time to the user gestures. Different performance practices will be applied on the code and different tools will be used to identify the most intensive parts of the code.

1.2 Overview

In the thesis we will follow the classical simulation pipeline module shown in figure 1.1. The first step is breaking down the physical phenomena into mathematical equations which in our case it is mainly the Navier-Stokes equations and the continuity equations. After this we convert all the continuos equations into a discrete model in a process known as discretization which in turn is usually solved using iterative solvers. The software development cycle starts after this starting from laying down the requirements and designing the software. After the design is done the implementation or coding the solution started. The final program is executed and the results are visualized in meaningful colors or shapes presented to the user. Based on the visualization and the numbers, the whole process could be repeated and tuned till we reach the desired results.

The two show cases or scenarios for the App are examples for a computational steering process. In a normal fluid simulation the simulation runs without any intervening or feedback from the user, like for example simulating the flow of air in a tunnel or simulating clouds movement. The problem with normal fluid simulation that it is time consuming and does not allow the user to alter the problem parameters or the fluid domain. If you are simulating a cylinder in a pipe with a fluid flow and then you want to adjust the diameter or location of the cylinder. In a none computational steering problem you will have to stop the simulation, adjust the cylinder dimensions or location and re-run the simulation. On the other hand, the computational steering is a concept that allows you to interact with the problem domain and actually change the problem parameters or boundaries in real time. Such system can be very challenging but rewarding [4]. Providing such interactivity based on the user feedback can increase rapidly the possibility of detecting and correcting any errors. As a result, the engineering and design

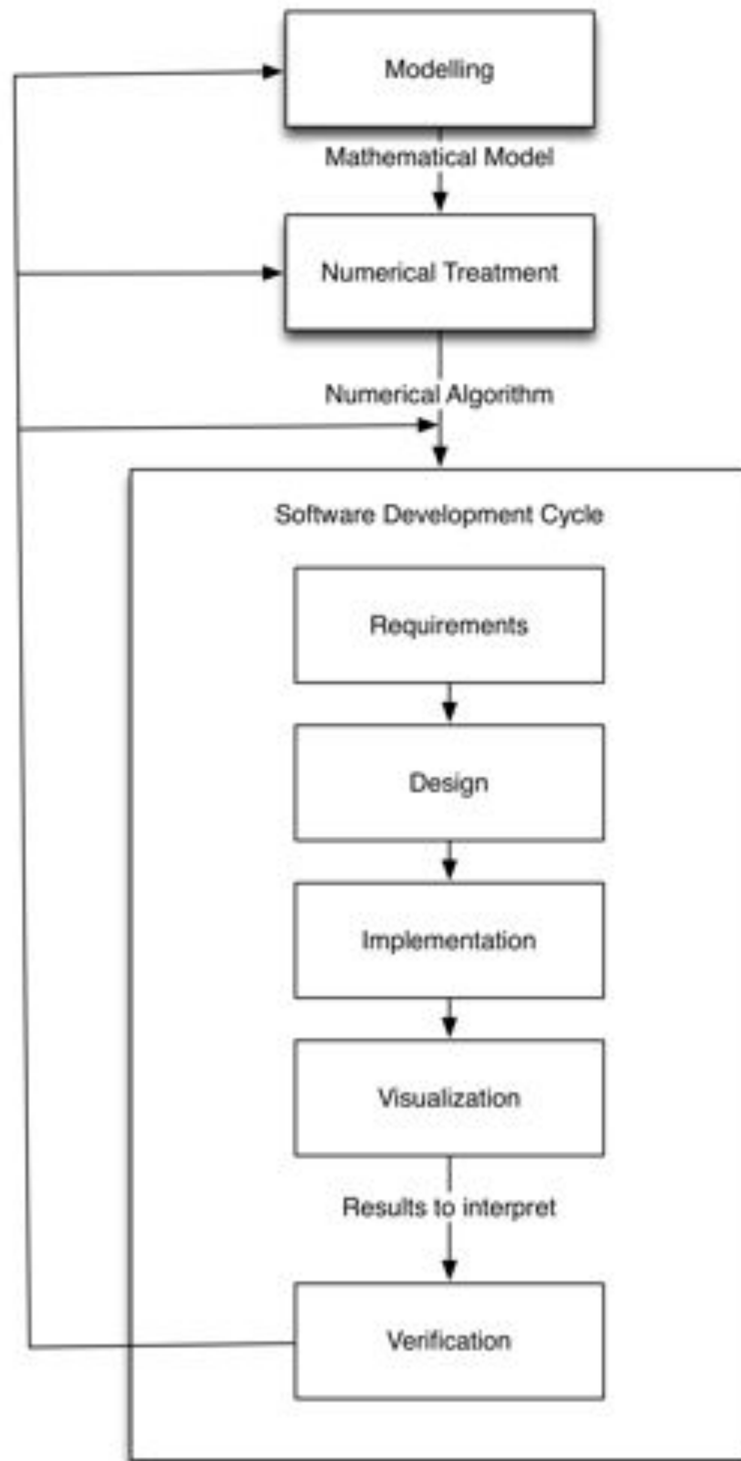

FIGURE 1.1: The Classical Simulation Pipeline showing all the different stages in the simulation process

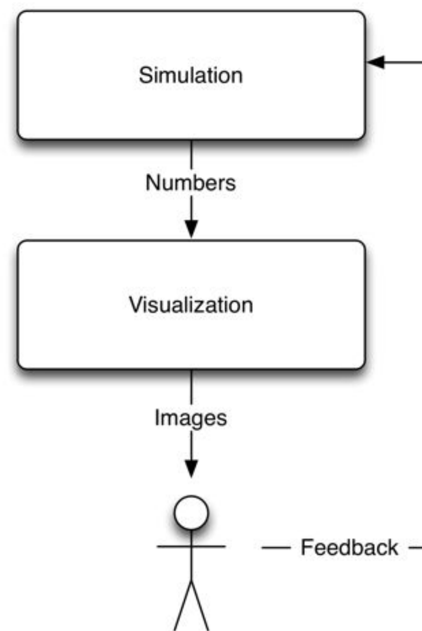

FIGURE 1.2: A simple scenario for a computational steering system

of the simulation core should allow real-time problem parameters manipulation without stopping the simulation. Figure 1.2 shows a simplified components in a computational steering problem and how they are interacting together.

In this thesis we are building above an already existing solution which is mainly the modeling and numerical treatment steps. In chapter 2 we will explain the main features of the CFD Lab code [3] which already existed and the main problems with it. A brief numerical and mathematical background is given to explain the main features and limitations of the code.

In chapter 3 we will start by explaining how the Core Code have been re engineered and designed to provide a solid reusable CFD Engine with an API written in Objective-C [5].

In chapter 4 we will explain how the visualization process was implemented and the techniques used in rendering the fluid flow into a colored image. Mapping the fluid domain to the screen coordinates will be explained together with detailed steps how to change the simulation result into a colored image that could be displayed on the iPhone screen.

In chapter 5 we will continue with one of the corner stones of this paper which is performance optimization for our Application and discuss the different practices which needs to be tackled to boost the performance of any real time simulation. A comparison will be made between the execution speed before and after optimization to determine the performance gain we obtained.

We will finish up by explaining how the engine how be using three examples which serves as examples for what kind of fluid simulations are possible using the developed engine. The first two show cases is based on the classical simulation pipeline where the user does not intervene with the simulation or change the problem parameters during the numerical computations. The last show case is a computational steering problem where the model needs to adjust itself based on the user input, and the visualization results is changed immediately to reflect the impact of the user actions. In this case the user has some control on the simulation process and can influence its execution.

1.3 Challenges

Real time Application

This is the most important aspect since it is a computational steering problem where the user feedback and interaction is a key feature in fluid simulation. The user actions should be propagated to the simulation and new results should be computed based on the user feedback. Once the user request is processed the new data should be available and displayed. The whole process of interaction and see the results should be done in a smooth transition to have a realtime process. The numerical modeling and the software architecture will have to have to be optimized when dealing with a problem of real time constraints.

Limited Processing power

Like any computational problem we are always limited by a certain processing power capabilities. The simulation engine should be able to run in real-time on a handheld device with very limited computing power making performance a very important aspect of the thesis. The limited processing power is a direct result of the need to a real time performance together with the limited hardware. For example simulating a huge grid can take days on a small device.

Memory limitations

When implementing any kind of application on a handheld device memory management is a very important aspect. Poor memory management can result in performance degrading and eventually lead to crash the execution of the application. Allocating and releasing memory will have to be correctly addressed. Memory leaks will have to be detected using specialized tools.

Chapter 2

Background and Previous work

2.1 Overview

The core CFD engine is based on the computational fluid dynamics lab course taught at the scientific computing chair [6]. It is recommended to read the two worksheets [2] and [3] for a more detailed description for the numerical scheme and fluid modeling which is defining the core of the CFD engine. In this chapter we will give a brief overview about the basics of how the fluid flow is being modeled and what type of physical equations are used to describe the fluid flow. An explanation about the numerical model as well as an abstract algorithm for the simulation process will be explained. The limitations and the type of possible fluid simulation that could be done with the engine is explained.

The core engine is discussing how to model the flow of an incompressible viscous fluid like water, where the the mass of the fluid can occupy only a constant volume. On the other hand; compressible fluid like air, the same mass of fluid can occupy different volumes. Viscosity explains the resistance of the fluid to flow or in a simpler words, it defines how easy the fluid can move. For example, honey has high viscosity while water has a very low viscosity.

2.2 Mathematical Model

The famous Navier-Stokes equations are used to describe the fluid flow. The Navier-Stokes equations consists of the two momentum equations 2.1 and 2.2. This form of the

Navier-Stokes equations shown below could be used to explain the flow of a fluid in two dimensions [1].

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}(\frac{\partial u^2}{\partial x^2} + \frac{\partial u^2}{\partial y^2}) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x \quad (2.1)$$

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}(\frac{\partial v^2}{\partial x^2} + \frac{\partial v^2}{\partial y^2}) - \frac{\partial (v^2)}{\partial y} - \frac{\partial (uv)}{\partial x} + g_y \quad (2.2)$$

Both momentum equations together with the continuity equation 2.3 describes the flow of the fluid.

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \quad (2.3)$$

Where

- **u**: The horizontal components of the fluid velocity.
- **v**: The vertical components of the fluid velocity.
- **p**: Pressure
- **Re**: Reynolds number
- $\mathbf{g}_x$ and $\mathbf{g}_y$: External forces in the horizontal and vertical direction affecting the fluid like gravitational force for example.

Together with the Navier-Stokes equations 2.1 and 2.2 we need initial conditions and boundary conditions. Initial conditions include the initial horizontal and vertical velocities, and the initial pressure. Boundary conditions are given on the four walls or boundaries, the core code provides four possible boundary conditions to choose between [1].

- **No-slip**: Any fluid flowing parallel to this boundary will be zero.
- **Free-slip**: The fluid can flow freely parallel to the boundary without friction.
- **Inflow**: Fluid is flowing in at the boundary.
- **Outflow**: Fluid is flowing out from the boundary.

