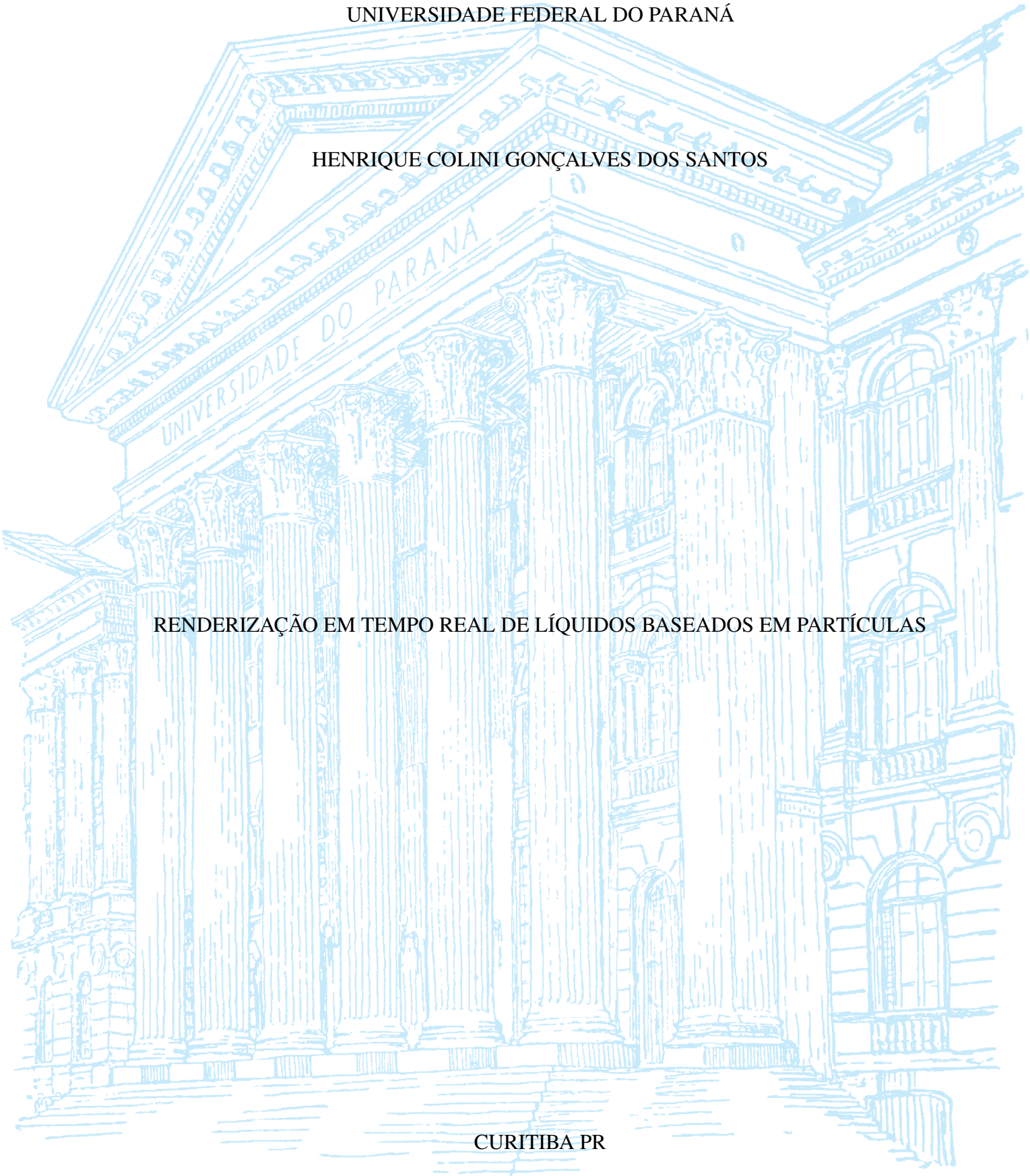
UNIVERSIDADE FEDERAL DO PARANÁ

HENRIQUE COLINI GONÇALVES DOS SANTOS



RENDERIZAÇÃO EM TEMPO REAL DE LÍQUIDOS BASEADOS EM PARTÍCULAS

CURITIBA PR

2025

HENRIQUE COLINI GONÇALVES DOS SANTOS

RENDERIZAÇÃO EM TEMPO REAL DE LÍQUIDOS BASEADOS EM PARTÍCULAS

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: Wagner Machado Nunan Zola.

CURITIBA PR

2025

Aos meus pais.

AGRADECIMENTOS

Este trabalho, assim como toda a minha trajetória acadêmica, só foi possível graças aos meus pais, Ivanclei e Noemi. Desde muito cedo, dedicaram tempo, esforço e recursos à minha educação e ao meu aprendizado. Em diversos momentos, enfrentaram dificuldades financeiras, desafios sociais e períodos de instabilidade; ainda assim, minha formação sempre foi tratada como prioridade, por reconhecerem nela o principal caminho para a construção do meu futuro. Sempre incentivaram minha curiosidade e criatividade e jamais limitaram meus sonhos. Por esses e tantos outros motivos, sou-lhes eternamente grato.

Registro também um agradecimento especial à pessoa que me acompanhou ao longo de toda a minha jornada universitária e que me ofereceu o suporte necessário para que eu permanecesse firme durante a elaboração deste trabalho: meu melhor amigo, atual noivo e futuro esposo, Mateus. Agradeço profundamente por estar sempre ao meu lado.

RESUMO

A renderização de fluidos é um tema central em computação gráfica, pois fenômenos líquidos estão presentes em inúmeras situações do cotidiano – da água que escorre por uma torneira à tinta utilizada na pintura de superfícies. Em razão disso, a Dinâmica de Fluidos Computacional tornou-se fundamental em áreas como engenharia, medicina, animação e desenvolvimento de jogos digitais, onde a representação visual de simulações é parte indispensável do processo.

Em aplicações interativas, simulações de líquidos são frequentemente modeladas por meio de conjuntos de partículas, seguindo uma abordagem Lagrangiana. Nesse contexto, este trabalho investiga diferentes técnicas de renderização em tempo real de líquidos baseados em partículas, oferecendo uma visão ampla tanto dos fundamentos quanto de aspectos práticos. Realizamos uma revisão dos principais métodos utilizados na área, desde o clássico *Marching Cubes* até estratégias modernas baseadas em *ray marching*.

Em seguida, implementamos cada método utilizando a API gráfica WebGPU, por meio da linguagem Rust, de modo a avaliar sua viabilidade em hardware contemporâneo. Por fim, apresentamos e discutimos resultados experimentais que comparam desempenho e fidelidade visual das abordagens estudadas.

Este trabalho se propõe a ser um recurso a ser utilizado como referência para pesquisadores e desenvolvedores interessados no tema, que podem ter dúvidas sobre a escolha do método que melhor atinge seus objetivos, e sobre aspectos práticos relacionados à implementação destes.

Palavras-chave: Renderização de fluidos. Renderização em Tempo Real. Abordagem Lagrangiana para Simulação de Fluidos. Líquidos Baseados em Partículas.

ABSTRACT

Fluid rendering is a major theme in computer graphics, given how ubiquitous liquid phenomena are in everyday situations – from the water that flows from a tap, to the paint used for coloring surfaces. Because of that, Computational Fluid Dynamics has become fundamental in areas such as engineering, medicine, animation, and video game development, in which the visual representation of such simulations is an indispensable part of the workflow.

In interactive applications, liquid simulations are frequently modeled as sets of particles, following a Lagrangian approach. In this context, this work investigates different techniques for real-time rendering of particle based liquids, offering a broad view of both theoretical foundations and practical aspects. We review the main methods used in this field, from the classic Marching Cubes algorithm to modern strategies based on ray marching.

Next, will implement each method using the WebGPU graphics API, through the Rust programming language, in order to evaluate its viability in modern hardware. Lastly, we will show and discuss experimental results that compare performance and visual fidelity of the studied approaches.

This work aims to be a reference resource for researchers and developers interested in the field, who might have questions regarding the choice of the methods that best suit their goals, and regarding practical aspects of their implementations.

Keywords: Fluid rendering. Real-time rendering. Lagrangian take on fluid simulations. Particle based liquids.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

UFPR	Universidade Federal do Paraná
CFD	*Computational Fluid Dynamics*
SPH	*Smoothed Particle Hydrodynamics*
PBVR	*Particle Based Volume Rendering*
SDF	*Signed Distance Field*
GPU	*Graphics Processing Unit*
CPU	*Central Processing Unit*

LISTA DE SÍMBOLOS

$\int$	Integral.
$\sum$	Somatório.
$\mathbf{x}$	Uma variável em negrito é um vetor.
$\|\mathbf{x}\|$	Norma do vetor $\mathbf{x}$.
$\mathbb{R}$	Conjunto dos números reais.
$\in$	Símbolo de "pertence a".
$\forall$	Símbolo de "para todo".
σ	Sigma, desvio padrão.
ρ	Rô, densidade.
π	Pi, constante matemática.
e	Constante de Euler.
A'	Se A é uma função, A' é sua derivada.
∇	Gradiente.
$\lfloor x \rfloor$	Piso de x.

SUMÁRIO

1 INTRODUÇÃO

Fluidos estão presentes em praticamente todos os aspectos do mundo natural e artificial: da água que percorre o delta de um rio ao café que preenche uma xícara; dos vórtices de plasma que se formam na superfície do Sol às tintas que se misturam em um balde; dos gases que saem pelo escapamento de um carro a inúmeras outras manifestações cotidianas e científicas. O estudo e a visualização desses fenômenos desempenham um papel essencial em áreas tão diversas quanto ciência, engenharia, medicina e entretenimento digital.

A Dinâmica de Fluidos Computacional (*Computational Fluid Dynamics*, CFD) consolidou-se como uma ferramenta indispensável para a resolução de problemas complexos que envolvem o comportamento de líquidos, gases e massas em diferentes escalas, incluindo as planetárias. Entre os diversos métodos numéricos disponíveis, os baseados em partículas, como o *Smoothed Particle Hydrodynamics* (SPH), destacam-se por sua versatilidade, maturidade conceitual, relativa simplicidade de implementação e ampla gama de aplicações (Liu e Liu, 2003).

A visualização desses sistemas constitui um ramo fundamental da Computação Gráfica. É importante observar que as técnicas de renderização podem ser divididas em duas categorias principais de acordo com o tempo de processamento necessário para a geração de imagens: a renderização em tempo real, que produz quadros com rapidez suficiente para transmitir a sensação de movimento contínuo; e a renderização não interativa ou *offline*, na qual cada imagem exige um tempo considerável de processamento, priorizando-se a máxima fidelidade visual (Akenine-Möller et al., 2018). Neste trabalho, concentraremos a análise na renderização em tempo real de líquidos baseados em partículas, particularmente relevante para aplicações interativas, como ferramentas científicas e jogos digitais.

Embora o foco seja a renderização de líquidos, muitas das técnicas discutidas podem ser estendidas para outras aplicações envolvendo dados volumétricos baseados em partículas. O uso de partículas como primitivos gráficos que preenchem o espaço revela-se um método eficiente e interativo para a Renderização Volumétrica, conforme demonstram pesquisas em *Particle-Based Volume Rendering* (PBVR) (Sakamoto et al., 2007).

Outro fator de motivação está na evolução das APIs gráficas voltadas à web. O WebGL, amplamente utilizado desde 2011, apresenta limitações frente às demandas atuais, incapaz de explorar de forma plena o paralelismo e as capacidades modernas das GPUs. O WebGPU, lançado oficialmente em navegadores como o Chrome em 2023, representa um avanço significativo: oferece acesso de baixo nível aos recursos da GPU, controle explícito sobre memória e paralelismo, e suporte a *compute shaders*. *Shaders* (do inglês "*shading*", "sombreamento") são programas que executam numa GPU. Como o nome sugere, estes são tradicionalmente empregados para o cálculo de cores e iluminação de superfícies. *Compute shaders* são uma generalização de *shaders*, que permitem que desenvolvedores realizem computações arbitrárias não limitadas à geração de imagens.

A adoção do WebGPU torna viável a renderização de sistemas complexos diretamente em navegadores modernos, ampliando o alcance dessa tecnologia. Os navegadores representam hoje a plataforma de software mais difundida do mundo, acessível em praticamente qualquer dispositivo conectado à internet. Diferentemente de soluções nativas que demandam hardware gráfico especializado, configurações complexas e conhecimento técnico avançado, aplicações baseadas em WebGPU podem ser executadas em contextos cotidianos de trabalho por engenheiros, cientistas ou astrônomos. Além disso, a tecnologia mostra-se promissora para jogos digitais

com gráficos avançados em um ambiente acessível a usuários de todas as idades. Esse caráter multiplataforma abre ainda novas perspectivas para a educação: estudantes podem explorar técnicas avançadas de visualização científica diretamente no navegador, sem a necessidade de instalar softwares complexos como o *ParaView* (Kitware, 2025) ou ferramentas proprietárias, facilitando o primeiro contato com a computação gráfica e incentivando novas gerações a se interessarem pela área.

Neste trabalho, realizaremos uma revisão geral da área de renderização 3D em tempo real, com ênfase em aspectos relacionados à visualização de sistemas de partículas de caráter volumétrico. A partir dessa base, faremos uma análise comparativa de desempenho e qualidade entre diferentes métodos de renderização de líquidos, através da implementação e experimentação com protótipos interativos implementados usando a API WebGPU através das linguagens de programação Rust e WGSL.

É importante destacar que o foco do trabalho não é a simulação dos fluidos em si, mas exclusivamente em sua renderização. Por este motivo, não entraremos em tantos detalhes aprofundados a cerca de aspectos matemáticos e práticos da construção de simulações; apenas aqueles que acabam sendo relevantes para a renderização.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, apresentaremos uma revisão histórica e conceitual essencial para o desenvolvimento deste trabalho. Inicialmente, será feita uma visão geral da renderização 3D, com destaque para aspectos relevantes à visualização de fluidos. Em seguida, analisaremos diferentes abordagens de simulação, com foco nos métodos baseados em partículas, como o SPH.

2.1 VISÃO GERAL DA RENDERIZAÇÃO 3D

Na computação gráfica, o termo *renderização* refere-se ao processo de geração de uma imagem a partir de uma cena virtual, para ser exibida em um dispositivo de saída, como um monitor. A *renderização em tempo real*, por sua vez, caracteriza-se pela capacidade de produzir essas imagens de forma contínua e suficientemente rápida para compor sequências animadas interativas, de modo que o observador perceba um fluxo visual suave, em vez de quadros individuais estáticos (Akenine-Möller et al., 2018).

Especificamente, a renderização tridimensional em tempo real busca criar imagens a partir de representações internas de um espaço 3D simulado no computador. O objetivo é reproduzir, de forma convincente, um ambiente visual compatível com a percepção natural dos usuários, que estão imersos em um mundo físico tridimensional.

2.1.1 Métodos Geométricos

A forma mais amplamente utilizada de renderização tridimensional é, sem dúvida, a baseada em malhas geométricas compostas por pontos, arestas e triângulos. A maioria das placas gráficas (GPUs) possui *hardware* otimizado para a *pipeline* de renderização fundamentada nesses primitivos geométricos.

