Departamento de Mecânica Computacional
Faculdade de Engenharia Mecânica
Universidade Estadual de Campinas - UNICAMP

IA369E - Tópicos em Engenharia de Computação VI

Simulação de Partículas usando CUDA e Ferramentas de Visualização OpenGL

Aluno: Edgar Andrés Patiño Nariño

10 de dezembro de 2013

Resumo

Este trabalho é uma aplicação do método de partículas para problemas de escorregamento de fluido, aplicando ferramentas de visualização 3D e interação interativa em tempo real, para o melhoramento do estudo das simulações de partículas. Para este fim nós usamos o código *Particle Simulation using CUDA*, feito pelo Prof. Simon Green, onde em este trabalho propõe-se sua modificação em uma nova simulação de partículas fazendo uma formulação própria do método *Dissipative Particle Dynamics* (*DPD*) e a alteração de sua visualização no OpenGL, acrescentando a implementação de novos exemplos de estudo como escorregamento continuo de partículas, queda de coluna de fluido e fluido-estrutura, além da implementação da visualização da grandeza da velocidade em cada partícula. O entorno de visualização é desenvolto usando a biblioteca de OpenGL GLUT e a ferramenta de visualizador por fragmentos de *Point Sprites* desenvolvida para a texturização em um ponto onde se usa a textura 2D, nosso caso em questão uma esfera no espaço para a representação de cada partícula. Este trabalho desenvolvido para IA369E - Tópicos em Engenharia de Computação VI, que foi orientado à estudo de sistema interativo de visualização de dados volumétricos, dirigido pela Professora Wu, Shin - Ting. Onde a conclusão más destacada foi conseguir a estabilização do método *DPD* usando a visualização da velocidade de cada partícula, e a alteração de alguns parâmetros da simulação.

1 Introdução

Durante os últimos 15 anos, vários novos métodos de simulação sem malha, baseados em partículas, foram objeto de grande atenção [1–3], pois podem fornecer soluções onde os métodos tradicionais com malha são fracos [4], como por exemplo em situações de fluxo de fluidos complexos [5], altas deformações de material [6, 7] e problemas multi-escala [8, 9]. Entre eles estão principalmente os recentes desenvolvimentos em *Smoothed Particle Hydridynamics*

(*SPH*) [10], os *Lattice gas methods* (*LGM*) [11], *Dissipative Particle Dynamics* (*DPD*) [12–14], *Moving Least Square (MLS)* [15], *Moving Least Square Reproducing Kernel Interpolant (MLSRK)* [1, 16] e *Discrete Element Method* (*DEM*).

Método DPD

Um método mais recente para simulação por partículas, o método *DPD*, proposto por Hoogerbrugge e Koelman em 1992 [9], embora menos eficiente do ponto-de-vista computacional devido a sua natureza aleatória. Apresenta uma ótima conservação de massa e de momentum, e permite simulações de sistemas mesoscópicos que envolvem interações fluido-estrutura, soluções coloidais, e problemas multi-escala, como demonstrado por outros autores [17–19]. Em função dessas características, o *DPD* tem sido aplicado com sucesso na modelagem numérica de sistemas microfluídicos de elevada complexidade como, por exemplo, a micro-bomba coloidal publicada por De Palma em 2006 [17], a simulação de fluxo multi-fase em redes de microcanais publicada por Liu em 2007 [18], e um microssistema de imuno-ensaios baseado em micro-esferas e uma câmara microfluídica (Fig. 1) publicado por Steiner em 2009 [5]. Esse sistema de imuno-ensaios foi modelado através de um conjunto de 500.000 partículas de flúido, e um experimento de 12 segundos de duração levou uma semana para ser simulado em um *cluster* de 32 processadores NEC Xeon EM64T com sistema operacional Linux, que nesse tempo executaram 650.000 interações. Esse elevado custo computacional pode ser um impedimento à aplicação do método *DPD*, apesar dos seus muitos aspectos positivos.

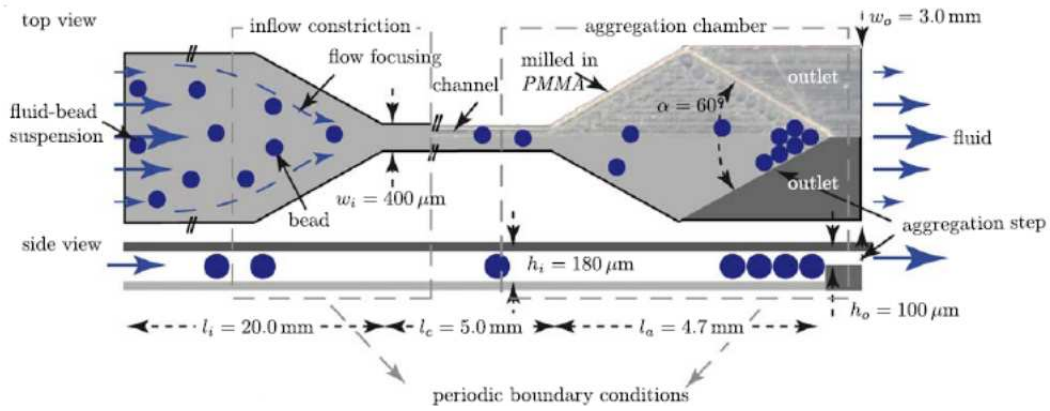


Figura 1: Microssistema de imuno-ensaios baseado em micro-esferas e uma câmara microfluídica [5]. As microesferas são recobertas com uma camada de proteínas de captura e marcadas com um fluoróforo. São arrastadas por um flúido até a câmara de leitura, onde se espera que formem um agregado regular que facilite a leitura por fluorescência.

Integração GPU e simulação de Partículas

Os métodos baseadas em partículas geral têm várias vantagens sobre os métodos baseados em malha [20]:

1. Eles só realizar cálculos quando é necessário.

2. Eles exigem menos armazenamento e largura de banda, já que as propriedades do modelo são armazenados apenas nas posições das partículas, em vez de em cada ponto do espaço.

3. Eles não são necessariamente restritos a um caixa finita.

4. A conservação da massa é simples (uma vez que cada partícula representa uma quantidade fixa de massa).

A principal desvantagem dos métodos baseadas em partículas é que requerem um grande número de partículas para obter resultados realistas, fazendo o processamento computacional muito pesado.
Felizmente, é relativamente fácil de paralelizar sistemas de partículas e os enormes capacidades de computação paralela de *GPU*'s modernas agora torna possível simular sistemas de grandes dimensões em taxas de tempo relativamente baixas. Apesar dos métodos de partículas em geral presentarem elevado custo computacional [10, 21], novas perspectivas se apresentaram a partir de 2007, quando a *NVIDIA* lançou o primeiro processador gráfico (*GPU*) de uso-geral (*GPGPU*), para ser usado como co-processador matemático em sistemas de cálculo científico, acelerando dezenas de vezes a solução de problemas complexos e diminuindo em muito o custo computacional, utilizando técnicas de processamento massivamente paralelo [22, 23]. Embora, o desenvolvimento de um simulador do fluxo de líquidos por métodos de partículas e seja executado em *GPGPU* justifica-se pelos seguintes motivos: As interações fluido-estrutura são fenômenos muito complexos, difíceis de se representar fielmente por modelos de ordem reduzida devido a não-linearidades dos fenômenos e às complexas geometrias dos dispositivos; Os métodos tradicionais de simulação numérica apresentam restrições de desempenho em simulações dinâmicas. Um simulador capaz de representar com boa fidelidade o fluxo de líquidos pode viabilizar a criação de modelos de ordem reduzida mais fiéis que os atuais [24–26].

