UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DIEGO VASCONCELOS SCHARDOSIM DE MATOS

Animação Física Simplificada em Web

RIO DE JANEIRO
2025

DIEGO VASCONCELOS SCHARDOSIM DE MATOS

Animação Física Simplificada em Web

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Cláudio Esperança

RIO DE JANEIRO

2025

"The display is the computer."

Jen-Hsun Huang

RESUMO

Este trabalho apresenta o desenvolvimento de um sistema de animação física simplificada para aplicações Web, fundamentado no método de Jakobsen (2001). O objetivo é investigar e demonstrar como técnicas de simulação leve podem produzir movimentos coerentes, estáveis e visualmente naturais, mesmo em ambientes com recursos computacionais limitados, como navegadores modernos. Para isso, são integradas abordagens clássicas de detecção e resposta a colisões, incluindo estratégias de Filtragem e Refinamento, bem como métodos geométricos amplamente utilizados em sistemas interativos. Além disso, o projeto incorpora otimizações estruturais, como subdivisão espacial e processamento em thread separada, buscando garantir escalabilidade e desempenho em cenários com múltiplos objetos dinâmicos. Os resultados obtidos evidenciam que é possível alcançar simulações eficientes e responsivas mantendo baixo custo computacional, tornando essas técnicas adequadas para jogos, visualizações interativas e aplicações educacionais na Web.

Palavras-chave: Computação gráfica; Animação Física; Detecção de Colisão; Resposta a Colisão; Corpos Rígidos e Deformáveis; Web.

ABSTRACT

This work presents the development of a simplified physical animation system for Web applications, based on Jakobsen (2001)'s method. The goal is to investigate and demonstrate how lightweight simulation techniques can produce coherent, stable, and visually natural motion, even in environments with limited computational resources, such as modern browsers. To achieve this, classical approaches to collision detection and response are integrated, including filtering and refining strategies, as well as geometric methods widely used in interactive systems. In addition, the project incorporates structural optimizations such as spatial subdivision and processing in a separate thread, aiming to ensure scalability and performance in scenarios with multiple dynamic objects. The results obtained show that it is possible to achieve efficient and responsive simulations while maintaining low computational cost, making these techniques suitable for games, interactive visualizations, and educational applications on the Web.

Keywords: Computer Graphics; Physics Animation; Collision Detection; Collision Response; Rigid and Deformable Bodies; Web.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

O campo da Computação Gráfica evoluiu significativamente nas últimas décadas, impulsionado pelo aumento da capacidade computacional e pela especialização do hardware gráfico. Esta área investiga métodos e algoritmos para gerar, manipular e representar imagens digitais de forma eficiente, desempenhando um papel central em aplicações de design, engenharia, entretenimento e visualização científica. Como destaca Azevedo (2003), a computação gráfica combina matemática e arte, oferecendo meios para representar fenômenos complexos e criar imagens inviáveis pelos métodos tradicionais. Ela pode ser encarada, portanto, como uma ferramenta que permite ao artista transcender as limitações das técnicas convencionais de desenho ou modelagem.

Em paralelo, o avanço dos programas interativos, especialmente jogos eletrônicos e simulações físicas em tempo real, intensificou a necessidade de técnicas capazes de combinar realismo visual com desempenho computacional. De acordo com Möller, Haines e Hoffman (2018) num programa interativo, uma imagem é exibida, o usuário reage, e essa reação influencia as próximas imagens a serem geradas. Esse ciclo de interação deve ocorrer a uma taxa suficientemente alta para que o usuário não veja imagens individuais, mas sim se sinta imerso em um processo dinâmico. A taxa na qual as imagens são exibidas é medida em quadros por segundo (FPS) ou Hertz (Hz). Para aplicações interativas, requer-se ao menos uma taxa de 6 FPS sendo que taxas mais elevadas tornam a experiência mais imersiva.

No contexto das aplicações interativas, destacam-se aquelas que realizam animação, isto é, a produção de imagens dinâmicas que representam objetos em movimento. Em geral, essas animações podem ser classificadas em dois grandes grupos: (i) animações *keyframe*, nas quais artistas definem manualmente os quadros-chave e as interpolações; e (ii) animações baseadas em física, em que os movimentos emergem da simulação de leis físicas, permitindo comportamentos naturais e interações complexas entre objetos.

O presente trabalho aborda a implementação de animação física 2D na Web. A escolha deste tema justifica-se por sua ampla aplicabilidade em jogos digitais, simuladores educacionais e interfaces visuais interativas. A popularidade do problema decorre de sua relevância prática e de seus desafios teóricos, que envolvem tanto a modelagem geométrica dos objetos quanto sua evolução temporal segundo princípios físicos.

Diversos jogos e motores gráficos modernos adotam animações físicas para produzir experiências imersivas e responsivas. Exemplos incluem sistemas de queda, *ragdolls*, tecidos, corpos rígidos e fluidos. Em todos esses casos, a simulação envolve um conjunto de corpos que possuem propriedades físicas, como massa, forma, velocidade e aceleração, e cujas interações são determinadas por colisões, forças e restrições. Uma animação física une dois domínios: a geometria, necessária para detectar e calcular contatos, e a física,

responsável pela evolução dinâmica dos corpos.

De forma geral, um sistema de animação física segue o seguinte esquema: (i) definição dos corpos e de suas propriedades físicas; (ii) integração numérica para atualizar suas posições e velocidades; (iii) detecção de colisões, que identifica se e como os objetos se interceptam; (iv) resposta às colisões, que corrige interpenetrações e determina novas velocidades; e, opcionalmente, (v) renderização dos resultados ao usuário. Por sua natureza iterativa e dependente de múltiplos módulos, trata-se de um problema complexo, sensível ao desempenho e à precisão.

A relevância dessa classe de problemas motivou o desenvolvimento de diversos motores de física que alavancam hardware gráfico para melhorar o desempenho. Entre os quais podemos citar motores físicos comerciais e de código aberto como o Havok (Havok, 2024) popularizou o uso profissional de física em jogos de grande porte, oferecendo um sistema robusto de colisões e restrições. O NVIDIA PhysX (NVIDIA Omniverse,) introduziu aceleração por GPU, permitindo simulações mais ricas. Motores como Bullet Physics (COUMANS, 2015) e Box2D (CATTO, 2011), ambos de código aberto, democratizaram o acesso a ferramentas de alta qualidade, tornando-se amplamente adotados em pesquisas, jogos independentes e aplicações embarcadas. Tais ferramentas demonstram a importância do tema e como algoritmos otimizados, aliados a hardware moderno, permitem simulações estáveis em tempo real.

Jakobsen (JAKOBSEN, 2001) propôs durante o desenvolvimento do jogo *Hitman: Codename 47* um esquema simplificado de simulação física que é capaz de modelar corpos rígidos e deformáveis, e tecidos, sem computar explicitamente matrizes de orientação, torques ou tensores de inércia. Seu método combina manutenção iterativa de restrições, resposta a colisões por projeção e uso de aproximações eficientes. Entretanto, o método omite diversos detalhes importantes, como estratégias de detecção de colisões, tratamento de um grande número de restrições e mecanismos de otimização para múltiplos objetos – lacunas que este trabalho busca explorar.

Independentemente da complexidade da simulação física, toda aplicação interativa requer um módulo final de renderização, responsável por apresentar ao usuário o estado atualizado da cena. No contexto Web, tecnologias como WebGL permitem que essa etapa seja realizada com aceleração gráfica, integrando o fluxo completo de animação física em tempo real.

Neste trabalho, o objetivo é desenvolver um protótipo inspirado no método de Jakobsen, apresentando soluções para detecção e resposta a colisões com foco em aplicações Web. São metas específicas:

- Revisar os principais conceitos de animação baseada em física e integração numérica;
- Implementar algoritmos de detecção de colisão

- Desenvolver uma simulação física baseada em partículas e restrições utilizando o método de Jakobsen;
- Aplicar técnicas de otimização e processamento multi-threaded;
- Avaliar o desempenho e a estabilidade do sistema em diferentes cenários.

O trabalho está organizado da seguinte forma: O Capítulo 2 fundamenta a animação baseada em física e detalha o método de Jakobsen. O Capítulo 3 descreve os algoritmos essenciais para a detecção de colisões. O Capítulo 4 aborda as técnicas de resposta a colisão, incluindo projeção de posição. O Capítulo 5 foca nas estratégias de otimização para tempo real, filtragem e refinamento, objetos delimitadores e processamento em thread separada. O Capítulo 6 apresenta a metodologia experimental e a discussão dos resultados. Por fim, o Capítulo 7 resume as contribuições, discute limitações e propõe trabalhos futuros.

2 ANIMAÇÃO BASEADA EM FÍSICA

A animação baseada em física é uma abordagem de geração de movimento que aparenta seguir princípios físicos básicos, mesmo que as equações envolvidas sejam tratadas de forma aproximada ou altamente simplificada. Diferentemente da animação tradicional, na qual o animador define manualmente posições e rotações ao longo do tempo (do inglês, *keyframe animation*), a animação física permite que o comportamento dos objetos emerja naturalmente das forças, restrições e interações entre os corpos simulados.

Nesse contexto, a animação física simplificada busca um equilíbrio entre precisão e desempenho: modelos matemáticos são utilizados como guia, mas a prioridade está na estabilidade visual e na resposta interativa. Diferentemente da simulação científica, onde precisão e correção numérica são cruciais, na animação para fins gráficos ou interativos o objetivo não é obter resultados fisicamente corretos, mas sim verossimilhança visual (plausibilidade). O foco está em transmitir sensação de peso, inércia e colisões de maneira convincente ao usuário, mesmo quando obtidos por heurísticas.

2.1 CONCEITOS E DEFINIÇÕES

O objetivo central da animação baseada em física é resolver numericamente as equações que descrevem o movimento de objetos em um mundo virtual. Tais equações derivam das leis fundamentais da mecânica clássica, formuladas por Isaac Newton.

> **Primeira lei de Newton (Inércia):** Na ausência de forças externas, um objeto em repouso permanece em repouso e um objeto em movimento continua em movimento com velocidade constante. Apenas forças externas podem alterar o estado de movimento.
>
> **Segunda lei de Newton (Princípio Fundamental da Dinâmica):** Para um corpo de massa constante m submetido a uma força $\vec{F}$, o movimento é descrito por:
> $$\vec{F} = m\vec{a} = m\frac{d\vec{v}}{dt} = m\frac{d^2\vec{x}}{dt^2},$$
> onde $\vec{x}$ é a posição do corpo, $\vec{v}$ é sua velocidade e t é o tempo.
>
> **Terceira lei de Newton (Ação e Reação):** Para toda força exercida em um corpo existe uma força de igual magnitude e direção oposta exercida no corpo que a gerou.

Em implementações simplificadas, efeitos complexos como atrito, restituição e deformação são aproximados por modelos empíricos. De maneira geral, o ciclo de uma simulação física engloba:

1. coleta e soma das forças aplicadas (gravidade, vento, atrito etc.);
2. integração temporal das equações de movimento;
3. detecção e resposta a colisões;
4. atualização das posições e posterior renderização.

2.2 DINÂMICA DE PARTÍCULAS

A dinâmica de partículas é uma abordagem na qual o sistema é composto por partículas independentes. Cada partícula possui posição, velocidade e massa, e suas interações são modeladas por forças (como gravidade ou molas) ou por restrições geométricas (como manter distâncias constantes).

O estado de uma particula no instante t é dado por:

$$X(t) = \begin{pmatrix} \vec{x}(t) \\ \vec{v}(t) \end{pmatrix}.$$

Seja $F(t)$ a soma das forças que atuam sobre a partícula e m sua massa. O movimento pode ser descrito por:

$$\frac{d}{dt}X(t) = \frac{d}{dt}\begin{pmatrix} \vec{x}(t) \\ \vec{v}(t) \end{pmatrix} = \begin{pmatrix} \vec{v}(t) \\ \dfrac{F(t)}{m} \end{pmatrix}.$$

Esta abordagem é flexível e serve como base para simular sistemas complexos, como tecidos, fluidos e corpos deformáveis, onde o comportamento macroscópico emerge da interação coletiva das partículas.

2.3 REPRESENTAÇÃO DE CORPOS RÍGIDOS

Um corpo rígido é um objeto cuja forma e volume permanecem invariáveis durante a simulação. Em termos matemáticos, a distância entre quaisquer dois pontos do corpo é constante, independentemente das forças aplicadas. Essa suposição simplifica o problema, permitindo representar o corpo apenas por grandezas globais: posição, orientação e velocidades linear e angular.

A representação matemática de um corpo rígido é dada por:

- **Posição** $\vec{p}$: coordenadas do centro de massa;
- **Orientação** R: matriz de rotação ou quatérnio;
- **Velocidade linear** $\vec{v}$: variação temporal da posição;
- **Velocidade angular** $\vec{\omega}$: variação temporal da orientação;
- **Massa** m e **tensor de inércia** I: medidas de resistência à aceleração.

