EPS
Escola Politècnica Superior
UdG

UNIVERSITAT DE GIRONA

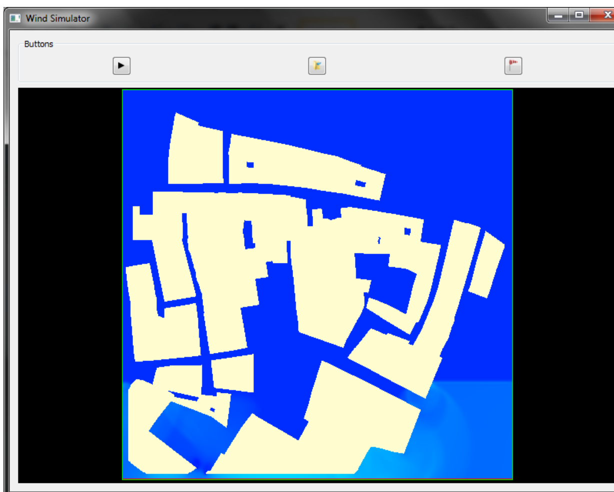
ENGINYERIA INFORMÀTICA

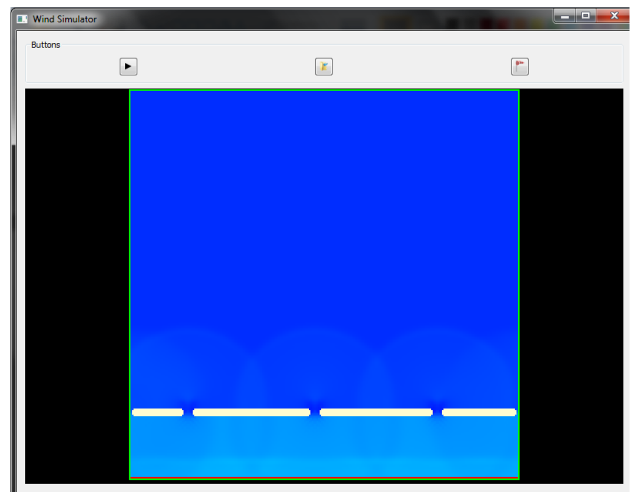
Simulació de Vent en Temps Real per Paisatges

Projecte/Treball Final de Carrera (Pla 1997)

MEMÓRIA

Eduard Rando Segura





TUTOR: Dr. Gustavo Patow
DEPARTAMENT: Informàtica i matemàtica aplicada
ÀREA: LSI

Juny 2014

Índex

Capítol 1: Introducció 5
Apartat 1.1: Motivacions 5
Apartat 1.2: Propòsits 5
Apartat 1.3: Objectius 6
Apartat 1.4: Capítols de la Memòria 7
Capítol 2: Estudi de viabilitat 8
Apartat 2.1: Recursos tècnics pel desenvolupament del projecte 8
Apartat 2.2: Recursos humans 8
Apartat 2.3: Tecnologia 8
Apartat 2.4: Cost econòmic 9
Apartat 2.5: Conclusions de l'estudi de viabilitat 10
Capítol 3: Metodologia 11
Capítol 4: Planificació 14
Capítol 5: Marc de treball i conceptes previs 16
Apartat 5.1: Mètode Lattice-Boltzmann 16
Apartat 5.1.1: Introducció 16
Apartat 5.1.2: Detalls LBM 18
Apartat 5.1.3: Condicions de contorn 22
Apartat 5.1.4: Forces externes 23
Apartat 5.2: GPUs 24
Apartat 5.2.1: Visió de la pipeline gràfica 24
Apartat 5.2.2: Estructura de les GPUs 26
Apartat 5.2.3: GPUs de propòsit general (GPGPU), pipeline computacional 26
Apartat 5.2.4: Llenguatges per la *pipeline* computacional 27
Apartat 5.2.5: Llenguatge *CUDA* i la seva arquitectura 27
Apartat 5.2.6: *OpenGL* 31
Capítol 6: Requisits del sistema 33
Apartat 6.1: Plantejament de la problemàtica 33
Apartat 6.2: Plantejament de la solució 34
Apartat 6.3: Requisits Funcionals 34
Apartat 6.4: Requisits No Funcionals 35
Capítol 7: Estudis i decisions 36
Apartat 7.1: Microsoft Visual Studio 2012 36

Característiques 37
Motius de selecció 43
Apartat 7.2: C++ 44
Motius de selecció 44
Apartat 7.3: *CUDA* 45
Motius de selecció 45
Apartat 7.4: *FreeImage* 45
Característiques 45
Formats Suportats 47
Motius de selecció 47
Apartat 7.5: OpenGL 48
Motius de selecció 48
Apartat 7.6: Qt 48
Motius de selecció 49
Apartat 7.7: Sistemes Operatius: Windows 7 49
Motius de selecció 49
Apartat 7.8: GANTT Project 50
Motius de selecció 50
Apartat 7.9: StarUML 50
Motius de selecció 50
Capítol 8: Anàlisi i disseny del sistema 51
Apartat 8.1: Fitxes i diagrames de cas d'ús 51
Diagrama de cas d'ús general 52
Diagrama de classes 55
Capítol 9: Implementació i proves 77
Apartat 9.1: WindSimulator 77
Mètodes de WindSimulator 78
Apartat 9.2: Interfície d'usuari 90
QtDisplayer 92
GLWidget 96
WindChanger 104
Apartat 9.3: Proves 106
Capítol 10: Resultats 108
Apartat 10.1: Objectius de l'aplicació 108

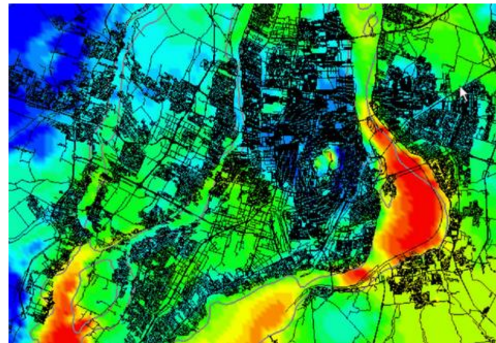
Realitzar una simulació en temps real mitjançant un llenguatge de programació en paral·lel, com CUDA. 108
Obtenció de la informació bàsica de l'escena. 109
Implementació del mètode de Lattice-Boltzmann per realitzar la simulació. 112
Permetrè al usuari modificar la direcció del vent en temps real. 114
Creació d'una interfície d'usuaria simple i intuïtiva. 116
Apartat 10.2: Validesa legal de l'aplicació. 116
Capítol 11: Conclusions 117
Capítol 12: Treball futur 119
Capítol 13: Bibliografia 120
Capítol 14: Annexos 121
Annex A: Conceptes bàsics de C++ 121
Estructura bàsica 121
Concepte classe 121
Herència 122
Sobrecarrega d'operadors 122
Capítol 15: Manual d'usuari i/o instal·lació 124

Capítol 1: Introducció

En aquest capítol introductori explicarem les motivacions, propòsits i objectius del nostre projecte final de carrera. Finalment, presentarem un breu resum dels diferents apartats que trobarem en aquesta memòria.

Apartat 1.1: Motivacions

La simulació de la realitat és un fenomen que va sorgir fa uns anys per tal de predir esdeveniments sense haver de malbaratar recursos. El problema inicial de la simulació va ser la necessitat de simplificar la realitat a causa de la manca de capacitat dels ordinadors de l'època.



Amb els anys això ha canviat, i avui dia qualsevol ordinador de sobre taula té prou recursos per fer simulacions força realistes. Així doncs, volem fer un pas de realisme a les simulacions i afegir les característiques del vent en aquestes.

Amb aquest projecte volem ajudar, per exemple, a estudis científics sobre la difusió de la contaminació en grans nuclis a causa de l'efecte del vent, càlculs de trajectòries amb forces externes degudes al vent, o incorporar en el món de la multimèdia efectes realistes de vent.

Apartat 1.2: Propòsits

Els propòsits per realitzar aquest projecte són els de desenvolupar una aplicació que ens permetí, en qualsevol ordinador amb prou capacitat de computació, realitzar una simulació en temps real de vent. La idea és la de poder proporcionar a l'aplicació un paisatge com a *input* i aquesta faci els càlculs necessaris per simular el comportament del vent segons la geometria de l'escena i ens proporcioni de forma visual el resultat del comportament del vent.

Apartat 1.3: Objectius

El principal objectiu d'aquest projecte és desenvolupar un sistema que permeti realitzar simulacions realistes de vent per un paisatge 2D, i estudiar com el vent és veu afectat per la geometria de l'escena. Un punt important, és que tot ha de ser en temps real.

Per aconseguir-ho, utilitzarem tècniques basades en el mètode de *Lattice-Boltzmann*, el qual consisteix en una xarxa regular que representa el fluid en posicions discretes, i estudiar com flueix. Escollint els paràmetres correctes de la simulació, es pot demostrar que aquest mètode convergeix a les equacions continues de *Navier-Stokes*, les qual són les més importants per descriure el comportament macroscòpic d'un fluid.

Per accelerar tots els càlculs, utilitzarem la capacitat i la potencia de les targes gràfiques, ajustarem l'algorisme per poder-lo utilitzar en paral·lel, tot tenint en compte les restriccions de les *GPUs*.

També haurem de generar un sistema per poder llegir les escenes 2D sobre les que realitzarem la simulació. Finalment, haurem de "pintar" el vent per tal de poder visualitzar el resultat de la simulació.

Per tant, els objectius bàsics d'aquest projecte són:

- Estudi i implementació de les tècniques de *Lattice-Boltzman* per a la simulació de vent.
- Per accelerar la simulació necessitarem estudiar algun llenguatge de programació en paral·lel, com per exemple *CUDA*.
- Estudi i recerca de llibreries per la lectura de les escenes.
- Estudi de llibreries per a la renderització dels resultats de la simulació. Com a exemple podem parlar de llibreries *OpenGL*.
- Estudi i desenvolupament d'interfícies d'usuari per l'aplicació.
- Documentar tot el procés per realitzar el projecte.

Apartat 1.4: Capítols de la Memòria

La memòria està estructurada en els següents apartats:

- **Capítol 1. Introducció**. En aquest capítol mostren les motivacions, els propòsits i objectius del projecte. També fem un resum dels diferents apartats.
- **Capítol 2**. **Estudi de viabilitat**. En aquest capítol justificarem la viabilitat del projecte en terme de recursos tant econòmics, com tecnològics o humans.
- **Capítol 3**. **Metodologia**. En aquest capítol explicarem el mètode seguit per desenvolupar el projecte.
- **Capítol 4**. **Planificació**. En aquest capítol explicarem el *timing* del projecte, és a dir, el pla de treball, les tasques planificades i el temps estimat per cada etapa.
- **Capítol 5**. **Marc de treball i conceptes previs**. En aquest capítol introduirem els conceptes necessaris per comprendre millor la resta de la memòria.
- **Capítol 6**. **Requisits del sistema**. En aquest capítol descriurem els requisits funcionals i no funcionals del sistema per assolir els objectius.
- **Capítol 7**. **Estudis i decisions**. En aquest capítol descriurem el maquinari, les llibreries i/o programari utilitzat.
- **Capítol 8**. **Anàlisi i disseny del sistema**. En aquest capítol mostrarem les necessitats del sistema i les solucions proposades per cada necessitat.
- **Capítol 9**. **Implementació i proves**. En aquest capítol mostrarem els problemes trobats durant el desenvolupament del projecte i com els hem solucionat. També mostrarem el model de classes del sistema i els algorismes més rellevants.
- **Capítol 10**. **Resultats**. En aquest capítol descriurem detalladament el procés de desenvolupament del projecte i en mostrarem els resultats obtinguts.
- **Capítol 11. Conclusions**. En aquest capítol reflexionarem sobre l'assoliment dels requisits establerts, explicarem les possibles desviacions de la planificació inicial i farem una crítica dels resultats obtinguts.
- **Capítol 12. Treballs futurs**. En aquest capítol valorarem les possibles ampliacions o millores del projecte per el futur.
- **Capítol 13. Bibliografia**. En aquest capítol, mostrarem les referencies utilitzades per desenvolupar el projecte.
- **Capítol 14. Annexos**. En aquest capítol, es mostrarà informació complementaria del projecte.
- **Capítol 15**. **Manual d'usuari i/o instal·lació**. En aquest capítol, explicarem com s'ha d'utilitzar la aplicació i, si fes falta, com s'ha d'instal·lar.

Capítol 2: Estudi de viabilitat

En aquest capítol valorarem la viabilitat del projecte des de diferents aspecte, com poden ser els recursos econòmics necessaris, els recursos humans, la viabilitat tecnològica, etc.

Apartat 2.1: Recursos tècnics pel desenvolupament del projecte

Els recursos tècnics requerits pel desenvolupament del projecte es redueixen essencialment, a un ordinador amb una tarja gràfica amb capacitat de paral·lelitzar processos. Pel que fa al sistema operatiu, ens és indiferent treballar en Linux, MacOs o Window, tots són vàlids per desenvolupar el projecte.

El projecte s'ha dut a terme amb un PC amb una arquitectura Intel Core i5-2430M, una CPU de quatre nuclis de 2.4 GHz amb un sistema operatiu Windows 7 de 64 bits. També disposa dues memòries RAM de 4GB a 1333Hz cadascuna i una tarja gràfica NVIDIA Geforce GT 540M de 2GB amb capacitat per executar CUDA.

Apartat 2.2: Recursos humans

El projecte requerirà dels següents rols:

- Analista: serà l'encarregat de definir i organitzar els passos a seguir per desenvolupar el projecte, és a dir, és qui s'encarregarà de dissenyar l'aplicació i documentar-la.
- Programador: serà l'encarregat de realitatzar les tasques definides per l'analista per poder tenir una aplicació final.
- Cap de projecte: serà l'encarregat de coordinar la feina i la ben entesa dels membres del grup.

Aquesta és la teoria, a la pràctica tant el rol d'analista com el de programador han estat desenvolupats per mi i el de cap de projecte, encarregat de l'orientació, ha estat desenvolupat pel tutor del projecte.

Apartat 2.3: Tecnologia

A nivell tecnològic, i assumint els requisits ja especificats en l'apartat 2.1, necessitarem algunes eines de software per desenvolupar el projecte.

En primer lloc, necessitarem un entorn de programació per a Windows, ja que aquest sistema operatiu no té cap compilador de forma nativa com si té, per exemple, Linux. L'entorn de programació que utilitzarem serà el Microsoft Visual Studio 2012 (en endavant, MSVC2012). També necessitarem diferents llibreries per dur a terme algunes de les tasques definides, però més endavant discutirem l'elecció d'aquestes llibreries.

Apartat 2.4: Cost econòmic

En aquest apartat definirem el cost econòmic dels recursos tècnics, la tecnologia i els recursos humans que hem explicat prèviament.

En el cas dels recursos tècnics, com treballarem amb un PC de la nostre propietat, ens estalviarem el cost de comprar-lo, tot i això haurem de tenir en compte l'amortització dels recursos utilitzats. Per calcular-la utilitzarem la següent formula:

$$Amort = \frac{Preu\ recurs * messos\ feina}{36}$$

En referència a la tecnologia, gracies al conveni SPARK entre la UdG i Microsoft, podrem disposar del MSVC2012 de forma gratuïta. Pel que fa a les diverses llibreries que utilitzem, sempre prioritzarem a utilitzar llibreries "Open Source" o de codi lliure.

Els recursos humans, vistos des d'una perspectiva teòrica, tenen diferent cost en funció de la feina que es desenvolupa. Per exemple el analista té un cost superior al del programador.

- Analista: 20 €/h.
- Programador: 10 €/h.

Les feines a desenvolupar són:

- Estudi de les eines.
- Disseny de l'estructura de classes i algorismes principals.
- Implementació.
- Proves.
- Memòria.

Així, els costos segons tasca i perfil serien:

Tasca	**Perfil**	**Hores**	**Cost/h**	**Cost total**
Estudi d'eines	Analista	100	20 €/h	2.000 €
Disseny classes/algorismes	Analista	200	20 €/h	4.000 €
Implementació	Programador	600	10 €/h	6.000 €
Proves	Programador	100	10 €/h	1.000 €
Memòria	Analista	200	20 €/h	4.000 €
TOTAL PROGRAMADOR		700	10 €/h	7.000 €
TOTAL ANALISTA		500	20 €/h	10.000 €
TOTAL				17.000 €

Com ja hem dit, aquest costos serien en cas de contractar a dues persones que realitzessin aquesta feina, però a la pràctica tota aquesta feina és realitzarà sense cap retribució econòmica.

Apartat 2.5: Conclusions de l'estudi de viabilitat

Pel que hem pogut observar en els apartats anteriors, la viabilitat del projecte està garantida a nivell tècnic i tecnològic. Això és així degut a que ja disposem dels requeriments necessaris només començar el projecte, i que el programari que utilitzem o bé permet, el seu ús per l'aprenentatge, o és de codi lliure.

A nivell econòmic, tindríem un cost força elevat d'aproximadament uns 17.000€, la qual cosa dificultaria la viabilitat del projecte. Però aquest no és el nostre cas.

Capítol 3: Metodologia

Avui en dia, hi ha moltes metodologies de desenvolupament eficients i diferenciables, des de les més antigues com la metodologia *Waterfall* fins a les més modernes tècniques *Agile*. En aquest projecte no s'ha seguit cap metodologia d'un tipus concret, ja sigui *Spiral*, *Scrum* o *Iterative*.

El que hem fet ha estat definir un tipus de metodologia que funcionés bé amb les característiques del projecte, tal i com es mostra en el següent diagrama de flux de la Figura 3.1:

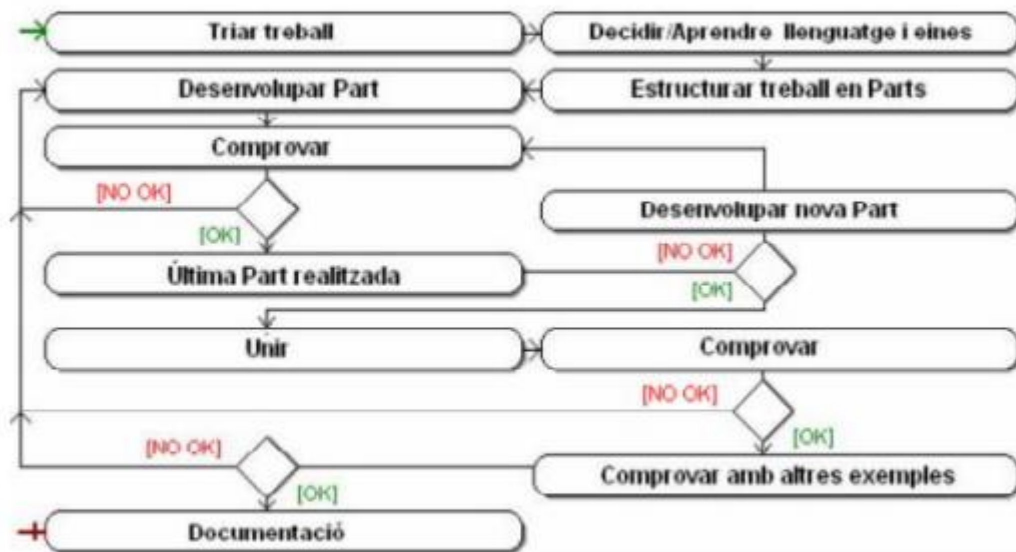


Figura 3.1: Diagrama de flux de la metodologia *skylineEngine*.

Com és pot veure en la figura 3.1, la metodologia disposa del següents passos:

1. Triar el treball a desenvolupar.
2. Decidir el llenguatge de programació i les eines a utilitzar.
3. Aprendre el llenguatge de programació i el funcionament de les eines escollides.
4. Estructurar el treball en parts segons les funcions que ha de realitzar.
5. Desenvolupar la part corresponent seguint l'ordre de l'estructura del treball.
6. Fer comprovacions per tal de confirmar que el funcionament és correcte al finalitzar la part.
    a. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és convenient.
    b. Si al fer les comprovacions el resultat és l'esperat es desenvolupa la part següent tornant al punt 5, en cas que s'hagin finalitzat les parts amb les respectives comprovacions s'inicia el punt 7.
7. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.
    a. Si al fer les comprovacions el resultat no és l' esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és convenient.
    b. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 8.
8. Generar diferents models d'exemple per a comprovar que el funcionament és el correcte.
    a. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és convenient.
    b. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 9.
9. Arrodonir la Documentació.

Tal i com podem veure, consisteix en dividir el projecte en mòduls i organitzar en el temps el desenvolupament, la verificació i la correcció. Tant durant el temps de desenvolupament com de verificació es fa un seguiment mitjançant tutories setmanals o bisetmanals depenent de l'etapa, ja que en els inicis la manera d'avançar és molt més lenta que al final i no sempre es necessari fer tutories tan freqüentment.

Quan s'acaba un mòdul, sempre que es pugui, es finalitza totalment de manera que no s'hagi de retocar, així garantim que els errors que puguin sorgir en el mòdul actual són únicament d'aquest i no de cap dels anteriors, o almenys que afectin el mínim possible.

Pel procés de disseny utilitzarem el llenguatge de modelat estàndard dins del camp de l'enginyeria del programari, l'UML (*Unified Modeling Language*). L'UML s'utilitza per definir un sistema, per detallar els seus elements, per documentar i construir. Per aconseguir això, l'UML disposa de nombrosos tipus de diagrames que mostren diversos aspectes dels elements representats.

Els motius d'escollir aquesta metodologia, han estat per les característiques del projecte mateix. En primer lloc, és tracte d'un projecte en el qual només hi estarà treballant una persona, per la qual cosa, no és necessari detallar excessivament cada part per coordinar la feina entre els integrants de l'equip de treball.

Així doncs, aquesta metodologia ens permet agilitzar el procés de desenvolupament al evitar haver de passar una gran quantitat de temps dissenyant prèviament a desenvolupar. Això no treu, que dissenyem les parts principals del projecte per evitar-nos problemes per falta de disseny a posteriori.

Capítol 4: Planificació

Aquest projecte va començar en el mes de juliol del 2013 amb la intenció de ser finalitzat i entregat en el juny de 2014. Amb aquest propòsit en ment, vam planificar les tasques de la següent manera:

1- Juliol 2013 – Octubre 2013. Estudi de programació en paral·lel en sistemes heterogènies, és a dir, aprendre les bases de programació de CUDA i realitzar diverses proves per consolidar els coneixements.
2- Octubre 2013 – Desembre 2013. Estudi de tècniques de simulació de fluids, més concretament l'estudi del mètode de *Lattice-Boltzmann* i desenvolupament del nucli del nostre projecte.
3- Novembre 2013 – Desembre 2013. Estudi de llibreries per realitzar la lectura de escenes i poder donar un input al nostre sistema.
4- Gener 2014 – Març 2014. Estudi de llibreries de renderització per mostrar els resultats de les simulacions.
5- Març 2014 – Maig 2014. Estudi de llibreries per realitzar interfícies d'usuari (UI) i desenvolupament de la UI.
6- Maig 2014 – Juny 2014. Arrodoniment de la memòria del projecte.

Per veure més clarament la planificació del projecte podem observar la Figura 4.1, on es pot veure el diagrama de Gantt.

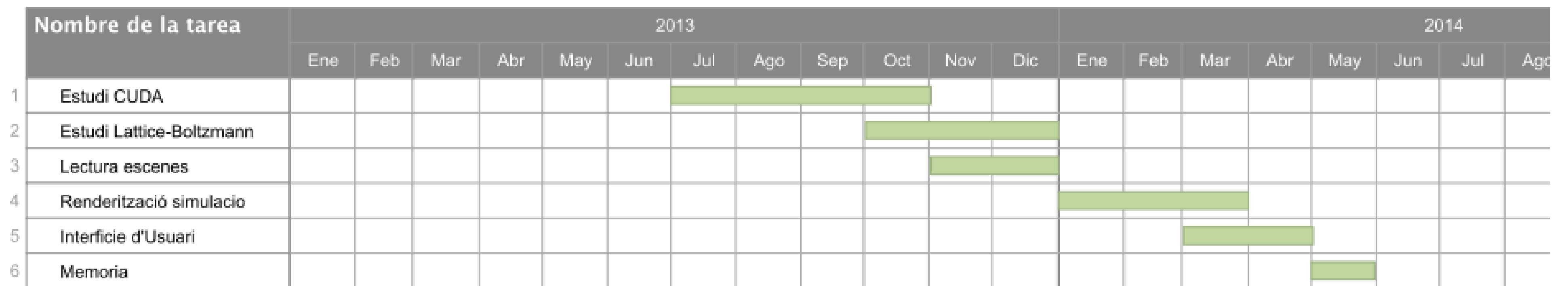


Figura 4.1: Diagrama de Gantt

Capítol 5: Marc de treball i conceptes previs

En aquest capítol explicarem els conceptes necessaris a priori per poder comprendre el projecte i poder-ne fer un bon seguiment. Els conceptes que requerim explicar són els següents:

- Mètode *Lattice-Boltzmann*
- *GPUs*
    - Estructura de la *GPU*
    - Llenguatge *CUDA* i la seva arquitectura
    - Llenguatge *OpenGL*

Apartat 5.1: Mètode Lattice-Boltzmann

En la simulació de fluids ens trobem amb la necessitat de treballar amb la seva mecànica, amb les equacions de *Navier-Stokes* i la seva complexitat. Per definir el comportament d'un fluid, hem de tenir en compte aspectes com la velocitat del fluid, la pressió entre les molècules, la viscositat o les forces externes que interaccionen amb en fluid. Per sort, hi ha el mètode de *Lattice-Boltzmann* que ens permet simplificar la complexitat de la simulació.

Per entendre millor el mètode, primer introduirem l'evolució des de les equacions de *Boltzmann* fins el mètode de *Lattice-Boltzmann*. Després explicarem en més detall el mètode.

Apartat 5.1.1: Introducció

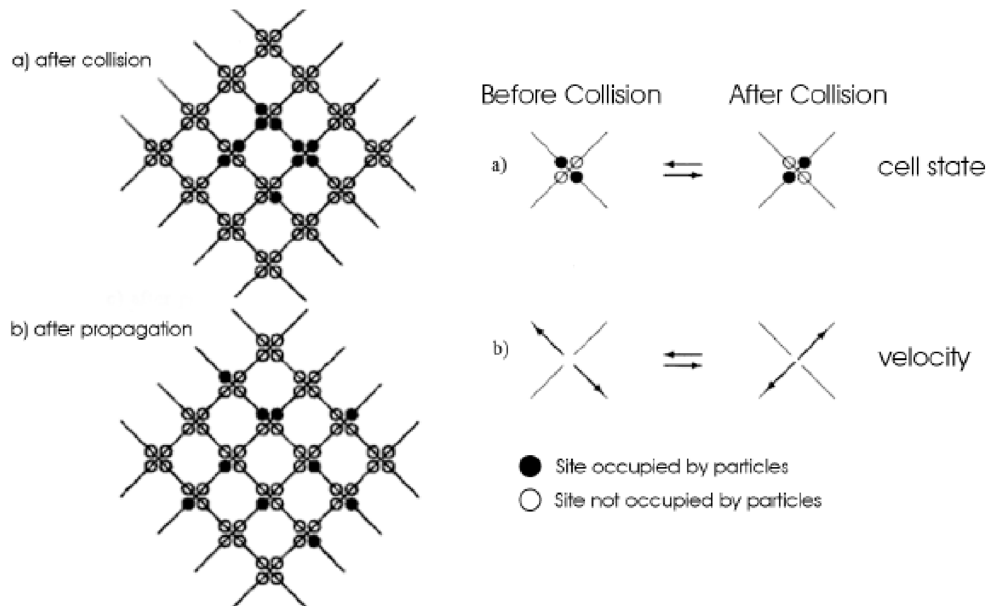


Figura 5.1: *Lattice Gas Cellular Automat*

El mètode de *lattice Boltzmann* té els arrels en les equacions de *Boltzmann*, proposades el 1872 per Ludwig Boltzmann. Les equacions de *Boltzmann* descriuen el comportament del gas a nivell microscòpic mitjançant la teoria de la cinètica. Aquest mètode proporciona una distribució estadística de partícules en un espai de fase d'una sola partícula.

En [Hardy et al., 1976] es va proposar el *Lattice Gas Cellular Automata* (LGCA) tal i com podem veure en la Figura 5.1. El LGCA va ser introduït com el model conceptual per el comportament microscòpic d'un fluid, capaç de resoldre les equacions de *Navier-Stokes*. El model està composat per un enreixat on cada intersecció és un booleà que indica l'estat de la partícula, tal i com es mostra en la Figura 5.1. Les posicions ocupades per partícules tenen un valor de 1, per contra si no hi ha partícules el valor és de 0. Dos processos ocorren en una posició, la propagació i la col·lisió de partícules. En la propagació, les partícules es mouen en la direcció de la seva velocitat cap a les posicions veïnes. El pas de col·lisions resol els conflictes per posicions que han rebut múltiples partícules després de la propagació.

El principal problema del LGCA és que és altament anisotròpic a causa de la invariància rotacional. Això simplement vol dir que els vòrtexs produïts pel model són de forma quadrada. Es van trigar deu anys, fins que [Frisch et al., 1986] va introduir el LGCA hexagonal, per resoldre alguns del problemes de la anisotropia. Tot i els esforços de *Frisch*, encara hi havia molt problemes en aquest mètode. Alguns dels problemes eren les llargues fluctuacions en el flux del fluid (soroll estadístic), la impossibilitat de simular en tres dimensions, i les simulacions estaven limitades a fluids amb una alta viscositat. El mètode de *lattice Boltzmann* (LBM) va aparèixer en resposta a les limitacions del LGCA.

La primera proposta per [McNamara i Zanetti, 1988], el LBM substituïa valor booleà de partícules per direccions del enreixat amb la distribució de densitat del fluid per evitar soroll estadístic. Tot i això, el LBM encara tenia problemes per simular en 3D i només podia simular fluids viscosos.

La viabilitat per simular en 3D va sorgir gracies a [Higuera i Jimenez, 1989]. Ells van suggerir canviar el procés de col·lisió, passant d'una operació no-lineal de col·lisió a una de lineal. Aquests canvis van permetre accelerar les simulacions permetent simulacions en 3D. [Quian i Lallemand, 1992] van proposar millores en les col·lisions del LBM per permetre simulacions de fluid amb poca viscositat. Van decidir eliminar les col·lisions del LBM per tal que només les conseqüències de les col·lisions importessin. [Quian i Lallemand, 1992, Chen et al., 1992] van suggerir una ultima millora en l'operació de col·lisió coneguda com l'aproximació de *Bhantnagar-Gross-Krook*. Aquesta versió de LBM és coneguda com a model *lattice-BGK* (LBGK) i proporciona un sol temps de relaxació. El LBGK és el LBM més popular utilitzat avui dia a causa de la seva simplicitat i eficiència.

El LBM és inherentment compressible. Com a conseqüència, aquest modela les equacions compressibles de *Navier-Stokes*. La compressibilitat dels fluids és una característica principal del LBM i és el que dona una avantatge de rendiment sobre els altres mètodes. Tot i així, [He i Luo, 1997] van reconèixer que existia una necessitat respecte els fluid incompressibles i van introduir una variant incompressible per el LBM. Una limitació del LBM incompressible és la de mantenir la velocitat del fluid baixa per tal de minimitzar els efectes de la compressibilitat.

Apartat 5.1.2: Detalls LBM

El LBM treballa sobre un enreixat. Existeixen forces variacions del LBM, cadascuna d'elles esta anomenada com DXQY, on 'X' són les dimensions i 'Y' són el nombre velocitats del enreixat o de vectors d'influència. LBM pot esser descrit com un tipus de autòmat de partícules, és a dir, el fluid es modela com un conjunt de partícules del mateix tipus. Totes les cel·les del enreixat són actualitzades a cada pas d'execució per regles simples, les quals tenen en compte l'estat de les cel·les del voltant.

El nostre fluid és l'aire, el qual en unitats molt petites, el podem considerar com un fluid incompressible. Per aquest motiu, farem servir les equacions de LBM per modelar fluids incompressibles, descrites per [He i Luo, 1997]. Normalment treballarem amb el model D3Q19 per simulacions en 3D o el model D2Q9 per simulacions 2D, podem veure en detall una cel·la d'aquests models en la Figura 5.2.

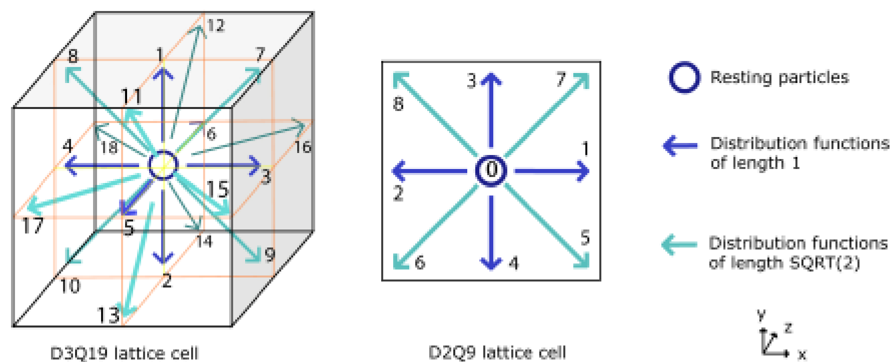


Figura 5.2: a l'esquerra tenim el model D3Q19. A la dreta el model D2Q9.

Els vectors d'influència es referencien com e_i on i és el numero del vector d'influència. En el cas del model D2Q9 els vectors d'influència són $e_0 - e_8$. Per cada cel·la $\vec{x}$ i temps t, les partícules del fluid es mouen a velocitats arbitraries modelades per les funcions de distribució de partícules $f_i(\vec{x}, t)$, on el valor de i correspon al numero del vector d'influència. Cada $f_i(\vec{x}, t)$ és l'estimació del numero de partícules que es mouran en el vector d'influència e_i. Fixem-nos que les partícules només es poden desplaçar segons els vectors d'influència. La magnitud dels vectors e_1a $e_4$és de 1 unitat del enreixat per cada fracció de temps. La magnitud dels vectors e_5a $e_8$és de $\sqrt{2}$ unitats del enreixat per cada fracció de temps. La magnitud del vector e_0 és de 0, ja que representa les partícules que es queden a la pròpia cel·la. Aquestes partícules, les de e_0, no és mouen enlloc en la següent fracció de temps, però algunes d'elles poden ser accelerades a causa de la col·lisió amb altres partícules, per tant la quantitat de partícules que romandran en la cel·la pot canviar.

A partir de les funcions de distribució de les partícules, podem calcular dos valors físics molt importants. Sumant totes les funcions de distribució per una cel·la, podem calcular la densitat de la cel·la, assumint que totes les partícules tenen la mateixa massa d'1.

$$\rho = \sum_{i=0}^{Y} f_i \quad (eq\ 1)$$

Una altra informació important per cada cel·la és la velocitat i sobretot la direcció en la que és mouen les partícules d'una cel·la. De nou hem de sumar totes les funcions de distribució de partícules, però aquest cop cada funció es multiplicarà primer pel vector d'influència corresponent. D'aquesta manera, la funció de distribució de partícules f_0 es multiplica per e_0, que és $(0{,}0)$, per tant sempre és zero. La funció de distribució f_1 es multiplica per e_1 $(1{,}0)$ i es suma a la funció de distribució f_2 multiplicat per e_2 $(-1{,}0)$, etc. Aquest resultat s'escala per la densitat inicial, la qual normalment val 1, ja que les funcions de distribució de partícules contenen el total de partícules.

$$\vec{v} = \frac{1}{\rho_0} \sum_{i=0}^{Y} f_i \vec{e_i} \quad (eq\ 2)$$

Una simulació consisteix en dos passos, la *propagació* i la *col·lisió*, que és repeteixen per cada fracció de temps. Aquest dos passos simulen els fenòmens de convecció i de difusió que ocorren a nivell macroscòpic en la física. El pas de *propagació* és el més senzill, només cal moure les partícules d'una cel·la a una altre, tal i com és mostra en la Figura 5.3.

