5
Fluidos Estáveis

Neste capítulo é apresentado um método para resolver numericamente as equações de Navier-Stokes para fluidos incompressíveis. Esse método foi desenvolvido por Joe Stam em 1999 e é conhecido como Stable Fluids (stam1999). Ele é um método semi-lagrangeano, que é incondicionalmente estável.

5.1
Navier-Stokes para fluidos incompressíveis e o método da projeção

Ao supor que o fluido possui temperatura e densidade constantes, ele é descrito pelo campo de velocidades $\mathbf{V}$ e pelo campo escalar de pressão p, que são variáveis no tempo e no espaço, denotando por X a coordenada espacial.

Dados os campos $\mathbf{V}$ e p em um tempo inicial $t = 0$, esses campos evoluem de acordo com as equações de Navier-Stokes para fluidos incompressíveis:

$$\frac{\partial \mathbf{V}}{\partial t} = -(\mathbf{V} \cdot \nabla)\mathbf{V} - \frac{1}{\rho}\nabla p + \mu \nabla^2 \mathbf{V} + F \tag{5-1}$$

$$\nabla \cdot \mathbf{V} = 0, \tag{5-2}$$

onde μ é a constante de viscosidade, ρ é a constante de densidade e F uma força externa que atua sobre o fluido.

Suponha-se que o líqüido está delimitado em um domínio D. Neste caso, as condições de fronteira são dadas por uma função u_D, definida no bordo de D, ∂D, de forma que a componente normal da velocidade em relação a ∂D seja zero, isto é, o material não atravessa o bordo.

Acontece que, para o fluido em questão, os campos de pressão e velocidade que aparecem na equação de Navier-Stokes estão relacionados, ou seja, é possível combinar as duas equações em uma única equação para a velocidade. Para mostrar esse resultado é necessário o seguinte teorema:

Teorema 5.1 (Decomposição Helmholtz-Hodge) *Um campo vetorial W, em D, onde D é uma região do espaço com bordo ∂D suave, pode ser unicamente decomposto na forma*

$$W = \mathbf{V} + \nabla p,$$

onde p é um campo escalar, $\mathbf{V}$ tem divergente livre (zero) e é paralelo a ∂D, isto é, $\mathbf{V} \cdot n = 0$ em ∂D, onde n é o vetor normal a ∂D.

Demonstração Usando a identidade

$$\nabla \cdot (p\mathbf{V}) = (\nabla \cdot \mathbf{V})p + \mathbf{V} \cdot \nabla p,$$

aplicando o teorema da divergência e o fato de que $\nabla \cdot \mathbf{V} = 0$, obtém-se

$$\int_D \mathbf{V} \cdot \nabla p \, dV = \int_D \nabla \cdot (p\mathbf{V}) \, dV = \int_{\partial D} p\mathbf{V} \cdot n \, dA = 0,$$

desde que $\mathbf{V} \cdot n = 0$ em ∂D. Assim, temos a seguinte relação de ortogonalidade

$$\int_D \mathbf{V} \cdot \nabla p \, dV = 0$$

que usamos para demonstrar a unicidade. Supondo $W = \mathbf{V}_1 + \nabla p_1 = \mathbf{V}_2 + \nabla p_2$. Tem-se

$$0 = \mathbf{V}_1 - \mathbf{V}_2 + \nabla(p_1 - p_2).$$

Fazendo o produto interno com $\mathbf{V}_1 - \mathbf{V}_2$ e integrando

$$0 = \int_D \|\mathbf{V}_1 - \mathbf{V}_2\|^2 + (\mathbf{V}_1 - \mathbf{V}_2) \cdot \nabla(p_1 - p_2) \, dV = \int_D \|\mathbf{V}_1 - \mathbf{V}_2\|^2 \, dV.$$

Usando o teorema de Stokes e a relação de ortogonalidade, segue que, $\mathbf{V}_1 = \mathbf{V}_2$ e conseqüentemente $\nabla p_1 = \nabla p_2$, ou seja, $p_1 = p_2 +$ constante

Se $W = \mathbf{V} + \nabla p$ teremos $\nabla \cdot W = \nabla \cdot \nabla p = \nabla^2 p$ e portanto, $W \cdot n = n \cdot \nabla p$. Agora, dado W, seja p definido pela solução do seguinte problema de Neumann

$$\nabla p = \nabla \cdot W \text{ em } D, \quad \text{com} \quad \frac{\partial p}{\partial n} = W \cdot n \quad \text{sobre } \partial D.$$

A solução existe e é única a menos da soma de constante a p. Com esse p defina $\mathbf{V} = W - \nabla p$, assim $\mathbf{V}$ satisfaz $\nabla \cdot \mathbf{V} = 0$ e $\mathbf{V} \cdot n = 0$ por construção de p.

