Projeto Comp. Gráfica

Carlos Eduardo Nogueira Silva
Felipe Gomes da Silva
Gabriel Martins Brum
Luis Henrique Salomão Lobato

Setembro, 2025

Sumário

1 Introdução **2**

2 Sistemas de Partículas **2**
2.1 Fundamentação 2
2.1.1 Problemas 3

3 Simulação de Fluidos em Computação Gráfica **3**
3.1 Fundamentação 3

4 Geração Procedural de Terrenos **4**
4.1 Noise 5
4.1.1 Fractional Brownian Motion no Perlin Noise (or fractal noise) 7
4.2 Mapeamento de Texturas 8

5 Volumes (baseado em particulas) **8**
5.1 Modelagem de forma 8
5.1.1 Passo de Densidade 8
5.1.2 Passo de Velocidade 9
5.2 Conclusão da modelagem de forma 10
5.3 Shading 11
5.3.1 Interação da luz com volumes 11
5.3.2 Implementação básica de ray marching 12

6 Justificativa **12**

7 Objetivos **12**

8 Metodologia **12**

9 Resultados **12**

10 Conclusão **12**

1 Introdução

Em 1952 o computador MANIAC (Mathematical Analyzer, Numerical Integrator, and Computer) ou IAS foi construindo, prometendo resolver problemas matemáticos antes tidos como ou muito difíceis, ou muito demandantes de processamento humano. Em geral, podemos associar os seguintes problemas que rodavam no IAS (citados por Dyson em Turing's Cathedral: The Origins of the Digital Universe [4]) ao objeto de estudo do nosso projeto, a física:

1. Explosões nucleares, medidas em microsegundos
2. Ondas de choque, que variavam em tempo de microsegundos a minutos
3. Metereologia
4. Evolução biológia
5. Evolução estelar

Para explicitar esses avanços e, em especial, explorar a computação gráfica aplicada na física sobre o tempo, escolheu-se trabalhar com o tema de simulação de ambientes, unindo temáticas como sistemas de partículas, fluidos e terrenos. Estes tópicos, apesar de a priori parecem desconexos e um pouco restritivos quanto ao tema, são extremamente relevantes para a computação gráfica, e são amplamente utilizados em jogos, filmes e outras mídias visuais, além de terem fomentado a base para todas as simulações e aplicações físicas futuras.

Faz-se uma menção honrosa aos corpos rígidos e tecidos, que também são temas extremamente relevantes e interessantes, mas que por limitações de tempo e escopo do projeto, não puderam ser abordados.

2 Sistemas de Partículas

"Particle Systems—A Technique for Modeling a Class of Fuzzy Objects" -> primeiro artigo referente a modelagem de eventos chamados "nebulosos". Em geral, o modelo inova ao trocar a anterior representação de superfícies dos trabalhos previos. Usa-se de um oscilador estocástico. Esta técnica foi empregada no filme Star Trek II: The Wrath f Khan - minuto 1.39

Este paper representa a primeira aparição de um sistema de partículas, alterando a representação de primitivas anterior, como polígonos com extremidades definidas, mas como nuvens de partículas primitivas que definem seu volume (futuramente a representação Point Cloud também seria utilizada). Esse volume de partículas também não é uma entidade por si só, cada partícula tem seu *lifetime* próprio com a passagem do tempo (esta modelada por uma série de oscilações estocásticas - randômicas | ou pseudo).

Este projeto deriva de ideias prévias relacionadas aos videogames (provavelmente a primeira aplicação de cg em escala), como o grandessíssimo Evans e Sutherland Flight Simulator, que, além de ser o projeto inventor dos frame buffers (lembrem do swap buffers do opengl), iniciou (primitivamente) o processamento de explosões com um sistema simples de colisão e a tentativa de renderizar esse elemento "fuzzy" (como o fogo) mas sem a aleaotiriedade esperada.

evans e sutherland

paper evans e sutherland

2.1 Fundamentação

Para renderizar esse sistema, o seguinte pipeline é seguido: (1) novas partículas são introduzidas no sistema, (2) cada partícula recebe seus atributos individualmente, (3) toda partícula que exista no sistema depois de passar do seu tempo de vida, são removidas; (4) as partículas

restantes se movem baseadas nos seus atributos dinâmicos e (5) a imagem do sistema é criada no frame buffer.

