DETECÇÃO DE COLISÃO USANDO *SWEEP AND PRUNE* COM K-*DOPS* EM *GPU*

Bruno Gallego Soares do Amaral

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Esperança

Rio de Janeiro
Dezembro de 2021

a

A alguém cujo valor é digno desta dedicatória.

Agradecimentos

A minha mãe, Conceição Aparecida Gallego Soares do Amaral, pela orientação educacional.

A minha esposa, Luana Moraes de Lima, companheira de todas as horas, pelo apoio e incentivo desde o início.

A meu orientador, Doutor Professor Claudio Esperança, por me aceitar como orientando no programa de mestrado da COPPE/UFRJ, por me acompanhar, incentivar e disponibilizar seu tempo e - principalmente - seu conhecimento, além do apoio que sempre demonstrou.

Aos gestores do Tribunal Regional do Trabalho da 1° Região: Rafael Pinho e Juliana Teixeira que flexibilizaram minha carga horária de trabalho viabilizando minha participação no programa de mestrado da UFRJ.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DETECÇÃO DE COLISÃO USANDO *SWEEP AND PRUNE* COM K-*DOPS* EM *GPU*

Bruno Gallego Soares do Amaral

Dezembro/2021

Orientador: Claudio Esperança

Programa: Engenharia de Sistemas e Computação

No ambiente computacional, a detecção de colisão consiste em verificar se dois objetos ocupam o mesmo lugar no espaço virtual, sejam em ambientes bidimensionais ou tridimensionais, portanto, é tema imprescindível em simulações de objetos rígidos, deformáveis e dinâmica de fluidos. Para as simulações que requerem tempo real, normalmente, são utilizados volumes envolventes [8] para aproximação do real objeto em cena, com objetivo de aumentar consideravelmente o desempenho, assim suportando centenas de objetos ou maiores taxas de quadros por segundo. Devido ao alto custo computacional [6] desses algoritmos e técnicas de aceleração, estes são temas de interesse, assim como o uso de *hardware* especializado, como, por exemplo, *GPU*s [25], para tratar do problema com eficiência superior às unidades centrais de processamento (*CPU*s).

São apresentados nesta dissertação experimentos acerca do modelo de detecção de colisão através do algoritmo *Sweep and Prune* [5], visto que estudos recentes [1] comprovaram a efetividade desse algoritmo. Nosso estudo adota os k-Politopos Discretos Orientados (k-*DOP*s) ao invés de caixas alinhadas com os eixos (*AABB*s) ou esferas como volumes envolventes, já que os $k-$*DOP*s melhor aproximam geometricamente objetos com formas irregulares [8], podendo ser mais efetivos na eliminação dos pares que não se encontram em colisão. Além da utilização de *GPU*, com objetivo de extrair melhor performance do paralelismo ofertado por esse *hardware*. Para realizar a simulação dos objetos com resposta às colisões, faz-se necessário aplicar a dinâmica de corpos rígidos, a qual é implementada neste trabalho através da técnica de casamento de formas ([55]).

Palavras-chave: animação baseada em física, detecção de colisão, *sweep and prune*, k-*dop*, *gpgpu*, *meshless deformation*, *shape matching*.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

COLLISION DETECTION USING SWEEP AND PRUNE WITH K-DOPS ON GPU

Bruno Gallego Soares do Amaral

December/2021

Advisor: Claudio Esperança

Department: Systems Engineering and Computer Science

Collision detection is an important consideration for physical simulation of rigid and deformable objects, or even fluids. For each type of simulation and object, different bounding volumes can be used in order to generalize the real object in the scene, with the goal to improve performance and to support huge quantities of objects. Due to the high cost of this procedure, many algorithms and techniques have been investigated, in addition to specific hardware such as GPUs, that tend to be more efficient for this task than CPUs.

In this thesis, an alternative approach to the collision detection model based on the Sweep and Prune algorithm is investigated, where k-Discrete Oriented Polytopes (k-DOPs) are used as bounding volumes instead of the more traditional axis-aligned bounding boxes, since they tend to approximate objects with irregular shapes better. Furthermore, GPUs are very amenable for implementing this approach since the algorithms have a high degree of parallelism. In order to build objects simulation with collisions, it is required to use any rigid body dynamic, whose was chosen meshless deformation based on shape matching technic.

Keywords: physically-based animation, collision detection, sweep-and-prune, k-dop, gpgpu, *meshless deformation*, *shape matching*.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O problema de Detecção de Colisão (*Collision Detection* ou *CD*), é de grande importância para animações baseadas em física (*physically-based animations* ou *PBA*), para determinar de forma precisa ou aproximada se a colisão entre objetos em cena ocorreu, em qual ou quais pontos de contato ocorreram a colisão e determinar a resposta de colisão, ou seja, realizar o tratamento adequado para a separação e dinâmica dos objetos, considerando aspectos físicos, como: momento linear, angular, massa, coeficiente de restituição e possíveis outros fatores físicos do objeto. O tema possui inúmeras aplicações em computação gráfica, seja em aplicações interativas ou não interativas. Dentre as aplicações interativas destacam-se os jogos eletrônicos e as simulações de fenômenos físicos em campos diversos como engenharia e geofísica. Já as aplicações não interativas incluem a confecção de animações para diversas finalidades como entretenimento, educação e propaganda.

A detecção de colisão de todos os pares de objetos em cena tem um alto custo computacional [6], principalmente para aplicações que demandam alta acurácia ou grande número de objetos [6]. Com isto, algoritmos e técnicas são constantemente aprimorados, com objetivo de aumentar o desempenho e suportar maior quantidade de objetos em cena. Diferentes abordagens de algoritmos foram desenvolvidas, a exemplo dos algoritmos [20] baseados em: funcionalidade (*Feature*), simplex (*Simplex*), espaço de imagem (*Image space*) e volume envolvente (*Bounding Volume*).

Dado este fato, na atualidade os algoritmos de detecção de colisão com maior eficiência são baseados em volumes envolventes [1, 20] e por este motivo foram empregados neste trabalho. Esta categoria de algoritmos é comumente dividida em duas ou três fases [40] de acordo com a necessidade. São elas: Fase Ampla (*Broad Phase*), opcionalmente uma Fase Intermediária (*Mid-Phase*), e Fase Detalhada (*Narrow Phase*). A *Broad Phase* é a primeira fase a ser executada. Esta comumente realiza uma verificação ampla, eliminando os pares de objetos consideravelmente afastados, para os quais não existe possibilidade de colisão. Nessa fase são usados volumes envolventes, que são geometrias que englobam e aproximam o objeto real e possuem alto desempenho para verificação de colisão,

pelo fato de geralmente serem geometrias simples. Após esta fase, a *Mid-Phase* pode pré-processar uma geometria complexa gerando uma hierarquia de volumes envolventes ou decompor objetos não convexos em convexos, com objetivo de refinar a fase anterior com uma rápida verificação mais detalhada, assim, ainda mantendo baixo custo computacional. Posteriormente, a *Narrow Phase* executa uma verificação detalhada dos objetos filtrados na(s) fase(s) anterior(es), (*Broad Phase* e/ou *Mid-Phase*), diagnosticando se de fato houve colisão com precisão e identificando os detalhes da colisão, pontos de contato, profundidade de penetração, arestas e faces.

Muitos algoritmos foram elaborados para serem usados na *Broad Phase*. Para auxiliar os algoritmos, volumes envolventes (*Bounding Volumes* ou BV) são geralmente utilizados para criar uma melhor aproximação do objeto real, sendo utilizados comumente na primeira e segunda fase da detecção de colisão (*Broad Phase* e *Mid-Phase*), com objetivo de obter maior performance, visto que o processo de poda será mais eficaz.

Um volume envolvente é geralmente uma forma geométrica simples que contém o objeto real. Assim, se dois volumes envolventes não se intersectam, então, os objetos reais respectivos são também disjuntos. A inversa, entretanto, não é verdadeira, ou seja, se dois volumes envolventes se intersectam, não necessariamente os objetos envolvidos também o fazem e, portanto, um teste mais acurado geralmente também é realizado para este caso. O objetivo é que o teste de interseção entre volumes envolventes seja mais barato computacionalmente que o teste com os objetos reais, que geralmente são malhas triangulares complexas. Dessa forma, pode-se descartar pares de objetos que não se intersectam a um custo relativamente mais baixo. Por outro lado, se a probabilidade de interseção é alta, o teste com volumes envolventes pode piorar o custo total.

Os volumes envolventes comumente empregados são: Caixas Envolventes Alinhadas com os Eixos (*Axis Aligned Bounding Boxes* ou *AABB*), e Esferas. Outras representações também são utilizadas em casos mais específicos, como Caixas Envolventes Orientadas (*Oriented Bounding Boxes* ou OBB), Cápsulas (Capsules), Politopos com k orientações discretas (k*-Discrete Oriented Polytopes* ou k*-DOPs*) ou Fecho Convexo (*Convex Hull*).

Estes variados volumes são utilizados pelo fato de que quanto maior a aproximação da geometria com o objeto real, maior será a acurácia de acerto de colisão propriamente dita na *Narrow Phase*. Quando a aproximação do volume envolvente não é muito fiel, pode acontecer que, na *Broad Phase*, muitos pares de colisão sejam identificados para serem verificados na próxima etapa, sendo que muitos não colidem de fato, gerando um maior custo computacional indesejável. Para exemplificar, imagine que sejam utilizados apenas esferas como volume envolvente para todos os objetos reais em cena. Para uma bola de basquete o volume esfera se engloba perfeitamente na bola, entretanto, para a tabela da cesta de basquete o volume esfera se engloba inapropriadamente, pois a tabela de basquete pode ser discretizada e simplificada geometricamente como um paralelepípedo. Sendo a tabela envolvida por uma esfera, sobraria muito volume envolvente para pouco espaço

ocupado no envólucro. Ao verificar a colisão destes objetos, quando a bola está próxima do cesto ou da tabela, podem ser identificadas muitas falsas colisões, consequentemente sempre sendo processada a fase seguinte. Em outro ponto de vista, se forem usados volumes como esfera para a bola ou um k-*DOP* e para a tabela de basquete um *AABB* ou k-*DOP*, a precisão de acerto na *Broad Phase* é aumentada consideravelmente.

Em complemento aos volumes envolventes estão os algoritmos de detecção de colisão, que fazem uso das aproximações dos volumes envolventes para acelerar a detecção de colisão de milhares de objetos. Neste contexto, aplicam-se os algoritmos Varrer e Podar (*Sweep and Prune* ou *SaP*) [1][2], também conhecidos como Ordenar e Varrer (*Sort and Sweep*), Grades Uniformes (*Uniform Grids*) [7] e Hierarquias de Volumes Envolventes (*Bounding Volume Hierarchies*) [8].

O *Sweep and Prune*, de forma geral, consiste em criar uma lista de volumes envolventes ordenados ao longo de um único eixo, trazendo o problema para o universo unidimensional e, posteriormente, varrer este eixo verificando se os intervalos limites desses volumes estão sobrepostos. Em caso positivo, apenas uma lista de objetos que se intersectam no primeiro eixo são mantidos. O processo se repete para os demais os eixos base do sistema de coordenadas. Os objetos que se mantiverem na lista após a verificação de todos os eixos são exatamente os objetos que possuem seus volumes envolventes ocupando um mesmo espaço, ou seja, em colisão. Dependendo do tipo de volume envolvente, o *SaP* realiza diferentes varreduras de eixos. No caso de esferas, o algoritmo ordena utilizando ou ponto central da esfera ou pontos tangente à esfera e um ou vários eixos são varridos, visto que para qualquer eixo de projeção pode haver um plano separador entre o par de esferas. Portanto, para o algoritmo concluir que dois volumes do tipo esfera não se intersectam, seria necessário verificar todos os eixos perpendiculares à tangente da esfera, o que não é prático, por questões de desempenho computacional. Logo, este algoritmo pode apenas concluir que o par de esferas se intersecta quando todos os eixos de projeção selecionados são varridos. No caso dos volumes *AABB*, geralmente se utiliza o ponto mínimo da *AABB* para a ordenação e são necessários apenas três eixos de varredura, um para cada intervalo mínimo e máximo da *AABB*. Neste caso, o algoritmo conclui com precisão se os volumes se intersectam ou não. No caso das *OBB*s, temos as mesmas limitações dos testes com esferas. Como k-*DOPs* são generalizações de *AABB*s, temos um algoritmo análogo, sendo a ordenação e a varredura, ambas realizadas nas k orientações do politopo discreto. Devido ao alto grau de paralelismo do *SaP*, este algoritmo é bem empregado em ambientes que tomam proveito desse recurso.

Um melhoramento já conhecido nesse algoritmo é a seleção do primeiro eixo que tende a eliminar/podar a maioria dos objetos. Uma dessas técnicas é conhecida como *análise de componente principal* ou PCA (do inglês *Principal Component Analysis*) [2].

PCA é uma das técnicas mais antigas e conhecidas em análise multivariada, introduzida por *Pearson* (1901), e desenvolvida por *Hotelling* (1933) [14]. A ideia principal do

PCA é reduzir a dimensionalidade do conjunto de dados, quando há grande número de variáveis inter-relacionadas, mas mantendo o máximo possível de variação no conjunto de dados. O cálculo da componente principal se resume a solução de um problema de autovalores e autovetores para uma dada matriz de covariância. Apesar de relativamente simples, tem grande variedade de aplicações, bem como diferentes variações. No contexto de detecção de colisão utilizando *SaP*, a *PCA* é usada para estimar um eixo que melhor separa os volumes envolventes.

Além da *PCA*, o *SaP* também explora a coerência temporal através da ordenação. Dado que os objetos em cena têm baixo deslocamento, os objetos mantêm-se quase ordenados em quadros sucessivos. Por este motivo outros algoritmos de ordenação que fazem uso da coerência temporal podem ser utilizados, a exemplo do *Insertion Sort*.

Outra classe de algoritmos relevantes para detecção de colisão é a que reúne os algoritmos baseados em grades. Estes consistem em discretizar o espaço em uma grade virtual consideravelmente pequena em relação aos objetos, de forma que cada célula (ou *voxel*) da grade – um quadrado ou cubo – contenha uma pequena porção do objeto real. Esta grade normalmente é virtual, ou seja, apenas as células ocupadas são efetivamente armazenadas. Este processo é chamado de voxelização [7]. Um problema dessa classe de algoritmos se refere ao tamanho da grade em função do tamanho dos objetos em cena. Caso a grade seja muito maior que algum objeto, perde-se precisão na detecção de colisão. Por outro lado, se a grade for muito pequena em relação a objetos grandes, haverá precisão mais alta, entretanto alto custo de processamento para processar todas as célula. O algoritmo funciona de forma iterativa e incremental. Cada objeto na cena é inserido na grade através do processo de voxelização. Uma tabela de espalhamento (*hash table*) é geralmente utilizada para registrar quais *voxels* estão sendo ocupados pelo objeto. Quando os demais objetos são inseridos, antes de serem incluídos na tabela de espalhamento, é verificado se o *voxel* se encontra vazio na tabela. Caso já esteja preenchido, então é identificado que este novo objeto está ocupando o mesmo espaço que algum outro já existente, formando um par de colisão. O processo se repete para todos os objetos em cena e como resultado obtém-se uma lista de pares em colisão.

Um terceiro algoritmo relevante na detecção de colisão, principalmente para aplicações envolvendo a técnica de traçado de raios (*Ray Tracing*) é a Hierarquia de Volumes Envolventes (*Bounding Volume Hierarchy* ou *BVH*). Uma *BVH* é uma árvore n-ária, na qual os nós são compostos de volumes envolventes, podendo envolver uma geometria complexa. Esta estrutura não é feita para detecção de grande quantidade de objetos, mas sim para pares de objetos complexos ou não convexos, se enquadrando bem na *Mid-Phase*. Assim, quando a raiz de duas *BVH*s colidem, ambas as árvores são percorridas no nível imediatamente inferior e novamente é verificado se os objetos filhos se intersectam. Uma grande vantagem dessa estrutura é sua adaptação ao traçado de raios (*Ray Tracing*) em geometrias complexas. Inicialmente é verificado se o raio intersecta apenas um único volume; o

nó raiz. Em caso negativo, o processo de detecção é interrompido com o diagnóstico de que não há colisão. Caso contrário, o raio é verificado com os nós filhos. Este processo é realizado até não haver colisão ou todos os nós serem verificados.

Outra técnica relevante, quando performance e paralelismo são requisitos, é conhecida como *GPGPU* (*General Purpose computing on Graphics Processing Units*) [25], que consiste na utilização de unidades de processamento gráfico para propósitos gerais, ou seja, não se limitando à renderização de gráficos [25] [34]. Na atualidade, as placas gráficas (*GPU*s) contêm milhares de processadores para realizar cálculos, processamento de imagem e inteligência artificial, utilizando o modelo de paralelismo "Única Instrução, Múltiplos Dados"(*SIMD - Single Instruction, Multiple Data*) [38], que consiste em processar a mesma instrução em todos os processadores, porém, com dados diferentes.

Ademais, as *GPU*s modernas comumente possuem grande quantidade de memória volátil permitindo que grande volumes de dados possam ser armazenados e processados, assim possibilitando o armazenamento de grande quantidade de volumes envolventes. Diante disto, as *GPU*s se mostram atraentes para o processamento de detecção, explorando o potencial de computação em placas gráficas modernas.

Após a seleção e execução dos algoritmos e técnicas da *Broad Phase*, uma análise detalhada da colisão é geralmente aplicada. Neste sentido, destaca-se o [56] [8] algoritmo de distância *Gilbert–Johnson–Keerthi* (*GJK*), no qual determina, de forma eficiente, a distância mínima de dois fechos convexos, verificando se de fato os objetos estão em colisão de forma discreta, podendo ser adaptado para forma contínua [8]. Este algoritmo inicia criando a diferença de *Minkowski* com os dois fechos convexos, gerando um terceiro fecho convexo. Caso o novo fecho contenha a origem em seu interior, significa que os objetos estão em colisão.

Sendo identificada a colisão dos objetos, é necessário realizar a separação dos mesmos para que a simulação seja próxima da realidade, portanto o algoritmo [58] *Expanding Polytope Algorithm* (*EPA*) é utilizado em conjunto com o *GJK* para identificar os detalhes da colisão, como a profundidade de penetração dos objetos envolvidos e a normal da colisão. Para atingir esse objetivo, o *EPA* recebe como entrada um *simplex* (um tetraedro criado anteriormente pelo *GJK*) e pesquisa pela aresta mais próxima da origem, pois esta contém exatamente os detalhes da colisão.

Com todos os dados da colisão obtidos, é possível realizar a separação ou dinâmica dos corpos envolvidos. Diversas técnicas de dinâmica de colisão foram abordadas para diferentes tipos de corpos (rígidos, deformáveis ou fluidos), como técnicas baseada em impulso, complementariedade [62], casamento de formas [55]. A técnica descrita em [55] é baseada em casamento de formas. Esta recebe um par de objetos e deforma ambas as geometrias dos objetos, separando-os. Posteriormente, é realizado o casamento ou associação da forma deformada para a forma original do objeto, realizando desta forma a translação e rotação necessária para que o objeto retorne à malha original. Este pro-

cesso é realizado em *n* iterações para convergência global de todos os objetos em cena. Este método se destaca por suportar tanto corpos rígidos quanto deformáveis, além da simplicidade de implementação e sua estabilidade.

Diante do exposto, o presente trabalho apresenta um experimento complementar para detecção de colisão na *Broad Phase* por ser uma fase crucial na eficiência da detecção de colisão. O algoritmo *SaP* se apresenta eficiente na *Broad Phase* com a utilização de esferas [2] e *GPU*s para o processamento do algoritmo. Neste caso, [2] utilizou a técnica de Análise da Componente Principal (*PCA - Principal Analysis Component*), identificando o melhor eixo de varredura para a execução do *SaP*, e varrendo os objetos ao longo desse único eixo para verificar a colisão entre esferas conforme o "Teorema do Eixo Separador" (*SAT - Separating Axis Theorem*), que é detalhado no próximo capítulo. Em [1] também utilizou-se o *SaP* com *AABB*s em paralelo usando *multithreading* em *CPU* apresentando bons resultados, além de utilizar também uma medida relativa de dispersão para verificar qual melhor eixo de varredura a ser verificado primeiro. Ademais, foi utilizado varredura em dois eixos, sendo o terceiro verificado caso os dois eixos apresentassem colisão.

O presente trabalho propõe um aprimoramento da abordagem de detecção de colisão utilizando o algoritmo *SaP* em *GPU* através do emprego de k-*DOP*s, visto que estes permitem construir volumes envolventes bem ajustados aos objetos reais, com custo computacional relativamente baixo. Isto implica em implementar e adaptar o *SaP* para varrer as k orientações dos politopos discretos, adicionando heurística para identificar um suposto melhor eixo para iniciar a varredura, a exemplo da Análise da Componente Principal (*PCA*) [50], obtendo maior eficácia no processo de filtragem. Pretende-se com isso obter maior desempenho no processo de detecção de colisão visto que a filtragem da *Broad Phase* deverá será maior que aquela usando os demais volumes envolventes apresentados nas pesquisas anteriores. Para realização completa da simulação, foram utilizados os algoritmos [56] [57] *GJK Enhanced* em conjunto com o [58] *EPA* na *Narrow Phase*. Sendo na dinâmica da colisão utilizado [55] técnicas de deformação sem malhas baseadas em associação de formas.

1.1 Objetivos

1.1.1 Objetivo Geral

Experimentar o algoritmo de detecção de colisão *SaP* em *GPU* sobre geometrias não triviais utilizando diferentes volumes envolventes.

1.1.2 Objetivos Específicos

- Realizar uma análise exploratória do comportamento do algoritmo *SaP* com k-*DOP* em *GPU*;
- Comparar o algoritmo utilizando variações de volumes envolventes, como esferas, *AABB*s e k-DOPs.
- Avaliar o tempo de desempenho do *Sweep and Prune* em relação à quantidade de pares de eliminados de acordo com o volume envolvente.
- Avaliar a relevância da utilização da *PCA* com o algoritmo *SaP*.

Capítulo 2

Trabalhos Relacionados

Neste capítulo são abordados os algoritmos de detecção de colisão mais relevantes para o tema da pesquisa, assim como técnicas de aceleração e dinâmica de corpos cabíveis.

2.1 Detecção de Colisão

Em simulação física, o uso de detecção de colisão é ponto fundamental para facilitar o trabalho de animação dos objetos. Através da detecção de colisão é verificado se objetos estão colidindo, ou seja, se estes se intersectam ou ocupam o mesmo volume no espaço, quer sejam os corpos rígidos, deformáveis ou fluidos. Neste contexto, obtêm-se também as características da colisão, como os pontos de contato, profundidade de penetração, velocidade dos corpos e direção dos movimentos lineares e angulares. A detecção de colisão pode ser classificada em dois tipos: detecção de colisão discreta e detecção de colisão contínua, também chamada de colisão dinâmica [8]. No tipo discreto, dada a posição corrente dos objetos, verifica-se se ocupam o mesmo espaço no ambiente, independentemente do deslocamento. No tipo contínuo, verificam-se as posições atual e anterior dos objetos, considerando seu deslocamento entre os quadros de renderização, ou seja, para cada quadro renderizado, obtém-se o deslocamento, e geralmente um volume envolvente como uma *AABB* é criado para englobar esse deslocamento, permitindo assim que se verifique a colisão entre os volumes. Isto ocorre para identificar se houve alguma perda no processo de detecção de colisão devido à rápida movimentação dos objetos. Por exemplo, considere a Figura 2.1, imaginando uma esfera se movendo da direita para a esquerda ao encontro de uma parede fina. Observa-se que, devido a rápida movimentação da esfera, o terceiro quadro não identifica que houve uma colisão da esfera com a parede, ocorrida entre os quadros 2 e 3. Este fator ocorre devido à colisão ser discreta. Na Figura 2.2, entretanto, o volume do deslocamento da esfera é traçado, formando uma *AABB*, assim a colisão contínua determina que houve uma colisão entre os quadros 2 e 3 considerando o deslocamento da esfera.

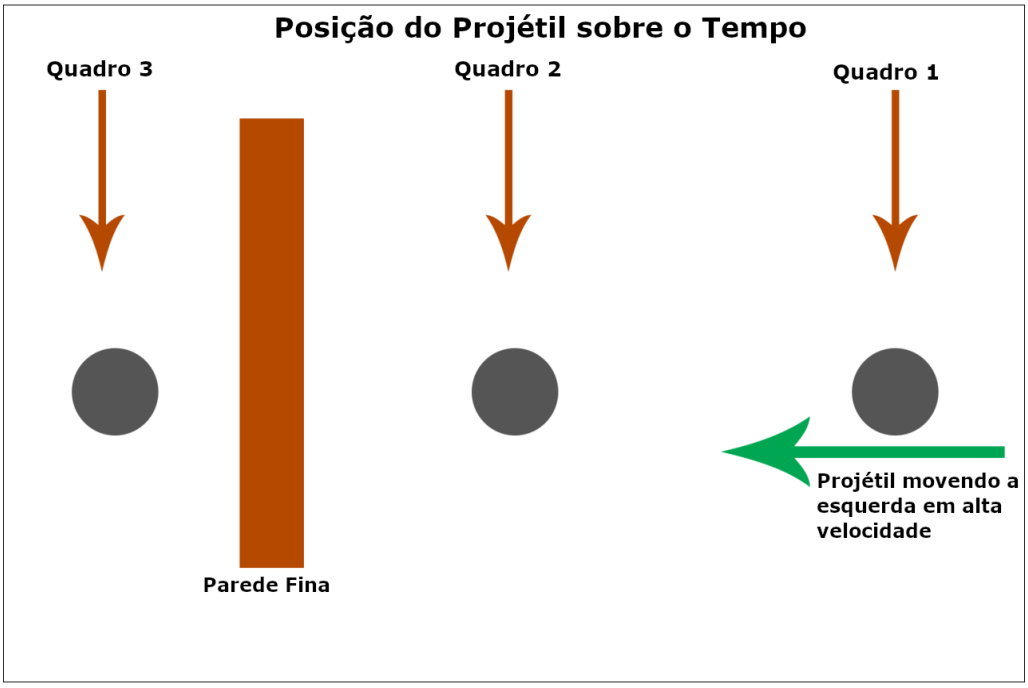

Figura 2.1: Verificação de colisão de colisão discreta (ilustração retirada de [39]).

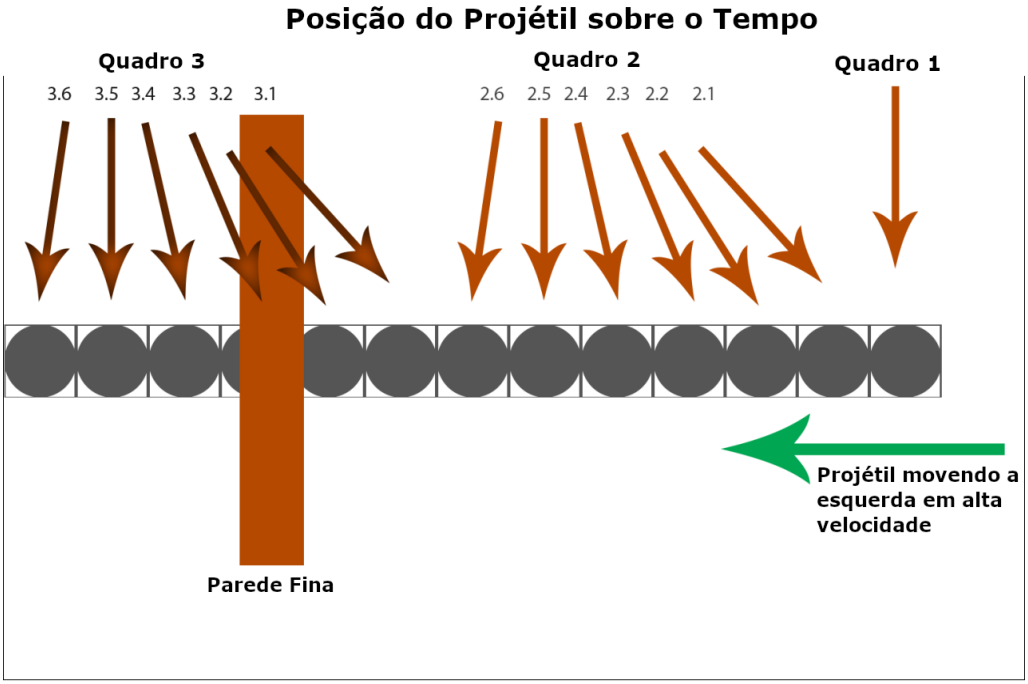

Figura 2.2: Verificação de colisão de colisão contínua (ilustração retirada de [39]).

2.2 Técnicas de Aceleração

Neste capítulo são destacadas as principais técnicas e um processo de aceleração para a detecção e resposta à colisão. As otimizações podem ser feitas em nível de processo, volume envolvente, hierarquia de volumes envolventes, algoritmos e subdivisão espacial.

2.2.1 Processo de Aceleração

Um algoritmo trivial para detecção de colisão consiste em verificar todos os objetos com relação a todos os demais objetos, ou seja, um algoritmo de complexidade assintótica quadrática (n^2). Se a quantidade n de objetos é muito grande, teríamos um custo computacional muito alto, degradando o desempenho e comprometendo uma aplicação executada em tempo real. Sendo assim, processos para detecção de colisão foram aprimorados e as abordagens para o problema geralmente consistem em três fases [40], sendo uma delas opcional.

A primeira etapa é a Fase Ampla (*Broad Phase*), consistido de uma rápida eliminação dos objetos que estão consideravelmente distantes. Nesta fase, comumente são usados volumes envolventes. Cada objeto é envolvido por um volume, preferencialmente que se ajuste bem ao objeto real e com complexidade geométrica baixa de forma a suportar testes de interseção rápidos. Assim, a detecção de colisão pode ser realizada com baixo custo computacional eliminando a necessidade de testar exatamente pares de objetos que garantidamente não podem estar em colisão. Por este motivo esta fase é muito explorada em pesquisas.

A fase seguinte é uma fase opcional denominada Fase Intermediária (*Mid-Phase*), que consiste em obter os pares que foram diagnosticados como em colisão na primeira fase e realizar um refinamento para verificar se de fato é possível que os objetos estejam colidindo. Para exemplificar, imagine dois personagens humanos em cena e que a *Broad Phase* tenha identificado uma colisão entre eles. Na *Mid-Phase* esse par é recebido e podem ser geradas hierarquias de volumes envolventes, separando cada membro dos personagens em vários volumes envolventes mais precisos e a verificação de colisão é realizada a partir desses novos volumes gerados na *Mid-Phase*.

Caso não seja descartada a colisão nas duas primeiras fases, um par de objetos é passado para última fase, chamada Fase Detalhada (*Narrow Phase*. Nessa última fase, geralmente é realizada uma verificação detalhada das geometrias reais dos objetos. Portanto, após todas as fases verificação de colisão, os pares de objetos são encaminhados para o módulo de *resposta à colisão*.

No módulo de resposta à colisão são calculados os pontos de contato, a velocidade dos objetos, momento linear e angular usando as propriedades dos objetos tais como massa, coeficiente de restituição, fricção e outras características que podem ser incluídas para simular a dinâmica de corpos pretendida pelo animador. Assim, a resposta pode consistir em separar objetos, deformar objetos, como os feitos de papel ou tecido, ou mesmo realizar fraturas no objeto, como por exemplo objetos de material quebradiço como vidro ou porcelana. A Figura 2.3 ilustra a sequência de passos (*pipeline*) comumente usada em aplicações típicas de animação física.