Visualização de métodos de Partículas usando OpenGL e GLUT

Outro problema nos métodos de partículas como é *DEM* e *DPD* é que são projetados para ser métodos adimensionais [19, 27] o que dificulta achar de formas simples os valores apropriados dos parâmetros dos modelos, complicando a estabilização e estudo. No caso especial de *DPD* além, tem um termo oscilatório (randômico) em sua formulação, que aumenta a dificuldade, e que faz que a visualização e modificação de parâmetros em tempo real uma questão critica e muito relevante para o estudo, otimização e validação das simulações. Assim, o emprego da máquina de estados como OpenGL e a biblioteca GLUT para visualização, transforma-se em uma importante ferramenta.
OpenGL e a ferramenta de visualização é "um programa de interface para hardware gráfico" [28, 29]. Na verdade, OpenGL uma biblioteca de rotinas gráficas e de modelagem, bi (2D) e tridimensional (3D), extremamente portável e rápida [30]. Entretanto, a maior vantagem na sua utilização é a rapidez, uma vez que usa algoritmos cuidadosamente desenvolvidos e otimizados pela Silicon Graphics, Inc., líder mundial em Computação Gráfica e Animação [30].

No entanto, GLUT (OpenGL Utility Toolkit) é uma biblioteca de funcionalidades para OpenGL cujo principal objetivo é a abstração do sistema operacional fazendo com que os aplicativos sejam multiplataforma. A biblioteca foi escrita por Mark Kilgard (autor de livros sobre computação gráfica) enquanto ele trabalhava para a empresa Silicon Graphics [28]. A biblioteca possui funcionalidades para criação e controle de janelas, e também tratamento de eventos de dispositivos de entrada (mouse e teclado). Também existem rotinas para o desenho de formas tridimensionais pré-definidas (como cubo, esfera, bule, etc).

Contribuições

Este trabalho tem como objetivo mostrar as vantagens de usar as ferramentas de visualização e iteração usando OpenGL e GLUT para o estudo de simuladores de partículas *DPD*, em simulações em tempo real. No entanto, este documento resume a ideia básica de uma formulação que propõe-se de *DPD*, sua implementação no código CUDA e sua visualização usando OpenGL e GLUT. Além, da implementação da ferramenta de visualização para processamento de fragmentos para desenho de partículas de *Point Sprite* para representar elas como esperas.
Para este efeito, este trabalho se consegue modificando o código feito pelo Prof. Simon Green, no que usa-se como método de simulação de partículas *DEM*, realizado em seu trabalho *Particle Simulation using CUDA* [20]. Portanto, usando esse código, mudamos ele para as novas formulações propostas com *DPD* , e os novos requisitos de visualização dos problemas a implementar. Esses consistem em visualizar a magnitude da velocidade das partículas usando una faixa de cores que delimite entre a velocidade máxima e mínima do problema, usando uma função que transforma os valores das grandeza das velocidades das partículas em cores *RGBA* (red green blue alpha) [28], como se pode olhar na Fig.14 do lado esquerdo. Onde a máxima velocidade fica de cor vermelho e a mínima velocidade de cor azul. Assim, modifica-se a iteração de OpenGL e CUDA, já que uma variável atualizada constantemente deve ser visualizada agora, o código anterior só visualizava as partículas com cores arbitrários definidos nos passos iniciais da simulação, como se mostra na Fig.14 no lado direito.
Com essas novas ferramentas e a visualização pronta, novos exemplos de estudo de interação fluidos-estrutura como são:

- Escorregamento continuo de partículas.
- Queda de coluna de partículas.
- Interação com obstaculo em queda de coluna de partículas.

Como efeito da novos exemplos e visualizações se consegue achar o estado de estabilidade do problema e se mostra um estado de estabilidade que só, posso-se conseguir com a nova implementação no modelo.

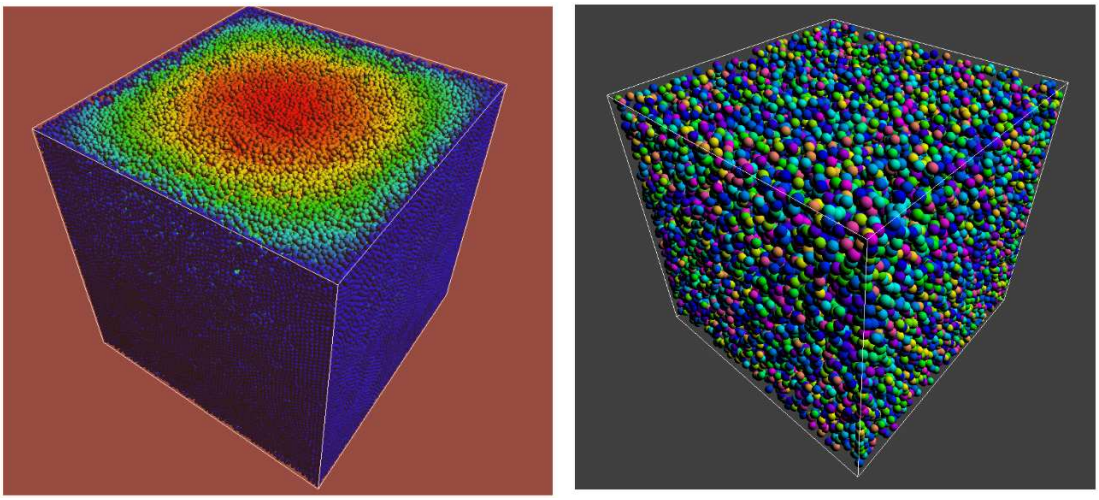


Figura 2: Visualização de partículas por Point Sprites. Na esquerda da figura, partículas coloreadas em relação com suas velocidades máximas e mínimas; Na direita visualização original, feito pelo Prof.Prof. Simon Green [20], onde os cores das partículas estão atribuídas de forma aleatório.

2 Generalidades programa Particle Simulation using CUDA

O comportamento da simulação de partículas feita em CUDA (*Particle Simulation using CUDA* [20]) esta dividido principalmente por três passos:

1. Integração temporal.
2. Interação Partícula-partículas (pesquisador de partículas).
3. Colisão de partículas.

Entanto que o renderizado das partículas é realizado usando OpenGL, fazendo uso do *Point Sprites* [28] e um *GLSL* (OpenGL Shading Language) [29] pixel shader que faz os pontos tornar-se esferas. O tema do uso do *Point Sprites* se aprofundara na próxima secção.

3 Integração temporal e interação CUDA-OpenGL

O passo de integração é o mais simples passo. Ele integra os atributos de partículas (posição e velocidade) para mover as partículas através do espaço. Usando a integração simplificada de Euler para obter a velocidade, sendo atualizada com base na força aplicada e da gravidade, e em seguida, a posição é atualizado com base na velocidade. Amortecimento e interações com o cubo delimitador também são aplicados nesta fase.
As posições e as velocidades das partículas são armazenadas em vetores de tipo **float4**. A posição é alocadas no OpenGL em um vetor de armazenamento de vértice (vertex buffer object,VBO), de modo que ela pode ser processados diretamente na GPU, no buffer usado pelo visualizador. Esto se consegue usando classe `ParticleSystem` e sua função `createVBO`, de esta maneira:

```
ParticleSystem::createVBO(uint size)
{
    GLuint vbo;
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    return vbo;
}
```

Assim, iniciar o vetor posição como `m_posVbo`, em este trabalho se adiciona uma variável que esta alocada também em um *VBO* e sera usada para alocar os cores das partículas em relação a suas velocidades `m_colorVBO`, dessa forma se muda uma variável constante no tempo como era o cor das partículas, que eram coloridas de maneira arbitraria. Estas variáveis são definidas como:

```
m_posVbo = createVBO(m_numParticles*4*sizeof(float));
registerGLBufferObject(m_posVbo, &m_cuda_posvbo_resource);
....
m_colorVBO = createVBO(m_numParticles*4*sizeof(float));
registerGLBufferObject(m_colorVBO, &m_cuda_colorvbo_resource);
```

Esta memória *VBO* é mapeada para uso de CUDA, com a função `registerGLBufferObject` para sua inicialização (como se mostra no código acima) a que direciona para `cudaGLMapBufferObject`, que armazenada para o uso da parte do CUDA do simulador, continuamente para sua atualização como:

```
{
    dPos = (float *) mapGLBufferObject(&m_cuda_posvbo_resource);
    dColor = (float *) mapGLBufferObject(&m_cuda_colorvbo_resource);
}
```

Assim, os vetores ficam em um buffer de dupla utilidade para que a atualização dos novos valores não afete as partículas ainda não processadas. Entanto, ao terminar a atualização dos vetores os dados podem ser usados de novo tanto na simulação como na visualização, em uma relação simular a mostrada na Fig.3, na que nas variáveis que ficam na GPU possam ser utilizadas tanto pela parte do visualizador como pelo simulador de uma maneira constante.