Nesse método, malhas pré-definidas de triângulos que definem a superfície de algum objeto são enviadas para a GPU, que então executa diversas etapas até a construção da imagem final: transformações de coordenadas, rasterização (processo de preencher cada triângulo com pixels), *shading* (cálculo da cor de cada pixel), entre outras. Os triângulos são versáteis, pois podem representar praticamente qualquer objeto tridimensional, desde que em quantidade suficiente para garantir fidelidade visual.

Apesar de sua ubiquidade, o paradigma geométrico tradicional apresenta problemas importantes quando aplicado à renderização de fluidos. Isso ocorre porque líquidos e gases não possuem superfícies fixas bem definidas, mas sim fronteiras dinâmicas, que se deformam constantemente em função do movimento das partículas que os compõem.

Representar tais fronteiras com malhas triangulares exige reconstruções frequentes e custosas, algo que apesar de comum na renderização offline, pode se tornar um gargalo em aplicações de tempo real.

2.1.2 Métodos Alternativos de Renderização

Embora a renderização baseada em triângulos seja dominante na computação gráfica interativa, surgiram diversas abordagens que buscam lidar com superfícies e volumes de forma mais direta e adequada para fenômenos dinâmicos, como os fluidos. Ao representar apenas a superfície de um objeto, perde-se uma dimensão inteira de informação – o seu conteúdo interno (Kaufman, 2003). No entanto, muitas aplicações dependem justamente dessa riqueza volumétrica:

simulações de gases na engenharia (Strakos et al., 2025), geração de imagens médicas (Zhang et al., 2011) ou a visualização de partículas em simulações de fluidos (Xiao et al., 2017).

Um exemplo relevante é a renderização baseada em pontos (*point-based rendering*), introduzida por Levoy e Whitted (1985). Nesse paradigma, as superfícies ou volumes são representados diretamente como conjuntos de pontos ou *surfels* (*surface elements*), que armazenam atributos como normais e cores. Tal abordagem é especialmente adequada para sistemas de partículas, já que dispensa a necessidade de reconstruir superfícies explícitas, permitindo a visualização direta do conjunto de partículas que define o fluido.

Voxels (*volume elements*) também se destacam como uma representação regular de dados volumétricos. Diferentemente dos polígonos, voxels preenchem a estrutura interna do objeto e permitem manipulação direta de propriedades como densidade, cor e opacidade. Em simulações de fluidos, os voxels podem ser usados para rasterizar partículas em grades tridimensionais e, em seguida, aplicar técnicas de visualização volumétrica. Uma importante técnica de renderização de superfícies de dados volumétricos, a *Marching Cubes* (que veremos em detalhes na Seção 3.1) utiliza o conceito de voxels como base para a construção das malhas.

Por fim, temos os métodos baseados na dispersão de raios: *raytracing* e *raymarching*. Em ambos os métodos, ao invés da rasterização direta de triângulos, raios são lançados a partir da câmera em direção à geometria da cena, colidindo ou não com algum objeto. A diferença entre *raytracing* e *raymarching* é sutil, porém importante: no *raytracing*, se calcula a distância entre o sensor da câmera e um objeto diretamente, normalmente um triângulo. Ou seja, se utilizam equações de intersecção de linhas e planos. *Raymarching* por sua vez não utiliza equações de intersecção: os raios "marcham"através do espaço 3D incrementalmente, coletando amostras de dados volumétricos, até que se decida que este atingiu um objeto (Glassner, 1989; Hadji-Kyriacou e Arandjelović, 2021). *Raymarching* é particularmente interessante do ponto de vista de renderização de fluidos, uma vez que estes não possuem fronteiras definidas explicitamente, o que dificulta (ou impossibilita) o uso de equações de intersecção.

2.2 SIMULAÇÕES DE FLUIDOS

Embora o foco deste trabalho não seja a simulação de fluidos em si, é fundamental compreender os princípios básicos da Dinâmica de Fluidos Computacional (*Computational Fluid Dynamics*, CFD), uma vez que diversos aspectos técnicos da modelagem influenciam diretamente as escolhas e restrições das técnicas de renderização empregadas.

A simulação computacional de fluidos pode ser classificada em duas abordagens principais, baseadas em diferentes descrições matemáticas do contínuo: as abordagens **Euleriana** e **Lagrangiana** (Bridson, 2015).

2.2.1 Abordagem Euleriana

Na abordagem **Euleriana**, o domínio do fluido é discretizado em uma malha tridimensional composta por pontos fixos e igualmente espaçados. As propriedades físicas do fluido – como velocidade, pressão e densidade – são armazenadas nesses pontos, e a evolução temporal é observada à medida que o fluido escoa através da malha. Essa é a formulação predominante na CFD tradicional, amplamente utilizada em contextos industriais e científicos.

Nessa representação, não há o movimento explícito de partículas individuais: o transporte de massa e de outras propriedades é obtido a partir da resolução de equações diferenciais parciais que modelam o comportamento contínuo do fluido, como as equações de Navier–Stokes.

Consequentemente, as técnicas de visualização associadas a sistemas Eulerianos são, em geral, baseadas em **renderização volumétrica direta**, na qual a informação tridimensional é

convertida em uma imagem bidimensional sem a necessidade de reconstrução de superfícies. Entre os métodos mais empregados nesse contexto está o *ray marching*, que consiste em lançar raios através do volume e acumular amostras de densidade e cor ao longo do percurso (Rauter et al., 2024).

2.2.2 Abordagem Lagrangiana

Na abordagem **Lagrangiana**, o fluido é representado por um conjunto discreto de partículas móveis. Cada partícula corresponde a uma pequena porção do fluido e carrega atributos como posição, velocidade, densidade e temperatura, sendo acompanhada individualmente ao longo da simulação. Essa descrição é conceitualmente intuitiva e guarda semelhanças com a dinâmica molecular e com métodos utilizados para modelar sólidos deformáveis.

O método Lagrangiano mais consolidado e amplamente utilizado é o *Smoothed Particle Hydrodynamics* (SPH). Nesse modelo, o movimento das partículas é determinado por integrações das equações de Newton, em que as forças são calculadas a partir de amostragens contínuas de grandezas como pressão e densidade. Cada partícula contribui para a construção de um campo de densidade contínuo, por meio de uma função de suavização W – normalmente uma aproximação de uma distribuição gaussiana centrada na origem (Liu e Liu, 2003). Estas funções, conhecidas como *kernels*, delimitam um raio máximo de interação entre duas partículas h, de forma que a integral tridimensional $\iiint W(\|(x, y, z)\|, h)\, dx\, dy\, dz = 1$.

Sistemas Lagrangianos, por sua natureza baseada em partículas discretas, oferecem maior flexibilidade em termos de renderização. Uma das abordagens mais diretas é o *splatting*, que consiste em projetar um primitivo geométrico simples – como um quadrado ou disco – na posição de cada partícula. Essa técnica é amplamente utilizada em implementações iniciais e processos de *debugging*, mas, com aprimoramentos adequados, pode produzir resultados visualmente realistas, conforme discutiremos nas seções seguintes.

Outra possibilidade surge a partir do próprio campo de densidades reconstruído durante a simulação SPH. Como a amostragem desse campo é uma operação recorrente e altamente otimizada, torna-se viável aplicar técnicas de **renderização volumétrica** diretamente sobre ele, utilizando, por exemplo, o *raymarching* para acumular contribuições de densidade e cor no espaço tridimensional. Essa abordagem permite a visualização contínua e suave do fluido, preservando sua natureza volumétrica e dinâmica.

O restante deste trabalho será, portanto, dedicado à abordagem Lagrangiana, com foco em diferentes estratégias de renderização de fluidos baseados em partículas. Algumas das técnicas apresentadas farão uso de estruturas e otimizações típicas de simulações SPH – como as utilizadas para acelerar amostragens espaciais –, mas o escopo permanecerá restrito aos aspectos visuais da representação, sem aprofundar-se nos detalhes físicos ou numéricos da simulação em si.

2.3 CONCLUSÃO

Neste capítulo, estabelecemos os fundamentos teóricos que sustentam o desenvolvimento deste trabalho. Foram revisados conceitos essenciais de renderização 3D e discutidas as abordagens clássicas e alternativas aplicáveis à visualização de fenômenos fluidos.

Também analisamos os princípios da Dinâmica de Fluidos Computacional, destacando as diferenças entre as abordagens Euleriana e Lagrangiana. Esta última mostrou-se particularmente relevante para o contexto da renderização de fluidos baseados em partículas, por permitir representações mais diretas e dinâmicas do comportamento do fluido.

A compreensão desses conceitos fornece a base necessária para o estudo e implementação das técnicas de renderização apresentadas nos capítulos seguintes, orientando as decisões metodológicas e tecnológicas adotadas neste trabalho.

3 ESTRATÉGIAS DE RENDERIZAÇÃO

Neste capítulo, apresentamos três categorias distintas de abordagens para a renderização em tempo real de líquidos baseados em partículas. As técnicas discutidas são independentes do método específico de simulação utilizado, pois o objetivo é descrever estratégias de renderização amplamente aplicáveis. Assume-se apenas a capacidade de amostrar a densidade do fluido em pontos arbitrários do espaço de forma relativamente eficiente; uma operação central na maior parte das simulações Lagrangianas, como o SPH e variantes mais modernas. Assim, é natural e razoável que os métodos de renderização explorados neste capítulo dependam diretamente dessas consultas ao campo de densidade.

3.1 MARCHING CUBES

O algoritmo *Marching Cubes*, proposto originalmente por Lorensen e Cline (1987) em 1987, é uma das soluções clássicas para o problema de representação de dados volumétricos, sendo, portanto, uma estratégia amplamente utilizada para a visualização de líquidos e outros fenômenos contínuos. É importante destacar que o algoritmo não realiza a renderização propriamente dita, mas sim a construção de uma malha geométrica composta por triângulos, a qual pode posteriormente ser processada por uma *pipeline* de renderização tradicional.

A ideia básica por trás deste algoritmo é a de que podemos representar visualmente um conjunto de dados volumétricos a partir da renderização de uma **isosuperfície**. Uma isosuperfície de uma função contínua $f : \mathbb{R}^n \to \mathbb{R}$ é um subconjunto $I \in \mathbb{R}^n$ tal que $\forall \mathbf{x} \in I$, $f(\mathbf{x}) = t$, onde t é uma constante chamada *threshold* ou limiar. No caso em que $n = 2$, isosuperfícies tipicamente são chamadas de **curvas de nível**. No caso de líquidos, renderizamos uma isosuperfície da função de densidade $\rho(\mathbf{x})$.

Como o nome sugere, o método baseia-se na ideia de "marchar" pelo volume, percorrendo-o através de uma grade tridimensional de cubos. Define-se um tamanho de célula fixo que será utilizado para preencher o domínio volumétrico do fluido. Em cada cubo, são amostrados os valores de densidade do fluido em seus oito vértices. Comparando estes valores com um limiar pré-definido, determina-se se cada vértice está dentro ou fora da superfície do fluido. A Figura 3.1 mostra as curvas de nível de densidade de uma simulação SPH, e a Figura 3.2 mostra a isosuperfície obtida ao escolher um valor de limiar.

Na implementação original do algoritmo, a "marcha" é feita iterativamente, através de três laços de repetição aninhados. Porém, como a configuração de um cubo não influencia em nada na configuração de outro cubo, a ordem da iteração dos cubos é arbitrária, podendo até mesmo ser executada paralelamente. Em um ambiente de GPU, isso tipicamente se traduz para o uso de uma *thread* por cubo.

Com base nessas classificações binárias, o cubo é associado a uma das 256 configurações possíveis (duas alternativas para cada vértice), definindo o conjunto de triângulos que melhor aproxima a superfície naquele trecho. Os vértices dos triângulos são então posicionados nas arestas do cubo por meio de interpolação linear entre os pontos vizinhos, produzindo uma aproximação contínua e suave da interface do fluido. A Figura 3.3 mostra a malha obtida após a escolha da configuração de vértices e alinhamento por interpolação linear, num cenário em duas dimensões análogo ao tridimensional que estamos analisando.

O Algoritmo 1, adaptado de uma versão de Bourke (1994) mostra o processo de geração dos triângulos em uma única célula da grade tridimensional. Ele utiliza algumas tabelas pré

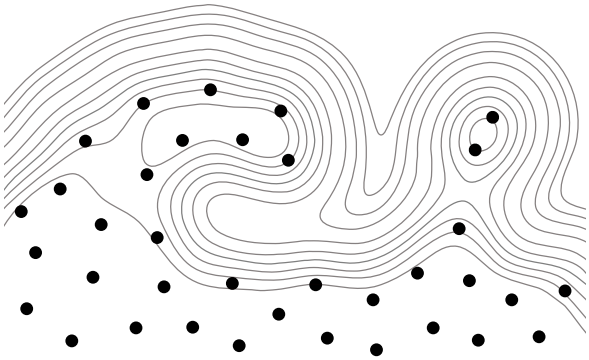


Figura 3.1: Curvas de nível da densidade do fluido em uma simulação SPH. Note que o valor varia continuamente, preenchendo todo o espaço.

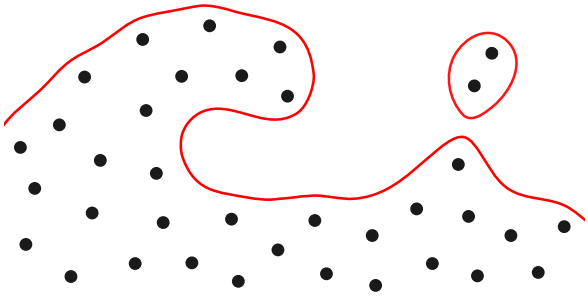


Figura 3.2: Uma das curvas de nível da imagem ao lado, escolhida como o limiar entre o líquido e seu exterior.

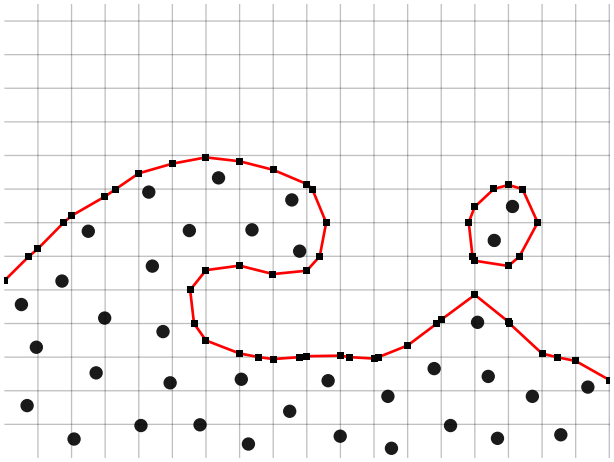


Figura 3.3: Malha de vértices obtida após a execução do algoritmo análogo ao *Marching Cubes* em duas dimensões, o *Marching Squares*. Note que as arestas se alinham conforme os valores amostrados nos cantos do quadrado, se ajustando à isosuperfície.

definidas que associam as classificações binárias à geometria a ser gerada. No algoritmo, a estrutura $TabelaArestas$ indica a presença de uma das 12 arestas do cubo para uma dada configuração. As $TabelaIndicesA$ e $TabelaIndicesB$ relacionam um índice de aresta (0-11) com os índices dos seus dois vértices (0-8). Finalmente, a $TabelaTriangulos$ aponta os índices dos trios de vértices presentes em uma configuração.