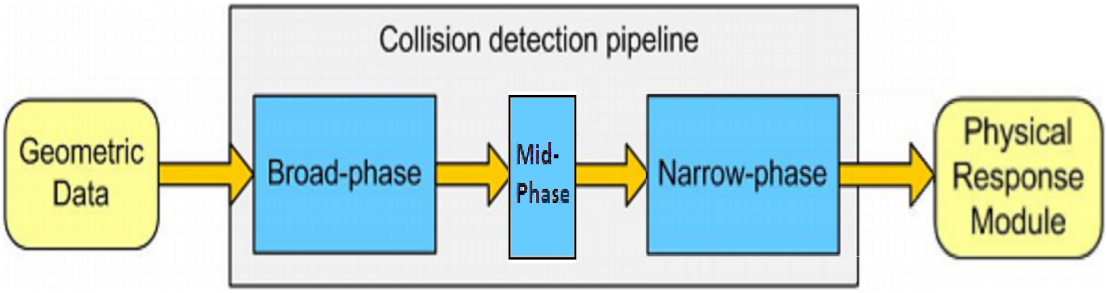

Figura 2.3: Sequência de passos comumente usada em aplicações de animação física.

2.2.2 Volumes Envolventes

Os volumes envolventes são geometrias simples que envolvem uma outra geometria complexa ou até mesmo uma hierarquia de volumes envolventes, ou seja, um volume envolvente englobando vários outros volumes recursivamente. Os volumes envolventes têm por objetivo aproximar um objeto real complexo a uma forma que facilite o cálculo de colisão, tornando-o mais eficiente para uma rápida detecção de colisão. Podem ser representados por diversos tipos de geometrias: esferas [30], caixas alinhadas com os eixos (*AABBs – axis-aligned bounding boxes*) [28], caixas orientadas (*OBBs - oriented bounding boxes*) [31], politopos com k orientações (k*-DOPs –* k*-discrete orientation polytopes*) [29], cápsulas [32] e fechos convexos [33]. Geralmente, quanto mais complexa é a representação do volume envolvente, maior será o custo computacional para calcular a colisão. Em contrapartida, mais próxima da representação do objeto real geralmente esta será. Assim, pesquisas com diferentes volumes em diferentes cenários são realizadas para diagnosticar quais volumes melhor se enquadram dadas as características dos objetos em cena. A Figura 2.10 no fim desta seção ilustra de forma exemplificativa os volumes envolventes e suas características.

A esfera é um dos volumes envolventes com maior relevância devido ao baixo custo computacional no teste de colisão, que requer apenas calcular se a distância D entre os centros das duas esferas é maior que a soma dos dois raios R_a e R_b. Para melhor desempenho não é utilizada a distância euclideana, mas sim a distância quadrática de todos os vetores, pelo fato da distância euclideana utilizar a função raiz quadrada, que tem custo computacional alto. Seja A o ponto central da esfera A, B o ponto central da esfera B e $D = A - B$, a colisão é identificada se e somente se

$$D_{\mathrm{x}}{}^2 + D_{\mathrm{y}}{}^2 + D_{\mathrm{z}}{}^2 \geq r_{\mathrm{a}}{}^2 + r_{\mathrm{b}}{}^2.$$

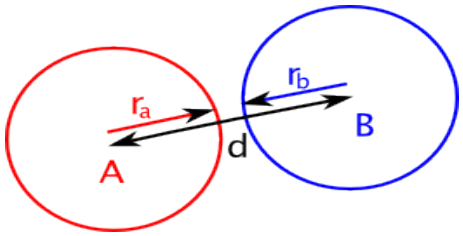

Figura 2.4: Cálculo de colisão entre esferas.

Esferas são bastante eficientes para representar objetos esféricos ou ovais, além de também serem muito utilizadas em simulações usando partículas. Por outro lado, são pouco indicadas para objetos compridos e finos, como um bastão ou alfinete, por exemplo, devido aos falsos positivos de colisão gerados por esse volume.

*AABB*s são caixas envolventes com faces alinhadas aos eixos/vetores base do sistema de coordenadas. São bem competitivas às esferas devido ao baixo custo computacional de calcular colisões. Possuem grande efetividade para representarem objetos cúbicos, no caso de três dimensões e retangulares em ambientes bidimensionais. O cálculo de detecção de colisão entre *AABB*s consiste em verificar se, para todos os eixos, o intervalo entre a coordenada mínima e máxima de uma *AABB* tem interseção não nula com o intervalo análogo da outra AABB. Isto pode ser visto também como a projeção de todos os vértices das *AABB*s em um eixo e verificar se eles se intersectam. Esse procedimento é conhecido como Teorema do Eixo Separador (*SAT - Separating Axis Theorem*) [51]. Este teorema define que, dado dois corpos convexos no espaço, estes não se colidem caso exista um eixo que contenha um hiperplano perpendicular que os separe. O Teorema do Eixo Separador é aplicado nos três eixos base do sistema de coordenadas (X, Y e Z). Caso os intervalos se intersectem nos três eixos, indica que não é possível encontrar um plano que separe os objetos em questão, portanto a colisão entre *AABB*s existe. A Figura 2.5 ilustra exemplos sem colisão e com colisão de *AABB*s.

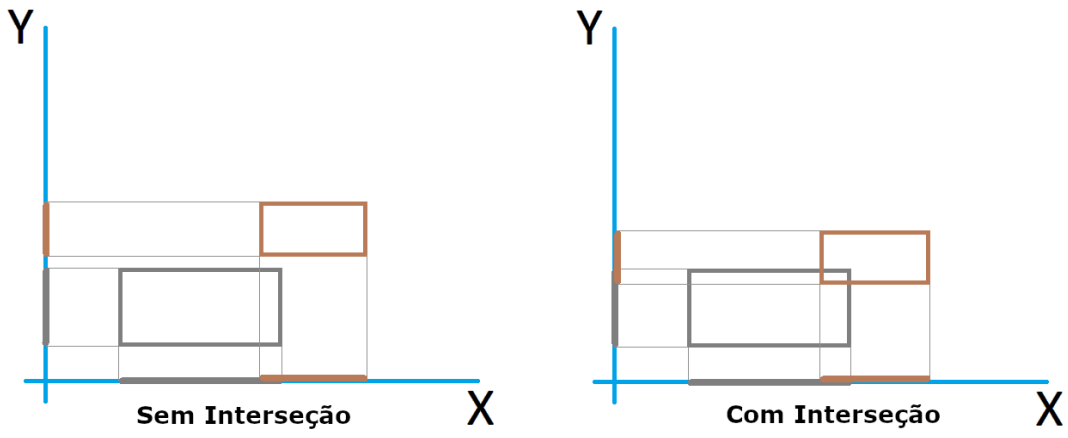

Figura 2.5: Verificação de colisão entre duas *AABB*s (ilustração retirada de[42]).

OBBs são paralelepípedos como as *AABB*s, porém suas faces possuem orientação livre, não necessariamente alinhadas com os eixos do sistema de coordenadas. Sua complexidade computacional na verificação de colisão é considerável se comparada com os volumes mencionados anteriormente. Geralmente é aplicado o *SAT* para cada eixo de orientação de ambas as *OBB*s, sendo necessário verificar seis eixos de projeção para cada par de *OBB*. Quando é utilizada a verificação de colisão para várias *OBB*s simultaneamente, geralmente, varrem-se apenas os eixos alinhados aos eixos base, portanto, a verificação deixa de ser precisa e elimina apenas os pares consideravelmente afastados. *OBB*s são bons volumes para envolver objetos cúbicos com orientações quaisquer. A Figura 2.6 ilustra um exemplo de detecção de colisão entre duas *OBB*s utilizando o *SAT*.

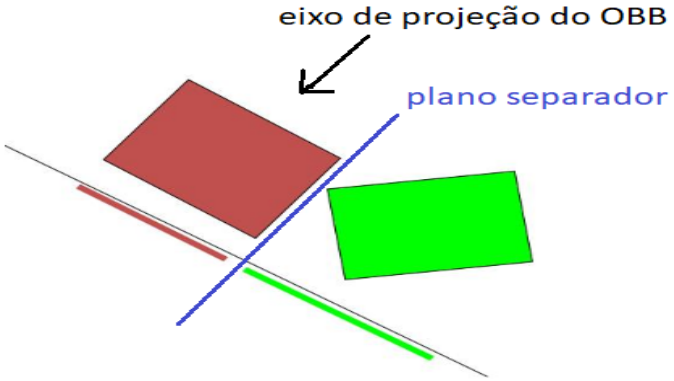

Figura 2.6: Verificação de colisão entre duas *OBB*s.

Os k-*DOPs* são k-politopos orientados discretos contendo n orientações, no qual cada orientações é composta por 2 politopos (mínimo e máximo), formando uma geometria convexa. Assim, por exemplo, uma *AABB* é um *6-DOP* em $\Re^3$, sendo cada politopo perpendicular a um vetor base do sistema de coordenadas. O cálculo de detecção de colisão se assemelha ao mesmo usado para *OBB*s visto anteriormente. Para cada eixo de orientação do k-*DOP* é verificado se existe um plano de parador através do *SAT*. Destaca-se também que há uma divergência literária na definição de k-DOP, no qual alguns autores definem k como o número de orientações. Portanto para alguns autores a *AABB* consiste em um *3-DOP*. A vantagem é que esses eixos são geralmente fixos para todos os objetos, assim, a varredura em todos os eixos fixados se aplica a todos os objetos. A Figura 2.7 exibe dois exemplo de k-*DOP*s.

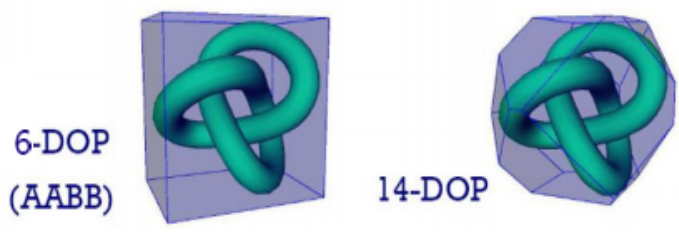

Figura 2.7: Generalização da *AABB* com k-*DOP* (ilustração retirada de [43]).

O fecho convexo (*convex hull*) é um conjunto de politopos que melhor englobam um objeto poliedral. Semelhante ao k-*DOP*, o fecho convexo também possui n orientações de planos, porém, se diferencia porque essas orientações não são fixas, podendo cada objeto, particularmente, conter um número variável de politopos em qualquer orientação. Sendo convexos, esses objetos admitem a utilização do Teorema do Eixo Separador (*Separating Axis Theorem* ou *SAT*). Como se observa nas Figura 2.8 e Figura 2.9, um objeto convexo (em rosa) e um objeto não convexo (em azul), não satisfazem o Teorema do Eixo Separador, visto que não é possível encontrar um plano separador entre os dois objetos, mesmo observando que esses não colidem.

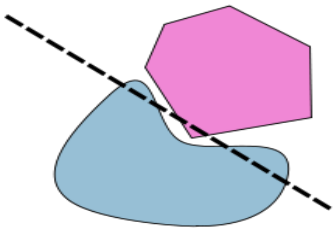

Figura 2.8: Objetos não convexos que não colidem podem não admitir um plano separador (ilustração retirada de [41]).

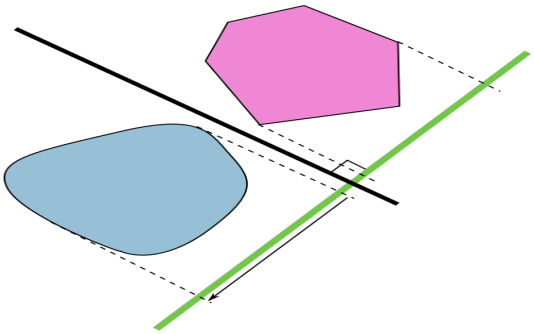

Figura 2.9: Detecção de colisão de dois fechos convexos (ilustração retirada de [41]).

Considerando todos os volumes envolventes mencionados neste capítulo, observe que, de forma geral e dependendo dos objetos em cena, quanto mais justo é o volume envolvente, melhor será a eliminação de pares de objetos efetuada na Fase Ampla (*Broad Phase*). Por outro lado, quanto menos justo é o volume envolvente, maior performance será obtida, devido à simplicidade da geometria. A Figura 2.10 ilustra os volumes envolventes abordados neste capítulo.

Figura 2.10: Representações de Volumes Envolventes (ilustração retirada de [8]).

2.2.3 Subdivisão Espacial

Técnicas de subdivisão espacial aceleram o processo de detecção de colisão por desconsiderarem objetos que estão relativamente distantes e paralelizando a execução dos algoritmos.

Octrees e *Quadtrees*

O objetivo destas subdivisões é representar o espaço por um quadrado (*quadtrees* - aplicadas em ambientes bidimensionais) ou cubo (*octrees* - aplicadas em ambientes tridimensionais) [36] que é subdividido recursivamente em 4 quadrados ou 8 cubos de tamanho uniforme. A subdivisão é feita de forma que cada quadrado/cubo contenha um conjunto relativamente baixo de objetos. Isto permite a paralelização do trabalho visto que cada conjunto pode ser processado de forma independente. Entretanto, objetos que interceptam mais de uma partição (quadrado/cubo) devem ser incluídos em todos os conjuntos correspondentes.

Na Figura 2.11 é possível observar um exemplo de subdivisão espacial hierárquica utilizando *Octrees*. Neste exemplo, o espaço é subdividido em oito regiões no primeiro nível da árvore. Posteriormente, para cada nó do segundo nível, ocorre uma outra subdivisão em oito regiões, sendo assim recursivamente até o nível de granularidade desejado. No último nível, os objetos separados nas regiões, denominadas *voxels*.

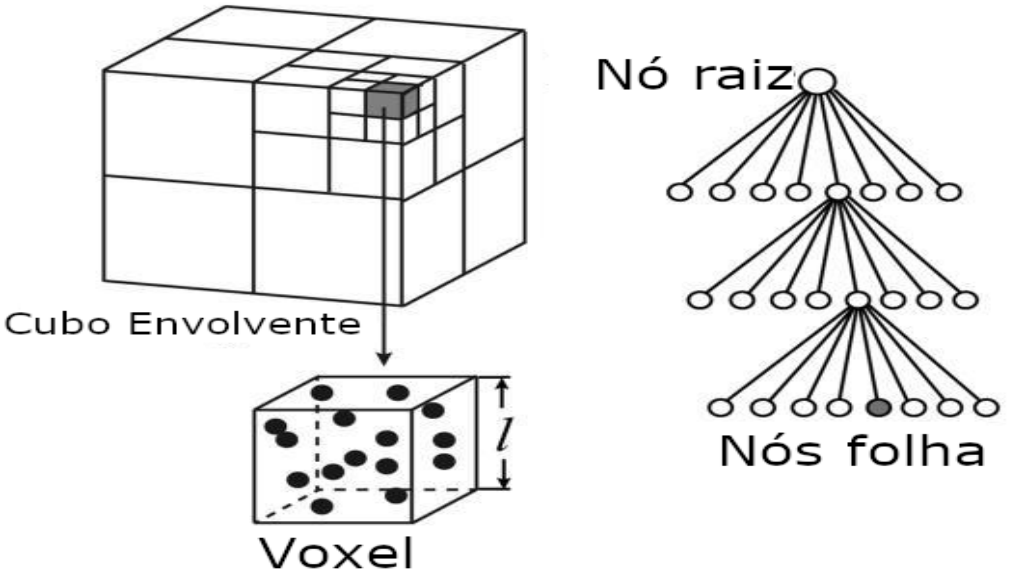

Figura 2.11: Exemplo de Subdivisão Espacial usando *Octree* (ilustração retirada de [16]).

Partição Binária do Espaço

As árvores de partição binária do espaço (*Binary Space Partition Trees* ou *BSP-Trees*) [36] decompõem recursivamente o espaço através de hiperplanos que podem ter posições e orientações quaisquer, sendo portanto mais flexíveis do que as quadtrees e octrees. As partições resultantes, entretanto, podem ser politopos convexos quaisquer, ao invés de quadrados ou cubos.

Na Figura 2.12, observa-se a região *A* sendo decomposta em duas regiões, *B* e *C*. A região *C* é subdividida em *D* e *E*, respeitando sempre a hierarquia das divisões anteriores. Com esta estrutura, rapidamente é possível identificar em qual região um ponto se encontra, eliminando outras regiões.

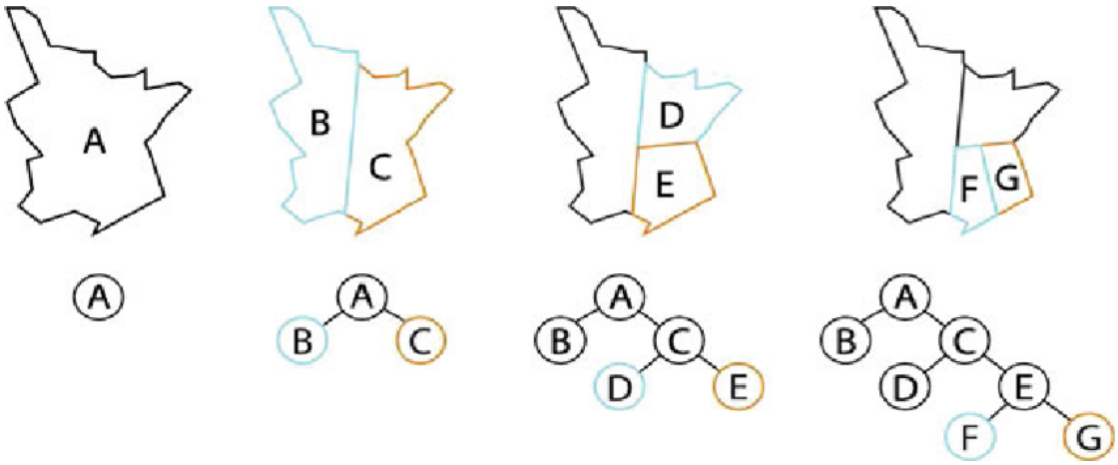

Figura 2.12: Utilização de BSP em regiões geográficas (ilustração retirada de [17]).

A detecção de colisão entre um raio e uma coleção de objetos (*ray cast*) também pode ser determinada navegando na estrutura de uma *BSP-tree*. Observe o exemplo mostrado na Figura 2.13 de um raio cruzando as regiões e colidindo com a esfera *A*. O raio percorre a *BSP-tree* a partir do nó raiz, assim é verificado se o raio intersecta a primeira região *1*. Sendo positivo, são verificadas as regiões *2* e *3*. Como o raio se encontra ao lado esquerdo da região *3*, apenas os objetos contidos ao lado esquerdo da região *3* são verificados, que neste caso é o objeto *D*. O mesmo procedimento ocorre na região *2*. O raio procede pela esquerda da região *2* e verifica que está à esquerda da região *4*. Nessa região, apenas é identificada a existência de uma esfera e é verificada se de fato ocorre a colisão no objeto. Observe que, de seis objetos, a verificação de colisão do raio é comparada apenas com dois objetos, o que aumenta a performance consideravelmente, visto que a verificação se o raio está à esquerda ou à direita do raio é relativamente rápida usando um operador de orientação, como detalhado no apêndice A.

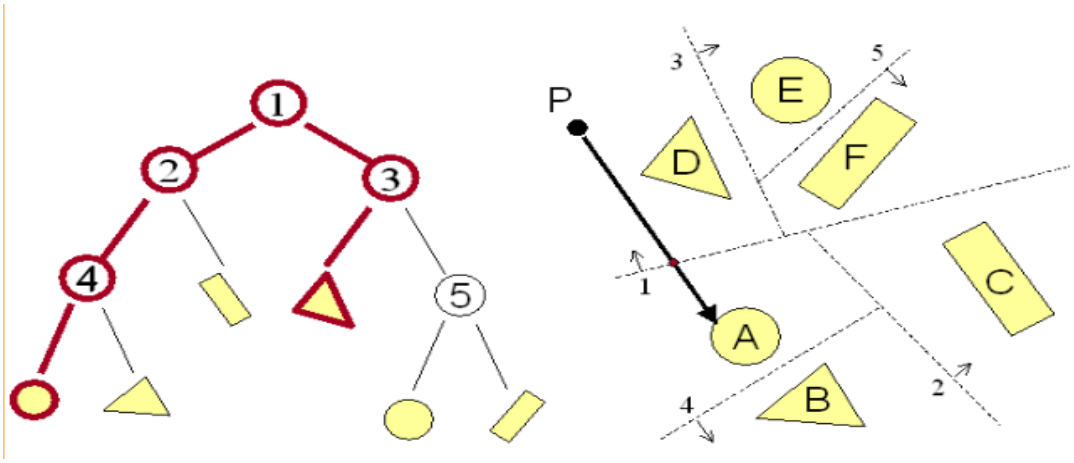

Figura 2.13: Utilização de *BSP* para detecção de colisão de um raio (ilustração retirada de [45]).

2.3 Hierarquia de Volumes Envolventes

Outra técnica de aceleração hierárquica empregada em detecção de colisão são as Hierarquias de Volumes Envolventes (*Bounding Volumes Hierarchies* ou *BVH*) [8], que utilizam um raciocínio semelhante ao das subdivisões espaciais hierárquicas. Uma *BVH* é uma árvore n-ária de volumes envolventes, sendo o nó raiz o volume que envolve todos os demais. Cada nó filho do nó raiz contém seus volumes envolventes e assim recursivamente, até um nível de detalhe suficiente para que o nível de granularidade ou precisão da detecção de colisão seja suficiente para a etapa posterior (*Narrow Phase*).

Algumas características são desejadas quando criando uma hierarquia de volumes envolventes: os nós contidos na sub-árvore sejam próximos um do outro; cada nó deve conter um volume mínimo do objeto real; a soma de todos os volumes envolventes deve ser

mínimo; a sobreposição entre os volumes vizinhos devem ser mínimas; a hierarquia deve ser balanceada; atenção deve ser dada aos nós próximos à raiz para um rápido descarte no caso em que não há colisão.

Visto que as características mencionadas anteriormente são desejadas, diferentes formas de construir o agrupamento dos objetos podem ser realizadas com objetivo de atendê-las, visto que, computacionalmente, é inviável a pesquisa para a melhor árvore [8]. Para aplicações interativas que exigem grande quantidade de quadros por segundo, as hierarquias são construídas na inicialização e não durante a execução da aplicação. Na etapa de construção, deve-se considerar o grau da árvore, podendo esta ser binária, ternária ou n-área. Árvores de grau mais alto possuem uma altura menor, consequentemente minimizando o tempo de percurso até a profundidade, entretanto tem maior custo de visitar cada nó. Existem três categorias primárias de método de construção de *BVH*: *top-down*, *bottom-up* e *insertion*.

O método *top-down* inicia particionando o conjunto de entrada em dois ou mais subconjuntos, envolvendo-os no volume envolvente selecionado. Recursivamente, envolve os demais subconjuntos. Pelo fato de ser o mais simples de implementar, este método é o mais popular [8].

O método *bottom-up* inicia com o conjunto de entrada pelas folhas da árvore, então agrupando-os em dois ou mais volumes envolvente, procedendo da mesma maneira até todos os objetos estarem agrupados em um único nó (raiz).

O método *insertion* constrói uma hierarquia incrementalmente, inserindo cada objeto por vez na árvore. Este método é considerado para aplicações em tempo real. Uma das vantagens é a dinâmica que o método oferece permitindo atualizações serem realizadas em tempo de execução. A Figura 2.14 ilustra os três métodos mais comuns de construção de *BVH*s.

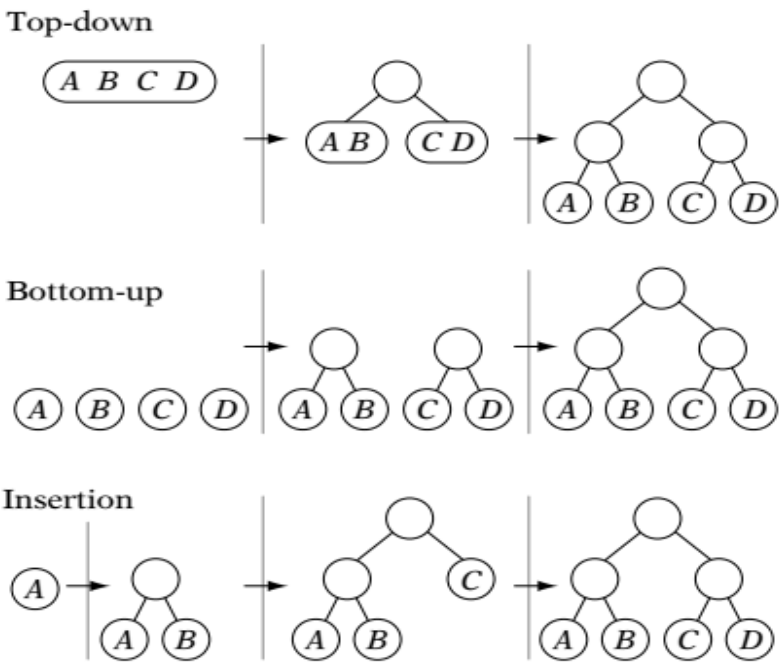

Figura 2.14: Métodos de construção de *BVH* (ilustração retirada de [8]).

A *BVH* recebe destaque pelo fato de poderem agrupar objetos complexos, contendo geometria não-convexa, formando um único volume envolvente para ser tratado na *Broad Phase*. A Figuras 2.15 e 2.16 apresentam um automóvel militar sendo envolvido hierarquicamente com volumes esfera (lado esquerdo) e com *AABB* (lado direito), e um robô sendo envolvido hierarquicamente com esferas contendo a representação hierarquica no lado direito, respectivamente.

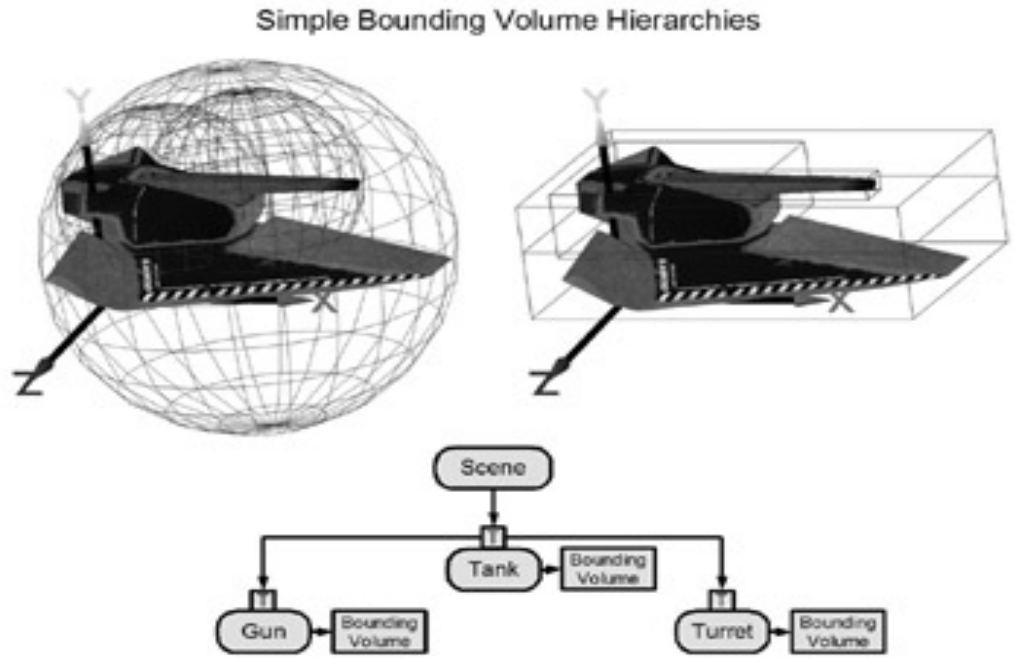

Figura 2.15: Exemplo de *BVH* envolvendo um automóvel militar (ilustração retirada de [54]).

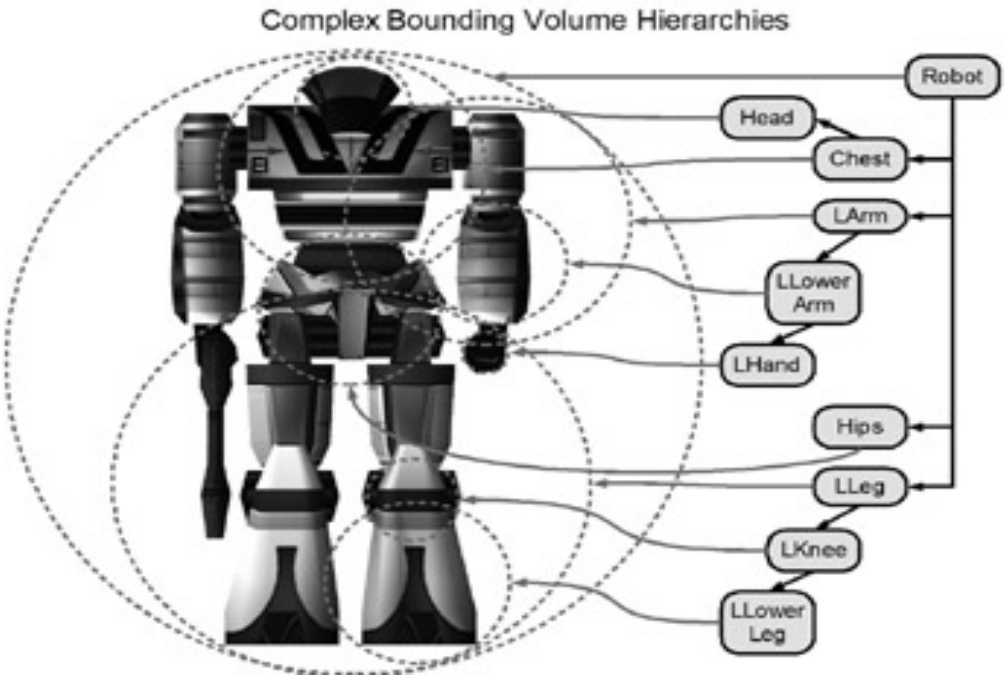

Figura 2.16: Exemplo de *BVH* envolvendo um robô (ilustração retirada de [54]).

Sendo construída a *BVH*, é possível verificar a detecção de colisão entre duas hierarquias ou até mesmo verificação de uma hierarquia com sua própria estrutura, alterando poucos passos do algoritmo original. O algoritmo tem como entrada as duas *BVH*s e se inicia verificando se a raiz de cada uma se colide. Não sendo identificada a colisão, o

algoritmo é finalizando indicando que não há colisão. O algoritmo percorre ambas as árvores, por nível, verificando a colisão dos volumes envolventes até o nível das folhas. A Figura 2.17 apresenta a sequência de passos para verificar a colisão entre duas *BVH*s distintas.

```
void BVHCollision(CollisionResult *r, BVTree a, BVTree b)
{

    if (!BVOverlap(a, b)) return;
    if (IsLeaf(a)) {
        if (IsLeaf(b)) {
            // At leaf nodes. Perform collision tests on leaf node contents
            CollidePrimitives(r, a, b);
            // Could have an exit rule here (eg. exit on first hit)
        } else {
            BVHCollision(a, b->left);
            BVHCollision(a, b->right);
        }
    } else {
        if (IsLeaf(b)) {
            BVHCollision(a->left, b);
            BVHCollision(a->right, b);
        } else {
            BVHCollision(a->left, b->left);
            BVHCollision(a->left, b->right);
            BVHCollision(a->right, b->left);
            BVHCollision(a->right, b->right);
        }
    }
}
```

Figura 2.17: Algoritmo de detecção de colisão entre duas *BVH*s (ilustração retirada de [54]).

2.4 Algoritmos para a Fase Ampla

Nesta seção são apresentados os algoritmos mais comumente empregados na Fase Ampla da detecção de colisão, que é o foco do presente estudo.

Força Bruta

O algoritmo de força bruta consiste em examinar todos os pares de objetos, tem implementação simples mas possui complexidade assintótica quadrática $O(n^2)$, sendo assim, recomendado apenas quando a quantidade de objetos em cena é relativamente pequena.

