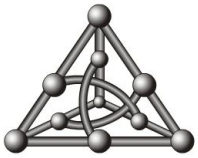
UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO E ESTATÍSTICA

Um Motor 3D para Simulação Dinâmica de Corpos Rígidos

Alexandre Soares da Silva

ORIENTADOR: Prof. Dr. Paulo A. Pagliosa



Campo Grande
2008

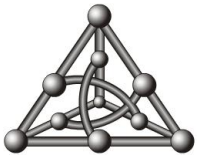
UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO E ESTATÍSTICA

Um Motor 3D para Simulação Dinâmica de Corpos Rígidos

Alexandre Soares da Silva

Dissertação apresentada ao Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação

ORIENTADOR: Prof. Dr. Paulo A. Pagliosa



Durante parte da elaboração deste projeto o autor recebeu apoio financeiro da CAPES.

Campo Grande
2008

Aos meus pais José e Marli, e ao meu orientador Paulo.

Agradecimentos

Quando pela primeira vez desenhamos uma esfera, simples e de aparência grosseira, na tela de nosso computador pessoal, eu sabia, bem lá no fundo, que não podíamos parar por ali. A partir daí, uma quantidade incontável de pessoas me apoiaram e contribuiram para que os estudos pudessem continuar. Embora não seja possível citar todas estas pessoas em alguns parágrafos, aqueles que direta ou indiretamente participaram desta caminhada têm minha eterna gratidão.

Agradeço aos meus pais, pois sempre confiaram e me apoiram, depositando amor, paciência e incentivo independente da situação. Deste modo, mesmo distantes, eu nunca estava realmente sozinho.

Meus sinceros agradecimentos ao meu orientador Paulo, exemplo de competência, caráter e dedicação não só durante o desenvolvimento do projeto, mas também durante todo período de convivência. Esta, uma pessoa de suma importância para a realização deste trabalho; sem ele o desenvolvimento e corretude deste trabalho não seriam possíveis.

Agradeço ao Márcio, pelos vários anos de companheirismo irrepreensível, que incansavelmente ajudou nas pesquisas necessárias contribuindo assim para que parte desse trabalho fosse possível.

Agradeço também a todos os professores, funcionários e colegas do curso de Bacharelado em Ciência da Computação da UFMS, por todas as vezes que, de alguma forma, contribuiram para o avanço deste projeto e consequentemente tornaram possível a continuação de nosso aprendizado.

Agradeço à CAPES e Fundect-MS pelo apoio financeiro concedido a mim e ao projeto. Também ao Prof. Henrique Mongelli por tornar disponível o equipamento utilizado.

Por fim, é essencial agradecer e dedicar o projeto a Deus por nos prover capacidade e dar o privilégio para podermos continuar nossa caminhada.

Resumo

Silva, A. S. *Um Motor 3D para Simulação Dinâmica de Corpos Rígidos.* Dissertação (Mestrado em Ciência da Computação), Universidade Federal de Mato Grosso do Sul, 2008.

O objetivo geral deste trabalho é o desenvolvimento de um motor 3D para aplicações de visualização e simulação dinâmica interativa de corpos rígidos em tempo real, incluindo jogos digitais. O motor 3D é resultante de modificações e extensões de um sistema de animação dinâmica chamado de AS, desenvolvido pelo Grupo de Visualização, Simulação e Games (GVSG) do DCT/UFMS. As principais extensões em AS são duas. A primeira é a implementação de um motor de física próprio para simulação de corpos rígidos, em substituição ao NVidia PhysX usado na versão original. Com isso, os resultados de outras pesquisas, tais como simulação de corpos elásticos e uso de unidades de processamento gráfico como co-processador do motor de física, podem ser mais facilmente incorporados ao sistema. O novo motor de física oferece suporte à definição de atores com múltiplas formas, vários tipos de junções e contato com atrito. A segunda é a implementação de um laço principal para tratamento de eventos de entrada, atualização da cena sendo simulada e renderização dos quadros da aplicação em tempo real. A atualização consiste na execução de ações definidas pelo desenvolvedor bem como na simulação pelo motor de física. A criação de atores e junções e a especificação de ações e das seqüências de eventos de uma aplicação do motor 3D podem ser especificadas em uma linguagem orientada a objetos própria de AS, chamada AL, em roteiros que são executados concorrentemente com o laço principal do motor.

Palavras-chave: *motor 3D*, *animação dinâmica*, *motor de física*.

Abstract

Silva, A. S. *Um Motor 3D para Simulação Dinâmica de Corpos Rígidos.* Master's Thesis, Universidade Federal de Mato Grosso do Sul, 2008.

The main purpose of this work is the development of a 3D engine for interactive, real time visualization and dynamic simulation of scenes composed of rigid bodies, including video games. The 3D engine results from modifications and extensions of a dynamic animation system, called AS, developed by the Group of Visualization, Simulation and Games (GVSG) of the DCT/UFMS. There are two main extensions. The first one is the implementation of a new physics engine for rigid bodies in substitution to the NVidia Physics used in the original version. As a consequence, the results of other researches such as the simulation of elastic bodies and the use of graphics processing units as co-processors can be integrated more directly into the system. The new physics engine supports the definition of actors with multiple shapes, various joint types, and frictional contact. The second extension is the implementation of game loop for input event handling, scene update, and rendering in real time. In the update stage the 3D engine executes the actions defined by the developer and invokes the physics engine in order to perform a dynamic simulation step of the scene. The creation of actors and joints and the specification of actions and sequences of events of a particular application can be specified by using an own object-oriented language, called AL, in scripts that are executed concurrently with the game loop of the 3D engine.

Key-words: *3D engine*, *dynamic animation*, *physics engine.*

Conteúdo

Lista de Figuras

CAPÍTULO 1

Introdução

1.1 Motivação e Justificativas

O Grupo de Visualização, Simulação e Games (GVSG) do DCT/UFMS desenvolveu um sistema de animação denominado AS destinado à simulação dinâmica de cenas com corpos rígidos [Oli06]. Simulação dinâmica utiliza leis da física para simular movimento, e um corpo rígido é um sistema de partículas no qual a distância entre duas partículas quaisquer não varia ao longo do tempo, não obstante a resultante das forças atuando no sistema. Neste sistema, uma linguagem de programação híbrida chamada LA é usada para descrever os objetos de uma cena a ser animada, bem como os roteiros e ações que modificam o estado da cena ao longo do tempo. O sistema é constituído de componentes responsáveis por compilar e executar uma animação e renderizar e exibir os quadros resultantes.

O compilador de AS toma como entrada arquivos contendo especificações das cenas em LA, e produz como saída um arquivo objeto contendo bytecodes que serão interpretados pela *máquina virtual de animação* (MVA). A MVA é o componente principal da versão original do sistema responsável pela interpretação e execução do fluxo de bytes correspondente às instruções do código objeto produzido pelo compilador da linguagem. Durante uma animação, a MVA renderiza quadros da cena que podem ser empacotadas por um ligador de arquivos de animação produzindo filmes em uma variedade de formatos (avi, mpeg, flic, dentre outros). A MVA é formada pelos subcomponentes ilustrados na Figura 1.1.

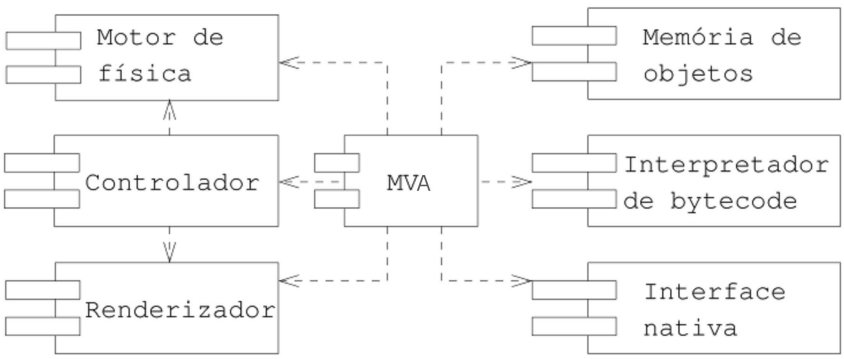


Figura 1.1: Arquitetura da MVA.

O presente trabalho faz parte de um projeto de pesquisa que visa estender as funcionalidades de AS a fim de que este possa ser aplicado ao desenvolvimento de aplicações interativas e em tempo real de visualização e simulação dinâmica de corpos rígidos e deformáveis, incluindo jogos digitais. No âmbito do referido projeto o objetivo geral deste trabalho consiste na adição de recursos de interação humano-computador em AS, bem como o desenvolvimento de um motor de física de corpos rígidos próprio para o sistema, em substituição ao PhysX [Cor08a] originalmente usado. O motor de física é a parte do sistema responsável pela atualização dos movimentos dos atores simulados em animação dinâmica. Para atender ao requisitos de performance exigidos na nova versão de AS, o núcleo de execução do sistema, baseado na *máquina virtual de animação*, deve ser modificado. A MVA se mostrou suficientemente especializada para a tarefa para a qual foi proposta, o que justificou seu desenvolvimento ao invés de se usar, por exemplo, uma máquina virtual Java no núcleo de AS. Contudo, para aplicações de simulação em tempo real, esta arquitetura pode não ser a mais adequada, dependendo do número de objetos presentes em uma cena. Por isso, optou-se por usar código nativo para execução de animações, mas, ao mesmo tempo, usufruir as vantagens do gerenciamento de memória e coleta de lixo automático provido pela MVA.

Neste trabalho o novo compilador da linguagem de animação gera código em MSIL, a linguagem de montagem independente de plataforma do framework `.NET` [Cor08b]. Este código objeto é convertido em código nativo pelo framework antes de ser executado pela primeira vez.

Entretanto, esta transição da linguagem de animação para código nativo não é feita diretamente. A descrição da cena em linguagem de animação é transformada, em primeiro lugar, em código `C#`, o qual alimentará o framework .NET. `C#` é uma linguagem de programação orientada a objeto e *type-safe* que tem suas origens na família de linguagens C. Possui coleta de lixo automática; tratamento de exceções e tipo seguros, dentre outras características de uma linguagem orientada a objetos. Estas características tornam quase que direta a conversão do código em LA para código `C#`.

O desenvolvimento de um motor de física próprio justifica-se pela necessidade de aplicação futura do sistema em simulações envolvendo corpos flexíveis. Diferentemente dos corpos rígidos, partículas de corpos flexíveis, por sua vez, podem apresentar deslocamentos relativos entre si, quando submetidos a ações externas.

Os arquivos binários do SDK do PhysX, desenvolvido pela AGEIA e adquirido recentemente pela NVIDIA, são disponibilizados gratuitamente para uso em aplicações não comerciais. Entretanto, não se trata de um framework, isto é, não é possível, sem o código fonte, estender o motor e acrescentar novas funcionalidades. Uma alternativa seria usar um motor de código aberto, como o Open Dynamics Engine (ODE) [eac]. ODE é uma biblioteca estável, com recursos necessários para simulações e acessível através de uma API C/`C++`. Contudo, este é um motor exclusivo para física de corpos rígidos. Após um detalhado estudo de sua arquitetura julgou-se que uma adaptação do ODE para tratar corpos flexíveis poderia ser tão complicada como começar um novo projeto que fosse fundação para simulação de corpos rígidos e flexíveis. Porém, o motor de física deste projeto conta com uma arquitetura própria, a qual não é baseada no ODE. Além disso, o domínio da tecnologia necessária à construção de um motor de física próprio pode ser útil no desenvolvimento de ferramentas de ensino de disciplinas de graduação e pós-graduação e como base de novas pesquisas. Por exemplo, uma versão para GPUs NVIDIA CUDA (*Computer Unified Device Architecture*) [Cor08c] do motor de física foi desenvolvida como produto de um projeto de mestrado por um integrante do GVSG/DCT-UFMS [Per08]. De qualquer maneira, algumas funções do ODE comuns ao motor desenvolvido, como por exemplo o algoritmo PCL descrito no Capítulo 2, foram tomadas como base para estudo e desenvolvimento do motor de física do projeto.

Para atender aos requisitos de interatividade, o sistema de animação deve ser dotado de

um laço responsável pelo tratamento de eventos do usuário, atualização e renderização da cena em tempo real. Este laço é chamado de laço principal. O laço principal é o responsável por contínuamente, a cada quadro de animação limitado pelo passo de tempo, alimentar o sistema com dados atualizados, exibir a cena e permitir que ocorram eventos de usuário. O novo sistema de animação possui um componente responsável por invocar e sincronizar o laço principal, invocar rotinas de atualização e tratamento de eventos do usuário. Nesta dissertação, tratamos cada componente do sistema de animação de maneira independente, e para evitar falta de clareza entre as principais funções de cada componente, referenciaremos o sistema completo sempre como motor para jogos ou simulações, ou simplesmente motor 3D. O termo motor de renderização será utilizado para mencionar o componente diretamente responsável pela gerência e renderização (exibição) da parte visual da cena. Na versão original do sistema, a MVA efetuava somente as etapas de atualização e renderização de cada quadro de animação, sem levar em consideração o tempo necessário para esta computação. Porém, aplicações interativas em tempo real devem exibir informações dentro de um passo de tempo determinado, para permitir que a interação seja contínua.

1.2 Objetivos e Contribuições

O objetivo geral deste trabalho é desenvolver um motor 3D para simulação dinâmica de cenas constituídas de corpos rígidos a partir de extensões ao sistema de animação do GVSG/DCT-UFMS. Pode-se caracterizar um motor 3D para jogos digitais ou simulações como sendo um *software* cujos componentes provêem as funcionalidades comuns e é capaz de executar todos os jogos ou simulações a partir deles desenvolvidos.

Os objetivos específicos são:

- uma API genérica para o motor 3D que seja capaz de acoplar motores de física, não apenas física de corpos rígidos ou motores de física de corpos rígidos já existentes, mas também motores de física de corpos deformáveis;
- adaptação da linguagem de animação LA, para oferecer suporte a tratamento de eventos de entrada do usuário;
- desenvolvimento de um motor de física para corpos rígidos, a ser utilizado no motor 3D; e
- implementação de um laço principal com execução de roteiros e ações em tempo real.

Como contribuições do projeto, podemos apontar os pontos principais como sendo:

- fornecer um ambiente de simulações fisicamente realísticas em 3D com cenas interativas em tempo real;
- criação de um ambiente que permite a visualização e simulação executadas diretamente em GPU, explorando sua natureza paralela, como por exemplo a utilização da tecnologia CUDA, como feito em [Per08].
- utilização da extensão da linguagem de animação LA desenvolvida pelo GVSG/DCT-UFMS como linguagem de descrição de roteiros, ações e eventos;
- fornecer uma ferramenta de ensino de computação gráfica e um *framework* que possa ajudar na visualização gráfica de resultados de alguns experimentos realizados pelo DCT-UFMS nas áreas de computação gráfica, IA, programação paralela e eventualmente até redes;

- com a implantação do motor de física de corpos deformáveis a ser desenvolvido pelo DCT-UFMS, pode-se realizar demonstrações em tempo-real de física de corpos flexíveis, e analisar sua viabilidade na utilização em jogos digitais; e
- estimular o estudo e desenvolvimento de ferramentas e aplicações de simulação e visualização em tempo real no âmbito do DCT-UFMS, inclusive jogos digitais.

1.3 Motores 3D e Motores de Física

Nesta seção apresentaremos conceitos do que são motores 3D, motores de física, e suas principais funcionalidades. Começaremos introduzindo o que é, e como funciona o laço principal de um motor 3D. Dentro deste laço, o dividimos em duas partes principais, uma que faz parte da rotina de renderização, como a cena pode ser exibida satisfatóriamente em tempo real, e outra que cuida das atualizações (incluídos ai eventos do usuário e física). Depois de definido o laço principal e sua funcionalidade, exibimos quais são os módulos de um motor 3D genérico, descrevendo qual o objetivo de cada módulo. Um dos módulos mais importantes, o motor de física, é exibido em seguida. Por fim, é dada uma visão geral do que é um motor de física para dinâmica de corpos rígidos.

1.3.1 O que é um Motor 3D para Simulações

Jogos digitais e simuladores de mundos tridimensionais são *softwares* e pertencem a uma classe chamada de aplicações de sofware em tempo real (*real-time software applications*). Formalmente, aplicações de *software* em tempo real significam aplicações que têm uma natureza ligada a um limite de tempo crítico, ou seja, aplicações nas quais a obtenção e resposta ao dados deve ser realizada em um tempo restrito [Dal03]. Considere como exemplo um programa que mostra informações, em um telão, sobre chegada e saída de aeronaves de um aeroporto; diversas linhas contém informações sobre número dos vôos, situação hora da aterrisagem, e assim por diante. O programa responde a eventos na hora - quando um avião atrasa ou aterrisa, por exemplo. As informações que chegam são de certo modo imprevisíveis e a aplicação deve processar e responder adequadamente. Além disso, estas informações devem ser exibidas no telão para representar os dados em função do tempo.

Em relação a programação de jogos ou simulações, motor é uma parte do projeto que provê certas funcionalidades ao seu programa. Analogamente ao motor de um veículo, do mesmo modo que uma pessoa quer dirigir o automóvel sem se preocupar com o funcionamento interno do motor, este motor serve para conduzir um projeto sem se preocupar com operações mais baixo nível como exibição de gráficos na tela, transformações de coordenadas 3D, comunicação com o adaptador de vídeo e som e assim por diante. Em outras palavras um motor deve realizar todo tabalho baixo nível pelo qual o programador de jogos não deve ser responsável. Um exemplo clássico de aplicação em tempo real é um jogo digital, pois geralmente deve exibir informações em um passo de tempo determinado para permitir que a interação seja contínua.

1.3.2 Laços em Tempo Real

Para simulações em tempo real, a simulação do mundo virtual e a intervenção do usuário podem ser consideradas tarefas pertencentes ao mesmo comportamento global, que é de "atualizar" o mundo virtual. Em outras palavras, o usuário é um caso especial da entidade mundo virtual. Para simplificação, estes tipos de programas serão vistos como aplicações que possuem uma rotina responsável pela atualização e outra pela renderização. Uma primeira

idéia seria utilizar uma abordagem que implementa ambas rotinas em um único laço, como mostra a Figura 1.2, então cada atualização é procedida por uma chamada de renderização, e assim por diante. Exibição e lógica estão implicitamente acopladas nesta abordagem.

Figura 1.2: Abordagem acoplada.

O problema deste tipo de laço é que em plataformas com configurações diferentes, ele pode não funcionar adequadamente; não executará uniformemente em várias configurações. Um computador com maior poder de processamento executaria o laço mais frequentemente, o que levaria a simulação a executar mais rápido, porém não necessariamente de modo que proporcionasse uma melhor experiência ao usuário. Outro problema é como o usuário percebe a degradação de performance. Este problema pode se manifestar se uma tarefa demora um tempo para ser executada maior do que o previsto.

Para solucionar o problema, é preciso analisar a natureza de cada componente. A renderização deve ser realizada o mais rápido possível; um computador mais rápido deve prover animação mais suave, melhor taxa de quadros por segundo, e assim por diante. Porém, o passo de atualização do mundo virtual não pode ser afetado por essa acelaração. Uma opção seria manter a atualização e renderização em sincronia, mas variando a granularidade da atualização de acordo com o tempo entre as chamadas sucessivas. Calcula-se o tempo decorrido desde a última atualização, então a parte responsável pela atualização utiliza esta informação para escalonar o passo dos eventos, e garantir que estas atualiaçãoes sejam realizadas na velocidade correta, independente do hardware. Embora esta solução seja válida para alguns casos específicos, ela é descartável. No momento que a taxa de quadros por segundo aumenta, não faz sentido incrementar a taxa de atualização do mundo.

Uma outra solução para o problema de sincronização é o modelo de laço chamado de desacoplado [VCF05]. Neste esquema uma *thread* executa a parte responsável pela renderização, enquanto outra cuida da atualização do mundo. Controlando a frequência que cada rotina é invocada, podemos assegurar que a parte de renderização terá o maior número possível de chamadas mantendo-se limitada por uma constante, independente do hardware para a resolução da etapa de atualização, Figura 1.3.

Imagine como exemplo, uma aplicação rodando a 60 FPS (*frame per second* ou quadros por segundo), enquanto a IA está em uma segunda thread a 15 FPS. Óbviamente, a atualização da IA será invocada de quatro em quatro quadros. No entanto, nada garante que atualizar a IA de quatro em quatro quadros garantirá quadros de renderização diferentes dentro deste intervalo. Mais quadros de renderização não significarão nada se o ciclo de IA for exatamente o mesmo para eles; na realidade, a animação estará rodando a 15 FPS. Para resolver o problema, pode-se dividir a IA em duas seções, o código real da IA executado utilizando-se um passo de tempo fixo e rotinas mais simples como por exemplo interpolação de animação e rotinas de atualização de trajetórias fixadas em uma taxa por quadro.

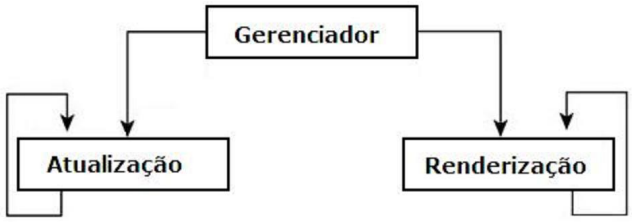


Figura 1.3: Abordagem desacoplada.

Entretanto, a abordagem que se baseia em threads também tem alguns problemas. Algumas CPUs não são muito eficientes em lidar com threads, principalmente quando funções temporais precisas são necessárias. Isto ocorre porque há variação da freqüência; o problema não está no overhead da chamada e criação das threads, mas sim nas funções reguladoras de tempo dos sistemas operacionais.

A opção mais popular é implementar threads utilizando laços regulares e temporizadores em um programa com apenas uma thread. A idéia é executar a atualizaçao e renderização seqüencialmente pulando algumas chamadas à atualização, para manter uma taxa fixa de chamadas. Em resumo, desacoplamos a rotina de renderização e atualização; a renderização é invocada o maior número de vezes possível enquanto que a atualização é sincronizada com o tempo. Devemos iniciar armazenando uma marca de tempo para cada chamada na etapa de atualização. Então, nas iterações subseqüentes do laço devemos calcular o tempo decorrido desde a última chamada (utilizando a marca de tempo) e compará-lo com o inverso da freqüência desejada. Este mecanismo é bem popular pois muitas vezes oferece maior controle do que threads e a implementação é mais simples. Não há necessidade de preocupação com memória compartilhada, sincronização, et cetera. O esquema é demonstrado na Figura 1.4.

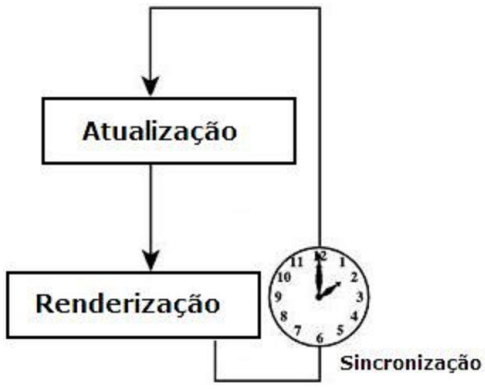


Figura 1.4: Única thread totalmente desacoplada.

Porém, a técnica mostrada é um pouco diferente de um laço de jogo ou simulação mais sofisticado. Assumimos que o tempo para cálculo de lógica não demora nada para se completar, não há tratamento de quando a janela perde o foco ou a aplicação é pausada, e assim por diante. Em um jogo digital devemos considerar também estes tempos. Mas a idéia é basicamente a mesma, apenas com um controle maior sobre os acontecimentos. Mais detalhes sobre laços para aplicações em tempo real na literatura podem ser encontrados em [VCF05].

1.3.3 Atualização e Renderização

Um motor 3D utilizado em um projeto de um jogo digital possui vários sub-motores, como por exemplo um motor de renderização, motor de som, motor de rede, motor de física, motor de IA e assim por diante [ZD04].

Rotinas relacionadas à simulação da cena (IA, física, animação), manipulação da rede e a resposta à eventos do usuário pertencem à mesma tarefa que é atualizar o mundo virtual. Para melhor entendimento, dividiremos o laço principal em duas partes: Atualização e Exibição.

Agumas decisões são fundamentais antes de se construir um motor 3D. A primeira decisão fundamental é se será um motor genérico, que pode ser utilizado para desenvolver tipos variados de jogos e aplicações, ou se deve ser focado em apenas um gênero, ou até mesmo para apenas um único jogo ou tipo de simulação; estas decisões influenciam principalmente a escolha das estruturas de dados a serem utilizadas. A segunda decisão importante é como a arquitetura da aplicação deve ser, além do fato de que se será desenvolvido exclusivamente para uma plataforma ou focado em múltiplas plataformas simultâneamente. A terceira consideração é a linguagem de programação, ou até mesmo conjunto de linguagens utilizadas, juntamente com as APIs escolhidas para cada módulo (Direct3D, OpenGL, DirectSound, et cetera).

Atualização

Começaremos distinguindo entre duas grandes entidades do mundo virtual. De um lado, entidades passivas, como por exemplo paredes e itens estáticos na cena. Formalmente, estes itens pertencem à cena, porém não têm um comportamento atribuído. São itens essenciais para algumas restrições com relação à cena e usuário, mas não tão importantes na etapa de atualização. Em simulações com grandes mundos virtuais, a rotina de atualização pré-seleciona uma subseção do mundo para que a porção responsável pelas restrições ao usuário ou determinado ator possa focar nestes poucos elementos, sendo assim mais eficiente.

A maior parte do tempo da rotina de atualização é gasta em outras entidades, as quais têm um comportamento definido. Elementos que se movimentam, portas que abrem e fecham e outros objetos que mudam de estado devem ser checados para manter a consistência da cena. Algumas aplicações dividem estes elementos em duas categorias: itens lógicos simples, como portas, elevadores, objetos que se movem a partir de uma animação pré-definida; e atores ou personagens com comportamento distinto (inteligentes). A diferença vem da complexidade dos códigos. Os elementos lógicos podem ser resolvidos com linhas simples de código, enquanto que os atores com comportamento necessitam de inteligência artificial, cálculos de dinâmica ou outros cálculos com um custo computacional maior.

Dadas estas classificações, o processo de atualização consiste de 4 passos [Dal03]:

- Um filtro seleciona os elementos relevantes à interatividade. Um ator a 15 kilômetros de distância do observador não parece ser um item importante do ponto de vista do usuário, nem uma parte de um jogo em um outro nível. No entanto nem todas entidades com estas características serão descartadas, para alguns casos é necessário calcular o coportamento para todas as entidades (como por exemplo, simulação dinâmica). Técnicas de escolha do nível de detalhes (LOD) [Lam03] também serão usadas, portanto é sempre preferível tê-las ordenadas;

- o estado do elemento ativo deve ser atualizado. Neste passo é feita a distinção entre entidades lógicas e inteligentes. No caso das inteligentes será necessário um processo mais apurado para atualizá-las;

- o terceiro passo requer que um plano de decisões do motor 3D seja tomado e implementado para gerar efetivamente as regras de comportamento; e
- atualizar o estado do mundo corretamente. Deve-se armazenar ou alterar dados da cena, como por exemplo, se algum ator se moveu, ou mesmo eliminá-lo das estruturas de dados caso ele não deva existir mais.

Apresentação

Dois dos módulos mais perceptíveis de um motor 3D são os responsáveis pelos gráficos e pelos efeitos sonoros. Nesta seção, ao nos referirmos à palavra renderizar, será em relação à gráficos, porém os conceitos básicos podem servir tanto para a parte gráfica como sonora da cena. O pipeline é parecido com o que controlava a lógica do jogo e pode ser dividido em: renderizar o mundo, renderizar os atores, e se for o caso, renderizar o jogador. Como é um pipeline genérico, poderia ser dividido em mais ou menos módulos, bem como alteradas as ordens em que os componentes da cena serão renderizados.

Renderização do Mundo

Renderizar em tempo real mundos de jogos completos compostos por centenas ou milhares de atores é praticamente inviável a menos que seja uma aplicação demasiadamente simples em relação ao número e complexidade dos atores. Nesta etapa o objetivo é renderizar os elementos passivos e dispositivos de lógica simples. Para qualquer aplicação semelhante, deve ser aplicado um filtro posteriormente à chamada de renderização para decidir o que será ou não será levado em conta. Um exemplo clássico é de uma região muito distante, ou zonas invisíveis do mundo que podem ser excluídas porque provêem pouca ou nenhuma informação ao usuário e contribuiria apenas para diminuir a taxa de quadros por segundo devido ao aumento da complexidade. Resumindo, qualquer pipeline de renderização consiste de duas partes: selecionar apenas as partes relevantes ao usuário e cuidar da renderização propriamente dita.

A rotina responsável pela seleção é implementada usando técnicas de *culling*, *clipping*, e ocultamento de superfícies (*occlusion*) [ZD04]. Deste modo, a representação resultante é apenas a parte visível do mundo em relação ao ponto de vista do observador. Esta rotina é a base de qualquer pipeline gráfico moderno. Não entraremos em detalhes sobre o pipeline gráfico, mas informações mais avançadas de como realizar a renderização de ambientes externos e internos podem ser encontrados em diversas fontes da literatura e internet [Ebe00, ZD04, MB98, RE01, Lun03].

Além do filtro de superfícies, um processo opcional, a seleção do nível de detalhes ou LOD, as vezes é aplicado aos dados visíveis para selecionar a relevância do dado e o nível de detalhamento adequado para renderizá-lo. Tome o exemplo de um ator formado de 10000 triângulos à 500 metros de distância do observador; muito provavelmente não necessitará de toda essa complexidade pois cada triângulo ocuparia um pequena fração de um pixel da imagem (pixel (aglutinação de Picture e Element) é o menor elemento num dispositivo de exibição, ao qual é possivel atribuir-se uma cor). Além disso, uma representação em menor resolução e mais eficiente pode ser utilizada para melhorar a performance.

Após selecionada a parte visivel da cena e à ela atribuída um nível de detalhes adequado, para efetivamente mostrá-la no dispositivo de exibição são necessárias mais duas etapas:

- Primeiro a geometria é armazenada em uma estrutura de dados eficiente, o que varia conforme a natureza da aplicação. Este passo chama-se *geometry packing* ou empacotamento da geometria [Dal03].

- O pacote de geometria é enviado ao hardware onde será processado. O hardware gráfico tem um grande impacto com relação ao empacotamento da geometria, o mecanismo de entrega dos dados influi muito na performance. Os principais mecanismos são encontrados nas duas principais APIs gráficas existentes , a OpenGL e DirectX [Corb].

A renderização do aúdio funciona de maneira um pouco diferente do que a de gráficos. Claramente, não é possível apenas filtrar o que é ou não visível. Se um ator está atrás do observador, ele não será visto, porém seus efeitos sonoros ainda poderão ser ouvidos. No entanto, filtros também são utilizados, geralmente parametrizados por alguma distância em relação a uma métrica volumétrica. Para saber quais sons são efetivamente audíveis pode-se calcular atenuação de cada um deles. Feito isso os fontes sonoras estão identificadas, basta enviar as dados de aúdio para o dispositivo responsável pelo som.

Atores

Na renderização de atores necessitamos de um pipeline específico devido às propriedades de animação. Entretanto, ainda é possível filtrar a quantidade de atores visíveis ao observador, da mesma maneira que na geometria do mundo. Este passo é essencial, porque a animação dinâmica da geometria é mais cara computacionalmente do que processar elementos passivos. Por este motivo, faz sentido que seja aplicada somente quando necessária. Independente do tipo de animação desejada, o resultado final são sempre dados geométricos estáticos que representam o estado instantâneo de como o ator deve parecer em um determinado quadro de animação. A partir deste momento a representação geométrica dos atores pode ser processada como a geometria regular do mundo.

Para movimentação de atores dinâmicamente, realizada por um motor de física, a animação dá-se de maneira diferente da descrita. Como no caso dos filtros para efeitos sonoros, não é possível filtrar o que é ou não visível, pelo simples motivo de que a próxima posição de um ator depende de sua posição anterior; por esta razão deve ser constantemente atualizada. Para os cálculos da dinâmica, geralmente são utilizados modelos mais simples do que os renderizáveis. Calcular as equações responsáveis pelas atualizações sobre uma geometria mais simples é menos caro computacionalmente, e ainda satisfatóriamente convincentes. Além disso, é possível organizar estes atores espacialmente e hierarquicamente para que seja mais rápido encontrar quais atores interagirão uns com os outros. Mais detalhes sobre o motor de física e cálculos da animação dinâmica são descritos no Capítulo 2.

A parte do laço principal responsável pela renderização de geometria inanimada, animada e pela sonorização é resumida pelo seguinte algorítmo:

```
Representação do Mundo 3D
  Selecionar subconjunto visível (gráficos)
    Remoção de superfície escondida
  Selecionar nível de detalhes adequado (LOD)
  Geometry packing do mundo 3D
  Renderizar geometria do mundo
Sonorização do Mundo 3D
  Selecionar fontes sonoras audíveis (som)
    Empacotar dados de audio
    Enviar para o hardware de som
Apresentação dos Atores
  Selecionar subconjunto visível
  Animar
  Geometry packing dos atores
  Renderizar dados dos atores
```

1.4 Partes de um Motor 3D

Um motor para jogos deve ser capaz de gerenciar o mundo virtual constituido de diferentes tipos de objetos ou atores, capaz de realizar detecção de colisão entre estes objetos, e até mesmo contar com física avançada. Atores controlados pelo computador são uma parte importante para maioria dos jogos, se estes não são jogos puramente multiplayer. Por isso, um componente de também IA também pode ser essencial. Muitos motores utilizam-se de linguagens de roteiros para criar conteúdo relativo ao jogo em si. Em vez de utilizar linguagens de programação baixo nível embutidas no motor 3D, há suporte à linguagens de roteiro mais fáceis de serem utilizadas por um programador de mundos, em vez de necessitar de um programador dedicado. Implementações que podem ser efetuadas por uma linguagem de roteiros variam desde determinar qual tecla foi apertada na ordem para acionar alguma nova ação, até interagir progressivamente em um ambiente. Finalmente, um motor 3D para jogos também é constituido de centenas de algoritmos implementados e adaptados variando em nível de sofisticação [Dal03]. Como dito anteriormente, o motor é formado de vários sub-motores, como uma visão geral é dada na Figura 1.5.

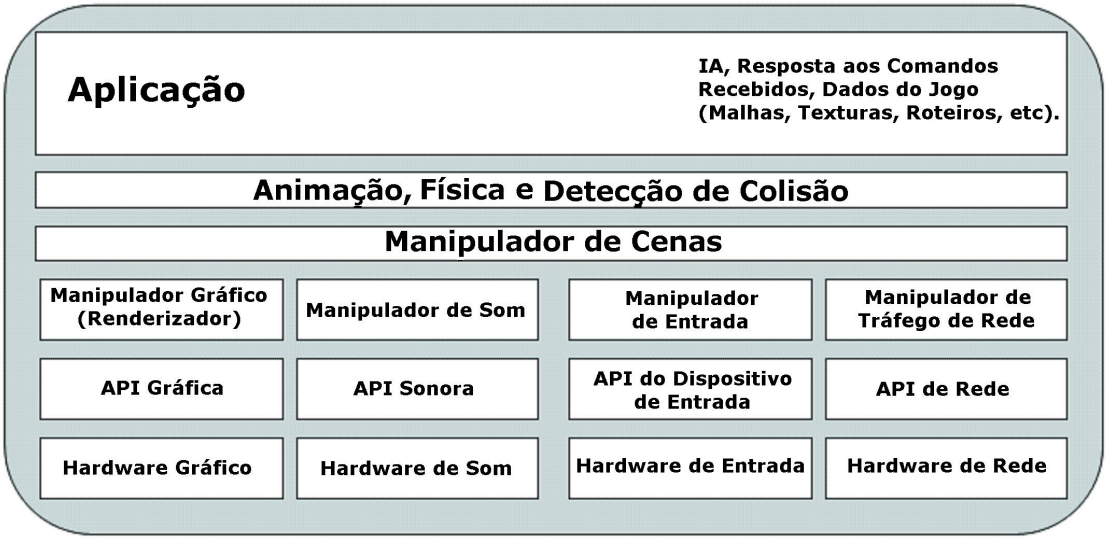


Figura 1.5: Esquema generalizado de um motor 3D para jogos.