3.1 Renderizado das partículas

0 renderizado das partículas é realizado pela maquina de estados OpenGL, fazendo uso do *Point Sprites* [28].

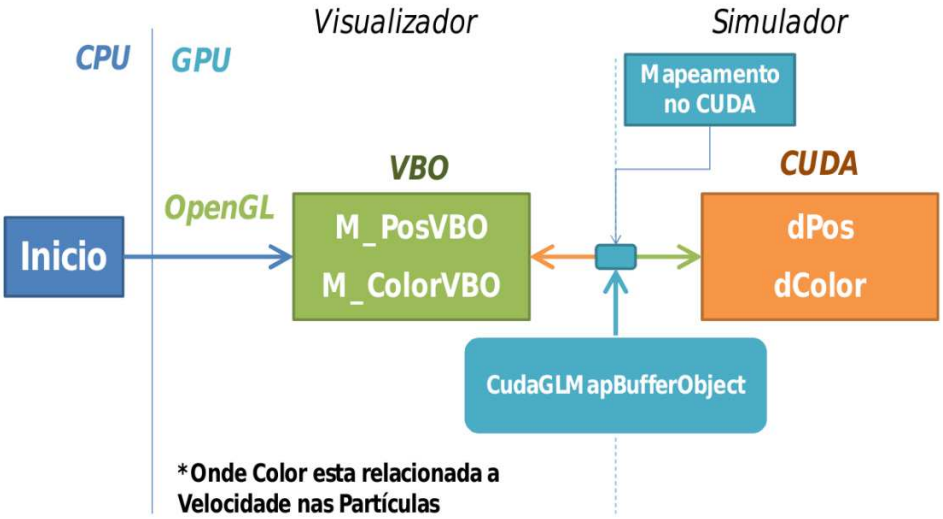


Figura 3: Interação CUDA-OpenGL, mostra a relação do manejo das variáveis na GPU para a parte do simulação e o Visualizador no código, usando CudaGLMapBufferObject.

Point Sprite

Point Sprite é um recurso interessante suportado pelo OpenGL 1.5 e posteriores. Embora OpenGL sempre apoio mapeado de texturas num ponto, antes da versão 1.5 esta significava coordenar uma textura única aplicada num ponto inteiro. Grande numero de pontos texturizados eram simplesmente muitas versões de um único texel filtrado. Com Point Sprites você pode colocar uma imagem de textura 2D em qualquer lugar na tela, desenhando um único ponto 3D [28].

Provavelmente a aplicação mais comum de point sprite é para sistemas de partículas. Um grande número de partículas que se movem na tela podem ser representados como pontos para produzir uma série de efeitos visuais. No entanto, o que representa esses pontos como pequenas imagens 2D sobrepostas pode produzir dramática de streaming filamentos animados.

Point Sprite é um recurso interessante suportado pelo OpenGL 1.5 e posteriores. Embora OpenGL sempre apoio mapeado de texturas num ponto, antes da versão 1.5 esta significava coordenar uma textura única aplicada num ponto inteiro. Grande numero de pontos texturizados eram simplesmente muitas versões de um único texel filtrado. Com Point Sprites você pode colocar uma imagem de textura 2D em qualquer lugar na tela, desenhando um único ponto 3D [28].

Provavelmente a aplicação mais comum de point sprite é para sistemas de partículas. Um grande número de partículas que se movem na tela podem ser representados como pontos para produzir uma série de efeitos visuais. [28].

No trabalho apresentado aqui, o Point Sprites é definido como:

```
glEnable(GL_POINT_SPRITE_ARB);
glTexEnvi(GL_POINT_SPRITE_ARB, GL_COORD_REPLACE_ARB, GL_TRUE);
...
```

Ela consiste na ativação da `GL_POINT_SPRITE_ARB`, encaminhando o conjunto de parâmetros da textura no `GL_COORD_REPLACE_ARB` para o ambiente, em nosso caso, não se define uma textura se não um *GLSL* (OpenGL Shading Language) [29] pixel shader que faz os pontos tornar-se esferas.

Visualização das velocidades nas partículas e a textura da esfera nas partículas

Para que as partículas fiquem coloreadas em relação as velocidades, se emprega a variável *Color* que ficara no buffer de dupla utilidade, nesta questão só sera usado na parte de visualização do código (OpenGL). Desta maneira, usando a ferramenta de OpenGL, onde a faixa de cores fica de em relação os velocidades como:

```
color[originalIndex] = float2RGBA((length(vel)-velMin)/(velMax-velMin));
```

Onde se normaliza a velocidade da partícula em questão em relação á velocidades máxima (`velmax`) e mínima (`velmin`) no conjunto de partículas no problema, sendo esta usada pela sub-rutina `float2RGBA` que relação esta nos cores RGBA como:

```
float4 float2RGBA( float f )
  {
    float q, t;
    f *= 4;
    t = f - floor(f);
    q = 1 - t;

    if (f < 0) return make_float4(0.0f, 0.0f, 1.0f, 0.0f);
    else if (f < 1) return make_float4(0.0f, t*1.0f, 1.0f, 0.0f);
    else if (f < 2) return make_float4(0.0f, 1.0f, q*1.0f, 0.0f);
    else if (f < 3) return make_float4(t*1.0f, 1.0f, 0.0f, 0.0f);
    else if (f < 4) return make_float4(1.0f, q*1.0f, 0.0f, 0.0f);
    else return make_float4(1.0f, 0.0f, 0.0f, 0.0f);
  }
```

Desta forma, se pode colorear as esferas com `glColor4f` em relação aos cor das partículas. Na Fig. 4 vemos um exemplo de como ficam coloridas as partículas.

Além disso, o código feito pelo Prof. Green [20] traz feito para o visualizador das partículas um pixel shader com *GLSL* (OpenGL Shading Language) [29] que faz os pontos tornar-se esferas em relação a textura requerida pelo Point Sprite. Desta maneira temos a forma do pixel shader (`spherePixelShader`) que gera a textura que se ligara para cada partícula, como:

```
// pixel shader for rendering points as shaded spheres
const char *spherePixelShader = STRINGIFY(
void main()
```

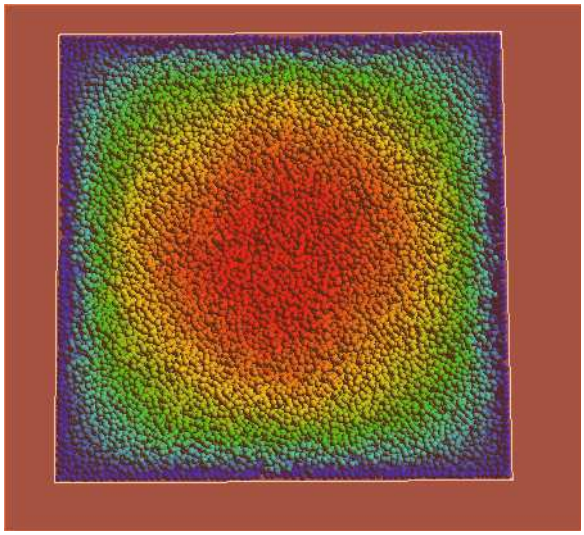


Figura 4: Exemplo de como ficam coloridas as partículas, para o problema de escorregamento de fluido.

```
{
    const vec3 lightDir = vec3(0.577, 0.577, 0.577);

    // calculate normal from texture coordinates
    vec3 N;
    N.xy = gl_TexCoord[0].xy*vec2(2.0, -2.0) + vec2(-1.0, 1.0);
    float mag = dot(N.xy, N.xy);

    if (mag > 1.0) discard;   // kill pixels outside circle

    N.z = sqrt(1.0-mag);

    // calculate lighting
    float diffuse = max(0.0, dot(lightDir, N));

    gl_FragColor = gl_Color * diffuse;
}
                                );
```

Desta maneira, a visualiza das partículas ficam como um conjunto de esferas no espaço, como se mostra na Fig.5.