2.3 Numerical solution

In order to solve the Navier-Stokes equations we will need to break the continuos momentum equations to a discrete form that could be solved using iterative numerical methods.

We will start by decomposing our domain into a two dimensional grid of discrete cells. The grid will have a specific size in the x direction and in the y direction. The bigger the grid, the more accurate the results are and the more computations needed. The more the computations the slower the simulation which is a critical thing in real-time applications therefore we will try to find a suitable grid size that gives a good solution and fast response time.

Boundary conditions and values are enforced at the domain boundary. After composing our fluid domain into a 2D grid of fluid points, we need to split the time evolution of the fluid into discrete time steps. The fluid mathematical equations are solved per time step based on the previous time step to determine how the fluid will evolve.

After decomposing our fluid domain into a discrete grid points. We will need to do the spatial discretization in the momentum equations and in the continuity equation. Momentum equation 2.1 is calculated at the vertical cell edge midpoints, Momentum equation 2.2 is calculated at the horizontal cell edge midpoints and the continuity equation 2.3 is calculated at the midpoint of the cell. The spatial discretization of the second derivative of u is given by [1]:

$$\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} := \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\partial x)^2} \tag{2.4}$$

The continuity equation 2.3 discretization is given below.

$$\left[\frac{\partial u}{\partial x}\right]_{i,j} := \frac{u_{i,j} - u_{i-1,j}}{\partial x} \qquad \left[\frac{\partial v}{\partial y}\right]_{i,j} := \frac{v_{i,j} - v_{i,,j-1}}{\partial y} \tag{2.5}$$

The time discretization for the momentum equations is done using the explicit Euler Method

$$v_{i,j}^{(n+1)} = G_{i,j}^{(n)} - \frac{\delta t}{\delta y}(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}) \tag{2.6}$$

$u_{i,j}{}^{(n+1)}$ is calculated in the same way using $F_{i,j}{}^{(n)}$

Where i and j are the coordinates of the midpoints of cell(i,j) where i = 1,..imax and j = 1,...jmax. $F_{i,j}{}^{(n)}$ and $G_{i,j}{}^{(n)}$ holds the discretized right hand sides of the equations 2.1 and 2.2. The dimensions of the grid in the horizontal and vertical direction are imax and jmax respectively.

In order to solve the pressure term $p^{(n+1)}$ in equation 2.6 we will use the discretized Poisson equation for the term $p^{(n+1)}$

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} = \frac{1}{\delta t}\left(\frac{F_{i,j}^n - F_{i-1,j}^n}{\delta x} + \frac{G_{i,j}^n - G_{i,j-1}^n}{\delta y}\right) \tag{2.7}$$

which is equivalent to

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} \tag{2.8}$$

The discretized Poisson equation 2.7 is solved using an iterative SOR solver. The fluid solver is where most of the intensive computations are done and it is one of the main time consuming calculations.

The number of iterations done on the the SOR solver is controlled using two variables. A predefined maximum of iterations and a tolerance value. After each iteration we calculate the residual error and compare it to the tolerance. If the residual is smaller than the tolerance or we reach the maximum number of iterations we stop iterating on the SOR solver. In our solution we will only use the iterations number and ignore the residual calculations. This will result in better performance but less accuracy for the fluid solver. Accuracy can be increase by increasing the number of iterations.

2.3.1 Obstacles Handling

Obstacles handling and representation is an important aspect for the interactivity part of the CFD engine, therefore we will briefly explain here how the obstacle handling is done. Each grid cell in the grid is classified as either a fluid cell or as an obstacle cell. Each cell holds a flag field which contains the information about the types of the surrounding four cells wither they are fluid cell or obstacle cells. The flag is represented as an integer where only the last five bits are used. Each bit can be either 0 to indicate

an obstacle cell present at this position or 1 to indicate a fluid cell. Some combinations are not allowed and tests are enforced to prevent such cases like the case of a forbidden boundary cell which will be defined later in this section.

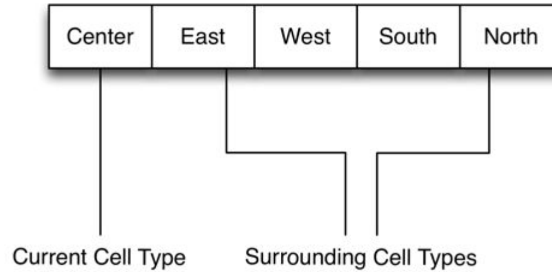

FIGURE 2.1: The figure shows the distribution of the flag field bits

The first bit indicates the cell type at the specific position while the other four bits indicate the four surrounding cell types as shown in figure 2.1

For example an integer value of 16 which is 10000 in binary will define a fluid cell and a value of 0 which is 00000 in binary will define an obstacle cell. In order for stable and accuracy computations we need as well to differentiate between the cells on the obstacle boundary and the cells on the boundary of the obstacle. Obstacles cells that have fluid neighbors are called the obstacle boundary cells for example a value of 01010 will define an obstacle value with two fluid cells present in the east and south directions. Using such structure for the flag field a static types of cells is constructed and this binary representation is totally hidden on a higher level. For example, a cell type of value C_F is used to indicate a fluid cell and cell type of value C_BE is used to indicate an obstacle cell with a fluid cell present at its east direction [3].

Some flag combinations are forbidden, like an obstacle cell surrounded by fluid cells from the four directions. This is the list of the main cell types relevant to our implementation and their definitions based on their position in the fluid and obstacle.

- **Fluid Cell**: Cell inside the fluid domain
- **Obstacle Cell**: Cell inside the obstacle domain
- **Forbidden boundary Cell**: Cell that has four fluid neighbors, this can't be allowed

The boundary values for the obstacle boundary cells are given by a set of equations. As an example, in order to set the boundary values on the north edge for an obstacle cell with only fluid cell in its north for a cell at grid coordinate i and j [3]

$$v_{i,j}=0, \quad u_{i-1,j}=-u_{i-1,j+1}, \quad u_{i,j}=-u_{i,j+1}, \quad G_{i,j}=v_{i,j}, \quad p_{i,j}=p_{i,j+1} \tag{2.9}$$

while for a boundary cell with only one fluid cell in its western direction. The boundary values are given by [3]

$$u_{i-1,j}=0, \quad v_{i,j-1}=\text{-}v_{i-1,j-1}, \quad v_{i,j}=-v_{i-1,j}, \quad F_{i-1,j}=u_{i-1,j}, \quad p_{i,j}=p_{i-1,j} \tag{2.10}$$

Where u is the horizontal velocity, v is the vertical velocity, p is the pressure value and F, G holds the discretized right-hand sides of the two momentum equations 2.1 and 2.2 at the given grid point coordinates. The positions of the velocities and the pressure value in obstacle cells is shown in figure 2.2

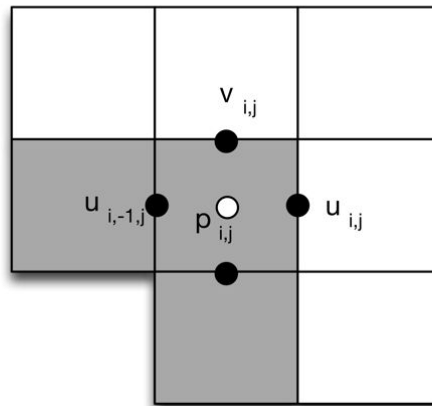

FIGURE 2.2: The figure shows the position of the calculated velocities and pressure where the while cells indicate fluid cells and grey indicates obstacle cells

Chapter 3

Software Architecture and Implementation

3.1 Overview

In this chapter we will explain about the general architecture for any fluid simulation application built using the CFD Engine. We start by explaining the Model View Controller design pattern and how it will be mapped with our architecture. The design and components of each of the model, view and controller will be explained in details. The properties or parameters of the objects will be listed. The communication and data exchange between the components will be discussed. The components responsible for the fluid visualization will be explained briefly and the in depth analysis for the visualization techniques and algorithms will be handled in chapter 4.

3.2 Software Architecture

Model View Controller

The model view controller is the main design pattern used in iPhone Applications and most of the Cocoa applications in general [7], therefore we will start by explaining the concept behind the pattern and how it will perfectly fits to application design. The MVC pattern separates the application logic and objects into three tiers or layers known as

the model, view and controller shown in figure 3.1 [7]. Not only the pattern separates the objects and components but it also defines how the objects will communicate and exchange data.

Each layer holds a collection of objects of similar functionality or classification closely. Adhering to the MVC pattern has massive benefits like maintainability, reusability and extendability. Further more, the pattern is somehow enforced and highly encouraged by Apple for iPhone applications, therefore we will stick to the pattern throughout the design process. In the next sections we will explain each layer in the MVC pattern and the components of each layer.

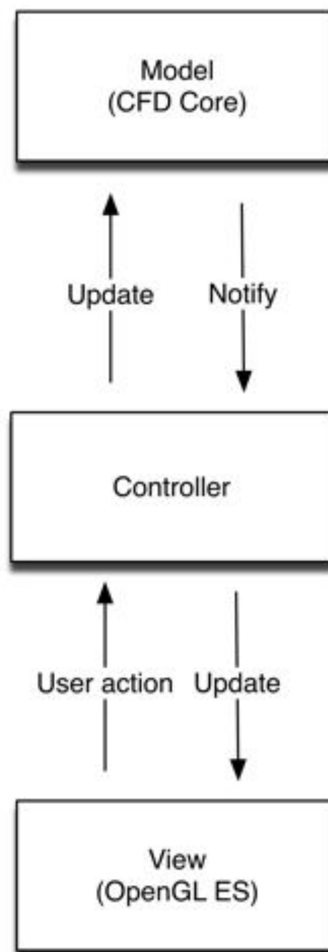

FIGURE 3.1: The MVC design pattern used in all the iPhone Apps

3.3 Model

The model layer encapsulates all the data objects used in the application and it is where all our simulation code and computations resides. Further more, the layer defines how the simulation results are communicated after the simulation is done. Inside the model, objects can communicate with each other to exchange data or information. The persistent stat of the application resides in the model layer once the application is initialized. The layer holds all the CFD core simulation computations and data. The model provides an easy to access API that allows for abstract representation of the model and makes it easier to port to other platforms. The core CFD Engine is written in C and the model API is written in Objective-C. The model is totally separated from visualization components and from the interface design. This makes reusing the CFD Engine in any Mac OS X application or iPhone App straight forward. Communication in the model with the view is handled through the controller which is responsible for passing over the messages between the two layers. The model receives commands from the controller, change its state and respond back to the controller to inform it with the changes. For example, the controller will send a command to the model to execute a specific number of simulation time steps. Once the model finishes the computations it will send a notification to the controller, and hands over the results of the simulation. The controller receives the notification, do the necessary computations and decides how to update the interface or inform the user of the new changes in the model.

