Caracterização de Propriedades Materiais Através da Análise do Movimento Browniano

Supervisor: Prof. Dr. Marcel Parolin Jackowski
Aluno: Luiz Fernando Oliveira Corte Real[1]

01 de dezembro de 2008

[1]Bolsista do CNPq - Brasil

Sumário

1 Introdução

Movimento browniano é o tipo de movimento realizado por uma partícula imersa num fluido, desconsiderando-se as correntes deste. É composto de translações e rotações, e seu caminho, teoricamente, não tem tangente em nenhum ponto; a cada instante de tempo, a direção de deslocamento muda.

A base teórica do movimento é principalmente devida a um artigo de Einstein, publicado em 1905 [1]. Nele, o cientista previu o movimento de partículas suspensas em um líquido baseado na teoria cinética molecular e, utilizando mecânica estatística, desenvolveu uma fórmula para o coeficiente de difusão das partículas em movimento browniano. Com isso, relacionou o movimento browniano com propriedades materiais do fluido onde ele ocorre.

Atualmente, a ressonância magnética (RM) de difusão [2] possibilita a determinação do coeficiente de difusão de materiais, mas não suas propriedades físicas. Com base no coeficiente de difusão e com dados da simulação do movimento browniano em um certo meio, é possível estimar as propriedades físicas do material em análise na ressonância de forma não-invasiva.

Entretanto, para simular o movimento browniano em situações diversas, é necessário poder simulá-lo para muitas moléculas ao mesmo tempo e em meios arbitrários. Para tanto, é necessário aplicar um algoritmo de detecção de colisão ao mesmo tempo preciso e eficiente. E se o objetivo é mostrar esta simulação ocorrendo em tempo real, a renderização da cena tridimensional de forma eficiente torna-se um fator crítico.

2 Materiais e métodos

A parte teórica da simulação envolve os conceitos de movimento browniano, difusão (intimamente ligado ao anterior), energia cinética, conservação de momento e mecânica estatística.

Para entender melhor o que é o movimento browniano, artigos e livros fundamentais a esse respeito foram estudados, dentre os quais podem ser citados [3], [4], [5] e [1]. Este último também contribuiu para o entendimento do conceito de difusão e foi fundamental para o trabalho de relacionar o movimento browniano com o coeficiente de difusão, juntamente com [6].

2.1 Equações de difusão

Partindo da simulação da movimentação molecular, pode-se utilizar a primeira lei de Fick e a equação para o deslocamento quadrático médio de Einstein para estimar o coeficiente de difusão numa certa região.

A primeira lei de Fick relaciona o fluxo de partículas J com o gradiente molar de concentração $\nabla\phi$:

$$J = -D\nabla\phi \tag{1}$$

onde D representa o coeficiente de difusão.

Já a equação de Einstein relaciona o deslocamento quadrático médio $\langle x^2 \rangle$ com o coeficiente de difusão D e o tempo t decorrido desde o estado inicial do sistema:

$$\langle x^2 \rangle = 6Dt \tag{2}$$

Com base nos dados obtidos e no coeficiente de difusão calculado, é possível aplicar a lei de Stokes-Einstein para derivar a viscosidade do material. Esta lei, juntamente com a primeira lei de Fick, permite derivar a seguinte relação entre o coeficiente de difusão e a viscosidade η do meio:

$$D = \frac{kT}{6\pi\eta a} \tag{3}$$

onde T é a temperatura e k é a constante de Boltzmann.

2.2 Distribuição de Maxwell

Ao longo do desenvolvimento do projeto foi abordado um problema não previsto: como gerar um estado inicial compatível com a realidade? Aqui, foi utilizada a distribuição de Maxwell [7] para estimar um módulo de velocidade inicial para cada molécula com base na temperatura fornecida.

A distribuição de Maxwell é dada por:

$$n = 4\pi N e^{\frac{-mv^2}{2kT}} v^2 \left(\frac{m}{2\pi kT}\right)^{1.5} \tag{4}$$

onde n é o número de moléculas esperado para a temperatura T numa velocidade v, N é o número total de moléculas do meio considerado, m é a massa de cada partícula e k é a constante de Boltzmann.

No programa, está sendo utilizada uma versão ligeiramente modificada da equação para obter-se a proporção esperada de moléculas a uma determinada velocidade:

$$p = 4\pi e^{\frac{-mv^2}{2kT}} v^2 \left(\frac{m}{2\pi kT}\right)^{1.5} \tag{5}$$

Como a distribuição de Maxwell fornece o número de moléculas numa determinada velocidade, para estimar a velocidade de uma molécula é necessário integrar esta função e, em seguida, calcular sua inversa [8].

Como esta função não admite uma inversa explícita, é necessário fazer tanto a inversão como a integração de modo numérico. Isso é feito por meio do seguinte algoritmo:

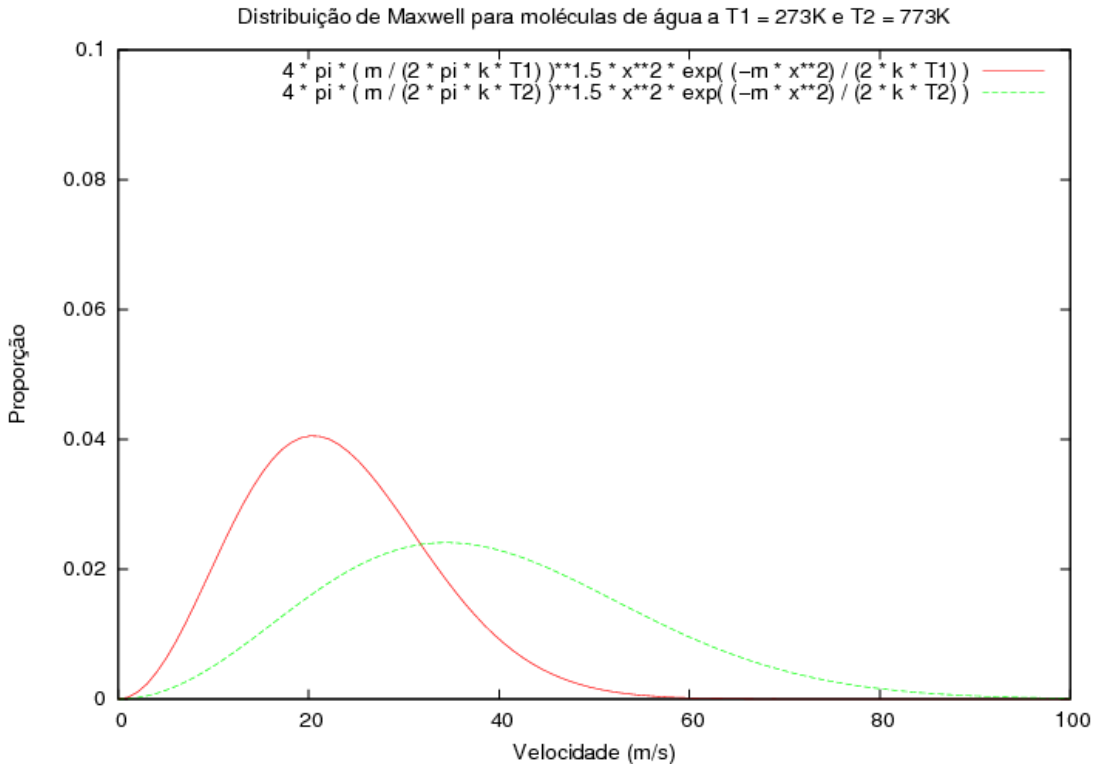

Figura 1: Gráfico da distribuição de Maxwell para duas temperaturas diferentes

1. Gere um número pseudo-aleatório: $p = Unif([0, 1))$

2. Calcule os valores independentes da velocidade na função:

$$c_1 = 4\pi \left(\frac{m}{2\pi kT}\right)^{1.5}$$
$$c_2 = \frac{-m}{2kT}$$

3. Sejam $s = 0$ e $v = 0$. Enquanto $s \leq p$ faça:

   (a) $s = s + [c_1 \cdot x^2 \cdot \exp(c_2 \cdot x^2)] \cdot \delta$

   (b) $v = v + \delta$

δ determina a qualidade da aproximação da integração, e está definido no programa como 0.1.

2.3 Tecnologias utilizadas

Foi decidido, inicialmente, que o *software* para simular o movimento browniano deveria ser desenvolvido em C++, no ambiente de desenvolvimento Microsoft Visual C++ Express 2005 e utilizando as bibliotecas GTK [9], para o desenvolvimento da interface gráfica, e VTK [10], para a renderização da simulação em três dimensões.

Tanto a linguagem como o ambiente de desenvolvimento e as bibliotecas não eram bem conhecidas; foi necessário estudá-las e desenvolver programas de teste utilizando-as, o que foi feito logo de início, em conjunto com os estudos sobre o movimento browniano.

Outra ferramenta que auxiliou bastante no processo de gerenciamento de dependências e compilação do *software* foi o CMake [11], cujo objetivo é, dado um conjunto de arquivos de código e um conjunto de dependências, gerar automaticamente um arquivo (ou um conjunto de arquivos) que possa(m) ser utilizado(s) para compilar a aplicação de forma automática.