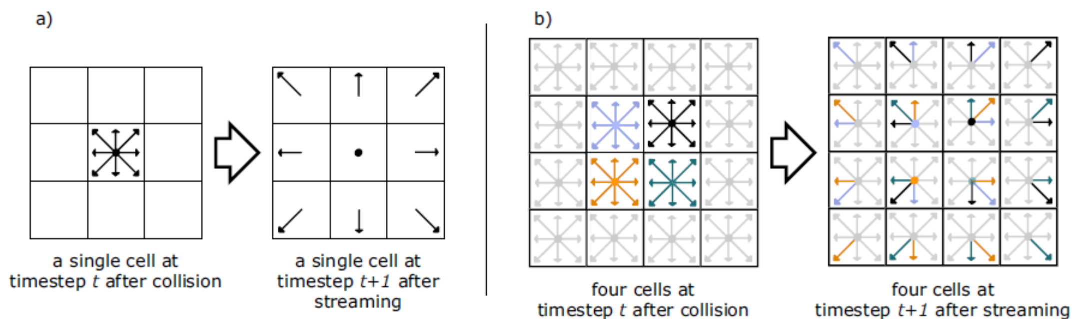


Figura 5.3: Pas de propagació

Com a exemple, una cel·la amb coordinades [i,j], la funció de distribució pel vector d'influència que apunta avall és copiaria en la funció de distribució d'avall de la cel·la [i+1,j]. El vector del centre no apunta enlloc, per tant les seves partícules no es copiaran. El pas de *propagació* és pot descriure matemàticament com:

$$f_i(\vec{x} + \vec{e_i}, t + 1) = f_i^{new}(\vec{x}, t) \quad (eq\ 3)$$

On $\vec{e_i}$ és el vector d'influència que apunta en la mateixa direcció que la funció de distribució. Per exemple, si la funció de distribució és f_1, llavors el vector d'influència seria $\vec{e_1}$ o (1,0).

En el pas de *col·lisió*, arriben partícules a una cel·la i col·lisionen amb altres partícules. Com la Figura 5.4 demostra gràficament, el pas de *col·lisió* no modifica la densitat o la velocitat d'una cel·la, només modifica la distribució de les partícules per totes les funcions de distribució de partícules de la cel·la. Per exemple, considerem una cel·la amb coordinades [x,y] on el fluid es mou en direcció positiva sobre l'eix x. La cel·la no perdrà cap partícula durant la *col·lisió*. No obstant, el moviment s'escamparà cap a altres vectors d'influència de la cel·la que apuntin en la direcció positiva del eix x. El vectors que apuntin en la direcció oposada seran més petits. Això esta il·lustrat a la Figura 5.4. En el següent pas de propagació, les cel·les veïnes de coordenades [x+1,y] rebran una major quantitat de partícules de la cel·la amb coordinades [x,y], mentre que les cel·les veïnes amb coordenades [x-1,y] rebran una menor quantitat de partícules.

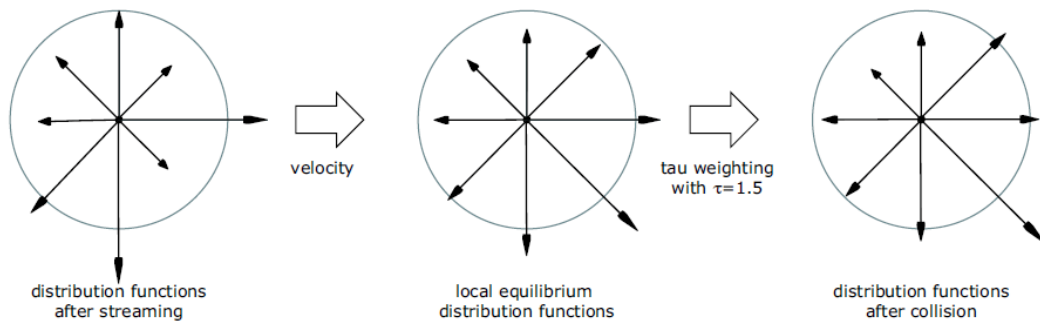


Figura 5.4: Pas de col·lisió

Per modelar aquest comportament, hem de calcular la funció de distribució d'equilibri f_i^{eq}i les noves funcions de distribució. [He i Luo, 1997] van suggerir que la funció de distribució d'equilibri

$$f_i^{eq}(p,\vec{v}) = w_i\left[\rho + \rho_0\left(\frac{3}{c^2}(\vec{e_i}\cdot\vec{v}) + \frac{9}{2c^4}(\vec{e_i}\cdot\vec{v})^2 - \frac{3}{2c^2}(\vec{v}\cdot\vec{v})\right)\right] \quad (eq\ 4)$$

funciona bé per reproduir el comportament de fluid incompressibles. Cada vector d'influència té una funció de distribució d'equilibri. La velocitat bàsica en l'enreixat és denomina per c. En implementacions bàsiques $c = \rho_0 = 1$. El pesos per el model D2Q9, depenen de la longitud dels vectors:

$$w_i = \begin{cases} 4/9 & i = 0, \\ 1/9 & i = 1,2,3,4, \\ 1/36 & i = 5,6,7,8. \end{cases}$$

El terme $\left[\frac{3}{c^2}(\vec{e_i} \cdot \vec{v}) + \frac{9}{2c^4}(\vec{e_i} \cdot \vec{v})^2 - \frac{3}{2c^2}(\vec{v} \cdot \vec{v})\right]$ en la funció de distribució d'equilibri és responsable d'això. Les noves funcions de distribució són:

$$f_i^{new}(\vec{x}, t) = (1 - \omega) f_i(\vec{x}, t) + \omega f_i^{eq}(\rho, \vec{v}) \quad (eq\ 5)$$

amb

$$\omega = \frac{2}{6v + 1}$$

Aquí v és la viscositat del fluid en unitats del enreixat. El rati de relaxació ω afecta en la rapidesa amb la qual el fluid arriba al equilibri. Per raons d'estabilitat, el valor de ω ha de estar dins del rang de 0 a 2. Per $\omega < 1$ el fluid serà més viscos, com la mel, per $\omega > 1$ el fluid serà menys viscos, com l'aigua.

En la teoria, els passos de propagació i col·lisió es solen combinar en una única formula coneguda com l'equació de *lattice Boltzmann*,

$$f_i^{new}(\vec{x}, t) - f_i(\vec{x} + \vec{e_i}, t + 1) = (1 - \omega) f_i(\vec{x}, t) + \omega f_i^{eq}(\rho, \vec{v}) \quad (eq\ 6)$$

La part esquerra de l'equació representa el pas de propagació. La part dreta es pot reconèixer com la combinació de la actual funció de distribució i l'equilibri de la cel·la.

La simulació de LBM ha de procedir de la següent manera:

1. Calcular la densitat d'acord amb l'equació 1.
2. Calcular la velocitat (Equació 2).
3. Calcular la distribució d'equilibri (Equació 4).
4. Actualitzar les funcions de distribució segons l'equació 6
5. Tornar al pas 1.

Apartat 5.1.3: Condicions de contorn

Fins ara hem vist el comportament de les partícules dins de les cel·les, però en els límits de la simulació, o més ben dit, en els contorns, hem de definir unes noves funcions de distribució pels contorns.

La condició de contorn estàndard per simulacions LBM són les parets *no-slip*. L'exemple d'aquestes condicions és que el fluid en el contorn no es mou. Per tant, cada cel·la de *Lattice Boltzamnn* al costat del contorn ha de tenir el mateix nombre de partícules movent-se en direcció al contorn que partícules movent-se en la direcció oposada. El resultat d'això serà una velocitat de zero en la cel·la, ho podem imaginar com si les partícules reflectissin en el contorn. El procés de reflexió és mostra en la Figura 5.5, a l'esquerra el cas *no-slip*. A la dreta de la Figura 5.5, podem veure el cas de *free-slip,* el qual només la velocitat normal al contorn és reflectida.

Una altre condició pot ser la *half-way bounce-back,* el qual es mostra en la Figura 5.6. La condició *half-way bounce-back* funciona reflectint la funció de distribució de partícules que entren en les cel·les de contorn cap a les direccions oposades.

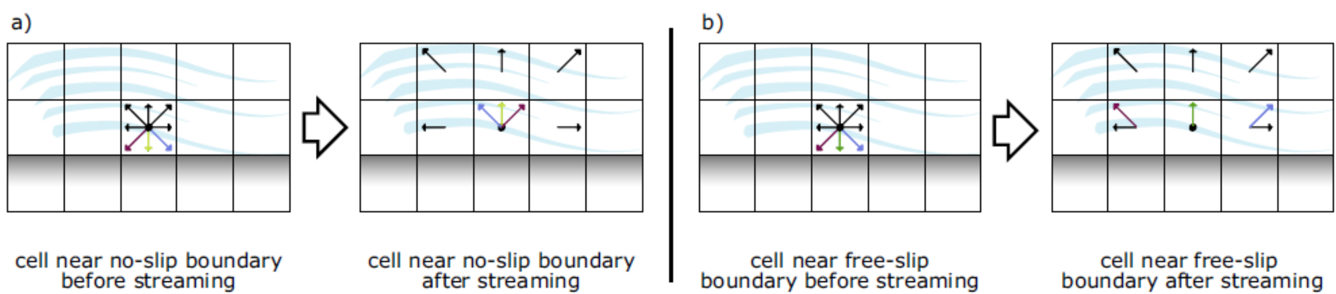


Figura 5.5: Els obstacles *No-Slip*, part esquerra de la figura, reflecteixen directament la funció de distribució entrant. En el cas de les condicions de contorn *free-slip*, part dreta, la funció de distribució és reflecteix al llarg del vector normal al contorn.

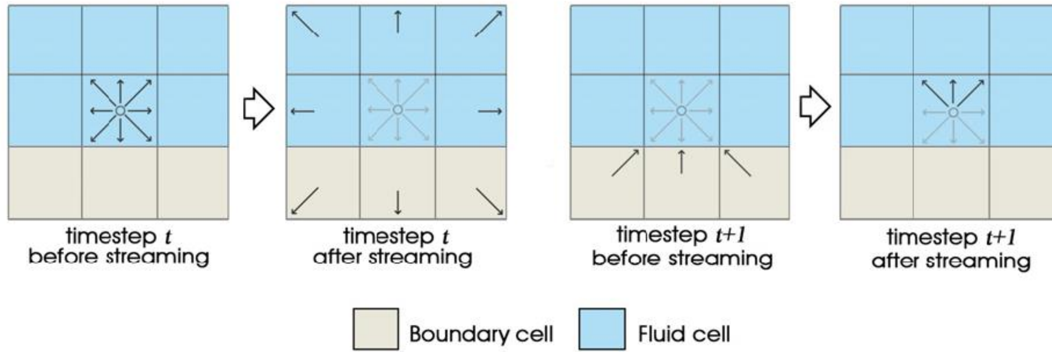


Figura 5.6: Condició de contorn *Half-way Bounce-back*.

Apartat 5.1.4: Forces externes

Tot i que per la realització d'aquest projecte no treballarem amb forces externes, creiem que per assolir un coneixement complet del mètode cal també explicar com hi poden interaccionar amb forces externes.

Així doncs, per obtenir una simulació realista necessitem afegir forces externes al model, forces com poden ser la gravetat o la força del vent. Les forces externes com la gravetat es poden incorporar al LBM de moltes maneres (veure [Buick i Greated, 2000]). Però, la manera més senzilla és la de modificar el càlcul de la velocitat a cada cel·la en el pas de col·lisió. D'aquesta manera, l'equilibri local és calcula amb la velocitat $\vec{v}$ modificada de la següent manera:

$$\vec{v} = \vec{v} + \tau\vec{F}$$

on $\vec{F}$ és la força externa, i τ és el temps de relaxació del LBM. Cal tenir cura quan calculem la velocitat en cada cel·la per cassos especial com calcular la trajectòria de partícules.

Una altre possibilitat és la de modificar les funcions d'equilibri. Per cada càlcul amb l'equació 4 la funció de distribució de partícules s'ha de modificar afegint la força externa per les direccions necessàries. Una combinació del dos mètodes també és possible.

Apartat 5.2: GPUs

Una Unitat de Procés Gràfic (en anglès: *Graphic Proces Unit* o GPU) és un dispositiu de representació gràfica dedicada d'un ordinador personal, estació de treball o consola.

Qualsevol PC o consola que tingui o no una *GPU*, ha de tenir sempre una *CPU* que executarà el sistema operatiu i els programes. Les *CPU*, per disseny, són de propòsit general. Les *CPUs* executen programes, com poden ser l'editor de textos Word, escrits en llenguatge de propòsit general com són el C++ o Java.

Gràcies a la seva arquitectura, les GPUs tenen la capacitat de paral·lelitzar processos i reduir el temps global d'execució. Inicialment es van pensar per accelerar la visualització de gràfics, és a dir, estaven dissenyades seguint una pipeline gràfica. Posteriorment, s'hi va incorporar la possibilitat d'executar codi de propòsit general, és a dir, es va modificar la pipeline per poder executar processos de còmput.

Apartat 5.2.1: Visió de la pipeline gràfica

A causa del disseny especialitzat de la GPU, aquesta és molt més rapida en cert tipus de tasques, com per exemple de gràfics (la visualització de models 3D), amb respecte a una CPU de propòsit general. Avui dia les *GPUs* poden processar deu milions de vèrtexs per segon i *rasteritzar* (transformar un model de vèrtexs i arestes a una imatge de píxels) milers de milions de fragments per segon. Seria ridícul intentar assolir aquestes velocitats amb una CPU. Per contra, les *GPUs* no poden executar els mateixos programes de propòsit general que una CPU.

Les *GPUs* modernes permeten programar-les a nivell de vèrtexs i píxels, a mès d'altres etapes que no detallarem en aquest document. Aquesta capacitat permet una representació més realista i detallada en un hardware genèric que tothom es pot permetre, en comparació amb les *GPUs* de *pipelines* de funcions fixes. En una *pipeline* de funcions fixes podem enviar informació a la *pipeline* i podem modificar l'estat, però no podem alterar el processat del vèrtexs i dels fragments. Un exemple d'aquesta restricció seria la del càlcul de il·luminació, el qual seria molt adequat calcular-la per cada vèrtexs o fragment.

Una *pipeline* programable permet un control complet de cada etapa mitjançant *shaders*, petits trossos de codi que un cop compilats, modificant alguna de les etapes de forma dinàmica. Els *shaders* de vèrtexs són programes que realitzen les transformacions dels vèrtexs i les normals, generació de coordinades de textura, i pre-càlculs d'il·luminació que normalment es calculen en l'etapa de processament geomètric. Els *shaders* de fragments o de píxels són programes que realitzen els càlculs en l'etapa de processament dels píxels de la *pipeline* gràfica i determinen exactament com s'il·lumina cada píxel, quina textura s'hi aplica, i si el píxel s'ha de dibuixar o no.

Els programes *shader* són enviats a la *GPU* per un programa executat en la *CPU*, però són executats en la *GPU* (veure Figura 5.7).

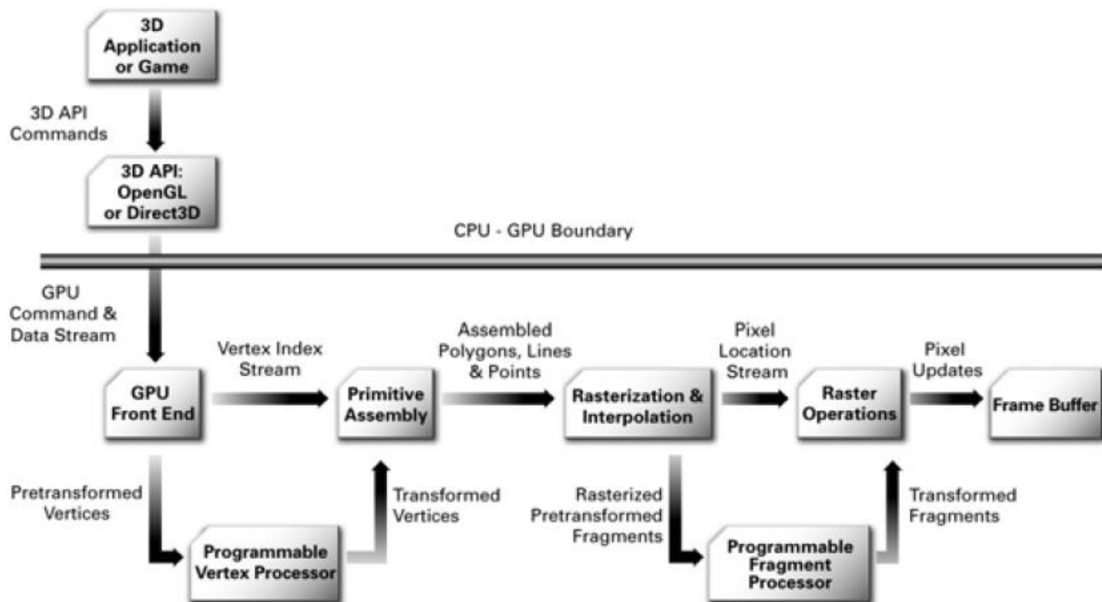


Figura 5.7: La *pipeline* gràfica programable simplificada.

Fins fa poc, molts dels programes de la *GPU* estaven escrits en llenguatge assemblador. Amb l'increment de comandes i característiques de les *GPUs*, ens hem trobat amb l'inconvenient de utilitzar aquest llenguatge.

Llenguatges per la *pipeline* gràfica

- OpenGL Shading Language (GLSL): llenguatge de codi lliure desenvolupat per OpenGL ARB basat en la sintaxis de C, desenvolupat amb la finalitat de proporcionar als desenvolupadors un control més directe sobre la *pipeline* gràfica.
- High Level Shading Language (HLSL): llenguatge propietari de Microsoft molt similar al GLSL.
- C for Graphics (Cg): llenguatge propietari de Nvidia. Aquest és va sorgir de la col·laboració entre Microsoft i Nvidia per desenvolupar el HLSL, per tant és una variant d'aquest anterior.

La característica diferenciadora principal respecte els llenguatges de propòsit general és que aquests llenguatges de *shading* estan basats en còmputs amb un flux de dades i exprimeixen la possibilitat de paral·lelització, a causa del control *SIMD* o *Single-Instruction Multiple-Data*, de les *GPUs*.

Apartat 5.2.2: Estructura de les GPUs

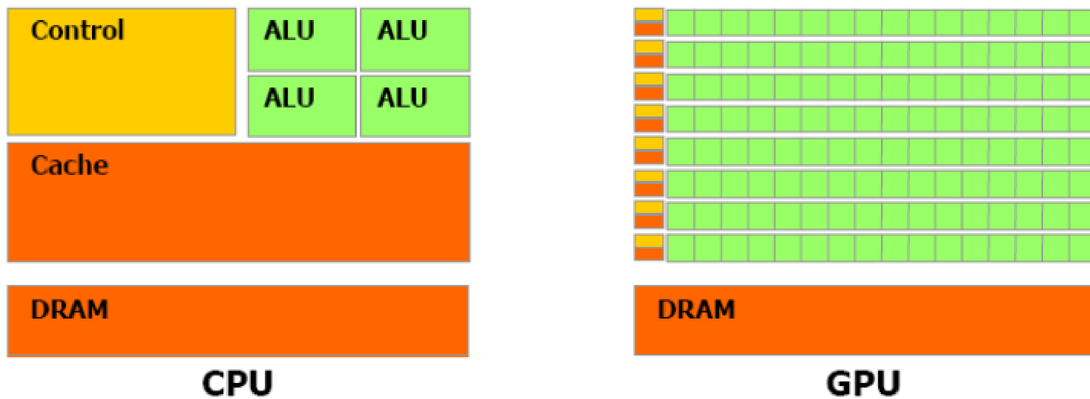


Figura 5.8: Comparació estructura CPU vs GPU

Com podem veure en la Figura 5.8, la principal diferencia entre la estructura clàssica de les *CPUs* i l'estructura de les *GPUs* és que les segones estan basades en l'ús de múltiples *Arithmetic Logic Units* (*ALUs*). El fet de tenir tantes *ALUS*, combinat amb una política *SIMD*, ens proporciona la capacitat de paral·lelitzar de les *GPUS*.

Apartat 5.2.3: GPUs de propòsit general (GPGPU), pipeline computacional

Les *GPUs* van començar a ser programables a partir del 2001. Les *GPUs* estan dissenyades específicament per gràfics, per aquest motiu són molt restrictives en termes d'operacions i programació. A causa de la seva naturalesa, les *GPUs* només són efectives en problemes que poden ser resolts mitjançant el processament per fluxos i on el hardware només es pot utilitzar de certes maneres.

Les *GPUs* només poden tenir processos amb independència de dades, però per contra en poden processar moltíssims de forma paral·lela. Això és molt efectiu quan el programador vol processar una gran quantitat d'informació de la mateixa manera. En aquest sentit, podem dir que les *GPUs* són processadors de fluxos (processador que pot treballar en paral·lel mitjançant un únic nucli sobre molts elements d'un flux, d'una sola vegada).

Un flux és simplement un conjunt d'elements que requereixen còmputs similars. Els fluxos proporcionen el paral·lelisme de les dades. El nucli és el conjunt d'operacions que s'aplicaran a cada element del flux. Ja que la GPU processa elements independents, no hi hauria d'haver dades compartides o estàtics, però ja veurem que no sempre és així. Per cada elements només podrem llegir de l'entrada de dades, realitzar operacions sobre l'element, i escriure els valors de sortida. Podem tenir múltiples entrada i múltiples sortides, però mai posicions de memòria de lectura i escriptura.

La intensitat aritmètica és defineix com el nombre d'operacions realitzades per paraula de memòria transferida. És molt important en les aplicacions *GPGPU* el tenir una intensitat aritmètica elevada, ja que sinó, la latència d'accés a memòria limitarà la velocitat de còmput.

Les aplicacions *GPGPU* ideals són aquelles que tenen un gran conjunt de dades amb el que treballar, un alt paral·lelisme, i una mínima dependència entre les dades.

Apartat 5.2.4: Llenguatges per la *pipeline* computacional

- **CUDA**: llenguatge propietari de CUDA que ens permet treballar amb codi de propòsit general, tot tenint accés al conjunt d'instruccions i la memòria dels elements en paral·lel de la GPU.
- **OpenCL**: llenguatge de codi lliure similar al CUDA, però amb la possibilitat de treballar sobre diferents tipus de plataforma. A diferencia de CUDA, OpenCL no està limitat a treballar només amb targes gràfiques de *Nvidia*.
- **OpenACC**: llenguatge de codi lliure similar als dos anteriors, però amb la finalitat de simplificar i fer més entenedor el codi a paral·lelitzar. Treballant amb OpenACC, podem definir macros en el nostre codi seqüencial, per indicar que aquell fragment de codi es paral·lelitzar

Apartat 5.2.5: Llenguatge *CUDA* i la seva arquitectura

CUDA és un llenguatge que treballa sobre una arquitectura de càlcul en paral·lel, desenvolupat per *NVIDIA*, que aprofita la potencia de la *GPU* per tal d'incrementar al màxim el rendiment del sistema.

Aquest llenguatge s'utilitza en sistemes on és realitza un "coprocessament" repartit entre la *CPU* i la *GPU*. *CUDA* ens permet un control total del conjunt d'instruccions de la *GPU* i de la memòria d'aquesta, però no només per treballar la part gràfica del sistema, sinó també per treballar amb aplicacions de propòsit general (*GPGPU*). *CUDA* proporciona unes quantes extensions per tal que el programador pugui implementar el paral·lelisme tot emprant altres llenguatges d'alt nivell com el *C*, *C++* o *Fortran*. Tot i poder treballar amb *CUDA* en qualsevol sistema operatiu, ens trobem amb la restricció de posseir una tarja gràfica de *NVIDIA*, més concretament una de les gammes *GeForce*, *ION Quadro* o *Tesla GPUs*.

Per poder endinsar-nos una mica més en el llenguatge, a part de recordar que per treballar amb la GPU necessitem definir funcions nucli, hem de fer un cop d'ull a com esta distribuïda la memòria en la GPU (veure Figura 5.9)

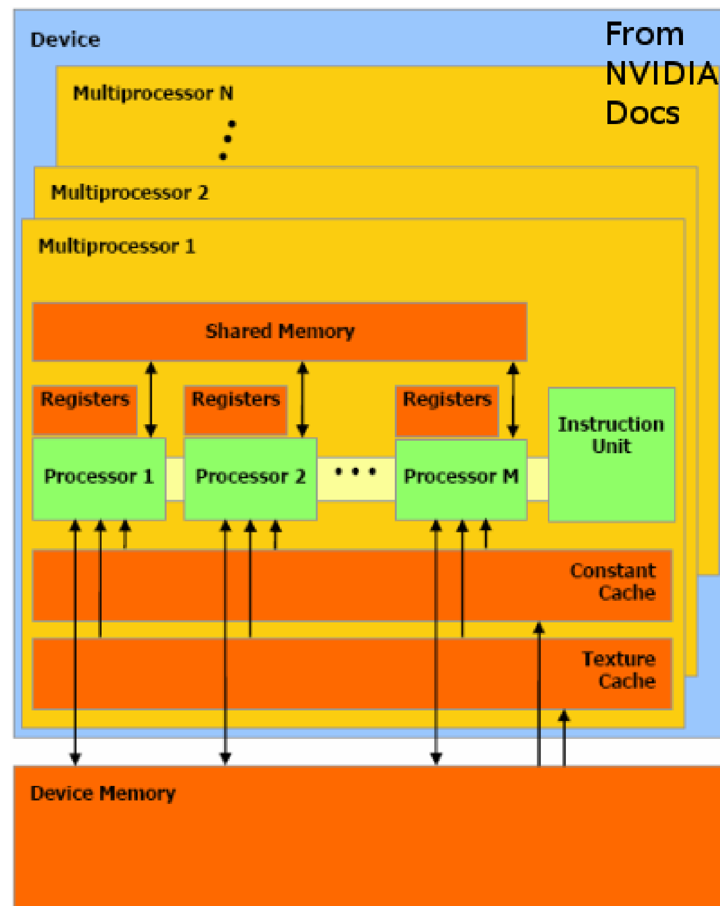


Figura 5.9: Distribució de memòria d'una GPU.

Com podem veure en la Figura 5.9, trobem principalment tres tipus de memòria el primer és la *Device memory* o *global memory*, en segon lloc tenim la *shared memory* i per últim tenim els *registers*:

- *Global memory*: és aquella memòria accessible per tots els *threads* executats en la *GPU*.
- *Shared memory*: és aquella memòria accessible per tots els *threads* que pertanyen al mateix *block*.
- *Registers*: és l'espai a memòria privat per cada *thread*.

Característiques principals de CUDA

Estructura bàsica

Tota aplicació que vulgui treballar amb CUDA, haurà de seguir els següents passos:

1- Reservar espai de memòria a la GPU i copiar les dades d'entrada.
2- Indicar a la GPU el nucli ha executar per cada thread
3- Copiar els resultats de la GPU i allibera l'espai ocupat.

Aquest són els passos principals. Per fer-los, haurem d'utilitzar les següents funcions que ens proporciona CUDA. Amb **cudaMalloc(@variable, mida)** podem reservar espai de memòria a la GPU. Per contra si volem alliberar espai hem d'utilitzar la funció **cudaFree(@variable)**. Per copiar dades utilitzarem la funció **cudaMemcpy(@destí, @origen, mida, tipus)**, segons el tipus que utilitzem podrem copiar de CPU a GPU o viceversa, així doncs, els possibles valors del tipus són *cudaMemcpyHostToDevice*, *cudaMemcpyHostToHost*, *cudaMemcpyDeviceToHost* i *cudaMemcpyDeviceToDevice*.

Declaració de funcions

Pel que fa a la declaració de funcions, tenim tres tipus de funcions diferents, les funcions de CPU, funcions d'enllaç entre CPU i GPU i les funcions de GPU.

	Execution	Called
__device__ float DeviceFunc()	Device (GPU)	Device (GPU)
__global__ void KernelFunc()	Device (GPU)	Host (CPU)
__host__ float HostFunc()	Host (CPU)	Host (CPU)

Les funcions d'enllaç, com el nom indica, són les funcions que cridem des de la *CPU* per començar a executar codi en la *GPU*, són les funcions nucli. Cal tenir en compte que aquestes funcions sempre han de tenir un tipus de retorn buit. També cal esmentar que la definició __host__ és prescindible, ja que el compilador ho sobreentén, però tot i això és útil si volem sobreescriure la mateixa funció tant al *Host* com a *Device*.

Quan treballem amb les funcions de la *GPU*, necessitarem identificar el *thread* actual per poder treballar amb les dades adequades, per fer-ho haurem de calcular l'identificador del *thread* de la següent manera:

```
Int i = threadIdx.x + blockDim.x * blockIdx.x;
```

Un cop calculat el *thread*, haurem de validar que aquest estigui dins del rang vàlid de *threads*, ja que en molts cassos tindrem més *threads* dels necessaris. En cas de tenir una estructura de blocs com una matriu 2D, calcularem la fila i columna del thread de la següent manera:

```
int i = ( blockIdx.y * blockDim.y ) + threadIdx.y;
int j = ( blockIdx.x * blockDim.x ) + threadIdx.x;
```

Tipus de memòria en la GPU

Múltiples tipus de memòria en la GPU:

Variable Declaration	Memory	Scope	Lifetime	Cycles
Int LocalVar;	Registre	Thread	Thread	~1
__device__ __shared__ int SharedVar;	Shared	Block	Block	~5
__device__ int GlobalVar;	Global	Grid	Application	~500
__device__ __constant__ int ConstantVar;	Constant	Grid	Application	~5

Les variables les podrem declarar des del codi de CPU, si han de compartir informació, o des del codi de la GPU si la informació només ha de ser accedida per la GPU. Cal tenir en compte que la memòria a la GPU és molt limitada, per exemple la memòria compartida podrà guardar una *array* de 8x8 fins a 32x32.

A causa d'aquesta limitació d'espai, haurem de seguir estratègies per optimitzar el nostre codi en funció del espai màxim de memòria compartida. La estratègia més utilitzada és la de ajustar la informació necessària de cada *thread* a la mida de la memòria compartida i la de paral·lelitzar el procés de transmissió de la informació des de la memòria global a la memòria compartida de cada bloc.

Crida de funcions

Finalment, veurem com es fan les crides a les funcions d'enllaç des del codi *CPU*. Per fer la crida, haurem d'indicar la informació auxiliar de les dimensions de la graella de blocs de la *GPU* i les dimensions de cada bloc. De forma estàndard, com podrem treballar fins a 3 dimensions, calcularem aquest paràmetres de la següent manera:

```
dim3 DimGrid(
   (n-1)/mida Bloc + 1, → Dimensió X de la graella (indicarà quants blocs hi ha)
   1,                   → Dimensió Y de la graella
   1                    → Dimensió Z de la graella
);

dim3 DimBlock(
   256, → nº de thread per blocX
   1, → nº de thread per blocY
   1 → nº de thread per blocZ
);
```

Per tant, per fer la crida farem:

```
vecAddKernel<<<DimGrid, DimBlock>>>(params);
```

D'aquesta manera proporcionarem a la *GPU* la informació necessària per crear els blocs necessaris, amb els *threads* necessaris a cada bloc per poder treballar. Com hem vist, podem definir els nombre de *threads* que vulguem. Per tant, és molt provable que desaprofitem *threads*, d'aquí la necessitat de controlar que l'índex de cada *thread* estigui dins del rang de les dades amb les que treballem. Per optimitzar l'ús de la *GPU*, caldrà calcular dinàmica ment el nombre de *threads* i de blocs per tal de malgastar el mínim possible.

Apartat 5.2.6: *OpenGL*

OpenGL (***Open Graphic Library***) és una especificació estàndard que defineix una *API* multillenguatge i multiplataforma per desenvolupar aplicacions de visualització 2D o 3D. Aquesta *API* s'utilitza per interactuar amb la *GPU* i aconseguir l'acceleració de hardware que aquesta ens proporciona.

OpenGL va ser desenvolupat per *Silicon Graphics Inc.*, iniciat al 1991 i finalitzat el Gener del 1992, com una solució de codi lliure. Des de llavors, s'ha emprat en nombrosos camps com són les eines *CAD*, la realitat virtual, visualitzacions científiques, visualització d'informació, simuladors de vol, i videojocs. Actualment, la gestió d'aquesta *API* corre a càrrec de l'organització sense ànims de lucre *Khronos Group*.

Com hem explicat anteriorment, *OpenGL* té un llenguatge de *shaders* propi anomenat GLSL, però per facilitar el seu ús als programadors, s'han elaborat enllaços de llenguatges i diverses extensions. Alguns exemples d'aquest enllaços poden ser el *WebGL* per *Java* o el *WGL*, *GLX* i *CGL* per *C*, entre d'altres. Pel que fa a les extensions, hem cercat algunes per treballar amb *C*:

- GLEW: aquesta és una extensió *d'OpenGL* que ens permet determinar quines extensions d'OpenGL suporta la plataforma de treball.
- GLFW: aquesta és una extensió *d'OpenGL* que ens permet gestionar les interrupcions tant de teclat com de ratolí. *OpenGL* per si mateix no implementa aquest tipus de gestió, però si volem realitzar una aplicació mínimament interactiva, necessitem poder definir el comportament d'aquestes interrupcions.
- GLM: aquesta és una extensió *d'OpenGL* que ens permet fer càlculs matemàtics de geometria espacial, molt útils per les representacions 3D.

Característiques principals *d'OpenGL*

Per entendre les característiques principals *d'OpenGL*, veiem un exemple d'una pipeline gràfica:

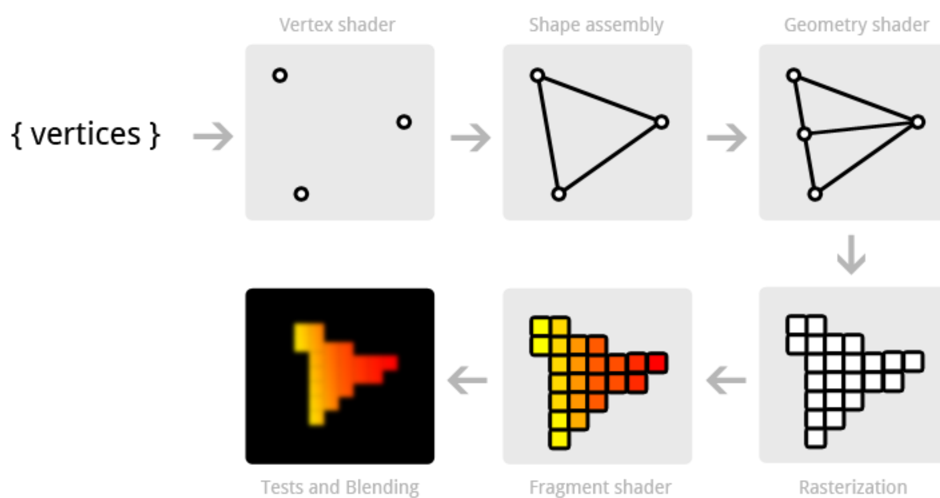


Figura 5.10: Exemple *pipeline* gràfica.