4 Interação Partícula-partícula

É relativamente simples de programar um sistema de partículas onde elas interagem uns com os outros. A maioria dos sistemas de partículas usadas em vídeo jogos de hoje se enquadram

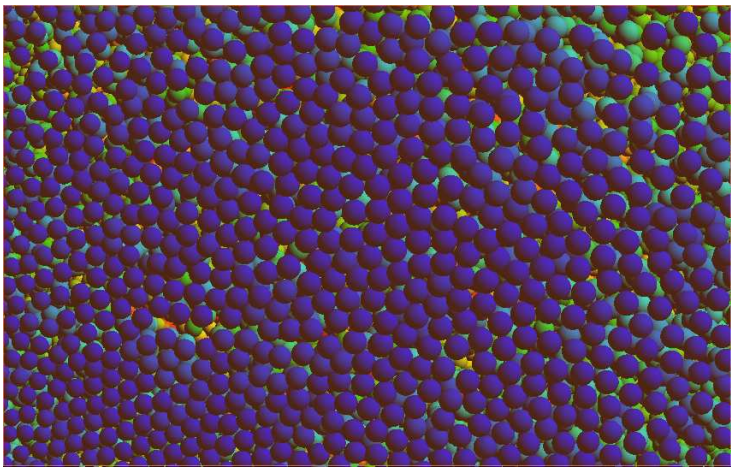


Figura 5: Resultado da visualização das partículas como esferas usando Point Sprite e o Pixel Shader.

nesta categoria. Neste caso, cada partícula é independente e podem ser simuladas trivialmente em paralelo.

O exemplo "n-corpo"incluído no CUDA SDK inclui iterações na forma de atração gravitacional entre corpos [31,32]. Ele demonstra que é possível obter um excelente desempenho em "n-corpos"para a simulação gravitacional usando CUDA ao realizar os cálculos de interação de uma forma de força bruta e Computar todas as interações para os n-corpos. O uso de memória compartilhada significa que este método não fica vinculado por largura de banda da memória.

No entanto, para as iterações locais (tais como colisões) que pode melhorar o desempenho usando subdivisão espacial. O aspecto chave aqui é que, para muitos tipos de iteração, a força de interação cai com a distância. Isto significa que podemos calcular a força de uma dada partícula por apenas comparação com todos os seus vizinhos dentro de um determinado raio, o que é feito aplicando um pesquisador de partículas. As Técnicas de subdivisão espacial do espaço da simulação para desenvolvimento dos pesquisadores de partículas é o modo mais fácil para encontrar os vizinhos de uma partícula [33].

4.1 Malha uniforme

Neste trabalho para a subdivisão espacial, se emprega o método de malha uniforme [34,35], que é uma forma simples e eficiente para subdivisão. Assim, uma malha uniforme subdivide o espaço de simulação para uma malha de células de tamanho uniforme. Por simplicidade, se utiliza uma malha em que o tamanho de célula é o mesmo que o tamanho da partícula (o dobro do seu raio). Isto significa que cada partícula pode abranger apenas um número limitado de células da malha (8 em 3 dimensões). Além disso, se se assumir nenhuma interpenetração entre as partículas, não há um limite superior fixo ligado ao número de partículas por célula de malha (4, em três dimensões).

Usa-se uma chamada malha "solta (loose)", em que cada partícula é atribuída a uma única célula de malha com base no seu ponto central. Uma vez que cada partícula pode, po-

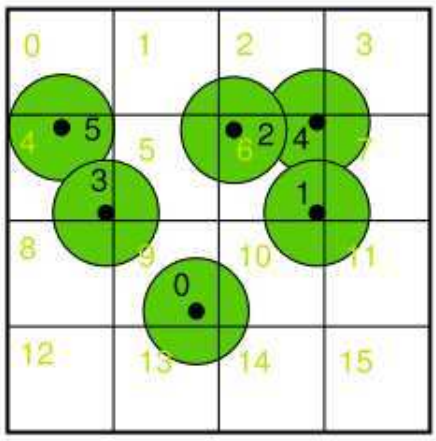
tencialmente, estar em sobreposição várias células da malha, o que significa que durante o processamento de colisões, devemos também analisar as partículas nas células vizinhas (3 x 3 x 3 = 27 no total) para ver se eles estão ou não tocando a partícula em questão. Este método permite armazenar as partículas nas células da malha simplesmente classificando-las por seu índice de malha [20, 33].
A estrutura da malha de dados é gerada a partir do zero em cada passo de tempo. É possível realizar alterações incrementais para a estrutura da rede na GPU, mas esta abordagem é simples e o desempenho é constante, independentemente do movimento das partículas [20].

4.1.1 Construção da malha usando ordenamento

O abordagem que implementamos em este trabalho é utilizado ordenação. O algoritmo consiste de vários Kernel's. O primeiro kernel "calcHash"calcula um valor de *hash* (mistura) para cada partícula com base em sua *ID* de célula. Neste caso, basta usar o *ID* de célula linear como o *hash*. O kernel armazena os resultados no vetor "particleHash"em memória global como um par `uint2` (*hash* celular, *ID* de partículas).
Então se classificam as partículas com base em seus valores de *hash*. A classificação é realizada utilizando o ordem da base rápido fornecido pela biblioteca CUDPP, que usa o algoritmo descrito em [36].
Para que esta lista de classificados seja útil, se tem que ser capaz de encontrar o início de qualquer célula na lista de classificados. Isto é conseguido através da execução de outro kernel chamado de "findCellStart", que utiliza um thread por partícula e compara o índice da célula onde a partícula esta ficando com o índice da célula da partícula anterior na lista ordenada. Se o índice é diferente, isto indica o início de uma nova célula, bem como o endereço de início é escrito para outra vetor com numeração diferente. O código usando classificação encontra também o índice do final de cada célula de uma maneira similar [37].
A Fig. 6 demonstra a criação da rede usando o método de classificação. Como uma otimização adicional, re-ordenar a posição e velocidade de vetor em ordem de ordenação para melhorar a coerência das pesquisas na memoria de textura durante o processamento de colisão.

5 Colisão de partículas

Uma vez que se tem construída a estrutura da malha que pode usar-se para acelerar a interações partícula-partícula. No código de exemplo, realiza colisões entre partículas simples, utilizando o método *DPD* [37]. Sendo esta parte, da simulação onde se calculam as equações que fazem a relação entra as partículas, e as relações numéricas que atualizaram a posição e a velocidade.



Index	Unsorted list (cell id, particle id)	List sorted by cell id	Cell start
0	(9, 0)	(4, 3)	
1	(6, 1)	(4, 5)	
2	(6, 2)	(6, 1)	
3	(4, 3)	(6, 2)	
4	(6, 4)	(6, 4)	0
5	(4, 5)	(9, 0)	2
6			
7			
8			
9			5
10			
11			
12			
13			
14			
15			

Figura 6: Construção da malha usando ordenamento.