■

Pelo teorema, qualquer campo vetorial W pode ser decomposto como a soma de um campo que conserva massa e um campo gradiente.

Com esse resultado, define-se o operador $\mathbb{P}$, que projeta um campo vetorial W em sua porção com divergente livre $\mathbf{V} = \mathbb{P}(W)$. Esse operador é definido implicitamente aplicando $\nabla \cdot$ aos dois lados da equação do teorema.

$$W = \mathbf{V} + \nabla p$$

$$\begin{aligned} \nabla \cdot W &= \nabla \cdot \mathbf{V} + \nabla \cdot \nabla p \\ \nabla \cdot W &= \nabla \cdot \nabla p \\ \nabla \cdot W &= \nabla^2 p. \end{aligned} \tag{5-3}$$

Esta é uma equação de Poisson. A solução desta equação é usada para calcular a projeção $\mathbf{V}$

$$\mathbf{V} = \mathbb{P}(W) = W - \nabla p \ .$$

Aplicando o operador aos dois lados da equação de Navier-Stokes para fluidos incompressíveis, obtém-se

$$\begin{aligned} \mathbb{P}\left(\frac{\partial \mathbf{V}}{\partial t}\right) &= \mathbb{P}\left(-(\mathbf{V}\cdot\nabla)\mathbf{V} - \frac{1}{\rho}\nabla p + \mu\nabla^2\mathbf{V} + F\right) \\ \frac{\partial \mathbf{V}}{\partial t} &= \mathbb{P}\left(-(\mathbf{V}\cdot\nabla)\mathbf{V} - \frac{1}{\rho}\nabla p + \mu\nabla^2\mathbf{V} + F\right) \\ \frac{\partial \mathbf{V}}{\partial t} &= \mathbb{P}\left(-(\mathbf{V}\cdot\nabla)\mathbf{V} + \mu\nabla^2\mathbf{V} + F\right) - \mathbb{P}\left(\frac{1}{\rho}\nabla p\right) \\ \frac{\partial \mathbf{V}}{\partial t} &= \mathbb{P}\left(-(\mathbf{V}\cdot\nabla)\mathbf{V} + \mu\nabla^2\mathbf{V} + F\right). \end{aligned} \tag{5-4}$$

Onde, usa-se os fatos que $\mathbb{P}(\mathbf{V}) = \mathbf{V}$ e que $\mathbb{P}(\nabla p) = 0$. Para esta última temos $\nabla p = 0 + \nabla p$ e, como a decomposição de Helmholtz-Hodge é única, chegamos a $\mathbb{P}(\nabla p) = 0$.

A equação (5-4) é a equação fundamental para a evolução do estado do fluido. A equação (5-4) basicamente descreve como funciona o algoritmo.

1. Calcular o termo dentro dos parênteses. O resultado é um campo W.

2. Projetar W sobre seu campo com divergente livre. Para aplicar a projeção resolve-se a equação $\nabla \cdot W = \nabla^2 p$, e projeta-se

$$\frac{\partial \mathbf{V}}{\partial t} = \mathbb{P}(W) = W - \nabla p.$$

3. Conhecendo $\frac{\partial \mathbf{V}}{\partial t}$, é possível avançar o estado do campo $\mathbf{V}$ para o próximo passo de tempo.

4. Comece novamente.

5.2 Método numérico na Resolução da Equação de Navier-Stokes

Nesta seção, é explicado o método numérico usado para resolver as equações de Navier-Stokes, usando o processador da placa gráfica. Para isso, resolve-se numericamente cada termo da equação (5-1), separadamente.

O resultado de uma etapa é o início da próxima. Assim, reduzimos a sobrecarga no envio de dados para a GPU e carregaremos no hardware gráfico somente dados relativos ao estado inicial e as condições de fronteiras da simulação.

A figura 5.1 mostra um exemplo da simulação. A estrela do mar e o peixe representam barreiras para o fluido. Em um determinado tempo t, o estado da simulação é descrito pelo campo de velocidades do fluido. A pressão e os outros atributos são todos calculados com base no campo de velocidade, como será visto na seqüência.

5.2.1 Conceito Principal

Para resolver a equação de Navier-Stokes para fluidos incompressíveis, é necessário definir um operador (S), que evolui para o estado do fluido em um passo de tempo. Esse operador será a composição de quatro operadores: advecção ($\mathbb{A}$), difusão ($\mathbb{D}$), aplicação das forças ($\mathbb{F}$) e restauração do campo com divergente livre ($\mathbb{P}$).