O Algoritmo 2 é utilizado para posicionar um vértice em um lugar entre os dois vértices de uma aresta. A posição é uma simples interpolação linear das posições, utilizando como peso os valores escalares presentes nas extremidades e o valor limiar desejado. Esta função garante que a malha seja gerada com planos que tangenciam a superfície implícita.

O método *Marching Cubes*, assim como outros métodos similares baseados na geração de polígonos, é muito utilizado para visualização de dados volumétricos estáticos. A utilização para renderização em tempo real de líquidos requer que a malha seja recriada novamente a cada quadro, o que possui um impacto de performance na ordem de $O(n^3)$, onde n é a resolução da simulação. Além disso, o algoritmo de geração da malha não é tão fácil de ser executado numa GPU, o que resulta na necessidade de enviar e receber conjuntos grandes de dados a cada quadro para implementações baseadas híbridas, CPU-GPU.

Algoritmo 1 Marching Cubes - execução em um único cubo

Entrada: célula de grade com posições $P[0..7]$, valores escalares $V[0..7]$, e valor de isosuperfície $isolevel$
Saída: lista $triangulos[]$ contendo os vértices dos triângulos da isosuperfície dentro da célula
$indiceCubo \leftarrow 0$
for $i \leftarrow 0$ **to** 7 **do**
    if $V[i] < isolevel$ **then**
        $indiceCubo \leftarrow indiceCubo + 2^i$
    end if
end for
if $TabelaArestas[indiceCubo] = 0$ **then**
    return 0 // A célula não intersecta a isosuperfície
end if
Criar vetor local $vertice[0..11]$
for $e \leftarrow 0$ **to** 11 **do**
    if $TabelaArestas[indiceCubo] \,\&\, 2^e$ **then**
        $a \leftarrow TabelaIndicesA[e]$
        $b \leftarrow TabelaIndicesB[e]$
        $vertice[e] \leftarrow InterpolaVertice(isolevel, P[a], P[b], V[a], V[b])$
    end if
end for
$triangulos \leftarrow$ lista vazia
for cada tupla (a, b, c) na lista $TabelaTriangulos[indiceCubo]$ **do**
    Criar triângulo $tri = (vertice[a], vertice[b], vertice[c])$
    Adicionar tri em $triangulos$
end for
return $triangulos$

Algoritmo 2 InterpolaVertice

Entrada: Valor de isosuperfície $isolevel$, pontos p_1, p_2, valores escalares v_1, v_2
Saída: Ponto interpolado p
if $|\, isolevel - v_1 \,| < \varepsilon$ **then**
    return p_1
else if $|\, isolevel - v_2 \,| < \varepsilon$ **then**
    return p_2
else if $|\, v_1 - v_2 \,| < \varepsilon$ **then**
    return p_1
end if
$\mu \leftarrow \dfrac{isolevel - v_1}{v_2 - v_1}$
$p \leftarrow p_1 + \mu \cdot (p_2 - p_1)$
return p

3.2 RENDERIZAÇÃO EM SCREEN SPACE

Uma das abordagens mais influentes para a renderização em tempo real de líquidos baseados em partículas é a técnica de pós-processamento em espaço de tela. Diferentemente de métodos que dependem da reconstrução explícita de superfícies tridimensionais, essa estratégia utiliza o *framebuffer* e operações de imagem para gerar superfícies contínuas e visualmente

coerentes a partir da projeção direta das partículas. Os trabalhos de van der Laan et al. (2009) foram alguns dos primeiros a descrever em detalhes essa técnica.

O método consiste em renderizar inicialmente as partículas como elementos individuais – geralmente por meio de *point sprites* ou *quads* orientados para a câmera –, semelhante a sistemas de partículas convencionais amplamente utilizados em jogos. Em seguida, aplicam-se filtros e operações matemáticas no espaço de tela para suavizar as transições entre partículas, criando a ilusão de uma superfície fluida e contínua.

Sua principal vantagem reside na eficiência computacional: todo o processo é executado diretamente na GPU, evitando a reconstrução de malhas geométricas complexas e minimizando a transferência de dados entre CPU e GPU. Adicionalmente, a técnica baseia-se em operações de processamento de imagem, que dispensam estruturas de dados volumétricas ou espaciais custosas. Essas características tornam a abordagem particularmente adequada para aplicações que demandam interatividade e desempenho estável, como jogos eletrônicos e simuladores em tempo real.

3.2.1 Etapas do pipeline

O método é dividido em quatro estágios principais:

1. **Projeção das partículas:** cada partícula é renderizada como uma esfera (ou *sprite*) com raio proporcional ao seu raio físico na simulação. O resultado dessa etapa é um mapa de profundidade (*depth map*) contendo o valor de profundidade mínimo de cada esfera ao longo do eixo da câmera. Note que não há a necessidade de utilizarmos uma malha densa de triângulos para renderizar esferas, uma vez que podemos escrever no mapa os valores equivalentes à profundidade de uma esfera a partir de simples equações trigonométricas. A Figura 3.4 representa a superfície idealizada de uma simulação de um líquido e a Figura 3.5 mostra como seria seu mapa de profundidade, visto a partir da câmera. A Figura 3.6 mostra o resultado de projetar as partículas como esferas, e consequentemente a Figura 3.7 mostra o verdadeiro mapa de profundidade obtido.

2. **Suavização da superfície:** o mapa de profundidade obtido contém descontinuidades entre partículas adjacentes. Para eliminar essas descontinuidades, aplica-se um filtro de difusão que suaviza as variações de profundidade preservando a curvatura da superfície. É possível utilizar um filtro tão simples quanto um Desfoque Gaussiano, porém há estratégias com melhores resultados, como veremos.

3. **Reconstrução de normais:** com a superfície suavizada, o mapa de normais é derivado do gradiente do campo de profundidade, permitindo o cálculo de iluminação de forma coerente.

4. **Renderização final:** por fim, o fluido é renderizado com base nos normais e profundidade reconstruídos, utilizando um modelo de iluminação adequado (por exemplo, *Blinn–Phong* ou *Fresnel-based shading*), resultando em uma aparência translúcida e contínua.

3.2.2 Suavização da Superfície

A etapa de suavização é fundamental para transformar a nuvem discreta de partículas em uma superfície contínua e visualmente plausível. Quando cada partícula é renderizada como uma pequena esfera no espaço da tela, o resultado tende a apresentar uma aparência granulada e

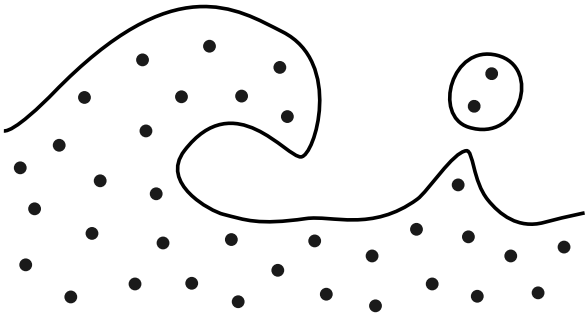


Figura 3.4: Uma superfície desejável em dado instante de uma simulação baseada em partículas.

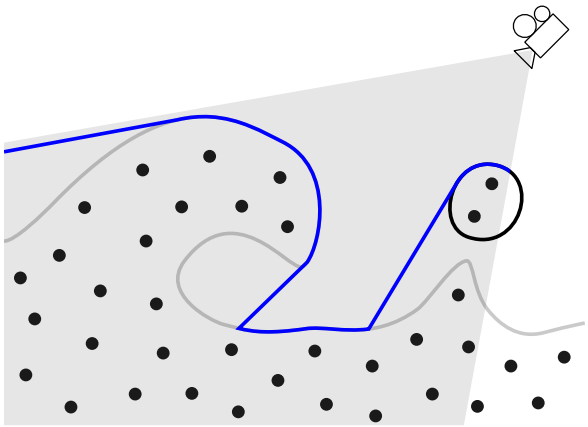


Figura 3.5: O mapa de profundidade desejado da cena, do ponto de vista da câmera. Note a mudança abrupta de profundidade nas bordas de objetos que bloqueiam o fundo.

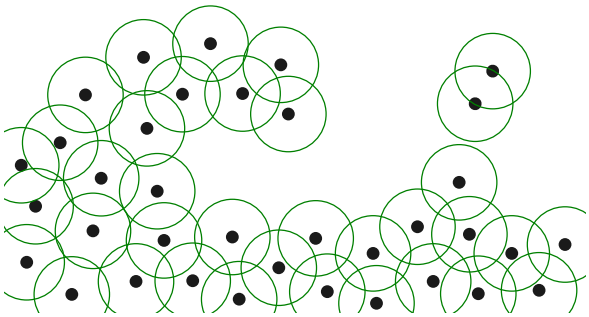


Figura 3.6: Configuração de partículas projetadas como esferas na cena.

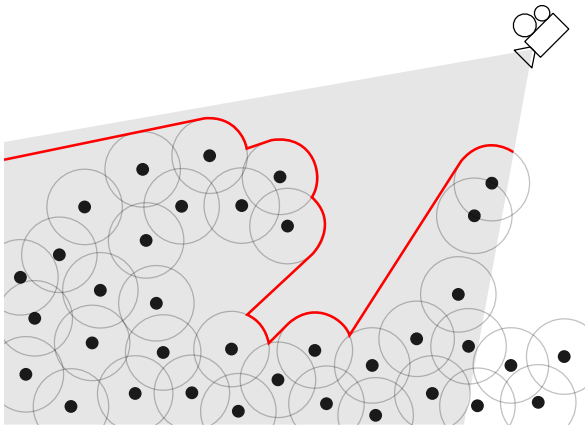


Figura 3.7: O mapa de profundidade obtido após a projeção da imagem ao lado. Note que não há transição suave entre as esferas, causando uma aparência de “geleia”.

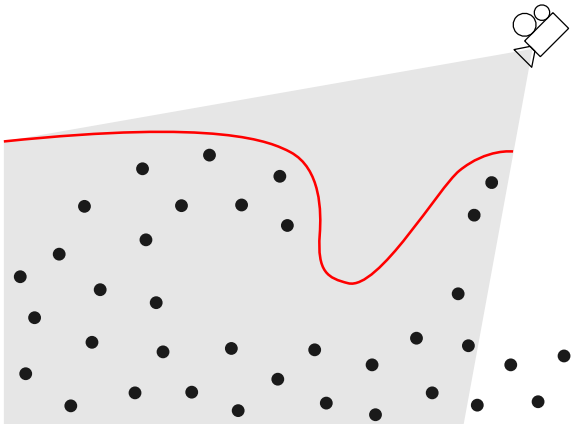


Figura 3.8: Mapa de profundidade suavizado utilizando desfoque gaussiano. Note que apesar da superfície das esferas não ter mais a transição rígida, o desfoque erroneamente mistura as divisas entre objetos que estão na frente ou atrás, causando artefatos de profundidade.

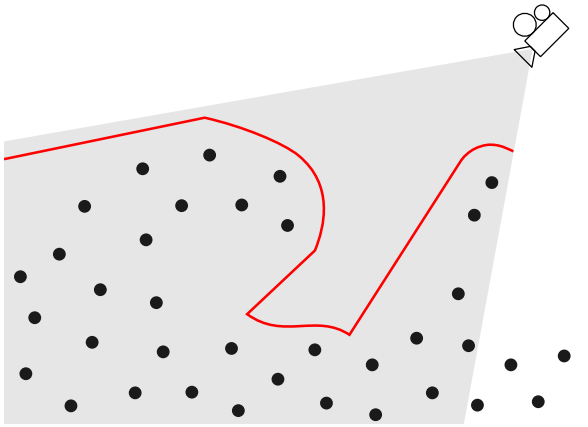


Figura 3.9: Mapa de profundidade suavizado com o filtro bilateral. Note que este filtro mantém corretamente os desníveis causados por objetos estarem na frente dos outros, se aproximando mais do mapa de profundidade desejado (Figura 3.5)

artificial, semelhante a uma substância gelatinosa (como a sobremesa sagu). O objetivo, portanto, é reconstruir uma superfície que pareça coerente e suave, aproximando-se da interface contínua esperada de um fluido físico.

Uma das formas mais simples e amplamente difundidas de suavização de superfícies é o desfoque gaussiano (*Gaussian blur*). Essa técnica faz parte de uma ampla classe de operações conhecidas como filtros de convolução. Em um filtro de convolução, o valor de cada ponto de uma imagem é recalculado como uma média ponderada dos valores de seus vizinhos, segundo uma máscara (ou *kernel* – não relacionado ao termo homônimo que se refere a funções que executam paralelamente em placas de vídeo, nem à função de suavização de uma simulação SPH) predefinida. De forma geral, para uma imagem discreta $I(x, y)$ e um kernel K, o resultado da convolução é dado por:

$$I'(x, y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} K(i, j)\, I(x + i, y + j)$$

onde o kernel tem $altura = 2a + 1$ e $largura = 2b + 1$. A escolha do kernel determina o comportamento do filtro – suavização, detecção de bordas, realce de detalhes, entre outros.

O filtro gaussiano utiliza como kernel a função gaussiana bidimensional:

$$K(i, j) = G(\|(i, j)\|, \sigma)$$
$$i \in [-a, a], j \in [-b, b]$$

onde σ é o parâmetro de suavização da função $G(r, \sigma)$:

$$G(r, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{r^2}{2\sigma^2}}$$

Essa função atribui pesos maiores a pixels próximos do centro e menores aos mais distantes, o que resulta em uma transição suave e natural entre regiões de intensidade. Naturalmente, como o *kernel* é uma função pura, independente de valores externos, e definida para um conjunto normalmente pequeno de pontos (normalmente regiões de tamanho 3×3, 5×5, etc), ele costuma ser pré computado e armazenado em uma matriz de pesos.

Uma característica vantajosa do filtro gaussiano é sua *separabilidade*: a convolução bidimensional pode ser decomposta em duas convoluções unidimensionais – uma no eixo x e outra no eixo y. Isso reduz o custo computacional, por pixel, de $O(ab)$ para $O(a + b)$, tornando o filtro eficiente e amplamente utilizado em aplicações de tempo real.[1]

Entretanto, o desfoque gaussiano apresenta um problema importante no contexto da renderização de fluidos: ele suaviza indiscriminadamente todas as regiões da imagem, inclusive nas bordas ou transições de profundidade. Isso significa que, ao aplicá-lo a um mapa de profundidade, as fronteiras entre o fluido e o ambiente – ou entre partes do fluido em diferentes distâncias da câmera – são borradas, resultando em artefatos visuais e perda de definição (como ilustrado na Figura 3.8).

