Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Simulação de Fluidos no Unity Utilizando o Método SPH

Claudio Henrique da Silva

JUIZ DE FORA
JULHO, 2016

Simulação de Fluidos no Unity Utilizando o Método SPH

Claudio Henrique da Silva

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Bernardo Martins Rocha

JUIZ DE FORA

JULHO, 2016

Simulação de Fluidos no Unity Utilizando o Método SPH

Claudio Henrique da Silva

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Bernardo Martins Rocha
Doutor em Modelagem Computacional pelo LNCC

Rafael Alves Bonfim de Queiroz
Doutor em Modelagem Computacional pelo LNCC

Rodrigo Luis de Souza da Silva
Doutor em Engenharia pela UFRJ

JUIZ DE FORA

28 DE JULHO, 2016

Aos meus amigos, que sempre me deram forças para seguir em frente.

Resumo

Este trabalho possui como objetivo realizar um estudo sobre o método de simulação de fluidos SPH (*Smoothed-Particle Hydrodynamics*), que utiliza a forma Lagrangeana para representar um fluido. A fórmula de Navier-Stokes foi utilizada para relacionar a aceleração de uma partícula com forças de pressão e viscosidade. Para atualizar as velocidades e posições das partículas no decorrer da simulação foi utilizado o método integrador Leap-Frog. Algumas estratégias de otimização do SPH foram discutidas, explorando diferentes maneiras de aprimorar a busca por partículas vizinhas (etapa crucial do método SPH). Também foi discutido como colisões entre partículas e os limites do domínio da simulação podem ser tratados. No decorrer do desenvolvimento do trabalho foi implementada uma versão do SPH em C++. Após explorar técnicas de otimização de código, este programa foi integrado ao motor de jogos Unity. Para realizar esta integração foi criada uma biblioteca que é gerenciada pelo Unity através de um script escrito em C#.

Palavras-chave: SPH, fluidos, Unity, Leap-Frog.

Abstract

This work has the objective to conduct a study about the SPH (*Smoothed-Particle Hydrodynamics*) method of fluid simulation, which uses the Lagrangian form to represent a fluid. The Navier-Stokes equation was used to relate the acceleration of a particle with pressure and viscosity forces. To update the velocities and positions of the particles during the simulation it was used the Leap-Frog method of integration. Some strategies of optimization of the SPH method were discussed, exploring different ways to improve the search for neighboring particles (a crucial step of the SPH method). It has also been discussed as collisions between particles and the simulation domain boundaries can be handled. During the development of this work a version of the SPH was implemented in C++. After exploring code optimization techniques, this program has been integrated with Unity game engine. To accomplish this integration it was created a library which is managed by Unity through a script written in C#.

Keywords: SPH, fluids, Unity, Leap-Frog.

Agradecimentos

Ao professor Bernardo, pelos conselhos, pela amizade e pela orientação.

Á todos os meus professores e colegas de trabalho, que me ajudaram a construir meu perfil profissional, me inspiraram sempre a buscar conhecimento e me ajudaram a alcançar conquistas que antes eu nunca havia imaginado ser capaz de alcançar.

Aos meus amigos, por me apoiarem sempre.

Ao Batman, por ter nos salvado inúmeras vezes e por suas eternas contribuições a paz mundial.

"The great thing in this world is not so much where we stand, as in what direction we are moving".

Oliver Wendell Holmes Sr.

Sumário

Lista de Figuras **8**

Lista de Tabelas **9**

Lista de Abreviações **10**

1 Introdução **11**
1.1 Motivação . . . 11
1.2 Objetivos deste trabalho . . . 14

2 Fundamentação teórica **15**
2.1 Fluidos . . . 15
2.1.1 Representações Euleriana e Lagrangeana . . . 15
2.1.2 Propriedades . . . 16
2.1.2.1 Densidade . . . 16
2.1.2.2 Velocidade . . . 17
2.1.2.3 Viscosidade . . . 17
2.2 Smoothed-Particle Hydrodynamics (SPH) . . . 17
2.2.1 Modelo de partículas . . . 18
2.2.2 Equação de Navier-Stokes . . . 19
2.2.3 Forças de pressão e de viscosidade . . . 19
2.2.4 Leap-Frog . . . 23
2.2.5 Busca por partículas vizinhas . . . 24
2.2.5.1 Atualização preguiçosa . . . 25
2.2.5.2 Divisão do espaço em células . . . 25
2.2.5.3 Divisão do espaço utilizando estruturas adaptativas . . . 26
2.2.6 Condições de contorno . . . 28

3 Implementação **30**
3.1 Ambiente de desenvolvimento . . . 30
3.2 Estrutura de dados adotada inicialmente . . . 30
3.3 Diagrama de execução . . . 31
3.4 Otimizações . . . 32
3.4.1 Mudança de estrutura de dados . . . 33
3.4.2 Flags de otimização de código . . . 35
3.4.3 Otimização da busca por partículas vizinhas . . . 36
3.4.4 Remoção de chamadas da função *sqrt* . . . 36
3.4.5 Remoção de chamadas da função *pow* . . . 37
3.5 Diagrama final de classes e métodos . . . 39
3.6 Integração com Unity . . . 40
3.6.1 Biblioteca SPH . . . 40
3.6.2 Configuração dos parâmetros da biblioteca no Unity . . . 42
3.6.3 Coloração de partículas . . . 45
3.6.4 Dam Break . . . 47

4 Resultados **50**
4.1 Exemplos de simulações 50
4.1.1 Dam Break 50
4.1.2 Simulação em três dimensões 51
4.1.3 Rotação do cenário 52

5 Considerações Finais **55**
5.1 Conclusões 55
5.2 Trabalhos futuros 55

Referências Bibliográficas **57**

Lista de Figuras

1.1 Visão superior de pratos com diferentes líquidos sobre um projetor durante um *Liquid Light Show* 11
1.2 Captura de tela do motor gráfico Unity executando o *Digital Liquid Light* . 12
1.3 Duas capturas da tela principal do *Digital Liquid Light*. 13

2.1 Comparação entre a representação Euleriana e a Lagrangeana de um fluido (Gourlay, 2012). 16
2.2 Exemplo de partículas distribuídas aleatoriamente no espaço. 18
2.3 Linha do tempo e os pontos onde a velocidade e a posição são calculadas. . 24
2.4 Exemplo do espaço de uma simulação 2D de fluidos divido em células de tamanho kh. As células destacadas são células que contém possíveis vizinhos da partícula i. 26
2.5 Exemplo de distribuição de partículas utilizando uma *quadtree*. 27
2.6 Exemplo de colisão entre uma partícula e um objeto. 29

3.1 Diagrama de classes utilizadas na versão inicial da implementação. 32
3.2 Diagrama de execução do programa implementado. 33
3.3 Diagrama de classes utilizadas na versão final da implementação. 39
3.4 Como o programa implementado foi integrado no Unity. 41
3.5 Captura de tela do Unity. 43
3.6 Captura de tela da aba *Scene* do Unity exibindo os limites da simulação e a área em que as partículas serão distribuídas aleatoriamente. 44
3.7 Representação do modelo HSV em três dimensões (a) e em duas dimensões (b). 46
3.8 Exemplo de simulação de um fluido utilizando a coloração por densidade. . 46
3.9 Domínio de cores que uma partícula i movendo a uma velocidade com módulo v_i pode assumir ao ser renderizada pela coloração por velocidade. . 47
3.10 Captura de tela do Unity. A barreira da simulação *Dam Break* está posicionada na posição $x = 0.5$. 48
3.11 Captura de tela da aba *Inspector* do Unity, que fornece uma interface para o usuário configurar a simulação do fluido no Unity. 49

4.3 Captura de tela de uma simulação onde o usuário pode rotacionar todo o cenário da simulação. 52
4.1 Frames de uma simulação com *Dam Break* gerada no Unity. 53
4.2 Frames de uma simulação em três dimensões gerada no Unity utilizando a coloração de partículas por velocidade. 54

Lista de Tabelas

1.1 Lista de ações que são interpretadas como comandos no *Digital Liquid Light* 13

2.1 Lista de parâmetros das fórmulas da Seção 2.2.3 23

3.1 Como as coordenadas das partículas são armazenadas na memória 34
3.2 Impacto da alteração na estrutura de dados no tempo de execução do programa 35
3.3 Descrição das diferentes flags de otimização suportadas pelo compilador . . 35
3.4 Impacto da utilização da flag de otimização *o3* 36
3.5 Impacto da otimização da busca por partículas vizinhas no tempo de execução do programa 37
3.6 Impacto da remoção de chamadas da função *sqrt* no tempo de execução do programa 38
3.7 Impacto da remoção de chamadas da função *pow* no tempo de execução do programa 38

4.1 Parâmetros utilizados para criar a simulação exibida na Figura 4.1 50
4.2 Parâmetros utilizados para criar a simulação exibida na Figura 4.2 51

Lista de Abreviações

AoS	Array of Structures
CUDA	Compute Unified Device Architecture
EDO	Equações Diferenciais Ordinárias
GPGPU	General-purpose computing on graphics processing units
FPS	Frames Por Segundo
GPU	Graphics Processing Unit
HSV	Hue, Saturation and Value
IDE	Integrated Development Environment
REV	Representative Element of Volume
SoA	Structure of Arrays
RGB	Red, Green and Blue
SPH	Smoothed-Particle Hydrodynamics

1 Introdução

1.1 Motivação

Liquid Light Show é uma técnica que cria animações de luzes e líquidos que foi muito utilizada em apresentações musicais dos anos 60. Para realizar as animações, o operador utilizava pratos transparentes sobre um projetor e adicionava líquidos de cores e viscosidades diferentes entre os pratos. Deste modo é possível manipular a imagem projetada conforme os pratos são movimentados e rotacionados. O operador então estava constantemente adicionando (e ás vezes removendo) líquidos e manipulando os pratos para criar uma animação que não apenas seguisse o ritmo da música que estava sendo tocada no momento, mas também que impressionasse a audiência. A Figura 1.1 mostra diversos líquidos adicionados sobre um prato durante um *Liquid Light Show*.

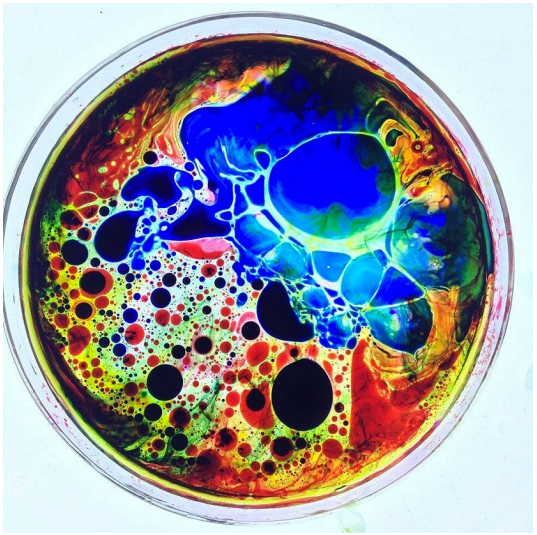


Figura 1.1: Visão superior de pratos com diferentes líquidos sobre um projetor durante um *Liquid Light Show* .

Esta técnica foi a principal inspiração para que, durante o segundo semestre de 2014 o autor desenvolvesse, sob orientação do professor Kenneth Bowen (University of Connecticut), o projeto *Digital Liquid Light*, que tinha como objetivo desenvolver uma aplicação que permitisse a uma pessoa interagir com líquidos que eram simulados em um

ambiente 2D. Para isto era utilizado um sensor de movimentos Kinect (Microsoft, 2016).

O *Digital Liquid Light* captura os movimentos do corpo do jogador permitindo que ele possa adicionar líquidos na tela, personalizar a aparência dos líquidos (viscosidade e cor), empurrar os líquidos com as mãos e também remover os líquidos da tela. Desta forma o jogador poderia criar seu próprio show de líquidos. Quando não havia nenhuma pessoa na frente do sensor para interagir com a aplicação o microfone do computador era utilizado para fazer com que os fluidos da simulação sofressem alterações de acordo com os sons do ambiente.

Na Figura 1.2 é possível visualizar como a aplicação utilizava as informações capturadas pelo Kinect. À direita está o esqueleto do jogador capturado pelo sensor de movimentos Kinect. À esquerda partes específicas do corpo, como mãos e cabeça, são representadas por esferas. Um algoritmo é responsável por analisar os posicionamentos destas esferas para identificar os comandos executados pelo jogador (veja a lista completa de comandos compatíveis na Tabela 1.1). Ao fundo da figura estão diversos fluidos com cores e viscosidades diferentes, sendo manipulados pelo jogador. O esqueleto do usuário e sua "representação em esferas" não são visíveis ao usuário: a Figura 1.3 mostra duas capturas da tela principal da aplicação, em momentos diferentes, onde o jogador só pode visualizar os fluidos.

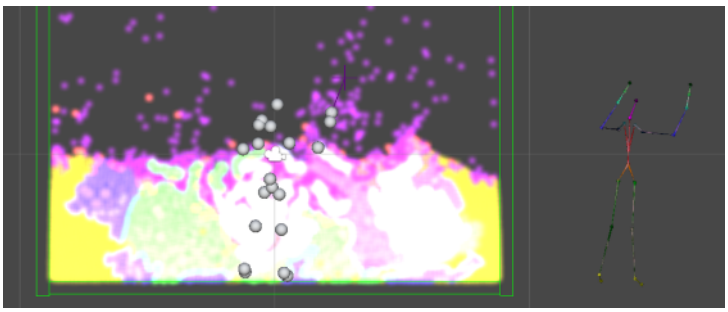


Figura 1.2: Captura de tela do motor gráfico Unity executando o *Digital Liquid Light*

A aplicação foi desenvolvida no motor gráfico Unity (Unity Technologies, 2016) e era compatível com o sistema operacional Windows (Microsoft, 2016). Unity é uma plataforma de desenvolvimento de jogos criada em 2005 que inicialmente suportava apenas o sistema OS X (Apple, 2016), mas que expandiu sua lista de plataformas com o decorrer dos anos, sendo hoje compatível com 21 plataformas diferentes, incluindo diferentes sis-

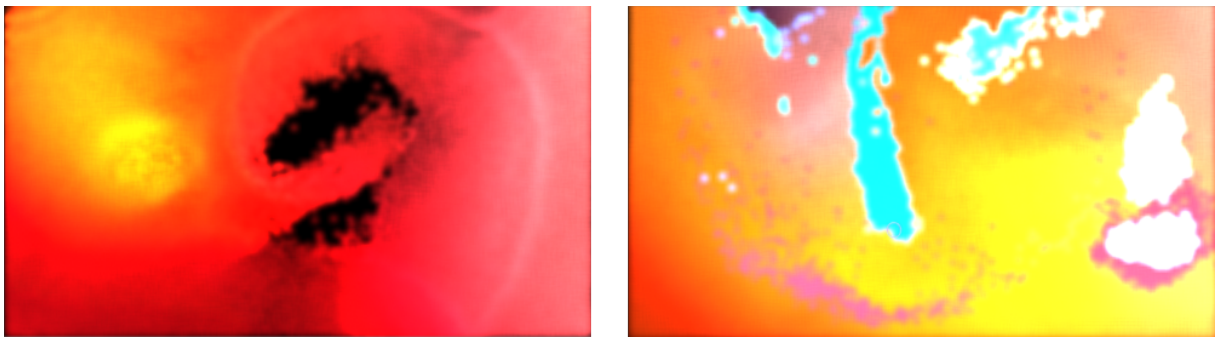


Figura 1.3: Duas capturas da tela principal do *Digital Liquid Light*.

Tabela 1.1: Lista de ações que são interpretadas como comandos no *Digital Liquid Light*

Ação	**Comando**
Colocar mão esquerda sobre a cabeça	Alterar a cor do fluido.
Tocar ombro esquerdo com a mão direita	Alternar entre os dois modos disponíveis: o primeiro permite que o jogador "empurre" os fluidos com as mãos, criando "ondas", e o segundo permite que o usuário adicione novos fluidos na tela.
Bater palmas	Alternar entre propriedades (como viscosidade) de fluidos previamente pré-definidas.
Tocar o ombro direito com a mão esquerda	Remover todos os fluidos da tela.

temas operacionais para computadores, consoles de videogame, navegadores de internet, plataformas móveis e também dispositivos de realidade aumentada. Esta característica é muito atrativa para desenvolvedores de jogos, pois ela permite que o desenvolvedor utilize o mesmo projeto para publicar seu jogo em todas as plataformas líderes do mercado, sem precisar re-escrever todo o código do jogo. Atualmente Unity é software líder global da indústria de jogos e possui diferentes licenças de uso, incluindo uma totalmente gratuita. Estas qualidades, juntamente com os recursos avançados oferecidos pelo motor, incentivaram seu uso não apenas por desenvolvedores de jogos, mas também por profissionais de arquitetura, engenharia e construção.