A visão clássica do que um motor de renderização faz é renderizar triângulos (ou polígonos). Com certeza, este é um requisito necessário, mas não único. Visto como uma caixa preta, um renderizador é um produto consumidor [ZD04]. Ele consome triângulos e produz saídas rasterizadas em um display gráfico. Como um consumidor, pode ser alimentado por dados em excesso e muito rapidamente, ou pode ficar ocioso esperando por algo para fazer. Um motor de renderização atual, de certa maneira deve controlar os dados que serão entrada para o renderizador; este processo é chamado de gerenciamento da cena e está um nível antes do renderizador Figura 1.5.

Em motores 3D para jogos, há diferentes subsistemas que provêm meios de portabilidade para todo o motor. São acessados através de especificações abstratas por uma interface (API), escondendo assim as dependências do sistema das implementações reais. A implementação de um destes subsistemas tem apenas que fornecer a interface para as funções desejadas, assegurar um comportamento consistente, encapsulando a forma de como este comportamento é executado. No diagrama genérico da Figura 1.5 temos os subsistemas de aúdio, de entrada, de rede e gráficos:

- O subsistema de aúdio mapeia diretamente o sistema de aúdio da camada inferior (hardware) através de uma API. Oferece serviços como criar canais de aúdio, toca um canal, ajusta o panorama(posição stéreo) de um canal, ajusta o volume, et cetera;

- o subsistema de entrada manipula toda entrada do usuário, se é por um mouse ou teclado, ou ainda *joystick*. Diferentemente do submotor de aúdio, que provê serviços quase tão baixo nível quanto dos sistemas operacionais comuns, o subsistema de entrada já traduz os eventos para um formato que pode ser utilizado diretamente pelo código do jogo;

- o subsistema de rede encapsula tanto a camada inferior do protocolo de rede, isto é, TCP/IP ou IPX, assim como o protocolo de jogo do motor, por exemplo *peer-to-peer* ou cliente servidor. O código do jogo invoca o subsistema de rede para replicar o eventos do jogo para os outros clientes. Estes eventos são chamados de eventos remotos;

- o susbsistema responsável pelo encapsulamento da API Gráfica depende da própria API gráfica, em vez de uma camada inferior do sistema. Existem diferentes implementações para cada API gráfica suportada. Adicionar suporte para outras APIs gráficas é uma questão de escrever mais implementações de subsistemas. Estas implementações não dependem diretamente do motor 3D, entretanto são portáveis para a plataforma alvo imediatamente já que a maioria dos códigos de renderização são construidas sobre uma camada de abstração de funcionalidades provida pelo subsistema citado.

Além destes, podemos ter módulos específicos para algum outro tipo de recurso necessário. Por exemplo, um subsistema de objetos contém a maioria dos serviços operando, ou gerenciando, objetos em toda a cena. Ou seja, gerenciando classes, instâncias, coleções de objetos, detecção de colisão, et cetera. Também deve prover facilidades como permitir ao código registrar novos tipos de objetos, que são então gerenciados automaticamente pelos outros subsistemas. Isto é importante para estensibilidade do sistema. Outro exemplo é a possibilidade de existir um subsistema de partículas contendo todo código referente ao sistema de partículas. Isso implicaria que o sistema de partículas é independente do restante dos sistemas, devido ao fato de que é construido sobre serviços abstratos oferecidos por subsistemas de baixo nível. O subsistema conteria todos serviços baixo nível para trabalhar com partículas e sistemas de partículas, desde definição e gerenciamento até animação de renderização.

Todo motor 3D deve ter um subsistema matemático, que ofereça rotinas matemáticas gerais, como operações com matrizes e vetores, quaternions, assim como algumas funções mais alto nível como transformar volumes completos de um quadro de referência à outro. É comum também haver um subsistema responsável pelos modelos de atores, oferecendo diversos serviços que lidam com a geometria dos objetos, como culling hierárquico, clipping, atravessar uma árvore BSP, entre outros. Embora estes sejam usados em sua maioria pelo código que renderiza a cena, não são parte próprias do código de renderização, já que são, de algum modo, naturalmente mais alto nível e ainda independentes do sistema. Por possuirem estas características, estes objetos podem ser usados por códigos residentes em diferentes níveis da hierarquia de abstração, isto é, por gerenciadores de cenas alto nível, assim como código de renderização baixo nível; resumidamente, este subsistema pode atuar como se fosse uma caixa de ferramentas do programador, contendo implementações universalmente acessíveis de algoritmos gráficos frequentemente necessário como os algoritmos : Sutherland-Hodgman clipper, culling utilizando volumes limitantes, traçado de raios em uma árvore BSP, e outros algoritmos tradicionais de remoçao de superfície escondida em computação gráfica. Definições mais detalhadas sobre destes algoritmos podem ser encontrados em [GV03].

Para depuração da simulação ou jogo digital ter uma janela console é boa opção de facilidade para desenvolvimento, permitindo ao motor 3D e ao programa ser configurado em

execução, dados serem listados, carregados e manipulados interativamente. Por exemplo, tornar um filtro de textura ligado e desligado utilizando apenas uma variável de console. O código do jogo pode registrar variáveis e comandos adicionais de console sem mexer no código do console propriamente dito, usando um conjunto de funções para registro.

1.5 Visão Geral de Um Motor de Física

Um motor de física é um conjunto de bibliotecas que auxiliam na criação de tais sistemas. A maioria dos motores de física existentes são para resolução da dinâmica dos corpos rígidos [Cor08a, eac]. Podem calcular a dinâmica de objetos sólidos, junções, contatos e colisões, atrito, entre outros. Em outras palavras, são bibliotecas auxiliares para a resolução de sistemas dinâmicos. Um sistema dinâmico é uma coleção de atores componentes da cena que se movem ou mudam de posição ou orientação com o passar do tempo. Tem-se quatro diferentes classificações para estas simulações:

- simulação *on-line*: O sistema deve ser executado tão rapidamente quanto na realidade.
- simulação *off-line*: O sistema executa mais lentamente do que a realidade.
- simulação interativa: O sistema é executado rápido o suficiente para permitir que o usuário a controle (dentro do laço principal da simulação).
- simulação em tempo real: Uma simulação onde é garantido que o sistema será atualizado uma taxa fixa de vezes por segundo.

Um motor de física calcula apenas atributos de uma simulação que estejam relacionados diretamente à dinâmica. O motor de física é portanto, parte de um sistema maior, como motores 3D ou simuladores de fenômenos naturais ou comportamento dinâmico.

Dentro do motor 3D para simulações ou jogos, um motor de física para corpos rígidos é a parte responsável pelos cálculos necessários à atualização das posições, orientações e velocidades dos atores dinâmicos. Dadas posição e orientação de um ator para um determinado passo de tempo t, o motor de física deve ser capaz de computar qual serão os novos estados físicos e espaciais deste ator no passo de tempo $t + \Delta t$. Nestes caso, tratando-se de corpos rígidos, deve ainda impedir que haja interpenetração entre os corpos; para um motor de física de corpos deformáveis, a deformação ocorrida nestes corpos também deve ser calculada. A partir de um motor de física é possível simular, naturalmente, uma série de processos físicos realísticamente. Para um simulador ou motor de física, deve-se primeiramente definir as seguintes etapas básicas em sua construção:

- Definir os processos ou processo físico que deverá ser simulado (movimento de corpos rígidos, movimento de fluídos, rotação de um motor, et cetera);
- Criar modelos ou um sistema que siga os requisitos dos processos definidos anteriormente, e representá-los matematicamente. Nesta representação, equações e inequações representarão o comportamento dos modelos no sistema e do próprio sistema;
- Escrever um programa capaz de calcular as equações e inequações do modelo matemático; e
- Obter os resultados após a execução do programa.

O esquema de como funciona um motor de física genérico para dinâmica dos corpos rígidos é mostrado na Figura 1.6. A representação do sistema é de acordo com as necessidades do

usuário, para um sistema de dinâmica de corpos rígidos esta descrição é composta de corpos rígidos, junções (conectividade entre diferentes corpos no sistema, Capítulo 2), controladores, pontos de contato (para armazenamento dos pontos de colisão), entre outros. Controladores são objetos, que mandam mensagens para a simulação para tentar e controlar o comportamento dos outros objetos sendo simulados. Por exemplo, um controlador simples pode ter como tarefa manter sempre um ângulo mínimo entre dois corpos rígidos de uma mesma junção. O motor de física toma como entrada a descrição do sistema, o estado atual desse sistema, as forças que atuarão sobre os objetos e um passo de tempo que será considerado como tempo transcorrido. Baseando-se neste passo de tempo, o motor de física devolve como saída o novo estado do sistema (seja posição, velocidade, rotação, et cetera). O resultado é utilizado pelo motor 3D que atualiza seus atores e envia os dados ao renderizador para exibir os resultados.

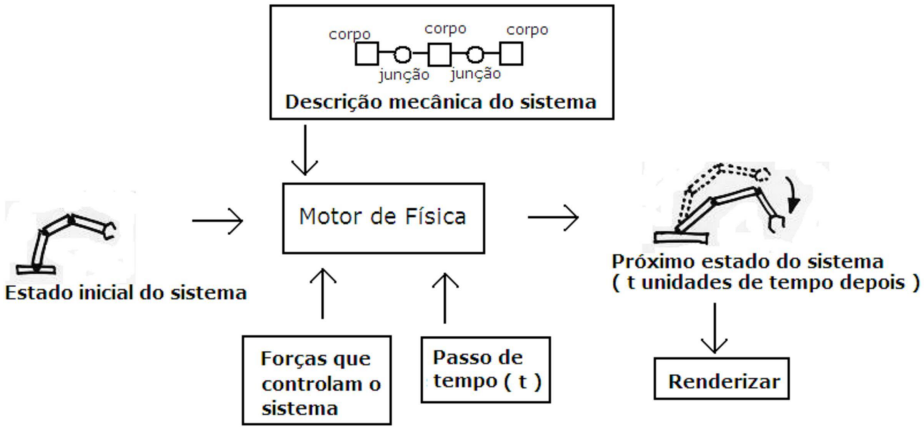


Figura 1.6: Diagrama conceitual de um motor de física.

1.5.1 Modularidade de Um Motor de Física

É consenso entre diversos autores [Ebe00, Erl04, Bar01] que simular corpos rígidos não só não é uma tarefa fácil, como também demanda tempo de estudo, além de possuir uma larga curva de aprendizado devido a quantidade de teoria envolvida. A descrição da modularidade dos motores de física mostra apenas uma visão das várias peças necessárias para a construção de um simulador. De fato, o motor de física é formado por diversos módulos, cada qual responsável por uma tarefa específica, isto é, há um módulo responsável por calcular o impulso de uma colisão, outro para determinar o momento do impacto, outro para atualizar as posições, e assim por diante.

No motor de física simulamos uma configuração que é uma coleção de objetos especificados pelo usuário, mais especificamente, atores e ações que interagirão com o sistema. Em um determinado momento, a configuração tem um estado, que é simplesmente a concatenação dos estados de todos os atores naquela configuração. O estado de um corpo rígido geralmente é descrito por sua posição, orientação e velocidade. Por esta razão, um quadro de animação é equivalente a uma "foto instantânea" em determinado ponto do tempo da simulação. Consecutivos quadros exibidos em um determinado espaço de tempo caracterizam uma animação [Erl04].

O laço responsável pela simulação é um laço simples com três estados. O estado **A** calcula as novas posições (na primeira execução não há novas posições, os objetos estão em um estado inicial) e rotações, o estado **B** testa a colisão entre os objetos, por fim, o estado **C** aplica as

forças e impulsos necessários. Olhando de maneira mais simplificada ainda, podemos dividir o simulador em dois componentes: um simulador e um detector de colisão.

A detecção de colisão é um problema puramente geométrico, enquanto que a simulação dinâmica, em sua maioria, preocupa-se com física inspirada em equações parciais diferenciais de movimento. Por esta razão, a simulação e a detecção de colisão são módulos distintos. Na descrição anterior, os estados **A** e **C** pertenceriam ao componente de simulação, ou seja, o próprio motor de física, e o estado **B** pertence à um componente de detecção de colisão, que apesar de ser importante ao motor de física é um módulo a parte.

Na colisão o objetivo é saber se a geometria de um objeto foi violada ou não. Por outro lado, a resposta à colisão lida com o aspecto físico. Neste caso, a principal pergunta é como alterar o movimento dos objetos de tal modo que eles não inter-penetrem uns nos outros. Para melhor entendimento, vamos analisar os componentes de detecção de colisão e dinâmica em pequenas partes. Separamos os componentes em módulos menores, cada qual responsável por uma tarefa mais específica. A modularidade apresentada aqui, foi baseada no design modular proposto por [Erl01] e também utilizado em [Erl04, KZZ02]. Primeiramente desenvolvida por razões educacionais, segundo o autor, esta abordagem permite um melhor entendimento do motor de física.

O componente de simulação é composto de três outro módulos chamados aqui de: Controlador de tempo, Calculador de movimento e Calculador de Restrições. Já o componente responsável pela detecção de colisão fica dividido em outros três: Fase geral (ou *Broad phase*), Fase reduzida(ou *Narrow phase*) e Determinação de Contato. Figura 1.7.

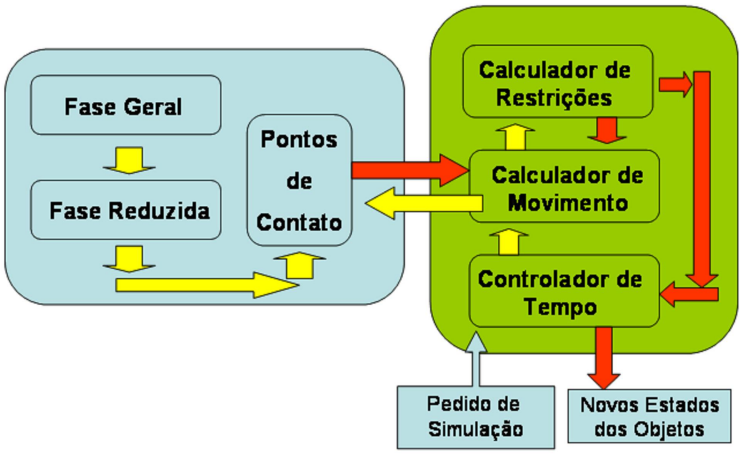


Figura 1.7: Design modular de propósito genérico.

Simulador

O componente chamado simulador é o responsável por prover movimento aos corpos rígidos em determinada fatia de tempo, e também, gerar forças que impeçam a inter-penetração entre eles. Dada uma fatia de tempo e estado corente dos corpos rígidos, este componente calcula as restrições necessárias para impedir inter-penetração, determina o movimento e com isso pode dizer quais as novas posições, orientações e velocidades dos corpos rígidos. Podemos dividir este componente em três módulos básicos:

- O controlador de tempo: é a parte central do simulador, responsável por determinar quando e como os outros módulos deste componente são invocados. O laço de simulação é iniciado após requisição do controlador de tempo, para alguma simulação, do tempo

inicial ao tempo final. Após o término da simulação, o estado da configuração corrente é retornado, conforme ilustra a Figura 1.7. O controlador de tempo pode utilizar um passo de tempo fixo e/ou adaptativo de acordo com algum algoritmo. Para algorítmos que mantém um passo de tempo fixo, o controlador de tempo pede ao calculador de movimento que simule o movimento para um passo de tempo fixo adiante, até que chegue ao final da simulação.

- Calculador de Restrições: O calculador de movimento invoca o calculador de restrições para conhecer as forças de restrição que atuam no sistema as quais são usadas nas ODEs para calcular o movimento contínuo dos objetos. Por hora, basta saber que as forças de restrição são obrigatórias para prevenir que os objetos inter-penetrem; restrições serão descritas com mais detalhes conforme avançamos no capítulo. Consideramos dois tipos de forças de restrições: forças de contato, restringindo os elos dos objetos articulados; e forças de contato que surgem da colisão entre objetos. Para forças de contato, o calculador de restrições precisa consultar o detector de colisões para obter as regiões de contato entre os objetos.

- Calculador de Movimento: é responsável pelo movimento contínuo de todos objetos da configuração de acordo com as equações que descrevem o movimento dos corpos rígidos (detalhes destas equações serão dadas posteriormente no Capítulo 2). Geralmente, estas equações são dadas na forma de equações diferenciais ordinárias (ODE)[Bar01]. O que compõe uma ODE será descrito mais adiante, no Capítulo 2, mas pode-se entender, por enquanto, que uma ODE aqui descreve o estado de um objeto.

  A ODE tipicamente depende das restrições e forças externas. Por esta razão, há uma ordem específica na qual o movimento é calculado. Primeiro calculamos todas as forças externas atuantes e depois todas as forças das restrições agindo no sistema. Note que, para o cálculo das forças de restrição, o calculador de movimento deve enviar o estado intermediário da configuração ao calculador de restrições, e este por sua vez, computa todas as forças de restrição e as retorna ao calculador de movimento. Após calculadas estas forças, o calculador de movimento finalmente aplicá-as às ODEs. Há uma estrutura composta por dois laços dentro do módulo de simulação para um único quadro de configuração; o laço mais externo no controlador de tempo e o laço interno dos passos de integração das ODEs dentro do calculador de movimento. A duração do passo responsável pela integração depende da configuração dos objetos, precisão desejada e estabilidade numérica do sistema [Erl04].

A Figura 1.7 indica resumidamente os passos realizados pelos módulos de um motor de física. A sequência natural dos passos está indicada pelas setas amarelas. Setas vermelhas representam uma resposta à determinada requisição de informação. O início dá-se com o pedido de simulação para determinado passo de tempo. Logo em seguida, o controlador de tempo invoca o Calculador de Movimento e este, por sua vez, após realizar o cálculo inicial das forças externas que agem no sistema, pede ao componente de Detecção de Colisão os pontos de Contato entre os atores (os quais gerarão restrições de contato). O componente de Detecção de Colisão, os devolve para que os pontos sejam passados ao Calculador de restrições. No calculador de restrições, são encontradas as forças de restrição necessárias para manter a integridade das propriedades dos corpos rígidos; estas forças são entregues ao Calculador de Movimento, e finalmente ele atualiza os estados dos atores da cena; voltando ao controlador de tempo, os dados atualizados são entregues ao invocador.

Detector de Colisões

O objetivo do detector de colisões é obter os pares de atores que estão colidindo ou colidirão dentro do passo de tempo desejado. Analogamente ao motor de renderização, se há uma grande quantidade de atores não é viável, computacionalmente, processar todos estes objetos presentes no mundo virtual o mesmo tempo, visto que alguns deles possivelmente não estarão sofrendo forças ou colidindo com algum outro corpo. A idéia é descartar a maioria dos objetos que, com certeza, não estarão em colisão; deste modo um número menor de objetos serão processados. Um outro detalhe é que, geralmente para um motor 3D de jogos digitais ou mesmo simulações que almejem tempo real, os modelos físicos dos atores não são os mesmos dos modelos gráficos. São modelos mais simplificados. Isto acontece porque, na maioria dos casos, o movimento e colisão calculados em objetos mais simples é real o suficiente para representar o movimento de seus respectivos modelos gráficos detalhados. Veja como exemplo a Figura 1.8, a Figura 1.8(A) e Figura 1.8(B) mostram a representação das formas geométricas de um automóvel que são utilizadas para cálculo de colisão. Com formas simples como caixas, cilindros esferas, fica menos custoso calcular os pontos de contato dos componentes do automóvel com o ambiente potencialmenete colidível. Na Figura 1.8(C) e Figura 1.8(D) são exibidas as representações gráficas do automóvel, estas que efetivamente, são mostradas ao usuário no dispositivo de saída.

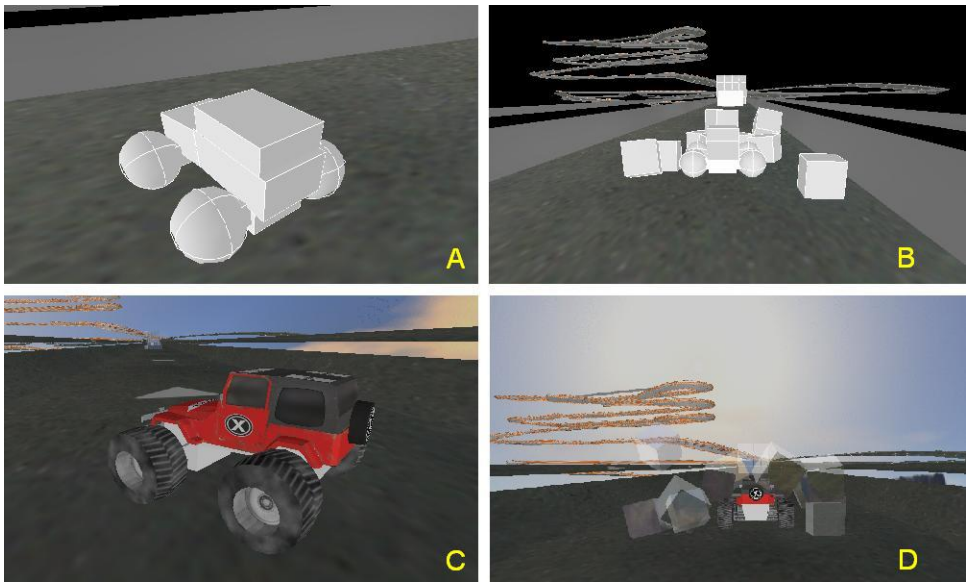


Figura 1.8: Representação do modelo físico e gráfico em um motor 3D.

Outro ponto importante de um detector de colisão, é como organizar os dados. O mundo virtual pode conter um vasto número de objetos interagindo, e uma comparação exaustiva entre todos pode ser caro demais. Para minimizar esse problema os objetos devem ser separados em "grupos de colisão" [Ebe00]. Por exemplo, salas ou caixas limitantes podem ser candidatas natural ao particionamento, cada uma agiria como um grupo. Apenas objetos dentro da mesma caixa ou sala seriam comparados. É claro que, objetos que se movem de um grupo à outro devem ser atualizados, mas antes ser comparado com os grupos de onde vieram e depois para a caixa ou sala adjacente na qual estão indo.

Dado um ator, também deve ser levado em conta como ele é estruturado em relação aos outros objetos da cena. Uma idéia seria utilizar uma representação que utilizase um grafo e volumes limitantes em seus nós. Assim, seria possível realizar o teste de rejeição mais grosseiro; e para os nós, podemos utilizar outro tipo de subdivisão, como árvores de

volumes limitantes. Neste projeto, dentro do propósito da simulação, não há um mundo significativamente extenso e nem objetos muito complexos foram utilizados; portanto não fez-se necessária a utilização de tais estruturas de dados. Porém, as mesmas podem ser modularmente incluídas sem problema algum caso sejam necessárias.

Após realizarmos os testes de colisão, devemos notificar o motor 3D ou motor de física sobre a informação obtida. Um modo razoável de reportar informação sobre a colisão é utilizando *callbacks*. Cada objeto envolvido na colisão possui um callback que é executado quando uma interseção é prevista ou detectada. Informações relevantes sobre a colisão (local, momento, vetores normais, atributos da superfície, et cetera.) é passado ao callback. A partir disso, é de responsabilidade da aplicação decidir o que fazer com a informação. O mecanismo de callback provê resposta à colisão e mantém a abstração que separa a detecção e a resposta à colisão. Este método de reportar informações é muito utilizado em sistemas de física e módulos de detecção de colisão que permita integração com outros módulos de sistemas físicos [Ebe00].

Podemos dividir a detecção de colisão em três fases distintas:

- **Broad Phase (Fase Geral)**
  A idéia mais simples para realizar a deteção de colisão entre os objetos do mundo seria comparar todas as possíveis interseções de todos objetos contra todos os outros, num total de $n(n-1)/2$ testes de interseção para n objetos. O propósito da fase geral do módulo de detecção de colisão é reduzir o número de comparações analisando o sistema em larga escala, de modo que pares de objetos bem próximos sejam reservados para testes na fase reduzida (narrow phase) da detecção de colisão. Como consequência desta poda, a carga computacional destinada a detecção de colisão pode ser reduzida consideravelmente. Exemplos clássicos de algorítmos desenvolvidos para este fim são busca exaustiva, sweep and prune, grids multi-níveis (ou tabelas hash hierárquicas) [Bar01]. Para determinar o grau de proximidade, os objetos geralmente são dotados de um volume limitante mais simples que a geometria do próprio objeto na forma de uma caixa, esfera, cilindro, et cetera. Por exemplo, no trabalho de [Per08], esferas limitantes foram utilizadas para efetuar as podas dos atores naa fase geral tanto em CPU como em GPU.

- **Narrow Phase (Fase Reduzida)**
  A fase reduzida da detecção de colisão examina os pares de objetos para descobrir se os objetos realmente colidem ou não. Algorítmos utilizados por esta etapa da detecção de colisão retornam uma resposta maior do que apenas a informação de que se houve ou não colisão. Os dados mais comuns de serem retornados são os pontos de contato, normal no ponto de contato, entre outras características da inter-penetração [Bar01, Ebe00]. Há uma infinidade de algorítmos para realização do cálculo da fase reduzida da detecção de colisão, mas a escolha do algoritmo não influencia de maneira alguma na modularidade do motor de física ou na determinação dos pontos de contato.

- **Determinação dos Pontos de Contato**
  A parte responsável pela determinação do contato calcula as regiões em contato entre dois objetos que se interpenetram ou tocam-se. Matematicamente falando, a região de contato é a interseção das superfícies dos dois objetos em contato ou inter-penetração. Determinar a região de contato é um problema puramente geométrico, no entanto não trivial pelas incertezas e singularidades das representações dos modelos [Erl04].

1.6 Revisão Bibliográfica

Com o surgimento dos sistemas de arquivos nos primeiros computadores pessoais, evidenciou-se que era necessário uma ferramenta para organizar melhor o conteúdo dos jogos digitais, minimizando o caos. Além do que, o trabalho poderia ser dividido entre várias pessoas, cada qual cuidaria de uma área específica do jogo. Além disso, com a divisão poderia-se substituir apenas os arquivos de dados do jogo e criar um novo jogo idêntico na jogabilidade mas diferente no conteúdo. Esta abordagem de motor para jogos baseados em dados tornou-se popular.

No fim dos anos 80 e inicio dos anos 90, alguns padrões de motor para jogos começaram a surgir. Suas especificações foram reveladas de modo que usuários podiam criar novas missões e conteúdo eles mesmos, aumentando a longevidade do jogo. O exemplo mais popular foi o motor do jogo *Doom* introduzido pela *Id Software* em 1993. Doom era realmente um adepto da filosofia dos motores para jogos: dados e comportamento (IA) eram implementados via arquivos externos, de modo que o arquivo executável era apenas carregador de fases ou mundos. Doom foi um marco na história dos motores, pois introduziu conceitos e algoritmos até hoje utilizados em jogos digitais. Naturalmente estes algoritmos foram adaptados para comportar mundos mais modernos e placas gráficas atuais [ZD04].

O conceito de motor evoluiu muito desde então. O código central de um jogo é o mínimo possível, provendo apenas funcionalidades e algoritmos essenciais ao jogo. Isto é, tarefas com relação ao pipeline de renderização, APIs de aúdio e de rede e interfaces de IA onde módulos externos podem ser acoplados. Alguns roteiros são escritos pelos programadores de jogos através de uma linguagem descritiva, e funcionam sem necessidade de recompilar qualquer parte do motor. Dentre estas, pode-se citar como exemplo a linguagem *Lua* [IdFC06]. Lua é uma linguagem de programação projetada para estender aplicações. É também freqüentemente usada como uma linguagem de propósito geral. Combina programação procedural com construções para descrição de dados, baseadas em tabelas associativas e semântica extensível. Lua é tipada dinâmicamente, interpretada a partir de bytecodes, tem gerenciamento automático de memória com coleta de lixo. Essas características fazem de Lua uma linguagem adequada para configuração, automação (*scripting*) e prototipagem rápida.

Há uma variedade de motores para jogos disponíveis comercialmente, como por exemplo: CryEngine [CRY], UnrealEngine 2 e 3 [Gama], Torque Engine [Gamb], Reality Engine [Stu] e motores de Quake, Quake 2 e 3 [Sof]. Há também os de arquitetura aberta que podem ser obtidos gratuitamente na internet. Entre eles, pode-se citar o OGRE 3D [ead], o CrystalSpace [eaa], Irrlicht [eab, Geb], o Nebula 2 Device [Lab], OpenSceneGraph [Sul] e o G3D [McG]. Além disso, há vários livros que descrevem detalhadamente como construir motores, inclusive com código fonte [Ebe00, Lam03, ZD04, Dal03]. Na internet existe um portal que reúne informações sobre motores 3D para jogos e motores de renderização [Dev].

Objetivando diminuir a quantidade de dados processada a cada quadro de animação, utiliza-se a ajuda de algoritmos como quadtrees, muito utilizadas em terrenos; octrees; BSP-trees, portais, e PVS. Destas, quadtrees e octrees já são técnicas de particionamento do espaço amplamente conhecidas no ramo da computação gráfica [ZD04]. Portais começaram sua popularidade por volta dos anos 90 com o jogo *Duke Nukem 3D*, num ambiente dividido em células (geralmente *indoor*) pode-se marcar os pontos de acesso à células adjacentes, permitindo calcular quais os conjuntos potencialmente visíveis naquele momento em tempo de execução [Lam03]. A idéia dos PVS é de pré-calcular as células ou folhas (caso utilizando uma estrutura de BSP-tree, por exemplo) que podem ser visíveis umas das outras, deste modo, temos pré-computadas as possibilidades de conjuntos visíveis antes de iniciar o jogo [ZD04]. Para organização de cenas tridimensionais, a tecnica mais utilizada é o grafo de cena ou *scene graph* [Ebe00, Dal03]. Segundo [Ebe00], podemos ter uma representação hierárquica da cena

de acordo com a localização espacial de seus componentes, onde esta cena é armazenada em uma árvore ou tipo de grafo direcionado acíclico. O resultado destes agrupamentos de objetos é chamado de grafo de cena.

Os algoritmos propostos servem também para outras finalidades diferentes de *culling* e organização espacial da cena. Por exemplo, pode-se armazenar atributos e propriedades dos objetos de um grafo de cena em seus nós hierárquicamente; BSP-trees podem ajudar no cálculo de detecção de colisão entre os próprios atores, inclusive com o mundo 3D. Detecção de colisão é uma parte importante principalmente para o motor de física, pois é necessária para cálculo de quando e como dois corpos se chocarão, para só depois calcular qual o efeito desta colisão, se houver. Há uma grande variedade de referências à motores de física de corpos rígidos, comerciais e de arquitetura aberta, entre os quais pode-se citar o Open Dynamics Engine (ODE) [eac] e o PhysX [Cor08a].

1.7 Organização do Texto

O restante do texto é organizado em 5 capítulos resumidos a seguir:

Capítulo 2 - Simulação Dinâmica de Corpos Rígidos

No capítulo 2 tratamos de animação e simulação dinâmica de corpos rígidos e do componente do sistema de animação objeto desta dissertação responsável pela simulação dinâmica de corpos rígidos, o motor de física. Em primeiro lugar, apresentamos os conceitos básicos relativos à simulação dinâmica dos corpos rígidos. Em seguida, discutimos as principais restrições existentes em uma simulação deste tipo: restrições de contato e restrições de junção. Modelados individualmente os problemas das restrições, chegamos a um sistema conhecido como PCL; apresentamos então uma maneira de unificar ambos os tipos de restrições em um único PCL, inclusive como modelar as matrizes, chamadas Jacobianas, que podem representar matematicamente tais restrições.

Capítulo 3 - Especificação de uma Simulação

Neste capítulo apresentamos a sequência de passos envolvidos na inicialização, execução e obtenção dos resultados de uma simulação dinâmica. Apresentamos também, as principais classes de objetos envolvidos na arquitetura do motor de física e seus mais importantes membros. Ao descrever os passos executados, mostramos alguns importantes métodos da API do motor de física, necessários para comunicação com o motor 3D. Detalhamos os algorítmos responsáveis pela resolução do PCL, integração e atualização dos estados dos objetos do motor 3D após cada iteração da simulação.

Capítulo 4 - Implementação do Ambiente

Abordamos neste capítulo a arquitetura, componentes e classes responsáveis pelo funcionamento do motor 3D. Em seguida, mostramos os componentes de uma cena a ser animada, representados por conjuntos de classes de objetos decritos em linguagem `C#`; adicionalmente, abordamos os conceitos necessários para a definição das entidades responsáveis pelo fluxo e controle de uma animação: seqüenciadores e eventos. Apresentamos como dá-se o fluxo de dados dentro do laço principal do motor 3D, qual a sequência de passos de execução, e quais políticas de sincronização foram necessárias dentro deste laço. Em seguida, mostramos como foi realizada a modelagem das principais classes do motor 3D. São introduzidas, no decorrer do capítulo as novas expressões da linguagem LA, estendida neste projeto.

Capítulo 5 - Exemplos

Neste capítulo, apresentamos alguns dos principais exemplos desenvolvidos para teste e demonstração das funcionalidades do motor 3D, com ênfase ao motor de física. São apresentados e descritos diversos exemplos, ilustrados os resultados obtidos, e ao fim de cada exemplo, comparamos seu desempenho com alguns dos motores de física existentes. Por último, descrevemos os códigos usados para geração das demonstrações.

Capítulo 6 - Conclusão

É apresentado um resumo do desenvolvimento do projeto, bem como dos objetivos e contribuições atingidas. Ainda, descrevemos as idéias de possíveis trabalhos que possam ser desenvolvidos tomando como base este projeto.

CAPÍTULO 2

Simulação Dinâmica de Corpos Rígidos

2.1 Introdução

O campo da animação e simulação baseada em física pode ser dividido em dois grandes grupos: cinemática e dinâmica. A cinemática estuda o movimento dos corpos sem considerar a massa e as forças que neles atuam, enquanto que na dinâmica o movimento é resultante da massa e forças atuantes nos corpos. Neste trabalho tratamos de animação e simulação dinâmica de corpos rígidos. Corpos rígidos podem ser classificados de várias maneiras. Um corpo rígido *discreto* é um sistema de $n_p > 0$ partículas no qual a distância relativa entre duas partículas quaisquer não varia ao longo do tempo, não obstante a resultante de forças atuando no sistema. Um corpo rígido *continuo* é um sólido indeformável com $n_p \to \infty$ partículas, delimitadas por uma superfície fechada que define o contorno de uma região do espaço de volume V. A simulação dinâmica de uma cena constituída de corpos rígidos em um intervalo $[t_0, t_f]$ consiste em, conhecidas a posição e velocidade de cada corpo no instante t_0, determinar sua posição e velocidade no instante $t_0 < t \leq t_f$ em função da resultante das forças atuantes no corpo em t. O componente do sistema de animação objeto desta dissertação responsável pela simulação dinâmica de corpos rígidos é o *motor de física*. O resumo de dinâmica de corpos rígidos apresentados neste capítulo foi baseado em manuscrito redigido pelo Prof. Paulo A. Pagliosa.

Começamos introduzindo os conceitos básicos da dinâmica dos corpos rígidos; para tal é necessário introduzir primeiro como se dá o movimento dinâmico em partículas. A partir daí, adicionamos restrições às partículas, o que em conjunto podem definir um corpo rígido. Mostramos então como chegar às equações de movimento dos corpos rígidos, incluindo restrições de movimento. Descrevemos também como introduzir impacto e atrito às equações. Ao final, mostramos como unificar restrições de junção e contato dos corpos rígidos para solucioná-las em um único PCL.