Dado que C++ é uma linguagem com um bom suporte a orientação a objetos, foi decidido que este paradigma deveria ser utilizado. Para que pudesse ser melhor aplicado, foi necessário utilizar uma biblioteca auxiliar [12] para o GTK – a biblioteca GTKmm –, cujo objetivo é tornar a interface desta última mais orientada a objetos.

Ainda com relação a orientação a objetos, foram aplicados, sempre que possível, padrões de projeto [13], a fim de tornar a aplicação mais flexível e fácil de entender para outros desenvolvedores. Dentre os padrões aplicados, podem ser citados o *Abstract Factory*, o *Singleton* e o *Observer*.

Um ponto bastante difícil era integrar as bibliotecas GTK e VTK na mesma aplicação. Após uma longa busca na Internet, foi encontrada a biblioteca vtkmm [14], cujo objetivo é fornecer uma área de desenho para o VTK dentro de um componente de interface GTK. Mesmo assim, ainda é necessário, até o momento, inserir programaticamente o componente VTK na interface, uma vez que não integração desta biblioteca com o desenvolvedor visual de interfaces Glade, que vem com a biblioteca GTK. Também foi necessário fazer algumas alterações na biblioteca para que ela compilasse tanto em Windows como em Linux e para que utilizasse a ferramenta CMake para compilação, para que fosse possível integrar a construção da biblioteca com a construção do projeto como um todo.

Pouco depois do início do projeto, notou-se a importância da geração de números pseudo-aleatórios. Para facilitar essa geração e garantir uma dispersão boa dos números gerados, utilizou-se uma biblioteca de geração de números aleatórios baseada no algoritmo *Mersenne Twister* [15].

Quando a aplicação já estava mais complexa, notou-se a necessidade de uma biblioteca de simulação de mecânica e dinâmica clássicas, com detecção e tratamento de colisões, para que as moléculas do simulador colidissem entre si e com o meio de difusão. Implementar uma nova biblioteca para este simulador seria muito trabalhoso, sujeito a erros e não era o objetivo principal deste trabalho; foi introduzida, então, a biblioteca ODE [16], cujo objetivo principal é a simulação da dinâmica newtoniana de corpos rígidos.

Com o objetivo de melhorar a qualidade do código de testes do programa, também foi introduzida, posteriormente, a biblioteca CppUnit [17], cujo objetivo é prover classes e métodos para facilitar a escrita, a execução e o aproveitamento do código de testes.

2.4 Atividades realizadas

O *software* foi sendo desenvolvido juntamente com a pesquisa sobre o tema. As funcionalidades foram pedidas aos poucos, de forma incremental e de acordo com o progresso da pesquisa, pelo prof. Marcel. Tendo em vista as modificações constantes no código, procurou-se implementar o programa aplicando alguns aspectos de XP[1]:

- **Testes a priori:** foram escritos testes para o código antes de escrevê-lo, sempre que possível.
- **Projeto incremental:** a arquitetura do programa foi pouco planejada, deixando os detalhes para serem pensados quando a funcionali-

[1] eXtreme Programming

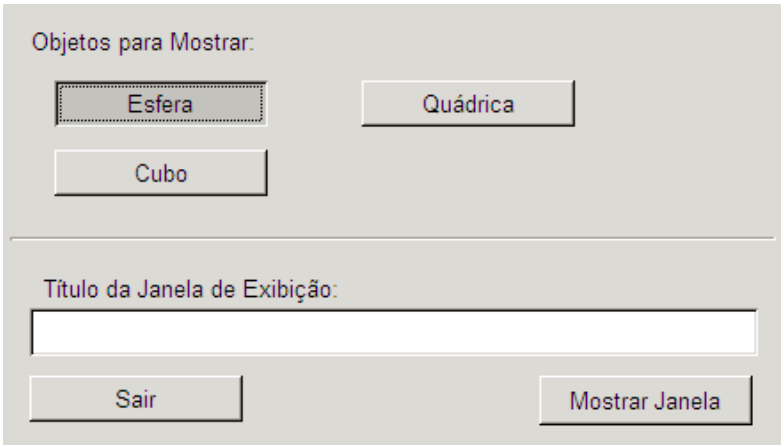

Figura 2: Tela única da primeira versão do programa

dade que os exigisse fosse implementada.

Tomou-se bastante cuidado, ao longo do desenvolvimento do *software*, para que este fosse independente da plataforma de execução, aproveitando, assim, os benefícios de utilizar bibliotecas para a abstração do desenho da interface e para a renderização de imagens tridimensionais.

2.4.1 Versões de teste

Foi desenvolvida uma versão inicial do programa simulador de movimento browniano utilizando-se o programa Glade, da biblioteca GTK, para desenhar a interface, e o programa `glade--`, para gerar o código em C++ para a interface desenhada. O programa Glade gera, a partir da interface desenhada, um arquivo no formato XML, contendo as especificações de cada componente (*widget*) da interface. Passa-se esse arquivo para o programa `glade--` para que este gere o código em C++.

A versão inicial do programa contava com uma interface simples, como mostra a Figura 2. O objetivo desta versão era tomar contato com as bibliotecas a serem utilizadas, mostrando objetos tridimensionais na tela e manipulando os *widgets* da interface.

Partindo desta versão, incrementou-se a interface, como se pode ver na Figura 3. Aqui, o objetivo era manipular os objetos criados, alterando suas posições e animando-os. Como já havia um pouco mais de complexidade no programa, com a introdução de um *parser* para as caixas de texto com as coordenadas dos objetos, introduziram-se testes unitários no programa.

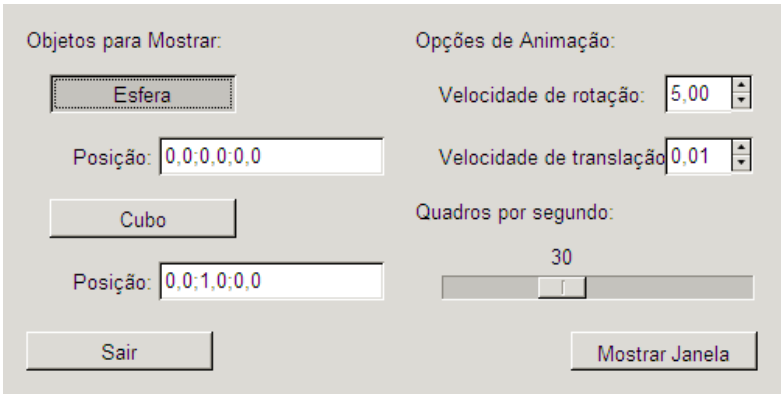

Figura 3: Tela única da segunda versão do programa, com suporte para o posicionamento dos objetos a serem desenhados

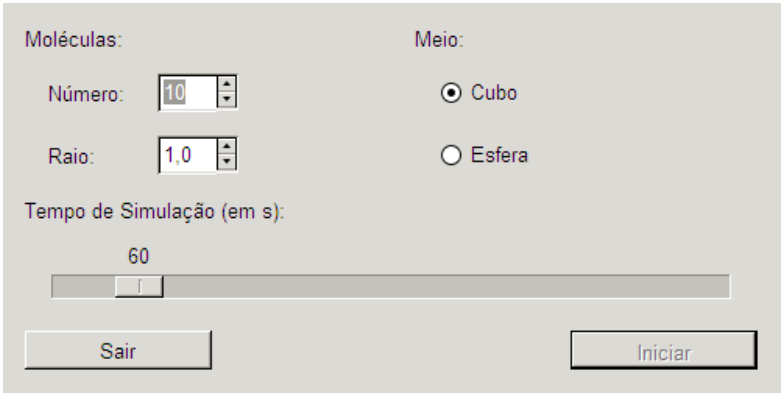

Figura 4: Interface do programa depois da implementação da simulação básica do movimento browniano, com as moléculas colidindo apenas com o ambiente de difusão

2.4.2 Primeira simulação

Depois de apresentar esta segunda versão, foi pedido que se mostrassem esferas movimentando-se dentro de um cubo ou de uma esfera maior – o primeiro passo para realizar a simulação do movimento browniano. O código cresceu bastante, e foi necessário introduzir alguma padronização nele (nomes de métodos e identação), para facilitar sua manutenção e seu desenvolvimento.

Foi pedido para que as esferas colidissem, inicialmente, apenas com o cubo (ou esfera) – o ambiente de difusão. Isso exigiu a implementação de detecção de colisão simples, onde apenas um dos objetos está em movimento e o ponto de colisão, juntamente com a normal deste ponto, pode ser facilmente calculada.

