UNIVERSIDADE FEDERAL FLUMINENSE

MARK EIRIK SCORTEGAGNA JOSELLI

NGrid
A New Data Structure for the Neighbourhood Gathering Problem on GPUs

Doctoral Thesis

NITEROI, 2013

UNIVERSIDADE FEDERAL FLUMINENSE

MARK EIRIK SCORTEGAGNA JOSELLI

NGrid
Uma Nova Estrutura de Dados para o Problema de Coleta de Vizinhos com uso de GPUs

NITERÓI

2013

UNIVERSIDADE FEDERAL FLUMINENSE

MARK EIRIK SCORTEGAGNA JOSELLI

NGrid
A New Data Structure for the Neighbourhood Gathering Problem on GPUs

Doctoral Thesis, submited to the Posgraduation in Computation program of the Universidade Federal Fluminense.

Orientador:
Esteban Walter Gonzalez Clua

Doctoral Thesis

NITEROI, 2013

UNIVERSIDADE FEDERAL FLUMINENSE

MARK EIRIK SCORTEGAGNA JOSELLI

NGrid
Uma Nova Estrutura de Dados para o Problema de Coleta de Vizinhos com uso de GPUs

Tese de Doutorado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Doutor em Computação.

Orientador:
Esteban Walter Gonzalez Clua

NITERÓI

2013

Doctoral Thesis

Mark Eirik Scortegagna Joselli

Doctoral Thesis, submited to the Posgraduation in Computation program of the Universidade Federal Fluminense.

Approved by:

Esteban Walter Gonzalez Clua/ IC-UFF (Chair)

Wagner Meira Junior / UFMG

Ricardo Farias / UFRJ

Anselmo Antunes Montenegro / IC-UFF

Maria Cristina Silva Boeres / IC-UFF

Niteroi, 15 de maio de 2013.

Acknowledgments

This thesis would not have been possible without the support of many people. The author wishes to express his love and gratitude to his beloved families; for their understanding and endless love.

The author wishes to express his gratitude to his advisor, Esteban Clua who was always helpful and offered invaluable assistance, support and guidance.

Deepest gratitude are also due to the members of the MediaLab, who had always given good advise and friendship.

A special thanks for FAPERJ and CNPQ for their financial support.

Abstract

Neighbourhood gathering methods are commonly required in many real-time simulation scenarios, such as: crowd simulation and game AI for simulating the vision of the entities; SPH fluid simulation for calculate the forces,density and viscosity interactions; and particle systems and physics simulation as the broad phase of the collision detection algorithm. In a naive implementation, it has a complexity of $O(n^2)$ required computations of the neighbourhood gathering algorithm, necessary for the proximity queries of all pair of entities in order to compute the relevant mutual interactions. In order to solve this problem, many works propose spatial data structures that subdivide the environment and classify the entities among the cells based on their position. Although this strategy minimizes the number of proximity queries to be treated, it is not efficient when a large number of particles are grouped in the same cell. This thesis proposes a novel and efficient data structure that maintains the entities into another paradigm of proximity data structure, called NGrid, where each cell contains only one entity and does not directly represent a discrete spatial subdivision. The NGrid proposal fills the lack of GPU bounded architectures, which usually processes all elements in the scene at each frame. Results shows that the use of the NGrid in different scenarios leads to a performance gain when compared to traditional data structure.

Keywords

1. Neighbourhood Gathering
2. Real-time Simulation
3. Games
4. Fluid Simulation
5. Particle System
6. Crowd Simulation
7. GPGPU
8. Parallel Computing
9. Data Structure

Resumo

Comumente métodos de selecionar a vizinhança são obrigatórios em muitos cenários Métodos de seleção de vizinhança são tradicionalmento utilizados em cenários de simulações em tempo real. Por exemplo, em problemas de simulação de multidões, visão de entidades em IA de games, simulação de fluidos com o método SPH e detecção de colisões em física de games. Em geral, as implementações desse método possuem complexidade da ordem de $O(n^2)$. Esta complexidade é associada as operações necessárias para consultar a proximidade de todos os pares das entidades, considerando as interações mútuas relevantes. Inúmeros trabalhos propõem a utilização de estruturas de dados espaciais para reduzir essa complexidade. Tais estruturas subdividem o ambiente, classificando as entidades entre as células com base em sua posição. Apesar dessa estratégia minimizar o número de consultas de proximidade a ser tratada, ela não é eficiente quando um grande número de partículas estão agrupados em uma mesma célula. Neste sentido, a presente tese propõe uma nova e eficiente estrutura de dados, na qual as entidades são mantidas em um paradigma baseado em proximidade. Em tal estrutura, denominada NGrid, cada célula contém apenas uma entidade e não representa diretamente a subdivisão do espaço discreto. A proposta da NGrid preenche a falta de estrutura de dados específicas para arquiteturas do tipo GPU, onde geralmente são processados todos os elementos da cena em cada quadro de renderização. Resultados obtidos demonstram a viabilidade da utilização da NGrid em diferentes cenários, com significativo ganho de desempenho em relação a estruturas de dados tradicionais.

Palavras-Chave

1. Coleta de vizinhança
2. Simulação em tempo real
3. Jogos Digitais
4. Simulação de Fluidos
5. Sistema de particulas
6. Simulação de multidões
7. GPGPU
8. Computação Paralela
9. Estrutura de dados

Glossary

AABB	:	axis-aligned bounding box
API	:	application programming interface
Boid	:	bird-like object
BSP	:	Binary space partitioning
BVH	:	Boundery Volume Hierarchy
CUDA	:	Compute Unified Device Architecture
DSP	:	Digital signal processors
GPGPU	:	General Processing on GPUs
GPU	:	Unidade de processamento gráfico
FPS	:	Frames por segundo
OBB	:	oriented bounding box
OpenCL	:	Open Computing Language
SaP	:	Sweep and Prune
SPH	:	Smoothed-Particle Hydrodynamics
UFF	:	Universidade Federal Fluminense

Contents

List of Figures **xv**

List of Tables **xvii**

List of Algorithms **18**

Introdução em Português **1**

Chapter 1

Introduction **4**

1.1 Contributions 6

1.2 Publications 6

1.3 Organization of this thesis 8

Chapter 2

The Neighborhood Gathering Problem **10**

2.1 Background 11

2.1.1 Real-Time simulation 11

2.1.2 GPU Computing 12

2.2 The Neighborhood Gathering Problem 15

2.3 The Neighbourhood Gathering for Real Time Simulations 17

2.3.1 Particle System 17

2.3.2 Crowd Simulation 20

2.3.3 SPH Fluid Simulation 22

2.3.4 Physics Engines 23
2.3.5 Digital Games 25
2.4 Summary 26

Chapter 3
Solutions for the Neighborhood Gathering Problem in the Literature 27
3.1 Sweep and Prune 27
3.1.1 Algorithm Details 27
3.2 Uniform Grid 29
3.2.1 Algorithm Details 30
3.3 Hierarchical Trees 32
3.3.1 Algorithm Details 32
3.4 Summary 34

Chapter 4
The NGrid 35
4.1 NGrid: a Proximity Data Structure 35
4.2 Properties of the NGrid 39
4.3 Determining the N-dimensions 41
4.4 Maintenance of the NGrid: Sorting Stage 43
4.4.1 Partial Odd-Even Sorting 44
4.4.2 Bitonic Sort 45
4.5 Gathering of the NGrid 47
4.5.1 Fixed N-radius 47
4.5.2 Adaptive N-radius 48
4.6 Analysis of the NGrid 49
4.7 Summary 50

Chapter 5

NGrid on a GPU Fluid Animation **51**

5.1 Architecture environment 51

5.1.1 Execution workflow 52

5.1.2 Neighborhood Gathering 52

5.1.3 Data configuration 52

5.1.4 Density Processing 53

5.1.5 Force Processing 53

5.1.6 Integration 54

5.2 Results 55

5.2.1 Simulation Configuration 55

5.2.2 Test Results 56

5.2.3 Comparison to other works in the Literature 59

5.3 Summary 61

Chapter 6

NGrid on a Multi-GPU Crowd Simulation **62**

6.1 Multiples GPUs architecture 63

6.2 Results 63

6.2.1 Simulation using a single GPU 65

6.2.2 Simulation with Multi-GPU 67

6.3 Summary 68

Chapter 7

NGrid on a Mobile GPU Particle System **69**

7.1 The Renderscript API 70

7.2 The Particle System 71

7.3 Architecture 72

7.4 Results 74

7.5 Summary 75

Chapter 8

NGrid on a GPU Game **76**

8.1 GpuWars Game Design 76

8.2 The Architecture 77

8.3 Physics Step 80

8.3.1 The narrow phase of the collision detection 82

8.3.2 The Integrator 82

8.4 AI Step 82

8.4.1 The GpuWars Game AI 83

8.4.1.1 Kamikaze Behavior 83

8.4.1.2 Group Behavior 84

8.4.1.3 Tricky Behavior 84

8.5 Results 85

8.6 Summary 86

Chapter 9

Conclusions **87**

9.1 Future Work 89

References **90**

Conteúdo

Lista de Figuras **x**

Lista de Tabelas **xii**

Introdução em Português **1**

Chapter 1

Introdução **4**

1.1 Contribuições 6

1.2 Publicações 6

1.3 Organização dessa Tese 8

Chapter 2

O Problema de Coleta de Vizinhança **10**

2.1 Conceitos Relacionados 11

2.1.1 Simulação em Tempo Real 11

2.1.2 GPGPU 12

2.2 O Problema da Coleta de Vizinhança 15

2.3 A Coleta de Vizinhança em Simulações em Tempo Real 17

2.3.1 Sistema de Párticulas 17

2.3.2 Simulação de Multidões 20

2.3.3 Simulação de Fluidos com SPH 22

2.3.4 Motores de Física 23

2.3.5 Jogos Digitais 25

2.4 Sumário 26

Chapter 3

Soluções na Literatura do Problema de Coleta de Vizinhança 27

3.1 Sap 27

3.1.1 Detalhes do Algoritimo 27

3.2 Grade Uniforme 29

3.2.1 Detalhes do Algoritimo 30

3.3 Arvores Hierarquicas 32

3.3.1 Detalhes do Algoritmo 32

3.4 Sumario 34

Chapter 4

NGrid 35

4.1 NGrid: Uma Estrutura de Dados Aproximada 35

4.2 Propriedades da NGrid 39

4.3 Determinando as N-dimensions 41

4.4 Manutenção da NGrid: Ordenação 43

4.4.1 Ordenação Parcial par-impar 44

4.4.2 Ordenação Bitonica 45

4.5 Coleta na NGrid 47

4.5.1 N-radius Fixo 47

4.5.2 N-radius Adaptativo 48

4.6 Analise da NGrid 49

4.7 Sumario 50

Chapter 5

NGrid em uma Animação de Fluidos 51

5.1 Arquitetura 51

5.1.1 Fluxo de Execução 52
5.1.2 Coleta de Vizinhançã 52
5.1.3 Configuração dos Dados 52
5.1.4 Processamento da Densidade 53
5.1.5 Processamento da Força 53
5.1.6 Integração 54
5.2 Resultados 55
5.2.1 Configuração da Simulação 55
5.2.2 Resultados de Testes 56
5.2.3 Comparação com outros Trabalhos da Literatura 59
5.3 Sumario 61

Chapter 6
NGrid em uma Simulação de Multidão em Multi-GPU **62**
6.1 Arquiteturas de Multiplas GPUs 63
6.2 Resultados 63
6.2.1 Simulaçã0 com uma GPU 65
6.2.2 Simulaçã0 com Multi-GPU 67
6.3 Sumario 68

Chapter 7
NGrid no Sistema de Partículas em uma Mobile GPU **69**
7.1 A API Renderscript 70
7.2 Sistema de Partícula 71
7.3 Arquitetura 72
7.4 Resultados 74
7.5 Sumario 75

Chapter 8

NGrid em um Game GPU **76**

8.1 GpuWars Game Design 76

8.2 A Arquitetura 77

8.3 Fisica 80

8.3.1 Fase fina da colisão 82

8.3.2 Integração 82

8.4 IA 82

8.4.1 A IA do GpuWars 83

8.4.1.1 Comportamento Kamikaze 83

8.4.1.2 Comportamento Group 84

8.4.1.3 Comportamento Tricky 84

8.5 Resultados 85

8.6 Sumario 86

Chapter 9

Conclusões **87**

9.1 Trabalhos Futuros 89

Referências **90**

List of Figures

2.1 Workflow of a real-time simulation. 12

2.2 Mobile Hardware Architecture [1]. 14

2.3 Regular grid for searching for neighboring particles. 16

2.4 Two particles near each other. 19

2.5 Workflow of a particle system 20

2.6 Flocking Boids Rules. 21

2.7 The entity and its vision. 21

2.8 Bounding Sphere. 25

3.1 Workflow of the sweep and prune algorithm. 28

3.2 The SaP method illustrated. 28

3.3 Two uniform grids with different cell sizes. 30

3.4 The uniform grid world subdivision. 31

3.5 The environment subdivided in a quadtree structure. 33

3.6 The entities organised in the tree structure. 33

4.1 An example of a distribution of entities in the NGrid. Small circles illustrate entities that are further away from the viewpoint. 36

4.2 The adaptation of the NGrid cells over a set on entities. 37

4.3 Example of the NGrid with N-radius of 1. 37

4.4 Example of the organization of a set of particles in the NGrid and the uniform grid. 38

4.5 Workflow of the NGrid. 39

4.6 Different NGrid dimensions for the same set of entities. 42

4.7 The sorting of a NGrid. 43

4.8 Partial sorting pass with 6 odd-even transposition steps. 46

4.9 Example of the gathering of a NGrid 4x4 and fixed N-radius of 1 48

4.10 Example of the Structure of the Extended Moore Neighborhood with 16 particles and adaptive N-radius 49

5.1 Fluid's data stored at GPU's global memory. 53

5.2 Screenshots of the simulation. 55

5.3 Use of Memory in the Simulation in MB. 58

5.4 Evolution of the Simulation in FPS. 59

5.5 Error during the simulation of 65k particles. 60

6.1 Simulation with 32K boids. 64

6.2 Evolution of the performance of the simulation in a log scale. 66

6.3 Error Evolution of 65k boids simulation. 67

6.4 Evolution of the simulation with one and two Gpus. 68

7.1 Architecture Overview. 73

7.2 Screenshot of the simulation with 1024 particles on a Android Tablet. . . . 74

8.1 Single-Thread Game Loop with a GPGPU Stage. 77

8.2 Game Loop of Architecture 78

8.3 The Different Process of the Architecture Threads. 81

8.4 The Kamikaze State Machine. 83

8.5 The Group State Machine. 84

8.6 The Tricky State Machine. 85

8.7 A Screenshot of the game. 85

8.8 Performance of the game 86

List of Tables

5.1 Scalability of the Simulation when using the Uniform Grid with Index Sort and the Neighborhood Grid. 57

5.2 Time in milliseconds spent with Tasks by each Gathering Method. M stands for maintenance and S stands for simulation 57

5.3 Usage of memory when using the uniform grid with index sort and the NGrid. 58

5.4 Erro of the NGrid in comparison with the uniform grid with different sort mechanisms. 59

6.1 Scalability of the Simulation when using the Uniform Grid with Index Sort and the Neighborhood Grid with Bitonic Sort. 65

6.2 Error of the NGrid with different sort mechanisms. 66

6.3 Scalability of the Simulation 68

7.1 Scalability of the Simulation. 75

List of Algorithms

1 Brute Force Neighborhood Gathering Algorithm 17

2 Algorithm to find the dimension of the NGrid 42

3 Odd even sort for a NGrid 45

4 The fixed N-radius gathering algorithm 48

5 The adaptive N-radius gathering algorithm 49

6 Particle's density processing. 53

7 Particle's forces processing. 54

8 Integration of the Particles. 55

9 Algorithm describing the high level steps during the simulation for various GPUs. 64

Introdução em Português

Jogos digitais e simulações gráficas normalmente gastam a maior parte dos recursos computacionais com renderização e visualização. Comumente, sobram poucos recursos computacionais para processar as tarefas complexas de comportamento das entidades na cena, requeridas normalmente pela física e/ou pela inteligência artificial. Este fato faz com que o comportamento e o número de objetos interativos venham a ser normalmente limitado. Além disso, a fim de processar um elevado número de entidades interativas, numa cena em tempo real, é necessário desenvolver uma solução para este problema específico. Isso requer estratégias de coleta de vizinhançca para a interação entre eles, o que afeta o desempenho do aplicativo.

A coleta de vizinhançca é o processo de coleta das entidades mais próximas uma determinada entidade n, e normalmente é feito para todas as entidade. Usualmente, para uma simulação, é necessário realizar este cálculo, entre um grande conjunto de entidades, exigindo um alto poder de processamento. A coleta de vizinhança é aplicada em muitos tipos de simulação, com diferentes finalidades. Para simular multidões e games, a coleta é normalmente usada para reunir as entidades mais próximas, a fim de simular a visão da entidade. Num cenário de fluido SPH, cada partícula tem de coletar as partículas mais próximas a fim de calcular a densidade, a viscosidade e as forças da partícula. Em um sistema de partículas, o algoritmo de detecção de colisão precisa das partículas mais próximos, a fim de verificar se existem colisões entre elas.

A abordagem direta de tal algoritmo tem complexidade $\mathrm{O}(n^2)$, uma vez que é necessário processar cada entidade contra todas as outras entidades. Muitos trabalhos, que necessitam de resultados em tempo real, tentam otimizar essa coleta, evitando a alta complexidade de consultas de proximidade através da aplicação de alguma forma de subdivisão espacial ou hierárquica no meio ambiente e classificando entidades em células com base na sua posição. Para acelerar essa busca dos dados em um ambiente paralelo (como as GPUs), a lista de entidades devem ser classificados de um modo que todas as entidades que estão nas mesmas células estejam proximas. Esta abordagem auxilia a reduzir o número de consultas de proximidade, mas é muito sensível ao número máximo de entidades que

podem caber numa única célula. A fim de proporcionar uma estrutura de dados para simulações em tempo real, este trabalho apresenta NGrid, uma nova estrutura de dados para o problema da coleta de vizinhança, adequado para ambientes de alto processamento paralelo, tal como as GPUs

Em aplicações em tempo real, muitos algoritmos tradicionalmente executadas na CPU muitas vezes são adequados para execução em paralelo, o que os torna adequados a serem implementadas na GPU. Existem várias áreas de pesquisa que usam a computação GPU, a fim de otimizar seu processamento, como a previsão do tempo [2], química [3] e algoritmos de inteligência artificial [4]. No entanto, as primeiras aplicações de GPGPU tinham que realizar a adaptação de shaders gráficos APIs, levando a uma curva de aprendizado difícil e codigos que, por vezes, não eram muito eficientes para as soluções propostas [5]. CUDA [**?**] e OpenCL [**?**] tecnologias como objectivo proporcionar uma nova camada de abstração sobre o hardware gráfico para facilitar a sua utilização para o processamento não-gráficos. Simulação em tempo real que explora este modelo de programação na GPU é uma linha de pesquisa promissora, uma vez que pode speedup a solução de computação de alto custo da simulação do comportamento.

Esta tese propõe uma nova estrutura de dados, chamada de NGrid, que otimiza e faz uso da proximidade de dados na memória. Enquanto o NGrid é projetada principalmente para o uso com os processadores manycores, como as GPUs, outros tipos de hardware paralelo, tais como processadores de múltiplos núcleos de CPU, podem tirar vantagem dela.

A NGrid é uma estrutura topológica discreta que mantém todas as partículas classificadas de tal maneira que as partículas, que estão próximos umas das outras de acordo com a métrica Euclidiana, são inseridas nas posições vizinhas do NGrid. O NGrid não tem células vazias, porque uma partícula sempre terá um vizinho, mesmo quando ele estiver muito longe e não influenciar o comportamento da outra entidade. Assim, o seu tamanho coincide com o número de partículas. Com isso, a estrutura de dados é muito adequado para o processamento paralelo, evitando conflitos de banco e aproveitando a coalescncia de dados, maximizando o desempenho de leitura de memória com base na localidade.

Contribuições

Esta tese apresenta o NGrid: uma nova estrutura de dados, adequada para processamento paralelo, elaborada para a simulação em tempo real que requerem um algoritmo de coleta de vizinhança. Esta estrutura é capaz de processar um grande número de entidades, sendo mais rápido do que as abordagens tradicionais, especialmente por causa

da estratégia de coalescência. Esta estrutura de dados tem sido bem sucedida em muitas simulações em tempo real, tais como, animação de fluidos (capítulo 5), simulação de multidão (capítulo 6), sistemas de partículas (capítulo 7) e jogos digitais simples (capítulo 8).

Os estudos de caso usam a NGrid, a fim de realizar a coleta de vizinhança. Ao fazer isso, a proposta atende também a falta de estruturas de dados da GPU, sendo uma solução eficiente para simulações em tempo real em soluções paralelo e distribuídas. A estrutura também tem sido aplicada com sucesso em arquiteturas móveis e arquitetura de multi-GPUs.

Organização da tese

Esta tese está organizada em duas partes, sendo a primeira relacionada proposta da arquitetura e a segunda para a sua aplicação e resultados. No Capítulo 2, o problema de coleta de vizinhaną é apresentado, apresentando conceitos relacionados sobre o problema e também onde ele se aplica. Este capítulo também fornece uma introdução sobre sistemas de tempo real e GPU Computing. O capítulo 3 apresenta algumas das soluções do problema coleta de vizinhaça que aparece na literatura. O Capítulo 4 apresenta a NGrid.

Em seguida, os capítulos de aplicação são apresentados, aonde todos os testes e implementações são detalhados e discutidos.

Capítulo 5 mostra a implementação e os testes da NGrid em uma animação de fluidos, com base na simulação de fluidos usando SPH, implementado usando CUDA. O capítulo também inclui o desempenho e testes de erro contra a estrutura de dados grade uniforme. O Capítulo 6 descreve a aplicação e os testes da NGrid numa simulação de multidões utilizando CUDA, numa única GPU e em um ambiente multi-GPU. Estes testes incluem a comparação do desempenho e do erro de NGrid com N-radius adaptativa e um fixo, e a grade uniforme. O Capítulo 7 mostra os detalhes da implementação e testes da NGrid aplicada sobre um sistema de partículas, utilizando uma plataforma móvel. Estes testes incluem também a comparação de desempenho com o método de força bruta. Seguido pelo capítulo 8, que descreve a implementação e testes do NGrid em um jogo usando CUDA. E finalmente, o capítulo 9 apresenta a conclusão deste trabalho, bem como as consideração finais.

Chapter 1
Introduction

Games and visual simulations normally spend most of the processing power in the scene with rendering and visualization. Commonly, fewer computation resources are left to process complex behavior tasks of the entities in the scene,which normally comes from physics and/or artificial intelligence. This fact makes the behavior and the number of interactive objects to be normally limited. Moreover, in order to process a high number of interactive entities in a real-time scene, it is necessary to develop a dedicated solution for this problem. These large number of elements in a scene usually requires neighborhood gathering strategies for the interaction among them, which affects the performance of the application.

The neighborhood gathering is the process of gathering the n closest entities to a certain entity, which is normally done for every entity. Usually, for a complete simulation, it is necessary to perform this calculation among a big set of entities, requiring a lot of processing power. The neighborhood gathering is applied in many simulation scenarios, with different purposes. In a crowd and game scenario, it is normally used to gather the closest entities in order to simulate the vision of the entity. In a SPH (Smoothed-Particle Hydrodynamics) fluid scenario, each particle needs its n closest particles in order to calculate the density, viscosity and forces of the particle. Also in a particle system, the collision detection algorithm needs the closest particles in order to check if there are any collisions between them.

The naive approach of such algorithm has complexity of $O(n^2)$, since it has to process each entity against the other entities. Many works that require real time results try to optimize this issue by avoiding the high complexity of proximity queries by applying some form of spatial or hierarchical subdivision to the environment and by classifying entities among the cells based on their position. To accelerate data fetching in a parallel hardware (such as GPUs) the entities list must be sorted in a way that all entities on the same cells are grouped together. This approach helps to lower the number of proximity queries, but

it is very sensible to the maximum number of entities that can fit in a single cell. In order to provide a data structure for massive real-time simulations, this thesis presents the NGrid, an new data structure for the neighborhood gathering problem, suitable for high parallel processing environments, such as GPUs

In real time applications, many non-graphics algorithms traditionally executed on the CPU are often suitable for parallel execution, which makes them appropriate to be implemented on the GPU. There are several research areas that use GPU computing in order to speedup its computation, such as weather forecast [2], chemistry [3] and behavioral AI algorithms [4]. However, the first applications of GPUs performing general-purpose computation (GPGPU) had to rely on the adaptation of graphics rendering APIs, leading to a difficult learning curve and sometimes not very efficient data structures for the proposed solutions [5]. CUDA [**?**] and OpenCL [**?**] technologies aim to provide a new abstraction layer on top of graphics hardware to facilitate its usage for non-graphics processing. Real-time simulation that explores this programming model on the GPU is a promising line of research, since it can speedup the high cost computation solution of the behavior simulation.

GPUs are a collection of SIMD processors designed to run streamed graphics pipelines. It is a computational model where the processing of each pixel is independent of the others and usually requires localized memory reads. There are rules of thumb to create efficient streamed applications, where, the most important one is to organize the data streams in a way that maximizes the memory read performance based on locality. These rules tend to result in more efficient usage of available memory and read ahead mechanisms of these devices.

This thesis proposes a novel structure, called the NGrid, which optimizes and make usage of data proximity for memory fetch. While the NGrid is mainly designed for the usage with manycores processors, such as GPUs, other parallel hardware, such as multicore CPU processors, may take advantage of it.

This NGrid is a discrete topological structure that keeps all particles sorted in such a way that particles, which are close to each others according to the Euclidian metric are inserted in neighboring positions at the NGrid. The NGrid does not have empty cells, because a particle always will have a neighbor, even when it can be far and does not influence the other entity's behavior. Hence, its size does match the number of particles. With that, the data structure is very suitable for parallel processing, avoiding bank conflicts and taking advantage of data coalescence by maximizing memory read

performance based on locality.

1.1 Contributions

This thesis presents the NGrid: a novel data structure, suitable for parallel processing, for real-time simulation that requires a neighborhood gathering algorithm. This structure is capable of processing a massive number of entities in a real-time application, being faster than traditional approaches, especially because of the coalescence strategy. This data structure has been successful applied in many real-time simulations, such as, fluid animation (chapter 5), crowd simulation (chapter 6), particle systems (chapter 7) and simple games (chapter 8).

The case studies use the NGrid in order to gather the neighborhoods of the entities. By doing that, this proposal will fulfill the lack of GPU data structures, being an efficient solution for other parallel and distributed solutions of real-time simulations. The structure also has been successfully applied in mobile architectures and in multi-GPUs architecture.

1.2 Publications

The initial results and the usage of the NGrid data structure of this thesis is present in the following peer review publications:

- Journals:
  - Journal of Computational Interdisciplinary Sciences [8] (Qualis B5): this work presents a flocking boids implementation with the processing of the work distributed among processors, which is particularly important since the NGrid will also have a mechanism for task division and distribution among multiples GPUs;
  - Computer in Entertainment [9] (Qualis B3): This work presents a architecture for task distribution among GPUs, which will be used when distributing task among multiples GPUs;
  - Computer in Entertainment [10] (Qualis B3): In this work an extended version of the paper [11] with the first concept of the NGrid is presented applied in a crowd scenario. Even though the author of this work is not the main author

of this paper, he has participated in the conception and implementation of the work;

- International Journal of Computational Fluid Dynamics [12] (Qualis B2): In this journal, a SPH fluid simulation with the use of the GPU is presented. This work uses a uniform grid as a neighbourhood gathering that is used as the traditional approach to compared against the NGrid on SPH fluid simulations (chapter 5);
- Lecture Notes in Computer Science [13] (Qualis C): This journal presents the results of the application of the NGrid in a mobile game framework.

- Conference procedings:
  - Brazilian Symposium on Games and Digital Entertainment 2008 [11] (Qualis B4): the first work, where a early concept of the NGrid appeared in a 2d flocking boids scenario. This work has received the best paper award for computer category;
  - Conferência de Ciências e Artes dos Videojogos 2009 [14] (Sem Qualis): This work presents a boid simulation with the use of uniform grid for neighbourhood gathering implemented on the GPU. This implementation is used as the traditional approach to compared against the NGrid on crowd simulations (chapter 6);
  - Brazilian Symposium on Games and Digital Entertainment 2009 [15] (Qualis B4): This work presents a game with all the logic implemented in the GPU with the use of uniform grid for neighbourhood gathering. The implementation of this paper is used as the traditional approach to compared against the NGrid on game scenarios. This work received 3rd best paper award of computer category;
  - Brazilian Symposium on Games and Digital Entertainment 2009 [4] (Qualis B4): This work presents an evolution of the NGrid on crowd simulation presented in [11, 10] allowing more complex space scenarios;
  - IEEE International Games Innovation Conference 2012[16] (Qualis B5): This work presents a framework for optimizing a game with the use of GPU Computing. One of the implemented techniques is the NGrid;
  - Brazilian Symposium on Games and Digital Entertainment 2012[17] (Qualis B4): This work presents the NGrid for crowd simulation implemented in a mo-

bile platform with the use of mobile GPU Computing. This work has received the second best paper award for computing category;

- Brazilian Symposium on Games and Digital Entertainment 2012 [18] (Qualis B4): This paper presents a SPH fluid simulation using multi-Gpus, which is particularly important since the NGrid also make use of multi-GPUs, and similar techniques are used in the Multi-GPU implementation.
- Workshop on Applications for Multi-Core Architectures 2012 [19] (Qualis B5): This work shows a distribution on multi-GPUs, which some ideas are used on the Multi-GPU implementation.