Este paper tratou alguns parâmetros dos sistemas de partículas:

- numero de partículas geradas (densidade): $NPartsf = MeanPartsf + Rand() \times VarPartsf \rightarrow$ media de partículas que o sistema deve ter somado a um um valor randômico de -1 a 1 * a variância de partículas desejado.
- variância: o programador pode alterar a media através de alguma função qualquer, alterando o tamanho do volume.

Em geral, cada nova partícula possui:

1. Posição inicial
2. Velocidade e direção iniciais
3. Tamanho inicial
4. Cor inicial
5. Transparência inicial
6. Formato
7. Tempo de vida (em frames)

2.1.1 Problemas

O artigo cita três problemas: (1) partículas não podem interagir com outras superfícies (2) só há interação de fato através de planos de projeção que limitam o crescimento das partículas. (3) toda partícula eh um emissor de luz, que adiciona luz para as partículas internas e externas, não sendo factível com a realidade.

3 Simulação de Fluidos em Computação Gráfica

Os maiores desafios da simulação de fluidos está nos aspectos fisicos que se aplicam, como por exemplo, convecção, difusão, turbulência e tensão superficial [3]. No entanto, essas simulações eram (ao menos em 2003) quase inviáveis para serem empregadas em tempo real, portanto a precisão acaba sendo deixada parcialmente de lado nestas simulações.

A simulação de fluidos começou, basicamente, com a equação de de Navier-Stokes que descrevem a dinamica dos fluidos (não vamos nos aprofundar, mas para efeito de curiosidade, esse sistema de equações diferenciais se baseia em derivadas parciais e permitem determinar os campos de velocidade e de pressão num escoamento de fluidos). Sua forma geral é dada por:

$$\rho \frac{D\mathbf{v}}{Dt} = -\nabla p + \mu \nabla^2 \mathbf{v} + \rho \mathbf{f}$$

3.1 Fundamentação

Uma explicação simples: a equação de Navier-Stokes descreve como o movimento de um fluido é influenciado pela pressão, viscosidade e forças externas. Nela, ρ é a densidade do fluido, $\mathbf{v}$ é o vetor velocidade, p é a pressão, μ é o coeficiente de viscosidade, e $\mathbf{f}$ representa forças externas (como gravidade). Ela permite calcular como o fluido se move e se comporta em diferentes situações. Essa equação pode ser vista como um vetor de forças que atuam sobre o fluido, sendo representado como vetores de velocidades no espaço.

Para quem quiser se aprofundar.

Em 1983, T. Reeves [12] introduziu sistemas de partículas como uma técnica para modelar uma classe de objetos difusos. Desde então, tanto a abordagem Lagrangiana baseada em partículas quanto a abordagem Euleriana baseada em grades têm sido usadas para simular fluidos em computação gráfica. Desbrun e Cani [3] e Tonnesen [15] utilizam partículas para animar objetos macios. As partículas também foram usadas para animar superfícies [16], controlar superfícies implícitas [1] e animar fluxos de lava [2]. Nos últimos anos, a abordagem Euleriana tem sido mais popular para a simulação de fluidos em geral [6], água [13, 7, 5], objetos macios [9] e efeitos de derretimento [2].

O artigo de Müller et al. (2003) [**?**] apresenta uma abordagem eficiente para simulação de fluidos baseada em Smoothed Particle Hydrodynamics (SPH). O método SPH representa o fluido como um conjunto de partículas, onde cada partícula carrega propriedades como massa, posição, velocidade e densidade. As interações entre partículas são calculadas usando funções de suavização (kernels), permitindo simular efeitos como pressão, viscosidade e forças externas.

As principais fórmulas utilizadas no SPH são:

- Densidade:
$$\rho_i = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h)$$
onde ρ_i é a densidade da partícula i, m_j é a massa da partícula j, W é o kernel de suavização e h é o raio de influência.

- Pressão:
$$\mathbf{f}_i^{\text{pressão}} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(|\mathbf{r}_i - \mathbf{r}_j|, h)$$
onde p_i e p_j são as pressões das partículas i e j.