Para contornar essa limitação, surge o **filtro bilateral** (Tomasi e Manduchi, 1998), uma extensão do filtro gaussiano que leva em consideração não apenas a distância espacial entre os pixels, mas também a diferença de intensidade (ou, no caso de mapas de profundidade, de valor de z). Assim, pixels que pertencem a regiões com profundidade semelhante são suavizados

[1] Para ser mais rigoroso, é importante frisar que tecnicamente, como a e b são constantes, em ambos os casos o tempo de computação por pixel é $O(1)$. A notação assintótica aqui foi empregada de maneira mais informal, para mostrar uma diferença real e perceptível do tempo necessário para filtros com tamanhos diferentes de kernel.

entre si, enquanto aqueles que cruzam descontinuidades de profundidade são preservados (como ilustrado na Figura 3.9). O filtro bilateral pode ser expresso como:

$$I'(x,y) = \frac{1}{W_p} \sum_{i=-a}^{a} \sum_{j=-b}^{b} G(\|(i,j)\|, \sigma_s)\, G(I(x,y) - I(x+i, y+j), \sigma_r)\, I(x+i, y+j)$$

$$W_p = \sum_{i=-a}^{a} \sum_{j=-b}^{b} G(\|(i,j)\|, \sigma_s)\, G(I(x,y) - I(x+i, y+j), \sigma_r)$$

onde G é novamente a função gaussiana, σ_s é o parâmetro gaussiano espacial (controla a taxa de desfoque geral), σ_r é o parâmetro gaussiano de domínio de intensidade (controla quanto a diferença de intensidade impacta na suavização) e W_p é o fator de normalização que garante que a soma dos pesos seja unitária.[2]

O inconveniente, contudo, é que o filtro bilateral *não é separável*, o que o torna mais caro computacionalmente, especialmente para kernels grandes. Apesar disso, aproximações eficientes e versões quase-separáveis têm sido propostas, permitindo um equilíbrio entre qualidade e desempenho (Banterle et al., 2012).

Outra abordagem conceitualmente distinta, porém intimamente relacionada, é o *Curvature Flow*; que é a proposta por van der Laan et al. (2009) em seu artigo original que introduz a técnica de renderização de líquidos que estamos analisando. Em vez de operar diretamente como um filtro de convolução, ele trata a suavização como um processo contínuo de evolução de superfícies. O método consiste em deslocar cada ponto da superfície ao longo de sua normal, em velocidade proporcional à curvatura média local. Em termos intuitivos, regiões convexas são levemente retraídas e depressões são suavemente preenchidas, resultando em uma superfície de curvatura menor a cada iteração e aparência mais fluida.

Esse processo é análogo ao comportamento físico da *tensão superficial* em líquidos, que busca reduzir a energia total da superfície, levando-a a formas suaves e estáveis. Embora o *Curvature Flow* tradicional seja formulado sobre superfícies tridimensionais contínuas, van der Laan et al. (2009) adaptaram o conceito para operar no espaço da tela – aplicando a evolução diretamente sobre o mapa de profundidade. Dessa forma, o valor de z de cada pixel é ajustado iterativamente de acordo com a curvatura estimada localmente, resultando em uma suavização visual sem borramento excessivo das bordas.

Os autores demonstram que essa técnica é capaz de produzir superfícies de fluidos visualmente contínuas e realistas, eliminando o aspecto granular típico de representações baseadas em partículas, sem recorrer a filtros caros como o bilateral. Apesar da simplicidade do conceito, o método envolve cálculos diferenciais relativamente complexos e, por isso, nem sempre é prático para aplicações em tempo real. Ainda assim, serve como uma importante inspiração para abordagens futuras de suavização baseadas em princípios geométricos e físicos.

3.2.3 Reconstrução das normais

Uma vez obtido o mapa de profundidade suavizado $z(x, y)$, as normais da superfície são aproximadas diretamente a partir dos gradientes locais:

[2] Nota-se que no artigo original de Tomasi e Manduchi (1998) a definição do filtro bilateral é mais geral e contínua, utilizando integrais que preenchem o espaço inteiro. Aqui, discretizamos as expressões substituindo-as por somatórios, calculando explicitamente as coordenadas dos vizinhos de um ponto (x, y).

$$\mathbf{n}(x, y) = \frac{(-\partial_x z,\ -\partial_y z,\ 1)}{\sqrt{(\partial_x z)^2 + (\partial_y z)^2 + 1}} \tag{3.1}$$

Essas normais são então utilizadas para o cálculo da iluminação, permitindo que o fluido apresente uma aparência coerente e realista sob diferentes condições de luz.

3.2.4 Considerações sobre desempenho

Diferentemente de métodos baseados em reconstrução de malha, o *Screen Space Fluid Rendering* apresenta complexidade linear em relação ao número de partículas projetadas. Além disso, todas as etapas do processo são adequadas à execução em *shaders* de fragmento, tornando a técnica altamente paralelizável. Em implementações modernas, o método pode renderizar dezenas de milhares de partículas a taxas superiores a 60 quadros por segundo.

3.2.5 Limitações

Por operar exclusivamente no espaço da tela, a técnica não preserva a geometria real do fluido fora da vista da câmera. Isso é um problema principalmente em situações nas quais o líquido atravessa as laterais do campo de visão da câmera, podendo apresentar artefatos causados pelo *warping ou clamping* da textura nas bordas. Além disso, a suavização deixa de funcionar de modo desejável caso a câmera se aproxime demais (ou entre) no líquido, revelando sua natureza discretizada em esferas – o que pode ser particularmente prejudicial para jogos digitais, onde a imersão do jogador é essencial. Ainda assim, o equilíbrio entre qualidade visual e desempenho a torna uma das soluções mais eficazes para renderização de fluidos em tempo real.

3.3 RAYMARCHING

Os métodos abordados anteriormente neste capítulo baseiam-se, de uma forma ou de outra, em estratégias clássicas de renderização a partir de primitivos geométricos. O *Marching Cubes*, de maneira explícita, constrói uma malha de triângulos que representa a fronteira do fluido, enquanto a técnica de *splatting* com pós-processamento em *Screen Space* utiliza partículas poligonais para compor o mapa de profundidade inicial.

Esta seção apresenta uma família moderna de abordagens para renderização 3D – a dispersão de raios. Alguns autores categorizam a dispersão de raios como uma técnica de espaço de tela (Wu et al., 2022), uma vez que opera por meio de processamento sobre a imagem final. Contudo, em sua forma mais pura, a dispersão de raios não se resume a um mero pós-processamento aplicado a uma cena gerada por polígonos – a imagem é *construída integralmente* pixel a pixel, tipicamente por meio de um *fragment shader*. Em tom coloquial, costuma-se dizer que essa categoria de técnicas permite "renderizar mundos inteiros com apenas dois triângulos" (Quilez, 2008).

O *raytracing* (traçado de raios) e o *raymarching* (marcha de raios) são técnicas de dispersão de raios. Ambas baseiam-se no mesmo princípio: calcular a cor de cada pixel individualmente a partir da simulação do percurso de raios de luz, idealizando o comportamento físico da dispersão de fótons no mundo real. Como discutido na Subseção 2.1.2, a distinção fundamental entre os dois está na forma como ocorre a intersecção entre o raio e a cena. O *raytracing* utiliza cálculos analíticos de intersecção com primitivos geométricos (quase sempre triângulos), enquanto o *raymarching* avança o raio incrementalmente no espaço até identificar uma colisão com a superfície implícita (Glassner, 1989; Hadji-Kyriacou e Arandjelović, 2021).

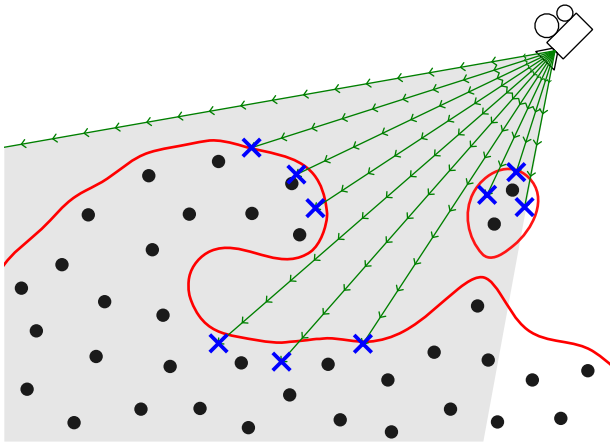


Figura 3.10: Representação de uma implementação ingênua de *raymarching* para líquidos baseados em partículas. A curva em vermelho é a isosuperfície do mesmo campo de densidade da Figura 3.1. As setas verdes representam raios "marchando" a partir da câmera. Os "X" em azul são o ponto em que o raio determinou estar dentro do líquido. Note que a maior parte do trabalho ocorre no espaço vazio longe da superfície.

Uma das principais vantagens do *raymarching* em relação ao *raytracing* é sua flexibilidade quanto aos tipos de superfícies representáveis. O *raytracing* depende da existência de soluções analíticas rápidas para as equações de intersecção, limitando-se a poucos tipos de primitivos. Já o *raymarching* pode operar sobre qualquer superfície definida por uma função implícita, o que o torna especialmente útil em contextos mais gerais. Por exemplo, ele é amplamente empregado na renderização de fractais tridimensionais, nos quais é fácil determinar se um ponto pertence ou não ao conjunto, mas extremamente difícil calcular uma intersecção direta com a geometria (Hart et al., 1989).

Essa característica torna o *raymarching* particularmente adequado à renderização de líquidos – em especial, aqueles simulados por métodos baseados em partículas, como o SPH, que produzem campos de densidade contínuos. Esses campos podem ser interpretados como funções implícitas, cuja superfície de nível define naturalmente a fronteira do fluido. A abordagem é conceitualmente semelhante à geração de malhas pelo *Marching Cubes*, mas o caráter volumétrico do *raymarching* permite resultados mais ricos: um raio lançado pela câmera pode ter seu percurso afetado gradualmente pela densidade variável do fluido. Em consequência, o método reproduz de forma mais natural fenômenos ópticos como transparência, reflexão e refração, resultando em visualizações mais realistas sem aumento expressivo da complexidade algorítmica.

O trabalho pioneiro de Xiao et al. (2017) representa um avanço significativo ao aplicar *raymarching* diretamente a fluidos simulados por partículas. Após observarem limitações inerentes às abordagens geométricas tradicionais – como a necessidade de reconstruir malhas via *Marching Cubes* ou a aparência artificial resultante do *splatting* com pós-processamento – os autores propuseram transpor para o domínio Lagrangiano uma estratégia até então restrita principalmente a simulações Eulerianas e à visualização de volumes contínuos. Essa adaptação não foi trivial: diferentemente de volumes estruturados, sistemas baseados em partículas não oferecem imediatamente uma função implícita suave apta ao avanço de raios. Ainda assim, os autores demonstram que, ao reconstruir um campo de densidade adequado e aplicá-lo a um esquema de *raymarching*, é possível obter uma superfície fluida coerente, estável e visualmente convincente, com um nível de detalhe superior ao das técnicas vistas anteriormente – e, crucialmente, mantendo desempenho suficiente para aplicações em tempo real.

Uma implementação ingênua de *raymarching* para líquidos consiste em emitir raios diretamente a partir da câmera e fazê-los avançar em passos de tamanho fixo, como ilustrado na

Figura 3.10. A cada passo, o raio amostra a densidade do fluido; caso essa densidade ultrapasse um limiar pré-definido, a cor correspondente é atribuída ao pixel associado àquela posição no espaço. O problema dessa abordagem é que, mesmo com a câmera ligeiramente afastada da região ocupada pelo fluido, o número de passos necessários até a superfície torna-se excessivo, desperdiçando tempo computacional ao atravessar grandes volumes de espaço vazio. Aumentar o tamanho do passo tampouco é uma solução aceitável: quanto maior a distância percorrida entre amostras, menor a precisão da reconstrução, pois o raio tende a superestimar a distância até o fluido, produzindo artefatos como o *banding* ilustrado na Figura 3.11, além de possibilitar o desaparecimento de estruturas finas demais para serem detectadas.

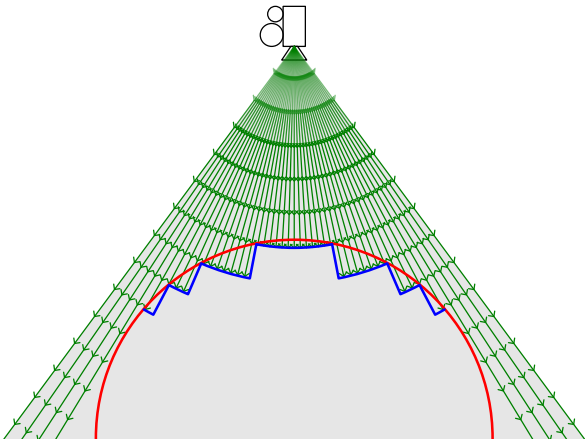


Figura 3.11: Problema de *banding* numa implementação ingênua de *raymarching*. O círculo vermelho representa uma isosuperfície esperada, enquanto a curva azul é a profundidade obtida.

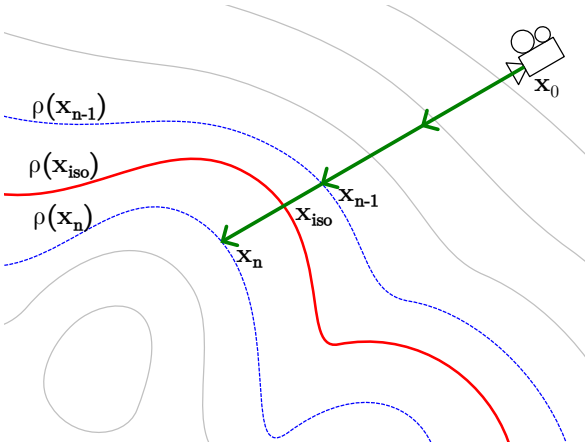


Figura 3.12: Solução para o problema de banding usando interpolação linear.

O artefato de *banding* é relativamente simples de mitigar. Utilizando as densidades nas duas últimas amostras do raio – isto é, nos pontos final e penúltimo – é possível realizar uma interpolação linear (conceitualmente semelhante ao procedimento descrito no Algoritmo 2 – *InterpolaVertice*) para "retornar" a posição estimada da superfície, assumindo localmente um campo de densidade aproximadamente linear. Assim,

$$t = \frac{\rho(\mathbf{x}_{iso}) - \rho(\mathbf{x}_{n-1})}{\rho(\mathbf{x}_n) - \rho(\mathbf{x}_{n-1})},$$

$$\|\mathbf{x}_{iso} - \mathbf{x}_0\| \approx (1-t)\,\|\mathbf{x}_n - \mathbf{x}_0\| + t\,\|\mathbf{x}_{n-1} - \mathbf{x}_0\|.$$

Aqui, $\mathbf{x}_0$ é a origem do raio, $\mathbf{x}_{iso}$ é o ponto de interseção com a isosuperfície, $\mathbf{x}_{n-1}$ e $\mathbf{x}_n$ correspondem às duas últimas posições amostradas, e $\rho(\mathbf{x})$ denota a densidade do fluido na posição $\mathbf{x}$. Observe que $\rho(\mathbf{x}_{iso})$ é, por definição, constante.