- Conference resumes:
  - CSBC GPU Forum 2012 [20]: This work shows a implementation of a game using the NGrid;
  - CSBC GPU Forum 2012[21]: This work shows a implementation of a crowd simulation using the NGrid;
  - CSBC GPU Forum 2012 [22]: this work shows the implementation of a SPH fluid simulation using the uniform grid for neighbourhood gathering;
  - CSBC GPU Forum 2012 [23]: This work shows a implementation of a particle system using the NGrid;
  - GPU Technology Conference 2012 [24]: This work shows the NGrid for simulation and games with the use of Gpu Computing.
  - GPU Technology Conference 2013 [25]: This work shows the NGrid for game simulation on mobile devices.
  - GPU Technology Conference 2013 [26]: This work shows the NGrid applied in a GPGPU game.

There is also a journal (on Computer Animation and Virtual Worlds Qualis B1) written and in reviewing process with the results of the application of the NGrid in SPH fluids.

1.3 Organization of this thesis

This thesis is organized in two parts, being the first related to the architecture proposal and the second for its application and results. In Chapter 2, the neighborhood gathering

problem is presented, providing a background on the problem being addressed by this thesis and also where it applies. This chapter also provides some introduction on real-time systems and GPU Computing. Chapter 3 presents some of the solutions of the neighborhood gathering problem that appears in the literature. Chapter 4 presents the novel data structure for neighborhood gathering, the NGrid.

Next, the application chapters are presented, where all the tests and implementations are detailed and discussed.

Chapter 5 shows the implementation and tests of the NGrid on a fluid animation, based on the SPH fluid simulation, using CUDA. The chapter also includes performance and error tests against the uniform grid data structure. Chapter 6 describes the implementation and tests of the NGrid on a flocking void simulation using CUDA, in a single-GPU and in a Multi-GPU ambient. These tests also include the comparison of the performance and error of NGrid with adaptive N-radius and a fixed one, and the uniform grid. Chapter 7 shows the details of the implementation and tests of the NGrid applied on a particle system using a mobile platform. These tests also include the performance comparison with the brute-force method. Followed by Chapter 8 that describes the implementation and tests of the NGrid on a simple Game using CUDA. Finally, Chapter 9 shows the conclusion of this work, as well as consideration and final analyzes.

Chapter 2
The Neighborhood Gathering Problem

The neighborhood gathering is the process of gathering closest entities of a certain element that are inside a given radius. In many simulations, this is done for every entity, having a complexity of O(n^2) in a brute force method. The neighborhood gathering method is applied in many scenarios, like crowd simulations, fluid simulations, particle system and the behavior of entities in games (AI and physics of the game). Even thought this problem appears in many scenarios, there are no works that discusses it in a generic way. This thesis tries to formalize and discuss it in a more generic way. Different authors, like [27] presents this gathering as the bottleneck of its simulation.

This problem appears often in the real time simulation literature, even though they do not deal with the problem in a generic way, since most authors only discuss the problem at the application level. This problem can be seen as a variant of the Fixed-Radius Nearest-Neighbors Search [28], where all points need to be found, given a fixed distance and a specified point in Euclidean space. In the case of real time simulations, this has to be done to the n entities in the scene [29]. This problem can also be seen as a modification of the Nearest neighbor search (NNS) [30], normally applied in pattern recognition [31], where the problem is to find the closest points in metric spaces of a certain point. Another similar problem is the KNNS (k-nearest neighbor search) [32], used in biology [33] and in image retrieval [34], where the problem is to identify the top k nearest neighbors to a certain point.

This chapter is divided as follows, first the background concepts of real-time simulations and GPU Computing will be presented. Then, the neighborhood gathering problem is described more formally, and later some of the real-time simulations that need a neighborhood gathering are presented. Finally, the summary of the chapter is presented.

2.1 Background

This section is dedicated to present some background concepts. Since the main application of the NGrid is in real-time simulation and games, the next subsection will present its concepts. Also, since the NGrid is mainly designed for GPU and manycore architecture, the main concepts on the subject will be presented as well.

2.1.1 Real-Time simulation

Graphical interactive real-time systems, like games and real-time simulations, are multimedia applications, and being so, they have time constraints to execute all of its processes and present to the end user the results. If a system does not fulfill this requirement, it will lose its interactivity and consequently it will fail. A common parameter for measuring a simulation is frames per second (FPS), which is the frequency (rate) that the system produces unique consecutive images called frames.The lower acceptable bound for being interactive is 16 FPS. Their are not higher bounds for a simulation FPS, but in PCs when the refresh rate of the monitor is less than the refresh of the simulation some discard of the rendered frame may occur [8].

The tasks that a real-time simulation normally should execute can be broken down into three general groups [35]: data acquisition, data processing and presentation. Data acquisition means gathering data from available input devices, such as, mice, joysticks, keyboards, and motion sensors. The data processing part refers to apply the user input into the simulation (user commands), apply the simulation rules, simulating the entities behavior (like physics and artificial intelligence), and related tasks. The presentation refers to providing feedback to the user about the current game state, through images and sounds. The main execution of a simulation or game can be seen on the workflow of Figure 2.1.

As listed previously, there are many tasks that a simulation must execute. A real-time simulation provides the illusion that everything is happening at once. In order to run a simulation in real-time, it must be an interactive application, and if it is unable to perform its work on time, the user experience will not be acceptable. This issue characterizes this kind of simulation as a heavy real-time application.

There are some previous works of the author that deals with the distribution of the simulation tasks between processors [8, 9, 35, 36, 37, 38], which are particularly important for this work, since it involves the distribution of workload between GPUs.

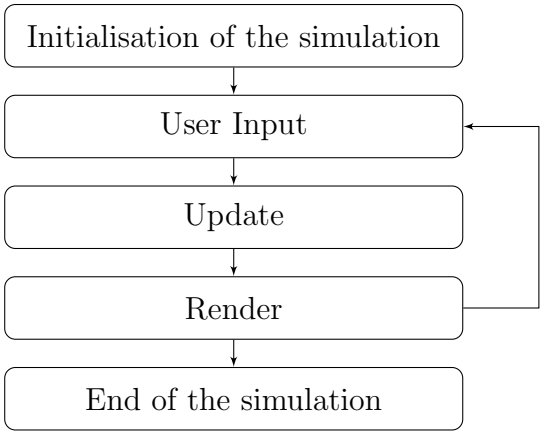

Figure 2.1: Workflow of a real-time simulation.

The neighborhood gathering is part of the data processing task group, since it is part of the update stage.

2.1.2 GPU Computing

GPUs are powerful processors originally dedicated to graphics computation. It is composed by several parallel processors, allowing it to present much better performance then modern CPUs in several applications scenarios. The Kepler K20 GPU card,for instance, can sustain a measured 4.7 TFLOPS/s against 60 GFLOPS/s of its contemporary CPU processors [39].

The GPUs architectures are being specially designed for processing tasks that require high arithmetic rates and data bandwidths. Because of the SIMD parallel architecture of the GPU, the development of this kind of application requires a different programming paradigm than the traditional CPU sequential programming model (the nVidia K20 [40], for example, has 2496 unified stream processors.). In order to take advantage of the GPU processing power, the developer needs to adapt its tasks to this kind of parallel paradigm, such as the data structure presented in this thesis.

The GPU can be used on the PC as a generic processor to process data and deal with computationally intensive tasks, through development of elaborate frameworks such as CUDA (Compute Unified Device Architecture) [**?**] and OpenCL (Open Computing Language) [**?**]. These frameworks facilitates the use of the GPU computing for generic processing. One main advantage in the use of these architectures is that they allow the use of the GPU in a more flexible way (both languages are based on the C language) without some of the traditional shader languages limitations (such as scatter memory

operations, i.e. indexed write array operations), and offering others features that are not even implemented on those languages (such as integer data operands like bit-wise logical operations AND, OR, XOR, NOT and bit-shifts) [41]. CUDA has a more mature architecture and is up to 30% faster than OpenCL [42, 43], but the disadvantage of the CUDA software architecture is that it is only available for the hardware of the proper vendors, i.e. CUDA only works on Nvidia Cards, while OpenCL is implemented in the main vendors of GPUs (AMD/ATI, nVidia and Intel), and even on CPU architectures. GPU computing can be applied in many different scenarios, like: fluid simulation [12], medical [44] and computer vision [45].

The GPGPU is getting more and more common and it is being applied in many fields, like geologic [46], medical [44] and computer vision [45]. The websites from CUDA [47] and gpgpu.org [48] show the latest development in the field.

The NGrid is mainly designed to be used in many cores processors like GPUs. It was developed having in mind different architectures using CUDA, Multiple GPUs and Mobile GPUs.

Nowadays, many GPU Computing systems are starting to have multiple GPU devices to solve their computational problems [49]. In order to distribute the workload across multiple GPUs, the developer must manage the data exchange between the main memory and these devices, guaranteeing consistency between the multiple copies of data, making the development for these architectures more difficult for the developer.

In order to allow a faster simulation, some kinds of problems are being solved using more than one GPU architecture. Among some works, we can cite [50], which solves a Fast Fourier Transform in 3D using multiples GPUs. Additionally, [51] computes a fast conjugate gradient in more than one GPU. Using more than one device to solve a problem requires a well-established process in order to share and processing data among these GPUs.

On mobile devices, the GPU is much less capable and powerful [52], and is typically integrated into the mobile processor system-on-a-chip (SoC), which also consists of one or several CPUs, DSP (digital system processor), and other available mobile-specific accelerators, as Figure 2.2 illustrates. This embedded GPU does not have a memory specific for it, having to share the system bus, with the others processors for accessing the memory. Consequently the memory bandwidth is also much lower when compared to the desktops GPUs [1].

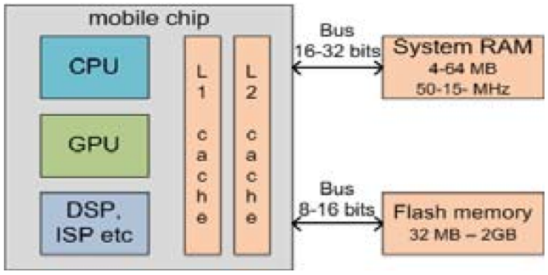

Figure 2.2: Mobile Hardware Architecture [1].

Currently, mobile GPUs emphasis more on lower power consumption [53] than performance. Some of these currently available GPUs devices are the Qualcomm's Adreno 200 GPU, the TI's PowerVR SGX 530/535 GPU and the nVidia Tegra3 GPU.

Normally, most works that uses mobile for parallel processing, deals with the use of the GPU for generic processing with the OpenGL ES [54] programable shaders, the vertex and fragment shader, as the programming interface [55]. The disadvantage of these approaches is the traditional shader languages limitations (such as scatter memory operations, i.e. indexed write array operations), and the lack of some features (such as integer data operands like bit-wise logical operations AND, OR, XOR, NOT and bit-shifts) [41]. The NGrid implemented in a mobile architecture will use the Renderscript API, that has also some of these disadvantages, like the limitation of scatter memory operations.

Renderscript is a new software development kit and API for Android firstly introduced by Google in the Honeycomb version of Android. Renderscript is an API for high-performance graphics processing on Android phones and tablets. It is used for fast 3D rendering and computing processing, having similar paradigm as GPU computing libraries and frameworks [56]. The main goal of Renderscript API is to bring a lower level, higher performance API to Android developers, in order to achieve better performance in visual animations and simulations [57].

The CUDA implementations of the NGrid have been designed to get the most of memory optimizations in order to speedup the data structure [58]. One of the techniques used is the data alignment. Since the device can read 4-byte, 8-byte, or 16-byte words from global memory into registers in a single instruction, all the data used is multiple of these values.

Also the global memory bandwidth, which is the main memory used in the GPU, is used most efficiently when the simultaneous memory accesses by the threads in a half-

warp can be coalesced into one or two memory transactions if the following rules are achieved: all the threads access 32/64/128-bit words, resulting in one 32/64/128-byte memory transaction, and all the memory accessed lies in the same memory segment, which must be equal to the transaction size. The NGrid is naturally organized to use coalesced memory access, since the NGrid sort the data according to access to the neighbors, as will be seen in chapter 4. Because of this organization the shared memory can be used inside the same block in order to help minimize the processing time, since it requires fewer warps to process. Also the same memory organization helps the optimization of the multi-GPU tests, that will be shown in Chapter 6.

2.2 The Neighborhood Gathering Problem

Neighborhood gathering problem is used in many real time simulation scenarios. In a virtual world, the n interactive entities may lay at a 2D or 3D in an Euclidian space. The method of neighborhood gathering consist of gathering the nearest set of entities with Euclidian distance less than a specify radius k, among the $n-1$ entities in the virtual world. This method is used together with others tasks of the simulation, like simulate behavior or physics. This work defines the problem based on the Fixed-Radius Near-Neighbors Search definition [29, 59, 60].

An entity is defined as an interactive element named i, placed in the virtual world, which has vector of positions in a determine time t:

$$f(i,t) = p_i \tag{2.1}$$

, where the p_i is a 3D vector:

$$p_i = [x_i, y_i, z_i] \tag{2.2}$$

In order to gather the set of nearest entities at a distance k of the entity p_i, the algorithm must test pairs of entities (p_i and p_b) in order to gather the set of entities that respects the equation:

$$G(i,b) = distance(p_i, p_b) < k \ AND \ i \neq b \Rightarrow add \ b \ to \ i \ list \tag{2.3}$$

, where b is the other entity and the distance is:

$$distance(p_i, p_b) = \sqrt{(x_i - x_b)^2 + (y_i - y_b)^2 + (z_i - z_b)^2}, \tag{2.4}$$

So, for solving the problem for all the simulated world, the equation:

$$\forall i, \forall b : G(i, b), \tag{2.5}$$

, must be respected in order to gather the neighborhoods' entities.

In order to illustrate this method, Figure 2.3 shows the i^{th} entity that will gather the entities with the smallest distance than the radius, which in this case is the set $\{A, E, F, K, O, P\}$ that will be gathered.

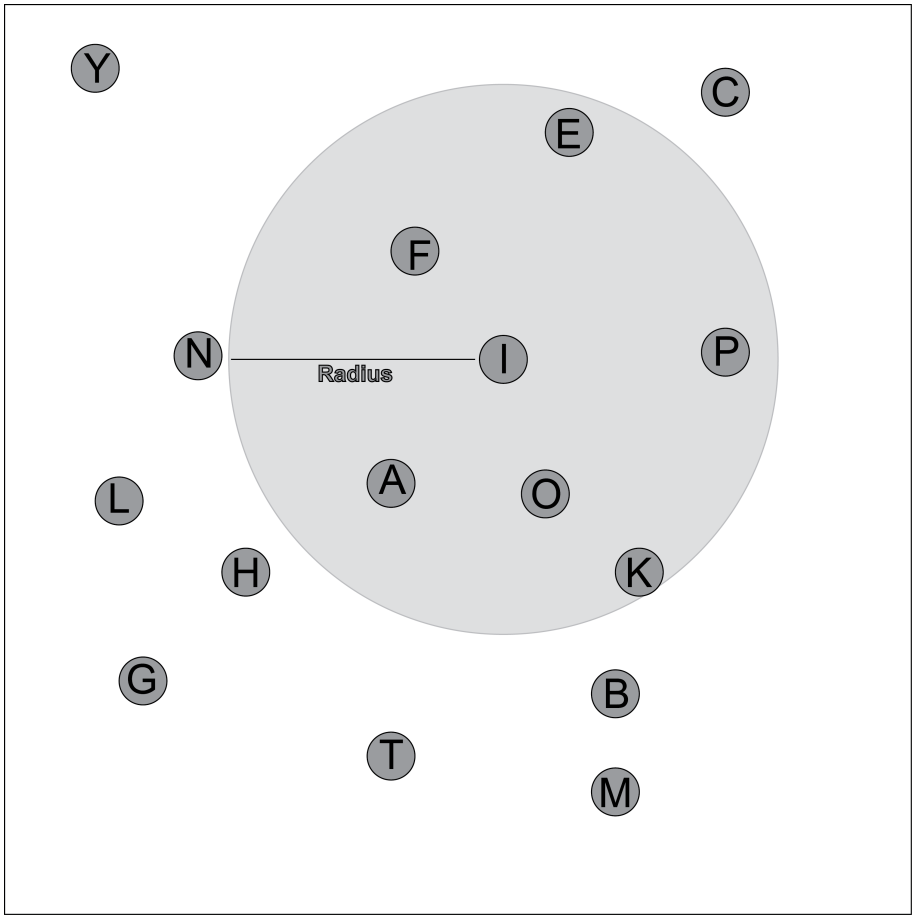

Figure 2.3: Regular grid for searching for neighboring particles.

The searching for neighboring entities is a bottleneck of many real time interactive applications. An entity can potently have Euclidean distance of k between every other entity. A world with n entities require $(n-1)+(n-2)+...+1 = n(n-1)/2 = \mathrm{O}(n^2)$. Due to the quadratic time complexity, naively testing every entity pair can quickly become too expensive even for moderate values of n. To speed up this process, the number of pairs tested must be reduced.

The naive approach of such algorithm has complexity of $\mathrm{O}(n^2)$, since it has to process each entity against the other entities. This implementation is showed in Algorithm 1.

Input: a set of entities P = { p_0,p_1,...,p_{n-1} }
Input: a radius k
Result: n set of neighbours N = { N_0,N_1,...,N_{n-1} }
for i=0;i<n;i++ **do**
for j=0;j<n;j++ **do**
if i ! = j AND distance(p_i,p_j) < k **then**
add j in i list, $N_i \Leftarrow j$
end if
end for
end for

Algorithm 1: Brute Force Neighborhood Gathering Algorithm

Since this problem is very time consuming, and would consume a lot of computation time, some solution have appeared in the literature that optimize this method. These solutions are based on the task of identifying smaller groups of entities that may be interacting and quickly exclude those that definitely are not. The available methods in the literature will be presented in the next chapter, and the NGrid will be presented in the subsequent chapter.

The neighborhood gathering problem comes from computational geometry field and does not appears, in this form, in the literature very often. However, some of the few available works are [61, 62, 63, 64, 65, 66]. This work concentrates on real-time simulations, where this problem is described with a different name, in the next section.

2.3 The Neighbourhood Gathering for Real Time Simulations

Neighborhood Gathering is required in different applications, such as: particle systems, crowd simulation, fluid simulation, physics simulation, and games, among many others.

2.3.1 Particle System

Dynamic particle systems is a computer graphics technique, commonly used in games, films and animations to animate fire [67], wind [68], smoke [69], clouds [70], and other "fuzzy" phenomena. These animations usually require a large number of entities, modeled as particles, interacting with each other and with the environment. Normally particles have simple rules for governing the individuals behavior with the complexity coming from the high number of particles.

This animation technique was first introduced in computer graphics by Reeves [71] for an animation used in 1983 in the movie Star Trek II: The Wrath of Khan, to simulate an explosion of a planet. This presentation shows the basic concepts of a particle system. He defined a particle system as a collection of many tiny particles, that during the simulation can be added to the system, moved, changed, or even removed from the system. Normally these particles are rendered as graphical primitives such as points, sphere and sprites. Later Reeves [72] extended his particle system by including more sophisticated particle motion for grass. Nowadays, all the major game engines and digital content creation system have some sort of particles system integrated into it.

The particle systems, on some animations, can have no mutual interaction, like on some simulations of wind and explosions [73]. In these cases, it does not need a data structure for neighborhood gathering. But in other cases, where the particles collides or interacts with each others , a neighbor gathering is needed.

Particles systems have the following workflow: first the particle system is initialized by creating the proper initializations; after entering the main loop of the simulation, where the particles may be created and destroyed according to the rules of the system; then the particles positions are updated according to the movement rules o the particles; and finally the particles are rendered on the screen. After the simulation finishes, the system ends the main loop and destroy the particles.

The motion of a Newtonian particle is governed by the second law of Newton ($F = m.a$). In this case, Particles are entities that have mass, position, and velocity, and respond to forces.

A Newtonian particle i has a know state:

$$S_i(t) = \begin{bmatrix} x_i(t) \\ m_i \\ P_i(t) \end{bmatrix}, \tag{2.6}$$

at the time t , consisting of position ($x_i(t)$), mass (m_i) and the linear momentum ($P_i(t)$). The particles could also have some others attributes like size, color, damping, friction, bounce and age.

The system is responsible to provide the knowledge the state of each of the particles $S(i,t)$, to determine the next state $S(i,t+\Delta t)$ of each particle into the scene, where Δt is the time step. This task involves the integration of the equation of motion of the

particle [74].

$$\frac{d}{dt}S_i(t) = \begin{bmatrix} v_i(t) \\ m_i \\ F_i(t) \end{bmatrix}, \tag{2.7}$$

, where

$$v_i = m_i^{-1} P_i \tag{2.8}$$

is the world linear velocity of the particle i, m_i is the mass of the body and F_i is the external force.

In order to calculate the interactions and collisions between the particles, a neighborhood gathering mechanism is needed. This collisions influentiates how the particle will behave during the simulation. For newtonian particles, it is required that the particles do not interpenetrate each other. Most particle system divides this step in two phases: a broad phase, and a narrow phase.

The broad phase is the neighborhood gathering step, which is responsible for avoiding the n^2 comparison between all the individuals, and also avoid doing a narrow phase of the collision detection between the n^2 individuals.

The narrow phase of the collision detection is responsible for doing the collision detection among the individual particles. The algorithm to perform this test is very simple in a particle system: if the particles are at a distance d from each other, less than the sum of radius $(r_1 + r_2)$, then they are colliding and must be treated accordantly. Figure 2.4 illustrates two particles near each other.

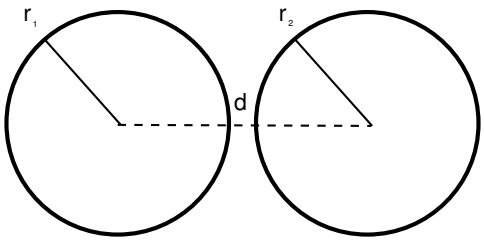

Figure 2.4: Two particles near each other.

The workflow of a newtonian particle system is illustrated in Figure 2.5.

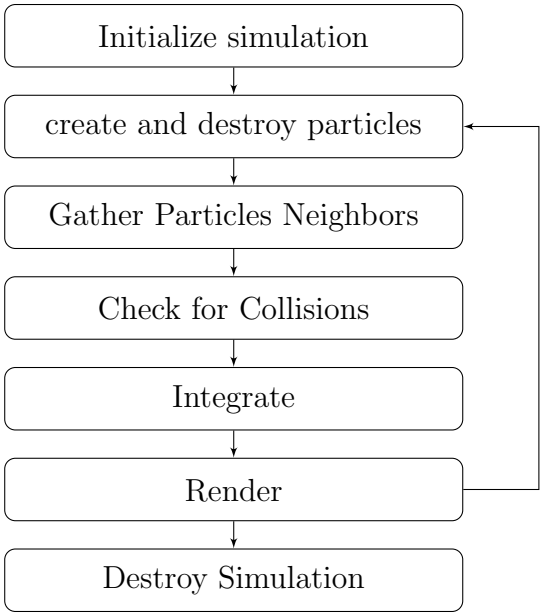

Figure 2.5: Workflow of a particle system

2.3.2 Crowd Simulation

In a typical natural environment it is common to find a huge number of animals, plants and small dynamic particles. This is also the case in other densely populated systems, such as sport arenas, communities of ants, bees and other insects, or even streams of blood cells in our circulatory system.

Crowd simulation are now appearing frequently on computer games, like Gran Theft Auto IV [75], and digital films, like trilogy of The Lord of the Rings [76]. Typical examples of the use of crowd simulation are the simulation of the behavior of group of animals [77], people walking on the street [78], soldiers fighting in a battle [79] and spectators watching a performance [80].

Computer simulations of crowds usually present a very limited number of independent entities, mostly with very predictable behavior. There are several approaches that aim to include more realistic behavioral models for crowd simulation such [77, 81, 82, 83, 84, 85]. All these models are based on the flocking boids approach [77], which also fundaments the crowd simulation test case on chapter 6.

Reynolds [77] presents the first distributed behavior model to simulate a flock of animals, that he called it boids (from bird-oid). This flock of boids is designed as a particle system, where each boid acts similar to particle. But instead of having the newtonian rules to control the behavior, it has also some behavior rules created by Reynolds. Each boid is implemented as a independent actor that behaves accordingly to its perception of

the environment, the physical rules (like gravity, inertia and collisions) and some set of behavior rules. In Reynold's work, these rules were 3 simply steering behavior of the boid, first a bird avoids too crowded local boid mates, second the boid tries to steer accordingly to the same place as its local boid mates, and third it tries to stay together with its group of boid. These rules can be seen on Figure 2.6.

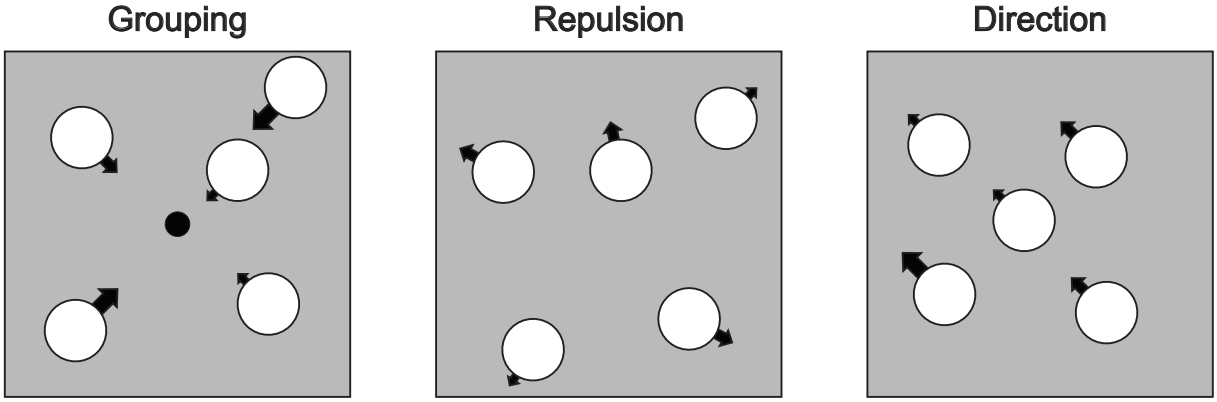

Figure 2.6: Flocking Boids Rules.

In order to achieve a believable simulation the algorithm tries to mimic what is observable in nature: many entities' behaviors depend on the vision of the entity. For that, it depends on a combination of internal and external factors (from the closest neighbors) that defines which actions are taken and how they are done. With this approach, internal state is represented by position, speed (also orientation) and the boid type, and external information refers to visible neighbors, depending on where the boid is looking at (orientation), and their relative distances. In order to mimic the vision of the individual, an neighborhood gathering algorithm is needed.

Figure 2.7 shows an entity with its' nearest entities to simulate its vision. The rules of the behavior of the crowd entities have evolved from these simple rules, to complex interactive agents, with memory and planning, but they still need some form of neighborhood gathering in order to simulate the agent's vision.

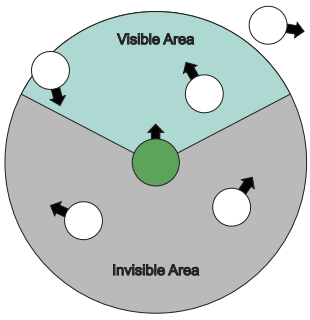

Figure 2.7: The entity and its vision.

2.3.3 SPH Fluid Simulation

Due to the graphics technology improvements, simulation of natural phenomena, such as water flows or smoke, has become possible to be performed in real time and interactive environments, like scientific applications and digital games. In fact, this kind of simulation is now quite popular in computer games, digital movies and animations. The use of aerodynamic effects and physics effects in games like races and flight simulators improves the player immersion.

Fluid simulation using SPH is a particle system, where the particles behavior follows the Navier-Stokes equations,

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}.\nabla\mathbf{v}\right) = -\nabla p + \rho\mathbf{g} + \mu\nabla^2\mathbf{v}, \tag{2.9}$$

and

$$\frac{\partial \rho}{\partial t} + \nabla.(\rho\mathbf{v}) = 0, \tag{2.10}$$

which are known as Navier-Stokes equations for modeling the flow of incompressible Newtonian fluids. In these equations, ρ represents the fluid's density, $\mathbf{v}$ the velocity field, p the pressure field, μ is the fluid's viscosity and g the resultant of external forces.

Equation (2.9) is known as the equation of motion and states that changes in linear momentum must be equal to all forces that act in the system. The convective term $\mathbf{v}.\nabla\mathbf{v}$ represents the change of a fluid's element properties that moves from one position to another.

Equation (2.10) is known as the *continuity equation* or *mass conservation* and states that in the absence of sinks and sources the mass in the system must be constant.

In this paper, the Navier-Stokes equations are solved using a mesh-less Lagrangian method called SPH. This method was introduced by Lucy [86] and Gingold and Monaghan [87] to perform simulations of astrophysical problems and latter extended for incompressive Newtonian fluids by [88].

In particle based Lagrangian methods, the convective term of Equation (2.9) and Equation (2.10) do not need to be solved since the material moves the flow and carries a fixed quantity of mass [89].

In SPH a compact support, radial and symmetrical smoothing kernel function is used to evaluate (anywhere in space) the field quantities defined only at a discrete set of

particles [90].

The evaluation of a continuous scalar field $A(\mathbf{x})$ is achieved by calculating a weighted summation of contributions for all particles $i \in [1...N]$, with position $\mathbf{x}_i$, mass m_i and additional attributes A_i using

$$A(\mathbf{x}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r}, h), \tag{2.11}$$

where ρ_i is the density of particle i, $\mathbf{r} = \mathbf{x} - \mathbf{x}_j$ and $W(\mathbf{r}, h)$ is the smoothing kernel.