Com veiem en la Figura 5.10, principalment necessitem definir els vèrtexs del model a representar. Per altre banda, també necessitarem definir quin color ha de tenir cada vèrtexs. Finalment, si és tracta d'un polígon més complex, haurem de definir cadascun dels triangles que el formen com un conjunt de cares.

Si volem dibuixar un polígon, OpenGL ens proporciona les següents primitives per dibuixar-lo:

```
glBegin(tipus);
    //Definim els vèrtexs del polígon i un color per tots o un color per cada vèrtexs
    glColor3f(red, green, blue);
    glVertex3f(coordX, coordY, coordZ);
    ...
glEnd();
```

Com a tipus, podem treballar amb punts, segments, triangles o polígons. Pel que fa a la definició dels vèrtexs i colors, tenim diferents instancies de les funcions, en l'exemple utilitzem les "3f" tres paràmetres de coma flotant, podem treballar des de 2 paràmetres (per coordenades 2D) a 4 paràmetres (per coordenades uniformes) i els valors poden ser short, int, float o double.

Quan treballem amb models més complexes, ens interessa un mètode més sofisticat per dibuixar, en aquest cas treballarem amb buffers de vèrtexs, de colors i de cares. Per fer-ho, el primer pas serà el d'activar els buffers de vèrtexs i de colors:

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
```

El següent pas serà el de inicialitzar els vèrtexs, els colors i les cares, i carregar aquesta informació, a excepció de les cares:

```
glColorPointer(num_components, tipus, stride, @buffer);
glVertexPointer(3, GL_FLOAT, 0, vertices);
```

Finalment dibuixarem el model a partir de les cares:

```
glDrawElements(elements_dibuix, mida, tipus_dades, @buffer);
p.e.: glDrawElements(GL_TRIANGLES, sizeof(indices), GL_UNSIGNED_INT, indices);
```

Capítol 6: Requisits del sistema

En aquest capítol, descriurem els requisits que ha de complir el sistema, tant funcionals com no funcionals. Aquesta etapa ha de permetre saber tot allò que cal fer perquè el projecte compleixi tots i cadascun dels seus objectius, tant des del punt de vista de programari com de maquinari. També emmarcarem els requisits en l'entorn on es desenvolupa el projecte.

Apartat 6.1: Plantejament de la problemàtica

El problema que volem solucionar és la simulació de vent a partir d'una escena (veure Figura 6.1).

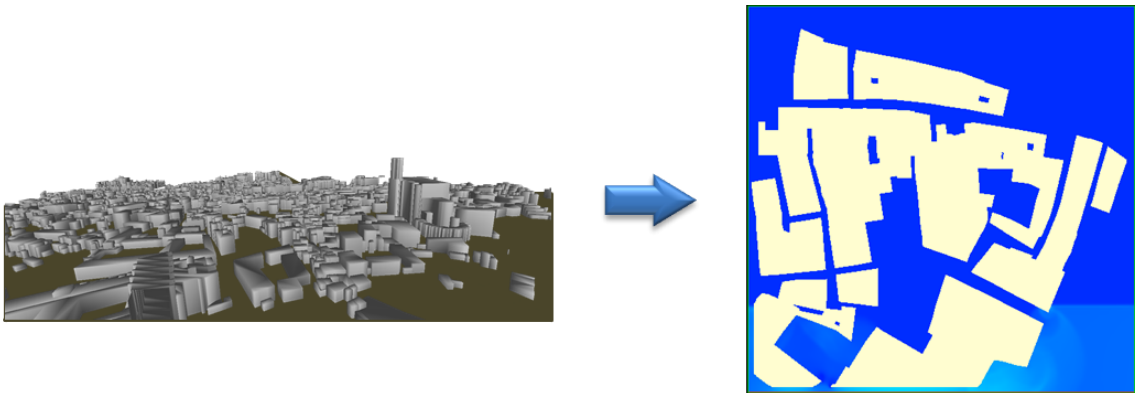
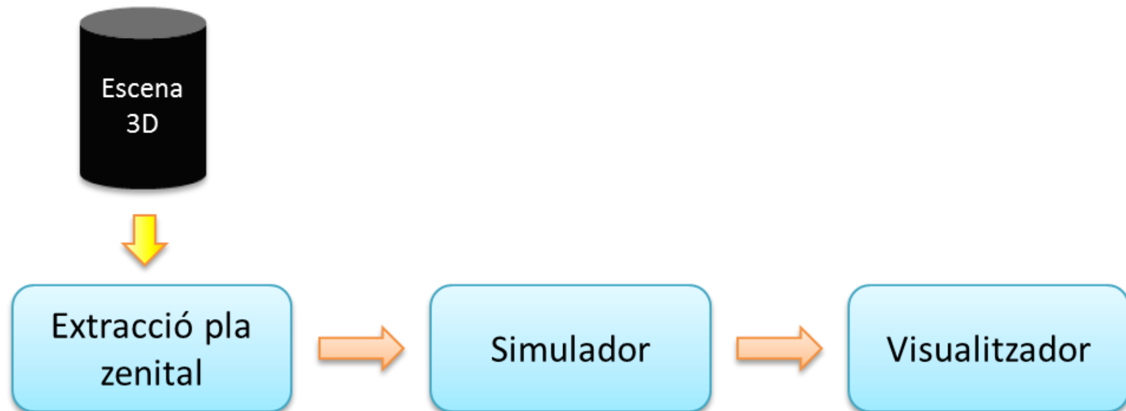


Figura 6.1: Exemple de simulació

Com podem observar en l'exemple, els principals punts de conflicte que hem de tractar, són l'extracció de la informació necessària de l'escena per poder realitzar la simulació, crear el nucli del sistema que permeti realitzar simulacions realistes i la posterior visualització dels resultats. A continuació explicarem els diferents requisits del projecte.

Apartat 6.2: Plantejament de la solució

La solució que plantegem és la següent:



Com veiem en l'esquema, partirem d'una escena 3D. El primer pas que realitzarem serà el d'extreure el pla zenital de l'escena. Gràcies a això obtindrem el circuit d'obstacles per on haurà de passar el vent. El següent pas serà el d'aplicar la simulació sobre el pla anterior. Com ja hem explicat prèviament, utilitzarem el mètode de *Lattice-Boltzmann* per realitzar una simulació realista. Finalment, mitjançant funcionalitats d'OpenGL per visualitzar, mostrarem els resultats de la simulació.

Apartat 6.3: Requisits Funcionals

Els requisits funcionals expliquen que ha de fer l'aplicació, és a dir, totes les funcionalitats que tindrà, sense especificar com les farà. Per tant, les funcionalitats a desenvolupar són:

- Obtenció de la informació bàsica de l'escena.
    - Lectura de fitxers d'imatge amb el pla zenital de l'escena.
    - Extracció de la geometria de la imatge.
- Implementació del mètode de Lattice-Boltzmann per realitzar la simulació.
- Permetré modificar la direcció del vent en temps real.
- Visualització dels resultats de la simulació.

Apartat 6.4: Requisits No Funcionals

En aquest apartat definirem com ha de ser l'aplicació i no què és el que ha de fer. Aquests requisits fan referència al que hem de tenir en compte a l'hora de dissenyar el sistema, com ara els recursos.

Els requisits no funcionals són els següents:

- La implementació del mètode de simulació ha de ser el més eficient possible per poder obtenir una simulació en temps real.
- La interacció usuari-aplicació ha de ser el més intuïtiva possible i de fàcil comprensió.

Els requeriments de hardware per la nostra aplicació són:

- Sistema operatiu Windows.
- Tarja gràfica *Nvidia* amb capacitat per CUDA.

Capítol 7: Estudis i decisions

En aquest capítol descriurem el maquinari, les llibreries i/o el programari utilitzats durant el desenvolupament del projecte, i justificarem la seva elecció. En el cas de les llibreries que hem emprat, n'explicarem el seu funcionament bàsic per una total comprensió d'aquesta.

Apartat 7.1: Microsoft Visual Studio 2012

Microsoft Visual Studio és un entorn de desenvolupament integrat (IDE, per les seves sigles en anglès) de *Microsoft*. Aquest s'utilitza per desenvolupar programes d'ordinador per la família de sistemes operatius *Microsoft Windows*, també permet desenvolupar pagines web, aplicacions web i serveis web. Visual Studio utilitza diverses plataformes de desenvolupament de software com *Windows API*, *Windows Forms*, *Windows Presentation Foundation*, *Windows Store* i *Microsoft Silverlight*. Aquest permet produir tant codi natiu com codi interpretat.

Visual Studio inclou un editor de codi amb suport de *IntelliSense* i *code refactoring*. El *debugger* que porta incorporat permet treballar tant a alt nivell com a nivell de màquina. Altres eines integrades que incorpora formularis de disseny per aplicacions amb interfície gràfica, dissenyador web, dissenyador de classes, i un dissenyador d'esquema de bases de dades. *Visual Studio* accepta *pluig-ins* per incorporar funcionalitats a quasi tots els nivells. Per exemple, permet incorporar eines per control de versions (com *Subversion* o *Visual SourceSafe*), per treballar amb nous llenguatges, per gestionar el cicle de desenvolupament del software, etc.

Visual Studio suporta diferents llenguatges de programació i permet editors de codi i suport per *debugger* de quasi qualsevol llenguatge de programació. Els llenguatges incorporats per defecte son *C*, *C++* i *C++/CLI* (via *Visual C++*), *VB.NET* (via *Visual Basic .NET*), *C#* (via *Visual C#*), i *F#* (des de *Visual Studio* 2010). El suport per altres llenguatges com *M*, *Python*, *Ruby* i *CUDA*, entre d'altres, només estan disponibles a partir d'instal·lacions a part. També té suport per *XML/XSLT, HTML/XHTML,JavaScript* i *CSS.*

Microsoft proporciona versions "*Express*" de *Visual Studio* sense cap cost. També proporciona de forma gratuïta llicencies comercials de *Visual Studio* a estudiants mitjançant el seu programa *DreamSpark*.

Característiques

Editor de codi

Com qualsevol altre *IDE*, *Visual Studio* incorpora un editor de codi amb suport per *syntax highlightning* i auto completat de codi mitjançant *IntelliSense* tant per variables, funcions, mètodes, bucles i queris. Els suggeriments d'auto completat és mostren en una *list box*.

L'editor de codi de *Visual Studio* també permet afegir marcadors en el codi per a una navegació ràpida. Altres ajudes de navegació són el poder col·lapsar blocs de codi i una cerca incremental, a part de la cerca normal de text i la cerca per regeix. L'editor de codi de *Visual Studio* també és compatible amb la refactorització de codi incloent paràmetres de reordenament, renombrament de variables i mètodes, extracció d'interfícies i encapsulació de classes dins de propietats, entre d'altres.

Visual Studio també incorpora la compilació en *background*. Al mateix temps que escrivim el codi, *Visual Studio* el compila en *background* amb la finalitat de trobar errors sintàctics i errors de compilació. Aquest mètode de compilació no genera cap codi executable, per això hem d'utilitzar un altre compilador.

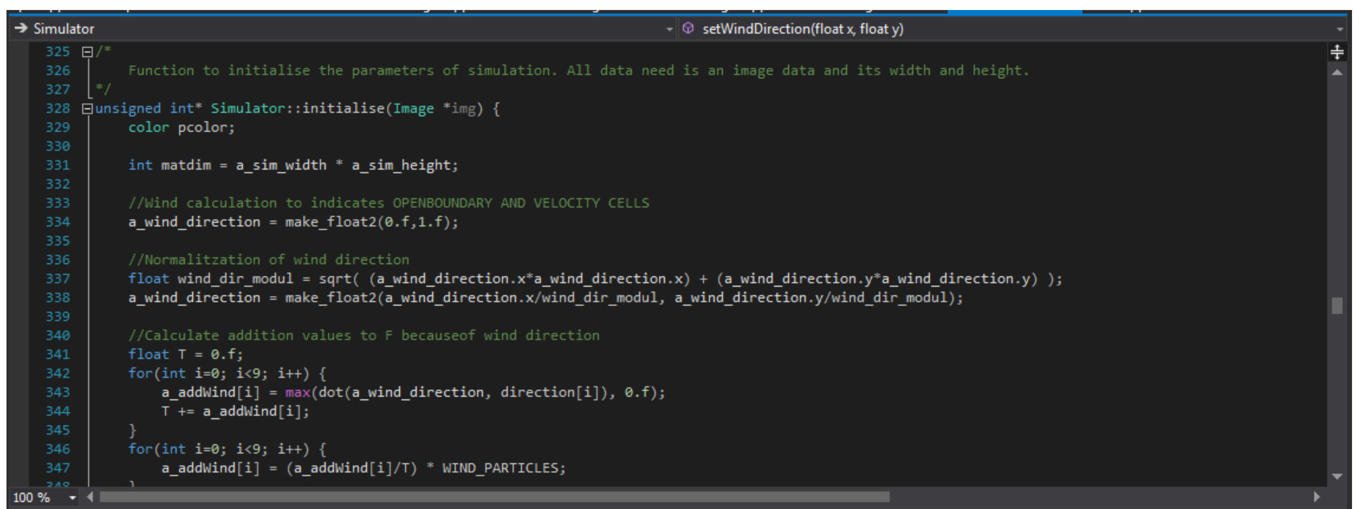


Figura 7.1: editor de codi de Visual Studio 2012

Debugger

Visual Studio inclou un *debugger* que pot treballar a alt nivell com a nivell màquina. Aquest funciona tant amb codi interpretat com amb codi natiu i pot ser utilitzat per *debugar* aplicacions escrites en qualsevol llenguatge suportat per *Visual Studio*. A més, aquest es pot enllaçar amb un procés en execució i monitoritzar i *debugar* el procés. Si el codi del procés és accessible, pot mostrar el codi que s'està executant a cada moment. El *debugger* es pot configurar per que s'executi quan una aplicació fora de l'entorn de Visual Studio falli.

El *debugger* permet afegir punts de control (els quals permeten parar l'execució temporalment) i observar l'estat de les variables. Els punts de control poden ser condicionats, això vol dir que s'activaran sempre que es compleixi la condició establerta. El codi es pot *debugar* línia a línia. També es pot entrar en les funcions per *debugar* el seu codi o passar per sobre i executar el codi intern d'una passada. Durant el procés de *debugació*, situant el cursor sobre una variable, podem veure el valor d'aquestes.

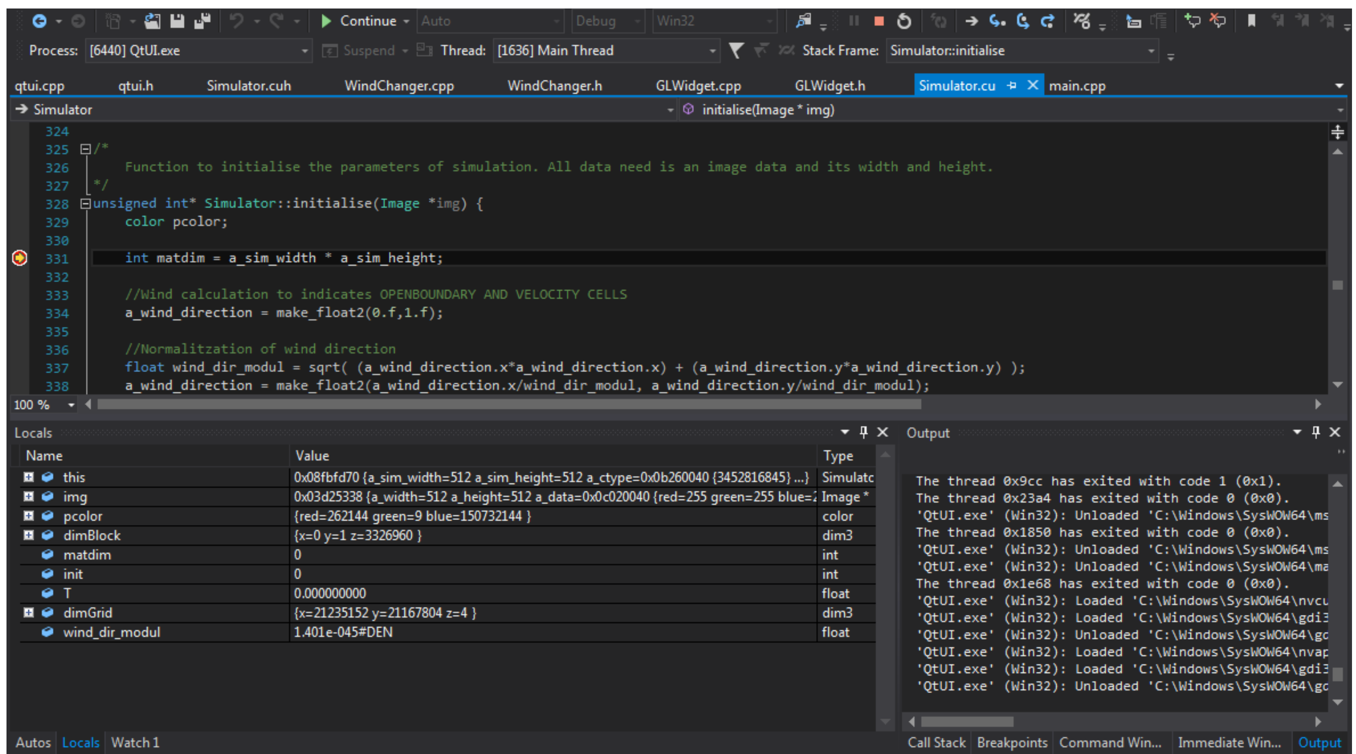


Figura 7.2: *debugger* de Visual Studio 2012

Dissenyadors

Visual Studio inclou un conjunt de dissenyadors visuals per ajudar en el desenvolupament d'aplicacions. Alguns dels disseny inclosos són:

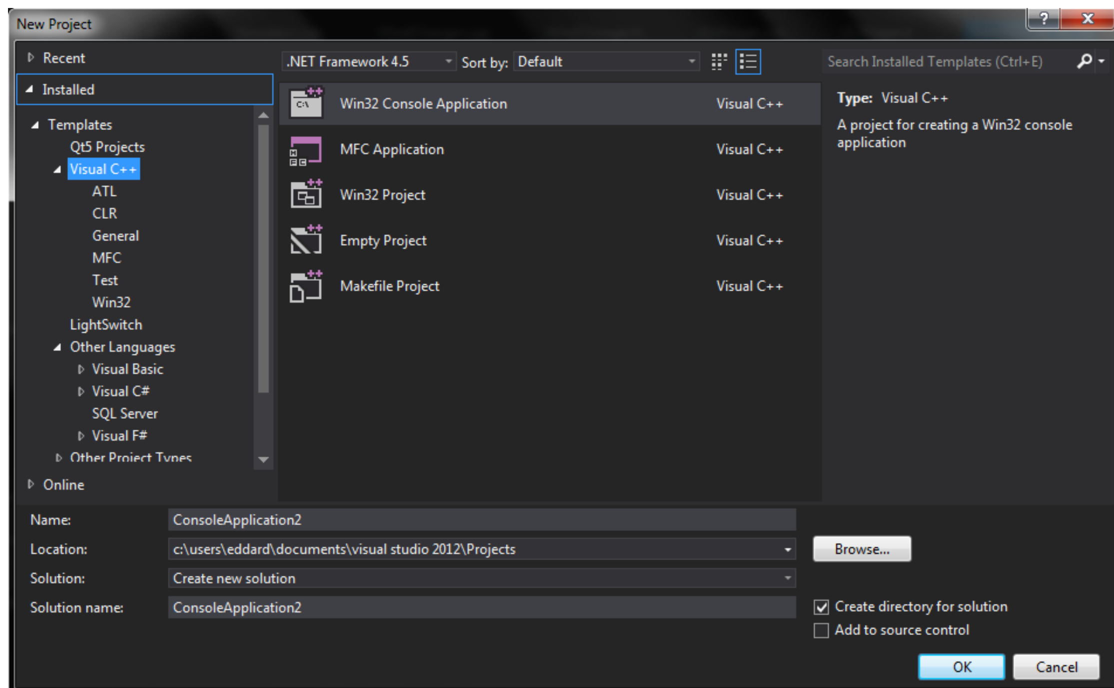


Figura 7.3: Menú de creació de nou projecte, on es pot seleccionar diferents dissenyadors.

Dissenyador *Windows Forms*

El dissenyador *Windows Forms* s'utilitza per construir aplicacions amb interfície gràfica mitjançant *Windows Forms*. EL disseny pot ser controlat encapsulant els controls dins de contenidors o bloquejant-los al costat del formulari. El controls que mostren informació (com *textbox*, *list box*, *grid view*, etc.) es poden fusionar amb gestors de dades com bases de dades o queris. Aquest controladors fusionats es poden crear arrossegant objectes des de la finestra de dades sobre el disseny de la interfície. La interfície queda enllaçada amb el codi mitjançant el model de programació d'esdeveniments. El dissenyador pot generar codi en *C#* o *VB.NET* per l'aplicació.

Dissenyador *WPF*

El dissenyador *WPF*, nom en clau *Cider*, es va introduir en la versió 2008. Al igual que el dissenyador *WindowsForms*, aquest suporta la metàfora del agafar i arrossegar. Aquest s'utilitza per les interfícies d'usuari propietàries de *Windows Presentation* Foundation. Suporta totes les funcionalitats pròpies de *WPF* incloent l'enllaç de dades i la gestió automàtica de disseny. El codi que genera per la interfície està en format *XAML*. Aquests fitxers de codi son compatibles amb *Microsoft Expression Design*.

Dissenyador/desenvolupament web

Visual Studio inclou un editor i un dissenyador de llocs webs que permeten crear pagines web a partir de *widgets*. S'utilitza per desenvolupar aplicacions *ASP.NET* i suporta *HTML*, *CSS* i *JavaScript*.

Dissenyador de Classe

El dissenyador de classe esta pensat per crear i editar les classes (incloent els seus membres i accessos) mitjançant models *UML*. Aquest dissenyador pot generar l'estructura buida de les classes i els seus mètodes en *C#* i *VB.NET*. També pot fer el procés invers i construir el diagrama de classes a partir del nostre codi.

Dissenyador de dades

El dissenyador de dades permet editar de forma gràfica l'esquema d'una base de dades, incloent taules tipades, claus primaries, claus foranes i restriccions. També el podem utilitzar per dissenyar queris a partir de la visualització de la base de dades.

Dissenyador *mapping*

Des del *Visual Studio* 2008, el dissenyador de *mapping* ens permet establir les correspondències entre les bases de dades i les classes que encapsulen les dades.

Altres eines

Navegador de pestanyes obertes

Podem tenir diverses pestanyes amb un document a cadascuna i podem intercanviar la pestanya amb la que treballem en cada moment. Podem invocar aquesta eina mitjançant *CTRL+TAB*.

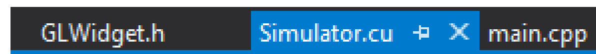


Figura 7.4: Sistema de pestanyes de Visual Studio 2012.

Editor de propietats

L'editor de propietats ens permet llista totes les propietats disponibles de tots els objectes de la solució i mitjançant una interfície gràfica ens permet editar-les.

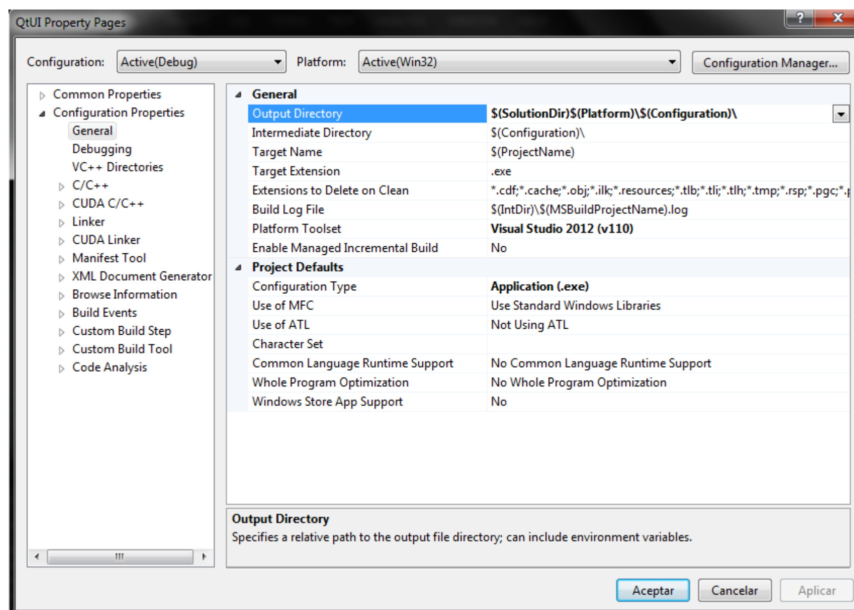


Figura 7.5: Editor de propietats de Visual Studio 2012.

Explorador de la solució

En *Visual Studio*, una solució és un conjunt de fitxers de codi i altres recursos utilitzats per crear una aplicació. Els fitxers d'una solució estan estructurats jeràrquicament. L'explorador de la solució ens permet gestionar i navegar dins els fitxers de la solució.

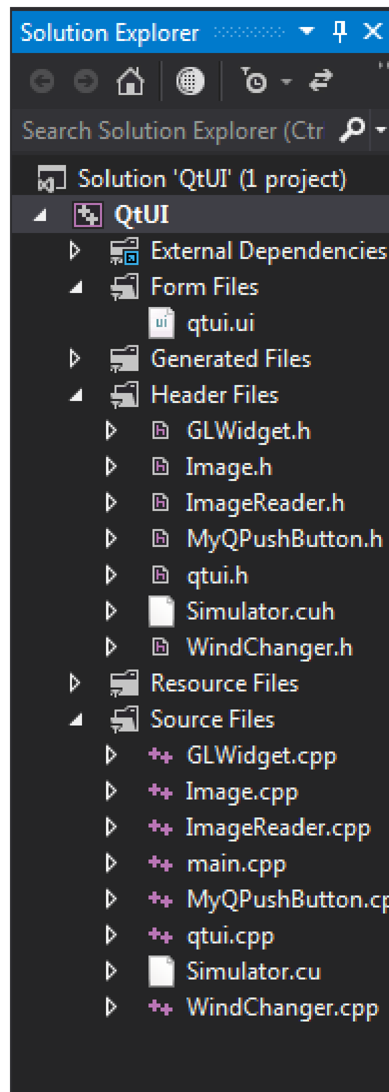


Figura 7.6: Explorador de la solució de Visual Studio 2012.

Explorador de dades

L'explorador de dades s'utilitza per gestionar bases de dades en instancies de *Microsoft SQL Server*. Aquest ens permet la creació i modificació de taules de la base de dades (emprant comandes *T-SQL* o el dissenyador de dades). També el podem utilitzar per crear queris i emmagatzemar procediments.

Explorador de servidor

L'explorador de servidors s'utilitza per administrar les connexions de base de dades en un equip accessible. També s'utilitza per navegar pels serveis de *Windows* en execució, per comptabilitzar el rendiment, per registrar els esdeveniments de *Windows* i les cues de missatges, o utilitzar-lo com una font de dades.

Framework per la generació de text

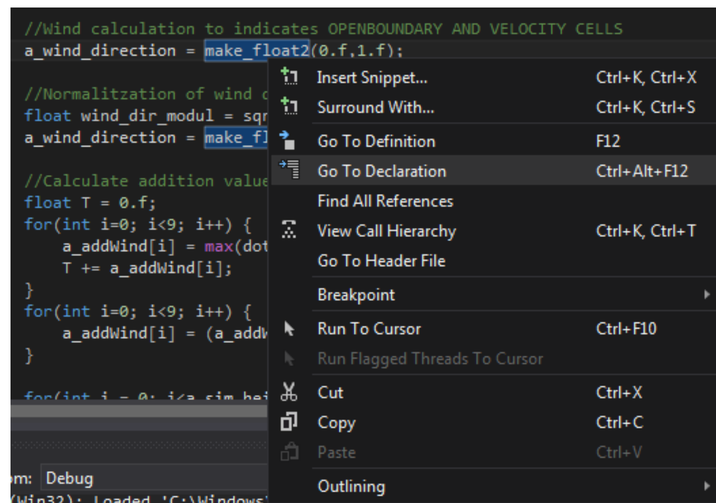
Visual Studio incorpora un *framework* per la generació de fitxers de text anomenat *T4*, el qual permet generar fitxers de text a partir de plantilles des de *l'IDE* o des de codi.

Novetats versió 2012

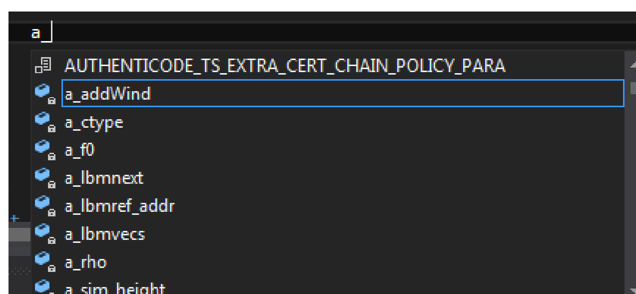
- **Coloració semàntica**: millora la coloració de la sintaxis, permet definir diferents colors per la sintaxis de *C++* com poden ser els macros, les enumeracions, el nom dels tipus, les funcions, etc.
- **Ressaltar referencies**: ens permet seleccionar un símbol i ens ressalta l'aparició d'aquest símbol dins del mateix fitxer.

```
//Wind calculation to indicates OPENBOUNDARY AND VELOCITY CELLS
a_wind_direction = make_float2(0.f,1.f);

//Normalitzation of wind direction
float wind_dir_modul = sqrt( (a_wind_direction.x*a_wind_direction.x) + (a_wind_direction.y*a_wind_direction.y) );
a_wind_direction = make_float2(a_wind_direction.x/wind_dir_modul, a_wind_direction.y/wind_dir_modul);