2.4 SIMULAÇÃO DE CORPOS DEFORMÁVEIS

Enquanto a dinâmica de corpos rígidos assume que a distância entre os pontos constituintes do objeto permanece inalterada, a simulação de corpos deformáveis lida com objetos que alteram sua forma sob a ação de forças externas. Esta categoria abrange uma vasta gama de fenômenos físicos, desde o comportamento elástico de uma bola de borracha até a dinâmica complexa de tecidos, cabelos e fluidos.

Na mecânica do contínuo, a deformação é descrita pela mudança na configuração métrica do material, gerando tensões internas que tendem a restaurar o objeto ao seu estado de repouso ou dissipar energia na forma de deformação plástica. No contexto da computação gráfica em tempo real, no entanto, modelos discretos simplificados, tipicamente baseados em partículas, são preferidos em relação a métodos mais complexos, como por exemplo os Métodos de Elementos Finitos (FEM, do inglês *Finit Element Methods*).

Sistemas Massa-Mola

A abordagem mais comum para simular deformações em jogos e aplicações interativas é o modelo Massa-Mola. Neste modelo, um objeto é discretizado em um conjunto de partículas pontuais com massa m, conectadas por molas ideais sem massa.

A força interna exercida por uma mola entre duas partículas i e j é descrita pela Lei de Hooke:

$$\vec{F}_{elastica} = -k_s(|\vec{x}_i - \vec{x}_j| - L_0)\frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|},$$

onde k_s é a constante de rigidez, L_0 é o comprimento de repouso da mola e $\vec{x}$ são as posições das partículas. Para garantir a estabilidade numérica e simular a perda de energia, adiciona-se frequentemente um termo de amortecimento:

$$\vec{F}_{amortecimento} = -k_d(\vec{v}_i - \vec{v}_j)\frac{\vec{x}_i - \vec{x}_j}{|\vec{x}_i - \vec{x}_j|},$$

onde k_d é o coeficiente de amortecimento e $\vec{v}$ são as velocidades.

Estruturação Topológica para Tecidos

Para simular superfícies deformáveis, como tecidos, as partículas são organizadas em uma grade. A estabilidade e o comportamento visual do tecido dependem de como essas molas (ou restrições) são conectadas. Uma topologia robusta geralmente emprega três tipos de conexões:

1. Molas Estruturais: Conectam partículas vizinhas diretas (horizontal e verticalmente). Elas resistem à tração e compressão básicas, mantendo a integridade da malha.

2. Molas de Cisalhamento: Conectam partículas diagonalmente vizinhas. Sua função é impedir que o tecido se distorça ou "deslize" sobre si mesmo, mantendo a estabilidade dos quadriláteros da malha.

3. Molas de Flexão: Conectam partículas alternadas (pulando um vizinho). Elas resistem à dobra do tecido, impedindo que ele se comporte como uma malha infinitamente flexível e conferindo-lhe uma certa rigidez à curvatura.

2.5 MÉTODOS NUMÉRICOS EM SIMULAÇÃO

A simulação de movimento depende da solução numérica das equações diferenciais que descrevem a dinâmica dos corpos. Diversos métodos de integração podem ser utilizados, cada um com um equilíbrio distinto entre precisão, estabilidade e custo computacional.

Método de Euler

O método de Euler é o mais simples e intuitivo. Ele atualiza a posição e a velocidade de acordo com a aceleração atual:

$$\vec{v}_{t+\Delta t} = \vec{v}_t + \vec{a}_t \Delta t,$$
$$\vec{x}_{t+\Delta t} = \vec{x}_t + \vec{v}_t \Delta t.$$

Apesar de sua simplicidade, o método de Euler tende a ser numericamente instável, especialmente em sistemas oscilatórios (como molas), pois o erro de integração cresce rapidamente ao longo do tempo.

Método Semi-implícito de Euler

Uma variação estável do método de Euler consiste em atualizar primeiro a velocidade e depois a posição, utilizando a nova velocidade no cálculo:

$$\vec{v}_{t+\Delta t} = \vec{v}_t + \vec{a}_t \Delta t,$$
$$\vec{x}_{t+\Delta t} = \vec{x}_t + \vec{v}_{t+\Delta t} \Delta t.$$

Essa pequena modificação melhora a conservação de energia e reduz a instabilidade numérica, sendo amplamente adotado em motores como o Box2D.

Integração de Verlet

Verlet é um método de segunda ordem que não armazena explicitamente a velocidade. A nova posição é calculada com base na posição atual, na posição anterior e na aceleração:

$$\vec{x}_{t+\Delta t} = 2\vec{x}_t - \vec{x}_{t-\Delta t} + \vec{a}_t \Delta t^2.$$

A velocidade, quando necessária para cálculos de amortecimento ou jogabilidade, é derivada implicitamente:

$$\vec{v}_t \approx \frac{\vec{x}_{t+\Delta t} - \vec{x}_{t-\Delta t}}{2\Delta t}.$$

Esse é o método usado por Jakobsen (2001), por ser um método estável e eficiente, especialmente em sistemas sujeitos a restrições geométricas além de eliminar a necessidade de armazenar explicitamente a velocidade, utilizando as posições atual e anterior para estimar a nova posição.

2.6 RESTRIÇÕES GEOMÉTRICAS

Jakobsen (2001) descreve uma restrição geométrica simples para compor estruturas complexas, a restrição linear (ou restrição de distância) mantém uma distância fixa d entre duas partículas i e j. Dessa forma o conjunto de partículas deve satisfazer a todo instante uma coleção de equações na forma:

$$|\vec{x_i} - \vec{x_j}| = d. \tag{2.1}$$

Mesmo que as posições das partículas estejam inicialmente corretas, depois de um passo de simulação a distância entre elas pode se tornar inválida (devido a forças externas, por exemplo). Para corrigir sua distância devemos movê-las (projetar) de forma a satisfazer Eq. 2.1, como ilustrado na Figura 1.

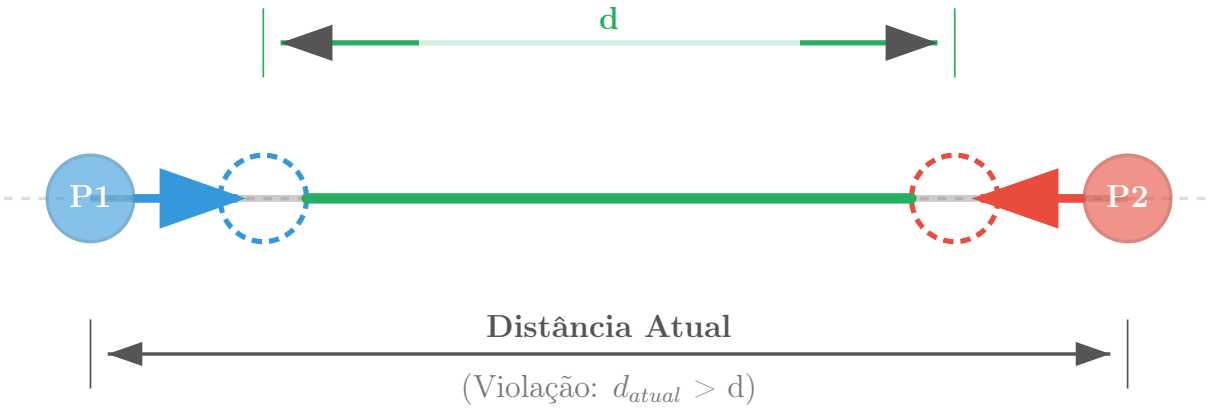


Figura 1 – A distância atual entre as partículas P1 e P2 está inválida e deve ser corrigida por projeção.

Jakobsen (2001) também descreve um esquema de articulação (dobradiça) fazendo que objetos compartilhem a mesma partícula como ilustrado na Figura 2. Dessa forma, seria possível modelar um corpo humanoide, Jakobsen (2001) também mostra que para realismo adicional pode-se limitar o ângulo de rotação de articulações através de restrições angulares satisfazendo o produto interno abaixo:

$$(x_2 - x_0) \cdot (x_1 - x_0) < \alpha.$$

Um método alternativo e simples que Jakobsen (2001) também mostra é usar restrições de distância auxiliares, que não foi usada neste trabalho, onde a distância entre as extremidades i e j não seja maior que um valor d_{min}:

$$|\vec{x}_i - \vec{x}_j| > d_{min}$$

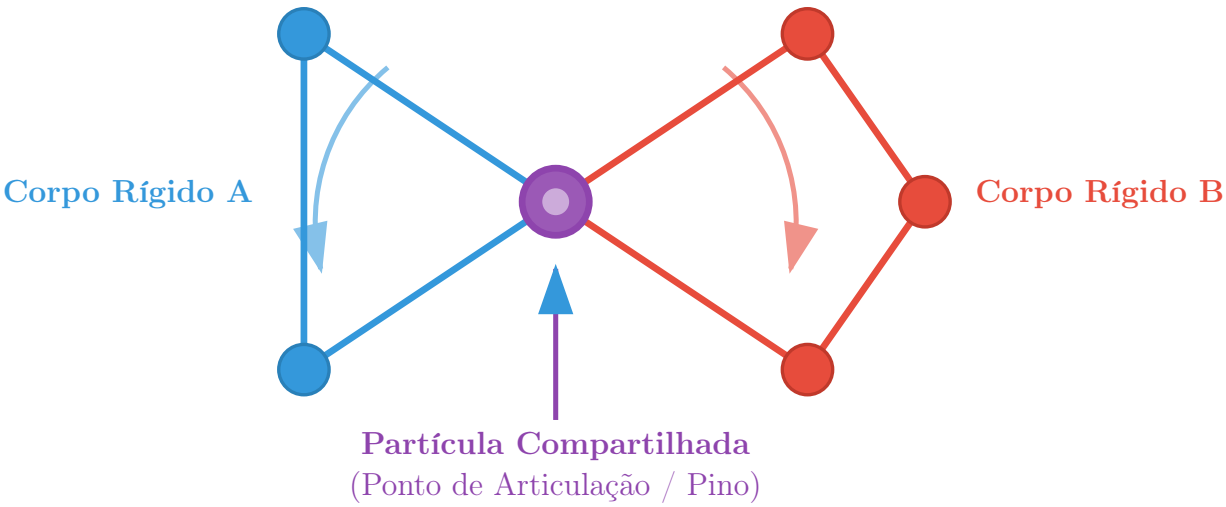


Figura 2 – A articulação é formada simplesmente fazendo os dois corpos compartilharem a mesma partícula

2.7 RESOLVENDO RESTRIÇÕES CONCORRENTES POR RELAXAMENTO

Na prática, uma simulação pode conter muitas restrições de todos os tipos vistos anteriormente. Para satisfazer todas elas devemos resolver todas elas sequencialmente, como as restrições entre partículas são interdependentes, não é possível satisfazê-las todas simultaneamente de maneira exata em um único passo.

Jakobsen (2001) propõe um método iterativo de relaxamento, também conhecido como relaxamento de *Gauss-Seidel*, para resolver as restrições de forma aproximada. É uma abordagem de solução indireta por iteração local que consiste em aplicar pequenas correções de posição para cada par de partículas conectado, repetindo o procedimento diversas vezes até que todas as restrições estejam aproximadamente satisfeitas. Cada iteração contribui para reduzir o erro acumulado, e a convergência ocorre rapidamente mesmo com poucas iterações (geralmente entre 3 e 5).

Apesar dessa abordagem parecer ingênua, ao resolver todas restrições localmente e repetir, o sistema global do sistema converge para uma configuração que satisfaz todas restrições. Quanto maior o número de iterações, mais rápido o sistema irá convergir para solução ideal e também a animação irá parecer mais rígida para o usuário.

Deve-se encontrar o movimento mínimo que satisfaça cada restrição, para restrição linear podemos fazer como no Algoritmo 1 que irá separar ou aproximar as partículas de

tal forma que satisfaçam a distância d. Onde ϵ representa o tamanho de passo local para

Algoritmo 1: Relaxamento para restrição linear

$\vec{\delta} \leftarrow \vec{x}_2 - \vec{x}_1$
$c \leftarrow \frac{\|\vec{\delta}\| - d}{\|\vec{\delta}\|}$
$\vec{x}_1 \leftarrow \vec{x}_1 - \epsilon\,\vec{\delta}\,c$
$\vec{x}_2 \leftarrow \vec{x}_2 + \epsilon\,\vec{\delta}\,c$

convergência para solução ideal. Quando $\epsilon = 1$ as partículas são movidas imediatamente para posição correta. Essa situação é comparável a um sistema de molas interconectadas entre partículas de rigidez que tendem para o infinito ou a uma haste rígida separando as duas partículas.