A cada passo de simulação o estado do fluido é descrito pelo campo velocidade $\mathbf{V}$ e é aplicado o operador de advecção. Então, $\mathbf{V} = \mathbb{A}(\mathbf{V})$, depois $\mathbf{V} = \mathbb{D}(\mathbf{V})$, $\mathbf{V} = \mathbb{F}(\mathbf{V})$, e finalmente $\mathbf{V} = \mathbb{P}(\mathbf{V})$, evoluindo o fluido em um passo de tempo.

A cada passo será aplicado mais um operador $\mathbf{V} = \mathbb{V}(\mathbf{V})$, conhecido como confinamento de vorticidade, que atua sobre a velocidade, injetando de volta a energia perdida com dissipação numérica.

Cada um desses operadores será melhor analisado, e, no próximo capítulo, será mostrada uma maneira de implementá-los, usando o hardware gráfico, com o auxílio da linguagem Cg.

5.2.2 Advecção

Olhando para a equação 5-4 é necessário calcular o termo $-(\mathbf{V} \cdot \nabla)\mathbf{V}$, que corresponde à advecção na equação de Navier-Stokes.

O fluido transporta a si mesmo e suas propriedades com o tempo, e esse transporte ocorre na direção do campo velocidade. Pensando em uma célula

Figura 5.1: Uma simulação com domínio de 65536 células de fluido, com condição de fronteira definida pelo bordo da figura do peixe dourado e pelo bordo da estrela do mar

fixa no fluído, ela possui certas propriedades em um determinado tempo t, e num tempo $t + \Delta t$, as propriedades da célula serão alteradas com o fluxo do fluido, na direção da velocidade.

Suponha-se que a célula, no tempo t, possui uma determinada quantidade q, de alguma propriedade, (pode ser velocidade, densidade, temperatura ou qualquer quantidade carregada pelo fluido). Então, o valor de q, no tempo $(t + \Delta t)$ dessa célula é dada pela equação:

$$q(x, t + \Delta t) = q(x - \mathbf{V}(x, t)\Delta t, t). \tag{5-5}$$

Dividindo o passo de tempo em dois, obtemos a seguinte equação

$$q(x, t + \Delta t) = q\left(x - \mathbf{V}(x, t)\frac{\Delta t}{2} - \mathbf{V}\left(x - \mathbf{V}(x, t)\frac{\Delta t}{2}, t\right)\frac{\Delta t}{2}, t\right). \tag{5-6}$$

Fazendo uso desta equação, o cálculo do advecção fica mais caro, porém, como esse não é o gargalo da aplicação, vale a pena usar esta equação, pois ela

aumentará muito a precisão do operador.

5.2.3 Difusão

Olhando para a equação 5-4 é necessário calcular o termo $\mu\nabla^2\mathbf{V}$, que corresponde à difusão pela viscosidade, na equação de Navier-Stokes.

Os fluidos com viscosidade mais elevada têm uma maior resistência ao fluxo. Essa resistência resulta na difusão do momentum, (perda de energia), isto é, perda de velocidade. Por isso, esse operador é chamado de difusão.

Aplicar o efeito da viscosidade do fluido é equivalente a usar a equação de difusão na velocidade.

$$\frac{\partial \mathbf{V}}{\partial t} = \mu\nabla^2\mathbf{V}, \tag{5-7}$$

onde μ é o coeficiente de viscosidade.

Não se resolve esta equação com integração explícita, pois o método ficaria com restrição no passo de tempo. Pensando em integração implícita, chega-se à seguinte equação:

$$(I - \mu\delta\nabla^2)u(x, t + \delta t) = u(x, t). \tag{5-8}$$

Esta equação pode ser resolvida com algum processo de relaxação.

5.2.4 Restauração do campo com divergente livre

Neste ponto é necessário calcular o gradiente da pressão ∇p e subtraí-lo do campo de velocidade. Para encontrar ∇p, basta resolver a equação (5-3), que é uma equação de Poisson. Precisamos calcular o divergente da velocidade intermediária. A figura 5.2 mostra o campo de divergência, que é extraído do campo de velocidade. Através dessa equação encontra-se o campo escalar de pressão p. A figura 5.2 mostra também esse campo. ∇p é calculado aplicando diferenças finitas, definidas pela fórmula

$$\nabla p = \left(\frac{\partial p}{\partial x}, \frac{\partial p}{\partial y}\right) = \left(\frac{p_{i+1,j} - p_{i-1,j}}{2\Delta x}, \frac{p_{i,j+1} - p_{i,j-1}}{2\Delta y}\right)$$

e em seguida, é subtraído o campo gadiente ∇p do campo de velocidades.