O problema do excesso de passos desperdiçados atravessando espaço vazio, entretanto, não é trivial de resolver. Em cenários estáticos, uma das soluções mais eficazes e amplamente utilizadas é o *raymarching* baseado em *Signed Distance Fields* (SDFs). Essa abordagem, conhecida desde a década de 1980 e hoje tão consolidada que muitos autores utilizam o termo "*raymarching*" como sinônimo de dispersão de raios sobre SDFs (Hadji-Kyriacou e Arandjelović, 2021), fundamenta-se em um princípio simples: dado um campo tridimensional que armazena, para cada ponto, a distância até a superfície mais próxima, o tamanho do passo tomado pelo raio é precisamente essa distância amostrada – e não um valor fixo. Com isso, o número total de passos diminui substancialmente, enquanto o tamanho do passo torna-se arbitrariamente pequeno

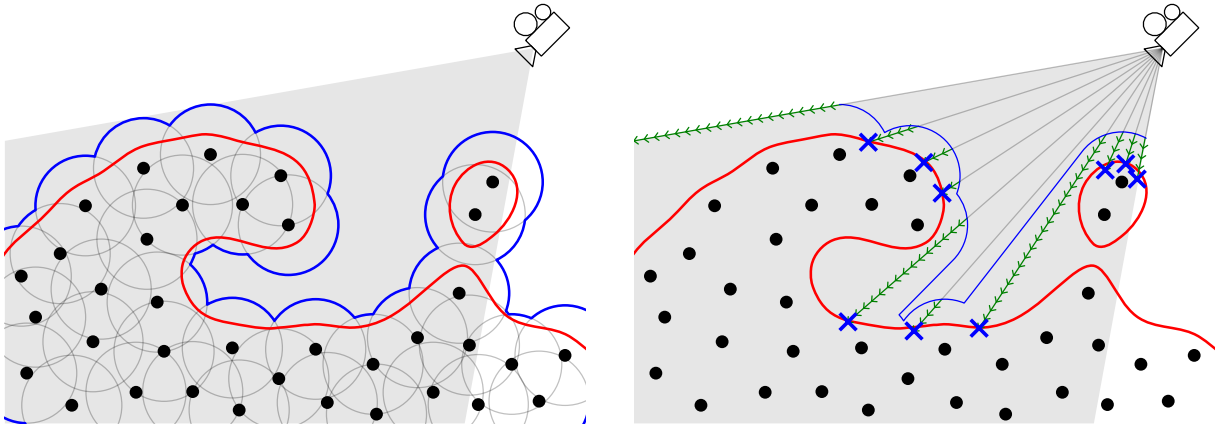


Figura 3.13: Otimização de Xiao et al. (2017) para resolver o problema dos passos inúteis em espaço vazio.

conforme o raio se aproxima da superfície. Ademais, como o valor amostrado representa, por definição, a menor distância possível até o objeto, não há risco de superestimativa.

Renderização via SDFs, contudo, exige – naturalmente – acesso a um SDF. Em aplicações de renderização *offline*, é viável calcular repetidamente a distância até diversos primitivos geométricos, para cada pixel. Em aplicações em tempo real com cenas estáticas, é comum armazenar o SDF em uma textura 3D pré-computada. No entanto, no contexto de simulações dinâmicas, recalcular todo o campo SDF a cada quadro seria computacionalmente proibitivo. Por esse motivo, é necessário buscar uma solução adaptada ao problema, suficientemente eficiente para operar em tempo real.

Em seu trabalho, Xiao et al. (2017) propõem uma solução engenhosa. Como discutido na Seção 3.2, técnicas tradicionais de renderização permitem obter, de forma eficiente, uma aproximação aceitável do mapa de profundidade do fluido. A geração desse mapa é particularmente rápida porque consiste em renderizar milhares de vezes o mesmo primitivo, um processo altamente otimizado em GPUs modernas. Se utilizarmos um raio grande o bastante para as esferas – maior que a menor distância possível entre a isosuperfície e sua partícula mais próxima –, esse mapa de profundidade pode servir como uma aproximação unidimensional de um SDF. Tal recurso permite eliminar grande parte do espaço vazio entre a câmera e a superfície, uma vez que essa distância serve como um ponto de partida para o *raymarching* tradicional. A otimização proposta pelos autores é ilustrada na Figura 3.13, que também evidencia o ganho de flexibilidade na escolha do tamanho do passo fixo durante a marcha. Ainda assim, é desejável minimizar ao máximo o número de passos remanescentes, o que pode ser alcançado empregando a interpolação linear discutida anteriormente.

A figura também revela, porém, que a solução dos autores não é perfeita. Raios com incidência rasante à superfície ainda percorrem longos trechos de espaço vazio, pois iniciar o avanço próximo ao fluido não garante proximidade na direção de *propagação* do raio. Essa distância adicional pode se tornar arbitrariamente grande, ocasionando uma perda de desempenho significativa. Em aplicações altamente interativas que permitem que o usuário se aproxime do líquido, posicionar a origem da câmera de tal modo que o mapa de profundidade obtido por *splatting* cubra completamente a tela resulta no pior caso possível, visto que a otimização é essencialmente descartada.

Wu et al. (2022) em uma publicação que continua diretamente o trabalho de Xiao et al. propuseram um conjunto de mecanismos adaptativos projetados especificamente para reduzir esse problema. O método de Wu et al. introduz estruturas auxiliares que permitem prever quando um raio está prestes a entrar (ou sair) de regiões relevantes de densidade.

O componente central dessa solução é o *binary density grid*, uma grade tridimensional binária que armazena, para cada célula, apenas a informação sobre a possibilidade de existir ou não uma isosuperfície em seu interior. Essa grade é construída por meio de uma estimativa conservadora da densidade máxima potencial de cada célula, garantindo que nenhuma região relevante seja descartada prematuramente. Durante o *raymarching*, essa estrutura permite que raios avancem com passos amplos através de grandes blocos de células marcadas como vazias, reduzindo drasticamente o custo de amostragens inúteis – inclusive nos casos em que o raio se desloca quase tangencialmente à superfície e, portanto, atravessaria longas extensões com pouca ou nenhuma contribuição de densidade.

Além disso, para mitigar situações em que o raio, mesmo após iniciar próximo ao fluido, ainda encontra regiões extensas de baixa densidade, os autores introduzem um segundo mecanismo: a classificação das partículas em *splashes* (baixa densidade) e *aggregations* (alta densidade). A partir dessa classificação, dois mapas de profundidade são gerados: D_{all}, contendo todas as partículas e utilizado para definir o ponto inicial do raio, e D_{agg}, contendo apenas partículas densas e empregado como aproximação para um ponto de saída antecipado. Quando um raio excede um número máximo de passos sem encontrar a isosuperfície, ele é imediatamente avançado até a profundidade indicada por D_{agg}, evitando que continue marchando por regiões quase vazias na direção escolhida. Esse mecanismo é particularmente eficaz para raios rasantes, que são os mais afetados pelo problema discutido, pois tendem a se deslocar por longos trechos paralelos ao fluido antes de encontrar uma região densa.

Por fim, os autores demonstram que a combinação entre a grade binária e o mapa de saída aproximado reduz o número total de passos de marcha em mais de 50% em cenas com muitos respingos e volumes esparsos. Isso resulta em desempenho substancialmente superior justamente nos casos problemáticos onde técnicas anteriores apresentam queda acentuada de performance. Dessa forma, o método resolve de maneira robusta o problema de raios que percorrem longas distâncias tangenciais, ao fornecer mecanismos explícitos para detectar, pular e abandonar regiões de baixa relevância geométrica ou densidade.

3.3.1 Reconstrução das Normais

Tal como no método em *screen space* de van der Laan et al. (2009), para gerar uma imagem que possui efeitos de iluminação corretos, precisamos computar os vetores normais da superfície do líquido. Xiao et al. (2017) mostram que é possível calcularmos normais diretamente a partir do passo da extração da profundidade da isosuperfície, seja pela Equação 3.1 ou simplesmente interpolando o gradiente da densidade normalizado. Porém, a imagem resultante possui um aspecto áspero indesejado, e portanto os autores propõem uma solução mais sofisticada que gera normais utilizando amostras de densidade adicionais.

Uma estimativa básica do normal $\mathbf{n}'_i$ na posição $\mathbf{x}_i$ é dado pela soma dos gradientes da densidade das partículas nas posições $\mathbf{p}_1, \ldots, \mathbf{p}_n$:

$$\mathbf{n}'_i = \sum_j^n \nabla W(\mathbf{x}_i - \mathbf{p}_i)$$

Apesar do normal $\mathbf{n}'_i$ apresentar resultados aceitáveis (melhores que a extração unidimensional em espaço de tela), com alguns cálculos mais intensos, utilizando *Principal Component Analysis* (Análise de Componentes Principais, PCA) é possível obter resultados mais suaves, que reduzem o efeito "globular" natural da discretização em partículas. Os passos para computar o normal aperfeiçoado são:

1. Cálculo da posição média $\bar{\mathbf{x}}_i$ das partículas vizinhas do ponto $\mathbf{x}_i$:

$$\bar{\mathbf{x}}_i = \frac{\sum_j \mathbf{p}_j W(\mathbf{x}_i - \mathbf{p}_j)}{\sum_j W(\mathbf{x}_i - \mathbf{p}_j)}$$

2. Cálculo da matriz de covariância C_i:

$$C_i = \frac{\sum_j W(\mathbf{x}_i - \mathbf{p}_j)(\mathbf{p}_j - \bar{\mathbf{x}}_i)(\mathbf{p}_j - \bar{\mathbf{x}}_i)^T}{\sum_j W(\mathbf{x}_i - \mathbf{p}_j)}$$

3. Extraia os três pares de autovetores e autovalores de C_i utilizando o método de iteração de Jacobi.

4. Seja $\mathbf{e}$ o autovetor de menor autovalor. Define-se o vetor $\mathbf{n}_i''$ como o vetor e ou $-e$, escolhendo aquele que tiver a orientação mais próxima de $\mathbf{n}_i'$.

5. Combinação dos normais $\mathbf{n}_i'$ e $\mathbf{n}_i''$:

$$w = min(1, e^{d \cdot (\mathbf{n}_i' \mathbf{n}_i'' - k)})$$

$$\mathbf{n}_i = (1 - w)\frac{\mathbf{n}_i'}{\|\mathbf{n}_i'\|} + w\frac{\mathbf{n}}{\|\mathbf{n}\|}_i''$$

   Onde d e k são parâmetros ajustáveis para renderização. Xiao et al. utilizaram $d = 10$ e $k = 0.998$.

6. O mapa de normais obtido pelo cálculo de $\mathbf{n}_i$ para cada pixel é então passado por um filtro bilateral, discutido na Seção 3.2.

É importante ressaltar que este cálculo é relativamente pesado, e sua utilização completa pode resultar numa redução perceptível da taxa de quadros. É possível utilizar a solução parcialmente, conforme a necessidade de fidelidade visual e a disponibilidade de recursos computacionais: usar $\mathbf{n}_i'$ apenas, fazer uma iteração de Jacobi menos precisa, não aplicar o filtro bilateral, etc.

3.4 RENDERIZAÇÃO

Nas três seções anteriores, vimos diferentes métodos empregados para a renderização de líquidos. Porém, é preciso ressaltar que, tecnicamente, nenhum deles explica o processo de renderização completo – falta a geração final da imagem, utilizando os dados obtidos nas computações anteriores. O motivo disso é que a geração da imagem final é bastante subjetiva. Renderizar água, tinta, mel ou lava envolvem decisões artísticas específicas e exigem o uso de estratégias diversas para replicar efeitos desejados.

Por exemplo, líquidos como água ou mel normalmente exigem o uso de transparência e refração. Com *raymarching*, isso pode ser feito aproveitando a natureza volumétrica da computação: o raio de luz pode ser *literalmente* refletido, refratado ou absorvido conforme as leis clássicas da ótica. Replicar estes efeitos com uma malha geométrica (*Marching Cubes*) não é tão fácil e exige aproximações criativas; o mesmo se aplica para estratégias em espaço de tela.

Outros exemplos incluem a necessidade de incluir bolhas e espuma em simulações de larga escala de água (oceanos, rios, piscinas agitadas); rugosidade em simulações de geleia;

mapeamento de texturas para a superfície rochosa de lava; emissão de luz para metais derretidos, entre muitos outros.

Dessa forma, não iremos nos aprofundar em detalhes visuais de líquidos específicos neste trabalho. As técnicas apresentadas são genéricas o bastante para permitir o estudo e aplicação caso a caso de técnicas estéticas mais avançadas. As três alternativas, por mais diferentes que sejam, resultam em mapas de profundidade e de normais – pontos de partida necessários para praticamente qualquer estratégia de geração de imagem escolhido, como o tradicional método de Blinn-Phong (Blinn, 1977).

3.5 CONCLUSÃO

Neste capítulo, examinamos três linhas fundamentais de pesquisa para a renderização em tempo real de líquidos baseados em partículas: a reconstrução geométrica por meio do *Marching Cubes*, a renderização baseada em filtros de imagens em *Screen Space*, e métodos modernos de *raymarching*.

O *Marching Cubes*, apesar de sua robustez e de sua importância histórica para a visualização volumétrica, mostrou-se pouco adequado para aplicações interativas devido ao custo excessivo de reconstruir malhas completas todo quadro. As técnicas em *Screen Space* oferecem desempenho significativamente superior ao explorar diretamente a rasterização tradicional e operações de filtragem na imagem final, mas possuem grandes limitações na qualidade da imagem gerada, principalmente devido à unidimensionalidade dos dados de profundidade disponíveis.

Por fim, investigamos técnicas de difusão de raios, em particular, *raymarching*. Esta técnica nos dá muito mais flexibilidade para a geração de imagens realistas. Porém, para ter performance aceitável em um contexto de tempo real, diversas otimizações são necessárias para reduzir ao máximo a quantidade de computações realizadas em cada pixel.

O cenário apresentado neste capítulo mostra que, embora não exista uma solução única que domine todos os aspectos de desempenho e qualidade, os métodos contemporâneos de *raymarching* representam o estado da arte para a renderização em tempo real de fluidos baseados em partículas. Eles acomodam com elegância tanto a estrutura irregular e dinâmica típica de simulações Lagrangianas quanto as exigências visuais de aplicações interativas modernas. No próximo capítulo, partiremos dessas bases conceituais para apresentar a metodologia proposta neste trabalho, que testou as diferentes técnicas em um estudo comparativo de performance e qualidade visual.

4 METODOLOGIA

Neste capítulo, descrevemos os materiais e procedimentos empregados para desenvolver e avaliar os métodos de renderização estudados anteriormente. Além de estabelecermos os objetivos da implementação, detalhamos as ferramentas computacionais utilizadas, discutimos aspectos técnicos relevantes de cada etapa do processo e, por fim, apresentamos a metodologia experimental adotada para mensurar desempenho e qualidade visual.

4.1 OBJETIVO E ABORDAGEM

No Capítulo 3, examinamos três famílias de abordagens para a renderização em tempo real de fluidos baseados em partículas. Com o intuito de transformar essas ideias em um estudo completo, este capítulo apresenta a implementação prática dessas três linhas de abordagem em uma simulação funcional de líquidos. Assim, este trabalho busca servir não apenas como referência teórica, mas também como guia técnico para o desenvolvimento de renderizadores em GPU.

Foram implementadas as seguintes estratégias:

1. ***Marching Cubes***, conforme o algoritmo original proposto por Lorensen e Cline (1987);
2. ***Splatting* com Pós-processamento**, utilizando a abordagem em espaço de tela de van der Laan et al. (2009);
3. ***Raymarching* em tempo real**, baseado no método moderno de Xiao et al. (2017).

Um objetivo adicional foi implementar todas essas abordagens inteiramente na GPU, evitando transferências custosas de dados entre CPU e GPU. Esse objetivo foi alcançado, embora algumas limitações e particularidades se tornem evidentes ao longo da discussão das implementações.