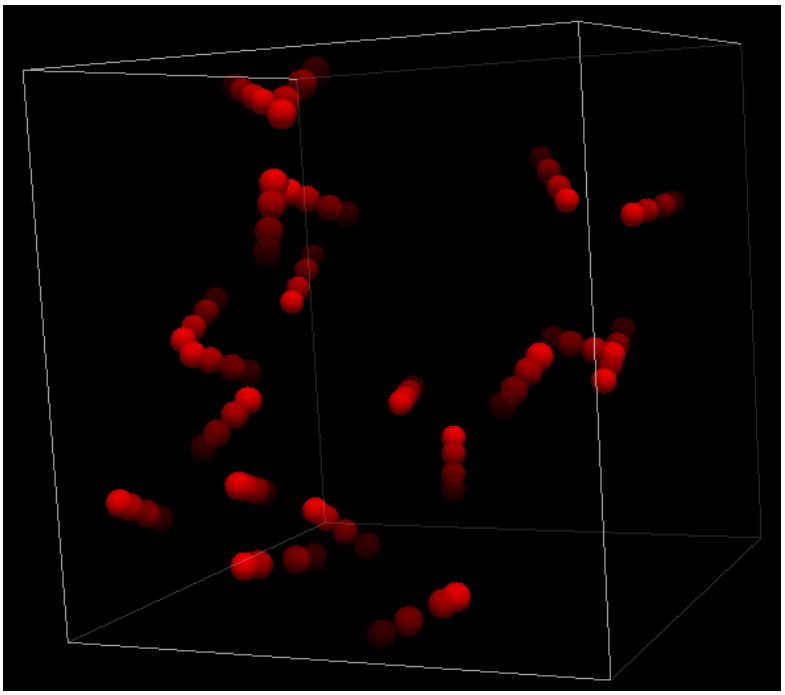

Figura 5: Sucessivos quadros da simulação básica do movimento browniano dentro de um cubo sobrepostos

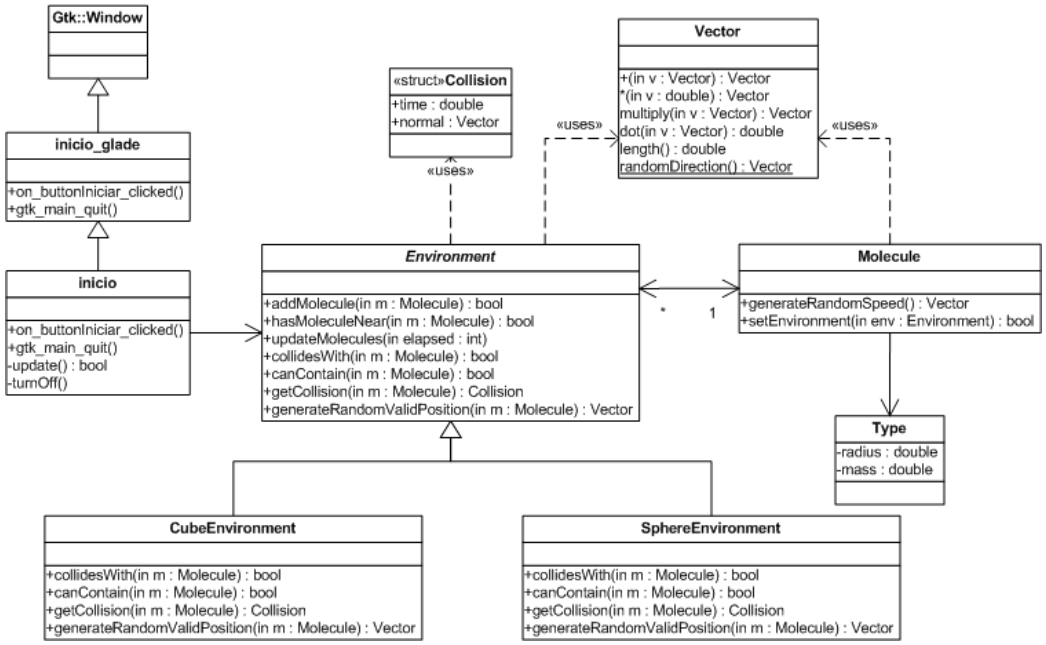

Figura 6: Diagrama de classes da última versão do programa a utilizar o `glade--` como gerador de código para a interface

O resultado destas alterações pode ser visto na Figura 4. O resultado da simulação pode ser visto na Figura 5.

Neste ponto, o paradigma de orientação a objetos começou a conflitar com o modo como o `glade--` gerava o código. Haviam classes bem definidas para as "moléculas" e para o ambiente de difusão, mas estas estavam razoavelmente acopladas a uma classe pouco coesa e com padrões de código diferentes, que era a classe da interface. O diagrama de classes do programa pode ser visto na Figura 6. Olhando para o diagrama, é possível notar que, além dos padrões de nomes de classes misturados, a classe `inicio`, cuja responsabilidade inicial era apenas desenhar a interface, estava ficando pouco coesa, tendo, também, a responsabilidade de criar os objetos da simulação e atualizá-los a cada quadro. Esses problemas foram determinantes para a decisão de não utilizar mais o gerador de código.

Outros fatores decisivos para tomar a decisão de não utilizar mais o gerador de código foram a falta de flexibilidade para mudanças na interface, uma vez que cada mudança exigia a mesclagem do código anterior com o recém-gerado, a recomendação de não utilizar o gerador feita pelos próprios desenvolvedores do GTK e a dificuldade de utilizar as ferramentas de auxílio à compilação determinadas pelo `glade--` no Windows.

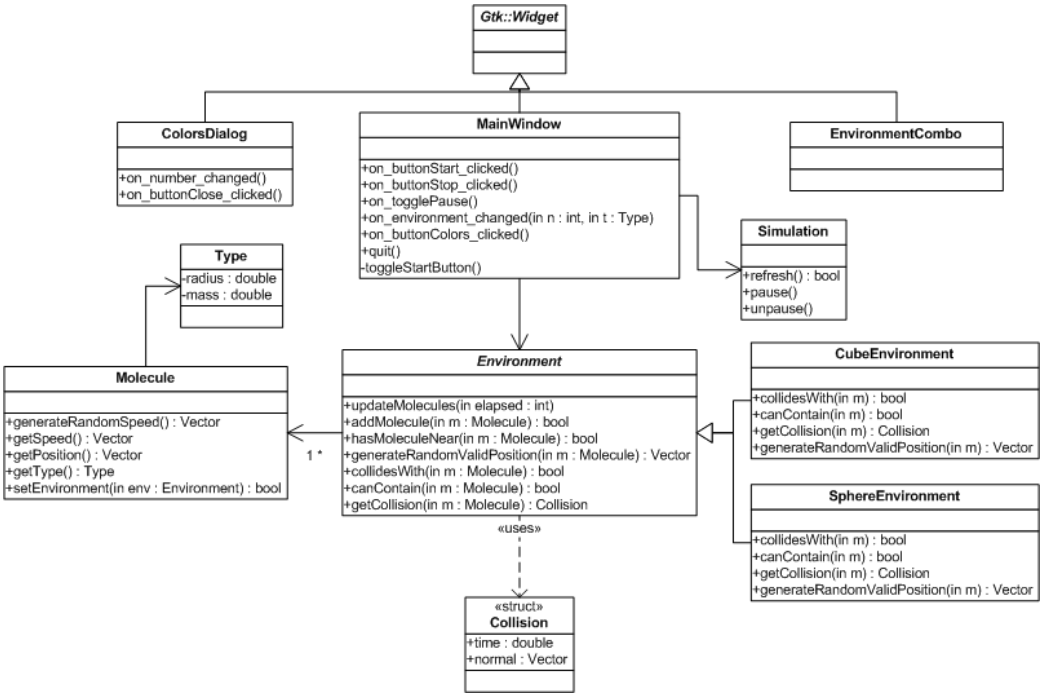

Figura 7: Diagrama de classes após as alterações para utilizar o carregamento dinâmico da interface

2.4.3 Primeira refatoração: carregamento dinâmico da interface

O programa foi alterado, então, para utilizar o carregamento dinâmico da especificação da interface, provido pela biblioteca GTKmm. Ela provê métodos de criar a interface dinamicamente a partir de sua especificação no formato utilizado pelo Glade. Além disso, para adicionar funcionalidade à interface, basta herdar a classe correspondente ao tipo de *widget* que terá novas funcionalidades e fazer com que a biblioteca use sua classe para carregar o componente especificado no arquivo do Glade.

Inicialmente, foram criadas uma classe para cada janela do programa e mais uma classe para um componente complexo da janela principal, que determina qual o tipo de ambiente de difusão que será utilizado. O diagrama de classes após as alterações pode ser visto na Figura 7.

A interface do programa também foi alterada nesta refatoração. Foi pedido que a simulação fosse exibida na mesma janela onde estavam os controles dos seus parâmetros. A interface modificada pode ser vista na Figura 8.