5.2.5 Confinamento de Vorticidade

O método baseado em (stam1999) sofre de dissipação numérica. Esse problema pode ser reduzido injetando a energia dissipada de volta no fluido,

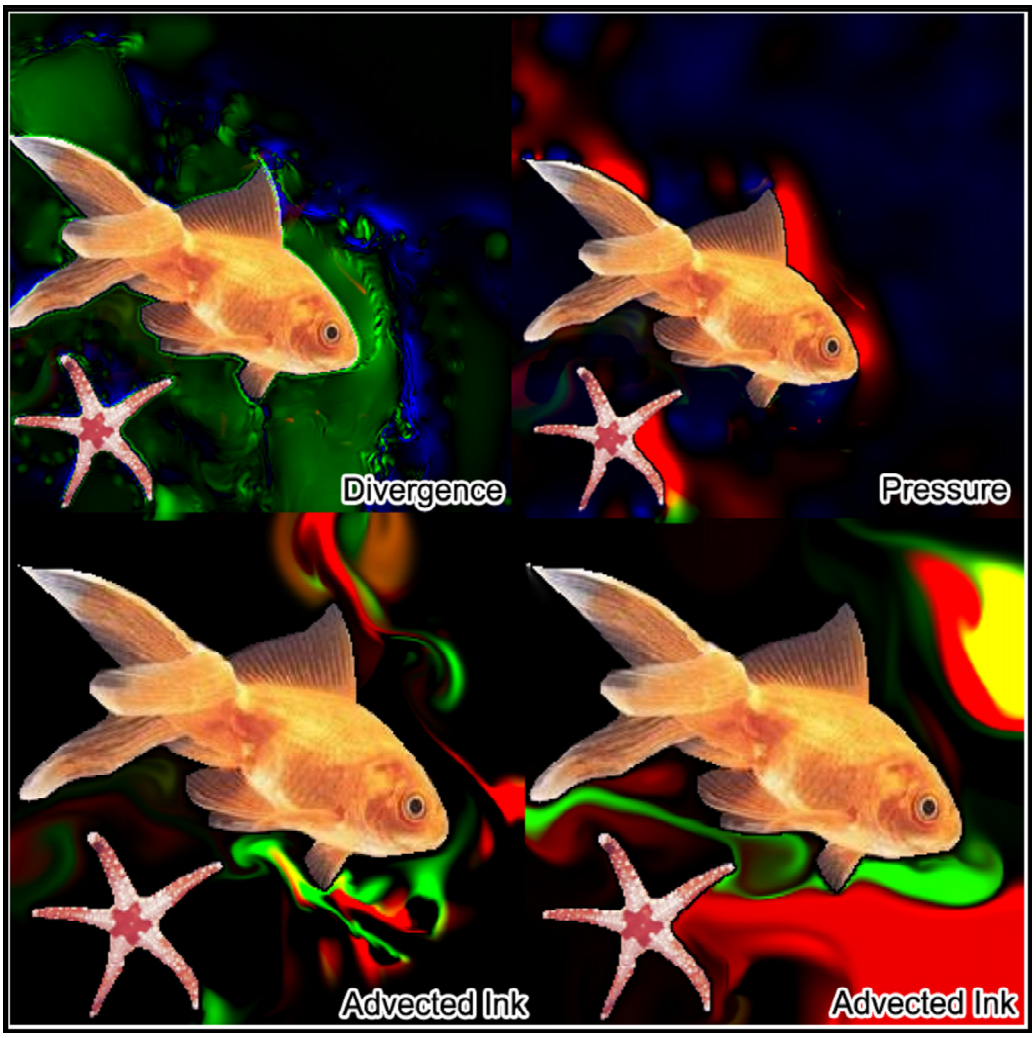

Figura 5.2: Um exemplo de visualização dos campos de divergência e pressão Em vermelho as altas pressões, e em verde as altas divergências

nas baixas freqüências, usando o método de confinamento de vórtice, descrito em (steinhoff1994).