- Viscosidade:
$$\mathbf{f}_i^{\text{visc}} = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(|\mathbf{r}_i - \mathbf{r}_j|, h)$$
onde μ é o coeficiente de viscosidade e $\mathbf{v}_i$, $\mathbf{v}_j$ são as velocidades das partículas.

Essas fórmulas permitem calcular as forças que atuam sobre cada partícula, resultando em simulações de líquidos realistas e eficientes para aplicações interativas. Há de se notar, entretanto, que essas fórmulas tendem a ser muito custosas se implementadas conforme a descrição, sem otimização linear inclusa.

Logo, diversos papers foram publicados visando manter a qualidade o suciente, mas reduzindo o custo computacional e permitir que se faça essa simulação em tempo real.

4 Geração Procedural de Terrenos

A geração procedural de terrenos é uma técnica amplamente utilizada em computação gráfica para criar ambientes naturais de forma automática e eficiente. Essa técnica é especialmente útil em jogos, simulações e visualizações onde a criação manual de terrenos seria impraticável devido à sua complexidade e escala.

Em especial, trataremos de uma técnica bastante utilizada, o Perlin Noise, e uma variação deste, o Fractional Brownian Motion (fBm).

4.1 Noise

A princípio, o "Noise" foi definido como uma primitiva de modelagem de textura para o Pixel Streaming Editor que o Ken Perlin [10] propôs, mas a implementação da técnica pode ser reproduzida em outras ferramentas.

Em um espaço vetorial onde cada ponto para as coordenadas x, y, z são inteiros, cada ponto deste conjunto pode ser associado a um valor "pseudorandom", onde x, y, z são valores gradientes. Deste modo, deve-se mapear cada sequência ordenada de três inteiros em uma sequencia ordenada de 4 números reais, de modo que:

$[a, b, c, d] = H([x, y, z])$ onde $[a, b, c, d]$ definem a equação com o gradiente $[a, b, c]$ e o valor "d" em $[x, y, z]$. $H()$ seria uma função HASH.

A partir disso, se [x,y,z] está neste set de inteiros, é definido o $Noise([x, y, z]) = d[x, y, z]$, interpretado como um sinal (apresenta frequência).

Com estas definições, supondo um vetor aleatório array, e que ele representa um Donut $color = white * Noise(array)$, com a aplicação desta "transformada" foi possível criar essas ondulações no donut com as texturas brancas, de forma aleatória.

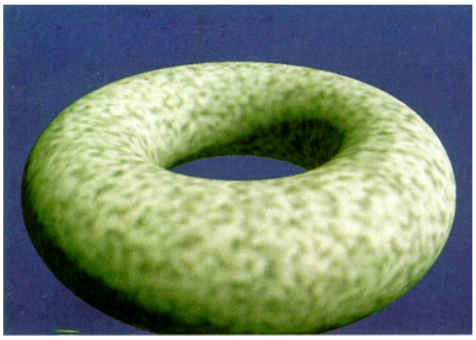

Figura 1: Donut com textura Noise aplicada

$color = Colorful(Noise(k * array))$ outro possível exemplo de aplicação,

Figura 2: Esfera com textura Noise aplicada

Figura 3: Cubo com textura Noise aplicada

Outra técnica seria o vetor diferencial do Noise(), que é definido pela taxa de variação instantânea do "ruído" através das três direções, chamados de Dnoise().

A partir desta aplicação fica possível criar outros tipos de perturbação, originando novas superfícies e texturas.

$$normal+ = Dnoise(array)$$

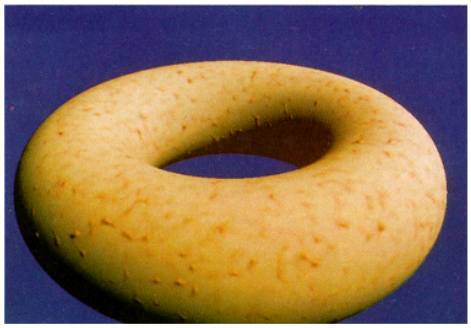

Figura 4: Donut com textura Dnoise aplicada

Como estes cálculos e tecnicas são aplicados em nível de pixel, a frequência de cada pixel é que está sendo utilizada, logo, qualquer frequência de taxa mais alta que não seja desejada é automaticamente removida.