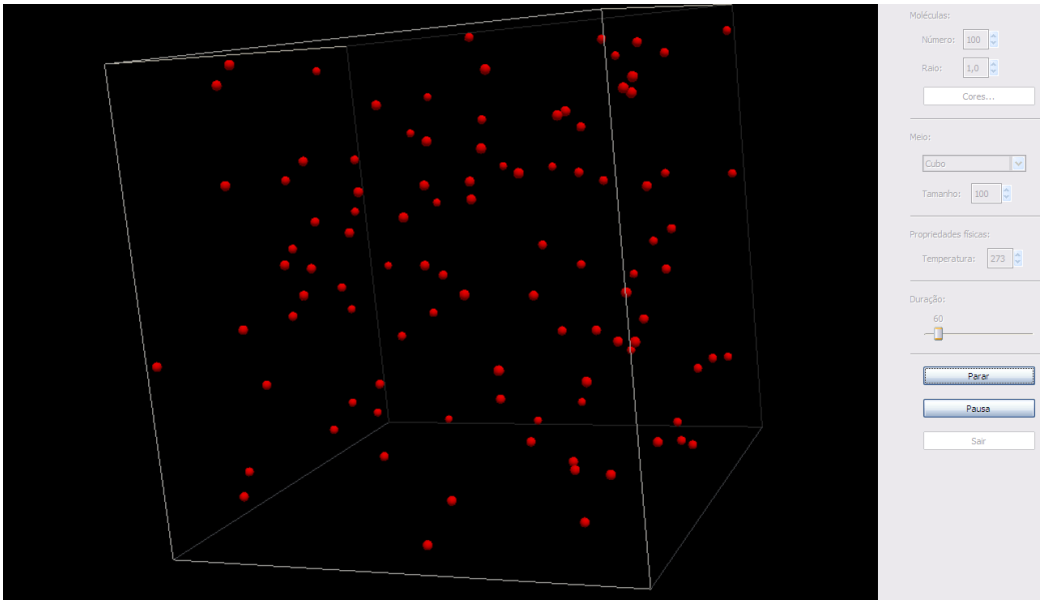

Figura 8: Interface do programa modificada para exibir a simulação ao lado dos controles dos seus parâmetros

2.4.4 Inclusão de novas bibliotecas

O próximo passo foi introduzir as bibliotecas ODE e CppUnit. Com isso, as classes `Environment` (e suas subclasses) e `Molecule` perderam o comportamento físico, passando apenas a adaptadoras entre as diversas bibliotecas utilizadas. As subclasses de `Environment` passaram a ter o papel de inserir no ambiente físico gerenciado pela ODE os planos de colisão necessários para modelar a forma do ambiente de difusão desejada.

A integração com a biblioteca CppUnit não trouxe muitos problemas. Entretanto, a biblioteca ODE exigiu o ajuste de alguns parâmetros referentes à conservação de energia, uma vez que, mesmo definindo o coeficiente de elasticidade das colisões para 1.0, ou seja, definindo que as colisões deveriam ser perfeitamente elásticas, havia perda de energia no sistema. Foi necessário definir um coeficiente de elasticidade maior do que 1.0 para que houvesse conservação de energia, excetuando-se erros de arredondamento menores do que 10^{-3} unidades de energia.

Outro problema relacionado à utilização da biblioteca ODE foi a necessidade de modelar os ambientes de difusão como intersecção de planos. Até a introdução desta biblioteca, a borda do ambiente esférico era calculada com exatidão, por meio da equação implícita da esfera: $\sqrt{x^2 + y^2 + z^2} = r$, onde r é o raio da esfera. Com a introdução da biblioteca, foi necessário utilizar a

seguinte parametrização:

$$\begin{cases} x(u,v) = r\cos(u)\cos(v) \\ y(u,v) = r\cos(u)\sin(v) \\ z(u,v) = r\sin(u) \\ u \in [\frac{-\pi}{2}, \frac{\pi}{2}) \\ v \in [0, 2\pi) \end{cases} \tag{6}$$

A partir desta parametrização, é possível calcular o vetor normal à esfera em um ponto qualquer de sua superfície. Com o vetor normal, define-se, então, um plano. Porém, como o número de planos necessários para representar uma esfera com perfeição é infinito e não é possível inserir infinitos planos na memória do computador, faz-se necessário aproximar a esfera por um número finito de planos. Para tanto, foi utilizado o seguinte algoritmo:

1. Sejam $\delta\phi = \frac{2\pi}{\Phi}$, $\delta\theta = \frac{2\pi}{\Theta}$
2. Para ϕ de 0 a π passo $\delta\phi$
   (a) Sejam $C_\phi = \cos(\phi)$, $c = \sin(\phi)$
   (b) Para θ de 0 a 2π passo $\delta\theta$
      i. Sejam $a = C_\phi\cos(\theta)$, $b = C_\phi\sin(\theta)$
      ii. Seja $v = (-a, -b, -c)$
      iii. Normalize v
      iv. Adicione no espaço o plano com a equação $v_x x + v_y y + v_z z - r = 0$, onde r é o raio da esfera

Φ e Θ determinam o número de planos que serão utilizados para aproximar a esfera. Quanto maiores seus valores, mais planos são utilizados.

Feita a integração com as bibliotecas, o foco foi melhorar a exibição da simulação. Neste ponto, foram introduzidas, gradualmente, diversas funcionalidades. A primeira delas foi permitir que o usuário escolhesse as cores das moléculas na tela.

2.4.5 Padronização das unidades

Já visando a coleta posterior de dados da simulação, foram determinadas as unidades dos parâmetros físicos da simulação. Foram adotadas as seguintes unidades:

- Massa: yoctogramas ($1\text{yg} = 10^{-24}\text{g}$)

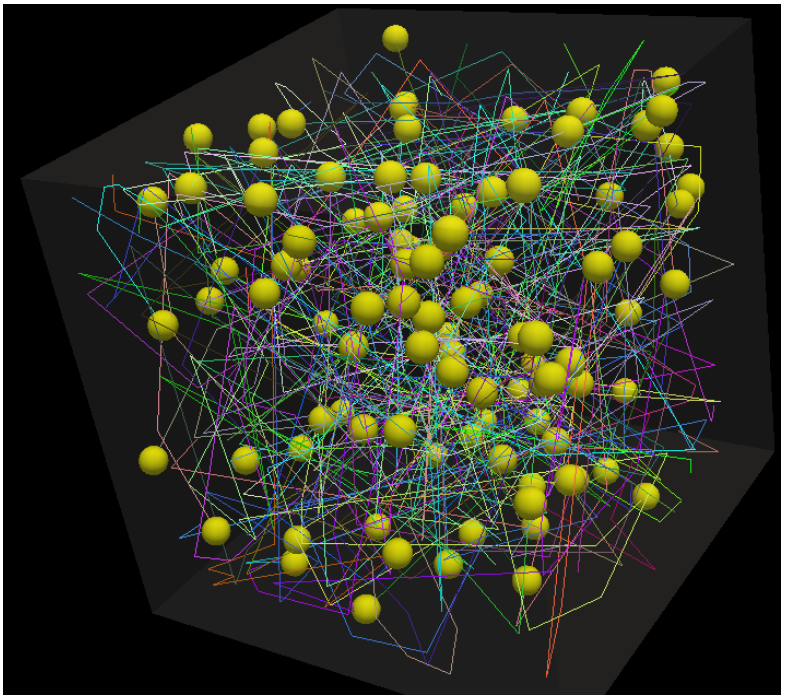

Figura 9: Exibição da simulação após a alteração do modo de exibição do ambiente de difusão e da inclusão da trajetória de difusão das moléculas

- Espaço: angstroms ($1\text{Å} = 10^{-10}\text{m}$)
- Tempo: nanossegundos ($1\text{ns} = 10^{-9}\text{s}$)

As unidades foram escolhidas de modo que o valor dos parâmetros ficasse próximo de 1.0, a fim de minimizar os erros de aproximação decorrentes do uso de números de ponto flutuante.

2.4.6 Desenho de trajetórias

Outra funcionalidade que foi pedida a fim de melhorar a visualização da simulação foi o desenho das trajetórias descritas pelas moléculas em difusão. Para isso, foi necessário criar uma nova classe – `Path` –, cujo objetivo é gerenciar os pontos das retas que compõem a trajetória. Também foi feita uma pequena alteração no modo como o ambiente de difusão é desenhado na tela: o modo de exibição *wireframe* foi trocado pelo modo de exibição translúcido, facilitando a percepção de tridimensionalidade do ambiente. O resultado dessas duas alterações pode ser visto na Figura 9.

Inicialmente, o desenho da trajetória foi feito de modo bastante ineficiente, com a alocação de cada polilinha ocorrendo a cada atualização.

Esta implementação foi necessária devido ao modo de operação da classe `vtkPolyline`, da biblioteca VTK; esta classe não era otimizada para desenhar uma polilinha dinâmica, o que era necessário para representar a trajetória de uma molécula em movimento. Pesquisou-se, então, como utilizar outras classes da biblioteca para chegar a esse resultado, e acabou-se por utilizar a classe `vtkPolyData` em conjunto com a classe `vtkPoints`. Esta última permite a adição e a alteração de um conjunto de pontos, enquanto a primeira define uma topologia sobre os pontos, no caso um conjunto de linhas.

2.4.7 Implementação da distribuição de Maxwell

Com o intuito de aproximar melhor a simulação da teoria mais recente sobre difusão, o modo como o módulo das velocidades iniciais das moléculas é gerado foi alterado. Inicialmente, calculava-se um módulo de velocidade com base na equação $mv^2 = 3kT$, onde m é a massa de uma molécula, v, sua velocidade, k, a constante de Boltzmann, e T, a temperatura ambiente. Porém, esta equação determina apenas a velocidade média de cada molécula num gás. O modelo cinético do gás prevê que as moléculas sigam uma distribuição de velocidade, conhecida como "distribuição de velocidades de Maxwell". Isso decorre da equipartição de energia quando uma colisão entre duas moléculas ocorre.

Aqui há dois pontos importantes a serem observados com relação à simulação. O primeiro é que está sendo assumido que a substância em difusão é um gás perfeito, que segue o modelo cinético. O segundo é que está se assumindo que as colisões que envolvem as moléculas são perfeitamente elásticas, ou seja, não há transferência de energia entre as moléculas e as paredes do meio de difusão, o que representaria um resfriamento no ambiente de difusão, e não há perda de energia na colisão entre duas moléculas.