2.2 Conceitos Básicos

Seja uma partícula de massa m localizada, em um instante de tempo t, em um ponto cuja posição no espaço é definida pelo vetor $\mathbf{x}= \mathbf{x}(t)$. Assume-se que as coordenadas de $\mathbf{x}$ são tomadas em relação a um sistema inercial de coordenadas Cartesianas com origem em um ponto $\mathcal{O}$, embora qualquer outro sistema de coordenadas (esféricas, cilíndricas, etc.) possa ser usado. Este sistema será chamado *sistema global* de coordenadas. A velocidade da partícula em relação ao sistema global é

$$\mathbf{v}(t) = \dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt} \tag{2.1}$$

e sua aceleração, derivando $\vec{v}$ também em relação ao tempo t:

$$\mathbf{a}(t) = \dot{\mathbf{v}} = \frac{d\mathbf{v}}{dt} = \ddot{\mathbf{x}} = \frac{d^2\mathbf{x}}{dt^2}. \tag{2.2}$$

A posição e a velocidade da partícula definem o estado da partícula em um instante :

$$\mathbf{S}(t) = \begin{bmatrix} \mathbf{x}(t) \\ \mathbf{v}(t) \end{bmatrix}. \tag{2.3}$$

O momento linear da partícula é definido como

$$\mathbf{p}(t) = m\mathbf{v}. \tag{2.4}$$

Seja $\mathbf{F}$=$\mathbf{F}$(t) a resultante das forças (gravidade, atrito, etc.) que atuam sobre a partícula em um instante de tempo t. A *segunda lei de Newton* afirma que o movimento da partícula é governado pela equação diferencial

$$\mathbf{F}(t) = \dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = \frac{d}{dt}(m\mathbf{v}). \tag{2.5}$$

Se a massa da partícula é constante:

$$\mathbf{F} = m\frac{d\mathbf{v}}{dt} = m\mathbf{v} = m\mathbf{a}. \tag{2.6}$$

Se não houver restrições sobre o movimento da partícula, esta possui 3 *graus de liberdade*, uma vez que o movimento no espaço pode ser expresso como uma combinação de translações nas direções de cada um dos três eixos de um sistema de coordenadas Cartesianas. O *momento angular* da partícula em relação à origem $\mathcal{O}$ do sistema global é definido como

$$\mathbf{L}(t) = \mathbf{x} \times \mathbf{p} = \mathbf{x} \times m\mathbf{v}. \tag{2.7}$$

O *momento* ou *torque* $\boldsymbol{\tau}$ da resultante de forças , em relação à origem $\mathcal{O}$ do sistema global, aplicado à partícula é

$$\boldsymbol{\tau}(t) = \mathbf{x} \times \mathbf{F}. \tag{2.8}$$

O torque nada mais é do que um momento de um sistema de forças que tende a causar rotação. Da mesma forma que, de acordo com a (2.5), a taxa de variação do momento linear ao longo do tempo é igual à resultante $\mathbf{F}$ das forças sobre a partícula, a taxa de variação do momento angular ao longo do tempo é igual ao momento de $\mathbf{F}$ aplicado à partícula:

$$\dot{\mathbf{L}} = \frac{d\mathbf{L}}{dt} = \frac{d}{dt}(\mathbf{x} \times \mathbf{p}) = \mathbf{x} \times \frac{d\mathbf{P}}{dt} + \frac{d\mathbf{x}}{dt} \times \mathbf{P} = \mathbf{x} \times \mathbf{F} = \boldsymbol{\tau}. \tag{2.9}$$

Seja agora um sistema de n partículas. A força total atuando sobre a i-ésima partícula é a soma de todas as forças externas $\mathbf{F}_i^e$ mais a soma das (n-1) forças internas $\mathbf{F}_{ji}$ exercidas pelas

demais partículas do sistema (naturalmente $\mathbf{F}_{ii} = 0$). A equação de movimento da partícula é

$$\frac{d\mathbf{p}_i}{dt} = m_i \mathbf{v}_i = \mathbf{F}_i^e + \sum_j \mathbf{F}_{ji}, \tag{2.10}$$

onde $\mathbf{p}_i$, m_i e $\mathbf{v}_i$ são o momento linear, a massa e a velocidade da partícula, respectivamente. Será assumido que $\mathbf{F}_{ji}$ satisfaz a terceira lei de Newton, ou seja, que as forças que duas partículas exercem uma sobre a outra são iguais e opostas. Somando-se as equações de movimento de todas as partículas do sistema obtém-se

$$\frac{d^2}{dt^2} \sum_i m_i \mathbf{r}_i = \sum_i \mathbf{F}_i^e + \sum_{i,j} \mathbf{F}_{ji}. \tag{2.11}$$

O primeiro termo do lado direito é igual à força externa total $\mathbf{F}$ sobre o sistema. O segundo termo anula-se, visto que $\mathbf{F}_{ij} + \mathbf{F}_{ji} = 0$. Para reduzir o termo do lado esquerdo, define-se um vetor posição $\mathbf{X}$ igual à média das posições das partículas, ponderada em proporção a suas massas:

$$\mathbf{X}(t) = \frac{\sum m_i \mathbf{x}_i}{\sum m_i} = \frac{\sum m_i \mathbf{x}_i}{M}, \tag{2.12}$$

onde M é a massa total. O vetor $\mathbf{X}$ define um ponto $\mathcal{C}$ chamado *centro de massa* do sistema. Com esta definição, a (2.10) reduz-se a

$$M \frac{d^2 \mathbf{X}}{dt^2} = \sum_i \mathbf{F}_i^e = \mathbf{F}, \tag{2.13}$$

a qual afirma que o centro de massa se move como se a força externa total estivesse atuando na massa total do sistema concentrada no centro de massa. Um sistema com n partículas movendo-se sem restrições possui $3n$ graus de liberdade. O momento linear total do sistema,

$$\mathbf{P}(t) = \sum_i m_i \frac{d\mathbf{x}_i}{dt} = m \frac{d\mathbf{X}}{dt} = m\mathbf{V}, \tag{2.14}$$

é a massa total vezes a velocidade $\mathbf{V} = \dot{\mathbf{X}}$ do centro de massa. A taxa de variação do momento linear total, $\dot{\mathbf{P}} = \mathbf{F}$, é igual à força externa total. Como conseqüência, se a força externa total é nula, o momento linear total de um sistema de partículas é conservado. O momento angular total em relação ao ponto $\mathcal{O}$ é

$$\mathbf{L}(t) = \sum_i \mathbf{x}_i \times \mathbf{p}_i = \mathbf{X} \times M\mathbf{V} + \sum_i \mathbf{x}_i' \times \mathbf{p}_i, \tag{2.15}$$

onde $\mathbf{x}_i' = \mathbf{x}_i - \mathbf{X}$ é o vetor do centro da massa à posição da i-ésima partícula e $\mathbf{p}_i = m_i \mathbf{v}_i$ é o momento linear da i-ésima partícula em relação ao centro de massa. Ou seja, o momento angular total é o momento angular do sistema concentrado no centro de massa mais o momento angular do movimento em torno do centro de massa. A taxa de variação do momento angular total,

$$\dot{\mathbf{L}} = \boldsymbol{\tau} = \sum_i \mathbf{x}_i \times \mathbf{F}_i^e, \tag{2.16}$$

é igual ao torque da força externa total em relação a $\mathcal{O}$. Para sistemas contínuos, isto é, com $n \to \infty$ partículas em um volume V, os somatórios nas expressões acima tornam-se integrais sobre V. Neste caso, a massa do sistema é definida por uma função de densidade $\rho = \rho(\mathbf{x}(t))$, tal que uma partícula na posição $\mathbf{x}$ concentra uma massa $dm = \rho dV$. Em particular, a posição do centro de massa $\mathcal{C}$ fica definida como

$$\mathbf{X}(t) = \frac{\int_V \mathbf{x}\, dm}{\int_V dm} = \frac{\int_V \rho \mathbf{x}\, dV}{M}, \tag{2.17}$$

onde $M = \int_V \rho dV$ é a massa total do sistema.

A simulação dinâmica de um sistema de partículas consiste, conhecidos o *estado inicial* - isto é, a posição e velocidade de cada partícula - e as forças atuantes em cada partícula, em determinar a evolução do estado do sistema ao longo do tempo.

2.3 Restrições de Movimentos

A configuração de um sistema de n partículas em um instante de tempo t é o conjunto da posição $\mathbf{x}_i$, $1 \leq i \leq n$ de todas as partículas do sistema em t. O *espaço de configurações* do sistema é o conjunto de todas as suas possíveis configurações. Em uma simulação, contudo, há *restrições* que, impostas ao movimento de um número de partículas, impedem que certas configurações sejam *válidas*, isto é, nem toda configuração do sistema pode ser atingida, mesmo com tempo e energia suficientes para tal. São consideradas neste texto somente restrições de movimento que podem ser descritas por uma condição expressa em função das posições das partículas do sistema e do tempo (ou seja, independem das velocidades e/ou acelerações das partículas).

Para ilustrar, seja uma partícula restrita a mover-se sobre uma superfície S em $\mathbb{R}^3$, conforme a Figura 2.1, a qual é representada implicitamente por uma função escalar C. Então, a restrição sobre a partícula pode ser escrita como

$$C(\mathbf{x}(t)) = 0. \tag{2.18}$$

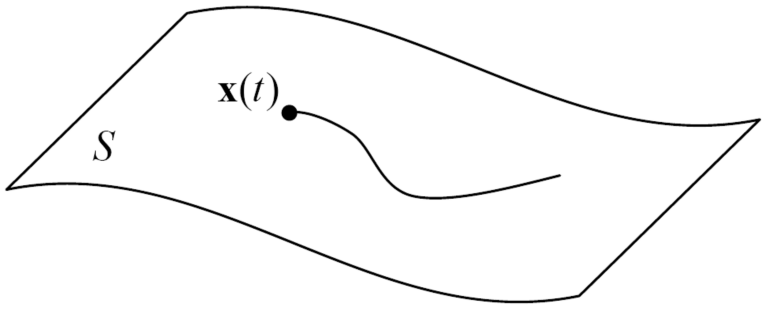


Figura 2.1: Partícula restrita sobre uma superfíce S.

Suponha, agora, que a restrição dada pela Equação (2.18) seja relaxada tal que a partícula possa se mover não somente sobre a superfície S, mas também acima desta, conforme a Figura 2.2. Esta condição pode ser escrita como

$$C(\mathbf{x}(t)) \geq 0. \tag{2.19}$$

Uma restrição expressa por uma condição envolvendo uma igualdade como a Equação (2.18) é dita ser *bilateral*, enquanto que uma restrição cuja condição é dada por uma desigualdade como a (2.19) é chamada *unilateral*. O movimento de uma partícula sujeita a uma restrição unilateral engloba o movimento irrestrito ou aquele sujeito à restrição bilateral correspondente. Se em um instante t a partícula satisfaz $C(\mathbf{x}(t)) \geq 0$, então, para algum período de tempo após t, o movimento da partícula pode ser classificado como irrestrito (a partícula está acima da superfície S). Similarmente, o movimento sujeito a uma restrição bilateral é um caso especial do movimento sujeito à restrição unilateral correspondente, uma vez que a restrição $C(\mathbf{x}(t)) = 0$ pode ser substituída pelas restrições $C(\mathbf{x}(t)) \geq 0$ e $-C(\mathbf{x}(t)) \geq 0$.

Usando-se outra nomenclatura, pode-se dizer que uma restrição em um sistema de n partículas cuja condição é definida por uma equação algébrica da forma

$$C(\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n, t) = 0. \tag{2.20}$$

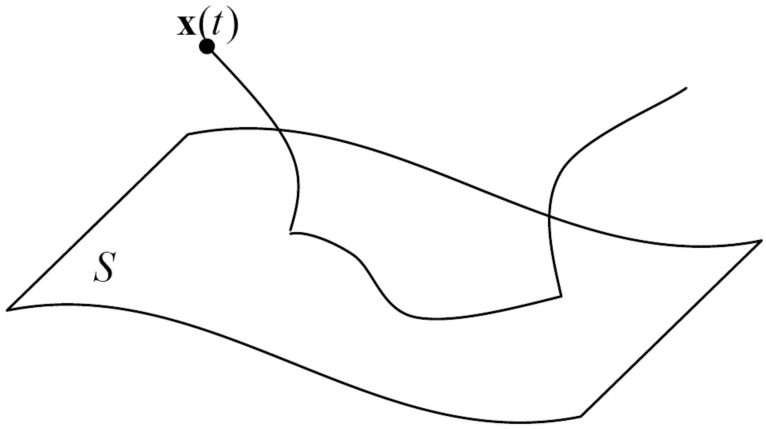


Figura 2.2: Partícula restrita sobre ou acima de uma superfíce S.

é também chamada de *holonômica*, onde a função escalar C representa, em cada instante t, *a superfície da restrição* em $\mathbb{R}^{3n}$. Uma restrição holonômica que não depende explicitamente do tempo, como a (2.18), é chamada *escleronômica*. De modo geral, uma restrição holonômica elimina um grau de liberdade do sistema, sendo o espaço de configurações válidas a interseção de todas as superfícies de restrição.

Restrições impõem duas dificuldades na solução de problemas mecânicos. Primeiro, as coordenadas $\mathbf{x}_i$ não são todas independentes, uma vez que são relacionadas através das condições das restrições; portanto, as equações de movimento das partículas do sistema, (2.10), não são todas independentes. Segundo, as restrições introduzem *forças* no sistema tais que o movimento satisfaça justamente as condições impostas pelas restrições. A fim de determinar a evolução do estado do sistema ao longo do tempo, as forças de restrições devem ser determinadas a cada instante da simulação.

Forças de restrições podem ser determinadas pelo *método das forças de penalidade* [Bar92] ou, mais exatamente, incorporando-se as restrições na equação de movimento do sistema. Para ilustrar, considere novamente o problema da partícula com movimento restrito sobre uma superfície S, Figura 2.1. Seja $\mathbf{F}_c(t)$ a força de restrição que atua sobre a partícula a fim de que esta permaneça em contato com a superfície. A equação de movimento da partícula é

$$m\ddot{\mathbf{x}} = \mathbf{F} + \mathbf{F}_c. \tag{2.21}$$

Se $\mathbf{x}$ é uma posição válida, isto é, que satisfaz a restrição dada pela (2.18), então as velocidades válidas são todas aquelas que satisfazem

$$\dot{C}(\mathbf{x}(t)) = \frac{\delta C}{\delta \mathbf{x}}.\dot{\mathbf{x}}(t) = 0. \tag{2.22}$$

A grandeza $\dfrac{\delta C}{\delta \mathbf{x}}$ (neste exemplo, o vetor gradiente, isto é, o vetor na direção normal à superfície S no ponto $\mathbf{x}$) é chamada *Jacobiano* de C e será denotado por $\mathbf{J}$:

$$\frac{\delta C}{\delta \mathbf{x}} = \mathbf{J} = \begin{bmatrix} \frac{\delta C}{\delta x} \\ \frac{\delta C}{\delta y} \\ \frac{\delta C}{\delta z} \end{bmatrix}. \tag{2.23}$$

As acelerações válidas, por sua vez, são todas aquelas que satisfazem

$$\ddot{C}(\mathbf{x}(t)) = \mathbf{J}.\ddot{\mathbf{x}}(t) + \dot{\mathbf{J}}.\dot{\mathbf{x}}(t) = 0, \tag{2.24}$$

onde $\dot{\mathbf{J}}$ é a matriz de derivadas parciais de segunda ordem de $\ddot{C}$ (isto é, a curvatura da superfície) em $\mathbf{x}$. Se C é a medida do deslocamento da partícula normal a S e $\dot{C}$ é a medida da velocidade da partícula normal a S , então $\ddot{C}$ é a medida da aceleração da partícula normal a S. A força de restrição é escolhida tal que $\ddot{C}$ é sempre zero. É assumido que a restrição é inicialmente satisfeita, isto é, $C(\mathbf{x}(0)) = 0$, e também que $\dot{C}(\mathbf{x}(0)) = 0$. Se C e $\dot{C}$ são nulas para t=0, então permanecerão nulas enquanto $\ddot{C}$ for zero.

Substituindo-se a aceleração da (2.21) na (2.24) obtém-se:

$$\mathbf{J}.\frac{\mathbf{F}+\mathbf{F}_c}{m} + \dot{\mathbf{J}}.\dot{\mathbf{x}} = 0. \tag{2.25}$$

Rearranjando os termos vem:

$$\mathbf{J}.\frac{\mathbf{F}_c}{m} = -\dot{\mathbf{J}}.\dot{\mathbf{x}} - \mathbf{J}.\frac{\mathbf{F}}{m} = 0. \tag{2.26}$$

Para calcular a força de restrição estabelece-se como condição adicional que nunca adicione ou remova energia do sistema. A energia cinética é

$$T = \frac{m}{2}\dot{\mathbf{x}}.\dot{\mathbf{x}}, \tag{2.27}$$

e sua derivada no tempo,

$$\dot{T} = m\ddot{\mathbf{x}}.\dot{\mathbf{x}} = \mathbf{F}\dot{\mathbf{x}} + \mathbf{F}_c.\dot{\mathbf{x}}, \tag{2.28}$$

é a taxa do *trabalho* realizado pela força externa $\mathbf{F}$ e de restrição $\mathbf{F}_c$ sobre a partícula. Se, como requerido, a restrição não altera a energia do sistema (como no caso da partícula mover-se sobre a superfície S sem atrito), o último termo da (2.28) deve ser zero, ou seja, o trabalho realizado pela força de restrição sobre a partícula deve ser nulo. Esta condição deve valer para *toda* velocidade válida, isto é, toda velocidade que satisfaz a (2.22):

$$\mathbf{F}_c\dot{\mathbf{x}} = 0, \quad \forall\dot{\mathbf{x}} \mid \mathbf{J}.\dot{\mathbf{x}} = 0. \tag{2.29}$$

(De modo geral, a condição imposta sobre $\mathbf{F}_c$ é conhecida como princípio dos trabalhos virtuais [RS04].) A (2.29) implica que $\mathbf{F}_c$ deve atuar sobre a partícula em uma direção normal à superfície S em $\mathbf{x}$ - isto é, na direção do gradiente de C - a fim de que a partícula permaneça sobre S, Figura 3. Portanto, a força da restrição pode ser escrita como

$$\mathbf{F}_c = \lambda\mathbf{J}, \tag{2.30}$$

onde λ, chamado de *multiplicador de Lagrange*, é um escalar a ser determinado. O valor de λ pode ser positivo ou negativo. A força de restrição $\mathbf{F}_c$ se opõe à força externa $\mathbf{F}$; na Figura 2.3(a), $\mathbf{F}_c$ atua na mesma direção da normal e $\lambda > 0$; na Figura 2.3(b), atua na direção oposta à normal e $\lambda < 0$. Substitindo-se a (2.30) na (2.26) obtém-se:

$$\lambda = \frac{m\dot{\mathbf{J}}.\dot{\mathbf{x}} + \mathbf{J}.\mathbf{F}}{\mathbf{J}.\mathbf{J}} \tag{2.31}$$

Determinado λ, calculam-se a força de restrição e a aceleração, (2.30) e (2.21), respectivamente. A evolução do estado da partícula é computada através da integração numérica da equação de movimento.

Considere agora o problema da partícula cujo movimento é sujeito à restrição unilateral da (2.19). Suponha que inicialmente a partícula esteja acima da superfície S e que não entre em contato com a superfície antes do tempo t_c. Durante o intervalo de tempo $[0, t_c)$ a restrição é dita *inativa* e o movimento da partícula satisfaz

$$m\ddot{\mathbf{x}} = \mathbf{F}. \tag{2.32}$$

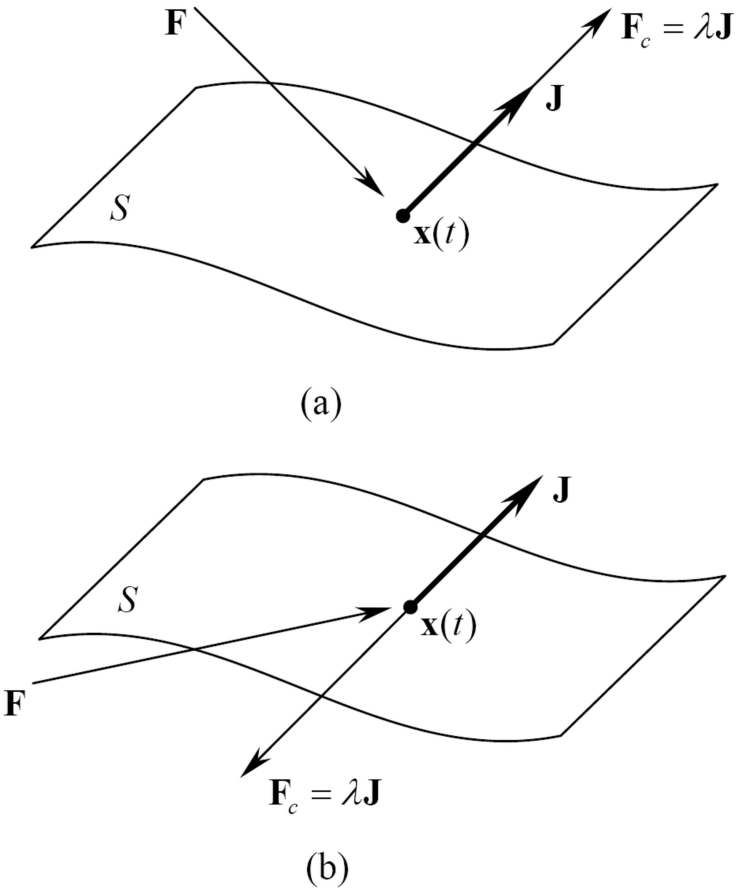


Figura 2.3: Forças no problema de restrição bilateral.

Depois que a partícula entra em contato com a superfície no tempo t_c , a restrição torna-se ativa e uma força de restrição deve atuar a fim de impedir que a partícula se mova abaixo da superfície. Assume-se que o gradiente $\mathbf{J} = \delta C/\delta\mathbf{x}$ aponte para a região do espaço "acima" de S , isto é, a região na qual a partícula é permitida mover-se quando não em contato com a superfície, Figura 2.4. Como no caso da restrição bilateral, a força de restrição $\mathbf{F}_c$ é suposta não realizar trabalho sobre a partícula, ou seja, deve atuar paralela à normal de S em $\mathbf{x}$; portanto, a força de restrição pode ser escrita igualmente como $\mathbf{F}_c = \lambda\mathbf{J}$. Diferente do problema anterior, contudo, a força de restrição deve atuar sempre na direção de $\mathbf{J}$, o que significa que $\lambda \geq 0$. Na Figura 2.4(a) a força de restrição $\mathbf{F}_c$ se opõe à força externa $\mathbf{F}$ ($\lambda > 0$), exatamente como ocorre na Figura 2.3(a). Para o caso da Figura 2.4(b), porém, *nenhuma* força de restrição atua ($\lambda = 0$) e a força externa acelera a partícula acima da superfície S. Se λ pudesse assumir um valor negativo, como ocorre na Figura 2.3(b), então a partícula, uma vez em contato com a superfície, mover-se-ia sobre esta como no problema da restrição bilateral.

O contato da partícula com a superfície S no instante t_c pode ser de dois tipos: *colisão* ou *repouso*, dependendo da velocidade da partícula normal à superfície em $\mathbf{x}(t_c)$, a qual é dada por $\dot{C}(\mathbf{x}(t_c))$. Se

$$\dot{C}(\mathbf{x}(t_c)) = \frac{\delta C}{\delta \mathbf{x}}.\dot{\mathbf{x}}(t_c) = \mathbf{J}.\mathbf{v}(t_c) < 0, \tag{2.33}$$

então a partícula *colide* com a superfície (se $\dot{C}(\mathbf{x}(t_c)) > 0$ a restrição torna-se inativa e a partícula deixa de estar em contato, movendo-se acima da superfície em $t > t_c$; o caso $\dot{C}(\mathbf{x}(t_c)) = 0$ é discutido posteriormente). Para evitar que a partícula se mova abaixo da superfície, a velocidade $\mathbf{v}$ deve passar por uma descontinuidade em t_c. Para tal, uma força de

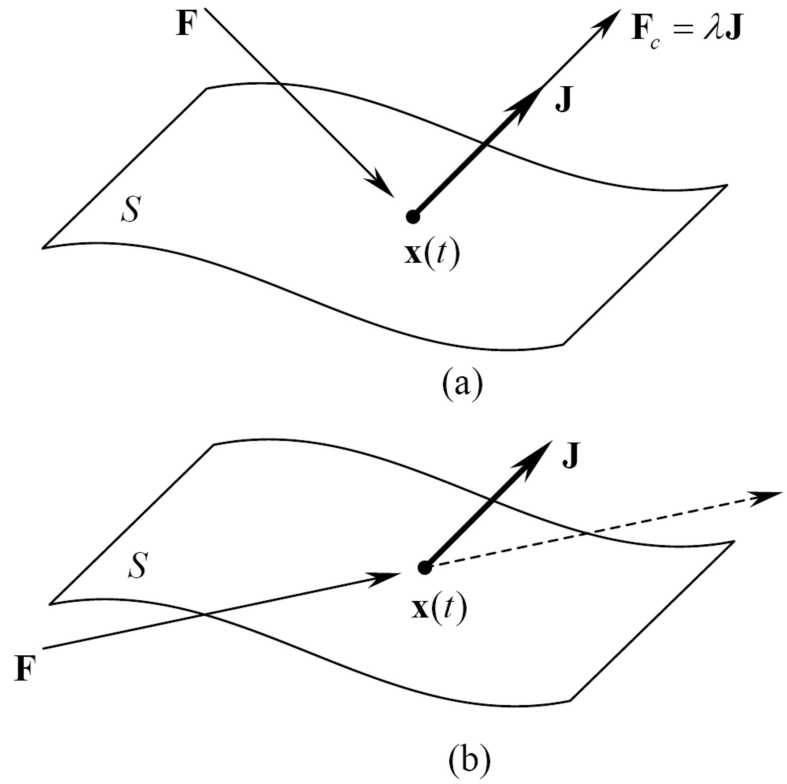


Figura 2.4: Forças no problema de restrição unilateral.

restrição *impulsiva* deve ser aplicada à partícula na direção normal a S em $\mathbf{x}(t_c)$. O impulso de uma força $\mathbf{F}$ atuando em um tempo infinitesimal Δt é definido como

$$\mathbf{F}_I = \int_{\Delta t} \mathbf{F} dt \tag{2.34}$$

e tem dimensões de força vezes tempo ou, equivalentemente, massa vezes velocidade. Aplicado à partícula, o impulso causa uma variação instantânea do momento linear $\Delta\mathbf{p} = \mathbf{F}_I$ e, conseqüêntemente, da velocidade $\Delta\mathbf{v} = \mathbf{F}_I/m$. Seja, então, o impulso $\lambda_I\mathbf{J}$ a ser aplicado à partícula em $\mathbf{x}(t_c)$, onde λ_I é um escalar positivo. A velocidade $\mathbf{v}^+(t_c)$ da partícula após a aplicação do impulso é

$$\mathbf{v}^+(t_c) = \mathbf{v}(t_c) + \frac{\lambda_I\mathbf{J}}{m}. \tag{2.35}$$

(Esta equação pode ser formalmente derivada definindo-se $\mathbf{v}^+(t_c) = \lim_{\Delta t\to 0} \mathbf{v}(t_c + \Delta t)$ quando uma força $\lambda\mathbf{J}/m$ atua na partícula de t_c até $t_c + \Delta t_c$, com $\lim_{\Delta t\to 0} \lambda\Delta t = \lambda_I$.) O escalar λ_I é determinado através da seguinte lei empírica para colisões:

$$v_n^+ = -\varepsilon v_n^-. \tag{2.36}$$

Na equação acima, v_n^+ e v_n^- denotam, respectivamente, a velocidade da partícula normal à superfície imediatamente após e imediatamente antes a aplicação da força de restrição impulsiva em $\mathbf{x}(t_c)$. O escalar $0 \leq \varepsilon \leq 1$ é chamado *coeficiente de restituição* da colisão e depende das propriedades materiais da partícula e da superfície. As velocidades normais v_n^+ e v_n^- são

$$v_n^- = \mathbf{J}.\mathbf{v}(t_c) = \dot{C}(\mathbf{x}(t_c)) \quad \text{e} \quad v_n^+ = \mathbf{J}.\mathbf{v}^+(t_c). \tag{2.37}$$

Substituindo as expressões acima na (2.36) obtém-se

$$\mathbf{J}.\mathbf{v}^+(t_c) = -\varepsilon\dot{C}(\mathbf{x}(t_c)). \tag{2.38}$$

Substituindo a (2.35) na (2.38) vem

$$\dot{C}(\mathbf{x}(t_c)) + \frac{\lambda_I \mathbf{J}.\mathbf{J}}{m} = -\varepsilon \dot{C}(\mathbf{x}(t_c)), \tag{2.39}$$

donde o valor de λ_I é

$$\lambda_I = \frac{m(\varepsilon+1)\dot{C}(\mathbf{x}(t_c))}{\mathbf{J}.\mathbf{J}}. \tag{2.40}$$

Se $\varepsilon > 0$, a partícula terá uma velocidade na direção da normal à superfície após a colisão; a restrição volta ao estado inativo e o movimento da partícula é considerado irrestrito até que esta entre em contato com a superfície novamente, Figura 2.2. Se $\varepsilon = 0$, a partícula não terá velocidade normal à superfície após a colisão; a restrição permanece ativa e uma força de restrição (não impulsiva) deverá atuar a fim de que a partícula não se mova abaixo da superfície. Tal força surge, portanto, sempre que

$$C(\mathbf{x}(t_c)) = \dot{C}(\mathbf{x}(t_c)) = 0. \tag{2.41}$$

Este tipo de contato é chamado contato de repouso.

No problema da restrição bilateral, $\dot{C} = 0$ foi mantida escolhendo-se λ tal que $\ddot{C} = 0$. Para o problema da restrição unilateral, $\dot{C} \geq 0$ é mantida escolhendo-se λ tal que $\ddot{C} \geq 0$, com a condição $\lambda \geq 0$. Um limite inferior para o valor de λ é dado pela (2.31):

$$\lambda \geq -\frac{m\dot{\mathbf{J}}.\dot{\mathbf{x}}(t) + \mathbf{J}.\mathbf{F}}{\mathbf{J}.\mathbf{J}}. \tag{2.42}$$

Para determinar o valor de λ impõe-se que este seja nulo sempre que $\ddot{C} > 0$, pois, neste caso, $C(\mathbf{x}(t))$ será uma função crescente e, como conseqüência, a partícula passará a se mover acima da superfície após o tempo T, o que tornará a restrição inativa. Com isso, não haverá força de restrição sobre a partícula e $\lambda = 0$. Esta restrição sobre λ pode ser expressa pela seguinte *condição de complementaridade*:

$$\lambda \ddot{C}(\mathbf{x}(t)) = 0. \tag{2.43}$$

Esta, em conjunto com as condições $\ddot{C} \geq 0$ e $\lambda \geq 0$, garante que $\ddot{C} > 0$ ou $\lambda = 0$ e, ou $\ddot{C} = 0$. Então, se o limite inferior dado pela (2.42) é positivo, o valor de λ é tomado como sendo igual a este limite inferior, o que resulta em $\ddot{C} = 0$; caso contrário, o valor de λ é tomado como sendo zero, o que resulta em $\ddot{C} > 0$. Em ambos os casos, a condição de complementaridade é satisfeita. Note que durante o período de tempo em que $\ddot{C} = 0$, o valor de λ é o mesmo do problema com restrição bilateral, o que restringe a partícula a mover-se em contato (de repouso) sobre a superfície. No problema com restrição unilateral, contudo, o valor de λ pára de decrescer em zero, momento em que a partícula deixa de estar em contato com a superfície. O caso geral de um sistema mecânico constituído de vários sistemas de partículas - especificamente corpos rígidos - sujeitos a várias restrições será tratado na próxima seção.

2.4 Dinâmica de Corpos Rígidos

Um corpo rígido discreto pode ser definido como um sistema de n partículas sujeito às seguintes restrições holonômicas:

$$r_{ij} - c_{ij} = 0, \quad 1 \leq i, j \leq n, \tag{2.44}$$

onde $r_{ij} = \|\mathbf{x}_i \mathbf{x}_j\|$ é a distância entre as partículas i e j e c_{ij} é uma constante positiva. Tais restrições, contudo, não são todas independentes (se fossem, estas eliminariam $n(n-1)/2$

graus de liberdade, número que, para valores grandes de n, excede os $3n$ graus de liberdade do sistema). De fato, para fixar um ponto em um corpo rígido não é necessário especificar sua distância a todos os demais pontos do corpo, mas somente a três outros pontos quaisquer não colineares. O número de graus de liberdade, portanto, não pode ser maior que nove. Estes três pontos de referência também não são, contudo, independentes, mas sujeitos às restrições:

$$r_{12} = c_{12}, \quad r_{23} = c_{23} \quad \text{e} \quad r_{13} = c_{13}, \tag{2.45}$$

o que reduz o número de graus de liberdade para seis.

Embora as equações de movimento tenham sido escritas até aqui em termos de coordenadas Cartesianas, ou seja, de translações a partir do ponto de origem e ao longo dos eixos de um sistema de referência, as coordenadas dos seis graus de liberdade de um corpo rígido não são descritas apenas por translações. A configuração de uma partícula de um corpo rígido será especificada com auxílio de um sistema de coordenadas Cartesianas cuja origem, por simplicidade, é o centro de massa $\mathcal{C}$ do corpo, e cujos eixos têm direções dadas, no instante t, por versores $\mathbf{r}(t) = (r_x, r_y, r_z)$, $\mathbf{s}(t) = (r_s, r_s, r_s)$ e $\mathbf{n}(t) = (r_n, r_n, r_n)$, com coordenadas tomadas em relação ao sistema global. Este sistema é chamado *sistema de massa* do corpo rígido . Três das coordenadas do corpo rígido no tempo t serão as coordenadas globais da posição do centro de massa $\mathbf{X}$(t); as três restantes serão a *orientação* do sistema de massa em relação ao sistema global. Uma das maneiras de se representar a orientação do sistema de massa em um instante t é através de uma matriz de rotação de um ponto do corpo em torno de seu centro de massa:

$$\mathbf{R}(t) = \begin{bmatrix} \mathbf{r}(t) & \mathbf{s}(t) & \mathbf{n}(t) \end{bmatrix} = \begin{bmatrix} r_x & s_x & n_x \\ r_y & s_y & n_y \\ r_z & s_z & n_z \end{bmatrix}, \tag{2.46}$$

onde as coordenadas dos versores $\mathbf{r}$, $\mathbf{s}$ e $\mathbf{n}$ formam as colunas da matriz (apesar de nove elementos, estes não são todos independentes e representam de fato as três coordenadas restantes de orientação do corpo). Uma propriedade importante da matriz de rotação $\mathbf{R}$é a *ortogonalidade*, isto é, $\mathbf{R}^T = \mathbf{R}^{-1}$, o que resulta $\mathbf{R}\mathbf{R}^T = \mathbf{1}$, onde $\mathbf{1}$ é a matriz identidade.