3.3.1 Model Design

The model is the CFD core together with an easy to use API with which the communication with the CFD engine is done. The previous code developed in the CFD Lab course [6] has ignored most of the software engineering practices including inheritance, usability and maintainability making it extremely hard to serve as a solid CFD Engine.

Rewriting the code or refactoring it would have taken a very long time and unnecessary effort. My approach was to build above the current code an understandable and easy to use API, which could serve as the front end for the previous CFD core code. Both the API and the core form the CFD Engine of the App as shown in figure 3.2.

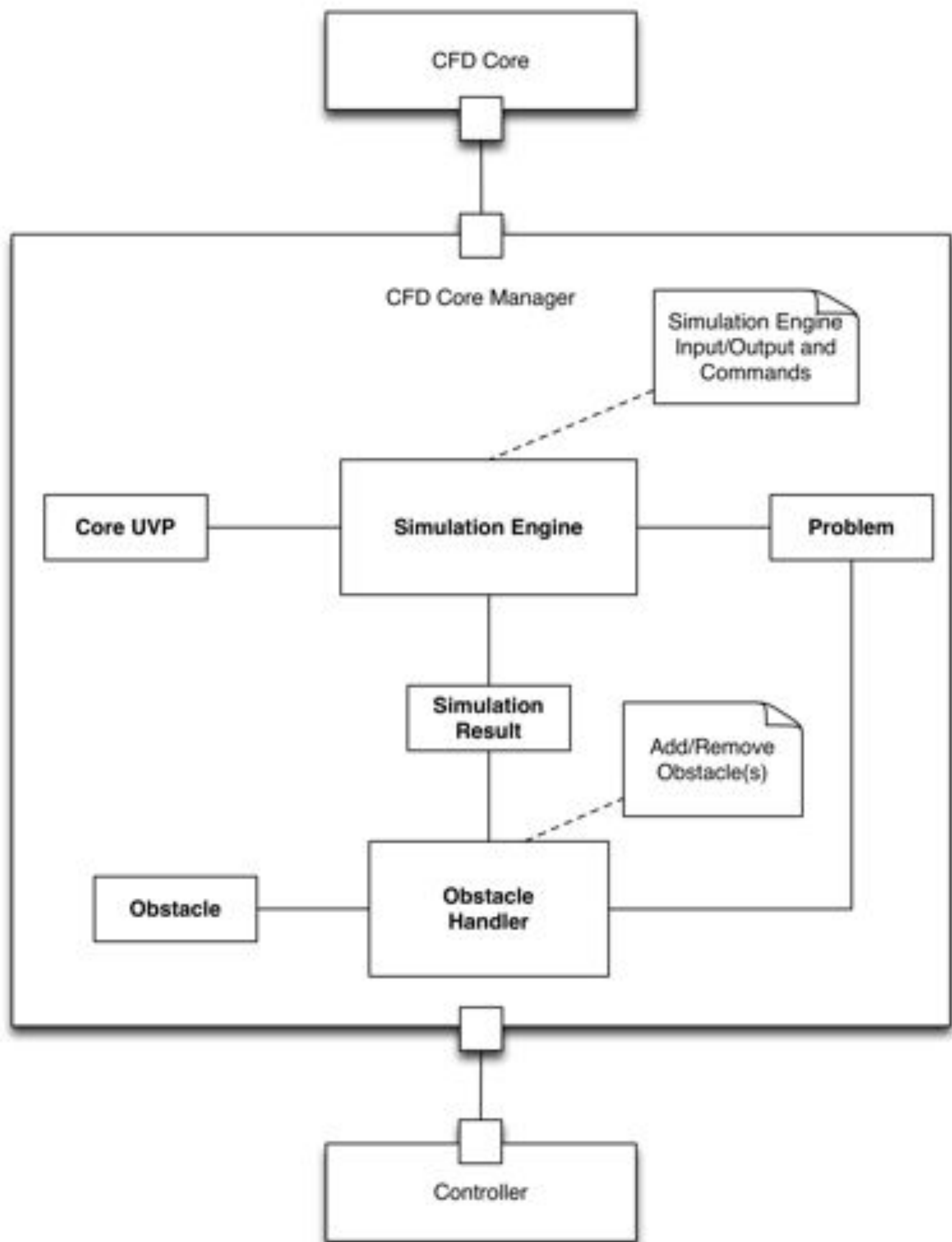

FIGURE 3.2: The figure shows the design for the model components and how it's connected to the Controller

The CFD core was developed previous in the CFD lab courses. The CFD Core Manager/Core API was added to provide a better software engineering approach and to help enforce encapsulation and modularity. The controller can interact and issue commands to the CFD core through the core API. For example changing the fluid solver will not affect anything in the rest of the application and the user will not even need to be aware of that. Having the CFD Core engine as one module with good documentation will help to reuse the engine in a game or any general fluid simulation without the need to know the architecture of the underlaying CFD engine. The API provides lots of services and different ways to control the fluid simulation behavior. An in depth explanation for the API will be given in the next sections. For now all you need to grasp is the overall view and how the API is sitting above the CFD code making any kind of performance optimization or changing the way the fluid simulation is handled independent from everything else.

We will start by explaining in details each component in the API and how it is communicating with the controller.

3.3.1.1 Core CFD

This is where all the previously developed C files and headers packed. Function calls was restructured to help decrease its complexity. Some function headers were changed and an initial code cleaning up. The core CFD is written in C for performance and portability reasons.

3.3.1.2 Core CFD Manager

The new API or Manager for the CFD Engine is written in Objective-C. This component allows the model layer to act as one abstract CFD simulation package that could be ported to any Mac or iPhone Application. We will go on directly by explaining each class in the API and how it could be used to provide a high performance fluid simulator in any iPhone App.

Problem

The problem class holds all the necessary values and information about the fluid geometrical domain and the initial conditions for the simulation. It holds as well the fluid properties and other necessary information for the simulation engine. The problem definition is broken into six components or objects

Geometry Data

This holds are the necessary geometrical properties of the fluid domain. It contains all the information about the grid and the discretization parameters.

- **dx**: Length dx of one cell in xdirection
- **dy**: Length dy of one cell in ydirection
- **xlength**: Size of the domain in x direction
- **ylength**: Size of the domain in x direction
- **imax**: Number of elements in the x direction
- **jmax**: Number of elements in the y direction

Pressure Iteration

The component holds all the parameters needed for the Poisson pressure iteration.

- **itermax**: Maximum number of pressure iterations in one time step
- **eps**: Accuracy criterion e (tolerance) for pressure iteration (res ¡ eps)
- **omg**: Relaxation factor w for SOR iteration
- **alpha**: Up winding factor (this has to be only between 0 and 1)
- **Re**: Reynolds number

Time Iteration

This explain all the necessary arguments to the momentum equations that uses the Explicit Euler method

- **dt**: The time step duration in seconds
- **tend**: The end time
- **tau**: Safety factor for time step size control T

External Forces

Any kind of external forces influencing the flow of the fluid. For example gravitational forces.

- **GX**: Force in the horizontal direction
- **GY**: Force in the vertical direction

Initial values

- **PI**: Initial pressure value
- **UI**: Initial horizontal velocity
- **VI**: Initial vertical velocity
- **wl, wr, wt, wb**: The left, right, top and bottom boundary conditions respectively; 1 for no-slip conditions, 2 for free-slip conditions, or 3 for outow conditions.
- themselves Flag: 2D array holding all the flags, one flag per cell to indicate its type (fluid or obstacle cell)

3.3.1.3 Simulation Result

The simulation result object is an abstract representation of all the fluid simulation results returning from the simulation engine. The values will be passed on to the controller which in turn hands it over to the view to be visualized as a colored image on the screen.

Further more, the object holds all the necessary information for the next time iteration. The simulation result holds a reference to the problem since the size of the grid is used to define the amount of memory needed to be allowed to all the results array.

Properties

- **P**: 2D array holding the pressure values for all the grid cells
- **U**: 2D array holding the horizontal velocity values for all the grid cells
- **V**: 2D array holding the vertical velocity values for all the grid cells
- **F**: 2D array holding on part of the discretized right-hand side of the momentum equation
- **G**: 2D array holding the second part discretized right-hand side of the momentum equation
- **RS**: 2D array holding the values for all the grid cells for the right-hand side for the pressure iteration

3.3.1.4 Simulation Engine

The simulation engine is the main backbone of the CFD Core API. The class is responsible for initializing the simulation with a given problem. Executing a series of commands and return back the results to the controller once the commands were executed successfully. The object provides different services to controller for the simulation core computations, like executing one or multiple simulation time steps, controlling the accuracy of the simulation and the number of frames per time step.

3.3.1.5 Obstacle Handler

The Obstacle handler is responsible for inserting and deleting obstacles from the fluid domain. The class provides an abstract layer for obstacle management in the fluid. The class keeps track of the obstacles present in the fluid in an array. After each time step the obstacle handler iterates on all the obstacles and decrease the age of the obstacles

by 1. If the age of the obstacle is zero then the obstacle handler removes the obstacle from the fluid as well as from its internal array.

3.3.1.6 Obstacle

An abstract representation for one obstacle in the fluid. Different attributes can be adjusted in the obstacle like size, shape, position and its age. The age is how long the obstacle will stay visible in the fluid before vanishing. When the age of the obstacle is zero that means this obstacle will be removed from the fluid.

3.4 View

3.4.1 View Design

The next tier in the MVC pattern is the view which is the objects and components that the user directly interact with and see on the device screen. Each component in the view should know how to plot itself on the screen and if applicable, interact with the user actions. The view in is the fluid image for the running simulation which the user sees on the device screen. In addition to that, the user can touch the fluid image which in turn triggers changes to the model and the user sees a new image reflecting the changes his action had on the fluid flow. The view has to be totally independent of the fluid representation in the model to enforce the concept of the MVC pattern.

Communication between the view and the model is done through the controller layer. For instance, once the user touches the fluid flowing in the screen, the view sends a request to the controller to update the model which in turn do the necessary conversions and asks the model to adjust the data model based on the user input.

All the coloring and rendering is done in the view layer. The view receives the user interaction, like touching the screen with one or more fingers. The view responds to touch events which are touch starts, touch moves and touch stopped.