//Calculate addition values to F becauseof wind direction
float T = 0.f;
for(int i=0; i<9; i++) {
    a_addWind[i] = max(dot(a_wind_direction, direction[i]), 0.f);
    T += a_addWind[i];
}
```

- **Nou explorador de solucions**: ara podem seguir la traça del codi saltant a la definició d'un mètode o a les crides del mètode.



- **Visualització automàtica de la llista *IntelliSense***: quan escrivim codi, de forma automàtica se'ns mostra un llistat amb els valors possibles del que volem escriure, sense necessitat de combinacions de tecles.



- **Filtrat dels membres de la llista**: *IntelliSense* utilitza *fuzzy logic* per determinar quines funcions/variables/tipus ens mostrarà en el llistat.
- **Fragments de codi**: podem configurar el *IntelliSense* per que ens generi de forma automàtica codi rellevant .

Motius de selecció

Hem decidit fer servir aquesta IDE i no una altre per diferents motius. En primer lloc, estem parlant d'una IDE amb molts anys de desenvolupament i per tant moltes funcionalitats. També té una comunitat molt gran darrera, la qual cosa ens permet trobar solucions a problemes amb la IDE de forma molt rapida. Per altre banda, el fet de ser gratuïta per estudiants ens ha estat molt temptador.

Per tant, al ser una eina molt completa i amb el suport tant de Microsoft com de la comunitat d'internet al darrera, l'entorn de desenvolupament escollit ha estat el Microsoft Visual Studio 2012.

Apartat 7.2: C++

C++ és un llenguatge de programació que va ser creat, com el seu predecessor C, als laboratoris Bell de AT&T. El seu autor principal va ser Bjarne Stroustrup. L'any 1980 es van afegir noves característiques al llenguatge C, entre les principals la integració de les classes, idea que va ser presa de Simula67 (per molts considerat el primer llenguatge orientat a objectes). A partir d'aquí va anar evolucionant fins que a l'any 1985 va ser consolidat com un llenguatge orientat a objectes i anomenat C++.

Actualment existeix un estàndard, anomenat ISO C + +, al qual s'han adherit la majoria dels fabricants de compiladors més moderns. N'existeixen també alguns intèrprets, com ara ROOT.

Una particularitat del C++ és la possibilitat de redefinir els operadors (sobrecàrrega d'operadors), i de poder crear nous tipus que es comportin com tipus fonamentals.

El nom C++ va ser proposat per Rick Mascitti l'any 1983, quan el llenguatge va ser utilitzat per primera vegada fora d'un laboratori científic. Abans s'havia fet servir el nom "C amb classes". En C++, l'expressió "C++" significa "increment de C" i es refereix al fet que C++ és una extensió de C.

C++ té els següents tipus fonamentals:

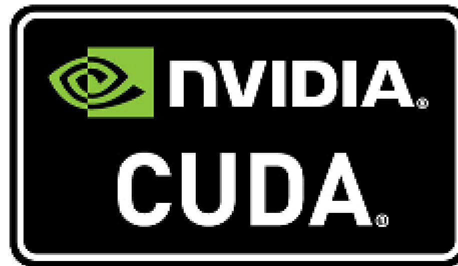
- Caràcters: *char* (també és un enter), wchar_t
- Enters: *short int, int, long int, long long int*
- Nombres de coma flotant: *float*, *double*, *long double*
- Booleans: *bool* (cert o fals)
- Buit: *void*

Per més informació, veure l'Annex A.

Motius de selecció

El motius per escollir el C++ com a llenguatge de programació del nostre projecte han estat molt variats. Per una banda, C++ és un llenguatge modern i molt complet, que ens permet desenvolupar quasi qualsevol codi. A més, aquest disposa de un gran nombre de llibreries per realitzar tasques ja desenvolupades per tercers. També hem de tenir en compte que el C++ esta present en la majoria de sistemes operatius. Per altre banda, al tenir coneixements previs tant de C com de C++ ens ha semblat convenient la seva elecció.

Apartat 7.3: *CUDA*



Com ja hem explicat en l'apartat 5.2.5, *CUDA* és un llenguatge per a *GPGPU* el qual ens permet paral·lelitzar processos de càlcul, els quals són repetitius, però treballen amb dades diferents. *CUDA* està basat en el llenguatge C, amb la particularitat que ens permet definir *kernels* que s'executaran en la *GPU* en comptes de a la *CPU*.

Motius de selecció

Vam escollir treballar amb *CUDA* pel fet que és un dels pocs llenguatges per a *GPGPU* amb suport per part d'una gran empresa com *NVidia*. És cert que aquesta decisió ens limita només a treballar amb targes gràfiques de l'empresa en qüestió, cosa que el *OpenCL* no, però el mercat de les targes gràfiques, avui dia, està dominat per NVidia, per tant no és cap problema.

Apartat 7.4: *FreeImage*

FreeImage és una llibreria de codi obert que pretén facilitar, als desenvolupadors, el treball amb formats d'imatges com *PNG*, *BMP*, *JPEG*, *TIFF* i d'altres en les seves aplicacions multimèdia. *FreeImage* és fàcil d'utilitzar, ràpid, segur de treballar en *multithreading*, compatible amb versions tant de 32 bits com de 64 bits de *Windows*, i multi plataforma (funciona també sobre *Linux* o *Mac OS X*).

Gràcies a la seva interfície *ANSI C*, *FreeImage* pot ser utilitzat en molts llenguatges diferents incloent *C*, *C++, VB*, *C#*, *Delphi*, *Java* i també en llenguatge de *scripting* com *Perl*, *Python*, *PHP*, *TCL* o *Ruby*.

Des del gener del 2000 al juliol del 2002, *FreeImage* va ser dissenyat i desenvolupat per *Floris van den Berg*. Avui dia l'encarregat del seu manteniment és *Hervé Drolon*.

Característiques

Fàcil d'utilitzar

La llibreria ha estat dissenyada per ser extremadament fàcil d'utilitzar. El seu lema és: "*make difficult things simple instead of simple things difficult*."

No limitat al PC local

La estructura única de *FreeImageIO* fa possible carregar imatges des de qualsevol lloc. Entre les possibilitats disponibles tenim fitxers independents, en memòria, en una base de dades o en internet, tot això sense haver de recompilar la llibreria.

Plugin Driven

El motor intern està fet completament modular utilitzant un sistema de *plugins* intel·ligent. Permet escriure fàcilment nous *plugins* i guardar-los en arxius DLL o incrustar els *plugins* directament en l'aplicació.

Conversió de colors

FreeImage proporciona un conjunt de funcions per convertir a bitmap des de un bitdepth a un altre. La llibreria suporta estàndards d'imatges de 1-, 4-, 8-, 16-, 24- i 32-bit, de la mateixa manera que enters, reals i imatges complexes.

Funcions de manipulació

Proporciona rutines bàsiques de manipulació d'imatges com poden ser rotacions, escalats, translacions o ajustos de colors, així com transformacions JPEG sense pèrdua d'informació.

Accés directa als bits dels *bitmaps* i a la paleta

Proporciona funcions que permeten un accés directe a la paleta de colors del *bitmap* i als propis bits del *bitmap*.

Suport per metadades

Permet parcejar metadades incorporades al *bitmap*. *FreeImage* suporta comentaris, *Exif* (incloent GPS i notes del fabricant),*IPTC*, *Adobe XMP*, *GeoTIFF* i metadades d'animacions *GIF*.

Suport per imatges d'alt rang dinàmic

FreeImage admet imatges RGB amb valors de punt flotant, així com imatges HDR de 48 bits, i ofereix operadors de mapeig de tons per convertir aquestes imatges a imatges LDR 24 bits.

Suport per a arxius *RAW* de càmeres digitals

FreeImage pot carregar arxius RAW de càmeres de fotos digitals (*CRW*/*CR2*, *NEF*, *RAF*, *DNG*, *MOS*, *KDC*, *DCR*, etc.), s'admeten pràcticament tots els formats *RAW*.

Permet consultar el codi font complet.

Està escrit en C++ portàtil, per la qual cosa la llibreria ha de compilar en tots els sistemes operatius: Windows 32 bits o 64 bits, Linux i Mac OSX.

FreeImage també inclou una àmplia documentació per ajudar a treballar-hi i aprendre sobre el codi proporcionat.

Integració amb *DirectX* i *OpenGL*.

Només cal una mínima programació per emmagatzemar un mapa de bits *FreeImage* en una superfície *DirectDraw* o utilitzar *FreeImage* per carregar les seves textures *Direct3D*/*OpenGL*.

Formats Suportats

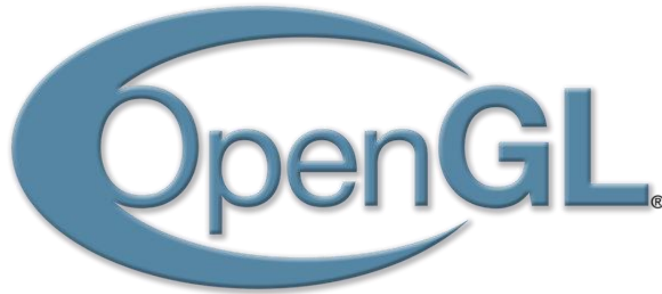
- BMP files [reading, writing]
- Dr. Halo CUT files [reading] *
- DDS files [reading]
- EXR files [reading, writing]
- Raw Fax G3 files [reading]
- GIF files [reading, writing]
- HDR files [reading, writing]
- ICO files [reading, writing]
- IFF files [reading]
- JBIG files [reading, writing] **
- JNG files [reading, writing]
- JPEG/JIF files [reading, writing]
- JPEG-2000 File Format [reading, writing]
- JPEG-2000 codestream [reading, writing]
- JPEG-XR files [reading, writing]
- KOALA files [reading]
- Kodak PhotoCD files [reading]
- MNG files [reading]
- PCX files [reading]
- PBM/PGM/PPM files [reading, writing]
- PFM files [reading, writing]
- PNG files [reading, writing]
- Macintosh PICT files [reading]
- Photoshop PSD files [reading]
- RAW camera files [reading]
- Sun RAS files [reading]
- SGI files [reading]
- TARGA files [reading, writing]
- TIFF files [reading, writing]
- WBMP files [reading, writing]
- WebP files [reading, writing]
- XBM files [reading]
- XPM files [reading, writing]

* només en escala de grisos
** només a partir de plugins externs, pot requerir llicencia comercial

Motius de selecció

Vam començar cercant llibreries que ens permetessin fer la lectura d'imatges en el nostre sistema, però totes estaven escrites per un format d'imatge en particular o eren molt complexes d'utilitzar. A diferencia d'aquestes, *FreeImage* té un major grau d'abstracció i és una llibreria molt simple que ens permet treballar amb totes aquestes llibreries més especifiques que havíem provat prèviament. Així doncs, la decisió va ser clara.

Apartat 7.5: OpenGL



Com ja hem explicat en l'apartat 5.2.6, *OpenGL* és una *API* que treballa sobre la *pipeline* gràfica de la GPU amb la finalitat de processar gràfics de forma ràpida.

Motius de selecció

Vam escollir treballar amb *OpenGL* perquè és una API ben documentada i amb una amplia comunitat on trobar solucions als problemes que ens puguin sortir. Per altre banda, es pot utilitzar de forma gratuïta,

Apartat 7.6: Qt

Qt és un marc de treball multi plataforma per la creació de programes amb Interfície gràfica d'usuari (GUI en anglès). Consisteix en unes llibreries escrites en C++.

Qt està disponible amb llicències comercial i amb llicència acadèmica. Totes les edicions suporten diversos compiladors incloent el GNU Compiler Collection C++ i el Visual Studio.

Actualment Qt és desenvolupat per Digia, que és propietària de la marca, i el Qt Project que té un sistema de govern obert i inclou tant desenvolupadors individuals com diverses empreses. Abans del llançament del Qt Project el desenvolupament el realitzava Nokia amb la seva divisió Qt Development Frameworks que es va crear a partir de la compra de la companyia noruega Trolltech, la creadora original de Qt. El febrer de 2011 Nokia va anunciar que abandonava el sistema operatiu mòbil Symbian i passava a basar la seva estratègia en smartphones en la plataforma de Microsoft. Un mes després Nokia anunciava la venda de la llicència i serveis de suport de Qt a l'empresa Digia.

Les Qt són especialment usades per KDE, Qtopia i OPIE i usen una versió estesa del llenguatge de programació C++, però els bindings existeixen per Python, Ruby, C, Perl i Pascal.

L'entorn de desenvolupament Qt s'utilitza en programari molt conegut com l'Autodesk Maya, l'Adobe Photoshop Elements, Skype, VLC media Player o VirtualBox. També per grans organitzacions i empreses com l'Agència Espacial Europea, DreamWorks, Google, HP, Samsung o Walt Disney Pictures, entra d'altres.

Funcionen en la majoria de plataformes i tenen un important esforç d'internacionalització. A part, té altres característiques que no són pròpiament de la interfície gràfica, com l'accés a bases de dades SQL, gestió de fils d'execució, una API unificada i un analitzador d'XML.

Motius de selecció

Qt és una llibreria molt completa per el desenvolupament d'aplicacions amb interfícies gràfiques. Una de les propietats de Qt, és la possibilitat d'afegir elements de visualització emprant OpenGL, la qual cosa ens motiva més a utilitzar Qt. Per altre banda, Qt és una eina molt estesà entre les empreses punteres, per la qual cosa ens ha semblat interessant aprofitar el projecte ampliar el nostre currículum.

Apartat 7.7: Sistemes Operatius: Windows 7

Per desenvolupar aquest projecte em fet servir el sistema operatiu de Windows, més concretament la versió Windows 7.

Motius de selecció

Els motius que ens han portat a escollir Windows 7 com a sistema operatiu per el projecte han estat en primer lloc, que és el sistema operatiu de la nostre maquina de treball. Per altre banda, aquest és un dels sistemes operatius més estes.

Apartat 7.8: GANTT Project

Gantt Project és una aplicació de programari lliure que permet realitzar diagrames de Gantt. Un diagrama de Gantt és una eina gràfica que té com a objectiu veure fàcilment la planificació d'un projecte mostrant el temps de dedicació per cadascuna de les tasques.

El diagrama de Gantt no indica les relacions existents entre activitats, però la posició de cada tasca al llarg del temps fa que es puguin identificar aquestes relacions i dependències.

Motius de selecció

Hem escollit aquest programa perquè és de programari lliure i els resultats que s'obtenen són de qualitat.

Apartat 7.9: StarUML

StarUML és un projecte open source per crear ràpidament diagrames UML. Aquesta aplicació esta en continua expansió degut a la seva filosofia de distribució i és totalment gratuïta.

Motius de selecció

El principal motiu d'escollir aquesta aplicació, a part del fet que ens permeti desenvolupar els diferents diagrames necessaris per el nostre projecte, ha estat el fet de ser gratuïta i no necessitar d'una llicencia per la seva utilització.

Capítol 8: Anàlisi i disseny del sistema

En aquest capítol veurem de manera detallada les necessitats del sistema desenvolupat i les solucions desenvolupades per a cada necessitat. Tot seguint la metodologia especificada, realitzarem la anàlisis i el disseny del sistema a partir de diferents diagrames i esquemes de la interfície d'usuari.

L'objectiu d'aquest projecte és desenvolupar un sistema que permeti la simulació de vent en temps real sobre un paisatge urbà. Hem de tenir en compte que per assolir aquest objectiu, necessitarem desenvolupar eines per fer la càrrega del paisatge, per realitzar la simulació i per visualitzar els resultats.

El projecte desenvolupat té un punt de partida molt clar, l'article publicat per Chen et al., "The Lattice-Boltzmann method".

Partint d'aquesta base teòrica, s'han desenvolupat els mòduls necessaris per assolir els objectius marcats.

Apartat 8.1: Fitxes i diagrames de cas d'ús

Un cas d'ús és una tècnica per a la captura de requisits potencials d'un nou sistema. Cada cas d'ús proporciona un o més escenaris que indiquen com hauria d'interactuar el sistema amb l'usuari o amb un altre sistema per aconseguir l'objectiu que es pretén.

Els actors que tindrem en compte en el nostre sistema seran, simplement, els usuaris finals que utilitzaran l'aplicació (veure figura 8.1).

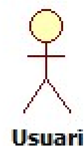


Figura 8.1: Actor del sistema.

Diagrama de cas d'ús general

En el diagrama de cas d'ús següent (veure figura 8.2), mostrem els requeriments principals del sistema.

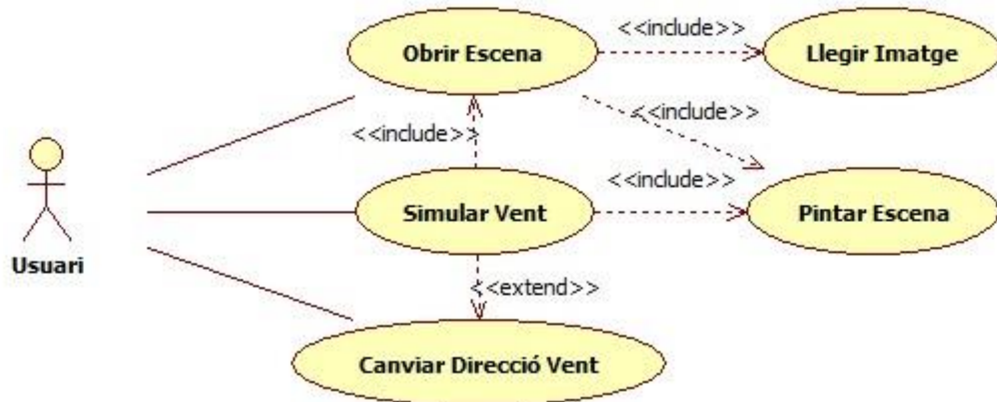


Figura 8.2: Diagrama de cas d'ús general.

A continuació veurem les fitxes de cada cas d'ús del sistema. Aquestes ens permetran fer-nos una idea del funcionament general de cada cas d'ús.

Fitxa de cas d'ús	Obrir Escena
Descripció	L'usuari vol obrir una escena sobre la qual realitzar una simulació
Actor	Usuari
Precondició	L'escena l'hem transformat prèviament en un imatge del pla zenital d'aquesta.
Flux Principal	1- Seleccionar una escena 2- Llegir la imatge de l'escena seleccionada 3- Extreure informació bàsica de la imatge 4- Pintar escena
Flux Alternatiu	
Postcondició	Escena oberta i visualitzada

Fitxa de cas d'ús	Llegir Imatge
Descripció	Permet llegir la informació d'una imatge emmagatzemada en local o en la red.
Actor	Usuari
Precondició	El fitxer seleccionat té extensió d'un fitxer d'imatge com .jpg, .png, etc.
Flux Principal	1- Obtenir el format del fitxer 2- Decodificar segons el format 3- Retornar una objecte Image amb la informació extreta.
Flux Alternatiu	
Postcondició	Tenim un objecte Image amb tota la informació de la imatge.

Fitxa de cas d'ús	Pintar escena
Descripció	Visualitza per pantalla l'escena i els efectes de la simulació sobre aquesta.
Actor	Sistema
Precondició	L'usuari ha obert una escena prèviament
Flux Principal	1- Obtenir estructura bàsica de l'escena en cel·les. 2- Per cada cel·la 2.1. Calcular posició de la pantalla 2.2. Per cada tipus de cel·la pintar d'un color o altre
Flux Alternatiu	
Postcondició	L'estat de l'escena s'ha representat per la pantalla de l'usuari.

Fitxa de cas d'ús	Simular vent
Descripció	A partir d'una escena farem una simulació del desplaçament del vent per l'interior d'aquesta.
Actor	Usuari
Precondició	L'usuari ha obert una escena prèviament
Flux Principal	▪ Obtenir estructura bàsica de l'escena en cel·les. ▪ Inicialitzar valors per realitzar la simulació. ▪ MENTRES no fi 3.1. Per cada cel·la aplicar el mètode de Lattice-Boltzmann 3.2. Retornar càlculs de densitats i de direcció del vent a cada cel·la. 3.3. Pintar nou estat de l'escena.
Flux Alternatiu	3.1.1- Si la cel·la és un obstacle no cal fer cap càlcul. 3.1.2- Si la cel·la és generadora de vent, afegirem partícules de vent en una direcció establerta per defecte o per l'usuari. 3.1.3- Si la cel·la és un límit, copiarem els desplaçament de partícules de les cel·les veïnes i aplicarem Lattice-Boltzman.
Postcondició	Calculem el nou estat de l'escena i el pintem per tal que l'usuari el pugui visualitzar.

Fitxa de cas d'ús	Canviar direcció del vent
Descripció	Modifiquem la direcció a partir de la qual simularem el vent
Actor	Usuari
Precondició	
Flux Principal	1- Seleccionar nova direcció del vent 2- Canviar variable amb direcció del vent 3- Si escena carregada 3.1. Actualitzar estructura de l'escena, caldrà modificar les cel·les límit i les generadores de vent.
Flux Alternatiu	
Postcondició	Hem modificat la direcció del vent per fer la simulació.

Diagrama de classes

Un diagrama de classes és un tipus de diagrama d'estructura estàtica que descriu l'estructura d'un sistema mostrant les classes del sistema, els atributs i les relacions entre elles.

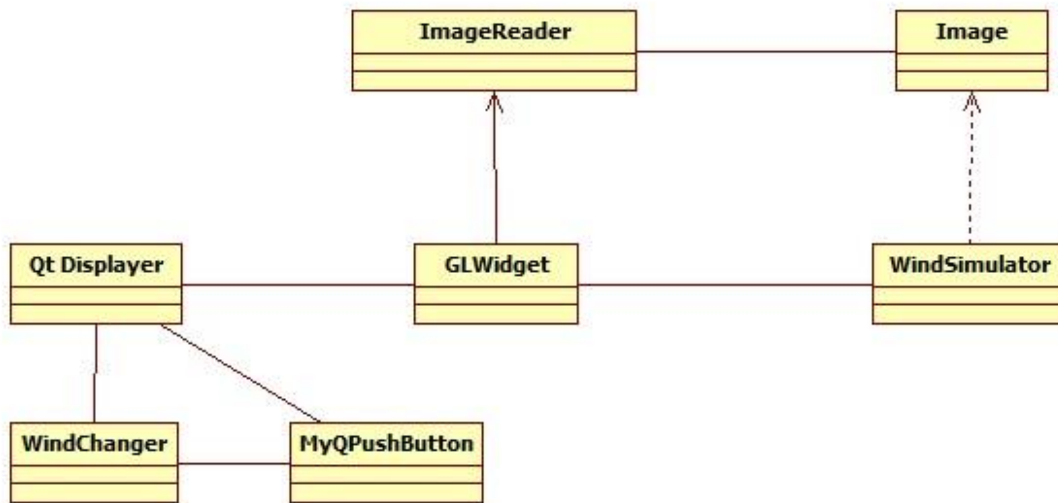


Figura 8.3: Diagrama de classes general.

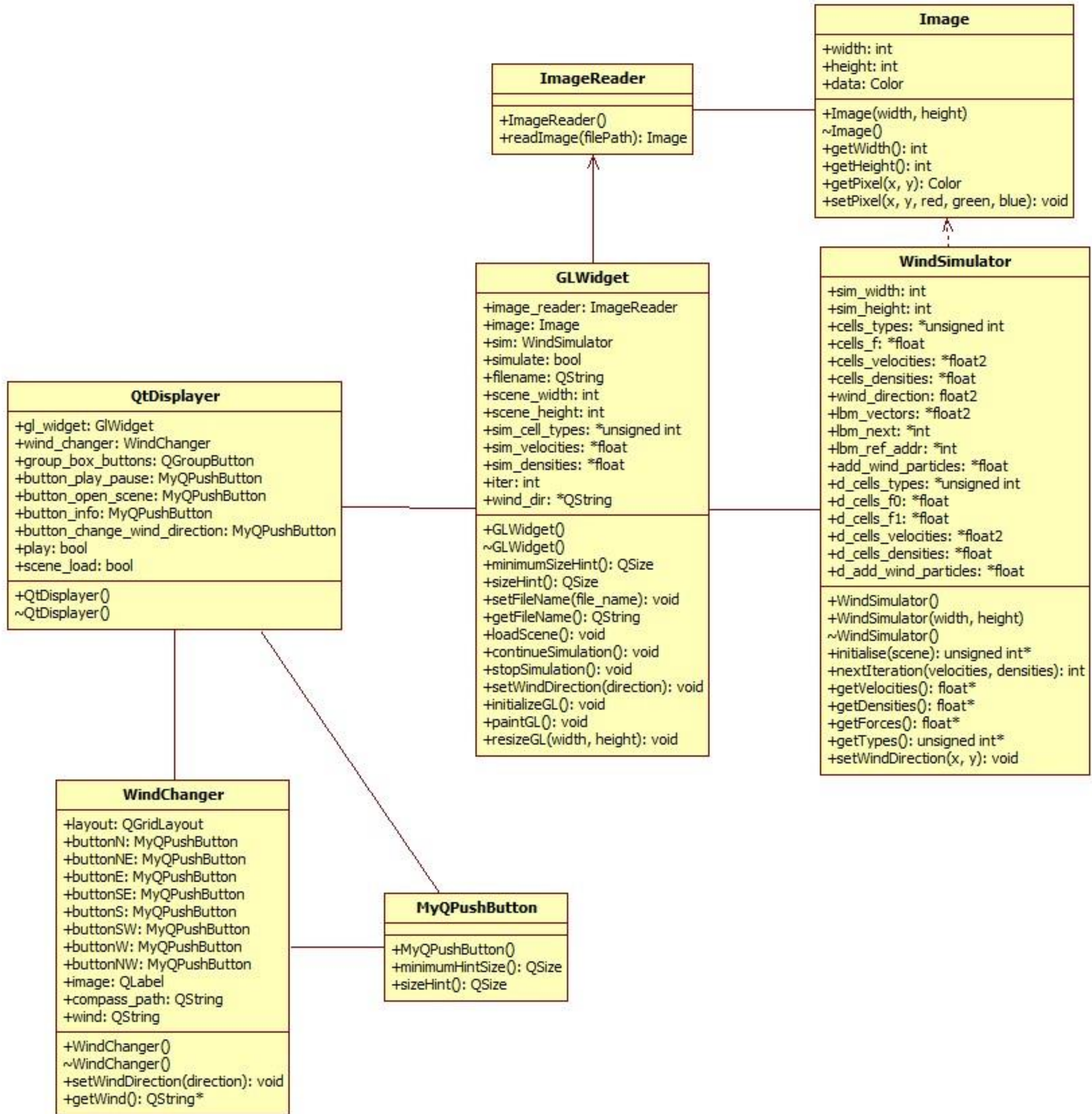


Figura 8.4: Diagrama de classes complet.

Image

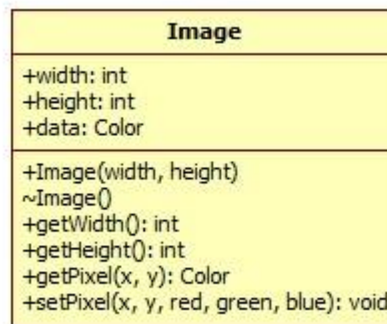


Figura 8.5: Fragment del diagrama de classes referent a Image.

Aquesta és la classe encarregada d'emmagatzemar la informació bàsica d'una imatge, a partir de la qual extraurem la informació de l'escena. Per fer més entenedora, hem encapsulat els valors RGB de cada píxel en un *struct* Color.

- **Image(width, height):** constructor de la classe que ens permet crear una instància d'un objecte de tipus Image. Rep dos paràmetres d'entrada, els quals determinen les dimensions de la imatge.
- **~Image():** destructor de la classe. S'encarrega d'alliberar l'espai de memòria reservat per una imatge.
- **int getWidth():** retorna l'ample de la imatge.
- **int getHeight():** retorna l'alçada de la imatge.
- **Color getPixel(x, y):** retorna una terna RGB amb el color corresponen al píxel amb coordenades x, y de la imatge.
- **void setPixel(x, y, red, green, blue):** mètode per modificar el color del píxel amb coordenades x, y de la imatge.

ImageReader

Figura 8.6: Fragment del diagrama de classe referent a ImageReader.

Aquesta és la classe encarregada de fer la lectura dels fitxers d'imatges i convertir-los en objectes Image. Per fer aquesta feina, aprofitarem la llibreria *FreeImage*.

- **ImageReader():** constructor de la classe, el qual s'encarrega d'inicialitzar els valors necessaris per poder llegir fitxers d'imatges.
- **Image readImage(filePath):** retorna una imatge amb la informació bàsica del fitxer d'imatge definit per el filePath.

QtDisplayer

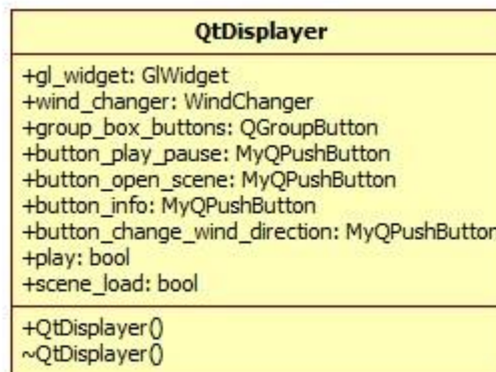


Figura 8.7: Fragment del diagrama de classes referent a QtDisplayer.

Aquesta és la classe encarregada de crear i gestionar la interfície gràfica de l'aplicació. Per fer-ho, utilitzarem la llibreria Qt. Qt proporciona moltes classes predefinides que podrem utilitzar i també permet definir *widgets* per poder customitzar elements de la interfície gràfica.

- **QtDisplayer():** constructor de la classe el qual s'encarrega d'inicialitzar els diferents elements de la interfície gràfica de l'aplicació.
- **~QtDisplayer():** destructor de la classe el qual s'encarrega d'alliberar l'espai ocupat a memòria pels diferents elements.

A part dels mètodes esmentats, Qt té predefinides diverses funcions per pintar els elements o gestionar el posicionament. Com no els hem necessitat modificar, en detallarem el funcionament.

GLWidget

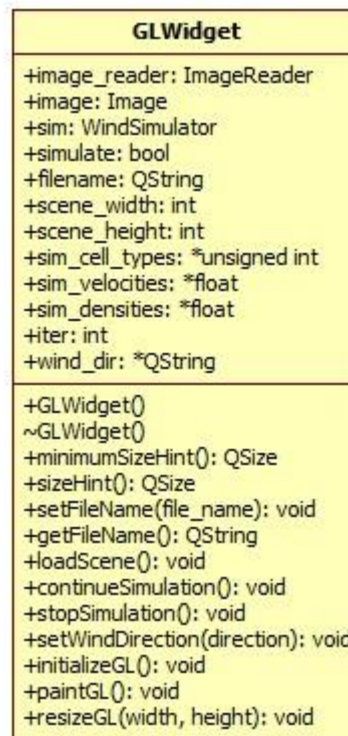


Figura 8.8: Fragment del diagrama de classes referent a GLWidget.

Aquesta és la classe encarregada de visualitzar, mitjançant OpenGL, l'escena i el seu estat. Aquesta és una de les grans classes, la qual enllaça amb quasi totes les altres. Dins de la interfície gràfica, hem de veure aquesta classe com a una subclasse del QtDisplayer, per ser més exacte és un *widget* que utilitza QtDisplayer.

Com podem en la figura 8.8, aquesta classe emmagatzema els diferent atributs per visualitzar la simulació de vent. Tot seguit descriurem cadascun dels atributs:

- **ImageReader image_reader**: aquest atribut és una referència a un objecte ImageReader, amb el qual realitzarem la lectura d'escenes.
- **Image image**: aquest atribut és una referència a un objecte Image, on emmagatzemarem la informació de l'escena obtinguda a partir del imatge_reader.
- **WindSimulator sim**: aquest atribut és una referència a un objecte WinSimulator, el qual serà l'encarregat de realitzar la simulació del vent.
- **bool simulate**: aquest atribut indicarà al sistema si la simulació ha d'estar activa o no. Gràcies a aquest valor podrem gestionar les interrupcions que realitzi l'usuari, com voler parar la simulació per valorar els resultats o per canviar la direcció del vent, etc.
- **QString filename**: aquest atribut guarda la localització del fitxer de l'escena.
- **int scene_width**: aquest atribut guarda l'amplada de l'escena carregada al sistema.
- **int scene_height**: aquest atribut guarda l'alçada de l'escena carregada al sistema.

- **unsigned int* sim_cell_types**: aquest atribut guarda l'estructura de tipus de l'escena. Aquest serà molt útil a l'hora de representar l'escena.
- **float* sim_velocities**: aquest atribut guarda les velocitats resultants de l'última iteració de la simulació.
- **float* sim_densities**: aquest atribut guarda les densitats resultants de l'última iteració de la simulació.
- **int iter**: aquest atribut guarda la iteració actual de la simulació.
- **QString wind_dir**: aquest atribut guarda un *string* on indicarà la direcció del vent actual. Els valors possibles són {'N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW'}. A partir d'aquest atribut podrem informar a la simulació dels canvis en la direcció del vent proposats per l'usuari.

A part d'aquests valors, també tenim altres variables en la classe les quals són especifiques de *OpenGL* per tal de visualitzar la simulació. Aquestes variables són els vèrtexs, colors de cada vèrtexs i els índex de cadascuna de les cares. Aquesta informació és crearà dinàmicament per cada escena seguint el següent esquema:

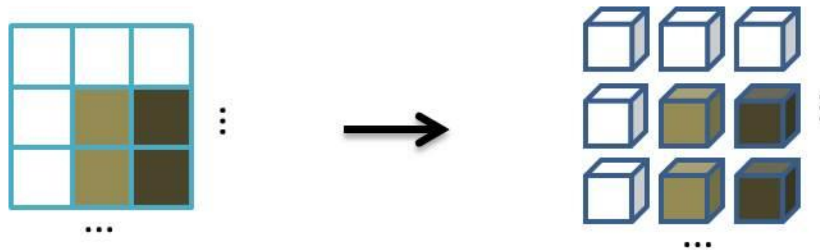


Figura 8.9: esquema creació model de la simulació.

Com veiem en la figura 8.9, el model que crearem per visualitzar la simulació serà un model 3D de cubs, on cada cub representarà un dels píxels de la imatge de l'escena. Aquest fet és imperceptible en els resultats, ja que la càmera que emprem per a visualitzar la representació és perpendicular al model 3D, però la creació d'aquest model té un motiu molt senzill: poder escalar el sistema per poder representar escenes 3D més endavant. Amb l'objectiu futur de crear un símil a un model de vòxels per escenes 3D, hem creat aquest model per representar la simulació.

Per altre banda, tenim diferents mètodes per gestionar la visualització de la simulació i, també, per controlar el procés de simulació. Tot seguit veurem cadascun dels mètodes de la classe:

- **GLWidget():** constructor de la classe encarregat d'inicialitzar els diferents elements per tal de visualitzar la simulació amb OpenGL i poder interactuar amb ella.
- **~GLWidget():** destructor de la classe, encarregat d'alliberar l'espai de memòria ocupat pels diferents elements creats.
- **QSize minimunSizeHint():** retorna la mida mínima per representar la simulació. Aquesta mida és basa en l'ample i l'alçada de la finestra que engloba els elements del GLWidget.
- **QSize sizeHint():** retorna la mida actual de la finestra que engloba els elements del GLWidget.
- **void setFileName(file_name):** mètode que permet definir el *path* de la imatge que voldrem utilitzar per realitzar la simulació.
- **QString getFileName():** retorna un *string* amb el *path* de la imatge seleccionada.
- **void loadScene():** mètode encarregat d'inicialitzar els paràmetres necessaris per poder realitzar la simulació. En la figura 8.4 podem veure el diagrama de seqüència d'aquest mètode.

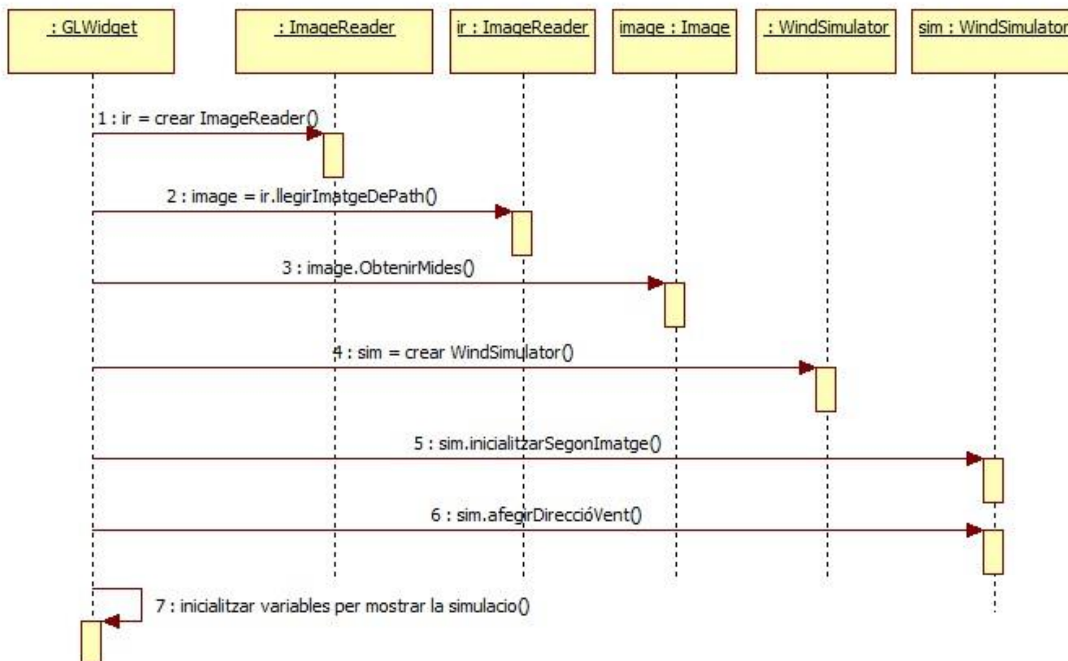


Figura 8.10: Diagrama de seqüència de loadScene.

Tal i com podem veure en la figura 8.10, el mètode és basa en la creació d'un lector d'imatges, realitzar la carrega de la imatge. A partir de la imatge, podrem inicialitzar els valors necessaris per la simulació. Un cop inicialitzat, modificarem la direcció del vent a una per defecte (N). Finalment, prepararem els valors necessaris per visualitzar la simulació.

```
loadScene() {
 Crear lector d'imatges.
 Llegir imatge del paisatge.
 Obtenir mides de la imatge.
 Crear simulador de vent.
 Inicialitzar el simulador a partir de la imatge.
 Modificar la direcció del vent a N per defecte.
 Preparar visualització de la simulació.
}
```

- **void continueSimulation():** mètode que permet continuar amb la simulació en cas d'haver estat aturada prèviament.
- **void stopSimulation():** mètode que permet aturar la simulació.
- **void setWindDirection(direction):** mètode que s'encarrega de modificar la direcció del vent de la simulació.
- **void initializeGL():** mètode encarregat d'inicialitzar els paràmetres necessaris per treballar amb OpenGL.
- **void paintGL():** mètode encarregat de calcular la següent iteració de la simulació, en cas de no haver-la aturat, i pintar-ne el resultat. Tot seguit veurem el pseudocodi d'aquesta funció.

El primer pas d'aquest mètode serà el de netejar els buffers de colors d'OpenGL i situar la càmera en la posició correcte. En cas de estar simulant, haurem de calcular la següent iteració de la simulació, actualitzar els colors segons els resultats de la iteració. Finalment pintarem la simulació, en cas de estar parada, pintarem la representació anterior, en cas contrari, pintarem els nous valors de la simulació.