4.2 FERRAMENTAS COMPUTACIONAIS

4.2.1 API Gráfica

Como mencionado no Capítulo 1, um dos fatores motivadores deste trabalho foi o amadurecimento da API gráfica WebGPU. Apesar do nome, WebGPU não é restrita ao contexto de navegadores: trata-se de uma interface moderna, de propósito geral, aplicável a ambientes nativos e projetada para abstrair de forma clara conceitos antes fragmentados entre APIs como Vulkan, Direct3D e OpenGL.

Neste trabalho, todas as implementações foram realizadas em Rust utilizando a biblioteca `wgpu`. Em ambientes nativos, o `wgpu` utiliza o *backend* mais adequado disponível (Vulkan, Metal, Direct3D12 ou OpenGL). Na web, utiliza diretamente o *backend* WebGPU quando suportado, ou recorre ao WebGL como alternativa.

O programa desenvolvido executa de maneira estável em ambiente nativo – neste trabalho, utilizando o *backend Vulkan*. Também foi gerada uma versão para navegador, embora seu desempenho seja drasticamente limitado pelas restrições inerentes ao ambiente web. Por tal motivo, não foi possível executar a renderização em tempo real neste ambiente, e assim ele não foi incluído nos testes.

4.2.2 Linguagens de Programação

A lógica executada na CPU foi desenvolvida em Rust, devido à sua combinação de alta performance com segurança de memória garantida em tempo de compilação. A escolha também é favorecida pelo fato de `wgpu` ser implementado integralmente em Rust.

Os *shaders* foram escritos em WGSL, a linguagem oficial do ecossistema WebGPU. Sua sintaxe é inspirada em Rust, mas seu papel é similar ao do GLSL. Embora o `wgpu` aceite GLSL com tradução automática, optamos por WGSL como prova de conceito, e pela naturalidade com a qual ela se integra ao Rust.

4.2.3 Ambiente de Desenvolvimento

Todos os componentes foram escritos e testados em um sistema Ubuntu 22.04, utilizando uma GPU RTX A4500. O desenvolvimento foi realizado no ambiente *RustRover*, da JetBrains.

4.3 SIMULAÇÃO DE LÍQUIDOS

Para avaliar os métodos de renderização, é necessário um conjunto consistente de dados de entrada. Em simulações baseadas em partículas, isso se reduz essencialmente às posições das partículas no espaço. Embora fosse possível utilizar dados pré-gerados, a análise baseada em fluidos estáticos seria pouco realista e não capturaria fenômenos importantes envolvendo movimento, sobreposição e mistura.

Além disso, mesmo para fluidos estáticos, seria necessário implementar estruturas de aceleração para amostragem de densidade – estruturas que fazem parte de qualquer simulação Lagrangiana. Como a amostragem de densidade é usada diretamente pelos renderizadores, mas a performance da simulação não é objeto de estudo, optou-se por incluir uma simulação simples e eficiente.

Foi utilizado o método *Position Based Fluids* (Macklin e Müller, 2013), conhecido por sua robustez e por permitir taxas de atualização interativas.

4.3.1 Amostragem de Densidades

Detalhes gerais do algoritmo por trás da simulação fogem do objetivo deste trabalho. Porém, é importante detalhar o funcionamento do cálculo e amostragem de densidades, que impacta diretamente a renderização: o tradicional método de divisão espacial em grade, por vezes chamado de *cell-linked list*.

O espaço da simulação, um paralelepípedo de tamanho $\mathbf{W} = (\text{largura, altura, profundidade})$ é segmentado em uma grade de dimensões $x \times y \times z$. Dessa forma, o tamanho de uma única célula da grade é $\frac{\mathbf{W}}{(x,y,z)}$. A cada quadro, as n partículas da simulação são separadas nestas células, construindo um vetor

$$C = [((i_1, j_1, k_1), 1), ((i_2, j_2, k_2), 2), ..., ((i_n, j_n, k_n), n)]$$

de pares $((i_p, j_p, k_p), p)$ contendo a posição da célula em que a p-ésima partícula se encontra, e seu índice. Então, este vetor é ordenado[1], utilizando a posição da célula como elemento comparativo da ordenação. O resultado é um vetor no qual os índices de todas as partículas dentro de uma célula (i, j, k) estão presentes contiguamente em um subvetor C_{ijk}, que pode ser

[1] Na implementação deste trabalho, esta operação de ordenação é realizada na GPU, tal como todas as outras. Foi utilizado o algoritmo *radix sort*, disponível pela biblioteca `wgpu-sort`.

rapidamente encontrado ao armazenarmos em vetores auxiliares os índices de início e fim de C_{ijk} em C de cada célula.

Isso é relevante, porque se escolhermos dimensões da grade x, y, z de tal forma que todos os componentes de $\frac{\mathbf{W}}{(x,y,z)}$ sejam no mínimo tão grandes quanto o raio de interação de cada partícula, h, determinar a densidade em um ponto $\mathbf{v}$ se resume a somar a contribuição de cada partícula numa vizinhança de no máximo $3 \times 3 \times 3$ células ao redor da que $\mathbf{v}$ se encontra. Sabendo que numa simulação SPH típica o número de partículas nesta vizinhança gira em torno de 20-30, calcular a densidade em cada ponto do espaço torna-se uma operação computacionalmente barata.

Dessa forma, o cálculo de densidade no ponto $\mathbf{v}$ que usamos pelos métodos de renderização segue a fórmula:

$$\sum_{i=a-1}^{a+1} \sum_{j=b-1}^{b+1} \sum_{k=c-1}^{} \sum_{p \in C_{ijk}}^{1} W(\|\mathbf{v} - \mathbf{x}_p\|, h)$$

$$(a, b, c) = \left\lfloor \frac{\mathbf{v}}{\mathbf{W}}(x, y, z) \right\rfloor$$

Onde $W(d, h)$ representa a densidade do líquido a d unidades de distância de uma partícula com raio de interação h, e $\mathbf{x}_p$ é a posição da partícula p. As operações para calcular (a, b, c) são feitas por componente.

4.4 IMPLEMENTAÇÃO DOS MÉTODOS

4.4.1 *Marching Cubes*

O algoritmo *Marching Cubes* gera uma malha que representa a isosuperfície da densidade do fluido. Cada bloco da grade 3D é processado independentemente, o que torna o algoritmo altamente paralelizável na GPU.

Para acelerar a execução, densidades e gradientes nos vértices da grade são pré-computados, evitando recomputações redundantes entre cubos adjacentes. A principal dificuldade prática reside na composição da lista global de triângulos: múltiplas *threads* geram triângulos simultaneamente, e o preenchimento de um mesmo vetor global pode levar a condições de corrida.

Existem muitas formas de resolver isso. É possível computar um passo anterior que decide quantos triângulos cada *voxel* irá gerar, e realizar uma soma de prefixos usada para determinar previamente os índices dos triângulos no vetor final que cada um irá ocupar. Uma outra possibilidade é a de preencher o vetor final com triângulos inválidos, mas isso possui um impacto grande na performance. Também é comum optar por realizar a geração da malha totalmente no lado da CPU.

Neste trabalho, adotou-se uma solução intermediária: cada bloco de *threads* calcula localmente quantos triângulos irá produzir e utiliza um único `atomicAdd` para reservar seu espaço na lista global. Essa abordagem produz bom desempenho sem complexidade excessiva.

Como o número total de triângulos não pode ser previsto com precisão, adotou-se uma alocação conservadora: cinco triângulos por célula, valor máximo possível segundo a tabela de casos (Bourke, 1994).

4.4.2 *Splatting* com Pós-processamento

Neste, realizamos o *splatting* de partículas esféricas, e processamos os valores de profundidade no *depth buffer* gerado para produzir a imagem. Como vimos em sua apresentação, este método permite a escolha de diferentes algoritmos de suavização do mapa de profundidade. Aqui, foi utilizado o filtro bilateral.

O filtro bilateral não é separável, e portanto seu custo cresce quadraticamente com o tamanho do núcleo. Conforme observado por van der Laan et al. (2009), uma aproximação separável do filtro produz artefatos previsíveis, mas aceitáveis para aplicações em tempo real – especialmente quando composições visuais adicionais, como transparência, são aplicadas.

Um detalhe de implementação importante de ser pontuado: mapas de profundidade costumam ser armazenados em um espaço normalizado em APIs gráficas modernas, como o *Normalized Screen Coordinates* (NDC), que é o caso do WebGPU. Neste espaço, distâncias não são lineares e não representam distâncias euclideanas reais. Aplicar um filtro de esmaecimento em valores de profundidade normalizados produz uma imagem com distorções evidentes e visualmente desagradáveis. Portanto, após a geração do mapa de profundidade, fizemos um passo que converte esta distância não linear normalizada para a distância até a câmera.

4.4.3 *Raymarching* em Tempo Real

A técnica de *raymarching* utilizada segue o método de Xiao et al. (2017), sendo a mais flexível, porém também a mais custosa. Ao contrário de *Marching Cubes*, que amostra densidades apenas nos vértices da grade, o *raymarching* requer diversas amostras por *pixel*. Assim, sua viabilidade depende de uma implementação extremamente eficiente da amostragem de densidade.

Para acelerar o início de cada raio, utilizamos novamente o *splatting* para gerar um mapa inicial de distâncias, reduzindo o percurso necessário até a superfície do líquido. Como no método anterior, as profundidades em NDC são convertidas para distâncias reais.

A estimativa de normais foi feita utilizando apenas o gradiente do campo de densidade do líquido, normalizado. O gradiente é calculado da mesma forma que a densidade, porém realizando um somatório de $\nabla W(\|\mathbf{v} - \mathbf{x}_p\|, h)$. Não foi necessário usar uma estratégia mais sofisticada do que essa.

4.4.4 Grupo de Controle

Além dos três métodos vistos acima, foi implementado um método de controle, que apenas faz *splatting* de círculos sem qualquer pós processamento. Para facilitar a visualização, os círculos foram coloridos conforme sua posição no espaço. Utilizar esta estratégia como grupo de controle nos ajuda a ter uma referência do custo de renderização mínimo possível na máquina utilizada.

4.5 METODOLOGIA DOS EXPERIMENTOS

Os métodos foram testados e comparados através de critérios de performance e de fidelidade visual. Para testar a performance, medimos o tempo de geração de quadros em cenários idênticos, usando os registros de *timestamp* da placa de vídeo. A análise de fidelidade visual é um pouco mais subjetiva, porém alguns critérios utilizados incluem: (1) dificuldade de discriminar partículas individuais, (2) suavidade de superfícies planas, (3) alinhamento de normais conforme a superfície do líquido, (4) ausência de artefatos visuais, e (5) representação fiel de profundidades.

Foram realizados testes com 10k, 20k, 40k, 80k, 160k e 320k partículas. Para cada número de partículas, se realizou um teste de geração de imagem estática, e um teste com líquido em movimento. Como a simulação é totalmente determinística, todos os métodos de renderização foram aplicados na exata mesma cena. A cena é uma modificação da clássica "*double dam break*" que envolve a colisão de dois blocos de líquido em um tanque: para manter a simulação em movimento, é aplicada nas partículas uma força de rotação constante, que gera um redemoinho de líquido batendo nas paredes.

O esquema de iluminação e cores utilizado é um simples Blinn-Phong com uma luz estática, idêntico para todos os métodos.

4.6 CONCLUSÃO DO CAPÍTULO

Neste capítulo, apresentamos os materiais, ferramentas e procedimentos necessários para concretizar os métodos estudados ao longo do trabalho. Definimos os objetivos da implementação, justificamos as escolhas tecnológicas, detalhamos a construção da simulação de suporte e discutimos individualmente as principais decisões envolvidas na adaptação de cada método de renderização ao ambiente WebGPU.

Os elementos estabelecidos aqui fornecem a base técnica sobre a qual os resultados do próximo capítulo se apoiam. Agora que os métodos foram implementados e a metodologia experimental definida, estamos prontos para analisar comparativamente o desempenho e a qualidade visual de cada abordagem.

No capítulo seguinte, apresentamos e discutimos os resultados obtidos.

5 RESULTADOS

Este capítulo apresenta os resultados obtidos a partir dos testes das implementações dos métodos de renderização de líquidos baseados em partículas estudados neste trabalho. No Capítulo 4, vimos os critérios qualitativos e quantitativos usados para realizar uma análise crítica, que apresentaremos aqui.

5.1 *MARCHING CUBES*

O método de *Marching Cubes* (`cubes`) mostrou resultados positivamente surpreendentes. Implementações iniciais haviam gerado imagens com uma aparência de "geleia", com partículas facilmente discrimináveis e a grade visível. Porém, ao aumentar o valor da densidade da isosuperfície, rapidamente a qualidade se tornou muito melhor.

A técnica é moderadamente rápida, e apresentou performance de tempo real. Superfícies planas são suaves, mas o arredondamento das bordas depende de uma resolução da grade alta. É difícil discretizar partículas individuais, que formam "gotas" com aspecto líquido remetente a tensão superficial.

Ao se aproximar do líquido, é revelada a natureza poligonal da malha, que possui resolução limitada. Pequenos volumes de fluido atravessam a grade rapidamente, o que gera um artefato visual temporal, com "gotas" que se deformam, e somem e reaparecem.

Um ponto que deve ser mencionado, porém, é que a depender da implementação deste algoritmo, o uso de memória de vídeo pode ser muito alto. Isso acontece porque alocamos o tamanho de memória de 5 triângulos por cada *voxel* da grade. Como aqui usamos uma grade de tamanho $256 \times 128 \times 128$ e 32 bytes por triângulo, foram necessários 134 MB.

A Figura 5.1 mostra imagens obtidas após a renderização com este método. A Tabela 5.1 mostra os tempos médios da renderização dos quadros em um cenário com líquido estático, e a Tabela 5.2 em um cenário com líquido em movimento. Por fim, a Figura 5.2 agrega os valores das tabelas em um gráfico.