Exemplo de cálculo com a frequência inversa, aplicando transformações em todos os octetos:

```
f=1
while (f < pixel_freq):
  normal + = Dnoise(f * point)
  f*=2
```

O autor descreve uma técnica para simular a aparência de mármore usando a função Noise(). O método parte do princípio de que a aparência do mármore resulta de camadas heterogêneas que foram deformadas por forças turbulentas antes de se solidificarem. A abordagem é, portanto, uma combinação de uma estrutura regular e simples (as camadas) com uma complexa estrutura estocástica (o ruído da turbulência). A base do modelo são as camadas, representadas por uma simples onda senoidal, sin(x). O autor usa a coordenada point[1] como o input para

essa função, e o valor resultante é então mapeado para cores através de uma função auxiliar marble_color(). Para adicionar o realismo das forças turbulentas, o autor introduz uma função turbulence(). Esta função é usada para perturbar a coordenada de entrada x antes que ela seja passada para a onda senoidal. O pseudocódigo que combina esses elementos é o seguinte:

```
function marble(point)
  x = point[1] + turbulence(point)
  return marble\_color(sin(x))
```

A função turbulence() é, por sua vez, uma soma de ruído em diferentes escalas, um processo que cria um padrão auto-semelhante ou 1/f. O algoritmo para a turbulence() é detalhado como:

```
function turbulence(p)
  t = 0
  scale = 1
  while (scale > pixelsize)
    t += abs(Noise(p / scale) * scale)
    scale *= 2
  return t
```

Este procedimento garante que a quantidade de ruído adicionada em cada escala seja proporcional ao seu tamanho, resultando na impressão visual de movimento browniano. Além disso, o uso da função abs() em cada iteração assegura que o gradiente da textura tenha limites descontínuos em todas as escalas, o que é interpretado visualmente como fluxo turbulento

Figura 5: Textura de mármore gerada com a função marble()

4.1.1 Fractional Brownian Motion no Perlin Noise (or fractal noise)

Fractional Brownian Motion (fBm) é uma técnica utilizada para gerar texturas e superfícies naturais em computação gráfica, especialmente em simulações de terrenos, nuvens e outros fenômenos naturais. A fBm é uma extensão do conceito de movimento browniano, que descreve o movimento aleatório de partículas em um fluido. Na fBm, esse movimento é caracterizado por uma escala fractal, permitindo a criação de padrões auto-similares em diferentes níveis de detalhe [10].

Para isso, a fBm é construída somando várias camads de ruído Perlin, cada uma com uma frequência e amplitude diferentes. A fórmula geral para calcular a fBm é dada por:

$$fBm(x,y,z) = \sum_{i=0}^{n-1} amplitude_i \cdot Noise(frequency_i \cdot (x,y,z))$$

4.2 Mapeamento de Texturas

Qualquer função que apresente o domínio entre as dimensões pode ser considerado uma "função espacial". A partir disso, cada função espacial pode ser interpretada como a representação de um material sólido.

Desta forma, ao avaliar estas funções nos pontos visíveis da superfície de um objeto, é possível obter a textura da superfície, de modo parecido a ter "contornado" o objeto. A textura obtida a partir deste tipo de extração foi tratada como "textura sólida". Este termo foi usado posteriormente para a descrição da variedade de objetos e explicação de outros conceitos.

5 Volumes (baseado em particulas)

Volumes em geral podem ser descritos de diversas maneiras, dentre elas, campos de particulas, campos de voxels, SDFs, entre outros. Os volumes serão abordados aqui de 2 formas, modelagem de forma e shading, de maneira a se melhor representar volumes como fumaça, fogo, agua, poeira, folhas, etc.