Jakobsen (2001) utiliza um modelo baseado em partículas e restrições geométricas, evitando explicitamente a solução de equações diferenciais rígidas e instáveis. Em vez disso, correções iterativas são aplicadas diretamente às posições das partículas, proporcionando estabilidade numérica elevada e comportamento visualmente plausível mesmo sob passos de tempo grandes.

3 DETECÇÃO DE COLISÕES

A detecção de colisões é um componente fundamental em sistemas de simulação física e em animações baseadas em partículas. Esse processo identifica quando dois ou mais objetos entram em contato, determina pontos de interseção e, quando necessário, fornece informações como vetores de penetração e normais que servirão de entrada para a etapa subsequente de resposta física. Em contextos interativos em tempo real, a precisão geométrica absoluta costuma ser sacrificada em favor da eficiência computacional, desde que o comportamento resultante se mantenha visualmente verossímil.

Este capítulo introduz os principais conceitos utilizados no âmbito deste trabalho, descrevendo as representações geométricas mais comuns para objetos convexos e apresentando dois algoritmos amplamente empregados em detecção de colisões: o Teorema do Eixo Separador (SAT, do inglês *Separating Axis Theorem*) e o algoritmo Gilbert-Johnson-Keerthi (GJK). Ambos operam eficientemente em formas convexas e constituem a base de vários motores físicos modernos.

3.1 POLÍGONOS CONVEXOS

Um objeto geométrico é definido como um conjunto não vazio, limitado e fechado de pontos. A propriedade de ser fechado implica que sua fronteira pertence ao próprio conjunto, enquanto a limitação garante que exista uma esfera de raio finito que contenha todos os seus pontos. Um plano, por exemplo, é fechado, mas não limitado.

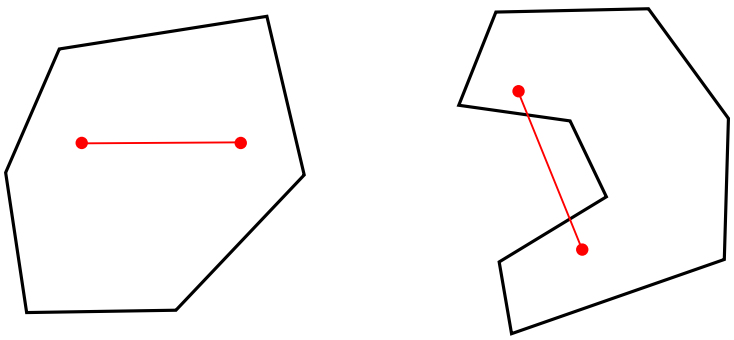


Figura 3 – Comparação entre polígono convexo (à esquerda) e polígono côncavo (à direita).

Uma forma é considerada convexa se, para quaisquer dois pontos contidos nessa forma, todo o segmento que os une também estiver contido nela, como ilustrado na Figura 3. Uma consequência prática é que, para qualquer linha que atravesse o objeto, esta o intersecta em no máximo dois pontos. Formas não convexas podem ser tratadas como composições de múltiplas partes convexas, o que permite a aplicação direta de algoritmos especializados.

3.2 TEOREMA DO EIXO SEPARADOR (SAT)

O Teorema do Eixo Separador é um dos métodos mais difundidos para detecção de colisão entre polígonos convexos. Além de identificar a presença ou ausência de interseção, o SAT também pode ser utilizado para calcular o vetor de translação mínima (MTV, do inglês *Minimum Translation Vector*), útil para correções geométricas e resposta física.

O SAT é um algoritmo genérico rápido que pode remover a necessidade de ter código de detecção de colisão para cada par tipo de forma, reduzindo assim o código e a manutenção. Ao traçar raios paralelos sob dois objetos (a partir de uma fonte luminosa, por exemplo) se as sombras formadas estiverem separadas então não há colisão, repita esse processo ao redor das formas, caso não encontre uma sombra que esteja separada uma da outra os objetos estão em colisão. O SAT se baseia no teorema geométrico que afirma:

Teorema 1. *Dois polígonos convexos A e B não se intersectam se, e somente se, existir um eixo (reta) sobre o qual as projeções de A e B não se sobrepõem.*

Tal eixo é denominado eixo separador, como ilustrado na Figura 4. Em termos computacionais, o algoritmo testa um conjunto finito de eixos candidatos. Para polígonos, os candidatos suficientes são as normais das faces (ou arestas, em 2D) de ambos os objetos.

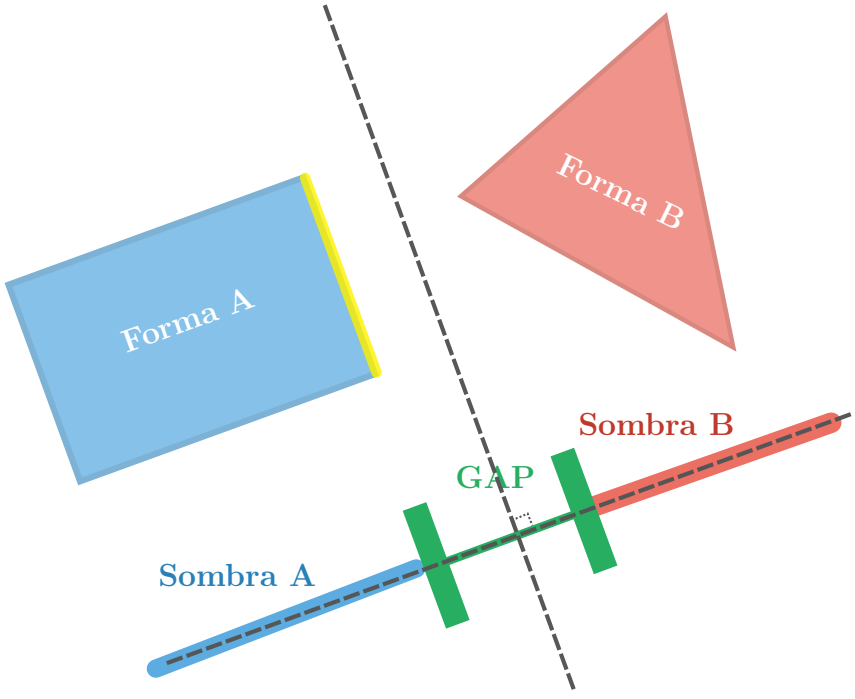


Figura 4 – Face escolhida em amarelo, eixo separador perpendicular a face.

Dado um eixo unitário $\hat{n}$ e um conjunto de vértices $\{v\}$ de um polígono, a projeção

gera um intervalo escalar $[min, max]$ definido por:

$$min = \min_i(\hat{n} \cdot \vec{v}_i)$$
$$max = \max_i(\hat{n} \cdot \vec{v}_i).$$

A verificação de sobreposição entre dois intervalos $[min_A, max_A]$ e $[min_B, max_B]$ é dada pela condição:

$$\text{Sobreposição} \iff max_A \geq min_B \quad \wedge \quad max_B \geq min_A,$$

como ilustrado na Figura 4. Se essa condição falhar em qualquer eixo candidato, os objetos estão separados e o algoritmo pode encerrar imediatamente.

O SAT pode ser estendido para calcular a penetração mínima. Caso todos os eixos apresentem sobreposição, a menor sobreposição encontrada corresponde à magnitude do vetor necessário para resolver a colisão, como mostrado no Algoritmo 2 , onde ϵ é um

Algoritmo 2: SAT com cálculo do MTV

```
Input: Polígonos convexos A e B
Output: d⃗ e δ, ou falso
d⃗ ← 0⃗
δ ← ∞
foreach aresta n̂ de A e B do
    n̂ ← normal unitária de n̂
    p1 ← Projeção de A
    p2 ← Projeção de B
    Δ ← sobreposição entre p1 e p2
    if Δ ≤ ε then
        return False
    if Δ < δ then
        δ ← Δ
        d⃗ ← n̂
return d⃗, δ
```

valor numérico para tratar erros de operação flutuante e deve-se ser um valor baixo. O método possui complexidade linear no número de arestas e é bastante eficiente para polígonos convexos em 2D. Em 3D, entretanto, o número de eixos candidatos cresce significativamente, reduzindo sua praticidade em relação a alternativas como o GJK.

3.3 ALGORITMO DE GILBERT-JOHNSON-KEERTHI (GJK)

O algoritmo de Gilbert-Johnson-Keerthi (GJK) é um algoritmo clássico para calcular distância euclidiana entre dois objetos poligonais convexos, também é capaz de determinar os respectivos pontos mais próximos entre eles. Este algoritmo possui uma complexidade linear em relação ao número de vértices dos objetos e não está restrito a uma dimensão

específica, podendo ser usado em qualquer espaço m-dimensional. Para calcular a distância é usado o subalgoritmo de Johnson, que resolve um sistema de equações lineares para combinação de vértices, faces e arestas. O algoritmo de GJK é amplamente utilizado em motores físicos devido à sua eficiência e robustez, especialmente em ambientes tridimensionais (GILBERT; JOHNSON; KEERTHI, 1988).

Com o passar dos anos, o subalgoritmo de distância de Johnson foi trocado a favor de outros mais sofisticados como, por exemplo, o subalgoritmo de Ericson apresentado na SIGGRAPH 2004 (ERICSON, 2004a). Este subalgoritmo se baseia numa abordagem geométrica ao invés de algébrica, com otimização usando regiões de Voronoi. Neste trabalho, detalharemos a versão aprimorada do GJK proposta por Gilbert e Foo (1990) que é capaz de tratar qualquer objeto curvo convexo, e o subalgoritmo de Ericson.

O algoritmo de GJK fundamenta-se na soma de Minkowski entre dois conjuntos convexos A e B, definida como

$$A + B = \{\vec{x} + \vec{y} \mid \vec{x} \in A,\ \vec{y} \in B\},$$

onde $\vec{x}$ e $\vec{y}$ representam vetores posição associados a pontos pertencentes aos conjuntos A e B, respectivamente.

Do ponto de vista geométrico, a soma de Minkowski pode ser interpretada como o conjunto de todos os pontos obtidos ao transladar o objeto B por cada vetor posição pertencente a A, mantendo sua forma e orientação, ou seja, faz-se uma cópia do objeto B centrado em cada ponto de A, como ilustrado na Figura 5.

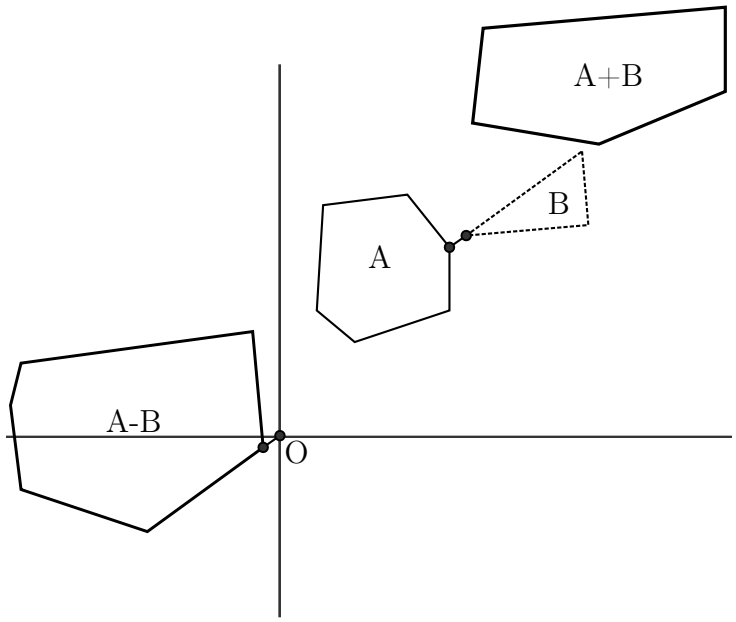


Figura 5 – Soma de Minkowski, entre os conjuntos A e B.

Uma propriedade muito útil da soma de Minkowski é o fato de que a soma de dois objetos convexos é um objeto convexo. Para um objeto A, usamos a notação $-A$ para

denotar o reflexo de A sobre a origem O. A diferença de Minkowski $A-B$ pode ser obtida calculando a soma de Minkowski de A e $-B$

$$A - B = A + (-B) = \{\vec{x} - \vec{y} \mid \vec{x} \in A, \vec{y} \in B\},$$

que pode ser pensado como um processo de varredura que calcula o vetor distância para cada ponto de B em A. Neste trabalho, usaremos esse termo para referir a essa operação quando necessário.

A diferença de Minkowski também é chamada de configuração do espaço de obstáculos (CSO, do inglês *Configuration Space Obstacles*). Nesse espaço a distância entre os objetos A e B é igual à distância entre a origem O e o CSO, como ilustrado na Figura 5, que chamaremos de *MinimumNormPoint(CSO)*. Logo, A e B se intersectam se, e somente se, sua CSO contém a origem (LINAHAN, 2015). Isso se deve ao fato de que se existe um ponto $p_A \subset A$ igual a $p_B \subset B$, então $p_A - p_B = \vec{0}$ é testemunha que há pelo menos um ponto que coincide com a origem O. Compare a Figura 6 à Figura 5.