A partir das coordenadas globais do centro de massa e da orientação do sistema de massa, a posição em coordenadas globais de um ponto $\mathcal{P}$ qualquer do corpo em um instante t é

$$\mathbf{x}(t) = \mathbf{X}(t) + \mathbf{R}(t)\mathbf{x}_0, \tag{2.47}$$

onde $\mathbf{x}_0$ são as coordenadas da posição de $\mathcal{P}$ em relação ao sistema de massa, Figura 2.5. A posição do centro de massa $\mathbf{X}$ e a orientação $\mathbf{R}$, as quais definem totalmente a configuração (de qualquer partícula) do corpo em t, são chamadas *variáveis espaciais*, ou *coordenadas generalizadas*, do corpo rígido e denotadas por

$$\boldsymbol{\chi}(t) = \begin{bmatrix} \mathbf{X}(t) \\ \mathbf{R}(t) \end{bmatrix}. \tag{2.48}$$

Uma outra maneira de se representar a orientação de um corpo rígido é através de *quaternions*. Um quaternion $\mathbf{q}$ é uma estrutura algébrica constituída de duas partes: um escalar s e um vetor $\mathbf{a} = (a_x, a_y, a_z)$, ou $\mathbf{q} = [s, \mathbf{a}]$. A multiplicação de dois quaternions $\mathbf{q}_1$ e $\mathbf{q}_2$ é definida como

$$\mathbf{q}_1\mathbf{q}_2 = [s_1, \mathbf{a}_1][s_2, \mathbf{a}_2] = [s_1 s_2 - \mathbf{a}_1 \cdot \mathbf{a}_2, s_1\mathbf{a}_2 + s_2\mathbf{a}_1 + \mathbf{a}_1 \times \mathbf{a}_2]. \tag{2.49}$$

Um quaternion unitário é um quaternion onde $s^2 + a_x^2 + a_y^2 + a_z^2 = 1$. Uma rotação de um ângulo θ em torno de um vetor $\mathbf{w}$ é representada pelo quaternion unitário

$$\mathbf{q} = [s, \mathbf{a}] = [\cos(\theta/2), \text{sen}(\theta/2) \| \mathbf{w} \|]. \tag{2.50}$$

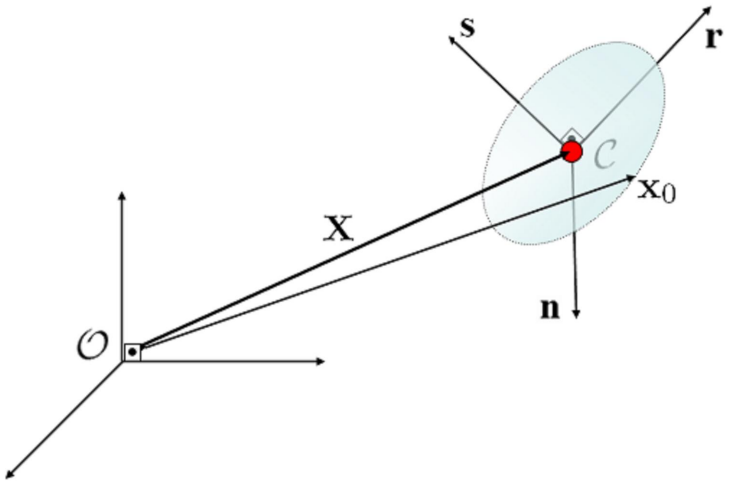


Figura 2.5: Sistema de massa de um corpo rígido.

A rotação inversa $\mathbf{q}_{-1}$ é definida invertendo-se o sinal de s ou de $\mathbf{a}$ na equação acima, mas não de ambos. Para rotacionar um ponto $\mathcal{P}(x, y, z)$ por um quaternion $\mathbf{q}$, escreve-se o ponto $\mathcal{P}$ como o quaternion $\mathbf{p} = [0, (x, y, z)]$ e efetua-se o produto

$$\mathbf{p}' = [0, (x', y', z')] = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}, \tag{2.51}$$

onde $\mathcal{P}'(x', y', z')$ é o ponto $\mathcal{P}$ rotacionado. Uma matriz de rotação 3×3 correspondente à rotação representada pelo quaternion pode ser obtida da expressão acima e escrita como

$$\mathbf{R} = \begin{bmatrix} 1 - 2(a_y^2 + a_z^2) & 2(a_x a_y - s a_z) & 2(a_x a_z + s a_y) \\ 2(a_x a_y + s a_z) & 1 - 2(a_x^2 + a_z^2) & 2(a_y a_z - s a_x) \\ 2(a_x a_z - s a_y) & 2(a_y a_z - s a_x) & 1 - 2(a_x^2 + a_y^2) \end{bmatrix}. \tag{2.52}$$

Em simulação dinâmica é preferível usar quaternions unitários a matrizes de rotação para representar a orientação de corpos rígidos. O principal motivo é que os erros numéricos acumulados nos nove coeficientes de $\mathbf{R}(\mathrm{t})$, à medida que a matriz é atualizada ao longo do tempo da simulação, faz com que esta não seja precisamente uma matriz ortogonal, ou seja, não represente exatamente uma rotação, o que exige uma normalização da matriz. Embora quaternions também tenham que ser normalizados durante a simulação, o custo dessa operação é menor. Usando-se $\mathbf{q}(\mathrm{t})$ como coordenada generalizada no lugar de $\mathbf{R}(\mathrm{t})$, a (2.48) fica escrita como

$$\boldsymbol{\chi}(t) = \begin{bmatrix} \mathbf{X}(t) \\ \mathbf{q}(t) \end{bmatrix}. \tag{2.53}$$

Note que, do mesmo modo que uma matriz de rotação, um quaternion não é uma representação mínima para a orientação de um corpo rígido, uma vez que esta requer três coordenadas e um quaternion tem quatro (não independentes, portanto). Outras formas de representação da orientação de um corpo rígido (por exemplo, ângulos de Euler [Sha05]), não são discutidas neste texto. Assim como a configuração, a velocidade de qualquer partícula de um corpo rígido pode ser igualmente definida em função de componentes de translação e rotação, os quais são denotados por:

$$\mathbf{u}(t) = \begin{bmatrix} \mathbf{V}(t) \\ \boldsymbol{\omega}(t) \end{bmatrix}. \tag{2.54}$$

A velocidade de translação ou *velocidade linear* de um corpo rígido é a velocidade $\mathbf{V}(\mathrm{t})$ de seu centro de massa. A velocidade de rotação ou *velocidade angular* de um ponto de um corpo

rígido em relação a um eixo que passa pelo centro de massa é descrita por um vetor $\boldsymbol{\omega}$(t). A direção de $\boldsymbol{\omega}$(t) define a direção do eixo de rotação e o módulo de $\boldsymbol{\omega}$(t) o ângulo percorrido por um ponto em torno deste eixo no instante t.

Pode-se estabelecer uma relação entre $\dot{\mathbf{R}}$ e a velocidade angular $\boldsymbol{\omega}$, do mesmo modo que há relação entre $\mathbf{X}$ e a velocidade linear $\mathbf{V}$. Para tal, primeiro demonstra-se que a taxa de variação ao longo do tempo de um vetor qualquer $\mathbf{r}$ fixo em um corpo rígido, isto é, que se move junto com este, é igual a [Bar01]

$$\dot{\mathbf{r}} = \boldsymbol{\omega} \times \mathbf{r}. \tag{2.55}$$

Agora, aplica-se a equação acima a cada uma das colunas de $\mathbf{R}$ na (2.46), ou seja, aos versores $\mathbf{r}$, $\mathbf{s}$ e $\mathbf{n}$, obtendo-se

$$\dot{\mathbf{R}} = [\boldsymbol{\omega} \times \mathbf{r}\boldsymbol{\omega} \times \mathbf{s}\boldsymbol{\omega} \times \mathbf{n}]. \tag{2.56}$$

A expressão acima pode ser simplificada notando-se que o produto vetorial entre os vetores $\boldsymbol{\omega}$ e $\mathbf{r}$ pode ser escrito como

$$\boldsymbol{\omega} \times \mathbf{r} = \begin{bmatrix} \omega_y r_z - \omega_z r_y \\ \omega_z r_x - \omega_x r_z \\ \omega_x r_y - \omega_y r_x \end{bmatrix} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \widehat{\boldsymbol{\omega}}\mathbf{r}, \tag{2.57}$$

onde $\widehat{\boldsymbol{\omega}}$ é a matriz anti-simétrica

$$\widehat{\boldsymbol{\omega}} = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{2.58}$$

A relação procurada entre $\dot{\mathbf{R}}$ e $\boldsymbol{\omega}$ é obtida escrevendo-se os produtos vetoriais da (2.56) como a multiplicação da matriz $\widehat{\boldsymbol{\omega}}$ pelos versores $\mathbf{r}$, $\mathbf{s}$ e $\mathbf{n}$, resultando

$$\dot{\mathbf{R}}(t) = \widehat{\boldsymbol{\omega}}(t)\mathbf{R}(t). \tag{2.59}$$

A partir desta relação, pode-se derivar a (2.47) em relação ao tempo e escrever a velocidade de um ponto $\mathcal{P}$ de um corpo rígido em um instante t como sendo

$$\dot{\mathbf{x}}(t) = \mathbf{V}(t) + \widehat{\boldsymbol{\omega}}(t)\mathbf{R}(t)\mathbf{x}_0 = \mathbf{V}(t) + \boldsymbol{\omega}(t) \times (\mathbf{x}(t) - \mathbf{X}(t)). \tag{2.60}$$

Similarmente, se a orientação de um corpo rígido em um instante t é representada por um quaternion unitário $\mathbf{q}$(t), então pode-se deduzir [Ebe04]

$$\dot{\mathbf{q}}(t) = \frac{1}{2}\mathbf{w}(t)\mathbf{q}(t), \tag{2.61}$$

onde $\mathbf{w}(t) = [0, \boldsymbol{\omega}(t)]$. As *velocidades generalizadas* do corpo rígido em t são:

$$\dot{\boldsymbol{\chi}}(t) = \begin{bmatrix} \dot{\mathbf{X}}(t) \\ \dot{\mathbf{q}}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{V}(t) \\ \frac{1}{2}\mathbf{w}(t)\mathbf{q}(t) \end{bmatrix}. \tag{2.62}$$

Na seção anterior, definiu-se o estado de uma partícula, (2.3), como sendo a posição e velocidade da partícula. Por extensão, o estado de um corpo rígido é definido em termos das posições e velocidades generalizadas, (2.54) e (2.62) , respectivamente. No entanto, em simulação dinâmica é mais comum e conveniente caracterizar o estado $\mathbf{S}$(t) de um corpo rígido pelas variáveis espaciais e usar, em lugar das velocidades generalizadas, as velocidades linear e

angular do corpo rígido, as quais são reconhecidas por alguns autores como sendo as próprias velocidades generalizadas. Assim:

$$\mathbf{S}(t) = \begin{bmatrix} \boldsymbol{\chi}(t) \\ \mathbf{u}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{X}(t) \\ \mathbf{q}(t) \\ \mathbf{V}(t) \\ \boldsymbol{\omega}(t) \end{bmatrix}. \tag{2.63}$$

Se a orientação de um corpo rígido em um instante t é dada por um quaternion unitário $\mathbf{q}(t) = [s, (a_x, a_y, a_z)]$, as velocidades generalizadas podem ser relacionadas com as velocidades linear e angular escrevendo-se o lado direito da (2.61) como sendo a matriz $\boldsymbol{\Theta}(t) \in \mathbb{R}^{4\times 3}$ dada por [Erl04]:

$$\boldsymbol{\Theta}(t) = \frac{1}{2} \begin{bmatrix} -a_x & -a_y & -a_z \\ s & a_z & -a_y \\ -a_z & s & a_x \\ a_y & -a_x & s \end{bmatrix}, \tag{2.64}$$

A (2.62) pode então ser escrita como

$$\dot{\boldsymbol{\chi}}(t) = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Theta}(t) \end{bmatrix} \begin{bmatrix} \mathbf{V}(t) \\ \boldsymbol{\omega}(t) \end{bmatrix} = \mathbf{D}(t)\mathbf{u}(t), \tag{2.65}$$

onde

$$\mathbf{D}(t) = \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{\Theta}(t) \end{bmatrix}. \tag{2.66}$$

2.5 Equação de Movimento de Corpos Rígidos

Da mesma forma que a (2.14) estabelece uma correspondência entre o momento linear e a velocidade linear, pode-se associar o momento angular em relação ao centro de massa $\mathcal{C}$ e a velocidade angular de um corpo rígido através da seguinte transformação linear [Gol80]:

$$\mathbf{L}(t) = \mathbf{I}(t)\ \boldsymbol{\omega}(t), \tag{2.67}$$

onde $\mathbf{I}$, o *tensor de inércia* do corpo rígido, descreve como a massa do corpo é distribuída em relação ao centro de massa $\mathcal{C}$. O tensor de inércia é representado por uma matriz simétrica

$$\mathbf{I} = \begin{bmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{bmatrix}, \tag{2.68}$$

onde

$$\begin{aligned} I_{xx} &= \int_V \rho(\mathbf{x}')({x'_y}^2 + {x'_z}^2)\, dV, \\ I_{yy} &= \int_V \rho(\mathbf{x}')({x'_x}^2 + {x'_z}^2)\, dV, \quad \text{e} \\ I_{zz} &= \int_V \rho(\mathbf{x}')({x'_x}^2 + {x'_y}^2)\, dV. \end{aligned} \tag{2.69}$$

são os momentos de inércia, e

$$\begin{aligned} \mathbf{I}_{xy} = \mathbf{I}_{yx} &= \int_V \rho(\mathbf{x}') x'_x r'_y d\mathbf{V}, \\ \mathbf{I}_{xz} = \mathbf{I}_{zx} &= \int_V \rho(\mathbf{x}') x'_x r'_z d\mathbf{V}, \quad \text{e} \\ \mathbf{I}_{yz} = \mathbf{I}_{zy} &= \int_V \rho(\mathbf{x}') x'_y r'_z d\mathbf{V} \end{aligned} \tag{2.70}$$

são os produtos de inércia e $\mathbf{x}' = \mathbf{x}(t) - \mathbf{X}(t) = (x'_x, x'_y, x'_z)$ é o vetor do centro de massa à partícula na posição $\mathbf{x}$ de um ponto do corpo, em coordenadas globais. Durante uma simulação, contudo, o tensor de inércia de um corpo rígido não é calculado através das integrais acima em cada instante t em que haja variação da orientação do corpo, pois o tempo de processamento para fazê-lo poderia ser proibitivamente custoso. Ao invés disto, o tensor de inércia é calculado, para qualquer orientação $\mathbf{q}(t)$ em termos de integrais computadas em relação ao sistema de massa antes do corpo rígido entrar em cena e, portanto, constantes ao longo da simulação (desde que não mude a densidade ou a geometria do corpo). Seja $\mathbf{I}_0$ este tensor de inércia. Pode-se mostrar que [Bar01]

$$\mathbf{I}(t) = \mathbf{R}(t)\mathbf{I}_0\mathbf{R}(t)^T. \tag{2.71}$$

onde $\mathbf{R}(t)$ é a matriz de rotação correspondente a $\mathbf{q}(t)$.

A taxa de variação do momento linear é igual a força externa total aplicada ao corpo rígido no instante t:

$$\dot{\mathbf{P}}(t) = M\dot{\mathbf{V}}(t) = \mathbf{F}(t), \tag{2.72}$$

onde $\dot{\mathbf{V}}$ é a *aceleração linear* do corpo. A taxa de variação do momento angular é igual ao torque da força externa total em relação ao centro de massa do corpo rígido em t:

$$\dot{\mathbf{L}}(t) = \frac{d}{dt}(\mathbf{I}(t)\boldsymbol{\omega}(t)) = \boldsymbol{\tau}(t). \tag{2.73}$$

Derivando-se em relação ao tempo vem

$$\begin{aligned} \boldsymbol{\tau}(t) &= (\mathbf{I}(t)\dot{\boldsymbol{\omega}}(t)) + (\dot{\mathbf{I}}(t)\boldsymbol{\omega}(t)) \\ &= (\mathbf{I}(t)\dot{\boldsymbol{\omega}}(t)) + (\dot{\mathbf{R}}(t)\mathbf{I}_0\mathbf{R}(t)^T + \mathbf{R}(t)\mathbf{I}_0\dot{\mathbf{R}}(t)^T)\boldsymbol{\omega}(t), \end{aligned} \tag{2.74}$$

onde $\dot{\boldsymbol{\omega}}$ é a aceleração angular do corpo. Da ortogonalidade da matriz de rotação, tem-se que $\dot{\mathbf{R}}^T = -\mathbf{R}^T\widehat{\boldsymbol{\omega}}$. Então, usando-se a (2.59) tem-se

$$\begin{aligned} \boldsymbol{\tau}(t) &= (\mathbf{I}(t)\dot{\boldsymbol{\omega}}(t)) + (\widehat{\boldsymbol{\omega}}(t)\mathbf{I}(t) - \mathbf{I}(t)\widehat{\boldsymbol{\omega}}(t))\boldsymbol{\omega}(t) \\ &= (\mathbf{I}(t)\dot{\boldsymbol{\omega}}(t)) + \boldsymbol{\omega}(t) \times \mathbf{I}(t)\boldsymbol{\omega}(t). \end{aligned} \tag{2.75}$$

Combinando-se as equações (2.72) e (2.75) obtém-se a equação de movimento de um corpo rígido:

$$\begin{bmatrix} \mathbf{1}M & \mathbf{0} \\ \mathbf{0} & \mathbf{I}(t) \end{bmatrix} \begin{bmatrix} \dot{\mathbf{V}}(t) \\ \dot{\boldsymbol{\omega}}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{F}(t) \\ \boldsymbol{\tau}(t) - \boldsymbol{\omega}(t) \times \mathbf{I}(t)\boldsymbol{\omega}(t), \end{bmatrix}, \tag{2.76}$$

ou

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{Q}, \tag{2.77}$$

onde

$$\mathbf{M}(t) = \begin{bmatrix} \mathbf{1}M & \mathbf{0} \\ \mathbf{0} & \mathbf{I}(t) \end{bmatrix} \tag{2.78}$$

é a *matriz de massa generalizada* e

$$\mathbf{Q}(t) = \begin{bmatrix} \mathbf{F}(t) \\ \boldsymbol{\tau}(t) - \boldsymbol{\omega}(t) \times \mathbf{I}(t)\boldsymbol{\omega}(t), \end{bmatrix} \tag{2.79}$$

são as *forças generalizadas* do corpo rígido.

Para uma cena com N corpos rígidos, as velocidades $\mathbf{u}_i$, $1 \leq i \leq \mathsf{N}$, de cada corpo podem ser combinadas em um vetor global $\mathbf{u} \in \mathbb{R}^{6\mathsf{N}}$:

$$\mathbf{u} = \begin{bmatrix} \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_i \\ \vdots \\ \mathbf{u}_\mathsf{N} \end{bmatrix} = \begin{bmatrix} \mathbf{V}_1 \\ \boldsymbol{\omega}_1 \\ \vdots \\ \mathbf{V}_i \\ \boldsymbol{\omega}_i \\ \vdots \\ \mathbf{V}_\mathsf{N} \\ \boldsymbol{\omega}_\mathsf{N} \end{bmatrix} \tag{2.80}$$

e as forças generalizadas $\mathbf{Q}_i$ em um vetor global $\mathbf{Q} \in \mathbb{R}^{6\mathsf{N}}$:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_1 \\ \vdots \\ \mathbf{Q}_i \\ \vdots \\ \mathbf{Q}_\mathsf{N} \end{bmatrix}. \tag{2.81}$$

Analogamente, as matrizes de massa generalizada podem ser combinadas em uma matriz global $\mathbf{M} \in \mathbb{R}^{6\mathsf{N}\times 6\mathsf{N}}$:

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_1 & \dots & \mathbf{0} & \dots & \mathbf{0} \\ \vdots & & \vdots & & \vdots \\ \mathbf{0} & \dots & \mathbf{M}_i & \dots & \mathbf{0} \\ \vdots & & \vdots & & \vdots \\ \mathbf{0} & \dots & \mathbf{0} & \dots & \mathbf{M}_\mathsf{N} \end{bmatrix} \tag{2.82}$$

A equação de movimento global pode então ser escrita como

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{Q}. \tag{2.83}$$

O papel do motor de física é, durante a simulação de uma cena com vários corpos rígidos, conhecidos os estados $\mathbf{S}_i(t)$ de cada corpo no tempo t, determinar os estados $\mathbf{S}_i(t + \Delta t)$ no tempo $t + \Delta t$, onde Δt é o *passo de tempo*. Para um sistema sem restrições de movimento, esta determinação consiste em se obter uma solução para a Equação (80), o que pode ser feito por qualquer método numérico de resolução de equações diferenciais ordinárias (EDO) de primeira ordem. O componente do motor responsável por isto é chamado *solucionador de EDO*. Basicamente, um solucionador de EDO toma como entrada (1) os estados no tempo t de todos os corpos da simulação, armazenados em uma estrutura de dados conveniente, (2) uma função que permita calcular, em t, a derivada

$$\frac{d}{dt}\mathbf{S}(t) = \begin{pmatrix} \mathbf{u}(t) \\ \dot{\mathbf{u}}(t) \end{pmatrix} = \begin{pmatrix} \mathbf{V}(t) \\ \frac{1}{2}\mathbf{w}(t)\mathbf{q}(t) \\ M^{-1}\mathbf{F}(t) \\ \mathbf{I}^{-1}(t)(\boldsymbol{\tau}(t) - \boldsymbol{\omega}(t) \times \mathbf{I}(t)\boldsymbol{\omega}(t)) \end{pmatrix} \tag{2.84}$$

do estado de cada corpo, e (3) os valores de t e Δt, e computa o estado $\mathbf{S}_i(t + \Delta t)$ de cada corpo rígido. Note que todas as grandezas na (2.84) são conhecidas no tempo t, sendo a força $\mathbf{F}$ e o torque $\boldsymbol{\tau}$ em relação ao centro de massa $\mathcal{C}$ de cada corpo rígido fornecido pela aplicação.

2.6 Dinâmica de Corpos Rígidos com Restrições

Em simulação dinâmica de corpos rígidos fundamentalmente são tratados dois tipos de restrições: (1) as impostas por junções entre (normalmente dois) corpos, e (2) as resultantes do contato entre corpos, abordadas na próxima seção. Uma junção entre dois corpos força que o movimento de um seja relativo ao do outro de alguma maneira que depende do tipo de junção. Alguns exemplos são ilustrados na Figura 2.6.

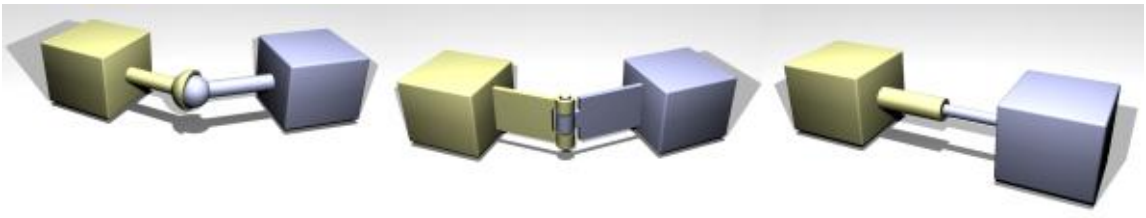


Figura 2.6: Exemplos de junções: esférica, de revolução e cilíndrica.

Uma junção esférica força que dois pontos sobre dois corpos diferentes sejam coincidentes, removendo três graus de liberdade de cada corpo. Uma junção de revolução pode ser usada para representar uma dobradiça entre dois corpos: cinco graus de liberdade de cada corpo são removidos, restando uma rotação que se dá em torno do eixo da dobradiça. Uma junção cilíndrica permite uma translação e uma rotação relativa de dois corpos em relação ao longo de um eixo, removendo quatro graus de liberdade de cada corpo. Se as translações e rotações permitidas por estes tipos de junções não são limitadas, então as restrições correspondentes são holonômicas.

Seja uma cena com $\mathbf{N}$ corpos e com $\mathbf{J}$ junções. Se a k-ésima junção, $1 \leq k \leq \mathbf{J}$, entre dois corpos A_i^k e A_j^k, $1 \leq i, j \leq \mathbf{N}$, remove m^k graus de liberdade dentre os doze possíveis graus de liberdade do par de corpos A_i^k e A_j^k, então esta introduz no sistema um conjunto de m^k restrições holonômicas

$$\mathbf{C}^k(\boldsymbol{\chi}^k) = \begin{bmatrix} \mathbf{C}_1^k(\boldsymbol{\chi}^k) \\ \mathbf{C}_2^k(\boldsymbol{\chi}^k) \\ \vdots \\ \mathbf{C}_{m^k}^k(\boldsymbol{\chi}^k) \end{bmatrix} = \mathbf{0}, \tag{2.85}$$

onde $\boldsymbol{\chi} \in \mathbb{R}^{14}$ é o vetor das coordenadas generalizadas dos corpos A_i^k e A_j^k:

$$\boldsymbol{\chi}^k = \begin{bmatrix} \boldsymbol{\chi}_i^k \\ \boldsymbol{\chi}_j^k \end{bmatrix} = \begin{bmatrix} \mathbf{X}_i^k \\ \mathbf{q}_i^k \\ \mathbf{X}_j^k \\ \mathbf{q}_j^k \end{bmatrix}. \tag{2.86}$$

(De forma geral, a equação de uma restrição holonômica é escrita em função de *todas* as coordenadas generalizadas do sistema, como na (2.20); contudo, nas expressões acima, sem perda de generalidade, são relacionadas apenas as coordenadas generalizadas dos corpos unidos pela junção.) Se $\boldsymbol{\chi}^k$ são coordenadas generalizadas válidas, então as velocidades válidas dos corpos A_i^k e A_j^k devem satisfazer as restrições cinemáticas da junção, as quais

resultam da derivação da (2.85) em relação ao tempo:

$$\begin{aligned}\dot{\mathbf{C}}^k(\boldsymbol{\chi}^k) = \frac{\delta \mathbf{C}^k}{\delta \boldsymbol{\chi}^k}\,\dot{\boldsymbol{\chi}}^k \\ &= \frac{\delta \mathbf{C}^k}{\delta \boldsymbol{\chi}^k}\,\mathbf{D}^k\,\mathbf{u}^k, \\ &= \mathbf{J}_j^k\,\mathbf{u}^k \\ &= \mathbf{0},\end{aligned} \tag{2.87}$$

sendo $\mathbf{u}^k \in \mathbb{R}^{12}$ o vetor das velocidades linear e angular dos corpos A_i^k e A_j^k,

$$\mathbf{u}^k = \begin{bmatrix}\mathbf{u}_i^k \\ \mathbf{u}_j^k\end{bmatrix} = \begin{bmatrix}\mathbf{V}_i^k \\ \boldsymbol{\omega}_i^k \\ \mathbf{V}_j^k \\ \boldsymbol{\omega}_j^k\end{bmatrix}, \tag{2.88}$$

e $\mathbf{D}^k \in \mathbb{R}^{14\times 12}$ a matriz

$$\mathbf{D}^k = \begin{bmatrix}\mathbf{D}_i^k & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_j^k\end{bmatrix} = \begin{bmatrix}\mathbf{1} & & & \\ & \boldsymbol{\Theta}_i^k & & \\ & & \boldsymbol{\Theta}_j^k & \\ & & & \mathbf{1}\end{bmatrix}, \tag{2.89}$$

onde $\boldsymbol{\Theta}$ é definido pela (2.64). As matrizes $\delta\mathbf{C}^k/\delta\boldsymbol{\chi}^k \in \mathbb{R}^{m^k\times 14}$ e $\mathbf{J}_J^k \in \mathbb{R}^{m^k\times 12}$,

$$\mathbf{J}_J^k \;=\; \frac{\delta \mathbf{C}^k}{\delta \boldsymbol{\chi}^k}\,\mathbf{D}^K k, \tag{2.90}$$

são os *Jacobianos* da junção k. A (2.87) é uma extensão da (2.22) e pode ser vista como o vetor de "velocidades normais" às superfícies representadas pelas funções de restrições $\mathbf{C}^k$. As "acelerações normais" são obtidas derivando-se a (2.87) em relação ao tempo:

$$\begin{aligned}\ddot{\mathbf{C}}^k(\boldsymbol{\chi}^k) \;=\; \frac{d^2}{dt^2}\mathbf{J}_J^k\,\mathbf{u}^k \\ &= \mathbf{J}_J^k\dot{\mathbf{u}}^k \;+\; \underbrace{\dot{\mathbf{J}}_J^k\mathbf{u}^k}_{\mathbf{c}^k} \\ &= \mathbf{J}_J^k\dot{\mathbf{u}}^k \;+\; \mathbf{c}^k \\ &= \mathbf{0},\end{aligned} \tag{2.91}$$

onde $\mathbf{u}^k \in \mathbb{R}^{12}$ é o vetor das acelerações linear e angular dos corpos A_i^k e A_j^k:

$$\dot{\mathbf{u}}^k = \begin{bmatrix}\dot{\mathbf{u}}_i^k \\ \dot{\mathbf{u}}_j^k\end{bmatrix} = \begin{bmatrix}\dot{\mathbf{V}}_i^k \\ \dot{\boldsymbol{\omega}}_i^k \\ \dot{\mathbf{V}}_j^k \\ \dot{\boldsymbol{\omega}}_j^k\end{bmatrix}. \tag{2.92}$$

Cada restrição da junção introduz uma força de restrição no sistema. Como no exemplo da partícula sujeita à restrição bilateral da seção anterior, admite-se que não haja atrito e que as forças de restrição generalizadas $\mathbf{Q}_J^k \in \mathbb{R}^{12}$ da junção, aplicadas aos corpos A_i^k e A_j^k, satisfaçam o princípio dos trabalhos virtuais, o que implica em

$$\mathbf{Q}_J^k \;=\; (\mathbf{J}_J^k)^T\boldsymbol{\lambda}_J^k, \tag{2.93}$$

onde $\boldsymbol{\lambda}_J^k \in \mathbb{R}^{m^k}$ é o vetor dos multiplicadores de Lagrange,

$$\boldsymbol{\lambda}_J^k = [\,\boldsymbol{\lambda}_1^k \ldots \boldsymbol{\lambda}_{m^k}^k\,]^T. \tag{2.94}$$

As restrições cinemáticas da (2.87) podem ser escritas como

$$\mathbf{J}_J^k \mathbf{u}^k = \begin{bmatrix} \mathbf{J}_{J_i}^k & \mathbf{J}_{J_j}^k \end{bmatrix} \begin{bmatrix} \mathbf{u}_i^k \\ \mathbf{u}_j^k \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Gamma}_{J_i}^k & \boldsymbol{\Omega}_{J_i}^k & \boldsymbol{\Gamma}_{J_j}^k & \boldsymbol{\Omega}_{J_j}^k \end{bmatrix} \begin{bmatrix} \mathbf{V}_i^k \\ \boldsymbol{\omega}_i^k \\ \mathbf{V}_j^k \\ \boldsymbol{\omega}_j^k \end{bmatrix} = \mathbf{0} \tag{2.95}$$

ou

$$\boldsymbol{\Gamma}_{J_i}^k \mathbf{V}_i^k + \boldsymbol{\Omega}_{J_i}^k \boldsymbol{\omega}_i^k + \boldsymbol{\Gamma}_{J_j}^k \mathbf{V}_j^k + \boldsymbol{\Omega}_{J_j}^k \boldsymbol{\omega}_j^k = \mathbf{0}, \tag{2.96}$$

onde as sub-matrizes $\mathbf{J}_{J_i}^k = [\boldsymbol{\Gamma}_{J_i}^k \quad \boldsymbol{\Omega}_{J_i}^k]$ e $\mathbf{J}_{J_j}^k = [\boldsymbol{\Gamma}_{J_j}^k \quad \boldsymbol{\Omega}_{J_j}^k]$ são as partições do Jacobiano da junção k que multiplicam as velocidades dos corpos A_i^k e A_j^k, respectivamente. As sub-matrizes $\mathbf{J}_{J_i}^k$ e $\mathbf{J}_{J_j}^k$ de cada junção k da cena podem ser organizadas em uma matriz global $\mathbf{J}_J \in \mathbb{R}^{\mathsf{R}\times 6\mathsf{N}}$,

$$\mathbf{J}_j = \begin{bmatrix} \mathbf{J}_{J_1}^1 & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}_{J_N}^1 \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}_{J_1}^r & \cdots & \mathbf{J}_{J_i}^r & \cdots & \mathbf{J}_{J_j}^r & \cdots & \mathbf{J}_{J_N}^r \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}_{J_1}^R & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}_{J_N}^R \end{bmatrix}. \tag{2.97}$$

onde $\mathbf{R} = \sum_{k=1}^{J} m^k$ é o número total de restrições de todas as junções da cena no instante t. Note que a matriz $\mathbf{J}_J$ engloba todas as restrições, independentemente de quais junções estas tenham se originado. A matriz global dos Jacobianos é bastante esparsa, uma vez que para uma restrição r(originária de uma junção qualquer k), a linha r correspondente contém apenas dois elementos não-nulos nas colunas i e j, que são as partições, relativas aos corpos A_i^k e A_j^k, respectivamente, do Jacobiano da junção k. Analogamente, pode-se concatenar em um vetor $\boldsymbol{\lambda}_J \in \mathbb{R}^{\mathsf{R}}$ global os multiplicadores de Lagrange correspondentes a todas as forças generalizadas de restrição devidas às junções,

$$\boldsymbol{\lambda}_J = [\boldsymbol{\lambda}_1^J \, \boldsymbol{\lambda}_r^J \ldots \boldsymbol{\lambda}_R^J]^T. \tag{2.98}$$

Portanto, o vetor global $\mathbf{Q}_J \in \mathbb{R}^{6\mathsf{N}}$ de forças generalizadas de restrição das junções é

$$\mathbf{Q}_J = \mathbf{J}_J^T \boldsymbol{\lambda}_J. \tag{2.99}$$

Adicionando-se este termo à (2.83), a equação de movimento global fica

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{Q} + \mathbf{Q}_J = \mathbf{Q} + \mathbf{J}_J^T \boldsymbol{\lambda}_J. \tag{2.100}$$

Assumindo-se que todas as restrições sejam inicialmente satisfeitas, as forças de restrições são escolhidas tal que a condição estabelecida pela versão global da (2.91),

$$\mathbf{J}_J \dot{\mathbf{u}} = -\dot{\mathbf{J}}_J \mathbf{u} = -\mathbf{c}_J, \tag{2.101}$$

se mantenha ao longo do tempo da simulação. As (2.100) e (2.101) equivalem ao seguinte sistema linear de equações com N+R incógnitas:

$$\begin{bmatrix} \mathbf{M} & -\mathbf{J}_J^T \\ \mathbf{J}_J & \mathbf{0} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{u}} \\ \boldsymbol{\lambda}_J \end{bmatrix} = \begin{bmatrix} \mathbf{Q} \\ -\mathbf{c}_J \end{bmatrix}. \tag{2.102}$$

Para resolver o sistema acima, pode-se reescrever a (2.100) como

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{M}^{-1}(\mathbf{Q} + \mathbf{J}_J^T\boldsymbol{\lambda}_J). \tag{2.103}$$

e substituir o resultado na (2.101), obtendo-se

$$\mathbf{J}_J\mathbf{M}^{-1}\mathbf{J}_J^T\boldsymbol{\lambda}_J = -\mathbf{J}_J\mathbf{M}^{-1}\mathbf{Q} - \mathbf{c}_J, \tag{2.104}$$

donde determina-se primeiro os multiplicadores de Lagrange $\boldsymbol{\lambda}_J$ (compare com a (2.31) na seção anterior) e depois, levando-se na (2.103), as acelerações $\dot{\mathbf{u}}$, as quais podem ser numericamente integradas a fim de se calcular o estado de cada corpo rígido da cena em $t + \Delta t$. Como demonstrado na literatura, pode-se, explorando o fato da matriz dos Jacobianos ser altamente esparsa, resolver a (2.102) em tempo linear (veja, por exemplo, [Bar94b]).

A formulação das equações de movimento vista até aqui é chamada de *formulação baseada em acelerações*, uma vez que envolve, naturalmente, as acelerações de partículas ou de corpos rígidos cuja determinação é necessária para cálculo da evolução do estado do sistema durante a simulação.

2.7 Restrições de Contato

Um contato entre dois corpos rígidos introduz uma restrição unilateral no sistema, isto é, cuja condição é expressa por uma inequação, o que implica que métodos distintos daqueles empregados no tratamento das restrições de junções devem ser considerados. Seja então uma cena com N corpos e com C contatos. Um contato k , $1 \leq k \leq \mathsf{N}$, entre dois corpos A_i^k e A_j^k, $1 \leq i, j \leq \mathsf{N}$, ocorre em um instante t_c quando as posições $\mathbf{x}_i^k(t)$ de uma partícula do contorno de A_i^k e $\mathbf{x}_j^k(t)$ de uma partícula do contorno de A_j^k são coincidentes, ou seja, $\mathbf{x}_c^k(t) = \mathbf{x}_i^k(t) = \mathbf{x}_j^k(t)$ (ou estiverem suficientemente próximas dada uma certa tolerância). Portanto, a força de restrição no ponto de contato $\mathbf{x}_c^k(t)$ deve ser tal que impeça que os corpos se interpenetrem, devendo se anular assim que os corpos não estiverem mais em contato. Assume-se que o motor de física possa contar com um componente responsá-vel pela *detecção de colisões*, o qual, no instante t_c , em que se pressupõe não haver interpenetração entre quaisquer corpos, seja capaz de determinar quantos e quais são os C pontos de contato entre os corpos rígidos da cena.