Primeiramente, para gerar detalhes nas baixas freqüências do fluido é necessário identificar de onde vêm os detalhes. Num fluxo incompressível, a vorticidade

$$\omega = \nabla \times \mathbf{V} \tag{5-9}$$

dá a estrutura das baixas freqüências. A dissipação numérica atua justamente onde $|\omega|$ é maior. Então, simplesmente soma-se essa energia de volta no campo velocidade.

$$N = \frac{\eta}{|\eta|} \qquad (\eta = \nabla|\omega|) \tag{5-10}$$

onde N aponta da menor para a maior concentração de vorticidade. A direção

e magnitude da força que é adicionada, fica

$$f_{conf} = \epsilon \Delta x (N \times \omega) \tag{5-11}$$

onde $\epsilon > 0$ é usado para controlar a quantidade de detalhes de baixa freqüência, que é somado ao fluxo e onde Δx é a distância entre os nós da malha.

Esse método é usado com sucesso em modelos muito complexos de fluxos, onde não é possível adicionar nós suficientes, para não perder os detalhes de baixa freqüência.

5.3 Implementação

A implementação do método usa a linguagem Cg. O método é uma extensão do trabalho descrito por (harris2004), principalmente no que diz respeito ao tratamento das fronteiras e uso de confinamento de vorticidade.

A discretização do fluido é feita usando uma malha bidimensional, e as propriedades do fluido são armazenadas em texturas na GPU. Cada pixel da textura corresponde a uma célula de fluido. A velocidade do fluido é codificada nos canais R e G da textura, e existem texturas intermediárias para armazenamento da pressão, divergente, e a velocidade intermediária, sem divergente livre.

Se a implementação fosse em CPU, cada parte da simulação possuiria um loop que iteraria sobre cada célula de fluido. Na GPU não é possível usar esse tipo de loop. Mas o pipeline de fragmentos da GPU é desenvolvido para executar o mesmo cálculo para cada fragmento que é processado pela GPU.

Depois que esses cálculos são realizados sobre os fragmentos, o resultado é a cor do fragmento, que é armazenada em um pixel buffer. Esse buffer pode ser tratado como uma textura, e é usado no passo subseqüente do algoritmo.

A cada passo do algoritmo são geradas primitivas "Quad"para disparar o pipeline gráfico. E em cada parte do algoritmo são carregados novos shaders. O resultado de cada etapa é uma textura, que será usada na etapa seguinte. Assim é reduzida a necessidade de comunicação CPU GPU.

A figura 5.3 mostra como funciona cada etapa do algoritmo. Primeiro disparamos o pipeline usando uma primitiva Quad. A primitiva é rasterizada e os fragmentos são processados pelo Fragment Shader, que possui acesso de leitura a memória da GPU. O resultado desse processamento é o pixel buffer, que é usado para alimentar o próximo passo da aplicação.

5.3.1

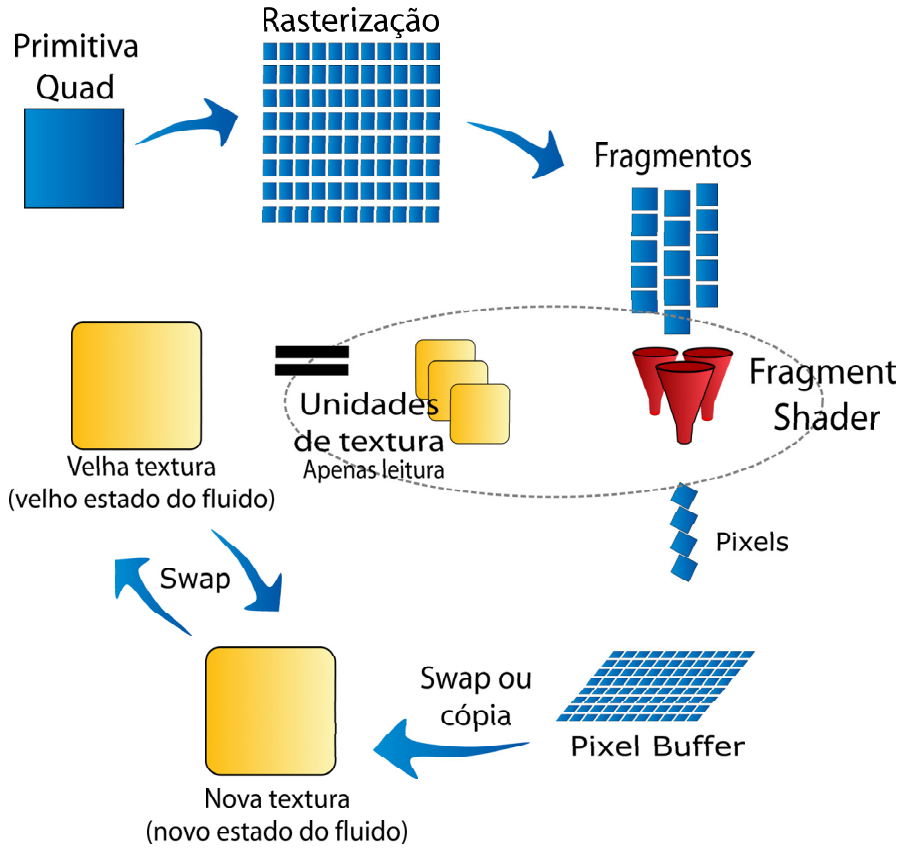

Figura 5.3: Pipeline de uma etapa da simulação, como os pixels são reutilizados a cada etapa.