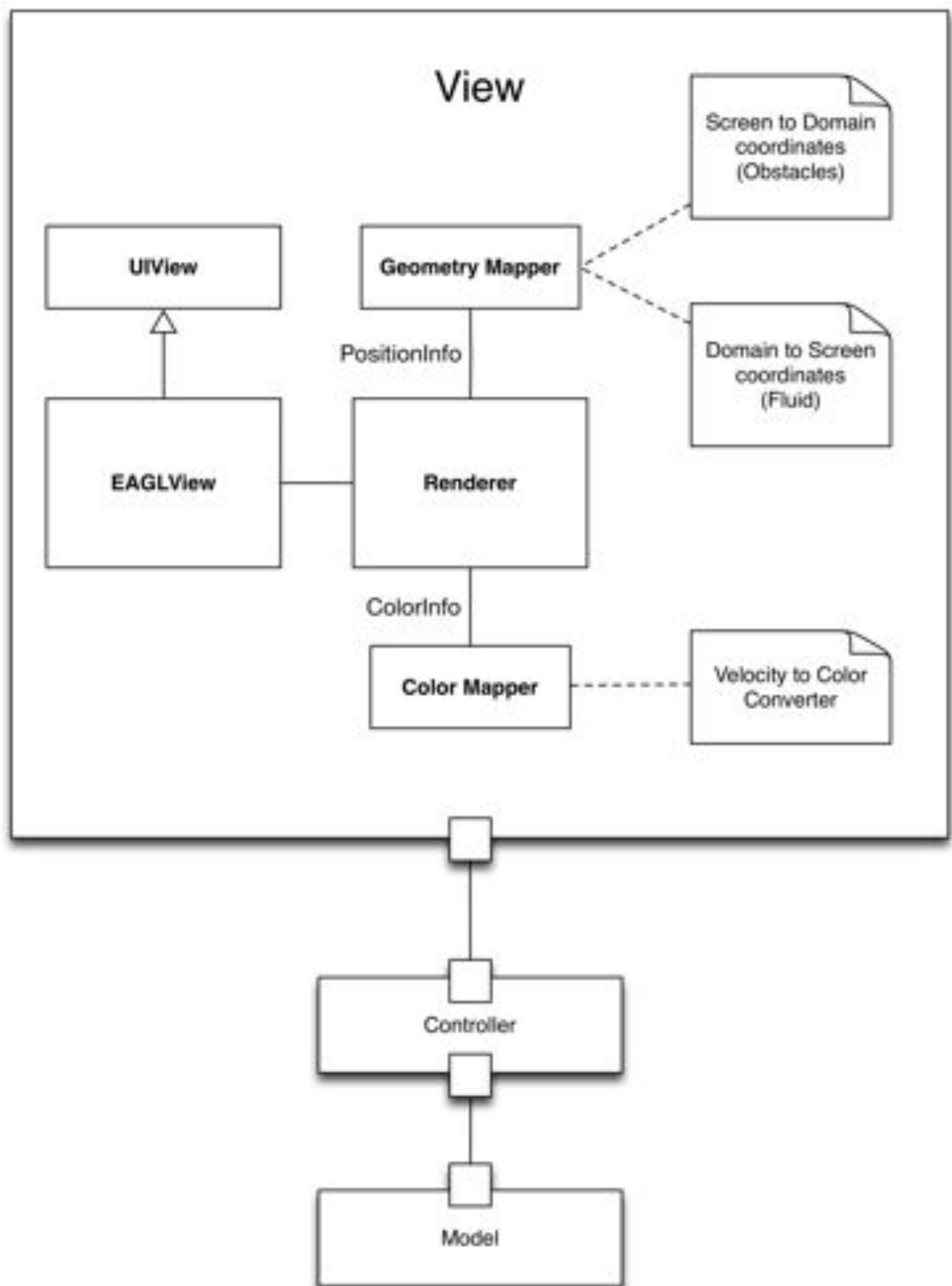

FIGURE 3.3: The figure shows the view components and how it is connected to the Controller and the model

In our App the view main functionality is to map the points in the fluid domain to the

points on the screen. The view is responsible for all the necessary transformations like scaling, rotations and translations to map a fluid cell in the fluid domain to a colored pixel in the screen domain. The same transformations are used when mapping the touch of the user on the phone screen to construct an obstacle in the fluid domain.The view is hosting the visualization core which maps the numerical values to colors and shapes on the screen. Those are the main functionalities of the Geometry Mapper and Color Mapper components shown in figure 3.3. The implementation of those components and the visualization techniques used will be discussed in details in the next chapter.

3.4.1.1 Geometry Mapper

Responsible for converting a grid cell with a specific position in the fluid domain to a pixel horizontal and vertical screen coordinates. The components applies a series of geometrical transformations(Rotate, Translate and Scale) to the fluid coordinate axis they overlap with the screen domain. After the fluid is plotted on the screen, we will need to adjust the camera projection and position to point to the

3.4.1.2 Color Mapper

The simulation of the fluid returns two values that could be visualized, the pressure and the fluid velocity. The velocity is given per each cell in the form of a vertical and horizontal velocities. In order to display the fluid flow to the user, we will need to visualize the velocity and pressure fields somehow. The pressure field will be ignored in our visualization engine and we will plot the velocity throughout the whole fluid domain. The model will convert the velocities for the whole grid to a colored image which can be presented to the user.

3.4.1.3 Renderer

The renderer is the module responsible for performing the full transformations necessary to convert the fluid simulation result to a colored image. The renderer fuse both the position and color information obtained from the geometry and color mapper objects. The renderer then forms colored vertices from the grid points. After that, the renderer

constructs triangle trips (explained in details in the next chapter) from the vertices and draw the render the final image on the screen.

3.4.1.4 EAGLView

This is the view represented to the user. The view has a reference to the renderer component, which is responsible for forming the images for the view. The other important usage for the view is handling the user interactions [8] which are

- **Touch started**: The user started touching the screen with on or more fingers.
- **Touch ended**: The user removed his fingers from the screen.
- **Touch moved**: The user moves his fingers across the screen without removing them.

The view will receive the touch events and pass them over over to the controller. The controller in turn transform the user touch into obstacles and sends over the new obstacles position details to the model. The model receives the request from the controller and construct the new obstacles in the fluid domain.

3.5 Controller

The last tier to discuss in the MVC pattern is the controller. The controller handles all the requests from the view and passes it over to the model and vice versa. For example, once the user triggers a touching event on the screen the view sends a notification to the controller of the event. The controller then converts the touch position to a position in the fluid domain and asks the model to add a new obstacle at that position. Once the user stops touching the screen the view sends a request to the controller to remove the obstacle from the fluid domain and the process continues. The other way round is the same, if the model wants to send new images to the view, it will send the request to the controller together with the necessary data, which will propagate it to the view.

The controller is the glue that holds the user view with the CFD simulation core. The controller send the necessary simulation requests to the core with the appropriate problem definition. After that, it waits for the results and pass it over to the visualization

engine to present the results on the screen to the user. The controller is responsible as well for receiving user actions and requests from the view, decide what to do and send over the commands to the model.

Chapter 4

Visualization

4.1 Definition

The visualization engine is one of the corner stones of the thesis since it is the first CFD engine ever to be implemented on the iPhone. The engine was developed from the scratch since there are no available open source CFD engines for the iPhone. The visualization engine is written in OpenGL ES which is the embedded version from the well known OpenGL graphics library [9]. The various rendering methods and color mapping algorithms will be explained in this chapter. The complete process of converting a 2D grid with velocities and pressure values to a 2D color image will be discussed in details.

Shown below in figure 4.1 is an overview for the visualization process steps. The information which we need to visualize is the U and V arrays which are holding the 2D grid velocities values. U is the velocity in the horizontal direction and V is the velocity in the vertical direction. In order to have a colored image on the screen from the velocity values per grid cell, we first need to calculate the position of each cell on the screen which is the Geometry Mapping step. The second thing is to convert the velocity values into meaningful colors, which is done in the Color Mapping step. A vertex object [9] is constructed to hold both the position and color information per pixel. Next step is to constructing all the vertices on the screen, where each vertex correspond to one and only one grid cell. If we just flushed those vertices to the screen we will end up with some holes in our image since the grid dimensions is usually smaller than the screen

dimensions. To fill in the gaps we use those vertices to construct triangle elements that can be collected into triangle strips. The triangle strips are then flushed to the screen buffer and visualized to the user. The triangle strips concept is explained later in the Triangle Elements Construction step.

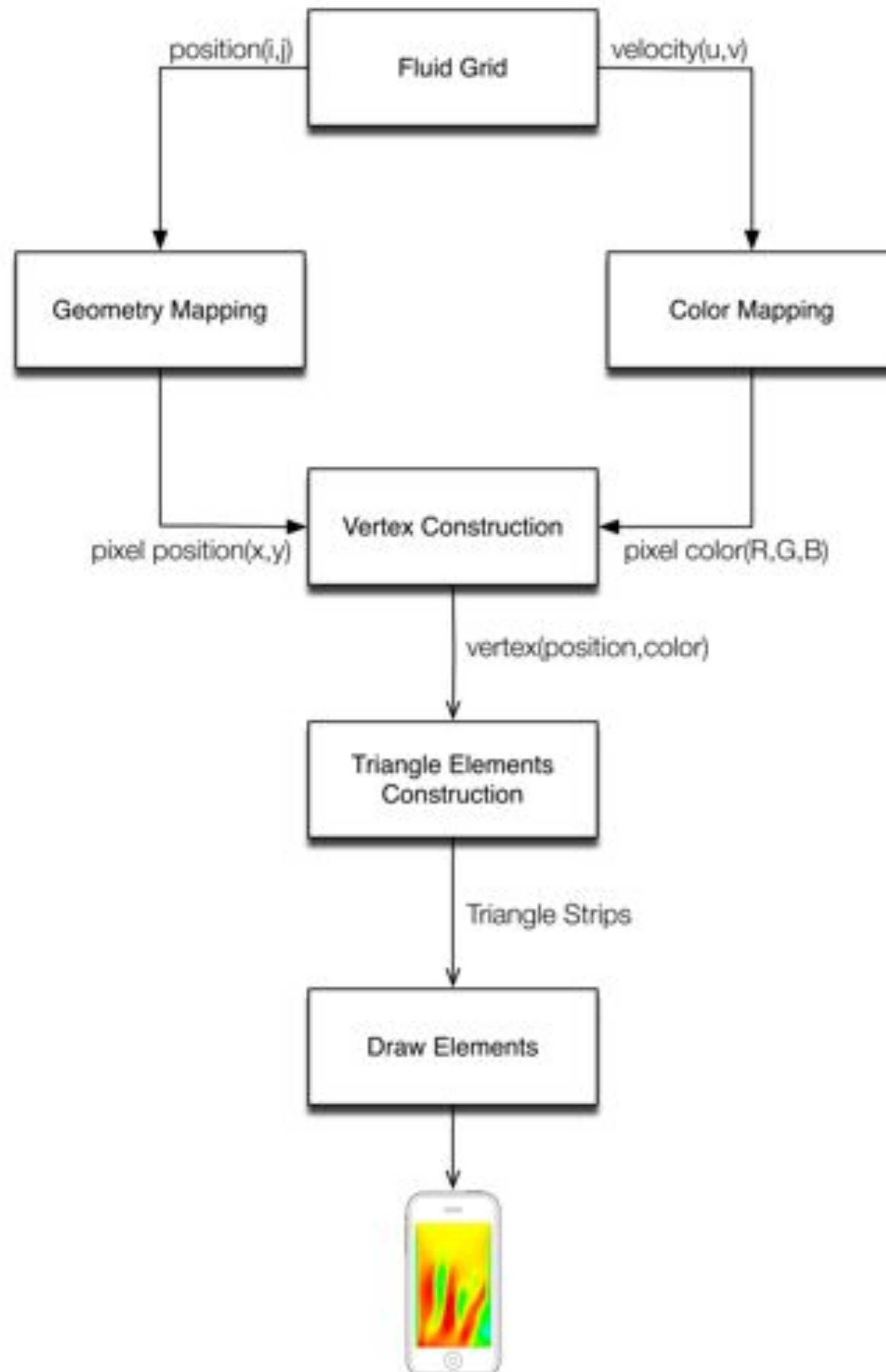

FIGURE 4.1: The different steps for the visualization process till the numerical data is changed into a picture on the phone screen

4.2 Mapping

Mapping is the process of converting the fluid grid to the screen pixels shown in figure 4.2. Each grid point is mapped to one pixel on the screen then a process of rendering and color interpolation is done to fill in the gaps between the pixels. We have two types of mapping; Geometry Mapping and Color Mapping. The Geometry Mapping component is responsible for mapping each grid cell to a pixel on the screen screen. Geometry Mapping is needed to define where to add/remove obstacles in the grid and how the grid should be positioned the phone screen. The second type of mapping is Color Mapping which defines the color to assign to each pixel based on the velocity fluid at the corresponding grid cell in the fluid domain.