O grande desafio durante o desenvolvimento do projeto foi realizar uma simulação fiel dos líquidos. Para facilitar e agilizar o desenvolvimento do projeto, foi utilizado um plugin desenvolvido por terceiros para realizar a simulação dos fluidos. Infelizmente, não haviam muitas opções disponíveis no mercado de plugins para Unity, e foi adotado o plugin que mais se adequou aos objetivos do projeto: o *Liquid Physics 2D* (Physical

Liquid Software, 2016), que possui como núcleo a biblioteca *LiquidFun* (Google, 2016), desenvolvido em C++.

O plugin trouxe grandes contribuições ao projeto, entretanto ele possuía algumas desvantagens que limitaram seu desenvolvimento. O plugin realizava simulações apenas em duas dimensões, o que impediu que ideias de novos efeitos (que utilizariam partículas espalhadas nos três eixos x, y e z) fossem implementadas. Ele também não era otimizado para o Unity e possuía alguns problemas de desempenho.

1.2 Objetivos deste trabalho

As dificuldades encontradas durante o desenvolvimento do *Digital Liquid Light* inspiraram a criação do presente trabalho, que possui como principal objetivo estudar como uma simulação de fluidos é realizada e, no decorrer dos estudos, implementar uma nova ferramenta para simulação de fluidos incompreensíveis em 2D e 3D.

Esta nova implementação será integrada no motor Unity no formato de plugin, permitindo que qualquer pessoa possa realizar simulações de fluidos. Através de uma interface amigável ao usuário, ele poderá definir os parâmetros da simulação e observar como o fluido se comporta com parâmetros diferentes.

As opções de visualização do editor do Unity permitirão ao usuário visualizar o fluido de diversos pontos de vista diferentes, rotacionando e aplicando zoom no ambiente, afim de se obter uma boa visualização do fluido observado. A ferramenta poderá ser utilizada não apenas para criar efeitos em jogos mas também para testar como fluidos se comportam com diversos parâmetros diferentes.

O plugin poderá ser utilizado por usuários que utilizam sistema operacional Linux, OS X ou Windows.

2 Fundamentação teórica

2.1 Fluidos

Um fluido pode ser definido como qualquer substância que pode assumir a forma de seu recipiente e que não resiste a deformações de cisalhamento. Até mesmo uma força muito pequena causa deformação na partícula de um fluido (Ferziger e Peric, 2001; Pritchard e McDonald, 2011).

A propriedade de um fluido poder assumir a forma de seu recipiente justifica a existência de técnicas específicas para simulação de fluidos: como fluidos assumem a forma de seu recipiente, um fluido está em colisão com tudo ao seu redor, inclusive com ele mesmo. Ou seja, se houver uma colisão de um objeto com uma parte do fluido, todo o corpo do fluido deverá responder a esta colisão. É importante ressaltar que o termo fluido pode ser aplicado tanto para líquidos quanto para gases, e embora cada um possa ser melhor representado em um modelo diferente, ambos possuem a mesma estrutura e compartilham propriedades similares (Gourlay, 2012).

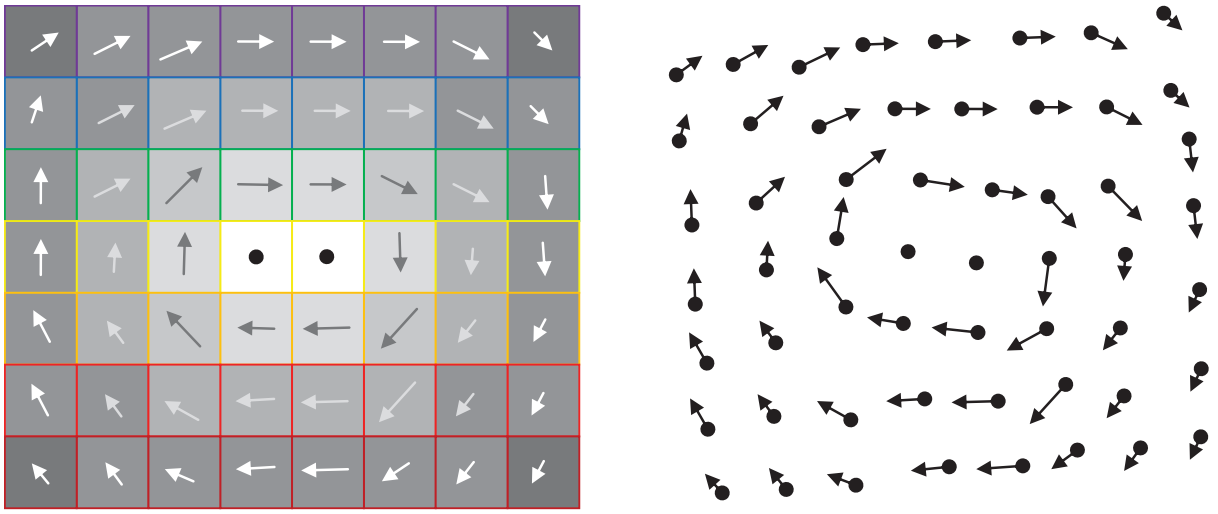
2.1.1 Representações Euleriana e Lagrangeana

Segundo Gourlay (2012) existem duas formas de representar um fluido: a forma *Euleriana* e a *Lagrangeana*. Na forma Euleriana o espaço que contém o fluido é dividido em uma malha (grade) onde cada região possui propriedades físicas como velocidade, densidade, temperatura e pressão. Neste modelo há a ausência da propriedade *posição*, pois cada região possui posição fixa.

Já na forma Lagrangeana, o fluido pode ser representado por um conjunto de partículas que, em adição as propriedades que uma região do modelo Euleriano possui (velocidade, densidade, temperatura e pressão), também possui a propriedade posição.

Métodos de simulação que utilizam a forma Euleriana geralmente demandam um alto esforço computacional, pois exigem um pré-processamento da malha (que representa

todo o espaço ocupado pelo fluido) e também uma constante atualização desta malha. Em contrapartida, métodos que utilizam a forma Lagrangeana exigem apenas um método eficiente de se detectar partículas vizinhas de uma dada partícula (Andrade, 2009). Por este motivo foi escolhido utilizar neste trabalho o método SPH (*Smoothed-Particle Hydrodynamics*), que é um método Lagrangeano. A Figura 2.1 (a) mostra a representação de um líquido na forma Euleriana, e (b) mostra a representação do mesmo fluido, porém utilizando a forma Lagrangeana.



(a) (b)

Figura 2.1: Comparação entre a representação Euleriana e a Lagrangeana de um fluido (Gourlay, 2012).

2.1.2 Propriedades

Renhe (2010); Pritchard e McDonald (2011) utilizam a *Hipótese do Contínuo* para realizar uma abstração de fluidos. Através deste modelo o fluido é representado por partículas discretas, que são consideradas um elemento representativo de volume (ou REV, *Representative Element of Volume*). Para um REV cada propriedade é representada através de um valor médio.

2.1.2.1 Densidade

Densidade é uma grandeza que representa a quantidade de massa presente em uma unidade de volume. Pela Hipótese do Contínuo, cada REV possui um valor definido para

densidade.

2.1.2.2 Velocidade

A velocidade é uma grandeza vetorial (possui módulo e sentido) que em um ponto qualquer do espaço do fluido, representa a velocidade instantânea de uma partícula que passa por este ponto durante um intervalo de tempo (Δt).

2.1.2.3 Viscosidade

Quanto maior a viscosidade de um fluido, maior sua resistência contra deformações por cisalhamento. Por exemplo, líquidos mais viscosos como o óleo são mais difíceis de serem derramados que líquidos menos viscosos como a água.

2.2 Smoothed-Particle Hydrodynamics (SPH)

Smoothed-Particle Hydrodynamics (SPH) é um método de interpolação de sistemas de partículas. Desde sua criação em 1977 o modelo foi explorado por diversos pesquisadores e, atualmente, é utilizado em problemas astrofísicos, mecânica dos sólidos e simulação de fluidos. Seu alto desempenho e alta fidelidade em suas simulações fez com que fosse adotado em diversos produtos da indústria do entretenimento, como por exemplo nos filmes *Ice Age* (Twentieth Century Fox Film, 2002), *The Lord of the Rings: The Return of the King* (New Line Productions, 2003) e *Superman Returns* (Warner Bros., 2006), e nos jogos *Alice: Madness Returns* (Electronic Arts, 2011) e *Borderlands 2* (Gearbox Software, 2012), (Green et al, 2011; Robb, 2012; Laursen, 2013).

Através do SPH as grandezas são distribuídas em uma vizinhança local de cada partícula utilizando núcleos simétricos radiais de suavização. De acordo com o SPH, uma grandeza escalar A é interpolada no local r por uma soma ponderada de contribuição de todas as partículas:

$$A_S(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h), \tag{2.1}$$

onde j itera através de todas as partículas do sistema, m_j é a massa da partícula j, r_j sua posição, ρ_j sua densidade e A_j o valor da grandeza em r_j. A função $W(r, h)$ é chamada

de núcleo de suavização com núcleo de raio h (Müller et al, 2003). A Equação (2.1) nos permite obter o valor da densidade na posição r:

$$\rho_S(r) = \sum_j m_j \frac{\rho_j}{\rho_j} W(r - r_j, h) = \sum_j m_j W(r - r_j, h). \tag{2.2}$$

2.2.1 Modelo de partículas

Os fluidos são representados por *partículas*, e cada partícula possui sua própria posição no espaço, densidade, velocidade e massa. Todo o sistema é representado por um número finito de partículas, distribuídas aleatoriamente no domínio do problema (Paiva et al, 2009). A Figura 2.2 mostra 2000 partículas distribuídas aleatoriamente no espaço.

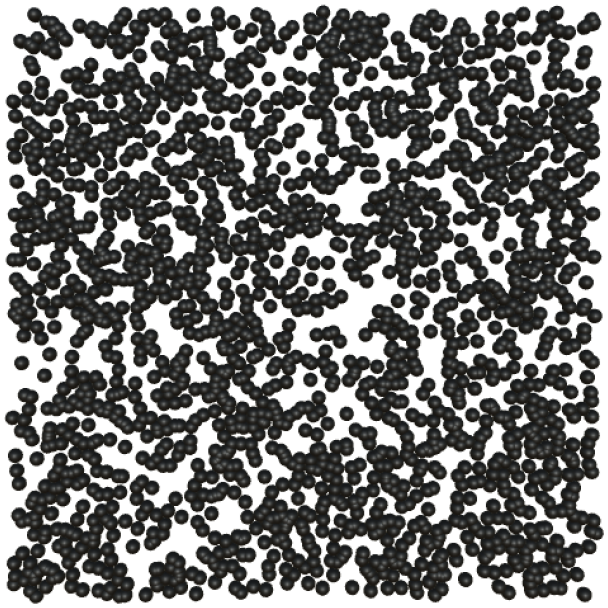


Figura 2.2: Exemplo de partículas distribuídas aleatoriamente no espaço.

A cada iteração o algoritmo atualiza as posições de cada partícula do fluido (mais detalhes de como esta atualização é realizada serão discutidos nas próximas seções), simulando assim o movimento do fluido.

É importante ressaltar que uma partícula possui propriedades como posição no espaço, massa, velocidade e densidade, mas a princípio não possui tamanho ou forma. Geralmente utilizam-se as posições das partículas apenas para renderizar a superfície do líquido. Durante o desenvolvimento deste trabalho partículas foram representadas por esferas apenas como forma de abstração.

2.2.2 Equação de Navier-Stokes

A Equação (2.3) representa a Equação de Navier-Stokes para fluidos incompressíveis, que formula a conservação de momento (Pnueli e Gutfinger, 1997; Müller et al, 2003):

$$\rho\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right) = -\nabla p + \rho g + \mu \nabla^2 v, \tag{2.3}$$

onde g é um campo de densidade de força externa e μ a viscosidade do fluido. Como as partículas movem com o fluido, a derivada substancial do campo da velocidade é simplesmente a derivada da velocidade das partículas pelo tempo, o termo convectivo $v \cdot \nabla v$ não é necessário para sistemas de partículas. A derivada da velocidade em função do tempo nada mais é do que a aceleração, então temos que $\frac{\partial v}{\partial t} = a$.

Na parte direita da Equação (2.3), $-\nabla p$ representa a força de pressão, ρg as forças externas e $\mu \nabla^2 v$ a força viscosidade. Podemos representar a soma destes três campos de densidade de forças da seguinte forma: $f = -\nabla p + \rho g + \mu \nabla^2 v$.

Portanto podemos reescrever a Equação (2.3) da seguinte forma:

$$\rho_i \frac{\partial v_i}{\partial t} = f_i \rightarrow \rho_i a_i = f_i \rightarrow a_i = \frac{f_i}{\rho_i}, \tag{2.4}$$

onde v_i é a velocidade da partícula i, e ρ_i e f_i são o campo da densidade e o campo de densidade de força avaliados no local da partícula i, respectivamente.

2.2.3 Forças de pressão e de viscosidade

Desconsiderando as forças externas do fluido (como por exemplo força da gravidade e forças de colisão), as forças que atuam em cada partícula são as forças de pressão (f^p) e de viscosidade (f^v). Para fins de organização, todos os parâmetros das equações envolvidas no cálculo destas forças são descritos na Tabela 2.1. Conforme discutido anteriormente, $-\nabla p$ representa a força de pressão, e é possível utilizar a Equação (2.1) para obter o valor desta força:

$$f_i^p = -\nabla p(r_i) = \sum_j m_j \frac{p_j}{\rho_j} \nabla W(r_i - r_j, h). \tag{2.5}$$

A Equação (2.5) não é simétrica: quando duas partículas interagem, o gradiente

do núcleo ($\nabla W(r_i - r_j, h)$) é zero no centro das partículas e portanto a partícula i utiliza a pressão da partícula j para calcular sua própria pressão, e vice-versa. Como a pressão geralmente não é igual nos locais das duas partículas, as forças de pressão não serão simétricas. Existem diversas formas de simetrizar esta equação na literatura. Müller et al (2003) propuseram a seguinte solução:

$$f_i^p = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(r_i - r_j, h), \tag{2.6}$$

que representa uma equação simétrica, pois utiliza a média aritmética da pressão das duas partículas que estão interagindo ($\frac{p_i+p_j}{2}$). A pressão de uma partícula pode ser obtida através da equação de estado de um gás ideal:

$$p_i = k\rho_i, \tag{2.7}$$

onde k é uma constante do gás que varia com a temperatura. Desbrun e Cani (1996) sugeriram a utilização de uma equação de estado modificada para simulações de fluidos:

$$p_i = k(\rho_i - \rho_0), \tag{2.8}$$

onde ρ_0 é a densidade de repouso do fluido.