Advecção

Para aplicar o operador de advecção ($\mathbb{A}$), carregamos o seguinte fragment program, que denominaremos de shader de Adveção, no pipeline gráfico, passando para ele a velocidade do fluido (em forma de textura) no parâmetro u. Já na variável rdx representa a distância entre as células da malha e o parâmetro x é a quantidade a ser advectada (velocidade, temperatura, etc.).

```
float4 main( uniform float timestep,
             uniform float rdx,
             uniform samplerRECT u,
             uniform samplerRECT x,
             float2 coord : WPOS
    ) : COLOR
    {
             float2 pos = coord - (timestep * rdx * 0.5 * f2texRECT(u, coord ));
             pos = pos - (timestep * rdx * 0.5 * f2texRECTbilerp(u, pos));
             return f4texRECTbilerp(x, pos);
    }
```

O estado do fluido está armazenado na memória da GPU, em uma textura que denominaremos de textura das velocidades. É necessário aplicar o operador

de advecção na velocidade. Isso é feito disparando o pipeline gráfico com um Quad, e usando o shader de adveção.

O shader recebe no parâmetro u a textura de velocidade, e como a quantidade a ser advectada é também a velocidade, passamos também no parâmetro x a mesma textura. O resultado desse processo, é o pixel buffer com os valores da velocidadade advectada. O pixel buffer é usado então como a nova textura de velocidade.

A função texRECTbilerp retorna o valor da textura interpolado. Como o Cg ainda não tem essa função nativa, é necessário implementá-la.

5.3.2 Difusão

Como em (harris2004), o método para resolver a equação da difusão é a iteração de Jacobi. A seguir está descrito o shader da iteração de Jacobi. A iteração de Jacobi está sendo usada para resolver a equação de difusão, que na verdade é o mesmo que resolver um sistema linear Ax=b.

Como a matriz associada à equação de difusão é bastante simples, não é necessária uma representação explícita da matriz. Na verdade, descrevemos a matriz usando apenas alguns parâmetros, que são passados para o shader: $\alpha = (\Delta x)^2/\nu\Delta t$ e $\beta = 4 + \alpha$.

O parâmetro b é a velocidade, e o chute inicial é também a textura de velocidade.

```
float4 main(half2 coords : WPOS,
            uniform half alpha,
            uniform half rBeta,
            uniform samplerRECT x,
            uniform samplerRECT b
        ):COLOR
        {
            // left, right, bottom, and top x samples
            half4 xL = h4texRECT(x, coords - half2(1, 0));
            half4 xR = h4texRECT(x, coords + half2(1, 0));
            half4 xB = h4texRECT(x, coords - half2(0, 1));
            half4 xT = h4texRECT(x, coords + half2(0, 1));
            // b sample, from center
            half4 bC = h4texRECT(b, coords);
            // evaluate Jacobi iteration
            return (xL + xR + xB + xT + alpha * bC) * rBeta;
        }
```

Assim, para realizar um passo de iteração de Jacobi, é necessário disparar o pipeline com uma primitiva Quad, e rasterizar com o shader da iteração de Jacobi com os parâmetros descritos acima. O resultado é um pixel buffer, que será a nova textura de velocidade. Quanto mais passos forem executados, maior será a precisão na solução da equação.

5.3.3
Aplicação de forças

A única força externa que estamos levando em conta é a força aplicada pelo usuário ao interagir com o mouse sobre o fluido. Conseguimos isso usando o seguinte fragment program, que soma valores à6666 velocidade.

```
float4 main(half2 coord :WPOS,
        uniform float2 x,
        uniform float2 dx,
        uniform float radius,
        uniform samplerRECT u
    ):COLOR
    {
        float dist = distance(x,coord);
        if (dist < radius)
            return f4texRECT(u, coord) + exp(-dist) * float4(dx,0,1);
        else
            return f4texRECT(u, coord);
    }
```

Quando o ponteiro do mouse se move sobre o fluido, desenhamos uma primitiva quad do tamanho da textura da velocidade, e aplicamos esse fragment program. Passamos na variável x a posição do ponteiro, dx a direção do movimento e a textura da velocidade em u.