The gradient and Laplacian of a smoothed attribute function $A(\mathbf{x})$ depends on the gradient and Laplacian of the kernel function, respectively

$$\bigtriangledown A(\mathbf{x}) = \sum_j m_j \frac{A_j}{\rho_j} \bigtriangledown W(\mathbf{r}, h), \tag{2.12}$$

$$\bigtriangledown^2 A(\mathbf{x}) = \sum_j m_j \frac{A_j}{\rho_j} \bigtriangledown^2 W(\mathbf{r}, h). \tag{2.13}$$

In SPH, the pressure is computed using a modified ideal gas law state proposed by Desbrun [88]

$$p_i = k(\rho_i - \rho_0), \tag{2.14}$$

where k is the stiffness constant of the fluid and ρ_0 corresponds to its rest density.

The SPH method divides the fluid into a set of discrete elements, referred as particles. In order to process the interaction between the particles using the Lagrangian approach, a neighborhood-gathering algorithm is needed in order to process the interaction for density, viscosity and force calculations. The neighborhoods gathering is normally done by some form of spatial subdivision and is responsible for avoiding the n^2 comparison between all the individuals in order to calculate the pressures, density and forces.

2.3.4 Physics Engines

A physics engine also follow as a base a particle engine, where a rigid body is a (possibly continuum) collection of particles, in which the relative distance between any two particles never changes, despite the external forces acting on the system.

In a scene composed of n rigid bodies, the state $S_i(t)$ of a body i at time t is defined by four variables that represents its coordinates and velocities of the body [74]:

$$S_i(t) = \begin{bmatrix} x_i(t) \\ mq_i(t) \\ P_i(t) \\ L_i(t) \end{bmatrix}, \tag{2.15}$$

where x_i is the world position in global coordinate of the center of mass C_i of the body, q_i is a quaternion that represents the rotation of the local reference frame of the body in relation to the world frame (with the origin in C_i), P_i is the world linear momentum and L_i is the world angular momentum of the body. The main functionality of a physics engine consists on determining the state $S_i(t + \Delta t)$ of each rigid body into the scene, for a known state $S_i(t)$, where Δt is the time step. This task involves the integration of the equation of motion of each rigid body i [74]:

$$\frac{d}{dt}S_i(t) = \begin{bmatrix} v_i(t) \\ \frac{1}{2}w_i(t)q_i(t) \\ F_i(t) \\ t_i(t) \end{bmatrix}, \tag{2.16}$$

where $v_i = m_i^{-1}P_i$ is the world linear velocity of the center of mass of the body C_i, m_i is the mass of the body, q_i is the quaternion $[0, w_i]$, $w_i = I_i^{-1}L_i$ is the angular velocity of the body, F_i is the external force, t_i is the external torque applied to the body and the I_i is a 3×3 matrix of the inertia tensor of the body at time t, respectively. The last one calculated in t:

$$I_i(t) = R_i(t)\, I_{0i}\, R_i(t)^T, \tag{2.17}$$

where R_i is the 3×3 rotation matrix correspondent to the quaternion q_i and I_{0i} is the inertia tensor computed in relation to the local system of the rigid body i at the time of its creation. If the density and geometry of the body is constant during the simulation, I_{0i} will also be constant.

Equation (2.16) is a first order ordinary differential equation (ODE); the component of a physics engine responsible by its integration (usually by applying a numerical method such as Runge-Kutta fourth-order) is called ODE solver. This method is used in the simulation step.

Generally, the motion of a rigid body is not free, but subject to constraints that

restraint one or more degrees of freedom (DOFs) of the body. Each constraint applied to a body introduces an unknown constraint force that should be determined by the physics engine in order to assure the restriction of the corresponding DOF. Constraints are related to joints between (usually) two bodies and/or (collision or resting) contact between two or more bodies [74]. In the case of our architecture, this kind of constraints are not calculated.

In order to compute the contact forces that will prevent interpenetration of bodies, a physics engine needs to know at each time t the set of contact points between each pair of bodies into the scene. The contact information includes the position and surface normal at the contact point, among others. This task is performed by a component integrated to the engine responsible for collision detection, which can be divided in a broad and a narrow phase [91].

This broad phase is responsible for avoiding the n^2 comparison between all the individuals, and also avoid doing a narrow phase of the collision detection between the n^2 individuals. This phase is a neighborhood gathering problem. The narrow phase of the collision detection is responsible for doing the collision detection among the rigid bodies, that is a collision check between all the polygons of the entities. In order to optimize this broad phase, and also have a similar bound structure for all the entities, a bound volume is commonly used. The bounding volume for any entity is a closed volume that completely contains the union of the entities' geometry. Some examples of bound volumes are the bounding sphere, the axis-aligned bounding box (AABB) and the oriented bounding box (OBB) [92]. An example of a geometry bound by a sphere can be seen on Figure 2.8.

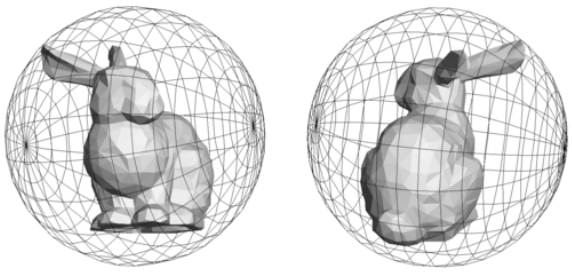

Figure 2.8: Bounding Sphere.

2.3.5 Digital Games

Computer games are multimedia applications that employ knowledge of many different fields, such as Computer Graphics, Artificial Intelligence, Physics, Network and others

[37].

The line of sight of NPC (non player characters) is the determination of every objects and entities that the NPC can see, and is used in order to simulate the vision of the NPC. This is used in order to simulate the behavior of the NPC.

In order to process the physics (the same as the physics engines) and AI steps (simulating the vision of the characters) some sort of neighborhood gathering method is needed. Most of the games tries to avoid the high complexity of proximity queries by applying some form of spatial subdivision to the environment and classifying entities among the cells based on their position.

2.4 Summary

In this chapter neighborhood gathering problem is presented and contextualize in different problems, related to the field of application of the NGrid.

There are also many other relevant publications that deals with neighborhood gathering, like raytracing [93], terrain visibility [94] and molecular dynamics [95] that are not detailed in this work but could also be good test cases the NGrid proposal, since they need a neighborhood gathering method.

The next chapter will present some solutions of the neighborhood gathering problem that appears in the literature.

Chapter 3
Solutions for the Neighborhood Gathering Problem in the Literature

Neighborhood gathering is still one of the bottlenecks of interactive environments, like games and real-time simulations. In scenarios with thousands of entities the processing of the neighborhood gathering can deteriorate the performance of the application. The brute-force approach will test all $O(n^2)$ pairs of entities. Hence, neighborhood gathering methods must ensure that this quadratic asymptotic behavior do not occur.

There are many methods that tries to avoid the quadratic complexity. Most of them tries to reach an average $O(n)$ complexity. This chapter presents the most used methods for neighborhood gathering mechanisms in the literature. First the sweep and prune method is presented, followed by the uniform grid and the hierarchical trees. Finally the summary is presented.

3.1 Sweep and Prune

The Sweep and Prune algorithm is a topological subdivision algorithm also called Sort and Prune algorithm or just SaP. The main idea of the algorithm, as the name says, is to do a dimension reduction, sort all the entities, and then prune it. This method appears more often in the literature as a broad phase mechanism for physics engines.

3.1.1 Algorithm Details

In order to determine that two objects are near each other, this approach reduces the 3D/2D problem into three/two 1D problems. This is done by determining the interval occupied by the entity along each of the axes (x,y and z for 3D and x and y for 2D), which is the contraction of the data structure. If the interval of two entities overlap, or is

less that the distance radius, in all of the axis, the entities must be near. This step test is called interception test. In order to determinate which intervals of the entities along the axis are close, the interval list is sorted. Since the entities move and rotate continuously, it is necessary to reconstruct and resort at every step of the simulation. The workflow of this method can be seen on Figure 3.1.

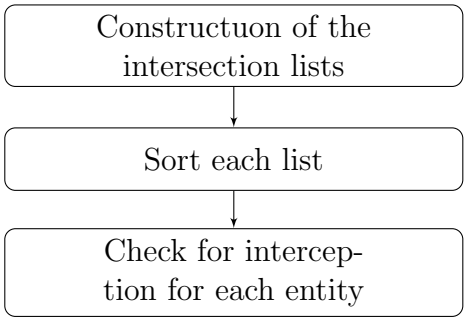

Figure 3.1: Workflow of the sweep and prune algorithm.

Figure 3.2 illustrates the SaP method in a 2D environment. Each entity is mapped to the X Axis and YAxis lists, and the neighbourhood gathering occurs by visiting the neighbours that appears in both lists with distance less than the radius of the neighbourhood gathering.

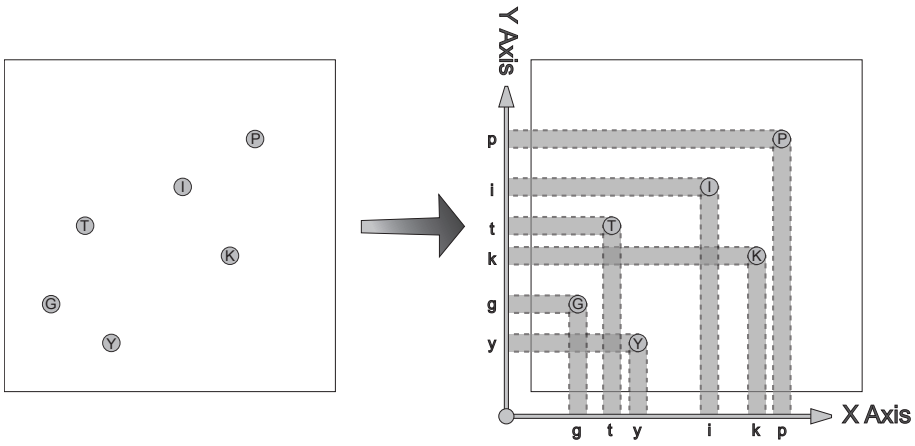

Figure 3.2: The SaP method illustrated.

The Sweep and Prune algorithm has an average O(n) complexity. Normally, the construction of the interval lists has an O(n) complexity and the sorting is commonly done by a quick-sort, which has a O($n * logn$) complexity, or simple a insertion sort, which can have an expected O(n), since the objects do not change position very often between frames. However, using the insertion sort for the sorting step can deteriorate to O(n^2) on a clustered axis because of small world movements causing large positional movements in the list [92]. The interception test has an average O($m * n$), where m is

the number of near objects. However, this numbers of near objects can be equal to the number of objects in the scene, having a $O(n^2)$ for the worst case for this method as well. Since, it needs one interval list for each axis, it has an space complexity of $O(2 * d * n)$, where d is the number of axis.

This method first appeared in the literature as part of a collision detection library, named as I-COLLIDE [96]. Followed by Baraff [97] that showed it applied on a physics engine.

This method appears more often in the literature as a broad phase algorithm for collision detection [92, 96, 98], but it also can be seen applied in the computation of line of sight algorithms [99], entity behaviour [100] and culling [101]. This method can be also seen in some open source physics engines, like Bullet [102] and Box2D [103].

There are not many implementation of the sweep and prune using parallel architecture. Among the few ones, Govindaraju et al. [104] presents a collision mechanism, with SaP as one of the steps of its collision detection, implemented in the GPU using depth and stencil buffer techniques. Liu et al [101] presents a culling system implemented in the GPU using CUDA, using a hybrid neighbourhood gathering mechanism, which combines the SaP method with a uniform grid mechanism.

There are also some variation of the SaP method, like the Kinect Sweep and Prune [105, 106], which uses the SaP with a kinetic data structure, transforming the SaP to an event-driven method. With the kinect data structure, the SaP list is only updated when there are collision responses, motion changes, or intersections. Also [100] presents some improvements for the SaP method by reducing the sort cost.

3.2 Uniform Grid

The uniform grid (also called a spatial hash) is a spatial subdivision technique, where the simulated world is subdivided in equally sized regular grids/boxes/buckets, called cells, among the 3/2 main axis of the simulated scene (x,y and z for 3D and x and y of 2D). This data structure appears very often in the literature applied in many real-time simulations. This method is the one choosen from the literature to compare the performance against the NGrid, since it is the most common in the literature.

3.2.1 Algorithm Details

This method discretize the simulated world into a set of equally sized regular volumes (or geometry in the case of 2D worlds), which are called cells. Normally, these cells should be larger than the diameter of the largest objects bound sphere, but there are some works that have smaller sized cells and the bigger objects are put in more than one cell. In order to gather the neighborhoods, all objects in the same cell and in neighboring cells are gathered (the number of neighboring cells are chosen according to the radius size).

Once the grid size is determinate, inserting the entity in the grid is a simple operation. In order to find the cell position simple divide each of the entity's coordinate position by the cell size, as can be seen of equation 3.1:

$$(c_x, c_y, c_z) = (\frac{p_x}{cellsize_x}, \frac{p_y}{cellsize_y}, \frac{p_z}{cellsize_z}) \tag{3.1}$$

This grid must be constructed in every step of the simulations, since particles move during the frames. Average case performance is O(n). However, the cell size directly affects the overall performance and efficiency of the algorithm, and can be a problem [107]. If the selected size is too small, many cells must update when an entity moves, reducing the performance, and also a larger amount of memory is needed for the uniform grid.

Another problem is when the cell size is too big resulting that many entities are in the same cell, or adjacent cells, requiring a hidden O(n^2) performance. Even when the scene has many entities that vary greatly in size, finding the optimum cell size becomes very hard [92]. Figure 3.3 illustrates two uniform grids, the first with a cell size too big and the second with a smaller cell size.

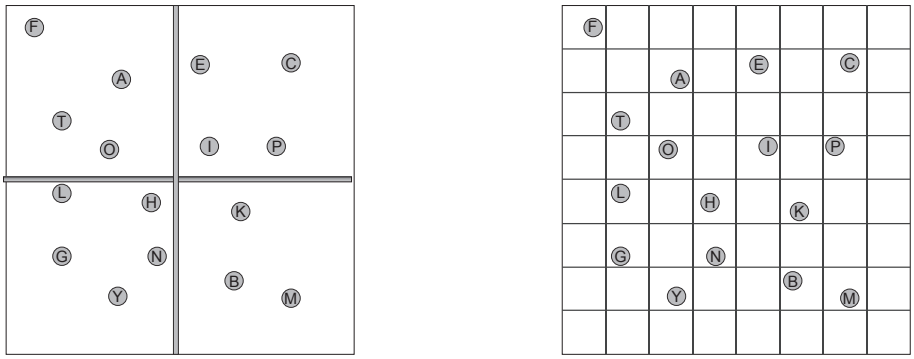

Figure 3.3: Two uniform grids with different cell sizes.

Another problem with the uniform grid is the memory consumption in bigger simu-

lated scenes. The memory requirements is $O(h * w)$, where h is height of the grid and w is the width of the grid. One solution for this problem is the construction of a hash table or bin table using the cell position as key [92, 108, 109], which gives a space complexity of almost $O(n)$. An example of a hash function can be seen on Equation 3.2:

$$i = [(i_x * 92837111)\, xor\, (i_y * 689287499)\, xor\, (i_z * 283923481)]\, mod\, n \tag{3.2}$$

, where i is the bucket number cell (i_x, i_y, i_z), which the entity is mapped to.

Figure 3.4 illustrates this process. The world is divided in cells, and each entity is indexed according to the cell position, and each entity gather the neighbors inside the same cell and the adjacent cells, depending on the neighborhood gathering radius.

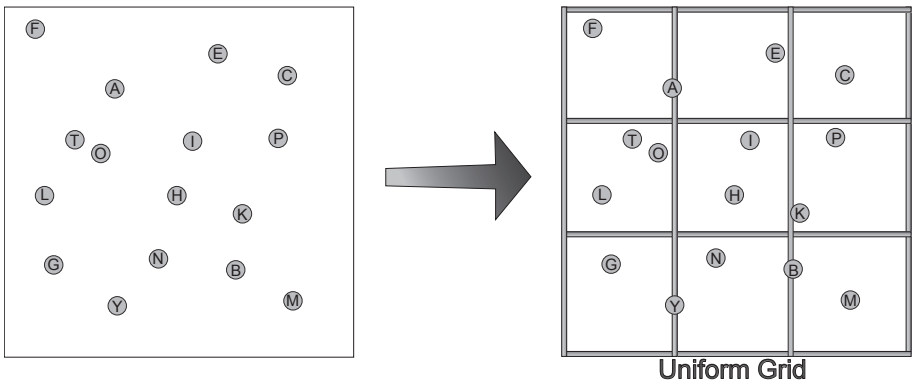

Figure 3.4: The uniform grid world subdivision.

There are many works that use this method in order to process the neighborhood gathering, and it is applied in many real-time simulations, like particle systems [110], crowd simulation [111], fluid simulation [12], raytracing [107], photon mapping [112] and physics engines, like ODE [113] and Chipmunk [114].

In a particle system, this work has appeared as part of the nVidia SDK [110], with the method explained in a book chapter [115]. The simulation was implemented in the GPU with the use of CUDA, and could process and render up to 65k particles in interactive frame rates.

In a crowd scenario, Reynolds first implemented a uniform grid [111, 116] that could simulate up to 280 boids at 60 fps in a Playstation 2 hardware. Reynolds implemented a more complex crowd simulation using uniform grid with a hash table that could simulate up to 15,000 boids in a Playstation 3 hardware [117] . A mix of GPU-CPU implementation can be seen on [118] that could simulate 1,600 boids and [119] that could simulate 16k boids in interactive frame rates.

Kurose and Takahashi [120] use this method to optimize the neighborhood gathering, but it also grows bigger in size as the fluid discretization grows. The uniform grid with SPH on GPU can also be seen in many others works, like [12], [121] and [122].

One way of reducing this problem is by using an index sort with the uniform grid. This index sort first sorts all the entities according to their cell indices and store them in an array. These indices of the sorted array are stored in each cell of the grid. With that, the cells just store one reference to the first particle with the corresponding cell index. This optimization is described in [112], and was implemented in the GPU by [123] for fluid simulation. Other optimizations based on memory convalescence can be seen on [27] and [124] applied in SPH fluid simulation.

3.3 Hierarchical Trees

Hierarchical tree data structure have been used in order to hierarchically arrange the entities in order to avoid the O(n^2) comparisons. There has been several Hierarchical tree data structures applied in real-time simulations, but the more common are the BVH, Kd-tree quadtree and octree.

The most common in real-time simulations are the octree and quadtree. A quadtree/octree is a tree, where each node (that is not a leaf) has connection with 4/8 others nodes of the data structure, hierarchically bellow the node.

3.3.1 Algorithm Details

Both the octree and quadtree data structures are built upon a recursive idea. Normally the quadtree is build for 2D simulations and the octree for 3D simulations. A quadtree is a tree data structure in which each internal node has exactly four children, while the octree has exactly eight children. These data structures can be seen as an extended binary tree [125], but instead of two child nodes it has four and eight child nodes.

While the root of the tree represents all the simulated space, each node in the quadtree is a square, and has four child nodes (unless it is a leaf node), representing the subspace of the region , while in the octree each node is a cube, and has eight child nodes. Each node in the quadtree either has exactly four children (eight in the case of the octree), or has no children (a leaf node).

A quadtree/octree works by dividing space into quadrants/boxes as entities enter that

space. It does this by following a simple rule, that for any node in the tree, the node will be divided further if more than a specified number of entities are contained within the node at any time. When a node is divided, it divides into 4/8 different rectangles/cubes by splitting the node in half according to the axis (x and y for the quadtree and x,y and z for the octree). Any entity that can fit entirely within one of these new nodes are pushed down (with entities that are on a node edges remaining in the higher-level node).

Figure 3.5 illustrates the process for a Quad-Tree. Each entity is divided recursively in four, until the there are the maximum specified number of entities per leaf. The structure is organized in the tree structure as Figure 3.6 shows. In this figures, the nodes represented by a blue circle represents the parent nodes, the nodes represented as a small square are empty leafs and the nodes in a green circle represent the leafs with entities attached to it. The letters correspond to the specific entities .

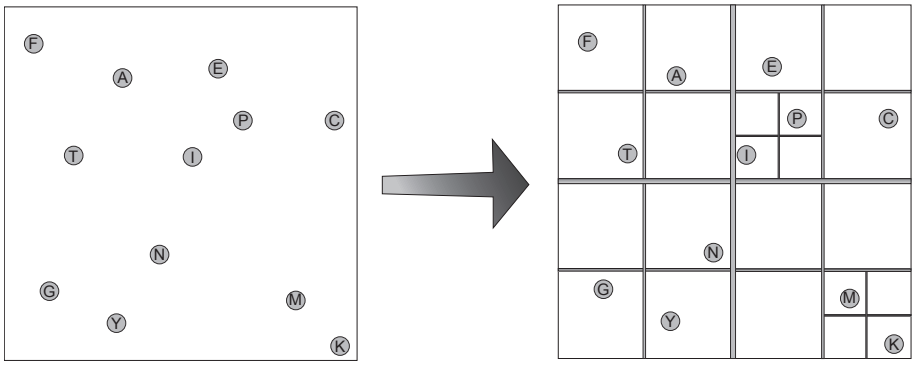

Figure 3.5: The environment subdivided in a quadtree structure.

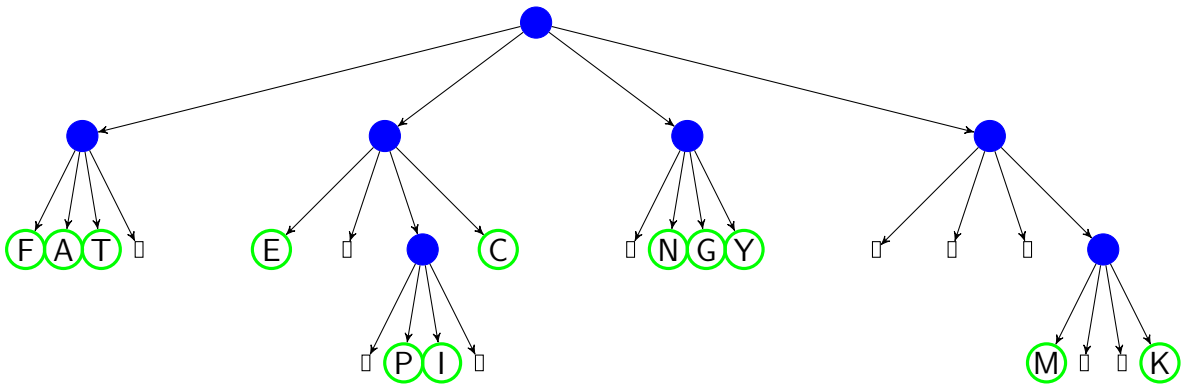

Figure 3.6: The entities organised in the tree structure.

The complexity of the Hierarchical tree depends on the distribution of the entities inside the simulated world. If the entities are uniformly distributed, so the leaves of the tree are all on the same level and the complexity of inserting all the n particles will be O(n*log n). Since the entities move during the simulation, this construction must be done

every simulation step. Also in order to do the neighborhood gathering, a search is needed, which can also be done in a O(n*log n) complexity in a uniformly distributed entities. The space complexity of Hierarchical tree is a expected O(n^2) [126]. Bad stochastic distribution of entities may degenerate the trees, with many empty cells.

The quadtree was first introduced in [127] and the octree in [128]. These data structure can be seen applied in many scenarios, like crowd [129], fluid simulation [130, 131], ray-casting [132, 133], image processing [134], geometric modelling [135] collision detection [136] and in a physics game engine [137].

Most works of Hierarchical trees implemented on the GPU deals with raytracing [133] and culling [138] problems. A Fluid simulation using quadtree with GPGPU can be seen on [139], which could simulate up to 16k particles at interactive framerate. An nbody problem on the GPU can be seen on [140].

3.4 Summary

This chapter has presented the most common data structures for dealing with the neighborhood gathering problem in real-time simulations. There are also some others structures that were not described in this work like BSP (Binary space partitioning) [141], hierarchical grids [92], Verlet list [142] and kd-tree [143]. But the more common on the selected real-time simulations scenario is the uniform grid, which will be used in this work as a comparison.

The next chapter will introduce the NGrid, which is another form of neighborhood gathering and it is the main contribution of this work.

Chapter 4
The NGrid

Many real time simulations neighborhood need gathering data structures. As highlighted in preview chapters, the naive straightforward implementation of the neighborhood gathering algorithm has complexity $O(n^2)$. Moreover, since entities are autonomous and can move during each frame, the neighborhood maintenance is a very computationally intensive task.

Different techniques, some of them presented in the last chapter, have been used to group and sort entities in order to accelerate the neighborhood gathering, especially for parallel and distributed solutions. Most of the implementations are based on some sort of spatial subdivision technique. On GPU and other parallel-based solutions, sorting algorithms may be used to reorder the particles, so it can benefit from the memory coalescence of such hardware with parallel architecture. Using this approach, Euclidean distant neighbors are stored near each other in the data structure, which is also the approach used in the NGrid to have a faster data structure for neighborhood gathering.

In this work a new approach for the neighborhood gathering problem is proposed. This chapter is organized as follows. First the NGrid is introduced in section 4.1. The properties of the NGrid are highlighted in section 4.2. The dimension size definition of the NGrid is presented in section 4.3. Then the sort mechanisms are presented in section 4.4 and in section 4.5 the gathering mechanisms are highlighted. Finally, section 4.6 presents an analyze of the NGrid and section 4.7 present the summary of the chapter.

4.1 NGrid: a Proximity Data Structure

The proposed data structure was mainly developed to be used in a parallel environment, like GPUs. This approach uses a proper grid data structure, which the author called NGrid and it stores information about all the entities of the simulation. In the NGrid, each entity is mapped to an individual cell (1:1 mapping) according to its spatial location.

Entities that are close in a geometric neighborhood sense are mapped to be close in the NGrid structure. In order to keep the NGrid properties (Section 4.2), a sorting mechanism is necessary. The following subsections describe the neighborhood data gathering using the NGrid data structure.

All the information about the entities are stored in arrays (the NGrid), where each entry holds the entire data for an individual entity. In this data structure, each cell fits one and only one entity, and has the same size as the entity. Figure 4.1 illustrates how a randomly distributed set of particles would be arranged in the NGrid when correctly sorted. The smaller entities represent particles that are further away from the viewpoint.

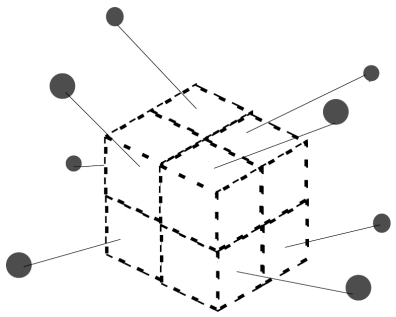

Figure 4.1: An example of a distribution of entities in the NGrid. Small circles illustrate entities that are further away from the viewpoint.

The NGrid can be described as a grid, where each cell has only one entity and it adapts its boundaries in order to completely hold each entity. Figure 4.2 illustrates the NGrid adaptation over a set of entities. From this figure it is possible to see that instead of adapting the entities to the cell, like the uniform grid does, the NGrid adapts the cell to the entities.

The NGrid structure is based on the Extended Moore Neighborhood [144] gathering algorithm using a 3D grid or a 2D matrix to hold the information of the entities. To reduce the cost of proximity queries, each entity only gathers the information about the entities surrounding its cell, based on the neighborhood search radius, called N-radius. Figure 4.3 illustrates the NGrid with a N-radius equal to 1.

This kind of spatial data structure with extremely regular information gathering enables a good prediction of the performance, since the number of proximity queries will always be similar over the simulation. This happens because instead of making the proximity queries over all entities inside a coarse grid/bucket/cell (variable quantity) and adjacent grid/bucket/cell, such as in traditional implementations, each entity would query only the surrounding individual neighbors. However, the NGrid has to be sorted continu-

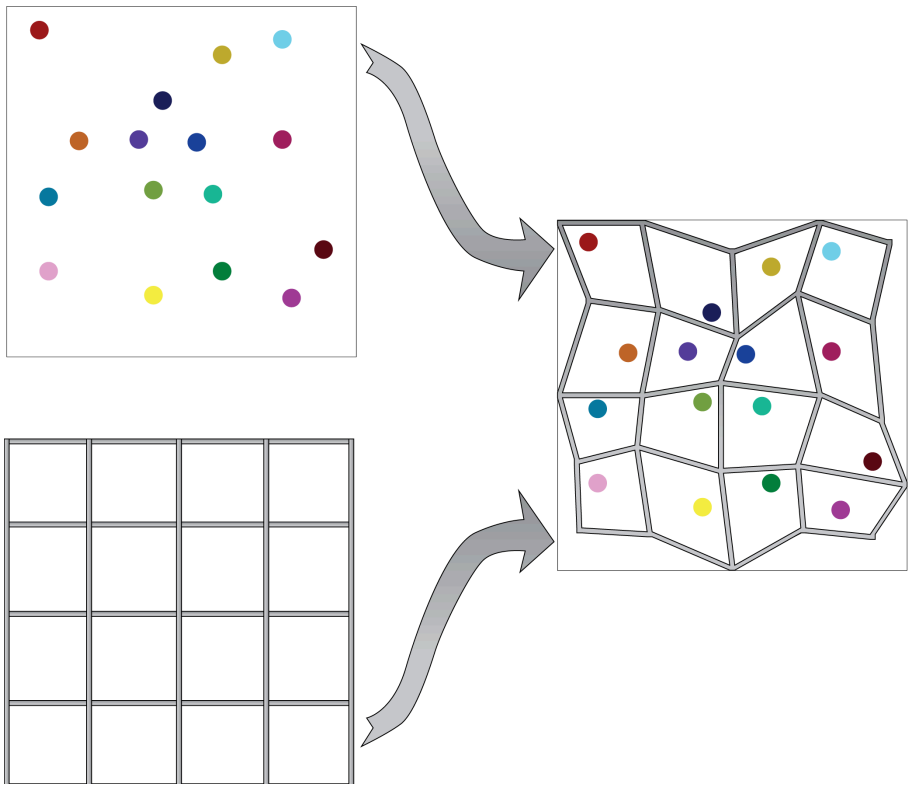

Figure 4.2: The adaptation of the NGrid cells over a set on entities.