Na equação de Navier-Stokes representada pela Equação (2.3) a força de viscosidade é definida como $\mu\nabla^2 v$. Podemos obter uma fórmula para esta força utilizando novamente a Equação (2.1):

$$f_i^v = \mu\nabla^2 v(r_i) = \mu \sum_j m_j \frac{v_j}{\rho_j} \nabla^2 W(r_i - r_j, h), \tag{2.9}$$

que também não é uma equação simétrica, pois a velocidade varia de partícula para partícula. Müller et al (2003) propuseram uma simetrização da Equação (2.9) utilizando diferenças de velocidade (pois as forças de viscosidade são dependentes apenas de diferen-

ças de velocidade e não de velocidades absolutas):

$$f_i^v = \mu \sum_j m_j \frac{v_j - v_i}{\rho_j} \nabla^2 W(r_i - r_j, h). \tag{2.10}$$

Segundo Crespo (2008) e Andrade (2009), a função núcleo do SPH possui grande impacto na simulação, cuja precisão numérica e estabilidade é diretamente afetada pelo núcleo utilizado. Müller et al (2003) propuseram o uso dos núcleos $W_{poly6}(r,h)$ e $W_{spiky}(r,h)$ para o cálculo das forças de pressão e $W_{viscosity}(r,h)$ para o cálculo das forças de viscosidade:

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & \text{se } 0 \leq r \leq h, \\ 0 & \text{caso contrário.} \end{cases} \tag{2.11}$$

$$W_{spiky}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & \text{se } 0 \leq r \leq h, \\ 0 & \text{caso contrário.} \end{cases} \tag{2.12}$$

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & \text{se } 0 \leq r \leq h, \\ 0 & \text{caso contrário.} \end{cases} \tag{2.13}$$

Na metodologia adotada foi considerado que N_i representa o conjunto de todas as partículas j cuja distância para a partícula i é menor que h, ou seja:

$$j \in N_i \text{ se e somente se } r < h. \tag{2.14}$$

O núcleo descrito pela Equação (2.11) foi aplicado na fórmula da densidade da partícula i, obtendo-se:

$$\rho_i = \frac{4m}{\pi h^8} \sum_j (h^2 - r^2)^3. \tag{2.15}$$

Aplicando-se a Equação (2.8) e os núcleos $W_{spiky}(r,h)$ (2.12) e $W_{viscosity}(r,h)$ (2.13) nas fórmulas de f_i^p (2.6) e f_i^v (2.10), obtêm-se:

$$f_i^p = \frac{15k}{4h^4} \sum_j m_j \frac{\rho_i + \rho_j - 2\rho_0}{\rho_j} \frac{(1 - q_{ij})^2}{q_{ij}} r_{ij} \text{ e} \tag{2.16}$$

$$f_i^v = \frac{40\mu}{\pi h^4} \sum_j m_j \frac{v_j - v_i}{\rho_j}(1 - q_{ij}), \tag{2.17}$$

onde

$$r_{ij} = r_i - r_j \text{ e} \tag{2.18}$$

$$q_{ij} = \frac{||r_{ij}||}{h}. \tag{2.19}$$

Com a finalidade de simplificar o código da implementação e aumentar seu desempenho, f_i^p foi somada a f_i^v (cujo resultado será definido por f_i^{p+v}). Deste modo é necessário executar apenas um laço (*loop*) para cobrir todas as partículas em N_i e calcular a contribuição das forças de pressão e viscosidade:

$$f_i^{p+v} = f_i^p + f_i^v \tag{2.20}$$

$$f_i^{p+v} = \frac{15k}{4h^4} \sum_j m_j \frac{\rho_i + \rho_j - 2\rho_0}{\rho_j} \frac{(1 - q_{ij})^2}{q_{ij}} r_{ij} + \frac{40\mu}{\pi h^4} \sum_j m_j \frac{v_j - v_i}{\rho_j}(1 - q_{ij}). \tag{2.21}$$

É possível simplificar a Equação (2.21) agrupando-se os termos e considerando que $v_j - v_i = -(v_i - v_j) = -v_{ij}$ e que todas as partículas possuem a mesma massa ($m_j = m$):

$$f_i^{p+v} = \frac{m}{4\pi h^4 \rho_j} \sum_j (1 - q_{ij}) \left[15k(\rho_i + \rho_j - 2\rho_0) \frac{(1 - q_{ij})}{q_{ij}} r_{ij} - 40\mu v_{ij} \right]. \tag{2.22}$$

Considerando que a soma de todas as forças externas que atuam sobre a partícula i, como a força da gravidade, seja dada por f_i^e, temos que $f_i = f_i^{p+v} + f_i^e$. Na implementação realizada durante este trabalho, foi considerada uma única força externa, a força da gravidade. Para facilitar os cálculos, adicionamos a contribuição da força da gravidade diretamente sobre a aceleração da partícula (por exemplo, adicionando $9,8 m/s^2$ à aceleração resultante das forças de pressão e de viscosidade). Portanto neste caso é possível considerar $f_i^e = 0$ o que implica que $f_i = f_i^{p+v}$. Para obter a aceleração da partícula i, a Equação (2.22) foi aplicada na Equação (2.4):

$$a_i = \frac{f_i}{\rho_i} = \frac{1}{\rho_i} \left[\frac{m}{4\pi h^4 \rho_j} \sum_j (1 - q_{ij}) \left[15k(\rho_i + \rho_j - 2\rho_0) \frac{(1 - q_{ij})}{q_{ij}} r_{ij} - 40\mu v_{ij} \right] \right]. \tag{2.23}$$

A velocidade e a aceleração de uma partícula podem ser obtidas através da derivada de sua posição ($v_i = \frac{\partial r_i}{\partial t}$) e da derivada de sua velocidade ($a_i = \frac{\partial v_i}{\partial t}$), respectivamente. Portanto é possível definir o seguinte sistema de Equações Diferenciais Ordinárias (EDO):

$$\begin{cases} \frac{\partial v_i}{\partial t} = \frac{f_i}{\rho_i} \\ \frac{\partial r_i}{\partial t} = v_i \end{cases}, \tag{2.24}$$

que ao ser aplicado o Método de Euler (Atkinson, 1989) se obtém as equações necessárias para atualizar a posição da partícula i após um instante de tempo Δt:

$$\begin{cases} v_i^{n+1} = v_i^n + \Delta t \frac{f_i^n}{\rho_i^n} \\ r_i^{n+1} = r_i^n + \Delta t v_i^n \end{cases} \rightarrow \begin{cases} v_i^{n+1} = v_i^n + \Delta t a_i^n \\ r_i^{n+1} = r_i^n + \Delta t v_i^n \end{cases}, \tag{2.25}$$

onde v_i^n, r_i^n, a_i^n, f_i^n e ρ_i^n representam, respectivamente, a velocidade, a posição, a aceleração, o campo de densidade de força, e a densidade da partícula i no instante n, e Δt o intervalo de tempo entre os instantes n e $n+1$.

Tabela 2.1: Lista de parâmetros das fórmulas da Seção 2.2.3

Parâmetro	**Descrição**
m_j	Massa da partícula j
p_i	Pressão da partícula j
ρ_j	Densidade da partícula j
$W(r,h)$	Núcleo de suavização do SPH com núcleo de raio h
r_i	Vetor posição da partícula i
h	Comprimento suave do SPH
μ	Viscosidade do fluido
v_j	Vetor velocidade da partícula j
ρ_0	Densidade de repouso do fluido
k	Constante do gás
r	Distância entre as partículas i e j (ou seja $r = \lvert r_i - r_j \rvert$)

2.2.4 Leap-Frog

As equações do sistema (2.25) representam o método de Euler, considerado o mais simples conhecido atualmente. *Leap-Frog* é um método integrador que possui uma simplicidade similar ao método de Euler, porém ele possui uma precisão de segunda ordem. Este método define as seguintes equações para calcular a velocidade e a posição de uma partícula

(Andrade, 2009; Paiva et al, 2009):

$$v_i^{t+\frac{1}{2}\Delta t} = v_i^{t-\frac{1}{2}\Delta t} + \Delta t a_i^t \text{ e} \tag{2.26}$$

$$r_i^{t+\Delta t} = r_i^t + \Delta t v_i^{t+\frac{1}{2}\Delta t}, \tag{2.27}$$

sendo v_i^t e r_i^t definidos como a velocidade e a posição da partícula i durante o instante t, respectivamente. É importante ressaltar que a velocidade inicial deve ser calculada utilizando o método de Euler.

O método recebeu o nome de *Leap-Frog* ("pulo de sapo" em tradução literal) pois calcula a velocidade e a posição em pontos de tempo intervalados, de forma que a velocidade "salta" na frente da posição e vice-versa (este comportamento é representado na Figura 2.3).

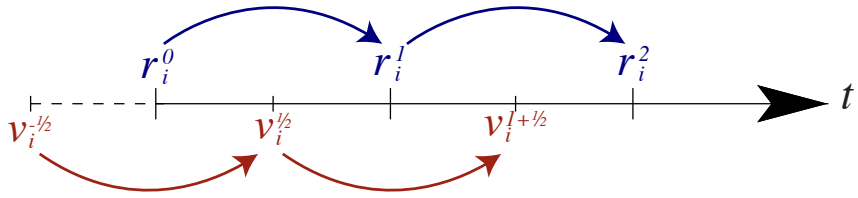


Figura 2.3: Linha do tempo e os pontos onde a velocidade e a posição são calculadas.

Para obter a velocidade durante o instante t, basta realizar uma simples aproximação:

$$v_i^t \approx \frac{v_i^{t-\frac{1}{2}\Delta t} + v_i^{t+\frac{1}{2}\Delta t}}{2}. \tag{2.28}$$

O método *Leap-Frog* foi utilizado neste projeto por sua precisão e estabilidade, assim como em outros trabalhos (Müller et al, 2003; Paiva et al, 2009).

2.2.5 Busca por partículas vizinhas

Conforme discutido anteriormente, uma partícula sofre forças de partículas vizinhas, afetando seu comportamento. Determinar quais partículas são vizinhas pode se tornar uma tarefa árdua, visto que é comum utilizar um grande número de partículas em simulações de fluidos e que todas as partículas estão em movimento, fazendo com que o conjunto de partículas vizinhas possa sofrer alterações a cada iteração.

Um algoritmo que executa a busca analisando todas as n partículas do sistema possui complexidade $O(n^2)$. É possível otimizar este algoritmo explorando a simetria entre as forças que atuam entre duas partículas: a força (de pressão ou de viscosidade) que a partícula i realiza sobre a partícula j (definida por $\overrightarrow{f_{i,j}}$) possui mesmo módulo da força que j realiza sobre i (definida por $\overrightarrow{f_{j,i}}$), ou seja $f_{i,j} = f_{j,i}$. Esta estratégia diminui o número de operações necessárias para calcular as forças que atuam sobre todas as partículas, mas não representa uma grande melhoria no tempo de execução do algoritmo se comparada a outras técnicas de otimização. Andrade (2009) descreve três formas eficientes de encontrar partículas vizinhas:

2.2.5.1 Atualização preguiçosa

A ideia desta estratégia é calcular quais são os vizinhos de uma partícula apenas quando seus vizinhos mudam. Isto só acontece se a partícula movimentar uma distância superior a $\frac{1}{2}h$. Sendo t o tempo médio que o conjunto de vizinhos de uma partícula sofre alteração, se t for comparável com o número de partículas, a busca por força bruta com atualização preguiçosa pode ser eficiente. Em geral este método é mais eficiente para partículas que se movem pouco (fazendo com que a busca por partículas vizinhas seja realizada raramente).

2.2.5.2 Divisão do espaço em células

Nesta técnica todo o espaço da simulação é dividido em células de mesma largura e altura, de forma regular, se tornando uma grade. Utilizando-se células de tamanho kh (onde k é um inteiro tal que $k > 0$), todas as partículas vizinhas de uma partícula p estão ou na mesma célula que p ou em uma célula adjacente à célula de p. Isto reduz consideravelmente o espaço de busca, pois descarta partículas que estão a uma distância maior que o raio de influência do SPH.

Deste modo a busca de vizinhos fica restrita a 3^d células, onde d é o número de dimensões do sistema. Ou seja, para simulações 2D é necessário analisar todas as partículas contidas em $3^2 = 9$ células da grade, e para simulações 3D é necessário analisar $3^3 = 27$ células. A Figura 2.4 mostra um exemplo de divisão do espaço de uma simulação 2D em células, destacando as 9 células que podem conter vizinhos da partícula p. Sendo

(p_x, p_y) as coordenadas da partícula p, os índices $[i, j]$ da célula que contém p são dados por:

$$i = \frac{p_x}{kh} \text{ e} \tag{2.29}$$

$$j = \frac{p_y}{kh}. \tag{2.30}$$

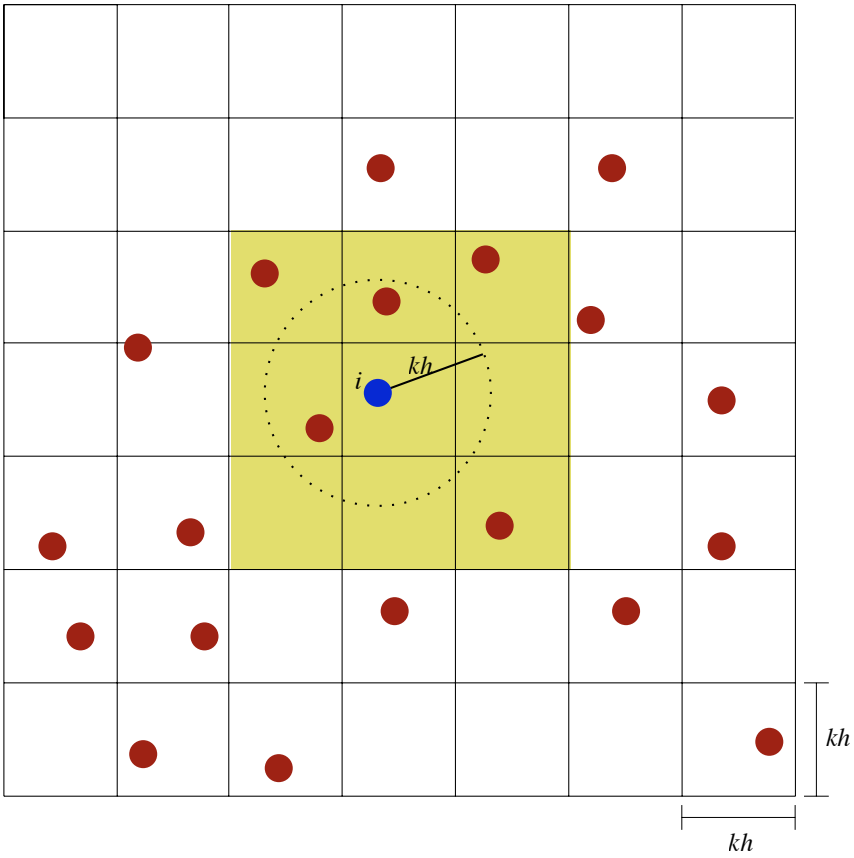


Figura 2.4: Exemplo do espaço de uma simulação 2D de fluidos divido em células de tamanho kh. As células destacadas são células que contém possíveis vizinhos da partícula i.

A grande desvantagem de dividir o espaço da simulação em células é que em simulações onde as partículas ficam mais dispersas muitas células ficam vazias, enquanto outras podem ficar com muitas partículas, fazendo com que, possivelmente, o método perca sua eficiência.

2.2.5.3 Divisão do espaço utilizando estruturas adaptativas

Esta técnica é muito parecida com a *Divisão do espaço em células* (Seção 2.2.5.2), contendo a mesma ideia de restringir o campo de busca dividindo-se o domínio da simulação, porém

utiliza uma estrutura adaptativa para tal. Para simulações em 2D geralmente é utilizada uma estrutura de dados conhecida como *quadtree*, e em simulações 3D é utilizada uma *octree* (Andrade, 2009).

Quadtree é uma árvore em que cada nó interno possui 4 filhos e cada nó representa um quadrado. Se um nó v possui filhos, então seus quadrados correspondentes são os quatro quadrantes do quadrado representado por v. *Quadtrees* representam um espaço 2D, mas podem ser generalizadas para dimensões maiores, sendo então chamadas de *octree* (Berg, 2008).

O espaço da simulação é representado por uma única célula que contém todas as partículas da simulação. Esta célula é subdividida recursivamente de forma que cada subdivisão contenha um número máximo de partículas por célula. A Figura 2.5 contém um exemplo de distribuição de partículas desta forma.

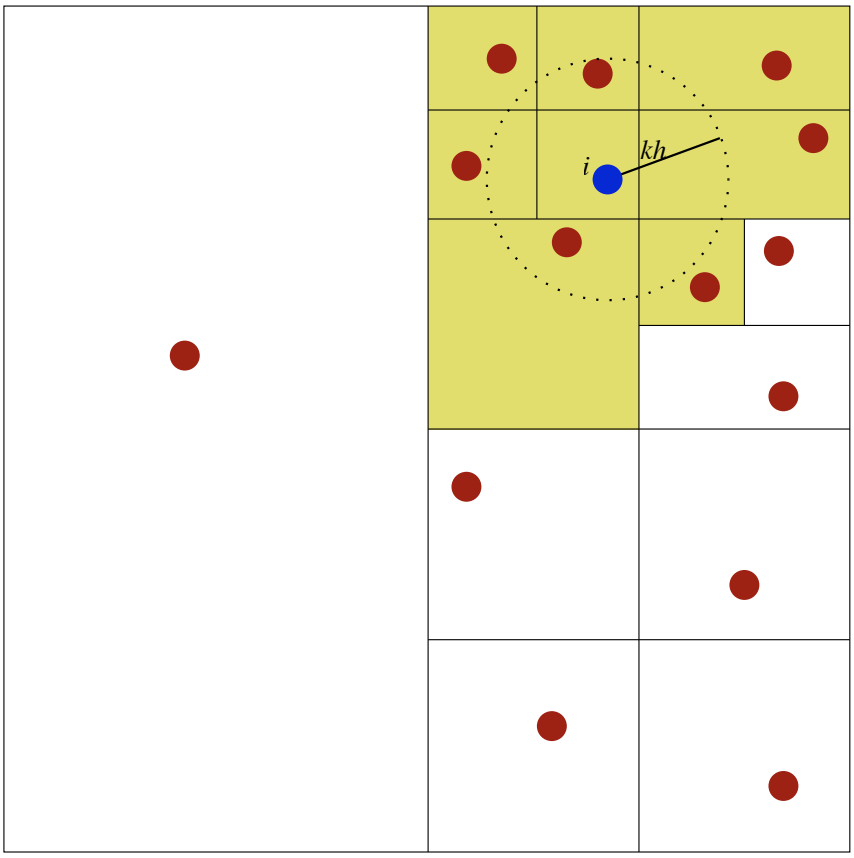


Figura 2.5: Exemplo de distribuição de partículas utilizando uma *quadtree*.