Varredura e Poda

O algoritmo de Varredura e Poda (*SaP* - Sweep and Prune) [2] [1], também conhecido como ordenar e varrer (*Sort and Sweep*), geralmente faz uso de volumes envolventes *AABB* e consiste em uma etapa de ordenação dos pontos mínimos ou máximos das *AABB*s em todos os três eixos base (em ambientes 3D), portanto nosso exemplo será baseado em *AABB* sem perda de generalidade para outros volumes envolventes. Após a ordenação, é realizada uma varredura em cada eixo dos pontos ordenados e é verificado se os intervalos dos pontos mínimo e máximo de cada *AABB* se intercalam. Caso eles se intercalem nos três eixos, indica que o par de *AABB*s está em colisão. Este algoritmo também já foi utilizado com volumes do tipo Esfera [2], fazendo uso de pontos tangente à esfera, sendo assim, faz-se necessário a varredura em um ou mais eixos, visto que seria necessário percorrer todos os eixos perpendiculares tangente à esfera para verificar se ela colide com outra utilizando o Teorema do Eixo Separador. Em [2], um único eixo é varrido e, neste caso, o algoritmo *SaP* pode concluir que os volumes não se intersectam, mas não a ausência de um plano separador. Para concluir absolutamente a não interseção, faz-se necessária uma verificação mais detalhada, geralmente postergada para a próxima etapa (*Mid-Phase* ou *Narrow Phase*). A generalização desse método também pode ser aplicada para volumes com k orientações fixas, no caso dos k-*DOP*s, entretanto, não foram encontrados trabalhos que descrevessem pesquisa neste sentido. Uma forma de otimização do algoritmo empregado em [2] e [1] é a seleção do eixo de varredura. Sendo assim, dependendo do eixo selecionado, pode-se se identificar rapidamente os pares de objeto que não estão em colisão. Em [2] foi utilizada a Análise da Componente Principal (*Principal Component Analysis* ou *PCA*), que é um método estatístico para estimar o eixo que melhor separa os objetos em cena. A Figura 2.18 abaixo ilustra o método *PCA* com a seleção do eixo contendo a maior dispersão das esferas dada a disposição dos objetos em cena. Observe que se o eixo selecionado for o eixo X ou Z, todas as esferas seriam marcadas como em colisão e repassadas para a próxima fase. Com a técnica de *PCA*, visualmente é possível identificar que duas esferas estão totalmente fora de intervalo para o eixo selecionado, sendo elas a primeira esfera próxima à origem e a quinta esfera contando do final do eixo em azul até a origem..

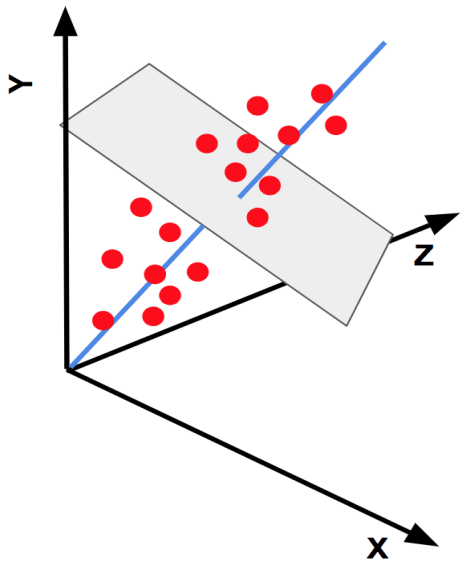

Figura 2.18: Em azul, o eixo perpendicular ao plano exibindo a primeira componente de maior variância em relação aos dados (ilustração retirada de [49]).

Em virtude do paralelismo, como forma de aceleração, e da implementação em *GPU*, faz-se necessário que a ordenação também seja paralelizada. Neste contexto, destacam-se dois algoritmos de ordenação paralela: ordenação por dígito (*Radix Sorting*) [3] e ordenação de rede (*Bitonic/Network Sorting*) [9]. Ambos são ordenadores paralelos com grande eficiência em *GPU*. Para uma ordenação crescente usando *Radix Sorting*, seleciona-se cada dígito do final do número para o início, e é realizado um histograma de todos os últimos dígitos da coleção de números. Estes dígitos, geralmente, são armazenados em dez 'recipientes' (0 - 9), no caso para os números de base decimal, podendo também utilizar bases octais ou hexadecimais. Posteriormente, é realizado o processo de varredura por prefixos (*prefix scan*) [50], consistindo em criar um vetor contendo as novas posições de cada elemento armazenado no histograma. Por último, é varrido cada elemento da coleção de números, de forma ordenada pelo recipiente inserido na posição correta em ordem, considerando o vetor de posições criado na etapa do *prefix scan*. Esse processo é feito para todos os dígitos contidos no número dentro da coleção de números (entrada). A Figura 2.19 exemplifica o processo.

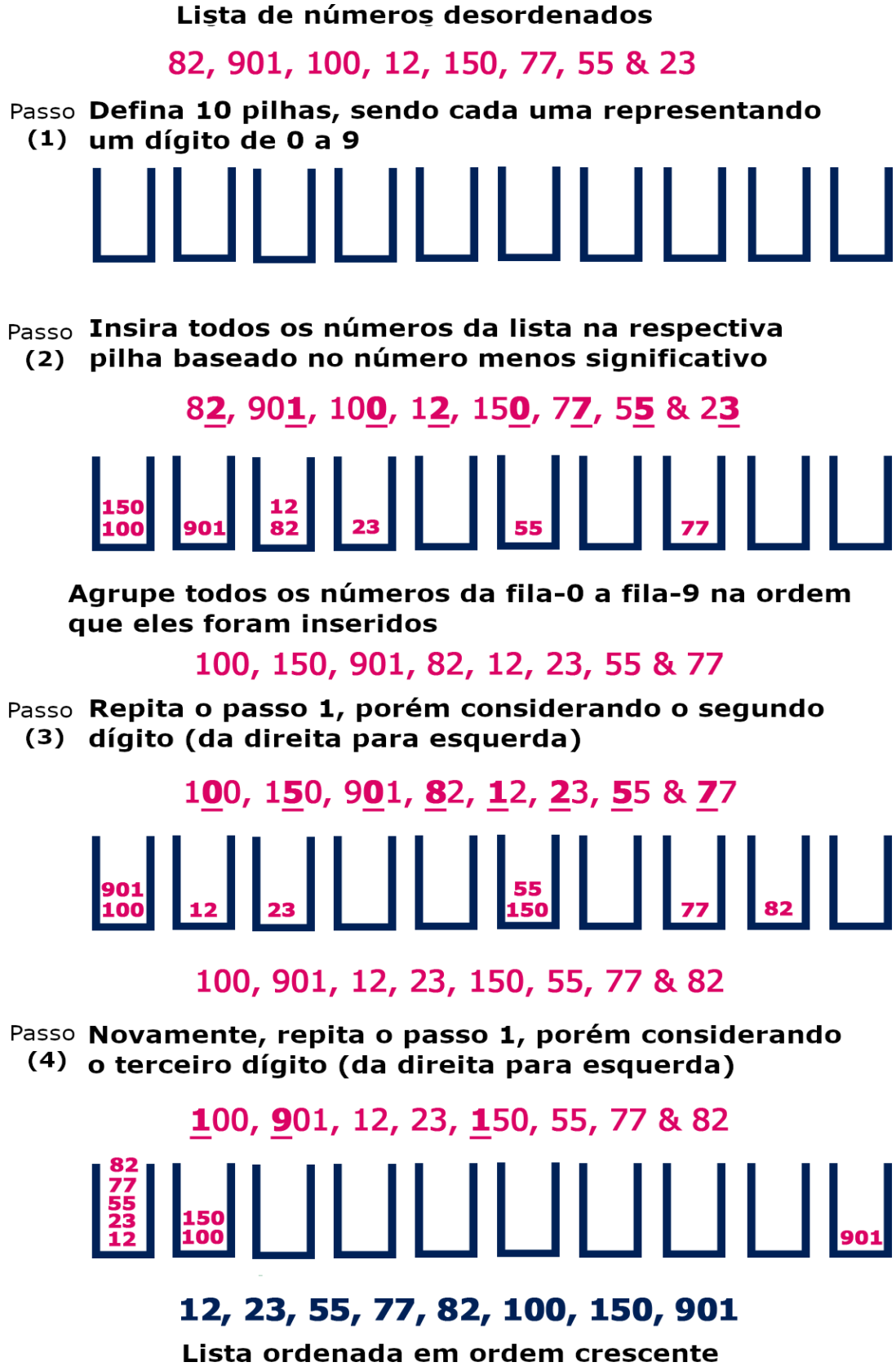

Figura 2.19: Processo de ordenação usando *Radix Sorting* (ilustração retirada de [3]).

Já a ordenação por rede (*Bitonic/Network Sorting*) é classificada como uma ordenação por comparação, consistindo em comparar os dois elementos consecutivos e ordená-los,

intercalando de baixo para cima e de cima para baixo. Posteriormente, são comparados os quatro elementos consecutivos da mesma forma, isto é, cria-se sequências de dois elementos, sendo uma sequência ascendente e outra descendente, que são comparadas e ordenadas. No próximo passo a sequência passará a ter o dobro do número anterior de elementos. Assim sucessivamente, até atingir metade dos elementos. Após esta etapa, criar sequências comparando com o elemento na posição somado a metade da quantidade total de elementos, compare-os e ordene. Execute o mesmo processo, porém, com a metade da quantidade no processo anterior. Realize esses passos até a quantidade ser igual a um. Após essas sequências, os elementos estarão ordenados. A Figura 2.20 detalha o procedimento.

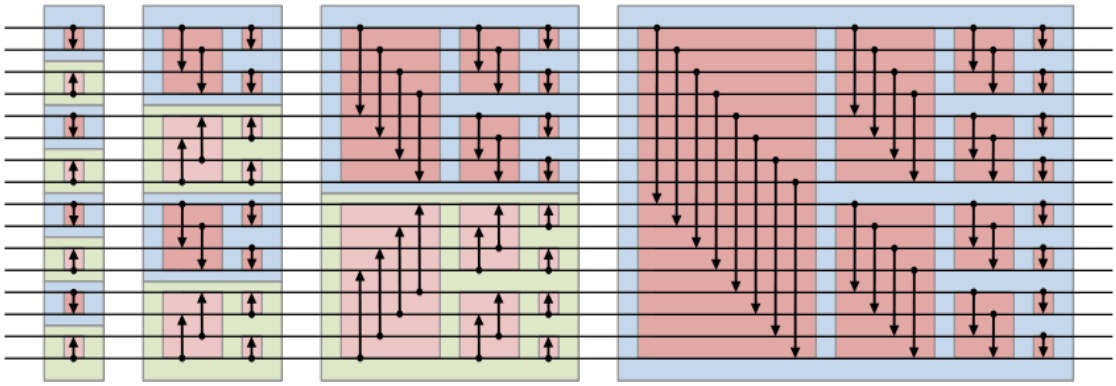

Figura 2.20: Processo de ordenação em rede *Bitonic/Network Sorting* (ilustração retirada de [44]).

Grade Uniforme

O algoritmo de detecção de colisão baseado em grade uniforme (*Uniform Grid*) comumente apresenta um desempenho superior ao de força bruta. Ele consiste em uma subdivisão virtual do espaço em células denominadas *voxels*, cada uma das quais contém um identificador único (*hash*). Ao incluir um objeto na cena, este é discretizado de forma a identificar quais células (*voxels*) o objeto ocupa, sendo armazenado em uma estrutura de dados que, dado um identificador único, informa quais elementos se encontram naquela posição. Geralmente é utilizada uma tabela de espalhamento (*hash table*). Quando um novo objeto é incluído na cena, ou ao renderizar um novo quadro na animação, o mesmo processo ocorre. Caso já exista alguma entrada na *hash table* para uma das células do novo objeto, isto indica que eles ocupam o mesmo espaço e, consequentemente, existe uma colisão entre eles. Uma lista ou vetor armazena os pares objetos em colisão. Este método apresenta uma complexidade assintótica $O(kn)$, onde n é o número de objetos e k é o número médio de células ocupadas por cada objeto. Uma das desvantagens deste algoritmo ocorre quando há uma desproporção grande no tamanho dos objetos, dado o fato que uma célula normalmente tem que ser pequena o suficiente para se ajustar ao

menor objeto, e portanto, objetos muito grandes na mesma cena elevariam o valor de k, aumentando o custo de processamento.

2.5 Algoritmos para a *Narrow Phase*

Após a execução da *Broad Phase*, no qual geralmente elimina grande parte dos pares de objetos realizando consideráveis melhoramentos de desempenho, é necessário executar uma análise detalhada dos pares restantes, aqueles que foram identificados como colisão pela *Broad Phase*. Essa análise é executada na *Narrow Phase*.

2.5.1 *GJK*

O algoritmo [56] [57] [59] *Gilbert–Johnson–Keerthi* (*GJK*) verifica se dois objetos poliedrais convexos estão em interseção. Este algoritmo faz uso das diferenças de *Minkowski* sobre o par de objetos envolvidos.

As diferenças de *Minkowski*, também chamada de dilatação morfológica, é definida pelo conjunto convexo composto pelas diferenças de um conjunto convexo **A** e outro conjunto convexo **B**. Dado o conjunto **M** composto pelas diferenças de *Minkowski*, temos $\mathbf{M} = \mathbf{A} - \mathbf{B} = \{\mathbf{a} - \mathbf{b} | \mathbf{a} \in \mathbf{A}, \mathbf{b} \in \mathbf{B}\}$. Observe na Figura 2.21 que o novo conjunto é composto apenas pelos vértices que compõem o fecho convex, eliminando - por exemplo - o vértice $(-1, 1)$ definido na operação $(2, 2) - (3, 1)$.

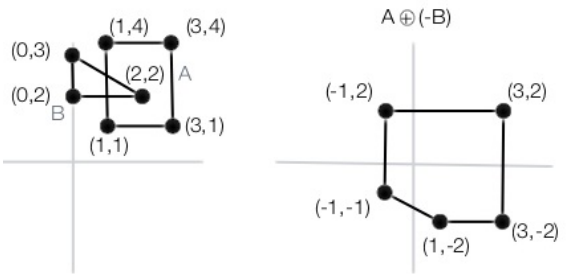

Figura 2.21: Diferença de Minkowski de $\mathbf{A} - \mathbf{B}$ (ilustração retirada de [61]).

Através desse novo fecho convexo definido pelas diferenças de *Minkowski*, no qual chamaremos de **M**, o *GJK* é executado iterativamente buscando encontrar um *simplex* Figura 2.22 que contenha a origem. *Simplex* é uma generalização do conceito de triângulo contendo o polígono mais simples para a dimensão em que se encontra, sendo no caso de 2D um triângulo e em 3D um tetraedro.

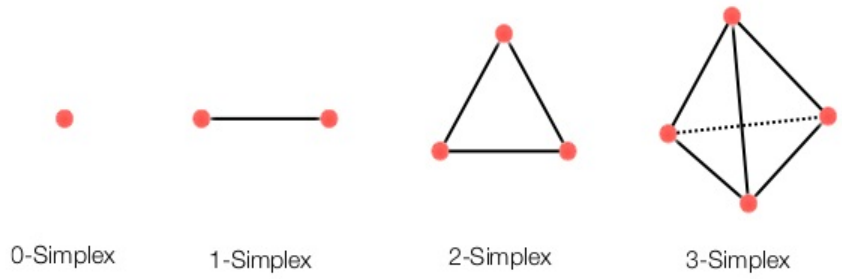

Figura 2.22: *Simplex* (ilustração retirada de [61]).

Caso o algoritmo consiga encontrar um *simplex* que contenha a origem, este é retornado, caso contrário o algoritmo sinaliza que o par de objetos não está em colisão.

Para encontrar o *simplex*, o algoritmo inicia selecionando um qualquer ponto do conjunto **M** definido anteriormente. Com este ponto arbitrário, é calculada uma direção para a origem, pesquisado o vértice de **M** que seja o mais distante nessa direção. Este conceito é chamado de função de suporte, conforme Figura 2.23, ou seja, dada uma direção, encontre um vértice extremo no fecho convexo.

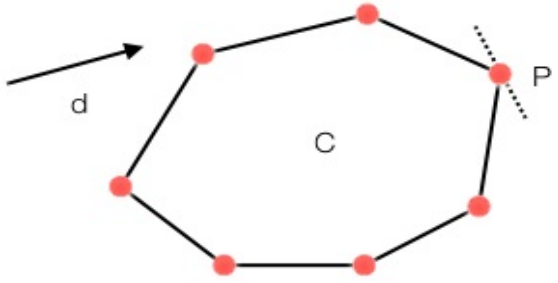

Figura 2.23: Função de Suporte (ilustração retirada de [61]).

Com dois pontos obtidos (uma semi-reta), o mesmo processo é realizado, utilizado-se a função de suporte, porém a direção é a normal da da semi-reta com a origem. Com a obtenção do terceiro ponto, uma face é criada e nesta utiliza-se a função de suporte para fornecer a direção normal da face (ou sua oposta), obtendo assim o quarto ponto, compondo um tetraedro. Caso o tetraedro não contenha a origem, o primeiro ponto selecionado aleatoriamente é removido e é feita uma busca com a função de suporte utilizando como direção a normal da face remanescente.

Para exemplificar, observe a Figura 2.24 em ambiente bidimensional. O algoritmo inicia com um ponto aleatório A e busca um ponto extremo em direção à origem utilizando a função de suporte, encontrando o vértice B, conforme descrito no passo 1. Com os vértices A e B, é pesquisado outro ponto de suporte em direção à origem, no qual é identificado o ponto D, conforme passo 2. Neste momento, o *simplex* (A, B e D) é encontrado e verificado que a origem ainda se encontra fora do *simplex*. Portanto, no passo 3, o último ponto (A) é removido e pesquisado um novo ponto para compôr o *simplex*

utilizando os pontos (B, D e F) apenas. No passo 4, o ponto F é adicionado e verificado que não há mais vértices em direção à origem. Como está fora do *simplex* verifica-se a não colisão.

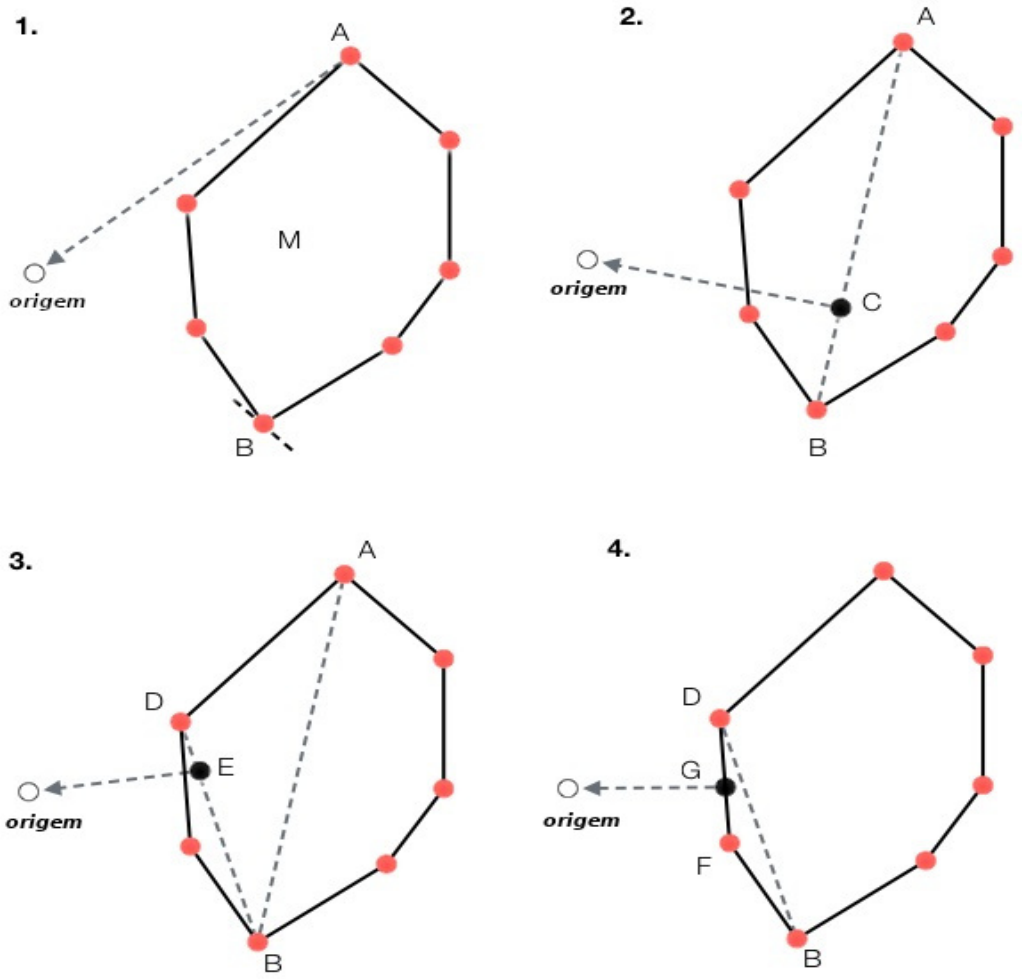

Figura 2.24: Execução do algoritmo *GJK* em 2D (ilustração retirada de [61] e traduzida).

O algoritmo *GJK* não requer conhecimento sobre a geometria, exceto seus vértices e a garantia que seja convexa. Entretanto, caso seja conhecida a estrutura da malha, é possível melhorar a performance do *GJK* [57] computando de forma mais eficiente a função de suporte. Para encontrar um extremo dada uma direção na função de suporte, ao invés de verificar todos os vértices da malha, é possível utilizar suas arestas para buscar sempre um vértice que irá obter um resultado melhor que o anterior. Este procedimento é conhecido como *GJK* Melhorado (*Enhanced GJK*).

2.5.2 Algoritmo de Expansão de Politopo

Em conjunto com o algoritmo *GJK*, é geralmente aplicado o Algoritmo de Expansão de Politopo (*Expanding Polytope Algorithm* - *EPA*) [58] [59], que requer como entrada um *simplex* e retorna detalhes da colisão, como profundidade de penetração, uma face de **M**

em 3D ou aresta de **M** para o caso 2D, além da direção normal da face. Para atingir esse objetivo, o *EPA* pesquisa a aresta do *simplex* mais próxima da origem, de forma semelhante ao *GJK*.

O *EPA* inicia com as quatro faces do tetraedro fornecidas pelo *GJK*, com suas respectivas normais, e - utilizando o conjunto **M** contendo as diferenças de *Minkowski*, busca iterativamente a face mais próxima da origem. Com a normal da face, esta é utilizada para buscar um novo ponto utilizando a função de suporte. Se a face de **M** encontrada for a mais próxima da origem, o algoritmo termina identificando os detalhes da colisão, senão uma nova iteração do *EPA* é empregada considerando o novo ponto encontrado.

Na Figura 2.25, observa-se do lado esquerdo dois polígonos em interseção, em destaque os vértices envolvidos na colisão em coloração preta. No lado direito, é apresentado o conjunto **M** definido pelas diferenças de *Minkowski* dos dois polígonos do lado esquerdo. Após a execução do *EPA*, este sinaliza que a aresta do fecho de **M** mais próxima à origem (destacada em azul) é a aresta em amarelo.

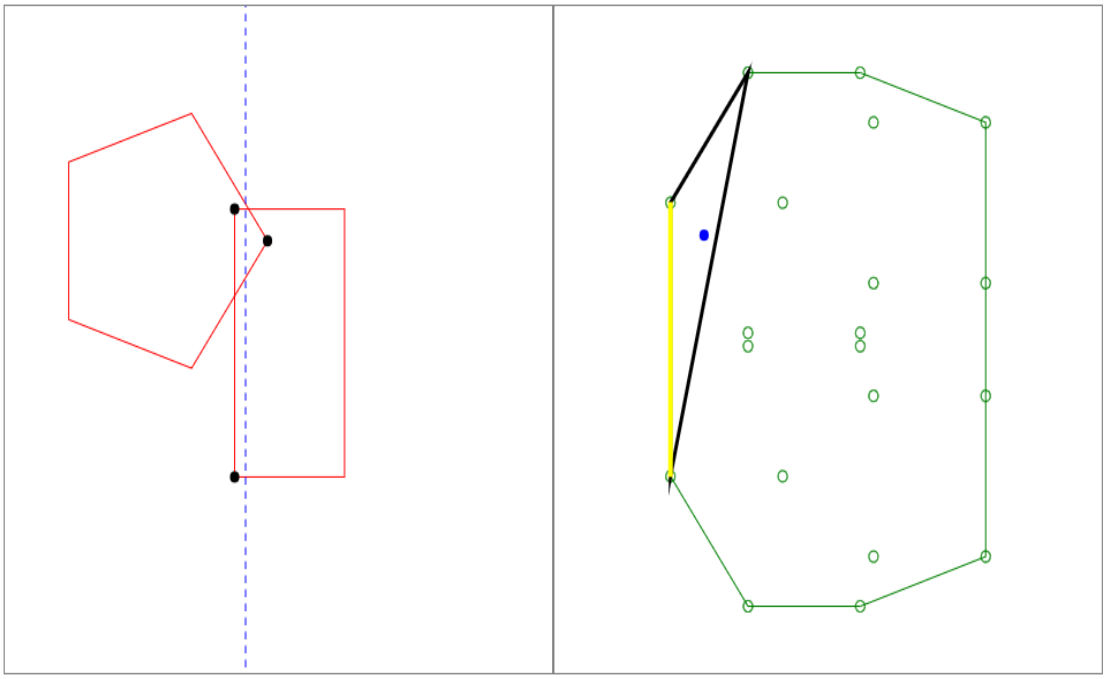

Figura 2.25: Execução do algoritmo *EPA* em 2D (ilustração retirada de [59]).

Neste próximo exemplo do emprego do *EPA* demonstrado na Figura 2.26 os polígonos do lado esquerdo se colidem em outra orientação. Observe que neste caso o *EPA* identifica a nova aresta em **M** onde ocorreu a colisão.

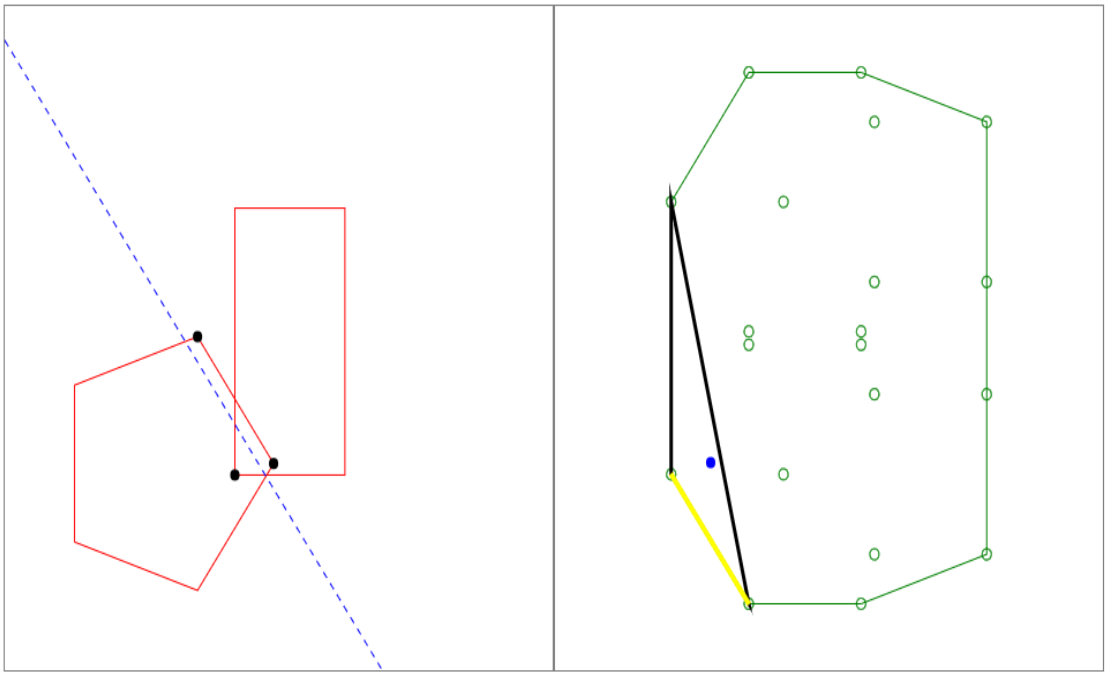

Figura 2.26: Execução do algoritmo *EPA* em 2D em outra orientação (ilustração retirada de [59]).

2.6 Algoritmos para dinâmica de corpos rígidos

2.6.1 Baseada em Impulso

Em consequência da detecção de colisão, faz-se necessário tratar a colisão de forma fisicamente plausível. A dinâmica de objetos é o campo de estudo que envolve o comportamento de objetos considerando os fatores físicos, como massa, centro de massa, tensor de inércia, coeficiente de restituição, fricção, forças atuantes, aceleração, velocidade, posição, momento linear e angular. Entre outros fatores que podem ser considerados, de forma que seja possível criar comportamentos e animações de acordo com a perspectiva do animador.

Cinemática

A cinemática estuda o movimento de corpos sem considerar seus fatores causadores (forças) [35]. Neste contexto, para a simulação de corpos rígidos em computação, geralmente estes são modelados considerando as seguintes componentes abaixo:

- Posição - define a posição do corpo, na forma vetorial, geralmente utilizando as coordenadas globais.
- Velocidade Linear - definida de forma vetorial, uma componente para cada eixo de coordenada. A velocidade também pode ser aproximada pelo deslocamento do

corpo dado um período de tempo $\frac{S_2-S_1}{T_2-T_1}$, onde S_2 é a posição atual do objeto, S_1 a posição anterior, T_2 o tempo atual e T_1 o tempo anterior. Em verdade, a derivada da posição em função do tempo pode ser computada como

$$\frac{ds}{dt} = \lim_{\Delta t \to 0} \frac{\Delta s}{\Delta t}.$$

- Velocidade Angular - definida de forma vetorial, uma componente para cada eixo de rotação, para identificar a velocidade de rotação do objeto nos eixos X, Y e Z (em cenários com três dimensões). A velocidade angular também pode ser aproximada pelo deslocamento angular do corpo dado um período de tempo $(\phi_2 - \phi_1)/(T_2 - T_1)$, onde ϕ_2 é o ângulo de rotação atual do objeto, ϕ_1 o ângulo anterior, T_2 o tempo atual e T_1 o tempo anterior, ou seja, a derivada da posição em função do tempo $\frac{\Delta \phi}{\Delta t} dt$

- Aceleração - também definida de forma vetorial pela mesma razão da velocidade. Esta armazena a aceleração no atual intervalo de tempo entre um passo da simulação e outro, ou seja, $(V_2 - V_1)/(T_2 - T_1)$, também representada por $\frac{\Delta v}{\Delta t} dt$, onde V é a velocidade do corpo.

- Massa - é um escalar que define a massa de um corpo, sendo utilizada para indicar a resistência de deslocamento linear de um objeto.

- Tensor de Inércia - normalmente representado computacionalmente por uma matriz, o qual define um tensor para cada eixo de rotação, ou seja, representa a resistência à rotação de um objeto com seus eixos. Esta resistência pode ser calculada através da distribuição de massa do corpo através de suas partículas ou definida por uma constante no caso de corpos rígidos. Diversos algoritmos foram desenvolvidos para cálculo do tensor de inércia, sendo os principais *voxelização* [48] e cálculo direto da distribuição de massa [48].

Computacionalmente, um corpo rígido geralmente é composto pelo centro de massa, que contém sua posição e massa, que pode ser calculada como a média aritmética de todas massas das partículas do corpo:

(1) Seja *M* a massa total do corpo dada por: $M = \sum_{i=1}^{max(P)} m_i$, onde P é o conjunto de partículas e *max* é uma função que fornece a quantidade de partículas no sistema.