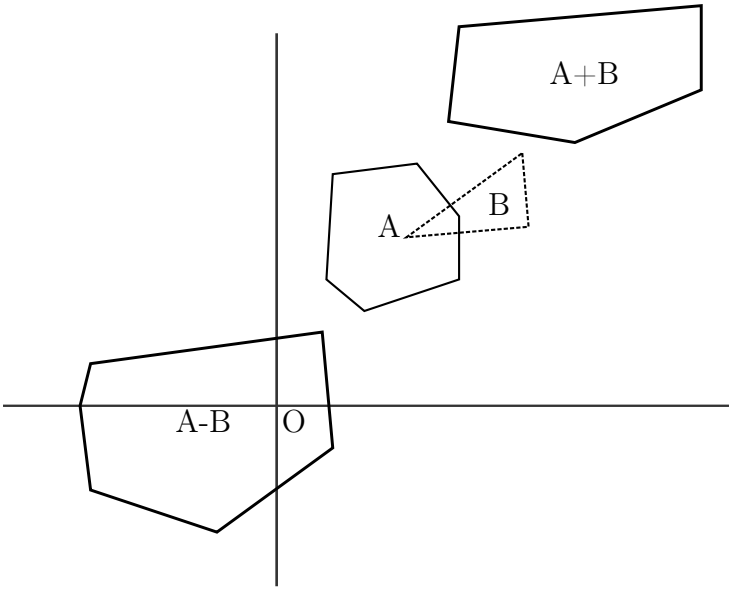


Figura 6 – Soma de Minkowski entre os conjuntos A e B em intersecção.

A grande coisa do algoritmo é que você não precisa calcular todo CSO, apenas uma amostra do CSO usando um mapeamento suporte. Uma função de suporte é definida tal que: seja A um conjunto qualquer, o suporte de A na direção $\vec{d}$ retorna o ponto que pertence a superfície de A mais distante da origem na direção $\vec{d}$, esse ponto é chamado de ponto de suporte. Gilbert e Foo (1990) mostraram que é possível representar qualquer objeto convexo com a função suporte correta, como por exemplo:

$$\text{Politopo} : S_A(\vec{d}) = max\{v \cdot a : a \in A\},$$
$$\text{Esfera com raio R} : S_A(\vec{d}) = R \cdot \vec{d}$$

Também é possível realizar aritmética com função suporte

$$S_{A+B}(\vec{d}) = S_A(\vec{d}) + S_B(\vec{d})$$
$$S_{-A}(\vec{d}) = -S_A(-\vec{d}),$$

dessa forma, seja D a diferença de Minkowski de dois objetos convexos A e B, o suporte de D na direção $\vec{d}$ é a diferença dos suportes de A e B: $S_{A-B}(\vec{d}) = S_A(\vec{d}) - S_B(-\vec{d})$, como ilustrado pela Figura 7.

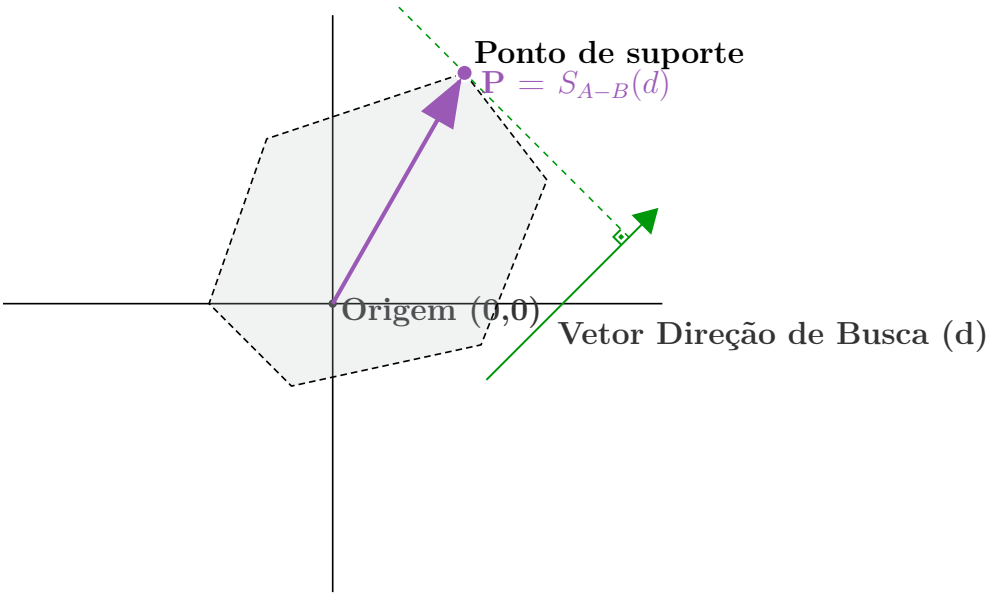


Figura 7 – A função suporte na forma implícita $A - B$ ao longo da direção $\vec{d}$.

A busca pelo ponto mais próximo da origem dentro do CSO é sustentado pelo teorema de Carathéodory (ROCKAFELLAR, 1970). O teorema afirma que se um ponto x está no interior do fecho convexo de um conjunto P então x está no interior de um simplex k-dimensional com vértices em P. Na literatura também pode ser encontrado como uma combinação convexa de no máximo d + 1 pontos de P.

Um Simplex é definido como um politopo de k+1 vértices em um espaço de k-dimensão. Dessa forma um ponto é 0-simplex, um segmento de reta é 1-simplex, um triângulo é 2-simplex, um tetraedro é 3-simplex, como na Figura 8.

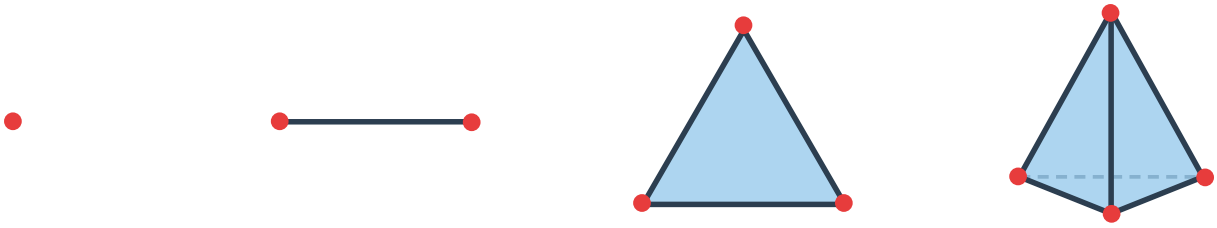


Figura 8 – Da esquerda para direita 0-simplex, 1-simplex, 2-simplex e 3-simplex

O teorema de Carathéodory em 2 dimensões afirma que podemos construir um triângulo usando pontos em P que envolve qualquer ponto no interior do fecho convexo de

P. Por exemplo, seja $P = (0,0), (0,1), (1,0), (1,1)$. O fecho convexo deste conjunto é um quadrado. Seja $x = (1/4, 1/4)$ no interior do fecho convexo de P. Podemos então construir um conjunto $(0,0), (0,1), (1,0) = P'$, cujo fecho convexo é um triângulo e envolve x, como ilustrado na Figura 9. (Wikipedia contributors, 2025)

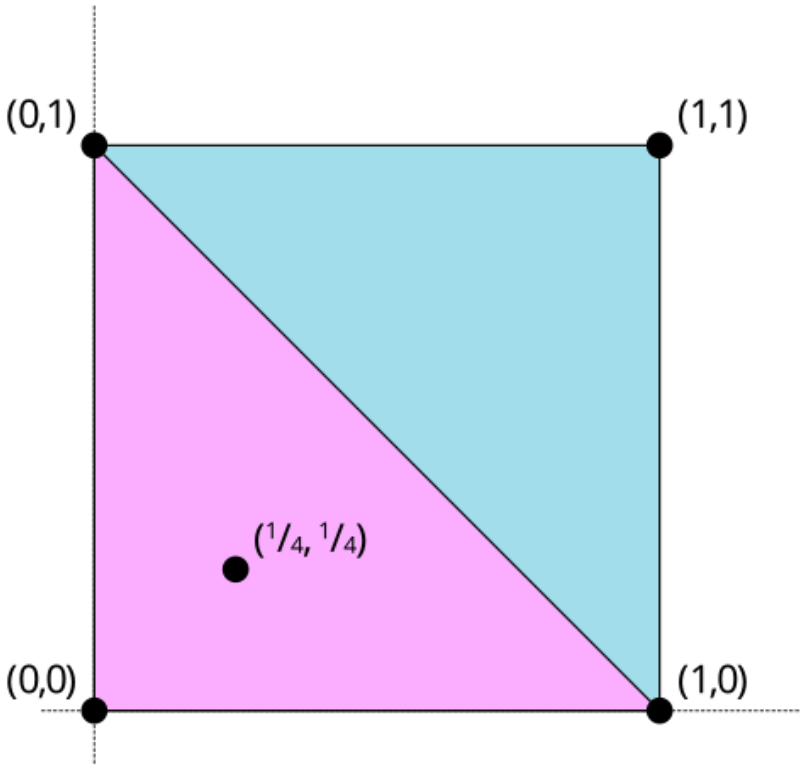


Figura 9 – Ilustração do teorema de Carathéodory. O ponto x está no interior do fecho convexo de P, e também está no interior do fecho convexo de P'.

Usaremos a mesma notação de Linahan (2015), $||P||$ para denotar a norma (distância à origem) de um ponto P. O algoritmo constrói iterativamente simplexos dentro do CSO mais próximos da origem até que o simplexo pare de mudar. O ponto P de norma mínima desses simplexos serve como aproximações sucessivas para *MinimumNormPoint(CSO)*.

Para politopos, o GJK termina após um número finito de iterações. Para objetos curvos, um termo épsilon deve ser adicionado para evitar loops infinitos e limitar o erro entre a distância de separação calculada e real. Foi estabelecido em que a distância ao quadrado entre a aproximação P e o *MinimumNormPoint(CSO)* real é limitada a um valor inferior a $P \cdot V$, onde V é o ponto de suporte do CSO na direção $-P$. Assim, se $||P||^2 - P \cdot V \leq \epsilon^2$, para alguma constante real muito pequena $\epsilon \geq 0$, concluímos que V não é mais extremo na direção $-P$ do que o próprio P e a distância do CSO à origem é ||P||.

O Algoritmo 3 começa com um ponto arbitrário P no CSO e um conjunto simplex vazio Q. Então calcula um ponto de suporte V na direção $-P$. Se V não estiver mais

Algoritmo 3: GJK

Input: Polígonos convexos A e B
Output: Distância euclidiana entre A e B
P $\leftarrow$ Ponto qualquer na CSO
Q $\leftarrow$ {}
$V \leftarrow S_{CSO}$(-S)
while $||P||^2 - P \cdot V > \epsilon^2$ **do**
  $Q' \leftarrow Q \cup \{V\}$
  P $\leftarrow$ MinimumNormPoint(ConvexHull(Q'))
  Q $\leftarrow$ Faça Q' o menor subconjunto convexo que contém P
  P $\leftarrow S_{CSO}(-P)$
return $||P||$

distante na direção $-P$ do que o próprio P, então P deve ser o ponto mais próximo da origem: o algoritmo termina com $||P||$ como a distância do CSO à origem. Caso contrário, o algoritmo adiciona V ao conjunto simplex atual Q e atualiza P para ser o ponto mais próximo da origem dentro do novo simplex (MinimumNormPoint(ConvexHull(Q'))). O simplex Q é reduzido ao menor subconjunto convexo de Q que ainda contém P, e o ponto de suporte V na direção $-P$ é atualizado para a próxima iteração.

O subalgoritmo de distância de Ericson (2004a) é responsável por computar o ponto mais próximo da origem dentro do novo simplex. Sua versão substituiu a versão algébrica de Johnson por uma geométrica, usando rotinas geométricas primitivas como *ClosestPointOnEdgeToPoint()* e *ClosestPointOnTriangleToPoint()*, que é implementado usando operações vetoriais. Apesar disso seu algoritmo é matematicamente equivalente ao de Johnson com um pequeno acrescimo de otimização: uma condição de parada antecipada que testa se o ponto P mais próximo da origem coincide com a origem (LINAHAN, 2015).

Para cada dimensão do simplex Q, o subalgoritmo de Ericson calcula se a origem está contida ou não no simplex usando regiões de Voronoi, remove pontos antigos do simplex e procura por uma nova direção que maximize as chances do próximo ponto de suporte conter a origem. Para cada dimensão é feito uma conta diferente, neste trabalho iremos trabalhar apenas com objetos em 2D, por isso trataremos apenas dos casos 1-simplex e 2-simplex, Linahan (2015) realiza a prova a corretude e também mostra para 3-simplex. Para detalhar essa rotina usaremos uma conveção de nomear os pontos do Simplex em ordem alfabética, com o ponto mais recente sendo A.