5.1 Modelagem de forma

No geral, ao se computar campos de particulas, espera-se que todas elas tenham seu *lifetime* bem definido para que se possa modelar de maneira precisa o seu comportamento. No entanto, em tempo real, esse trabalho se torna muito custoso, especialmente para volumes que representam fumaca ou folhas ou ate mesmo agua. O movimento desses elementos passa a ser calculado, entao, convertendo as velocidades que circulam o objeto em forcas de corpo. Vemos que poeira simplesmente pode ser modelada sendo carregada pelo vetor de velocidades, sem qualquer resistência significativa. Entretanto, no caso da fumaça, as particulas são substituidas por uma densidade de particulas que aproximam a quantidade delas presentes (normalmente um valor entre 0 e 1). Essa descrição é dada por essa função (conhecida como equação de advecção-difusão):

$$\frac{\partial \rho}{\partial t} = -(u \cdot \nabla)\rho + D\nabla^2\rho + S$$

Onde u é o campo de velocidades, D é o coeficiente de difusão e S é a fonte de particulas. A equação de advecção-difusão pode ser resolvida numericamente usando métodos como diferenças finitas ou volumes finitos. Esses métodos envolvem a discretização do domínio em uma grade e a atualização dos valores da densidade em cada célula da grade ao longo do tempo com base na equação diferencial parcial [14].

Esse cálculo normalmente é atribuido a um grid (matriz) bidimensional ou tridimensional, onde cada célula da grade armazena informações sobre a densidade de particulas, velocidade e outras propriedades relevantes. A resolução da grade afeta diretamente a precisão e o custo computacional da simulação. Grades mais finas proporcionam maior detalhe, mas exigem mais memória e poder de processamento.

5.1.1 Passo de Densidade

O método linear proposto por Stam at al [14] inicia-se com algum estado para a velocidade e densidade e, então, atualiza seus valores baseado em eventos externos (forças, fonte de energia,

fontes de particulas e etc.). A cada passo de tempo, o método segue três etapas principais, passando pela equação de advecção-difusão de maneira "inversa", começando do termo final e indo para o inicial:

- Primeiro termo (Source/Fonte): Adiciona densidade ao sistema baseado em fontes externas. Isso pode incluir a adição de fumaça de uma chaminé ou a introdução de calor em uma área específica.
- Segundo termo (Diffusion/Difusão): Simula a dispersão da densidade ao longo do tempo (se $D > 0$). Uma possível implementação é usar o método de Gauss-Seidel para resolver a equação de difusão, dada por

  $$x_{k+1} = x_k + a\nabla^2 x_{k+1}$$

  onde a é uma constante que depende do coeficiente de difusão e do passo de tempo.
- Terceiro termo (Advection/Advecção): Podemos modelar o centro de cada célula da grade como se fosse uma particula que se move baseada na velocidade do campo. Assim, a densidade é transportada ao longo do campo de velocidades. Entretanto, temos que converter novamente as particulas para a celula proveniente. Uma maneira de fazer isso é usando o método de traçado de linha (backtrace), onde cada célula da grade é atualizada com a densidade da célula de onde a particula veio, baseado na velocidade do campo, ou seja, nos tracamos a linha de volta. Esse método é conhecido como "semi-Lagrangian advection" e é estável para grandes passos de tempo.

O codigo no final vai ter essa cara:

```
void dens_step(int N, float *x, float *x0,
float *u, float *v, float diff, float dt) {
  add_source (N, x, x0, dt);
  SWAP(x0, x); diffuse (N, 0, x, x0, diff, dt);
  SWAP(x0, x); advect (N, 0, x, x0, u, v, dt);
}
```

5.1.2 Passo de Velocidade

O passo de velocidade é similar ao passo de densidade, mas com algumas diferenças importantes. Podemos dizer que a velocidade do campo se altera de 3 maneiras distintas: forças externas, difusão e advecção própria (ou auto-advecção - no qual o campo move a si mesmo). O código desta etapa pode ser escrito como:

```
void vel_step(int N, fl oat *u, float *v,
float *u0, float *v0, float visc, float dt) {
  add_source(N, u, u0, dt); add_source(N, v, v0, dt);
  SWAP(u0, u); diffuse(N, 1, u, u0, visc, dt);
  SWAP(v0, v); diffuse(N, 2, v, v0, visc, dt);
  project(N, u, v, u0, v0);
  SWAP(u0, u); SWAP( v0, v);
  advect(N, 1, u, u0, u0, v0, dt);
  advect(N, 2, v, v0, u0, v0, dt);
  project(N, u, v, u0, v0);
}
```

Note que temos um novo procedimento neste passo chamado `project`, que é responsável por garantir que o campo de velocidades seja incompressível, ou seja, que a divergência do campo seja zero. Isso é importante para simular fluidos realistas, onde a massa é conservada. Este método pode ser implementado resolvendo a equação de Poisson para a pressão e, em seguida, ajustando o campo de velocidades com base no gradiente da pressão.