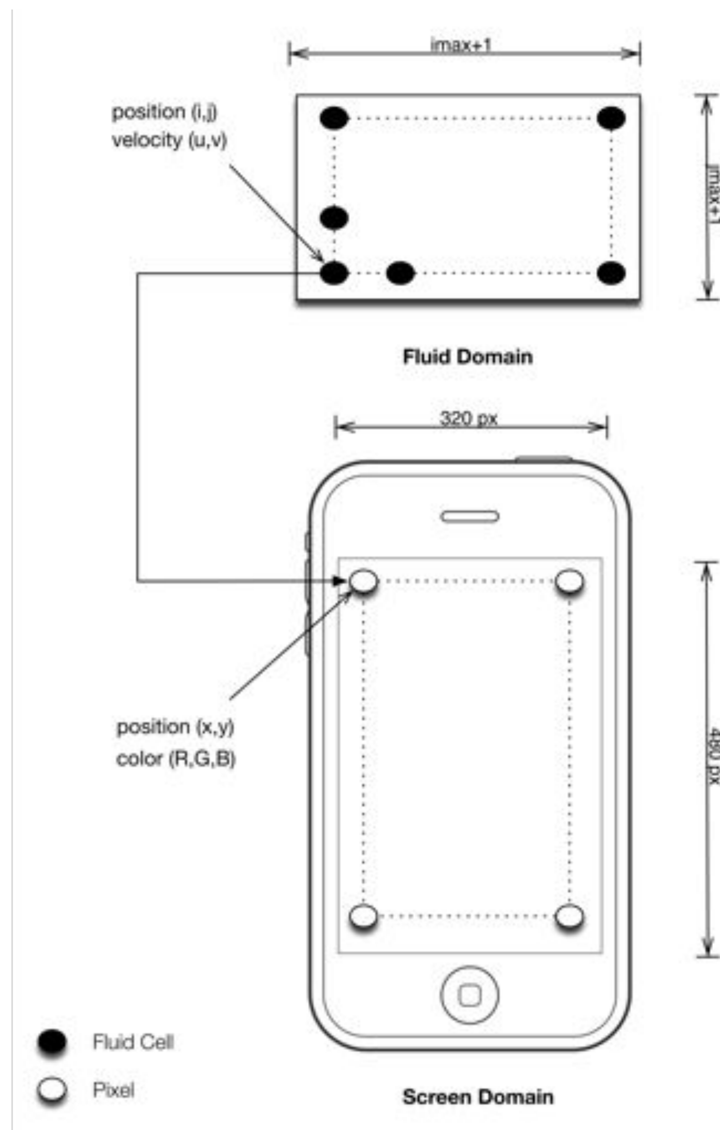

FIGURE 4.2: Each point in the fluid grid is mapping to one pixel on the phone screen

Where the phone display screen dimensions are 320 px x 480 px and the grid dimensions are (imax+1)*(jmax+1) where imax is the number of cell points in the horizontal direction and jmax is the number of cells in the vertical direction excluding the boundary cells. The grid size dimensions are defined in the problem parameters.

4.2.1 Geometry Mapping

Since fluid simulation will be running on the iPhone in landscape mode so we can use the full screen resolution of the device. The origin in OpenGL ES is in the center of the screen and we need the new center to be the top left corner so we can easily insert our grid starting from this point. Each cell in our grid is mapped to one pixel on the screen using a geometrical transformation which can be explained in the following steps:

- **Translation**: We will move the origin from the center of the phone to the top left corner as shown in figure 4.3

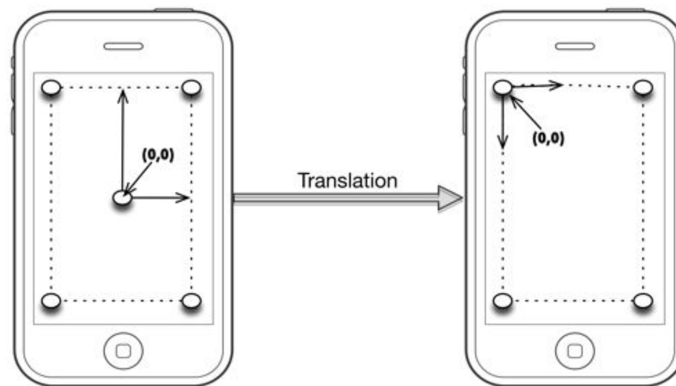

FIGURE 4.3: The figure shows the screen origin and the new origin after the translation

- **Rotation**: We rotate the origin 90 degrees clockwise so that our grid can extend in the whole screen domain as shown in figure 4.4

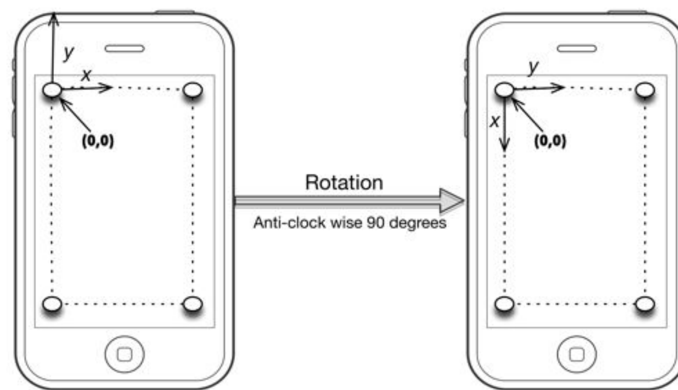

FIGURE 4.4: The figure shows the already translated origin and the new origin after the rotation

- **Scaling**: After we overlay the fluid domain on the screen domain we will need to adjust the positions of the cells in the fluid domain to fill the whole screen domain. If the fluid domain is smaller than the screen, we will have to scale up the fluid domain coordinates as shown in figure 4.5. No scaling is done if the grid size is the same size of the screen dimensions because in that case, every grid cell will correspond to one pixel on the screen. The maximum dimensions for the fluid domain is 320 x 480. Any grid size higher will have a small impact on the visualization although, finer grid will help to have a more accurate numerical solution.

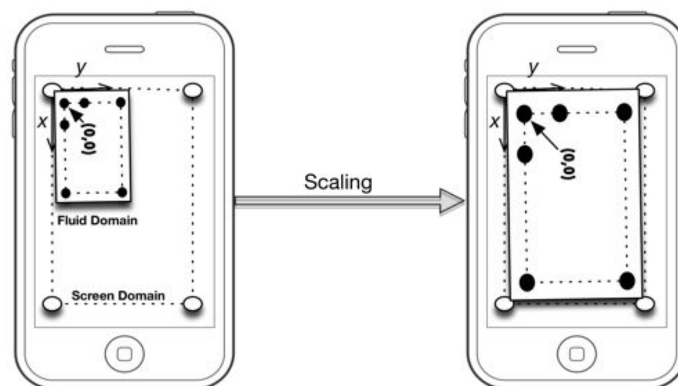

FIGURE 4.5: The figure shows how the fluid domain is mapped to the screen domain and how it will fill the whole screen after scaling

4.2.2 Color Mapping

Color Mapping is the process of converting the numerical data into meaningful colors in order to visualize the numerical data to the user in a way he can understand and interpret.

4.2.2.1 Velocity to RGB Color Mapping

After the last scaling step now we have our grid cells mapped to pixel positions. But each grid point has two velocity components u and v which needs to be visualized on the screen in colors. The process is shown in figure 4.6

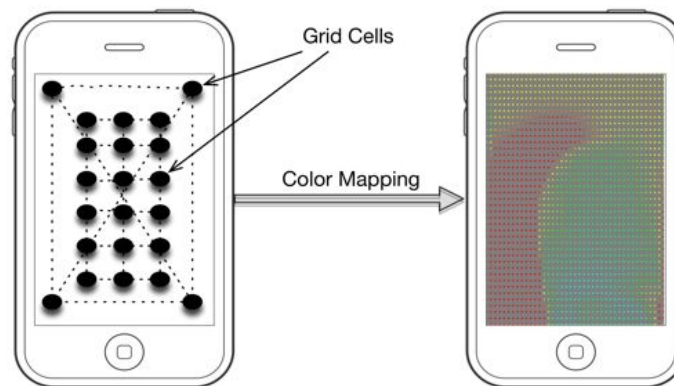

FIGURE 4.6: Each grid cell position with velocity values will be mapped to a pixel position with color values

The first step in is to convert the two components of the velocity field (u,v) into one scalar value which can be converted to a color. The absolute velocity from the u and v fields is calculated using the following equation

$$V = \sqrt{u^2 + v^2} \tag{4.1}$$

After obtaining the absolute velocity V we need to generate an RGB colored pixel from it. In order to convert the velocity value we will use a Color Map. The Color Map or Color Ramp, is a color scale corresponding to different velocities values. The Color Map used is the standard cold to hot scheme, where the hot is the red and the cold is blue. We will compute 3 extra colors evenly distributed in our scale, which are namely Cyan in the first quarter, Green in the middle and Yellow in the last quarter. The input to

our function is the maximum possible value for the absolute velocity and the minimum possible value for the absolute velocity. The output is an RGB color object. The color ramp used [10] is shown in the figure below 4.7

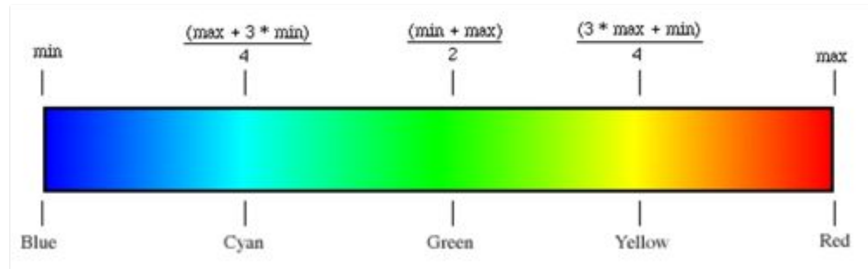

FIGURE 4.7: Color Map with the velocity to color mapping values shown [10]