Para o caso de 1-simplex temos a rotina *ClosestPointOnEdgeToPoint()* que testa se a origem está contida numa aresta

Para o caso de 2 simplex temos a rotina *ClosestPointOnTriangleToPoint()* que testa para dois tipos de região de Voronoi, região de Vértice ou região de Aresta

1. O origin está contida numa região de Vértice. Equação de exemplo para V_A:
   - $AX \cdot AB \leq 0$

- $AX \cdot AC \leq 0$

2. O origin está contida numa região de Aresta. Equação de exemplo para E_{AB}:

   - $(BC \times BA) \times BA \cdot BX \geq 0$
   - $AX \cdot AB \geq 0$
   - $BX \cdot BA \geq 0$

Dependendo do problema a ser resolvido, é possível modificar o GJK para apenas determinar se há ou não intersecção entre dois objetos A e B, conhecido como Boolean GJK (BGJK). Este algoritmo se baseia no fato de que A e B se intersectam se, e somente se, sua CSO contém a origem, desta forma o objetivo do GJK se reduz para determinar se a CSO contém ou não a origem. Neste trabalho foi implementado uma versão do BGJK apresentada por Muratori (2006) que é baseado nos avanços de Gilbert e Foo (1990) e Ericson, feito no Algoritmo 4.

Algoritmo 4: Boolean GJK

```
Input: Polígonos convexos A e B
Output: True se colidindo, False caso contrário
S ← S_CSO(Direção aleatória)
Q ← {S}
d ← −S
while iterações < MAX_ITER do
    S ← S_CSO(d)
    if S · d < 0 then
        return False
    Q ← Q ∪ {S}
    if DoSimplex(Q, d) then
        return True
return False
```

Inicializando um simplex Q com um ponto de suporte S numa direção aleatória do CSO (é comum ser escolhido a direção do centro de A para B). Então S é usado para gerar uma nova direção $\vec{d}$ oposta a S, isso é feito para maximizar a área do Simplex e as chances de conter a origem. O loop principal consiste em três etapas principais: primeiro calcular um novo ponto de suporte na direção atual e verificar se esse ponto ultrapassou a origem ($S \cdot \vec{d} < 0$), se sim a origem não pode estar contida no Simplex Q, caso contrário adicionamos S ao Simplex e continuamos tentando incluir a origem.

Por fim, chamamos a rotina *DoSimplex* que é responsável por: calcular se a origem está contida ou não no Simplex Q usando regiões de Voronoi, remover pontos antigos do simplex e procurar por uma nova direção que maximize as chances do próximo ponto de suporte conter a origem. Para cada dimensão é feito uma conta diferente, neste trabalho

iremos trabalhar apenas com objetos em 2D, por isso trataremos apenas dos casos 1-simplex e 2-simplex, Linahan (2015) realiza a prova a corretude e também mostra para 3-simplex. Para detalhar essa rotina usaremos uma conveção de nomear os pontos do Simplex em ordem alfabética, com o ponto mais recente sendo A.

4 RESPOSTA A COLISÕES

A resposta a colisões é uma etapa fundamental em qualquer sistema de simulação física interativa. Após detectar que dois corpos estão em interpenetração, torna-se necessário aplicar um conjunto de correções que restaurem a plausibilidade física do movimento, evitando instabilidades numéricas.

Este capítulo discute os princípios clássicos e as formulações modernas de resposta a colisão, estabelecendo a relação entre os métodos geométricos de detecção e a abordagem baseada em partículas e restrições proposta por Jakobsen (2001), que fundamenta este trabalho.

4.1 MÉTODOS DINÂMICOS NA SIMULAÇÃO FÍSICA

A resposta a colisões consiste, essencialmente, em duas operações principais:

1. Correção de Posição: eliminar a interpenetração entre dois corpos.
2. Correção de Velocidade: remover ou ajustar componentes da velocidade que induziriam novo contato imediato.

A literatura apresenta diversas abordagens para modelar o movimento de corpos rígidos e deformáveis. Embora este trabalho se baseie no método simplificado de Jakobsen (2001), é importante contextualizar outras categorias amplamente utilizadas em motores físicos modernos:

Método do Impulso

O Método do Impulso trata colisões aplicando impulsos instantâneos que alteram diretamente as velocidades dos corpos para preservar o momento linear e angular. Essa abordagem é utilizada em motores como Havok (Havok, 2024) e Bullet (COUMANS, 2015). O impulso é calculado em função da velocidade relativa no ponto de contato, resultando em um método eficiente para simulações em tempo real.

Método de Penalidades

Nos Métodos de Penalidades, colisões são tratadas como interpenetrações que geram forças de repulsão proporcionais à profundidade de penetração alterando diretamente a aceleração. Essas forças geralmente seguem modelos de mola e amortecimento. Trata-se de um método simples, porém sensível à escolha dos parâmetros de rigidez, podendo causar instabilidade numérica.

Método Baseado em Restrições com Multiplicadores de Lagrange

Os métodos baseados em restrições formulam os contatos como equações que devem ser satisfeitas exatamente. São resolvidos usando multiplicadores de Lagrange, essa abordagem é robusta e adequada para sistemas complexos, mas exige a solução de sistemas lineares, tornando sua aplicação onerosa em plataformas Web.

4.2 PROCESSO DE SEPARAÇÃO

A metodologia de Jakobsen (2001) pode ser compreendida como uma precursora da moderna dinâmica baseada em posição (PBD, do inglês *Position Based Dynamics*) (MÜLLER et al., 2007). Diferentemente das abordagens que atuam sobre a velocidade ou aceleração, Jakobsen (2001) trata colisões como restrições geométricas adicionais que devem ser satisfeitas durante o processo iterativo de relaxamento da simulação. Assim, objetos penetrando um ao outro são corrigidos exclusivamente por modificações de posição.

A resposta à colisão envolve dois passos principais. O primeiro consiste em separar os elementos geométricos (vértices, arestas ou faces) que se encontram em interpenetração, o que caracteriza um processo estritamente geométrico. O segundo passo corresponde a um processo iterativo de relaxamento, no qual os elementos afetados ajustam suas posições de acordo com as restrições impostas pelo sistema físico.

Para dois objetos convexos A e B em colisão, o esquema de detecção de colisão deve retornar os pontos de contato de cada objeto, o tamanho da penetração e a direção $\vec{d}$ de separação. Com essas informações devemos tratar duas configurações possíveis: colisão vértice-vértice ou colisão vértice-aresta.

Colisão Vértice-Vértice

Se o contato ocorre pontualmente entre duas partículas p e q, a correção é dividida igualmente (assumindo massas iguais):

$$\Delta\vec{p} = -\frac{1}{2}\vec{d}, \qquad \Delta\vec{q} = +\frac{1}{2}\vec{d}. \tag{4.1}$$

Colisão Vértice-Aresta

Esta é a configuração mais comum. Um vértice p de um corpo colide contra uma aresta definida pelas partículas $\vec{x}_1$ e $\vec{x}_2$ de outro corpo. O ponto de impacto p cai entre dois vértices $\vec{x}_1$ e $\vec{x}_2$ e o nosso objetivo é corrigir as suas posições para uma configuração válida $\vec{x}'_1$, $\vec{x}'_2$ de tal forma que garanta uma coerência geométrica, como mostra na Figura 10. Logo, pela equação da reta, p pode ser descrito como uma interpolação linear

$$p = \alpha\vec{x}_1 + (1-\alpha)\vec{x}_2, \qquad 0 \leq \alpha \leq 1, \tag{4.2}$$

ou seja, quando $\alpha = 0, p = x_2$, quando $\alpha = 1, p = x_1$.

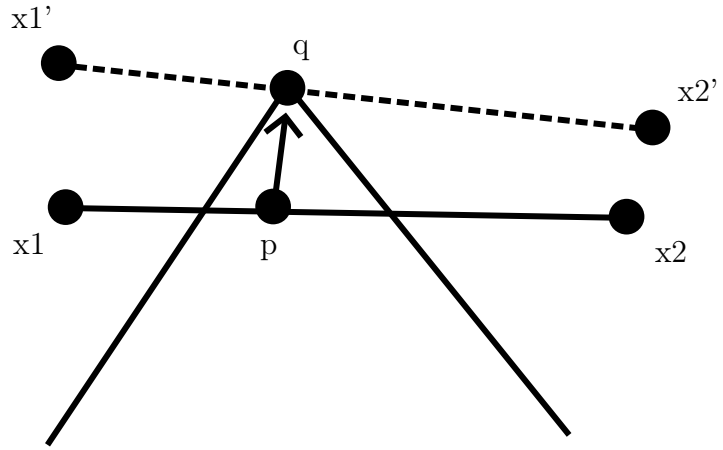


Figura 10 – Processo de separação caso colisão vértice-aresta. Correção das posição para configuração válida. Perceba como p está mais próximo de x_1, esse vértice é movido mais

Durante a separação, a correção $\Delta\vec{p}$ deve alterar indiretamente $\vec{x}_1$ e $\vec{x}_2$ de forma proporcional a essa parametrização. A partir da Eq. 4.2, derivamos o valor de α calculando quanto o ponto de contato p está ao "longo" da aresta que vai do ponto $\vec{x}_2 \to \vec{x}_1$. Dessa forma, Jakobsen (2001) computa as novas posições movendo as partículas proporcionalmente a α:

$$\begin{aligned} \vec{x}_1 &+= \alpha\Delta_p \\ \vec{x}_2 &+= (1-\alpha)\Delta_p \\ \alpha &= \frac{(\vec{p}-\vec{x}_2)\cdot(\vec{x}_1-\vec{x}_2)}{\|\vec{x}_1-\vec{x}_2\|^2}. \end{aligned}$$

Isso garante que a geometria original é preservada e que o ponto $\vec{p}$, definido implicitamente pelos vértices da aresta, é deslocado exatamente pela quantidade desejada.

4.3 ALGORITMO DE EXPANSÃO DE POLITOPOS (EPA)

Enquanto o SAT fornece naturalmente o MTV, o algoritmo BGJK apenas informa se há interseção (retornando um *Simplex* interno à diferença de Minkowski). Para obter a profundidade e a normal da colisão necessárias para a resposta física, utiliza-se o Algoritmo de Expansão de Politopos (EPA, do inglês *Expanding Polytope Algorithm*).

O Algoritmo 5 expande o *simplex* final do BGJK iterativamente até encontrar a fronteira da diferença de Minkowski mais próxima da origem. A distância entre o ponto mais próximo com a origem é a profundidade de penetração. Além disso, o vetor normal para o ponto mais próximo é a direção de separação (ponto de contato). A solução ingênua é usar a normal da face mais próxima da origem, porém um *simplex* não precisa conter nenhuma das faces do polígono original, o que pode levar ao cálculo de uma normal incorreta. A

Algoritmo 5: EPA

Input: Simplex
Output: $\hat{n}$, δ
while *iterações* $<$ *MAX_ITER* **do**
    e $\leftarrow$ Encontrar aresta mais próxima a origem
    p $\leftarrow$ Calcular novo ponto de suporte na direção da normal de e
    $\delta \leftarrow p \cdot normal(e)$
    if $|\delta - length(e)| < \epsilon$ **then**
        return *normal(e)*, δ
    Adicionar ponto ao simplex

tolerância ϵ (ex: 10^{-4}) e o limite de iterações são cruciais para evitar loops infinitos em casos de precisão numérica flutuante ou formas curvas aproximadas.

4.4 LIMITAÇÕES

Ao adotar métodos simplificados, abre-se mão de características essenciais de motores físicos completos. Entre as limitações mais relevantes estão:

- Ausência de conservação precisa de energia e momento, o que reduz o realismo de certas interações.
- Incapacidade de simular materiais complexos (ex.: fricção anisotrópica, torques realistas, elasticidade avançada).
- Dependência de parâmetros empíricos, sem interpretação física clara.
- Menor robustez para geometrias arbitrárias, especialmente polígonos côncavos ou mal escalonados.
- Dificuldade de lidar com sistemas altamente conectados (estruturas rígidas, máquinas, esqueletos).

Essas limitações não invalidam o uso das técnicas, mas reforçam a necessidade do uso da aplicação a cenários onde a prioridade é a responsividade, e não a precisão física.

Jittering

Em sistemas baseados em posições a estabilidade depende fortemente do processo de correção de posições. Um dos problemas mais recorrentes é o *jitter*, um tremor ou oscilação indesejada no posicionamento dos corpos, especialmente perceptível quando múltiplas restrições são aplicadas simultaneamente ou quando o sistema é altamente rígido.

Empilhamento

Métodos simplificados têm dificuldade em manter pilhas estáveis de objetos, principalmente quando as correções não são distribuídas de forma global e consistente. O empilhamento tende a "escorregar" ou colapsar devido à falta de precisão numérica adequado.