Fazendo-se uma analogia com o problema do movimento de uma partícula sujeito à restrição unilateral dada pela (2.19), o contorno do corpo A_j^k pode ser considerado como sendo a superfície S e a partícula como sendo a partícula $\mathbf{x}_i^k(t)$ do corpo A_j^k. A inequação expressa que a partícula A_j^k não pode "mover-se abaixo", isto é, não pode penetrar no interior do corpo A_j^k. Em um instante $t < t_c$ a partícula $\mathbf{x}_i^k(t)$ está "acima" do contorno de A_j^k e a restrição está inativa; no instante t_c os corpos entram em contato e a restrição fica ativa.

Seja $\mathbf{N}^k$ a normal ao contorno de A_j^k no ponto de contato $\mathbf{x}_c^k$, supostamente apontada para o exterior do corpo. A distância relativa entre as partículas $\mathbf{x}_i^k(t)$ e $\mathbf{x}_j^k(t)$, medida ao longo da normal $\mathbf{N}^k$ no momento do contato é

$$\mathbf{C}^k(\boldsymbol{\chi}^k) = \mathbf{N}^k(\mathbf{x}_i^k(t_c) - \mathbf{x}_j^k(t_c)) = 0. \tag{2.105}$$

onde $\boldsymbol{\chi}^k$ é o vetor de coordenadas generalizadas dos corpos A_i^k e A_j^k, (2.86). A velocidade relativa normal entre as partículas $\mathbf{x}_i^k(t)$ e $\mathbf{x}_j^k(t)$ no ponto de contato, contudo, não é necessariamente nula, sendo dada pela derivada da (2.105) em t:

$$\dot{\mathbf{C}}^k(\boldsymbol{\chi}^k) = \mathbf{N}^k(\dot{\mathbf{x}}_i^k(t_c) - \dot{\mathbf{x}}_j^k(t_c)). \tag{2.106}$$

As velocidades das partículas são, de acordo com a (2.60), $\dot{\mathbf{x}}_i^k(t_c) = \mathbf{V}_i(t_c) + \widehat{\boldsymbol{\omega}}_i \dot{\mathbf{x}'}_i^k(t_c)$ e $\dot{\mathbf{x}}_j^k(t_c) = \mathbf{V}_j(t_c) + \widehat{\boldsymbol{\omega}}_j \dot{\mathbf{x}'}_j^k(t_c)$
as quais, substituídas na (2.106), resulta em

$$\begin{aligned}
\dot{\mathbf{C}}^k(\boldsymbol{\chi}^k) &= \mathbf{N}^k(\mathbf{V}_i(t_c) + \widehat{\boldsymbol{\omega}}_i(t_c) \times \mathbf{x'}_i^k(t_c) - \mathbf{V}_j(t_c) - \widehat{\boldsymbol{\omega}}_j(t_c) \times \mathbf{x'}_j^k(t_c)) \\
&= \mathbf{N}^k(\mathbf{V}_i(t_c) - \mathbf{x'}_i^k(t_c) \times \widehat{\boldsymbol{\omega}}_i(t_c) - \mathbf{V}_j(t_c) + \mathbf{x'}_j^k(t_c) \times \widehat{\boldsymbol{\omega}}_i(t_c)) \\
&= \mathbf{N}^k.\mathbf{V}_i(t_c) - \mathbf{x'}_i^k(t_c) \times \mathbf{N}^k.\widehat{\boldsymbol{\omega}}_i(t_c) - \mathbf{N}^k.\mathbf{V}_j(t_c) + \mathbf{x'}_j^k(t_c) \times \mathbf{N}^k.\widehat{\boldsymbol{\omega}}_j(t_c)), \\
&= \mathbf{N}^k.\mathbf{V}_i(t_c) - \mathbf{x'}_i^k(t_c) \times \mathbf{N}^k.\widehat{\boldsymbol{\omega}}_i(t_c) - \mathbf{N}^k.\mathbf{V}_j(t_c) + \mathbf{x'}_j^k(t_c) \times \mathbf{N}^k.\widehat{\boldsymbol{\omega}}_j(t_c)) \\
&\quad \begin{bmatrix} \mathbf{N}^k & \mathbf{x'}_i^k(t_c) \times \mathbf{N}^k \end{bmatrix} \begin{bmatrix} \mathbf{V}_i(t_c) \\ \widehat{\boldsymbol{\omega}}_i(t_c) \end{bmatrix} + \begin{bmatrix} -\mathbf{N}^k & -\mathbf{x'}_j^k(t_c) \times \mathbf{N}^k \end{bmatrix} \begin{bmatrix} \mathbf{V}_j(t_c) \\ \widehat{\boldsymbol{\omega}}_j(t_c) \end{bmatrix}
\end{aligned} \tag{2.107}$$

onde $\mathbf{x'}_i^k = \mathbf{x}_c^k - \mathbf{X}_i$ e $\mathbf{x'}_j^k = \mathbf{x}_c^k - \mathbf{X}_j$. Se $\dot{\mathbf{C}}^k > 0$, os corpos estão se afastando e não haverá mais o contato em $\mathbf{x}_i^k$ em $t > t_c$; se $\dot{\mathbf{C}}^k = 0$, então os corpos permanecerão em contato de repouso em $t > t_c$. A condição $\dot{\mathbf{C}}^k < 0$ caracteriza uma contato de *colisão* ou *impacto* entre A_i^k e A_j^k no instante t_c; neste caso, haverá interpenetração dos corpos se as velocidades $\dot{\mathbf{x}}_i^k$ e $\dot{\mathbf{x}}_j^k$ não forem imediatamente modificadas. Esta situação será tratada posteriormente.

Suponha, então, que os corpos estão em contato de repouso. Se não há atrito entre A_i^k e A_j^k, no instante t_c atua em $\dot{\mathbf{x}}_c^k$ uma força de restrição generalizada $\mathbf{Q}_\mathsf{C}^k = \lambda_\mathsf{N}^k \mathbf{J}_\mathsf{C}^{k\,T}$, onde λ_N^k é um escalar a ser determinado e $\mathbf{J}_\mathsf{C}^{k\,T} \in \mathbb{R}^{1\times 12}$ é o Jacobiano do k-ésimo contato. Visto que $\dot{\mathsf{C}}^k = \mathbf{J}_\mathsf{C}^k \mathbf{u}^k$, tem-se, da (2.107), que

$$\mathbf{J}_\mathsf{C}^k = \begin{bmatrix} \mathbf{J}_{\mathsf{N}_i}^k & \mathbf{J}_{\mathsf{N}_j}^k \end{bmatrix} = \begin{bmatrix} \underbrace{\mathbf{N}^k}_{\Gamma_{\mathsf{N}_i}^k} & \underbrace{\mathbf{x}_c^k - \mathbf{X}_i \times \mathbf{N}^k}_{\Omega_{\mathsf{N}_i}^k} & \underbrace{-\mathbf{N}^k}_{\Gamma_{\mathsf{N}_j}^k} & \underbrace{-\mathbf{x}_c^k - \mathbf{X}_j \times \mathbf{N}^k}_{\Omega_{\mathsf{N}_j}^k} \end{bmatrix}, \tag{2.108}$$

a partir da qual pode-se escrever

$$\mathbf{Q}_\mathsf{C}^k = \begin{bmatrix} \mathbf{F}_\mathsf{N}^k & \mathbf{x}_c^k - \mathbf{X}_i \times \mathbf{F}_\mathsf{N}^k & -\mathbf{F}_\mathsf{N}^k & -\mathbf{x}_c^k - \mathbf{X}_j \times \mathbf{F}_\mathsf{N}^k \end{bmatrix} \tag{2.109}$$

onde

$$\mathbf{F}_\mathsf{N}^k = \lambda_\mathsf{N}^k \mathbf{N}^k \tag{2.110}$$

é a força normal no ponto de contato (note que as forças generalizadas são definidas pela força normal e o torque da força normal em relação ao centro de massa de cada corpo). O multiplicador de Lagrange λ_N^k deve ser tal que $\mathbf{F}_\mathsf{N}^k$

- deve prevenir a interpenetração dos corpos;
- deve ser uma força repulsiva; e
- anula-se quando os corpos se separarem.

Para satisfazer a primeira condição, deve-se analisar $\ddot{\mathbf{C}}^k$, a qual mede como os dois corpos estão acelerando um em relação ao outro na direção da normal no ponto de contato. Se $\ddot{\mathbf{C}}^k > 0$, o contato entre os corpos será quebrado emt $t > t_c$; se $\ddot{\mathbf{C}}^k = 0$, estes permanecerão em contato de repouso; se $\ddot{\mathbf{C}}^k < 0$, os corpos estão acelerando um na direção do outro e haverá interpenetração em $t > t_c$, o que deve ser evitado. Portanto, a primeira condição impõe a restrição

$$\ddot{\mathbf{C}}^k \geq 0 \tag{2.111}$$

onde

$$\ddot{\mathbf{C}}^k = \frac{d}{d}(\mathbf{J}_{\mathbf{C}}^k \mathbf{u}^k) = \mathbf{J}_{\mathbf{C}}^k \dot{\mathbf{u}}^k + \dot{\mathbf{J}}_{\mathbf{C}}^k \mathbf{u}^k = (\mathbf{J}_{\mathbf{C}}^k \dot{\mathbf{u}}^k) + c_{\mathsf{C}}^k = a_{\mathsf{C}}^k. \tag{2.112}$$

A segunda e terceiras condições são satisfeitas com as seguintes restrições:

$$\lambda_{\mathsf{N}}^k \geq 0 \tag{2.113}$$

e

$$\ddot{\mathbf{C}}^k \lambda_{\mathsf{N}}^k = 0. \tag{2.114}$$

As forças de restrição de todos os C contatos de repouso da cena em um instante da simulação devem ser determinadas simultaneamente. Para tal, organiza-se, como feito anteriormente, as sub-matrizes $\mathbf{J}_{\mathbf{N}_i}^k$ e $\mathbf{J}_{\mathbf{N}_j}^k$ de cada contato k da cena em uma matriz global $\mathbf{J}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{C}\times 6\mathsf{N}}$,

$$\mathbf{J}_{\mathsf{C}} = \begin{bmatrix} \mathbf{J}_{\mathbf{N}_1}^1 & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}_{\mathbf{N}_N}^1 \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}_{\mathbf{N}_1}^k & \cdots & \mathbf{J}_{\mathbf{N}_i}^k & \cdots & \mathbf{J}_{\mathbf{N}_j}^k & \cdots & \mathbf{J}_{\mathbf{N}_N}^k \\ \vdots & & \vdots & & \vdots & & \vdots \\ \mathbf{J}_{\mathbf{N}_1}^R & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{J}_{\mathbf{N}_N}^R \end{bmatrix}. \tag{2.115}$$

A exemplo da matriz da (2.97), a matriz $\mathbf{J}_{\mathsf{C}}$ também é bastante esparsa, pois, para um contato k entre dois corpos A_i^k e A_j^k, a linha k correspondente contém como elementos não-nulos apenas as sub-matrizes $\mathbf{J}_{\mathbf{N}_i}^k$ e $\mathbf{J}_{\mathbf{N}_j}^k$, respectivamente nas colunas i e j. Da mesma forma, pode-se concatenar em um vetor global $\boldsymbol{\lambda}_{\mathsf{C}} \in \mathbb{R}^{\mathsf{C}}$ os multiplicadores de Lagrange correspondentes a todas as forças generalizadas de restrição de contato de repouso,

$$\boldsymbol{\lambda}_{\mathsf{C}} = [\lambda_{\mathbf{N}}^1 \ \ldots \ \lambda_{\mathbf{N}}^k \ \ldots \ \lambda_{\mathbf{N}}^{\mathsf{C}}]^T \geq \mathbf{0}. \tag{2.116}$$

O vetor global $\mathbf{Q}_{\mathsf{C}}$ de forças generalizadas de restrição dos contatos é

$$\mathbf{Q}_{\mathsf{C}} = \mathbf{J}_{\mathsf{C}}^{\mathsf{T}} \boldsymbol{\lambda}_{\mathsf{C}}, \tag{2.117}$$

o qual, adicionado à (2.100), resulta na equação de movimento

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{Q} + \mathbf{Q}_{\mathsf{J}} + \mathbf{Q}_{\mathsf{C}} = \mathbf{Q} + \mathbf{J}_{\mathsf{J}}^{\mathsf{T}} \boldsymbol{\lambda}_{\mathsf{J}} + \mathbf{J}_{\mathsf{C}}^{\mathsf{T}} \boldsymbol{\lambda}_{\mathsf{C}}. \tag{2.118}$$

As forças de contato e acelerações devem satisfazer as condições

$$\begin{aligned} &\mathbf{J}_{\mathsf{J}}\dot{\mathbf{u}} + \mathbf{c}_{\mathsf{J}} = \mathbf{0}, \\ &\mathbf{a}_{\mathsf{c}} \geq \mathbf{0} \quad \text{complementar a} \quad \boldsymbol{\lambda}_{\mathsf{C}} \geq \mathbf{0}, \end{aligned} \tag{2.119}$$

onde $\mathbf{a}_{\mathsf{c}} = \mathbf{J}_{\mathsf{C}}\dot{\mathbf{u}} + \mathbf{c}_{\mathsf{C}}$. Em notação matricial pode-se escrever:

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{a}_{\mathsf{C}} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & -\mathbf{J}_{\mathsf{J}}^T & -\mathbf{J}_{\mathsf{C}}^T \\ \mathbf{J}_{\mathsf{J}} & \mathbf{0} & \mathbf{0} \\ \mathbf{J}_{\mathsf{C}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \dot{\mathbf{u}} \\ \boldsymbol{\lambda}_{\mathsf{J}} \\ \boldsymbol{\lambda}_{\mathsf{C}} \end{bmatrix} + \begin{bmatrix} -\mathbf{Q} \\ \mathbf{c}_{\mathsf{J}} \\ \mathbf{c}_{\mathsf{C}} \end{bmatrix} \tag{2.120}$$

A condição de complementaridade deve ser entendida como $a_{\mathsf{C}}^k \lambda_{\mathsf{N}}^k = 0, k = 1, 2, 3..., \mathsf{C}$, isto é, a aceleração relativa normal e a força de restrição correspondente devem ser ambas não-negativas, e se uma é positiva então a outra deve ser nula. A determinação das forças de restrições e acelerações que satisfazem a (2.118) e as condições da (2.125) é um *problema de complementaridade linear* (PCL), neste caso, *misto* ou *generalizado*, pois nem todas as variáveis estão sujeitas às condições de complementaridade. Um PCL misto pode ser transformado em um PCL puro [Cli99] e resolvido por vários algoritmos, entre os quais o algoritmo

de Lenke [Ebe04]. O componente do motor de física responsável pela tarefa é chamado *solucionador de PCL* e será abordado mais adiante. Para problemas de contato sem atrito, pode-se provar que sempre há uma solução para o PCL correspondente [AP97]. Contudo, para problemas de contato com atrito cuja formulação é baseada em acelerações a existência de uma solução não é garantida. Anitescu e Potra [AP97] propuseram uma formulação *baseada em velocidade* para o problema e provaram a existência de uma solução para o caso geral, embora nem sempre única, independentemente da configuração e número de contatos. Na formulação baseada em velocidade, usa-se um passo do método de integração de Euler para aproximar a aceleração do sistema como sendo

$$\mathbf{M}\dot{\mathbf{u}} = \frac{\mathbf{u}(t+\Delta t) - \mathbf{u}(t)}{\Delta t}, \tag{2.121}$$

onde $\mathbf{u}(t)$ é a velocidade no início do passo de tempo corrente da simulação, cujo intervalo é Δt, e $\mathbf{u}(t+\Delta t)$ é a velocidade no próximo passo de tempo. Substituindo-se na (2.118) vem

$$\begin{aligned} \mathbf{M}(\mathbf{u}(t+\Delta t) - \mathbf{u}(t)\Delta t) &= (\mathbf{Q} + \mathbf{Q}_\mathsf{J} + \mathbf{Q}_\mathsf{C})\Delta t \\ &= \mathbf{Q}\Delta t + \mathbf{J}_\mathsf{J}^\mathsf{T}\boldsymbol{\lambda}_\mathsf{J}\Delta t + \mathbf{J}_\mathsf{C}^\mathsf{T}\boldsymbol{\lambda}_\mathsf{C}\Delta t. \end{aligned} \tag{2.122}$$

Note, na equação acima, que as forças do lado direito são multiplicados por Δt, ou seja, tais termos representam *impulsos generalizados* ao invés de forças. Além disso, somente as restrições cinemáticas são consideradas para determinação dos impulsos (no lugar de forças) de restrição (em junções e contatos) e velocidades no próximo passo de tempo (no lugar de acelerações). Assim, velocidades válidas são aquelas que satisfazem

$$\mathbf{J}\mathbf{u}(t+\Delta t) = 0 \tag{2.123}$$

e, para que não haja interpenetração entre os corpos,

$$\mathbf{w}_\mathsf{C} \geq \mathbf{0} \quad \text{complementar a} \quad \lambda_\mathsf{C} \geq \mathbf{0}, \tag{2.124}$$

onde $\mathbf{w}_\mathsf{C} = \mathbf{J}_\mathsf{C}\mathbf{u}(t+\Delta t)$. A formulação pode ser escrita matricialmente como

$$\begin{aligned} &\mathbf{J}_\mathsf{J}\dot{\mathbf{u}} + \mathbf{c}_\mathsf{J} = \mathbf{0}, \\ &\mathbf{a}_\mathsf{c} \geq \mathbf{0} \quad \text{complementar a} \quad \boldsymbol{\lambda}_\mathsf{C} \geq \mathbf{0}, \end{aligned} \tag{2.125}$$

onde $\mathbf{a}_\mathsf{c} = \mathbf{J}_\mathsf{C}\dot{\mathbf{u}} + \mathbf{c}_\mathsf{C}$. Em notação matricial pode-se escrever:

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{w}_\mathsf{C} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & -\mathbf{J}_\mathsf{J}^T & -\mathbf{J}_\mathsf{C}^T \\ \mathbf{J}_\mathsf{J} & \mathbf{0} & \mathbf{0} \\ \mathbf{J}_\mathsf{C} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}(t+\Delta t) \\ \boldsymbol{\lambda}_\mathsf{J}\Delta t \\ \boldsymbol{\lambda}_\mathsf{C}\Delta t \end{bmatrix} + \begin{bmatrix} -\mathbf{M}\mathbf{u}(t) - \mathbf{Q}\Delta t \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \tag{2.126}$$
$$\mathbf{w}_C \geq \mathbf{0} \quad \text{complementar a} \quad \boldsymbol{\lambda}_\mathsf{C} \geq \mathbf{0}.$$

2.8 Impacto

Um vez que a solução do PCL ((2.126)) envolve impulsos ao invés de forças, impactos podem ser tratados mais naturalmente com a formulação baseada em velocidades. Isso porque, como no exemplo da partícula, para modificar instantaneamente a velocidade no ponto de impacto deve-se aplicar um impulso generalizado aos corpos em contato, justamente o que está acontecendo no PCL. Seja então um contato k cujo *coeficiente de restituição* é $0 \leq \varepsilon_k \leq 1$. Usando-se a lei de impacto de Newton, tem-se que a velocidade normal dos corpos A_i^k e A_i^k, após a colisão, é

$$\mathbf{J}_\mathsf{C}^k\mathbf{u}^k(t+\Delta t) \geq -\varepsilon_k\mathbf{J}_\mathsf{C}^k\mathbf{u}^k(t) \tag{2.127}$$

Se $\varepsilon_k = 1$, então nenhuma energia cinética é perdida na colisão e o choque é perfeitamente elástico; se $\varepsilon_k = 0$, toda a energia cinética é perdida e os corpos permanecerão em contato de repouso após a colisão. O coeficiente de restituição é uma propriedade comumente associada ao material de um corpo rígido. Neste caso, o valor de ε_k pode ser tomado como uma combinação (valor máximo, mínimo, média, etc.) dos coeficientes de restituição dos corpos em contato.

Uma das condições do PCL (2.126) é que, para o k-ésimo contato, $\mathbf{J}_{\mathsf{C}}^k \mathbf{u}^k(t + \Delta t) \geq 0$, ou seja, *qualquer* velocidade normal não-negativa associada aos corpos em contato no próximo passo de tempo é válida, desde que as demais condições sejam também satisfeitas. Contudo, se a velocidade normal $\mathbf{J}_{\mathsf{C}}^k \mathbf{u}^k(t)$ no passo de tempo corrente (antes da aplicação do impulso) for negativa, então o que a inequação (2.127) estabelece é que a velocidade normal no próximo passo de tempo (após a aplicação do impulso) não pode assumir *qualquer* valor não-negativo, mas maior ou igual a uma proporção do oposto de $\mathbf{J}_{\mathsf{C}}^k \mathbf{u}^k(t)$, sendo esta dada pelo coeficiente de restituição. Esta condição pode ser expressa como

$$\mathbf{J}_{\mathsf{C}}^k \mathbf{u}^k(t + \Delta t) + b_{\mathsf{C}}^k \geq 0, \tag{2.128}$$

onde

$$b_{\mathsf{C}}^k = \varepsilon_k \mathbf{J}_{\mathsf{C}}^k \mathbf{u}^k(t). \tag{2.129}$$

(Note, na equação acima, que $b_{\mathsf{C}}^k \leq 0$, senão os corpos não estariam em contato no passo de tempo corrente.) Para todos os C pontos de contato da cena pode-se escrever

$$\mathbf{w}_{\mathsf{C}} = \mathbf{J}_{\mathsf{C}} \mathbf{u}(t + \Delta t) + \mathbf{b}_{\mathsf{C}} \geq \mathbf{0}, \tag{2.130}$$

o que equivale a adicionar o vetor $\mathbf{b}_{\mathsf{C}} \in \mathsf{C}$,

$$\mathbf{b}_{\mathsf{C}} = [b_{\mathsf{C}}^1 \ \ldots \ b_{\mathsf{C}}^k \ \ldots \ b_{\mathsf{C}}^{\mathsf{C}}]^T, \tag{2.131}$$

ao PCL (2.126), resultando:

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{w}_{\mathsf{C}} \end{bmatrix} = \begin{bmatrix} \mathbf{M} & -\mathbf{J}_{\mathsf{J}}^T & -\mathbf{J}_{\mathsf{C}}^T \\ \mathbf{J}_{\mathsf{J}} & \mathbf{0} & \mathbf{0} \\ \mathbf{J}_{\mathsf{C}} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}(t + \Delta t) \\ \boldsymbol{\lambda}_{\mathsf{J}} \Delta t \\ \boldsymbol{\lambda}_{\mathsf{C}} \Delta t \end{bmatrix} + \begin{bmatrix} -\mathbf{M}\mathbf{u}(t) - \mathbf{Q}\Delta t \\ \mathbf{0} \\ \mathbf{b}_{\mathsf{C}} \end{bmatrix}, \tag{2.132}$$
$$\mathbf{w}_C \geq \mathbf{0} \quad \text{complementar a} \quad \boldsymbol{\lambda}_{\mathsf{C}} \geq \mathbf{0}.$$

2.9 Atrito

Para um contato k sem atrito entre dois corpos A_i^k e A_j^k, as forças generalizadas de restrição $\mathbf{Q}_{\mathsf{C}}^k$ que surgem no ponto de contato $\mathbf{x}_C^k$, (2.109), têm quatro componentes vetoriais: duas forças paralelas à normal $\mathbf{N}^k$ do plano de contato - uma atuando no centro de massa de A_i^k no sentido de $\mathbf{N}^k$ e outra atuando no centro de massa de A_j^k no sentido oposto - e os dois torques resultantes da aplicação no ponto de contato da força normal respectiva de cada corpo em relação ao centro de massa do corpo.

Em adição, quando há atrito entre os corpos no k-ésimo contato, surgem também *forças de atrito* tangenciais ao plano de contato — portanto perpendiculares a $\mathbf{N}^k$ —, que juntamente com os torques correspondentes definem as forças generalizadas de atrito no ponto de contato $\mathbf{x}_C^k$. A força de restrição no contato passa a ser então a soma da força normal $\mathbf{F}_{\mathbf{N}}^k$ e da força de atrito $\mathbf{F}_{\mathbf{t}}^k$. De acordo com a lei empírica de atrito de Coulomb, a força de contato resultante permanece no interior ou na superfície de uma região cônica em torno da normal no ponto

de contato chamada *cone de atrito*, definido como o conjunto de todas as forças de contato tais que

$$\|\mathbf{F}_{\mathbf{t}}^k\| \leq \mu_k \underbrace{\|\mathbf{F}_{\mathbf{N}}^k\|}_{\lambda_{\mathbf{N}}^k}, \tag{2.133}$$

onde μ_k é o coeficiente de atrito do k-ésimo contato e a magnitude da força normal é $\lambda_{\mathbf{N}}^k$. A exemplo do coeficiente de restituição, o coeficiente de atrito é uma propriedade do material de um corpo rígido, sendo o valor de μ_k uma combinação dos coeficientes de atrito dos corpos em contato. A fim de incorporar o atrito no PCL (2.132), a definição não-linear do cone de atrito é aproximada por uma representação poliedral em forma de pirâmide, chamada *pirâmide de atrito* [Bar94a, TPSL97], conforme ilustrado na Figura 2.7. Os versores ortogonais $\mathbf{t}_1^k$ e $\mathbf{t}_2^k$ - os quais podem ser tomados arbitrariamente no plano de contato para atrito isotrópico - e seus opostos são perpendiculares a cada uma das arestas da base da pirâmide. Juntamente com o versor normal $\mathbf{N}^k$, $\mathbf{t}_1^k$ e $\mathbf{t}_2^k$ definem, no ponto de contato, um sistema de coordenadas Cartesianas chamado *referencial do contato*. (Pirâmides com um número par maior de faces são possíveis, veja [Erl04]. A aproximação mais simples com quatro faces é adotada aqui para diminuir o número de operações - em tempo real -executadas pelo motor de física.) Da

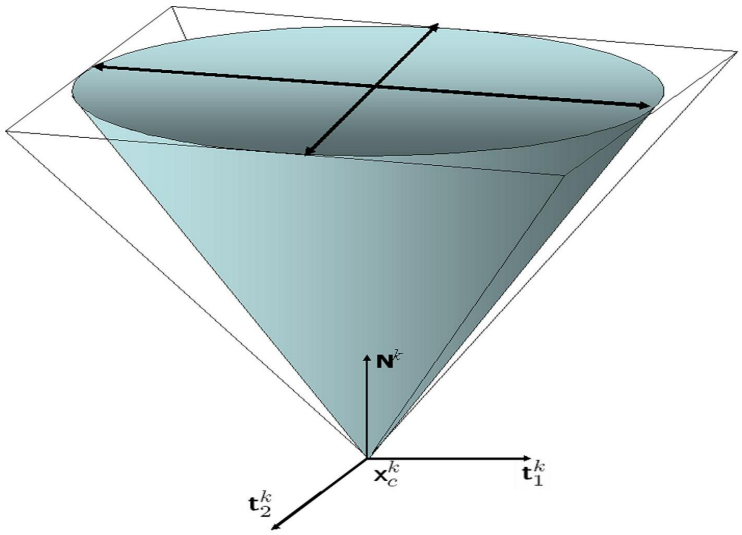


Figura 2.7: Pirâmide de atrito.

mesma forma que $\mathbf{J}_{\mathbf{C}}^k\mathbf{u}^k$ representa a velocidade relativa normal, pode-se definir um operador $\mathbf{J}_{\mathbf{C}}^k \in \mathbb{R}^{2\times 12}$ tal que a velocidade relativa tangencial entre os corpos A_i^k e A_j^k no k-ésimo contato pode ser escrita em função dos versores $\mathbf{t}_1^k$ e $\mathbf{t}_2^k$ como

$$\mathbf{J}_{\mathbf{t}}^k\mathbf{u}^k = \begin{bmatrix} \mathbf{J}_{\mathbf{t}_1 i}^k & \mathbf{J}_{\mathbf{t}_1 j}^k \\ \mathbf{J}_{\mathbf{t}_2 i}^k & \mathbf{J}_{\mathbf{t}_2 j}^k \end{bmatrix} \mathbf{u}^k = \begin{bmatrix} \mathbf{t}_1^k & \mathbf{x}_c^k - \mathbf{X}_i \times \mathbf{t}_1^k & -\mathbf{t}_1^k & -\mathbf{x}_c^k - \mathbf{X}_j \times \mathbf{t}_1^k \\ \mathbf{t}_2^k & \mathbf{x}_c^k - \mathbf{X}_i \times \mathbf{t}_2^k & -\mathbf{t}_2^k & -\mathbf{x}_c^k - \mathbf{X}_j \times \mathbf{t}_2^k \end{bmatrix}, \tag{2.134}$$

Similarmente, pode-se escrever, a partir do operador $\mathbf{J}_{\mathbf{t}}^k$, que as forças generalizadas de atrito são

$$\mathbf{Q}_{\mathbf{t}}^k\mathbf{u}^k = \begin{bmatrix} \mathbf{J}_{\mathbf{t}_1 i}^k & \mathbf{J}_{\mathbf{t}_1 j}^k \\ \mathbf{J}_{\mathbf{t}_2 i}^k & \mathbf{J}_{\mathbf{t}_2 j}^k \end{bmatrix} \begin{bmatrix} \lambda_{t1}^k \\ \lambda_{t2}^k \end{bmatrix} = \begin{bmatrix} \mathbf{F}_{\mathbf{t}1}^k & \mathbf{x}_c^k - \mathbf{X}_i \times \mathbf{F}_{\mathbf{t}1}^k & -\mathbf{F}_{\mathbf{t}1}^k & -\mathbf{x}_c^k - \mathbf{X}_j \times \mathbf{F}_{\mathbf{t}1}^k \\ \mathbf{F}_{\mathbf{t}2}^k & \mathbf{x}_c^k - \mathbf{X}_i \times \mathbf{F}_{\mathbf{t}2}^k & -\mathbf{F}_{\mathbf{t}2}^k & -\mathbf{x}_c^k - \mathbf{X}_j \times \mathbf{F}_{\mathbf{t}2}^k \end{bmatrix}, \tag{2.135}$$

onde

$$\begin{aligned} \mathbf{F}_{\mathbf{t}1}^k &= \lambda_{\mathbf{t}1}^k \mathbf{t}_1^k \\ \mathbf{F}_{\mathbf{t}2}^k &= \lambda_{\mathbf{t}2}^k \mathbf{t}_2^k \end{aligned} \tag{2.136}$$

De acordo com a lei de Coulomb, o atrito *dinâmico* ocorre quando a velocidade relativa tangencial é diferente de zero; neste caso, a força de atrito atinge seu valor máximo (na superfície da pirâmide de atrito) e uma direção oposta ao movimento. Isto significa que

$$\lambda^k_{\mathbf{t}_\beta} = \begin{cases} -\mu_k \lambda^k_{\mathbf{N}} & \text{se} \quad [\mathbf{J}^k_{\mathbf{t}_{\beta i}} \quad \mathbf{J}^k_{\mathbf{t}_{\beta j}}]\mathbf{u}^k > 0, \\ +\mu_k \lambda^k_{\mathbf{N}} & \text{se} \quad [\mathbf{J}^k_{\mathbf{t}_{\beta i}} \quad \mathbf{J}^k_{\mathbf{t}_{\beta j}}]\mathbf{u}^k < 0, \end{cases} \tag{2.137}$$

para $\beta = 1{,}2$. O atrito *estático* ocorre quando a velocidade relativa tangencial é nula; neste caso, a força de atrito (no interior da pirâmide de atrito) é tal que

$$\lambda^k_{\mathbf{t}_\beta} < |\mu_k \lambda^k_{\mathbf{N}}|. \tag{2.138}$$

As condições da (2.137) e (2.138) são, de fato, condições mais gerais de complementaridade. Para incorporar o atrito no PCL (2.132) adiciona-se $\mathbf{J}^k_{\mathbf{t}}$ à (2.108), resultando na matriz Jacobiana $\mathbf{J}^k_{\mathsf{C}} \in \mathbb{R}^{3\times 12}$ do k-ésimo contato,

$$\mathbf{J}^k_{\mathsf{C}} = \begin{bmatrix} \mathbf{J}^k_{\mathsf{N}_i} & \mathbf{J}^k_{\mathsf{N}_j} \\ \mathbf{J}^k_{\mathbf{t}_{1i}} & \mathbf{J}^k_{\mathbf{t}_{1i}} \\ \mathbf{J}^k_{\mathbf{t}_{2i}} & \mathbf{J}^k_{\mathbf{t}_{2i}} \end{bmatrix} = \begin{bmatrix} \underbrace{\mathbf{N}^k}_{\Gamma^k_{\mathsf{N}_i}} & \underbrace{\mathbf{x}^k_c - \mathbf{X}_i \times \mathbf{N}^k}_{\Omega^k_{\mathsf{N}_i}} & \underbrace{-\mathbf{N}^k}_{\Gamma^k_{\mathsf{N}_j}} & \underbrace{-\mathbf{x}^k_c - \mathbf{X}_j \times \mathbf{N}^k}_{\Omega^k_{\mathsf{N}_j}} \\ \underbrace{\mathbf{t}_1{}^k}_{\Gamma^k_{\mathbf{t}_{1i}}} & \underbrace{\mathbf{x}^k_c - \mathbf{X}_i \times \mathbf{t}_1{}^k}_{\Omega^k_{\mathbf{t}_{1i}}} & \underbrace{-\mathbf{t}_1{}^k}_{\Gamma^k_{\mathbf{t}_{1j}}} & \underbrace{-\mathbf{x}^k_c - \mathbf{X}_j \times \mathbf{t}_1{}^k}_{\Omega^k_{\mathbf{t}_{1j}}} \\ \underbrace{\mathbf{t}_2{}^k}_{\Gamma^k_{\mathbf{t}_{2i}}} & \underbrace{\mathbf{x}^k_c - \mathbf{X}_i \times \mathbf{t}_2{}^k}_{\Omega^k_{\mathbf{t}_{2i}}} & \underbrace{-\mathbf{t}_2{}^k}_{\Gamma^k_{\mathbf{t}_{2j}}} & \underbrace{-\mathbf{x}^k_c - \mathbf{X}_j \times \mathbf{t}_2{}^k}_{\Omega^k_{\mathbf{t}_{2j}}} \end{bmatrix}, \tag{2.139}$$

Adicionando-se as forças generalizadas de atrito na (2.109) vem

$$\mathbf{Q}^k_{\mathsf{C}} = (\mathbf{J}^k_{\mathsf{C}})^T \underbrace{\begin{bmatrix} \lambda^k_{\mathbf{N}} \\ \lambda^k_{\mathbf{t}_1} \\ \lambda^k_{\mathbf{t}_2} \end{bmatrix}}_{\boldsymbol{\lambda}^k_{\mathsf{C}}} \tag{2.140}$$

Introduzindo-se $\boldsymbol{\lambda}^k_{min} = [0 \; -\mathbf{u}^k\boldsymbol{\lambda}^k_{\mathbf{N}} \; -\mu^k\boldsymbol{\lambda}^k_{\mathbf{N}}]^T$ e $\boldsymbol{\lambda}^k_{max} = [\infty \; \mathbf{u}^k\boldsymbol{\lambda}^k_{\mathbf{N}} \; \mathbf{u}^k\boldsymbol{\lambda}^k_{\mathbf{N}}]^T$, as restrições do contato k podem ser escritas em uma mesma notação unificada, como a mostrada a seguir

$$\begin{aligned} &\lambda^k_{\mathsf{C}_l} = \lambda^k_{min_l} \Rightarrow (\mathbf{J}^k_{\mathsf{C}}\mathbf{u}^k)_l \geq 0, \\ &\lambda^k_{\mathsf{C}_l} = \lambda^k_{max_l} \Rightarrow (\mathbf{J}^k_{\mathsf{C}}\mathbf{u}^k)_l \leq 0, \\ &\lambda^k_{min_l} < \lambda^k_{\mathsf{C}_l} < \lambda^k_{max_l} = \lambda^k_{min_l} \Rightarrow (\mathbf{J}^k_{\mathsf{C}}\mathbf{u}^k)_l = 0, \end{aligned} \tag{2.141}$$

onde o índice l=1,2,3 indica a linha da matriz ou o elemento do vetor correspondente. Para todos os C pontos de contato, as sub-matrizes da (2.139) podem ser combinadas na matriz global $\mathbf{J}_{\mathsf{C}}$, cuja dimensão é, agora, $3\mathsf{C}\times 6\mathsf{N}$, e os multiplicadores de Lagrange das forças de contato combinados em $\boldsymbol{\lambda}_{\mathsf{C}} \in \mathbb{R}^{3\mathsf{C}}$. O LCP (2.132) fica agora sujeito às condições de complementaridade (2.141).