2.4.8 Refatoração da classe `MainWindow`

É possível notar, na Figura 7, que, mesmo após a alteração do programa para utilizar o carregamento dinâmico de interface, a classe `MainWindow` ficou com muitas responsabilidades. Visando a melhorar a arquitetura do *software*, foi feita uma refatoração do código dessa classe, com a criação de classes específicas para determinados controles da interface e distribuição de responsabilidades entre as já existentes. Também foram alterados os nomes de diversas classes e métodos, padronizando definitivamente o código, e alguns métodos foram divididos em métodos menores, facilitando seu entendimento e manutenção. O resultado dessa refatoração pode ser visto na

Figura 10.

2.4.9 Visibilidade das colisões aumentada

Após a refatoração, passou-se à implementação de mais melhorias de visualização da simulação. O passo seguinte foi implementar a mudança de cor das moléculas quando duas moléculas colidissem. Foi adicionada a possibilidade de o usuário escolher qual cor a molécula adquiriria quando colidisse, bem como a velocidade de transição para a cor normal. Quando uma molécula colide, atribui-se-lhe a cor de colisão e, em seguida, a cada atualização, muda-se essa cor seguindo a fórmula

$$c_i = c_{i-1} + (c_n - c_{i-1}) * f \tag{7}$$

onde c_{i-1} representa um canal de cor (dentre vermelho, verde e azul) da cor atual, c_i, um canal de cor da nova cor, c_n, um canal de cor da cor final (a cor normal) da molécula e f, o fator de velocidade de transição da cor. Assim, a cor da molécula começa mudando rapidamente logo após a colisão, numa velocidade que vai decaindo conforme a cor vai se normalizando. De acordo com o fator velocidade escolhido, é possível ter uma noção de quantas colisões estão ocorrendo e, portanto, da difusibilidade do material no ambiente escolhido.

2.4.10 Coleta das primeiras estatísticas

Após mais uma melhoria de visualização, o foco passou ao ponto mais importante do programa: a coleta de estatísticas. A primeira estatística coletada, de ordem unicamente computacional, deveria ser o número de quadros por segundo da animação. Foi escolhida esta estatística pela facilidade de implementação da sua coleta. Para dar mais flexibilidade ao usuário, também foi adicionada, na interface, a opção de mostrar ou não o número de quadros por segundo.

O cálculo do número de quadros por segundo é, na verdade, uma estimação baseada no intervalo entre a última atualização e a atualização corrente. Seu resultado é atualizado a cada quadro e impresso na tela da simulação, como mostra a Figura 11.

Depois do número de quadros por segundo, passou-se à coleta de dados da simulação. A primeira estatística coletada foi o número de colisões por segundo. Para implementá-la, foram necessárias algumas refatorações no código.

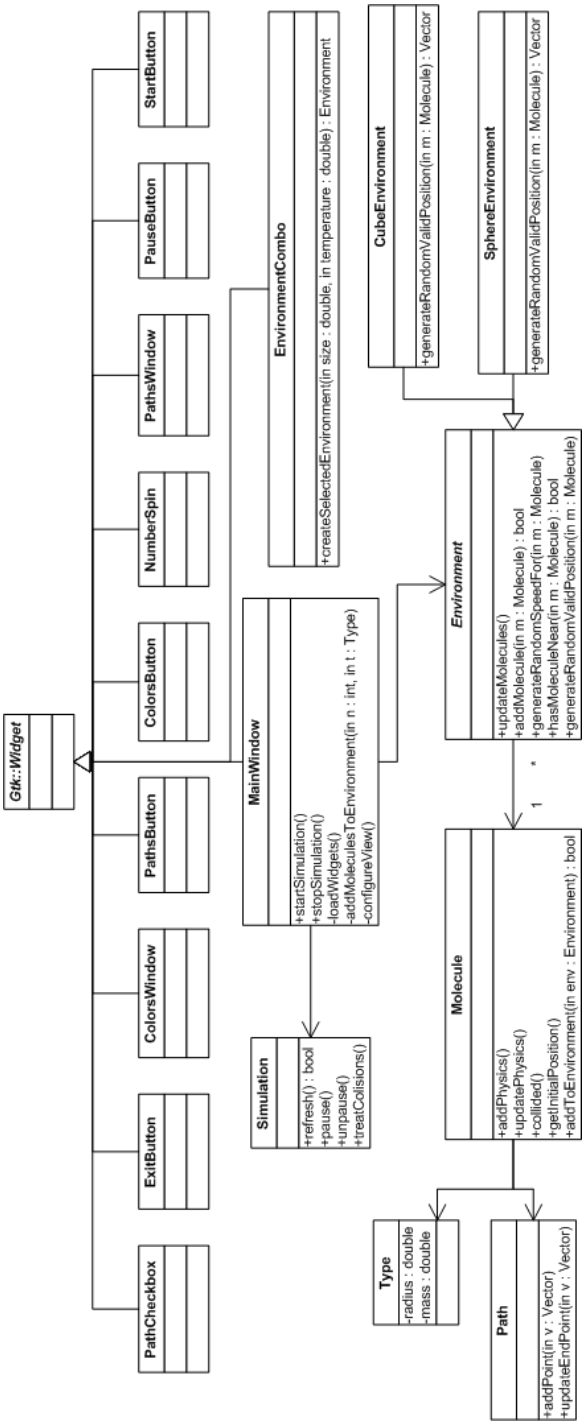

Figura 10: Diagrama de classes do sistema após a refatoração da classe `MainWindow`

Figura 11: Cálculo do número de quadros por segundo sendo mostrado na tela da simulação

2.4.11 Refatoração para coleta de estatísticas e alteração da interface

Juntamente com a coleta de estatísticas, foi pedido que se reformulasse a interface do programa, que já estava ficando com muitos controles devido às novas funcionalidades; foi pedido para que se adicionassem menus contendo parte dos parâmetros configuráveis do programa, deixando apenas os mais relevantes no painel lateral.

Para implementar essas alterações, refatorou-se novamente a classe `MainWindow`, retirando dela a responsabilidade de manter os parâmetros da simulação e, assim, isolando mais a camada de visualização da camada de modelo.

A fim de facilitar o entendimento posterior do código e aumentar a flexibilidade do sistema, nesta refatoração foram introduzidos padrões de projeto e arquiteturais sempre que possível.

Foi criada a classe `Configuration` para manter a responsabilidade pelos parâmetros da simulação, implementando o padrão *Singleton*, garantindo, assim, que o mesmo objeto é acessado pelos diversos controles da interface e pelas classes do modelo da simulação.

A classe `Simulation` foi quebrada em duas: uma responsável pelo controle da simulação e outra, pela visualização, numa instância do padrão arquitetural MVC[2], onde o modelo são as classes `Environment`, `Molecule`, `Path`, `Type` e `Configuration`.

A responsabilidade de criar o ambiente de simulação foi retirada da classe `EnvironmentCombo` e colocada numa nova classe, `EnvironmentFactory`, numa instância do padrão de projeto *Factory Method*. Com isso, isolou-se um pouco mais a parte de visualização da parte de modelo.

O resultado desta refatoração pode ser visto na Figura 12. É possível notar que, mesmo depois da refatoração, ainda há um certo acoplamento

[2] *Model-View-Controller*

entre o modelo e a visualização da simulação devido à presença das referências às classes da biblioteca VTK nas classes do modelo. De acordo com o padrão arquitetural MVC, a camada de visualização deve ter acesso às classes de modelo, e não o contrário, como ocorre após a refatoração.

2.4.12 Troca de esfera por cilindro

Desde a inserção da biblioteca ODE no sistema, a simulação da difusão no ambiente esférico ficou problemática. Aparentemente, o excesso de planos fazia com que a simulação travasse.

Como a esfera não era um ambiente de difusão de muito interesse para estudo, dado que a difusão dentro dela é praticamente homogênea, decidiu-se retirá-la do *software*, inserindo, no lugar, um ambiente cilíndrico. Neste novo ambiente, o usuário pode controlar a altura do cilindro; o raio permanece constante. Isso permite estudar diferenças de difusão de acordo com a relação diâmetro/altura do ambiente, o que pode ser interessante no caso de se estudar, por exemplo, a difusão em vasos capilares de diâmetros diferentes.

2.4.13 Coleta de mais estatísticas

Após a coleta do número de quadros por segundo, passou-se à coleta de estatísticas referentes à difusão: número de colisões por segundo e coeficiente de difusão. Para tanto, foi feita uma pequena refatoração no modo como o número de quadros por segundo era calculado e exibido na tela, de modo a flexibilizar a inserção de novas estatísticas na tela e facilitar a seleção de quais estatísticas mostrar.

Para implementar a coleta do número de colisões numa atualização da simulação, utilizou-se o padrão de projeto *Observer*, onde a classe responsável pelo cálculo desta estatística observa o simulador, o qual envia um sinal para todas as classes registradas como observadoras de colisão toda vez que uma colisão ocorre.

Já o coeficiente de difusão exigiu poucas mudanças na arquitetura: depois de refatorar a coleta de estatístitcas, bastou criar uma classe para seu cálculo e criar um atributo na classe `Molecule` que armazena a posição inicial da molécula. O coeficiente de difusão calculado por esta classe é o escalar D obtido pela equação de Einstein (2).