Tunneling

Ocorre quando objetos em alta velocidade atravessam outros sem detectar colisão. Métodos baseados exclusivamente em detecção discreta apresentam maior risco, especialmente quando o passo temporal é grande ou a geometria é fina.

5 OTIMIZAÇÕES

A eficiência computacional é um dos fatores determinantes para o desempenho de um motor de física em tempo real. Em jogos, animações interativas, simulações físicas e aplicações gráficas, a necessidade de atualizações contínuas e a obrigatoriedade de operar dentro de limites rigorosos de tempo tornam indispensável o uso de técnicas de otimização em todos os estágios do pipeline de simulação.

Como qualquer objeto pode potencialmente colidir com qualquer outro, uma simulação contendo n objetos requer $(n-1)+(n-2)+\cdots+1 = n(n-1)/2 = O(n^2)$ testes de pares no pior caso. Devido à complexidade quadrática, testar ingenuamente cada par torna-se impraticável mesmo para valores moderados de n.

Reduzir o custo associado ao teste de pares afetará o tempo de execução apenas linearmente. Para realmente acelerar o processo, o número de pares testados deve ser reduzido. Essa redução é realizada separando o tratamento de colisões de múltiplos objetos em duas fases: filtragem e refinamento.

Neste capítulo, descrevemos as estratégias clássicas de otimização aplicadas à detecção e resolução de colisões, com foco na divisão entre filtragem e refinamento, no uso de estruturas espaciais, na adoção de passos temporais fixos e na execução multi-thread de simulações físicas.

5.1 OBJETOS DELIMITADORES

Em sistemas de simulação física e detecção de colisões, a representação geométrica dos objetos influencia diretamente a eficiência dos cálculos. Formas complexas, com muitos vértices ou superfícies não convexas, tornam tais testes significativamente mais custosos. Uma solução comum é empregar aproximações convexas que possibilitam testes rápidos sem sacrificar excessivamente a precisão da simulação.

A principal motivação para o uso de objetos delimitadores é que formas mais simples (como caixas ou esferas) permitem testes de sobreposição muito mais baratos do que a geometria original que envolvem. Além disso, se eles se intersectam não necessariamente os objetos se intersectam, mas caso não se intersectam necessariamente os objetos não se intersectam, como ilustrado na Figura 11. Esse fato pode ser usado para acelerar os testes de colisão, como veremos posteriormente.

Segundo Möller, Haines e Hoffman (2018), nem todos os objetos geométricos servem como objetos delimitadores eficazes. As propriedades desejáveis para objetos delimitadores incluem:

- Testes de interseção de baixo custo

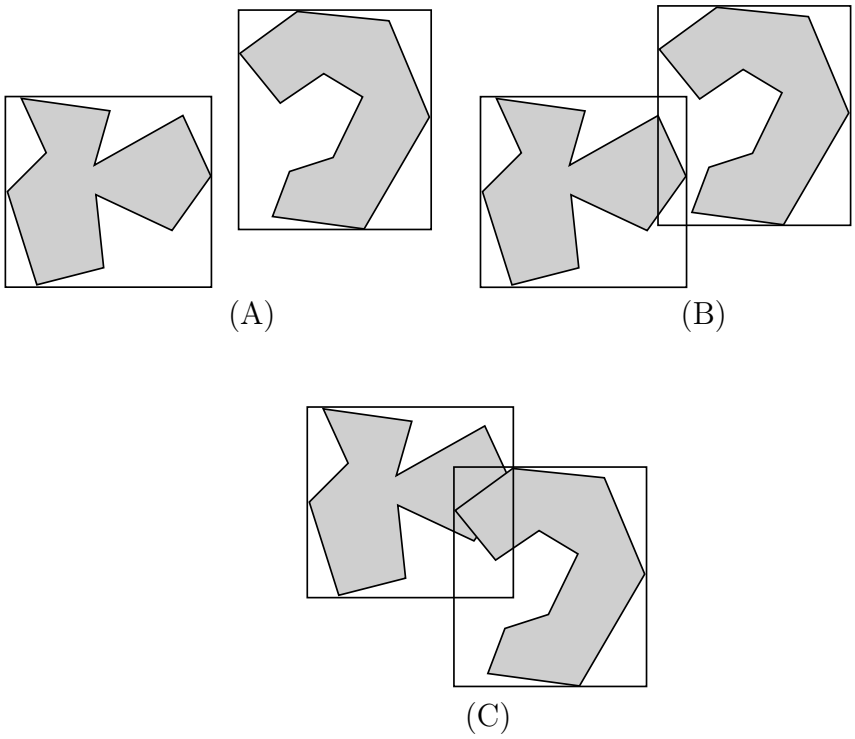


Figura 11 – Em (A) os objetos delimitadores não se intersectam, em (B) eles se intersectam porém os objetos não se intersectam, em (C) eles se intersectam e os objetos se intersectam.

- Ajuste preciso
- Cálculo econômico
- Fácil de girar e transformar
- Consome pouca memória

como ilustrado na Figura 12.

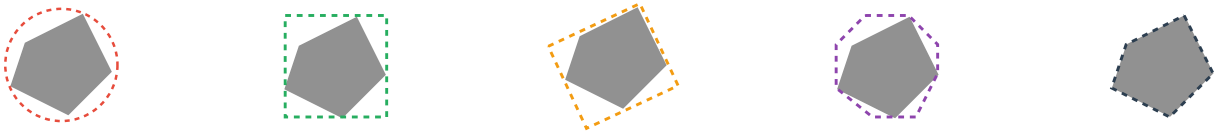


Figura 12 – Da esquerda para direita esfera delimitadora, caixa delimitadora alinhada aos eixos, caixa delimitadora orientada, K-DOP e fecho convexo.

Esfera Delimitadora

As esferas constituem o volume delimitador mais simples, definidas apenas por um centro c e um raio r:

$$\text{Sphere} = \{x \in \mathbb{R}^3 \mid \|x - c\| \leq r\}.$$

Testes de colisão entre esferas são rápidos, porém inadequados para objetos de proporções irregulares. Elipsoides oferecem melhor ajuste, mas aumentam o custo de teste. Por isso, essas formas são frequentemente utilizadas em fases preliminares da detecção, ou como nós intermediários em hierarquias de volumes delimitadores (BVH, do inglês *Bouding Volume Hierarchy*).

Caixa Delimitadora Alinhada ao Eixo Coordenado

A caixa delimitadora alinhada aos eixos (AABB, do inglês *Axis Align Bounding Box*) é um dos objetos delimitadores mais comuns. Trata-se de um paralelepípedo (ou retângulo, em 2D) cujas faces são paralelas aos eixos do sistema de coordenadas. Seu teste de interseção é simples como no Algoritmo 6.

Algoritmo 6: Teste de Intersecção AABB

Input: A e B: volumes AABB
Output: Verdadeiro se houver colisão
if $A.x_{max} < B.x_{min}$ ***ou*** $A.x_{min} > B.x_{max}$ **then**
  return *False*
if $A.y_{max} < B.y_{min}$ ***ou*** $A.y_{min} > B.y_{max}$ **then**
  return *False*
if $A.z_{max} < B.z_{min}$ ***ou*** $A.z_{min} > B.z_{max}$ **then**
  return *False*
return *True*

Caixa Delimitadora Orientada

Uma Caixa Delimitadora Orientada (OBB, do inglês *Oriented Bouding Box*) é uma caixa retangular que pode estar arbitrariamente rotacionada em relação aos eixos do sistema de coordenadas. A representação mais comum é feita por um ponto central c, por três vetores ortogonais $\hat{u}_i$ que compõem sua orientação e por extensões e_i, que são assumidas serem positivas.

OBBs geralmente oferecem melhor ajuste, especialmente para objetos alongados ou rotacionados, reduzindo falsos positivos. Porém, o teste de interseção é mais caro que o das AABBs e geralmente é feito usando o SAT.

Fecho Convexo

O Fecho Convexo (FC) de um conjunto finito de pontos p é definido como a menor região convexa que contém todos os pontos em p, sendo frequentemente utilizado como um objeto delimitador. Determinar o fecho convexo é um problema recorrente em computação geométrica, especialmente quando se deseja organizar pontos em estruturas mais simples ou acelerar operações posteriores, como testes de colisão.

Existem vários algoritmos, entre os quais o *Quickhull* é um algoritmo bastante conhecido para o cálculo do fecho convexo de um conjunto finito de pontos em qualquer dimensão, adotando uma estratégia de divisão e conquista semelhante ao *quicksort* (BARBER; DOBKIN; HUHDANPAA, 1996).

O Quickhull parte de um conjunto de pontos S e constrói o polígono (ou poliedro) convexo que os contém. O processo para 2 dimensões é ilustrado na Figura 13 e pode ser descrito em linhas gerais como no Algoritmo 7. Apresenta complexidade média $O(n \log n)$

Algoritmo 7: Quickhull 2D

Input: Polígono Convexo

Output: Lista dos vértices do fecho convexo

1. Encontre os pontos de menor e maior coordenada em x; eles pertencem ao fecho convexo.
2. Use a linha formada pelos dois pontos para dividir o conjunto em dois subconjuntos de pontos, que serão processados de forma recursiva.
3. Para cada subconjunto, encontre o ponto mais distante da linha; ele forma um triângulo que exclui pontos interiores.
4. Repita recursivamente os dois passos anteriores nas duas linhas formadas pelos dois novos lados do triângulo.
5. O processo termina quando todos os subconjuntos estão vazios.

em 2D, podendo chegar a $O(n^2)$ em casos degenerados. Em 3D, adapta-se a construções poliedrais mais complexas, mantendo o mesmo princípio recursivo.

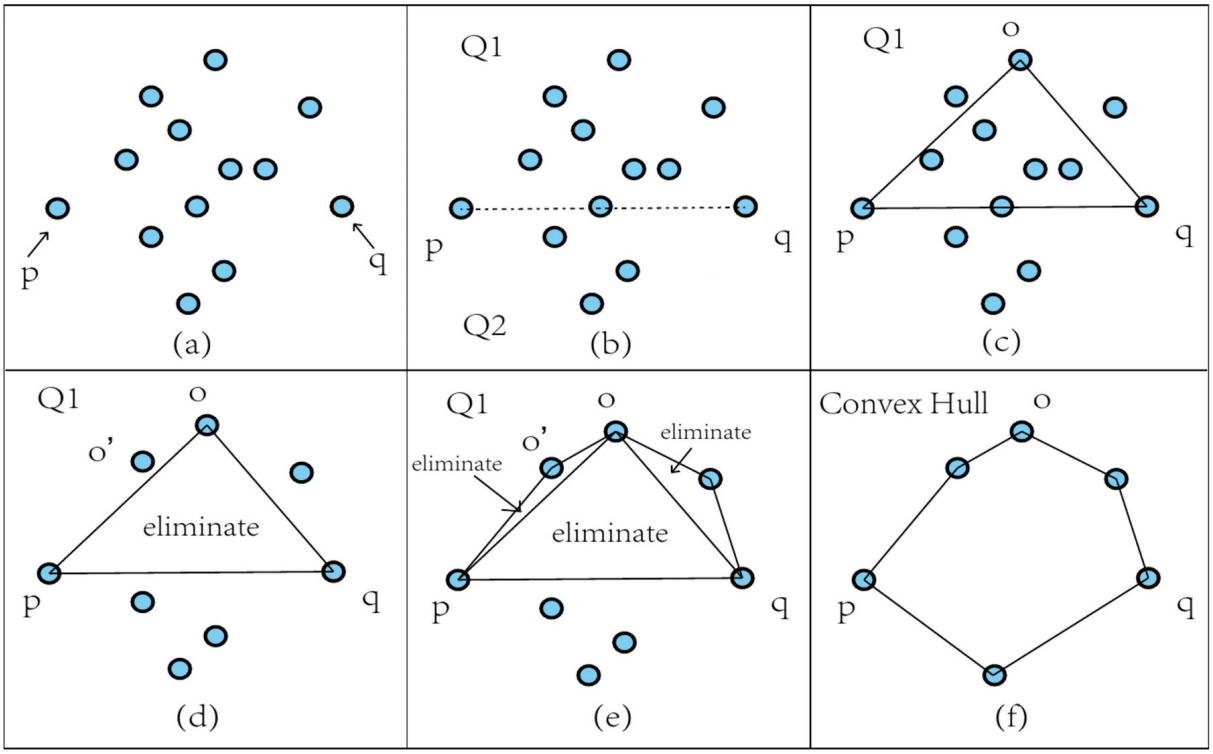


Figura 13 – Etapas do quickhull Fonte: (ZENG; ZHONG; PAN, 2024).