```
paintGL() {
   netejar buffers de colors.
   situar la càmera.
   SI simulate LLAVORS
         Calcular següent iteració de la simulació.
         Actualitzar colors segons els resultats.
         Iter++.
   FSI
   pintar model amb els colors actualitzats.
}
```

Per actualitzar els colors del model ho podrem fer a partir de les velocitats o de les densitats. Per defecte ho hem fet a partir de les densitats mitjançant un rang de valors per la densitat. Per tant per calcular el color d'una cel·la FLUID fem:

```
Red = 0.f;
Aux = densitat de la cel·la – MIN;
Green =  Aux/(MAX-MIN);
Blue = 1.f;
```

D'aquesta manera obtenim diferents tonalitats segons la densitat de cada moment. Finalment, per decidir el rang de valors acceptats per la representació ho hem fet a partir de l'experiència de diverses simulacions, és a dir, hem fet diverses proves i ens hem quedat amb els valors més petits i més grans trobats més un marge de seguretat que hi hem afegit.

- **void resizeGL(width, height):** mètode encarregat de modificar les dimensions de la finestra del widget.

WindChanger

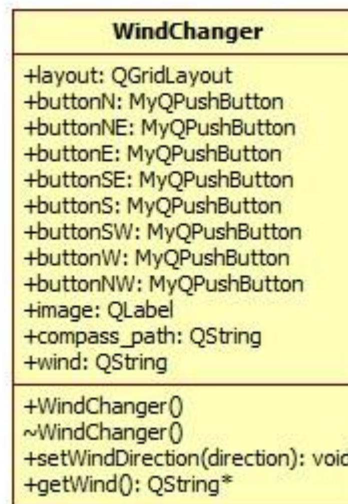


Figura 8.11: fragment del digrama de classes referent a WindChanger

Aquesta és la classe encarregada de permetre a l'usuari poder modificar la direcció del vent de la simulació. Al igual que la classe anterior, aquesta és un *widget* que utilitza el QtDisplayer.

- **WindChanger():** constructor de la classe encarregat d'inicialitzar els elements que permetran visualitzar i interactuar amb el modificador de vent.
- **~WindChanger():** destructor de la classe encarregat d'alliberar l'espai de memòria ocupat pels elements del widget.
- **void setWindDirection(direction):** mètode encarregat de modificar la direcció del vent segons la selecció del usuari.
- **QString* getWind():** retorna la direcció actual del vent en forma de *string*.

MyQPushButton

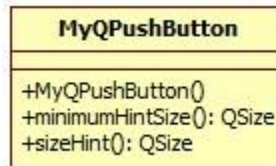


Figura 8.12: fragment del digrama de classes referent a MyqPushButton.

Aquesta és la classe encarregada de sobreescriure els botons de Qt.

- **MyQPushButton():** constructor de la classe, encarregat d'inicialitzar un botó.
- **QSize minimunSizeHint():** retorna la mida mínima del botó.
- **QSize sizeHint():** retorna la mida actual del botó.

WindSimulator

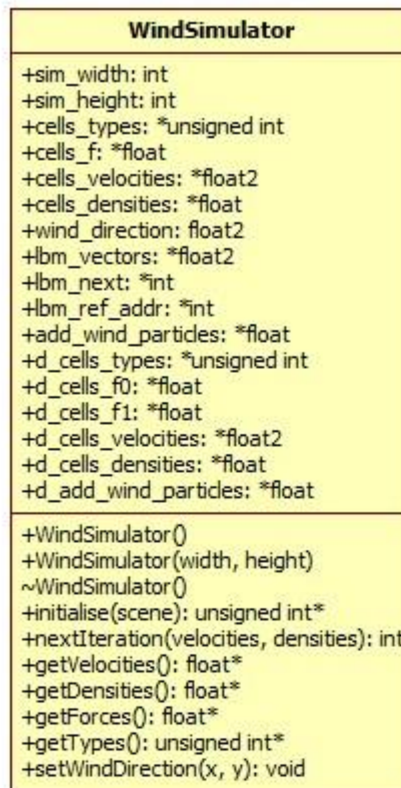


Figura 8.13: fragment del digrama de classes referent a WindSimulator

Aquesta és la classe encarregada de realitzar la simulació del vent. En aquesta classe és on implementarem el mètode de Lattice-Boltzmann, més concretament implementarem una versió en paral·lel utilitzant CUDA.

Com podem en la figura 8.13, aquesta classe emmagatzema els diferent atributs per realitzar la simulació de vent. Tot seguit descriurem cadascun dels atributs:

- **int sim_width**: aquest atribut indicarà al llarg de la simulació l'ample de l'escena sobre la qual estem simulant. Aquest atribut, juntament amb l'alçada de la simulació, seran molt importants perquè determinaran l'espai de memòria necessari per gestionar els resultats de la simulació.
- **int sim_height**: aquest atributs indicarà l'alçada de l'escena sobre la qual estem simulant i servirà per gestionar l'espai de memòria necessari pels resultats de la simulació.
- **unsignet int *cells_types**: aquest atribut hauria de ser una matriu la qual emmagatzema el tipus de cadascuna de les cel·les de la simulació. Però, per temes de disseny del propi CUDA, treballarem amb una *array*, és a dir, guardarem seqüencialment cadascuna de les files de la matriu. Pel que fa als tipus de les cel·les, aquest seran enters, els quals significaran:
  - 0 → FLUID o cel·la de fluid, és a dir, cel·la per on es pot desplaçar el fluid.
  - 1 → NOSLIP o cel·la amb obstacle, és a dir, cel·la per on no pot desplaçar-se el fluid i hi col·lisiona.
  - 2 → VELOCITY o cel·la generadora de vent, és a dir, cel·la encarregada d'afegir noves partícules de vent segons la direcció del vent indicada.
  - 3 → OPENBOUNDARY_N o cel·la de límit obert superior, és a dir, cel·la de la part superior de la simulació encarregada d'eliminar el fluid que li arriba.
  - 4 → OPENBOUNDARY_S o cel·la de límit obert inferior, anàlogament a l'anterior, però per la part inferior de la simulació.
  - 5 → OPENBOUNDARY_E o cel·la de límit obert de la part dreta, anàlogament a les anteriors, però per la part dreta de la simulació.
  - 6 → OPENBOUNDARY_W o cel·la de límit obert de la part esquerra, anàlogament a les anteriors, però per la part esquerra de la simulació.
  - 7 → OPENBOUNDARY_VEL o cel·la de límit obert per les VELOCITY, és a dir, cel·la de cas especial la qual posarem darrera les VELOCITY per tal de rebotar de forma integra les partícules de vent generades.

Com podem veure en el llistat anterior, tenim diferenciats els límits segon siguin en la part superior, part inferior, part dreta o part esquerra. Això ha estat necessari per garantir un bon comportament dels límits durant la simulació. L'objectiu principal dels límits és el de permetre al vent fluir fora de la simulació i així evitar l'increment descontrolat de partícules en l'escena. Per simular aquest comportament, necessitem copiar les forces de les cel·les interiors als límits, motiu pel qual hem de saber en quina direcció hem d'estirar les dades. Per tenir-ne un visió més clara veurem un exemple gràfic a la figura 8.14.

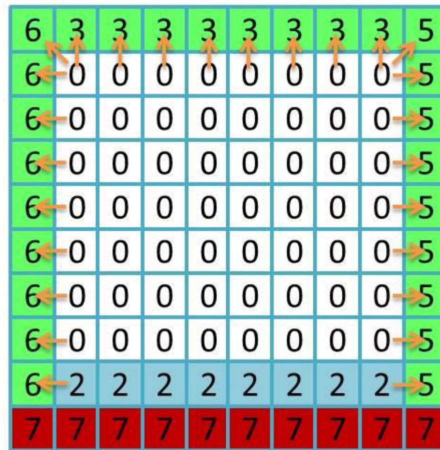


Figura 8.14: Exemple gràfic de la necessitat de tenir límit N, S, E i W.

- **float *cells_f**: aquest atribut és l'encarregat de guardar la distribució de partícules en la simulació. Com ja hem explicat prèviament, segons el model que utilitzem, tindrem 5 o 9 direccions en les quals es podran distribuir les partícules. Doncs, aquest atribut és el que s'encarrega de registrar aquests desplaçaments.

  Com que el model que estem implementant és el D2Q9, necessitarem registrar 9 direccions per cada cel·la. La manera com hem estructurat aquesta informació ha estat a partir d'una taula de mida 9n on les n primeres posicions són les f_0 de cada cel·la, les següents n posicions són les f_1 de cada cel·la i així successivament fins guardar les 9 distribucions.

- **float2 *cells_velocities**: aquest atribut, anomenat *cells_velocities* en el diagrama de classes, és l'encarregat de guardar la tendència de moviment de cada cel·la, és a dir, és la suma de distribucions per cada direcció. Amb aquest valor podem valorar la direcció de propagació del fluid. Com CUDA permet definir el tipus float2, hem evitat haver de crear una taula amb el doble de posicions per guardar x i y de cada cel·la.

- **float *cells_densities**: aquesta atribut, anomenat *cells_densities* en el diagrama de classe, és l'encarregat de guardar la quantitat de partícules que tenim a cada cel·la. Per fer-ho, farem la suma de distribucions de cada cel·la.

- **float2 wind_direction**: aquest atribut és l'encarregat d'indicar a la simulació en quin sentit s'afegiran noves partícules de vent. Cada cop que modifiquem aquest atribut, haurem de recalcular la informació de les cel·les més externes, és a dir, dels límits de l'escena.

Els següent grup d'atributs que veurem, guarden valors auxiliars per a la simulació. Aquest atributs no guarden informació d'interès per l'usuari final, però són importants per a la implementació realitzada del LBM.

- **float2 lbm_vectors[9]**: aquest atribut guarda els vectors directors de cada força de la simulació, serien els vectors $\vec{e_i}$ del LBM.
- **int lbm_next[9]**: aquest atribut guarda el desplaçament de l'índex necessari per trobar la cel·la veïna per cada direcció, és a dir, per la posició 0 guardarem un desplaçament de 0, ja que aquesta sempre apunta a ella mateixa. En canvi, si volem saber la veïna de l'esquerra guardarem un desplaçament de +1, per la dreta -1, per la veïna superior –*sim_width*, etc.
- **int lbm_ref_addr[9]**: aquest atribut guarda el desplaçament de l'índex per accedir a cadascuna de les forces, és a dir, per les forces 0 el desplaçament serà de 0, per les forces 1 el desplaçament serà de *sim_width*sim_height*, per les forces 2 serà 2*sim_*width*sim_height*, etc.
- **float add_wind_particles[9]**: aquest atribut guarda la quantitat de partícules que haurem d'afegir per cadascuna de les direccions. Aquest valors estaran predeterminats per la direcció del vent.

Per últim, tenim els atributs per treballar amb la GPU. Aquest atributs només es podran utilitzar des d'un *kernel*, però per evitar de redefinir-los cada cop que vulguem cridar un *kernel*, s'han guardat com atributs de la classe.

- **unsigned int *d_cells_types**: aquest serà el punter al tipus de cada cel·la en la GPU.
- **float *d_cells_f0**: aquest serà el punter a les forcés d'entrada de cada cel·la en la GPU.
- **float *d_cells_ f1**: aquest serà el punter a les forcés de sortida de cada cel·la en la GPU.
- **float2 *d_cells_velocities**: aquest serà el punter a les velocitats de cada cel·la en la GPU.
- **float *d_cells_densities**: aquest serà el punter a les densitats de cada cel·la en la GPU.
- **float *d_ add_wind_particles**: aquest serà el punter als increments de vent per cada direcció en la GPU.

Un cop vistos els atributs de la classe, passarem a veure els mètodes de la mateixa. Com hem fet anteriorment, veurem el pseudocodi i algun diagrama de seqüència dels mètodes més importants.

- **WindSimulator():** constructor de la classe, encarregat d'inicialitzar els paràmetres, tant de CPU com els de GPU, per realitzar la simulació.
- **WindSimulator(width, height):** constructor de la classe encarregada d'inicialitzar els paràmetres per realitzar la simulació, tot indicant les mides del paisatge a simular.

  A grans trets, aquest mètode haurà de reservar espai de memòria pels atributs de la simulació, calcular els desplaçaments de memòria per accedir correctament a les forces i reservar espai de memòria a la GPU per paral·lelitzar el LBM.

```
Simulator(int width, int height) {
    Reservar espai de memòria a CPU.
    Calcular desplaçaments de memòria.
    Reservar espai de memòria a GPU.
}
```

- **~Simulator():** destructor de la classe encarregat d'alliberar tant espai de memòria CPU com de memòria de la GPU.

- **unsigned int* initialise(scene):** retorna la malla amb el tipus de cada cel·la de l'escena. Per retornar aquest valor prèviament s'han inicialitzat els paràmetres dependents de l'escena. En la figura 8.5, podem veure el diagrama de seqüència d'aquest mètode.

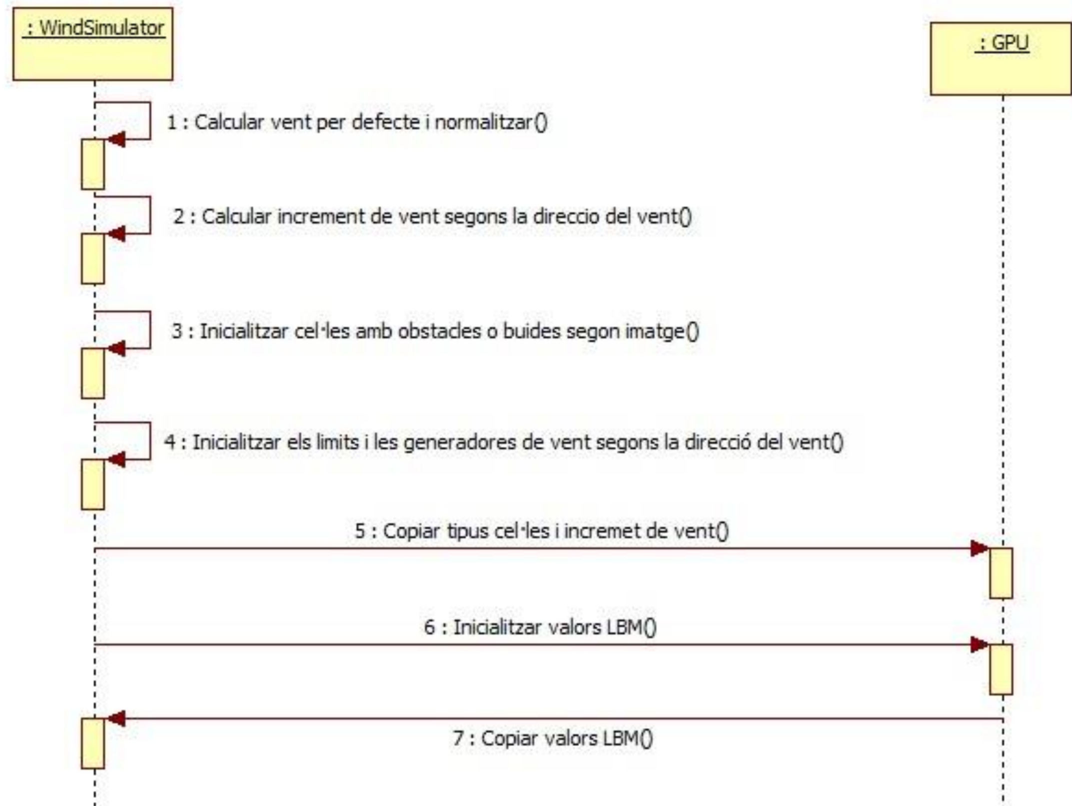


Figura 8.15: Diagrama de seqüencia de *initialise*.

Com podem veure en la figura 8.15, el procés d'inicialitzar s'encarrega bàsicament de convertir la imatge de l'escena en un format de dades útil per poder realitzar la simulació, és a dir, hem de convertir una malla de píxels en una malla de tipus cel·les.

Partint d'una imatge de l'escena a simular, realitzarem el següent procés:

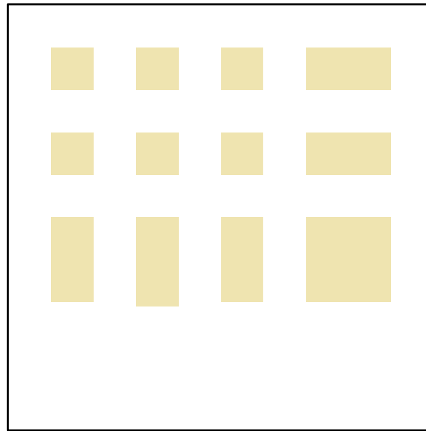


Figura 8.16: imatge d'una escena simple.

El primer pas és convertir tots els píxels blancs en cel·les FLUID (0) i els de color en NOSLIP (1):

0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	1	0
0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	1	0
0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	1	0
0	1	0	1	0	1	0	1	1	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Figura 8.17: transformació de píxels a tipus FLUID o NOSLIP.

El segon pas és el de modificar, en aquest ordre, els límits superior i inferior, i després els laterals. Els valors que posarem seran els de OPENBOUNDARY_N, OPENBOUNDARY_S, OPENBOUNDARY_E i OPENBOUNDARY_W, en l'exemple els pintarem de verd per diferenciar aquest pas del anterior:

6	3	3	3	3	3	3	3	3	5
6	1	0	1	0	1	0	1	1	5
6	0	0	0	0	0	0	0	0	5
6	1	0	1	0	1	0	1	1	5
6	0	0	0	0	0	0	0	0	5
6	1	0	1	0	1	0	1	1	5
6	1	0	1	0	1	0	1	1	5
6	0	0	0	0	0	0	0	0	5
6	0	0	0	0	0	0	0	0	5
6	4	4	4	4	4	4	4	4	5

Figura 8.18: addicció dels límits a la malla de tipus.

Com podem veure en les cantonades teníem el conflicte de quin tipus havíem de tenir, al final hem decidit sempre generar les cel·les en aquest ordre i tenir-ho en compte.

L'últim pas és el d'afegir les cel·les VELOCITY segons la direcció del vent i corregir els límits darrera aquestes cel·les per evitar comportaments erronis. En el cas per defecte, com la direcció del vent és (0,1), modificarem les cel·les de la part inferior com si fos l'entrada del vent a l'escena. En la imatge distingirem les cel·les VELOCITY amb un color blau i els límits corregits en vermell.

6	3	3	3	3	3	3	3	3	5
6	1	0	1	0	1	0	1	1	5
6	0	0	0	0	0	0	0	0	5
6	1	0	1	0	1	0	1	1	5
6	0	0	0	0	0	0	0	0	5
6	1	0	1	0	1	0	1	1	5
6	1	0	1	0	1	0	1	1	5
6	0	0	0	0	0	0	0	0	5
6	2	2	2	2	2	2	2	2	5
7	7	7	7	7	7	7	7	7	7

Figura 8.19: càlcul de les cel·les VELOCITY i modificació dels límits.

Com veiem en el codi, modificarem les cel·les VELOCITY segons la direcció del vent. Per fer-ho calcularem el producte escalar entre la direcció del vent i les direccions predefinides N, S, E i W, si el resultat és positiu haurem de incorporar cel·les VELOCITY en aquesta direcció. Tot seguit veurem uns exemples per explicar com ho hem dissenyat:

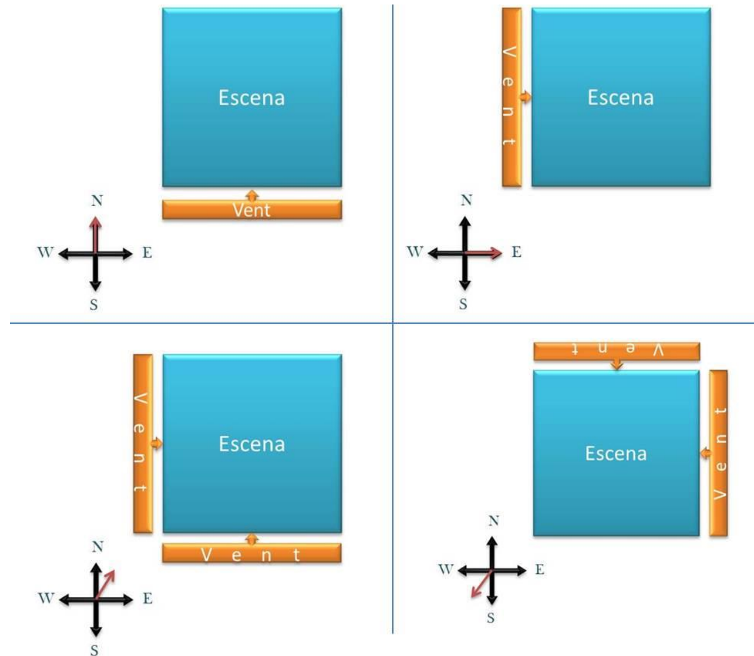


Figura 8.20: exemple de càlcul de cel·les VELOCITY.

- **int nextIteration(velocities, densities):** retorna si el pas de simulació ha anat bé. Aquest mètode s'encarrega de realitzar un pas de simulació i en retorna els valors resultants per els paràmetres d'e/s.

Per calcular una nova iteració de la simulació el que farem serà copiar a la GPU els resultats de la iteració anterior, cridar el *kernel* encarregat de calcular una iteració de LBM, cridar el *kernel* encarregat de preservar el bon funcionament de la simulació en les cel·les límits i copiar els resultats a CPU.

```
Int nextIteration(velocities, densities) {
      Copiar informació iteració anterior a GPU.
      Cridar kernel LBMStep.
      Crida kernel InitOpenLBM.
      Copiar resultats a CPU i tractar-los.
}
```

- **float* getVelocities():** retorna les velocitats actuals de la simulació.
- **float* getDensities():** retorna les densitats actuals de la simulació.
- **float* getForces():** retorna les forces que estan interactuant actualment en la simulació.
- **unsigned int* getTypes():** retorna la malla amb el tipus de cada cel·la de la simulació.
- **void setWindDirection(x, y):** mètode que modifica la direcció del vent de la simulació.

  Per modificar la direcció del vent, hem de tenir en compte que hem de recalcular les cel·les VELOCITY i els límits. El primer pas que realitzarem serà el de actualitzar i normalitzar la direcció del vent, després calcularem els nous increments de vents a partir de la nova direcció. A continuació modificarem la taula de tipus per tal de reorientar les cel·les VELOCITY modificarem la taula de tipus per tal de reorientar les cel·les VELOCITY segons la nova direcció del vent. Per acabar, copiarem a la GPU els atributs modificats. Així doncs, aquest mètode seguirà el següent pseudocodi:

```
void setWindDirection(x, y) {
    Actualitzar i normalitza wind_direction.
    Recalcular els increments de vent segons nova direcció.
    Eliminar les cel·les VELOCITY i OPENBOUNDARY_VEL.
    Calcular noves cel·les VELOCITY i OPENBOUNDARY_VEL.
    Copiar nova informació a la GPU.
}
```

Tot i no aparèixer en el diagrama de classes, ja que són mètodes privats, hi han tres *kernels* que utilitzarem per paral·lelitzar el mètode de LBM.

- **__global__ void InitLBM(cells_types, f, velocities, densities):** mètode encarregat d'inicialitzar els valors de la simulació en la GPU.

  En aquest mètode inicialitzarem les forces, velocitats i densitats de cada cel·la. Depenen del tipus de la cel·la les inicialitzarem d'una manera o altre. En el cas dels límits, farem el càlcul de les forces en equilibri i a partir d'aquestes calcularem les velocitats i densitats. En el cas dels obstacles o NOSLIP, inicialitzarem els valors a zero, mai accedirem a aquesta informació, però per mostrar els resultats ens serà molt adient. Finalment, en el cas de les cel·les FLUID i VELOCITY només calcularem les forces en equilibri, ja que a partir d'aquestes el càlcul de la següent iteració en calcularà la resta de valors.

```
void InitLBM(cells_types, f, velocities, densities) {
   Calcular índex del thread.
   SI thread vàlid LLAVORS
         SI tipus cel·la[thread] és límit LLAVORS
               Calcular forces, velocitats i densitats
                Inicials.
               SORTIR mètode.
         FSI
         SI tipus cel·la[thread] és NOSLIP LLAVORS
               Inicialitzar forces, velocitats i densitats a
                zero.
               SORTIR mètode.
         FSI
         //per la resta, FLUID i VELOCITY
         Calcular forces en equilibri
   FSI
}
```

  Per calcular les forces en equilibri utilitzarem la equació ja explicada en la teoria del mètode LBM, que és la següent:

$$f_i^{eq}(p,\vec{v}) = w_i\left[\rho + \rho_0\left(\frac{3}{c^2}(\vec{e_i}\cdot\vec{v}) + \frac{9}{2c^4}(\vec{e_i}\cdot\vec{v})^2 - \frac{3}{2c^2}(\vec{v}\cdot\vec{v})\right)\right] \quad (eq\ 4)$$

- **__global__ void initOpenLBM(cells_types, f, velocities, densities):** mètode encarregat d'inicialitzar els límits de la simulació en la GPU.

  Per un bon funcionament de la simulació, necessitem copiar les densitats i velocitats de les cel·les veïnes interiors en els límits. Per fer-ho primer calcularem l'índex del *thread* per saber quina cel·la estem tractant. En cas de no ser un límit, acabarem l'execució del *thread*. Si és un límit, copiarem les densitats i velocitats de les veïnes interiors, per poder-ho fer hem identificat els límits com N, S, E o W. Un cop copiats, calcularem les forces mitjançant la funció d'equilibri sobre els valors copiats.

```
void initOpenLBM(cells_types, f, velocities, densities) {
      calcular índex del thread.
      SI thread vàlid LLAVORS
            SI no és límit LLAVORS
                  SORTIR del mètode
            FSI
            // per els límits
            Copiar densitats i velocitats de cel·les
             interiors, segons el límit (N,S,E,W).
            Calcular forces aplicant la funció
             d'equilibri a les velocitats i densitats
             copiades.
      FSI
}
```

- **__global__ void LBMstep(cells_types, densities, velocities, f0, f1, omega, addWind):** mètode encarregat de realitzar un pas de simulació segons el mètode de Lattice-Boltzmann.

  Per desenvolupar aquest mètode, el primer pas serà calcular l'índex del *thread* per tal de saber quina cel·la hem de tractar. En cas de ser d'un tipus diferent a FLUID o VELOCITY, l'execució del *thread* haurà acabat. En cas contrari, calcularem la propagació de partícules respecte la iteració anterior, si la cel·la és VELOCITY afegirem els increments de vent a les forces. Per acabar, calcularem la velocitat i la densitat de la propagació i calcularem les forces resultats aproximant-les al equilibri, ja que tot fluid tendeix al equilibri.

```
void LBMstep(cells_types, densities, velocities, f0, f1,
 omega, addWind) {
      calcular índex del thread.
      SI thread vàlid LLAVORS
            SI tipus cel·la[thread] diferent de FLUID o
             VELOCITY LLAVORS
                  SORTIR mètode.
            FSI
            Calcular propagació.
            SI tipus cel·la[thread] és VEOCITY LLAVORS
                  Afegir increments de vent.
            FSI
            Calcular velocitat i densitat de la
             Propagació.
            Calcular les forces resultants aproximant-les
             al equilibri.
      FSI
}
```

Capítol 9: Implementació i proves

En aquest capítol explicarem les passos que hem seguit per desenvolupar el simulador de vent. Més concretarem, explicarem com hem implementat el mètode de *Lattice-Boltzmann* i mostrarem parts de codi per exemplificar-lo.

Així doncs, en primer lloc farem una explicació de la implementació del *WindSimulator*.

Apartat 9.1: WindSimulator

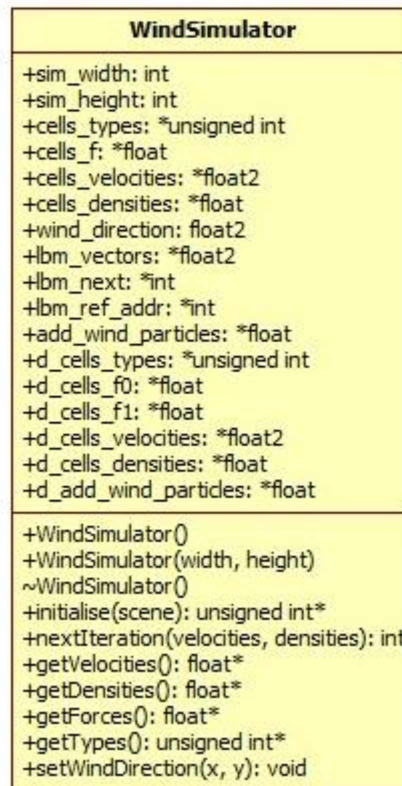


Figura 9.1: Classe WindSimulator.

En la figura 9.1, podem veure els atributs i mètodes públics de la classe. A part d'aquests mètodes, tenim de forma privada els *kernels* que emprarem per paral·lelitzar la simulació en la GPU. Per tant, ens centrarem a explicar el codi darrera els mètodes principals i com hem implementat el LBM en CUDA.

Mètodes de WindSimulator

En aquest apartat veurem els mètodes de la classe i explicarem els més importants. Com alguns del mètodes són llargs, el que farem serà desenvolupar el mètode en pseudocodi i mostrar el codi original de les parts importants. Per començar, mostrarem les constants de la classe.

```
//Parameters
const float OMEGA = 1.8f;
const float WIND_PARTICLES = 0.045f;

//LBM D2Q9 constants
const float LBM_W[9] = {(4.f/9.f),
  (1.f/9.f), (1.f/9.f), (1.f/9.f), (1.f/9.f),
  (1.f/36.f), (1.f/36.f), (1.f/36.f), (1.f/36.f)};
const int LBM_VX[9] = {0, 1, 0, -1, 0, 1, -1, -1, 1};
const int LBM_VY[9] = {0, 0, 1,  0, -1, 1, 1, -1, -1};
const int LBM_INV[9] = {0, 3, 4, 1, 2, 7, 8, 5, 6};

const float2 left_dir = make_float2(1.0,0.0);
const float2 right_dir = make_float2(-1.0,0.0);
const float2 top_dir = make_float2(0.0,-1.0);
const float2 bottom_dir = make_float2(0.0,1.0);

const float2 direction[9] = {
          make_float2(0.f,0.f), make_float2(1.f,0.f),
          make_float2(0.f,1.f), make_float2(-1.f,0.f),
          make_float2(0.f,-1.f), make_float2((1.f/sqrt(2.f)),(1.f/sqrt(2.f))),
          make_float2((-1.f/sqrt(2.f)),(1.f/sqrt(2.f))),
          make_float2((-1.f/sqrt(2.f)),(-1.f/sqrt(2.f))),
          make_float2((1.f/sqrt(2.f)),(-1.f/sqrt(2.f)))
};
```

Entre les constants podem veure OMEGA, que és el percentatge amb el qual el fluid recupera el seu estat d'equilibri, i WIND_PARTICLES, que és el percentatge de partícules de cada iteració. També tenim altres valors per preparar la simulació, com els pesos per a cada distribució, la descomposició en X i Y dels vectors directors, els índexos de les cel·les invertides (els quals utilitzarem per calcular els rebots), les direccions E, W, N i S per assignar la direcció del vent i els vectors $\vec{e_i}$ normalitzats.

- Simulator():

```
Simulator::Simulator() {
   //NOTHING
}
```

Aquest és el constructor de la classe. Com es pot veure no fa res. Això és així perquè necessitem més informació per poder inicialitzar els atributs de la classe, com per exemple valors com les mides de l'escena.

- Simulator(int width, int height) :

```
Simulator(int width, int height) {
    Reservar espai de memòria a CPU.
    Calcular desplaçaments de memòria.
    Reservar espai de memòria a GPU.
}
```

Aquest és el constructor de la classe sobreescrit per tal de construir el simulador a partir de les mides de l'escena. Com podem veure, ens encarreguem, principalment, de reservar espai de memòria tant a CPU com a GPU i de copiar valors a la GPU. A part, també calculem els desplaçament per a cada veí $\vec{e_i}$ i el desplaçament per a cada agrupació de forces. Per calcular els desplaçament ho farem de la següent forma:

```
int nxt_x = 1, nxt_y = a_sim_width;
for(int i=0; i<9; i++){
   a_lbmvecs[i] = make_float2((float) LBM_VX[i], (float) LBM_VY[i]);
   a_lbmnext[i] = LBM_VX[i]*nxt_x + LBM_VY[i]*nxt_y;
   a_lbmref_addr[i] = i*matdim;
}
```

A tall d'exemple, per reservar espai de memòria a la GPU i copiar-ne constants ho farem de la següent manera:

```
//Allocating and Moving data to GPU
CUDA_SAFE_CALL(cudaMalloc((void**) &d_celltype, sizeof(unsigned int) * matdim));

CUDA_SAFE_CALL(cudaMemcpyToSymbol(weight, &LBM_W, sizeof(float)*9));
```

- ~Simulator():

```
Simulator::~Simulator() {
   //Clean up GPU memory
   CUDA_SAFE_CALL(cudaFree(d_celltype));
   CUDA_SAFE_CALL(cudaFree(d_rho));
   CUDA_SAFE_CALL(cudaFree(d_f0));
   CUDA_SAFE_CALL(cudaFree(d_f1));
   CUDA_SAFE_CALL(cudaFree(d_u));
   CUDA_SAFE_CALL(cudaFree(d_addWind));

   delete [] a_ctype;
   delete [] a_f0;
   delete [] a_u;
   delete [] a_rho;

   delete [] a_lbmvecs;
   delete [] a_lbmnext;
   delete [] a_lbmref_addr;
   delete [] a_addWind;
}
```

Aquest és el destructor de la classe, el qual s'encarrega d'alliberar espai de memòria, tant a la CPU com a la GPU.

- unsigned int* initialise(Image *img) :

L'objectiu principal d'aquest mètode és el de calcular els tipus de cada cel·la, inicialitzar els valors de la GPU i calcular els increments de vent per defecte.