2.10 Unificando Junções e Contatos

As restrições de junções e contatos do PCL (2.132) podem ser tratadas de maneira unificada como a seguir. Primeiro, estabelece-se para cada restrição de junção valores mínimo e máximo para a magnitude do impulso correspondente, tal como feito para as restrições de contato. Para uma restrição r devida a uma junção tem-se

$$-\infty < \boldsymbol{\lambda}^r_{\mathsf{J}} < \infty \tag{2.142}$$

Logo, independentemente do tipo de restrição pode-se escrever

$$w^r = \mathbf{J}^r \mathbf{u}^r(t + \Delta t) + b^r \tag{2.143}$$

tal que

$$\begin{aligned} \lambda^r = \lambda^r_{min} &\Rightarrow w^r \geq 0, \\ \lambda^r = \lambda^r_{max} &\Rightarrow w^r \leq 0, \\ \lambda^r_{min} < \lambda^r \lambda^r_{max} = \lambda^r_{min} &\Rightarrow w^r = 0, \end{aligned} \tag{2.144}$$

O termo b^r vale zero para uma restrição devida a uma junção ou é dado pela (2.129) para uma restrição de contato. Ignorando-se os índices referentes a junções e contatos no PCL (2.132), tem-se, para todas restrições do sistema,

$$\mathbf{w} = \mathbf{J}\mathbf{u}(t + \Delta t) + \mathbf{b} \geq \mathbf{0} \quad \text{complementar a} \quad \boldsymbol{\lambda}_{min} \leq \boldsymbol{\lambda} \leq \boldsymbol{\lambda}_{max}. \tag{2.145}$$

A equação do movimento fica

$$\mathbf{M}\mathbf{u}(t + \Delta t) = \mathbf{M}\mathbf{u}(t) + \mathbf{Q}\Delta t + \mathbf{J}^T \boldsymbol{\lambda} \Delta t. \tag{2.146}$$

Isolando $\mathbf{u}(t + \Delta t)$ na equação acima obtém-se

$$\mathbf{u}(t + \Delta t) = \mathbf{u}(t) + \mathbf{M}^{-1}\mathbf{J}^T \boldsymbol{\lambda} \Delta t + \mathbf{M}^{-1}\mathbf{Q}\Delta t. \tag{2.147}$$

Substituindo-se na (2.145) resulta

$$\begin{gathered} \mathbf{w} = \underbrace{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T}_{\mathbf{A}} \underbrace{\boldsymbol{\lambda}\Delta t}_{\mathbf{x}} + \underbrace{\mathbf{J}(\mathbf{u}(t) + \mathbf{M}^{-1}\mathbf{Q}\Delta t) + \mathbf{b}}_{\mathbf{f}}. \\ ou \\ \mathbf{w} = \mathbf{A}\mathbf{x} + \mathbf{f} \quad \text{complementar a} \quad \boldsymbol{\lambda}_{min} \leq \boldsymbol{\lambda} \leq \boldsymbol{\lambda}_{max}. \end{gathered} \tag{2.148}$$

O LCP (2.148) define os dados de entrada do solucionador de PCL do motor de física, o qual determina o vetor $\mathbf{x}$, as magnitudes dos impulsos generalizados de restrição. A partir destes, efetua-se a atualização da velocidade dos corpos da cena em $t + \Delta t$ de acordo com a (2.147). O restante do cálculo do novo estado é feito com a atualização da configuração de todos os corpos rígidos,

$$\boldsymbol{\chi}(t + \Delta t) = \boldsymbol{\chi}(t) + \mathbf{D}\mathbf{u}(t + \Delta t)\Delta t. \tag{2.149}$$

2.11 Modelagem das Jacobianas

Esta seção trata de como podem ser modeladas as junções, e no final, como modelar a Jacobiana de restrição de contato de forma que as mesmas estruturas de dados possam ser utilizadas tanto para junções como para contatos. Antes disso, será realizada uma breve introdução de como podemos corrigir possíveis erros comuns que venham a surgir nestas junções. As submatrizes Jacobianas de junção seguem um padrão, portanto ficará fácil entender como a modelagem pode ser aplicada a diversos tipos de junções, inclusive à contatos.

Para a k-ésima restrição de junção, a restrição cinemática entre dois corpos, A_i^k e A_j^k, é descrita pela (2.95). Perceba que as sub-matrizes $\mathbf{J}^k_{J_i} = [\boldsymbol{\Gamma}^k_{J_i} \quad \boldsymbol{\Omega}^k_{J_i}]$ e $\mathbf{J}^k_{J_j} = [\boldsymbol{\Gamma}^k_{J_j} \quad \boldsymbol{\Omega}^k_{J_j}]$, partições da Jacobiana da junção k, multiplicam as velocidades dos corpos A_i^k e A_j^k, respectivamente. Ou seja, representam as velocidades do ponto de apoio da junção k para A_i^k e A_j^k. Obviamente, para manter os pontos de apoio das junções juntas, estes pontos devem mover-se com a mesma velocidade, e além disso, a soma das duas velocidades deve ser zero. A partir desta observação é baseada a estratégia para a montagem das Jacobianas das junções: dadas as velocidades dos corpos, montar uma matriz equação, de tal modo que a velocidade relativa na direção dos pontos de apoio da junção seja sempre zero.

Conectividade

A conectividade e movimentos de todas as junções serão descritos utilizando pontos âncora e eixos de junção. Um ponto âncora é um ponto no espaço onde dois pontos, um de cada corpo incidente, estão sempre perfeitamente alinhados. A localização de um ponto âncora relativo a um corpo A_i, é dados por um vetor relativo à base do corpo, $\mathbf{x}^i_{anc}$. A posição do ponto âncora no sistema de coordenadas globais, para uma junção k do corpo A_i, é dado por

$$\mathbf{x}^{gc}_{anc_k} = \mathbf{X}_i + \mathbf{R}(\mathbf{q}_i)\mathbf{x}^i_{anc_k}, \tag{2.150}$$

para um ponto âncora de um corpo A_i, temos que

$$\tilde{\mathbf{x}}^{gc}_{anc_k} = \mathbf{x}^{gc}_{anc} - \mathbf{X}_i, \tag{2.151}$$

onde ^{gc} denota coordenadas em relação ao sistema global.

Um eixo de junção descreve a direção de movimento permitido, tais como um eixo de rotação ou direção de escorregamento. O eixo da junção é dado por um vetor tridimensional, chamado de $\mathbf{s}^{wcs}_{axis}$. Esta maneira de descrevermos a conectividade é muito similar as "estruturas de pares de coordenadas" descritos por Featherstone [Fea87]. Realizando uma comparação, pontos da âncora correspondem à localização da origem dos pontos de apoio da junção, e eixos de junção correspondem à orientação dos pontos de apoio.

Erro da Junção

As restrições cinemáticas são restrições de velocidade, não das posições. Isto quer dizer que tanto erros numéricos como erros originários de aproximações internas podem adentrar no cálculo das posições na medida em que a simulação avança. Imagine que ocorreu algum erro de posicionamento, de modo que há um deslocamento no posicionamento dos pontos de apoio da junção ou um desalinhamento destes. Este erro pode ser reduzido ajustando a velocidade dos pontos de apoio, de tal sorte que o erro seja menor no próximo passo da simulação. Com isto, incluimos nossas restrições cinemáticas um termo de correção do erro da velocidade para a k-ésima junção, b^k_J:

$$\mathbf{J}^k_J\mathbf{u}^k = \mathbf{b}^k_J. \tag{2.152}$$

onde $\mathbf{b}^k_J \in \mathbb{R}^{m^k} = [b^k_{J_1} \ \dots \ b^k_{J_{m^k}}]$ é o vetor de erros da junção k que remove m^k graus de liberdade do sistema (b^k_J tem um valor para para restrição holonômica).

Estes erros de junção podem ser determinados para as N junções, bastanto para isso, incluir para cada junção k, $1 \leq k \leq \mathsf{N}$, seu respectivo termo $b^k_{J_m}$, transformando a (2.96) em:

$$\mathbf{\Gamma}^k_{J_i}\mathbf{V}^k_i + \mathbf{\Omega}^k_{J_i}\boldsymbol{\omega}^k_i + \mathbf{\Gamma}^k_{J_j}\mathbf{V}^k_j + \mathbf{\Omega}^k_{J_j}\boldsymbol{\omega}^k_j = \mathbf{b}^k_J, \tag{2.153}$$

Imagine, por exemplo, duas partículas i e j, que podem se mover sobre uma linha, e entre elas há uma junção, de modo que suas posições (uma em relação a outra) devem sempre ser iguais. A restrição cinemática então será

$$\mathbf{v}_i - \mathbf{v}_j = 0. \tag{2.154}$$

Supondo que algum erro esteja presente

$$\mathbf{x}_{err} = \mathbf{x}_i - \mathbf{x}_j, \tag{2.155}$$

onde $|\mathbf{x}_{err}| > 0$. Para ajustar as velocidades para que este erro seja eliminado dentro de um tempo Δt, precisamos que

$$\begin{aligned} \underbrace{\mathbf{v}_i - \mathbf{v}_j}_{\mathbf{Ju}} &= \underbrace{\frac{\mathbf{x}_{err}}{\Delta t}}_{\vec{b}}, \\ \mathbf{Ju} &= \mathbf{b}. \end{aligned} \tag{2.156}$$

Se junções ou limites estiverem sujeitos a algum erro inicial, e as relações iniciais estiverem em repouso, então os termos do erro irão acelerar as conexões. Não apenas os erros serão corrigidos, mas também os corpos continuarão a se mover posteriormente. Mas a correção de erros não deve adicionar energia cinética ao sistema, pois se forem, as conexões ligadas à junção irão começar a acelerar inesperadamente. De fato, a correção de erros tem o mesmo efeito que utilizar a lei da colisão de Newton para solucionar colisões simultâneas [Bar89]. Uma alternativa prática e aceitável para utilizar o parâmetro de redução de erro é discutido em [Erl04].

Parâmetro de Redução de Erros (ERP)

O tipo de abordagem adotada nesta dissertação para simular junções pertence à uma classe de algoritmos conhecidos como métodos *Full-Coordinate* [Erl04, Sha01], onde cada corpo em um mecanismo de junção é descrito por um conjunto completo das coordenadas de movimento dos corpos rígidos. Uma outra alternativa seria a utilização de métodos *Reduced-Coordinate*, por exemplo o algorítmo de *Featherstone* [Fea87]. A idéia central deste é que seja necessário descrever apenas os movimentos relativos dos corpos entre as junções. Consequentemente só precisaríamos das coordenadas relativas das junções.

Preferimos trabalhar com métodos *Full-Coordinate* porque a notação é mais fácil de ler e de trabalhar, enquanto que no outro tem termos mais longos e complicados; é também o método utilizado, por exemplo, pelo ODE. Além disso, embora os métodos que utilizem *Reduced-Coordinate* têm menos variáveis para trabalhar, eles geralmente são implementados por algorítmos recursivos. Estes algorítmos recursivos são limitados por estruturas na forma de árvore, o que torna difícil e necessário incontáveis extensões computacionais para que estes algorítmos lidem com laços fechados e contatos [Erl04].

Do ponto de vista da animação computacional, erros numéricos em um método *Full-Coordinate* são notados com mais facilidade. Sendo o motivo os erros no sistemas de coordenadas do corpo, dividiremos a junção e introduziremos um efeito chamado de *drifting*. *Drifting* é um termo utilizado quando uma coisa é "levada" ou "arrastada" por alguma força, neste caso as conexões da junção, que deveriam estar juntas, mas são arrastadas. Métodos de *Reduced coordinates* não sofrem de problemas de *drifting*, pois não importa quão grande os erros numéricos serão, a simulação sempre exibirá os corpos conectados adequadamente. Concluindo, com métodos *Full-Coordinate* podemos ter problemas de *drifting*, e há duas maneiras que eles podem ocorrer em um simulador:

- O usuário interage com uma junção, e esquece de setar a posição ou orientação correta de todos as conexões da mesma; e
- durante a simulação, erros podem transformar-se lentamente no efeito das conexões e serem arrastadas para fora de suas junções.

Na Seção 2.11.1 descreveremos as restrições cinemáticas de diferentes tipos de junções, e introduzimos alguns termos para correção de erros. Todos estes são multiplicados por um coeficiente, k_{cor} que denota a medida ou magnitude da taxa de correção de erro. A idéia

básica é simples, para cada junção temos um parâmetro de redução de erro, k_{erp},

$$0 \leq k_{erp} \leq 1. \tag{2.157}$$

Seu valor é uma medida de quanto o erro pode ser reduzido no próximo passo da simulação. O valor 0 quer dizer que não há correção de erro alguma, e um valor igual a 1 quer dizer que o erro deve ser totalmente eliminado. Se deixarmos que a duração do tempo no próximo passo da simulação seja denotado por um passo de tempo característico, Δt, então a seguinte constante é uma medida da taxa de variação,

$$k_{fps} = \frac{1}{\Delta t}. \tag{2.158}$$

O coeficiente k_{cor} agora pode ser determinado como,

$$k_{cor} = k_{erp} k_{fps}. \tag{2.159}$$

Deixar $k_{erp} = 1$ não é recomendado, já que diversas aproximações internas podem fazer com que o erro não seja completamente eliminado [Erl04]. O Open Dynamics Engine [eac] utiliza a mesma abordagem para correção de erros, e recomenda utilizar um valor de k_{erp} por volta de 0.8.

2.11.1 Montagem para Junções e Contatos

Deduziremos agora as Jacobianas e termos de correção de erros para alguns tipos de junções utilizadas no motor 3D. Estas informações são cruciais para entendimento das restrições cinemáticas explicadas nas seções anteriores. Aqui são listados três tipos básicos: junções esféricas (*ball-in-socket*), junção de revolução (*hinge* ou *revolution*), junção fixa (*fixed*); por fim mostraremos a Jacobiana para restrição de contato, o que faz sentido, já que na aplicação um contato é tratado como um tipo especial de junção. Este tratamento unificado é necessário para permitir que sejam utilizadas submatrizes Jacobianas que seguem um padrão em comum, tanto para junções como para contatos.

Junção Esférica

Uma junção esférica permite rotações arbitrárias entre dois corpos, assim como na Figura 2.8. Sabendo que uma junção esférica remove 3 graus de liberdade do sistema, concluimos que a Jacobiana da esfera, $\mathbf{J}^k_{ball}$ de uma junção k, é uma matriz Jacobiana 3×12. Assim como

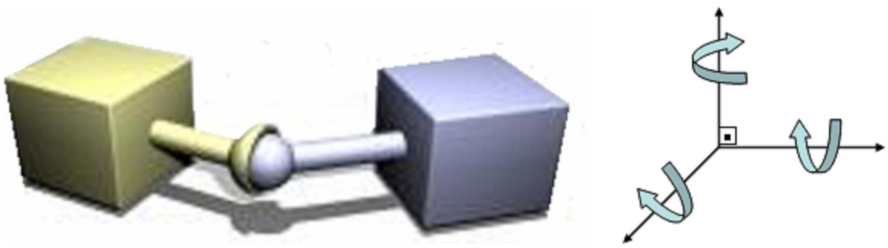


Figura 2.8: Exemplo de junção esférica.

fizemos na Seção 2.11, a submatriz Jacobiana é dada por

$$\mathbf{J}^k_{ball} = [\mathbf{\Gamma}^k_{J_i}, \quad \mathbf{\Gamma}^k_{J_j}, \quad \mathbf{\Omega}^k_{J_i}, \quad \mathbf{\Omega}^k_{J_j}] \tag{2.160}$$

onde

$$\mathbf{\Gamma}^k_{J_i} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \tag{2.161}$$

$$\mathbf{\Gamma}^k_{J_j} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.162}$$

$$\mathbf{\Omega}^k_{J_i} = (\mathbf{R}(\mathbf{q}_i)\tilde{\mathbf{x}}^k_{anc_i})\hat{}, \tag{2.163}$$

$$\mathbf{\Omega}^k_{J_j} = -(\mathbf{R}(\mathbf{q}_j)\tilde{\mathbf{x}}^k_{anc_j})\hat{}, \tag{2.164}$$

onde $\mathbf{\Gamma}^k_{J_i} \in \mathbb{R}^{3\times3}$, $\mathbf{\Omega}^k_{J_i} \in \mathbb{R}^{3\times3}$, $\mathbf{\Gamma}^k_{J_j} \in \mathbb{R}^{3\times3}$, $\mathbf{\Omega}^k_{J_j} \in \mathbb{R}^{3\times3}$, e o termo de correção de velocidade Mb^k_{ball} é dado por ,

$$\mathbf{b}^k_{ball} = k_{cor}(-\mathbf{X}_j - \mathbf{R}(\mathbf{q}_j)\mathbf{x}^k_{anc_j} + \mathbf{X}_i + \mathbf{R}(\mathbf{q}_i)\mathbf{x}^k_{anc_i}). \tag{2.165}$$

Junção de Revolução

Uma junção de revolução, também chamada de dobradiça ou *hinge*, permite apenas movimento relativo em torno de um eixo de junção especificado, Figura 2.9. Descrevemos esse tipo de junção por um ponto âncora no eixo de rotação e um eixo de junção, $\mathbf{s}^{gc}_{axis}$, dado por um vetor direção no sistema de coordenadas globais. Temos apenas um grau de liber-

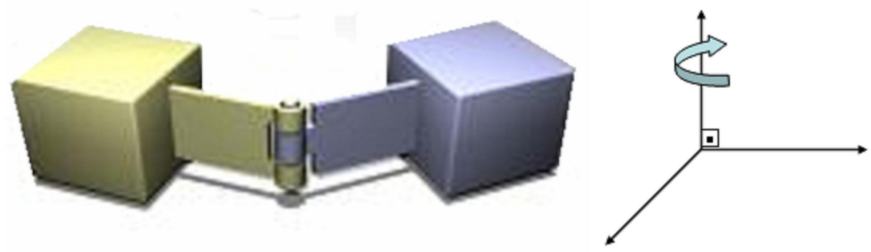


Figura 2.9: Exemplo de junção de revolução.

dade, o que significa que a junção de revolução tem 5 restrições no movimento relativo e consequentemente a Jacobiana, J^k_{hinge}, é uma matriz 5×12,

$$\mathbf{J}^k_{hinge} = [\mathbf{\Gamma}^k_{J_i}, \quad \mathbf{\Gamma}^k_{J_j}, \quad \mathbf{\Omega}^k_{J_i}, \quad \mathbf{\Omega}^k_{J_j}], \tag{2.166}$$

onde $\mathbf{\Gamma}^k_{J_i} \in \mathbb{R}^{5\times3}$, $\mathbf{\Omega}^k_{J_i} \in \mathbb{R}^{5\times3}$, $\mathbf{\Gamma}^k_{J_j} \in \mathbb{R}^{5\times3}$, $\mathbf{\Omega}^k_{J_j} \in \mathbb{R}^{5\times3}$, e o termo de correção de velocidade $\mathbf{b}^k_{hinge} \in \mathbb{R}^5$. Uma junção de revolução tem as mesmas restrições posicionais de uma junção esférica, então podemos imediatamente utilizar as 3 primeiras linhas da Jacobiana de restrição da junção esférica e também sua medida de erro; temos apenas que estender a Jacobiana de revolução com mais duas linhas que irão restringir a liberdade rotacional da junção esférica para apenas um eixo de rotação.

A estratégia para adicionar mais duas restrições rotacionais é porque, se desejamos permitir rotações apenas sobre o eixo de junção, somente a velocidade angular dos dois corpos em relação ao eixo de junção é permitido que seja diferente de 0, ou seja,

$$\mathbf{s}_{axis}.(\boldsymbol{\omega}_i - \boldsymbol{\omega}_j) \neq 0. \tag{2.167}$$

A velocidade angular relativa com qualquer outro eixo ortogonal a $\mathbf{s}_{axis}$ deve ser 0. Em particular, se deixarmos dois vetores, $\mathbf{t}_{ort1}$ e $\mathbf{t}_{ort2} \in \mathbb{R}^3$ tomados em relação ao sistema global

de coordenadas cartesians, serem versores ortogonais, e exigirmos que eles sejam ortogonais ao eixo de junção $\mathbf{s}_{axis}^{gc}$, então

$$\begin{aligned} \mathbf{t}_{ort1}.(\boldsymbol{\omega}_i - \boldsymbol{\omega}_j) = 0, \\ \mathbf{t}_{ort2}.(\boldsymbol{\omega}_i - \boldsymbol{\omega}_j) = 0. \end{aligned} \tag{2.168}$$

Destas equações podemos retirar as duas restrições cinemáticas e escrever a Jacobiana de revolução como,

$$\boldsymbol{\Gamma}_{J_i}^k = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{2.169}$$

$$\boldsymbol{\Gamma}_{J_j}^k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{2.170}$$

$$\boldsymbol{\Omega}_{J_i}^k = \begin{bmatrix} -(\mathbf{R}(\mathbf{q}_i)\tilde{\mathbf{x}}_{anc_i}^k)\tilde{} \\ (\mathbf{t}_{ort1})^T \\ (\mathbf{t}_{ort2})^T \end{bmatrix}, \tag{2.171}$$

$$\boldsymbol{\Omega}_{J_j}^k = \begin{bmatrix} (\mathbf{R}(\mathbf{q}_i)\tilde{\mathbf{x}}_{anc_i}^k)\tilde{} \\ -(\mathbf{t}_{ort1})^T \\ -(\mathbf{t}_{ort2})^T \end{bmatrix}. \tag{2.172}$$

Para o termo de medição de erro, já temos as três primeiras medidas herdadas da junção esférica, que cuidam dos erros posicionais. Mais duas medições de erros são necessárias por causa do desalinhamento rotacional sobre os eixos diferentes do eixo da junção.

Se armazenarmos o eixo da junção em relação às bases de ambos corpos, $\mathbf{s}_{axis}^i$ e $\mathbf{s}_{axis}^j$, e calcular então a direção do eixo de junção no sistema de coordenadas mundiais em relação a cada um dos corpos incidentes, teremos,

$$\begin{aligned} \mathbf{s}_i^{gc} = \mathbf{R}(\mathbf{q}_i)\mathbf{s}_{axis_i}^k \\ \mathbf{s}_j^{gc} = \mathbf{R}(\mathbf{q}_j)\mathbf{s}_{axis_j}^k \end{aligned} \tag{2.173}$$

Se $\mathbf{s}_i^{gc} = \mathbf{s}_j^{gc}$, óbviamente não há erro na orientação relativa da articulação entre os corpos. Se houver erro, então os corpos devem ser rotacionados de forma que $\mathbf{s}_i^{gc}$ e $\mathbf{s}_j^{gc}$ fiquem iguais. Isto pode ser obtido da seguinte maneira: imagine que o ângulo entre dois vetores é θ_{err}. Podemos então corrigir o erro relativo com uma rotação de θ_{err} em volta do eixo,

$$\mathbf{x}_{err} = \mathbf{s}_i^{gc} \times \mathbf{s}_j^{gc}. \tag{2.174}$$

Suponha que desejamos corrigir o erro por um ângulo de θ_{cor} em um tempo Δt que pode ser o tamanho do passo de tempo da simulação. Então necessitaríamos de uma velocidade angular relativa de magnitude

$$\begin{aligned} \|\boldsymbol{\omega}_{cor}\| = \frac{\theta_{cor}}{\Delta t} \\ \frac{k_{err}\theta_{err}}{\Delta t} \\ k_{err}\frac{1}{\Delta t}\theta_{err} \\ k_{err}k_{fps}\theta_{err} \\ k_{cor}\theta_{err} \end{aligned} \tag{2.175}$$

A direção desta velocidade angular corretiva é ditada pelo vetor $\mathbf{x}_{err}$, pois

$$\begin{aligned}\boldsymbol{\omega}_{cor} &= \|\boldsymbol{\omega}_{cor}\|\frac{\boldsymbol{\omega}_{err}}{\|\mathbf{x}_{err}\|} \\ &= k_{cor}\theta_{err}\frac{\mathbf{x}_{err}}{\|\mathbf{x}_{err}\|} \\ &= k_{cor}\theta_{err}\frac{\mathbf{x}_{err}}{\sin\theta_{err}}.\end{aligned} \tag{2.176}$$

No último passo, $\mathbf{s}_i^{gc}$ e $\mathbf{s}_j^{gc}$ são vetores unitários tais que,

$$\|\mathbf{x}_{err}\| = \|\mathbf{s}_i^{gc} \times \mathbf{s}_j^{gc}\| = \sin\theta_{err}. \tag{2.177}$$

Espera-se que o erro seja pequeno, por esta razão é razoável que utilizemos uma aproximação de um ângulo pequeno, onde $\theta_{err} \approx \sin\theta_{err}$, isto é,

$$\boldsymbol{\omega}_{cor} = k_{cor}\mathbf{x}_{err}. \tag{2.178}$$

Sabemos que $\mathbf{x}_{err}$ é ortogonal a $\mathbf{s}_{axis}^{gc}$, então o projetamos nos vetores $\mathbf{t}_{ort1}$ e $\mathbf{t}_{ort2}$, e terminamos encontrando a seguinte medição de erro,

$$\mathbf{b}_{hinge}^k = k_{cor}\begin{bmatrix}(\mathbf{X}_j + \mathbf{R}(\mathbf{q}_j)\mathbf{x}_{anc_j}^k - \mathbf{X}_i - \mathbf{R}(\mathbf{q}_i)\mathbf{x}_{anc_i}^k \\ \mathbf{t}_{ort1}.\mathbf{x}_{err} \\ \mathbf{t}_{ort2}.\mathbf{x}_{err}\end{bmatrix}. \tag{2.179}$$

Junção Fixa

Quanto ao caso das junções fixas, sabemos que uma junção deste tipo restringe dois corpos completamente de qualquer movimento relativo, e além disso tem 0 DOFs, da qual sabemos que a Jacobiana, $\mathbf{J}_{fixed}^k \in \mathbb{R}^{6\times12}$ é igual a

$$\mathbf{J}_{fixed}^k = [\boldsymbol{\Gamma}_{J_i}^k, \quad \boldsymbol{\Gamma}_{J_j}^k, \quad \boldsymbol{\Omega}_{J_i}^k, \quad \boldsymbol{\Omega}_{J_j}^k]. \tag{2.180}$$

A k-ésima junção, do tipo fixa, é descrita por um ponto âncora, e inicialmente um vetor

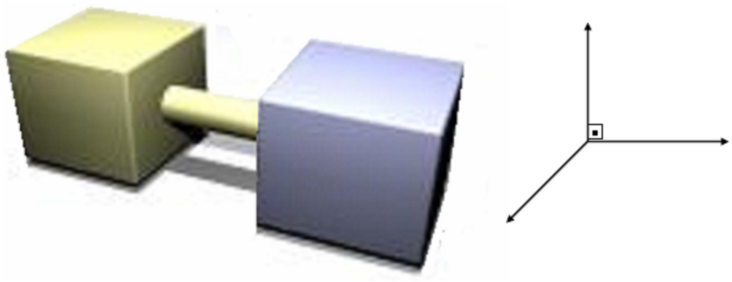


Figura 2.10: Exemplo de junção fixa.

deslocamento. Estes são armazenados na base do corpo A_i,

$$\mathbf{x}_{off_i}^k = \mathbf{X}_i - \mathbf{X}_j. \tag{2.181}$$

Note que o vetor deslocamento é calculado quando a junção foi inicialmente criada, e é constante. O deslocamento correspondente em coordenadas do sistema global é encontrado por,

$$\mathbf{x}_{off}^{gc} = \mathbf{R}(\mathbf{q}_i)\mathbf{x}_{off_i}^k. \tag{2.182}$$

Como temos uma junção fixa, ambos corpos pertencentes a ela devem rotacionar com a mesma velocidade angular, e as velocidades lineares devem obedecer a seguinte relação,

$$\mathbf{v}_j = \mathbf{v}_i + \boldsymbol{\omega}_j \times \mathbf{x}_{off}^{gc}. \tag{2.183}$$

Reunindo tudo isso podemos montar a matriz Jacobiana como,

$$\boldsymbol{\Gamma}_{J_i}^k = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{2.184}$$

$$\boldsymbol{\Gamma}_{J_j}^k = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{2.185}$$

$$\boldsymbol{\Omega}_{J_i}^k = \begin{bmatrix} & (\tilde{\mathbf{x}}_{off}^{wcs^k})\hat{} & \\ -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{2.186}$$

$$\boldsymbol{\Omega}_{J_j}^k = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2.187}$$

Similarmente, o termo de correção de erro é direto. Para corrigir o erro posicional, utilizamos as três primeiras entradas da junção esférica, $\mathbf{b}_{ball}^k$. Para correção do desalinhamento, como na junção de revolução, temos de descobrir uma velocidade angular para corrigir o erro de desalinhamento de θ_{err} radianos. A magnitude desta velocidade angular corretiva é, como anteriormente,

$$\begin{aligned} \|\boldsymbol{\omega}_{cor}\| &= \frac{\theta_{cor}}{\Delta t} \\ &\frac{k_{err}\theta_{err}}{\Delta t} \\ k_{err}&\frac{1}{\Delta t}\theta_{err} \\ k_{err}k_{fps}&\theta_{err} \\ k_{cor}&\theta_{err} \end{aligned} \tag{2.188}$$

Como na junçao de revolução, o valor desta velocidade angular corretiva é ditada por um eixo de rotação dado por algum versor $\mathbf{x}_{err}$

$$\begin{aligned} \boldsymbol{\omega}_{cor} &= \|\boldsymbol{\omega}_{cor}\|\mathbf{x}_{err} \\ &k_{cor}\theta_{err}\mathbf{x}_{err}. \end{aligned} \tag{2.189}$$

No entanto, ao contrário da anterior, a velocidade angular corretiva será encontrada da seguinte maneira: seja o desalinhamento rotacional dado por um quaternion, $\mathbf{q}_{err}$, então temos

$$\begin{aligned} \mathbf{q}_{err} &= [s, \mathbf{v}], \\ \mathbf{q}_{err} &= [\cos(\frac{\theta_{err}}{2}), \sin(\frac{\theta_{err}}{2})\mathbf{x}_{err}]. \end{aligned} \tag{2.190}$$

É esperado que o erro seja pequeno, portanto uma ângulo pequeno de aproximação é razoável, temos

$$\frac{\theta_{err}}{2}\mathbf{x}_{err} \approx \sin(\frac{\theta_{err}}{2})\mathbf{x}_{err} = \mathbf{v}. \tag{2.191}$$

Utilizando isso para a velocidade angular corretiva, encontramos que

$$\mathbf{w}_{cor} = k_{cor} 2\mathbf{v}. \tag{2.192}$$

Estas serão as três ultimas entradas para $\mathbf{J}_{fixed}$. Podemos agora escrever o vetor $\mathbf{b}_{fixed}$ como

$$\mathbf{b}^k_{fixed} = k_{cor} \begin{pmatrix} (\mathbf{X}_i + \mathbf{R}(\mathbf{q}_i)\mathbf{x}^k_{anc_i} - \mathbf{X}_j - \mathbf{R}(\mathbf{q}_j)\mathbf{x}^k_{anc_j} \\ 2\mathbf{v} \end{pmatrix}, \tag{2.193}$$

Pontos de Contato

Restrições de contato são distintas das restrições de junção, mas também são descritas por uma matriz Jacobiana. Esta matriz, chamada aqui de $J_{contact}$, pode ser expressa no mesmo padrão de submatrizes como as Jacobianas de junção, é possível inclusive, inserir um termo corretor de erro.

Sabemos que a Jacobiana de contato tem $1 + \beta$ restrições. Neste caso, como utilizamos $\beta = 2$, temos 3 restrições como mostrado na Seção 2.9. Portanto, é uma matriz Jacobiana de dimensão $\beta \times 12$,

$$\mathbf{J}^k_{contact} = [\mathbf{\Gamma}^k_{J_i}, \quad \mathbf{\Gamma}^k_{J_j}, \quad \mathbf{\Omega}^k_{J_i}, \quad \mathbf{\Omega}^k_{J_j}]. \tag{2.194}$$

A primeira linha corresponde as restrições da força normal e as β restantes correspondem as restrições de atrito tangencial, ou seja, no k-ésimo ponto de contato temos:

$$\mathbf{\Gamma}^k_{J_i} = \begin{bmatrix} \mathbf{N}^k \\ \mathbf{t}^k_1 \\ \mathbf{t}^k_2 \end{bmatrix}, \tag{2.195}$$

$$\mathbf{\Gamma}^k_{J_j} = \begin{bmatrix} -\mathbf{N}^k \\ -\mathbf{t}^k_1 \\ -\mathbf{t}^k_2 \end{bmatrix}, \tag{2.196}$$

$$\mathbf{\Omega}^k_{J_i} = \begin{bmatrix} \mathbf{x}_c - \mathbf{X}_i \times \mathbf{N}^k \\ \mathbf{x}_c - \mathbf{X}_i \times \mathbf{t}^k_1 \\ \mathbf{x}_c - \mathbf{X}_i \times \mathbf{t}^k_2 \end{bmatrix}, \tag{2.197}$$

$$\mathbf{\Omega}^k_{J_j} = \begin{bmatrix} -\mathbf{x}_c - \mathbf{X}_j \times \mathbf{N}^k \\ -\mathbf{x}_c - \mathbf{X}_j \times \mathbf{t}^k_1 \\ -\mathbf{x}_c - \mathbf{X}_j \times \mathbf{t}^k_2 \end{bmatrix}. \tag{2.198}$$

Se as restrições de contato forem violadas então um vetor corretivo de erro, $\mathbf{b}^k_{contact} \in \mathbb{R}^{1+\beta}$, pode ser expresso como

$$\mathbf{b}^k_{contact} = k_{cor} \begin{bmatrix} d^k_{penetration} \\ \mathbf{0} \end{bmatrix} \tag{2.199}$$

onde $d^k_{penetration}$ é a profundidade da penetração. As observações pertinentes aos padrões das submatrizes tanto das restrições de contato como das de junção, nos permite criar os tipos de restrições utilizando praticamente as mesmas estruturas de dados.

2.12 Comentários Finais

Neste capítulo foram introduzidos conceitos básicos da dinâmica dos corpos rígidos. Partimos inicialmente do movimento de partículas. Introduzimos, em seguida, restrições ao sistema mostrando a relação entre dinâmica dos corpos rígidos e de partículas. Partindo desta relação, foi possível mostrar como chegar às equações de movimento dos corpos rígidos. Incluímos dois tipos de restrições de movimentos aos corpos rígidos, necessárias para prover maior realismo à cena no quesito animação dinâmica: restrições de junção e de contato. Incluímos atrito às restrições de contato e mostramos, então, como é possível unificar as restrições de junções e contatos em um sistema a ser solucionado como um único PCL. Por fim, mostramos como encontrar as matrizes jacobianas para as junções presentes no motor.