Esta técnica possui como desvantagem o fato de que quando a dispersão de partículas é muito grande, o consumo de memória é maior do que o consumo de uma grade

regular (discutida na seção anterior). Realizar o percurso em árvore também possui custo computacional maior do que se acessar uma célula de uma grade diretamente. Durante a simulação as partículas estão sempre se movimentando, sendo necessário atualizar a estrutura adotada constantemente, portanto é importante notar que a criação e manutenção de uma grade é mais simples que a de uma estrutura adaptativa.

2.2.6 Condições de contorno

Quando uma partícula colide com um objeto sólido (por exemplo as paredes do cenário de simulação), ela é empurrada na direção oposta do objeto e a componente de sua velocidade que é perpendicular à superfície do objeto é refletida (Müller et al, 2003).

Então, quando a colisão é detectada, a partícula precisa "voltar" na direção da sua posição original. O movimento da partícula i, que possui posição r_i e velocidade v_i é representado pela equação: $r_i = r_i + v_i \Delta t$. Explicitando as componentes x, y e z dos vetores se obtém:

$$\left.\begin{aligned} r_{i,x} &= r_{i,x} + v_{i,x}\Delta t \\ r_{i,y} &= r_{i,y} + v_{i,y}\Delta t \\ r_{i,z} &= r_{i,z} + v_{i,z}\Delta t \end{aligned}\right\} \rightarrow r_{i,eixo} = r_{i,eixo} + v_{i,eixo}\Delta t. \tag{2.31}$$

Para empurrar a partícula de volta, precisamos saber qual é o instante em que ocorreu a colisão (t_{col}). É possível explicitar a variação de tempo na fórmula da velocidade, $v = \frac{\Delta x}{\Delta t} \rightarrow \Delta t = \frac{\Delta x}{v}$. Sendo s o vetor que representa a posição da superfície do objeto colidido, e $eixo\perp$ o eixo perpendicular à superfície deste objeto, t_{col} pode ser obtido da seguinte forma:

$$\Delta t = \frac{\Delta x}{v} \rightarrow t_{col} - 0 = \frac{r_{i,eixo\perp} - s_{eixo\perp}}{v_{i,eixo\perp}} \rightarrow t_{col} = \frac{r_{i,eixo\perp} - s_{eixo\perp}}{v_{i,eixo\perp}}. \tag{2.32}$$

Após obter o valor de t_{col}, para empurrar a partícula é necessário realizar a seguinte operação sobre as três componentes do vetor posição da partícula:

$$r_i = r_i - v_i t_{col}. \tag{2.33}$$

A equação abaixo inverte a componente da velocidade da partícula que é perpendicular à superfície do objeto colidido com a partícula:

$$v_{i,eixo\perp} = -v_{i,eixo\perp}. \tag{2.34}$$

Quando uma partícula colide com um objeto, sua velocidade pode ser reduzida devido a fenômenos físicos como aquecimento, deformação e som provocados pelo impacto. Esta redução é controlada pelo parâmetro de *amortecimento* (representado por α) que é um número real tal que $0 \leq \alpha \leq 1$. Portanto, para concluir o tratamento da colisão é realizada a seguinte operação:

$$v_i = v_i\alpha. \tag{2.35}$$

A Figura 2.6 representa uma colisão entre uma partícula e uma parede localizada na posição $x = 1$. A parte (1) da figura representa a partícula antes da colisão, (2) a mesma partícula assumindo uma posição inválida, atravessando os limites do cenário (esta situação não ocorre graças as condições de contorno), e (3) a partícula depois da colisão com seu vetor de velocidade ajustado. Note que o eixo perpendicular à superfície de colisão ($eixo\perp$) neste caso é o eixo x.

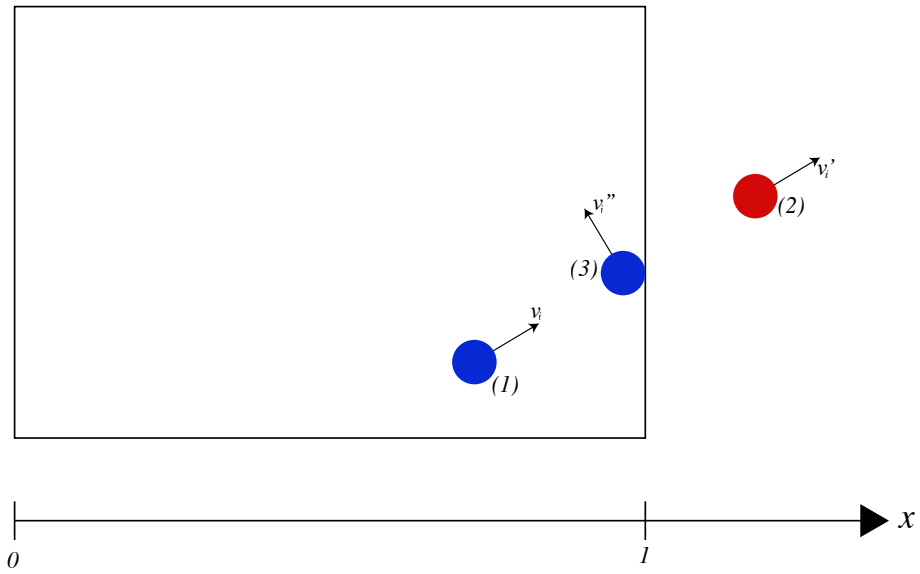


Figura 2.6: Exemplo de colisão entre uma partícula e um objeto.

3 Implementação

3.1 Ambiente de desenvolvimento

Durante o desenvolvimento deste trabalho foi implementado o SPH utilizando a linguagem de programação C++. Para editar o código foi utilizado o IDE (*Integrated Development Environment*, Ambiente de Desenvolvimento Integrado em português) Xcode (Apple, 2016) e para compilá-lo foi utilizado o compilador Apple LLVM 7.1.

No início do desenvolvimento o programa gerava arquivos de extensão *VTK*, contendo as coordenadas de cada partícula, de forma que cada arquivo representava o sistema de partículas em um dado instante de tempo (uma única simulação era representada por múltiplos arquivos). Estes arquivos eram carregados no Paraview (Kitware et al, 2016), que pode representar cada partícula por um ponto no espaço e alternar as posições dos pontos conforme realiza a leitura dos arquivos, em *loop*, criando uma animação.

Inicialmente o código consumia muito tempo para ser executado. Após as otimizações discutidas na Seção 3.4 serem implementadas, o programa passou a ser apto a gerar simulações em tempo real. Esta versão otimizada foi integrada como biblioteca no motor gráfico Unity (mais detalhes sobre a integração estão disponíveis na Seção 3.6). Nesta etapa foram criados scripts na linguagem C# para realizar a comunicação entre o Unity e a biblioteca implementada, utilizando o IDE Monodevelop (Xamarin, 2016).

Durante todo o desenvolvimento foi utilizado o sistema de controle de versões *Git* (Torvalds e Hamano, 2016), utilizando repositórios hospedados nos servidores do BitBucket. Isto permitiu rastrear cada mudança realizada no código fonte do projeto.

3.2 Estrutura de dados adotada inicialmente

Foram utilziadas técnicas de *Programação Orientada a Objetos* para representar diferentes estruturas no programa. A Figura 3.1 ilustra a estrutura de classes criada durante as primeiras versões do programa. Entretanto o programa sofreu inúmeras otimizações (as

mais importantes são descritas na Seção 3.4) e os atributos e métodos das classes foram alterados.

A classe *SPH* possui uma lista de todas as partículas da simulação, os valores das constantes do SPH (Tabela 2.1) e métodos para realizar iterações, definir os limites do cenário, obter o número de partículas do sistema e adicionar novas partículas.

A classe *Ponto* foi criada para representar um ponto no espaço ou um vetor com origem no ponto $(0, 0, 0)$. Ela possui apenas três números reais (x, y e z), representados por variáveis do tipo *float*, e métodos para acessar e alterar esses valores, e para realizar operações entre dois pontos.

A classe *Particula* representa uma partícula no sistema. Possui posição, velocidade e aceleração, além de variáveis do tipo *float* (números reais) para armazenar sua massa e densidade.

A classe *VTK* foi criada apenas para salvar todas as partículas (e suas posições no espaço) no instante de tempo atual em um arquivo no formato *VTK*. Para criar uma simulação eram gerados inúmeros arquivos neste formato, onde cada arquivo representava um *frame* da animação do fluido.

3.3 Diagrama de execução

A Figura 3.2 representa o diagrama de execução do programa desenvolvido. Primeiro o SPH precisa ser inicializado, definindo-se as constantes para os parâmetros da Tabela 2.1, os limites do cenário e também as posições iniciais das partículas. Depois são realizadas sucessivas iterações. Em cada iteração, para cada partícula, é calculada a densidade utilizando a fórmula (2.15) e as forças de pressão e viscosidade que atuam sobre ela através da Equação (2.22). Então as velocidades e as novas posições das partículas são obtidas através do *Leap-Frog* (conforme discutido na Seção 2.2.4). Por último, são verificadas as condições de contorno (Seção 2.2.6) para garantir que as partículas possuam um comportamento adequado quando colidirem com os limites do cenário.

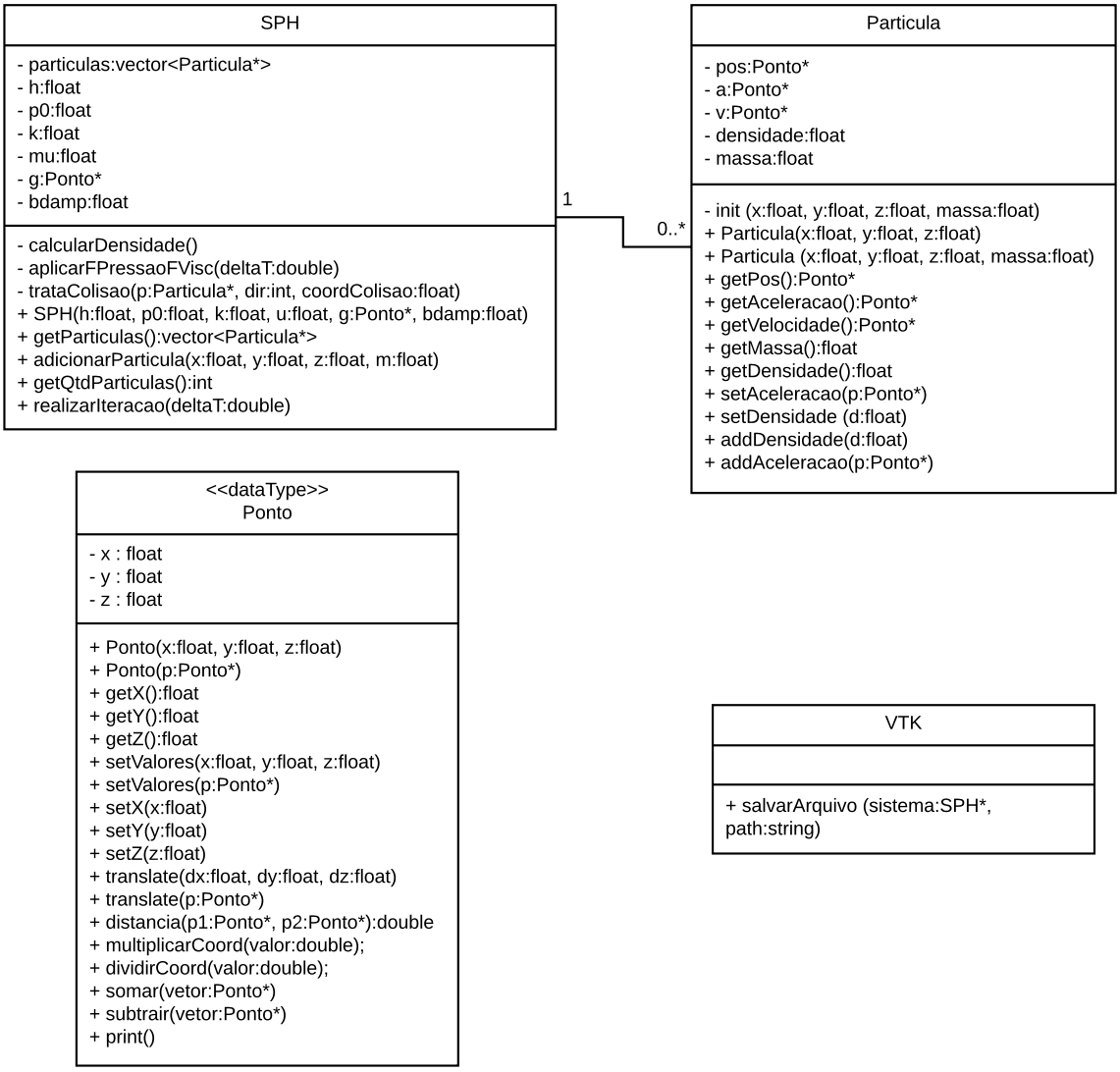


Figura 3.1: Diagrama de classes utilizadas na versão inicial da implementação.

3.4 Otimizações

Após obter a primeira versão funcional da implementação do SPH criada durante este trabalho, foram realizadas diversas melhorias no código para que ele tornasse capaz de realizar simulações com um grande número de partículas e também ser executado em tempo próximo ao real. Abaixo estão documentadas as principais otimizações realizadas e seus impactos no tempo de execução do programa. É importante ressaltar que as otimizações não estão listadas abaixo em ordem de prioridade, mas sim na ordem em que foram aplicadas ao projeto. Para comparar os efeitos de cada otimização, foi registrado o tempo de execução do programa antes e depois da melhoria ser aplicada. Para obter valores consistentes, foram realizadas 5 execuções para cada instância de testes e então

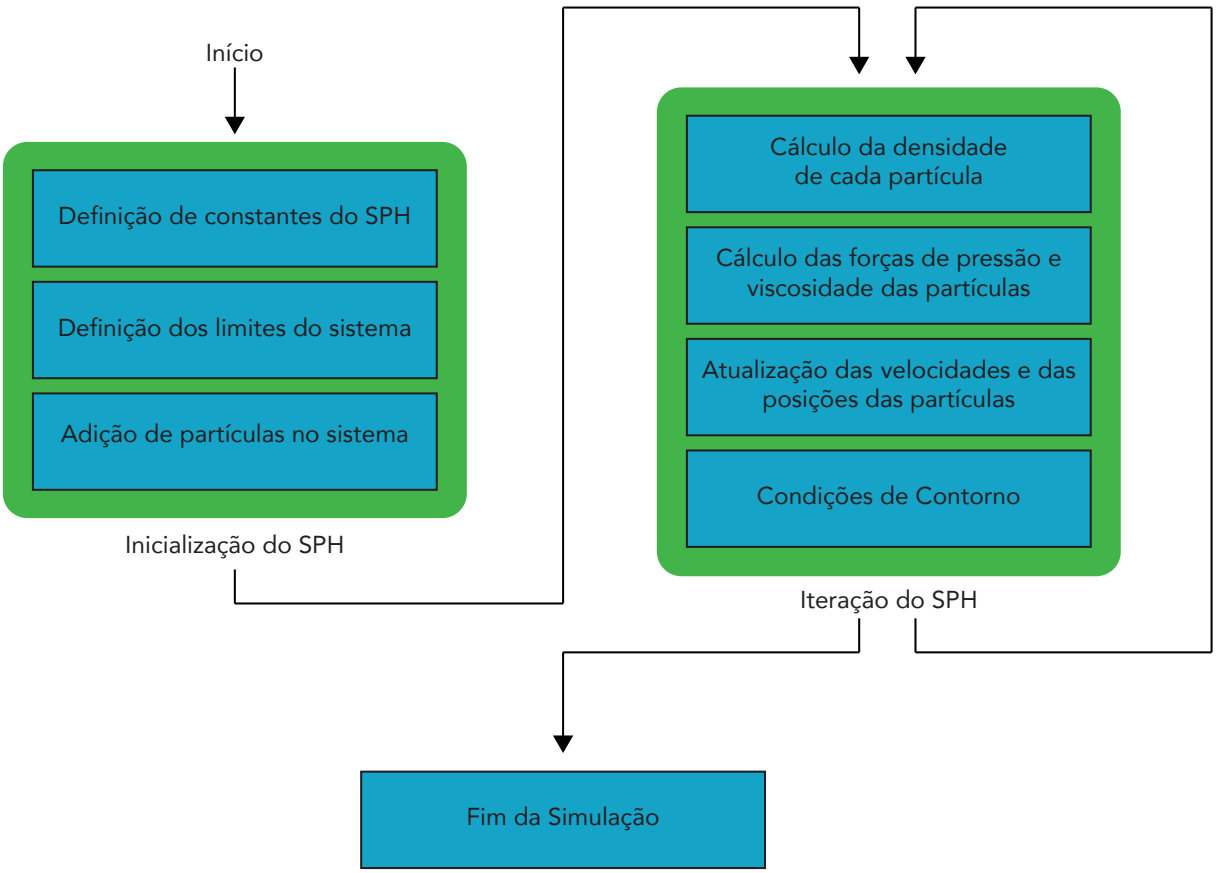


Figura 3.2: Diagrama de execução do programa implementado.

obtida a média do tempo de execução.