Obtendo como resultado o pixel buffer, então usamos esse como a nova textura de velocidade, que foi alterada num círculo com centro na posição do ponteiro do mouse, e a alteração é suavizada por uma gaussiana. Assim, a simulação continua de forma mais suave.

5.3.4
Restauração do campo com divergente livre

Aqui também seguiremos os passos de (harris2004). Vamos usar o método de Jacobi para resolver a equação de Poisson da Pressão, onde $\alpha = -(\Delta x)^2$, e $\beta = 4$, Δx é o espaçamento entre as células da malha. Guardamos a pressão em uma textura (Não é exatamente a pressão que obtemos, mas ela multiplicada por um fator. Como estamos interessados no gradiente, isso não importa.), usamos uma fórmula de diferenças finitas para calcular ∇p e subtraímos esse gradiente da velocidade. Teremos então uma nova velocidade com o divergente livre.

5.3.5
Confinamento de vorticidade

Para facilitar a implementação desse passo vamos dividi-lo em três partes:

1. A partir da velocidade calculamos o ω como na Eq. (5-10) e o armazenamos em uma textura.

2. Calculamos o valor das duas componentes de N e armazenamos em outra textura.

3. Somamos f_{conf} da Eq. (5-11) no campo de velocidades.

Para realizar esses cálculos precisaremos de três fragments programs: para calcular o rotacional $\nabla \times u$

```
float4 main(half2 coords : WPOS,
            uniform samplerRECT u
      ):COLOR
      {
            float4 wL=texRECT(w, coords- half2(1,0));
            float4 wR=texRECT(w, coords+ half2(1,0));
            float4 wB=texRECT(w, coords- half2(0,1));
            float4 wT=texRECT(w, coords+ half2(0,1));
            return ((wR.y-wL.y)+(wT.x-wB.x))
      }
```

para calcular as componentes de N

```
float4 main(half2 coords : WPOS,
            uniform samplerRECT curl
      ):COLOR
      {
            float cL=abs(texRECT(curl, coords- half2(1,0)).x);
            float cR=abs(texRECT(curl, coords+ half2(1,0)).x);
            float cB=abs(texRECT(curl, coords- half2(0,1)).x);
            float cT=abs(texRECT(curl, coords+ half2(0,1)).x);
            float2 N = float2((wR.x-wL.x),(wT.x-wB.x));
            normalize(N);
            return float4(N,0,0);
      }
```

para adicionar a f_{conf} à velocidade

```
float4 main(half2 coords : WPOS,
            uniform float h,
            uniform float eps,
            uniform samplerRECT curl,
            uniform samplerRECT n,
            uniform samplerRECT u
      ):COLOR
      {
            float2 U = texRECT(u,coords).xy;
```

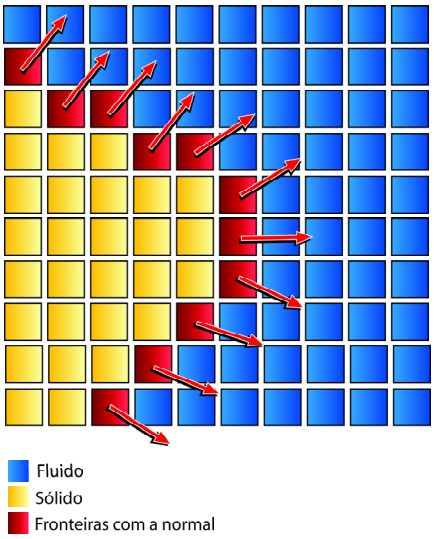

Figura 5.4: Representação da condição de fronteira com normal

```
    float2 N = texRECT(N,coords).xy;
    float1 C = texRECT(N,coords).x;
    return h*eps*float4(U.x+N.x*C,U.y+N.y*C,0,0);
}
```

5.4 Condições de fronteira

Nesta seção vamos mostrar uma maneira de acoplar um número arbitrário de fronteiras. O fluido reagirá a essas fronteiras como se fossem barreiras impostas por corpos sólidos. Não trataremos aqui fronteiras extremamente finas. Para este caso o método de simulação teria que possuir algum esquema adaptativo.

A condição de fronteira que usaremos para a velocidade é conhecida como *no-slip condition*, isto é, a velocidade deve ser zero na fronteira, e para a pressão devemos usar a condição de fronteira de Neumann $\partial p/\partial n = 0$, isto é, a mudança de pressão na direção normal à fronteira é zero.