CAPÍTULO 3

Arquitetura do Motor de Física

3.1 Introdução

A arquitetura do motor de física proposta nesta dissertação pode ser dividida em três componentes principais, os quais chamaremos de Engine, Integrator e LCPSolver. Além destes, há um componente adicional, que chamaremos de Collision, necessário para detecção de colisão entre os corpos do atores; porém Collision não é parte do motor de física, é apenas um componente a parte que auxilia o motor de física a desempenhar sua tarefa. Engine é o componente responsável por organizar e encapsular todos os outros, ou seja, o ponto de partida para iniciarmos uma simulação. Além disso é neste componente que está presente a API do motor de física, necessária para que o motor 3D consiga utilizá-lo. LCPSolver, ao receber os pontos de contato encontrados por Collision, tem a tarefa de encontrar forças de restrição de modo que elas mantenham a integridade das propriedades restritivas dos corpos rígidos. O algorítmo usado pelo LCPSolver é chamado de SOR LCP. Feito isso, o componente Integrator deve atualizar as velocidades dos corpos rígidos integrando as acelerações obtidas e também atualizar posições e orientações integrando suas velocidades.

Lógicamente, há uma sequência de inicializações que devem ser realizadas antes de ativarmos uma simulação. Por conta disto, o motor 3D conta com classes manipuladoras de corpos rígidos, junções e formas que fazem a interface com o motor de física, permitindo que algumas destas inicializações sejam efetuadas no momento da criação de atores e seus atributos ainda fora do motor de física. Esta interface é implementada através de métodos externos os quais invocam funções da API do motor de física. Adicionalmente, há classes que representam corpos rígidos, formas e junções também no motor 3D, porém estas contém atributos e propriedades, em sua maioria, diferentes das classes presentes no motor de física adequadas à natureza de cada componente.

Descrevemos neste capítulo, a sequência de passos envolvidos na inicialização, execução e obtenção dos resultados de uma simulação dinâmica. Apresentamos também, as principais classes de objetos envolvidos na arquitetura do motor juntamente com seus mais importantes membros. Ao descrever os passos executados pelo motor de física, mostramos alguns importantes métodos ativados pela API do motor de física e qual o papel de cada um. Além disso, conforme avançamos na descrição da sequência de passos realizados pelo motor de

física, detalhamos algorítmos responsáveis pela resolução do PCL, integração das equações de movimento e pela atualização dos estados dos objetos do motor 3D após cada iteração da simulação.

3.2 Inicialização do Motor de Física

O motor de física é compostos por uma biblioteca de classes e funções responsáveis pelo cálculo da dinâmica e pela API pela qual podemos acessá-lo. Para manipular essa API, o motor 3D conta com classes implementadas em `C#` responsáveis por ligar o motor 3D ao motor de física, além de conduzir a criação e simulação dos atores de modo organizado. Estas classes são conhecidas como objetos `eShapeSpace`, `eJointSpace` e `eRigidBodySpace`, detalhados mais adiante.

As bibliotecas do motor de física são implementadas em `C++`. `C++` foi escolhida como linguagem para implementação desse componente porque, além de ser uma linguagem amplamente conhecida cientifica e academicamente, oferece uma performance adequada ao propósito, que é utilizar o motor de física para aplicações em tempo real; calcular a animação dinâmicamente é uma tarefa computacionalmente intensiva, para a qual `C++` é indicada. Além disso, como o desenvolvedor não pode realizar alterações em métodos mais baixo nível pertencentes ao motor de física, tudo é invocado a partir da API sem a necessidade de utilizar a mesma linguagem do motor 3D. Devido a estas características, não faz-se necessária a utilização de coleta de lixo automática, e outras vantagens que linguagens como `C#` oferecem.

A linguagem `C#` permite que invoquemos métodos ou funções externas presentes em uma *lib* ou *dll*, contanto que sejam prototipadas previamente. Este processo é utilizado para invocação da API do motor de física cujo código foi descrito em `C++` e compilado na forma de uma *dll*. Em `C#`, as chamadas à API do motor 3D são declaradas como métodos externos dentro das classes responsáveis pelo gerenciamento dos objetos relacionados a dinâmica dos corpos rígidos.

O motor de física recebe, a cada chamada, o passo de tempo sobre o qual deverá realizar a simulação, o tempo atual, e óbviamente todos os atores e junções existentes naquele momento. A entrada para a chamada é realizada pelo método da API `void PerformSimulation(float time, float dt)`, podendo ser invocado pela cena, em `C#`, através de um método externo de mesma assinatura, presente na classe `Scene`.

3.2.1 Gerenciador do Motor de Física

Dentro do motor de física, temos declarados globalmente em `C++` o ponteiro para objeto `Engine* engine`. A classe `Engine` do motor de física, é encapsulador de objetos do tipo `ShapeSpace`, `JointSpace` e `RigidBodySpace`, e também dos principais componentes do motor de física; é através de um objeto desta classe, em `*engine`, que iniciamos a sequência de passos que calcula o novo estado dos corpos rígidos. Como ilustrado pela Figura 3.1, `Engine` possui além dos membros citados, a declaração do ponteiro `collision` para um tipo `Collision`. `Collision` é a classe criada para realização da detecção de colisão do motor; pode ser vista como um componente que ajuda o motor de física, calculando os pontos de contato das colisões entre os corpos rígidos. Apesar de conceitualmente não fazer parte de um motor de física, é necessário uma referência à algum detector de colisão para que saibamos quais serão os pontos de contato entre os atores da cena. Outro atributo importante de `Engine` é o ponteiro para um objeto `Integrator`, chamado de `integrator`. O objeto em `*integrator` é quem realiza a integração da equação de movimento e atualiza os estados dos corpos rígidos, como descrito no Capítulo 2. Como métodos importantes temos dois métodos

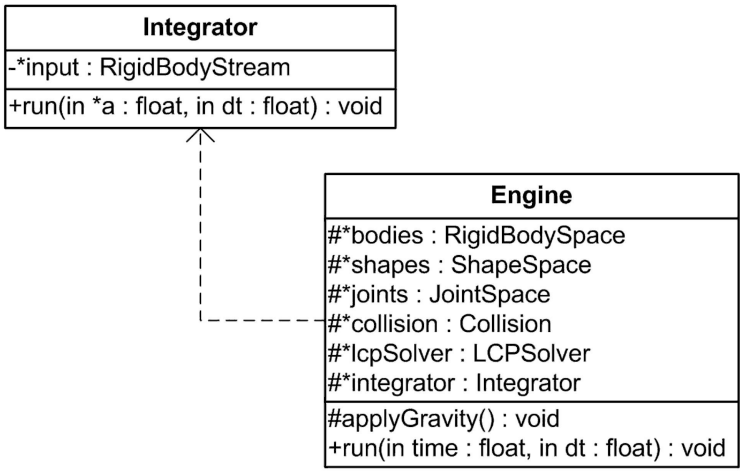


Figura 3.1: Representação da estrutura estática de Engine.

virtuais: `void applyGravity()` e `void run(float time, float dt)`. O primeiro aplica a aceleração da gravidade à todos os corpos rígidos do motor de física. O segundo é o ponto de entrada para o início da simulação; dentro deste método estão presentes as chamadas que invocam o detector de colisões, encontram e solucionam as restrições, e integram o movimento dos corpos rígidos.

O objeto apontado por `engine` deve ser, antes de tudo, instânciado através da API. O comando da API responsável por isto é `void InitializePhysics(bool useGPU)`. Onde o parâmetro `useGPU` é um valor booleano que diz se o motor de física será instânciado para execução somente em CPU, ou se deve utilizar CUDA para realizar seus cálculos paralelamente utilizando GPU, neste caso, utilizado em conjunto com uma implementação em CUDA, desenvolvida por [Per08].

3.2.2 RigidBodySpace

Um `RigidBodySpace` é um container gerenciador de corpos rígidos, Figura 3.2. Nesta classe estão presentes os principais métodos da API relativos à manipulação de corpos rígidos. `RigidBodySpace` deriva de `RigidBodyStream`. O principal atributo de `RigidBodyStream`, por sua vez é um atributo da classe `RigidBodyData`. `RigidBodyData` é composto de conjuntos de ponteiros para diversas listas de informações sobre os corpos rígidos. As listas contém propriedades dos corpos rígidos, ordenadamente. Por exemplo, `*invMass` é uma lista contendo inverso da massa do primeiro até o n-ésimo corpo rígido; `*invInertia` contém as matrizes do inverso do tensor de inércia para os mesmos n corpos, na mesma ordem. Dentre estas listas de atributos destaca-se a lista `*state` que contém referências para uma estrutura que representa o estado de um corpo rígido (Capítulo 2) composto por posição, orientação, velocidade angular e velocidade linear. `RigidBodyStream` acrescenta à sua classe base, métodos para facilitar a manipulação destas listas como se fossem, na realidade, fluxos de dados dos corpos rígidos.

Uma observação importante é que, um objeto RigidBodyData também é atributo declarado na classe `RigidBody`, abstração de um corpo rígido no motor de física. Porém, diferentemente da classe `RigidBodyStream`, onde `RigidBodyStream::RigidBodyData` contém ponteiros para uma lista de informações sobre todos os corpos rígidos, os ponteiros do objeto do tipo `RigidBody::RigidBodyData` contém ponteiros para as informações do próprio objeto do tipo `RigidBody`. Por exemplo, `*invMass` contém um ponteiro para a posição da lista de inversos da massa em que se encontra o inverso da massa para a id do objeto `RigidBody`

ao qual o atributo pertence. Resumindo, cada ponteiro aponta para a posição dos atributos do próprio objeto apenas, entretanto, as informações são compartilhadas entre o objeto `RigidBodyStream` e `RigidBody`.

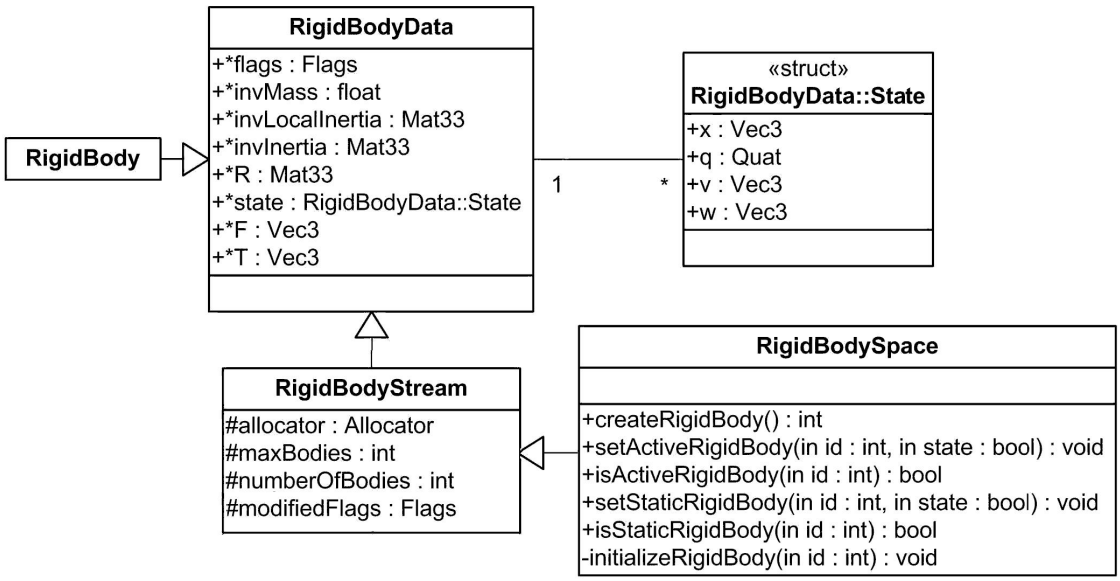


Figura 3.2: Representação da estrutura estática da classe RigidBodySpace.

O motor 3D conta com uma classe de nome parecido, `eRigidBodySpace`, porém em `C#`. Esta classe é a responsável por gerenciar os corpos rígidos em um nível mais alto. Ela faz a interface através da qual o motor 3D comunica-se com o motor de física. Ela é quem invoca os métodos da API de `RigidBodySpace` do motor de física; estes métodos são declarados como métodos externos, e acoplam diretamente o método correspondente de `RigidBodySpace` em `C++`. É importante ressaltar que, dentro desta classe não temos membros específicos para o cálculo da dinâmica como temos na classe `RigidBodySpace` em `C++`; por exemplo, o inverso do tensor de inércia de um corpo rígido não existe em `C#`, mas a massa, densidade e pose local existem. As propriedades usadas no nível mais alto do motor 3D são aquelas que podem ser criadas ou modificadas pelo usuário; no caso das propriedades que são usadas apenas para computação e utilizadas nos cálculos da dinâmica, não é necessário representá-las nos objetos dentro do motor 3D, somente dentro do motor de física. Entretanto, o desenvolvedor pode sim, perguntar por estes atributos únicos ao motor de física. Neste caso o motor resgata do motor de física o valor do atributo desejado e o retorna ao desenvolvedor.

Importante classe gerenciadora dos corpos rígidos no motor, `eRigidBodySpace` apresenta alguns dos seus principais métodos externos importados do motor de física. São eles:

- `static extern int CreateRigidBody()`- Quando o usuário cria um corpo rígido, além de criar um objeto `eRigidBody` no motor 3D, este método que inicializa no motor de física, o fluxo de informações e estados na lista de propriedades dos corpos rígidos para este ator, presente em `RigidBodySpace` do motor de física.

- `static extern void SetActiveRigidBody(int, bool)` - Modifica flags do corpo rígido para que ele seja visto como um objeto ativo, e seja assim, dinâmico. Caso um corpo rígido não esteja ativo, ele não participa dos cálculos que atualizam os estados dos corpos rígidos. Passa como parâmetro um id que representa o identificador do corpo rígido e um booleano inicando se o mesmo está ativo ou não.

- `static extern void SetStaticRigidBody(int, bool)` - Seta o corpo rígido como

sendo um objeto estático. Um objeto estático não sofre forças e nem deslocamentos. Os argumentos são os mesmos de `SetActiveRigidBody()`.

- `static extern void SetPose(int , ref Matrix34 )` - Seta a posição e orientação do corpo rígido. O segundo argumento é do tipo `Matrix34`, a classe que corresponde à uma matriz 3x4, e dentro desta matriz estão representadas posição(uma parte 3x1) e orientação (uma parte 3x3). `pose` será portanto a posição e orientação a ser atribuída ao corpo rígido.

- `static extern void SetLinearVelocity(int, ref Vector3)` - Seta a velocidade linear do centro de massa do corpo rígido representado pelo identificador passado como argumento 1. A velocidade é o segundo argumento. `Vector3` é a classe utilizada no motor 3D para representar um vetor ou ponto tridimensional no espaço afim.

- `static extern void SetAngularVelocity(int id, ref Vector3 w)` - Seta a velocidade angular do corpo rígido como sendo `w`.

- `static extern void AddForce(int id, ref Vector3 force)` - Aplica uma força com intensidade e direção `force` no corpo rígido de `id`.

- `static extern void AddTorque(int, ref Vector3)` - Aplica uma torque com intensidade e direção dados pelo argumento do tipo vetor no corpo rígido de identficador representado pelo argumento 1.

Adicionalmente há mais métodos para setar e obter valores dos objetos da classe, porém não há necessidade de listá-los aqui, pois seguem a mesma linha de pensamento dos anteriores. Em `C++`, os métodos correspondentes são invocados através do objeto `*bodies` do tipo `RigidBodySpace`, atributo de `*engine`.

3.2.3 ShapeSpace

Um objeto `ShapeSpace` é análogo à um objeto `RigidBodySpace`, mas como um container de formas, Figura 3.3. Nesta classe estão presentes os principais métodos da API relativos à manipulação das formas dos corpos rígidos. `ShapeSpace` deriva de `ShapeStream` que por sua vez deriva de `ShapeData`. `ShapeData` é uma composição de formas. Uma forma no motor de física é representada pela classe `Shape`. Os principais atributos de `Shape` são:

- `int typeId` - Identificador que representa o tipo de forma. Pode ter o valor de `TypeId::Sphere`, `TypeId::Box`, `TypeId::Plane` ou `TypeId::Capsule`, as quais representam as primitivas esferas, caixa, plano e capsula respectivamente.

- `int bodyId` - Inteiro identificador do corpo ao qual a forma pertence.

- `int collisionGroupId` - Identificador do grupo de colisão. Formas pertencentes à um mesmo grupo de colisão não colidem entre si. Este teste é realizado no módulo detector de colisões.

- `Matrix34 localPose` - Representa a pose local da forma. Em coordenadas locais com origem no ponto central do frame da forma, a matriz representa sua orientação e posição.

- `float boundRadius` - Raio do volume limitante utilizado na detecção de colisão. Para este projeto, o volume limitante usado tem forma esférica.

- `union SphereInfo, BoxInfo, PlaneInfo, CapsuleInfo` - União que armazena alguns atributos sobre o tipo de shape. O objeto Shape só pode ter um dos tipos descritos, portanto assumirá como atributo apenas um dos 4 tipos.

A classe `ShapeData`, base de `ShapeStream`, declara o atributo `*shape`, que é ponteiro para o primeiro elemento da lista de formas do motor de física. Em outras palavras, `ShapeData` declara a lista de formas. Já `ShapeStream` por sua vez, declara atributos que permitem manipular esta lista como um fluxo de dados: armazena o número de formas, o número máximo permitido, flags de modificação, número de formas *bounded*. Formas no motor de física podem ser de dois tipos em relação ao volume limitante que ajuda na detecção de colisão: *bounded* ou *unbounded*. No caso, formas bounded são aquelas que possuem algum volume limitante; formas unbounded não possuem este volume limitante, fazem parte de um caso especial no qual este volume não é necessário e testes de colisão sempre são realizados para estes tipos. Um exemplo de forma unbounded seria um plano, o qual possui comprimento e largura infinitos, porém não possui altura. `ShapeSpace`, derivada de `ShapeStream` contém

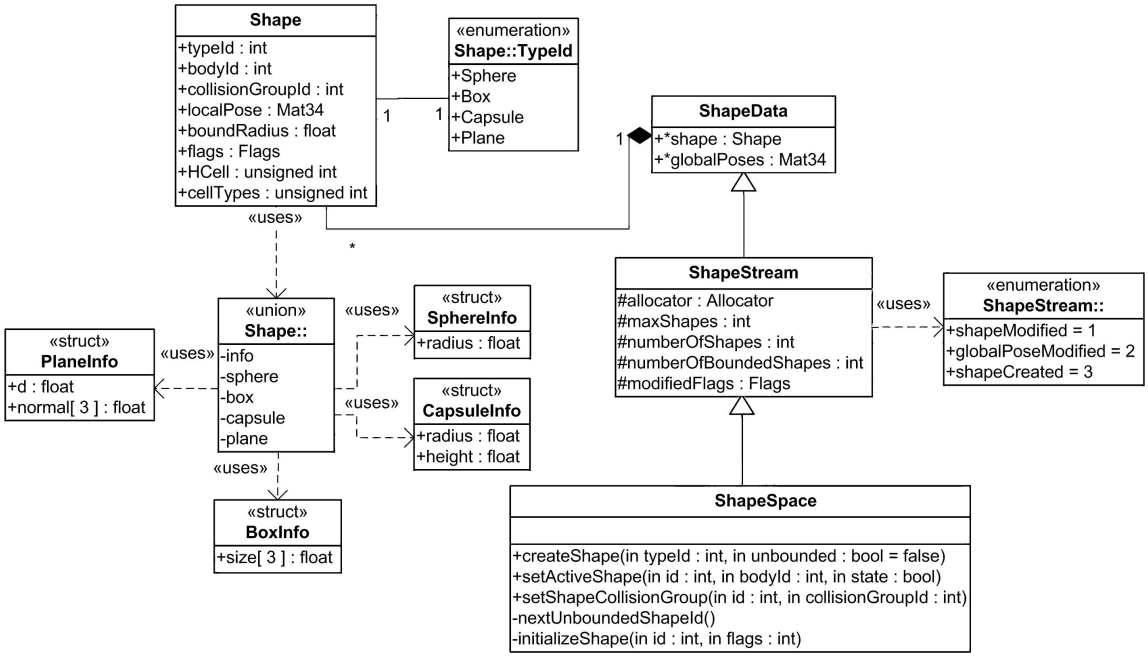


Figura 3.3: Representação da estrutura estática da classe ShapeSpace.

a API utilizada para comunicação com o motor 3D. Da mesma maneira que ocorre com `RigidBodySpace` e `eRigidBodySpace`, `ShapeSpace` possui uma classe de nome parecido, `eShapeSpace` em `C#`, dentro do motor 3D. `eShapeSpace` é quem governa a criação das formas no motor e no motor de física. Quando o desenvolvedor cria uma forma, o objeto `eShapeSpace` cria uma representação da forma no motor 3D (instância da classe `eShape` em `C#`); esta representação conta apenas com atributos que podem ter alguma importância ao desenvolvedor. Além disso, `eShapeSpace` utiliza a API de `ShapeSpace` para criar e adicionar à lista de formas do mesmo mais um objeto do tipo `Shape`.

Alguns dos principais métodos externos de `eShapeSpace`, importados do motor de física são:

- `int createShape(int typeId, bool unbounded)` - Quando o usuário cria uma forma, além de criar um objeto do tipo `eShape` no motor 3D, este método que ini-

cializa no motor de física, os atributos da forma presente na lista de formas do motor de física, `*shape` presente em `ShapeSpace`. O parâmetro `typeId` é o código que indica qual o tipo de primitiva que representa a forma (esfera, caixa, plano ou capsula), o segundo parâmetro indica se será uma forma *unbounded*, o que ocorre caso seu valor seja `true`; para o valor `false` a forma será *bounded*. Este parâmetro tem `false` como valor default.

- `void setActiveShape(int id, int bodyId, bool state)` - Seta as flags do objeto `Shape` indicada pelo índice `id` para que indique que a mesma está ativa ou inativa. Uma forma desativada não faz parte da etapa de detecção de colisão. Este método coloca também a forma no mesmo grupo de colisão que seu corpo; o segundo parâmetro passado como argumento deve ser o índice identificador do corpo ao qual a forma pertence. O parâmetro `state` é um booleano que, se tiver valor verdadeiro, define a forma como ativa, caso contrário inativa.

- `int nextUnboundedShapeId() const` - Retorna o índice identificador da lista de formas da próxima forma unbounded. Unboundeds são armazenadas sempre do final da lista de formas em direção ao início (ao contrário das boundeds, que iniciam-se no início da lista em direção ao final).

- `void initializeShape(int id, int flags)`- Inicializa atributos da forma de índice `id`, de acordo com seu tipo que pode ser `Shape::Sphere`, `Shape::Box`, `Shape::Plane` ou `Shape::Capsule`. As flags recebidas como segundo parâmetro também são atribuídas às flags da forma.

3.2.4 JointSpace

Ainda falta criarmos e inicializarmos o objeto presente em `JointSpace *joints` de `*engine`. Uma `JointSpace` é um container onde são armazenadas informações sobre todas as restrições presentes em cada junção do sistema. Um `JointSpace` tem informação sobre o número de junções e o número total de restrições provocadas por essas junções. Adicionalmente, herda de `JointData`, um ponteiro para uma lista de objetos `ConstraintInfo`, chamado `*constraints`, Figura 3.4. Cada objeto `ConstraintInfo` desta lista tem informações relativas a cada restrição entre dois atores, dentre elas: índice dos dois atores relacionados, a jacobiana da restrição, informações sobre atrito, entre outros. Cada junção pode ter um número diferente de restrições, mas é importante que saibamos à qual junção cada restrição pertence. Para tal, a classe `JointData` também conta com um ponteiro, chamado `joints`, para uma lista que contém a soma prefixa do número de restrições de cada junção; resumidamente, para a i-ésima junção , indicada na posição i, onde iniciam-se suas restrições dentro da lista `*constraints`. O comando da API responsável por adicionar essas restrições à `JointSpace` é `bool AddConstraintInfo(int count, ConstraintInfo[ ] info)`, onde `count` é número de restrições da junção e `info` é um vetor contendo estas restrições. A invocação dessa API, no motor do projeto é responsabilidade da classe estática chamada `eJointSpace`.

3.3 Execução da Simulação

Criados os corpos rígidos, formas e junções do motor de física, podemos iniciar a simulação invocando o método `void PerformSimulation( )`. Este método pertence a API do motor de física, e nada mais faz do que invocar o método `Engine::void run( )` do objeto `*engine`.

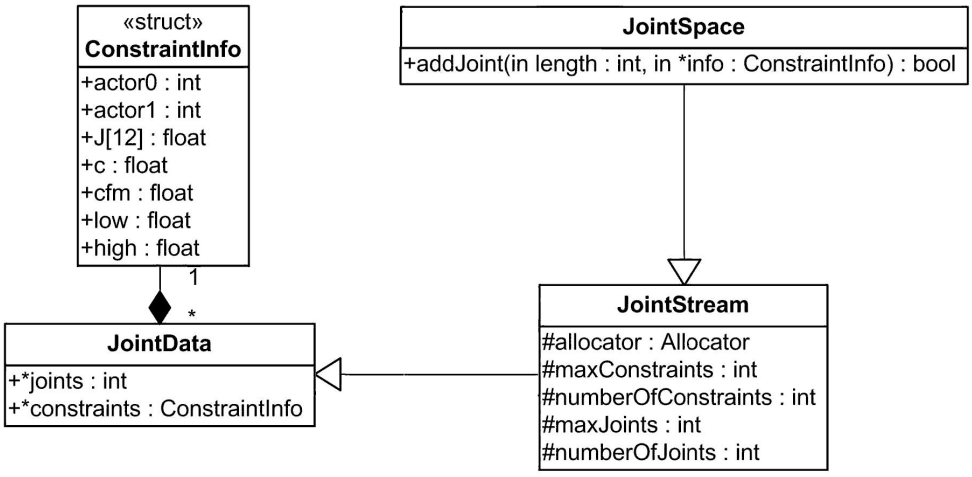


Figura 3.4: Representação da estrutura estática de JointSpace.

Recebe um argumento `float time` que refere-se ao tempo corrente da simulação, e outro argumento, `float dt`, que é o passo de tempo sobre o qual ocorrerá o movimento.

O primeiro passo de `PerformSimulation()` é invocar o módulo detector de colisões. Os objetos presentes em `*bodies` e `*shapes` são passados ao detector de colisões, `*collision`. Este por sua vez realiza três etapas:

1. Primeiro, copia a lista de corpos rígidos e de formas para sua própria estrutura de dados. Instancia também todas as estruturas de dados das células de colisão. Células de colisão é uma forma de subdivisão espacial para que alguns testes de colisão sejam eliminados, [Per08].

2. A segunda etapa é a execução da fase geral da detecção de colisão; nesta etapa são retornados pares de atores que possivelmente colidirão (mas não certamente), os pares de atores eliminados são os que, com toda certeza, não estão ou estarão colidindo.

3. Para os pares de atores retornados, é invocada a fase reduzida da detecção de colisão; nesta fase, é realizada a colisão precisa entre as formas dos atores envolvidos. Caso realmente haja colisão, são calculados os pontos de contato, e estes são adicionada à uma lista de contatos. Entre as informações da cada contato (formado por dois pontos de contatos) da lista, estão a posição onde houve o contato, normal no ponto de contato, profundidade de penetração, índice dos dois corpos envolvidos, parâmetros da superfície (física), entre outros.

Cada contato é, na realidade, uma junção de contato; esta abordagem é utilizada porque fica mais direta a montagem do PCL se tratarmos junções e contatos de maneira uniforme. Contatos são representados pela estrutura de nome `Contact`. Como atributos importantes de `Contact` podemos citar `SurfaceParameters surface` e `ContactGeom geom`. `Surface` é a classe que representa as propriedades físicas do material da superfície do corpo rígido; dentre eles os coeficientes de atrito (`mu` e `mu2`) e de ressalto (`bounce` e `bounce_vel`). `ContactGeom` contém a representação das informações sobre a geometria das superfícies em cotato; dentre seus principais atributos estão:

- `Vec3 pos` - `Vec3` é a classe utilizada no motor de física (mesma função de `Vector3` no motor 3D, em `C#`) para representar um vetor ou ponto tridimensional no espaço afim. O objeto `pos` armazena a posição, em coordenadas globais, do local onde ocorre o contato.

- `Vec3 normal` - Armazena o vetor normal à superfície de um objeto ao qual pertence o ponto de contato. Como um ponto de contato armazena o contato sempre entre dois objetos, portanto duas superfícies, por convenção adotamos a regra de que a normal refere-se à superfície do ator de menor índice na lista de atores.
- `float depht` - Armazena a profundidade de penetração. Dado o ponto de contato, é a medida que a superfície de um dos atores adentrou à superfície do outro.
- `int rb0, rb1` - `rb0` e `rb1` são os índices dos dois atores envolvidos no ponto de contato. `rb0` contém sempre o ator de menor índice na lista de atores.

3.3.1 Forças de Restrição

No próximo passo da simulação, é criada então uma lista de objetos `Contact`, preenchida obtendo-se os contatos calculados por `*collision`. `Engine` possui declarado o atributo `lcpSolver* LCPSolver` (instânciado no construtor de `Engine`), objeto responsável por solucionar o PCL. Os principais atributos e métodos declarados em `LCPSolver` e `Contact` são mostrados em Figura 3.5. Observe na Figura 3.5 que, `LCPSolver` é composto de um objeto

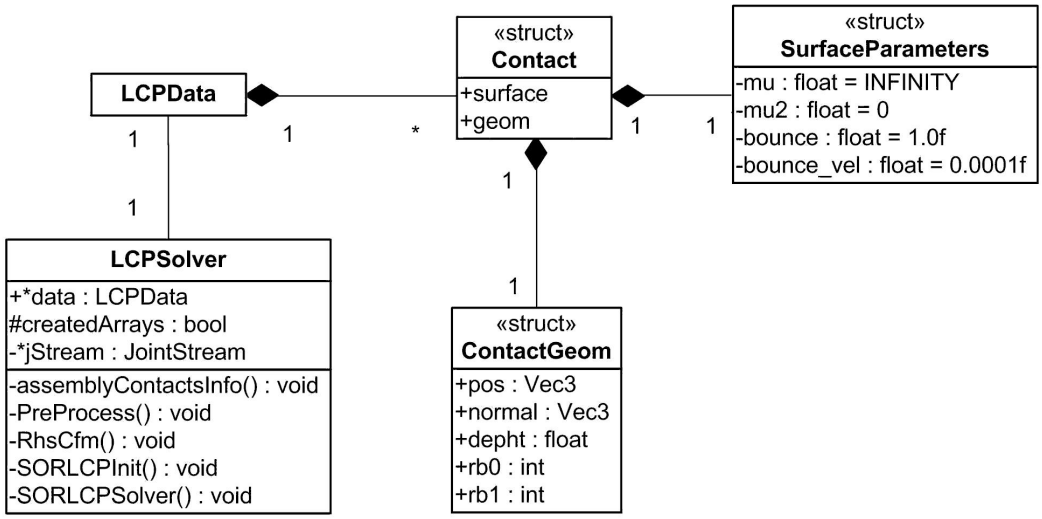


Figura 3.5: Representação da estrutura estática das classes LCPSolver e Contact.

do tipo `LCPData`. `LCPData` mantém informações sobre estruturas de dados do PCL (mais especificamente o algorítmo SOR LCP, descrito no Capítulo 2), tais como número de restrições, número de junções, número de contatos, número de corpos, e ainda outros importantes como,

- `FPS` - Taxa de quadros por segundo da simulação, é calculada como sendo `1/dt`.
- `ERP` - *Error Reduction Parameter*, parâmetro de redução de erros utilizado para corrigir eventuais erros de aproximação das restrições de junção, como proposto no Capítulo 2.
- `float *cforce` - Representa o vetor $\mathbf{x}$, da expressão $\mathbf{w} = \mathbf{A}\mathbf{x} + \mathbf{f}$, do Capítulo 2. Neste vetor serão retornadas os impulsos das restrições encontradas solucionando o PCL.
- `float* lambda` - Representa o valor $\boldsymbol{\lambda}$, da (2.148).
- `float *rhs` - Vetor que representa $\mathbf{f}$, da expressão $\mathbf{A}\mathbf{x} + \mathbf{f} \geq \mathbf{0}$ do sistema.

- `float* iMJT` - Matriz, na forma de um vetor linha a linha. Armazena a multiplicação da matriz de massa inversa pela jacobiana de restrição de cada restrição de cada corpo, resultando ao todo em $\mathbf{M}^{-1}\mathbf{J}^T$, Capítulo 2, que é parte da matriz $\mathbf{A}$.

- `float* Ad` - Vetor com os elementos da diagonal da matriz $\mathbf{A}$, para facilitar os cálculos do SOR LCP.

- `ConstraintInfo* cInfo` - Vetor com informações do tipo `CostraintInfo` de cada restrição de junção.

- `Contact* contacts` - Lista de contatos, fornecida anteriormente pelo detector de colisões. É necessária para a montagem das restrições de contato, tratadas como uma junção de contato.

O passo seguinte é invocar o método `setInput(RigidBodyStream& bSt, JointStream& jSt, Contact* contact, int numberOfContacts)` do objeto `*lcpSolver`; onde `bSt` é a lista de corpos rígidos de `Engine`,(`*bodies`); `jSt` é a lista de junções de `Engine`,(`*joints`); `contact` é a lista de contatos retornados pelo detector de colisões; e `numberOfContacts` é o número de elementos desta lista. Este método inicia o `LCPData`(cuja instância é armazenada em `*data`) do objeto `*lcpSolver`, com os devidos valores recebidos como parâmetros. Devemos também invocar também, o método `void SetProperties(float w, int iteration, float dt, Vec3 gravity)`, que trata de setar outras propriedades importantes do PCL como o fator de relaxamento do SOR LCP `w_sor` com o parâmetro `w`, o número máximo de iterações com o parâmetro `iteration`, o passo de tempo com `dt`, e o valor da aceleração da gravidade com `gravity`.

Após a montagem dos dados, devemos invocar o método `LCPSolver::void run()` de `*lcpSolver` para que o PCL seja solucionado. `run()` aloca memória necessária para `*cforce` e `*lambda` presentes em `*data`. Neste método os seguintes passos são efetuados:

- `*lcpSolver` invoca seu método `void assemblyContactsInfo()`. Neste, para cada contato em `*contact`, 3 estruturas `ConstraintInfo` são computadas: uma correspondente à restrição de contato propriamente dita, e as outras duas estão relacionadas à restrição imposta pelo atrito onde cada uma é representada por um versor tangente do cone de atrito ($\eta = 2$ neste caso), de acordo com o Capítulo 2. Após calculadas, as 3 estruturas `ConstraintInfo` são adicionadas ao objeto `*jStream` de `*lcpSolver`. Na realidade, o objeto `*jStream` referenciado aqui é do tipo `JointSpace`, especialização de `JointStream`.

- Para cada corpo, atualiza-se o inverso do tensor de inércia (atributo `invInertia` de `RigidBodyData`), pois `R` (orientação do corpo rígido) pode ter sido modificado. Cada corpo rígido cuida de sua própria atualização invocando `updateGlobalInertia()` que simplesmeste faz: `*invInertia = *R* *invLocalInertia *` R^T, onde R^T é a matriz transposta de R.

  Em seguida é computada a parte de cada corpo que contribui para o vetor $\vec{b}$ do PCL, e armazenado em um vetor temporário (`tmp`); no código fonte, $\vec{b}$ é chamado de `rhs`. São armazenados em `tmp` seguidamente os valores $M^{-1} * F_{ext} + velocidadelinear/dt$ e $I^{-1} * T_{ext} + velocidadeangular/dt$; onde F_{ext} são forças atuantes corpo, velocidade linear é um dos seus estados (armazenados em `*state`), I^{-1} é o inverso do tensor de inércia (`*invInertia` de `RigidBody`), T_{ext} é o torque atuante no corpo, e velocidade angular é o estado `w` de `*state`. O método de `LCPSOlver` que realiza estas operações é o método `void PreProcess()`.

- `*lcpSolver` invoca `SORLCPInit()`. Neste método, percorremos a lista de restrições (`*cinfo` do atributo `*data`) e montamos `*iMJT`, $\mathbf{M}^{-1}$ multiplicada pela jacobiana das restrições. O vetor $\vec{b}$ é finalizado também e ao final do processamento de todos os $J\lambda$, $\vec{b}$ estará montado. Além disso, pré-computamos a diagonal da matriz $\mathbf{A}$, introduzimos a correção de erros no vetor $\vec{b}$ e escalonamos $\mathbf{J}$ e $\vec{b}$ pela diagonal de $\mathbf{A}$.