3.4.1 Mudança de estrutura de dados

Até então o código utilizava classes criadas especificamente para o programa, como a classe *Particula* ou a classe *Ponto*. A classe principal do programa (que é chamada *SPH*, por gerenciar todas as partículas do sistema) possuía uma lista de instâncias da classe *Particula*, que por sua vez possuía instâncias da classe *Ponto* para representar sua posição, velocidade e aceleração, e valores inteiros para representar sua massa e sua densidade. Esta modelagem é chamada de *Array of Structures* (AoS), pois representa uma coleção de estruturas para representar uma entidade (na implementação, uma coleção de instâncias da classe *Particula* representava o sistema SPH).

O programa estava gastanto muito tempo para acessar as propriedades específicas de uma partícula. Por exemplo, para acessar a coordenada x da partícula p_i, o programa precisava primeiro carregar na memória o objeto da classe *Particula* que está na posição i

de sua lista de partículas, acessar o objeto *Ponto* que representa a posição desta partícula e finalmente acessar o inteiro que representa a coordenada x deste ponto, realizando um longo caminho para apenas acessar uma propriedade da partícula.

Para resolver este problema foi abandonada a estratégia de representar as partículas e pontos por classes específicas. A classe principal do programa deixou de armazenar uma lista de instâncias da classe *Particula*, e passou então a armazenar uma lista diferente para cada propriedade das partículas: uma lista exclusiva para as posições das partículas, outra para as velocidades, outra para as densidades, e assim por diante. Desta forma a modelagem do programa deixou de ser do tipo AoS e passou a ser do tipo *Structure of Arrays* (SoA), que possui como característica ter uma estrutura de diferentes coleções de dados para representar uma entidade (na implementação o sistema do SPH passou a ser representado por diversas listas, uma para cada propriedade das partículas). Converter um código AoS para SoA faz com que seus dados sejam armazenados de forma contínua na memória, permitindo que o processador utilize a memória auxiliar de *cache* para acessar os dados mais rapidamente e que o compilador possa fazer otimizações de acesso automaticamente.

Para todas as propriedades foram utilizadas listas simples de números reais. Para as propriedades numéricas (como a densidade e a massa), o acesso pode ser realizado diretamente (por exemplo, a densidade da partícula i está localizada na posição i da lista de densidades das partículas). Para as propriedades vetoriais (como posição, velocidade e aceleração), as componentes x, y e z de cada propriedade vetorial foram armazenadas sequencialmente em uma lista específica para cada propriedade. A Tabela 3.1 representa como as coordenadas da partícula são armazenadas na memória (sendo (x_i, y_i, z_i) as coordenadas da partícula i). Desta forma, para acessar as coordenadas x, y e z da partícula i é necessário acessar as posições $3i$, $3i + 1$ e $3i + 2$, respectivamente.

Tabela 3.1: Como as coordenadas das partículas são armazenadas na memória

x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	(...)	x_i	y_i	z_i

A Tabela 3.2 mostra o impacto que esta melhoria teve no tempo de execução em uma certa simulação de fluidos.

Tabela 3.2: Impacto da alteração na estrutura de dados no tempo de execução do programa

Execução	**Tempo de execução sem a otimização (em segundos)**	**Tempo de execução com a otimização (em segundos)**
1	40,3292	34,2643
2	41,1609	34,1512
3	41,5534	34,3588
4	41,1119	34,4323
5	41,0264	34,0057
Desvio padrão	0,4440	0,1691
Média	**41,1119**	**34,2643**

3.4.2 Flags de otimização de código

O compilador pode automaticamente otimizar o código de acordo com a flag de otimização adotada. A Tabela 3.3 descreve os diferentes níveis de otimização que o compilador Apple LLVM versão 7.1 oferece para a linguagem C++. Para o projeto foi utilizada a otimização nível *o3*, pois esta é a que oferece mais melhorias no desempenho do programa e, embora possa afetar a confiabilidade do programa compilado, este nível não apresentou alterações na fidelidade das simulações geradas pela implementação deste trabalho.

Tabela 3.3: Descrição das diferentes flags de otimização suportadas pelo compilador

Flag	**Descrição**
o0	Nenhuma otimização será realizada. Esta é a flag padrão adotada pelos compiladores.
o1	Ativa otimizações básicas.
o2	Realiza mais otimizações que o nível *o1*, realizando quase todas as otimizações suportadas que não envolvem um compromisso entre espaço e velocidade.
o3	Ativa todas as otimizações do nível *o2* e as que envolvem um compromisso entre espaço e velocidade. O programa compilado ficará maior com esta flag, mas ele provavelmente será mais rápido.
os	Possui como objetivo otimizar o tamanho do programa binário gerado pelo compilador. Realiza todas as otimizações da flag *o2* que não afetem o tamanho do programa.
ofast	Otimiza ainda mais que o nível *o3*, realizando otimizações que podem até mesmo gerar código incorreto até mesmo para programas criado com padrões corretos.

A Tabela 3.4 mostra como as otimizações realizadas pelo compilador afetaram o programa, ao executar uma simulação de um fluido formado por 216 partículas, realizando 1999 iterações (com um passo de tempo de 0,0001 segundos) e gerando 400 arquivos *VTK*.

Esta otimização gerou uma grande melhoria (cerca de 95%) no tempo de execução, o que permitiu realizar simulações mais longas e/ou com mais partículas que antes eram inviáveis de se executar, pois isto exigiria grande esforço computacional.

Tabela 3.4: Impacto da utilização da flag de otimização *o3*

Execução	**Tempo de execução utilizando flag *-o0* (em segundos)**	**Tempo de execução utilizando flag *-o3* (em segundos)**
1	118,5060	6,3398
2	119,4050	6,2867
3	117,4290	6,3681
4	117,8040	6,2955
5	117,6280	6,3886
Desvio padrão	0,8083	0,0444
Média	**117,8040**	**6,3398**

3.4.3 Otimização da busca por partículas vizinhas

Conforme discutido na Seção 2.2.5, a eficiência da busca de partículas vizinhas afeta diretamente no desempenho do SPH. Até então a busca era realizada analisando todas as partículas da simulação, o que exigia muitos recursos computacionais. Para otimizar a busca o domínio foi dividido em células, utilizando uma grade conforme discutido na Seção 2.2.5.2.

A Tabela 3.5 possui o tempo médio de execução de uma simulação com 1331 partículas e 1999 iterações, que gera 400 arquivos *VTK* e que utiliza um passo de tempo de 0,0001 segundos, utilizando a busca por vizinhos não otimizada (sem grade) e realizando a busca otimizada (com grade, e variando o tamanho do lado das células $k * h$ tal que $1 \leq k \leq 6$). Com $k = 4$ foi possível diminuir o tempo de execução em 58%.

3.4.4 Remoção de chamadas da função *sqrt*

Calcular a raiz quadrada de um número através da função *sqrt* (implementada pela biblioteca *math.h*) é muito custoso. No SPH frequentemente é necessário comparar a distância de uma partícula com o comprimento suave do SPH (Equação (2.14)). Para otimizar o programa, evitando o cálculo de raízes quadradas, os dois lados da condição da Equação

Tabela 3.5: Impacto da otimização da busca por partículas vizinhas no tempo de execução do programa

#	**Sem grade**	**Com grade (utilizando células de lado $k * h$)**					
-	**-**	$1 * h$	$2 * h$	$3 * h$	$4 * h$	$5 * h$	$6 * h$
1	188,4610	557,5280	109,3210	79,9807	77,1796	84,8060	94,8703
2	185,7260	556,7110	114,2930	80,5250	78,5435	85,5694	95,6042
3	188,2040	556,6950	108,1900	80,4290	77,8409	84,8734	95,5645
4	185,7320	556,0380	109,9580	80,2617	77,4949	84,5411	95,7673
5	186,8950	557,2540	108,3550	81,9495	78,4200	84,7191	95,0996
Desvio padrão	1,3063	0,5756	2,4936	0,7664	0,5854	0,3934	0,3784
Média	**186,8950**	**556,7110**	**109,3210**	**80,4290**	**77,8409**	**84,8060**	**95,5645**

(2.14) foram elevados ao quadrado:

$$j \in N_i \text{ se e somente se } r^2 < h^2. \tag{3.1}$$

Como $r = |r_i - r_j| = \sqrt{(r_{i,x} - r_{j,x})^2 + (r_{i,y} - r_{j,y})^2 + (r_{i,z} - r_{j,z})^2}$, a Equação (3.1) pode ser reescrita da seguinte forma:

$$j \in N_i \text{ se e somente se } (r_{i,x} - r_{j,x})^2 + (r_{i,y} - r_{j,y})^2 + (r_{i,z} - r_{j,z})^2 < h^2. \tag{3.2}$$

Desta forma o programa não precisa calcular nenhuma raiz quadrada. A Tabela 3.6 mostra que esta otimização teve um baixo impacto (cerca de 2%) para uma simulação com 1331 partículas, 19999 iterações e tempo de passo de 0,0001 segundos, e que gera 400 arquivos *VTK*. Porém ela não deve ser desprezada. Na Seção 3.4.5 será discutida uma otimização semelhante, mas que gerou resultados mais promissores.

3.4.5 Remoção de chamadas da função *pow*

Durante o cálculo das forças de pressão de viscosidade, da velocidade e da aceleração das partículas é necessário calcular diversas potências. Para isto estava sendo utilizada a função *pow* (presente na biblioteca *math.h*), porém esta função consome muitos recursos computacionais para ser executada. Uma solução deste problema é substituir as chamadas

Tabela 3.6: Impacto da remoção de chamadas da função *sqrt* no tempo de execução do programa

Execução	**Tempo de execução sem a otimização (em segundos)**	**Tempo de execução com a otimização (em segundos)**
1	77,180	74,2011
2	78,544	76,6129
3	77,841	75,4131
4	77,495	76,1931
5	78,420	76,5223
Desvio padrão	0,5854	1,0053
Média	**77,841**	**76,1931**

da função *pow* por sucessivas multiplicações. Por exemplo, a potência h^8 presente na Equação (2.11) pode ser calculada da seguinte forma:

```
1 float h2 = h * h; // h^2
2 float h4 = h2 * h2; // h^4
3 float h8 = h4 * h4; // h^8
```

A Tabela 3.7 possui o tempo médio de execução de uma simulação com 1331 partículas, que executa 1999 iterações, que utiliza um passo de tempo de 0,0001 segundos e que gera 400 arquivos *VTK*, executada antes e depois desta otimização ser aplicada. Esta alteração resultou em uma economia de tempo de cerca de 26%.

Tabela 3.7: Impacto da remoção de chamadas da função *pow* no tempo de execução do programa

Execução	**Tempo de execução sem a otimização (em segundos)**	**Tempo de execução com a otimização (em segundos)**
1	74,2011	56,073
2	76,6129	56,720
3	75,4131	56,765
4	76,1931	57,202
5	76,5223	56,586
Desvio padrão	1,0053	0,4056
Média	**76,1931**	**56,720**

3.5 Diagrama final de classes e métodos

Após otimizar o código algumas classes deixaram de ser utilizadas e outras sofreram alterações. A Figura 3.3 descreve os métodos e atributos das classes na versão final do programa. Nesta versão está sendo utilizado o método *Leap-Frog* (definido na Seção 2.2.4) para atualizar as posições das partículas, portanto foi necessário alocar um novo vetor (denominado *velMeioParticulas*) para armazenar os valores das velocidades obtidos pela Equação (2.26). O vetor *velParticulas* armazena os valores obtidos pela equação (2.28).

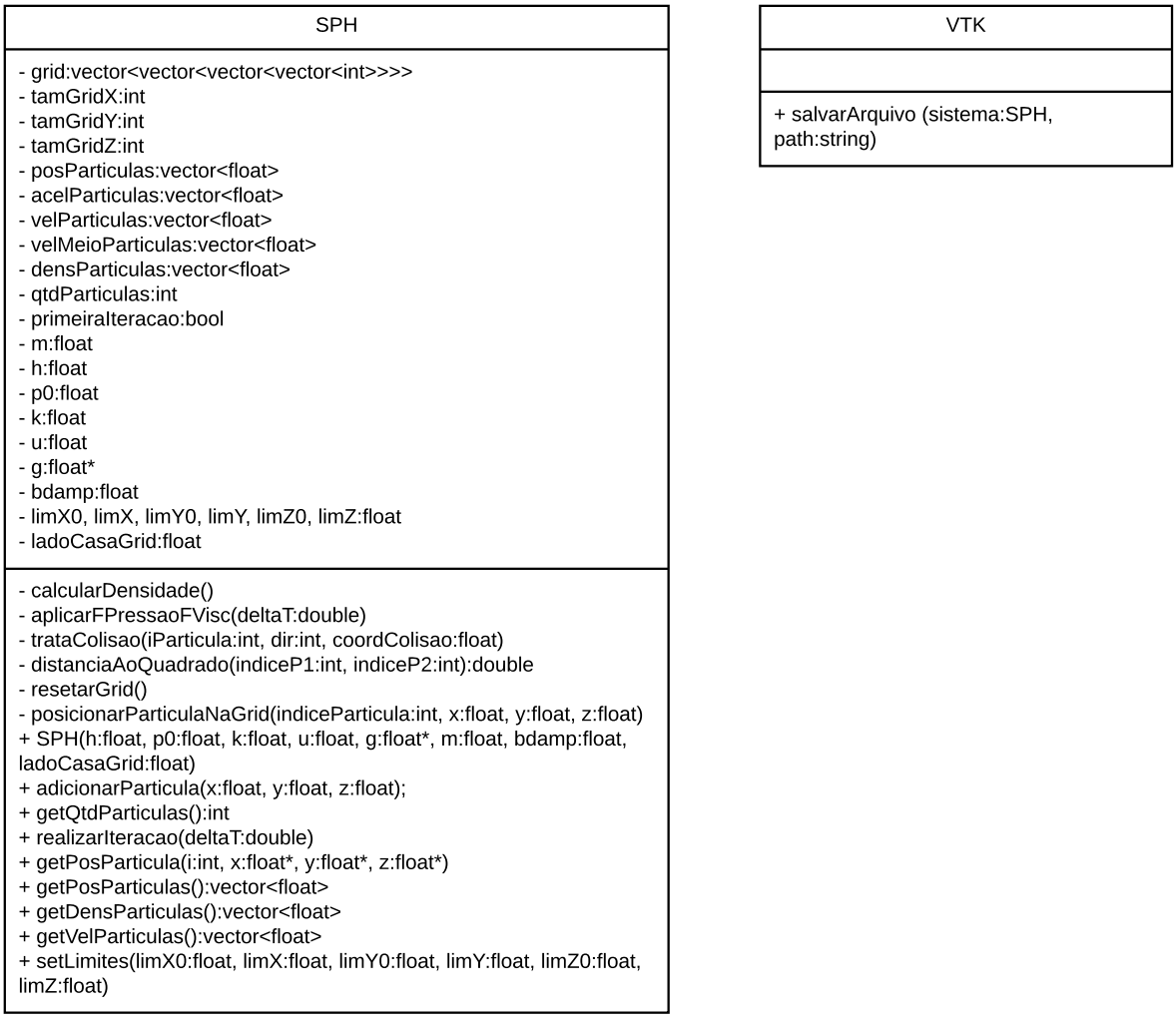


Figura 3.3: Diagrama de classes utilizadas na versão final da implementação.

3.6 Integração com Unity

3.6.1 Biblioteca SPH

Depois de otimizar a implementação do SPH, ela foi integrada ao motor de jogos Unity. O motor suporta nativamente as linguagens de programação C# e Javascript, mas não C++. Por este motivo o programa implementado em C++ foi compilado no formato de uma **biblioteca**, que pode ser utilizada por um script escrito em C# ou Javascript e então ser utilizado no Unity.