O procedimento `project` pode ser implementado da seguinte maneira:

```
void project(int N, float *u, float *v, float *p, float *div) {
  int i, j, k;
  float h = 1.0f / N;
  for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
      div[IX(i,j)] = -0.5f*h*(u[IX(i+1,j)]
      - u[IX(i-1,j)] + v[IX(i,j+1)] - v[IX(i,j-1)]);
      p[IX(i,j)] = 0;
    }
  }
  set_boundary(N, 0, div); set_boundary(N, 0, p);

  for (k = 0; k < 20; k++) {
    for (i = 1; i <= N; i++) {
      for (j = 1; j <= N; j++) {
        p[IX(i,j)] = (div[IX(i,j)] + p[IX(i-1,j)]
        + p[IX(i+1,j)] + p[IX(i,j-1)] + p[IX(i,j+1)]) / 4;
      }
    }
    set_boundary(N, 0, p);
  }

  for (i = 1; i <= N; i++) {
    for (j = 1; j <= N; j++) {
      u[IX(i,j)] -= 0.5f*(p[IX(i+1,j)] - p[IX(i-1,j)])/h;
      v[IX(i,j)] -= 0.5f*(p[IX(i,j+1)] - p[IX(i,j-1)])/h;
    }
  }
  set_boundary(N, 1, u); set_boundary(N, 2, v);
}
```

5.2 Conclusão da modelagem de forma

Podemos concluir essa seção com esse código maravilhoso que junta tudo:

```
void simulate(int N, float *u, float *v, float *u0, float *v0,
float *dens, float *dens0, float diff, float visc, float dt) {
  get_inputs(N, u0, v0, dens0);
  vel_step(N, u, v, u0, v0, visc, dt);
  dens_step(N, dens, dens0, u, v, diff, dt);
  draw_dens(N, dens);
}
```

a rotina `draw_dens` será abordada na seção de shading.

5.3 Shading

Volumes são tradicionalmente renderizados usando técnicas de ray marching, onde um raio é lançado através do volume e amostras são coletadas ao longo do caminho do raio. Essas amostras são então usadas para calcular a cor e a opacidade do pixel correspondente na imagem final. A equação de rendering para volumes pode ser expressa como:

$$C = \int_{t_{near}}^{t_{far}} T(t)\sigma(t)L(t)dt$$

onde C é a cor final do pixel, $T(t)$ é a transmissão ao longo do caminho do raio, $\sigma(t)$ é a densidade de absorção no ponto t, e $L(t)$ é a luz incidente no ponto t. A transmissão $T(t)$ é dada por:

$$T(t) = e^{-\int_{t_{near}}^{t} \sigma(s)ds}$$

Existem diversas técnicas para otimizar o ray marching, como o uso de grids adaptativos, onde a resolução do grid é maior em áreas de interesse e menor em áreas menos importantes. Outra técnica é o uso de pré-computação de iluminação, onde a iluminação é calculada previamente e armazenada em uma textura 3D, que pode ser rapidamente acessada durante o ray marching. Além disso, técnicas de denoising podem ser aplicadas para reduzir o ruído nas imagens renderizadas, especialmente em simulações de volumes complexos como fumaça e fogo, no entanto, não serão abordadas aqui.

5.3.1 Interação da luz com volumes

A luz pode ser absorvida, emitida ou espalhada ao interagir com os volumes gerados até aqui, sendo cada um dos fenômenos descritos por uma equação específica:

- Absorção: A luz é absorvida pelo meio, reduzindo sua intensidade. Isso é descrito pela lei de Beer-Lambert:

  $$I(t) = I_0 e^{-\int_0^t \sigma_a(s)ds}$$

  onde $I(t)$ é a intensidade da luz no ponto t, I_0 é a intensidade inicial, e $\sigma_a(s)$ é o coeficiente de absorção no ponto s.

- Emissão: O meio pode emitir luz, contribuindo para a intensidade total. A contribuição da emissão pode ser modelada como:

  $$L_{emit}(t) = \int_0^t j(s)e^{-\int_s^t \sigma_a(u)du}ds$$

  onde $j(s)$ é a taxa de emissão no ponto s.