(2) A posição do centro de massa definida em $C_p = \sum_{n=1}^{max(p)} \frac{P_p(i)}{max(P)}$

(3) A massa do centro de massa definida por $C_m = \sum_{n=1}^{max(p)} \frac{P_m(i)}{M}$

Além disto, um corpo rígido também possui uma orientação, dado o fato que pode sofrer rotações, portanto faz-se necessário mapear a orientação *R* e a velocidade angular ω do objeto, bem como o tensor de inércia. Em ambientes tridimensionais, os objetos podem rotacionar três eixos, X, Y e Z, sendo necessário um vetor para cada eixo. Na forma vetorial os três eixos são mapeados em uma matriz 3x3, onde cada linha ou coluna, dependendo do sistema matricial empregado (por linha ou coluna), define a rotação em um eixo. Visto que, para realizar uma rotação, é necessário especificar um eixo de rotação, para os três eixos isto é convergido para um ponto - o ponto de rotação. Para este ponto é comumente utilizada a localização do centro de massa do objeto rígido. Sendo utilizado o centro de massa, a rotação se dá em função das coordenadas locais do objeto, logo sendo requerido uma translação antes de rotação e posteriormente outra translação para retornar o objeto em seu local de origem rotacionado no centro de massa do corpo. Este cálculo se apresenta da seguinte maneira:

Sendo *R* a matriz de orientação do corpo, *S* o vetor posição do centro de massa do corpo, P as partículas do corpo e *i* o índice da partícula: $P_i = ((P_i - S)R) + S$, ou seja, para cada partícula do corpo, esta será transladada em função do centro de massa, rotacionada e devolvida à posição original com a rotação.

Para auxiliar os cálculos, normalmente a massa é armazenada como inversa $\frac{1}{m}$, visto que constantemente exige o cálculo pela divisão da massa. Como a divisão é custosa computacionalmente, armazenando a massa inversa substitui-se o operador de divisão pela multiplicação, tornando mais eficiente. Sendo assim, da mesma maneira é o tensor de inércia. Além disto, utiliza-se o sistema internacional de unidades, o qual define as seguintes medidas:

- Tempo - Segundos (s)
- Deslocamento - Metro (m)
- Massa - Quilograma (Kg)
- Velocidade (linear) - Metro/Segundo $\frac{m}{s}$
- Velocidade (angular) - Radianos/Segundo $\frac{r}{s}$
- Aceleração (linear) - Metro/Segundo ao quadrado $\frac{m}{s^2}$
- Aceleração (angular) - Radianos/Segundo ao quadrado $\frac{r}{s^2}$
- Força - N (*Newton* ou Quilograma vezes metro/massa ao quadrado $\frac{Kg.m}{m^2}$
- Densidade - Massa/Distância^3 $\frac{M}{m^3}$

A dinâmica de corpos rígidos baseada em impulso tem forte relação com as leis de *Newton*, portanto faz-se necessário destacar os conceitos que norteiam as leis de *Newton*.

1. Lei da Inércia - todo corpo continua em seu estado de repouso ou de movimento uniforme em uma linha reta, a menos que seja forçado a mudar aquele estado por forças aplicadas sobre ele. Em outras palavras, todo corpo em repouso, se mantém em repouso, até que alguma força seja aplicada a ele. O inverso também se aplica, todo corpo em movimento, se mantém em movimento, até que alguma força igual contrária se aplique a ele. Esta lei é geralmente representada computacionalmente através da massa para o momento linear e do tensor de inércia para o momento angular, ambos representando a resistência respectivamente.

2. Lei da Superposição de Forças, também conhecida como Princípio Fundamental da Dinâmica - a mudança de movimento é proporcional à força motora imprimida e é produzida na direção de linha reta na qual aquela força é aplicada, ou seja, esta lei define que o módulo da aceleração produzida sobre um corpo é proporcional ao módulo da força aplicada sobre ele, e inversamente proporcional à massa do corpo. Matematicamente a aceleração pode ser expressa em função da força: $|\vec{a}| = \frac{|\vec{F}|}{m}$. Rearranjando a equação colocando a aceleração em função da força pode ser expressada em $\vec{F} = \vec{a}.m$

3. Lei da ação e reação - para toda ação há sempre uma reação oposta e de igual intensidade: as ações mútuas de dois corpos um sobre o outro são sempre iguais e dirigidas em sentidos opostos. Entende-se por esta lei que para um corpo iniciar um movimento é necessário que um outro corpo interaja provocando uma ação e reação entre os corpos em contato. Computacionalmente esta lei é presentada pela conservação do momento, detalhada mais adiante.

Quando ocorre um contato entre dois corpos, duas grandezas físicas devem ser consideradas, sendo uma delas a conservação do *momentum*. Na dinâmica de objetos rígidos o termo *momentum* é definido como uma quantidade de movimento, sendo imprescindível na transferência de movimentos em um sistema envolvendo dois ou mais corpos, seja na translação (momento linear), seja na rotação (momento angular). Matematicamente o momento linear $\vec{p}$ é dado pela massa vezes a velocidade $\vec{p} = m\vec{v}$, enquanto o momento angular $\vec{l}$, é definido pelo produto vetorial do tensor de inércia $\vec{I}$ vezes a velocidade angular ω, ou seja, $\vec{l} = \vec{I} \times \vec{\omega}_i$. Para que seja conservado o momento linear durante o contato, faz-se necessário relacionar o momento dos dois objetos em contato. Esta relação se dá através da igualdade de $m_1 v_{1\text{-}} + m_2 v_{2\text{-}} = m_1 v_{1+} + m_2 v_{2+}$, onde *m* é a massa, *v* é a velocidade e os índices *1* e *2* referentes aos objetos envolvidos, sendo a sinalização referente à velocidade antes da colisão e após a colisão.

Além da conservação do momento, também deve ser considerado o tipo de colisão entre os objetos, podendo esta ser elástica ou perfeitamente inelástica. No caso da colisão inelástica, os objetos mantêm a mesma velocidade que estavam antes da colisão,

geralmente alterando apenas o sentido do deslocamento. Para colisões elásticas, é necessário calcular o coeficiente de restituição para encontrar a velocidade dos objetos $e = -(v_{1+} - v_{2+})/(v_{1-} - v_{2-})$, onde *e* é o coeficiente de restituição a ser utilizado como fator na velocidade após a colisão, *v* a velocidade e os índices *1* e *2* referentes aos objetos envolvidos e os sinais indicando se a velocidade é antes ou após a colisão.

Métodos de Integração

Método de *Euler*

O método de integração de *Euler* tem como base a série de *Taylor* que permite aproximar o valor de uma função em algum ponto, porém exige algum conhecimento sobre a função e sua derivada. Essa expressão pode ser aproximada por uma série polinomial infinita, dada por:

$f(x + \Delta x) = \sum_{n=0}^{\infty} \frac{f^{(\mathrm{n})}(x)}{n!} \Delta x^{\mathrm{n}}$

expandindo esta série, temos:

$f(x + \Delta x) = f(x) + f'(x)(\Delta x) + \frac{f''(x)}{2!}\Delta x^2 + \frac{f'''(x)}{3!}\Delta x^3 + ...$

, onde *y* está em função de *x* mais um deslocamento Δx. No contexto de simulação, a função pretendida na integração é a velocidade em função do tempo, ou seja, $v(t + \Delta t) = v(t) + v'(t)\Delta t$. Esta é a integração pelo método de *Euler*, considerando apenas o termo de ordem mais alto (até a primeira derivada) e desprezando as demais derivadas da série, compreendendo assim um erro de truncamento, entretanto, quanto menores forem os demais termos, menos influentes estes serão na aproximação, portanto, é razoavelmente aceitável. Além disto, Δt já se inicia em um valor pequeno em simulações interativas, exponenciando esse valor nos demais termos da série de *Taylor*, menor ainda este será. Geometricamente o método de *Euler* aproxima o novo valor extrapolando a direção da derivada da função em relação ao passo anterior, conforme Figura 2.27.

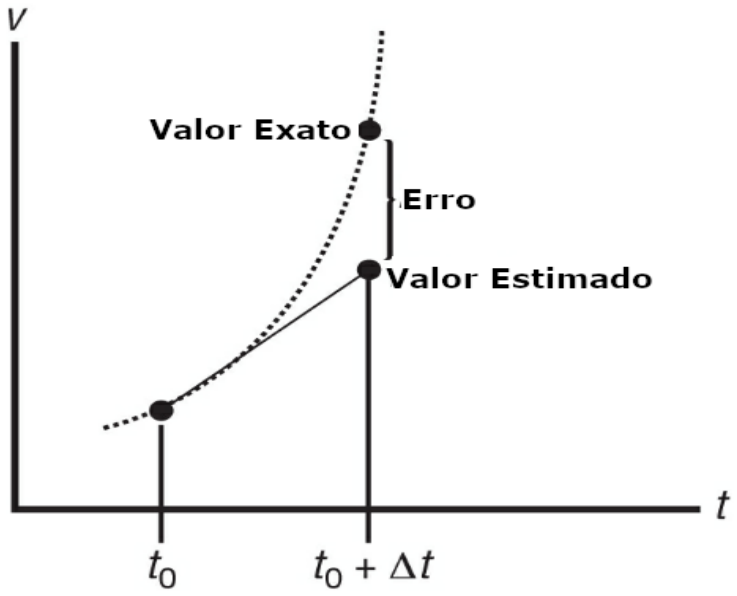

Figura 2.27: Aproximação utilizando método de *Euler* (ilustração retirada de [35]).

Método Runge-Kutta

Este método de integração se assemelha ao método de *Euler*, porém com maior precisão, visto que este utiliza não somente a primeira ordem mais alta do polinômio de *Taylor*, mas também a segunda ordem, ou até maiores. No método *Runge-Kutta* é primeiro calculado o método de *Euler*, visto o erro de truncamento que ocorre nesse processo, é selecionada a metade do método do *Euler* e a partir desse ponto é calculada a segunda ordem do polinômio de *Taylor*, buscando melhor aproximação. A Figura 2.28 apresenta os passos do método de *Runge-Kutta*. Em (a), é dada a função $f(x)$ em função do tempo t_0 e mais um deslocamento h. Em (b), é estimada a função utilizando a aproximação de *Euler*. Em (c), a metade da estimativa de *Euler* é obtida para aproximar a segunda derivada. Em (d), é calculada a nova estimativa no ponto $x(t_0) + k_2h$. Observe que a aproximação foi bem superior ao método de *Euler*. Portanto, temos:

$P_1 = f(t) + f'(t)h$
$P_2 = f(t) + 0.5P_1$
$f(t + h) = f(t) + P_2h$

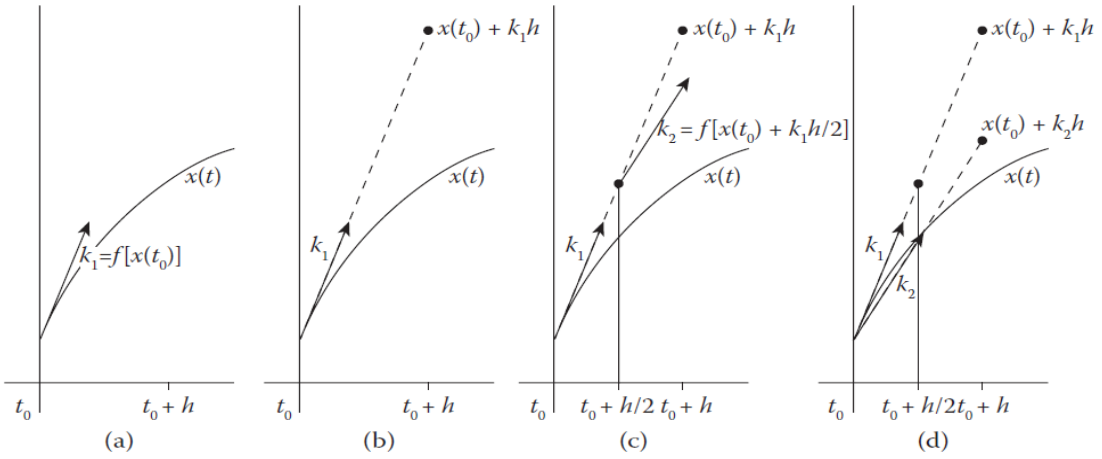

Figura 2.28: Aproximação utilizando método de *Runge-Kutta* (ilustração retirada de [48]).

Método de *Verlet*

Os métodos apresentados acima são baseados na aceleração e no tempo, portanto, faz-se necessário conhecer os estados da velocidade e posição. Diferentemente, o método *Verlet* assume que a aceleração é determinada pela posição, não pela velocidade [48]. Para este método assumimos que dois estados são conhecidos: a aceleração e a posição. Dado x a posição, e a a aceleração, assumimos que $x(t)$, $x(t+1)$, $a(t)$ e t o tempo são conhecidos e queremos calcular $x(t+2)$. Com as duas posições anteriores definidas por $x(t)$ e $x(t+1)$, pode-se estimar a velocidade através da aproximação

$$v \approx \frac{x(t+\Delta t) - x(t)}{\Delta t}. \tag{2.1}$$

Para entender a derivação do método *Verlet* suponhamos os valores $x(t)$, $x(t+\Delta t)$ e $x(t+2\Delta t)$, poderíamos estimar a velocidade entre as duas primeiras posições como

$$v(t+0.5\Delta t) = \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{2.2}$$

e a segunda posição como

$$v(t+1.5\Delta t) = \frac{x(t+2\Delta t) - x(t+\Delta t)}{\Delta t}. \tag{2.3}$$

Assim, poderemos estimar a aceleração entre as duas velocidades $a(t+\Delta t)$

$$\begin{aligned} &= \frac{v(t+1.5\Delta t) - v(t+0.5\Delta t)}{\Delta t} \\ &= \frac{\frac{x(t+2\Delta t)-x(t+\Delta t)-x(t)}{\Delta t} - \frac{x(t+\Delta t)-x(t)}{\Delta t}}{\Delta t} \\ &= \frac{x(t+2\Delta t) - 2x(t+\Delta t) + x(t)}{\Delta t^2}. \end{aligned} \tag{2.4}$$

Isto pode ser resolvido como:

$$x(t + 2\Delta t) = 2x(t + \Delta t) - x(t) + a(t + \Delta t)\Delta t^2 \tag{2.5}$$

O ponto chave para a integração *Verlet* é que nos permite escrever a posição $t + 2\Delta t$ como uma função da posição no tempo t e $t + \Delta t$, e a aceleração em $t + \Delta t$.

Este método é derivado da diferenciação de segunda ordem, portanto, o método *Verlet* também é considerado como método de segunda ordem, com precisão relativamente boa. Um problema talvez já observado nesse método é a necessidade de obter uma posição e aceleração iniciais. Alguns métodos são utilizados para se obter a segunda posição inicial, assim satisfazendo os requisitos preliminares para executar o método *Verlet*.

Método *Velocity Verlet*

Um aprimoramento ao método *Verlet* e matematicamente equivalente é o método *Velocity Verlet* dado por:

$$x(t + \Delta t) = x(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2, \tag{2.6}$$

$$v(t + \Delta t) = v(t) + \frac{1}{2}a(t + \Delta t) + a(t)\Delta t. \tag{2.7}$$

O *Verlet* em sua forma de velocidade tem algumas vantagens sobre o método *Verlet*. Além de considerar a velocidade na integração da posição, reduz erros de arredondamento e não tem o problema da inicialização do método *Verlet* original.

Personalizando os métodos de integração

De maneira geral, os métodos de integração podem ser personalizados com a adição de parâmetros para controlar propriedades da simulação física, a exemplo da velocidade da física. Para simular fricção, pode ser adicionado um parâmetro **d**, geralmente chamado de *damping*, utilizado para reduzir a velocidade final do objeto a cada passo da integração. Este parâmetro é aplicado multiplicando seu valor em $v(t + \Delta t)$. Sendo limitado entre [0,1) para nunca aumentar a velocidade final, mas sim manter ou reduzir a velocidade do objeto.

2.6.2 Deformação sem malha Baseada em Casamento de Formas

Esta técnica, também conhecida como *Meshless deformation Based on Shape Matching* [55], apresenta uma abordagem para corpos rígidos e deformáveis através de um modelo geométrico, sem a necessidade de informações sobre a malha e de um esquema de integração explícito. Os modelos existentes baseado nas regras da física, em conjunto com a estabilidade e esquemas de integração implícito possuem alto custo computacional [55],

além dessas abordagens não permitirem simulação interativa de objetos com geometrias relativamente complexas.

O princípio básico desta técnica tem como entrada um conjunto de partículas do corpo com suas respectivas massas e uma configuração inicial, que são as posições iniciais da partículas. Ao final de cada iteração da simulação, as partículas são deslocadas para posição final $\mathbf{g_i}$. Para calcular a posição final de cada partícula, é casada a configuração inicial (ou forma) definida por $\mathbf{x}_\mathrm{i}^0$ com a configuração definida pela atual posição da partícula $\mathbf{x}_\mathrm{i}$, conforme apresentada na Figura 2.29.

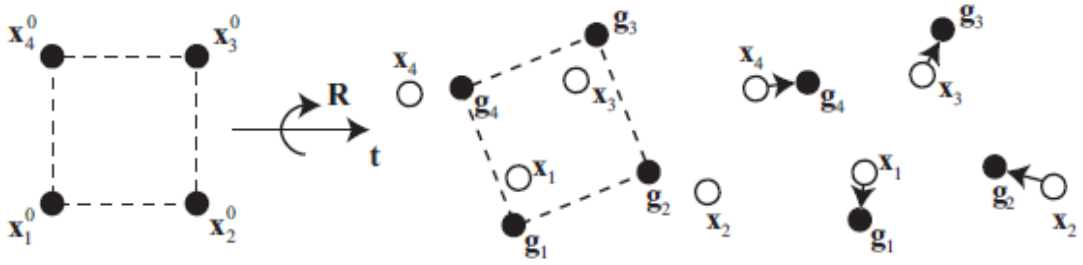

Figura 2.29: Associação da forma inicial com a forma atual (ilustração retirada de [55]).

O problema de casar uma forma inicial à forma atual consiste em: dados dois conjuntos de partículas $\mathbf{x}_\mathrm{i}^0$ (inicial) e $\mathbf{x}_\mathrm{i}$ (atual), encontrar uma matriz de rotação $\mathbf{R}$ e os vetores de translação que minimizam

$$\sum_{j=1}^{n} \mathbf{w}_\mathrm{i}(\mathbf{R}(\mathbf{x}_\mathrm{i}^0 - \mathbf{t}_0) + \mathbf{t} - \mathbf{x}_\mathrm{i})^2, \tag{2.8}$$

onde n é a quantidade de partículas, $\mathbf{w}_i$ é o peso de cada partícula e $\mathbf{t}$ é o vetor de translação. O vetor de translação ótimo é a diferença do centro de massa da configuração inicial em relação ao centro de massa da forma atual, que é fisicamente plausível. Este pode ser computado da seguinte maneira: Seja cm o centro de massa do corpo e $\mathbf{m}_\mathrm{i}$ a massa da da i-ésima partícula, temos que

$$\mathbf{t}_0 = \mathbf{x}_\mathrm{cm}^0 = \frac{\sum_i \mathbf{m}_\mathrm{i}\mathbf{x}_\mathrm{i}^0}{\sum_i \mathbf{m}_\mathrm{i}}, \mathbf{t} = \mathbf{x}_\mathrm{cm} = \frac{\sum_i \mathbf{m}_\mathrm{i}\mathbf{x}_\mathrm{i}}{\sum_i \mathbf{m}_\mathrm{i}}. \tag{2.9}$$

Para encontrar a matriz de rotação $\mathbf{R}$, o problema se restringe a encontrar uma matriz de transformação rígida $\mathbf{A}$, de forma que minimize a equação

$$\mathbf{A} = (\sum_{i=1}^{n} \mathbf{m}_\mathrm{i}\mathbf{p}_\mathrm{i}\mathbf{q}_\mathrm{i}^T)(\sum_{i=1}^{n} \mathbf{m}_\mathrm{i}\mathbf{q}_\mathrm{i}\mathbf{p}_\mathrm{i}^T)^{-1} = \mathbf{A}_\mathrm{pq}\mathbf{A}_\mathrm{qq}, \tag{2.10}$$

sendo $\mathbf{p}_\mathrm{i} = \mathbf{x}_\mathrm{i} - \mathbf{x}_\mathrm{cm}$ e $\mathbf{q}_\mathrm{i} = \mathbf{x}_\mathrm{i}^0 - \mathbf{x}_\mathrm{cm}^0$.

O segundo termo $\mathbf{A}_\mathrm{qq}$ é uma matriz simétrica contendo apenas a escala. Portanto, a

parte rotacional está contida em $\mathbf{A}_{pq}$, que pode ser encontrada através de decomposição polar $\mathbf{A}_{pq} = \mathbf{RS}$, no qual $\mathbf{S}$ é a parte simétrica, dada por $\mathbf{S} = \sqrt{\mathbf{A}^{T}{}_{pq}\mathbf{A}_{pq}}$ e a parte rotacional $\mathbf{R} = \mathbf{A}_{pq}\mathbf{S}^{-1}$. Sendo assim, é possível encontrar a posição final através de

$$\mathbf{g}_i = \mathbf{R}(\mathbf{x}_i^0 - \mathbf{x}_{cm}^0) + \mathbf{x}_{cm}. \tag{2.11}$$

Capítulo 3

Método Proposto

Foram realizados experimentos através de simulações com diferentes volumes envolventes buscando melhor eficácia e eficiência entre eles, considerando a quantidade de eliminação de pares de colisão e o custo computacional na etapa da *Broad Phase*. Além disto, também é analisado o resultado obtido com a técnica de *PCA* e os algoritmos aplicados na *Narrow Phase*.

O algoritmo empregado na *Broad Phase* é baseado em *SaP* [2] [1] usando alternativamente *18-DOP*s, *AABB*s ou Esferas como volumes envolventes, com processamento em *GPU*, sendo selecionado o melhor eixo utilizando a *PCA* [50] [13] [14]. A *AABB* pode ser considerada como 6-DOP. O volume envolvente *18-DOP* utiliza, além dos 3 eixos e 6 semiespaços plano mencionados anteiormente da *AABB*, os eixos inclinados em 45° graus nos planos XY, XZ e YZ, totalizando 9 eixos e 18 semiespaços plano.

De forma geral, as simulações primeiro executam a criação dos volumes envolventes, seguida de uma ordenação dos mesmos, para a qual é empregado o algoritmo de ordenação por dígito (*Radix Sorting*) [3] em *GPU* para um único eixo. Posteriormente, é feita uma varredura nos volumes envolventes, com objetivo de identificar quais volumes se sobrepõem quando projetado nesse eixo. Sendo assim, obtém-se uma lista de pares em potencial colisão. A partir desse resultado, é analisado o tempo necessário para realizar a ordenação, bem como a taxa de de eliminação dos pares de objetos que não se intersectam, usando os diferentes volumes envolventes.

Na *Narrow Phase*, tem-se como entrada os pares em potencial colisão remanescentes da etapa anterior e é realizada uma verificação de colisão detalhada das malhas utilizando o algoritmo *GJK*. Sendo positivo o resultado do *GJK*, ou seja, confirmando a colisão do par de objetos, é utilizado o algoritmo *EPA* para obtenção dos detalhes da colisão, tais como a profundidade de penetração e a direção da colisão.

Com os detalhes da colisão fornecidos pelo sistema de detecção de colisão, é feita a simulação da dinâmica de corpos. Neste caso, foi empregada a técnica *Meshless Deformation Based on Shape Matching* por ser um algoritmo geométrico, relativamente simples de implementar, com estabilidade e suporte para corpos rígidos e deformáveis.

Para implementação do método proposto, foi utilizada a linguagem C/C++ pelo fato de ser uma linguagem consolidada, robusta, apropriada para sistemas de alta performance. Não foram utilizadas bibliotecas matemáticas externas (ex.: álgebra linear), além da padrão fornecida pela linguagem C/C++. A ferramenta de ambiente de desenvolvimento integrado (*IDE*) utilizada foi *Microsoft Visual Studio 2019 - Community Edition* na versão 16.9. A seguir são enumeradas as bibliotecas utilizadas, bem como suas funções:

- *Open Graphic Library* (*OpenGL*) - utilizada no processo de renderização da cena e dos objetos;
- *Open Computing Language* (*OpenCL*) - utilizada para acesso e processamento em *GPU*;
- *OpenGL Extension Wrangler* (*GLEW*) - utilizada em conjunto com a *OpenGL* para facilitar acesso às suas extensões;
- *GLFW* - utilizada para facilitar a utilização do sistema de janelas do sistema operacional em conjunto com a *OpenGL*, além do processamento dos componentes de entrada e saída, como teclado e *mouse*;
- *Dear ImGUI* (*ImGUI*) - utilizada para criação e exibição de controles/componentes de formulários e texto na tela;
- *GoogleTest* - utilizada para execução de testes unitários automatizados em diferentes plataformas, como *Windows*, *Linux* e *MacOS*;
- *Microsoft Test* (*MSTest*) - utilizada para execução de testes unitários automatizados integrada com a ferramenta *Microsoft Visual Studio 2019 - Community Edition*;

Dada a complexidade apresentada para o desenvolvimento do projeto de simulações, nosso trabalho, o qual denominamos de *Spectrum Engine*, foi arquitetado em quatro módulos:

- *Foundation* - define a base do sistema, contendo a definição de tipos de dados, funções utilitárias, gerenciamento de arquivos, imagens, alocação de memória personalizada e dispositivos de entrada e saída;
- *Physics* - este módulo contemplada todos os aspectos matemáticos e físicos, como a definição de matrizes, vetores, volumes envolventes, procedimentos da detecção de colisão (*Broad Phase* e *Narrow Phase*), algoritmo para dinâmica de corpos e métodos numéricos em geral;

- *Rendering* - este módulo engloba todos os aspectos de renderização, tendo como base a *API OpenGL*, assim como suas abstrações para suportar outras *API*s de renderização e códigos de *shader*. Este módulo se destaca principalmente pelo uso da técnica de *Instancing*, a qual realiza a renderização otimizada de grande quantidade de instâncias de objetos que possuem mesma malha, porém com transformações diferentes (posição, orientação e escala) para cada instância;

- *Front-End* - este módulo é responsável pela apresentação do sistema e suas visões, como a visão de logs, inspeção de objetos, barra de status, e principalmente a visão da cena. Este módulo se destaca principalmente pelo uso da biblioteca *Dear ImGUI*, a qual é responsável por exibe os componentes de interface com o usuário.

Na Figura 3.1 é possível observar os módulos com suas relações dentro da *Spectrum Engine*.

O módulo *front-end* tem como dependência todos os demais módulos, pois este necessita do módulo *Foundation* para criar a interação de interface com os dispositivos de entrada e saída, alocação de memória, assim como demanda do módulo que contém funções matemáticas (*Physics*) para realizar pequenos cálculos de ambientes 2D durante a apresentação da interface, em conjunto com a biblioteca de renderização, por exemplo, para obter o *framebuffer* da *API* de renderização.

O módulo *Physics* pode ser considerado o principal módulo do nosso trabalho e depende apenas do módulo *Foundation*, o qual é utilizado para a alocação personalizada de memória, leitura de arquivos contendo o código que será compilado em tempo de execução e executado pela *GPU*.

O módulo *Rendering* tem como dependência apenas o módulo *Foundation* para acessar a alocação personalizada de memória e leitura de arquivos de *shader*.

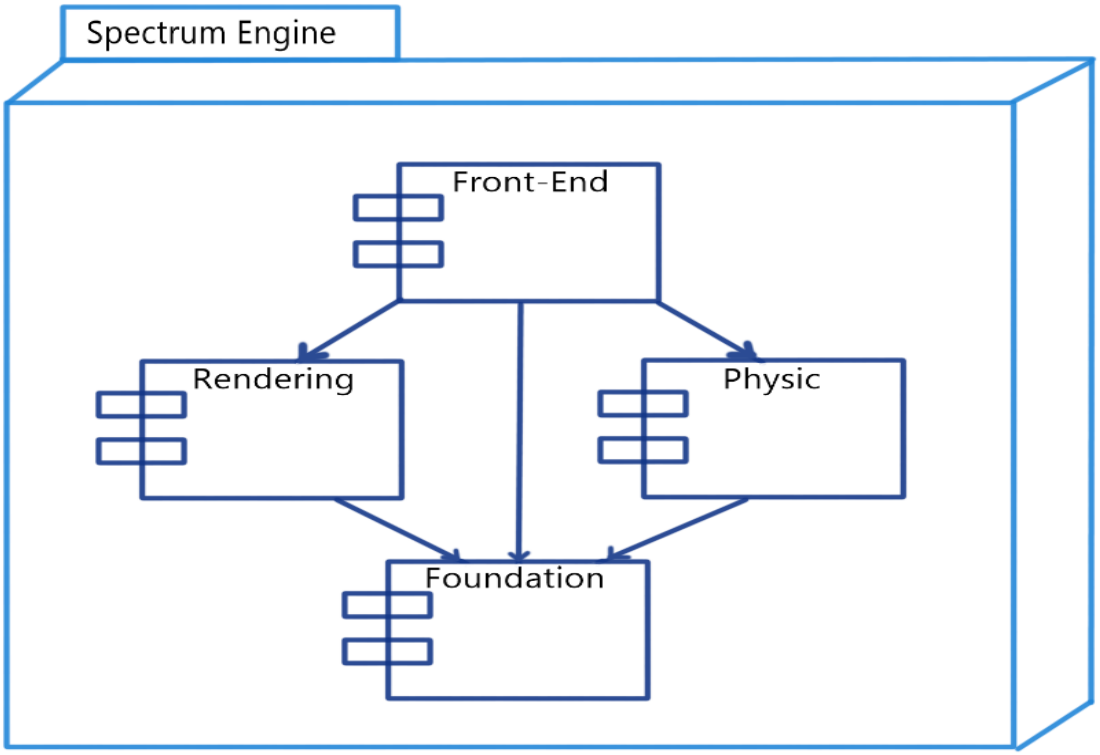

Figura 3.1: Diagrama de componentes da *Spectrum Engine*.

3.1 Considerações Iniciais

Na fase de inicialização da aplicação o arquivo contendo código *OpenCL* é lido do disco, carregado e compilado de forma específica para cada *GPU*. Desta forma, o código pode ser executado durante a simulação sem requerer nova inicialização.

Com objetivo de esclarecer todo processo de simulação físico realizado, a Figura 3.2 apresenta todos os passos que ocorrem, de forma macro, a cada quadro.

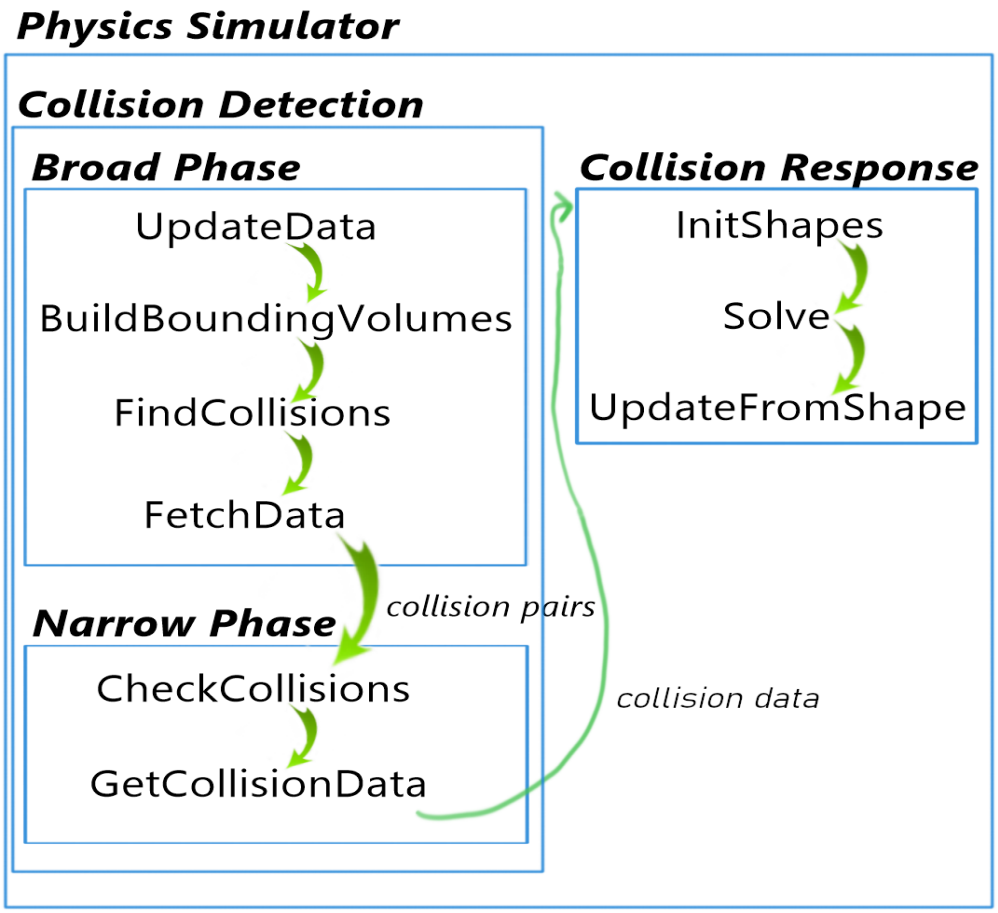

Figura 3.2: Passos realizados dentro do simulador físico executado a cada passo.