5.1 Posicionamento das partículas

Para fazer a formulação *DPD* é preciso primeiro plantear o posicionamento do problema entre as partículas i e j. Considera-se o contato entre as partículas como na Fig. 7, onde se calcula a distancia de sobreposição entre partículas dado como δ_{ij}, de esta maneira, o melhor caminho é fazer a descomposição das componentes globais do sistema coordenado desde um sistema local de componentes normal-tangencial. Onde a distancia entre centros das partículas (D_{ij})

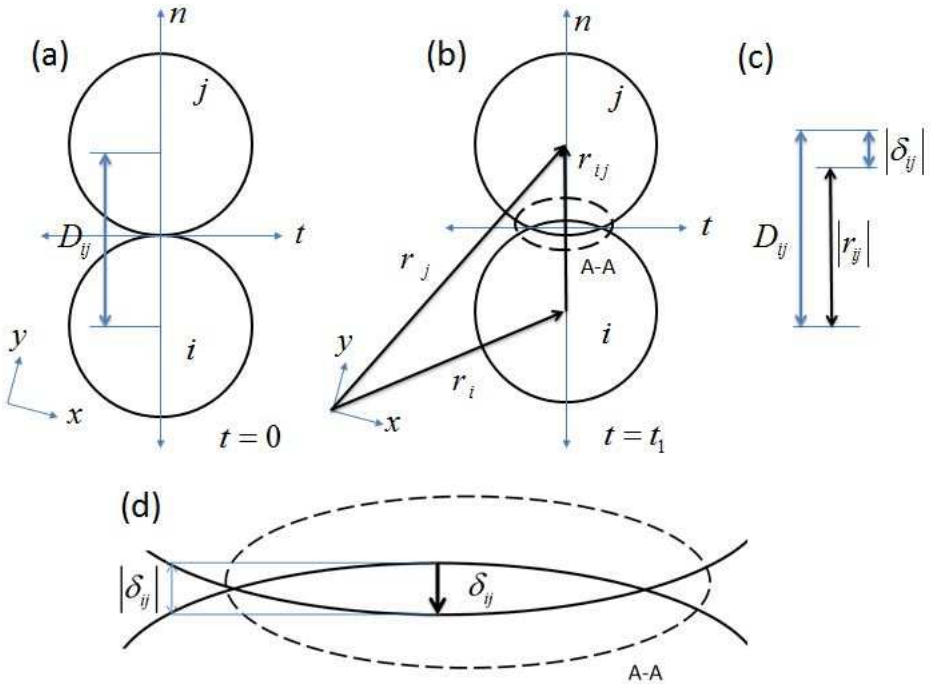


Figura 7: distancia de sobreposição entre partículas dado como δ_{ij} [38].

no primeiro instante (Fig. 7 (a)) de contato $\Delta t \approx 0$ é igual à soma de raios das partículas dado como:

$$D_{ij} = R_i + R_j \tag{1}$$

Onde R_i e R_j são os raios de i e j respectivamente. Entanto, o vetor distancia entre centros na direção normal também chamada distancia relativa entre as partículas (r_{ij}) depois do contacto entre as partículas é dada pelos vetores de posição $\mathbf{r}_i$ e $\mathbf{r}_j$ (Fig. 7 (b)) como:

$$\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_j \tag{2}$$

Assim, pode ser formulado o vetor unitário normal (η_{ij}) como:

$$\eta_{ij} = \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|} \tag{3}$$

De esta maneira, a distancia de sobreposição entre partículas é dada pelo vetor δ_{ij} (Fig. 7 (c) e (d)) calculado como:

$$\delta_{ij} = (D_{ij} - |\mathbf{r}_{ij}|)\,\eta_{ij} \tag{4}$$

5.2 Formulação própria *DPD*

Minha Hipóteses sobre *DPD* é que é derivado do comportamento da formulação *DEM*, já que os dois são derivados da Dinâmica Molecular (*DM*). Onde a o comportamento em *DPD* esta dado pela soma das forças conservativa ($\mathbf{F}^c$), força dissipativa ($\mathbf{F}^d$) e força aleatória (ou randômica) ($\mathbf{F}^r$), como a força:

$$\mathbf{F}_{ij} = \mathbf{F}^c + \mathbf{F}^d + \mathbf{F}^r \tag{5}$$

5.2.1 Força conservativa

Se conhecemos a força conservativa ou força repulsiva, a qual considera-se que é um tipo de força repulsiva na direção normal às partículas em contato, dada como:

$$\begin{aligned} \mathbf{F}^c_{ij} &= -K_n\left[(D_{ij} - |\mathbf{r}_{ij}|)\,\eta_{ij}\right] \\ \mathbf{F}^c_{ij} &= -K_n * D_{ij}\left[\left(1 - \tfrac{|\mathbf{r}_{ij}|}{D_{ij}}\right)\eta_{ij}\right] \end{aligned} \tag{6}$$

Onde que γ_t é coeficiente de amortecimento tangencial, K_t modulo de mola tangencial e lembremos que a variável D_{ij} é uma constante em função dos raios das partículas em iteração (1). Então considerando a função peso $W(r)$ como:

$$W(r) = 1 - \frac{|\mathbf{r}_{ij}|}{D_{ij}} \tag{7}$$

E mudando $K_n * D_{ij}$ como a contante A_{ij}, a equação 6 pode ser transformada como:

$$\mathbf{F}^c = A_{ij} W(r) \eta_{ij} \tag{8}$$

A que é muito similar ás utilizadas na literatura [14, 39, 40].

5.2.2 Força Dissipativa

Em *DPD* a força dissipativa tradicionalmente é definida como:

$$\mathbf{F}^d = \gamma W^d(r) v_{ij}^n \tag{9}$$

Onde se considera $W^d(r) = [W(r)]^2$ [5,12,41]. De esta forma, $W^d(r)$ é dada como:

$$\begin{aligned} W^d(r) &= \left(1 - \frac{|\mathbf{r}_{ij}|}{D_{ij}}\right)^2 \\ W^d(r) &= 1 - 2\frac{|\mathbf{r}_{ij}|}{D_{ij}} + \left(\frac{|\mathbf{r}_{ij}|}{D_{ij}}\right)^2 \end{aligned} \tag{10}$$

Então, a equação 9 pode ser re-definida com a equação 10 como:

$$\begin{aligned} \mathbf{F}^d &= \gamma W^d(r) v_{ij}^n \\ \mathbf{F}^d &= \gamma \left[1 - 2\frac{|\mathbf{r}_{ij}|}{D_{ij}} + \left(\frac{|\mathbf{r}_{ij}|}{D_{ij}}\right)^2\right] v_{ij}^n \\ \mathbf{F}^d &= \gamma v_{ij}^n + \gamma v_{ij}^n \left[\left(\frac{|\mathbf{r}_{ij}|}{D_{ij}}\right)^2 - 2\frac{|\mathbf{r}_{ij}|}{D_{ij}}\right] \end{aligned} \tag{11}$$

5.2.3 Força Aleatoria

Para a força aleatória, esta dada na literatura [5, 12, 14] como:

$$\mathbf{F}^r = -\sigma W^r(r) \xi_{ij} \eta_{ij} \tag{12}$$

Onde ξ_{ij} é uma variável aleatória com meia em zero (0) e desviação igual a um (1), e σ é a magnitude da variáveis aleatória ξ_{ij} e $W^r(r) = W(r)$.

5.2.4 Aplicação de CUDA pela varável aleatória ξ_{ij}

Para a variável ξ_{ij} usa-se a distribuição normal, usando a função da biblioteca de *CUDA* para formar números randômicos no *Device* chamada de *CURAND* [42], e tiramos de ela a função *curand normal*, fazendo seu chamamento como:

```
__device__ float
curand_normal (curandState *s);
```

Esta é uma função que entrega uma distribuição normal **float** com meia $0,0$ e desviação padrão $1,0$. Este resultado pode ser escalado e deslocou-se para produzir valores de distribuição normal com qualquer média e desvio padrão.

5.2.5 Condições de fronteiras

O domínio dos problemas a implementar no visualizador são feitos em três dimensões e se representa em um cubo de 2x2x2 (adimensional), neste volumem se delimita a iteração entre as partículas em todos os problemas. Para a simulação se implementam para as fronteiras

solidas, por exemplo no caso as paredes do cubo, o caso de não penetração de parede com uma condição de *no-slip* [43–45], onde ao atingir a partícula sob a fronteira (parede), ela é mudada na direção do vetor velocidade, sendo esta completamente deslocada em direção contraria, mas no caso das grandezas das componentes da velocidade elas se mantem com as novas direção da velocidade.
Outro caso do fronteira que se implementa em este trabalho, é a condição de cavidade para fluxo continuo de partículas, com este as partículas que querem penetrar uma parede, são deslocadas a outra posição do domínio do problema mas suas componentes de velocidades não são mudadas, no problema de escorregamento continuo que usa esta condição, esto acontece entre as paredes superior e inferior, que se tratara mais a profundidade na seguente secção.