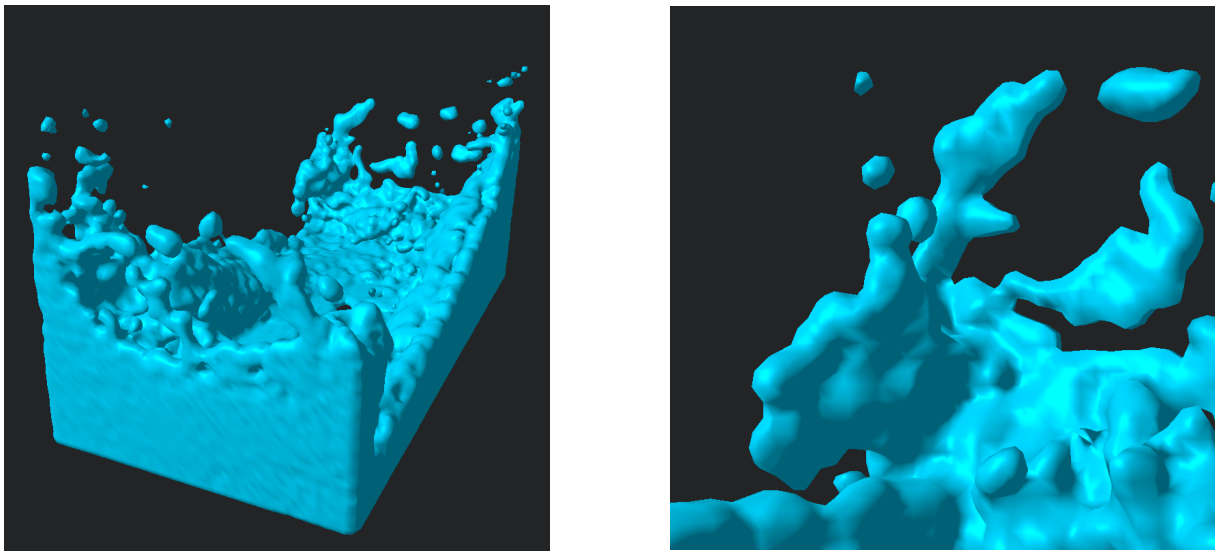


Figura 5.1: Renderização de 20k partículas com o método `cubes`. À esquerda: cena completa. À direita: mesma cena, vista de perto.

n	Média (ms)	Desvio padrão (ms)
10k	4.150	0.002
20k	4.439	0.002
40k	4.414	0.003
80k	4.453	0.002
160k	4.292	0.007
320k	4.386	0.009

Tabela 5.1: Tempos médios da renderização de quadros do método `cubes`, líquido estático

n	Média (ms)	Desvio padrão (ms)
10k	4.020	0.062
20k	4.303	0.091
40k	4.355	0.121
80k	4.585	0.176
160k	4.633	0.260
320k	4.871	0.346

Tabela 5.2: Tempos médios da renderização de quadros do método `cubes`, líquido em movimento

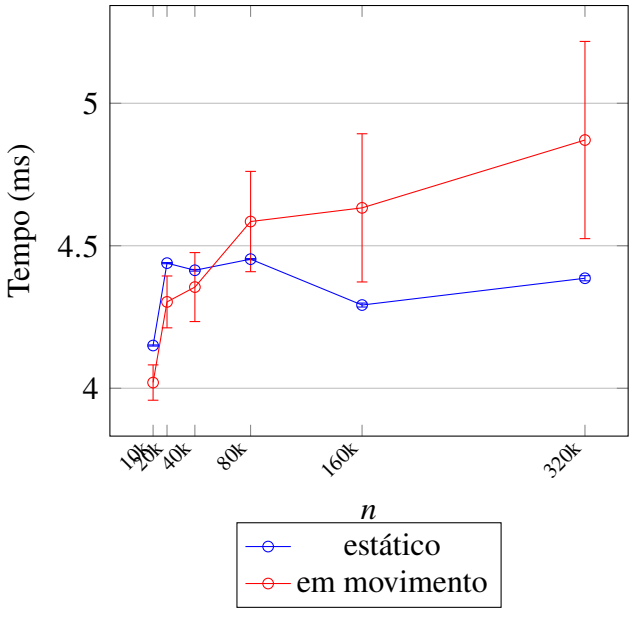


Figura 5.2: Gráfico mostrando os tempos médios da renderização de quadros do método `cubes`

5.2 *SPLATTING*

A técnica de *splatting* com pós processamento (`splatting`) demonstrou qualidade aceitável. Sua performance é levemente pior que `cubes`, e como previsto, possui muitos artefatos devido à não-separabilidade do filtro bilateral. Dito isso, é notável que esse problema não é tão perceptível de um ponto de vista perpendicular ao líquido, sendo visível principalmente nas suas extremidades.

A técnica funciona melhor quando o líquido não passa na frente de si mesmo, que pode ser o caso de usos de uma simulação abaixo do nível do espectador, como por exemplo uma piscina. Nesses casos, as normais são suaves e superfícies planas permanecem planas.

Ao olhar de perto, o efeito da suavização da superfície quebra: não é possível "borrar" tanto a imagem a ponto de desaparecem as fronteiras entre as partículas sem usar um filtro bilateral com alcance muito grande, computacionalmente inviável.

A Figura 5.3 mostra imagens obtidas após a renderização com este método. A Tabela 5.3 mostra os tempos médios da renderização dos quadros em um cenário com líquido estático, e a Tabela 5.4 em um cenário com líquido em movimento. Por fim, a Figura 5.4 agrega os valores das tabelas em um gráfico.

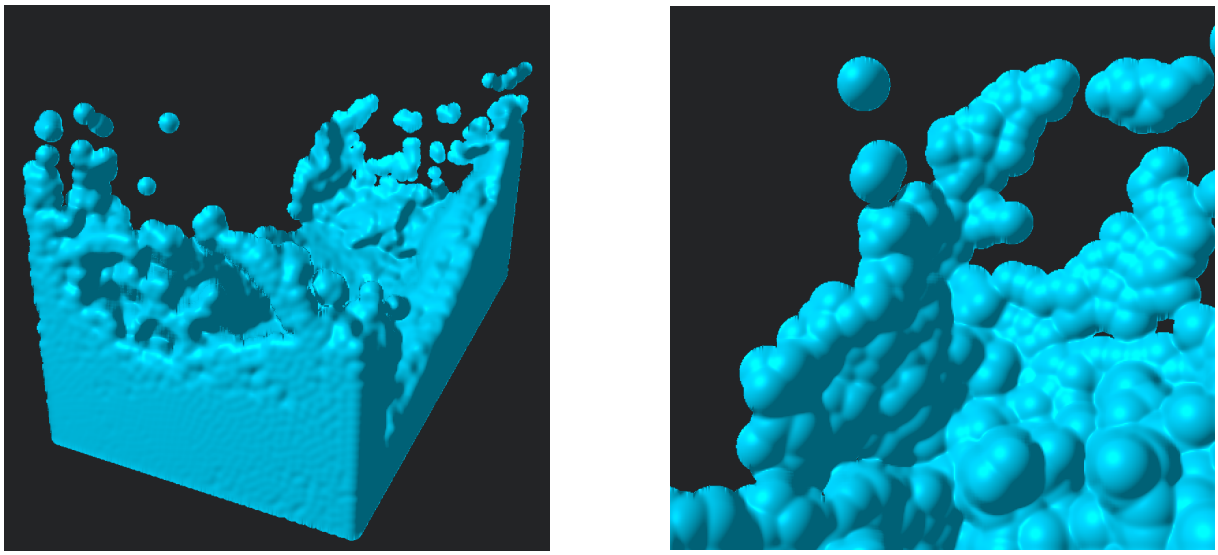


Figura 5.3: Renderização de 20k partículas com o método `splatting`. À esquerda: cena completa. À direita: mesma cena, vista de perto.

n	Média (ms)	Desvio padrão (ms)
10k	0.477	0.008
20k	0.624	0.006
40k	0.602	0.003
80k	0.671	0.005
160k	0.701	0.009
320k	0.869	0.014

Tabela 5.3: Tempos médios da renderização de quadros do método `splatting`, líquido estático

n	Média (ms)	Desvio padrão (ms)
10k	0.444	0.025
20k	0.589	0.022
40k	0.583	0.020
80k	0.676	0.019
160k	0.725	0.024
320k	0.896	0.030

Tabela 5.4: Tempos médios da renderização de quadros do método `splatting`, líquido em movimento

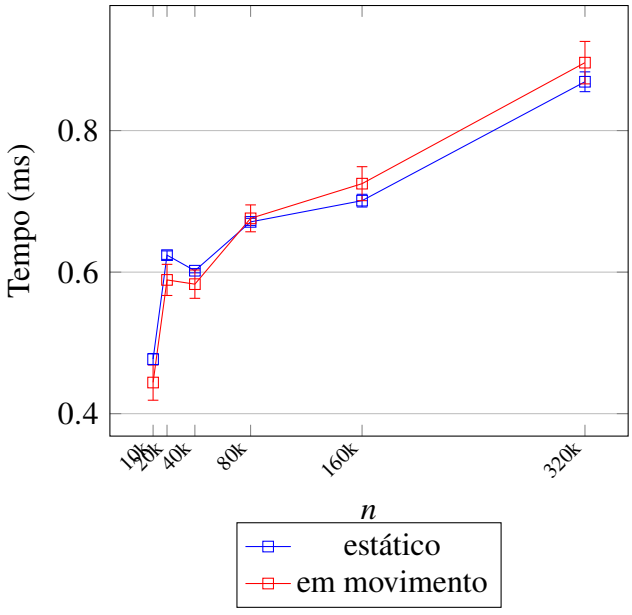


Figura 5.4: Gráfico mostrando os tempos médios da renderização de quadros do método `splatting`

5.3 *RAYMARCHING*

O método `raymarching` pode ser descrito visualmente como "*marching cubes* com resolução infinita". O grande triunfo do *raymarching* no geral é sua capacidade imbatível de renderizar superfícies curvas sem o uso de uma quantidade cada vez maior de polígonos. Não importa o quão próximo se esteja do objeto, sua aparência arredondada não passa a ser discretizada. As normais são perfeitamente suaves, as partículas não são discretizáveis a não ser que estejam dispersas no ar, e mesmo assim se misturam com o restante do fluido de forma realista. De todos os métodos, é o que melhor transmite o aspecto visual de um líquido real, funcionando bem de qualquer ponto de vista.

Esta grande fidelidade visual tem um custo alto, quando comparamos com os outros métodos. Por realizar múltiplas amostras de densidade por pixel, a performance deste método é, sem dúvida, a pior dentre todos. Com 320 mil partículas, o tempo de renderização gira em torno de 15 ms, perigosamente próximo ao valor de 16,66 ms: o limite máximo para uma renderização suave a 60 quadros por segundo.

A performance *raymarching* depende muito do tamanho da simulação. Isso acontece porque em mundos maiores, raios com incidência rasante ao líquido precisam viajar por muito mais tempo até atingir a superfície do outro lado, ou sair dos intervalos da simulação.

A Figura 5.5 mostra imagens obtidas após a renderização com este método. A Tabela 5.5 mostra os tempos médios da renderização dos quadros em um cenário com líquido estático, e a Tabela 5.6 em um cenário com líquido em movimento. Por fim, a Figura 5.6 agrega os valores das tabelas em um gráfico.

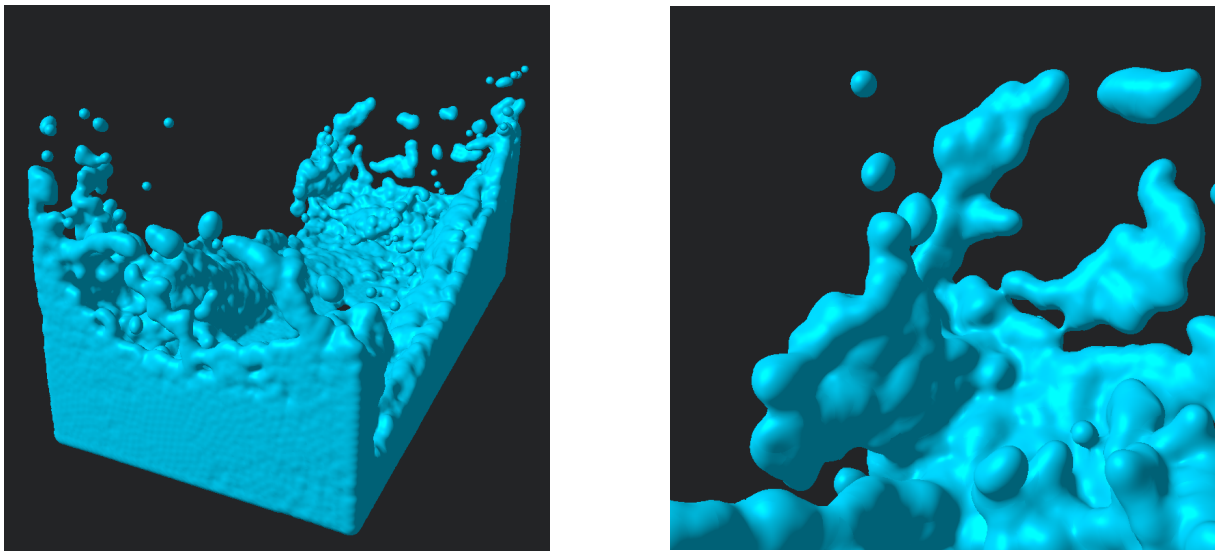


Figura 5.5: Renderização de 20k partículas com o método `raymarching`. À esquerda: cena completa. À direita: mesma cena, vista de perto.

n	Média (ms)	Desvio padrão (ms)
10k	8.470	0.489
20k	8.954	0.280
40k	9.686	0.288
80k	10.999	0.227
160k	12.480	0.455
320k	13.607	0.544

Tabela 5.5: Tempos médios da renderização de quadros do método `raymarching`, líquido estático

n	Média (ms)	Desvio padrão (ms)
10k	7.529	1.097
20k	8.296	1.004
40k	9.450	1.218
80k	10.794	1.468
160k	12.617	2.091
320k	15.330	2.873

Tabela 5.6: Tempos médios da renderização de quadros do método `raymarching`, líquido em movimento

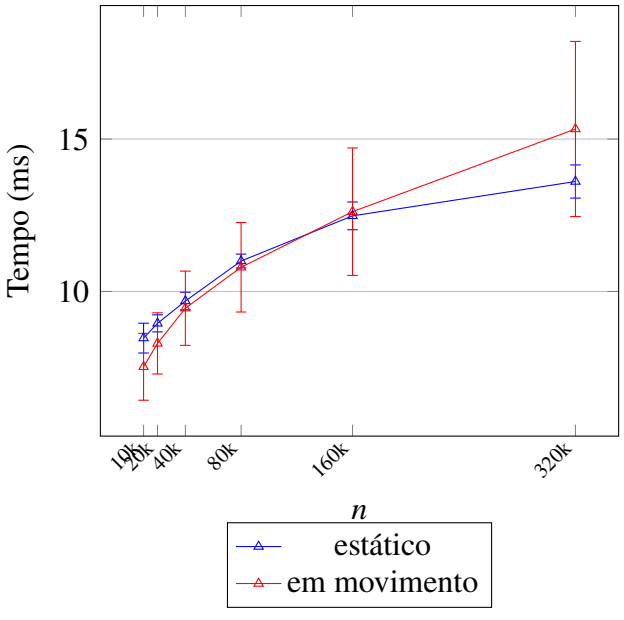


Figura 5.6: Gráfico mostrando os tempos médios da renderização de quadros do método `raymarching`

5.4 CONTROLE

A renderização trivial usada como controle não será avaliada conforme fidelidade visual, e serve o propósito de indicar o tempo de renderização mínimo da simulação, e quanto ele escala conforme aumenta o número de partículas. Mesmo com 320 mil partículas ou mais, é possível renderizar um quadro em menos de 1 milissegundo.

A Figura 5.7 mostra imagens obtidas após a renderização com este método. A Tabela 5.7 mostra os tempos médios da renderização dos quadros em um cenário com líquido estático, e a Tabela 5.8 em um cenário com líquido em movimento. Por fim, a Figura 5.8 agrega os valores das tabelas em um gráfico.