2.4.14 Continuação da alteração da interface

Feita a implementação da coleta das estatísticas, era necessário terminar de alterar a interface, já que a alteração feita anteriormente não incluia a adição de funcionalidade aos novos componentes.

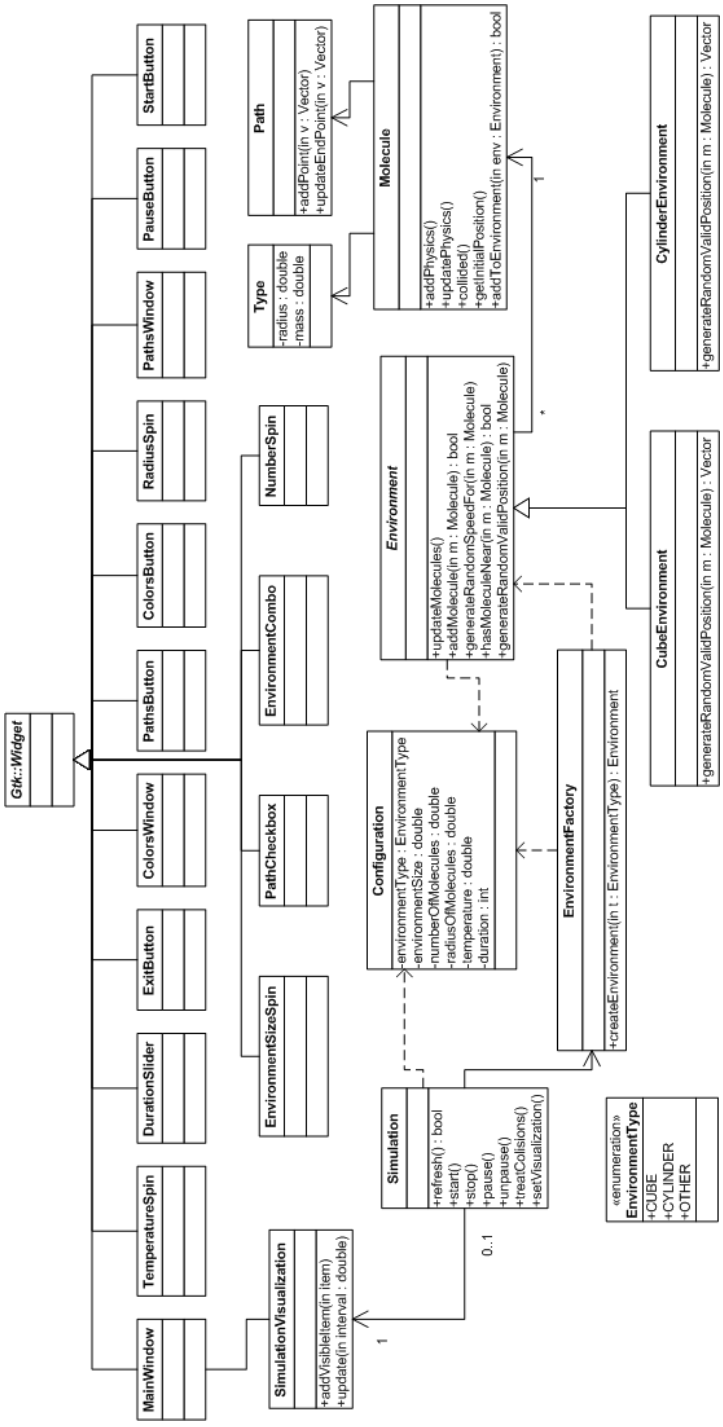

Figura 12: Diagrama de classes do programa após a segunda refatoração da classe `MainWindow`, implementando parcialmente o padrão arquitetural MVC

Como haviam parâmetros que podiam ser configurados em mais de um lugar da interface (o ambiente de difusão, por exemplo), notou-se a necessidade de implementar algum mecanismo de atualização de componentes da interface toda vez que um parâmetro fosse alterado. Para tanto, foi utilizado, novamente, o padrão *Observer*, desta vez com os componentes da interface registrando-se na classe `Configuration` para receberem um sinal sempre que um parâmetro fosse alterado. O diagrama de classes do sistema após a implementação dessa funcionalidade pode ser visto na Figura 13.

Um problema enfrentado na implementação desta nova funcionalidade foi o fato de a atualização programática do valor de um certo *widget* da interface gerar uma chamada de *callback* da biblioteca GTK – a mesma *callback* ativada quando o widget tinha seu valor alterado pelo usuário. Ou seja, cada clique do usuário fazia com que o sistema gerasse um clique em todos os *widgets* da interface. Para corrigir isso, utilizou-se um *lock* na classe `Configuration`, acessado pelo método `isSynchronizing`, que faz com que a *callback* de um *widget* não seja executada efetivamente quando o *lock* está ativo.

3 Resultados e produtos obtidos

Até o momento, o *software* obtido é capaz de simular a difusão de até duzentas moléculas simples, de formato esférico e raio entre 0,31 Å e 2,98 Å, que são, respectivamente, os raios médios calculados do átomo de Hélio e de Césio, o menor e o maior dentre os calculados [18]. Esta simulação pode ser feita dentro de um cubo de tamanho entre 10^2 Å e 10^5 Å ou dentro de um cilindro de raio 10^2 Å e altura entre 10^2 Å e 10^5 Å. A temperatura neste meio pode variar de 0,01 K a 5000,0 K.

Executando a simulação num computador com um processador de dois núcleos a 1,66 GHz, com 512 MB de memória, a taxa de atualização observada foi sempre em torno de sessenta quadros por segundo, que é a taxa esperada, embora a atualização da cena tridimensional nem sempre seja feita na mesma velocidade.

Foi possível constatar uma dificuldade razoável para refatorar o programa, em parte devido à falta de ferramentas para ajudar neste processo. No entanto, as refatorações ajudaram bastante a melhorar a arquitetura e a incluir novas funcionalidades no sistema. A arquitetura ainda está longe da ideal, mas já se mostra bastante flexível. Com a adoção do padrão MVC, será possível aproveitar o programa mesmo sem a interface gráfica, para uma simulação exclusivamente numérica.

Foi difícil manter os testes *a priori*, conforme recomenda a metodologia ágil adotada. Grande parte do código, especialmente a parte escrita depois

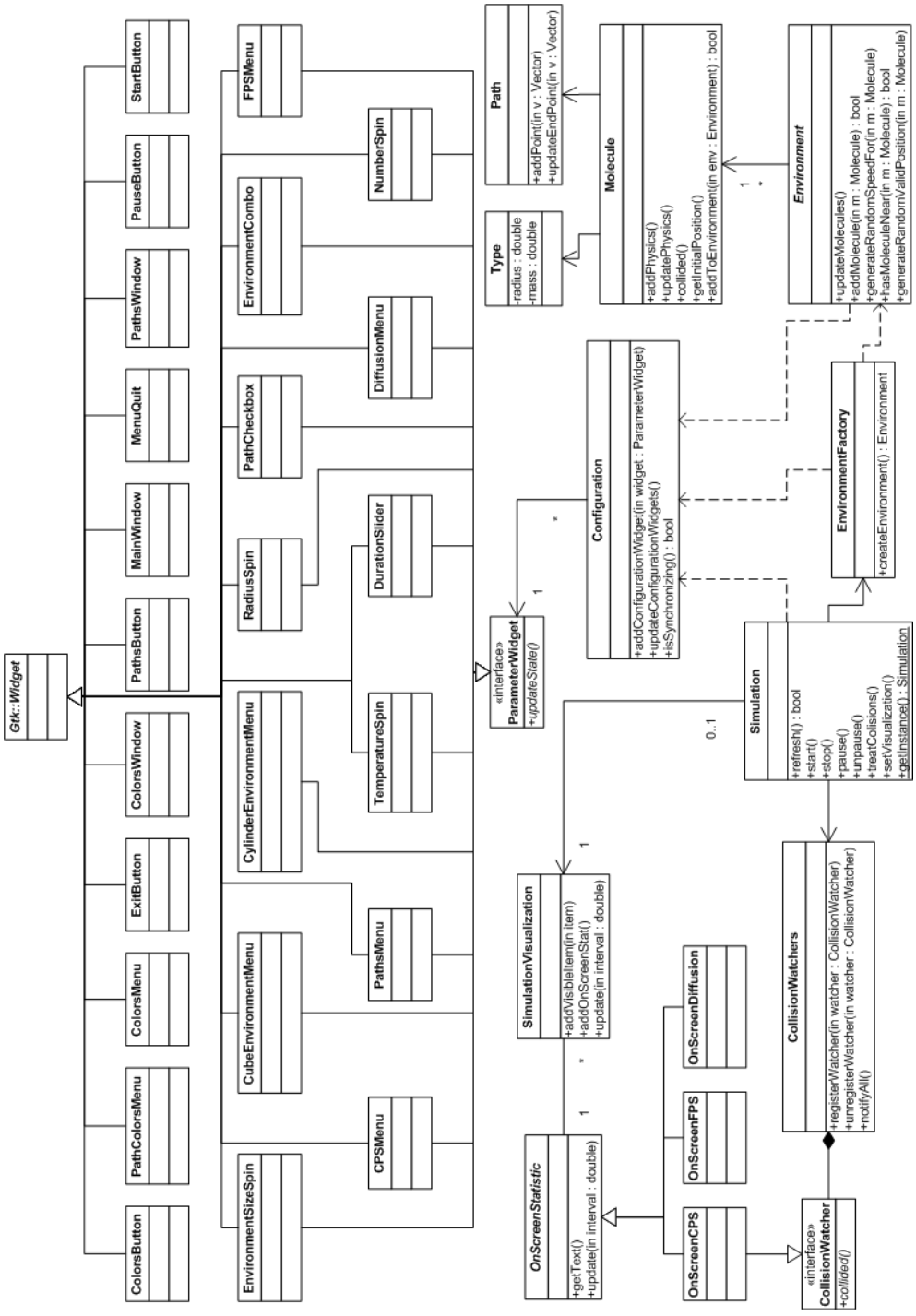

Figura 13: Diagrama de classes após a inserção de funcionalidade nos novos componentes da interface e da coleta de estatísticas