Como se observa, nosso simulador é dividido no detector de colisões (*Collision Detection*) e na resposta a colisões (*Collision Response*), sendo a detecção de colisão subdivida nas etapas de *Broad Phase* e *Narrow Phase*.

Antes de iniciar a simulação, o módulo de física é inicializado com os objetos e suas respectivas malhas. As informações das malhas são copiadas, inseridas e mantidas em *GPU*. Também deve ser considerado que a cada 30 quadros é executado o procedimento de seleção do eixo a ser utilizado pelo *Sweep and Prune* através da *PCA*.

O simulador inicia no procedimento *UpdateData*, o qual transfere para a *GPU*, através de comandos *OpenCL*, as transformações dos objetos, contendo posição, orientação e escala. Posteriormente, é executado um procedimento que transforma todos os vértices de todas as malhas para as posições globais utilizando a estrutura de transformação do objeto (posição, orientação e escala), assim armazenando todos os vértices transformados em *GPU*. Após esta etapa, o procedimento de construir volumes envolventes, detalhado nas próximas seções, é executado utilizando os vértices das malhas computados anteriormente. Com os volumes envolventes construídos, é possível executar o comando de identificar colisões de volumes envolventes, consistindo em construir os pontos extremos dos volumes envolventes em um eixo selecionado pela *PCA*, ordenar os volumes envolventes,

realizar uma varredura para ignorar os pares que estão consideravelmente distantes, mantendo apenas uma lista de pares em potencial colisão, cujos os volumes envolventes estão em interseção. Com a lista de pares na memória da *GPU*, esta é recuperada para a *CPU* - através do procedimento *FetchData* - para que a etapa *Narrow Phase* seja executada.

Com os pares em potencial colisão, a etapa *Narrow Phase* inicia com o procedimento *CheckCollisions* iterando sobre todos os pares e verificando de forma precisa se os objetos envolvidos em cada par estão de fato em colisão, através do algoritmo *GJK*. Sendo confirmado o contato entre os objetos no par de colisão, esses permanecem para serem processados pela próxima etapa *GetCollisionData*, a qual obtém as informações de colisão, como profundidade de penetração e a normal da colisão.

Com as informações de contato e o diagnóstico de colisão obtidos, a resposta a colisão (*Collision Response*) itera sobre todos os objetos em colisão e cria uma estrutura, armazenando as informações da forma inicial dos objetos antes da resposta à colisão. Este procedimento foi denominado *InitShapes*.

Para cada par em colisão, o procedimento *solve* é executado diversas vezes, fazendo com que os objetos sejam separados através da técnica de casamento de formas (*Shape Matching*). Nosso método executa exatamente 10 vezes este procedimento para uma convergência global entre todos os objetos envolvidos em cena, visto que um mesmo objeto pode estar contido em vários pares de colisão em um único quadro, e a técnica de *Shape Matching* pode não separar totalmente um dado par de objetos em uma única execução.

Para finalizar a resposta à colisão, o procedimento *UpdateFromShape* é executado uma única vez. Esse procedimento é semelhante ao *solve* mencionado anteriormente, porém - além de resolver a resposta à colisão - ele atualiza definitivamente a posição e orientação do objeto na estrutura de transformação. Após toda a execução do simulador de física, o processo de renderização é realizado.

A seguir os procedimentos abordados nesta seção são descritos em detalhes.

3.2 Construindo Volumes Envolventes

Uma das primeiras etapas do processo de simulação física de corpos rígidos é a criação da lista dos volumes envolventes **VE**, sendo um volume para cada objeto em cena. Antes de ser computada, é feito um pré-processamento que atualiza todos os vértices de todos os objetos em coordenadas locais para coordenadas globais, de acordo com as informações de transformação. Essa estrutura é composta pela posição central do objeto (vetor de 3 posições, sua orientação (*quaternion*) e escala (vetor de 3 posições). Este processamento recebe como entrada uma lista de vértices de todos os objetos e se obtém como resultado uma lista de vértices atualizados em coordenadas globais.

O cálculo de **VE** é realizado em *GPU*, tendo como parâmetros de entrada a lista de vértices atualizados de cada objeto, retornando uma lista de volumes envolventes.

Caso **VE** contenha AABBs, são obtidos os vértices extremos em cada orientação, formando uma estrutura de valores min(**x**, **y**, **z**) e max(**x**, **y**, **z**). Este processo pode ser facilmente generalizado para k-DOPs, onde para cada uma das k orientações serão armazenados os valores mínimo ($min(k_i)$) e máximo ($max(k_i)$), onde *i* é o índice de uma orientação.

Caso **VE** contenha esferas, é suficiente computar os pontos extremos $min(\mathbf{c} - \mathbf{r})$ e $max(\mathbf{c} + \mathbf{r})$ em uma direção, onde **c** é a posição central do objeto (centro de massa) e **r** é a distância do vértice mais distante do centro de massa.

Todo processo de construção de **VE** tem custo computacional de $O(n)$, onde n é o maior número de vértices que um objeto possui. No nosso caso $n = 24$ para todos os objetos.

3.3 Selecionando o melhor eixo com *Principal Component Analysis*

O algoritmo *Sweep and Prune* consiste em ordenar e varrer a lista de objetos dado um eixo qualquer. Entretanto, uma escolha ruim desse eixo pode levar o *SaP* a ser ineficiente, caso as projeções dos objetos sobre este eixo não sejam em grande parte disjuntas, ou seja, caso não seja possível encontrar muitos planos separadores na primeira passada.

Para contornar esse problema, nossa implementação faz uso da *Principal Component Analysis*, ou simplesmente *PCA*, uma técnica estatística de análise exploratória de dados, que busca encontrar o autovetor associado ao maior autovalor da matriz de covariância. Em nosso contexto, os dados são o centro de massa de cada objeto em cena.

Nossa implementação inicia criando uma matriz Mx3, onde M é a quantidade de objetos em cena e os valores são as posições do centro de massa de cada objeto (x, y, z). Com a matriz criada, é utilizado o método numérico de decomposição em valores singulares, também conhecido como *SVD*. Este método consiste em multiplicar a matriz fornecida pela sua transposta dividindo por M. Isto resulta em uma matriz de covariância $M_{c3x3} = (MM^{\mathrm{T}})/n$. Encontrando os autovalores e autovetores de M_c, obtém-se o autovetor que possui o maior autovalor associado a ele. Esse autovetor é normalizado e é exatamente o vetor (eixo) que melhor distribui os dados (objetos em cena), acelerando o procedimento *Sweep and Prune*. A Figura 2.18 apresenta um exemplo do eixo selecionado em azul e os dados na cor vermelha.

3.4 Ordenando com *Radix Sorting*

Este algoritmo foi escolhido por se adequar bem ao paradigma de paralelismo SIMD. A versão sequencial desse algoritmo tem complexidade $O(n)$ e, em paralelo, sua comple-

xidade cai para $O(nk/p)$ onde p é o número de processadores e k a quantidade máxima de dígitos dos números a serem ordenados. Além disso, é um algoritmo que pode ser implementado sem o uso comparações, o que é benéfico quando implementado em *GPU*s, conforme demonstrado em [2].

O processo de ordenação, antes de sua execução, cria uma lista de índices **IN** para a lista de volumes envolventes **VE**. Desta forma, o processo pode ser conduzido ordenando **IN** ao invés de **VE**, movimentando menos memória. Como o eixo a ser varrido pelo *Sweep and Prune* é dinâmico, selecionado pela *PCA*, ordena-se os volumes envolventes baseado nesse eixo. Caso os volumes envolventes sejam do tipo *AABB* ou k-DOP, é utilizado o eixo mais próximo ao selecionado pela *PCA*. Caso os volumes sejam do tipo esfera, é utilizado um dos 9 eixos associados ao 18-DOP. Os elementos a serem ordenados correspondem às projeções de cada volume envolvente sobre o eixo. Seja u o vetor correspondente à direção do eixo escolhido e seja P um volume envolvente. Então, cada elemento é dado por um intervalo $[min(P, u), max(P, u)]$, onde

$$min(P, u) = \min_{p \in P} p \cdot u, \text{e}$$

$$max(P, u) = \max_{p \in P} p \cdot u.$$

Esses elementos são construídos por um procedimento denominado *BuildElements* e são ordenados pelos seus valores mínimos utilizando *Radix Sorting*. Nossa implementação é baseada em [3], sendo que os valores são representados em ponto fixo com 9 dígitos decimais, correspondentes a potências entre 10^{-4} até 10^4, ou seja, 5 dígitos para a parte inteira e 4 dígitos para a parte decimal.

Nos nove primeiros passos do algoritmo, os dígitos correspondentes são tratados de forma análoga à descrita em [3], com 10 baldes (*"buckets"*) sendo utilizados para cada um dos 10 algarismos. Um último passo, entretanto, é realizado para tratar números negativos, no qual a contagem (*Count)* emprega 2 baldes nos quais são depositados os valores negativos e positivos, respectivamente. Nesta última rodada, o procedimento *Prefix Scan* é executado da mesma maneira à dos demais passos, mas o procedimento *Reorder* é diferenciado, pois ao ordenar considerando apenas o sinal (positivo ou negativo), os números negativos estarão em ordem inversa. Portanto a única diferença no *Reorder* ocorre durante cálculo do índice para os números negativos.

Para elucidar melhor o processo, considere um conjunto com os números inteiros $\{4, -2, 3, -1, -4, -2, 3, 5, 0\}$. Na primeira iteração do *Radix Sorting*, a qual considera o último dígito, os números serão ordenados da seguinte maneira: $\{0, -1, -2, -2, 3, 3, 4, -4, 5\}$. Observa-se que o *Radix Sorting* mantém a ordem de entrada dos elementos no caso de duplicatas, e por este motivo o 4 antecede o -4. Para tratar os números negativos, é realizada mais uma iteração, porém sem considerar a grandeza dos dígitos, mas apenas o sinal. Diferentemente de um dígito decimal que o domínio consiste de 0-9, o sinal pode ser

positivo ou negativo. Para o elemento zero é indiferente ser tratado como positivo ou negativo em nossa implementação. Isto permite obter a ordenação final copiando para o resultado o balde correspondente aos números negativos de forma invertida, resultando em $\{-4, -2, -2, -1, 0, 3, 3, 4, 5\}$. Em particular, na etapa *Count* são criados grupos de dois baldes para contar a quantidade de ocorrências de números positivos e negativos. Cada *thread* da *GPU* cria dois baldes. Assumimos que nossa *GPU* fictícia tenha apenas 3 *threads*. Assim temos que cada *thread* irá processar 3 elementos, visto que o conjunto todo de entrada é composto por 9 elementos. A figura 3.3 ilustra o processo.

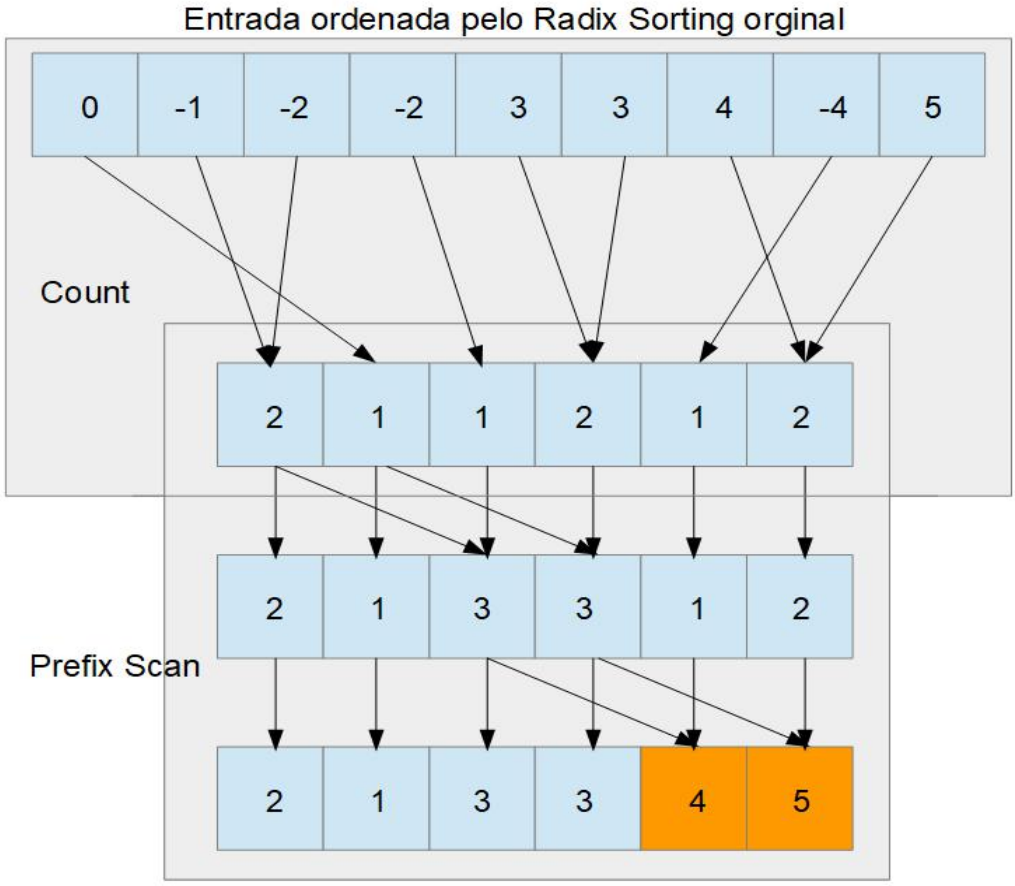

Figura 3.3: Procedimentos *Count* e *Prefix Scan* do *Radix Sorting* para números negativos.

Observa-se que na tabela de *offsets* do *prefix scan*, os 2 últimos valores (marcados em laranja) contêm o somatório de todos os números positivos e negativos, respectivamente em cada balde, ou seja, 4 números negativos e 5 positivos neste exemplo.

Uma vez computada a tabela de *offsets*, obtida pelo processo *Prefix Scan*, esta é usada como entrada para o procedimento *Reorder*. No caso da última etapa que consiste em separar os positivos dos negativos, é usado o Algoritmo 1 que é executado para cada *thread* da *GPU*. O processo é ilustrado na Figura 3.4 que mostra os valores computados em cada

Algorithm 1 Procedimento *Reorder* do *Radix Sorting* para números negativos.

procedure REORDERNEGATIVES
  $negCount \leftarrow offsetTable[lastIndex - 1]$ ▷ Quantidade de Negativos
  $ix \leftarrow$ *thread index*
  for $i = lastElement_{\text{ix}}$ **to** $firstElement_{\text{ix}}$ **do**
    if $isPositive(input_{\text{i}})$ **then** ▷ Verifica se o elemento é positivo
      $output_{\text{i}} = negCount + offsetTable[i] - 1$
    else
      $output_{\text{i}} = negCount - offsetTable[i]$
    $dec(offsetTable[i])$ ▷ Decrementa o valor contido em offsetTable[i].

thread. Nesta figura, o primeiro vetor contém os elementos ordenados desconsiderando o sinal e o segundo vetor é a tabela de *offsets* gerada pela etapa *Prefix Scan*. As cores das setas indicam os valores considerados por cada *thread*.

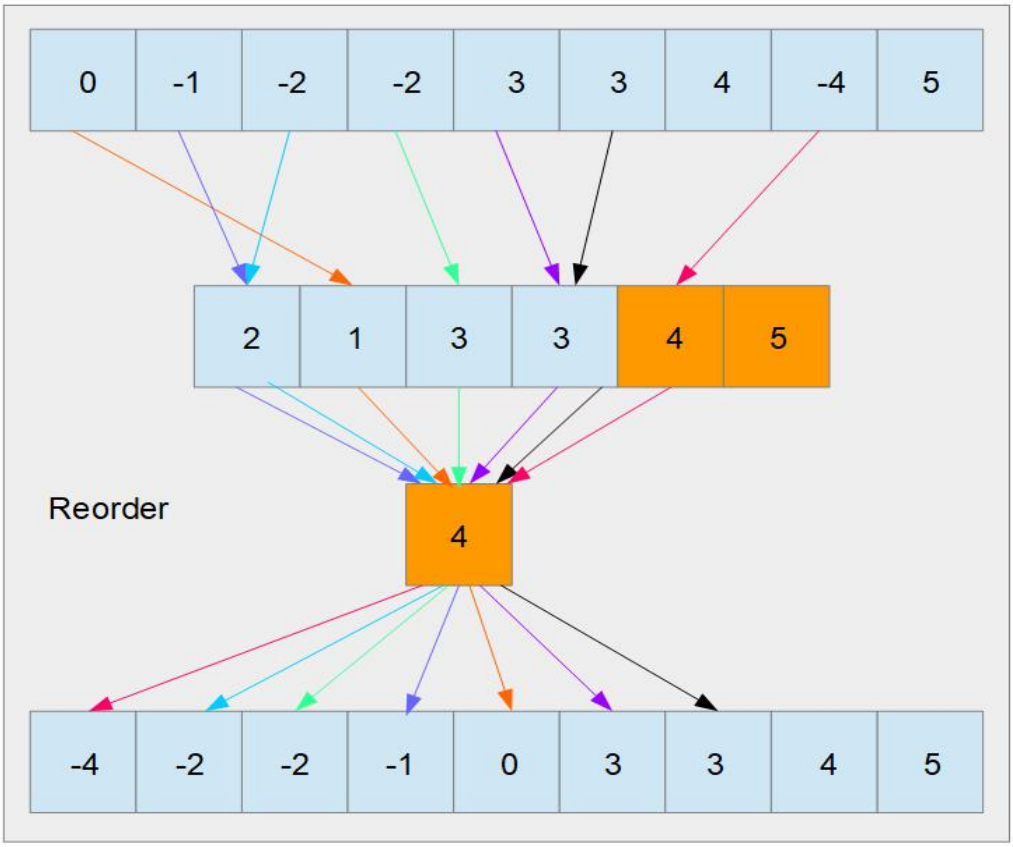

Figura 3.4: Procedimento *Reorder* do *Radix Sorting* para números negativos.

3.5 Implementação do algoritmo *Sweep and Prune*

Com a lista de volumes envolventes **VE** ordenada através do *Radix Sorting*, o algoritmo *SaP* inicia uma varredura no eixo selecionado pela *PCA*. Cada *thread* da *GPU* é responsável por um elemento $\mathbf{VE}_t$, onde t é o índice da *thread*, realizando uma busca por outros elementos que iniciam (contêm o mínimo) no intervalo mínimo e máximo do ele-

mento em questão ($\mathbf{VE}_i$).

Algorithm 2 Detecção de Colisão de volumes envolventes na *Broad Pahse*.

```
procedure CHECKCOLLISIONS
    pairs ← {}
    for i = min(VE_t) to max(VE_t) do
        if min(VE_i) ∩ VE_t then                ▷ Verifica se os volumes se intersectam
            paris.insert((VE_i, VE_t))
```

É verificado apenas se o elemento mínimo ($\mathbf{VE}_i$) de cada volume está contido no intervalo do volume envolvente da *thread* [*min*($\mathbf{VE}_t$), *max*($\mathbf{VE}_t$)]. Isto é feito para que não haja pares duplicados na lista, como (A, B) e (B, A). Sendo identificada a interseção entre o par $\mathbf{VE}_i$ e $\mathbf{VE}_t$, os objetos correspondentes são adicionados à lista de pares em potencial colisão que é retornada pelo *SaP*.

Na Figura 3.5, é ilustrado o algoritmo sendo executado em 2D. Como cada *thread* executa um elemento, teremos quatro *threads* neste caso. A *thread* 1 realiza a verificação no intervalo do elemento 1, portanto identifica apenas que o elemento 2 está em seu intervalo, sendo assim, o processo continua verificando se os objetos 1 e 2 se intersectam nos demais eixos utilizando o volume envolvente. Como se observa, o par em questão não se intersecta. A *thread* identifica que não há mais elementos no intervalo do elemento 1 e finaliza sua execução. Paralelamente, a *thread* 2 é executada e identifica que apenas o elemento 3 está em seu intervalo, sendo descartado por não se intersectar nos demais eixos. O elemento 1 é ignorado pela *thread* 2, pois esse elemento não inicia dentro do intervalo do elemento 2. Isto ocorre para que não haja verificação duplicada. As *threads* 3 e 4 também são executadas em paralelo e ambas identificam que não há nenhum outro elemento que inicia dentro de seus intervalos respectivamente.

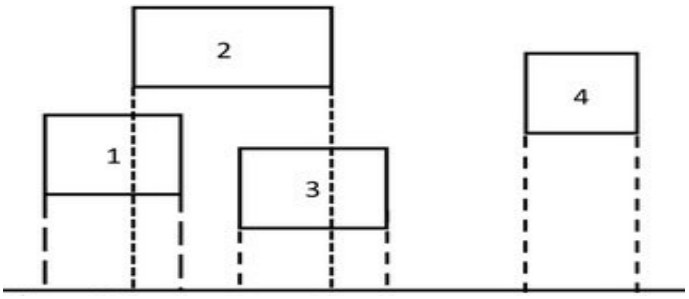

Figura 3.5: Exemplo de execução do algoritmo *SaP*.

3.6 Detectando uma colisão precisa com *GJK* e *EPA*

Após a execução da *Broad Phase* retornar um conjunto de pares de objetos possivelmente em colisão, a *Narrow Phase* é computada em *CPU* através do algoritmo *GJK*. Pela simplicidade e ganhos na performance, foi adotado o *GJK* melhorado (*Enhanced GJK*), que

se caracteriza por calcular a função de suporte fazendo uso das informações de conectividade da malha do objeto, percorrendo-a para encontrar um ponto extremo dada uma direção, diferentemente do *GJK* original que verifica todos os vértices para encontrar o ponto extremo.

Nosso *GJK* é implementado de acordo com a Seção 2.5.1. Uma questão importante com relação à performance do algoritmo se refere ao número de iterações necessário para determinar ou descartar a colisão. Com efeito, a quantidade máxima de iterações do *GJK* (ou *EPA*) é o número total de vértices da diferença de Minkowski. Como em nossa simulação os objetos são malhas com 24 vértices, temos um máximo de $24^2 = 576$ iterações no total.

No caso do *GJK*, este verifica todas as combinações dos vértices para certificar se a origem se encontra dentro ou fora do fecho convexo da diferença de *Minkowski* **M**. Quando a origem está próxima ao centro de **M**, facilmente é encontrada pelo *GJK*, visto que este inicia criando uma das faces do tetraedro próximo ao centro de **M** e o vértice extremo na direção da origem. Entretanto, quando a origem está próxima da fronteira de **M**, mais iterações são necessárias para encontrar a origem.

Com o objetivo de estimar o número máximo de iterações do *GJK* em nossas simulações, foi realizado um estudo empírico com 10 milhões de execuções, que revelou um máximo de 17 iterações para o término do *GJK*, sendo que, em média, 4 iterações são necessárias, muito menor portanto que o máximo teórico.

Um estudo similar foi realizado para as iterações de *EPA*. Foram realizadas as mesmas 10 milhões de execuções de *EPA*, que destacou um número máximo de 29 iterações para sua o término do *EPA*, sendo que, em média, 3 iterações são necessárias, muito menor que o máximo teórico.

Outra diferença de nossa implementação é no tratamento de casos onde a origem está muito próxima a uma face ou aresta de **M**. Neste caso o *GJK* pode retornar um tetraedro degenerado, composto de 4 vértices, sendo um deles repetido. Para contornar esta situação, utilizamos novamente a função de suporte procurando um vértice diferente na direção a normal à face composta pelos três vértices distintos do tetraedro degenerado, visto que qualquer outro vértice é aceitável uma vez que a origem já está contida na face composta pelos 3 vértices distintos.

Sendo diagnosticada a interseção do par de objetos, faz-se necessário obter os detalhes da colisão para que seja possível realizar a dinâmica dos corpos. Para este objetivo, o *EPA* é executando, tirando proveito do resultado fornecido pelo *GJK*, ou seja, usando como entrada os vértices do tetraedro gerado pelo *GJK* e computando a normal e a profundidade de penetração da colisão. Conforme vimos na Seção 2.5.2, esses dados são obtidos a partir da face de **M** mais próxima da origem.

3.7 Resposta à colisão

A resposta à colisão recebe como entrada os dados da colisão, que em nossa simulação são fornecidos pelo *EPA*, como a normal da colisão e a profundidade de penetração. Para uma convergência global, isto é, para que todos os corpos rígidos em cena fiquem separados, sem interpenetração, faz-se necessário executar por diversas vezes o processo de detecção de colisão e resposta para um passo de tempo (*time step*), permitindo assim uma física plausível.

A resposta à colisão implementada difere ligeiramente do esquema da Figura 3.2, que tem propósito meramente didático. O procedimento exato é descrito no algoritmo 1.

Algorithm 3 Procedimento de Resposta à Colisão.

```
1: procedure COLLISION RESPONSE(pais)        ▷ Pares de Colisão da Broad Pahse
2:     InitShapes()
3:     for i = 1 to 10 do
4:         CheckCollisions()                  ▷ Execução do GJK
5:         GetCollisionData()                 ▷ Execução do EPA
6:         ShapeMatching()                    ▷ Atualização das Partículas
7:     UpdateFromShape()                      ▷ Atualização do Corpo Rígido
```

O algoritmo inicia com a execução do procedimento *InitShapes* que itera por todos os pares em potencial colisão identificados na etapa da *Broad Phase* para inicializar todos os objetos. Essa inicialização consiste em criar um sistema de partículas correspondentes aos vértices de cada corpo rígido, que serve de suporte à técnica de casamento de forma (*Shape Matching*). Esse sistema de partículas é composto por:

- índice do objeto;
- posição inicial do objeto;
- quantidade de partículas/vértices;
- posição inicial de todas as partículas;
- posição final de todas as partículas.

Note-se que a posição inicial e final de cada partícula é armazenada em coordenadas locais do objeto.

Após a inicialização com *InitShapes*, é iniciado um processo iterativo com 10 repetições. Este número de iterações foi necessário para alcançar um resultado visual plausível em nossos experimentos. Para cada iteração, é realizada uma nova varredura em todos os pares de objetos definidos pela *Broad Phase* e um processo de resolução é executado para cada par, denominado *Solve*, consistindo de três fases: *CheckCollisions*, *GetCollisionData* e *ShapeMatching*.

A primeira etapa (*CheckCollisions*) consiste em verificar se o par ainda se encontra em colisão. Em geral, é utilizado o algoritmo *GJK* para esse fim, mas um tratamento diferenciado é realizado caso um dos objetos do par de colisão seja o chão. Neste caso é consideravelmente mais eficiente apenas calcular se o vértice extremo na direção $[0, -1, 0]^T$ é menor ou igual a zero, indicando se está em contato com o chão. Para encontrar o vértice extremo é utilizada a função de suporte conforme descrito na seção 3.6.

Caso seja identificada uma colisão pelo *GJK*, a execução do *EPA* também é realizada para obter a normal e a profundidade de penetração. Para colisões envolvendo o chão, esses dados são obtidos de forma trivial, com base no vértice com menor coordenada y.

Sejam dois objetos A e B em colisão como mostrado na Figura 3.6 e seja N a normal retornada pelo *EPA*. Então, os vértices de suporte a e b são extremos nas direções N e $-N$, respectivamente. Usando esses dois pontos, define-se dois planos (linhas pontilhadas na Figura 3.6), e todas as partículas p_i associadas a vértices de A, tais que $(p_i - b) \cdot N < 0$, são projetadas, isto é, deslocadas para coincidir com o plano que passa por b. Analogamente, todas as partículas associadas a vértices q_j de B, tais que $(q_j - a) \cdot N > 0$, são projetadas sobre o plano que passa por a. O resultado dessa projeção é ilustrado na Figura 3.6(b).

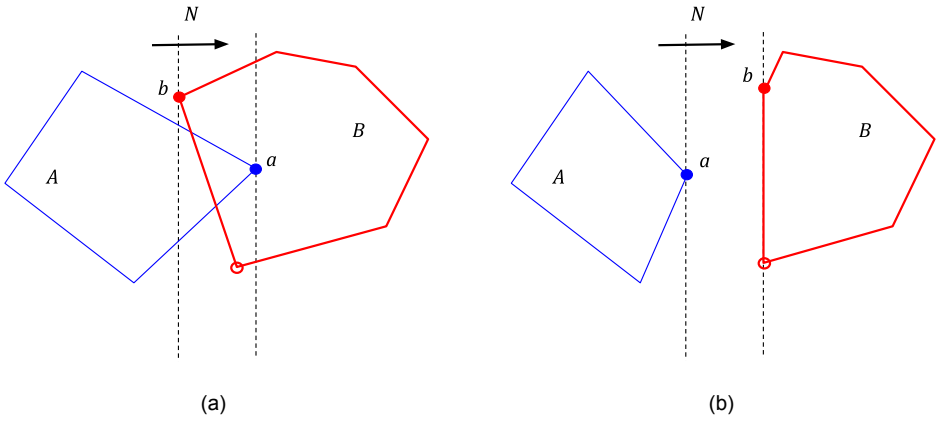

Figura 3.6: (a) identificação dos planos de colisão para cada objeto. (b) partículas atualizadas.

Com as partículas do objeto atualizadas, o processo de *Shape Matching* é executado, conforme Seção 2.6.2, sendo o procedimento de encontrar a raiz quadrada fazendo uso de métodos numéricos de decomposição polar de matriz, conforme definido no apêndice B.

Caso o método de encontrar a raiz quadrada **S** não convirja até um número máximo de iterações, esta etapa é ignorada, o que não é um problema visto que são realizadas 10 etapas de *Shape Matching*.

Após realizar as 10 iterações de detecção e resposta à colisão, a etapa final, *UpdateFromShape*, é executada. Ela consiste em converter a matriz de rotação **R** em um *quaternion*, que será associado a nova orientação do corpo rígido. Além de atualizar a orientação

do corpo, também é atualizada a posição utilizando o centro de massa das partículas atualizadas.

Algumas condições excepcionais devem ser tratadas, em particular, caso o *GJK* ou *EPA* não convirjam, então a iteração corrente é ignorada. Também pode ocorrer de a raiz quadrada da matriz calculada ser singular ou muito próxima de singular. Neste caso não é possível obter uma matriz inversível, logo o procedimento deve ser ignorado. Outra exceção que pode ocorrer é haver uma pequena escala indesejável na diagonal principal quando $\mathbf{A}_{\text{pq}}$ é multiplicada pela matriz $\mathbf{S}^{-1}$ para obter a matriz final de rotação **R**. Caso a escala seja superior a 1% em qualquer um dos eixos, o procedimento também é ignorado, de forma a evitar a propagação de erro numérico para as demais iterações no mesmo quadro. Em algumas circunstâncias, as 10 iterações de *Shape Matching* podem não convergir devido a uma transformação específica do objeto, então é atualizada apenas a translação do objeto, sendo ignorada a rotação.

3.8 O Integrador

Nossa implementação utilizou uma variação do integrador numérico *Verlet*, conhecida como *Velocity Verlet*, conforme descrita na seção 2.6.1. O Δt é definido como uma unidade de integração e foi desconsiderado das fórmulas 3.1 e 3.2. Em nossa simulação há apenas a aceleraçao da gravidade, portanto substituímos $a(t)$ para **g**. Também foi adicionado um parâmetro de amortização da velocidade do objeto **d** denominado *damping* e definido por $\{d|0 \leq d \geq 1 \in \mathbb{R}\}$ para simular fricção com o ar. Portanto, a fórmula final do integrador *Velocity Verlet* incluindo o parâmetro de **damping** é dada por

$$x(t + \Delta t) = x(t) + v(t) + \frac{1}{2}g, \tag{3.1}$$

$$v(t + \Delta t) = (v(t) + g)\mathbf{d}. \tag{3.2}$$

Capítulo 4

Notas de Metodologia

A engenharia de software busca continuamente melhorias na qualidade da produção de software, seja qual for a linha da computação. Recentemente, o desenvolvimento de testes automatizados vem ganhando mais destaques no processo de qualidade de software [60].