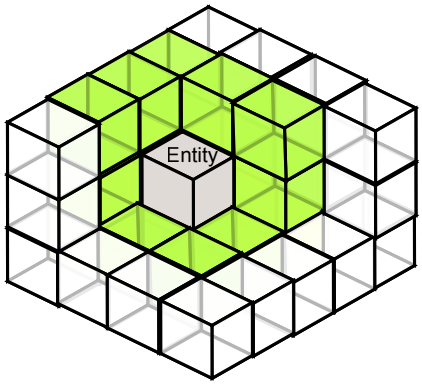

Figure 4.3: Example of the NGrid with N-radius of 1.

ally in such a way that those entities, which are neighbors in geometric space are stored in individual cells that are close in the NGrid, so that each entity should gather information only about its closest neighbors. In order to better illustrate how the entities would be organized by the NGrid and by the Uniform Grid, Figure 4.4 illustrates a set of particles and how they are organized in both structures.

The NGrid starts, at the beginning of the simulation, by determining the dimensions sizes (which are called N-dimensions) and does a full sort on all the dimensions, but during the simulation (and depending on the sorting step), some misalignment may occur over

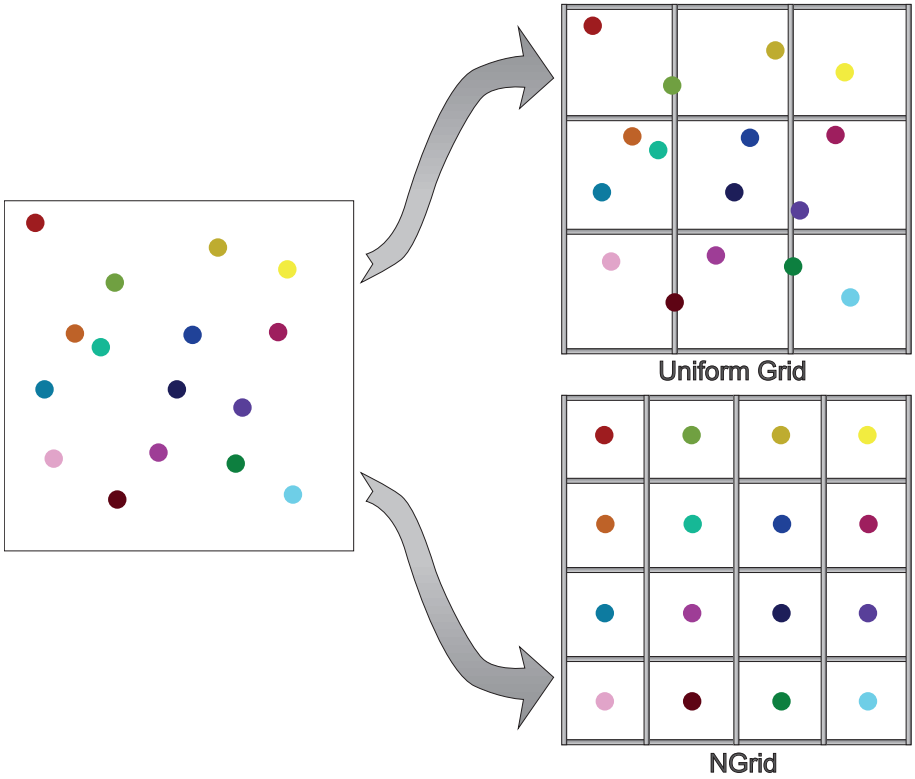

Figure 4.4: Example of the organization of a set of particles in the NGrid and the uniform grid.

the data structure because the entities can move during the simulation. This can make the gathering step to miss some of the neighbor entities. However, since the algorithm needs a sorting mechanism, this misalignment is very small when used a proper NGrid dimension size and time step - less than 1% of all entities are misaligned in all the test cases (when compared with the full sorted NGrid), and in the next step this misalignment is fixed by the sorting mechanism. The outline workflow of the NGrid processing is illustrate at Figure 4.5.

This type of structure cannot guarantee that the processed set of entities' neighbors corresponds to the closest set of neighbors in the Euclidean way, since the NGrid can be misaligned or the chosen N-radius can be small. But with the right sorting algorithm and the right radius this kind of algorithm can yield visually believable simulations. The tests from this thesis show that this approximated neighborhood gathering is an interesting model for optimizing real-time visual simulations, that needs a neighborhood gathering algorithm and needs a fast visual feedback for the animation.

The structure can be used in 3D or 2D scenarios. In a 2D scenario the NGrid is a simplification of the 3D proximity with only the X and Y (or Z) dimension. In this case, the proximity data structure used is a 2D NGrid. The 2D term mentioned refers only to

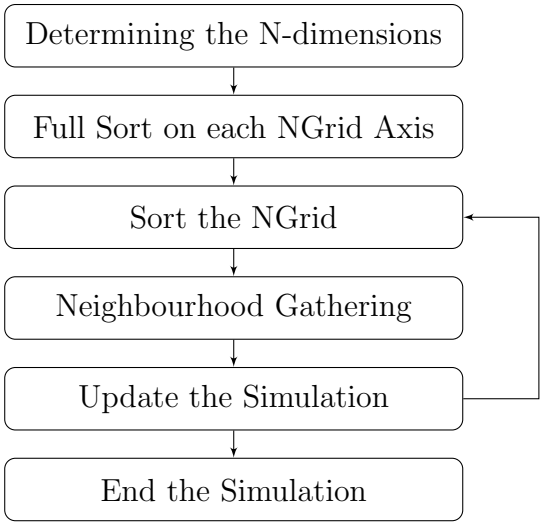

Figure 4.5: Workflow of the NGrid.

the spatial nature of the data structure, which is still suitable for a simple 3D simulation where the entities do not traverse the third dimension too much, such as particles spread at a terrain.

4.2 Properties of the NGrid

The NGrid is a data structure designed specifically for the Neighborhood Gathering problem. Instead of partition the world like the spatial subdivision techniques, the NGrid is a topological subdivision data structure, that has the following properties:

Property 1: The number of cells in the NGrid will be equal to the number of the entities being simulated, so in a simulated world with n entities, the NGrid will also have n cells.

Property 2: The number of entities on each NGrid's cell is equal to one. With that, there are not empty cells on the NGrid.

Properties 1 and 2 correspond to the properties of the number of cells of the NGrid. The NGrid is a topological data structure, where each entity is mapped to one and only one cell, keeping every cell with one entity. This way, the NGrid does not requires any additional memory space, since the entities' position array can be used as the NGrid.

Property 3: The size of each cell correspond to the size of the entity that it holds.

Property 4: The position of the cell on the simulated world correspond to the position of the entity that it holds. With that, the position of the cell on the NGrid

correspond to the position on each N-dimension.

Properties 3 and 4 corresponds to the properties of the NGrid's cells, which have the same size and position of the entity it holds. This means that the cell on the NGrid has a different approach as the cell on spatial subdivision techniques, where the entities are mapped to the corresponding cell. In the case of the NGrid the cell maps itself to the entity, having the same properties as the entity it holds.

Property 5: The NGrid is an array that have one N-dimension value for each axis (x and y for 2D and x,y and z for 3D), and the size of each N-dimension value can be fixed or dynamic.

Property 6: The size of the N-dimension value of each axis of the NGrid can vary, but the multiplication of the N-dimensions must be equal to the number of NGrid cells. So for a 2D NGrid, $Ndimension_x * Ndimension_y = n$ and for a 3D NGrid, $Ndimension_x * Ndimension_y * Ndimension_z = n$.

Properties 5 and 6 corresponds to the N-dimension properties of the NGrid. The NGrid can have the N-dimension fixed as a value in the beginning of its usage, or it can use an algorithm for its determination. The algorithm approach can also be use during the simulation, to determinate if the entities distribution on the environment has changed. This property and the algorithm are better explained in section 4.3.

Property 7: For each dimension of the NGrid, every element on the same dimension must be organized in a way that for a array of values A, $\forall \mathbf{A}[i] \in \mathbf{A},\ i > 0 \Rightarrow \mathbf{A}[i-1] \leq \mathbf{A}[i]$, where $\mathbf{A}$ represents the value corresponding to the axis (X,Y or Z).

Property 7 correspond to the maintenance step of the NGrid, which need to be sorted in order to adapt its cells to the entities' cells, since it moves during the simulation. This property is better explained in section 4.4.

Property 8: In order to gather the neighbors each entity have a separated N-radius for each dimension. This N-radius may be fixed or dynamic. The N-radius determine how many neighbors' cells the gather method will visit.

The property 8 is related to the gathering of the neighbors. The NGrid can use a fixed N-radius, which can achieve good results, but it must be determinate in a way that does not produces a lot of errors, and can be adaptive in a way that can be determinate to each entity in all directions. Both methods are better explained in section 4.5.

4.3 Determining the N-dimensions

The N-dimensions values can be determined by two ways, a fixed way, by defining a number or algorithmically, which analyses the position of every entity and determine the dimensions sizes of the NGrid. The fixed dimension size can be very simple to implement, since it only needs to calculate the dimension size so that for a 2D NGrid,

$$Ndimension_x * Ndimension_y = n \tag{4.1}$$

and for a 3D NGrid,

$$Ndimension_x * Ndimension_y * Ndimension_z = n \tag{4.2}$$

where n is the number of entities in the world. The fixed dimension size could lead to gathering errors, since the N-dimension value do not fit well the dispersion of the entities in the world. Also the distribution of the entities in the world could change during the simulation, requiring an adaptation of N-dimension sizes, since it could not be done previously, at a fixed N-dimension.

Figure 4.6 illustrates an NGrid with a 8×2 N-dimensions and a NGrid with a 4x4 N-dimensions. It is possible to notice from the figures that the 4×4 sample adapts better in the environment, keeping the neighbors closer among each other.

In order to determine the N-dimensions size dynamically, an algorithm is required. The algorithm starts by determining similar values for each N-dimension size. For example for a quadratic number of entities n, the N-dimension in 2D would be $\sqrt[2]{n}$ for each N-dimension value. The algorithm needs, as an input, the position of every entity, then it sorts the entities in every dimension according to the defined dimension. After sorted, the algorithm determine the average of the difference between the value of the distance between each neighbor position in each N-dimension, which is called the step. If the step in one N-dimension is higher (in this work is set to more than 20 %, which was the best result among tests) than the other N-dimensions, these dimensions must change, because different step values mean that the entities are not distributed according to the N-dimensions of the NGrid. Determining if the difference steps are high or low comes from the size of the simulated world. This algorithm is illustrated in the Algorithm 2.

Also, since the distribution of the entities's position in the world may change during the simulation, this method is used during the simulation together with the sorting stage to change dynamically the dimensions size whenever needed.

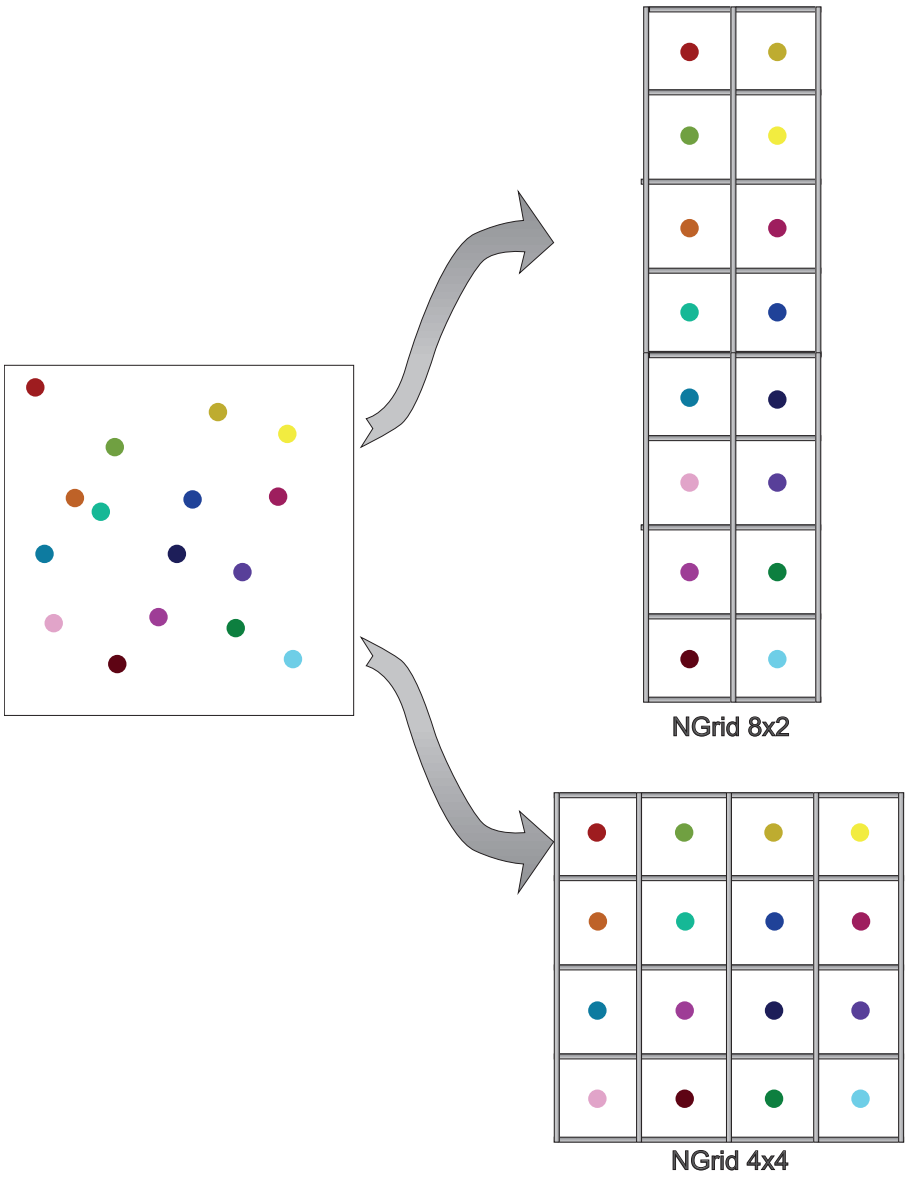

Figure 4.6: Different NGrid dimensions for the same set of entities.

Input: INPUT: Entities Positions Array
Result: Entities Positions Array
Result: N-dimensions
Determinate similar N-dimensions for each axis
repeat
for all Dimensions **do**
Sort
Calculate the average step
end for
until steps difference is small

Algorithm 2: Algorithm to find the dimension of the NGrid

4.4 Maintenance of the NGrid: Sorting Stage

Since the particles move at each frame, the NGrid may become misaligned. In order to maintain the NGrid in such a way that neighbors in geometric space are stored in cells close to each other, it has to be sorted in every step of the simulation. This section presents the process of sorting such data structure.

The position information of each entity is used to perform a lexicographical sort based on the three dimensional coordinates of the position's vector. The goal is to store the entity with the smaller values for Z, Y and X in the closer-bottom-left cell of the grid, and the entity with the highest values of Z, Y and X in the far-top-right cell respectively. Using these three values to sort the grid, the furthest lines will be filled with the entities with the higher values of Z while the top lines will be filled with the entities with higher values of Y and the right columns will store those with higher values for X,according with figure 4.7.

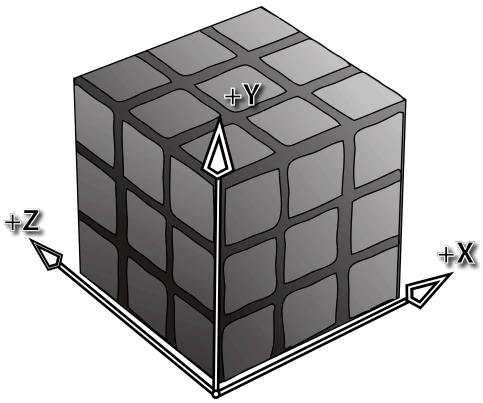

Figure 4.7: The sorting of a NGrid.

This sorting strategy provides data for the approximate neighborhood query, which is optimal in terms of data coalescence and bank conflicts avoidance. When performing a sorting over an one dimensional array of floating point values, the goal is that, given an array $\mathbf{A}$, the following rule must apply at the end:

$$\forall \mathbf{A}[i] \in \mathbf{A},\ i > 0 \Rightarrow \mathbf{A}[i\text{-}1] \leq \mathbf{A}[i] \tag{4.3}$$

. Extending this rule to a grid $\mathbf{G}$, each cell has three floating point values X, Y and Z, and the following rules are defined:

1. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ k > 0, \mathbf{G}[i][j][k-1].Z \leq \mathbf{G}[i][j][k].Z$;
2. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ k > 0, \mathbf{G}[i][j][k-1].Z = \mathbf{G}[i][j][k].Z \Rightarrow \mathbf{G}[i][j][k].X \leq \mathbf{G}[i][j][k-1].X$;

3. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ k > 0, \mathbf{G}[i][j][k].Z = \mathbf{G}[i][j][k].Z\ and\ \mathbf{G}[i][j][k].X = \mathbf{G}[i][j][k-1].X \Rightarrow \mathbf{G}[i][j][k].Y \leq \mathbf{G}[i][j][k-1].Y$;

4. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ j > 0, \mathbf{G}[i][j-1][k].Y \leq \mathbf{G}[i][j][k].Y$;

5. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ j > 0, \mathbf{G}[i][j-1][k].Y = \mathbf{G}[i][j][k].Y \Rightarrow \mathbf{G}[i][j][k].Z \leq \mathbf{G}[i][j-1][k].Z$;

6. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ j > 0, \mathbf{G}[i][j-1][k].Y = \mathbf{G}[i][j][k].Y\ and\ \mathbf{G}[i][j-1][k].Z \leq \mathbf{G}[i][j][k].Z \Rightarrow \mathbf{G}[i-1][j][k].X \leq \mathbf{G}[i][j][k].X$;

7. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ i > 0, \mathbf{G}[i][j][k].X \leq \mathbf{G}[i-1][j][k].X$;

8. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ i > 0, \mathbf{G}[i-1][j][k].X = \mathbf{G}[i][j][k].X \Rightarrow \mathbf{G}[i][j][k].Y \leq \mathbf{G}[i-1][j][k].Y$;

9. $\forall \mathbf{G}[i][j][k] \in \mathbf{G},\ i > 0, \mathbf{G}[i-1][j][k].X = \mathbf{G}[i][j][k].X\ and\ \mathbf{G}[i][j][k].Y = \mathbf{G}[i-1][j][k].Y \Rightarrow \mathbf{G}[i][j][k].Z \leq \mathbf{G}[i-1][j][k].Z$;

The NGrid data structure is independent of the sorting algorithm used, as long as the rules above are always valid, eventually or even partially achieved during simulation, depending on the desired neighborhood precision.

In the case the simulated world is a 2D, the sorting pass is a simplified one with just the X and Y passes. The next subsections will present the sorting strategies that are used together with the NGrid, and partial odd-even sort and a bitonic sort.

4.4.1 Partial Odd-Even Sorting

The odd-even sort is a relative simple sorting mechanism that was originally created for parallel processors [145]. This strategy works by comparing all the odd adjacent pairs in the same NGrid dimension, and if a pair is in the wrong order, being the first larger than the second, the entities are swapped. The next step repeats this process for the even pars. Then it repeats until the array is correctly sorted. This mechanism is described in Algorithm 3.

Here, the odd-even is used as a partial sort strategy, the odd-even transposition sort, with only one odd-even pass per update. The odd-even transposition sort is similar to the bubble sort algorithm and it is possible to complete a partial pass, traversing the whole data structure, in $O(n)$ sequential time or $O(1)$ parallel complexity when running on n threads (if available on the environment). Because there are two steps, one for odd and other for even elements (for each axis), this algorithm is suitable for parallel execution.

```
Input: Entities Positions Array
Result: Entities Positions Array
  repeat
    sorted = true
    for all D=Dimensions do
      for all E=Odd Entities in D do
        if Position[D][E] > Position[D][E+1] then
          Calculate the average step
          sorted = false
        end if
      end for
      for all E=Even Entities in D do
        if Position[D][E] > Position[D][E+1] then
          Calculate the average step
          sorted = false
        end if
      end for
    end for
  until sorted == false
```

Algorithm 3: Odd even sort for a NGrid

Figure 4.8 shows a schematic presentation of a partial odd-even transposition sort pass. The dark cells represent the cells that are sorted during the step.

This sorting pass must be spread into six steps, one for the odd and one for the even elements of each axis. The first step runs the sorting process between each entity position vector of the even columns against its immediate neighbor in the subsequent odd column. If the rules described by Rule 1, Rule 2 or Rule 3 are violated, the entities switch cells in the grids. The other six sorting steps perform the same operation for the odd column of the Z and the similar steps over the Y and X axis.

From the tests performed in this work, it was possible to see that with this partial sort more than 10% of entities are in the wrong Ngrid position when comparing with a full sort on the entire NGrid. So this sorting mechanism only seems viable on simulation that does not need a lot of precision, or that the entities does not change position very often. Otherwise the usage of the bitonic sort is suggested, as present in next session.

4.4.2 Bitonic Sort

This work also uses a bitonic sort [146], which makes a full sort in each dimension. The algorithm, created by Ken Batcher in 1968 [147], consists of two parts. First, the unsorted sequence is built into a bitonic sequence; then, the series are split multiple

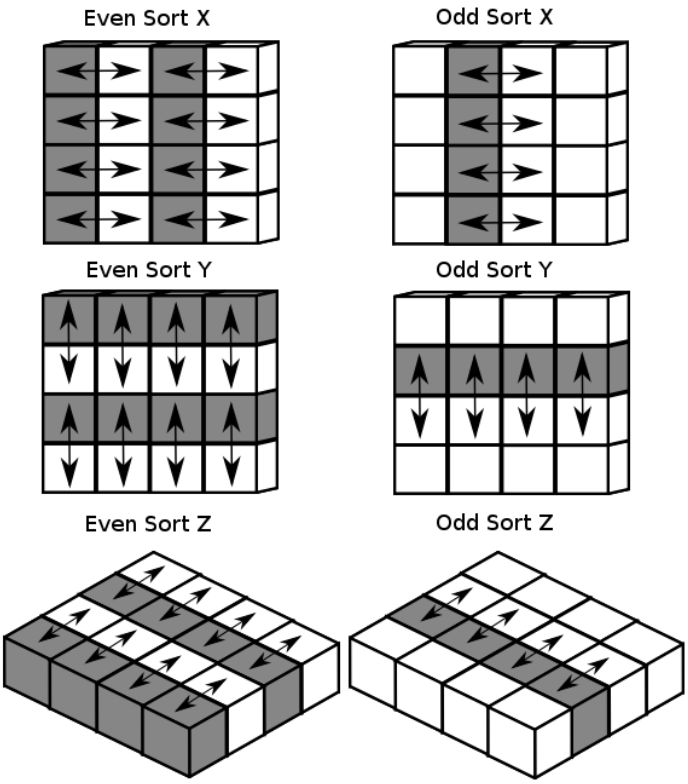

Figure 4.8: Partial sorting pass with 6 odd-even transposition steps.

times into smaller sequences until the input is in sorted order. The bitonic sort [146] is a parallel sorting algorithm that is very efficient when sorting a small number of elements [148], which is the case of the NGrid since the sort strategy is applied to each dimension separately.

The bitonic sort is a comparison-based sorting algorithm, and it is mainly designed to run in parallel. The approach is a divide and conquer strategy, where, first, a comparison is built for sorting a bitonic sequence, dividing into two subsequences, where all elements of the first are smaller or equal than those of the second. The subsequences themselves are sorted by recursive application of BitonicSort, and a BitonicMerge is used to combine those sorted subsequences.

The used implementation is an optimized and adapted version based on a previous work of nVidia [149]. This sort is divided into 3 passes, one for each dimension (X,Y and Z) of the NGrid.

The complexity of this algorithm is $O(n * log(n)^2)$ where n is the number of elements to sort in sequential time. This comparisons are performed by m CUDA threads making such parallel implementation of the algorithm to perform with a complexity of $O(log(n)^2)$,

if there were $m = n$ stream processors on the processor.

This sorting stage does not make a full sort on the NGrid but only a full sort on each dimension (X,Y and Z) of the grid. If a change is made, for instance, in one entity position on the Y pass, another pass for the X would be needed in order to keep the NGrid with a full sort. In the tests the author have seen that this misalignment is very small, usually less than 1% of the entities changes place in one step of the simulation. In the next step this error will be fixed, and the use of a full sort on the neighborhood grid would impose some loss in performance without visible gain in the simulation.

4.5 Gathering of the NGrid

After establishing the NGrid correct dimension and it is already sorted, it is necessary a gathering mechanism. Since the NGrid does not have a spatial structure, but a topological one, it cannot use the radius, but will use instead N-radius, which is the number of NGrid neighbors that the entities will visit.

The NGrid uses two forms of neighborhood gathering, a fixed N-radius and an adaptive one.

4.5.1 Fixed N-radius

The fixed N-radius, as the name says, is the N-radius fixed at the beginning of the simulation. This gathering mechanism is based on the *Extended Moore Neighborhood* that is used in the Cellular Automata theory [150] . The algorithm for gathering such a neighborhood can be seen at Algorithm 4. This algorithm takes as input the fixed N-radius and the NGrid, the array with the stored particles, which were already sorted.

Figure 4.9 illustrates the structure that was built with the NGrid in a 2D matrix holding arbitrary information for 16 individual entities. In this gather mechanism each entity only gathers the information about the entities surrounding its cell, based on a constant search of N-radius. In the example of Figure 4.9, the chosen fixed N-radius is 1, so the entity represented as I (in light gray) would have access to the 8 highlighted surrounding cells (represented in dark gray).

Using the fixed N-radius could cause that some neighbors are missing by the method, like Figure 4.9 illustrates. The entity I should gather entities that are less than the radius, which should be { A, E, O, P, N, K, B } and not { A, E, C, O, P, N, H, K } like it is

```
Input: Grid - Entities Positions Array
Input: Nradius - the fixed N-radius
Input: indexZ - index of the entity on dimension Z
Input: indexY - index of the entity on dimension Y
Input: indexX - index of the entity on dimension X
  for z=-Nradius;z<=Nradius;z++ do
    for y=-Nradius;y<=Nradius;y++ do
      for x=-Nradius;x<=Nradius;x++ do
        if (z!=0 OR x != 0 OR y != 0) then
          Do computation with (Grid[indexZ+z][indexY+y][indexX+x])
        end if
      end for
    end for
  end for
```

Algorithm 4: The fixed N-radius gathering algorithm

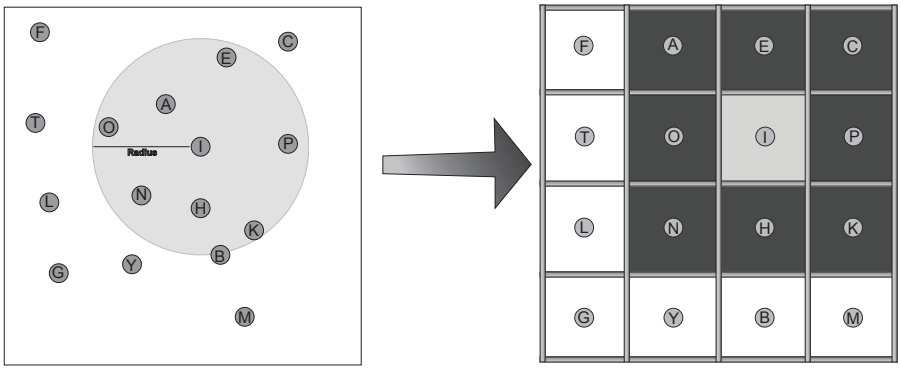

Figure 4.9: Example of the gathering of a NGrid 4x4 and fixed N-radius of 1

gathered by the algorithm. This mechanism misses a entity (B) and it gather a not used entities (C) This could be avoided by increasing the fixed-radius or by using the adaptive N-radius, presented in the next subsection.

4.5.2 Adaptive N-radius

Instead of having a fixed N-radius, the adaptive N-radius tries to adapt the N-radius in order to gather the neighbors that have a lesser distance than the radius. This is done by having an adaptive N-radius, where the value of the N-radius is adapted for each neighbor direction and it increases until all the entities inside the radius are gathered. This is done by following a simple algorithm, for every direction of the entity's neighbors on the NGrid, the algorithm increases the N-radius until it has gathered all the entities that are less than the radius distance of it.

```
Input: Grid - Entities Positions Array
Input: radius - the position radius
  for all Entities directions do
    distance = 0
    while distance <= radius do
      distance = calculate distance between entities
      Do computation with neighbor
    end while
  end for
```

Algorithm 5: The adaptive N-radius gathering algorithm

Figure 4.10 illustrates the structure that was built with the NGrid in a 2D 4×4 matrix holding the information for all 16 individual entities. The proximity queries are done by the adaptive N-radius, so each entity only gathers the information about the entities surrounding its cell, based on the Euclidean radius. In the example of Figure 4.10, the adaptive N-radius varies from 0 to 2, so the entities represented as I (in light gray) would gather the 8 highlighted surrounding cells (represented in dark gray), which are { A, E, O, P, N, H, K, B }.

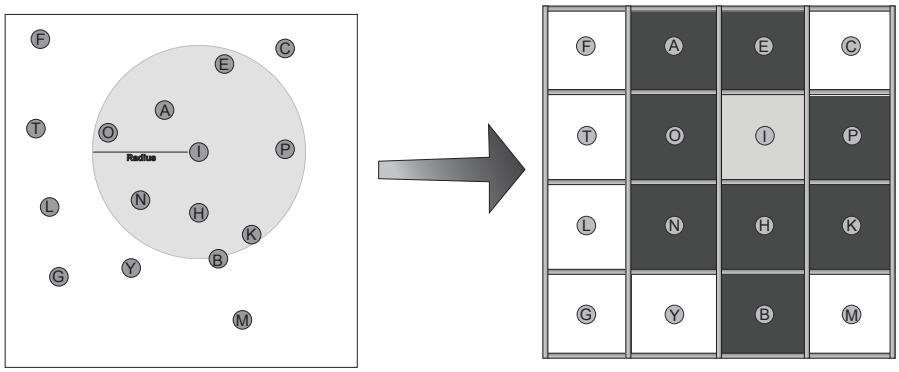

Figure 4.10: Example of the Structure of the Extended Moore Neighborhood with 16 particles and adaptive N-radius

Using this method, the NGrid gathers practically the same neighbors that the uniform grid or other traditional methods, unless the NGrid has a misaligned dimension, but since the NGrid is constantly ordered and checked the dimension, this does not happens very often.