```
unsigned int* initialise(Image *img) {
      Calcular vent per defectei normalitzar.
      Calcular increment de vent segons la direcció.
      Inicialitzar cel·les amb obstacles o buides segons
       la imatge.
      Inicialitzar les cel·les límits i les generadores
       de vent segons la direcció d'aquest.
      Copiar a GPU els tipus i els increments de vent.
      Crida al kernel per inicialitzar valors LBM a GPU.
      Copiar de GPU els valors inicialitzats.
}
```

Si anem per ordre, el primer que ens interessa és el càlcul dels increments de vent. Per fer-ho hem seguit el següent principi: calcularem la descomposició de la direcció del vent per cada $\vec{e_i}$ i en calcularem el valor acumulat, després en guardarem el percentatge de cada direcció com l'increment. El codi resultant és el següent:

```
//Calculate addition values to F becauseof wind direction
float T = 0.f;
for(int i=0; i<9; i++) {
   a_addWind[i] = max(dot(a_wind_direction, direction[i]), 0.f);
   T += a_addWind[i];
}
for(int i=0; i<9; i++) {
   a_addWind[i] = (a_addWind[i]/T) * WIND_PARTICLES;
}
```

El següent pas important és com transformarem la imatge en informació útil per la simulació. Això ho farem calculant el tipus de cada cel·la. Per fer-ho seguirem el següent codi:

```
for(int i = 0; i<a_sim_height; i++) {
   for(int j=0; j<a_sim_width; j++) {
      pcolor = img->getPixel(j,i);
      if((pcolor.red==255)&&(pcolor.green==255)&&(pcolor.blue==255)) {
         a_ctype[i*a_sim_width + j] = H_FLUID;
      } else a_ctype[i*a_sim_width + j] = H_NOSLIP;
   }
}
//Modify external cells to OPENBOUNDARY
//Bottom & Top cells
for(int i=0; i<a_sim_width; i++) {
   a_ctype[(matdim-1)-i] = H_OPENBOUNDARY_S; //Bottom
   a_ctype[i] = H_OPENBOUNDARY_N;            //Top
}
//Left side & Right side cells
for(int i=0; i<a_sim_height; i++) {
   a_ctype[i*a_sim_width] = H_OPENBOUNDARY_W;         //Left side
   a_ctype[((i+1)*a_sim_width)-1] = H_OPENBOUNDARY_E; //Right side
}
//Modify neighbours of external cells if them follow the wind direction
int init;
if(dot(a_wind_direction,bottom_dir) > 0.f) {
   init = matdim - (2*a_sim_width) + 1;
   for(int i=0; i<(a_sim_width - 2); i++) a_ctype[init + i] = H_VELOCITY;
   for(int i=0; i<a_sim_width; i++) a_ctype[(matdim-1)-i] = H_OPENBOUNDARY_VEL;
}
if(dot(a_wind_direction,top_dir) > 0.f) {
   init = a_sim_width + 1;
   for(int i=0; i<(a_sim_width - 2); i++) a_ctype[init + i] = H_VELOCITY;
   for(int i=0; i<a_sim_width; i++) a_ctype[i] = H_OPENBOUNDARY_VEL;
}
if(dot(a_wind_direction,left_dir) > 0.f) {
   init = a_sim_width + 1;
   for(int i=0; i<(a_sim_height - 2); i++) a_ctype[init + (i*a_sim_width)] = H_VELOCITY;
   for(int i=0; i<a_sim_height; i++) a_ctype[i*a_sim_width] = H_OPENBOUNDARY_VEL;
}
if(dot(a_wind_direction,right_dir) > 0.f) {
   init = (a_sim_width - 1) * 2;
   for(int i=0; i<(a_sim_height - 2); i++) a_ctype[init + (i*a_sim_width)] = H_VELOCITY;
   for(int i=0; i<a_sim_height; i++) a_ctype[((i+1)*a_sim_width)-1] = H_OPENBOUNDARY_VEL;
}
```

Per acabar amb el mètode, veurem un exemple de crida a un *kernel* de la GPU:

```
CUDA_SAFE_CALL(cudaMemcpy(d_celltype, a_ctype, sizeof(unsigned int)*matdim,
cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy(d_addWind, a_addWind, sizeof(float)*9, cudaMemcpyHostToDevice));

//Executing GPU code
dim3 dimBlock(BLOCKSIZE);
dim3 dimGrid(matdim / BLOCKSIZE);
InitLBM<<<dimGrid, dimBlock>>>(d_celltype, d_f0, d_u, d_rho);

//Initialising values from GPU to CPU
CUDA_SAFE_CALL(cudaMemcpy(a_f0, d_f0, sizeof(float)*matdim*9, cudaMemcpyDeviceToHost));
CUDA_SAFE_CALL(cudaMemcpy(a_u, d_u, sizeof(float2)*matdim, cudaMemcpyDeviceToHost));
CUDA_SAFE_CALL(cudaMemcpy(a_rho, d_rho, sizeof(float)*matdim, cudaMemcpyDeviceToHost));
```

Com podem veure, el primer pas és copiar la informació necessària a la GPU, després cridem el *kernel* corresponen indicant el numero de blocs i quants *threads* té cada bloc. Finalment copiem els resultats de la GPU a la CPU.

- int nextIteration(float *velocities, float *densities):

```
int Simulator::nextIteration(float *velocities, float *densities) {
    int matdim = a_sim_width * a_sim_height;

    //Moving data to GPU
    CUDA_SAFE_CALL(cudaMemcpy(d_celltype, a_ctype, sizeof(unsigned int)*matdim,
cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(d_rho, a_rho, sizeof(float)*matdim, cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(d_f0, a_f0, sizeof(float)*matdim*9, cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(d_u, a_u, sizeof(float2)*matdim, cudaMemcpyHostToDevice));

    dim3 dimBlock(BLOCKSIZE);
    dim3 dimGrid(matdim / BLOCKSIZE);
    LBMstep<<<dimGrid,dimBlock>>>(d_celltype, d_rho, d_u, d_f0, d_f1, OMEGA, d_addWind);
    InitOpenLBM<<<dimGrid, dimBlock>>>(d_celltype, d_f1, d_u, d_rho);

    CUDA_SAFE_CALL(cudaMemcpy(a_f0, d_f1, sizeof(float)*matdim*9, cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL(cudaMemcpy(a_u, d_u, sizeof(float2)*matdim, cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL(cudaMemcpy(a_rho, d_rho, sizeof(float)*matdim, cudaMemcpyDeviceToHost));

    //Getting return values
    for(int i=0; i<matdim; i++) {
        velocities[i*2] = a_u[i].x;
        velocities[(i*2)+1] = a_u[i].y;
        densities[i] = a_rho[i];
    }

    return 0;
}
```

Aquest és el mètode encarregat de calcular la següent iteració de la simulació. A grans trets, s'encarrega de copiar la informació de la iteració anterior a la GPU per poder calcular la nova iteració i finalment copia els resultats a la CPU per poder-los visualitzar. Com ja hem esmentat anteriorment, per realitzar una simulació realista en els límits, després de cada iteració haurem de cridar el *kernel* InitOpenLBM.

- float* getVelocities():

```
float* Simulator::getVelocities() {
	int matdim = a_sim_width * a_sim_height;
	float *aux = new float[matdim*2];
	int iter;

	for(int i = 0; i < matdim; i++) {
		iter = i*2;
		aux[iter] = a_u[i].x;
		aux[iter+1] = a_u[i].y;
	}

	return aux;
}
```

Aquest és el mètode encarregar de consultar les velocitats resultants de l'última iteració de la simulació. Com veiem, abans de retornar les velocitats les hem de tractar. Això és a causa de com les llibreries de CUDA implementen el tipus float2, el qual representa una tupla de *floats*, però a la resta de classes no tenim aquest tipus. Per aquest motiu convertim la *array* de float2 a una *array* de *floats* de mida 2n amb la seqüencia: $x_0, y_0, x_1, y_1, \dots, x_n, y_n$.

- float* getDensities():

```
float* Simulator::getDensities() {
	return a_rho;
}
```

Aquest és el mètode encarregat de consultar les densitats resultants de l'última iteració de la simulació.

- float* getForces():

```
float* Simulator::getForces() {
	return a_f0;
}
```

Aquest és el mètode encarregat de consultar les forces resultants de l'última iteració de la simulació.

- unsigned int* getTypes():

```
unsigned int* Simulator::getTypes() {
	return a_ctype;
}
```

Aquest és el mètode encarregat de consultar la malla de tipus de la simulació.

- void setWindDirection(float x, float y):

Aquest és el mètode encarregat de modificar la direcció del vent de la simulació.

```
void setWindDirection(x, y) {
    actualitzar i normalitza wind_direction.
    recalcular els increments de vent segons nova direcció.
    eliminar les cel·les VELOCITY i OPENBOUNDARY_VEL.
    Calcular noves cel·les VELOCITY i OPENBOUNDARY_VEL.
    Copiar nova informació a la GPU.
}
```

Com veiem en el pseudocdi, tenim 5 passos diferenciats. En primer lloc hem d'actualitzar i normalitzar el vector director del vent.

```
a_wind_direction = make_float2(x,y);
//Normalitzation of wind direction
float wind_dir_modul = sqrt( (a_wind_direction.x*a_wind_direction.x) +
  (a_wind_direction.y*a_wind_direction.y) );
a_wind_direction = make_float2(a_wind_direction.x/wind_dir_modul,
   a_wind_direction.y/wind_dir_modul);
```

En segon lloc hem de calcular els nous increments de vents a partir d'aquesta nova direcció.

```
//Calculate addition values to F becauseof wind direction
float T = 0.f;
for(int i=0; i<9; i++) {
    a_addWind[i] = max(dot(a_wind_direction, direction[i]), 0.f);
    T += a_addWind[i];
}
for(int i=0; i<9; i++) {
    a_addWind[i] = (a_addWind[i]/T) * WIND_PARTICLES;
}
```

El següent pas és el d'eliminar les antigues cel·les VELOCITY i OPENBOUNDARY_VEL, ja que hem de simular la nova entrada de vent.

```
//Remove H_VELOCITY cells from types
init_a = a_sim_width + 1;                  //Init for top
init_b = matdim - (2*a_sim_width) + 1;     //Init for bottom
for(int i=0; i<(a_sim_width-2); i++) {
    a_ctype[init_a + i] = H_FLUID;
    a_ctype[init_b + i] = H_FLUID;
}

init_a = a_sim_width + 1;                         //Init for left
init_b = (a_sim_width - 1) * 2;                   //Init for right
for(int i=0; i<(a_sim_height-2); i++) {
    a_ctype[init_a + (i*a_sim_width)] = H_FLUID;
    a_ctype[init_b + (i*a_sim_width)] = H_FLUID;
}

//Remove H_OPENBOUDNARY_VEL cells from types
//Bottom & Top cells
for(int i=0; i<a_sim_width; i++) {
    a_ctype[(matdim-1)-i] = H_OPENBOUNDARY_S;  //Bottom
    a_ctype[i] = H_OPENBOUNDARY_N;             //Top
}
//Left side & Right side cells
for(int i=0; i<a_sim_height; i++) {
    a_ctype[i*a_sim_width] = H_OPENBOUNDARY_W;            //Left side
    a_ctype[((i+1)*a_sim_width)-1] = H_OPENBOUNDARY_E;    //Right side
}
```

Finalment, hem de calcular les noves cel·les VELOCITY i afegir-hi els límits personalitzats.

```
//Calculate H_VELOCITY cells with new wind_direction
int init;
if(dot(a_wind_direction,bottom_dir) > 0.f) {
    init = matdim - (2*a_sim_width) + 1;
    for(int i=0; i<(a_sim_width - 2); i++) a_ctype[init + i] = H_VELOCITY;
    for(int i=0; i<a_sim_width; i++) a_ctype[(matdim-1)-i] = H_OPENBOUNDARY_VEL;
}
if(dot(a_wind_direction,top_dir) > 0.f) {
   init = a_sim_width + 1;
   for(int i=0; i<(a_sim_width - 2); i++) a_ctype[init + i] = H_VELOCITY;
   for(int i=0; i<a_sim_width; i++) a_ctype[i] = H_OPENBOUNDARY_VEL;
}
if(dot(a_wind_direction,left_dir) > 0.f) {
   init = a_sim_width + 1;
   for(int i=0; i<(a_sim_height - 2); i++) a_ctype[init + (i*a_sim_width)] = H_VELOCITY;
   for(int i=0; i<a_sim_height; i++) a_ctype[i*a_sim_width] = H_OPENBOUNDARY_VEL;
}
if(dot(a_wind_direction,right_dir) > 0.f) {
   init = (a_sim_width - 1) * 2;
   for(int i=0; i<(a_sim_height - 2); i++) a_ctype[init + (i*a_sim_width)] = H_VELOCITY;
   for(int i=0; i<a_sim_height; i++) a_ctype[((i+1)*a_sim_width)-1] = H_OPENBOUNDARY_VEL;
}
```

En últim lloc, tenim la implementació dels diferents *kernels* de CUDA. Tots els mètodes que segueixen requereixen de la funció d'equilibri esmentada amb anterioritat. Per aquest motiu, el primer que farem serà veure com l'hem implementada.

```
__device__ float equilibrium(float rho, float2 uu, float weight, float2 vec) {
       float uvec = dot(uu, vec);
       return weight * (rho - 1.5f*dot(uu,uu) + 3.f*uvec + 4.5f*uvec*uvec);
}
```

Aquesta és la implementació de la funció d'equilibri. Fixem-nos en el detall en la declaració d'aquesta, '__device__', amb això estem definint aquesta funció com a privada dins la GPU.

- **__global__ void InitLBM(cells_types, f, velocities, densities):**

```
__global__ void InitLBM(unsigned int *celltype, float *f_0, float2 *uu, float *rho) {
    int pos = threadIdx.x + blockIdx.x * BLOCKSIZE;

    if(pos < SIM_WIDTH * SIM_HEIGHT) {
        int type = celltype[pos];

        if(D_OPENBOUNDARY_N == type || D_OPENBOUNDARY_S == type || D_OPENBOUNDARY_E == type
           || D_OPENBOUNDARY_W == type || D_OPENBOUNDARY_VEL == type) {
            uu[pos] = make_float2(0.f,0.f);
            rho[pos] = 0.f;
            for(int i=0; i<9; i++) {
                f_0[ref_addr[i] + pos] = equilibrium(1.f, make_float2(0.f,0.f), weight[i],
vectors[i]);
                uu[pos] += f_0[ref_addr[i] + pos] * vectors[i];
                rho[pos] += f_0[ref_addr[i] + pos];
            }
            return;
        }

       if(D_NOSLIP == type) {
            for(int i=0; i<9; i++) f_0[ref_addr[i] + pos] = 0.f;
            uu[pos] = make_float2(0.f, 0.f);
            rho[pos] = 0.f;
            return;
            //boundary cells won't be accessed,
            //but we init the vel to avoid random values on output images
        }

        float2 u = make_float2(0.f, 0.f);
        for(int i=0; i<9; i++) {
            f_0[ref_addr[i] + pos] = equilibrium(1.f, u, weight[i], vectors[i]);
            //Init to rest
        }
    }
}
```

Aquest és el *kernel* encarregat d'inicialitzar els valors de la simulació en la GPU. Com podem veure, segons quin tipus de cel·la sigui, inicialitzarà les forces, densitats i velocitats d'una manera o altre. El cas més especial és el dels límits, els quals inicialitzem en l'equilibri com si fossin una cel·la FLUID, per tal que durant la simulació les cel·les interiors tinguin partícules de vent per propagar.

- **__global__ void initOpenLBM(cells_types, f, velocities, densities):**

```
__global__ void InitOpenLBM(unsigned int *celltype, float *f_0, float2 *uu, float *rho) {
    int pos = threadIdx.x + blockIdx.x * BLOCKSIZE;

    if(pos < SIM_WIDTH * SIM_HEIGHT) {
        unsigned int ct = celltype[pos];

        if(ct == D_FLUID || ct == D_NOSLIP || ct == D_VELOCITY) return;
        if(ct == D_OPENBOUNDARY_N) {
            if(pos == 0) { //Top Left corner
                rho[pos] = rho[(pos + 1) + SIM_WIDTH];
                uu[pos] = uu[(pos + 1) + SIM_WIDTH];
            } else if(pos == (SIM_WIDTH * SIM_HEIGHT - SIM_WIDTH)) { //Bottom Left corner
                rho[pos] = rho[(pos + 1) - SIM_WIDTH];
                uu[pos] = uu[(pos + 1) - SIM_WIDTH];
            } else {
                rho[pos] = rho[pos + SIM_WIDTH];
                uu[pos] = uu[pos + SIM_WIDTH];
            }
        } else if(ct == D_OPENBOUNDARY_S) {
            if(pos == 0) { //Top Left corner
                rho[pos] = rho[(pos + 1) + SIM_WIDTH];
                uu[pos] = uu[(pos + 1) + SIM_WIDTH];
            } else if(pos == (SIM_WIDTH * SIM_HEIGHT - SIM_WIDTH)) { //Bottom Left corner
                rho[pos] = rho[(pos + 1) - SIM_WIDTH];
                uu[pos] = uu[(pos + 1) - SIM_WIDTH];
            } else {
                rho[pos] = rho[pos - SIM_WIDTH];
                uu[pos] = uu[pos - SIM_WIDTH];
            }
        } else if(ct == D_OPENBOUNDARY_E) {
            if(pos == (SIM_WIDTH - 1)) { //Top Right corner
                rho[pos] = rho[(pos - 1) + SIM_WIDTH];
                uu[pos] = uu[pos - 1];
            } else if(pos == (SIM_WIDTH * SIM_HEIGHT - 1)) { //Bottom Right corner
                rho[pos] = rho[(pos - 1) - SIM_WIDTH];
                uu[pos] = uu[(pos - 1) - SIM_WIDTH];
            } else {
                rho[pos] = rho[pos - 1];
                uu[pos] = uu[pos - 1];
            }
        } else { //D_OPENBOUNDARY_W
            if(pos == 0) { //Top Left corner
                rho[pos] = rho[(pos + 1) + SIM_WIDTH];
                uu[pos] = uu[(pos + 1) + SIM_WIDTH];
            } else if(pos == (SIM_WIDTH * SIM_HEIGHT - SIM_WIDTH)) { //Bottom Left corner
                rho[pos] = rho[(pos + 1) - SIM_WIDTH];
                uu[pos] = uu[(pos + 1) - SIM_WIDTH];
            } else {
                rho[pos] = rho[pos + 1];
                uu[pos] = uu[pos + 1];
            }
        }

        for(int i=0; i<9; i++) {
            f_0[pos + ref_addr[i]] = equilibrium(rho[pos], uu[pos], weight[i], vectors[i]);
        }
    }
}
```

Aquest *kernel* és l'encarregat de copiar la informació de les cel·les interiors pels límits. Tot i semblar un mètode complex, és molt senzill. L'objectiu principal és el de copiar les velocitats i densitats de les cel·les veïnes interiors a cada límit, l'únic problema és que hem de tractar tota la casuística de quina posició ocupa el *thread* actual, és a dir, els límits N hauran de copiar el veí de sota, les S el de sobre o, per exemple, el de la cantonada superior esquerra ha de copiar del veí en diagonal una cel·la a sota i una a la dreta.

- **__global__ void LBMstep(cells_types, densities, velocities, f0, f1, omega, addWind):**

```
__global__ void LBMstep(unsigned int *celltype, float *rho, float2 *u, float *f_0,
                         float *f_1, float omega, float *addWind) {
    int pos = threadIdx.x + blockIdx.x * BLOCKSIZE;

    if(pos < SIM_WIDTH * SIM_HEIGHT) {
        unsigned int ct = celltype[pos];
        if(D_NOSLIP == ct || D_OPENBOUNDARY_N == ct || D_OPENBOUNDARY_S == ct ||
D_OPENBOUNDARY_E == ct || D_OPENBOUNDARY_W == ct || D_OPENBOUNDARY_VEL == ct) return;
//Early exit for boundary cells

        //Stream
        float in_f[9];

        in_f[0] = f_0[pos];
        for(int i=1; i<9; i++) {
            //if next cell is boundary, flip actual df
            if(D_NOSLIP == celltype[pos+next[i]]) in_f[i] = f_0[ref_addr[inv[i]] + pos];
            else if(D_OPENBOUNDARY_VEL == celltype[pos+next[i]]) in_f[i] = equilibrium(1.f,
                     make_float2(0.f, 0.f), weight[i], vectors[i]);
            else in_f[i] = f_0[ref_addr[i] + pos+next[i]]; // stream normal
        }
        __syncthreads();

        //Collide
        if(D_VELOCITY == ct) {
            for(int i=1; i<9; i++) {
                in_f[i] += addWind[i];
            }
        }
        float n_rho = in_f[0];
        float2 n_u = make_float2(0.f, 0.f);

        for(int i=1; i<9; i++) {
            n_rho += in_f[i];
            n_u += in_f[i] * vectors[i];
        }
        for(int i=0; i<9; i++)
            in_f[i] -= omega * (in_f[i] - equilibrium(n_rho, n_u, weight[i], vectors[i]));

        __syncthreads();

        for(int i=0; i<9; i++)
            f_1[ref_addr[i] + pos] = in_f[i];

        rho[pos] = n_rho;
        u[pos] = n_u;
    }
}
```

Aquest *kernel* és l'encarregat de realitzar un pas de simulació segons el mètode de *Lattice-Boltzmann*. Com veiem, tenim una primera fase on realitzem la propagació de partícules respecte la iteració anterior i després en calculem els nous valors.

El punt més críptic d'aquest mètode és la variable next[]. Aquesta determina per cada força quina és la cel·la connexa, és a dir, d'on provenen les noves partícules. Ho podrem veure més clarament amb un exemple.

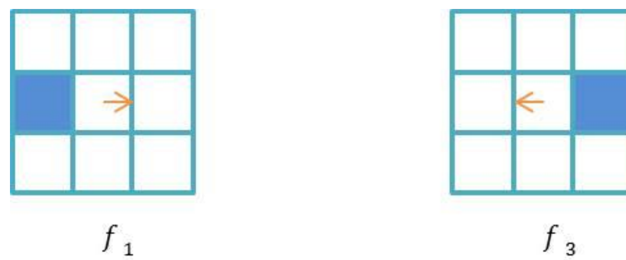


Figura 9.1: Exemple de *next* per f_1 i f_3.

Així doncs, si ens trobem en l'exemple de f_3 i next fos un obstacle, la nova força seria la f_1 de la iteració anterior.

Figura 9.2: Exemple de col·lisió per f_1 i f_3.

Apartat 9.2: Interfície d'usuari

En aquest apartat, veurem com serà la interfície d'usuari i com hi podrà interactuar l'usuari. Com ja hem especificat anteriorment, el disseny de la interfície ha de ser simple i senzill d'entendre per qualsevol usuari. Tot seguit, en la figura 9.3, podrem veure un esquema aproximat del que serà la interfície d'usuari.

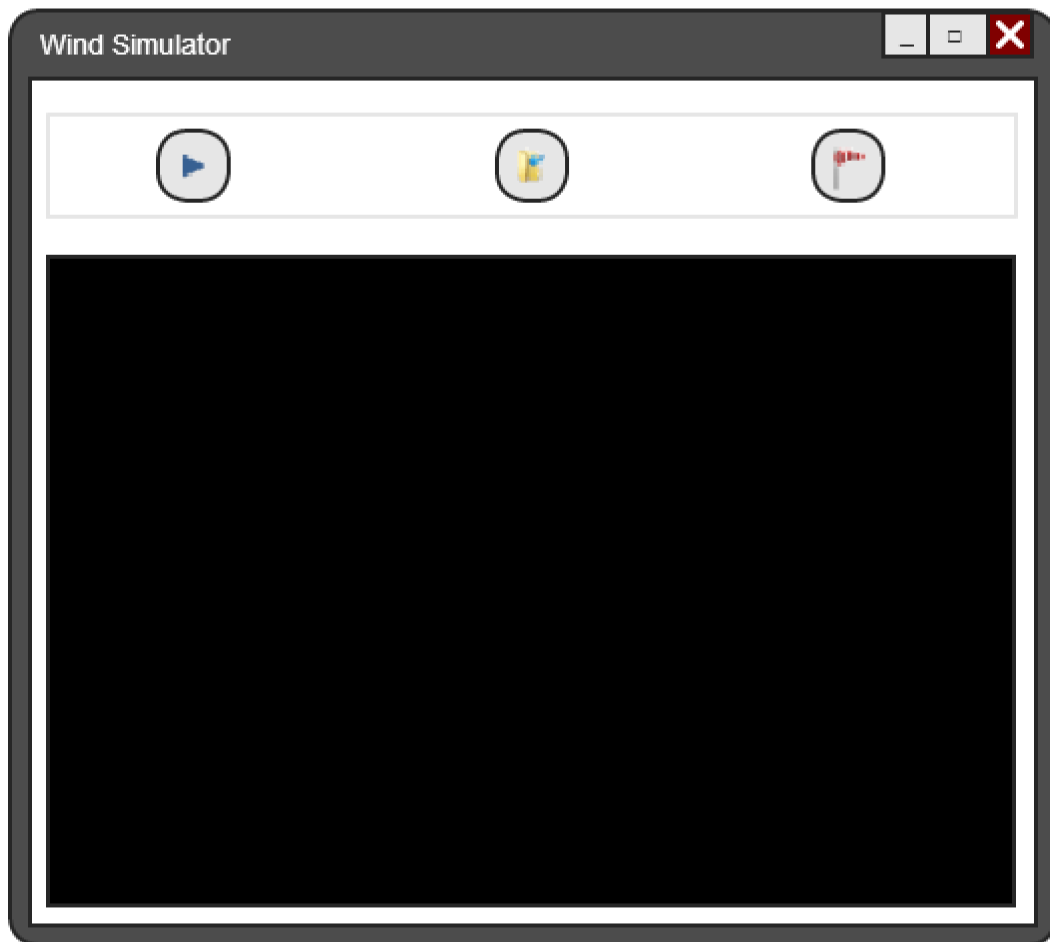


Figura 9.3: Esquema de la interfície d'usuari.

Tal i com podem veure en la figura 9.3, la nostra interfície és basa en 3 parts. En primer lloc tenim els típics botons per gestionar la finestra, els quals ens permetran minimitzar, maximitzar o tancar la finestra. Per altre banda tenim una secció amb botons, que permetran a l'usuari interactuar amb el simulador. Finalment, tenim una finestra on podrem visualitzar elements mitjançant OpenGL.

Si ens fixem més concretament en la secció de botons, de forma intuïtiva podem saber que farà cada botó. El primer permetrà a l'usuari activar la simulació o parar-la. El segon, permetrà a l'usuari seleccionar una imatge sobre la qual realitzar la simulació. Finalment, el tercer, permetrà a l'usuari modificar la direcció del vent de la simulació. Per seleccionar el vent es mostrà una nova finestra, tal i com veiem en la figura 9.4.

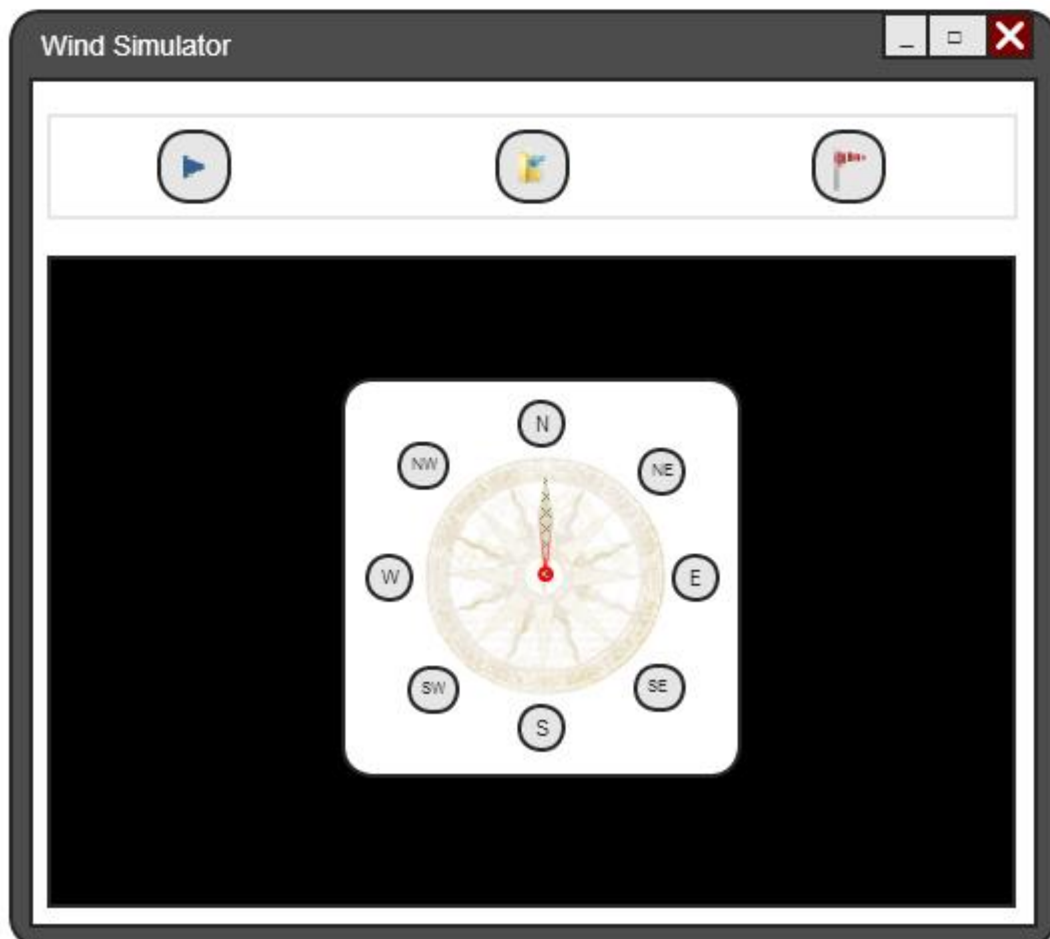


Figura 9.4: Esquema de la interfície d'usuari si cliquem el botó per canviar la direcció del vent.

Per realitzar la implementació d'aquesta interfície, s'han utilitzat les llibreries Qt, tal i com s'especifica en l'apartat d'estudis i decisions. Si ens fixem en el diagrama de classes, la interfície esta implementada en les classes QtDisplayer, GLWidget, WindChanger i MyQPushButton. Tot seguit veurem el codi més important d'aquestes classes per desenvolupar la interfície.

QtDisplayer

La classe QtDisplayer, tal i com s'ha explicat prèviament, és la encarregada de crear la interfície d'usuari i de gestionar les interaccions de l'usuari. Així doncs, tot seguit veurem els mètodes més important de la classe.

- QtDisplayer(QWidget * parent):
  Aquest és el constructor de la classe i s'encarrega d'inicialitzar tots els elements de la interfície. Els passos que seguirem seran els d'inicialitzar els atributs basics, crear un *layout* per posicionar els diferents elements i crear aquest elements, que són el conjunt de botons, el GLWidget i el WindChanger.

```
QtDisplayer::QtDisplayer(QWidget *parent)
	: QWidget(parent)
{
	ui.setupUi(this); //create the window

	play = true;
	scene_load = false;

	QVBoxLayout *mainLayout = new QVBoxLayout(this);

	createButtonsGroupBox();
	mainLayout->addWidget(m_groupBoxButtons);

	m_glwidget = new GLWidget(parent);
	mainLayout->addWidget(m_glwidget, 0);

	createWindChanger();
}
```

Com podem veure en el codi, en primer lloc creem la finestra sobre la que representarem la resta d'elements i inicialitzem els *flags* que gestionaran la simulació. Després crearem el *layout* i els elements que aquest contindrà. Tot seguit veurem el codi per crear aquests elements.

El primer element del que veurem el codi serà del conjunt de botons. Bàsicament, la feina que haurem de realitzar és:

- Crear un nou *layout* que gestioni els diferents botons
- Per cada botó crear un nou *MyQPushButton*.
- Assignar al botó una icona adient.
- Assignar al botó un esdeveniment que l'activi i la funció a executar quan s'activi.
- Assignar el botó al *layout*.

```
void QtDisplayer::createButtonsGroupBox() {
   m_groupBoxButtons = new QGroupBox(tr("Buttons"));

   QHBoxLayout *layout = new QHBoxLayout;

   m_buttonPlayPause = new MyQPushButton(this);
   m_buttonPlayPause->setIcon(QIcon("Images/PlayButton_nb.png"));
   connect(m_buttonPlayPause, SIGNAL(clicked()), this, SLOT(buttonPlayPauseClicked()));

   m_buttonOpenScene = new MyQPushButton(this);
   m_buttonOpenScene->setIcon(QIcon("Images/OpenSceneButton_nb.png"));
   connect(m_buttonOpenScene, SIGNAL(clicked()), this, SLOT(buttonOpenSceneClicked()));

   m_buttonChangeWindDir = new MyQPushButton(this);
   m_buttonChangeWindDir->setIcon(QIcon("Images/ChangeWindButton_nb.png"));
   connect(m_buttonChangeWindDir, SIGNAL(clicked()), this, SLOT(buttonChangeWindClicked()));

   layout->addWidget(m_buttonPlayPause, 0, Qt::AlignHCenter);
   layout->addWidget(m_buttonOpenScene, 0, Qt::AlignHCenter);
   layout->addWidget(m_buttonChangeWindDir, 0, Qt::AlignHCenter);

   m_groupBoxButtons->setLayout(layout);
}
```

El següent element a construir serà el *GLWidget*, per fer-ho utilitzarem el constructor de la classe que encapsula la informació necessària per la seva creació. Més endavant veurem el codi darrera aquest mètode.

```
m_glwidget = new GLWidget(parent);
mainLayout->addWidget(m_glwidget, 0);
```

Finalment, crearem el *WindChanger*, element encarregat de modificar la direcció del vent de la simulació. Per fer-ho utilitzarem el constructor de la classe per crear-lo i en modificarem la visualització amb la finalitat que sembli una finestra emergent quan es pressioni el botó corresponent. El codi corresponent és el següent.

```
void QtDisplayer::createWindChanger() {
    m_windchanger = new WindChanger(this);
    m_windchanger->setFixedSize(QSize(200,200));

    QPalette *Pal = new QPalette();
    Pal->setColor(QPalette::Background, Qt::white);
    m_windchanger->setAutoFillBackground(true);
    m_windchanger->setPalette(*Pal);

    m_windchanger->setParent(this);
    m_windchanger->hide();

    m_windchanger->setWindDirection(QString("N"));

    QPoint localPoint(300, 200);
    QPoint windowPoint = this->mapTo(this->window(), localPoint);
    m_windchanger->move(windowPoint);

    connect(m_windchanger, SIGNAL(windChanged()), this, SLOT(updateWindDirection()));
}
```

- Void buttonPlayPauseClicked():

  Aquest és el mètode encarregat de gestionar el comportament del botó *PlayPause*, és a dir, el botó el qual permetrà al usuari aturar la simulació o continuar-la. Com veurem tot seguit, el codi haurà de comprovar si la simulació és pot continuar o s'ha d'aturar. En cas de poder continuar, s'ha de tenir en compte el cas de que no hi hagi cap escena carregada. Finalment, s'haurà d'actualitzar la icona del botó per fer-lo més intuïtiu.

```
void QtDisplayer::buttonPlayPauseClicked() {
    if(play) {
        if(m_glwidget->getFilename() != QString(QLatin1String(""))) {
            m_glwidget->continueSimulation();
            play = false;
            m_buttonPlayPause->setIcon(QIcon("Images/PauseButton_nb.png"));
        } else {
            QMessageBox::information(this, tr("Error Console"), "No scene load, open a scene 
first");
        }
    } else {
        m_glwidget->stopSimulation();
        play = true;
        m_buttonPlayPause->setIcon(QIcon("Images/PlayButton_nb.png"));
    }
}
```

- Void buttonOpenSceneClicked():

  Aquest és el mètode encarregat de gestionar el comportament del botó *OpenScene*, és a dir, el botó el qual permetrà al usuari carregar una escena. El comportament d'aquest mètode és el següent: si la simulació està activa, l'aturarem. A continuació obtindrem el *path* que l'usuari hagi indicat i carregarem l'escena en qüestió.

```
void QtDisplayer::buttonOpenSceneClicked() {
    QString filename;

    if(!play) {
        m_glwidget->stopSimulation();
        play = true;
        m_buttonPlayPause->setIcon(QIcon("Images/PlayButton_nb.png"));
    }

    filename = QFileDialog::getOpenFileName(this,tr("Open Scene"),"C://","All Files (*.*)");
    if(filename != QString("")) {
        scene_load = true;
        m_glwidget->setFilename(filename);
        m_glwidget->loadScene();
    }
}
```

- Void buttonChangeWindClicked():

  Aquest és el mètode encarregat de gestionar el comportament del botó ChangeWind, és a dir, el botó el qual permetrà al usuari modificar la direcció del vent de la simulació. El comportament d'aquest mètode serà el d'aturar la simulació si esta activa i mostrar el widget per modificar la direcció del vent.

```
void QtDisplayer::buttonChangeWindClicked() {
    if(scene_load) {
        m_glwidget->stopSimulation();
        m_windchanger->show();
    } else {
        QMessageBox::information(this, tr("Error"), "No scene load, open a scene first");
    }
}
```

- Void updateWindDirection():

  Aquest és el mètode que s'executarà en resposta quan és modifiqui la direcció del vent dins del widget WindChanger. El comportament d'aquest serà el d'obtenir la direcció decidida per l'usuari, modificar la direcció del vent en la simulació i, si cal, continuar amb la simulació.