Embora seja de conhecimento que o tempo despendido na elaboração e desenvolvimento de testes automatizados, pesquisas recentes [60] apresentam os ganhos e as vantagens na produção, manutenção e evolução do software. A exemplo podem ser citados:

- Permitir testar partes do software sem depender de outras partes estarem completas;
- Engenheiros podem paralelamente testar e corrigir defeitos identificados;
- Simplificar o processo de depuração do código se limitando à pequenas áreas do código;
- Aptidão em testar áreas internas do código que dificilmente são alcançadas com testes manuais;

No segmento de testes automatizados, são concebidos - de maneira geral - três categorias de teste: unitários, integrados e funcionais, conforme observado na Figura 4.1.

Os testes unitários tem objetivo de testar isoladamente a menor unidade de um sistema computacional, no caso uma função, sem suas dependências.

Já os testes integrados, consistem em um nível mais elevado, considerando que as unidades estão em funcionamento garantido pelos testes unitários. Sua ênfase é cobrir a integração entre as pequenas unidades do sistemas, garantindo um sistema consistente e coeso.

Em nível mais abstrato estão os testes de interface ou testes funcionais, que tem por objetivo cobrir funcionalidades completas do sistema. Pode ser citado como exemplo no nosso trabalho a funcionalidade de habilitar e desabilitar a simulação para que um quadro seja analisado pontualmente.

Figura 4.1: Categorias de testes automatizados (ilustração retirada de [60]).

A linha de pesquisa em computação gráfica se apresenta com grau de dificuldade elevado para testes manuais devido as grandes possibilidades de combinação que podem ocorrer durante a execução do software. Extensivos testes manuais seriam necessários para garantir e manter a corretude do software. Realizando testes unitários automatizados é possível reduzir essa carga de teste de forma que cada componente/função seja testada de maneira que a interação de todos os componentes sejam reduzidas de erro.

Nosso trabalho fez uso extensivo de testes unitários automatizados com objetivo de assegurar a qualidade e garantir que toda modificação realizada no sistema tenha impacto reduzido e identificado problemas introduzidos o quanto antes pela modificação.

A abordagem de testes unitários são convidativas principalmente quando o sistema proposto é baseado em cálculos complexos, no qual podem ser facilmente isolados e testados unitariamente, sem a necessidade de abstrações (*mocks*) para isolar os componentes do sistema, principalmente dado que toda biblioteca de álgebra linear foi escrita desde o início, e por este motivo foi selecionada a categoria de testes automatizados para ser aplicada em nosso trabalho.

Foram criados mais de 500 testes unitários cobrindo principalmente os módulos *Foundation* e *Physics* contendo principalmente os aspectos da álgebra linear, físicos e estruturas de dados, visto que nosso trabalho tem amplo foco em cálculos.

Para exemplificar, foram criados testes unitários para testar cada passo do algoritmo *Radix Sorting*, ou seja, foram criados testes para o *count*, *prefix scan* e o *reorder*. Após cada passo confirmar sua corretude, foi realizado um teste completo do algoritmo de ordenação, fornecendo entradas estáticas, processadas em *GPU* e obtido o resultado, sendo este validado por comparação com valores esperados.

Outro exemplo que se destaca dos demais foi o teste para computar os auto-vetores e auto-valores de uma matriz de ordem 3x3. Dada a complexidade do método numérico, por ser iterativo e podendo não convergir, foram tomados exemplos da literatura e comparado com os resultados obtidos pelo software *Matlab* e biblioteca *Eigen* para verificar

a corretude de algumas matrizes, conforme Figura 4.2.

```
SP_TEST_METHOD(CLASS_NAME, eigenValuesAndVectors)
{
    Mat3 matrix = {
        4.0f, 2.0f, 0.0f,
        2.0f, 5.0f, 3.0f,
        0.0f, 3.0f, 6.0f
    };
    Vec3 eigenValues;
    Mat3 eigenVectors;
    sp_uint iterations;
    matrix.eigenValuesAndVectors(eigenValues, eigenVectors,
        iterations, SP_UINT_MAX, SP_EPSILON_FOUR_DIGITS);

    Vec3 expectedValues(1.4516f, 8.9089f, 4.6395f);
    Mat3 expectedVectors = {
        -0.54801f, 0.69826f, -0.46056f,
        0.27285f, 0.6697f, 0.69069f,
        0.79072f, 0.25284f, -0.55753f
    };

    for (sp_uint i = 0; i < 3u; i++)
        Assert::IsTrue(isCloseEnough(expectedValues[i],
            eigenValues[i], SP_EPSILON_THREE_DIGITS),
            L"Wrong value", LINE_INFO());

    for (sp_uint i = 0; i < 9u; i++)
        Assert::IsTrue(isCloseEnough(expectedVectors[i],
            eigenVectors[i], SP_EPSILON_THREE_DIGITS),
            L"Wrong value", LINE_INFO());
}
```

Figura 4.2: Exemplo de teste unitário automatizado para calculo dos auto-vetores e auto-valores de uma matriz 3x3.

Capítulo 5

Experimentos e Discussão

Neste trabalho foram realizados experimentos utilizando seis simulações idênticas, 3 para cada tipo de volume envolvente (*18-DOP*, *AABB* e Esfera), contendo 512 objetos idênticos em cada simulação, sendo as outras 3 simulações contendo os mesmos objetos porém escalados no eixo y em 300%, ou seja, os objetos cujo tamanho original é (2.0, 2.0, 2.0) foram redimensionados para (2.0, 6.0, 2.0). O objeto, em sua forma original, consiste em um octaedro truncado, conforme Figura 5.1, contendo 24 vértices e 26 faces. O objeto escalado é exibido na Figura 5.2. Todas as simulações foram finalizadas no quadro 7.000, dado que todos os objetos terminam de cair aproximadamente no quadro 5.500. Todos os objetos são soltos em posições da forma (0, y_i, 0), onde $y_i = 10i$ para i entre 1 e 512. A orientação de cada objeto é aleatória, porém todas as simulações foram executadas com as mesmas orientações previamente computadas, ou seja, foi gerada uma semente de orientações que é utilizada em todas as 6 simulações. A aceleração da gravidade foi considerada 9,8 por passo de integração. Todos os tempos mencionados nos gráficos estão em milissegundos.

Destaca-se também que, na utilização da *PCA*, esta seleciona um eixo mais próximo dos 9 eixos correspondentes pelo volume do tipo *18-DOP*. Para os volumes do tipo *AABB*, é selecionado um eixo mais próximo dos 3 eixos da *AABB*. Embora para os volumes do tipo *Esfera*, não haja restrição de um eixo específico, nossa implementação se limitou a varrer um eixo dentre os 9 eixos correspondentes às direções do *18-DOP*.

Considerando o alto grau de coerência temporal das simulações, optamos por realizar a análise por componentes principais (*PCA*) apenas uma vez a cada 30 quadros.

O integrador numérico utilizado para todas as simulações foi o *Velocity Verlet* com o parâmetro de *damping* definido como 0.95 e o valor da velocidade física em 0.003 unidades por passo de tempo.

Embora o algoritmo *GJK* seja executado exatamente 10 vezes para cada par em potencial colisão identificado pelo algoritmo *Sweep and Prune*, conforme descrito em detalhes na seção 3.1, o *Sweep and Prune* é executado somente uma vez por quadro. Optamos por manter a execução do *Sweep and Prune* uma única vez por quadro devido ao custo

computacional de preparação do algoritmo com a atualização dos dados para a *GPU*, transformação dos vértices e sua execução em si.

Para implementação e execução dos experimentos foi utilizado o *hardware* de um *notebook Acer PH315-52-79VM*, contendo: um processador *Intel Core i7-7700HQ* com 4 *cores* e 8 *threads*; um módulo de 16Gb de memória do tipo *DDR4 SDRAM*; placa de vídeo discreta *NVIDIA GeForce GTX 1060* com 6Gb de memória.

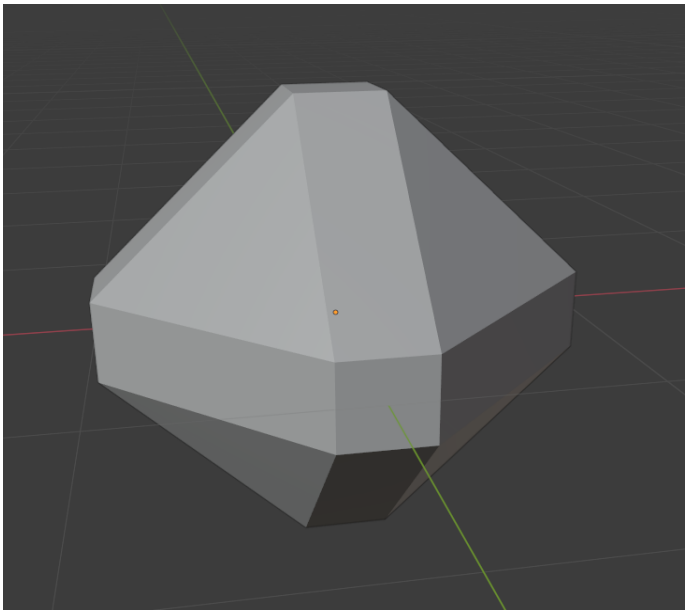

Figura 5.1: Exemplo da geometria do objeto sem escala.

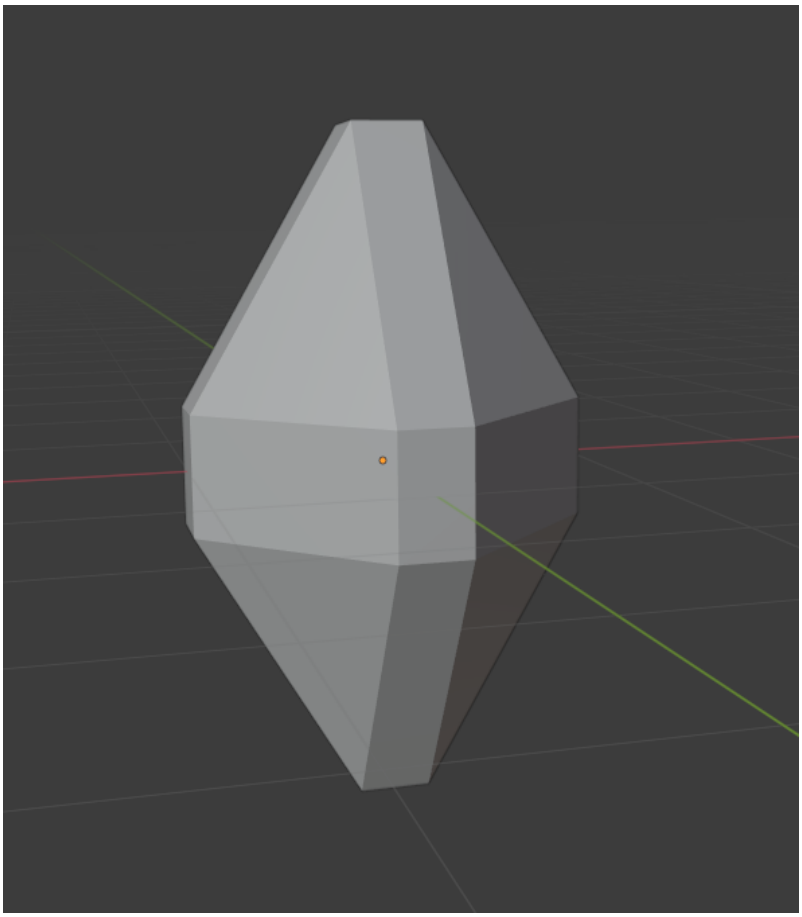

Figura 5.2: Exemplo da geometria do objeto com escala.

Foram realizadas diversas análises da simulação física, levantando as seguintes medidas:

- *CONSTRUÇÃO DOS VOLUMES*: tempo de execução para a construção dos volumes envolventes (Esfera, *AABB* e *18-DOP*) em *GPU*, conforme visto na seção 3.2;
- *SWEEP AND PRUNE*: tempo de execução de todas as etapas do algoritmo *Sweep and Prune* em *GPU*, conforme visto na seção 3.5, isto é:
  - *PROJEÇÃO NO EIXO*: tempo para projeção dos limites mínimos de cada objeto no eixo selecionado pela técnica *PCA*;
  - *RADIX SORTING*: tempo para ordenação dos elementos criados no item *PROJEÇÃO NO EIXO*, conforme visto na seção 3.4;
  - *PODA*: tempo para eliminação dos pares consideravelmente afastados no processo do algoritmo *Sweep and Prune*;
- *GJK+EPA*: tempo de execução da *Narrow Phase* utilizando o *GJK* seguido do *EPA*, caso necessário, conforme visto na seção 3.6;

- *SHAPE MATCHING*: tempo de resposta à colisão utilizando a técnica *Shape Matching*, conforme visto na seção 3.7;
- *RENDERIZAÇÃO*: tempo de renderização de toda a cena;
- *PARES*: quantidade de pares de objetos não descartados pelo *Sweep and Prune* para cada volume envolvente;

A seguir são apresentadas três imagens da simulação, sendo a Figura 5.3 exibindo o início da simulação, a Figura 5.4 aproximadamente o meio da simulação e a Figura 5.5 o final da simulação.

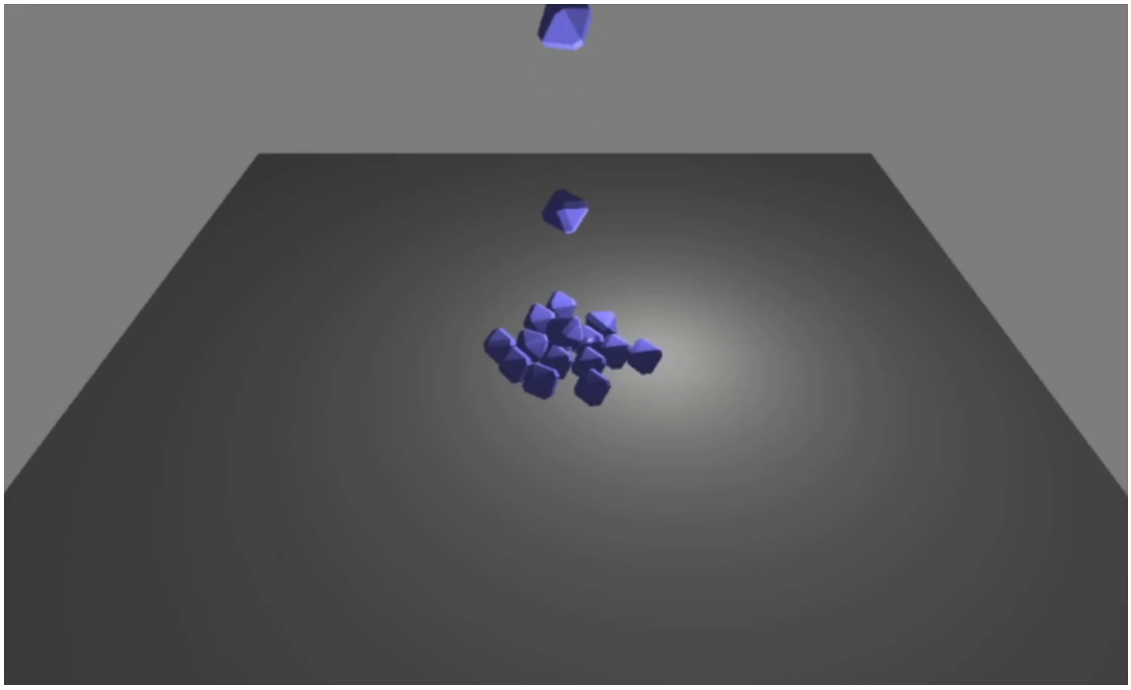

Figura 5.3: Início da simulação.

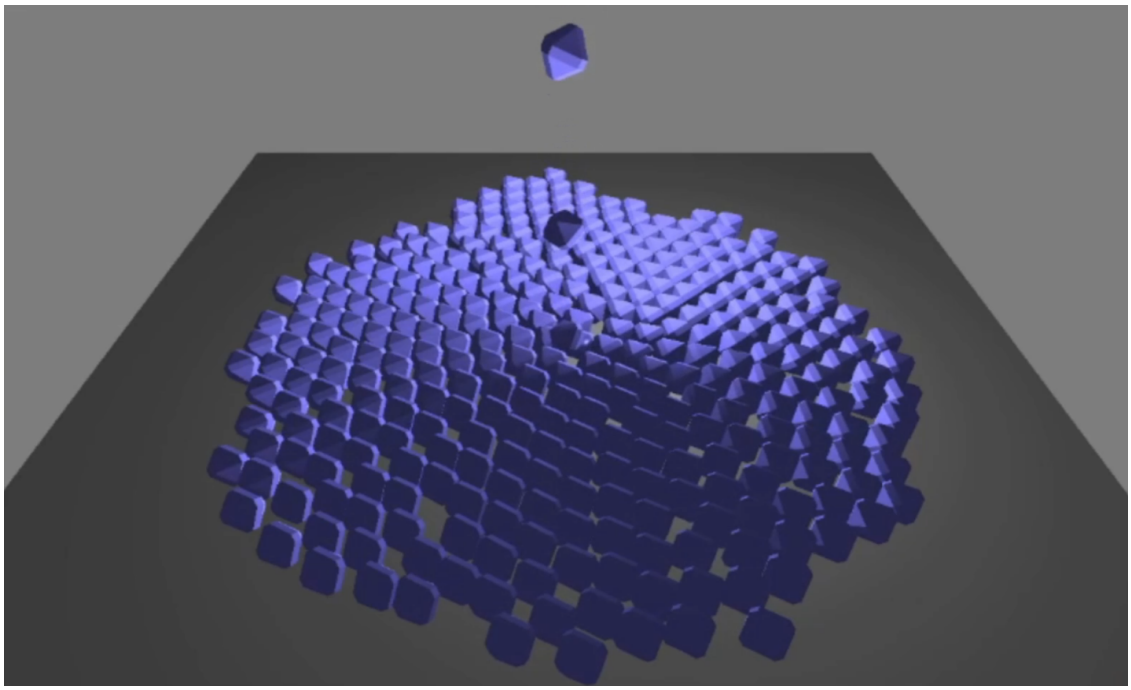

Figura 5.4: Meio da simulação.

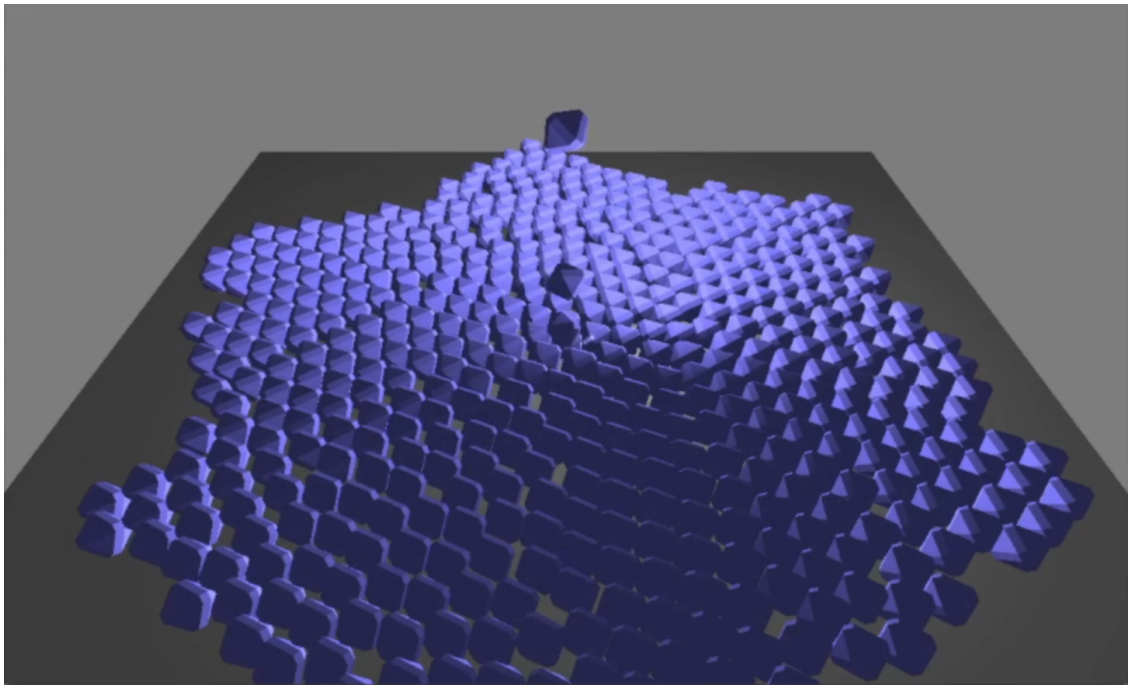

Figura 5.5: Final da simulação.

5.1 Análise dos pares não descartados

Uma medida importante em nosso experimento é a quantidade de pares em potencial colisão, ou seja, os pares que não foram descartados pelo algoritmo *Sweep and Prune*. A

quantidade de pares em possível colisão está intimamente relacionada com a geometria do objeto, e por esta razão foram analisadas duas simulações com geometrias diferentes, uma simulação com objetos contendo a geometria original (sem escala) e outra simulação com objetos escalados. Nas Figuras 5.6 e 5.7 é apresentado a quantidade de pares em potencial colisão para cada tipo de volume envolvente por quadro.

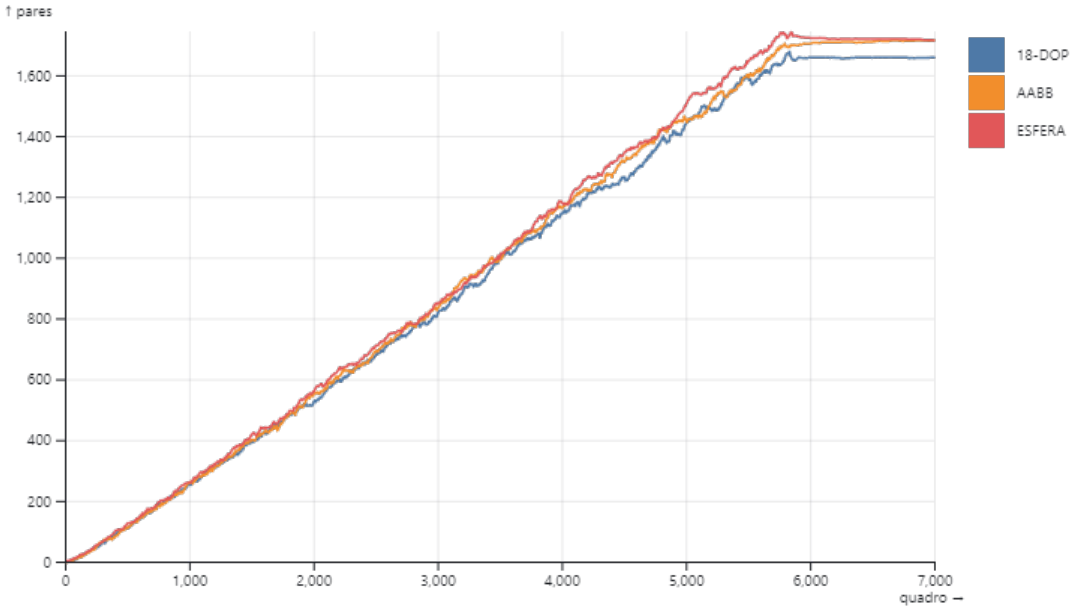

Figura 5.6: Número de pares de objetos em potencial colisão não descartados pelo *Sweep and Prune* por quadro para cada tipo de volume, considerando objetos sem escala.

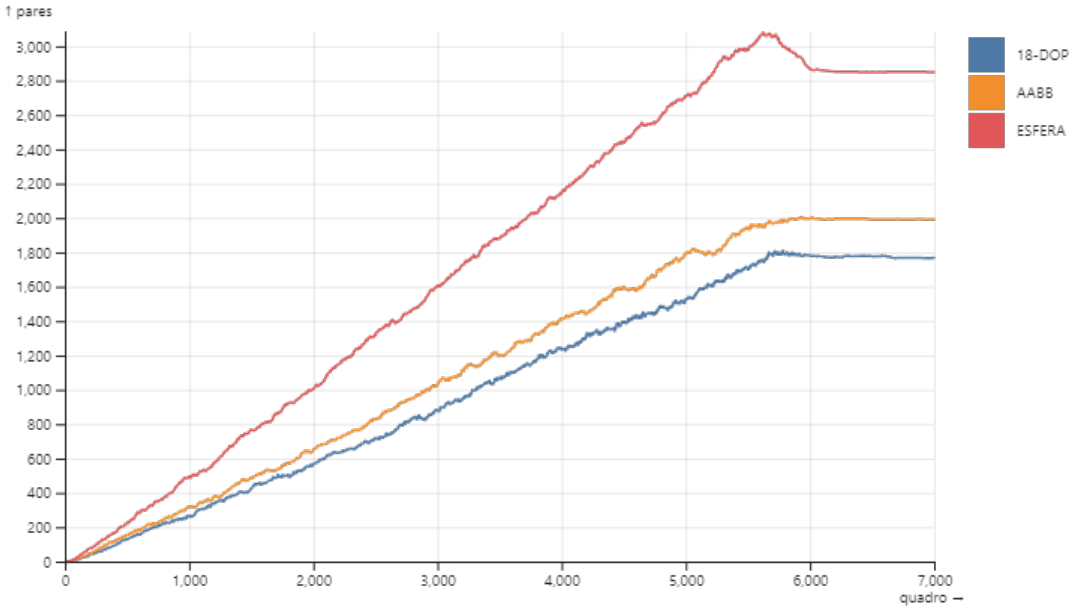

Figura 5.7: Número de pares de objetos em potencial colisão não descartados pelo *Sweep and Prune* por quadro para cada tipo de volume, considerando objetos com escala.

Observa-se que na simulação contendo os objetos com a geometria sem escala (Fi-

gura 5.6) uma quantidade similar de pares em potencial colisão é registrada para todos os tipos de volume envolvente. Este fato ocorre devido à geometria dos objetos ser quase globular. Portanto, tanto esferas quanto *AABB*s e *18-DOP*s envolvem relativamente bem a geometria. Já na Figura 5.7, a qual exibe os pares em potencial colisão para a simulação contendo objetos com a geometria escalada, há uma discrepância na quantidade de pares para o volume envolvente do tipo *18-DOP* quando comparado com os demais volumes envolventes.

A tabela 5.1 exibe a quantidade média de pares em potencial colisão por quadro, bem como a quantidade máxima de pares em colisão por quadro para todos os tipos de volume envolvente, medidas na simulação com objetos não escalados. Por sua vez, a tabela 5.2 mostra os dados análogos medidos para a simulação com objetos escalados.

	Qtd Média/Quadro	Qtd Máxima
18-DOP	**957**	**1.683**
AABB	978	1.717
Esfera	995	1.744

Tabela 5.1: Tabela contendo a quantidade média de pares em potencial colisão por quadro e a quantidade máxima encontrada em todos os quadros para cada tipo de volume envolvente, para a simulação com objetos sem escala.

	Qtd Média/Quadro	Qtd Máxima
18-DOP	**1.040**	**1.834**
AABB	1.152	2.012
Esfera	1.762	2.993

Tabela 5.2: Tabela contendo a quantidade média de pares em potencial colisão por quadro e a quantidade máxima encontrada em toda simulação para cada tipo de volume envolvente, para a simulação com objetos com escala.

Podemos observar, portanto, que na simulação contendo objetos escalados (Tabela 5.2) com volume do tipo *18-DOP* foi registrado em média 59% menos potenciais colisões que na simulação usando esferas e 9,72% menos potenciais colisões quando comparado com a simulação usando *AABBs*. Esta, por sua vez registrou 34,61% menos potenciais colisões em média que na simulação com esferas.

Entretanto, na simulação cujos objetos não estão escalados os dados se assemelham, sendo que os volumes do tipo *18-DOP* se destacam ligeiramente na quantidade média por quadro, registrando em média 957 potenciais colisões contra 978 da *AABB* e 995 da *Esfera*.

5.2 Análise do tempo por quadro

Nesta análise foi medido o tempo de execução de todos os processos da simulação por quadro, consistindo nos tempos de: *CONSTRUÇÃO VOLUMES*, *SWEEP AND PRUNE*, *GJK+EPA*, *SHAPE MATCHING* e *RENDERIZAÇÃO* para cada tipo de volume envolvente, com e sem escala, totalizando 6 gráficos, conforme Figuras 5.8, 5.9, 5.10, 5.11, 5.12 e 5.13.