O IDE Xcode foi configurado para compilar o projeto no formato *.bundle*, que é o formato de bibliotecas para o sistema operacional OS X. Este arquivo foi carregado dentro do Unity e então foi criado uma classe em C#, chamada *SPH*, que é executada pelo Unity e é responsável por carregar a biblioteca compilada e utilizá-la para executar uma simulação de fluidos. A Figura 3.4 representa um diagrama de como os arquivos da implementação original foram carregados no Unity: O código em C++ foi compilado como uma biblioteca, gerando o arquivo *SPHLib.bundle* que é utilizado pelo código *SPH.cs* (escrito em C#), que por sua vez é executado no Unity.

Abaixo está uma versão simplificada do código desta classe. Alguns trechos foram omitidos e outros são representados aqui como pseudocódigo para facilitar o entendimento. O código completo também trata opções mais avançadas que serão discutidas no decorrer desta seção.

```
1  public class SPH : MonoBehaviour {
2      [DllImport ("SPHLib")]
3      private static extern void initSPH (float h, float p0, float k,
           float u, float[] g, float m, float bdamp, float
          ladoCasaGrid);
4      private static extern void setLimites (float limX0, float limX,
           float limY0, float limY, float limZ0, float limZ);
5      private static extern void adicionarParticula(float x, float y,
           float z);
6      private static extern void realizarIteracao(double deltaT);
7      private static extern void getPosParticulas([In, Out] ref float
          [] posParticulas);
8      private static extern void getDensParticulas([In, Out] ref
          float[] densParticulas);
9      private static extern void getVelParticulas([In, Out] ref float
          [] velParticulas);
10
```

```
11      void Start () {
12          ConstantesSPH constantes = this.GetComponent<ConstantesSPH
               >();
13          initSPH(constantes.h, constantes.densidadeRepouso,
               constantes.k, constantes.viscosidade, constantes.g,
               constantes.massaParticula, constantes.amortecimento,
               constantes.KLadoCasaGrid*constantes.h);
14          setLimites (-limites.x, limites.x, -limites.y, limites.y, -
               limites.z, limites.z);
15
16          for (int i = 0; i < qtdParticulas; i++) {
17              esferas [i] = GameObject.CreatePrimitive (PrimitiveType
                   .Sphere).transform;
18              pos = vetor randomico no dominio do fluido;
19              esferas [i].localPosition = pos;
20              adicionarParticula (pos.x, pos.y, pos.z); // Adiciona a
                    particula no sistema SPH gerenciado pela biblioteca
21          }
22      }
23
24      void Update () {
25          realizarIteracao (tempoIteracao);
26          for (int i = 0; i < qtdParticulas; i++) {
27              pos.x = posicoesParticulas [i * 3    ];
28              pos.y = posicoesParticulas [i * 3 + 1];
29              pos.z = posicoesParticulas [i * 3 + 2];
30              esferas [i].localPosition = pos;
31          }
32      }
33  }
```

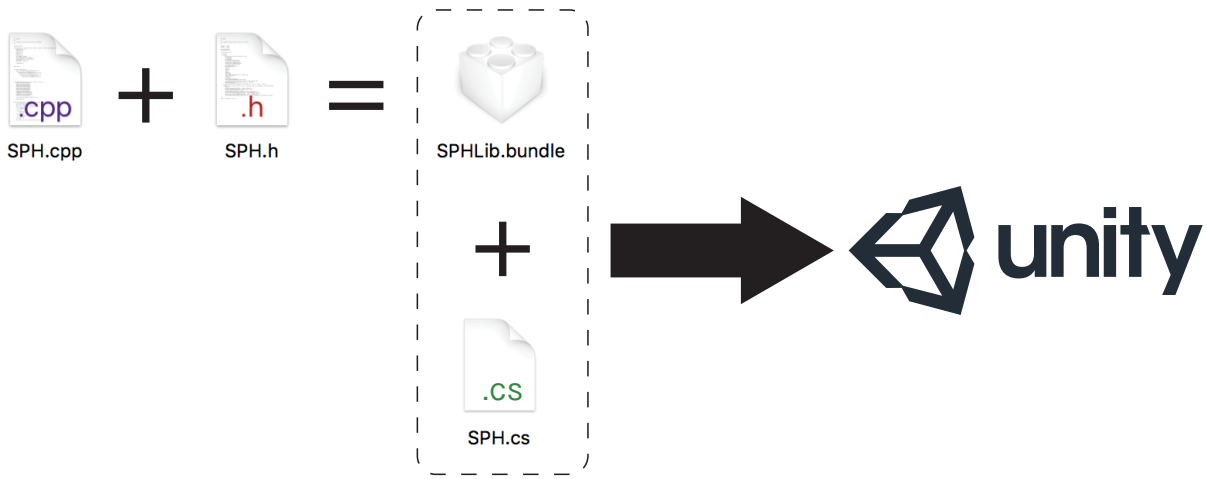


Figura 3.4: Como o programa implementado foi integrado no Unity.

As linhas 2–9 são responsáveis por importar todos os métodos necessários para realizar uma simulação de fluidos da biblioteca, cujo arquivo foi nomeado como ***SPH-Lib***.*bundle*, por isto o comando *DllImport* da linha 2 recebe como parâmetro a string

"SPHLib".

A função *Start* é executada pelo Unity durante a inicialização do jogo desenvolvido pelo motor gráfico. A linha 12 carrega os parâmetros definidos pelo usuário, que serão utilizados para inicializar o SPH (linhas 13–14, que realizam chamadas de métodos da biblioteca). O laço realizado entre as linhas 16–21 cria todas as partículas da simulação, gerando uma posição randômica para cada uma. Além de adicionar a partícula no sistema SPH (linha 20), o algoritmo também cria uma instância de uma esfera que representará a partícula no Unity (linha 17).

A função *Update* é chamada pelo Unity logo depois que a função *Start* conclui sua execução, e é executada repetidamente até que o jogo seja interrompido pelo usuário. A linha 25 realiza uma iteração do SPH utilizando como o intervalo de tempo o valor armazenado na variável *tempoIteracao*. O laço realizado pelas linhas 26–31 são responsáveis por atualizar as posições de cada esfera que representa uma partícula (estas esferas foram criadas no método *Start*). Deste modo a biblioteca realiza a simulação do SPH e o Unity representa as partículas por esferas, em tempo real.

3.6.2 Configuração dos parâmetros da biblioteca no Unity

A Figura 3.5 é uma captura de tela da interface do Unity, utilizando seu layout de visualização padrão. As abas *Hierarchy*, *Scene*, *Project* e *Console* exibem uma listagem hierárquica dos objetos presentes na cena atual, uma visualização da cena, uma interface para gerenciamento dos arquivos do projeto e a saída do console de texto, respectivamente. A aba *Inspector* exibe vários componentes de entrada que permitem ao usuário alterar as propriedades de um objeto e gerenciar seus componentes.

Para executar a implementação do SPH no Unity o usuário precisa criar um objeto vazio na engine, e então adicionar o componente *SPH* ao objeto. Para isto basta selecionar o menu *GameObject* e a opção *Create Empty* no Unity, e então arrastar a classe *SPH* (discutida da seção anterior) sobre o objeto. Quando o usuário selecionar este objeto a aba *Inspector* do Unity passará a exibir diversos campos para o usuário ser capaz de configurar a simulação do fluido. A Figura 3.11 é uma captura de tela desta área.

O campo *Dimensoes* possui duas opções: *D2D* e *D3D* que representam 2 dimen-

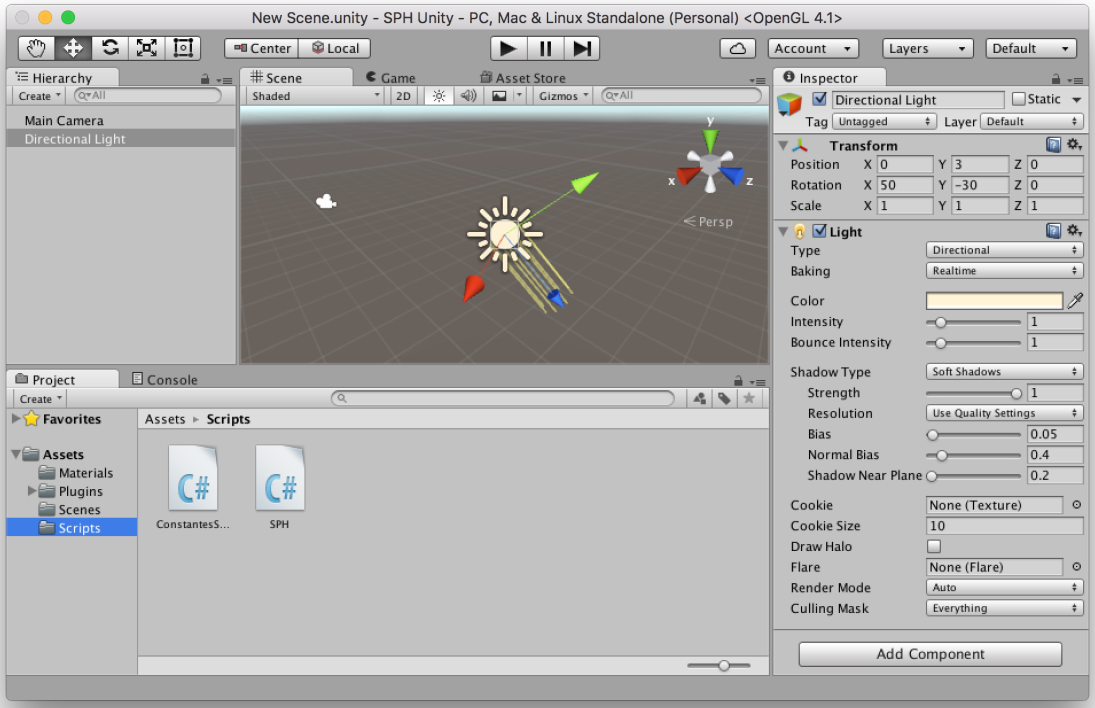


Figura 3.5: Captura de tela do Unity.

sões e 3 dimensões, respectivamente. Se o usuário selecionar 2 dimensões as partículas serão distribuídas aleatoriamente no plano $x \times y$ e todas as partículas terão a coordenada z de sua posição definida como zero. Como não haverá nenhuma força atuando sobre as partículas no eixo z (a gravidade não atua no eixo z e não haverá partículas distribuídas ao longo do eixo z para realizar alguma força neste eixo) nenhum tratamento especial além da distribuição inicial das partículas foi necessário para representar uma simulação em 2D.

Qtd Particulas e *Tempo Iteracao* definem o número de partículas que serão utilizadas na simulação e o intervalo de tempo (em segundos) que será utilizado em cada iteração do SPH, respectivamente.

Caso a opção *Desenhar Limites Cenario* estiver ativada, os limites do cenário (cujo tamanho é definido pelo parâmetro *Limites Cenario*) são exibidos em verde na aba *Scene* do Unity. Quando a simulação iniciar, as partículas serão distribuídas aleatoriamente dentro de um cubo que é representado em azul na aba *Scene*, e o usuário pode alterar a posição e as dimensões deste cubo realizando alterações nas variáveis *Pos*

Cubo Liquido e *Tam Cubo Liquido*, respectivamente. Caso o usuário defina uma posição inválida para o cubo (por exemplo, o posicionando fora dos limites da simulação), ele passará a ser exibido em vermelho para alertar o usuário. A Figura 3.6 mostra uma captura de tela do resultado da seguinte configuração: *Limites Cenario* $= (1, 0.5, 0.5)$, *Pos Cubo Liquido* $= (-0.25, 0, 0)$ e *Tam Cubo Liquido* $= (0.3, 0.3, 0.3)$.

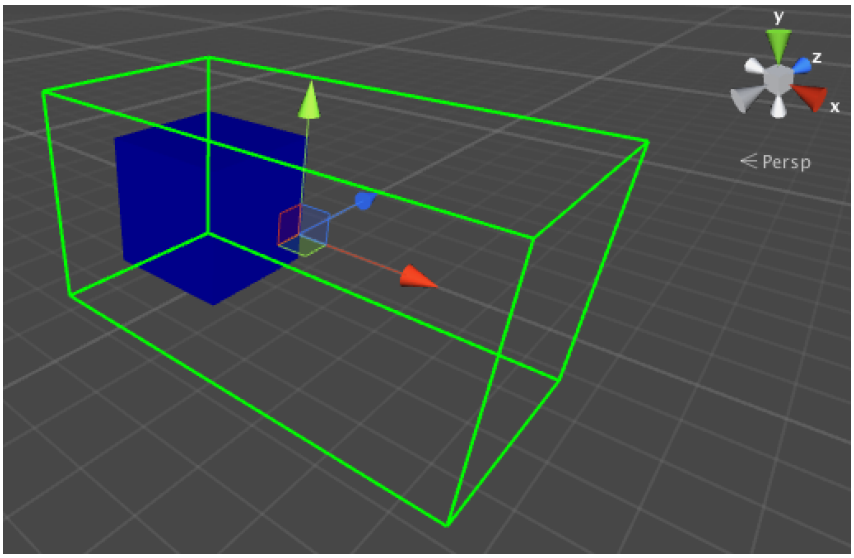


Figura 3.6: Captura de tela da aba *Scene* do Unity exibindo os limites da simulação e a área em que as partículas serão distribuídas aleatoriamente.

Os parâmetros *Pos X Barreira Dam Break* e *Ativar Dam Break* serão discutidos na Seção 3.6.4. *Escala Esfera* define o diâmetro das esferas que representarão as partículas no Unity. No parâmetro *Material Particula* deve ser selecionado um *Material* do Unity (que é uma definição de como uma superfície deverá ser renderizada, incluindo por exemplo, referências para texturas e tons de cores) que será utilizado para renderizar estas esferas. Os campos *Cor Particula Vel Max*, *Cor Particula Dens Min* e *Cor Particula Dens Max* são discutidos na Seção 3.6.3.

Gravidade é um vetor de 3 dimensões que define a aceleração da gravidade. *Amortecimento* e *K Lado Casa Grid* representam a variável α da Equação (2.35) e a variável k definida na Seção 2.2.5.2, respectivamente. Os demais campos da Imagem 3.11 são os parâmetros do SPH que são descritos na Tabela 2.1.

3.6.3 Coloração de partículas

Um dos recursos extras que foi implementado no algoritmo em C# que utiliza a biblioteca do SPH foi a possibilidade de alterar a coloração das partículas. Para isto o usuário precisa selecionar uma opção no parâmetro *Cor Particula* na aba *Inspector* do Unity (Figura 3.11). Existem as seguintes opções disponíveis: *Padrao*, *Preto*, *Vermelho*, *Verde*, *Azul*, *Densidade*, *Velocidade* e *Randômico*. Nenhuma destas opções afeta a simulação do fluido, porém algumas delas, como as opções de coloração por densidade e por velocidade, podem auxiliar o usuário a compreender determinadas propriedades das partículas.

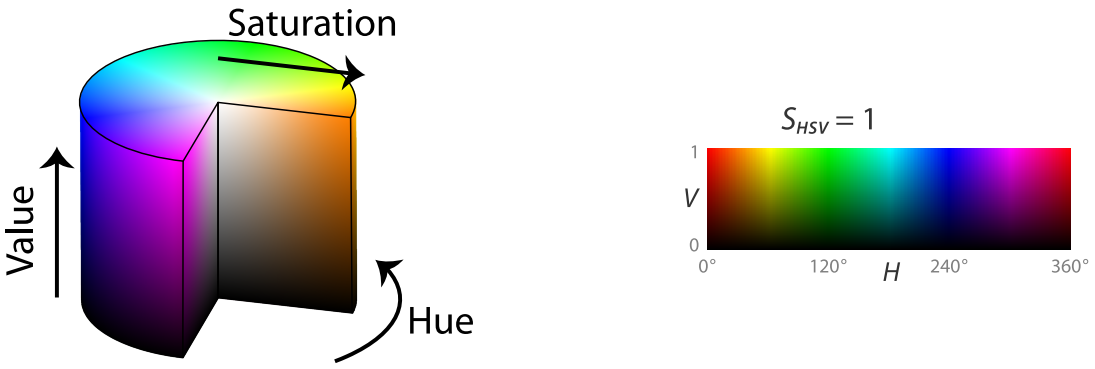
Ao utilizar a opção *Padrao* a cor das partículas será a mesma definida pelo *Material* definido no campo *Material Particula*. As opções *Preto*, *Vermelho*, *Verde* e *Azul* alteram a cor de todas as partículas para a respectiva cor.