6 Ferramentas interativas e Visualização

Usando a ferramenta OpenGL e GLUT se consegue visualizar em tempo real as partículas. Esta visualização esta composta por um cubo como se pode ver na Fig.8, onde as linhas dos cantos de cor branco, representam as paredes do cubo, que como se pode reparar contem as partículas em seu interior, desta forma o domínio dos problemas que se estudam neste trabalho ficaram sempre dentro do cubo.

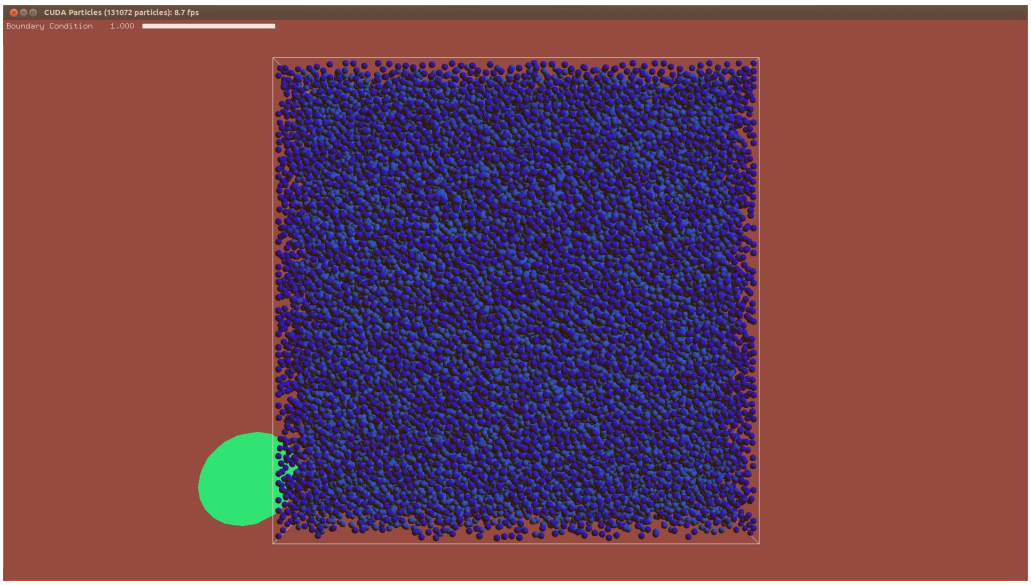


Figura 8: Visualizador de simulador de partículas por CUDA.

6.1 Menus auxiliares e comandos por teclas

O visualizador tem duas aplicações para modificar em tempo real os problemas simulados, o primeiro como se mostra na Fig.14, é ativado fazendo uso do botão secundário do mouse, o que chamaremos de menu principal, neste se podem selecionar as fazendo clique com o botão principal do mouse sob a alternativa a usar, além como se mostra no menu (Fig.14) ao lado das palavras que definem a ação fazer (Reset block, Reset random, Hole e etc.) em colchetes

quadrados os comandos em letras no teclado para aplicar as mesmas ações sem precisar de ativar o menu. Desta forma, só é preciso usar os comandos o teclado para ativar as mudanças em tempo real que sejam precisas.

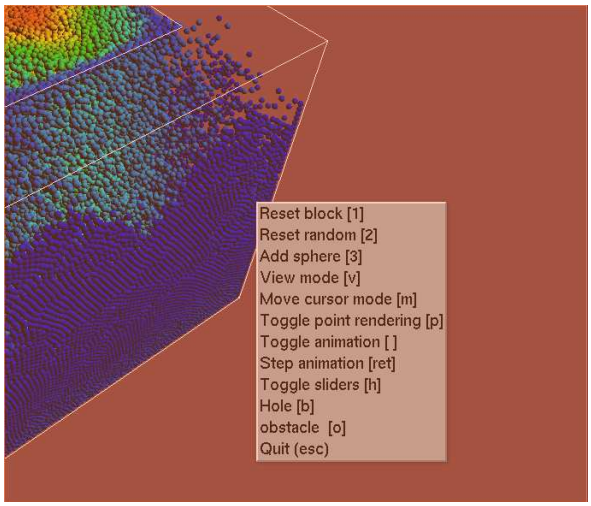


Figura 9: Menu auxiliar usando o botão secundário do mouse e comandos no teclado.

6.1.1 Escorregamento continuo de partículas

Uns dos problemas que se podem ativar com o menu principal com o comando **obstacle [o]**, é o problema de escorregamento continuo de partículas. Ele consiste de uma condição onde o fluido esta fluindo constantemente no domínio (do quadrado) pela ação da gravidade, o seja as partículas que se estão deslocando para a fronteira inferior (parede inferior do cubo) com uma velocidade dada, são deslocadas a fronteira superior (parede superior do cubo) mantendo as componentes da velocidade, desta forma se gera uma condição continua de fluxo das partículas, como se pode ver na Fig.10. Para modificar esta condição, se implemento um menu que se ativa na parte superior da janela de (ver Fig. 8), onde a variação do tamanho do orifício se pode aumentar ou reduzir empregando, gerando condições de parede solida ou de fluxo continuo de partículas nas paredes inferior e superior (1.0 representa a parede totalmente com fluxo de partículas), como se mostra na Fig.10. Entanto, na Fig.11 se mostra condições intermediarias de orifício, mudando a condição no menu superior.

6.1.2 Queda de coluna de partículas

O simulador feito pelo Prof. Green [20], presenta o exemplo de coluna de partículas já pronto, que é uma coluna de partículas que caí pela ação da gravidade, mas que na parede inferior do cubo ela se comporta como uma superfície solida, elas começam a descer como se mostra na Fig.12, onde se olham três passos de tempo entre a condição iniciar até a colisão com todas as paredes frontais do cubo, onde se usa nas partículas a visualização da velocidade e o método de *DPD*.

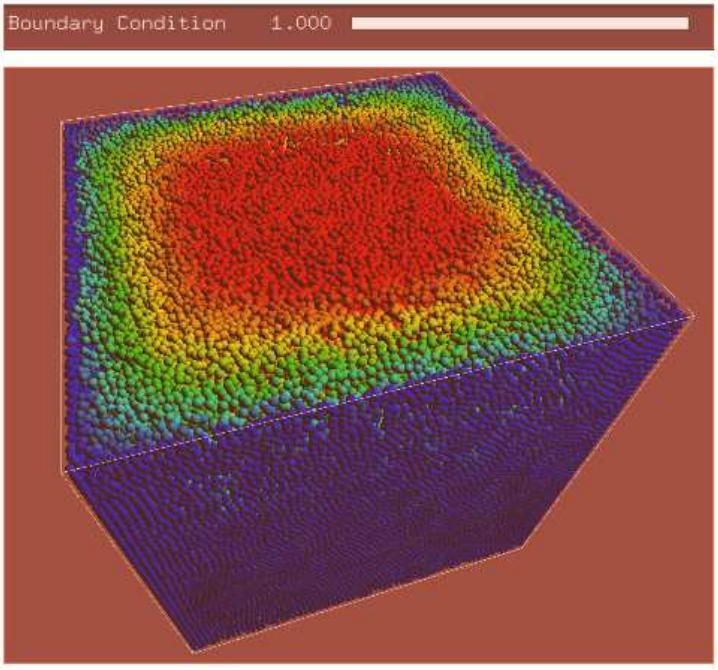


Figura 10: Condição de fronteira de orifício para escorregamento de fluido.

Este exemplo se ativa, desativando no menu principal a opção **Hole [b]** (que desativam os orifícios) e no mesmo menu posteriormente **Reset block [1]**, esto para evitar que o simulador não fique em um estado intermediaria com o exemplo de *Escorregamento continuo de partículas*, ainda que pode ser interessante seu estudo, mas neste trabalho não sera tratado.