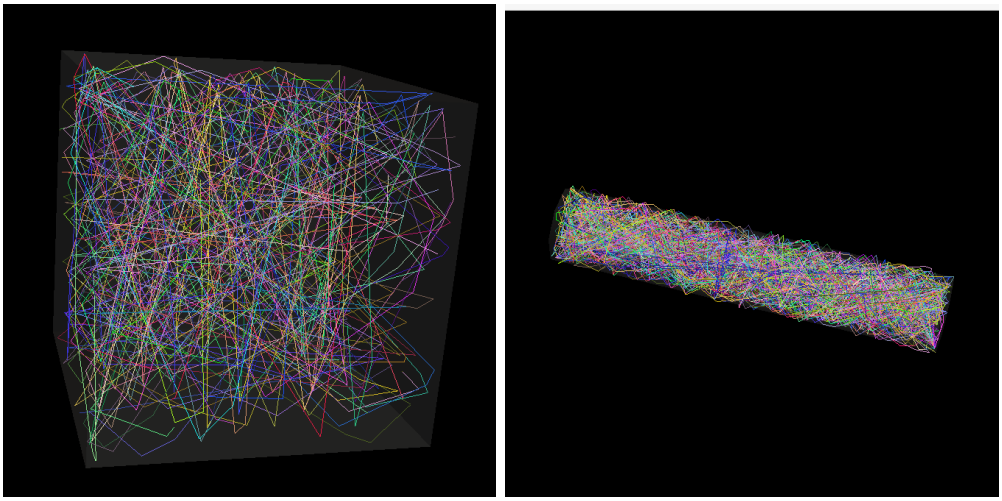

Figura 14: Comparativo de difusividade num cubo de lado 1000Å e num cilindro de raio 100Å e altura 1000Å por meio das trajetórias descritas por 100 moléculas em 5ns de simulação a 273K.

do carregamento dinâmico da interface, não foi desenvolvida conforme recomendado. No fim, apenas as classes `Vector` e `Environment` tiveram testes unitários escritos para elas.

Como a simulação ainda é bastante simples, não foi possível validar os valores obtidos com a equação de Einstein para a difusão. O modelo da simulação desenvolvido até o momento considera as moléculas como esferas rígidas, regidas apenas pelas leis da mecânica. Ainda não foram consideradas forças de origem elétrica e magnética. Entretanto, já é possível observar algumas associações entre variáveis que eram esperadas.

A Figura 14 mostra o resultado da simulação do movimento browniano em ambientes diferentes. É possível notar que no cubo a difusividade não segue um padrão claro, enquanto que, no cilindro, é possível notar que a difusividade foi maior no sentido da altura. Também é possível perceber que a densidade de moléculas no cilindro é bem maior que no cubo pelo nível de preenchimento do ambiente de difusão pelas trajetórias.

A Figura 15 mostra o resultado da simulação para diferentes condições de temperatura e pressão (número de moléculas no volume). A variação de cor indica há quanto tempo uma molécula colidiu pela última vez, dando uma idéia do coeficiente de difusão; quanto mais vermelha a molécula, mais recentemente ela colidiu. Assim, é possível notar que, nas figuras 15(a) e 15(c), a difusão é mais rápida (poucas colisões), devido à baixa temperatura em (a) e à baixa pressão em (c).

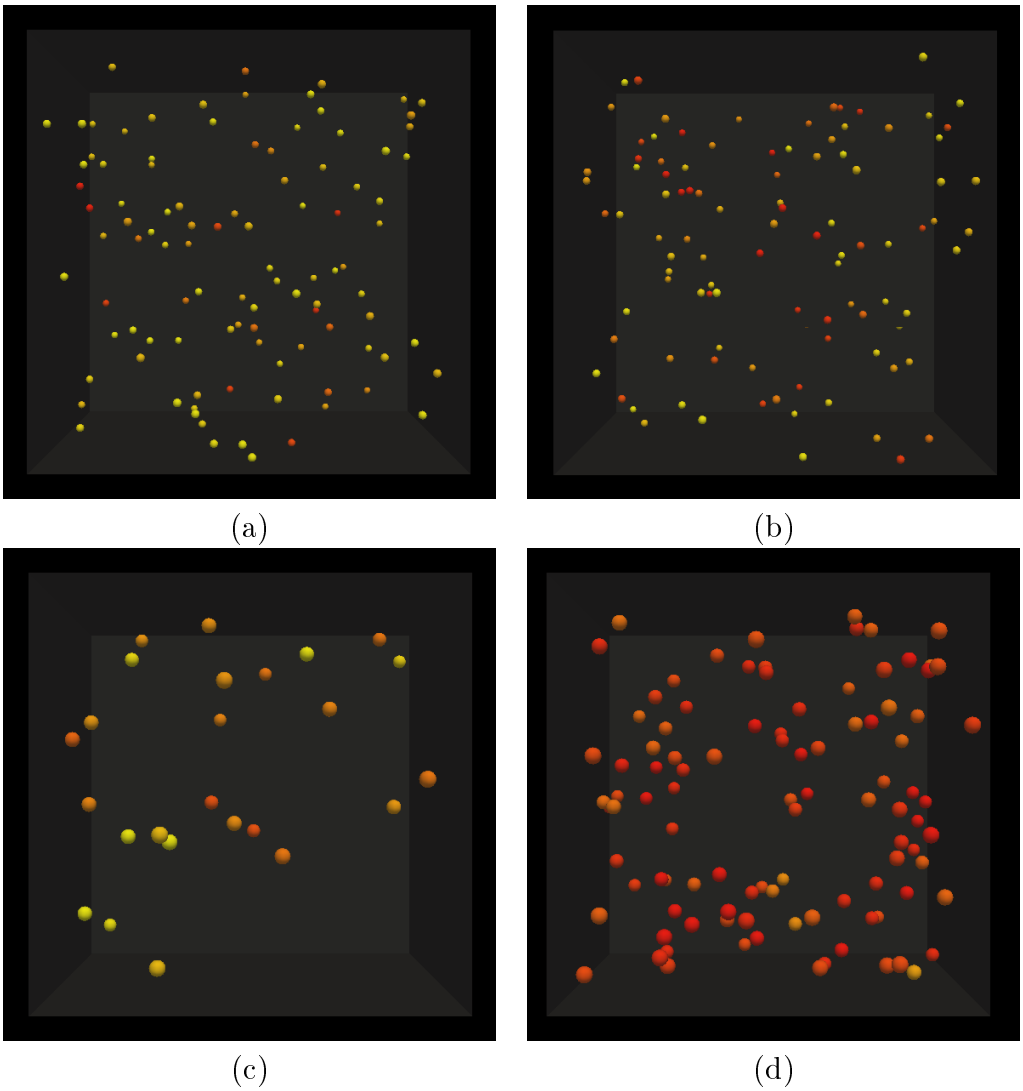

Figura 15: Comparativo da difusividade sob diferentes condições de temperatura e pressão. Em (a) e (b), 100 moléculas estão num cubo de lado igual a 1000Å mas, em (a), a temperatura é de 73K enquanto que, em (b), a temperatura é de 273K. Em (c) e (d), o ambiente de difusão é um cubo de 150Å de lado a 273K mas, no primeiro, temos apenas 25 moléculas, contra 100 do último.

4 Discussão

Sabe-se que ainda há muito o que fazer para poder simular a difusão de moléculas de água. Além de ser necessário implementar moléculas mais complexas, é preciso, por exemplo, modelar as forças inter e intra-moleculares.

Para tentar aumentar a precisão do modelo, pode-se utilizar o potencial de Lennard-Jones [19] para simular as forças inter-moleculares. Este potencial procura representar as forças atrativas que existem entre duas moléculas (ou átomos) quando estes estão próximos, forças estas que decaem com o aumento da distância entre as moléculas, e as forças repulsivas que surgem quando a distância entre elas fica menor do que a soma dos raios delas. O potencial de Lennard-Jones é dado pela seguinte equação:

$$E(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{8}$$

Nesta equação, r representa a distância entre as moléculas, ϵ regula a intensidade da força de atração e σ determina a distância na qual as forças entre as moléculas é nula; se o raio for menor, as moléculas se repelem; se for maior, as moléculas se atraem.