A opção *Densidade* faz com que o algoritmo defina cores diferentes para as partículas, de acordo com a sua densidade. Para isto o algoritmo utiliza a representação de cores HSV (*Hue, Saturation and Value*), que utiliza os três parâmetros que originaram seu nome para definir uma cor. A Figura 3.7 mostra representações do HSV em planos de três (a) e duas (b) dimensões. O algoritmo implementado fixa os valores de *Saturation* e $Value$, e alterando o valor do parâmetro Hue de acordo com o valor da densidade de cada partícula. Para isto o usuário deverá informar uma estimativa da menor e da maior densidades que uma partícula pode ter durante uma simulação através dos parâmetros *Cor Particula Dens Min* (denotado por ρ_{min}) e *Cor Particula Dens Max* (denotado por ρ_{max}), respectivamente. O algoritmo poderia obter estes valores automaticamente, mas isto consumiria esforço computacional desnecessário. Para fins de otimização foi decidido solicitar uma estimativa destes valores ao usuário. Desta forma a cor de uma partícula i que possui densidade ρ_i é definida da seguinte forma:

$$\begin{cases} h = \frac{\rho_i - \rho_{min}}{\rho_{max} - \rho_{min}} \\ s = 1 \\ v = 1 \end{cases}, \tag{3.3}$$

onde h, s e v são os parâmetros do HSV já normalizados ($0 \leq h, s, v \leq 1$).

A Figura 3.8 mostra uma captura de tela de uma simulação de um fluido em duas



(a) (b)

Figura 3.7: Representação do modelo HSV em três dimensões (a) e em duas dimensões (b).

dimensões utilizando a coloração de partículas por densidade, com 2000 partículas e os parâmetros *Cor Particula Dens Min* = 0 e *Cor Particula Dens Max* = 800.

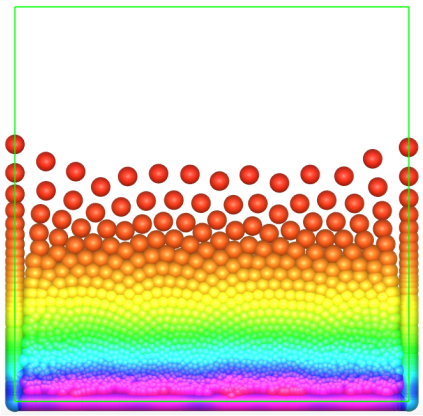


Figura 3.8: Exemplo de simulação de um fluido utilizando a coloração por densidade.

De maneira similar, também foi implementada a coloração por velocidade. Neste caso o algoritmo calcula o módulo da velocidade de cada partícula e a colore de acordo como valor aproximado da magnitude da velocidade máxima informado pelo usuário através do parâmetro *Cor Particula Vel Max* (definido por v_{max}): quanto mais próximo da velocidade máxima, mais a cor da partícula tenderá para o branco, e quanto mais distante, mais tenderá para o azul. Para isto foi adotado a representação de cores RGB (*Red, Green and Blue*), onde existem três parâmetros r, g e b que representam cada cor que origina seu nome, e estas cores são combinadas para se obter diferentes cores. Sendo

$\overrightarrow{v_i}$ a velocidade de uma partícula, sua cor é definida da seguinte forma:

$$\begin{cases} r = g = \frac{|\overrightarrow{v_i}|}{v_{max}} \\ b = 1 \end{cases}, \tag{3.4}$$

onde r, g e b estão normalizados ($0 \leq r, g, b \leq 1$). Desta forma quando v_i é zero a partícula terá cor azul ($r = g = 0$ e $b = 1$) e quando v_i é igual a v_{max} a partícula será renderizada na cor branca ($r = g = b = 1$). A Figura 3.9 mostra como a cor de uma partícula varia de acordo com o módulo de sua velocidade.

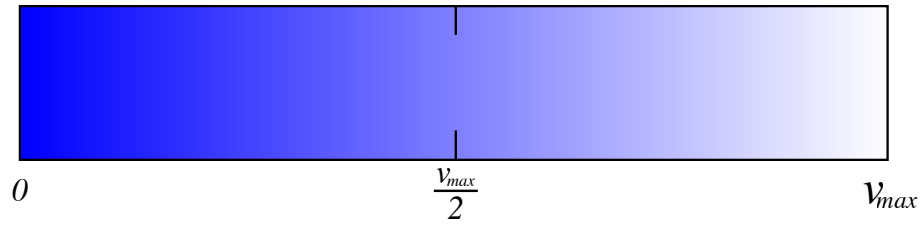


Figura 3.9: Domínio de cores que uma partícula i movendo a uma velocidade com módulo v_i pode assumir ao ser renderizada pela coloração por velocidade.

Apenas para testes também foi criada a coloração randômica, que apenas define uma cor randômica para cada partícula durante o início da simulação (ao utilizar esta opção as cores não são atualizadas no decorrer da simulação).

3.6.4 Dam Break

Dam Break (quebra de barreira, em tradução literal) é uma simulação onde as partículas são posicionadas, inicialmente, apenas em uma parte isolada do espaço, como se houvesse uma barreira vertical imaginária delimitando o espaço. No decorrer da simulação o fluido se espalha pelo cenário.

A integração com o Unity permite ao usuário realizar esta simulação. Para isto, ele precisa definir a posição x da barreira através do campo *Pos X Barreira Dam Break*. Conforme mostrada na Figura 3.10, a barreira é exibida na aba *Scene* do Unity em verde. O cubo azul que representa a área em que as partículas serão distribuídas deve ficar posicionado a esquerda da barreira. Como as partículas são posicionadas aleatoriamente na inicialização da simulação, a distribuição inicial de partículas geralmente não é uma

distribuição de equilíbrio das partículas e portanto as partículas começam a se mover muito rapidamente durante o início da simulação. Por este motivo, a barreira não é removida imediatamente após o início da simulação. A barreira só será "quebrada"quando o usuário ativar a opção *Ativar Dam Break*, clicando na caixa de seleção referente a esta opção. A Seção 4.1.1 possui um exemplo deste tipo de simulação.

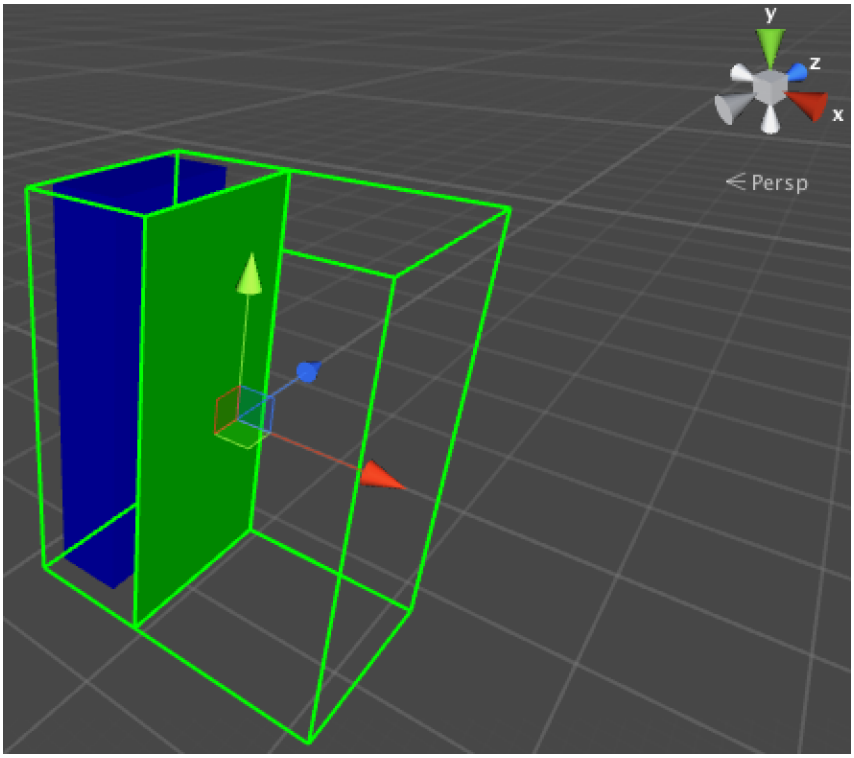


Figura 3.10: Captura de tela do Unity. A barreira da simulação *Dam Break* está posicionada na posição $x = 0.5$.

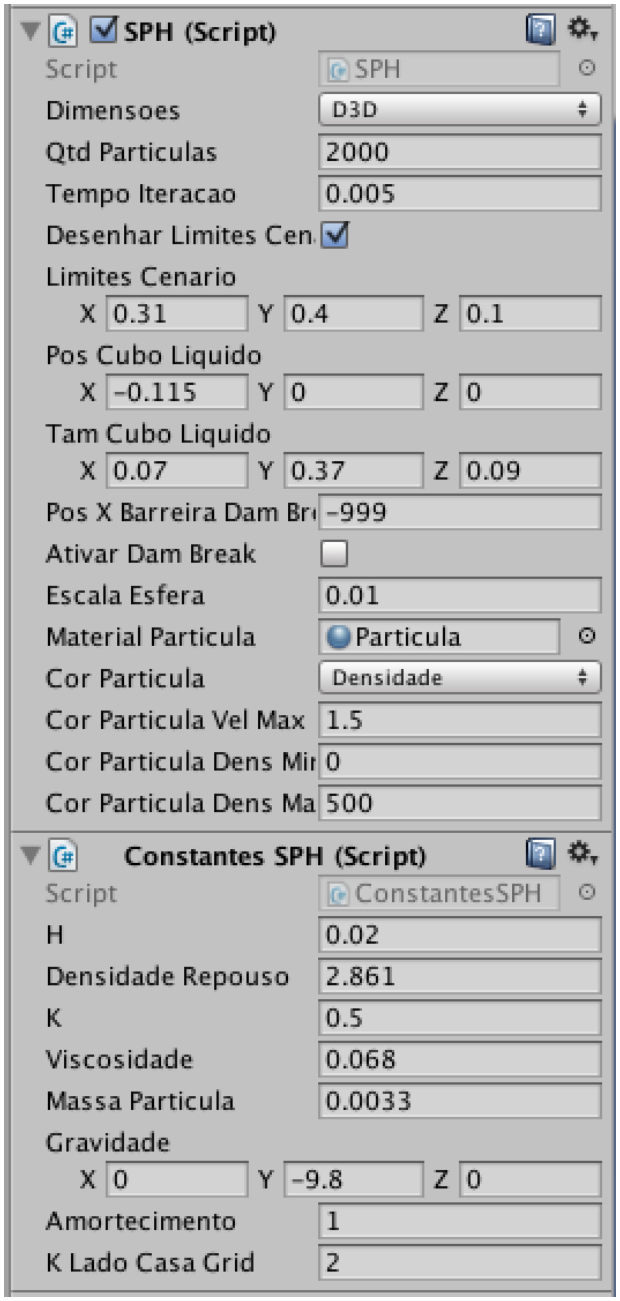


Figura 3.11: Captura de tela da aba *Inspector* do Unity, que fornece uma interface para o usuário configurar a simulação do fluido no Unity.

4 Resultados

As simulações descritas nesta seção foram executadas no Unity, utilizando um Macbook Pro com processador Intel Core i5 2.4GHz e 8GB de memória RAM.

4.1 Exemplos de simulações

4.1.1 Dam Break

A Figura 4.1 mostra alguns frames de uma simulação gerada pelo programa implementado no Unity, que realiza o *Dam Break* discutido na Seção 3.6.4, e que utiliza a coloração por densidade (definida na Seção 3.6.3) e os parâmetros descritos na Tabela 4.1. A simulação foi executada com uma média de 17 FPS (Frames Por Segundo).

Tabela 4.1: Parâmetros utilizados para criar a simulação exibida na Figura 4.1

Parâmetro	**Valor**
h	0.068
Densidade Repouso (ρ_0)	2.861
k	5
Viscosidade (μ)	0.068
Massa Particula (m_i)	0.0033
Aceleração da gravidade	$(0, -9.8, 0)$
α (amortecimento)	1
K Lado Casa Grid	2
Dimensões	D2D
Quantidade de partículas	3000
Tempo Iteracao	0.004
Limites Cenario	$(1.9, 1.9, 0.1)$
Pos Cubo Liquido	$(-0.615, 0, 0)$
Tam Cubo Liquido	$(0.5, 1.5, 0.09)$
Pos X Barreira Dam break	-0.35
Escala Esfera	0.025
Cor Particula	*Densidade*
Cor Particula Dens Min	0
Cor Particula Dens Max	100

4.1.2 Simulação em três dimensões

Uma simulação em três dimensões com 5000 partículas é mostrada na Figura 4.2 e foi criada utilizando os parâmetros descritos na Tabela 4.2. Nesta simulação foi utilizada a coloração por velocidade (definida na Seção 3.6.3), fazendo com que as partículas com velocidades altas sejam representadas em branco, e as partículas com velocidades pequenas em azul. A simulação foi executada com uma média de 20 FPS. Nesta simulação as partículas foram distribuídas aleatoriamente numa região localizada acima do "chão"do cenário. No início da simulação todas as partículas estão em repouso, portanto todas elas são coloridas em azul escuro. Com o decorrer da simulação as partículas vão caindo pelo cenário e aumentando suas velocidades, fazendo com que elas mudem de cor para tons mais claros de azul, até que elas atingem a velocidade máxima e passam a ser coloridas em branco. Depois que as partículas colidem com os limites inferiores elas espalham pelo cenário e voltam a ficar em repouso (voltando a serem representadas em azul escuro).

Tabela 4.2: Parâmetros utilizados para criar a simulação exibida na Figura 4.2

Parâmetro	**Valor**
h	0.02
Densidade Repouso (ρ_0)	2.861
k	0.5
Viscosidade (μ)	0.068
Massa Particula (m_i)	0.0033
Aceleração da gravidade	$(0, -9.8, 0)$
α (amortecimento)	1
K Lado Casa Grid	4
Dimensões	D3D
Quantidade de partículas	5000
Tempo Iteracao	0.004
Limites Cenario	$(1.5, 1.5, 1.5)$
Pos Cubo Liquido	$(0, 0, 0)$
Tam Cubo Liquido	$(0.1, 0.1, 0.1)$
Pos X Barreira Dam break	-9999
Escala Esfera	0.025
Cor Particula	*Velocidade*
Cor Particula Vel Max	2.5

4.1.3 Rotação do cenário

Também foi realizado um experimento onde o usuário poderia rotacionar os limites do cenário, fazendo com que fluido reagisse ás alterações de rotação. Para executar esta simulação a rotação em si não era realizada diretamente nos limites do cenário, mas sim no vetor de gravidade. Ao rotacionar o vetor de gravidade e a visualização da simulação, foram obtidos resultados em que a rotação parecia estar sendo aplicada diretamente nos limites do cenário. A Figura 4.3 é uma captura de tela obtida durante este experimento (nesta simulação foi utilizada uma opção de coloração onde todas as partículas receberam uma cor fixa durante toda a simulação).

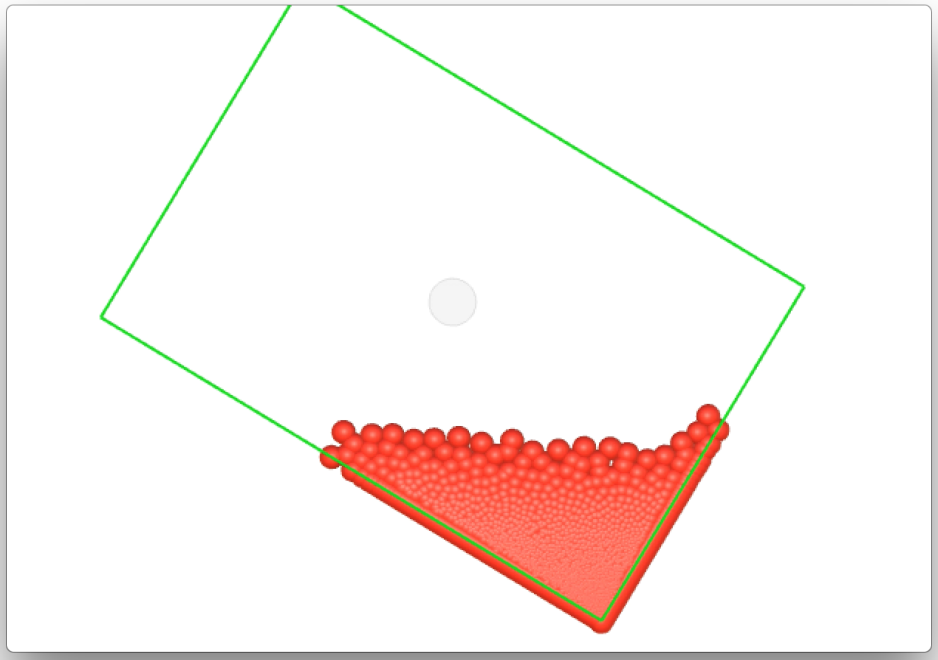


Figura 4.3: Captura de tela de uma simulação onde o usuário pode rotacionar todo o cenário da simulação.