6.1.3 Interação com obstaculo em queda de coluna de partículas

Para este caso se implementa as mesmas ações que no exemplo de *queda de coluna de partículas*, só que usando o menu principal ativamos o comando **obstacle [o]**, o qual gerara um cubo de 0.2x0.2x0.2, sob a parede inferior do cubo, onde seu centro para (x, y, x) ficara na posição

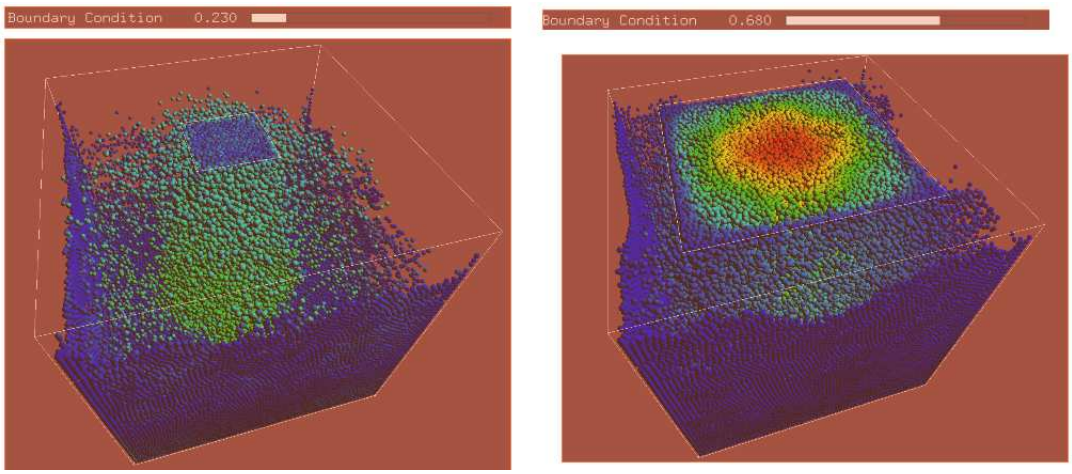


Figura 11: Condição de fronteira, duas variações do orifício para escorregamento de fluido, modificando menu ativado pelo teclado.

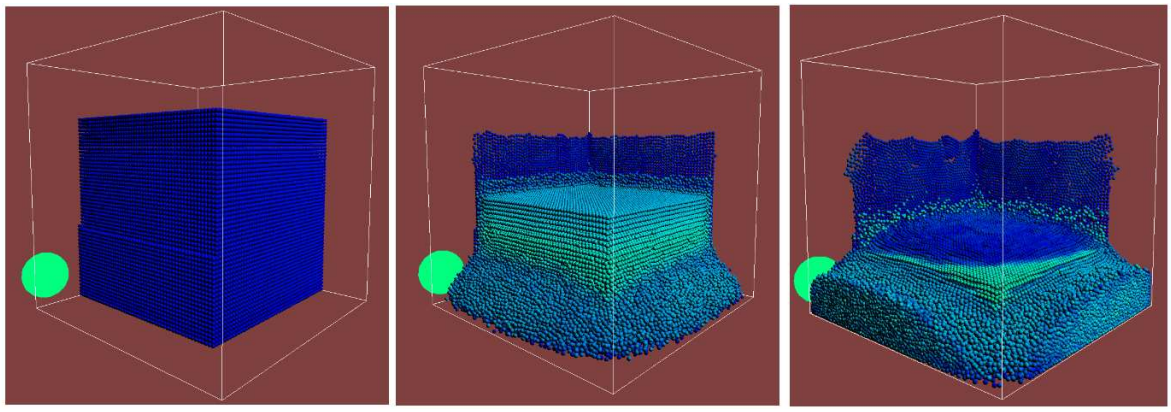


Figura 12: Passos de tempo no exemplo de coluna de partículas, até elas atingir todos as paredes frontais.

$(0.7, -0.9, 0.7)$ considerando a coordenada $(0, 0, 0)$ no centro do cubo que define o domínio do problema. Este esta disposto com a condição de fronteira solida para evitar a penetração das partículas na fronteira. Na Fig.13 que mostra o comportamento do obstaculo em interação com o escorregamentos das partículas na coluna.

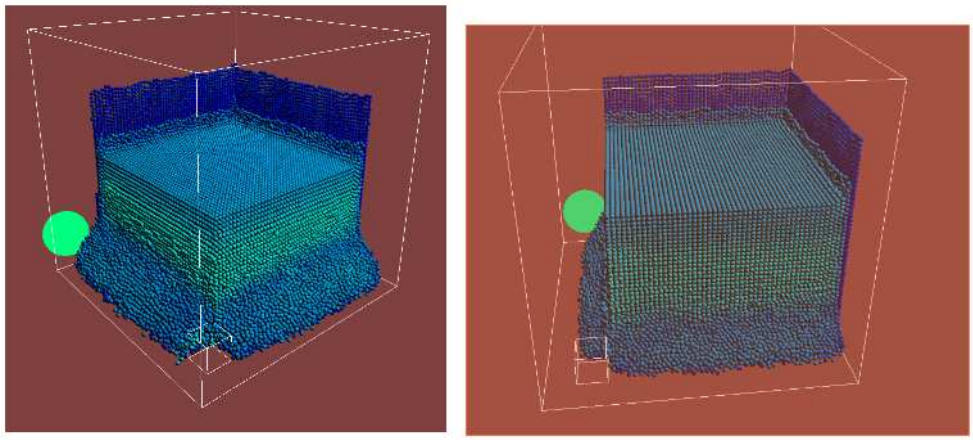


Figura 13: Interação com obstaculo em queda de coluna de partículas.

6.1.4 Menu modificação de parâmetros na simulação

O menus de modificação de parâmetros é ativado usando tecla **[h]** no teclado, este comando mostrara um menu na parte superior como se olha na Fig.14.

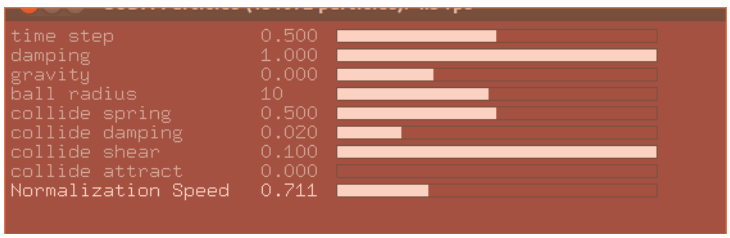


Figura 14: Se mostra o visualizador de simulador de partículas por CUDA.

Onde se pode modificar algumas propriedades relacionadas as colisões das partículas na formulação *DPD* na simulação em tempo real, como passos de tempo (*time step*), constantes nas equações de força (todos os parâmetros *collide*) e a velocidade de normalização (*Normalization Speed*) que definira a intensidade com a qual se transforma a relação de velocidades entre partículas em cores *RGBA* entre máximos e mínimos.

7 Discussão

O visualizador de partículas mostra um comportamento interessante como se pode ver nas Fig.10 e 11 onde se pode reparar que se forma um perfil de velocidades como é de esperar em um problema de fluxo continuo de partículas. O que valida primeiro o modelo, e duas condições importantes, a primeira o uso dos parâmetros da formulação *DPD* e a segunda a forma de implementar as condições de fronteira. Situação que foi facilitado com a visualização da velocidade nas partículas e o uso dos menus, todo realizado em tempo.
O emprego das ferramentas de integração CUDA-OpenGL usando a ferramenta CudaGLMapBufferObject, mostram simular e visualizar em tempo real das simulações das partículas, nos exemplos implementados de uma forma adequada, o que ajuda a que os tempos e a iteração do usuário sejam menores, e facilitem a modificação no tempo de execução. Assim, aumenta a possibilidade de de geral maiores estudos dos problemas executados e diminuir a procura de comportamentos desejados nas simulações, já que todo se modifica na hora.
Desta forma, se justifica a importância das ferramentas como CUDA, OpenGL e GLUT, especialmente sobre o uso de visualizadores comerciais ou livres, onde geralmente só se visualiza em estado de pós-simulação e onde a iteração com a mesma é quase nula em modificação de problemas e parâmetros. Já que esta presenta uma visualizações direta e a interação com a simulação dá um beneficio maior. Em especial em partículas os visualizadores especializados tanto comercial como livres, em especial para problemas de simulação física, são muito escaços, o que faz em partículas uma necessidade para seu implementação e estudo.