Vamos incorporar essas condições à simulação como mais um operador no algoritmo. A cada passo do algoritmo vamos aplicar as condições de fronteira para ajustar a velocidade e a pressão na fronteira. Realizaremos isso criando mais um *fragment program*. Desse modo, a cada passo da simulação usaremos os operadores descritos acima, e, em seguida, aplicaremos o programa que força as condições de fronteiras, avançando assim o estado da simulação.

Codificaremos a fronteira em uma textura, assim como codificamos a velocidade do fluido, classificando cada tipo de pixel dessa textura. Vamos utilizar apenas o canal R de uma textura. Cada pixel da textura é então classificado como fluido (R negativo), sólido (R=0), ou fronteira (R variando

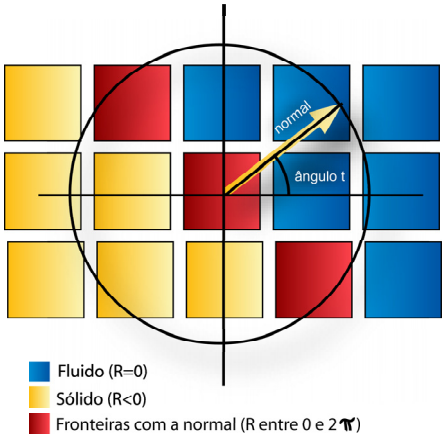

Figura 5.5: Valores da textura para o fragment program da fronteira.

Figura 5.6: Interpolação bilinear para suavizar o erro na fronteira.

de 0 a 2π). Na fronteira vamos armazenar a direção do vetor normal, a fronteira parametrizando a direção como sendo:

$$n = (cos(t), sin(t)), \tag{5-12}$$

onde t é o valor que foi armazenado na textura, ou seja, para cada fragmento poderemos responder se ele corresponde à fronteira ou ao fluido. Dessa maneira podemos escrever um fragment program que a cada passo de simulação copia para a fronteira o valor da pressão do ponto da malha para onde aponta a normal. Assim teremos a condição de Neumann respeitada, e da mesma forma copiamos para a fronteira o valor da velocidade com sentido invertido, ou seja, teremos que a velocidade vale zero nos pixels entre a borda e o fluido. Ao copiarmos essa velocidade novamente usamos a função texRECTbilerp. Para interpolar o valor da pressão e da velocidade, também interpolaremos a pressão.

Agora o código em CG desse fragment program

```
float4 main(uniform samplerRECT w,
            uniform samplerRECT x,
            uniform float scale,
            float2 coord : WPOS
        ):COLOR
        {
            float t = texRECT(w,coord ).r;
            float4 r;

            //boundary
            float2 n = float2 (sin(t),cos(t));
            r = scale * f4texRECTbilerp(x,coord +n);

            //fluid
            if (t<0) r = texRECT(x, coord );

            //solid
            if (t == 0.0) r = float4(0,0,0,0);

            return r;
        }
```

Usamos o mesmo programa, tanto para a pressão, como para a velocidade. Para a pressão, em w passamos a textura contendo a codificação da fronteira, em x passamos a textura contendo a pressão. O fator scale vale 1. Já para a velocidade o fator scale vale -1, e passamos a textura com a velocidade na variável x.

6
Resultados e Conclusão

Neste capitulo serão analisados os resultados obtidos com a simulação de fluido e também a performance da simulação.

A tabela 6 mostra a velocidade em frames por segundo, de vários cenários de simulação. A velocidade da simulação depende principalmente de dois fatores: o primeiro é o tamanho do domínio de simulação. Foram realizados testes com domínios de $128 \times 128 = 16.384$,$256 \times 256 = 65.536$, $512 \times 512 = 262.144$ e $1024 \times 1024 = 1.048.576$ de células de fluido. O segundo fator é o número de iterações na resolução dos sistemas lineares: quanto mais iterações, melhor a simulação, e pior a velocidade.

Para aplicações onde é necessária somente a simulação visual, as simulações com apenas 20 iterações já dão um bom resultado, permitindo assim gerar gráficos para jogos em tempo real. Já para aplicações em físicas são necessários mais passos.

Também foi realizada uma comparação quanto ao uso das condições de fronteira. Neste caso é possível ver que este não é o gargalo do algoritmo. Mesmo assim, é possivel otimizar esta parte da simulação, porém, valendo mais a pena estudar novos processos para reduzir o tempo gasto com as iterações de Jacobi.