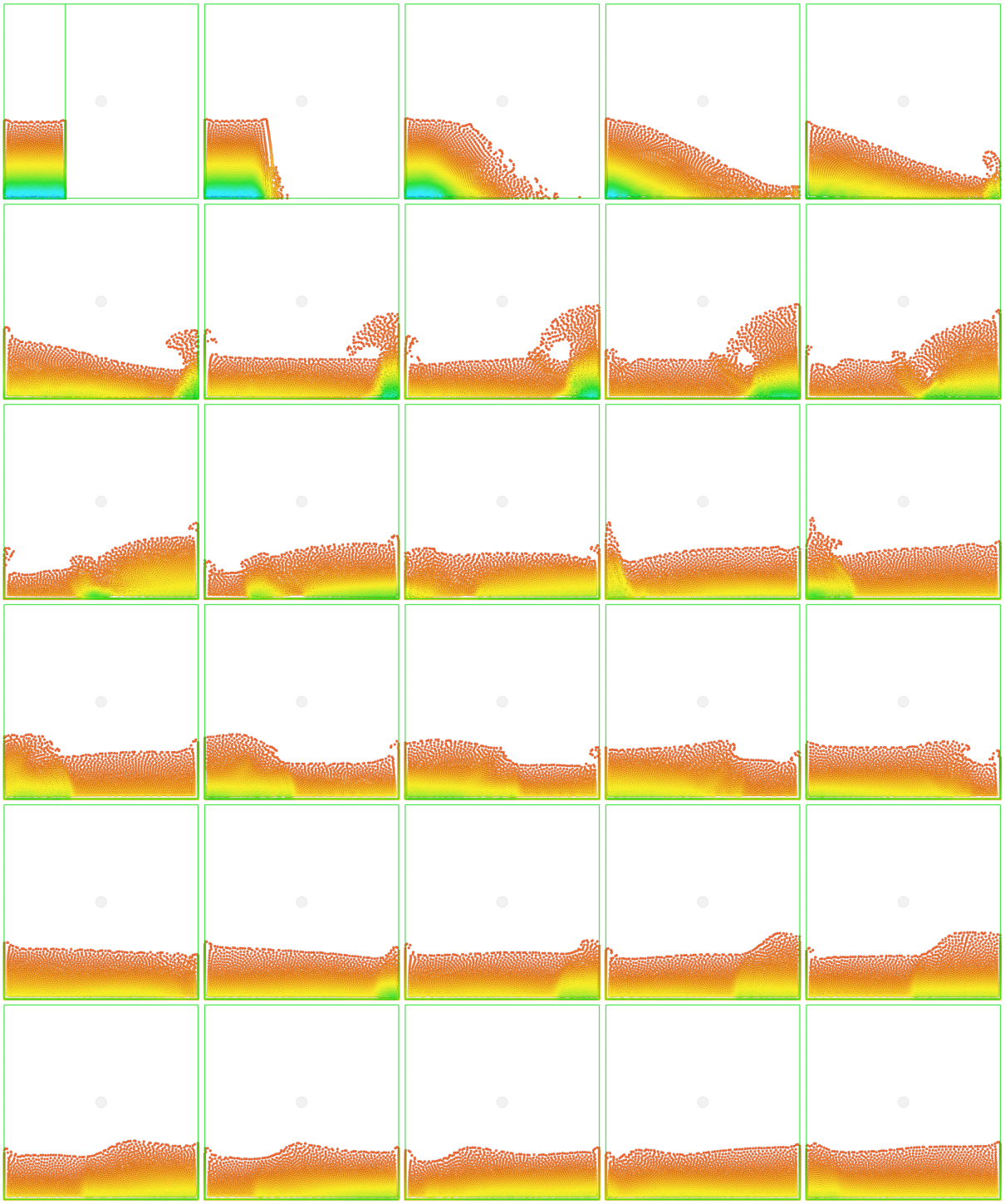


Figura 4.1: Frames de uma simulação com *Dam Break* gerada no Unity.

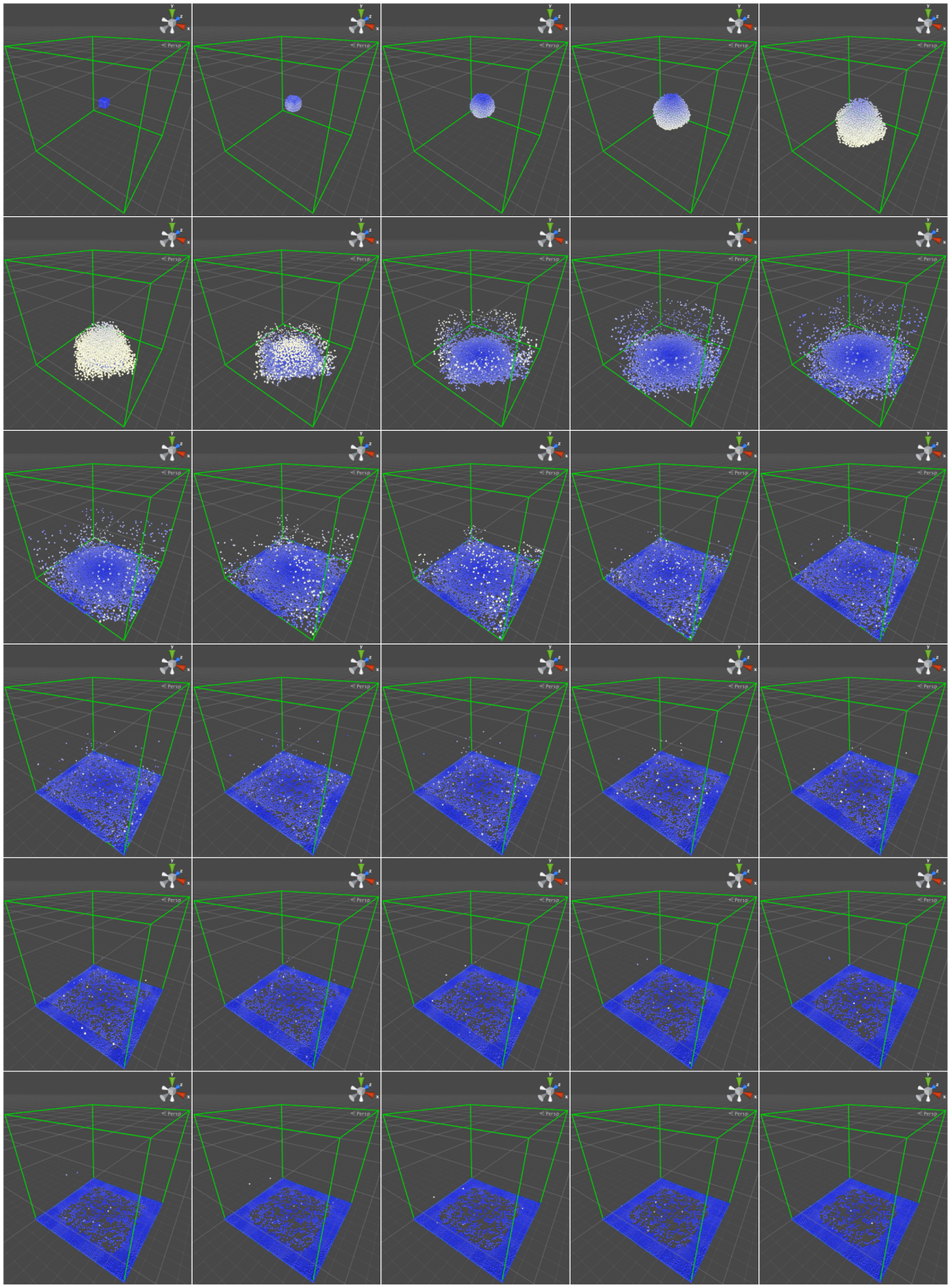


Figura 4.2: Frames de uma simulação em três dimensões gerada no Unity utilizando a coloração de partículas por velocidade.

5 Considerações Finais

Este trabalho foi iniciado com um estudo do método SPH, utilizando funções de núcleo baseadas no trabalho de Müller et al (2003). A equação de Navier-Stokes foi utilizada para relacionar a aceleração das partículas com as forças de pressão e de viscosidade. O método Leap-Frog foi adotado para realizar o movimento das partículas, atualizando os valores das velocidades e posições das mesmas. A implementação do SPH foi convertida para uma biblioteca que foi integrada ao Unity, oferecendo ao usuário diferentes maneiras de visualização das partículas do fluido.

5.1 Conclusões

Embora o método SPH seja simples ele é estável e capaz de realizar simulações de fluidos realistas. As otimizações de código discutidas na seção 3.4 permitiram executar simulações de 12.000 partículas no Unity a uma taxa média de 10FPS utilizando um Macbook Pro com processador Intel Core i5 2.4GHz e 8GB de memória RAM.

Ao integrar o programa implementado com o Unity foi criada uma maneira fácil e rápida para o usuário configurar os parâmetros do SPH e analisar os efeitos das alterações realizadas. As diferentes opções de coloração de partículas (discutidas na sessão 3.6.3) permitem ao usuário analisar a mudança dos valores da densidade e da velocidade das partículas, e melhor compreender o comportamento do fluido.

O trabalho foi uma ótima oportunidade para o desenvolvedor aplicar o conhecimento obtido durante sua formação de Bacharel de Ciência da Computação, e dar continuidade ao projeto *Digital Liquid Light*, iniciado em 2014.

5.2 Trabalhos futuros

O próximo passo deste trabalho é realizar a renderização do fluido, deixando de representar as partículas como simples esferas. Um dos modos de renderizar um fluido é triangular a

isosuperfície através do algoritmo *Marching Cubes* (Müller et al, 2003).

Também será interessante adicionar a propriedade *cor* aos fluidos, realizando simulações com fluidos com cores e outras propriedades diferentes para analisar como os fluidos se misturariam e trabalhar com misturas de cores.

Embora a integração com o Unity possua uma interface amigável para o usuário, após iniciar a simulação o usuário pode apenas manipular a visualização da simulação, rotacionando e/ou aplicando zoom, e ativar o *Dam Break*. A iteração entre o usuário e a simulação durante sua execução é muito limitada.

Laursen (2013) desenvolveu um trabalho semelhante, criando sua própria implementação do SPH diretamente no Unity e totalmente em C#. Em sua implementação o usuário pode interagir diretamente com os fluidos, adicionando forças no sistema ao clicar na tela e arrastar o mouse. Os fluidos também iteragem com objetos simples na cena, como cubos e esferas, de forma que objetos possam ser empurrados pelos fluidos ou até boiarem.

Explorar técnicas de computação paralela ao realizar as iterações do SPH irá trazer grandes benefícios para o desempenho do algoritmo, permitindo realizar simulações com um número muito maior de partículas do que a implementação serial consegue manipular em tempo real. Plataformas de computação paralela como a CUDA (*Compute Unified Device Architecture*) (NVIDIA, 2016) permitem um programa utilizar a GPU (*Graphics Processing Unit*; Unidade de Processamento Gráfico, em português) através de uma abordagem chamada GPGPU (General-purpose computing on graphics processing units; Computação de propósito geral em unidades de processamento gráfico, em português) (Laursen, 2013). A utilização da GPU desta forma gera grandes melhorias de desempenho, sendo muito utilizada em aplicações de computação gráfica.

Adicionando todos estes novos recursos ao trabalho resultará em uma ferramenta que permitirá desenvolvedores a realizarem simulações de fluidos em seus jogos facilmente e obter iteração direta entre os fluidos e objetos ou até mesmo o jogador. Este resultado poderia ser utilizado em uma nova implementação do *Digital Liquid Light*, superando as barreiras encontradas na primeira versão do projeto.

Referências Bibliográficas

Andrade, L. F. d. S. **Animação de jatos oscilantes em fluidos viscosos usando SPH em GPU**. 2014. Tese de mestrado - Universidade de São Paulo.

Apple Inc. **OS X**. Disponível em: http://www.apple.com/osx/, Julho 2016.

Apple Inc. **Xcode**. Disponível em: https://developer.apple.com/xcode/, Julho 2016.

Atkinson, K. E. **An Introduction to Numerical Analysis**. JOHN WILEY & SONS INC, 1989.

de Berg, M.; Cheong, O.; van Kreveld, M. ; Overmars, M. **Computational Geometry**. Springer Berlin Heidelberg, 2008.

Crespo, A. J. C. **Application of the Smoothed Particle Hydrodynamics model SPHysics to free-surface hydrodynamics**. 2008. Tese de doutorado - Universidade de Vigo.

Desbrun, M.; Cani, M.-P. **Smoothed particles: A new paradigm for animating highly deformable bodies**. In: Computer Animation and Simulatioń96 (Proceedings of EG Workshop on Animation and Simulation), p. 61–76. Springer, aug 1996.

Electronic Arts Inc. **Alice: Madness returns**. Disponível em: http://www.ea.com/alice, Julho 2011.

Ferziger, J. H.; Peric, M. **Computational Methods for Fluid Dynamics**. Springer-Verlag GmbH, 2001.

Gearbox Software, LLC. **Borderlands 2**. Disponível em: http://www.gearboxsoftware.com/game/borderlands-2/, Julho 2012.

Google Inc. **Liquidfun**. Disponível em: http://google.github.io/liquidfun/, Julho 2016.

Gourlay, M. J. **Fluid simulation for video games (part 1)**. Intel Developer Zone Disponível em: https://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1, Julho 2012.

Green, S.; Tonge, R.; Sainz, M.; Johnston, D. ; Schoemehl, D. **Fluid simulation in alice: Madness returns**. NVIDIA Developer Disponível em: https://developer.nvidia.com/content/fluid-simulation-alice-madness-returns, Junho 2011.

Kitware Inc; Sandia National Laboratory ; Los Alamos National Laboratory. **Paraview**. Disponível em: http://www.paraview.org/, Julho 2016.

Laursen, J. C. M. **Improving the Realism of Real-Time Simulation of Fluids in Computer Games**. 2013. Tese de Doutorado - Aalborg University.

Microsoft Corporation. **Kinect**. Disponível em: https://developer.microsoft.com/en-us/windows/kinect, Julho 2016.

Microsoft Corporation. **Windows**. Disponível em: https://www.microsoft.com/windows/, Julho 2016.

Müller, M.; Charypar, D. ; Gross, M. **Particle-based fluid simulation for interactive applications**. In: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, p. 154–159. Eurographics Association Aire-la-Ville, Switzerland, 2003.

NVIDIA Corporation. **CUDA**. Disponível em: http://www.nvidia.com/object/cuda_home_new.html, Julho 2016.

New Line Productions. **The lord of the rings: The return of the king**. Disponível em: http://www.lordoftherings.net/home.htm, Julho 2003.

Paiva, A.; Petronetto, F.; Tavares, G. ; Lewiner, T. **Simulação de Fluidos sem Malha: Uma introdução ao método SPH**. IMPA, 2009.

Physical Liquid Software. **Liquid physics 2d**. Disponível em: https://www.assetstore.unity3d.com/en/#!/content/23907, Julho 2016.

Pnueli, D.; Gutfinger, C. **Fluid Mechanics**. Cambridge University Press, 1997.

Pritchard, P. J.; McDonald, A. T. **Fox and McDonald's Introduction to Fluid Mechanics**. JOHN WILEY & SONS INC, 2011.

Renhe, M. C. **Morphing empregando modelos de difusão de fumaça**. 2010. Dissertação de mestrado - Universidade Federal do Rio de Janeiro.

Robb, D. **Smoothed particle hydrodynamics simulations of freely moving solid objects in a free-surface flow with applications to river ice dynamics**. 2012. Tese de Doutorado - McGill University.

Torvalds, L.; Hamano, J. **Git**. Disponível em: https://git-scm.com/, Julho 2016.

Twentieth Century Fox Film Corporation. **Ice age**. Disponível em: http://www.iceagemovies.com/, Julho 2002.

Unity Technologies. **Unity**. Disponível em: http://unity3d.com/pt, Julho 2016.

Warner Bros. Entertainment Inc. **Superman returns**. Disponível em: http://www.lordoftherings.net/home.htm, Julho 2006.

Xamarin. **Monodevelop**. Disponível em: http://www.monodevelop.com/, Julho 2016.