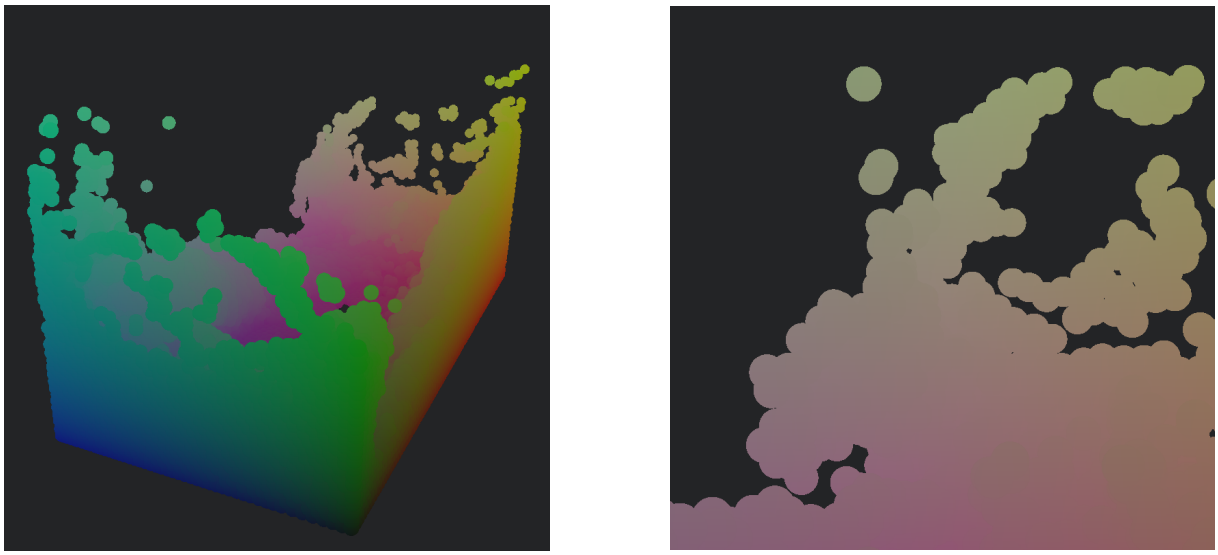


Figura 5.7: Renderização de 20k partículas com o método `controle`. À esquerda: cena completa. À direita: mesma cena, vista de perto.

n	Média (ms)	Desvio padrão (ms)
10k	0.038	0.001
20k	0.042	0.003
40k	0.055	0.003
80k	0.083	0.004
160k	0.146	0.004
320k	0.287	0.003

Tabela 5.7: Tempos médios da renderização de quadros do método `controle`, líquido estático

n	Média (ms)	Desvio padrão (ms)
10k	0.029	0.004
20k	0.045	0.005
40k	0.061	0.004
80k	0.089	0.003
160k	0.149	0.002
320k	0.295	0.004

Tabela 5.8: Tempos médios da renderização de quadros do método `controle`, líquido em movimento

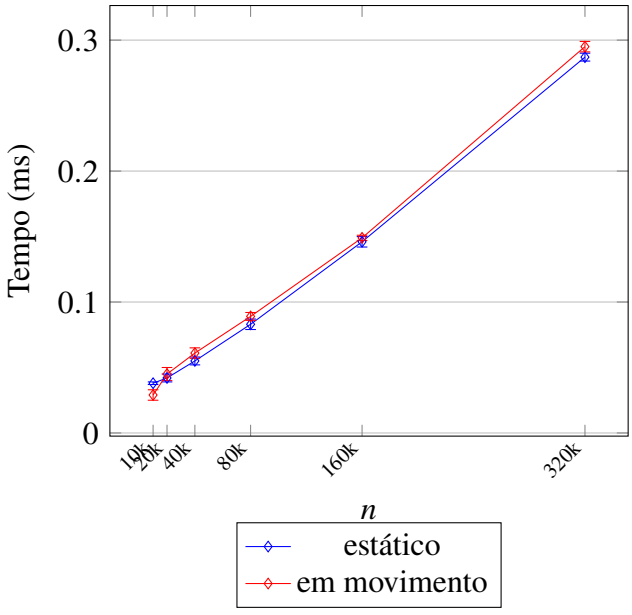


Figura 5.8: Gráfico mostrando os tempos médios da renderização de quadros do método `controle`

5.5 RESULTADOS GERAIS

Os gráficos nas Figura 5.9 e Figura 5.10 agregam os tempos médios de renderização obtidos por todos os métodos, no cenário com líquido estático e com líquido em movimento, respectivamente.

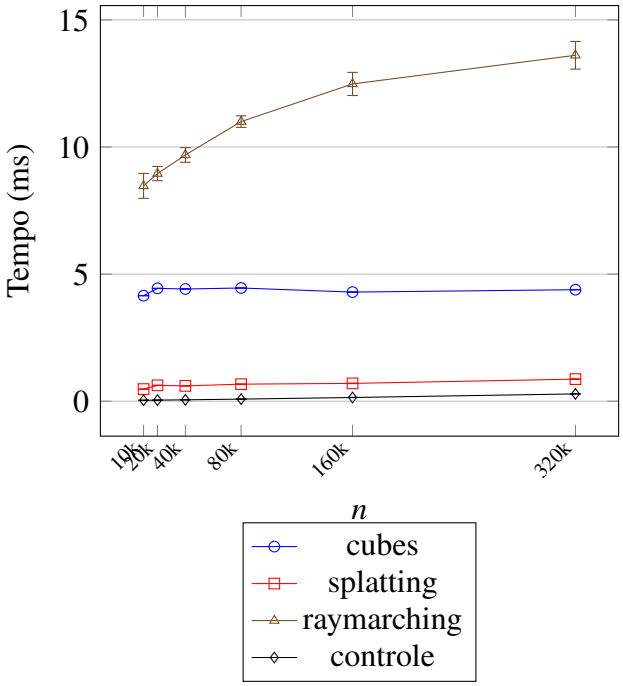


Figura 5.9: Tempos médios da renderização de quadros de todos os métodos, líquido estático

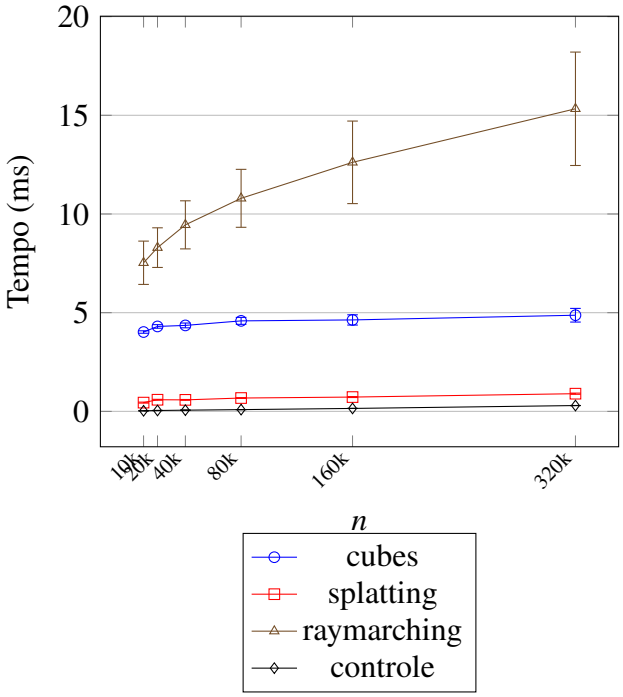


Figura 5.10: Tempos médios da renderização de quadros de todos os métodos, líquido em movimento

5.6 CONCLUSÃO

Os resultados apresentados neste capítulo evidenciam que cada método de renderização estudado oferece um conjunto distinto de benefícios e limitações, tanto em termos de qualidade visual quanto de desempenho. Em linhas gerais, observamos uma relação clara entre fidelidade visual e custo computacional, o que permite situar cada técnica em um espectro próprio de aplicabilidade.

O método `cubes`, baseado em *Marching Cubes*, demonstrou-se surpreendentemente eficaz dentro das restrições impostas pela discretização espacial. Ele produz superfícies coerentes, visualmente agradáveis e com desempenho consistente em tempo real. No entanto, sua dependência da resolução da grade limita a suavidade de detalhes mais finos, além de introduzir artefatos temporais quando pequenos volumes de fluido atravessam rapidamente as células da malha. Seu custo de memória de vídeo também pode ser relativamente alto, a depender da implementação usada para o armazenamento dos vértices.

A técnica de `splatting` apresentou desempenho competitivo e qualidade visual aceitável, sobretudo em cenários onde o fluido não se sobrepõe a si mesmo ao longo da linha de visão. Ainda que o filtro bilateral introduza artefatos decorrentes de sua natureza não separável, esses defeitos são pouco perceptíveis em condições favoráveis de visualização. Contudo, a técnica mostra-se limitada na representação de detalhes microscópicos, evidenciando a dificuldade de eliminar visualmente a granularidade inerente à representação por partículas.

Por sua vez, o método de `raymarching` alcançou os melhores resultados em termos de realismo. Sua capacidade de renderizar superfícies suaves, contínuas e visualmente ricas, independentemente da proximidade da câmera, destaca sua superioridade estética. Entretanto, essa fidelidade tem um custo elevado: o método apresentou tempos de renderização significativamente maiores, que podem dificultar seu uso em cenários interativos mais exigentes, ou mesmo inviabilizar se os recursos computacionais da GPU não forem bons o bastante.

No conjunto, os resultados mostram que nenhuma técnica é universalmente superior; cada uma ocupa um espaço próprio na prática da computação gráfica. Métodos geométricos como `cubes` equilibram bem qualidade e desempenho, `splatting` oferece rapidez com qualidade moderada, e `raymarching` entrega o melhor resultado visual ao custo de performance significativamente mais alta.

6 CONCLUSÃO

Este trabalho realizou uma investigação abrangente sobre técnicas de renderização em tempo real de líquidos baseados em partículas, abordando tanto aspectos teóricos quanto práticos da área. Através da implementação e avaliação comparativa de três métodos principais – *Marching Cubes*, *Splatting* com pós-processamento e *ray marching* – o estudo demonstrou claramente os trade-offs entre desempenho e qualidade visual que caracterizam essa área da computação gráfica.

Os resultados obtidos confirmam que não existe uma solução universalmente superior para o problema da renderização de fluidos. Cada método ocupa um espaço próprio no espectro de aplicações: o Marching Cubes oferece um equilíbrio razoável entre qualidade e desempenho, sendo adequado para aplicações que requerem malhas explícitas; o Splatting apresenta a melhor relação custo-benefício para cenários menos críticos visualmente; enquanto o Raymarching estabelece-se como estado da arte em termos de fidelidade visual, ainda que ao custo de maior demanda computacional.

A contribuição deste trabalho vai além da comparação teórica, oferecendo implementações concretas que demonstram a viabilidade dessas técnicas usando tecnologias modernas como WebGPU e Rust. Isto representa um avanço significativo em direção ao objetivo de tornar a visualização científica avançada acessível através de navegadores web, eliminando barreiras de instalação e configuração que historicamente limitaram o acesso a essas ferramentas.

Algumas limitações deste trabalho incluem os resultados inconclusivos de testes com simulação e renderização na web – apesar de termos utilizado a API WebGPU, sua aplicação foi em ambiente nativo, e não num navegador. Expandir os testes para incluírem execução na web, além de diferentes configurações de hardware e cenários mais diversificados seria valioso. Estudos futuros também podem investigar a construção de outros fenômenos visuais, como espuma, bolhas e interações com superfícies complexas.

Em síntese, este trabalho não apenas fornece uma análise crítica atualizada das técnicas de renderização de fluidos, mas também estabelece uma base prática para o desenvolvimento de aplicações interativas que exigem visualização realista de fenômenos líquidos. Através de nossa abordagem teórica e prática, o estudo serve como um recurso valioso para pesquisadores, desenvolvedores e educadores interessados em computação gráfica e visualização científica, contribuindo para a democratização do acesso a tecnologias avançadas de renderização.

REFERÊNCIAS

Akenine-Möller, T., Haines, E. e Hoffman, N. (2018). *Real-Time Rendering*. A K Peters/CRC Press, 4 edition.

Banterle, F., Corsini, M., Cignoni, P. e Scopigno, R. (2012). A low-memory, straightforward and fast bilateral filter through subsampling in spatial domain. *Computer Graphics Forum*, 31(1):19–32.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198.

Bourke, P. (1994). Polygonising a scalar field (marching cubes). `https://paulbourke.net/geometry/polygonise/`. Acessado em setembro de 2025.

Bridson, R. (2015). *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press, Boca Raton, FL, 2 edition. eBook - PDF.

Glassner, A. S., editor (1989). *An Introduction to Ray Tracing*. The Morgan Kaufmann Series in Computer Graphics. Academic Press / Morgan Kaufmann.

Hadji-Kyriacou, A. e Arandjelović, O. (2021). Raymarching distance fields with cuda. *Electronics*, 10(22).

Hart, J. C., Sandin, D. J. e Kauffman, L. H. (1989). Ray tracing deterministic 3-d fractals. Em *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, página 289–296, New York, NY, USA. Association for Computing Machinery.

Kaufman, A. (2003). Volume visualization: Principles and advances. *Stony Brook University*, 28.

Kitware, I. (2025). Paraview — open-source, multi-platform data analysis and visualization application. `https://www.paraview.org/`. Accessed: 2025-11-30.

Levoy, M. e Whitted, T. (1985). The use of points as a display primitive. Relatório Técnico 85-022, Computer Science Department, University of North Carolina at Chapel Hill, Chapel Hill, NC.

Liu, G. e Liu, M. (2003). *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. World Scientific Publishing Co. Pte. Ltd.

Lorensen, W. E. e Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169.

Macklin, M. e Müller, M. (2013). Position based fluids. *ACM Trans. Graph.*, 32(4).

Quilez, I. (2008). Rendering worlds with two triangles with raytracing on the gpu in 4096 bytes. Em *NVScene 2008*, páginas 1–57. Presentation slides.

Rauter, M., Zimmermann, L. e Zeilinger, M. (2024). Accelerating transfer function update for distance map based volume rendering. Em *2024 IEEE Visualization and Visual Analytics (VIS)*, páginas 171–175.

Sakamoto, N., Nonaka, J., Koyamada, K. e Tanaka, S. (2007). Particle-based volume rendering. Em *2007 6th International Asia-Pacific Symposium on Visualization*, páginas 129–132.

Strakos, P., Jaros, M., Brzobohaty, T., Faltynkova, M., Meca, O. e Riha, L. (2025). Scalable volume rendering of billion-cell cfd simulations using vfx pipelines. Em *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Posters*, SIGGRAPH Posters '25, New York, NY, USA. Association for Computing Machinery.

Tomasi, C. e Manduchi, R. (1998). Bilateral filtering for gray and color images. Em *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, páginas 839–846.

van der Laan, W. J., Green, S. e Sainz, M. (2009). Screen space fluid rendering with curvature flow. Em *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, página 91–98, New York, NY, USA. Association for Computing Machinery.

Wu, T., Zhou, Z., Wang, A., Gong, Y. e Zhang, Y. (2022). A Real-Time Adaptive Ray Marching Method for Particle-Based Fluid Surface Reconstruction . Em Ghosh, A. e Wei, L.-Y., editores, *Eurographics Symposium on Rendering*. The Eurographics Association.

Xiao, X., Zhang, S. e Yang, X. (2017). Real-time high-quality surface rendering for large scale particle-based fluids. Em *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, New York, NY, USA. Association for Computing Machinery.

Zhang, Q., Eagleson, R. e Peters, T. M. (2011). Volume visualization: A technical overview with a focus on medical applications. *Journal of Digital Imaging*, 24(4):640–664.