Também é possível modelar forças de origem elétrica por meio do potencial de Coulomb e, com isso, modelar algumas forças intra-moleculares. O potencial de Coulomb é dado pela seguinte equação:

$$E(r) = \frac{1}{4\pi\epsilon_0} \frac{q_1 q_2}{r} \tag{9}$$

Nesta equação, ϵ_0 é a permissividade elétrica do meio, q_1 e q_2 são as cargas das partículas envolvidas e r é a distância entre elas.

Com um modelo mais complexo, será possível tentar validar o coeficiente de autodifusão calculado para a água numa certa temperatura, comparando o calculado com o já conhecido experimentalmente.

Com a validação do coeficiente de difusão calculado para a água, será possível partir para a simulação da difusão de água sob restrições físicas mais complexas e para a difusão de outras substâncias, como cloreto de sódio. Também será possível estimar outras propriedades materiais das substâncias modeladas, como viscosidade, utilizando a equação 3.

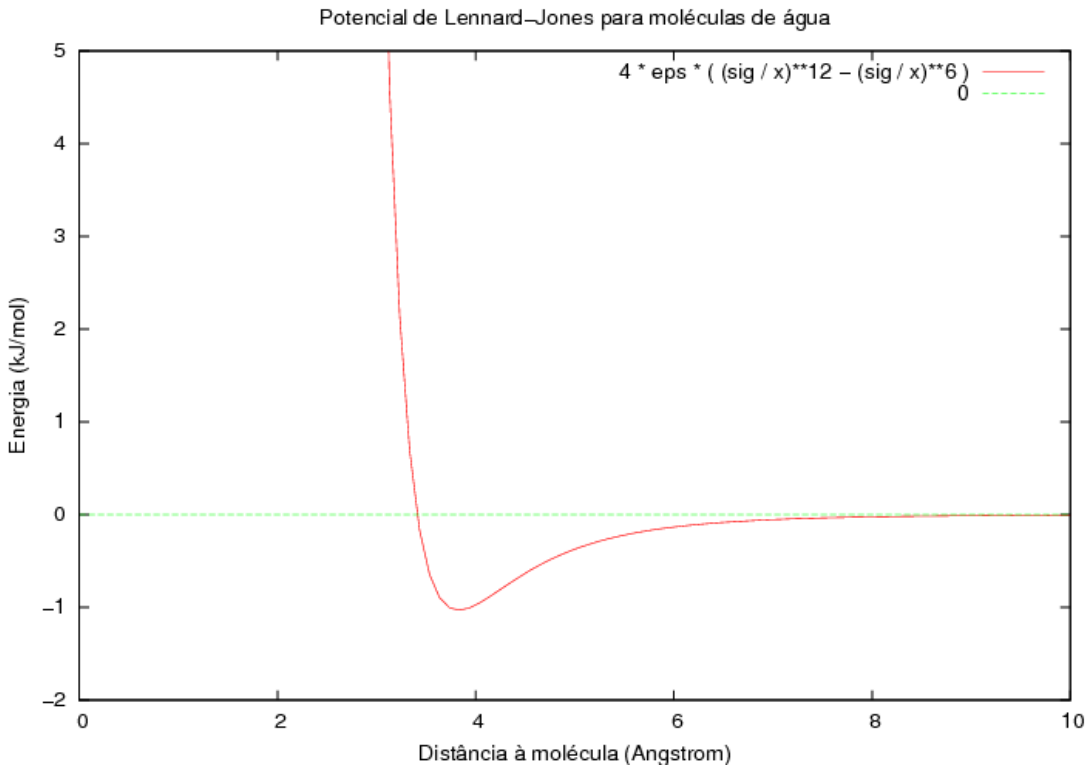

Figura 16: Gráfico da função potencial de Lennard-Jones para moléculas de água

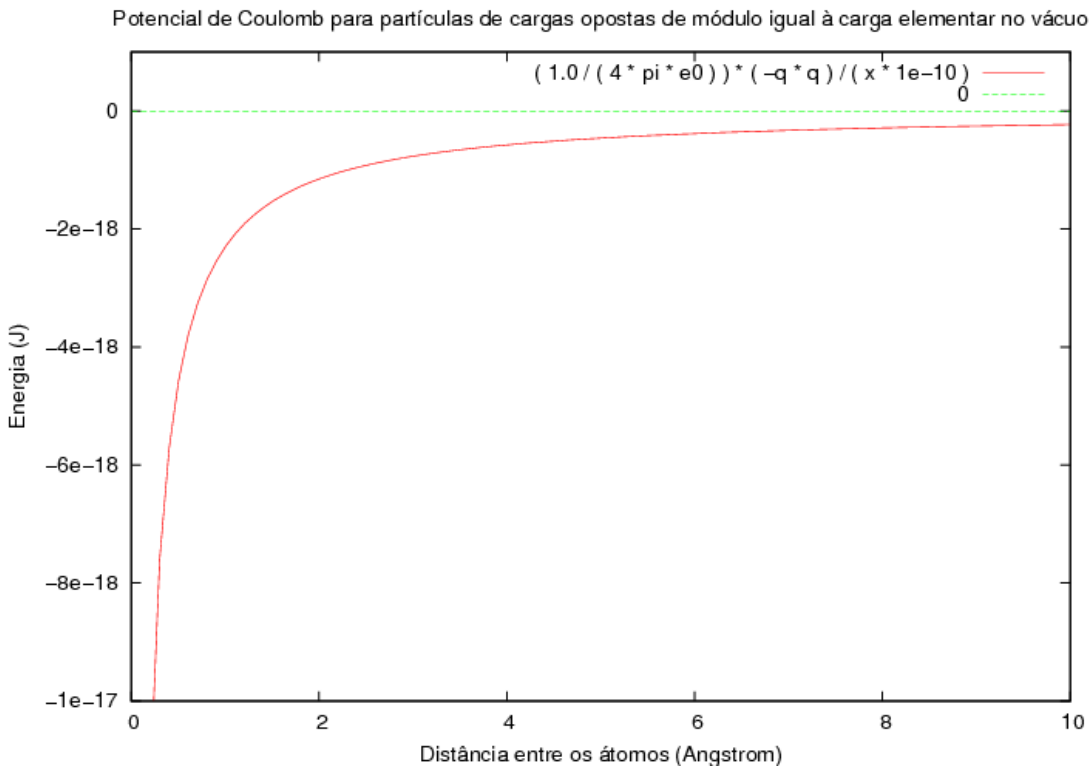

Figura 17: Gráfico da função potencial de Coulomb para duas partículas de cargas opostas e módulo igual à carga elementar (1.602×10^{-19} C) no vácuo

5 Referências

[1] EINSTEIN, A. Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen. *Ann. Phys. 17*, p. 549-560, 1905.

[2] LE BIHAN, D. *et. al.* Imaging of diffusion and microcirculation with gradient sensitization: design, strategy, and significance. *J Magn Reson Imaging*, 1:7-28, 1991.

[3] BROWN, R. *A brief account of microscopical observations made in the months of June, July and August, 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies.* Edinburgh new Philosophical Journal, p. 358-371, Julho-Setembro, 1828.

[4] LANGEVIN, P. Sur la théorie du mouvement brownien. *C. R. Acad. Sci. 146*, p. 530-533, Paris, 1908.

[5] NELSON, E. *Dynamical Theories of Brownian Motion.* 2. ed. Princeton: Princeton University Press, 2001.

[6] FICK, A. *Phil. Mag.*, 10:30, 1855.

[7] BERTULANI, C. A. *Teoria Cinética dos Gases.* 2008. Disponível em: <http://www.if.ufrj.br/teaching/fis2/teoria_cinetica/teoria_cinetica.html>. Acesso em: 10 abril 2008.

[8] ROSS, S. M. *Simulation.* 4. ed. Burlington : Elsevier Academic Press, 2006.

[9] KRAUSE, A. *Foundations of GTK+ Development*: Expert's Voice in Open Source. Berkeley: Apress, 2007.

[10] KITWARE INC. *The VTK's User's Guide, Version 4.4.* New York, 2004.

[11] KITWARE INC. *Mastering CMake.* New York, 2008. 4 ed.

[12] CUMMING, M. *et al. Programming with gtkmm.* 2006. Disponível em: <http://gtkmm.org/docs/gtkmm-2.4/docs/tutorial/pdf/programming-with-gtkmm.pdf>. Acesso em: 6 março 2008.

[13] GAMMA, E. *et al. Design Patterns*: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[14] Disponível em: <http://sourceforge.net/projects/vtkmm/>. Acesso em: 16 dezembro 2007.

[15] BEDAUX, J. *C++ Mersenne Twister pseudo-random number generator*. 2003. Disponível em: <http://www.bedaux.net/mtrand/>. Acesso em: 22 agosto 2008.

[16] SMITH, R. *Open Dynamics Engine*: v0.5 User Guide. 2006. Disponível em: <http://ode.org/ode-latest-userguide.pdf>. Acesso em: 6 março 2008.

[17] BAKKER, B. *et al. CppUnit Documentation*: Version 1.11.6. 2006. Disponível em: <http://cppunit.sourceforge.net/doc/1.11.6/>. Acesso em: 22 agosto 2008.

[18] SLATER, J. C. *J. Chem. Phys.*, 41:3199, 1964.

[19] MATTICE, W. *Lennard-Jones potential*. 1999. Disponível em: <http://gozips.uakron.edu/ mattice/ps674/lj.html>. Acesso em: 22 outubro 2008.