5.2 FILTRAGEM E REFINAMENTO

Em sistemas de simulação física com muitos objetos, o custo computacional da detecção de colisão pode rapidamente tornar-se inviável. Uma abordagem ingênua, baseada na verificação de todos os pares possíveis de objetos, apresenta complexidade $O(n^2)$, inviável mesmo para quantidades moderadas de entidades dinâmicas. Observa-se, entretanto, que em cenários realistas apenas uma pequena fração dos objetos encontra-se efetivamente próxima o suficiente para colidir em um dado instante. A partir dessa constatação, a detecção de colisão é tradicionalmente decomposta em duas etapas conceitualmente distintas: Filtragem (*Broad Phase*) e Refinamento (*Narrow Phase*).

A etapa de Filtragem tem como objetivo reduzir drasticamente o número de pares candidatos à colisão por meio de testes conservadores e de baixo custo computacional. Nessa fase, não se busca determinar com exatidão se dois objetos colidem, mas apenas descartar, com segurança, aqueles pares cuja colisão é geometricamente impossível. Para isso, empregam-se objetos delimitadores simples, como caixas alinhadas aos eixos, em conjunto com estruturas de localização espacial que exploram a coerência espacial da cena. A premissa fundamental é que objetos só podem interagir se estiverem suficientemente próximos.

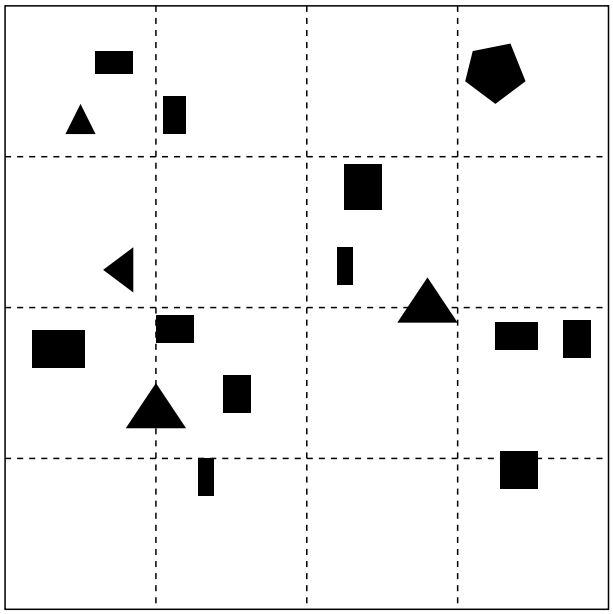
As técnicas de partição espacial constituem um dos principais instrumentos da Filtragem. Essas técnicas subdividem o domínio contínuo da simulação em regiões menores, tipicamente convexas, cada uma associada a um conjunto de referências aos objetos que nela se encontram total ou parcialmente. Dois objetos só podem colidir se compartilharem ao menos uma dessas regiões, o que reduz significativamente o número de testes de pares necessários. Tal redução, embora dependente da distribuição espacial dos objetos, permite alcançar tempo esperado próximo de $O(n)$ ou $O(n \log n)$ em cenários bem comportados.

A literatura apresenta diversas estruturas de dados voltadas à organização espacial, incluindo abordagens hierárquicas como *Quadtrees*, *Octrees*, *Binary Space Partitioning Trees* (BSP), *K-d Trees* e *Ball Trees*. As grades uniformes (*Uniform Grids*) constituem um esquema muito usado de subdivisão espacial.

Uma grade uniforme subdivide o espaço em células de tamanho fixo e igual, associando cada objeto às células que sua caixa delimitadora intercepta, conforme ilustrado na Figura 14. Essa abordagem assume implicitamente que a escala dos objetos é relativamente homogênea, hipótese que influencia diretamente sua eficiência.

Uma implementação direta da grade uniforme poderia empregar uma matriz bidimensional, na qual cada célula armazena uma lista de objetos. Contudo, essa representação apresenta limitações relevantes, como elevado consumo de memória em cenários esparsos, tamanho fixo do mundo e baixa flexibilidade para ambientes potencialmente infinitos. Para contornar essas restrições, adota-se neste trabalho uma abordagem baseada em ordenação linear das chaves de célula por varredura.

Figura 14 – Divisão espacial em grade uniforme



Nessa estratégia, na inicialização da grade, cada célula da grade é identificada por coordenadas inteiras (i, j), obtidas a partir das coordenadas contínuas (x, y) do objeto, segundo:

$$i = \left\lfloor \frac{x}{C} \right\rfloor, \quad j = \left\lfloor \frac{y}{C} \right\rfloor,$$

onde C representa o tamanho da célula. Como neste trabalho um objeto é representado por um conjunto de pontos, não temos acesso direto ao seu centro, é calculado as coordenadas de todas as células interceptadas por sua AABB, garantindo a propriedade conservadora da Filtragem.

As coordenadas são linearizadas em uma chave única da célula (*cell key*) usando uma abordagem eficiente que utiliza um mapeamento linear simples:

$$Key_{cell} = i + j \times C$$

e os pares $(Key_{cell}, Index_{obj})$ são salvos num array, onde $Index_{obj}$ representa o índice do objeto no array original ou referência da memória.

Uma vez construída a grade, a Filtragem consiste em ordenar os pares pela chave da célula. Nessa etapa pode-se utilizar diversos algoritmos de ordenação encontrados na literatura como o *Radix Sort*. Neste trabalho será usado o algoritmo nativo do navegador que depende da implementação, como TimSort para o V8 (Google Chrome) ou o MergeSort para o SpiderMonkey (Mozilla Firefox).

Após a ordenação, os objetos que compartilham a mesma célula aparecem em posições contíguas no array, o quê favorece a localidade espacial e temporal, menos *cache misses* e *page faults*. A partir dessa organização, é possível gerar os pares candidatos à colisão percorrendo o array ordenado e agrupando os objetos por célula. Para evitar testes redundantes, é necessário eliminar duplicatas, se dois objetos A e B caem na mesma

célula é preciso testar apenas o par (A, B) e não repetir para o par (B, A). Essa etapa é essencial para preservar os ganhos de desempenho proporcionados pela filtragem espacial.

Essa abordagem permite armazenar apenas as células efetivamente ocupadas, reduzindo o consumo de memória e permitindo a simulação em espaços conceitualmente ilimitados.

Em ambientes dinâmicos, nos quais objetos se movem continuamente e podem ser criados ou removidos, a manutenção incremental da grade torna-se um aspecto crítico. Embora seja possível atualizar apenas as células afetadas por cada objeto em movimento, essa abordagem exige controle adicional e estruturas auxiliares. Neste trabalho, opta-se deliberadamente por uma estratégia mais simples: a grade é completamente reinicializada a cada passo de tempo. Essa decisão representa um trade-off arquitetural, privilegiando simplicidade e previsibilidade de execução em detrimento de possíveis otimizações incrementais, escolha particularmente adequada ao contexto de execução em JavaScript e ambientes baseados na Web.

O desempenho de métodos baseados em grade uniforme é fortemente influenciado pela escolha do tamanho da célula. Idealmente, cada objeto deveria ocupar aproximadamente uma única célula; nesse caso, em duas dimensões, um objeto pode interceptar no máximo quatro células adjacentes. Segundo Ericson (2004b), escolhas inadequadas para esse parâmetro podem degradar significativamente o desempenho, destacando-se quatro fatores principais:

1. células excessivamente pequenas aumentam o custo de atualização da estrutura;
2. células grandes demais agrupam muitos objetos, reduzindo a eficácia da filtragem;
3. objetos de geometria complexa podem exigir subdivisão adicional;
4. cenários com grande variação de escala demandam abordagens hierárquicas ou híbridas.

Assim, a adoção de uma grade uniforme não constitui uma solução universal, mas uma decisão arquitetural dependente das características da cena simulada.

Ao término da Filtragem, obtém-se um conjunto P de pares candidatos à colisão, com redução significativa do custo computacional em relação ao método quadrático. Ainda assim, no pior caso, quando muitos objetos se concentram em uma única célula, a complexidade pode degradar novamente para $O(n^2)$, evidenciando o caráter heurístico dessa etapa.

A etapa de Refinamento realiza testes geométricos precisos nos pares de candidatos P, essa fase é chamada de Narrow Phase. Utiliza-se algoritmos mais sofisticados para determinar se os objetos estão de fato colidindo, como SAT, GJK e EPA, vistos anteriormente. Essa fase nem sempre é obrigatória pois, sua aplicação pode se tratar de objetos retangulares que caibam justamente nas dimensões de sua caixa delimitadora.

5.3 COMO DETERMINAR O INTERVALO DE INTEGRAÇÃO DA FÍSICA

Em física é preciso estabelecer um valor para o passo de tempo, normalmente chamado de dt (do inglês, *Delta Time*). Isso depende muito, se o seu problema tiver muitos objetos e restrito ao tempo, usa-se um dt alto, mas se não estiver restrito ao tempo usa-se o melhor dt possível para obter uma simulação suave. Em muitos casos não é interessante usar todo tempo da máquina para simulação física, mas também para renderização.

No geral existem dois esquemas de integração física. A forma mais simples é usar uma integração de passo fixo usando sempre o mesmo dt e garante que a simulação seja reprodutível. Outra forma é integração com passo variável, ela é a forma mais comum de realizar integração física pois é sempre computada o mais rápido possível e o dt depende do tempo decorrido do relógio, o problema disso que dependendo da máquina pode fazer mais passos ou menos passos de simulação, em geral fazer mais passos da simulação trás um resultado mais plausível.

5.4 EXECUÇÃO PARALELA E PARALELISMO DE DADOS NA SIMULAÇÃO FÍSICA

A simulação física em aplicações interativas apresenta elevado custo computacional, especialmente quando envolve integração temporal, detecção de colisões e resolução iterativa de restrições. No contexto da Web, esse custo torna-se ainda mais relevante devido às limitações da thread principal, responsável não apenas pela execução do código da aplicação, mas também pela renderização gráfica, tratamento de eventos de entrada e atualização da interface do usuário. Dessa forma, torna-se necessário discutir estratégias de paralelismo que permitam manter a responsividade da aplicação sem comprometer a estabilidade e a previsibilidade da simulação física.

Uma abordagem amplamente adotada consiste na separação conceitual entre as rotinas de renderização e simulação física. Enquanto a renderização é executada na thread principal, geralmente sincronizada com o ciclo de atualização visual por meio de mecanismos como *requestAnimationFrame*, a simulação física pode ser executada em uma thread separada, utilizando *Web Workers*. Essa separação reduz a probabilidade de bloqueios na interface e permite que o cálculo físico seja realizado de forma assíncrona em relação ao processo de desenho.

Nesse modelo, a thread responsável pela simulação apenas escreve o estado físico mais atual e a renderização consome apenas o estado físico mais recente disponível, como ilustrado na Figura 15.

A simulação física apresenta dependências de dados que dificultam sua paralelização completa. Na resolução de colisões e restrições, múltiplas interações podem envolver um mesmo corpo rígido ou partícula. Por exemplo, colisões entre os pares (A, B) e (B, C) compartilham o corpo B, o que exige coordenação cuidadosa para evitar inconsistências.

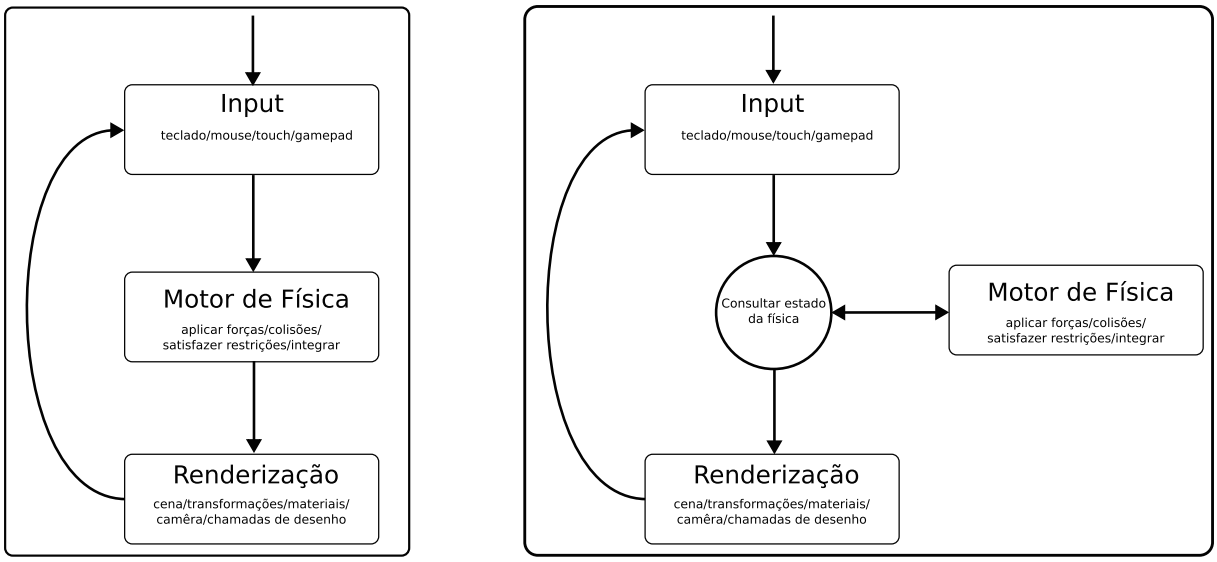


Figura 15 – À esquerda esquema tradicional single-thread. À direita esquema com simulação rodando numa thread dedicada a passo fixo.