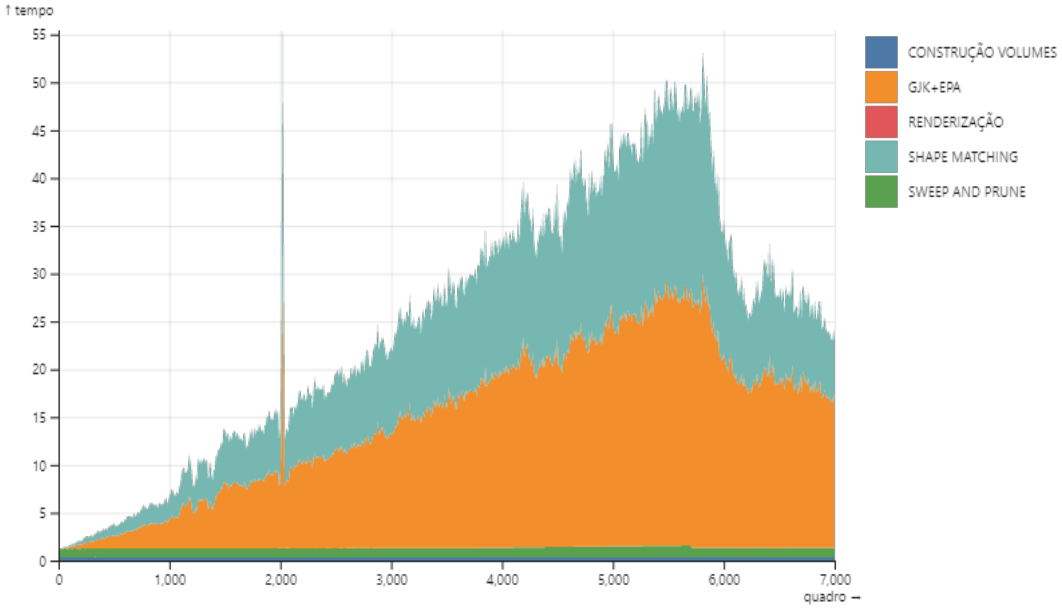

Figura 5.8: Tempo de execução em milissegundos para *18-DOP* com objetos sem escala

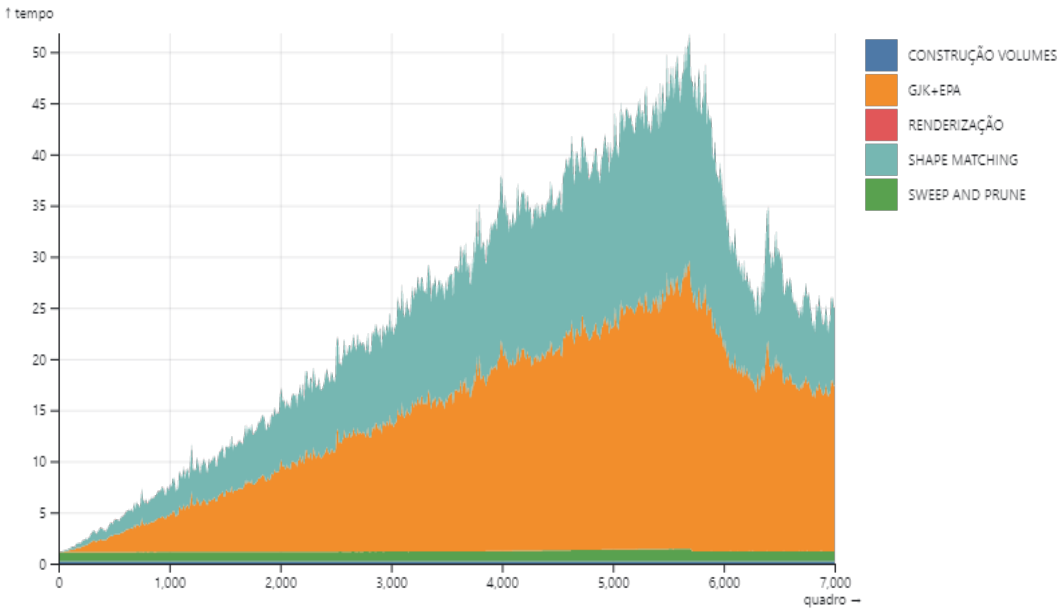

Figura 5.9: Tempo de execução em milissegundos para *AABB* com objetos sem escala

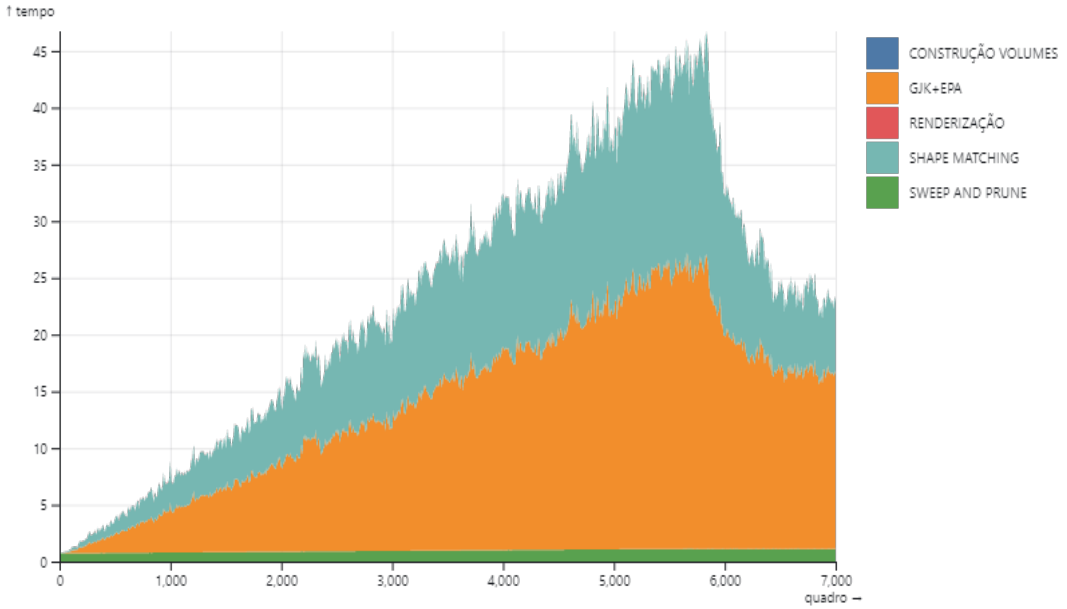

Figura 5.10: Tempo de execução em milissegundos para *Esferas* com objetos sem escala

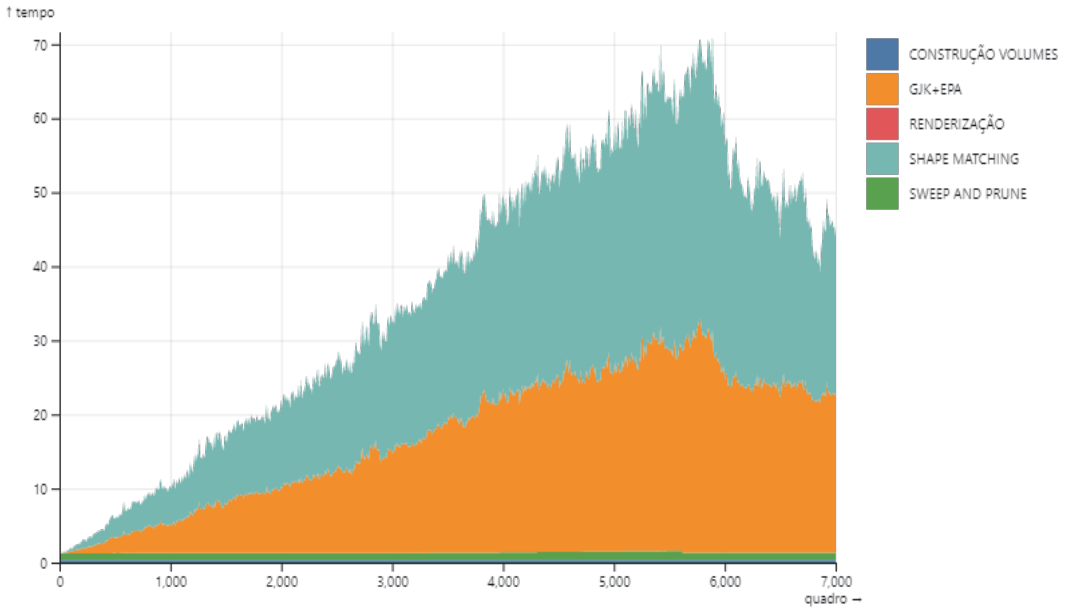

Figura 5.11: Tempo de execução em milissegundos para *18-DOP* com objetos com escala

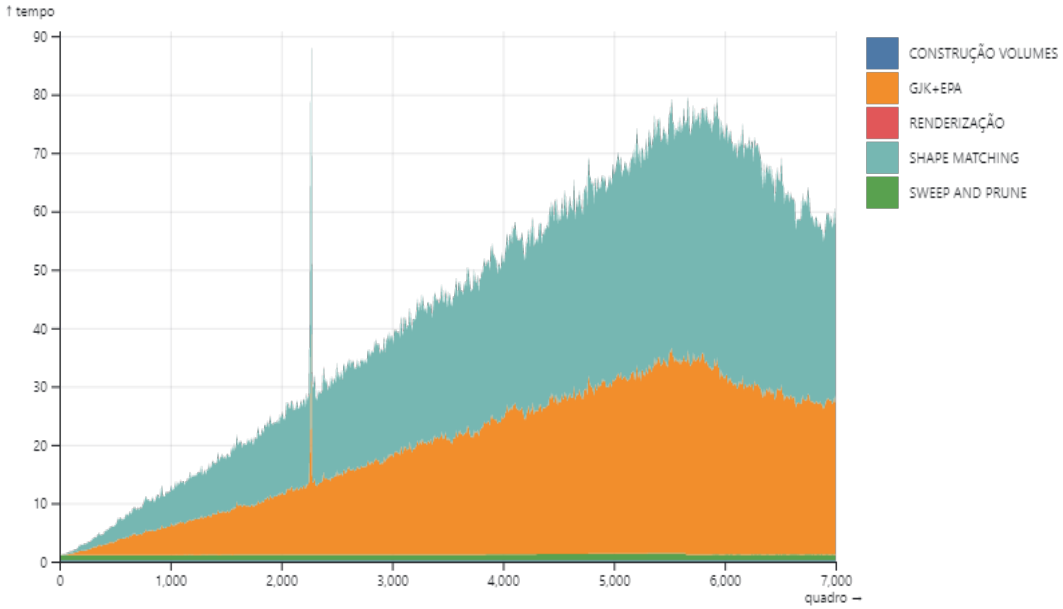

Figura 5.12: Tempo de execução em milissegundos para *AABB* com objetos com escala

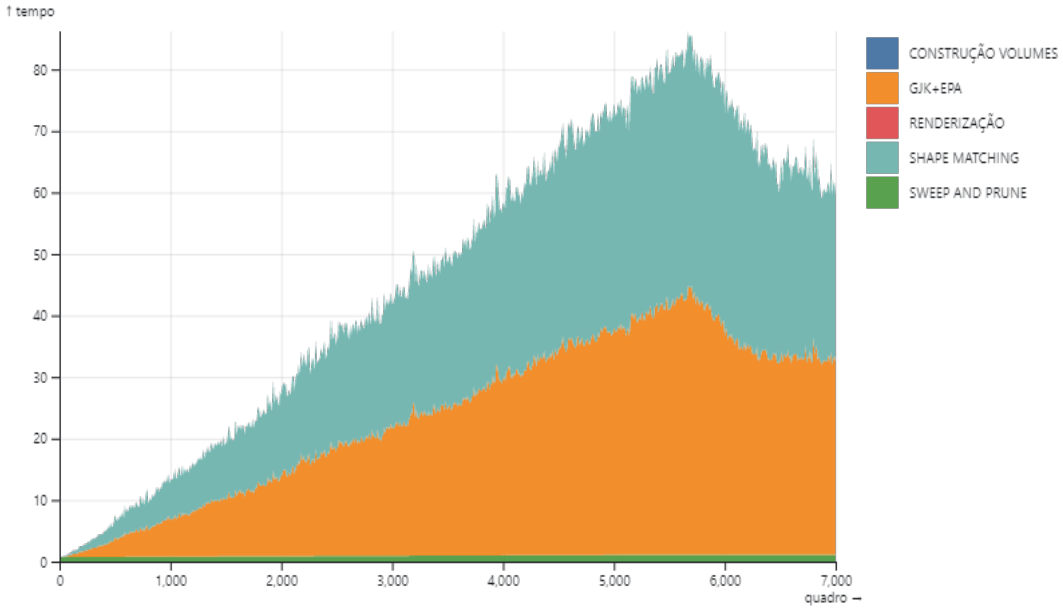

Figura 5.13: Tempo de execução em milissegundos para *Esferas* com objetos com escala

Observa-se que o tempo de construção dos volumes envolventes e de renderização são muito baixos em relação ao tempo das demais etapas, não impactando significativamente o tempo por quadro. A etapa do *Sweep and Prune* toma um tempo aproximadamente constante de 1.5 milissegundo por quadro, sendo representado por uma fina faixa verde na base do gráfico. Claramente, as etapas *GJK+EPA* e *SHAPE MATCHING* correspondem ao maior dispêndio de tempo por quadro, que é cada vez mais significativa à medida que cresce o número de pares em colisão. Próximo ao final da simulação, aproximadamente no quadro 5.500, há uma leve queda no tempo dos procedimentos *GJK+EPA* e *SHAPE*

MATCHING, visto que a partir desse momento os objetos já terminaram de cair e estão todos em contato com o chão e o eixo selecionado pela *PCA* é alterado para algum eixo no plano XZ. Posteriormente, a partir do quadro 6.000, é observada uma maior estabilidade no tempo dos procedimentos.

Observa-se também que o *GJK+EPA* ainda supera o tempo de *SHAPE MATCHING*, dado que o *GJK* é sempre executado em todos os pares em potencial colisão, sendo um filtro para a execução do *SHAPE MATCHING*, ou seja, o *SHAPE MATCHING* só é executado para os pares em colisão de fato. Logo, pode-se afirmar que o *GJK* é influenciado diretamente pelos pares falsos-positivos encaminhados pelo *Sweep and Prune*, enquanto o *SHAPE MATCHING* não.

5.3 Análise do tempo por volume envolvente

Nesta seção é apresentada uma análise do tempo total acumulado em milissegundos de todas as etapas da simulação (*CONSTRUÇÃO VOLUMES*, *SWEEP AND PRUNE*, *GJK+EPA*, *Shape Matching* e *RENDERIZAÇÃO*) por quadro, conforme Figuras 5.14 e 5.15.

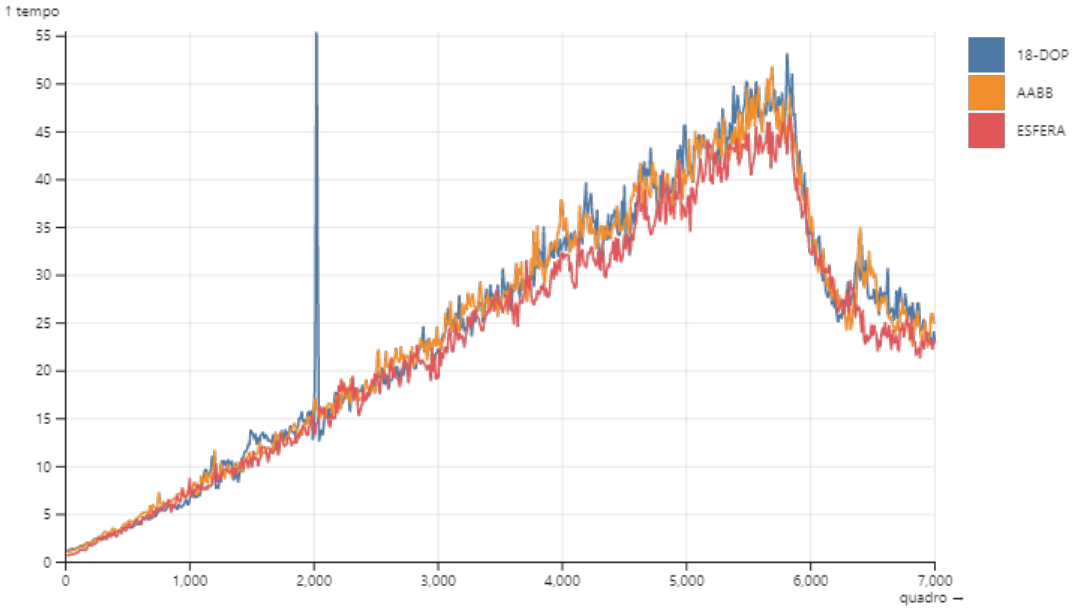

Figura 5.14: Tempo total de execução em milissegundos por quadro para os três tipos de volumes envolventes com objetos sem escala.

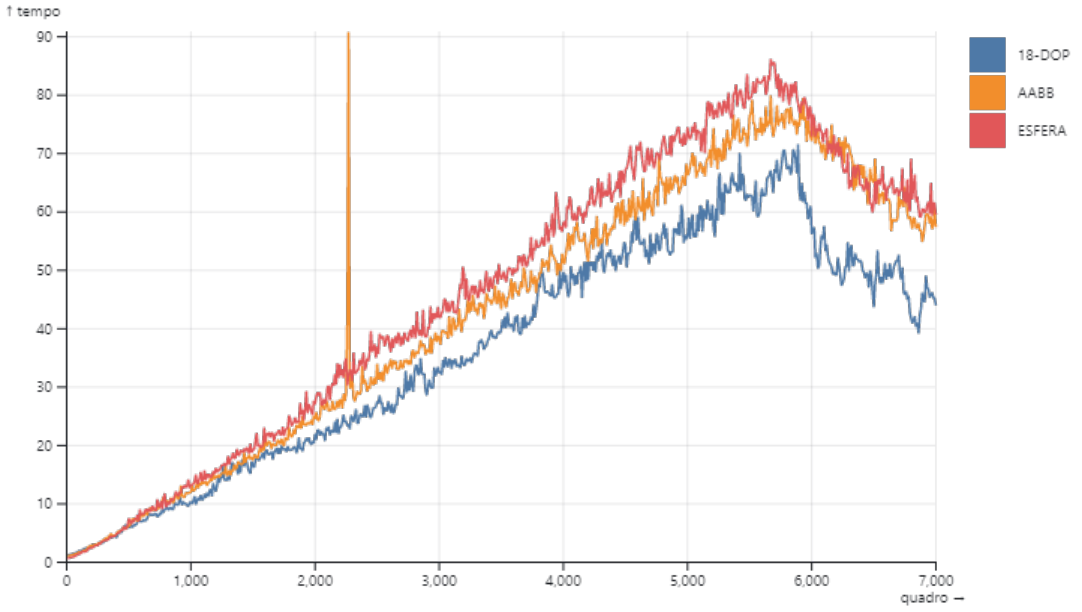

Figura 5.15: Tempo total de execução em milissegundos por quadro para os três tipos de volumes envolventes com objetos com escala.

Observa-se que quando a simulação é executada com objetos sem escala (Figura 5.14), a Esfera é ligeiramente mais eficiente que os demais volumes, sendo o volume do tipo *18-DOP* o menos eficiente na maioria dos quadros. Já com os objetos escalados (Figura 5.15), a performance obtida com os volumes do tipo *18-DOP* é consideravelmente superior à performance obtida com os demais volumes envolventes. É exibido na Tabela 5.3 o tempo médio em milissegundos por quadro para cada tipo de volume envolvente para simulações de objetos com e sem escala.

	Sem Escala	Com Escala
18-DOP	26,24ms	**38,22ms**
AABB	26,13ms	44,12ms
Esfera	**24,98ms**	46,71ms

Tabela 5.3: Tabela contendo o tempo médio em milissegundos por quadro em cada simulação.

Portanto, para a simulação contendo objetos com a geometria original, sem escala, o volume do tipo esfera resultou num desempenho superior em aproximadamente 5% comparado aos demais volumes envolventes. No cenário contendo os objetos escalados, o volume do tipo *18-DOP* obteve desempenho superior em aproximadamente 22,21% comparado com ao tipo Esfera, e 15,44% comparado ao tipo *AABB*. Por sua vez, o volume do tipo *AABB* registrou um desempenho superior em 5,87% comparado com o tipo Esfera.

Essa diferença observada no tempo médio por quadro para simulações com objetos escalados se dá pelo fato de que, quanto menos seletivo for o volume envolvente, mais pares falso-positivo de possíveis colisões são encaminhados para etapa posterior (*Narrow*

Phase).

Para simulações com objetos escalados, os volumes do tipo *AABB* e principalmente os do tipo *Esfera* resultam em muitos pares falso-positivos serem encaminhados para a próxima fase. Nas Figuras 5.16 e 5.17 são exibidos os gráficos de pares falso-positivos encaminhados para a etapa *GJK+EPA* para cada tipo de volume envolvente, para simulações contendo objetos com e sem escala.

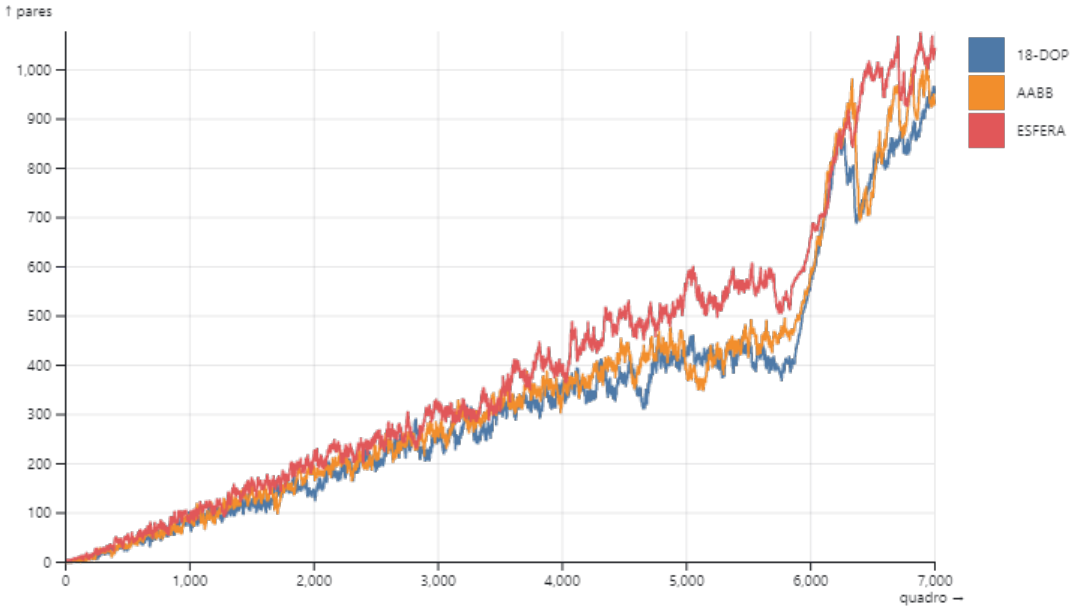

Figura 5.16: Quantidade de pares falso-positivo por quadro para cada volume envolvente com objetos sem escala.

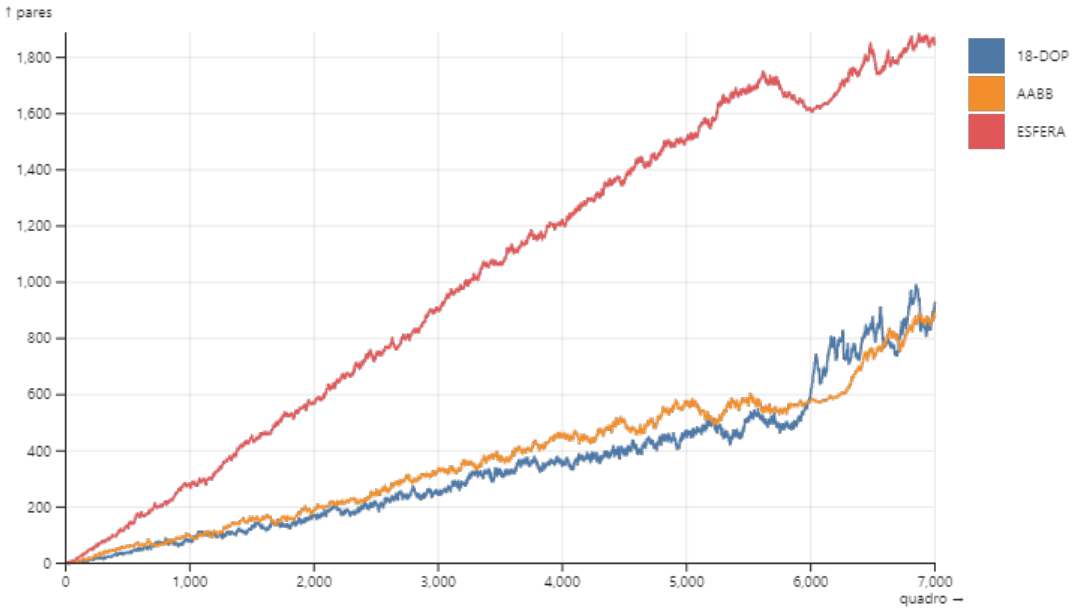

Figura 5.17: Quantidade de pares falso-positivo por quadro para cada volume envolvente com objetos escalados.

Observa-se na Figura 5.16 que o número de pares falso-positivos é similar para todos os volumes envolventes, sendo o tipo esfera ligeiramente superior. Enquanto na Figura 5.17 os volumes do tipo *AABB* e *18-DOP* permanecem com uma quantidade similar de pares, sendo que o volume do tipo *AABB* apresenta uma quantidade de pares falso-positivos ligeiramente superior comparado com o volume do tipo *18-DOP*, enquanto o volume do tipo Esfera se destaca com grande quantidade de pares falso-positivos sendo encaminhados para a *Narrow Phase*.

A seguir é ilustrado o impacto da grande quantidade de pares falso-positivos encaminhados para a *Narrow Phase*. Nas Figuras 5.18 e 5.19 é exibido o tempo em milissegundos para o procedimento *GJK+EPA* por quadro.

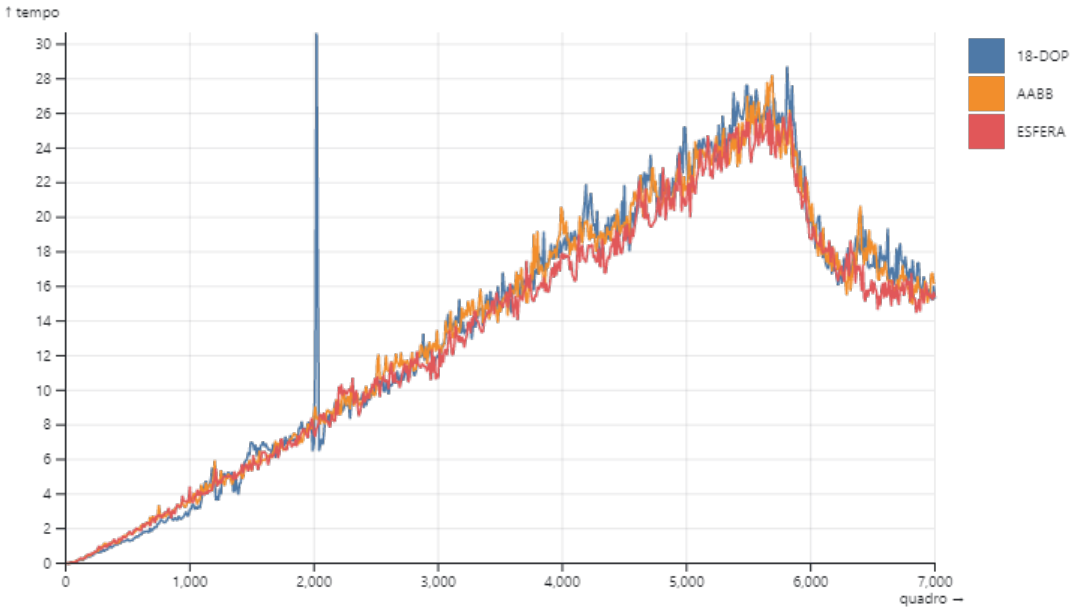

Figura 5.18: Tempo de execução em milissegundos do *GJK+EPA* de objetos sem escala

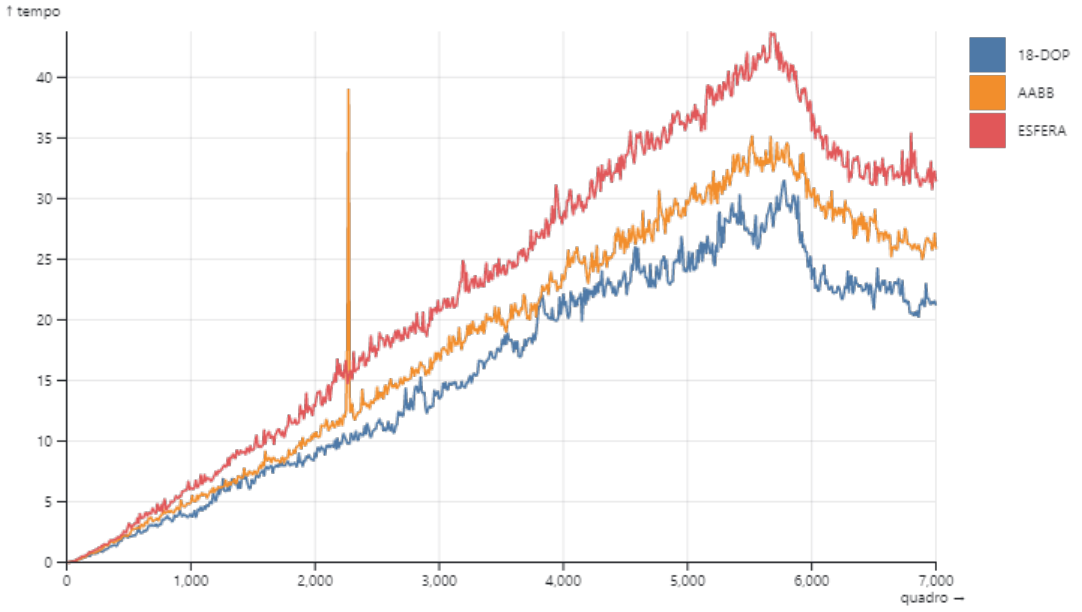

Figura 5.19: Tempo de execução em milissegundos do *GJK+EPA* de objetos com escala

Como previsto, observa-se na Figura 5.18 que os dados são similares para todos os tipos de volume envolvente. Entretanto, na simulação contendo objetos com escala, o impacto dos pares falso-positivos pode ser observado no tempo de execução do *GJK+EPA*.

5.4 Análise da construção dos volumes envolventes

Na Figura 5.20 é exibido um gráfico contendo o tempo de construção de cada tipo de volume envolvente ao longo de toda simulação. Observa-se que o maior tempo é tomado pela construção dos volumes envolventes do tipo *18-DOP* com uma constante de aproximadamente 0.34 milissegundos. Isto se justifica dado que, para todos os vértices da malha, faz-se necessário encontrar o vértice mais distante nas nove orientações do *18-DOP*. Para o caso da construção das *AABB*s, apenas três orientações são necessárias, portanto manteve um tempo consideravelmente inferior com 0.21 milissegundos. Para a esfera, é suficiente encontrar o vértice mais distante do centro da malha para que seja possível encontrar o raio da esfera, sendo esta consideravelmente mais rápida que os demais, atingindo a marca de 0.02 milissegundos. Destaca-se também que, para o volume do tipo esfera, não há necessidade de computar o volume envolvente em cada quadro, sendo suficiente computar o volume da esfera para um único objeto, visto que todos os objetos contém a malha idêntica ao demais objetos, além dos objetos serem simétricos em todos os eixos. Portanto, o volume do tipo *18-DOP* é 11 vezes mais lento que as esferas e 61% mais demorado que as *AABB*s. Já o tempo de construção das *AABB*s é 7 vezes maior comparado às esferas. Embora haja essa discrepância no tempo de construção dos volumes envolventes, entendemos que esse tempo pode ser reduzido para os volumes do

tipo *18-DOP* e *AABB*, conforme descrito no capítulo 6.

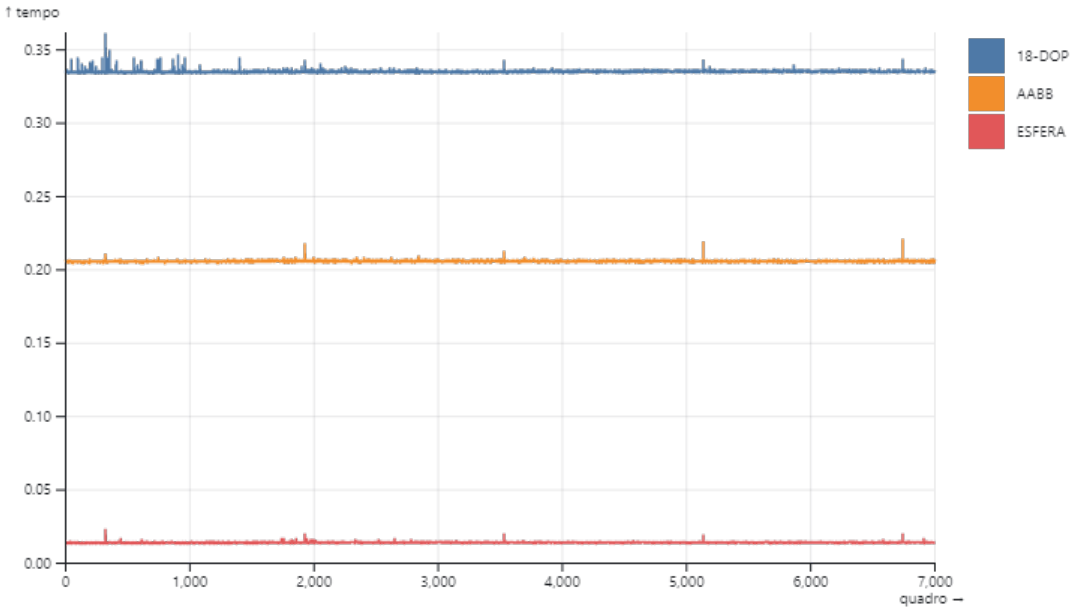

Figura 5.20: Tempo de construção em milissegundos de cada volume envolvente.

5.5 Análise do Sweep and Prune e *PCA*

Foi realizado também um levantamento acerca do tempo de execução do algoritmo *Sweep and Prune* por quadro, para cada tipo de volume envolvente abordado no experimento, considerando objetos com e sem escala. O tempo avaliado é um acumulado das três etapas do *Sweep and Prune* (*PROJEÇÃO EIXO*, *RADIX SORTING*, *PODA*). Observa-se na Figura 5.21 que o tempo total do *Sweep and Prune* é similar para os tipos de volume envolvente *AABB* e *18-DOP*, entretanto menor para as Esferas em quase todos os quadros. Porém, após o reprocessamento da *PCA*, quando todos os objetos atingem o chão, aproximadamente no quadro 5.500, os tempos dos volumes do tipo *18-DOP* e *AABB* passam a ser menores em aproximadamente 20%, tornando-os menores que o tempo registrado pelas esferas. Posteriormente ao quadro 5.500, todos os tipos volume envolvente passam ter um tempo constante. A Figura 5.22 apresenta o mesmo gráfico, porém utilizando objetos escalados e se observa que não há diferença significativa quando comparado ao gráfico com objetos sem escala.