4.6 Analysis of the NGrid

The NGrid can be divided into two stages, a construction, which only happens in the beginning of the simulation, and a maintenance stage, which happens at every step of the

simulation.

The construction of the NGrid, using the algorithm can have, in the worst case, $\sqrt[2]{n}$ changes in the dimensions, before finding the right one, having a O($n^{3/2} * logn$) (with $n * logn$ as the sorting cost). This complexity can appears to be huge, when compared to traditional approaches, which has complexity close to O(n), but these approaches have to construct its data structure every step of the simulation, while the construction of the NGrid is done only once on the beginning of the simulation.

During the simulation loop, the NGrid maintenance is done by sorting it according to each dimension, having the same complexity of the sorting mechanism. In the case of the partial Odd-even sort, it has O(n) complexity and for the bitonic sort case it has a O(n*log n) complexity. Even the check for a change on the grid dimensions is done during the sort mechanism, which does not influentiate on the complexity. Others solutions for the neighborhood gathering also requires a sorting step, like the SaP and the uniform grid, having similar complexity. Also, the construction of a quadtree/octree requires a O(n*log n) insertion.

One of the advantages of the NGrid is that it does not require the construction step during the simulation, only the maintenance step. Also the NGrid does not require any additional data for the NGrid array, since it uses the same position array, arranged according to the NGrid properties.

4.7 Summary

This chapter has presented the NGrid, the novel data structure for neighborhood gathering. Using this structure, the following of the work consists on applying it in different scenarios, which the following chapters will describe.

The next chapters will present the tests and concepts of the NGrid applied in different architectures and scenarios.

Chapter 5
NGrid on a GPU Fluid Animation

Realism in real-time graphical simulations includes the search for real behaviors and physics. Due to the graphics technology improvements, simulation of natural phenomena, such as water flows or smoke, has become possible to be performed in real time and interactive environments, like scientific applications and digital games. In fact, this kind of simulation is now quite popular in computer games, digital movies and animations. The use of aerodynamic effects and physics effects in games like races and flight simulators improves the player immersion. It is important to state that in real-time simulations, there is a trade-off between realism and interactivity, and most of the times the realistic behavior is put aside in order to have interactive frame rates.

This chapter implements a fluid animation based on the SPH presented on section 2.3.3. The NGrid is used in a dynamic 3D dimension, a fixed N-radius of 4 and the bitonic sort and the partial odd-even sort as a possible maintenance step. The uniform grid is also implemented and used in the experiments and precision of the calculus.

This chapter is divided as follows first the architecture using the NGrid is presented. Then the results are detailed and discussed. Finally the summary of the chapter is presented.

5.1 Architecture environment

The proposed architecture implements fluid simulation using a novel GPU computing solution, based on the NGrid data structure, allowing a high performance increase during simulation. This architecture environment is explained in the following subsections.

5.1.1 Execution workflow

The proposed workflow can be described as follows: at the beginning, the architecture sorts all particles according to its position. The sorted particles gather their neighborhoods according to the radius and calculate their pressure and density. Based on these results, the system calculates the internal forces at the particles and adds any external forces that may be influencing it, such as gravity and users' input. Finally the system calculates the new velocity and positions, integrating the whole system.

5.1.2 Neighborhood Gathering

Fluids are represented using a set of particles that interact with each other. This is always true for fluids' particles, as each of them needs to find its neighborhood particles for calculating variables such as pressure and density, according to the SPH method.

This chapter presents the Fluid animation using the NGrid with a dynamic 3D dimensions, a fixed N-radius of 4 and two sort mechanisms were implemented as the maintenance stage: the partial Odd-even sort and the bitonic sort.

In order to evaluate this data structure for fluid animation, this chapter has implemented the traditional uniform grid with a index sort method in the GPU with the use of CUDA. This implementation uses a uniform grid to discretize the simulated world into cells. In order to optimize this structure, especially for GPUs, these particles are sorted into an index array according to the cells index.

5.1.3 Data configuration

Different state settings that rule the fluid simulation must be updated at every frame, according to the simulation behavior. During fluid simulation, each particle has its state composed of a float4 for the position, a float4 for the forces, a float2 for density/pressure and a float4 for its velocity. These data are stored at GPU's global memory, as shown in Figure 5.1. The fluid simulation is performed entirely on the GPU, avoiding data transfers between CPU-GPU, since it is typically the bottleneck in most of the CPU-GPU systems.

In this work, the simulation of each particle is mapped to one GPU thread for both the sorting and simulation steps, so it is important to mention that the grids are double buffered; consequently each of these tasks do not write data over the input structures that can still be read by others parallel GPU threads. This work could also use atomic

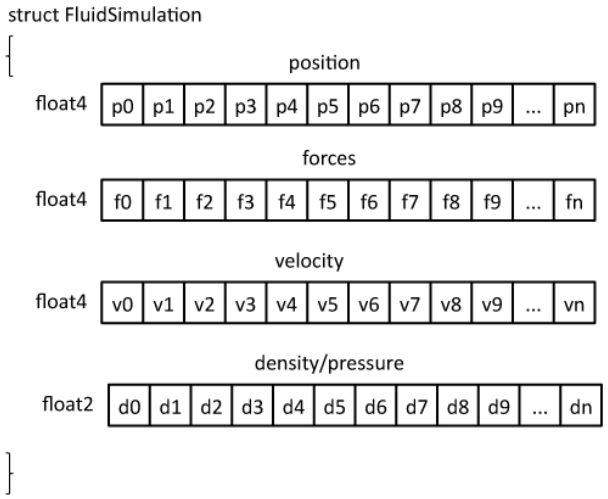

Figure 5.1: Fluid's data stored at GPU's global memory.

operations for the grid operations, but these kinds of operations are still very costly for massive simulations.

5.1.4 Density Processing

After the sorting step, it is necessary to process each particle, using its neighborhood particles information, in order to calculate the particle's density. Performing this step requires a different GPU kernel, as for calculating other fluid properties the density information must be available. Employing a grid radius of h requires traversing the three dimensions' (Z,Y and X) neighborhood and processing this neighbors according to Algorithm 6.

```
for z=-h;z<=h;z++ do
  for y=-h;y<=h;y++ do
    for x=-h;x<=h;x++ do
      if (x != 0 OR y != 0 OR z != 0) then
        Grid[indexZ][indexY][indexX].Density +=
        ComputeDensity(Grid[indexZ+z][indexY+y][indexX+x])
      end if
    end for
  end for
end for
```

Algorithm 6: Particle's density processing.

5.1.5 Force Processing

After processing each particle's density, internal and external forces are processed. In this work, internal forces are represented by pressure and viscosity, while external forces are

the gravitational forces and the user's inputs. The Algorithm 7 shows how these forces are processed.

```
for z=-h;z<=h;z++ do
  for y=-h;y<=h;y++ do
    for x=-h;x<=h;x++ do
      if (x != 0 OR y != 0 OR z != 0) then
        Grid[indexZ][indexY][indexX].Forces +=
        ComputeViscosityForce(Grid[indexZ+z][indexY+y][indexX+x])
        Grid[indexZ][indexY][indexX].Forces +=
        ComputePressureForce(Grid[indexZ+z][indexY+y][indexX+x])
      end if
    end for
  end for
end for
Grid[indexZ][indexY][indexX].Forces += GlobalExternalForce
```

Algorithm 7: Particle's forces processing.

5.1.6 Integration

After fluid forces are computed, it is necessary to integrate the particles velocity and position. This step is responsible for integrating the equations for the motion of a particle [151]. In the proposed architecture, it consists of a simple formulation, since it does not take into account the angular velocities and torque, which are not necessary for the fluid simulation. Then, it updates the particle velocity based on the forces that are applied to it, which are sent to the integrator, and then it updates the position based on its velocities, using a method based on Euler integration (this approach is one of the simplest forms of integration) using a finite time step.

Due to the fact that there are no dependencies among the particles during fluid simulation, they can be updated independently from each other. Following, internal and external forces are integrated into fluid's particle velocity and finally in its position.

It is also during this step that the collision between the particles and the bounds of the box in which reflect the velocity, of the particles that are colliding with the box's wall, in relation to the normal to the wall scaled by a damping factor d, are calculated. The Algorithm 8 illustrates this step.

```
for Each particle do
    Calculate new velocity
    Check collison with the wall
    Calculate new position
end for
```

Algorithm 8: Integration of the Particles.

5.2 Results

This section presents the results obtained from the proposed data structure. A series of screenshots of a fluid animation, built with the presented architecture applying raytracing in a set of 65k particles using pov-ray [152], can be seen in Figure 5.2.

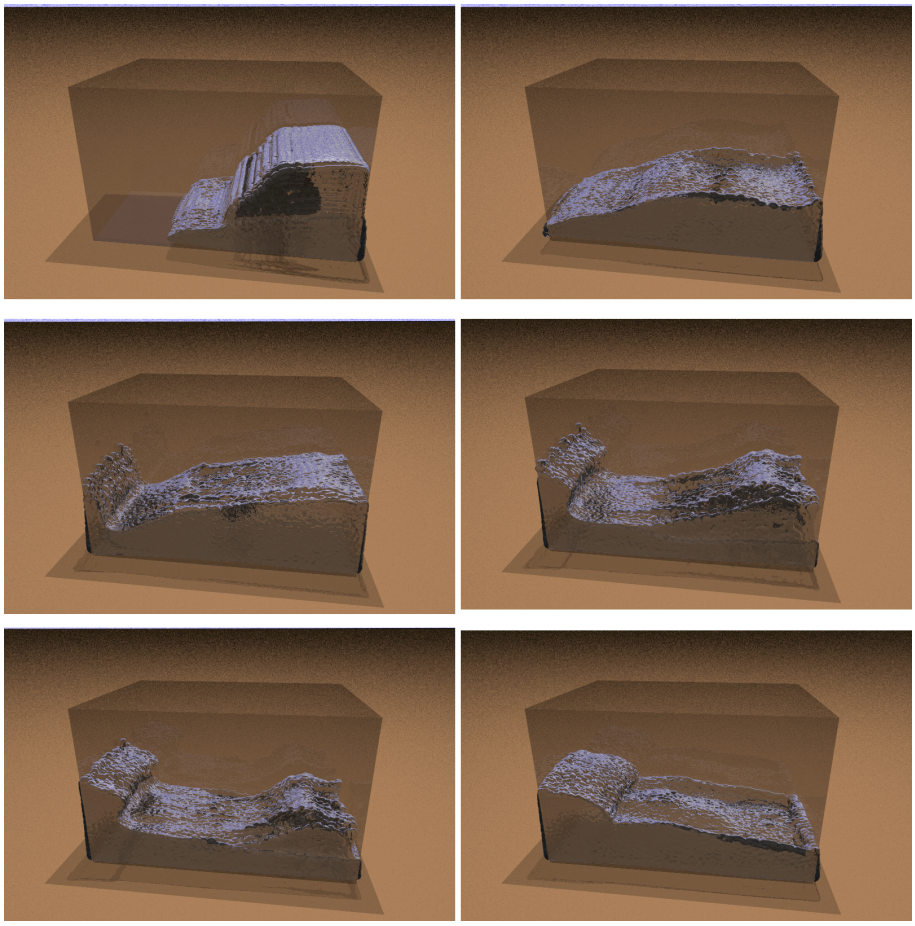

Figure 5.2: Screenshots of the simulation.

5.2.1 Simulation Configuration

For the tests, a PC with a Ubuntu 10.10 Linux distribution equipped with an Intel i7 3.07GHz using 8 GB of RAM and a NVidia GeForce GTX 580 with 512 cores were used.

Simulations tests with different configurations were performed. Fluid rendering is done in two ways: by applying raytracing in the particles using pov-ray (for Figure 5.2), and by using simple primitives (for performance tests). To assure that results are consistent, each test was repeated 10 times and the standard deviation of the average times was confirmed to be within 5%.

The test scene consists on the fluid with different number of particles, and a box with fixed size, where the fluid will be dropped. The interaction with the box is done apart from the fluid calculation, during the integration step. When the particle collide with this box, it reflects its velocity with a damping factor d. The fluid particles are updated at a fixed speed of 30 interactions per second and the rendering is processed at every frame time.The fluid is dropped in different configurations to test different initial conditions, like the classical dan-break, where the fluids are concentrated in a dam and the dam is released, and drop, where the fluids are dropped from a high spot into a receptacle.

To evaluate the scalability of the data structure, the number of particles being simulated varied from 1 thousands to 1 million. The N-radius was set to 4 (which was the radius with less mean error in experiments). The cell size of the traditional uniform grid influences the total the number of visited neighbors in each interaction, which in the tests for this work have used a size of $h = 1$ for this (which was the size with empirical better performance). The fluid particle radius are also set to 1. The fluid is also influenced by the gravity force, which were set to 9.8 m/s^2. The damping factor d was set to 0.9.

5.2.2 Test Results

Table 5.1 presents the results for the simulations using the NGrid method and the uniform grid with index sort considering a varying number of particles in both methods, all processing is done in the GPU. The label FPS represents the *frames per second* which measure the time necessary to update and render the simulation (of simple particles spheres). *Speedup* is defined by the relation $S = \frac{X_1}{Y_2}$, being X_1 the FPS for the NGrid and Y_2 the FPS for the Uniform Grid. As expected, the fluid simulation using the NGrid method presents better results than the simulation using the uniform grid with index sort.

Table 5.2 presents the results for the simulations using the NGrid method and the uniform grid with index sort considering a varying number of particles In both methods, all processing is done in the GPU. The label M represents the time elapsed for the maintenance of the neighborhood gathering algorithm (sorting the NGrid or building the uniform grid). On the other hand, the label S is the time spent in the simulation

Table 5.1: Scalability of the Simulation when using the Uniform Grid with Index Sort and the Neighborhood Grid.

# Particles	Uniform Grid With Index Sort	NGrid with Bitonic Sort		NGrid with Odd-Even Sort	
	FPS	FPS	Speedup	FPS	Speedup
1,024	703	1388	1.97	1263	1.88
4,096	571	1201	2.10	1109	1.94
16,384	377	826	2.19	788	2.09
65,536	131	384	2.95	352	2.68
262,144	30	109	3.63	89	2.96
1,048,576	3	27	9.00	22	7.33

processing (density, force and integration calculations). Here, the time for rendering the particles is not taken into account. From these results, it can be seen that the time spent with the maintenance of the NGrid is lower than the time spent building a uniform grid and maintaining it.

Table 5.2: Time in milliseconds spent with Tasks by each Gathering Method. M stands for maintenance and S stands for simulation

# Particles	Uniform Grid with Index Sort		NGrid with Bitonic Sort		NGrid with Odd-Even Sort	
	M	S	M	S	M	S
1,024	0.0429	0.71	0.015	0.23	0.016	0.23
4,096	0.1175	1.68	0.015	0.23	0.017	0.23
16,384	0.1281	6.55	0.016	0.79	0.018	0.79
65,536	0.1728	60.49	0.017	2.94	0.019	2.94
262,144	0.1891	127.33	0.018	10.84	0.022	10.84
1,048,576	0.3166	401.88	0.026	25.27	0.031	25.27

Table 5.3 and Figure 5.3 shows the memory usage for the presented data structure. From these results, it can be seen that this data structure uses much less memory space, since it does not need a lot of memory to keep the data structure in a linear form, while the uniform grid with index sort does need at least 2 MB for keeping the data structure. This data is required for the extras arrays for the index sort array and the uniform grid array in the GPU.

Table 5.4 shows the mean error and variance as a function of the number of particles keeping the radius equal to 4 (which produced the smaller error with greater performance). This mean error is calculated by processing the particles with the NGrid and comparing its positions with the one calculated by the uniform grid method (by calculating the distance between both). From these results, it can be seen that the error is small with the bitonic sort mechanism (less than 0.05, while the radius size is 1.0) as well as the

Table 5.3: Usage of memory when using the uniform grid with index sort and the NGrid.

# Particles	uniform grid with index sort	NGrid
1,024	2.10 MB	5.6 KB
4,096	2.12 MB	22.4 KB
16,384	2.19 MB	89.6 KB
65,536	2.5 MB	360 KB
262,144	3.5 MB	1.4 MB
1,048,576	7.7 MB	5.6 MB

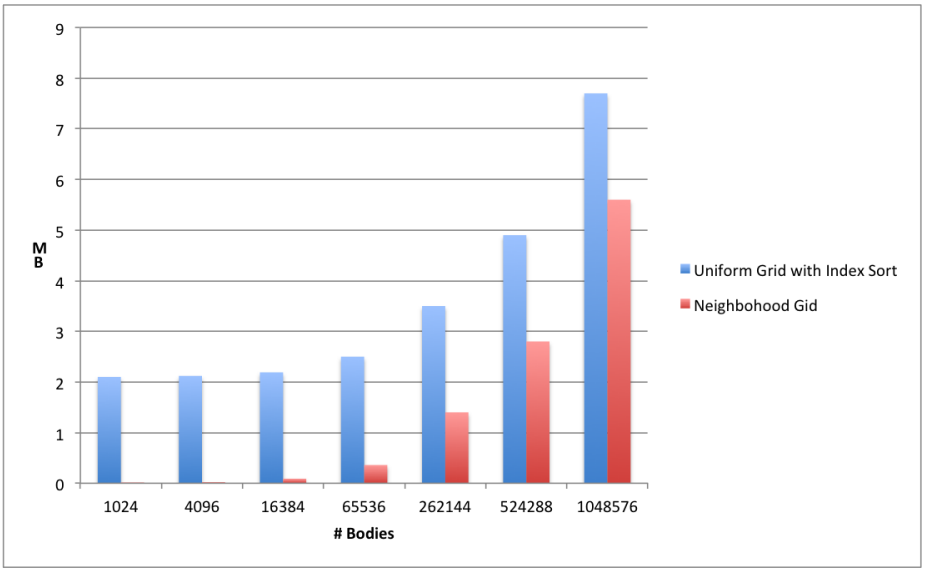

Figure 5.3: Use of Memory in the Simulation in MB.

variance. The mean error from density calculation and force steps were also measured, being not included in this work, since they follow the same evolution as the mean error included in Table 5.3, absent 10% for the density calculus error.

Figure 5.4 shows the evolution in a log scale of the performance of the simulation with the uniform grid and the NGrid with bitonic sort mechanism varying the number of particles. This graph shows that the NGrid data structure is faster for fluid simulation and that it scales better than the uniform grid with index sort.

Figure 5.5 shows the mean error of the particles position during 1000 frames of the simulation with 65k particles in the scene using the NGrid with bitonic sort. From these results it can be seen that the error does not varies considerable during the simulation, maintaining a low variance of the error.

The number of particles neighbors visited for each method was also measured, as the number of particles neighbors that are actually used in the simulation. The authors have noticed that the uniform grid with index sort visits from 100 up to 275 particles in order

Table 5.4: Erro of the NGrid in comparison with the uniform grid with different sort mechanisms.

# Particles	NGrid wit Odd Even Sort		NGrid with Bitonic Sort	
	Mean Error	Variance	Mean Error	Variance
1,024	0.0009836	0.000025	0.000642	0.000004
4,096	0.012236	0.000936	0.002437	0.000108
16,384	0.082615	0.001233	0.011948	0.000393
65,536	0.23655	0.011676	0.020246	0.000689
262,144	0.57226	0.100328	0.025120	0.001765
1,048,576	0.66419	0.169628	0.035663	0.001940

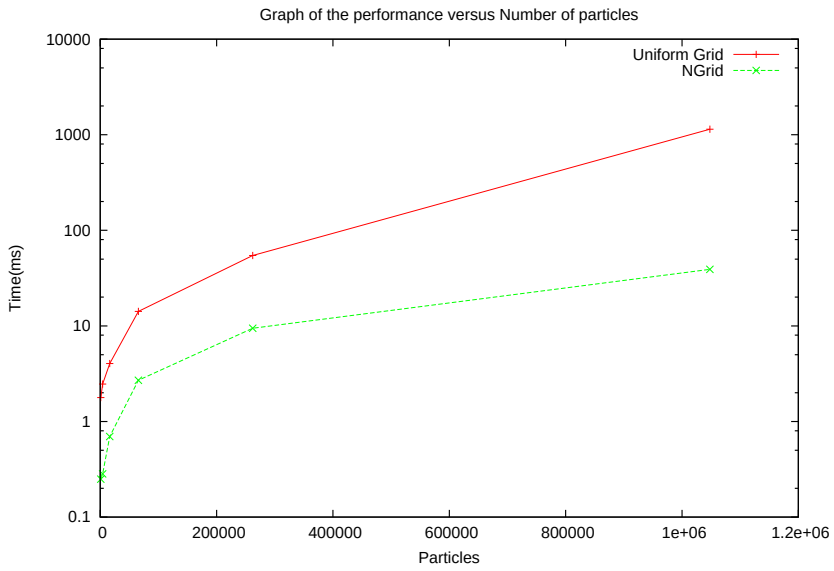

Figure 5.4: Evolution of the Simulation in FPS.

to use from 17 up to 35 particles neighbors in its calculation. While the neighborhood grid always visits 102 particles in order to use from 17 to 35 neighbors. Also a test comparing the presented architecture implemented in the GPU and the uniform grid implemented in the CPU shows a speedup of 100 times (with 100K particles).

5.2.3 Comparison to other works in the Literature

In [27], the authors show two optimizations of a parallel CPU implementation for the uniform grid with index sort and the spatial hashing, with great performance and present also a comparison between this optimization and the known methods. Although it is hard to compare to other's work, for the lack of the code and the tested architecture, the

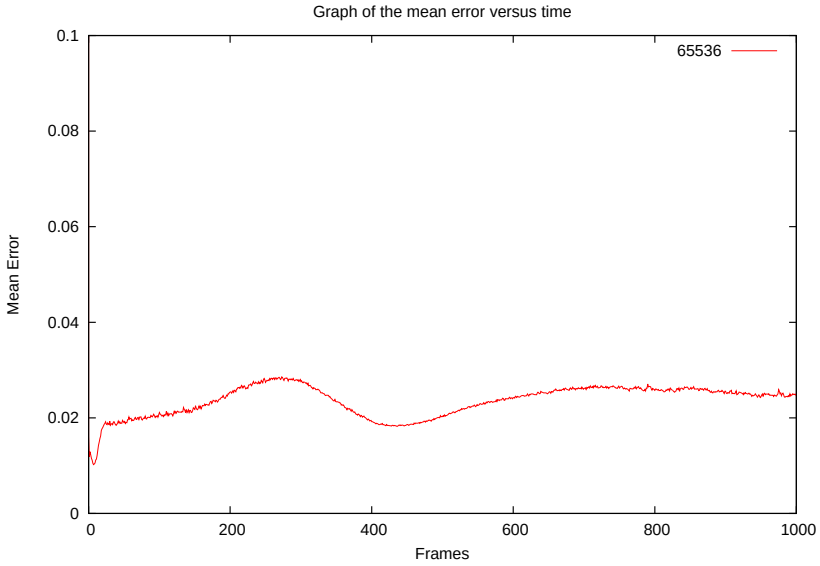

Figure 5.5: Error during the simulation of 65k particles.

simulation with the NGrid can achieve a 174 FPS rate (which consists on the processing of the neighborhood gathering, behavior processing and rendering) of 262K particles, while the work [27] has a performance of 16.5 ups (update per lab second which consists on the processing of the neighborhood gathering and behavior processing without rendering and with a fixed time step) of 170K particles.

Goswami et al [124] show an implementation of the SPH with CUDA using a z-indexing for the spatial hashing, which appears to be the fastest real time SPH implementation in the literature. Based on the z-index hash map, it can achieve better performance since it reduces the memory overhead for building the data structures. This work could achieve a performance of 10 FPS for the physical calculations of 255K particles (without the rendering) on a Geforce GTX 280 (which has 640 cores) while the present chapter can achieve 16 FPS for the same steps with 262K in a less powerful card, an nVidia Geforce GT 650M (that has only 384 cores).

The work [153] combine in one method the density and pressure force computation, which is updated with a one frame delay. Even though it doubles the performance, it still has the uniform grid cost, which is higher than the NGrid method. And also by using this technique the simulation can become unstable [154].

5.3 Summary

This chapter has presented a fluid animation using the NGrid and its comparison with the uniform grid. The NGrid has proved to have a nice speedup with a small error for fluid animations. Even though this structure does not fulfil all the steps of a SPH fluid animation, it is still good enough for animation of fluids, since it has a small error and the behaviour of the fluids appears to the human eye similar to the SPH implementation.

This chapter used two sorting mechanisms with the NGrid, the bitonic sort and a partial odd-even sort. The bitonic sort has a much better speedup and a small amount of error when compared to the partial odd-even. Even with that results, the NGrid with odd-even implementation is still important for architectures that can not implement a full sorting mechanism, like the mobile architect presented in chapter 7.

The next chapter will be showing crowd simulation implementation and tests using the NGrid.

Chapter 6
NGrid on a Multi-GPU Crowd Simulation

In a typical natural environment it is common to find a huge number of animals, plants and small dynamic particles. This is also the case of other densely populated systems, such as sport arenas, communities of ants, bees and other insects, or even streams of blood cells in our circulatory system. Computer simulations of these systems usually present a very limited number of independent entities, mostly with very predictable behavior. There are several approaches that aim to include more realistic behavioral models for crowd simulation as in [77, 81, 85, 82, 83, 84]. All these models are based on the flocking boids approach [77], which also fundaments this work. While high end games traditionally use crowd environments, due to its high end hardware resources, mobile games avoid them.

Algorithms for flocking simulation are driven by the need to avoid the $O(n^2)$ complexity of the proximity queries between entities. Several approaches have been proposed to cope with this issue [111, 118, 155] but none of them has reached an ideal level of scalability. As far as this thesis verified, no work until the present date has proposed a real time simulation of more than just a few hundreds thousands of complex entities interacting with each other in real-time.

This chapter implements a flocking animation based on the Reynolds rules presented on [111] and Section 2.3.2. The NGrid is used in dynamic 3D dimension, a dynamic N-radius and a fixed N-radius of 4 are used. The bitonic sort is implemented as the maintenance step. The uniform grid is also implemented and used to evaluate the performance and the calculus precision. This chapter has also implemented the NGrid simulation on a Multi-GPU environment to test its performance and scalability in a different architecture.

This chapter is divided as follows: first the architecture using the NGrid is presented. Then the results are presented and discussed, including implementations in multiple GPUs. Finally the summary of the chapter is presented.

6.1 Multiples GPUs architecture

The crowd entities are represented using a collection of particles that interact with each according to the boids rules. This interaction needs to be performed at each step of the simulation, and each entity needs to find its neighborhood particles for calculating its behavior. The crowd simulation is divided in three steps, a maintenance, which will construct/maintain the neighborhood gathering mechanism (in this case the NGrid); the simulation, which applies the boid rules for each entity; and the rendering, which render the boids on the screen.

The strategy used in this work for allowing more than one GPU to process entities is to split up the crowd domain among the available GPUs. In this case, the entities located in the boarder of the NGrid are processed differently from the one inside the grid. In this case, entities, which are not on the edge of the NGrid, can be processed normally, as all the dependency data is available on the same GPU. However, entities that are located on the edge of the NGrid cannot be processed, as some of its dependency data are located on another GPU's memory. This required a data transferring using the NVidia GPU Compute Direct with CUDA language [156], which allows Peer to Peer (P2P) communication to share the memory that enables a single view of the whole GPUs memory in the host.

Processing crowd simulation using a set of GPUs is done by a collection of ordered tasks, maintenance, simulation and rendering. The maintenance requires much less computation power than the simulation and the rendering, as presented in the fluid tests on Table 5.2. Since the maintenance does not require much processing power and would require a more complex synchronization mechanism, it will not be distributed among the GPUs. Also the rendering step must be done in the GPU connected with the video monitor, so it is not distributed as well. The outline process is highlighted in Algorithm 9.

6.2 Results

In this work, we implemented and tested the flocking boids case-study using the NGrid and also evaluated the rendering of all boids. The rendering consists of a simple display list that is repeated for each entity/boid using the position and orientation information gathered from a texture that is bound from the output VBO of the CUDA simulation in

```
Input: Grid - Entities Positions Array
Input: numGPUS - the number of available GPUs on the system
  Sort the NGrid according to the X axis
  Sort the NGrid according to the Y axis
  Sort the NGrid according to the Z axis
  for all available GPU do
    share the Grid data
    Process the boid rules
  end for
  Synchronize
  Render the entities
```

Algorithm 9: Algorithm describing the high level steps during the simulation for various GPUs.

a vertex shader as can be seen on Figure 6.1.

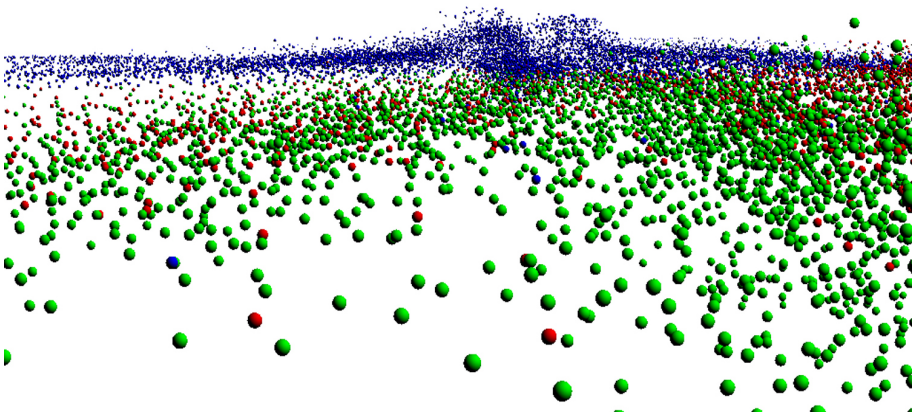

Figure 6.1: Simulation with 32K boids.