The process is called color mapping and is a dynamic real-time coloring algorithm since the coloring is done without the need to know the maximum and minimum values. The algorithm [10] used is given below.

```
- (UIColor *)getColorForVelocity:(double)v{
        UIColor *color;                 //initial is white
        double red = 1.0;
        double blue = 1.0;
        double green = 1.0;
        double dv;
        if (v < vmin)
                v = vmin;
        if (v > vmax)
                v = vmax;
        dv = vmax - vmin;
        if(v < (vmin + 0.25 * dv)) {
                red = 0;
                green = 4 * (v - vmin) / dv;
        }else if (v < (vmin + 0.5 * dv)) {
                red = 0;
                blue = 1 + 4 * (vmin + 0.25 * dv - v) / dv;
        }else if (v < (vmin + 0.75 * dv)) {
                red = 4 * (v - vmin - 0.5 * dv) / dv;
                blue = 0;
        }else {
                green = 1 + 4 * (vmin + 0.75 * dv - v) / dv;
                blue = 0;
        }
        color = [UIColor colorWithRed:red green:green blue:blue alpha:1.0];
        return color;
}
```

4.2.2.2 Rendering

After the Color Mapping process we now have 2D array of vertices holding position and color information for the fluid grid in the screen domain. Till now, we only have illuminated colored pixels on the screen and empty spaces or blanks between them if the grid dimensions is smaller than the iPhone display dimensions.

Those inner pixels will need to be interpolated from the neighboring points. In order to do that, we will construct triangular polygons between each three points (square polygons is currently not available in OpenGL ES). The graphics library will interpolate the colors inside each polygon based on the colors of its three vertices. Figure 5.1 demonstrates an example for a 4 x 3 grid size. For the 12 grid cells we will need 12 triangles to construct a triangle mesh that cover up all the missing points in between the grid cells.

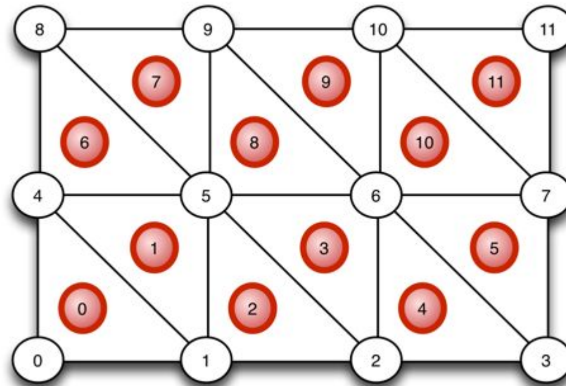

FIGURE 4.8: The figure shows the grid points in white and the triangle elements are numbered in red circles

To construct those polygons or triangle elements between the points we can use two approaches:

- **Triangles** [9]: We simply need to construct triangles between each three points in the grid. After this we will hand over the triangles geometry and vertices color to the graphics library to be rendered on the screen. For example, for the grid in figure 5.1 our triangles elements list will be given by:

```
Triangle 0  ->  (0, 1, 4)
Triangle 1  ->  (1, 4, 5)
Triangle 2  ->  (2, 1, 5)
Triangle 3  ->  (2, 6, 5)
Triangle 4  ->  (2, 6, 3)
Triangle 5  ->  (7, 6, 3)

Triangle 6  ->  (4, 8, 5)
Triangle 7  ->  (9, 8, 5)
Triangle 8  ->  (9, 6, 5)
Triangle 9  ->  (9, 6, 10)
Triangle 10 ->  (7, 6, 10)
Triangle 11 ->  (7, 11, 10)
```

From the above list and the figure it is easy to spot that the triangles share some common vertices. Extra vertices means that the graphics card will spend more time rendering the scene. This will have an impact on the over all performance of the visualization engine.

- **Triangle Strips** [9]: In order to draw connected triangles sharing one or two vertices, we can save lots of processing time if we use a strip of connected triangles. In our example we will have two triangle strips, one per row. One triangle strip is shown below in figure 4.9.

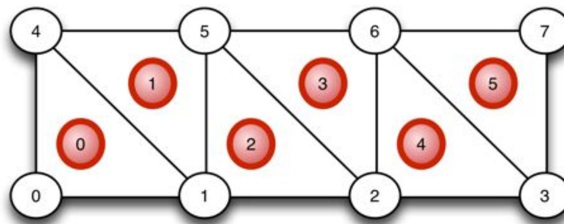

FIGURE 4.9: Each row in the grid will have one triangle strip like the one above

Where the triangle strip is given by (0, 4, 1, 5, 2, 6, 3, 7). Each vertex in the triangle strip is connected to the two preceding vertices to form a triangle. For

given triangle strip in figure 4.9, a triangle will be formed between (0,4,1), (5,1,4), (2,5,1)..etc

In order to generate the array of triangle strips to cover the whole grid, we will have to duplicate the vertices on the common edge between each two neighboring horizontal strips. How to convert our grid into an array of triangle strips is explained in figure 4.10

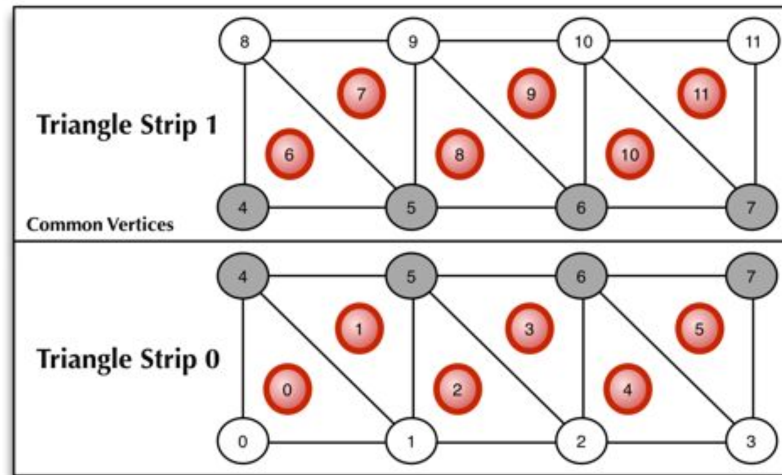

FIGURE 4.10: the grey vertices are the common vertices between the two strips that have to be duplicated

The strip will consist of an interconnected grid vertices listed as:

```
Triangle Strip 0  ->  (0, 4, 1, 5, 2, 6, 3, 7)
Triangle Strip 1  ->  (4, 8, 5, 9, 6, 10, 7, 11)
```

Therefore only two strips are needed to define the geometry of the whole domain with total of 16 vertices. While in the case of the normal triangles, we needed 3 vertices per triangle and we have 12 triangles to cover up the whole domain which is 36 vertices in total, compared to 16 in the case of using triangle strips which is less than half of the total vertices needed if we use the triangles method. Using triangle strips will result in a better and faster rendering process.

After forming the triangle strips array covering the whole fluid domain, they are sent to the graphics library to be plotted on the screen. The figure below shows how the screen will be before we do the rendering using triangle strips and how the screen will look after we do the rendering to fill in the missing gaps between the points using triangle strips.

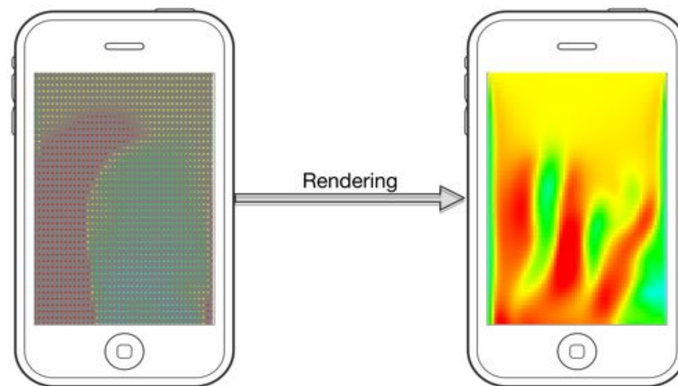

FIGURE 4.11: The figure shows how the user screen will look like before and after rendering

Chapter 5

Performance

5.1 Overview

Performance and code optimization is a very crucial part of the thesis since our main objective is to achieve a real-time interactivity with the user. Different optimization techniques has been applied to each module of the software on the design and code level. We will discuss each part of the optimization process starting with the core engine at the bottom of the simulation pipeline up to the visualization engine.

5.2 Hardware

The two devices used for testing are iPod Touch 2nd Generation and iPhone 3G S. The iPod touch was used during the performance optimization phase since it has a lower processing power than the iPhone 3G S. The CFD engine will be able to run on any iPhone or iPod touch model with an adequate real-time performance. Below is a detailed comparison between the technical specifications of the two devices:

	iPhone 3G S	iPod Touch 2nd Generation
Processor	833 MHz (underclocked to 600 MHz) ARM Cortex-A8 Samsung S5PC100	620 MHz (underclocked to 532 MHz) Samsung ARM11 core with internal ARM7 core for Jazelle acceleration
Graphics	PowerVR SGX GPU	PowerVR MBX Lite 3D GPU
Memory	256 MB DRAM	128 MB DRAM
Display	320 x 480 px, 2:3 aspect ratio	

5.3 Code Profiling

Code profiling is the process of analyzing the code and calculating each part and how much computations are done. Instruments was used to identify the most intensive and computing power consuming in the code.

Before jumping into general optimizations, the correct practice is to identify the code computations bottlenecks and start optimizing those parts. For instance, optimizing a for loop that will be called only once in the initialization could be ignored in most cases. If this part of the code is executed multiple times then any performance optimization could result in an overall performance gain for the whole App.

Instruments which is part of the XCode development tools package was used for performance profiling. Instruments gives a detailed insight on the work load on each part of the code and automatically highlights the most computationally extensive parts of the code. The figure below shows a screenshot for Instruments highlighting the most extensive parts in the simulation step.

FIGURE 5.1: The figure shows a screenshot for instruments in action

5.4 Memory Allocation and Memory Leaks

This might seem as performance irrelevant from the first look, but in practice allocating and deallocating objects takes a considerable amount of processing time. The less objects allocated, the faster the execute. Memory leaks could analyzed either using the static analysis [11] which is available in XCode version 3.2 or Instruments. The static analysis detects the memory leaks and possible bugs in the code before it is run.

Common mistakes were allocating memory and not releasing the objects afterwards. Another main performance down grade was using UIColor objects to hold the colors per pixel and passing them over to the OpenGL to rendered. If we have a grid of 100 * 100 then the App will allocate 10,000 UIColor objects, which will result in a massive performance downgrading. To workaround that, it is possible to pass directly to the colors per pixel as an RGB color float values.