- Espalhamento: A luz pode ser espalhada em diferentes direções ao interagir com o meio. O espalhamento pode ser descrito pela função de fase $p(\omega', \omega)$, que define a probabilidade de um raio de luz ser espalhado da direção ω' para a direção ω. A contribuição do espalhamento pode ser modelada como:

  $$L_{scatter}(t) = \int_0^t \sigma_s(s) \int_{S^2} p(\omega', \omega)L(s, \omega')d\omega' e^{-\int_s^t \sigma_a(u)du}ds$$

  onde $\sigma_s(s)$ é o coeficiente de espalhamento no ponto s, e $L(s, \omega')$ é a radiância incidente na direção ω' no ponto s [8].

Todas essas técnicas se acumulam na equação de rendering volumétrico expressa no inicio da seção 5.3, que pode ser resolvida numericamente usando métodos como ray marching ou path tracing. A implementação dessas técnicas pode variar dependendo do contexto específico e dos requisitos de desempenho e qualidade visual, mas o objetivo geral é capturar a complexidade da interação da luz com volumes de maneira eficiente e realista.

5.3.2 Implementação básica de ray marching

A implementação básica de ray marching envolve o lançamento de um raio através do volume e a amostragem da densidade e outras propriedades do volume ao longo do caminho do raio. O código a seguir ilustra uma implementação simples de ray marching para volumes:

```
vec4 ray_march(vec3 ray_origin, vec3 ray_direction,
float t_near, float t_far) {
  vec4 color = vec4(0.0);
  float t = t_near;
  float step_size = 0.1;
  while (t < t_far) {
    vec3 sample_position = ray_origin + t * ray_direction;
    float density = sample_density(sample_position);
    vec4 sample_color = compute_color(density);
    color += sample_color * exp(-density * step_size);
    t += step_size;
  }
  return color;
}
```

Este código define uma função ‘ray_march‘ que toma a origem e a direção do raio, bem como os limites próximo e distante do volume. A função itera ao longo do caminho do raio, amostrando a densidade do volume em cada passo e acumulando a cor resultante, levando em consideração a absorção da luz. A função ‘sample_density‘ é responsável por retornar a densidade do volume em uma posição específica, enquanto a função ‘compute_color‘ calcula a cor com base na densidade amostrada. A variável ‘step_size‘ controla a distância entre as amostras ao longo do raio, e pode ser ajustada para equilibrar a qualidade visual e o desempenho. Note que esta é uma implementação simplificada e pode ser expandida para incluir efeitos adicionais, como emissão e espalhamento, conforme discutido anteriormente [11].

6 Justificativa

Justificativa

7 Objetivos

obj

8 Metodologia

Metodologia

9 Resultados

Resultados

10 Conclusão

Concluido

Referências

[1] Jules Bloomenthal. *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.

[2] Mark Carlson, Peter J. Mucha, Robert Van Horn, and Dimitris Metaxas. Melting and flowing. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2002.

[3] Mathieu Desbrun and Marie-Paule Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Eurographics Workshop on Computer Animation and Simulation*, 1996.

[4] George Dyson. Turing's cathedral: The origins of the digital universe. *Nature*, 482(7386):461–462, 2012.

[5] Doug Enright, Ronald Fedkiw, Joel Ferziger, and Ian Mitchell. Animation and rendering of complex water surfaces. *ACM Transactions on Graphics*, 21(3):736–744, 2002.

[6] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH 2001*, 2001.

[7] Nick Foster and Dimitris Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing*, 58(5):471–483, 1996.

[8] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, January 1984.

[9] Matthias Müller, Julie Dorsey, Leonard McMillan, Robert Jagnow, and Barbara Cutler. Stable real-time deformations. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2002.

[10] Ken Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296, 1985.

[11] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 3rd edition, 2016.

[12] William T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.

[13] Jos Stam. Stable fluids. In *SIGGRAPH 99*, 1999.

[14] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003. CDROM/PDF.

[15] Dan Tonnesen. *Dynamically coupled particle systems for geometric modeling, reconstruction, and motion simulation*. PhD thesis, University of Copenhagen, 1998.

[16] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *SIGGRAPH '91*, 1991.