For the tests, a PC with a Ubuntu 10.10 Linux distribution equipped with an Intel i7 3.07GHz using 8 GB of RAM and two NVidia GeForce GTX 580 with 512 cores were used. Simulations tests with different configurations were performed. To assure that results are consistent, each test was repeated 10 times and the standard deviation of the average times was confirmed to be within 5%.

In order to better evaluate the NGrid in a crowd simulation, this work has divided the test in two subsections. The first dedicated to the NGrid in a single GPU, and the second to the multi-GPU implementation.

6.2.1 Simulation using a single GPU

The tests with single GPU are designed specifically to show the performance of the NGrid in a crowd simulation and to show the error test with the use of the adaptive N-radius. The same machine was used, with only one GPU doing the processing and rendering of the simulation. To evaluate the scalability of the architecture, this work has varied the number of entities/boids being simulated (from 1 thousand to 1 million). The N-radius used was set to 4 and also the adaptive N-radius was used, both of them using the bitonic sort as the maintenance stage. The cell size of the traditional uniform grid influences the total the number of visited neighbors in each interaction, and in the tests this work have used a size of $h = 1$ for this (which was the size with better performance). At preliminary tests, it has been observed that the number of different boid types had no observable influence on the performance, so a fixed number of 4 types was used for all experiments.

Table 6.1 presents the results for the simulations using the NGrid method and the uniform grid with index sort considering a varying number of entities in both methods, all processing is done in the GPU. The label FPS represents the *frames per second* which measure the time necessary to update and render the simulation (of simple particles spheres). *Speedup* is defined by the relation $S = \frac{X_1}{Y_2}$, being X_1 the FPS for the NGrid and Y_2 the FPS for the Uniform Grid. As expected, the crowd simulation using the NGrid method presents better results than the simulation using the uniform grid with index sort. Also, it can be seen that the time spent using the adaptive is higher, since it will visit more neighbors, but it is gives a nice speedup over the uniform grid.

Table 6.1: Scalability of the Simulation when using the Uniform Grid with Index Sort and the Neighborhood Grid with Bitonic Sort.

# Particles	Uniform Grid With Index Sort	NGrid with Adaptative N-radius		NGrid with N-radius 4	
	FPS	FPS	Speedup	FPS	Speedup
1,024	1344	2784	2.07	2995	2.27
4,096	1103	2514	2.28	2692	2.44
16,384	588	1492	2.53	1690	2.87
65,536	211	561	2.66	656	3.11
262,144	38	166	4.36	197	5.11
1,048,576	4	41	10.25	47	11.75

Figure 6.2 shows the evolution in a log scale of the performance of the boid simulation with the uniform grid and the NGrid with bitonic sort mechanism using the adaptive N-radius and a N-radius of 4 varying the number of boids. This graph shows that the NGrid data structure is faster for crowd simulation and that it scales better than the uniform

grid with index sort. Also can be seen that the adaptive N-radius is slower than the fixed N-radius, since it can potentially gather more neighbors, but this adaptive N-radius can also decrease the error on the simulation as Table 6.2 will show.

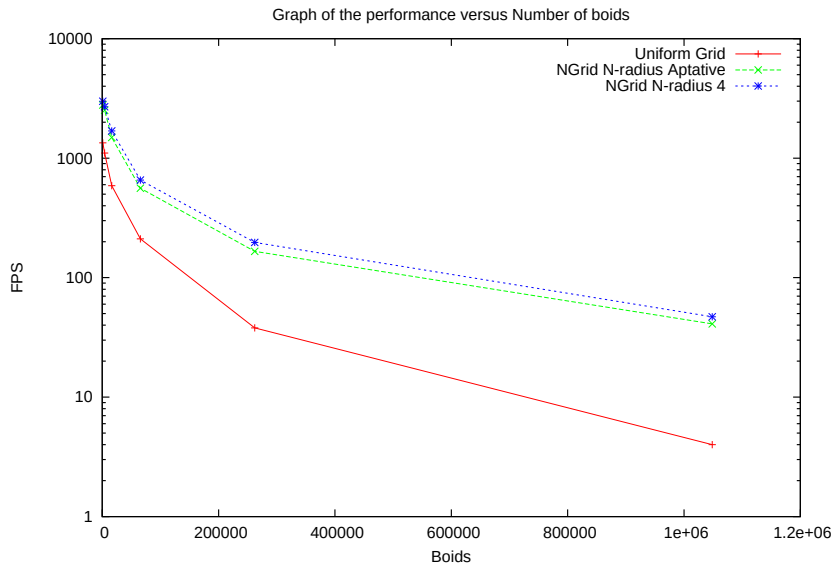

Figure 6.2: Evolution of the performance of the simulation in a log scale.

Table 6.2 shows the mean error and variance as a function of the number of entities. This mean error is calculated by processing the entities with the NGrid (with the fixed N-radius of 4 and the adaptive N-radius) and comparing its positions with the one calculated by the uniform grid method (by calculating the distance between both). From these results, it can be seen that the error is smaller with the adaptive N-radius as well as the variance.

Table 6.2: Error of the NGrid with different sort mechanisms.

# Particles	NGrid wit N-radius 4		NGrid with Adaptative N-radius	
	Mean Error	Variance	Mean Error	Variance
1,024	0.0008876	0.000022	0.000461	0.000002
4,096	0.0093627	0.000721	0.001956	0.000055
16,384	0.077372	0.009801	0.008438	0.000199
65,536	0.201974	0.009369	0.011952	0.000401
262,144	0.509362	0.089647	0.019449	0.000936
1,048,576	0.611983	0.123675	0.027566	0.001022

Figure 6.3 shows the mean error of the particles position during 1000 frames of the simulation with 65k particles in the scene using the NGrid with bitonic sort and adaptive

N-radius. From these results it can be seen that the error does not varies considerable during the simulation, maintaining a low variance of the error. This means that the NGrid can be used for animations of crowds, since the error is small.

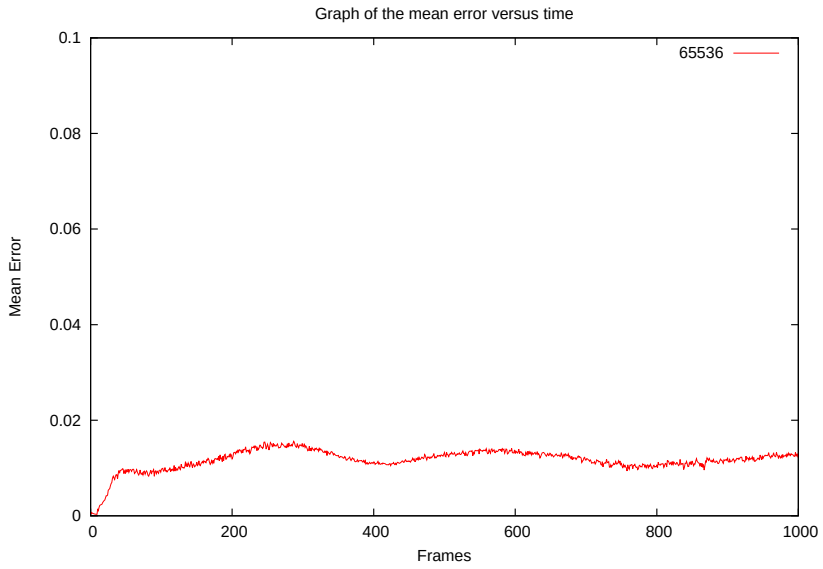

Figure 6.3: Error Evolution of 65k boids simulation.

6.2.2 Simulation with Multi-GPU

The tests with dual GPU were designed specifically to show the speedup of using more than one GPU and also to show the adaptability of the NGrid on different architectures. Both of the GPUs are used, one for maintenance, simulation and rendering, and the other for simulation only. To evaluate the scalability of the architecture, this work has varied the number of entities/boids being simulated (from 4 thousand to 1 million). The N-radius used was the adaptive N-radius using the bitonic sort as the maintenance stage.

Table 6.3 shows the results of using a single and dual-GPU for processing the simulation. According to the presented result, it is possible to see that using more than one GPU has increased the overall performance of the simulation using the NGrid. This speedup is mostly because of the increased number of available processors.

Figure 6.4 shows the evolution in a log scale of the performance of the boid simulation with the single and dual gnu system varying the number of particles. This graph shows that the NGrid data structure is faster and that it scales well in a multi-GPU system.

Table 6.3: Scalability of the Simulation

# Boids	Single-GPU FPS	Multi-GPU FPS	Performance Gain
4,096	2514	2564	1.02
16,384	1492	1641	1.10
65,536	561	701	1.25
262,144	166	237	1.43
1,048,576	41	66	1.61

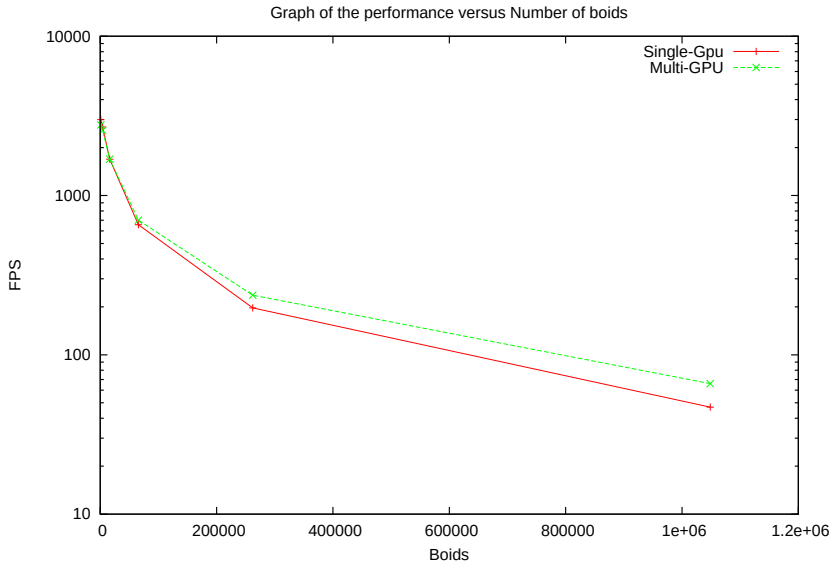

Figure 6.4: Evolution of the simulation with one and two Gpus.

6.3 Summary

This chapter has presented the NGrid applied in a crowd scenario, implemented using the adaptive N-radius, using multiples GPUs. The next chapter presents the NGrid applied in a mobile architecture.

Chapter 7
NGrid on a Mobile GPU Particle System

Mobile devices, like smartphone and tablets, and Digital TVs have many others constraints [157], when compared to PC, like: hardware constraints (processing power and screen size); user input, (buttons, voice, touch screen and accelerometers); and different operating systems, like Android, BB, iPhone OS, Symbian and Windows Mobile. Developing and porting algorithms for this kind of device must take these issues into account.

During the past few years, mobile phones and other mobile devices evolved from simple phone and messaging devices to high end smartphones with serious computing capabilities. Nowadays, most of these devices are equipped with multicore processors like dual- core CPUs and GPUs, which are designed for both low power consumption and high performance computation. Moreover, most devices still lack libraries for generic multicore computing usage, such as CUDA or OpenCL, but they still have the shader abilities. For the Android specific case, there is the Renderscript, which is a API for computation. However, computing certain specific kind of tasks in these mobile GPUs, and other available multicores processors may be faster and much more efficient than their single threaded CPU counterparts. This chapter shows that the NGrid proposal scales well and fits this kind of architecture.

Google introduced in the Honeycomb version of Android the Renderscript API (application programming interface) [158]. Renderscript is an API for achieving better performance on Android phones and tablets. Using this API, applications can use the same code to run on different hardware architectures like different CPUs (Central Processing Unity), ARM (Advanced RISC Machine) v5, ARM v7, and X86, GPUs (Graphic Processing Unit) and DSPs (Digital Signal Processors). The API decides which processor will run the code on the device at runtime, choosing the best processor for the available code. This chapter models the NGrid data structure adapted to make it suitable for this new

architecture and compares it to the traditional brute force algorithm.

There are no works on the literature that deals with the use of renderscript, like presented in this work [17], but the Android SDK [158] makes available a series of sample codes, for building simple animations based on particle systems, like a fountain and a brute force physics simulation that can render and process up to 900 interacting particles.

This chapter is divided as follows: first some basic concepts of the Rendescript API are presented. Then the architecture of the simulation is shown. The tests are presented in section 7.4 and finally in section 7.5 discusses the the summary of the chapter.

7.1 The Renderscript API

Renderscript is a new software development kit and API for Android firstly introduced by Google in the Honeycomb version of Android. Renderscript is an API for high-performance processing on Android phones and tablets. It is used for fast 3D rendering and computing processing, having similar paradigm as GPU computing libraries and frameworks [56]. The main goal of Renderscript API is to bring a lower level, higher performance API to Android developers, in order to achieve better performance in visual animations and simulations [57].

The API was mainly design for portability, performance and usability [159]:

- Portability: it is possible to have the same code being able to run on different architectures like different CPUs, ARM v5, ARM v7, and X86, GPUs and even Digital signal processors (DSP). With that, the developer can have access to all of the API features without having to write code to support different architectures or a different amount of processing cores.
- Performance: the main goal of the architecture is to achieve a better performance on the available processors. This performance gain comes from executing the code natively on the device.
- Usability: the API has a developer friendly compute API similar to CUDA, and a familiar language in C99.

Renderscript code is compiled on the device at runtime, so the developer does not need to recompile the application for different processor types, making easier its usage [159].

Its language is an extension of the C99 language that is translated to an intermediate code at compile time, and then to machine code at runtime. The API scale the generated code to the amount of processing cores available on the device. The decision of choosing which processor will run the code is made on the device at runtime, being completely transparent for the developer. Normally simple scripts will be able to run on the available GPUs, while more complex scripts will run on the CPU. The CPU is also a fallback, so that if none other available suitable device, it will run the code.

All the tasks implemented in Renderscript are automatic portable for parallel processing on the available processors of the device, like the CPU, GPU and even DSP. Renderscript is especially useful for apps that do image processing, mathematical modeling, or any operations that require lots of mathematical computation, similar to GPU computing paradigm. The main use of renderscript is to gain performance in critical code where the traditional Android framework and OpenGL ES APIs are not fast enough.

The Renderscript is composed of two APIs: a computing API (responsible for processing the computation), and a rendering API (responsible for the renderization of the scene, working together with OpenGl ES 2.0). The Renderscript code is called from an Android Activity inside the virtual machine. If the code can execute on a GPU or on a multi-core CPU, it may be assigned to run on that. The script runs asynchronously and sends its results back into the Virtual Machine.

The renderscript files are defined with a .rs extension. Android build tools compile the Renderscript .rs file to intermediate byte-code, and package it inside the application's .apk file. On the device, the byte-code is compiled (while loading the application) to machine code that is optimized for the exact device that it will use for processing. This machine code is cached for future use, so it is only compiled in the first run. This eliminates the need to target a specific architecture during the development process, while speeding up the application processing.

In order to have a particle system, with the NGrid, processed using the Renderscript, a specific architecture is needed, which is present in the next section.

7.2 The Particle System

The simulated particle system for validating this proposal at a low profile computing is a simple newtonian particle system, where each particle is a sphere. The system has some physics properties, like collision detection, and gravity. The input of gravity is made by

the device accelerometer, and some collision input is made also though touch input. There are three main steps to the performing the simulation: integration, maintenance of the NGrid and processing the collisions.

The integration is a simple step. It integrates the particle position and velocity in order to move the particles though space. In this work, the Euler integration is used. The velocity is updated accordantly to the applied forces and the gravity force, and then the position is updated accordantly to the calculated velocity. Also the particles interaction with the bounding walls of the screen are applied during this integration scheme.

The maintenance of the NGrid is done by a sorting step. In this case the parallel odd-even sorting strategy is used, as explained in section 4.5.1.

The simple collisions between particles are done using the DEM method [160]. This collision model consist of diverse forces, including a dashpot force, which causes damping, and a spring force, which are the forces that the particles move apart.

7.3 Architecture

As explained in Chapter 4, the NGrid can have different configurations. Since the mobile devices are simpler and have more constraints, for this work the NGrid is used with the following reduced properties, a 2D fixed dimension, a fixed N-radius of 4 and a partial odd-even sort.

In order to fully operate at the mobile device, this simulation tasks are divided in four different ambients, which are illustrated on Figure 7.1:

- the Android framework, where the application is created, and the renderscript context is also created. Besides, this ambient is responsible for gathering the inputs and to sent it to the computing renderscript;
- the computing renderscript is where the variables for the simulation are created, the call for the renderscript computing engine are made and the sort the entities is done;
- the renderscript computing engine will process the behavior of the scene distributing its process among the available processors;
- OpenGL: will render all the objects, applying shaders and visual effects to them.

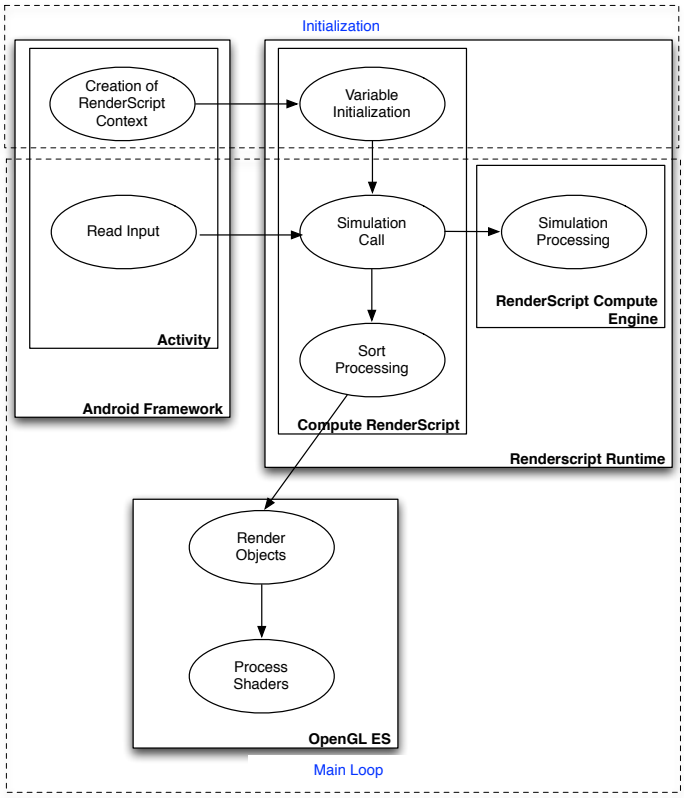

Figure 7.1: Architecture Overview.

This architecture works in two main steps: the initialization, and the main loop. In the initialization step, the Android Framework creates all the context, the necessary data structures for communication between the Android framework and the Renderscript, and the data structures needed for the computation. Then, the compute Renderscript initializes all the variables, like the particles properties, initial position, initial velocity and size.

The main loop step is repeated during the entire simulation and has the following sequence of tasks, first the input, coming from the accelerometer and touch, is gathered from the Android framework and sent to the renderscript to process accordantly. Then the simulation is processed, i.e. the collision is processed according to the NGrid neighbors and the integration is processed. Afterwards, the maintenance of the NGrid is done by the sorting step. All these updated positions are put on a VBO (Vertex Buffer Object) and sent to a vertex shader in order to render the individuals without using the Activity. The the visualization part is done by the OpenGL ES, by rendering the particles and applying the shaders.

7.4 Results

In this chapter, the particle system is implemented and evaluated, the evaluation includes the rendering of all particles. The rendering consists of a simple primitive of a sphere representing each particle, which is bound from the output VBO (vertex buffer object) of the simulation in a vertex shader. A screenshot from the simulation can be seen on Figure 7.2.

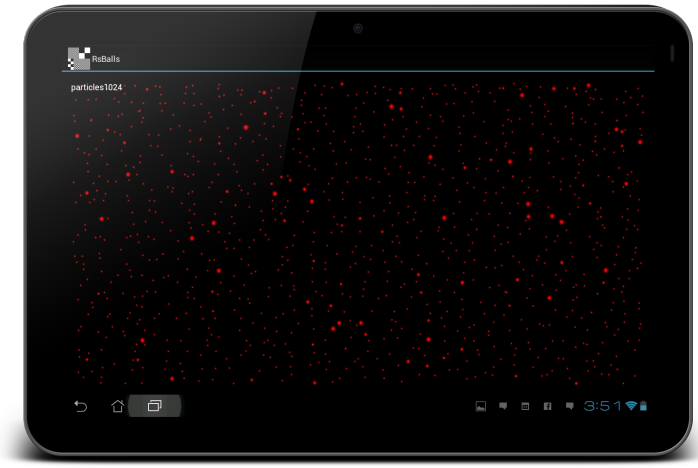

Figure 7.2: Screenshot of the simulation with 1024 particles on a Android Tablet.

In order to evaluate the performance of the NGrid in mobile devices, this work has implemented also the brute force algorithm, which has a $O(n^2)$ complexity. Since the Renderscript is very new and have some scatter constraints, there are lack of spatial subdivisions techniques implemented in this technology, so most works uses the brute force algorithm.

For the tests, we have used a Asus Tranformer TF101, which is a 10.1 inches tablet with an Android 4.03 operating system that has a Nvidia Tegra 2 T20 chipset with a Dual-core 1 GHz Cortex-A9 CPU and a ULP (ultra-low power) GeForce GPU and 1GB RAM memory. Simulations tests with different configurations were performed. To assure that results are consistent, each test was repeated 10 times and the standard deviation of the average times was confirmed to be within 5 %.

Table 7.1 show the results of different simulation configurations, by varying the number of boids in the scene. In these results, the label FPS represents the *frames per second* which measure a time necessary to update and render the simulation. *Speedup* is defined by the relation $S = \frac{X_1}{Y_2}$, being X_1 the FPS for the NGrid and Y_2 the FPS for the Brute

Force algorithm.

Table 7.1: Scalability of the Simulation.

# Particles	Brute Force FPS	NGrid FPS	Performance Gain
64	585	604	1.03
128	290	444	1.53
256	221	351	1.59
512	151	245	1.62
1024	88	180	2.04
2048	28	116	4.14
4096	5	94	18.8
8192	2	76	38
16384	0.8	44	55

From these results, as expected, the simulation using the method implemented with the NGrid presents a better result than the simulation using the method based at the brute force algorithm. These tests also shows that even with more that 16k interactive particles in the scene, the simulation with NGrid can still maintain the interaction, since it maintain the lower bound for interaction [35] (higher than 16FPS).

7.5 Summary

This chapter has presented the NGrid used in a particle system implemented in a mobile architecture with the use of Renderscript.

The next chapters will present the implementation and tests of a game using the NGrid on a GPU architecture.

Chapter 8
NGrid on a GPU Game

Computers, new video game consoles (such as the Microsoft Xbox 360 and the Sony Playstation 3) and GPUs feature multi-core processors. For this reason, putting the game tasks isn parallel is getting crucial for performance gain As a proof of concept, this chapter shows the NGrid applied in a game with its tasks execution in parallel, with the sequential execution kept to a minimum.

The development of programmable GPUs has enabled new possibilities for general purpose computation (GPGPU) which can be used to enhance the level of realism of virtual simulations. Many games and applications that uses GPGPU to process some parts of its tasks in the GPU and another on the CPU. This may bring limitation on the simulation, because it requires a lot of data transfers between the CPU and GPU, and this can be the bottleneck of the simulation. This chapter implements all the methods of the game entirely on the GPU with the use of CUDA architecture keeping the GPU-CPU communication to a minimum. In order to process the physics and AI steps some sort of neighborhood gathering method is needed. This chapter validates the NGrid in a game context, with an adaptive N-radius and a bitonic sort as the maintenance stage.

This chapter is divided as follows, first the game design and requirements of the test case is presented. Then, the architecture used by the game is described. Finally the results are presented followed by the chapter summary.

8.1 GpuWars Game Design

The GpuWars is a massive 2D game prototype shooter with a top-down 2D perspective. The game is similar to 2D shooters like Geometric Wars [161] and E4 [162]. The main enhancements of GPUWars is that it uses GPU to process its calculations, allowing to process and render thousands of enemies, while similar games only process hundreds.

The game play is very simple: the player plays as a GPU card (which is called

"GPUship") inside the "computer universe", and he needs to process (by shooting them) polygons, shaders and data (the enemies) from a game. Every time the "GPUship" make physical contact with an enemy it looses time and as a consequence, looses FPS. The objective is to process the maximum number of data in the smaller amount of time, and keep the game interactive with a minimum 12 frames per second.

The GpuWars uses the keyboard as the input device, where one set of controls are used to control the movement of the "GPUship" and another set to control the direction of the shots. The shots are implemented as physical entities and the enemies are simulated as physical-behavior entities.

8.2 The Architecture

Computer games are multimedia applications that employ knowledge of many different fields, such as Computer Graphics, Artificial Intelligence, Physics, Network and others [163]. The game loops are the underlying structure that games and real time simulations are built upon. These loops are regarded as real-time because games and simulations (and similar kinds of multimedia applications) have time constraints to run the tasks that rely on them. This means that if those tasks do not run fast enough, the experience that the application must provide will be compromised.

Most GPGPU works uses a sequential game loop called Single-Thread Game Loop with a GPGPU Stage [37], which processes some of the data processing tasks of the game loop inside the GPU. Figure 8.1 illustrates this game loop. There are some variations of the same loop by putting the tasks in multi-threats [164].

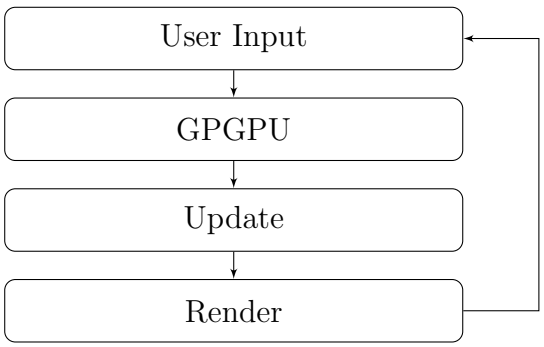

Figure 8.1: Single-Thread Game Loop with a GPGPU Stage.

The game loop of this architecture differs from those works, since it does not have an update stage in the CPU. This loop work as follows: First the CPU gathers the input and sends it to the GPU. The GPU treat this data, making the necessary adjustments,

like the transformation of the players' position and the creation of the players' shots. The GPU starts updating the bodies by applying the physics update behavior and their logic behavior, which corresponds to the artificial intelligence step. These updates are put on a VBO (Vertex Buffer Object) and sent to the shaders for rendering. The GPU also sends variables to the CPU in order to execute sound effects and to tell if it should terminate the application. This game loop is illustrated in figure 8.2.

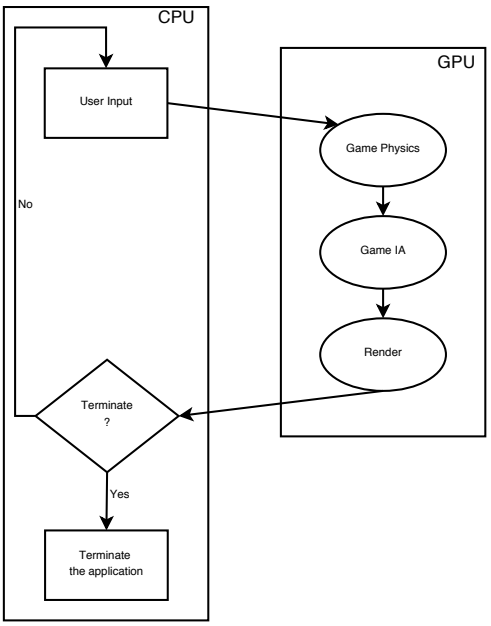

Figure 8.2: Game Loop of Architecture

To resume, the CPU is responsible for:

- Creating a window;
- Gathering the users input and sending it to the GPU;
- Making the GPU calls;
- Executing the music and sound effects;
- Terminating the simulation/application/game, i.e., destroying the window and releasing the data.

While the GPU is responsible for:

- Applying the physics on the bodies;
- Processing the artificial intelligence;

- Determining the game status, like the player scores;
- Determining the end of the simulation/application/game.

The data that is exchanged between the CPU and GPU, which normallyy happen every frame, is encapsulate in a special structure, in order to keep the communication between the CPU and the GPU to a minimum, since this process can be a bottleneck of any simulation that has communication between CPU and GPU [165]. In order to implement this architecture some data structure is needed. The description of the data required for each entity follows:

- One vector (x,y,z) with the entity position;
- One vector (x,y,z) with the entity force;
- One vector (x,y,z) with the entity direction/orientation;
- One float as the entity type;
- One float with the entity energy;
- One float for the entity mass.

These data are grouped together into three vectors of float4 in order to optimize the data exchange between the GPU executions. To assure the desired high performance, all information, whenever possible, are organized and mapped as textures, using a ping-pong strategy.

Random numbers are used in games to avoid deterministic behavior by its entities. In the proposed architecture, random numbers are used in order to determinate random behavior of the entities (the creation of new entities, the initial status of these entities and the actions of the entities). Since most GPUs do not have native pseudo random number generation, this work developed a pseudo-random number generation based on a nVidia work [166].

GPGPU programs are divided into threads. In order to process the main game logic that needs to be executed sequentially, the proposed architecture has a special GPU thread, which is responsible for it, and is the same that treats the user inputs. This processing includes tasks that: updates the simulation according to the users input, i.e. treats the input; creates new entities, if necessary (which are created in other GPU threads);

determines the scores (in case that the simulation is a game); determines the game over or the end of the simulation. The others threads are responsible for updating the entities, like collision detection and response, and the entities behavior.

There are three types of entities simulated by this work: physical entity, behavior entity and physical-behavior entity. The physical entity will only simulate the physical aspects of an entity related to collisions. The behavior entity will only simulate the behavior of the entity and will not simulate the collisions. The physical-behavior entity will simulate both physical and behavioral aspects.