5.5 CFD Core Engine optimization

The core engine is the part where most of the computations are done, therefore to have a real-time behavior this is the first component that had to be optimized. The core takes more than 70% of the processing time. Optimization had to be done on the different parts of the code using some optimization techniques that will be explained in the next section.

Loop Fusion

Fusing or merging loops was done in various parts of the code. Loop fusion should normally be done by the compiler but this way we enforce the loop merging instead of relying on the compiler. For example, loop fusion was down in the calculations of the velocities method. The two for loops responsible for calculating the horizontal and vertical velocities for the whole grid is given below:

```
for(i = 1; i < imax; i++){
        for(j = 1; j <= jmax; j++)
                if(Flag[i][j] & C_F)
                        U[i][j] = F[i][j] - dt*(P[i+1][j] - P[i][j])/dx;
}
```

```
for(i = 1; i <=imax; i++){
        for(j = 1; j < jmax; j++)
                if(Flag[i][j] & C_F)
                V[i][j] = G[i][j] - dt*(P[i][j+1] - P[i][j])/dy;
        }
}
```

We notice that both loops are nearly identical to each other and there is no dependency between them. Thus we can merge them into one loop, and execute the uncommon part separately afterwards. After the merge, the two loops are given by

```
//Loop Fusion
for(i = 1; i < imax; i++){
        for(j = 1; j < jmax; j++)
                if(Flag[i][j] & C_F){
                        U[i][j] = F[i][j] - dt*(P[i+1][j] - P[i][j])/dx;
                        V[i][j] = G[i][j] - dt*(P[i][j+1] - P[i][j])/dy;
                }
}
//the remaining iterations
for(i = 1; i < imax; i++){
        if(Flag[i][jmax] & C_F)
                U[i][jmax] = F[i][jmax] - dt*(P[i+1][jmax] - P[i][jmax])/dx;
}

for(j = 1; j < jmax; j++){
        if(Flag[imax][j] & C_F)
                V[imax][j] = G[imax][j] - dt*(P[imax][j+1] - P[imax][j])/dy;
}
```

Value Pre calculating

Pre-calculating some values instead of re-calculating them in every loop. For example for a grid size 100 * 100, the calculation will be done 10,000 times. If we pre calculate some expression and save it in a variable we will calculate it once instead of 10,000 times. For example before the optimization is done, the SOR iteration is given by:

```
int i,j;
double temp;
setPValues(imax,jmax,P,Flag,*dp);
for(i=1; i<=imax; i++){
        for(j=1; j<=jmax; j++){
                P[i][j]=(1-omg)*P[i][j] + (omg / (2 * (1/(dx*dx)+1/(dy*dy))))*
```

```
                        ((P[i+1][j]+P[i-1][j]) / (dx*dx)+ (P[i][j+1]+P[i][j-1])
                        / (dy*dy)- RS[i][j]);
        }
}
```

We notice here that there are some calculations that have a fixed value per SOR iteration. Those calculations are done in each for loop iterations. Like the calculations of the dx*dx or the inverse of dx*dx. After moving all those calculations outside the for loop we get

```
int i,j;
//dxdx,dydy are calculated once instead of 6 * imax * jmax times
double dxdx = dx * dx;
double dydy = dy * dy;
//term and negavtiveOmg are evaluated only once instead of jmax * imax
double term = (omg / (2 * (1/(dxdx)+1/(dydy))));
double negativeOmg = (1-omg);
setPValues(imax,jmax,P,Flag,*dp);
for(i=1; i<=imax; i++) {
        for(j=1; j<=jmax; j++){
                P[i][j]= negativeOmg * P[i][j] + term * ((P[i+1][j]+P[i-1][j])
                        / (dxdx) + (P[i][j+1]+P[i][j-1]) / (dydy)- RS[i][j]);
        }
}
```

The same technique is use throughout the code whenever possible

Compiler flags

Using compiler flags for optimization is not a common practice on the iPhone but in desktop applications the compiler flags would significantly boost the overall performance. For more information about the available compiler flags[12]. Flags -O2 or -O3 could be used but they do not have a big impact on performance on the iPhone in the case of our CFD engine.

Numerical optimization

Our main focus is to get a decent visualization and provide some real time interactivity with the fluid. To simplify the calculations we used fixed iterations number in the SOR solver. The calculations of the residual error have been removed. Having a fixed number of iterations means a less accurate result but faster execution time. We will use

a medium number of iterations that could give us an accurate result to some extend and an adequate response time.

5.6 Performance optimization gain

The frame per sec rate was calculated for different grid sizes before and after the optimization process. The device used is iPhone 3G S with an SOR of 20 iterations. The table below shows the difference between the execution speed before and after the optimization process. From the calculated numbers, we notice that for a low grid resolution, the performance gain η is nearly 1.8, where the performance gain is calculated using equation 5.1

$$\eta = \frac{F_A}{F_B} \tag{5.1}$$

Where F_A is the frame rate per second after the optimization and F_A is the frame rate before optimization.

The more we increase the grid the less this gain will be, till it drops to around 1.3 for a full resolution grid. This is because on a very fine grid size, most of the computations are done in the SOR solver.

	Speed (Frame/Sec)	
Grid Size (imax * jmax)	Before optimization	After optimization
20 * 30	40.6	73.3
40 * 60	17.6	23.2
80 * 80	6.7	9.1
80 * 120	4.3	6.1
320 * 480 (Full Resolution)	0.3	0.4

Chapter 6

Test Cases

6.1 Non-interactive Simulations

In this section we will try simulating some of the classical examples of a fluid simulation problems, like the Karman vortex street and the flow over a step scenario. The problem parameters used for both scenarios is are given below [2]

- **Pressure Iteration**: itermax = 10, eps = 0.001, omg = 1.7, alpha = 0.9, Re = 100
- **Geometry Data**: xlength = 1, ylength = 1
- **Time Iteration**: dt = 0.05, tau = 0.5
- **External Forces**: GX = 0, GY = 0
- **Initial values**: PI = 0, UI = 1 VI = 0, wl = 3, wr = 3, wt = 1, wb = 1

6.1.1 Flow over a step

The fluid is flowing from the left to the right in a channel. The obstacle is in a form of a square in the lower left corner of the channel and it is occupying half the channel height.

Results and Performance

The simulation was tested on both the iPhone and iPod. The frame rate was calculated for different grid sizes given below and figure 6.1 shows the simulation running with a fine grid on the iPhone.

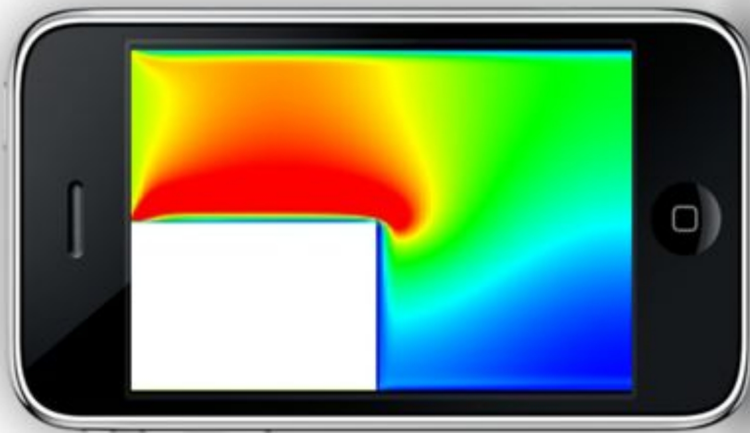

FIGURE 6.1: The visualization shows the flow of the fluid in the channel. The obstacle is highlighted in white

	Speed (Frame/Sec)	
Grid Size (imax * jmax)	iPhone 3G S	iPod Touch 2nd Generation
20 * 30	81	68
40 * 60	25	18
80 * 80	10	6
80 * 120	7	3
320 * 480 (Full Resolution)	0.4	0.2

6.1.2 Karman Vortex Street

The Karman Vortex Street is one of the classical CFD simulation used to model the von Karman vortex phenomena, which can be seen in clouds movements, missiles, bridges or transmission lines as an effect of the wind blowing past them.

The simulation was tested on both the iPhone and iPod. The frame rate was calculated for different grid sizes given below and the figure 6.2 shows the simulation running with a fine grid on the iPhone.

Results and Performance

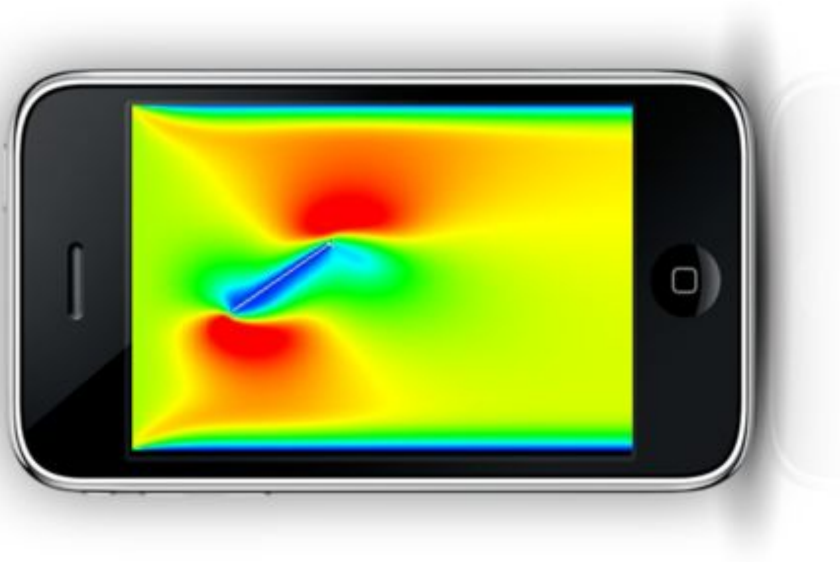

FIGURE 6.2: The visualization shows the flow of the fluid in the channel with a thin plate hanging in the middle of the channel

	Speed (Frame/Sec)	
Grid Size (imax * jmax)	iPhone 3G S	iPod Touch 2nd Generation
20 * 30	73.3	66.4
40 * 60	23.2	17.3
80 * 80	9.1	6.3
80 * 120	6.1	4.4
320 * 480 (Full Resolution)	0.4	0.25

6.2 Interactive Simulations

This section shades the light on the interactivity capabilities available in the CFD simulation engine. There are two types of possible interaction with the fluid domain. The first is by inserting obstacles in the fluid domain once the user touch the screen. The second scenario is instead of inserting obstacles, we will add velocity to the fluid cells depending on how fast we swipe over the screen. We will explain in the next two sections in details how each scenario is implemented

6.2.1 Insert an Obstacle on touch

The user can interact with the fluid by applying a touch event to the phone screen. We can apply up to four simultaneous touch events on the screen. Each touch is modeled as an obstacle in the grid at the touching position. The obstacle could be modeled as a circle or a square. We will use the circle representation to reflect the nature of the human finger. Once the user start touching the screen the following process is done in order to add the obstacle at the given touch position on the screen.