```
void QtDisplayer::updateWindDirection() {
    QString *dir =  m_windchanger->getWind();
    m_glwidget->setWindDirection(*dir);
    if(!play) m_glwidget->continueSimulation();
}
```

Un cop vist la classe principal encarregada de gestionar la interfície d'usuari, veurem els diferents subelements esmentats.

GLWidget

La classe GLWidget, tal i com hem explicat prèviament, és l'encarregada de visualitzar la simulació de l'escena i de fer d'enllaç entre els canvis proposats per l'usuari i la simulació. Així doncs, tot seguit veurem els mètodes més important de la classe.

- GLWidget(QWidget *parent):

  Aquest és el constructor de la classe, com a tal, és l'encarregat d'inicialitzar els atributs i de la classe. A més, s'encarrega de crear un temporitzador per tal d'actualitzar la visualització de l'escena periòdicament.

```
GLWidget::GLWidget(QWidget *parent) : QGLWidget(parent) {
    qtPurple = QColor::fromCmykF(0.39, 0.39, 0.0, 0.0);
    filename = QString(QLatin1String(""));
    simulate = false;
    iter = 0;
    vertices = new GLfloat[0];
    colors = new GLfloat[0];
    indices = new GLuint[0];
    indices_length = 0;

    wind_dir = new QString("N");

    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(update()));
    timer->start(1);
}
```

- ~GLWidget():

  Aquest és el destructor de la classe, s'encarrega d'alliberar l'espai de memòria ocupat pels atributs de la classe. Els elements a eliminar de la classe són: els vèrtexs del model, els colors, les cares i la imatge d'on hem carregat l'escena.

```
GLWidget::~GLWidget() {
    //Delete data
    delete [] vertices;
    delete [] colors;
    delete [] indices;

    delete image;
}
```

- void loadScene():

  Aquest mètode és l'encarregat de carregar una escena al sistema. Per fer-ho, en primer lloc si hi ha una imatge d'una escena anterior l'eliminem. A continuació llegim la nova imatge, mitjançant l'ImageReader, a partir del path introduït previament per l'usuari. El següent pas consisteix en crear i inicialitzar el simulador, i assignar-li la direcció del vent. Després inicialitzarem les variables on guardarem els resultats de cada iteració i crearem el model de la simulació per visualitzar-la.

```
void GLWidget::loadScene() {
    //Delete previous image and simulation, if no first exec.
    if(first_exec) first_exec = false;
    else delete image;

    iter = 0;
    //Read new image
    //To read the image, we need to cast the QString to char*
    QByteArray ba = filename.toLatin1(); //First we convert to QByteArray
    char *image_path = ba.data();        //Then we extract the pointer to the data
    image = ir.readImage(image_path);

    scene_width = image->getWidth();
    scene_height = image->getHeight();
    //Create new simulator
    sim = new Simulator(scene_width, scene_height);
    sim_types = sim->initialise(image);
    //Set simulation default wind direction
    if(wind_dir == QString("N")) sim->setWindDirection(0.f,1.f);
    else if(wind_dir == QString("NE")) sim->setWindDirection(1.f,1.f);
    else if(wind_dir == QString("E")) sim->setWindDirection(1.f,0.f);
    else if(wind_dir == QString("SE")) sim->setWindDirection(1.f,-1.f);
    else if(wind_dir == QString("S")) sim->setWindDirection(0.f,-1.f);
    else if(wind_dir == QString("SW")) sim->setWindDirection(-1.f,-1.f);
    else if(wind_dir == QString("W")) sim->setWindDirection(-1.f,0.f);
    else if(wind_dir == QString("NW")) sim->setWindDirection(-1.f,1.f);
    else printf("Error, bad wind direction\n");
    //Init sim data
    sim_velocities = new float[scene_width * scene_height *2];
    sim_densities = new float[scene_width * scene_height];
    //Prepare display data
    int width, height;
    width = this->width();
    height = this->height();
    float width2 = 0.99f, height2 = 0.99f;

    float edgeX = width2 / (float)scene_width;
    float edgeY = height2 / (float)scene_height;
    cube_edge = std::min(edgeX, edgeY);
    cubes_offset = 0.1f * cube_edge;
    cubes_offset = 0.f; //!!!!
    //Initialise data info to draw
    initVertices(scene_width, scene_height);
    initColors(scene_width, scene_height, sim_types);
    initIndices(scene_width, scene_height);
    //Load vertex and color info
    glColorPointer(3, GL_FLOAT, 0, colors);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}
```

- void continueSimulation():

  Aquest mètode és l'encarregat de permetre a la simulació continuar l'execució. Simplement aquest mètode gestiona un booleà el qual hi assignarà un cert. D'aquesta manera, en el mètode paintGL, segons aquest booleà és calcularan noves iteracions de la simulació o no, segons calgui.

```
void GLWidget::continueSimulation() {
    simulate = true;
}
```

- void stopSimulation():

  Aquest mètode és l'encarregat d'aturar la simulació. Anàlogament al mètode anterior, aquest hi assignarà un fals.

```
void GLWidget::stopSimulation() {
    simulate = false;
}
```

- void initializeGL():

  Aquest mètode és l'encarregat d'inicialitzar diversos valors d'OpenGL per tal de representar el model que s'ha creat de la simulació. La part que més interesa és la de l'activació dels buffers pels vèrtexs i els colors.

```
void GLWidget::initializeGL() {
    QColor *c = new QColor(0.f,0.f,0.f,0.f);
    qglClearColor(*c);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glShadeModel(GL_SMOOTH);

    //Enable array structure to draw
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);
}
```

- void paintGL():

  Aquest és el mètode encarregad de visualitzar el model creat per a la simulació. Per fer-ho, utilitzarem mètodes d'OpenGL. El comportament bàsic d'aquest mètode és el de netejar els buffers dels elements pintats per pantalla, calcular una nova iteració de la simulació, actualitzar els colors segons els resultats de la nova iteració i pintar-la. Això serà així, sempre i quan l'usuari no hagi aturat la simulació, en cas d'estar aturada, no calcularem cap iteració nova de la simulació i repintarem el model tal i com estava.

```
void GLWidget::paintGL() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -10.0);

    if(simulate) {
        if(sim->nextIteration(sim_velocities, sim_densities) != 0) {
            printf("Error in iteration %d of simumlation", iter);
        }
        //Recalculate colors of simulation and draw all scene again
        updateColors(sim_types, scene_width, scene_height, sim_densities);
        iter++;
    }

    glDrawElements(GL_TRIANGLES, indices_length, GL_UNSIGNED_INT, indices);
}
```

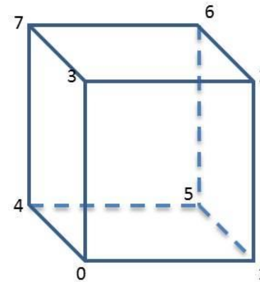
- void setWindDirection(QString dir):

  Aquest mètode és l'encarregat de modificar la direcció del vent de la simulació segons el valor entrat per l'usuari. Per fer-ho, comprovarem que la nova direcció sigui diferent a l'anterior i, en cas afirmatiu, assignarem aquesta direcció al simulador.

```
void GLWidget::setWindDirection(QString dir) {
    if(dir != *wind_dir) {
        if(dir == QString("N")) {
            wind_dir = new QString(dir); sim->setWindDirection(0.f,1.f);
        } else if(dir == QString("NE")) {
            wind_dir = new QString(dir); sim->setWindDirection(1.f,1.f);
        } else if(dir == QString("E")) {
            wind_dir = new QString(dir); sim->setWindDirection(1.f,0.f);
        } else if(dir == QString("SE")) {
            wind_dir = new QString(dir); sim->setWindDirection(1.f,-1.f);
        } else if(dir == QString("S")) {
            wind_dir = new QString(dir); sim->setWindDirection(0.f,-1.f);
        } else if(dir == QString("SW")) {
            wind_dir = new QString(dir); sim->setWindDirection(-1.f,-1.f);
        } else if(dir == QString("W")) {
            wind_dir = new QString(dir); sim->setWindDirection(-1.f,0.f);
        } else if(dir == QString("NW")) {
            wind_dir = new QString(dir); sim->setWindDirection(-1.f,1.f);
        } else {
            QMessageBox::information(this, tr("Error"), "Bad wind direction");
        }
    }
}
```

- void initVertices(int w, int h):

Aquest és el mètode encarregat d'incialitzar els vèrtexs del model de l'escena. Com ja s'ha explicat amb anterioritat, el model consisteix en transformar cada píxel de la imatge en un cub. Els vèrtexs de cada cub estràn ordenats de la següent manera:



Així doncs, partint d'aquesta estructura, crearem tants cubs com píxels tingui la imatge de l'escena. El primer que farem serà calcular la posició del vèrtex 0 del primer píxel, el de la part superior esquerra, i a partir d'aquesta posició anirem crean la resta.

```
void GLWidget::initVertices(int w, int h) {
   vertices = new GLfloat[8*3*w*h];
   //Calculate the init position, it'll be top left
   float initX = -1.f * ( ((float)w / 2.f) * cube_edge);
   float initY = ( ((float)h / 2.f) * cube_edge);
   int init_index;
   float posX, posY;
   for(int i=0; i<h; i++) {
      for(int j=0; j<w; j++) {
         init_index = (i*w + j) * 8*3; //Each iteration will write 24 values (8 vertices * 3 components x,y,z
         //Calculate coord for vertex 0 of current cube
         posX = initX + j*(cube_edge + cubes_offset);
         posY = initY - i*(cube_edge + cubes_offset);

   vertices[init_index + 0] = posX;             vertices[init_index + 1] = posY;              vertices[init_index + 2] = 0.f; //Vertex 0
   vertices[init_index + 3] = posX + cube_edge; vertices[init_index + 4] = posY;              vertices[init_index + 5] = 0.f; //Vertex 1
   vertices[init_index + 6] = posX + cube_edge; vertices[init_index + 7] = posY + cube_edge;  vertices[init_index + 8] = 0.f; //Vertex 2
   vertices[init_index + 9] = posX;             vertices[init_index + 10] = posY + cube_edge; vertices[init_index + 11] = 0.f; //Vertex 3

   vertices[init_index + 12] = posX;             vertices[init_index + 13] = posY;             vertices[init_index + 14] = -cube_edge; //Vertex 4
   vertices[init_index + 15] = posX + cube_edge; vertices[init_index + 16] = posY;             vertices[init_index + 17] = -cube_edge; //Vertex 5
   vertices[init_index + 18] = posX + cube_edge; vertices[init_index + 19] = posY + cube_edge; vertices[init_index + 20] = -cube_edge; //Vertex 6
   vertices[init_index + 21] = posX;             vertices[init_index + 22] = posY + cube_edge; vertices[init_index + 23] = -cube_edge; //Vertex 7
      }
   }
}
```

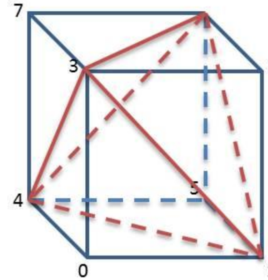
- void initColors(int w, int h, unsigned int *types):

  Aquest és el mètode encarregat d'incialitzar el color de cada vèrtex del model creat. Per tant, treballarem de forma anàloga al mètode anterior amb l'excepció que ara tindrem el tipus de cada cel·la, i segons aquest decidirem el color que tindra cada cub.

```
void GLWidget::initColors(int w, int h, unsigned int *types) {
    float red, green, blue;
    colors = new GLfloat[8*3*w*h];
    int init_index;
    for(int i=0; i<h; i++) {
        for(int j=0; j<w; j++) {
            init_index = (i*w + j) * 8*3; //Each iteration will write 24 values (8
vertices.color * 3 components x,y,z)

            switch(types[i*w + j]) {
                case H_FLUID:
                    red = 0.f; green = 0.f;     blue = 1.f;
                    break;
                case H_NOSLIP:
                    red = 1.f; green = 0.99216f; blue = 0.81569f;
                    break;
                case H_VELOCITY:
                    red = 1.f; green = 0.f; blue = 0.f;
                    break;
                case H_OPENBOUNDARY_N:
                    red = 0.f; green = 1.f; blue = 0.f;
                    break;
                case H_OPENBOUNDARY_S:
                    red = 0.f; green = 1.f; blue = 0.f;
                    break;
                case H_OPENBOUNDARY_E:
                    red = 0.f; green = 1.f; blue = 0.f;
                    break;
                case H_OPENBOUNDARY_W:
                    red = 0.f; green = 1.f; blue = 0.f;
                    break;
                case H_OPENBOUNDARY_VEL:
                    red = 0.f; green = 1.f; blue = 0.f;
                    break;
                default:
                    break;
            }
    colors[init_index+0]=red; colors[init_index+1]=green; colors[init_index+2]=blue;   //v0
    colors[init_index+3]=red; colors[init_index+4]=green; colors[init_index+5]=blue;   //v1
    colors[init_index+6]=red; colors[init_index+7]=green; colors[init_index+8]=blue;   //v2
    colors[init_index+9]=red; colors[init_index+10]=green; colors[init_index+11]=blue; //v3

    colors[init_index+12]=red; colors[init_index+13]=green; colors[init_index+14]=blue; //v4
    colors[init_index+15]=red; colors[init_index+16]=green; colors[init_index+17]=blue; //v5
    colors[init_index+18]=red; colors[init_index+19]=green; colors[init_index+20]=blue; //v6
    colors[init_index+21]=red; colors[init_index+22]=green; colors[init_index+23]=blue; //v7
        }
    }
}
```

- void initIndices(int w, int h):

    Aquest és el mètode encarregat de crear les cares del model de la simulació. Aquestes cares, seran triangles els quals units formaran un cub. El següent esquema ens mostrarà els triangles dels quals estaran formats els cubs.



    Així doncs, seguint l'estructura d'exemple, crearem les cares necessàries per crear un cub complet. Cada cara serà una terna amb els vèrtexs que la formen.

```
void GLWidget::initIndices(int w, int h) {
   indices = new GLuint[6*2*3*w*h]; //We got 6 faces, each face 2 triangle and each triangle 3 vertices
   indices_length = 6*2*3*w*h;

   int init_index, init_vertex;
   for(int i=0; i<h; i++) {
       for(int j=0; j<w; j++) {
          init_index = (i*w + j) * 6*2*3;//Each iteration will write 32 values (6 faces * 2 trianglePerFace * 3 verticesPerTriangle
          init_vertex = (i*w + j) * 8;//Each cube works with 8 vertices, vertices[(0,1,2,3,4,5,6,7)cube0,(,8,9,10,11,12,13,14,15)cube1,...]

          indices[init_index + 0] = init_vertex + 0; indices[init_index + 1] = init_vertex + 1; indices[init_index + 2] = init_vertex + 3;
          indices[init_index + 3] = init_vertex + 0; indices[init_index + 4] = init_vertex + 3; indices[init_index + 5] = init_vertex + 4;
          indices[init_index + 6] = init_vertex + 0; indices[init_index + 7] = init_vertex + 4; indices[init_index + 8] = init_vertex + 1;

          indices[init_index + 9]  = init_vertex + 2; indices[init_index + 10] = init_vertex + 3; indices[init_index + 11] = init_vertex + 1;
          indices[init_index + 12] = init_vertex + 2; indices[init_index + 13] = init_vertex + 1; indices[init_index + 14] = init_vertex + 6;
          indices[init_index + 15] = init_vertex + 2; indices[init_index + 16] = init_vertex + 6; indices[init_index + 17] = init_vertex + 3;

          indices[init_index + 18] = init_vertex + 5; indices[init_index + 19] = init_vertex + 1; indices[init_index + 20] = init_vertex + 4;
          indices[init_index + 21] = init_vertex + 5; indices[init_index + 22] = init_vertex + 6; indices[init_index + 23] = init_vertex + 1;
          indices[init_index + 24] = init_vertex + 5; indices[init_index + 25] = init_vertex + 4; indices[init_index + 26] = init_vertex + 6;

          indices[init_index + 27] = init_vertex + 7; indices[init_index + 28] = init_vertex + 4; indices[init_index + 29] = init_vertex + 3;
          indices[init_index + 30] = init_vertex + 7; indices[init_index + 31] = init_vertex + 3; indices[init_index + 32] = init_vertex + 6;
          indices[init_index + 33] = init_vertex + 7; indices[init_index + 34] = init_vertex + 6; indices[init_index + 35] = init_vertex + 4;
      }
   }
}
```

- void updateColors(unsigned int *types, int w, int h, float *densities):

  Aquest és el mètode encarregat d'actualitzar els colors del model a partir de les densitats. Per fer-ho, seguirem el mateix esquema que quan inicialitzàvem els colors, amb l'única diferencia que en les cel·les H_FLUID el càlcul del color serà de la següent manera:

```
case H_FLUID:
    red = 0.f;
    aux = densities[i*w + j] - MIN_VAL;
    green = (aux/(MAX_VAL-MIN_VAL));
    blue = 1.f;
    break;
```

WindChanger

La classe WindChanger, tal i com hem explicat prèviament, és l'encarregada de permetre a l'usuari modificar la direcció del vent de la simulació. Com ja hem pogut veure en la figura 9.4, l'esquema de la interfície d'usuari, aquesta classe ha de crear una finestra emergent, la qual permetrà a l'usuari escollir entre 8 direccions diferents i l'escollida s'assignarà a la simulació. Així doncs, tot seguit veurem els mètodes més important de la classe.

- WindChanger(QWidget *parent):

  Aquest és el constructor de la classe encarregat d'inicialitzar els elements de la classe. Com és tracta d'un subelement de la interfície d'usuari, l'objectiu principal d'aquest mètode serà el de crear la seva visualització. Els passos que seguirà són:

  - Crear un *layout* que gestionara els diferent elements.
  - Crear cadascun dels botons amb el logo corresponent.
  - Assignar al botó la funció a executar quan es premi.
  - Inserir la imatge de fons d'una brúixola per fer-ho més intuïtiu.
  - Assignar tots aquests elements al *layout*.

```
WindChanger::WindChanger(QWidget *parent) : QWidget(parent) {
    m_layout = new QGridLayout(this);

    m_buttonN = new MyQPushButton(this);
    m_buttonN->setIcon(QIcon("Images/NButton_nb.png"));
    connect(m_buttonN, SIGNAL(clicked()), this, SLOT(buttonNClicked()));

    m_buttonNE = new MyQPushButton(this);
    m_buttonNE->setIcon(QIcon("Images/NEButton_nb.png"));
    connect(m_buttonNE, SIGNAL(clicked()), this, SLOT(buttonNEClicked()));

    m_buttonE = new MyQPushButton(this);
    m_buttonE->setIcon(QIcon("Images/EButton_nb.png"));
    connect(m_buttonE, SIGNAL(clicked()), this, SLOT(buttonEClicked()));

    /* Igual per SE, S, SW I W*/

    m_buttonNW = new MyQPushButton(this);
    m_buttonNW->setIcon(QIcon("Images/NWButton_nb.png"));
    connect(m_buttonNW, SIGNAL(clicked()), this, SLOT(buttonNWClicked()));

    m_image = new QLabel();
    m_image->setBackgroundRole(QPalette::Dark);
    m_image->setScaledContents(true);
    QPixmap pix("Images/Compass2_nb.png");
    m_image->setPixmap(pix);

    m_layout->addWidget(m_buttonNW, 0, 0, Qt::AlignRight);
    m_layout->addWidget(m_buttonN, 0, 1, Qt::AlignHCenter);
    m_layout->addWidget(m_buttonNE, 0, 2, Qt::AlignLeft);
    m_layout->addWidget(m_buttonW, 1, 0, Qt::AlignHCenter);
    m_layout->addWidget(m_image, 1, 1, Qt::AlignHCenter);
    m_layout->addWidget(m_buttonE, 1, 2, Qt::AlignHCenter);
    m_layout->addWidget(m_buttonSW, 2, 0, Qt::AlignRight);
    m_layout->addWidget(m_buttonS, 2, 1, Qt::AlignHCenter);
    m_layout->addWidget(m_buttonSE, 2, 2, Qt::AlignLeft);
}
```

- Void setWindDirection(QString dir):

Aquest és el mètode encarregat de modificar la imatge de fons de la brúixola segons la direcció del vent actual, per tal que l'usuari conegui la direcció de vent actual. Així doncs, segons la direcció inserirem una imatge de fons o un altre.

```
void WindChanger::setWindDirection(QString dir) {
    if(dir == QString("N")) {
        m_compassPath = new QString("Images/CompassN_nb.png");
    } else if(dir == QString("NE")) {
        m_compassPath = new QString("Images/CompassNE_nb.png");
    } else if(dir == QString("E")) {
        m_compassPath = new QString("Images/CompassE_nb.png");
    } else if(dir == QString("SE")) {
        m_compassPath = new QString("Images/CompassSE_nb.png");
    } else if(dir == QString("S")) {
        m_compassPath = new QString("Images/CompassS_nb.png");
    } else if(dir == QString("SW")) {
        m_compassPath = new QString("Images/CompassSW_nb.png");
    } else if(dir == QString("W")) {
        m_compassPath = new QString("Images/CompassW_nb.png");
    } else if(dir == QString("NW")) {
        m_compassPath = new QString("Images/CompassNW_nb.png");
    } else {
        QMessageBox::information(this, tr("Error"), "Bad wind direction");
    }

    QPixmap pix(*m_compassPath);
    m_image->setPixmap(pix);
}
```

- QString* getWind();

Aquest és el mètode encarregat de consultar, des de l'exterior de la classe, la direcció del vent seleccionada per l'usuari. Així doncs, aquest mètode directament retornarà l'atribut que guarda la direcció del vent.

```
QString* WindChanger::getWind() {
    return m_wind;
}
```

- Void buttonNClicked():

  Aquest és el mètode encarregat de gestionar el comportament quan es premi el botó N. El funcionament d'aquest mètode es basa en modificar l'atribut que guarda la direcció del vent, actualitzar la imatge de fons segons la nova direcció, emetre el senyal que s'ha modificat la direcció del vent i ocultar el Widget per modificar la direcció del vent.

```
void WindChanger::buttonNClicked() {
    m_wind = new QString("N");
    setWindDirection(*m_wind);
    emit windChanged();
    this->hide();
}
```

  Per la resta de botons seguiran l'esquema esmentat, amb l'única diferencia de la direcció que assignarem. Per aquest motiu ens estalviarem de mostrar el codi per el botó NE, E, SE, S, SW, W i NW.

Fins ara hem vist la classe QtDisplayer que s'encarrega de gestionar la interfície, i els subelements GLWidget i WindChanger. Quedaria per veure la classe MyQPushButton, però simplement és una personalització dels botons per tal de modificar les mides per defecte de forma genèrica. Així doncs, no explicarem aquesta classe, degut a la seva falta d'importancia en general per l'objectiu del projecte.

Apartat 9.3: Proves

En aquest apartat, explicarem les proves que hem realitzat duran la implementació per tal de validar el codi.

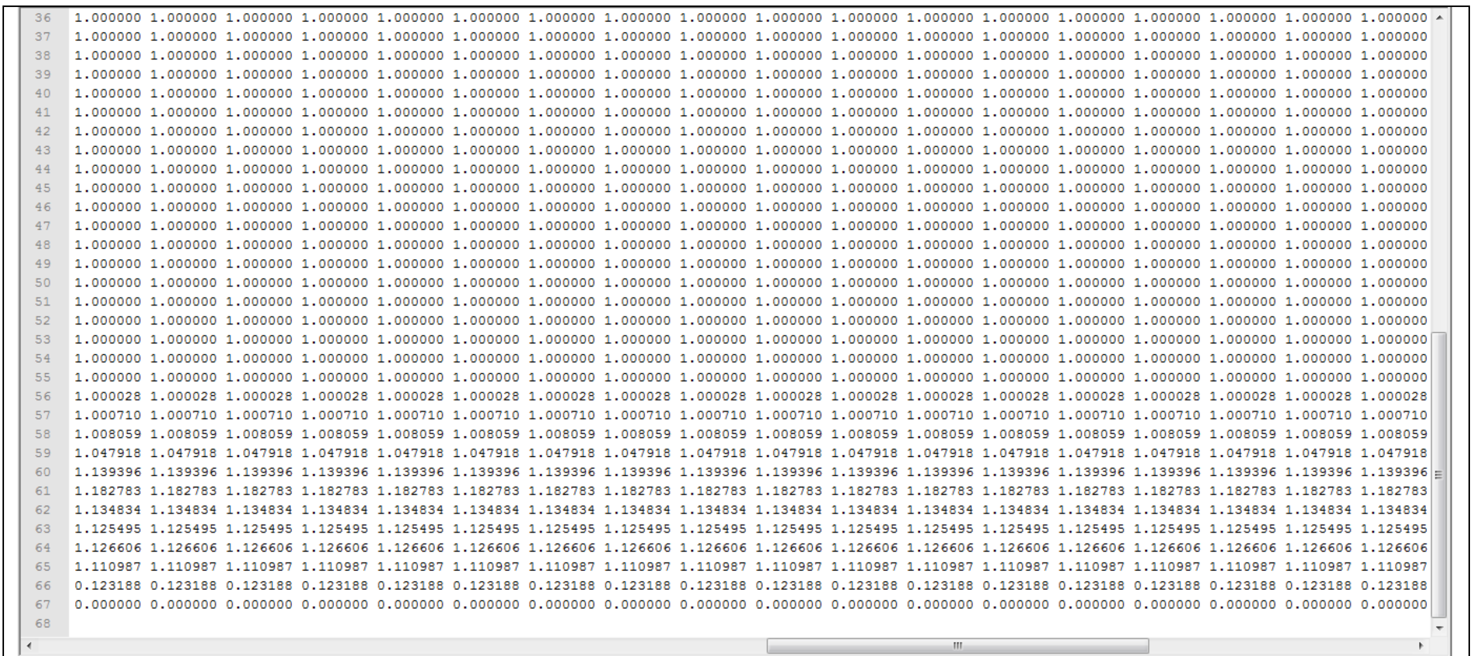
Seguint la metodologia, hem realitzat un procés iteratiu per mòduls en el projecte, és a dir, ens hem centrar en cada pas del projecte a desenvolupar un subconjunt de funcionalitats i un cop ens havíem assegurat la correcta implementació, passàvem a millorar el mòdul o a desenvolupar el seguent. Gracies a això, i també a les eines de debug proporcionades pel Visual Studio, no hem realitzat grans troços de codi per validar la correcta implementació. Tot i això, si que hem fet algunes validacions.

Durant el desenvolupament del mòdul del simulador, a l'estar treballant amb la GPU i tenir un menor control per debugar, vam desenvolupar funcions per escriure fitxers de sortida amb els valors de les densitats, velocitats, forces i tipus de la simulació.

Tot seguit veurem un exemple d'aquestes funcions, el qual genera un fitxer de sortida amb les densitats. Per fer-ho, creem un objecte de tipus FILE, creem un enllaç en mode escriptura per treballar-hi i tot recorrent les densitats, les escrivim en el fitxer.

```
void GLWidget::writeDensities(int iter, int width, int height, float *densities){
    FILE *output;
    char filename[256];

    sprintf_s(filename, "outputD_%04d.txt", iter);

    if( (output = fopen(filename, "w")) == NULL ) {
        printf("Unable to open file %s. Ignoring output.", filename);
        return;
    }

    fprintf(output, "P3\n%d %d\n%d\n", width, height, 255);

    for(int i=0; i<height; i++) {
        for(int j=0; j<width; j++) {
            fprintf(output, "%f ", densities[i*width + j]);
        }
        fprintf(output, "*\n");
    }

    fclose(output);
}
```

Aquest és el resultat d'un dels fitxers de sortida:



```
36 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
37 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
38 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
39 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
40 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
41 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
42 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
43 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
44 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
45 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
46 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
47 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
48 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
49 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
50 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
51 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
52 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
53 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
54 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
55 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
56 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028 1.000028
57 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710 1.000710
58 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059 1.008059
59 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918 1.047918
60 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396 1.139396
61 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783 1.182783
62 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834 1.134834
63 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495 1.125495
64 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606 1.126606
65 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987 1.110987
66 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188 0.123188
67 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
68
```

Figura 9.5: Fitxer de resultats.

Capítol 10: Resultats

En aquest capítol mostrarem el grau d'assoliment dels objectius mitjançant exemples del funcionament del projecte desenvolupat. L'objectiu d'aquest capítol serà el de mostrar que l'aplicació resultant del projecte fa les tasques especificades originalment.

A part, també descriurem la validesa legal de l'aplicació. Per fer-ho mostrarem que aquest compleix la llei orgànica de protecció de dades de caràcter personal (LOPD) i la llei de serveis de la societat de la informació i comerç electrònic (LSSICE).

Apartat 10.1: Objectius de l'aplicació

En aquest apartat recordarem quins eren els objectius de l'aplicació i en mostrarem exemples que demostrin el seu assoliment. Com a objectius, no només ens quedarem amb els inicials, els quals eren massa genèrics, sinó que hi incorporarem també els requeriments funcionals de l'aplicació. D'aquesta manera veurem amb claretat el funcionament.

Realitzar una simulació en temps real mitjançant un llenguatge de programació en paral·lel, com CUDA.

Al inici del projecte, es va decidir que buscar un mètode eficient per tal que la simulació fos en temps real. Tot i les bondads del mètode de Lattice-Boltzmann, es va decidir que la millor opció era la programació en paral·lel per assolir una simulació en temps real.

Amb aquest objectiu en ment, es van estudiar les diverses opcions en llenguatges en paral·lel, tal i com s'ha demostrat en l'apartat 5.2 GPUs, amb la decisió final d'escollir CUDA com a llenguatge de programació en paral·lel.

Finalment, tal i com es pot veure en l'apartat 9.1, s'ha implementat el mòdul WindSimulator de tal manera que la simulació s'executi en paral·lel en els nuclis de la GPU i no en la CPU.

Així doncs, amb els exemples vistos, podem concloure que aquest objectiu s'ha assolit.

Obtenció de la informació bàsica de l'escena.

El primer pas que s'havia de resoldre per poder realitzar la simulació era la manera d'obtenir la informació bàsica de l'escena. Tal i com s'ha explicat, aquesta ha estat la solució escollida:

Partim d'una escena, que en el nostre exemple és tracta d'un model d'un fragment del barri vell de Girona, obtingut pel departament d'IMAE de la UdG.

Figura 10.1: A l'esquerra imatge real de l'escen. A la dreta model del barri vell.

En el següent pas, ja que aquesta primera versió del projecte no ha contemplat la possibilitat de treballar amb simulacions en 3D, haurem d'extreure una imatge del pla zenital de l'escena.

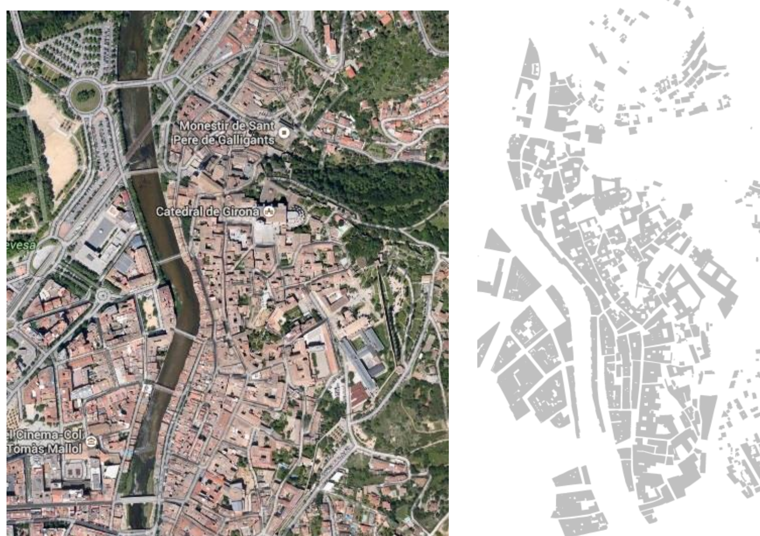


Figura 10.2: Imatge zenital de l'escena.

Un cop tenim la imatge del pla sagital, ja podríem executar l'aplicació i emprar el mòdul ImageReader, explicat en l'apartat l'apartat 8, per obtenir la informació bàsica per la simulació. El problema amb el que ens trobem, és que la memòria disponible a la GPU és limitada, i no podem treballar amb qualsevol imatge.

La memòria total de la targeta gràfica utilitzada és de 2GB. Si tenim en compte que per cada píxel hem de guardar el tipus, la densitat, la velocitat i les vuit forces, això ens determinar un llindar màxim del nombre de píxels que pot tenir la imatge. A més, és necessari definir una mida mínima de píxels pels carrers més estret per tal que el vent i pugui circular. Així doncs, per evitar problemes, utilitzarem una imatge més acotada.

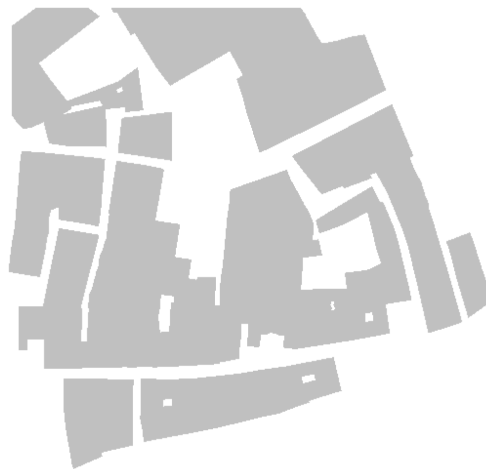


Figura 10.3: Imatge de l'escena acceptable per la GPU.

Un cop dins l'aplicació, i havent pres el botó per obrir una escena.

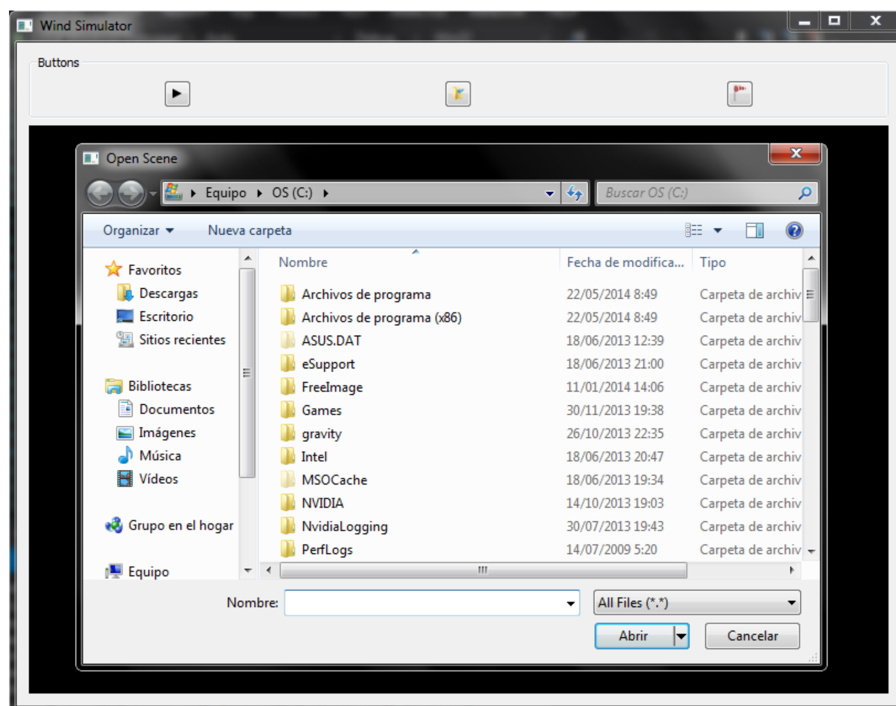


Figura 10.4: Visualització de l'aplicació al obrir una escena.

S'ha de buscar el fitxer de la imatge a obrir i acceptar. El resultat d'aquesta acció serà el de carregar l'escena al simulador.

Figura 10.5: Visualització de l'aplicació un cop l'usuari ha obert una escena.

Així doncs, tal i com hem vist amb aquest exemple, l'objectiu de carregar la informació bàsica d'una escena ha estat assolit.

Implementació del mètode de Lattice-Boltzmann per realitzar la simulació.

El mòdul central del projecte és el simulacor de vent. Per implementar-lo, hem seguit el mètode de Lattice-Boltzmann, explicat en l'apartat 5.1. Com ja hem explicat prèviament, aquest mòdul requereix d'un conjunt del tipus els quals indicaran com està estructurada l'escena sobre la qual es farà la simulació. La simulació s'executara iteració rere iteració, i el resultat de cada iteració serà el conjunt de densitats i velocitats de cada cel·la.

Tot seguit veurem un exemple de l'evolució de la simulació seguint l'escena de l'exemple anterior. Com podrem veure, el vent segueix una direcció determinada, en aquest exemple serà la direcció N.

Figura 10.6: Estat inicial de la simulació.

Figura 10.7: Simulació després d'unes iteracions.

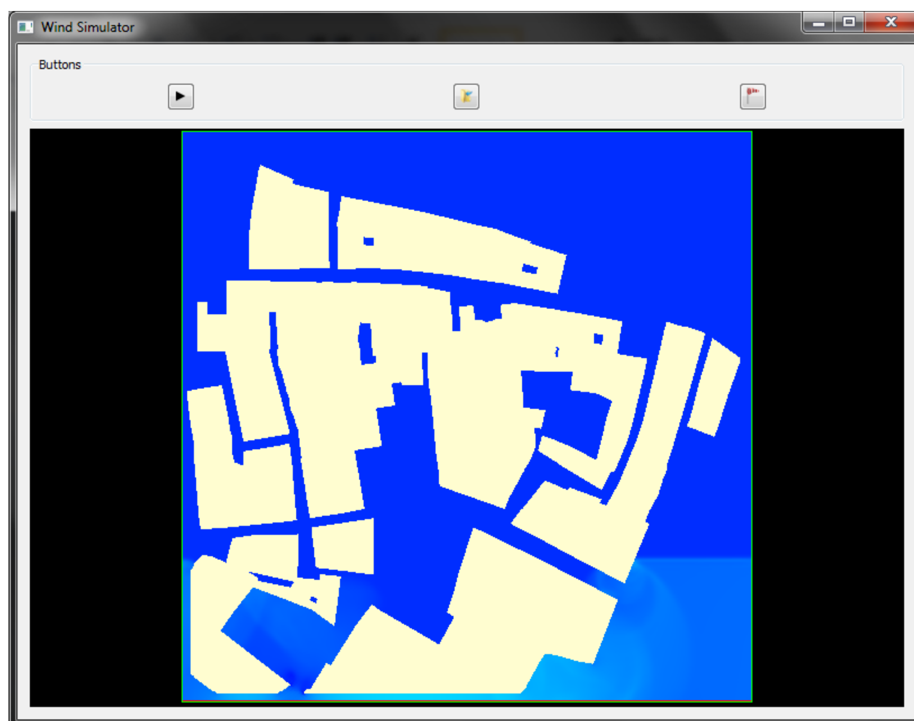


Figura 10.8: Simulació en execució.

Com podem veure en les figures, el vent s'inicia homogèniament des de l'extrem inferior de l'escena, propagant-se per cada cel·la liure que troba i rebotant quan troba un obstacle. Si ens fixem en els extrems lateral, podem observar que no hi ha cap mena de rebot: aquest límits treballen com si el vent s'hi pogués propagar i sortir de l'escena sense influir en la simulació.

Permetrè al usuari modificar la direcció del vent en temps real.

Una de les funcionalitats que havia d'incorporar en el projecte era la de modificar la direcció del vent, per poder tenir un major control de la simulació. La implementació d'aquesta funcionalitat la podem trobar en l'apartat 9.1, al mètode setWindDir().

Tot seguit veurem un exemple gràfic de com podem modificar la direcció del vent i com afecta a la simulació. En primer lloc, tindrem una simulació en execució.

Figura 10.9: Simulació en execució.

Un cop tenim una simulació en execució, s'ha de pressionar el botó de més a l'esquerra, el qual permet a l'usuari modificar la direcció del vent.

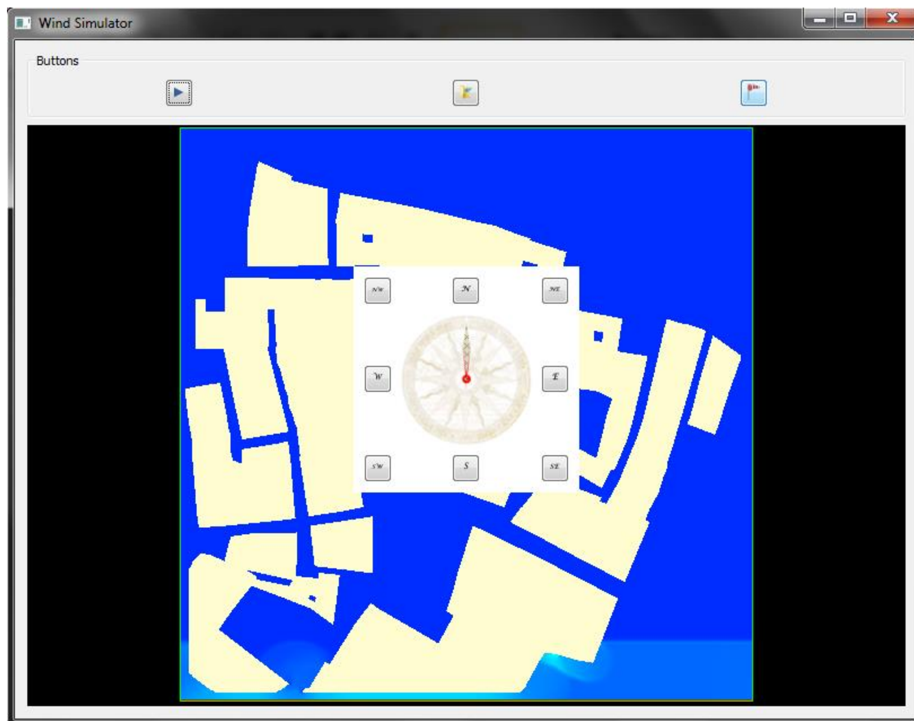


Figura 10.10: Simulació quan l'usuari a pressionat el botó per modificar la direcció del vent.

El següent pas consisteix en decidir quina serà la nova direcció del vent i pressionar el botó corresponent. Un cop pressionat, és modificarà l'estructura de tipus, tal i com es va definir en la pàgina 69 en endavant, per fer que el vent entri en l'escena per la nova direcció indicada. En aquest exemple modificarem la direcció del vent a S, és a dir, de dalt cap a baix.

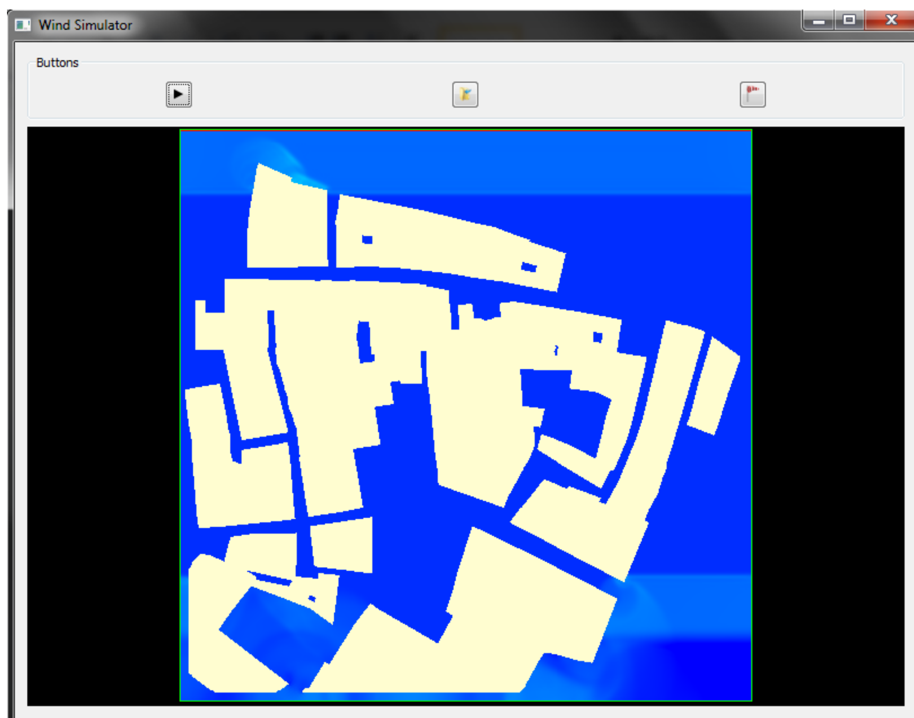


Figura 10.11: Simulació amb la nova direcció del vent (S).

Com podem veure en la figura 10.11, un cop pres el botó les cel·les de la part inferior han deixat de generar partícules de vent, en canvi, les de la part superior ara són les encarregades de generar-ne. Tal i com podem veure, els efectes de les iteracions anteriors perduren en la simulació.

Creació d'una interfície d'usuaria simple i intuïtiva.

L'últim objectiu que ens vam marcar, va ser el de realitzar una interfície d'usuari, la qual havia de ser molt senzilla, però a l'hora intuïtiva. Tal i com s'ha explicat en l'apartat 9.2 i com s'ha pogut veure en els exemples, s'ha desenvolupat una interfície amb tres botons i una finestra on mostrar la simulació.

Des del nostre punt de vista, la interfície compleix amb els requisits, tot i que ens faltaria realitzar alguns testos d'usabilitat.

Apartat 10.2: Validesa legal de l'aplicació.

Un aspecte a tenir en compte quan desenvolupemt una aplicació, és la validesa legal d'aquesta, és a dir, si compleix la legislació actual pel que fa a la llei orgànica de protecció de dades de caràcter personal (LOPD) i la llei de serveis de la societat de la informació i comerç electrònic (LSSICE).

Si parlem de la protecció de dades, l'aplicació no emmagatzema cap tipus d'informació del usuari, motiu pel qual complim lo esmentat per la LOPD. Per altre banda, si ens centrem en el comerç electrònic, la nostre aplicació no inclou cap servei de pagament, motiu pel qual complim lo esmentat per la LSSICE.

Capítol 11: Conclusions

En aquest treball de final de carrera, es va fixar l'objectiu de desenvolupar una aplicació que permetés la simulació de vent en paisatges 3D. Com a objectius bàsics es va decidir:

- Estudi i implementació de les tècniques de *Lattice-Boltzman* per a la simulació de vent.
- Per accelerar la simulació necessitarem estudiar algun llenguatge de programació en paral·lel, com per exemple *CUDA*.
- Estudi i recerca de llibreries per la lectura de les escenes.
- Estudi de llibreries per a la renderització dels resultats de la simulació. Com a exemple podem parlar de llibreries *OpenGL*.
- Estudi i desenvolupament d'interfícies d'usuari per l'aplicació.
- Documentar tot el procés per realitzar el projecte.

A part, durant el disseny d'aquesta, es van decidir un conjunt de requisits que havia de complir l'aplicació. Aquest van ser:

- Obtenció de la informació bàsica de l'escena.
  - Lectura de fitxers d'imatge amb el pla zenital de l'escena.
  - Extracció de la geometria de la imatge.
- Implementació del mètode de Lattice-Boltzmann per realitzar la simulació.
- Permetré modificar la direcció del vent en temps real.
- Visualització dels resultats de la simulació.
- La implementació del mètode de simulació ha de ser el més eficient possible per poder obtenir una simulació en temps real.
- La interacció usuari-aplicació ha de ser el més intuïtiva possible i de fàcil comprensió.

Com s'ha demostrar al llarg d'aquesta memoria, s'han complert tots els objectius i requisits. S'ha desenvolupat un simulador amb la capacitat de tractar escenes 2D i realizar-hi simulacions de vent. La principal problemática amb la que ens hem trobat, ha estat la manca de productes similars de codi lliure en la red. Així doncs, partint de zero, s'ha realitzat una implementació en CUDA del mètode de Lattice-Boltzmann, la qual ens permet simular fluids en temps real. Tot això ha provocat que es rebaixessin les prespectives del projecte i, pel moment, només tractar escenes 2D i no 3D.

Una de les feines més difícils de tot projecte és la seva planificació, ja que només l'experiència permet decidir quant temps requerirà cada tasca. Per aquest motiu, veurem el diagrama de Gantt real del projecte i el compararem amb teòric.

Figura 11.1: Diagrama de Gantt real del projecte.

Si comparem el diagrama de la Figura 11.1 amb el de la figura 4.1, podem veure que, a grans trets, hem complert la planificació. Les principals diferencies estan en que es va allargar el temps dedicat per estudiar i desenvolupar la implementació del mètode de Lattice-Boltzmann, motiu pel qual vam endarrerir la seguent tasca a desenvolupar. Per últim, el temps dedicat a la memòria ha estat molt més elevat en comparació amb el temps teòric calculat.

Capítol 12: Treball futur

En aquest capítol explicarem les possibles millores i canvis en el projecte pel futur. La idea d'aquestes millores és la de dotar a la aplicació de noves funcionalitats i millorar el funcionament i l'eficiencia de les ja assolides.

La primera millora que necessitaria l'aplicació seria la d'augmentar la mida de les escenes amb les que treballar. La idea seria fragmentar l'escena en parts viables per la GPU i simular-les individualment i gestionar l'intercanvi de fluid en les seccions de l'escena.

Figura 12.1: Exemple de escena fragmentada

En segon lloc, ara que tenim una implementació funcional per escenes 2D, s'hauria d'aplicar en escenes 3D. Això implicaria fer canvis en el mòdul de simulació i en el visualitzador i passar d'un model D2Q9 a D3Q27.

Una altre millora interesant, seria la d'afegir una funcionalitat per poder definir regions d'entrada de vent, és a dir, en canvi de fer l'entrada de vent al llarg de tot el límit de l'escena, poder definir regions més petites que puguin simular rafages de vent puntuals.

També es podria incorporar en les escenes obstacles mòbils, és a dir, obstacles que presentessin resistència, però que és puguessin desplaçar, com per exemple cotxes, autobusos i camions.

Capítol 13: Bibliografia

[NCZ14] NVidia CUDA Zone. (2014). CUDA *programming Language – Official Website*. Recuperat 06 juny 2014, des de https://developer.nvidia.com/cuda-zone

[CTD14] CUDA Toolkit Documentation v6.0. (2014). *CUDA documentation*. Recuperat 06 juny 2014, des de http://docs.nvidia.com/cuda/index.html#axzz33rvEqnGb

[FFDSG14] Fast Fluid Dynamics Simulation on the GPU. (2014). *Fluid simulation on GPU theory*. Recuperat 06 juny 2014, des de http://http.developer.nvidia.com/GPUGems/gpugems_ch38.html

[FI14] FreeImage. (2014). *FreeImage – Official Website*. Recuperat 06 juny 2014, des de http://freeimage.sourceforge.net/

[OGL14] OpenGL. (2014). *OpenGL – Official Website*. Recuperat 06 juny 2014, des de http://www.opengl.org/

[OADO14] OpenGL API Documentation Overview. (2014). *OpenGL documentation*. Recuperat 06 juny 2014, des de http://www.opengl.org/documentation/

[QP14] Qt Project. (2014). *Qt – Official Website*. Recuperat 06 juny 2014, des de http://qt-project.org/

[QPD14] Qt Project Documentation. (2014). *Qt documentation*. Recuperat 06 juny 2014, des de http://qt-project.org/doc/

[LBMW14] Lattice Boltzmann methods Wikipedia. (2014). *Article Wikipedia de Lattice Boltzmann methods*. Recuperat 06 juny 2014, des de http://en.wikipedia.org/wiki/Lattice_Boltzmann_methods

[SO14] StackOverflow. (2014).StackOverflow. Recuperat 06 juny 2014, des de http://stackoverflow.com/

[HL97] He, X. and Luo, L.-S (1997). *Lattice Boltzmann model for the incompressible navierstokes equation*. J. Stat. Phys., 88(972).

[JM08] Jordi Marco (2008). *Accurate Water Simulation fo Visibility in Driving*. Master Thesis.

Capítol 14: Annexos

Annex A: Conceptes bàsics de C++

Estructura bàsica

Tot programa en C + + ha de tenir la funció *main*() (llevat que s'especifiqui en temps de compilació un altre punt d'entrada, que en realitat és la funció que té el *main*())

```
int main ()

{}
```

La funció main ha de tenir un dels següents prototips:

- *int main ()*
- *int main (int argc, char ** argv)*
- *int main (int argc, char ** argv, char ** env)*

La primera és la forma per omissió d'un programa que no rep paràmetres ni arguments. La segona forma té dos paràmetres: *argc*, un nombre que descriu el nombre d'arguments del programa (incloent-hi el nom del programa mateix), i *argv*, un punter a una matriu de punters, de *argc* elements, on l'element *argv[i]* representa l'últim argument lliurat al programa. En el tercer cas s'afegeix la possibilitat de poder accedir a les variables d'entorn d'execució de la mateixa manera que s'accedeix als arguments del programa, però reflectits sobre la variable *env*.

El tipus de retorn de *main* és int. En acabar la funció *main*, s'ha d'incloure el valor de retorn (per exemple, *return* 0;, encara que l'estàndard preveu només dos possibles valors de retorn: *EXIT_SUCCESS* i *EXIT_ERROR*, definides en l'arxiu *cstddef*), o sortir per mitjà de la funció *exit*. Alternativament es pot deixar en blanc, en aquest cas el compilador és responsable d'afegir la sortida adequada.

Concepte classe

Els objectes en C++ són abstrets mitjançant una classe. Segons el paradigma de la programació orientada a objectes un objecte consta de:

- Mètodes o funcions
- Atributs o Variables Membre

Un exemple de classe que podem explicar com a exemple és la classe gos. Cada gos comparteix unes característiques (atributs). El seu nombre de potes, el color del seu pelatge

o la seva mida són alguns dels seus atributs. Les funcions que ho facin bordar, canviar el seu comportament... aquestes són les funcions de la classe.

De mateixa forma que en el C, la definició d'una classe és realitza en un fitxer .h on tindrem els mètodes, tant públic com privats, i els atributs de la classe. D'altre banda, tindrem un fitxer .cpp on implementarem els mètodes de la classe.

Herència

Existeixen dos tipus d'herència entre classes en el llenguatge de programació C + +. Aquestes són:

- Herència simple (un sol pare)
- Herència múltiple (més d'un pare)

Sobrecarrega d'operadors

La sobrecàrrega d'operadors és una forma de fer polimorfisme. És possible definir el comportament d'un operador del llenguatge perquè treballi amb tipus de dades definits per l'usuari. No tots els operadors de C++ són factibles de sobrecarregar, i, entre aquells que poden ser sobrecarregats, s'han de complir condicions especials. En particular, els operadors *sizeof i::* no són sobrecàrregues.

No és possible en C++ crear un operador nou.

Els comportaments dels operadors sobrecarregats s'implementen de la mateixa manera que una funció, llevat que aquesta tindrà un nom especial: Tipus de dades de tornada *operator*<*token* del operador> (paràmetres)

Els següents operadors poden ser sobrecarregats

<table>
<tr><th>Operadors unaris</th><th>Operadors Binaris</th><th>Operadores d'Assignació</th></tr>
<tr><td>• Operador * (d'indirecció)
• Operador -
> (d'indirecció)
• Operador +
• Operador -
• Operador ++
• Operador –</td><td>• Operador ==
• Operador +
• Operador -
• Operador *
• Operador /
• Operador %
• Operador <<
• Operador >>
• Operador &
• Operador ^
• Operador |
• Operador []
• Operador ()</td><td>• Operador =
• Operador +=
• Operador -=
• Operador *=
• Operador /=
• Operador %=
• Operador <<=
• Operador >>=
• Operador &=
• Operador ^=
• Operador |=</td></tr>
</table>

Atès que aquests operadors són definits per un tipus de dades definit per l'usuari, aquest és lliure d'assignar qualsevol semàntica que desitgi. No obstant això, es considera de primera importància que les semàntiques siguin tan semblants al comportament natural dels operadors com perquè l'ús dels operadors sobrecarregats sigui intuïtiu. Per exemple, l'ús de l'operador unari - hauria canviar el "signe" d'un "valor".

Els operadors sobrecarregats no deixen de ser funcions, pel que poden retornar un valor, si aquest valor és del tipus de dades amb què treballa l'operador, permet l'encadenament de sentències. Per exemple, si tenim 3 variables A, B i C d'un tipus T i sobrecarreguem l'operador = perquè treballi amb el tipus de dades T, hi ha dues opcions: si l'operador no torna res una sentència com "A = B = C, "(sense les cometes) donaria error, però si es torna un tipus de dades T a l'implementar l'operador, permetria concatenar tots els elements es volguessin, permetent alguna cosa com" A = B = C = D =...;"

Capítol 15: Manual d'usuari i/o instal·lació

En aquest capítol, especificarem el funcionament de l'aplicació i com utilitzar-la. L'aplicació serà un executable, per la qual cosa no caldrà cap mena d'instal·lació prèvia, per contra, es requerirà d'una targeta gràfica amb capacitat per executar codi CUDA i actualitzar els drivers necessaris per treballar amb CUDA.

En primer lloc, veurem la pantalla inicial de l'aplicació i s'explicaran les funcionalitats bàsiques que es poden utilitzar.

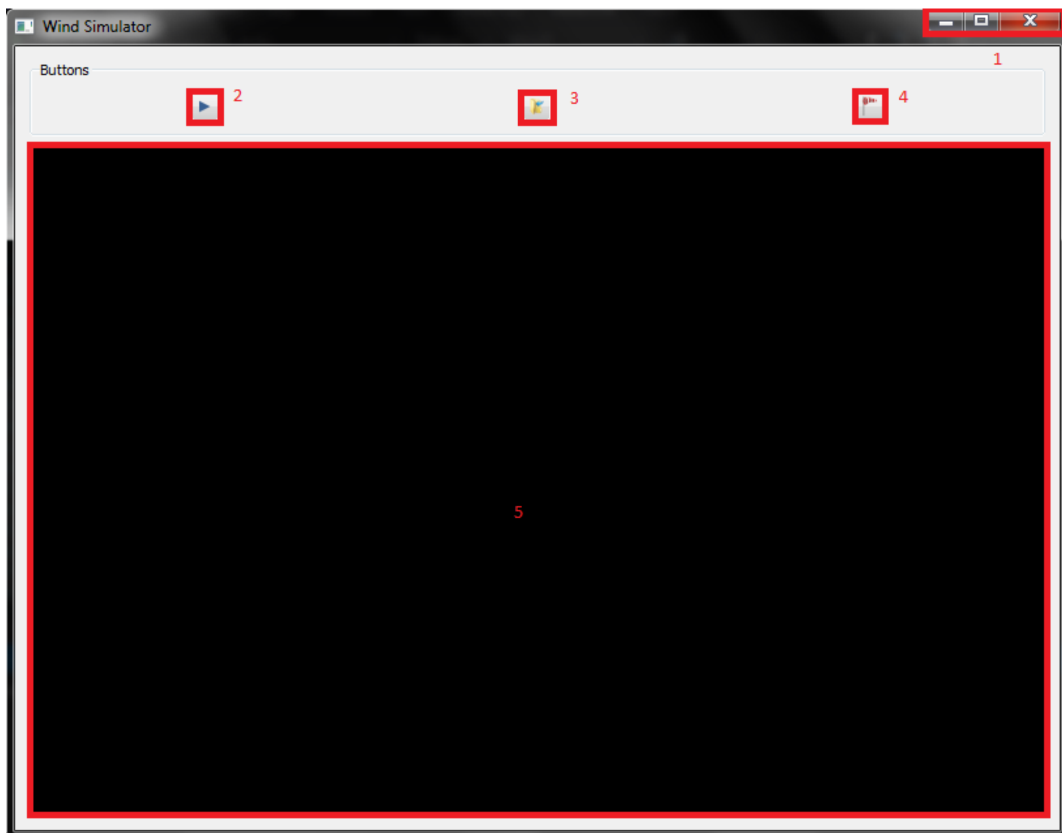


Figura 15.1: Pantalla d'inici de l'aplicació.

Com es pot veure n la figura 15.1, hi ha 5 elements bàsics:

1. Botons per gestionar el comportament bàsic d'una finestra, els quals són minimitzar, maximitzar o tencar la finestra.
2. Botó per iniciar la simulació o aturar-la. Cal tenir en compte que si no hi ha cap escena carregada, es mostrarà un missatge d'error.
3. Botó per realitzar la lectura d'una escena. Aquest obrirà una finestra emergent per seleccionar un fitxer del sistema.
4. Botó per modificar la direcció del vent. Aquest obrirà una finestra emergent per poder seleccionar la nova direcció del vent.
5. Visualitzador de l'estat de la simulació. En aquest element, podrem veure l'escena un cop carregada i com evoluciona duran la simulació.

Tot seguit veurem la seqüència lògica per fer funcionar l'aplicació. El primer pas serà carregar una escena.

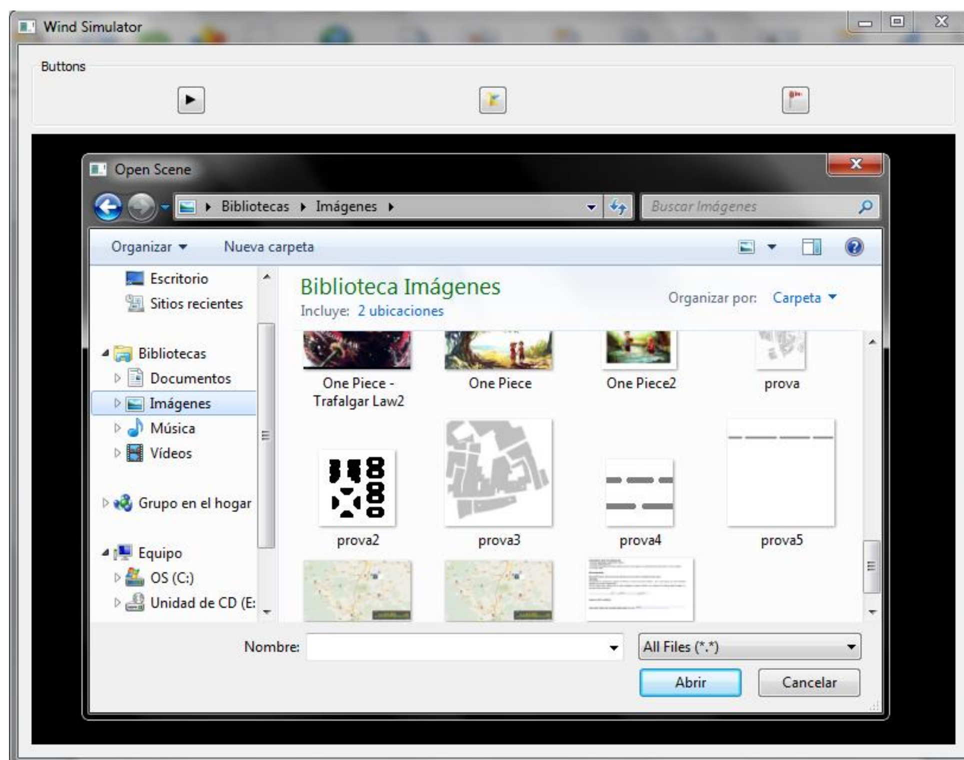


Figura 15.2: Carregant una escena al simulador.

Un cop seleccionada l’escena, aquesta es mostrarà en el visualitzador.

Figura 15.3: Escena carregada al simulador.

Ara que ja tenim l’escena carregada, podem iniciar la simulació o definir prèviament la direcció que tindrà el vent. Per defecte sempre tindrem un vent direcció N. Així doncs, primer seleccionarem una nova direcció del vent, tot pressionant el botó corresponent.

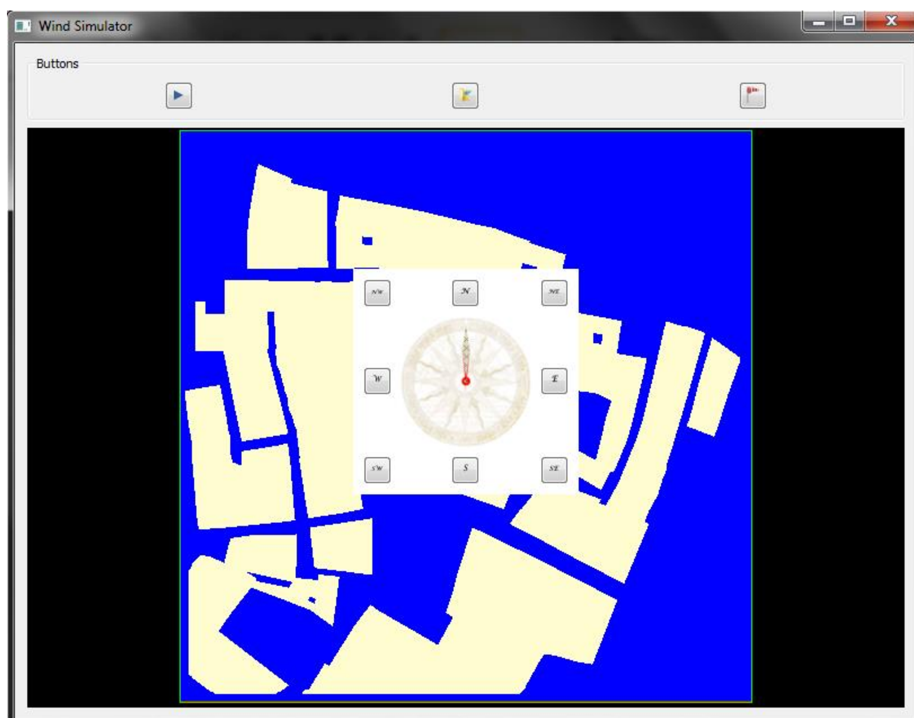


Figura 15.4: Modificant la direcció del vent.

El següent pas serà el de seleccionar una de les direccions disponibles. Un cop seleccionada, s'ocultarà la finestra de selecció de la direcció del vent i s'actualitzarà l'escena. Finalment, ja podem iniciar la simulació.

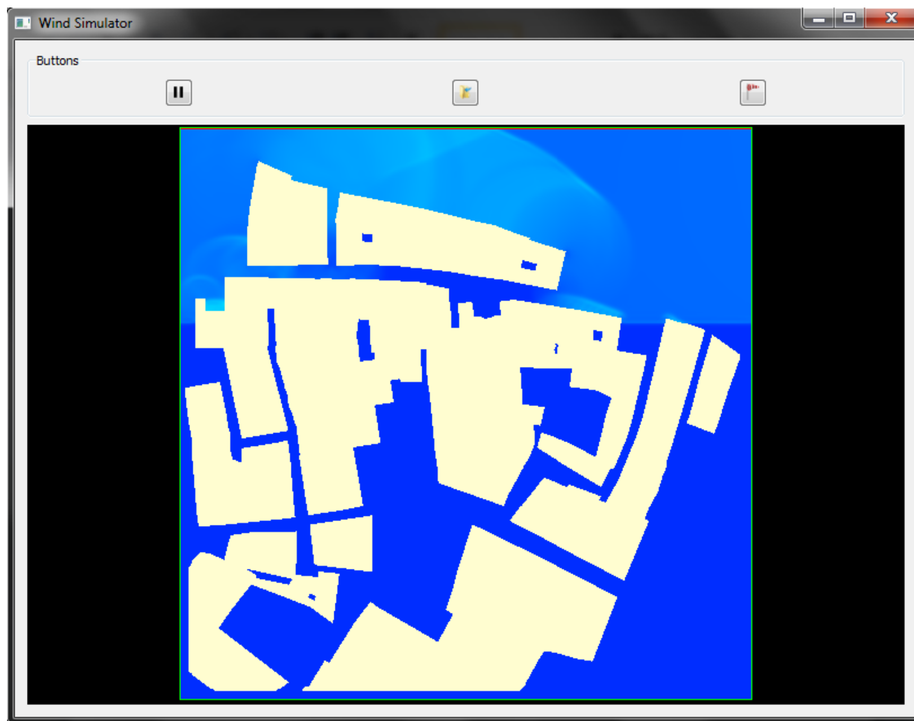


Figura 15.5: Simulació activa.

Com ja hem explicat prèviament, si no tinguéssim cap escena carregada i vulguessim iniciar una simulació, obtindríem el següent resultat.

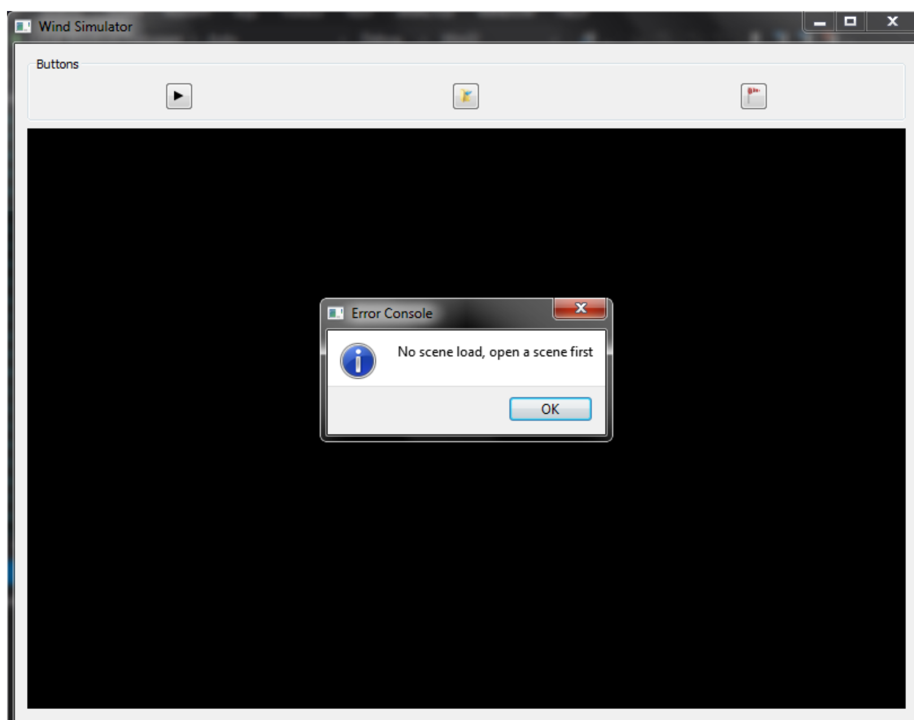


Figura 15.6: Intent de simulació sense cap escena carregada.