The positions and type of all entities are gathered into a VBO (Vertex Buffer Object) and sent to a vertex shader in order to render the entities without using the CPU. In order to deal with the creation of the entities, the architecture keeps a list with the values to indicate available positions for entities creation.

Using this structure, the GPU processes some empty threads (threads that practically do not process anything), and also threads running different codes, which can affect the general performance, because of the threads synchronization mechanism inside the GPU block. Here is proposed the usage of the NGrid structure, which during its' sort stage groups the empty threads together. Figure 8.3 illustrates the process of the different threads.

The proposed architecture is built in a way such that it can be used, with proper modifications, for both 3D games and GPGPU particle simulation. It was implemented using the following technology: CUDA [**?**] for GPGPU processing; OpenGL for rendering; GLSL (OpenGL Shading Language) for shaders; and GLUT (OpenGL Utility Toolkit) for window creation and input gathering. But the concepts presented here could also be adapted to others technologies. In the next sections shows how the NGrid is applied to the physics and the AI step.

8.3 Physics Step

This step is responsible for the physics behaviour, i.e. how the bodies process and resolve all bodies collisions and response. The physics of this architecture is based on the physics on particle systems [110, 167, 168] and in a hybrid physics engine[37].

Collision detection is a complex operation. Normally, to reduce this computation cost, this task is performed in two steps: first, the broad phase, and second, the narrow phase.

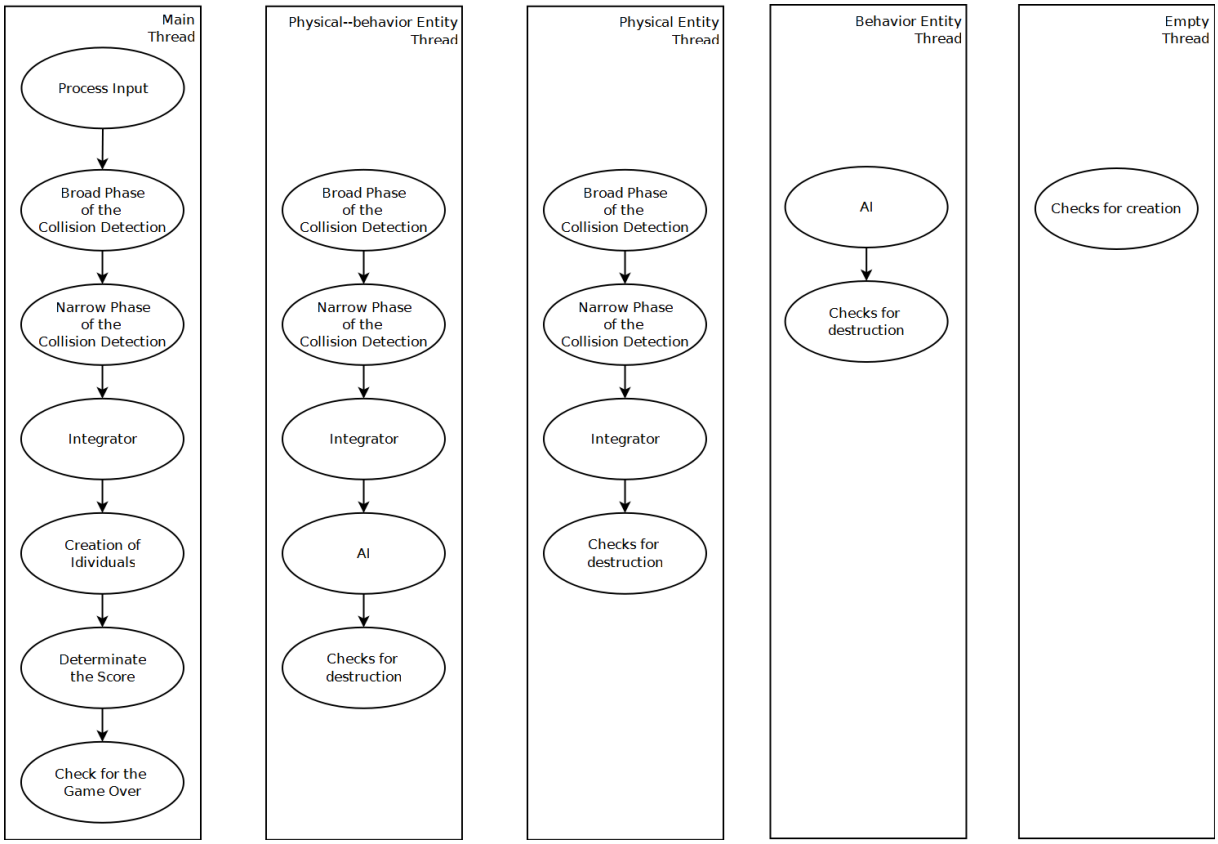

Figure 8.3: The Different Process of the Architecture Threads.

In the broad phase, the collision library detects which bodies have a chance of colliding among themselves. In the narrow phase, a more refined algorithm to do the collisions tests are performed between the pairs of bodies that passed by the broad phase. This work implements as a broad phase neighborhood gathering using the NGrid with 2D fixed dimension, an adaptive N-radius and a bitonic sort as the maintenance stage.

The physics step is responsible for:

- Executing the broad phase of the collision detection, using the NGrid;
- Executing the narrow phase of the collision detection, i.e. applying the collision test in each of the previously approved body pair;
- Forwarding the simulation step for each body by computing the new position and velocity according to the forces and the time step, i.e., solving the motion equations;

8.3.1 The narrow phase of the collision detection

The narrow phase of the collision detection is responsible for doing the collision detection among the rigid bodies. In this work, instead of doing the collision check between all the polygons of the entities, it is implemented a basic primitive area element, that complex models are put inside. The bounds are used to surround every model, simplifying the narrow phase of the collision detection. Two types of bounds were implemented: a circle bound and a bounding rectangle. The circle bound is used whenever it is possible. This is done in order to save memory, since the circle bound only needs the position vector and a radius, while the bounding rectangle needs four variables.

8.3.2 The Integrator

This method is responsible for integrating the equations of motion of a rigid body [151]. In the proposed architecture, it consists of a simple formulation; since it does not take into account the angular velocities and torque. This method updates crowd entity velocity based on the forces that are applied to it, which are sent to the integrator, and then it updates the position based on its velocities, using an integration method based on Euler integration (this type of integration is one of the simplest forms of integration). Mathematically, it evaluates the derivative of a function at a certain time, and linearly extrapolate based on that derivative to the next time step.

8.4 AI Step

Game Artificial Intelligence (AI) is used to produce the illusion of intelligence in the behavior of non-player characters (NPC), as, for instance, in the case of the text case GpuWars, of the enemies. The algorithms used for Game AI are typically built upon know methods from the Artificial Intelligence field, but game AI focus more on the gameplay instead of precision. Besides, game AI has more computational constraints then pure AI applications, since the game needs also to process the game physics and to render of the results.

There are several ways to implement the game AI, such as Finite State Machines (FSM), fuzzy logic, neural networks, and many other [169]. This work uses Finite State Machine. FSMs are powerful tools used in many parts of computer games [170, 171, 172], like the NPC behavior, the characters animation states and the game menu states.

A FSM models structured behavior and is composed by states, the transitions between those states, and the actions. The architecture can be used to implements agent-based behaviors like finite state machines and crowd behavior.

The behaviors are affected by the size of vision (which uses the grid made by the broad phase of the collision detection), velocity, energy and type, which are variables available for each type of entity.

8.4.1 The GpuWars Game AI

This game implements 3 different behaviors using FSM: the kamikaze, the group and the tricky behaviors, which are present in the next subsections.

The behaviors are affected by the size of vision (which uses the same NGrid made by the broad phase of the collision detection), velocity and energy (which are variables available for each type of enemy). By modifying these values, this work implements seven different types of enemies.

8.4.1.1 Kamikaze Behavior

The kamikaze approach is a behavior that simulates suicidal attacks. It is created by using a state machine that has four states: wandering, attacking, checking energy and dead. It is illustrated on Figure 8.4.

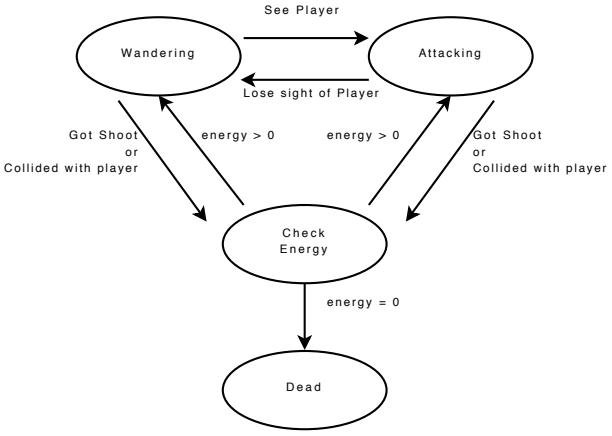

Figure 8.4: The Kamikaze State Machine.

The kamikaze is a very simple behavior. It wanders until it sees the "GPUShip", then it goes attacking it by throwing itself against it. This approach is well suited for GPU architecture, since little information about the scene is necessary.

8.4.1.2 Group Behavior

The group behavior, creates a conduct pattern that makes groups, avoid bullets and attacks. It was modeled with a state machine that has six states: wandering, grouping, attacking, checking energy, avoiding bullets and dead (Figure 8.5).

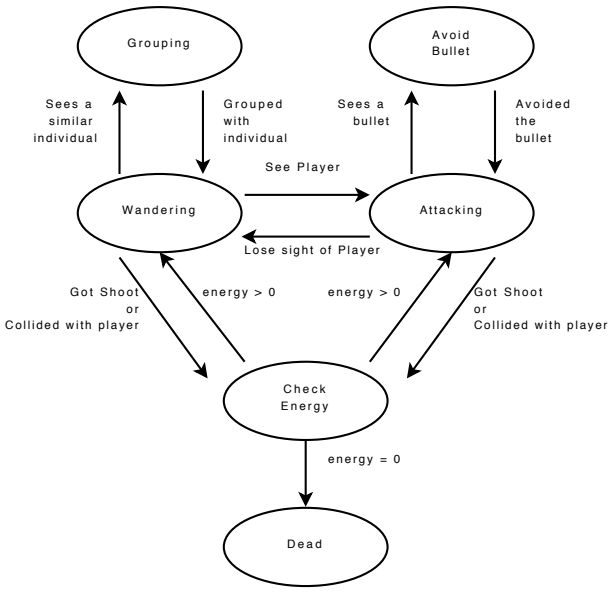

Figure 8.5: The Group State Machine.

This behavior is also very simple. The entity wanders trying to find similar entities, i.e., entities of the same type, and the "GPUShip". If it sees a similar entity, it goes closer to it and makes a group. In cases where it can see the player, it attacks the player by throwing itself against it. If the entity sees a bullet coming in its direction, it tries to avoid it.

8.4.1.3 Tricky Behavior

The tricky behavior is the most complex behavior of the game. This behavior also tries to group similar entities, but it is the only that recoveries energy. It has a state machine with seven states: wandering, grouping, attacking, avoiding bullets, checking energy, escaping and dead (Figure 8.6).

This type of enemy wanders trying to find the "GPUShip" or similar entities. If it sees a similar individual, it goes closer to it and makes a group. If it is seeing the player, it throws itself against it. If the entity sees a bullet coming in its direction it tries to avoid it. If it has little energy it tries to escape from the player neighborhood to recover the lost energy.

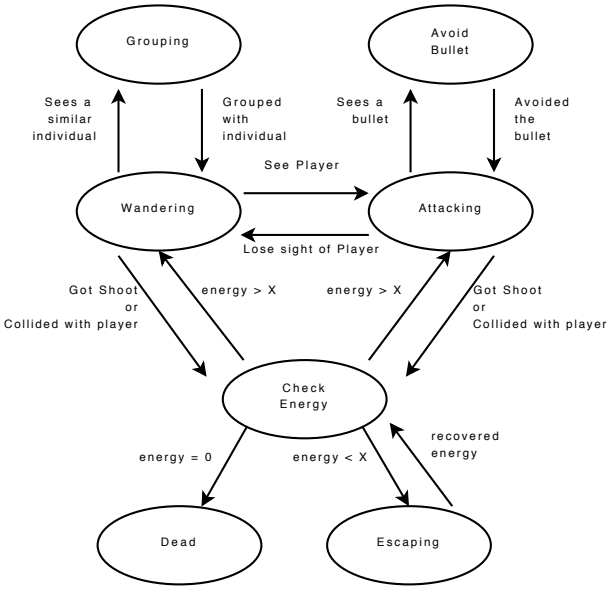

Figure 8.6: The Tricky State Machine.

8.5 Results

For the sake of replicability, the experiments were executed on a common hardware that can run CUDA. In that sense, it was used a Macbook with an AMD Turion Dual-core with 3GB RAM memory and equipped with nVidia Geforce mobile 9400M GPU card (which has only 8 stream processors).

The number of enemies determines the performance of the game. This work has decided to have a maximum bound of 8192 enemies. This number was not chosen because of any limitation of the architecture, but it was chosen in order to present a massive amount of enemies and still have the fun factor in the game. A screenshot of the game can be seen of Figure 8.7.

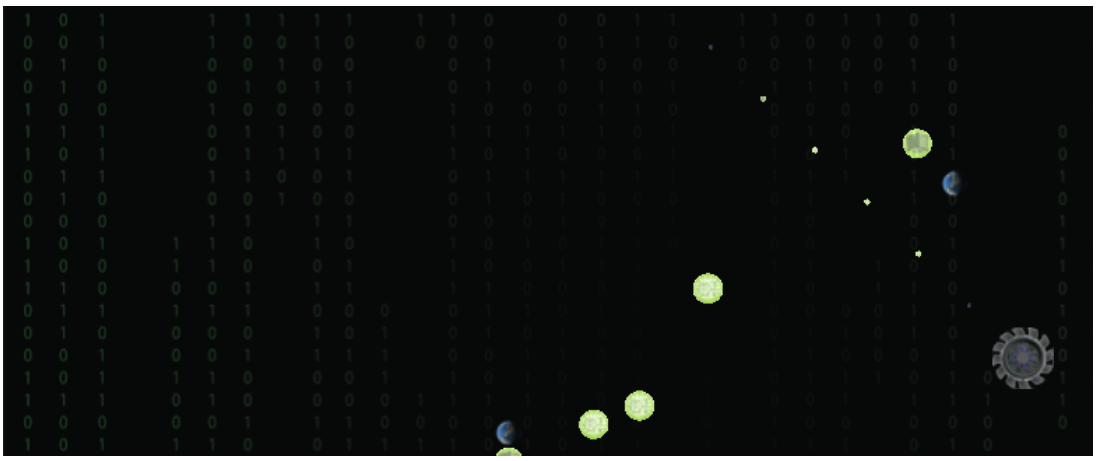

Figure 8.7: A Screenshot of the game.

To better view the performance, Figure 8.8 shows a graph with the performance in FPS of the game in 5 minutes of the game.

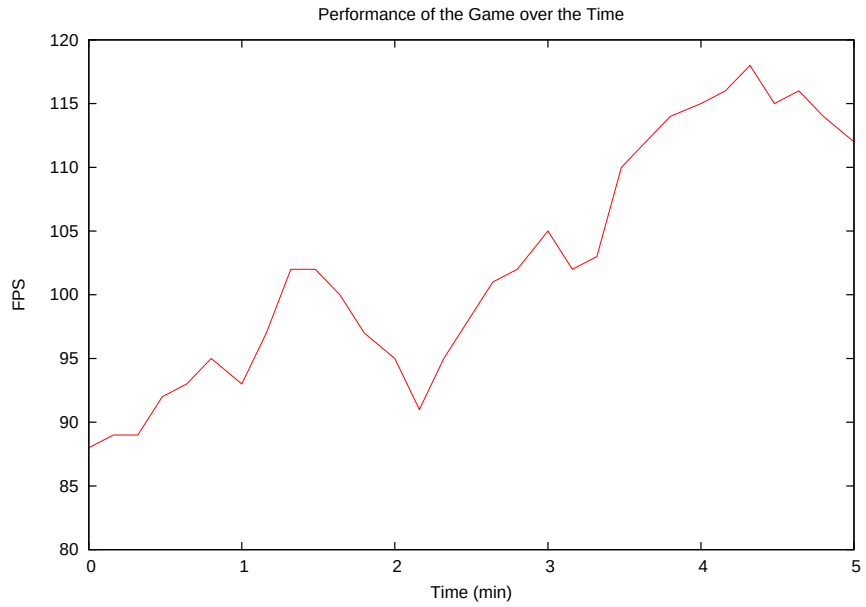

Figure 8.8: Performance of the game

From this figure it can be seen that the performance of the game ranges from 88 to 118 frames per second (FPS). This performance is considered optimal in a game [35].

As far as the author of this work is concerned, there are not any works in the literature that implements all the game tasks using the GPU, like this chapter presents. This work has also implemented the GpuWars using a uniform grid, which the game ranges from 45 to 58 FPS. So the NGrid has a speedup of 1.95 times over the uniform grid.

8.6 Summary

This chapter has presented a game with all tasks implemented in the GPU, using the NGrid. The next chapter will be presenting the conclusion of this work.

Chapter 9
Conclusions

The main motivation for developing a new data structure for the neighborhood gathering on GPUs was the search for a method that could delivery the neighbors consuming less power and memory resources, in order to have a more complex scene, delivered in real-time. One of the benefits of the NGrid is the coalescence memory and bank conflicts avoidance, which comes with the NGrid properties.

The development and evolution of multi-core processors, GPUs and video games indicates that multithread architectures are a trend. In addition, GPUs have evolved into more generic processors, allowing them to be used to process different tasks of the game logic.

The games industry and real-time visualization demands for faster processing, since they demand for real time processing. The neighborhood gathering can have a high complexity. With that has been developed many different data structures for this neighborhood gathering, which were presented by this work.

A lot of works on the field are researching on optimizations and adaptations for the already developed data structures, in order to achieve more simulation with less consuming. The optimization of these structures on the GPU, rely on the organization of the memory gathering in order to have memory convalescence and avoid bank conflicts.

The NGrid was developed focusing specifically on the speeding of the neighborhood gathering. The NGrid has naturally in its memory organization the optimization for GPUs avoiding banks conflicts and having the memory convalescence. This organization also allows a effective use of GPUs shared memory and facilitation on division of work among GPUs. The NGrid has a fast maintenance stage, which only requires the sorting of the structure according to the coordination axis. It also has different configurations, in order to adapt to the different architecture and simulation scenarios.

The NGrid was tested over different GPU architectures and scales: a mobile GPU,

an CUDA capable GPU and a multi-GPU ambient, and it has shown great adaptability over these different ambients, as results have demonstrated, it has expressive speedup over traditional approaches in all those environments.

The NGrid was also tested in different problems that require some form of neighborhood gathering, which was particle system, a simple game shooter, a fluid animation and a crowd simulation. In all those scenarios the NGrid was faster than the traditional approach, giving impressive speedups. Also, the NGrid has much less memory footprint, since it does not require any extra data for organizing its data structures, unlike the others data structures.

As shown in the crowd simulation tests and the fluid animation tests, the NGrid is a topological structure, which gives the approximate neighborhood gathering. The NGrid has an amount error, which could lead to visual issues. These errors have showed to be small in the test, but they still exists. Most of these errors are from the misaligned NGrid, which could be avoided with a full sort step. With that, the NGrid is recommended to be used in games and animations, like the movement of a flock of birds, the flow of a river, or the simple games that do not have many collisions, but not in scientific simulation which requires high precision interactions.

The following contributions are considered from this thesis:

- Formalization of the neighborhood gathering as a computional problem. All the works that the author have seen only approach this problem in the application of the neighborhood gathering, and this work formalized it as a generic problem with different applications;
- Provide a literature overview of the more common data structure that could be applied to neighborhood gathering;
- Provide a new data structure capable of processing a massive number of entities, in real time, using the GPU;
- Provide an architecture for distribution of data over multi-GPUs ambients;
- Provide an architecture for visual simulation using the GPU on mobile architectures.

9.1 Future Work

This work has focus on the creation and application of the NGrid in games and visual simulations. There are others problems that could be investigated as possible cases that the NGrid could be applied, like ray casting.

Also the NGrid has an amount of error from the misaligned grid, which derivate at every time an entity change position in one axis, where it should be sorted again for the others axis. This could be avoided with a more powerful sort step. One approach is to use the Dynamic Parallelism of the newest Kepler GPUs architecture, which allows dispatch of CUDA kernel inside the kernel being processed. This way, every time an entity changes the NGrid position, it is sorted in all the axis, for the right NGrid position.

The proposed data structure was developed to be used with a GPU computing architecture, based on CUDA and OpenCL, but it also could be applied to others multi-cores scenarios, like parallel computers, multi-cores CPUs and even the latest video games (Playstation 3 and Xbox 360).Also the NGrid could be applied in others distributed systems.

References

[1] CHENG, K.-T.; WANG, Y.-C. Using mobile gpu for general-purpose computing a case study of face recognition on smartphones. *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, p. 1 –4, april 2011. ISSN Pending.

[2] MICHALAKES, J.; VACHHARAJANI, M. Gpu acceleration of numerical weather prediction. *IEEE International Symposium on Parallel and Distributed Processing*, p. 1–7, 2008.

[3] UFIMTSEV, I. S.; MARTINEZ, T. J. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *Journal Chemistry Theory Computation*, v. 4 (2), p. 222 – 231, 2008.

[4] JOSELLI, M. et al. A neighborhood grid data structure for massive 3d crowd simulation on gpu. *Games and Digital Entertainment, Brazilian Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, p. 121–131, 2009d.

[5] JOSELLI, M. *Uma arquitetura de motor de física para games 3d com processamento híbrido entre CPU e GPU e distribuição dinâmica de carga*. Dissertao (Mestrado) — Universidade Federal Fluminense, Instituto de Computação, Niteroi - Rio de Janeiro, 2007.

[6] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture documentation version 2.2*. 2009. `http://developer.nvidia.com/object/cuda.html`.

[7] GROUP, K. *OpenCL - The open standard for parallel programming of heterogeneous systems*. 2009. `http://www.khronos.org/opencl/`.

[8] JOSELLI, M. et al. An architeture with automatic load balancing for real-time simulation and visualization systems. *JCIS - Journal of Computational Interdisciplinary Sciences*, p. 207–224, 2010.

[9] JOSELLI, M. et al. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Comput. Entertain.*, ACM, New York, NY, USA, v. 7, n. 4, p. 50:1–50:15, jan. 2010c. ISSN 1544-3574. Disponível em: <http://doi.acm.org/10.1145/1658866.1658869>.

[10] PASSOS, E. B. et al. A bidimensional data structure and spatial optimization for supermassive crowd simulation on gpu. *Comput. Entertain.*, ACM, New York, NY, USA, v. 7, p. 60:1–60:15, January 2010. ISSN 1544-3574. Disponível em: <http://doi.acm.org/10.1145/1658866.1658879>.

[11] PASSOS, E. et al. Supermassive crowd simulation on gpu based on emergent behavior. *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, p. 81–86, 2008.

[12] JR, J. R. da S. et al. A heterogeneous system based on gpu and multi-core cpu for real-time fluid and rigid body simulation. *International Journal of Computational Fluid Dynamics*, v. 26, n. 3, p. 193–204, 2012. Disponível em: <http://www.tandfonline.com/doi/abs/10.1080/10618562.2012.683789>.

[13] JOSELLI, M. et al. Mobilewars: A mobile gpgpu game. *Lecture Notes in Computer Science*, Springer, to appear, p. to appear, 2013.

[14] JOSELLI, M.; CLUA, E. An architecture for massive crowd simulation with physics behavior on the gpu. *Conferencia de Ciências e Artes dos Videojogos*, p. 324–333, 2009b.

[15] JOSELLI, M.; CLUA, E. Gpuwars: Design and implementation of a gpgpu game. *Brazilian Symposium on Games and Digital Entertainment*, IEEE Computer Society, Los Alamitos, CA, USA, p. 132–140, 2009.

[16] JOSELLI, M. et al. Techniques for designing gpgpu games. *Games Innovation Conference (IGIC), 2012 IEEE International*, p. 78–82, August 2012b.

[17] JOSELLI, M. et al. A flocking boids simulation and optimization structure for mobile multicore architectures. In: *SBGames 2012 - Trilha de Computao ()*. Braslia: [s.n.], 2012c.

[18] SILVA, J. et al. An architecture for real time fluid simulation using multiple gpus. *XI Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2012 - Computing Track*, 2012.

[19] SADJADI, M. et al. Architecture of request distributor for gpu clusters. *Applications for Multi-Core Architectures (WAMCA), 2012 Third Workshop on*, 2012.

[20] JOSELLI, M. et al. Gpuwars: Design and implementation of a gpgpu game. *Proceedings of the GPU Forum 2012 (CSBC)*, 2012d.

[21] JOSELLI, M. et al. A neighborhood grid data structure for massive 3d crowd simulation on gpu. *Proceedings of the GPU Forum 2012 (CSBC)*, 2012e.

[22] SILVA, J. R. da et al. A heterogeneous multicore cpu and gpu architecture for two-way real time fluid simulation. *Proceedings of the GPU Forum 2012 (CSBC)*, 2012.

[23] JOSELLI, M. et al. A novel data structure for particle system simulation based on gpu with the use of neighborhood grids. *Proceedings of the GPU Forum 2012 (CSBC)*, 2012f.

[24] JOSELLI, M.; CLUA, E. Techniques for designing gpgpu games. *2012 GPU Technology Conference (GTC 2012)*, 2012.

[25] JOSELLI, M. et al. A flocking boids simulation and optimization structure for mobile multicore architectures. *2013 GPU Technology Conference (GTC 2013)*, 2013a.

[26] JOSELLI, M. et al. Design and development of a gpu computing framework. *2013 GPU Technology Conference (GTC 2013)*, 2013b.

[27] IHMSEN, M. et al. A parallel sph implementation on multi-core cpus. *Comput. Graph. Forum*, v. 30, n. 1, p. 99–112, 2011.

[28] CHAZELLE, B. An improved algorithm for the fixed-radius neighbor problem. *Inf. Process. Lett.*, v. 16, n. 4, p. 193–198, 1983.

[29] DICKERSON, M. T.; EPPSTEIN, D. Algorithms for proximity problems in higher dimensions. *Comput. Geom. Theory Appl.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 5, n. 5, p. 277–291, jan. 1996. ISSN 0925-7721. Disponível em: <http://dx.doi.org/10.1016/0925-7721(95)00009-7>.

[30] CLARKSON, K. L. Fast algorithms for the all nearest neighbors problem. *FOCS*, p. 226–232, 1983.

[31] YIANILOS, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In: *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1993. (SODA '93), p. 311–321. ISBN 0-89871-313-7. Disponível em: <http://dl.acm.org/citation.cfm?id=313559.313789>.

[32] COVER, T.; HART, P. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, IEEE, v. 13, n. 1, p. 21–27, jan. 1967. ISSN 0018-9448. Disponível em: <http://dx.doi.org/10.1109/TIT.1967.1053964>.

[33] PAN, F. et al. Comprehensive vertical sample-based knn/lsvm classification for gene expression analysis. *J. of Biomedical Informatics*, Elsevier Science, San Diego, USA, v. 37, n. 4, p. 240–248, ago. 2004. ISSN 1532-0464. Disponível em: <http://dx.doi.org/10.1016/j.jbi.2004.07.003>.

[34] ZHANG, H. et al. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In: *In CVPR*. [S.l.: s.n.], 2006. p. 2126–2136.

[35] JOSELLI, M. et al. An adaptative game loop architecture with automatic distribution of tasks between cpu and gpu. *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, p. 115–120, 2009c.

[36] JOSELLI, M. et al. Automatic dynamic task distribution between cpu and gpu for real-time systems. *IEEE Proceedings of the 11th International Conference on Computational Science and Engineering*, p. 48–55, 2008b.

[37] JOSELLI, M. et al. A new physics engine with automatic process distribution between cpu-gpu. *Sandbox 08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, p. 149–156, 2008.

[38] JOSELLI, M. et al. An architecture with automatic load balancing and distribution for digital games. In: *Proceedings of the 2010 Brazilian Symposium on Games and Digital Entertainment*. Washington, DC, USA: IEEE Computer Society, 2010b. (SBGAMES '10), p. 59–70. ISBN 978-0-7695-4359-8. Disponível em: <http://dx.doi.org/10.1109/SBGAMES.2010.19>.

[39] AILA, T.; LAINE, S.; KARRAS, T. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. *NVIDIA Technical Report*, n. NVR-2012-02, jun. 2012.

[40] NVIDIA. *TESLA K20 GPU ACTIVE ACCELERATOR*. 2012. http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Active-BD-06499-001-v02.pdf.

[41] OWENS, J. D. et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, v. 26(1), p. 80–113, 2007.

[42] DU, P. et al. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Comput.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 38, n. 8, p. 391–407, ago. 2012. ISSN 0167-8191. Disponível em: <http://dx.doi.org/10.1016/j.parco.2011.10.002>.

[43] ADAMS, B. et al. Adaptively sampled particle fluids. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 26, n. 3, jul. 2007. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/1276377.1276437>.

[44] MUYAN-OZCELIK, P. et al. Fast deformable registration on the gpu: A cuda implementation of demons. In: GAVRILOVA, M. et al. (Ed.). *the 1st technical session on UnConventional High Performance Computing (UCHPC) in conjunction with the 6th International Conference on Computational Science and Its Applications (ICCSA)*. Los Alamitos, California: IEEE Computer Society, 2008. p. 223–233.

[45] TUNACODE. *CUVILib: CUDA Vision and Imaging Library*. 2010. http://www.cuvilib.com/.

[46] KADLEC, B.; TUFO, H.; DORN, G. Knowledge-assisted visualization and segmentation of geologic features using implicit surfaces. *IEEE Computer Graphics and Applications*, IEEE Computer Society, Los Alamitos, CA, USA, v. 99, n. PrePrints, 2009. ISSN 0272-1716.