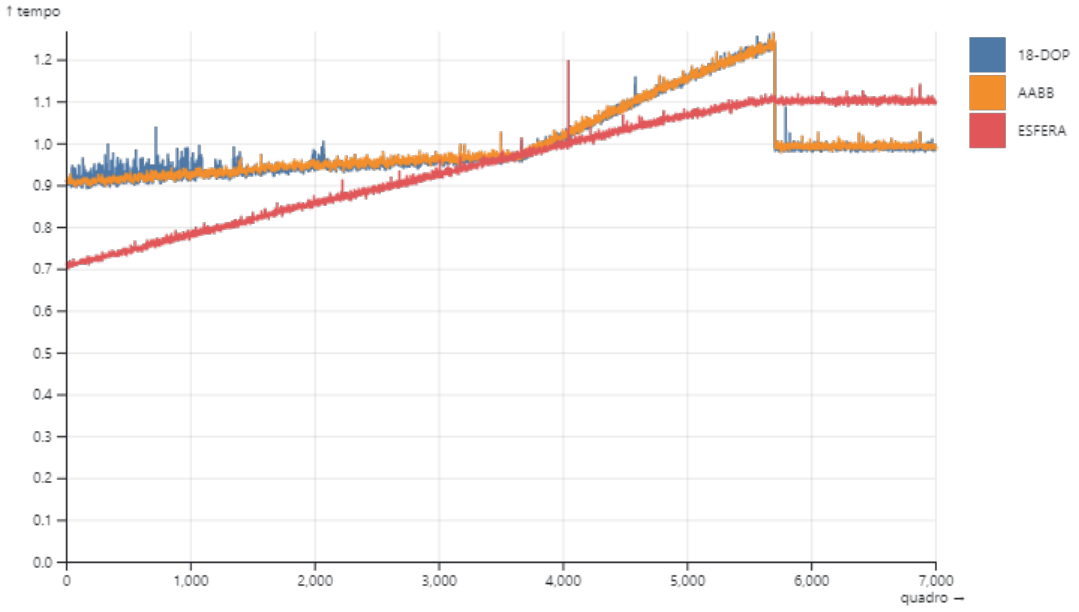

Figura 5.21: Tempo de execução em milissegundos do algoritmo *Sweep and Prune* para objetos sem escala em y.

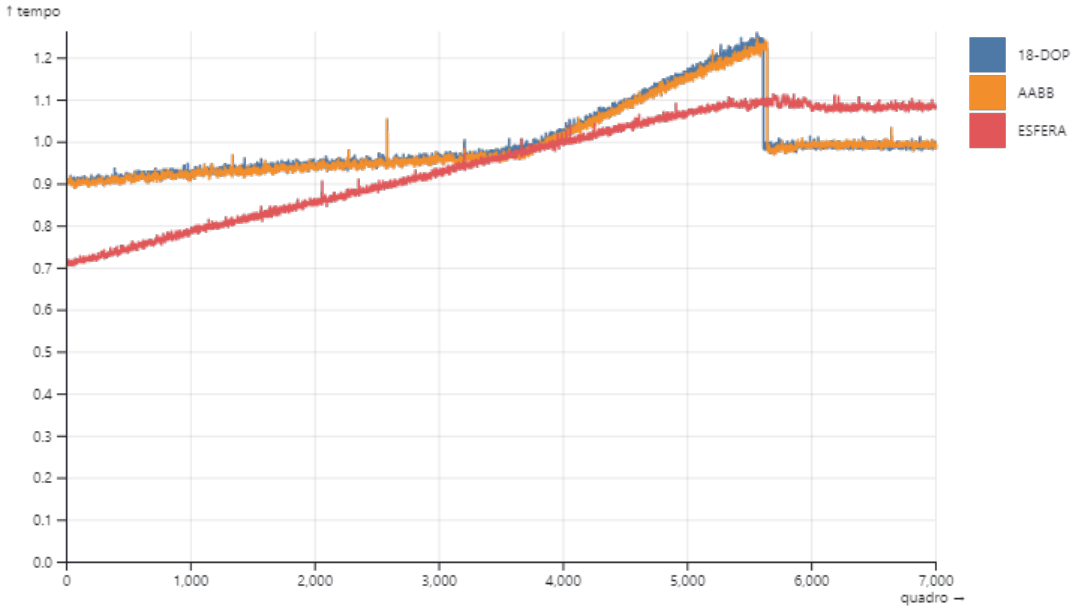

Figura 5.22: Tempo de execução em milissegundos do algoritmo *Sweep and Prune*, exibindo a média a cada 10 quadros para objetos com escala em y.

Na Figura 5.23 é analisado o tempo de cada fase do *Sweep and Prune* (*PROJEÇÃO EIXO*, *RADIX SORTING* e *PODA*).

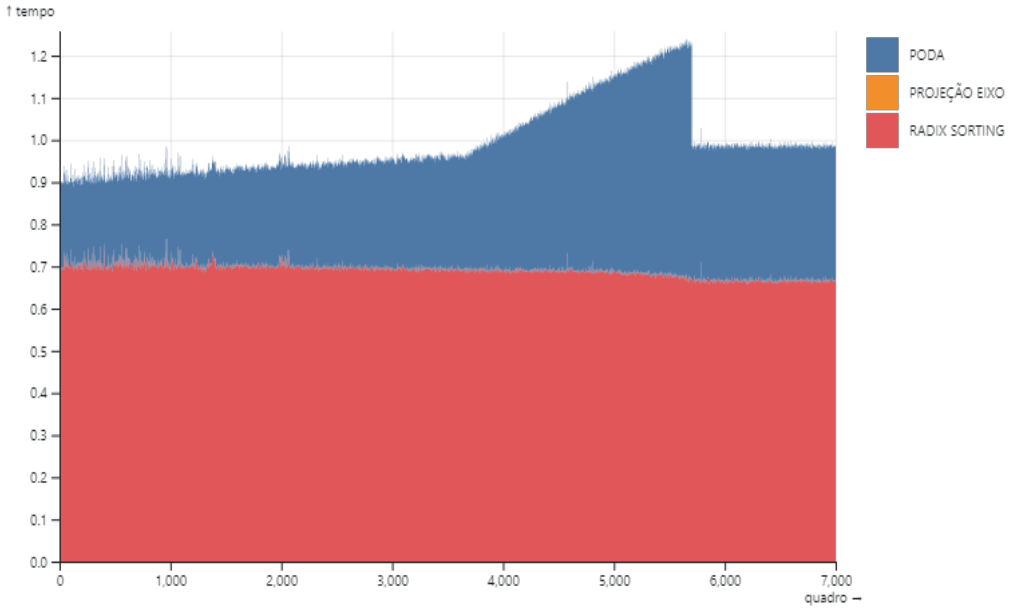

Figura 5.23: Tempo de execução em milissegundos das fases do algoritmo *Sweep and Prune*

Observa-se que há um ganho significativo no processo de *PODA* quando o eixo selecionado pela *PCA* é alterado aproximadamente no quadro 5.500, criando um acréscimo de aproximadamente 23% no desempenho. Constata-se também que o tempo tomado pela ordenação é predominante em relação às demais etapas do *Sweep and Prune*, sendo a etapa de projeção dos limites mínimos no eixo desprezível, sendo exibida em laranja na base do gráfico.

Capítulo 6

Conclusão

Diante do exposto, conclui-se que em geral, os volumes do tipo *18-DOP* são competitivos com *AABB*s e esferas no pior caso e, em cenários com objetos não globulares, francamente preferíveis, já que os *18-DOPs* apresentaram desempenho 61% superior com objetos escalados. Com objetos globulares, o volume Esfera se destaca ligeiramente.

Observamos que os ganhos referentes à escolha do volume envolvente se devem de forma quase exclusiva à redução do número de falsos positivos e ao menor esforço computacional decorrente na *narrow phase* que, nos nossos experimentos, é implementada com o algoritmo GJK. Concebivelmente, se dispusermos de um algoritmo de detecção exata de colisão mais eficiente, esses ganhos podem ser reduzidos ou mesmo eliminados.

Uma outra ideia seria não utilizar uma *narrow phase*, mas computar a resposta às colisões usando o resultado aproximado da *broad phase*. Por exemplo, para objetos globulares, pode-se usar os próprios volumes envolventes como suporte para a fase de resposta. No caso de esferas, por exemplo, os deslocamentos de vértices podem ser feitos por projeção num plano separador.

Nossos experimentos se limitou a utilizar 3 tipos de volumes envolventes: *k-DOP* e esferas, sendo $k = \{6, 18\}$. Outros tipos de volumes envolventes, como *OBBs* e outras variações de k, poderiam ser comparados em nossa simulação.

Além disto, novos experimentos contendo objetos cuja geometria contém quantidades de faces variadas seriam de grande interesse, pois a quantidade de faces impacta diretamente a performance do *GJK* e *EPA*, dado que esses algoritmos examinam as faces para identificar contatos.

Toda a *Broad Phase* é executada em *GPU*, o que leva a uma melhor performance, entretanto a *Narrow Phase* é executada em *CPU*. Novos experimentos com a execução da *Narrow Phase* em *GPU* seriam também de grande interesse, principalmente devido à abordagem usada no presente trabalho onde a quantidade de colisões é significativa e a taxa de quadros por segundos é degradada devido ao restante da implementação (*Narrow Phase* e dinâmica de corpos) ser em *CPU*. Uma possível alternativa é utilizar um resolvedor de complementariedade linear em *GPU* [63] para resolver os contatos com

convergência global.

Outra possibilidade observada para suportar maior quantidade de objetos mantendo taxas iterativas seria criar uma estrutura de decomposição espacial, como uma *Octree* por exemplo, onde os objetos em cada partição poderiam ser processados de forma independente, permitido a exploração de paralelismo na execução do algoritmo *Sweep and Prune* seguido de *GJK+EPA*. Dependendo da distância do observador a cada partição, o sistema de detecção de colisão poderia ser mais relaxado, visto que os detalhes da física não seriam perceptíveis, ou seja, poderia haver menos iterações de *GJK+EPA*.

No aspecto de disposição dos objetos em cena, onde há alto grau de colisões, nos quais os objetos estão menos dispersos, *18-DOP* é uma alternativa mais eficiente devido à menor taxa de pares falso-positivos encaminhados a próxima etapa, tornando-se um filtro mais seletivo de pares em potencial colisão. Enquanto em ambientes com objetos relativamente dispersos, o volume do tipo Esfera se destaca pelo fato de atingir grau de eliminação de pares em potencial colisão similar ao *18-DOP*, sendo assim mais eficiente devido ao alto desempenho na construção dos volumes envolventes e verificação de colisão dos pares de volumes envolventes.

Nossa simulação realizou a dinâmica de objetos utilizando técnica *Meshless Deformations Based on Shape Matching* para corpos deformáveis. Experimentos comparativo com outras técnicas de dinâmicas de corpos, como dinâmica baseada em impulso ou dinâmica utilizando discretização em esferas [7], podem ser realizados para comparar o comportamento das colisões, visto que a técnica empregada em nosso trabalho mantém alta coesão entre os objetos, isto é, a separação dos objetos em colisão utilizando a técnica supracitada é feita de forma que o contato entre os objetos seja mantido mínimo na maioria das vezes, acarretando em grande volume de colisões.

Nosso trabalho utilizou apenas detecção de colisão discreta. Com a aplicação da detecção de colisão contínua sendo empregada na *Broad Phase* os objetos poderiam ser escalados no sentido do deslocamento, possivelmente gerando volumes envolventes agudos/escalados nessa direção. Portanto entendemos que, um volume envolvente que engloba melhor geometrias desse tipo, como o *18-DOP*, obterão melhor desempenho comparado com os demais volumes, como *AABB* e Esferas.

No quesito de construção de volumes envolventes, os tipos *18-DOP* e *AABB* são consideravelmente mais lentos comparado com esferas, o que é compensado com o alto grau de eliminação de pares falso-positivos de objetos que são encaminhados para a próxima etapa de simulação.

A construção de volumes envolventes é realizada em *GPU*, sendo cada *thread* responsável pela construção do volume envolvente de um único objeto, ou seja, para a construção de um volume *18-DOP* a *thread* precisa encontrar os limites mínimo e máximo para os 9 eixos, sendo para *AABB*s 3 eixos e para esferas apenas um. Entendemos que a construção de volumes envolventes sendo processada em uma *thread* para cada eixo, ao

invés de para cada objeto, aumentaria o grau de paralelismo, podendo levar a desempenho mais competitivo na construção dos volumes envolventes do tipo *18-DOP* e *AABB*.

Com a alta dispersão dos objetos em cena, também seria interessante avaliar o comportamento da taxa de atualização da técnica da *PCA*, a qual foi fixa em nosso trabalho, a saber, uma atualização a cada 30 quadros. Em nossas simulações, a técnica de *PCA* ganha relevância apenas quando todos os objetos terminam de cair e o eixo de projeção do *Sweep and Prune* é alterado para algum eixo no plano XZ, gerando uma melhora significativa no desempenho do *Sweep and Prune*.

Conforme observado na Figura 5.21, a técnica de *PCA* alterou o eixo apenas no quadro 5.500 quando todos os objetos estavam no plano e se manteve no eixo (0, 1 0) durante todo este período. Caso outra técnica ou uma melhoria na *PCA* identificasse antecipadamente que outro eixo fosse mais indicado, seria possível diminuir o tempo do *Sweep and Prune*, quando metade ou mais dos objetos já se encontram no plano.

Referências Bibliográficas

[1] CAPANNINI, G. "Adaptive Collision Culling for Massive Simulations by a Parallel and Context-Aware Sweep and Prune Algorithm", *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, v. 27, n. 7, jul. 2018.

[2] HARADA, T. "Real-time Collision Culling of a Million Bodies on Graphics Processing Units", *ACM Transactions on Graphics*, v. 29, n. 6, dez. 2010.

[3] HARADA, T., HOWES, L. "Introduction to GPU Radix Sort". In: *Heterogeneous Computing with OpenCL*, 2011.

[4] OLIVEIRA, R., ESPERANÇA. "Exploiting Space and Time Coherence in Grid-based Sorting", *XXVI Conference on Graphics, Patterns and Images*, 2013.

[5] TERDIMAN, P. "Sweep and Prune", -, sep 2007.

[6] SERPA, Y., RODRIGUES, M. "Parallelizing Broad Phase Collision Detection for Animation in Games: A Performance Comparison of CPU and GPU Algorithms", *SBC - Proceedings of the SBGames 2014*, v. 29, n. 6, nov. 2014. Disponível em: <https://ieeexplore.ieee.org/abstract/document/7000035>.

[7] HARADA, T. "Real-Time Rigid Body Simulation on GPUs". In: *GPU Gem*, 3 ed., cap. 29, -, Addison-Wesley Professional, aug 2007.

[8] ERICSON, C., BYWALEC, B. *Real-Time Collision Detection*. 1 ed. Los Angeles, Elsevier, 2004.

[9] KNUTH, D. *"The Art of Computer Programming"*. 3 ed. , Addison-Wesley Professional, abr. 1998.

[10] KESSENICH, J., SELLERS, G., SHREINER, D. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*. 9 ed. , Addison-Wesley Professional, jul. 2016.

[11] KAELI, D., MISTRY, P., SCHAA, D., et al. *Heterogeneous Computing with OpenCL 2.0*. 1 ed. , Morgan Kaufmann, jun. 2015.

[12] MUNSHI, A., GASTER, B., MATTSON, T., et al. *OpenCL Programming Guide*. 1 ed. , Addison-Wesley Professional, jul. 2011.

[13] JOLLIFFE, I. "Principal Component Analysis". In: *International Encyclopedia of Statistical Science*, Berlin, Heidelberg, Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-04898-2.

[14] JOLLIFFE, I. *Principal Component Analysis, Second Edition*. Springer, 2002.

[15] WICHE, R. "How to create awesome accelerators: The Surface Area Heuristic". abr. 2018. Disponível em: <https://medium.com/@bromanz/how-to-create-awesome-accelerators-the-surface-area-heuristic-e [Online; accessed 29-September-2019].

[16] YANG, J., LI, R., XIAO, Y., et al. "3D reconstruction from non-uniform point clouds via local hierarchical clustering". In: *3D reconstruction from non-uniform point clouds via local hierarchical clustering*, 07 2017. doi: 10.1117/12.2281528.

[17] LIRA DOS SANTOS, A., TEIXEIRA, J., FARIAS, T., et al. "Understanding the Efficiency of kD-tree Ray-Traversal Techniques over a GPGPU Architecture", *International Journal of Parallel Programming*, v. 40, 06 2011. doi: 10.1007/s10766-011-0186-1.

[18] CORPORATION, N. "PhysX". 2019. Disponível em: <https://www.geforce.com/hardware/technology/physx>. [Online; accessed 04-October-2019].

[19] CORPORATION, M. "Havok". 2019. Disponível em: <https://www.havok.com/>. [Online; accessed 04-October-2019].

[20] AVRIL, Q., GOURANTON, V., ARNALDI, B. "A Broad Phase Collision Detection Algorithm Adapted to Multi-cores Architectures", *VRIC 2010 Proceedings*, 04 2010.

[21] GROUP, K. "OpenCL Overview". 2019. Disponível em: <https://www.khronos.org/opencl/>. [Online; accessed 06-October-2019].

[22] KAELI, D., MISTRY, P., SCHAA, D., et al. *Heterogeneous Computing with OpenCL 2.0: Third Edition*. 3 ed. , Morgan Kaufmann, 01 2015.

[23] KIRK, D., HWU, W.-M. *Programming Massively Parallel Processors: A Hands-on Approach*. 3 ed. , Morgan Kaufmann, 12 2016.

[24] MUNSHI, A., GASTER, B., MATTSON, T. G., et al. *OpenCL Programming Guide*. 1 ed. , Addison-Wesley Professional, 07 2011.

[25] OWENS, J., LUEBKE, D., GOVINDARAJU, N., et al. "A Survey of General-Purpose Computation on Graphics Hardware", *Computer Graphics Forum*, v. 26, 03 2007. doi: 10.1111/j.1467-8659.2007.01012.x.

[26] QI, B., PANG, M. "An enhanced sweep and prune algorithm for multi-body continuous collision detection", *The Visual Computer*, 06 2019. doi: 10.1007/s00371-019-01718-2.

[27] QI, B., PANG, M. "A Robust and Efficient Algorithm for Multi-body Continuous Collision Detection". In: *A Robust and Efficient Algorithm for Multi-body Continuous Collision Detection*, 10 2018. doi: 10.1109/CW.2018.00020.

[28] XING, Y.-S., LIU, P., XU, S. "Efficient collision detection based on AABB trees and sort algorithm", *2010 8th IEEE International Conference on Control and Automation, ICCA 2010*, pp. 328–332, 06 2010. doi: 10.1109/ICCA.2010.5524093.

[29] ZHIGANG, F., JIANXUN, J., JIE, X., et al. "Efficient collision detection using bounding volume hierarchies of OBB-AABBs and its application", *2010 International Conference on Computer Design and Applications, ICCDA 2010*, v. 5, 06 2010. doi: 10.1109/ICCDA.2010.5541315.

[30] CHANG, J.-W., WANG, W., KIM, M.-S. "Efficient collision detection using a dual OBB-sphere bounding volume hierarchy", *Computer-Aided Design*, v. 42, pp. 50–57, 01 2010. doi: 10.1016/j.cad.2009.04.010.

[31] ZHANG, R., LIU, X., WEI, J. "Collision detection Based on OBB Simplified modeling", *Journal of Physics: Conference Series*, v. 1213, pp. 042079, 06 2019. doi: 10.1088/1742-6596/1213/4/042079.

[32] KONG, D., LIU, Y., CUI, N. "Collosion Detection Research Based on Capsule Bounding Volume", *Journal of Computational Information Systems*, v. 10, pp. 7–2014, 04 2014.

[33] CANT, R., LANGENSIEPEN, C., FOSTER, L. "A Composite Convex Hull Collision Detector for the Open Dynamics Engine". In: *A Composite Convex Hull Collision Detector for the Open Dynamics Engine*, 05 2008. ISBN: 0-7695-3114-8.

[34] GREGORY, J. *Game Engine Architecture, Third Edition*. Elsevier, 07 2018. ISBN: 9781315267845. doi: 10.1201/9781315267845.

[35] BOURG, D. M., BYWALEC, B. *Physics for Game Developers*. O'Reilly, 04 2013. ISBN: 9781449392512.

[36] SAMET, H. *Foundations Of Multidimensional And Metric Data Structures*. Morgan Kaufmann, 01 2006.

[37] CORPORATION, N. "NVIDIA TURING GPU ARCHITECTURE". 2019. Disponível em: <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>. [Online; accessed 12-October-2019].

[38] TAL, D., RISHE, N., NAVATHE, S., et al. "On Parallel Architectures." v. 28, 01 1990.

[39] "Continuous Collision Detection". 2019. Disponível em: <https://forum.unity.com/threads/continuous-collision-detection.297909/>. [Online; accessed 03-December-2019].

[40] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. *Real-Time Rendering*. CRC Press, 01 2019. ISBN: 9781315365459. doi: 10.1201/9781315365459.

[41] "Hyperplane separation theorem". 2019. Disponível em: <https://en.wikipedia.org/wiki/Hyperplane_separation_theorem>. [Online; accessed 05-December-2019].

[42] "The Separating Axis Theorem". 2015. Disponível em: <https://darrensweeney.net/2015/09/12/the-seperating-axis-theorem/>. [Online; accessed 05-December-2019].

[43] HOUNSELL, M. "Detecção de Colisão para um Simulador de Robô Manipulador". 2007.

[44] "Bitonic sorter". 2019. Disponível em: <https://en.wikipedia.org/wiki/Bitonic_sorter>. [Online; accessed 05-December-2019].

[45] "Ray Tracing". 2019. Disponível em: <http://cseweb.ucsd.edu/~sht005/ray_tracing_info.html>. [Online; accessed 05-December-2019].

[46] "Bullet Real-Time Physics Simulation". 2020. Disponível em: <https://pybullet.org/>. [Online; accessed 14-January-2020].

[47] RAJA, C., BALASUBRAMANIAN, S., RAGHAVENDRA, P. "Heterogeneous Highly Parallel Implementation Of Matrix Exponentiation Using GPU", *International Journal of Distributed and Parallel systems*, v. 3, 04 2012. doi: 10.5121/ijdps.2012.3209.

[48] HOUSE, D., KEYSER, J. *Foundations of Physically Based Modeling and Animation*. Taylor & Francis, 2017. ISBN: 9781315373140. Disponível em: <https://books.google.com.br/books?id=ZpmhswEACAAJ>.

[49] "Principal Component Analysis". 2018. Disponível em: <https://www.learnopencv.com/principal-component-analysis/>. [Online; accessed 05-February-2020].

[50] LADNER, R. E., FISCHER, M. J. "Parallel Prefix Computation", *J. ACM*, v. 27, pp. 831–838, 1980.

[51] LIANG, C., LIU, X. "The Research of Collision Detection Algorithm Based on Separating axis Theorem". 2015.

[52] ARKHIPOV, D., WU, D., LI, K., et al. "Sorting with GPUs: A Survey", 09 2017.

[53] SATISH, N., HARRIS, M., GARLAND, M. "Designing efficient sorting algorithms for manycore GPUs". v. 23, 05 2009. doi: 10.1109/IPDPS.2009.5161005.

[54] DUSTIN CLINGMAN, KENDALL SHAWN, M. S. "Practical Java Game Programming", 06 2004.

[55] MÜLLER, M., HEIDELBERGER, B., TESCHNER, M., et al. "Meshless deformations based on shape matching", *ACM Trans. Graph.*, v. 24, pp. 471–478, 07 2005. doi: 10.1145/1186822.1073216.

[56] GILBERT, E. G., JOHNSON, D. W., KEERTHI, S. 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, pp. 193–203, 1988. doi: 10.1109/56.2083.

[57] CAMERON, S. "Enhancing GJK: computing minimum and penetration distances between convex polyhedra". In: *Proceedings of International Conference on Robotics and Automation*, v. 4, pp. 3112–3117 vol.4, 1997. doi: 10.1109/ROBOT.1997.606761.

[58] BERGEN, G. "Proximity queries and penetration depth computation on 3d game objects", 01 2001.

[59] “2D GJK and EPA algorithms”. 2021. Disponível em: <https://observablehq.com/@esperanc/2d-gjk-and-epa-algorithms>. [Online; accessed 06-February-2021].

[60] LIYANAGE, K. “ANALYZING THE IMPACT OF UNIT TESTING PRACTICES ON DELIVERY QUALITY IN AN AGILE ENVIRONMENT”, 11 2019. doi: 10.13140/RG.2.2.14559.41124.

[61] “Visualizing the GJK Collision detection algorithm”. 2016. Disponível em: <https://www.haroldserrano.com/blog/visualizing-the-gjk-collision-algorithm>. [Online; accessed 06-February-2021].

[62] ERLEBEN, K. “Velocity-based shock propagation for multibody dynamics animation”, *ACM Trans. Graph.*, v. 26, 06 2007. doi: 10.1145/1243980.1243986.

[63] NGUYEN, H. “Gpu gems 3”, p. 723–740, 01 2007.

Apêndice A

Operador de Orientação

Consiste em um operador que define se um dado ponto/localização está a direita, esquerda ou se são colineares, dada uma reta ou um plano. Esse operador se distingue consideravelmente do ambiente do caso 2D para 3D. No caso bidimensional, o cálculo é realizado obtendo-se os três pontos e incluindo em uma matriz coluna 3x3, sendo a primeira linha um vetor tridimensional unitário. Com a matriz, calcula-se o determinante da matriz.

$$D = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ ponto1.x & ponto2.x & ponto3.x \\ ponto1.y & ponto2.y & ponto3.y \end{bmatrix} = \{-1, 0, 1\}$$

Caso o determinante seja zero, ou algo próximo à zero (*Epsilon*), significa que os três pontos são colineares. Caso o determinante seja maior que zero o terceiro ponto está a esquerda em relação à reta formada pelos dois pontos anteriores. Caso contrário, o ponto está a direita.

No cenário tridimensional, não faz sentido se um ponto está a esquerda ou direta de uma reta, mas sim de um plano. Portanto, o cálculo pode ser simplificado utilizando a regra da mão direita, ou seja, os vértices das faces estão sempre ordenados em sentido anti-horário. Em uma renderização onde se considera o uso dessa regra, para calcular a orientação de um ponto, faz-se necessário apenas calcular a normal do plano em relação ao ponto e o produto escalar utilizando a normal. Se o vetor normal for positivo, logo o ponto está a esquerda (pois a face está virada para o ponto), caso contrário o ponto está a direita. Para calcular a normal da face do plano, faz-se necessário apenas calcular o produto vetorial de três pontos contidos no plano ordenados seguindo a regra da mão direita.

$$O = DOT(\vec{Normal}, (\vec{Target} - \vec{Point})) = \{+, -\}$$

Onde *DOT* é o operador produto escalar, *Target* é o ponto que se deseja conhecer a orientação e *Point* é um ponto no plano.

Apêndice B

Raiz Quadrada de Matriz

Encontrar a raiz quadrada de uma matriz não é um processo muito trivial, necessitando de métodos numéricos iterativos de decomposição polar, e por este motivo foi incluído nesse apêndice para que o leitor não perca o foco na leitura quando é apresentada a implementação de *shape matching*, no qual faz uso deste procedimento.

Nossa implementação para encontrar a raiz quadrada de uma matriz faz uso do método de rotações de Jacobi. Uma das razões foi a eficiência em comparação ao método de decomposição QR ou *Single Value Decomposition* (*SVD*). Além de sua estabilidade numérica e ampla aplicabilidade.

Para encontrar a raiz quadrada de uma matriz $\mathbf{M}$, tomamos proveito de que $\mathbf{M}$ é diagonalizável pelo fato de ser uma matriz simétrica, logo

$$\begin{aligned}\mathbf{M} &= \mathbf{A} * \mathbf{D} * \mathbf{A}^{-1} \\ &= \mathbf{A} * \mathbf{D}^{\frac{1}{2}} * \mathbf{D}^{\frac{1}{2}} * \mathbf{A}^{-1} \\ &= \mathbf{A} * \mathbf{D}^{\frac{1}{2}} * \mathbf{A}^{-1} * \mathbf{A} * \mathbf{D}^{\frac{1}{2}} * \mathbf{A}^{-1} \\ &= \sqrt{\mathbf{M}} * \sqrt{\mathbf{M}},\end{aligned}$$

portanto,

$$\sqrt{\mathbf{M}} = \mathbf{A} * \mathbf{D}^{\frac{1}{2}} * \mathbf{A}^{-1}, \tag{B.1}$$

onde $\mathbf{A}$ são os auto-vetores da matriz $\mathbf{M}$ e $\mathbf{D}$ é a matriz diagonal de $\mathbf{M}$.

Para encontrar os auto-vetores $\mathbf{A}$ para matrizes simétricas, iniciamos o processo de diagonalização de $\mathbf{M}$ com rotações de Jacobi. Este é um processo iterativo no qual busca zerar os valores que não compõem a diagonal primária da matriz. Como critério de parada desse método, definimos que a soma dos quadrados dos elementos que não compõem a diagonal primária da matriz sejam próximos de zero. A matriz de auto-vetores, definida por $\mathbf{A}$ é inicializada como matriz identidade.

A cada iteração do método de rotações de Jacobi, selecionamos o maior valor contido em $\mathbf{M}$ nos elementos acima da diagonal principal. A seleção do maior valor é feita dado

que tende a zerar também os próximos elementos da matriz.

A partir do elemento (i, j) selecionado, é criada uma matriz **R** de rotação de Jacobi sobre esse elemento, no qual é multiplicada por **M** e multiplicada pela sua transposta, definindo a matriz **J**x de Jacobi, ou seja,

$$\mathbf{J} = \mathbf{R}_{i,j} * \mathbf{M} * \mathbf{R}^{T}{}_{i,j}. \tag{B.2}$$

Obtendo a matriz **J**, esta é multiplicada por **A**, sendo $\mathbf{A} = \mathbf{A} * \mathbf{J}$. Após a convergência do algoritmo de rotações de Jacobi, os auto-vetores estarão contidos em **A**. Para calcular a matriz diagonal **D**, temos que

$$\mathbf{D} = \mathbf{A}^{-1} * \mathbf{M} * \mathbf{A}. \tag{B.3}$$

Com a matriz de auto-vetores e a diagonal, é possível obter a raiz quadrada de **M** facilmente, através de

$$\sqrt{\mathbf{M}} = \mathbf{A} * sqrtd(\mathbf{D}) * \mathbf{A}^{-1}, \tag{B.4}$$

conforme equação B.1, onde *sqrtd* é uma função que executa a raiz quadrada dos elementos da diagonal principal de uma matriz.

Glossário

$AABB$ - Aligned Axis Bounding Volume

AL - Active List

API - Application Programming Interface

BP - Broad Phase

BSP - Binary Space Partition

BV - Bounding Volume

BVH - Bounding Volume Hierarchy

CD - Collision Detection

CPU - Central Processing Unit

EPA - Expanding Polytope Algorithm

GJK - Gilbert–Johnson–Keerthi

$GPGPU$ - General Purpose computing on Graphics Processing Units

GPU - Graphic Processing Unit

$k - DOP$ - k-Discreted Oriented Polytope

$MIMD$ - Multiple Instructions, Multiple Data

$MISD$ - Multiple Instructions, Single Data

NP - Narrow Phase

OBB - Oriented Bounding Volume

PBA - Phisically-Based Animation

PCA - Principal Component Analysis

RTD - Ray Tracing

SaP - Sweep and Prune

SAT - Separating Axis Theorem

$SIMD$ - Single Instruction, Multiple Data

$SISD$ - Single Instruction, Single Data