8 Conclusões

Se consegue que o visualizador interativo usando método de *DPD*, seja uma ferramenta útil para o estudo e estabilização do método, para vários exemplos.
A visualização da grandeza das velocidades nas partículas, foi uma aplicação bem sucedida, devido a que se consegue achar condições para o funcionamento ótimo do modelo. No caso especial da aplicação das condições de contorno.
As ferramentas iterativas, como menus, zoom, deslocamentos e rotação. Ajudam para uma melhor observação dos fenômenos, já que em partículas a observação individual e detalhada se torno critica.
A execução de vários exemplos em um mesmo visualizador e além de sua modificação em tempo real, acrescenta e facilita o estudo dos fenômenos em partículas, de uma maneira mais

geral.
A interação em GPU de OpenGL e CUDA, é uma interessante ferramenta tanto para a visualização como na simulação de fenômenos físicos.

Para perspectivas futuras, a geração de corte, para visualização das partículas, em posições intermediárias e conseguir visualizar Linhas de fluxo que mostrem a trajetória das partículas através do tempo, ajudariam a um melhor estudo do problema.
Além, melhorar a implementação das colisão nas partículas usar um método mais complexo como *SPH* ou *MLS*. E por ultimo, Implementação de problemas mas complexos, para casos de fenômenos com comportamento só possíveis de simular em estado de multi-escala.

Referências

[1] S Li and WK Liu. *Meshfree particle methods*. Springer, New York, New York, USA, 2007.

[2] GR Liu. *Mesh free methods: moving beyond the finite element method*. CRC Press, Boca Raton, 2009.

[3] GR Liu and YT Gu. *An introduction to meshfree methods and their programming*. 2005.

[4] D M Heyes, J Baxter, U Tüzün, and R S Qin. Discrete-element method simulations: from micro to macro scales. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 362(1822):1853–65, September 2004.

[5] Thomas Steiner, Claudio Cupelli, Roland Zengerle, and Mark Santer. Simulation of advanced microfluidic systems with dissipative particle dynamics. *Microfluidics and Nanofluidics*, 7(3):307–323, January 2009.

[6] L D Liberskyb. Smoothed Particle Hydrodynamics: Some recent improvements and applications. 7825(96), 1996.

[7] W. Benz and E. Asphaug. Simulations of brittle solids using smooth particle hydrodynamics. *Computer Physics Communications*, 87(1-2):253–265, May 1995.

[8] Nasr M. Ghoniem†, Esteban P. Busso, Nicholas Kioussis, and Hanchen Huang. Multiscale modelling of nanomechanics and micromechanics: an overview. *Philosophical Magazine*, 83(31-34):3475–3528, October 2003.

[9] Petros Koumoutsakos. Multiscale Flow Simulations Using Particles. *Annual Review of Fluid Mechanics*, 37(1):457–487, January 2005.

[10] GR Liu and MB Liu. *Smoothed particle hydrodynamics: a meshfree particle method*. 2003.

[11] U Frisch, B Hasslacher, and Y Pomeau. Lattice-gas automata for the Navier-Stokes equation. *Physical review letters*, 423:11–12, 1986.

[12] PJ Hoogerbrugge, J Koelman, Home Search, Collections Journals, About Contact, My Iopscience, and I P Address. Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics. *EPL (Europhysics Letters)*, 155, 2007.

[13] Robert D Groot and Patrick B Warren. Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. 107(11):4423–4435, 1997.

[14] Pep Español. 8.6 DISSIPATIVE PARTICLE DYNAMICS. In S. Yip, editor, *Handbook of Materials Modeling*, pages 2503–2512. Springer, Netherlands, 2005.

[15] P Lancaster and K Salkauskas. Surfaces generated by moving least squares methods. *Mathematics of computation*, 37(155):141–158, 1981.

[16] WK Liu, S Li, and T Belytschko. Moving least-square reproducing kernel method Part II: Fourier analysis. *Computer Methods in Applied Mechanics and ...*, 7825(96), 1996.

[17] Pietro De Palma, P. Valentini, and M. Napolitano. Dissipative particle dynamics simulation of a colloidal micropump. *Physics of Fluids*, 18(2):027103, 2006.

[18] Moubin Liu, Paul Meakin, and Hai Huang. Dissipative particle dynamics simulation of multiphase fluid flow in microchannels and microchannel networks. *Physics of Fluids*, 19(3):033302, 2007.

[19] Pep Español and Mar Serrano. Dynamical regimes in DPD. page 8, October 1998.

[20] Simon Green. Particle simulation using CUDA. *NVIDIA Whitepaper, December 2010*, (May), 2010.

[21] UD Schiller. *Dissipative Particle Dynamics. A Study of the Methodological Background.* PhD thesis, University of Bielefeld, 2005.

[22] J Sanders and E Kandrot. CUDA by Example. *An Introduction to General-Purpose GPU Programming ...*, 2011.

[23] D Kirk, WH Wen-mei, and W Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann/Elsevier, 2010.

[24] M Elwenspoek and TSJ Lammerink. Towards integrated microliquid handling systems. *Journal of ...*, 4:227–245, 1999.

[25] T Bourouina and JP Grandchamp. Modeling micropumps with electrical equivalent networks. *Journal of Micromechanics and ...*, 398, 1999.

[26] AM Bos and PC Breedveld. 1985 Update of the bond graph bibliography. *Journal of the Franklin Institute*, (May):79–82, 1985.

[27] P.A. Cundall and O.D.L. Strack. A discrete numerical model for granular assemblies. *Geotechnique*, 1(29):47–65, 1979.

[28] R Wright, N Haemel, GM Sellers, and B Lipchak. *OpenGL SuperBible: comprehensive tutorial and reference*. 2010.

[29] J Kessenich, D Baldwin, and R Rost. The opengl shading language. *Language version*, 2004.

[30] J Neider, T Davis, and M Woo. *OpenGL. Programming guide*, volume 4. 1997.

[31] C NVIDIA. CUDA C Best Practices Guide. (January), 2010.

[32] Programming Guide. Nvidia cuda™. 2012.

[33] C Ericson. Real-time collision detection. *Chemistry & ...*, 2004.

[34] Sung-Eui Yoon and Peter Lindstrom. Mesh layouts for block-based caches. *IEEE transactions on visualization and computer graphics*, 12(5):1213–1220, 2006.

[35] Bruce Palmer and Jarek Nieplocha. Efficient Algorithms for Ghost Cell Updates on Two Classes of MPP Architectures. *IASTED PDCS*, 2002.

[36] Nadathur Satish, Computer Sciences, Mark Harris, Michael Garland, and Santa Clara. Designing Efficient Sorting Algorithms for Manycore GPUs. (May):1–10, 2009.

[37] H Nguyen. *Gpu gems 3*. 2007.

[38] Stefan Luding. Introduction to discrete element methods. *European Journal of Environmental and Civil ...*, (Md):785–826, 2008.

[39] E. Moeendarbary, T. Y. Ng, and M. Zangeneh. Dissipative Particle Dynamics: Introduction, Methodology and Complex Fluid Applications — a Review. *International Journal of Applied Mechanics*, 01(04):737–763, December 2009.

[40] Patrick B Warren. Dissipative particle dynamics. *Current Opinion in Colloid & Interface Science*, 3(6):620–624, December 1998.

[41] M. Serrano, P. Español, and Ignacio Zúñiga. Collective effects in dissipative particle dynamics. *Computer Physics Communications*, 121-122:306–308, September 1999.

[42] C NVIDIA. Curand library. *NVIDIA Corporation, Santa Clara, California*, 2008.

[43] a. Berkenbos and C.P. Lowe. Accurate method for including solid–fluid boundary interactions in mesoscopic model fluids. *Journal of Computational Physics*, 227(9):4589–4599, April 2008.

[44] Dmitry a. Fedosov, Igor V. Pivkin, and George Em Karniadakis. Velocity limit in DPD simulations of wall-bounded flows. *Journal of Computational Physics*, 227(4):2540–2559, February 2008.

[45] D.C. Visser, H.C.J. Hoefsloot, and P.D. Iedema. Comprehensive boundary method for solid walls in dissipative particle dynamics. *Journal of Computational Physics*, 205(2):626–639, May 2005.