[47] NVIDIA. *CUDA Zone*. 2010. http://www.nvidia.com/object/cuda_home_new.html.

[48] GPGPU.ORG. *GPGPU.org*. 2010. http://www.gpgpu.org.

[49] KIM, J. et al. Achieving a single compute device image in opencl for multiple gpus. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2011. (PPoPP '11), p. 277–288. ISBN 978-1-4503-0119-0.

[50] NUKADA, A.; MARUYAMA, Y.; MATSUOKA, S. High performance 3-d fft using multiple cuda gpus. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. New York, NY, USA: ACM, 2012. (GPGPU-5), p. 57–63. ISBN 978-1-4503-1233-2.

[51] CEVAHIR, A.; NUKADA, A.; MATSUOKA, S. Fast conjugate gradients with multiple gpus. In: *Proceedings of the 9th International Conference on Computational Science: Part I*. Berlin, Heidelberg: Springer-Verlag, 2009. (ICCS '09), p. 893–903. ISBN 978-3-642-01969-2.

[52] AKENINE-MOLLER, T.; STROM, J. Graphics processing units for handhelds. *Proceedings of the IEEE*, v. 96, n. 5, p. 779 –789, may 2008. ISSN 0018-9219.

[53] THERDSTEERASUKDI, K. et al. Utilizing rf-i and intelligent scheduling for better throughput/watt in a mobile gpu memory system. *ACM Trans. Archit. Code Optim.*, ACM, New York, NY, USA, v. 8, n. 4, p. 51:1–51:19, jan. 2012. ISSN 1544-3566. Disponível em: <http://doi.acm.org/10.1145/2086696.2086730>.

[54] MUNSHI, A.; GINSBURG, D.; SHREINER, D. Opengl(r) es 2.0 programming guide. *Addison-Wesley Professional*, 2008.

[55] KIM, T.-Y.; KIM, J.; HUR, H. A unified shader based on the opengl es 2.0 for 3d mobile game development. In: *Proceedings of the 2nd international conference on Technologies for e-learning and digital entertainment*. Berlin, Heidelberg: Springer-Verlag, 2007. (Edutainment'07), p. 898–903. ISBN 978-3-540-73010-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1772177.1772270>.

[56] HUANG, Y.; CHAPMAN, P.; EVANS, D. Privacy-preserving applications on smartphones. In: *Proceedings of the 6th USENIX conference on Hot topics in security*. Berkeley, CA, USA: USENIX Association, 2011. (HotSec'11), p. 4–4. Disponível em: <http://dl.acm.org/citation.cfm?id=2028040.2028044>.

[57] GUIHOT, H. *Pro Android Apps Performance Optimization*. Apress, 2012. ISBN 9781430239994. Disponível em: <http://books.google.com.br/books?id=ROlZAs4WLYwC>.

[58] FALCAO, G.; SILVA, V.; SOUSA, L. How gpus can outperform asics for fast ldpc decoding. In: *Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009. (ICS '09), p. 390–399. ISBN 978-1-60558-498-0. Disponível em: <http://doi.acm.org/10.1145/1542275.1542330>.

[59] BENTLEY, J. L. *A survey of techniques for fixed radius near neighbor searching*. Stanford, CA, USA, 1975.

[60] BENTLEY, J. L.; STANAT, D. F.; JR., E. H. W. The complexity of finding fixed-radius near neighbors. *Inf. Process. Lett.*, v. 6, n. 6, p. 209–212, 1977.

[61] LENHOF, H.-P.; SMID, M. An animation of a fixed-radius all-nearest-neighbors algorithm. In: *Proceedings of the tenth annual symposium on Computational geometry*. New York, NY, USA: ACM, 1994. (SCG '94), p. 387–. ISBN 0-89791-648-4. Disponível em: <http://doi.acm.org/10.1145/177424.178099>.

[62] VIGNERON, A. Reporting intersections among thick objects. *Information Processing Letters*, v. 85, n. 2, p. 87 – 92, 2003. ISSN 0020-0190. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020019002003472>.

[63] TURAU, V. Fixed-radius near neighbors search. *Inf. Process. Lett.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 39, n. 4, p. 201–203, set. 1991. ISSN 0020-0190. Disponível em: <http://dx.doi.org/10.1016/0020-0190(91)90180-P>.

[64] DICKERSON, M. T.; DRYSDALE, R. S. Fixed-radius near neighbors search algorithms for points and segments. *Inf. Process. Lett.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 35, n. 5, p. 269–273, ago. 1990. ISSN 0020-0190. Disponível em: <http://dx.doi.org/10.1016/0020-0190(90)90056-4>.

[65] LENHOF, H.-P.; SMID, M. Enumerating the k closest pairs optimally. In: *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1992. (SFCS '92), p. 380–386. ISBN 0-8186-2900-2. Disponível em: <http://dx.doi.org/10.1109/SFCS.1992.267752>.

[66] SALOWE, J. Shallow interdistance selection and interdistance enumeration. In: DEHNE, F.; SACK, J.-R.; SANTORO, N. (Ed.). *Algorithms and Data Structures*. Springer Berlin Heidelberg, 1991, (Lecture Notes in Computer Science, v. 519). p. 117–128. ISBN 978-3-540-54343-5. Disponível em: <http://dx.doi.org/10.1007/BFb0028255>.

[67] HORVATH, C.; GEIGER, W. Directable, high-resolution simulation of fire on the gpu. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 28, n. 3, p. 41:1–41:8, jul. 2009. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/1531326.1531347>.

[68] KUESTER, F. et al. Visualization of particle traces in virtual environments. In: *Proceedings of the ACM symposium on Virtual reality software and technology*. New York, NY, USA: ACM, 2001. (VRST '01), p. 151–157. ISBN 1-58113-427-4. Disponível em: <http://doi.acm.org/10.1145/505008.505038>.

[69] MCGUIRE, M.; FEIN, A. Real-time rendering of cartoon smoke and clouds. In: *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*. New York, NY, USA: ACM, 2006. (NPAR '06), p. 21–26. ISBN 1-59593-357-3. Disponível em: <http://doi.acm.org/10.1145/1124728.1124733>.

[70] KERMEL, L. et al. Tackling computer generated clouds in 'madagascar: The crate escape'. In: *ACM SIGGRAPH 2008 talks*. New York, NY, USA: ACM, 2008. (SIGGRAPH '08), p. 5:1–5:1. ISBN 978-1-60558-343-3. Disponível em: <http://doi.acm.org/10.1145/1401032.1401039>.

[71] REEVES, W. T. Particle systems a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, ACM, New York, NY, USA, v. 2, n. 2, p. 91–108, abr. 1983. ISSN 0730-0301. Disponível em: <http://doi.acm.org/10.1145/357318.357320>.

[72] REEVES, W. T.; BLAU, R. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In: *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1985. (SIGGRAPH '85), p. 313–322. ISBN 0-89791-166-0. Disponível em: <http://doi.acm.org/10.1145/325334.325250>.

[73] SIMS, K. Particle animation and rendering using data parallel computation. In: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1990. (SIGGRAPH '90), p. 405–413. ISBN 0-89791-344-2. Disponível em: <http://doi.acm.org/10.1145/97879.97923>.

[74] FEIJO, B.; PAGLIOSA, P. A.; CLUA, E. W. G. Visualizao, simulao e games. *Breitman, K. and Anido, R. (Ed.), Atualizaes em Informtica, Editora PUC-Rio, Rio de Janeiro, Brasil*, p. 127–185, 2006.

[75] NORTH, R. G. *Grand Theft Auto IV, Rockstar Games*. 2008. `http://www.rockstargames.com/IV/`.

[76] AITKEN, M. et al. The lord of the rings: the visual effects that brought middle earth to the screen. In: SIGGRAPH: ACM SPECIAL INTEREST GROUP ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES. *ACM SIGGRAPH 2004*. New York, NY, USA: ACM Press, 2004. Disponível em: <http://dx.doi.org/10.1145/1103900.1103911>.

[77] REYNOLDS, C. W. Flocks, herds and schools: A distributed behavioral model. In: *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1987. p. 25–34. ISBN 0-89791-227-6.

[78] BERG, J. van den et al. Interactive navigation of multiple agents in crowded environments. In: *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2008. p. 139–147. ISBN 978-1-59593-983-8.

[79] JIN, X. et al. Interactive control of real-time crowd navigation in virtual environment. In: *VRST '07: Proceedings of the 2007 ACM symposium on Virtual reality software and technology*. New York, NY, USA: ACM, 2007. p. 109–112. ISBN 978-1-59593-863-3.

[80] NVIDIA. *Skinned Instancing*. 2008. `http://developer.download.nvidia.com/SDK/10/direct3d/Source/SkinnedInstancing /doc/SkinnedInstancingWhitePaper.pdf`.

[81] MUSSE, S. R.; THALMANN, D. A model of human crowd behavior: Group inter-relationship and collision detection analysis. In: EUROGRAPHICS. *Workshop Computer Animation and Simulation of Eurographics*. 1997. p. 39–52. Disponível em: <http://citeseer.ist.psu.edu/584571.html>.

[82] SHAO, W.; TERZOPOULOS, D. Autonomous pedestrians. In: SCA. *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. New York, NY, USA: ACM, 2005. p. 19–28.

[83] RODRIGUES, R. A. et al. An interactive model for steering behaviors of groups of characters. *Appl. Artif. Intell.*, Taylor & Francis, Inc., Bristol, PA, USA, v. 24, n. 6, p. 594–616, jul. 2010. ISSN 0883-9514. Disponível em: <http://dx.doi.org/10.1080/08839514.2010.492167>.

[84] PELECHANO, N.; ALLBECK, J. M.; BADLER, N. I. Controlling individual agents in high-density crowd simulation. In: SCA. *SCA 07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007. p. 99–108. ISBN 978-1-59593-624-4.

[85] TREUILLE, A.; COOPER, S.; POPOVIć, Z. Continuum crowds. In: SIGGRAPH. *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*. New York, NY, USA: ACM, 2006. p. 1160–1168. ISBN 1-59593-364-6.

[86] LUCY, L. B. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal, vol. 82*, p. 1013–1024, 1977.

[87] GINGOLD, R. A.; MONAGHAN, J. J. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices, vol. 181*, p. 375–389, 1977.

[88] DESBRUN, M.; GASCUEL, M. paule. Smoothed particles: A new paradigm for animating highly deformable bodies. *In Computer Animation and Simulation 96 (Proceedings of EG Workshop on Animation and Simulation)*, Springer-Verlag, p. 61–76, 1996.

[89] MüLLER, M.; CHARYPAR, D.; GROSS, M. Particle-based fluid simulation for interactive applications. In: *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003. p. 154–159. ISBN 1-58113-659-5.

[90] MONAGHAN, J. J. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics. Vol. 30*, p. 543–574, 1992.

[91] LENNERZ, C.; SCHMER, E.; WARKEN, T. A framework for collision detection and response. *in 11th European Simulation Symposium, ESS99*, p. 309–314, 1999.

[92] ERICSON, C. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558607323.

[93] GOURMEL, O. et al. Bvh for efficient raytracing of dynamic metaballs on gpu. In: *SIGGRAPH 2009: Talks*. New York, NY, USA: ACM, 2009. (SIGGRAPH '09), p. 51:1–51:1. ISBN 978-1-60558-834-6. Disponível em: <http://doi.acm.org/10.1145/1597990.1598041>.

[94] HAVERKORT, H.; TOMA, L.; ZHUANG, Y. Computing visibility on terrains in external memory. *J. Exp. Algorithmics*, ACM, New York, NY, USA, v. 13, p. 5:1.5–5:1.23, fev. 2009. ISSN 1084-6654. Disponível em: <http://doi.acm.org/10.1145/1412228.1412233>.

[95] DONEV, A.; TORQUATO, S.; STILLINGER, F. H. Neighbor list collision-driven molecular dynamics simulation for nonspherical hard particles. i. algorithmic details. *J. Comput. Phys.*, Academic Press Professional, Inc., San Diego, CA, USA, v. 202, n. 2, p. 737–764, jan. 2005. ISSN 0021-9991. Disponível em: <http://dx.doi.org/10.1016/j.jcp.2004.08.014>.

[96] COHEN, J. D. et al. I-collide: an interactive and exact collision detection system for large-scale environments. In: *Proceedings of the 1995 symposium on Interactive 3D graphics*. New York, NY, USA: ACM, 1995. (I3D '95), p. 189–ff. ISBN 0-89791-736-7. Disponível em: <http://doi.acm.org/10.1145/199404.199437>.

[97] BARAFF, D. An introduction to physically based modeling: Rigid body simulation i - unconstrained rigid body dynamics. *In An Introduction to Physically Based Modelling, SIGGRAPH '97 Course Notes*, p. 97, 1997.

[98] ROCHA, R. de S.; RODRIGUES, M. A. F. An evaluation of a collision handling system using sphere-trees for plausible rigid body animation. In: *Proceedings of the 2008 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2008. (SAC '08), p. 1241–1245. ISBN 978-1-59593-753-7. Disponível em: <http://doi.acm.org/10.1145/1363686.1363972>.

[99] FISHMAN, J.; HAVERKORT, H.; TOMA, L. Improved visibility computation on massive grid terrains. In: *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, NY, USA: ACM, 2009. (GIS '09), p. 121–130. ISBN 978-1-60558-649-6. Disponível em: <http://doi.acm.org/10.1145/1653771.1653791>.

[100] TRACY, D. J.; BUSS, S. R.; WOODS, B. M. Efficient large-scale sweep and prune methods with aabb insertion and removal. In: *Proceedings of the 2009 IEEE Virtual Reality Conference*. Washington, DC, USA: IEEE Computer Society, 2009. (VR '09), p. 191–198. ISBN 978-1-4244-3943-0. Disponível em: <http://dx.doi.org/10.1109/VR.2009.4811022>.

[101] LIU, F. et al. Real-time collision culling of a million bodies on graphics processing units. In: *ACM SIGGRAPH Asia 2010 papers*. New York, NY, USA: ACM, 2010. (SIGGRAPH ASIA '10), p. 154:1–154:8. ISBN 978-1-4503-0439-9. Disponível em: <http://doi.acm.org/10.1145/1866158.1866180>.

[102] COUMANS, E. *Bullet Physics Library*. 2012. Disponível em: `http://www.bulletphysics.com`.

[103] Box2D. 2012. Disponível em: <http://www.box2d.org/screenshots.html>.

[104] GOVINDARAJU, K. N. et al. CULLIDE: interactive collision detection between complex models in large environments using graphics hardware. In: *Graphics Hardware 2003*. [S.l.: s.n.], 2003. p. 25–32.

[105] COMING, D. S.; STAADT, O. G. Kinetic sweep and prune for collision detection. In: *Proceedings of the Second Workshop in Virtual Reality Interactions and Physical Simulations (VRIPHYS'05)*. [S.l.: s.n.], 2005.

[106] COMING, D. S.; STAADT, O. G. Kinetic sweep and prune for multi-body continuous motion. *Computers & Graphics*, v. 30, n. 3, p. 439–449, 2006.

[107] KALOJANOV, J.; SLUSALLEK, P. A parallel algorithm for construction of uniform grids. *Proceedings of the Conference on High Performance Graphics 2009*, ACM, New York, NY, USA, p. 23–28, 2009. Disponível em: <http://doi.acm.org/10.1145/1572769.1572773>.

[108] TESCHNER, M. et al. Optimized spatial hashing for collision detection of deformable objects. p. 47–54, 2003.

[109] OAT, C.; BARCZAK, J.; SHOPF, J. Efficient spatial binning on the gpu. *Tech Report*.

[110] NVIDIA. *CUDA Particles*. 2008. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/ particles/doc/particles.pdf`.

[111] REYNOLDS, C. Interaction with groups of autonomous characters. In: GDC. *Game Developers Conference 2000*. 2000. Disponível em: <http://www.red3d.com/cwr/papers/2000/pip.html>.

[112] PURCELL, T. J. et al. Photon mapping on programmable graphics hardware. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, Eurographics Association, p. 41–50, 2003.

[113] SMITH, R. *Open Dynamics Engine*. 2012. Disponível em: `http://www.ODE.org/`. 20/12/2012.

[114] LEMBCKE, S. *Chipmunk Game Dynamics*. 2012. Disponível em: `http://chipmunk-physics.net/`. 20/12/2012.

[115] GRAND, S. L. Broad-phase collision detection with cuda. In: NGUYEN, H. (Ed.). *GPU Gems 3*. [S.l.]: Addison Wesley Professional, 2007. cap. 32.

[116] REYNOLDS, C. Steering behaviors for autonomous characters. In: GDC. *Game Developers Conference 1999*. 1999. Disponível em: <citeseer.ist.psu.edu/reynolds99steering.html>.

[117] REYNOLDS, C. Big fast crowds on ps3. In: SANDBOX. *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*. New York, NY, USA: ACM, 2006. p. 113–121. ISBN 1-59593-386-7.

[118] CHIARA, R. D. et al. Massive simulation using gpu of a distributed behavioral model of a flock with obstacle avoidance. In: *Vision, Modeling, and Visualization (VMV)*. [S.l.: s.n.], 2004. p. 233–240.

[119] SILVA, A. R.; LAGES, W. S.; CHAIMOWICZ, L. Improving boids algorithm in gpu using estimated self occlusion. In: *Proceedings of SBGames'08 - VII Brazilian Symposium on Computer Games and Digital Entertainment*. [S.l.]: Sociedade Brasileira de Computação, SBC, 2008. p. 41–46. ISBN 85-766-9217-1.

[120] KUROSE, S.; TAKAHASHI, S. Constraint-based simulation of interactions between fluids and unconstrained rigid bodies. *Proceedings of Spring Conference on Computer Graphics*, p. 197–204, 2009.

[121] SUN, H.; HAN, J. Particle-based realistic simulation of fluid–solid interaction. *Comput. Animat. Virtual Worlds*, John Wiley and Sons Ltd., Chichester, UK, v. 21, n. 6, p. 589–595, nov. 2010. ISSN 1546-4261. Disponível em: <http://dx.doi.org/10.1002/cav.379>.

[122] BAYRAKTAR, S.; GÜDÜKBAY, U.; ÖZGÜC, B. Gpu-based neighbor-search algorithm for particle simulations. *J. Graphics, GPU, & Game Tools*, v. 14, n. 1, p. 31–42, 2009.

[123] GREEN, S. *Particle-based Fluid Simulation*. 2008. Http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_ParticleFluids.pdf. 17/10/2011.

[124] GOSWAMI, P. et al. Interactive sph simulation and rendering on the gpu. In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010. (SCA '10), p. 55–64. Disponível em: <http://dl.acm.org/citation.cfm?id=1921427.1921437>.

[125] MEHTA, D. P.; SAHNI, S. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. [S.l.]: Chapman & Hall/CRC, 2004. ISBN 1584884355.

[126] LEE, H. *Quadtree with Quadratic Storage*. [s.n.], 1987. Disponível em: <http://books.google.com.br/books?id=PMsfauXvGqwC>.

[127] FINKEL, R. A. et al. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, Springer Berlin / Heidelberg, v. 4, n. 1, p. 1–9, mar. 1974. ISSN 0001-5903. Disponível em: <http://dx.doi.org/10.1007/BF00288933>.

[128] LABORATORY, R. P. I. I. P.; MEAGHER, D. *Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. [s.n.], 1980. Disponível em: <http://books.google.com.br/books?id=CgRPOAAACAAJ>.

[129] HE, X.; CHEN, L.; ZHU, Q. Transactions on edutainment vi. In: PAN, Z.; CHEOK, A. D.; MüLLER, W. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2011. cap. A novel method for large crowd flow, p. 67–78. ISBN 978-3-642-22638-0. Disponível em: <http://dl.acm.org/citation.cfm?id=2043065.2043075>.

[130] BUNLUTANGTUM, R.; KANONGCHAIYOS, P. Adaptive grid refinement using view-dependent octree for grid-based smoke simulation. In: *Proceedings of the 4th international conference on Motion in Games*. Berlin, Heidelberg: Springer-Verlag, 2011. (MIG'11), p. 204–215. ISBN 978-3-642-25089-7.

[131] OGUZ, O.; DURUPINAR, F.; GüDüKBAY, U. Dynamic point region quadtrees for particle simulations. *Inf. Sci.*, Elsevier Science Inc., New York, NY, USA, v. 218, p. 133–145, jan. 2013. ISSN 0020-0255. Disponível em: <http://dx.doi.org/10.1016/j.ins.2012.06.028>.

[132] LUX, C.; FRöHLICH, B. Gpu-based ray casting of stacked out-of-core height fields. In: *Proceedings of the 7th international conference on Advances in visual computing - Volume Part I*. Berlin, Heidelberg: Springer-Verlag, 2011. (ISVC'11), p. 269–280. ISBN 978-3-642-24027-0. Disponível em: <http://dl.acm.org/citation.cfm?id=2045110.2045140>.

[133] WEI, C.; GAIN, J.; MARAIS, P. Interactive gpu-based octree generation and traversal. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. New York, NY, USA: ACM, 2012. (I3D '12), p. 211–211. ISBN 978-1-4503-1194-6. Disponível em: <http://doi.acm.org/10.1145/2159616.2159657>.

[134] CHEN, H. H.; HUANG, T. S. A survey of construction and manipulation of octrees. *Comput. Vision Graph. Image Process.*, Academic Press Professional, Inc., San Diego, CA, USA, v. 43, n. 3, p. 409–431, set. 1988. ISSN 0734-189X. Disponível em: <http://dx.doi.org/10.1016/0734-189X(88)90092-8>.

[135] PUECH, C.; YAHIA, H. Quadtrees, octrees, hyperoctrees: a unified analytical approach to tree data structures used in graphics, geometric modeling and image processing. In: O'ROURKE, J. (Ed.). *Symposium on Computational Geometry*. [S.l.]: ACM, 1985. p. 272–280. ISBN 0-89791-163-6.

[136] JIMENEZ, P.; THOMAS, F.; TORRAS, C. 3D collision detection: a survey. *Computers & Graphics*, v. 25, n. 2, p. 269–285, abr. 2001. Disponível em: <http://www.ingentaconnect.com/content/els/00978493/2001/00000025/00000002/art00130>.

[137] FARSEER. *Farseer Physics Engine*. 2012. Disponível em: `http://farseerphysics.codeplex.com/`. 20/12/2012.

[138] DICK, C.; KRÜGER, J.; WESTERMANN, R. GPU ray-casting for scalable terrain rendering. In: *Proceedings of Eurographics 2009 - Areas Papers*. [S.l.: s.n.], 2009. p. 43–50.

[139] HEGEMAN, K.; CARR, N. A.; MILLER, G. S. P. Particle-based fluid simulation on the gpu. *Lecture Notes in Computer Science*, Springer, v. 3994, p. 228–235, 2006.

[140] BÉDORF, J.; GABUROV, E.; ZWART, S. P. A sparse octree gravitational n-body code that runs entirely on the gpu processor. *J. Comput. Phys.*, Academic Press Professional, Inc., San Diego, CA, USA, v. 231, n. 7, p. 2825–2839, abr. 2012. ISSN 0021-9991. Disponível em: <http://dx.doi.org/10.1016/j.jcp.2011.12.024>.

[141] LUQUE, R. G.; COMBA, J. a. L. D.; FREITAS, C. M. D. S. Broad-phase collision detection using semi-adjusting bsp-trees. In: *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2005. (I3D '05), p. 179–186. ISBN 1-59593-013-2. Disponível em: <http://doi.acm.org/10.1145/1053427.1053457>.

[142] VERLET, L. Computer "experiments" on classical fluids. ii. equilibrium correlation functions. *Phys. Rev.*, v. 165, p. 201–214, 1968.

[143] DUVENHAGE, B. Using an implicit min/max kd-tree for doing efficient terrain line of sight calculations. In: *Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. New York, NY, USA: ACM, 2009. (AFRIGRAPH '09), p. 81–90. ISBN 978-1-60558-428-7. Disponível em: <http://doi.acm.org/10.1145/1503454.1503469>.

[144] GRAY, L. et al. A mathematician looks at wolframs new kind of science. *Notices of the American Mathematical Society 50 (2) (2003) 200211, URL http://www.ams.org/notices/200302/fea-gray.pdf. URL http://www.ams.org/notices/200302/fea-gray.pdf*, v. 50, p. 200–211, 2003.

[145] HABERMANN, A.; SCIENCE., C.-M. U. P. P. D. O. C. *Parallel Neighbor-Sort (or the Glory of the Induction Principle)*. Defense Technical Information Center, 1972. (Research paper). Disponível em: <http://books.google.com.br/books?id=HFnkNwAACAAJ>.

[146] BATCHER, K. E. Sorting networks and their applications. In: *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*. New York, NY, USA: ACM, 1968. p. 307–314.

[147] BATCHER, K. E. Sorting networks and their applications. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. New York, NY, USA: ACM, 1968. (AFIPS '68 (Spring)), p. 307–314. Disponível em: <http://doi.acm.org/10.1145/1468075.1468121>.

[148] BLELLOCH, G. E. et al. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 1998.

[149] NVIDIA. Bitonic sort demo. *Tech Report*, 2007. Disponível em: <http://www.nvidia.com/content/cudazone/cuda_ sdk/Data-Parallel_ Algorithms.html# bitonic>.

[150] SARKAR, P. A brief history of cellular automata. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 32, n. 1, p. 80–107, 2000. ISSN 0360-0300.

[151] EBERLY, D. H. *Game Physics*. [S.l.]: Morgan Kaufmann, 2004.

[152] BUCK, D. K.; COLLINS, A. A. POV-Ray - The Persistence of Vision Raytracer. Disponível em: <http://www.povray.org/>.

[153] ZHANG, Y.; SOLENTHALER, B.; PAJAROLA, R. Adaptive sampling and rendering of fluids on the gpu. In: *Proceedings of the Fifth Eurographics / IEEE VGTC conference on Point-Based Graphics*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008. (SPBG'08), p. 137–146. ISBN 978-3-905674-12-5. Disponível em: <http://dx.doi.org/10.2312/VG/VG-PBG08/137-146>.

[154] SOLENTHALER, B.; PAJAROLA, R. Density contrast sph interfaces. In: *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008. (SCA '08), p. 211–218. ISBN 978-3-905674-10-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1632592.1632623>.

[155] COURTY, N.; MUSSE, S. R. Simulation of large crowds in emergency situations including gaseous phenomena. In: CGI. *CGI '05: Proceedings of the Computer Graphics International 2005*. Washington, DC, USA: IEEE Computer Society, 2005. p. 206–212. ISBN 0-7803-9330-9.

[156] CORPORATION, N. *NVIDIA CUDA Programming Guide*. 2012.

[157] JOSELLI, M.; CLUA, E. grmobile: A framework for touch and accelerometer gesture recognition for mobile games. *Proceedings of the 2009 VIII Brazilian Symposium on Games and Digital Entertainment*, IEEE Computer Society, Washington, DC, USA, p. 141–150, 2009c. Disponível em: <http://dx.doi.org/10.1109/SBGAMES.2009.24>.

[158] ANDROID, G. *Android Renderscript*. 2012. http://developer.android.com/guide/topics/renderscript/index.html.

[159] OSTRANDER, J. *Android Ui Fundamentals: Develop & Design*. Pearson Education, 2012. ISBN 9780132929028. Disponível em: <http://books.google.com.br/books?id=z5jAHfSeeJMC>.

[160] HARADA, T. Real-Time Rigid Body Simulation on GPUs. In: NGUYEN, H. (Ed.). *GPU Gems 3*. Addison Wesley Professional, 2007. cap. 29. Disponível em: <http://my.safaribooksonline.com/9780321545428/ch29>.

[161] CREATIONS, B. *Geometry Wars Retro Evolve*. 2009. http://www.bizarrecreations.com/games/ geometry_wars_retro_evolved/.

[162] INC., Q. E. *Every Extend Extra Extreme*. 2009. `http://www.qentertainment.com/eng/ 2007/09/every_extend_extra_extreme.html`.

[163] VALENTE, L.; CONCI, A.; FEIJÓ, B. Real time game loop models for single-player computer games. *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, p. 89–99, 2005.

[164] JOSELLI, M. et al. An architeture with automatic load balancing for real-time simulation and visualization systems. *JCIS - Journal of Computational Interdisciplinary Sciences*, p. 207–224, 2009.

[165] KRUEGER, J. A gpu framework for interactive simulation and rendering of fluid effects. *IT - Information Technology*, v. 4, p. (accepted), 2008. Disponível em: <http://www.sci.utah.edu/publications/krueger08/ GPU_framework.pdf>.

[166] PODLOZHNYUK, V. *Parallel mersenne twister*. 2007. `http://developer.download.nvidia.com/ compute/cuda/sdk/website/projects/ MersenneTwister/doc/MersenneTwister.pdf`.

[167] MICROSOFT. *Advanced Particles*. 2007. Siggraph 2007: Real-Time Rendering in 3D Graphics and Games course.

[168] KIPFER, P.; SEGAL, M.; WESTERMANN, R. Uberflow: a gpu-based particle engine. *Graphics Hardware 2004*, p. 115–122, 2004.

[169] BOURG, D. M.; SEEMANN, G. Ai for game developers. *O'Reilly Media, Inc.*, 2004.

[170] DYBSAND, E. A finite state machine class. *Game Programming Gems*, p. 237–248, 2000.

[171] RANKIN, J. R.; VARGAS, S. S. Fps extensions modelling esgs. In: *ACHI '09: Proceedings of the 2009 Second International Conferences on Advances in Computer-Human Interactions*. Washington, DC, USA: IEEE Computer Society, 2009. p. 152–155. ISBN 978-0-7695-3529-6.

[172] LI, F.; WOODHAM, R. J. Video analysis of hockey play in selected game situations. *Image Vision Comput.*, Butterworth-Heinemann, Newton, MA, USA, v. 27, n. 1-2, p. 45–58, 2009. ISSN 0262-8856.