- Get the screen coordinates of the user touch on the screen (Multi touch events are allowed).
- Convert the position of the touch on the screen to a grid cell position in the fluid domain.
- Insert an obstacle at the given position in the grid by marking the cells around the touch position as obstacle cells, and set the velocity to zero. In order to mark the cells of the obstacle shaped as a circle as shown in figure 6.3. We have to start from the circle center with radius equals one and increment gradually to the obstacle radius, marking all the cells as obstacle cells in a circular manner on each step. The finer our degree step size the more fine circle shape we will get.

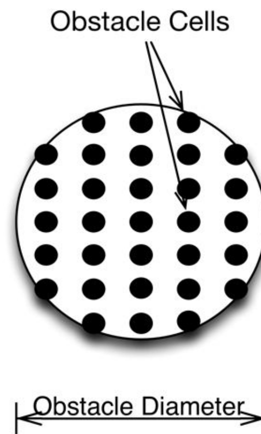

FIGURE 6.3: The figure shows how the circle shaped obstacle is approximates by a mesh of points

A simplified version of the algorithm that adds a circular obstacle to the fluid grid is given below, where kobstacleSize is a constant defining the radius of the circle.

```
for(int j = 1; j<kobstacleSize; j++){
        for(int i=0; i<360; i++){
                l = sin(i)*j+xcenter;
                m = cos(i)*j+ ycenter;
                //make sure that the obstale is not outside the grid
                if(l < currentProblem.jmax && m < currentProblem.imax){
                        //mark the cell as an obstacle
                        currentProblem.Flag[l][m] = C_B;
                        //clear the velocities at the grid point
                        currentResult.U[l][m] = 0.0;
                        currentResult.V[l][m] = 0.0;
                }
        }
}
```

- After inserting the obstacle in the grid, the age of the obstacle is used to determine when the obstacle should be removed from the fluid. The obstacle is not removed immediately, but rather starts to shrink gradually before it vanishes completely from the fluid. After each time step all the current obstacles in the fluid domain age are decremented by one. Once the obstacle age hits zero it is removed from the fluid domain completely.

Results and Performance

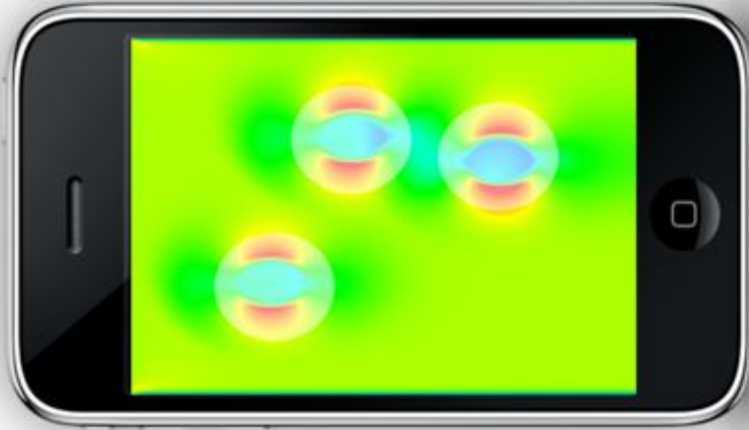

FIGURE 6.4: The figure shows the case where the user touches the fluid using three fingers and does not swipe after this

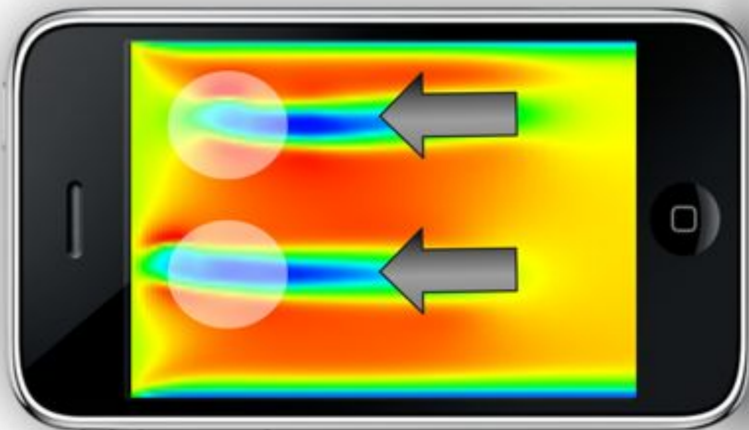

FIGURE 6.5: The figure shows the case where the user swipes his figures across the fluid

The two figures above shows a typical usage behavior. Figure 6.4 is an example for a touch and remove sequence where the user just tap on the screen using a couple of fingers. The fade grey circles show the touch positions. It is clear from the simulation

that the velocity value will increase around the touch positions which starts to fade away once the touch stops. The high velocity is indicated by the red color. Figure 6.5 shows a use case where the user swipes with his two fingers across the screen. The black arrows indicates the direction of the swipe. Blue colors indicates that the fluid velocity is low at those points.

It is important to note that the process of inserting obstacles in the fluid and shrinking them till they are removed from the fluid will dramatically affect the whole numerical solution. The SOR solver given a stable fluid flow usually takes few iterations to converge while when insert obstacle we change the whole domain definition therefore we need high number of SOR iterations to get an acceptable numerical solution. The interactive simulation is run for grid size of 80 x 120 (ratio 1:4 to the real screen dimensions) to determine the maximum number of iterations that could be done and still get a real-time interactivity with the fluid flow. Too small iterations will result in an incorrect numerical solution, and increasing the iterations will result in an accurate but slow solution and we lose the real time behavior for the fluid.

Number of SOR Iterations	Speed (Frame/Sec)
5	7.11
10	6.04
20	4.62
40	3.21

From the results in the table above it is clear that increasing the SOR iterations influence the performance.

6.2.2 Add Velocity on touch

The second scenario is that instead of inserting obstacles in the fluid, we will add extra velocity to the fluid depending on how fast the user will push it by touching. This is a realistic effect which occurs when you try to touch any fluid flow, it will move away due to the increase in its velocity. This test case is limited to only one touch event. The process of adding velocities to the fluid can be explained in the following steps

- Once the user touch the screen, the touch screen coordinates is saved in a temporary variable called currentTouchPosition.
- Once the user starts moving his finger in the screen the end touch position is saved in a temporary variable called endTouchPosition.
- After a specific period of time, let's say every 10 frames or every 1 second we will measure the velocity of the touch movement from the start to the end. The velocity is calculated using the following equation

  $$V\binom{u}{\nu} = \left| \frac{Touch\binom{x_1}{y_1}_{end} - Touch\binom{x_2}{y_2}_{start}}{t} \right| \tag{6.1}$$

  Where V$\binom{u}{\nu}$ is the velocity at the end position, Touch$\binom{x_1}{y_1}_{end}$ is the end touch screen coordinates , Touch$\binom{x_2}{y_2}_{start}$ is the start touch screen coordinates and T is the time between the two events.
- Add the velocity V$\binom{u}{\nu}$ at the end position Touch$\binom{x_1}{y_1}_{end}$ to the cells velocities around the touch position. The same shape as the obstacle discussed before is used to determine the range of the affected cells.
- If the touch is still moving, clear the endTouchPosition. The new startTouchPosition is now the endTouchPosition and restart the whole process again from the beginning.

  If the touching ended, then clear both the endTouchPosition and the startTouchPosition.

Summary

We started by explaining the numerical background necessary to understand the basic behavior of the CFD engine. Our CFD engine is used only to describe non compressible fluids like water or honey. It is clear that the previous available code had no performance optimization nor did it follow any software engineering practices, making it hard to debug, maintain and extend. A totally new API had to be built above previous code to provide easy to access methods and add computational steering support to the CFD engine. The mathematical and numerical model is explained briefly and more can be found in literature[1].

We continue after this by explaining the over all architecture and design of the application which is following the classical Model View Controller design pattern. The MVC is the main design pattern used in most of the iPhone applications where in our App the model is holding all the simulation engine, the view is responsible for displaying the data to the user in a meaningful way and the controller is handling the communication between the two components. In addition to display the data the view is responsible for handling the user actions and pass them over to the controller. The controller is responsible for adding and removing the obstacles from the fluid. The controller keeps a track for the current obstacles present in the fluid in a queue.

In the visualization we explained in details the process of converting the fluid cells to a colored image to be displayed on the screen. The process of rendering the fluid from a given fluid grid was discussed in details. We explain after this our cold-hot coloring scheme which could be easily changed to suit any other coloring scheme for a different fluid.

Performance optimization is an important part of the paper and some steps were explained briefly how the core CFD engine and visualization engine were optimized. We used triangle strips instead of using triangle elements in the visualization process to increase the performance for the rendering process.

We tested the core engine for different interactive and none interactive simulations to determine the minimum grid size to get a decent visualization feedback for both cases. The engine was tested on both an iPhone and an iPod for different simulation scenarios.

The engine proved to offer adequate performance for grid sizes under 100 * 100 with real time interactivity. It was explained how changing the number of the SOR iterations can affect the stability of our solution and the execution time as well. By using an average SOR iterations number we manage to get a solution with good accuracy and a decent frame rate that made it possible to provide real-time interactivity to the user. The engine provides an adequate real time performance for a mesh with sizes below 100 * 100. It is clear that the iPhone 3G S is faster than the iPod touch in terms of computational power.

The CFD engine was implemented for the first time on a handheld device and it offers real time interactivity to the user. Further improvements could be done, like controlling the fluid flow direction with the device sensor. The visualization engine could be extended to include more coloring schemes to support different fluid types. The CFD core engine could be further optimized by using a multi-grid solver instead of the SOR solver. The coloring algorithms which maps the velocity values to the colors could be optimized by using a static lookup table, where each velocity directly correspond to an RGB color this has a complexity of N(1) which will improve the coloring process performance.

Bibliography

[1] Tilman Neunhoeffer Michael Griebel, Thomas Dornsheifer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction (Monographs on Mathematical Modeling and Computation)*. SIAM, 1997.

[2] Dr. Miriam Mehl. Praktikum wissenschaftliches rechnen computational fluid dynamics, worksheet 1. 2009.

[3] Dr. Miriam Mehl. Praktikum wissenschaftliches rechnen computational fluid dynamics, worksheet 2. 2009.

[4] Feng Xian. Computational steering systems in grid computing environments.

[5] Stephen G. Kochan. *Programming in Objective-C*. Addison Wesley, second edition, 2009.

[6] Dr. Miriam Mehl. Lab course computational fluid dynamics, 2009.

[7] Apple Inc. Model-view-controller.

[8] Jeff LaMarche Dave Mark. *Beginning iPhone Development: Exploring the iPhone SDK*. Apress, 2009.

[9] Benjamin Lipchak Richard S Wright. *OpenGL SuperBible*. Addison Wesley, fourth edition, 2007.

[10] Paul Bourke. Colour ramping for data visualisation, 2009.

[11] Apple Inc. Static analysis in xcode 3.2.

[12] gcc.gnu.org. Optimize options.