No método de Jakobsen (2001), em particular, a ordem de aplicação das restrições influencia o resultado final, tornando a execução paralela ainda mais sensível a variações na ordem de processamento.

Embora a execução da simulação em múltiplas threads pareça uma solução natural para aumentar o desempenho, a programação concorrente introduz uma série de desafios. Um dos principais problemas está relacionado à distribuição de carga (*load balancing*). Em simulações físicas, o custo computacional não é uniforme: regiões do espaço com maior densidade de corpos exigem mais testes de colisão e mais iterações de resolução de restrições, enquanto regiões esparsas demandam pouco processamento. Como essa distribuição varia dinamicamente ao longo do tempo, torna-se difícil dividir o trabalho de forma equilibrada entre várias threads.

Outro desafio fundamental refere-se à consistência dos dados. Em um ambiente multithread, diferentes threads podem acessar e modificar simultaneamente os mesmos estados físicos, como posições e orientações dos corpos. Essa situação pode levar a condições de corrida, resultados inconsistentes e instabilidade numérica. Em simulações físicas, utilizar estados parcialmente atualizados é particularmente problemático, pois pequenas inconsistências podem se propagar e amplificar ao longo das iterações.

Além disso, a necessidade de mecanismos de sincronização, como travas (*locks*), operações atômicas e barreiras, introduz sobrecarga adicional e pode reduzir significativamente os ganhos esperados com o paralelismo. Em muitos casos, o custo de sincronização supera o benefício obtido com a execução paralela, especialmente quando a granularidade das tarefas é fina, como na resolução individual de colisões.

Outro aspecto relevante é o não-determinismo introduzido pela execução concorrente. A ordem de escalonamento das threads pode variar entre execuções, levando a resultados

físicos ligeiramente diferentes mesmo sob condições iniciais idênticas. Esse comportamento dificulta a depuração, a reprodução de simulações e a implementação de funcionalidades como *replays* ou sincronização em aplicações distribuídas.

Por essas razões, muitas engines físicas modernas optam por paralelizar apenas etapas bem definidas e fracamente acopladas da simulação, como a detecção de colisão na fase de Filtragem, a resolução de colisões na fase de Refinamento e a resposta a colisão em um único thread lógico.

Nem todo paralelismo está associado à execução em múltiplas threads. O paralelismo de dados, explorado por meio de instruções SIMD (*Single Instruction, Multiple Data*), permite aplicar a mesma operação a múltiplos elementos simultaneamente dentro de um único fluxo de execução. Esse modelo é particularmente adequado para etapas da simulação física que envolvem operações homogêneas sobre grandes conjuntos de dados, como a integração temporal das posições, a atualização de velocidades implícitas e testes simples de colisão.

Ao contrário do paralelismo baseado em threads, SIMD não introduz condições de corrida nem exige mecanismos complexos de sincronização, uma vez que todas as operações ocorrem de forma determinística dentro de um único contexto de execução. No contexto da Web, o suporte a SIMD em WebAssembly permite explorar esse tipo de paralelismo de forma eficiente, desde que os dados estejam organizados em layouts de memória apropriados, como *Structure of Arrays* (SoA).

O uso de WebAssembly (WASM) oferece vantagens significativas para a implementação de simulações físicas intensivas. Por possuir tipagem estática, memória linear e maior previsibilidade de desempenho, WASM é mais adequado para laços computacionais intensivos do que JavaScript tradicional. Quando combinado com SIMD, um único thread de execução em WASM pode atingir desempenho comparável ou superior a implementações JavaScript multithread, sem incorrer nos problemas de sincronização e não-determinismo associados à concorrência.

Essa abordagem justifica arquiteturas nas quais a simulação física completa é executada em um único worker utilizando WASM, enquanto a thread principal permanece responsável pela renderização e interação com o usuário. Dessa forma, o paralelismo é explorado verticalmente, por meio de instruções vetoriais e melhor uso do hardware subjacente, em vez de horizontalmente, por meio da criação de múltiplas threads concorrentes.

O WebGPU introduz a possibilidade de explorar paralelismo massivo por meio da GPU, sendo particularmente adequado para tarefas altamente paralelizáveis e com baixo acoplamento entre dados. Na simulação física, isso inclui etapas como a detecção de colisão em fase de filtragem, a construção de grades uniformes e a geração de pares candidatos à colisão. Essas operações envolvem grandes volumes de dados independentes e se beneficiam diretamente do modelo de execução da GPU.

Entretanto, a resolução iterativa de restrições físicas, especialmente em métodos como

o de Jakobsen (2001), apresenta dependências complexas que limitam sua adequação à execução na GPU. Assim, o uso de WebGPU deve ser encarado como complementar, e não como substituto direto, da simulação física tradicional executada na CPU.

Com base nas considerações discutidas, uma arquitetura híbrida mostra-se mais adequada para simulações físicas simplificadas na Web. Nessa arquitetura, a renderização é realizada na *main thread*, a simulação física principal é executada em um worker dedicado utilizando WebAssembly, e o paralelismo de dados é explorado por meio de SIMD. Opcionalmente, etapas específicas e fracamente acopladas, como a fase ampla da detecção de colisões, podem ser delegadas à GPU via WebGPU.

Essa abordagem reduz a complexidade associada à programação concorrente, melhora a previsibilidade dos resultados e mantém um equilíbrio entre desempenho e robustez, sendo particularmente adequada para aplicações Web interativas e educacionais.

6 EXPERIMENTOS

Este capítulo apresenta os experimentos realizados com o objetivo de avaliar o desempenho, a estabilidade e a precisão do sistema desenvolvido. Os experimentos também investigam os impactos das otimizações aplicadas nas etapas de *Broad Phase* (utilizando uma grade espacial uniforme), na *Narrow Phase* (empregando SAT e GJK) e na paralelização do cálculo físico por meio de múltiplas threads. O objetivo é verificar a viabilidade dessas técnicas em um ambiente de execução web, onde o custo computacional e a responsividade são fatores críticos.

Os experimentos foram conduzidos em um computador com as seguintes especificações:

- Processador: AMD Ryzen 5 1600X @ 3.3GHz
- Memória RAM: 16 GB DDR4
- Sistema Operacional: Ubuntu 24.04 LTS
- Plataforma de execução: Navegador Firefox 121
- Implementação: Typescript + Web Workers (multi-threading), Vuejs e p5js

6.1 CONFIGURAÇÃO DOS CENÁRIOS

Três grupos de experimentos foram definidos, cada um com foco em um aspecto distinto do sistema proposto:

Experimento 1 — Detecção de Colisões Entre Objetos Convexas

O segundo experimento teve como objetivo comparar os algoritmos de detecção de colisão SAT e GJK em termos de precisão e custo computacional. Foram utilizados objetos convexos de 3 a 8 vértices (em 2D). Cada cenário variou de 2 até 100 objetos móveis, gerando colisões dinâmicas com rotações e translações aleatórias.

Os tempos médios de detecção e a taxa de acertos foram medidos com e sem a utilização de uma etapa de Broad Phase baseada em grade uniforme.

- O algoritmo SAT demonstrou desempenho satisfatório em colisões bidimensionais com poucos vértices.
- A introdução da *Broad Phase* reduziu significativamente o número de pares testados na *Narrow Phase*, resultando em ganho médio de até 65% em desempenho.

Experimento 2 — Simulação Multi-Threaded

O terceiro experimento avaliou os benefícios do uso de concorrência na simulação física. A implementação utilizou a API *Web Workers* para distribuir a atualização das partículas e as verificações de colisão entre múltiplas threads.

A execucação da física em uma thread separada apresentou estabilidade na renderização.

Experimento 3 - Comparação Com Box2D

6.2 RESULTADOS E DISCUSSÃO

Os resultados obtidos indicam que o método de Jakobsen (2001) apresenta um bom equilíbrio entre estabilidade e simplicidade de implementação, sendo especialmente adequado para simulações de tecidos e cadeias articuladas em tempo real.

Os algoritmos SAT e GJK apresentaram comportamentos complementares: o SAT mostrou-se mais simples e eficiente em 2D, enquanto o GJK foi superior para colisões tridimensionais complexas. A combinação de ambos na *Narrow Phase*, precedida pela otimização em grade uniforme na *Broad Phase*, resultou em ganhos expressivos de desempenho sem perda significativa de precisão.

A utilização de múltiplas threads proporcionou melhorias significativas na taxa de atualização da simulação, especialmente em cenários densos com mais de 100 corpos dinâmicos. O gráfico da Figura ilustra a relação entre número de threads e o ganho de desempenho observado.

INSERIR FIGURA

7 CONCLUSÕES

Os experimentos realizados confirmam a viabilidade de um sistema de animação física simplificada, eficiente e estável, totalmente implementado em ambiente web. A combinação entre o integrador de Verlet, as otimizações de detecção de colisão e a execução multi-threaded proporcionou resultados consistentes e visualmente plausíveis em tempo real.

Tais resultados demonstram que é possível construir um motor físico leve e acessível, adequado para aplicações educacionais, jogos independentes e simulações científicas interativas, sem a necessidade de bibliotecas externas ou dependências proprietárias.

Como trabalhos futuros, propõe-se:

- Realizar testes em ambientes 3D
- Implementar hierarquias de volumes limitadores (BVH);
- Migrar a execução paralela para WebGPU Compute Shaders, permitindo simulação massiva em GPU.

REFERÊNCIAS

AZEVEDO, A. C. E. **Computação gráfica - Teoria e prática**. [S.l.]: Editora Campus, Ltda, 2003.

BARBER, C. B.; DOBKIN, D. P.; HUHDANPAA, H. The quickhull algorithm for convex hulls. **ACM Transactions on Mathematical Software (TOMS)**, Acm New York, NY, USA, v. 22, n. 4, p. 469–483, 1996.

CATTO, E. **Box2D: A 2D Physics Engine for Games**. 2011. Acesso em: nov. 2025. Disponível em: https://box2d.org.

COUMANS, E. **Bullet Physics Simulation**. 2015. Acesso em: nov. 2025. Disponível em: https://pybullet.org.

ERICSON, C. **The Gilbert-Johnson-Keerthi (GJK) Algorithm**. 2004. SIGGRAPH Presentation.

ERICSON, C. **Real-time collision detection**. [S.l.]: Crc Press, 2004.

GILBERT, E.; FOO, C.-P. Computing the distance between general convex objects in three-dimensional space. **IEEE Transactions on Robotics and Automation**, v. 6, n. 1, p. 53–61, 1990.

GILBERT, E.; JOHNSON, D.; KEERTHI, S. A fast procedure for computing the distance between complex objects in three-dimensional space. **IEEE Journal on Robotics and Automation**, v. 4, n. 2, p. 193–203, 1988.

Havok. **Havok Physics**. 2024. Acesso em: nov. 2025. Disponível em: https://www.havok.com/havok-physics/.

JAKOBSEN, T. Advanced character physics. In: **Proceedings of the Game Developer's Conference 2001**. San Jose, CA: Game Developers Conference, 2001. Presented at Game Developer's Conference 2001.

LINAHAN, J. **A Geometric Interpretation of the Boolean Gilbert-Johnson-Keerthi Algorithm**. 2015. Disponível em: https://arxiv.org/abs/1505.07873.

MÖLLER, T.; HAINES, E.; HOFFMAN, N. **Real-time rendering**. 4. ed. [S.l.]: CRC Press, 2018.

MÜLLER, M. et al. Position based dynamics. **Journal of Visual Communication and Image Representation**, v. 18, n. 2, p. 109–118, 2007. ISSN 1047-3203. Disponível em: https://www.sciencedirect.com/science/article/pii/S1047320307000065.

MURATORI, C. **Implementing GJK (2006)**. 2006. https://caseymuratori.com/blog_0003. Acessado em: 12 de Outubro de 2025.

NVIDIA Omniverse. **PhysX SDK**. https://github.com/NVIDIA-Omniverse/PhysX. BSD-3 license, Acesso em: nov. 2025.

ROCKAFELLAR, R. T. **Convex Analysis**. Princeton, NJ: Princeton University Press, 1970. (Princeton Landmarks in Mathematics and Physics).

Wikipedia contributors. **Carathéodory's theorem (convex hull) — Wikipedia, The Free Encyclopedia**. 2025. https://en.wikipedia.org/w/index.php?title=Carath%C3%A9odory%27s_theorem_(convex_hull)&oldid=1313647530. [Online; accessed 18-February-2026].

ZENG, T.; ZHONG, C.; PAN, T. Clustering explanation based on multi-hyperrectangle. **Scientific Reports**, v. 14, 12 2024.