- Por fim, `lcpsolver` invoca `SORLCPSolver()`. Aqui o PCL é calculado como foi mostrado no cálculo do algorítmo SOR LCP convencional. Os resultados, ou acelerações, são retornadas dentro do atributo `*cforce` de `*data`. Estas são as acelerações lineares e angulares necessárias para mantermos a integridade das propriedades dos corpos rígidos.

3.3.2 Integração da Equação de Movimento

Encontradas as forças de restrição por `*lcpSolver`, a próxima etapa é integrar os movimentos respeitando estas forças. Esta é a função do objeto `Integrator *integrator` de `Engine`. A classe `Integrator` declara o atributo `RigidBodyStream* input`; `*engine` invoca o método `void setInput(RigidBodyStream& b)` de `*integrator`, e este faz com que `*input` aponte para o parâmetro `b` recebido pelo método. Em outras palavras, a lista de corpos rígidos de `*integrator` será a mesma de `*engine`.

`Integrator` declara o método `void run(float *Force, float dt)`, onde `*Force` é a lista de impulsos das restrições calculadas por algum objeto `LCPSolver`, mais especificamente o atributo `*cforce` de `*data`, que é atributo de `*lcpSolver`. O segundo argumento é o passo de tempo `dt`. O método percorre todos os corpos rígidos da lista `RigidBodyStreams *input` e invoca `RigidBody::void update(float* a, float dt)` para cada um deles. Onde, `*a` aponta para o início de um vetor que representa duas triplas: a primeira é aceleração linear (3 posições) e a segunda aceleração angular (3 posições), calculadas pelo PCL para o corpo em questão (armazenadas em `*cforce`). A função deste método é atualizar a velocidade linear, angular, posição e orientação do corpo rígido invocador. A classe `RigidBody` herda de sua classe base o atributo `State *state` (Figura 3.2); `*state` armazena o estado do corpo rígido, com a seguinte relação: `x` é a posição do corpo rígido, `q` é a orientação, `v` é a velocidade linear e `w` é a velocidade angular. O método computa a atualização das velocidades integrando as acelerações atuais somadas às das restrições:

- `v = (a + invMass * F)* dt`; e

- `w = ((a+3) + invInertia * T) * dt`,

onde `a` é o vetor contribuição da aceleração linear e angular calculada pelo PCL, `F` é a força atuante no corpo, `T` o torque atuante no corpo, `invMass` e `invInertia` são a inversa da matriz de massa e o inverso do tensor de inércia, respectivemente. Em seguida, atualiza-se a posição, e orientação do corpo rígido,

- `x += v * dt`; e

- `q += Quat(w) * q * (0.5f * dt)`,

onde `Quat` é a classe de objetos que representa um quaternion. Por fim, `q` é transformado em uma matrix 3x3 e atribuido à `R`. Zera-se os acumuladores de força `T` e `F`. Resta limpar a lista `*joints` de `*engine` (pois provavelmente não será a mesma na próxima iteração do laço principal), e retornar.

3.4 Atualizando Atores do Motor

Após a simulação, os corpos rígidos do objeto do tipo `RigidBodySpace` presentes em `*engine` estão atualizados, porém ainda devemos repassar os resultados ao motor 3D que requisitou os cálculos ao motor de física.

Após o término de `PerformeSimulation()`, já fora do motor de física, portanto em `C#` percorremos a lista de atores do motor e invocamos o método `Actor::UpdateGlobalPose()` de cada um. Este método acessa o atributo `body`, representante do objeto `eRigidBody` de cada ator, e este atributo invoca `SetActorGlobalPose(ref Matrix34 globalPose)`, onde `globalPose` é a matriz 3x4 do ator, passada por referência, que representa sua pose global (posição e orientação em relação ao sistema global de coordenadas).

Caso não for um corpo rígido estático (se for estático , não se movimenta, portanto não faz sentido atualizarmos a pose do ator, que nunca mudará), calculamos a pose atualizada para o ator e seu corpo rígido da seguinte maneira:

- Calculamos a matriz inversa da matriz `localPose` do corpo rígido do ator em questão, inversa neste contexto quer dizer uma matriz que faz a transformação espacial contrária à pose local. Como `eRigidBody::localPose` é composta de uma parte de rotação e outra de translação (orientação e posição), para encontrar a inversa basta calcular a transposta da parte da rotação e inverter o sinal da translação, compondo a matriz novamente a seguir. Chamaremos a matriz inversa de `localPose` de `invLocalPose`.

- O problema a ser resolvido agora é o seguinte: Quando o motor de física foi invocado para prover movimento aos corpos rígidos, a pose sobre o qual ele realiza a integração é o frame de massa do corpo rígido. Frame de massa é a pose do corpo rígido em relação ao sistema global de coordenadas, mudá-lo por si só não provê movimento ao corpo rígido; o frame de massa do i-ésimo corpo rígido fica armazenada externamente no motor de física no atributo `RigidBodySpace* bodies` do objeto `*engine`, e é formado pelos atributos `R[i](rotação)` e `state[i].x (posição)` (atributos da lista de corpos `*bodies`), onde `i` é o índice do i-ésimo corpo rígido.

  Fatalmente, como as coordenadas globais do corpo (frame de massa) rígido são dependentes do seu ator, para atualizar o ator não basta atribuir a transformação sofrida, porque esta foi computada em relação a um sistema global. Para tal, temos que realizar o caminho inverso, anulando a transformação local do corpo rígido e ai sim aplicar a transformação global: efetuamos o movimento local inverso do corpo rígido, levando-o à origem de seu sistema (no caso o frame do ator), depois aplicamos o deslocamento global obtido pelo motor de física; isto levará o ator à posição global correta.

  A transformação inversa do sistema local do corpo rígido é `invLocalPose` que calculamos no item anterior. O frame de massa tem que ser resgatado do motor de física; `eShapeSpace` possui declarado o método externo `static extern void GetPose(int id, out Matrix34 massglobalpose)` que faz esse papel. Onde, `id` é o índice identificador do corpo rígido desejado, `massglobalpose` é a matriz onde a pose será retornada. Finalmente, a composição de matrizes que forma a pose do ator é `massglobalpose * invLocalPose`. Espacialmente falando, isto quer dizer que efetua-se primeiro a transformação presente em `invLocalPose` seguida da transformação de `massglobalpose`.

- Percorremos a lista de formas do corpo rígido, chamada `shapes`, e cada uma invoca o método `void UpdateGlobalPose(Matrix34 globalPose)`, onde globalPose é a globalPose do ator, que acabou de ser atualizada. Neste método, invoca-se o método de classe de `eShapeSpace` de assinatura `static extern void UpdatePose(int id, ref Matrix34 m)`, prototipação do método externo (portanto implementado no motor

de física) `ShapeSpace::UpdateShapePose(int id, ref Matrix34 m)`. Onde `id` é o índice identificador da shape chamadora e `m` é a matriz `globalPose`, passada por referência.

- Dentro do motor de física, `ShapeSpace *shapes` de `*engine` é possuidora da lista de poses globais das formas — lembrando que `ShapeSpace` possui declarada a lista de poses globais ordenada por índice de todas as formas, a lista é armazenada em `*globalPoses`, herdado de `ShapeData` — que deve ser atualizada. Cada pose global de cada shape é atualizada com a pose recebida como argumento multiplicada pela própria pose local da forma (óbviamente será a pose local desta shape tomada em relação à pose global do ator que a possui), em outras palavras, a pose global da shape é obtida efetuando suas transformações locais, seguida das transformações do seu corpo rígido. O atributo `ShapeStream::flags` de `*shapes` é modificado com um valor indicando que houve mudança nas poses das formas .

Ao final, temos atores, seus corpos rígidos e todas formas com a pose atualizada conforme os dados provenientes do motor de física.

3.5 Comentários Finais

Neste capítulo apresentamos a arquitetura e funcionamento do motor de física, componente do motor 3D. Apresentamos a sequência de corpos rígidos, formas e junções que devem ser inicializadas antes de iniciarmos a simulação propriamente dita. Feito isto, mostramos a sequência de passos que inicia-se a partir do método que dá início à simulação, `PerformeSimulation()`; ao receber como parâmetros o tempo corrente e o passo de tempo desejado para a iteração, o método, através da classe Engine, inicia uma sequência de 3 passos distintos: detectar colisões, solucionar um PCL para manter a integridade dos corpos rígidos, e integrar o movimento dos mesmos. Mostramos que o motor de física, dentro da classe Engine conta com o objetos do tipo Collision, LCPSolver e Integrator, responsáveis por executar cada um dos 3 passos descritos, respectivamente. Collision efetua a detecção de colisão em duas etapas: uma fase geral, onde são descartados elementos que não entrarão em colisão com toda certeza testando-se seus volumes limitantes, e uma fase reduzida, onde caso realmente haja colisão, são calculados os pontos de contato para cada uma. LCPSolver, recebe os pontos de contato calculados pelo objeto do tipo Collision, monta as restrições de movimento geradas por elas na forma de um PCL e o resolve; a solução deste problema retorna acelerações necessárias para manter a integridade das restrições. Enfim, o objeto do tipo Integrator recebe estas acelerações, que são integradas em relação ao tempo juntamente com o restante das forças atuantes em cada corpo rígido, gerando os novos estados do corpo rígido: velocidade linear, velicidade angular, posição e orientação.

CAPÍTULO 4

Implementação do Ambiente

4.1 Introdução

A arquitetura do motor 3D proposta nesta dissertação difere da arquitetura do ambiente original utilizado na LA original, tanto na estrutura dos componentes da cena, como na política de sincronização dos seqüenciadores. Isto ocorre porque para a geração da animação em tempo real, a arquitetura original não é adequada. A geração de código objeto, não é mais baseada na MVA; nesta nova versão, o código descritivo recebido do usuário é transformado em uma linguagem intermediária, `C#`, e depois compilado gerando código MSIL. O código fonte no qual foi implementado o motor 3D é autônomo, ou seja, qualquer parte que seja (estruturas, métodos, ou tipos de dados, por exemplo) não pertencem a um jogo ou simulação específica. O código pode ser utilizado para outros projetos, ou até mesmo outros motores, sem que seja necessário remover partes dele. Apesar de internamente os componentes do motor não funcionarem do mesmo modo que no projeto original de [Oli06], as clases de objetos da API de animação, em sua maioria, ainda podem ser descritas em LA normalmente. As exceções são as classes que manipulam eventos do usuário e as que descrevem roteiros.

Neste capítulo, abordamos a arquitetura, componentes e classes responsáveis pelo funcionamento do motor 3D. Apresentamos também os objetos que constituem uma cena a ser animada. Estes objetos são representados por conjuntos de classes de objetos, decritos em linguagem `C#`; adicionalmente, abordamos os conceitos necessários para a definição das entidades responsáveis pelo fluxo e controle de uma animação: seqüenciadores e eventos. Apresentamos como se dá o fluxo de dados dentro do laço principal do motor 3D, qual a sequência de passos de execução, e quais políticas de sincronização foram adotadas dentro deste laço. A seguir, mostramos como foi realizada a modelagem das principais classes do motor 3D. É mostrada qual a relação entre atores, corpos rígidos e suas propriedades, junções dentro do motor de física, assim como a estrutura de atores, luzes, materiais e janela de renderização dentro do motor de renderização. Ao descrever o funcionamento das classes de objetos, paralelamente são introduzidas as novas expressões da linguagem LA, estendida neste projeto.

4.2 Arquitetura do Ambiente

O ambiente é composto de uma interface gráfica, onde o usuário é capaz de criar, modificar, salvar e abrir arquivos texto os quais descrevem as cenas, roteiros e ações da simulação. Neste ambiente é possível descrever, compilar e executar o roteiro desejado. Após a compilação, a exibição gráfica da simulação é feita em uma janela do próprio ambiente. A representação gráfica dos atores é mostrada utilizando-se OpenGL [Gro]. OpenGL é uma API gráfica composta por um conjunto de algumas centenas de funções, que fornecem acesso a praticamente todos os recursos o hardware de vídeo. Internamente, age como uma máquina de estados, que de maneira bem específica dizem à OpenGL o que fazer.

Como o principal objetivo do simulador é exibir o comportamento dinâmico dos corpos rígidos, não foram criados para os atores modelos gráficos próprios para exibição, apesar do motor de renderização permitir que cada ator tenha seu próprio modelo gráfico. Em vez disso, renderizamos suas formas, que possuem geometria da superfície mais simples e portanto são utilizadas para calcular colisão e dinâmica. Estas formas são transformadas em uma malha de polígonos triângulares e então passadas à OpenGL para exibição no dispositivo de saída. Apesar da exibição ser realizada por estas geometrias simples, nada impede que sejam utilizadas geometrias mais realística da superfície representada, isto não é feito simplesmente porque não foram modeladas tais representações gráficas, o que demandaria um pouco mais de tempo, fugindo dos objetivos principais do projeto.

Há também o compilador para a linguagem LA extendida. Este compilador é responsável por compilar e transformar o código da LA para `C#`. Cenas, roteiros e ações podems ser descritas pelo usuário em LA, incluindo-se ai a criação dos atores, eventos, novos roteiros, novos objetos, entre outras coisas permitidas pela linguagem. Ao pedir para compilar o código, a aplicação fará a análise léxica e sintática do código fonte e, caso houver alguma proibição, dará um aviso de erro mostrando a linha onde tal erro ocorreu e qual o código correspondente àquele tipo de erro. Se não houver erros de compilação, o texto presente no editor do ambiente de programação será transformado em código `C#` para que possa ser executado pelo framework .NET. Este compilador juntamente com o restante da aplicação permite que a animação seja pausada a qualquer momento, um novo trecho de código seja inserido, compilado *just-in-time* e executado a partir da etapa corrente da animação como se fosse parte do código original.

No motor de física de corpos rígidos foram implementadas funcionalidades e algoritmos essenciais ao propósito desta aplicação, que resume-se à testar a simulação de corpos rígidos em um ambiente descritivo. Os principais algoritmos implementados, para que servem, e como funcionam foi descrito no Capítulo 2. A arquitetura do motor pode ser definida pelos seguintes componentes: compilador da linguagem de animação, motor de renderização(visualizador de arquivos de animação), motor de física (incluindo módulos responsáveis pela integração e resolução das equações de restrições), e aplicação, Figura 4.1. O compilador da nova linguagem de animação é derivado do compilador de LA (CLA), que por sua vez era derivado do compilador da linguagem L. Ele toma como entrada arquivos contendo especificações em LA de uma ou mais cenas a serem animadas (arquivos scn), e produz como saída arquivos na linguagem `C#`. O motor de renderização, conta com uma janela onde a cena é exibida. Para renderizar as imagens dos atores da cena na janela, atrelado à esta janela há um renderizador o qual toma a a cena corrente e renderiza um quadro da animação a cada passo de tempo. A implementação atual do renderizador é baseada em OpenGL, porem pode ser estendida para suportar outras APIs gráficas. Para a animação dinâmica, o responsável é o motor de física. A parte principal da arquitetura, responsável pelo gerenciamento dos atores e relações entre eles está construída em `C#`. A parte do motor de física responsável por rotinas matemáticas específicas, como por exemplo, integração da equação de movimento e resolução

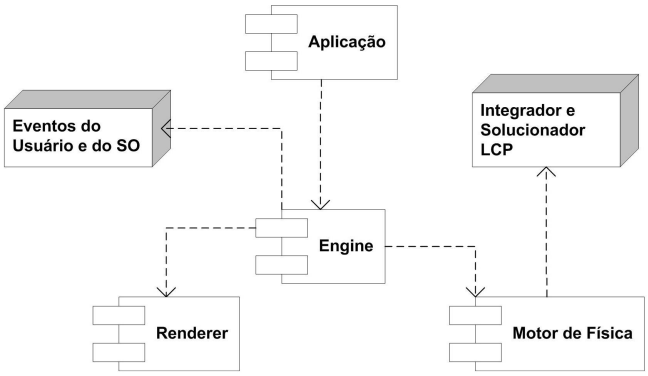


Figura 4.1: Componentes do motor 3D.

do PCL, estão implementadas em `C++`; estas rotinas são utilizadas em `C#`, transformando-se as rotinas `C++` em dlls e exportado-as ao código `C#`.

Todos estes componentes são gerenciados pelo módulo principal, chamado de Engine. Este módulo é o responsável por receber e tratar os eventos do usuário, calcular o passo de tempo, invocar o motor de renderização e invocar o motor de física, tudo sincronizadamente. No caso das ações, deve impedir concorrência entre threads (detalhadas mais adiante) ao processar e criar ações. Dentro deste módulo, estão presentes as estruturas de dados que tratam dos principais objetos que podem ser criados; um gerenciador de corpos rígidos (classe `eRigidBodySpace`); lista e todas ações e roteiros criados; lista de todos modelos; quais as cenas e câmera correntes, janela onde será realizada a renderização; entre outros. A aplicação, ao ser iniciada, cria automaticamente um objeto único do tipo `Engine`, que por sua vez, trata de criar os componentes básicos do motor 3D automaticamente (câmera, janela, gerenciador de corpos rígidos, etc).

4.3 Componentes e Funcionamento do Motor

4.3.1 Fluxo de Execução do Laço Principal

O laço do motor 3D é uma variação da opção mais popular de laços para motores, onde o laço principal é regular e utilizamos um temporizador em sua thread, Subseção 1.3.2. É bem verdade que o motor contém outras threads, para roteiros, mas estas interferem na etapa de atualização e não são controladoras do motor de renderização e de física como a thread do laço principal. A simulação inicia-se após a criação da cena, representada por algum objeto da classe `Scene`. Ao invocar `Scene::Start()` do objeto, o motor 3D verifica se existe, para esta cena, algum roteiro atribuído. Caso houver, o código contido no roteiro será parseado, compilado, e os objetos e ações descritos transformados em objetos do motor 3D e adicionados à cena. É possível também, caso o usuário deseje, não criar roteiros, mas adicionar itens à cena; neste caso, eles serão adicionados à cena imediatamente. A diferença é que, com um roteiro é possível controlar, esperar por alguma resposta de outro objeto, criar novos roteiros e ações em tempo de execução, pausar o roteiro por determinado tempo, entre outras vantagens.

Iniciada a cena, esta entra em estado *Running*. Este estado indica que a cena foi iniciada e está em funcionamento. A janela de exibição é criada no motor 3D e à ela são atribuídos os métodos responsáveis por simular e desenhar a cena. Isto é feito atribuindo à esta janela um callback que invoca as rotinas de renderização e atualização sempre que necessárias, e esta necessidade é controlada pelo motor 3D. A seguir, iniciamos o relógio, que terá o papel

de "contar" e sincronizar o tempo decorrido entre cada iteração do laço principal. Iniciamos daí o laço principal; este laço só será quebrado no momento que a aplicação receber uma mensagem (seja do SO, seja criada pelo usuário) que exija que o programa seja terminado. Do contrário, o laço principal executa os seguintes passos repetidamente:

- Executa as ações.
  - O motor 3D bloqueia a lista de ações para que possa atualizá-las. Isto é necessário porque o laço principal e os roteiros executam em threads diferentes; ao tentar atualizar a lista de ações ativas (em *Running*), alguma outra thread pode estar criando ou modificando as ações existentes, o que geraria inconsistência dos dados no motor. O motor 3D vai percorrendo a lista de todas as ações existentes, para formar uma nova lista, somente com as ações que não estão com seu ciclo de vida terminado; ainda nesta mesma etapa, o motor 3D aproveita para invocar o método `Action::Update()` de cada ação, para atualizá-las.

    Se uma ação estiver em estado *Terminated*, ou seja, seu ciclo de vida chegou ao fim, nada há a fazer, e esta ação não participará da nova lista de ações. Caso a ação esteja suspensa (estado *Suspended*), ela é enfileirada na nova lista, pois apesar de estar pausada, seu ciclo de vida ainda não chegou ao final. Caso a ação esteja ativa e executando (*Running*), cada ação invoca o método `Action::Update()`, e é adicionada à nova lista de ações. Ao final, temos uma nova lista de ações, com atualização de cada uma. Só então a desbloaqueamos.

- Executamos a simulação física, invocando o motor de física (Capítulo 3), resumidamente:
  - Primeiramente atualizamos o passo de tempo decorrido desde a última atualização, este tempo decorrido é chamado aqui de dt. Logo em seguida, bloqueamos o acesso à modificação de elementos da cena; como vamos alterá-la, portanto outras threads que não a do laço principal não podem alterá-la concorrentemente. Depois o motor conta com duas alternativas para o passo de tempo da simulação física: passo de tempo fixo, passo de tempo variável. No passo de tempo não fixo, o motor 3D não leva em consideração o tempo necessário na etapa de renderização e atualização de eventos e ações, a parte da dinâmica é então simulada utilizando um passo de tempo (aqui com valor default de 1/60) sempre igual. A segunda variação, o não fixo e opção default deste motor, o motor de física calcula a variação dt para a iteração atual, caso o tempo decorrido for inferior ao tempo mínimo determinado pelo passo de tempo desejado, não é realizada a invocação da atualização da dinâmica. Caso o tempo seja maior ou igual ao passo de tempo desejado, invoca-se o motor de física para o passo de tempo desejado, e o tempo excedente é somado ao próximo dt da iteração seguinte do laço principal.
  - O cálculo principal da dinâmica é computado através do método externo (implementado no motor de física) `Scene::PerformSimulation()`.
  - Após a simulação, a cena é desbloqueada para alterações por outra thread.

- A janela invoca o renderizador para exibição dos resultados (atores em suas respectivas poses) e estes são renderizados pelo motor de renderização.

- Aplicação recolhe novas mensagens e eventos que venham a ser criados neste espaço de tempo. Caso uma das mensagens seja para terminar a aplicação, o laço é quebrado. A aplicação então libera todos os recursos alocados e a simulação termina. Caso contrário, todos os passos são repetidos.

4.3.2 Cena e Seus Seqüenciadores

Uma cena é representada pela classe `Scene`. É constituída por uma lista de atores, uma lista de junções e uma de luzes; uma cena é um objeto renderizável, implementando a interface `IRenderable`. Portanto, percebe-se que, além de tratar-se de uma representação do aspecto físico do mundo virtual, também encapsula membros referentes à renderização. As listas de atores,luzes e junções são instânciadas a partir de classes paramétricas que representam uma lista de componentes, `ComponentList`, e são também um container de objetos, `IContainer`. A Figura 4.2 mostra a representação, em notação UML, da cena. Referente ao aspecto gráfico,

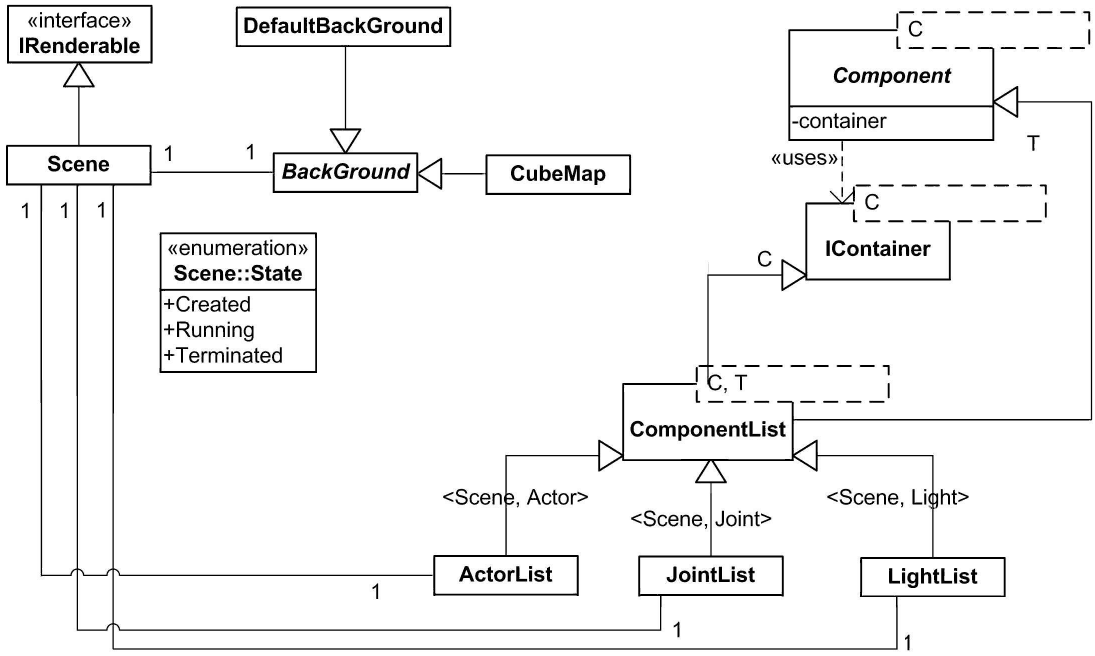


Figura 4.2: Representação estática da estrutura da classe Scene.

a cena conta ainda com uma iluminação ambiente default, e um atributo do tipo `BackGround`, que é a classe dos objetos que indicam o tipo do cenário ou cor de fundo.

Toda cena tem um tempo de vida, contado a partir de sua criação. Uma cena pode estar em 3 estados diferentes: `Scene::State::Created`, quando a cena foi criada, mas não entrou em execução ainda; `Scene::State::Running`, quando a cena foi criada e está sendo executada e por fim `Scene::State::Terminated`, quando seu ciclo de vida chega ao final.

Uma cena pode ter um roteiro a ser seguido, atributo do tipo `Script`, chamado `script`. Se não possuir roteiro algum (nulo), seus componentes são criados normalmente, na sequência como constar no código fonte. Caso contrário, toda a sequência de criação, espera, inicio de ações, criação de novos roteiros, entre outras ações possíveis de serem descritas em um roteiro, estarão descritas no texto do roteiro que será invocado ao inicio da cena.

Todo roteiro tem acesso à cena ao qual pertence, por este motivo, é necessário que a cena tenha algum dispositivo para controlar o acesso concorrente dos roteiros, pois eles podem criar e destruir seus componentes. Para tal, conta com o atributo `Lock lock`; `lock` é um objeto de sincronização; o usuário, ao descrever uma rotina passível de concorrência, deve invocar `Lock::Acquire()`, este comando faz com que o comando da cena seja adquirido. Se outra thread deseja efetuar alguma outra rotina que sofra com concorrência também deve invocar `Lock::Acquire()` para a mesma cena, no entanto, se alguma thread já tiver adquirido o comando da cena, o invocador de `Lock::Acquire()` vai esperar que a cena seja liberada. O comando do usuário que libera uma cena é `Scene::Lock::Release()`. Após

`Release()`, qualque thread que esteja esperando pela liberação da cena continua por causa de um `Lock::Acquire()`, toma controle sobre a mesma até que a libere também. É fácil perceber que, se o desenvolvedor não tomar os devidos cuidados pode provocar uma situação de *deadlock*, mais detalhes são dados na seção que trata de sequênciadores, Subseção 4.3.4.

Resumidamente, os principais atributos de uma cena, em `C#` são:

- `float TotalTime` - Tempo total, em milisegundos, de tempo de vida da cena. Pode ser utilizado caso o usuário deseje consultar o tempo de vida da cena.
- `Color AmbientLight` - Objeto da classe `Color`, que representa uma quádrupla RGBA, cor de fundo da cena. Tem como valor default `Color.Gray`, objeto do motor 3D que exibe a cor cinza.
- `string name` - Objeto cadeia de caracteres que armazena o nome da cena. Seu nome é criado quando na instanciação da mesma, passando uma string ao seu construtor.
- `ActorList actors` - Lista de todos atores pertencentes à cena. A cena é quem invoca o renderizador para exibição de cada um deles, portanto a necessidade da existência da lista de atores dentro dela.
- `JointList joints` - Lista de todas as junções entre atores. É importante para passagem das junções ao motor de física, e renderização da estrutura das junções pelo motor de renderização.
- `LightList lights` - Lista de todas as luzes que provêm iluminação à cena. A iluminação é ativada pelo motor de renderização.
- `BackGround scenario` - Cenário de fundo da cena. Pode ser um mapa de ambiente, que exibe uma imagem projetiva de texturas, mais conhecido como *environmentmap* [WS99], pode ser também um bitmap de fundo ou a opção default que é apenas uma cor de fundo RGB.
- `Script script` - Roteiro de entrada da cena. Caso o roteiro for não nulo, sua execução será iniciada, caso contrário este atributo é ignorado.
- `Lock locks` - Objeto sincronizador da cena. Se alguém invocou `Acquire()` de `lock` para ter controle sobre a cena, nenhuma outra thread pode obter controle sobre esta cena até que seja invocado `Release()`.
- State state - Estado corrente da cena. Pode ser iniciada (`Created`), rodando(`Running`) ou terminada(`Terminated`).

4.3.3 Atores, Junções e Luzes

Atores,luzes e junções são todos componentes de uma cena, que têm a classe base em comum `SceneComponent`. `SceneComponent`, por sua vez, deriva da classe paramétrica como `Component<Scene>`, ou seja, é um componente de uma cena que possui um container para tal.

Um ator é representado pela classe `Actor`. Também é um objeto renderizável, implementando a interface `IRenderable`. Possui uma pose global, a matriz `gloBalPose`, que armazena sua posição e orientação espacial em relação ao sistema global de coordenadas. Um modelo faz parte de um ator, e nele estão informações sobre o material — em relação à sua aparência, classe `Material` — que pode ser utilizado pelo renderizador. Os principais atributos de um material são:

- `public Color Ambient` - É a cor da radiosidade do ambiente que o material representa. No caso do material de alguma superfície é a intensidade de luz ambiente que este refletirá. Caso for material de uma fonte de luz, representa a intensidade da cor de luz ambiente que esta fonte emite.
- `public Color Diffuse` - É a cor da contribuição difusa da iluminação que o material representa. No caso do material de alguma superfície é a intensidade da contribuição difusa da luz que este refletirá. Caso for material de uma fonte de luz, representa a intensidade da contribuição difusa da cor da luz que esta fonte emite.
- `public Color Specular` - É a cor da contribuição especular da iluminação que o material representa. No caso do material de alguma superfície é a intensidade da contribuição especular da luz que este refletirá. Caso for material de uma fonte de luz, representa a intensidade da contribuição especular da cor da luz que esta fonte emite.
- `public float Shine` - `Shine` é o coeficiente indicador da intensidade do "*spot*" de uma superfície ao ser iluminada.
- `private Opacity MaterialOpacity` - `Opacity` é um enumerador que pode assumir dois valores: `OPAQUE` e `TRANSPARENT`. O primeiro indica que o material é opaco, e o segundo que é translúcido; o grau de tranparência é definido pelo fator A da quádrupla RGBA que representa a cor do material. A parcela de transparência, em porcentagem, é dada pela fórmula (1-A)*100%. O atributo `MaterialOpacity` armazena um destes valores. Nota: Para simular este efeito de transparência não foram utilizados shaders, apenas *blending* [WS99], uma técnica que calcula a contribuição de cor do material de um objeto e mescla com a contribuição de cor ao fundo desse objeto.
- `private bool BGReflection` - Valor booleano que se caso verdadeiro e o fundo da cena for constituído de uma imagem, aplica a textura da imagem na superfície do material com se o mesmo fosse um espelho. Caso o fundo da cena não possuir uma textura ou o valor de `BGReflection` for `false`, nada faz.

O modelo que o ator possui, indicado pelo atributo `model`, pode possuir um conjunto ou uma malha de triângulos. A classe `Model` representante de um modelo, tem métodos que permitem manipular ou carregar uma malha de triângulos como representação gráfica do ator. Todo ator possui um objeto `eRigidBody` chamado `body`, que é a abstração de corpo rígido; um corpo rígido conhece quem é o ator do qual ele faz parte. `eRigidBody` é composto de um `BodyMaterial`, possuidor das características físicas do material do corpo, como por exemplo o coeficiente de atrito. Possui ainda densidade, massa, tensor de inércia local, pose local e uma lista de formas que o compõe chamada `shapes`, Figura 4.3. As formas são objetos do tipo `eShape` representantes do formato físico de vários objetos que compõem o corpo rígido. Na ambiente uma forma pode ter o formato de uma esfera, de uma caixa, um cilindro ou mesmo um plano. `Shape` é um `Component<Scene>` e também implementa `IRenderable`, portanto além de ser um componente do motor 3D, sua forma é renderizável; como não possuímos modelados neste projeto, modelos gráficos sofisticados para exibição dos atores, nos limitamos a renderizar suas formas quando na renderização destes. No momento que o renderizador deve desenhar os atores, ele pergunta ao ator se ele possui um modelo atribuído, caso possua esta será a representação a ser renderizada, caso contrário são renderizadas as formas do ator apenas.

Entre as possíveis especializações de `Shape` temos as classes `SphereShape`, `BoxShape`, `CapsuleShape` e `PlaneShape` que representam as formas para esfera, caixa, capsula e plano respectivamente. A representação, em notação UML, das classes citadas é ilustrada na Figura 4.4. A representação de uma junção é a classe abstrata `eJoint`, que deriva de

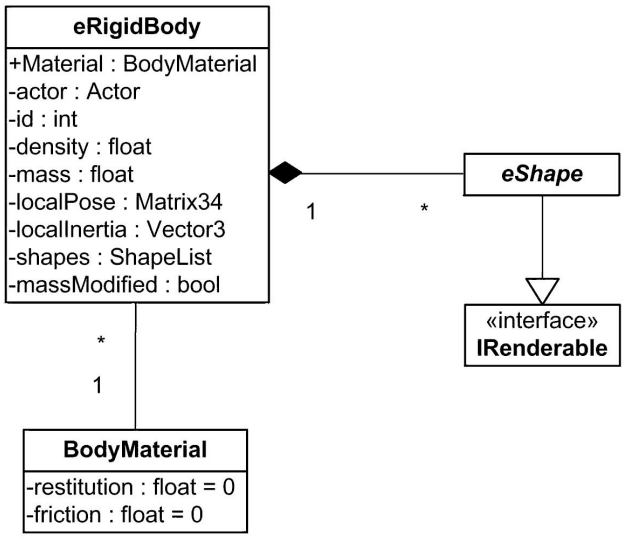


Figura 4.3: Representação estática da estrutura da classe eRigidBody.

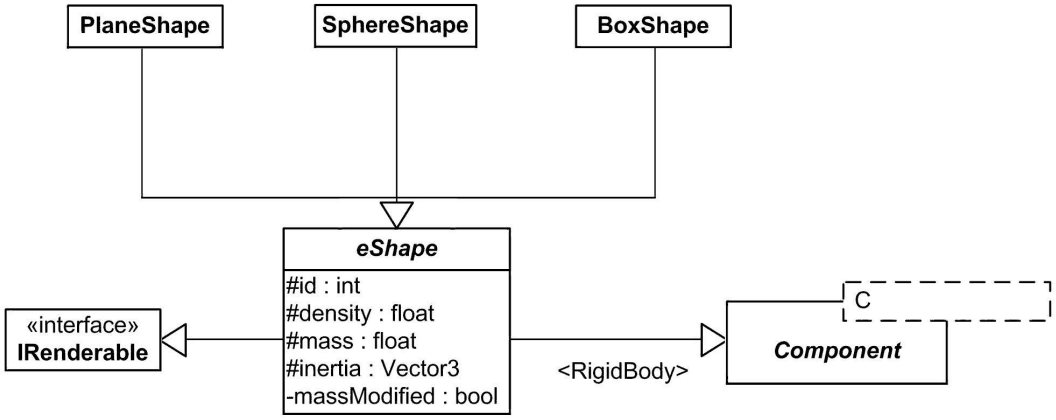


Figura 4.4: Representação estática da estrutura da classe eShape.

`SceneComponent`, Figura 4.5. Os objetos que podem ser criados são as especializações para os tipos de junção de revolução, junção fixa e junção esférica, respectivemente representadas pela classes `RevoluteJoint`, `FixedJoint` e `SphericalJoint`. A junção tem referência à dois objetos `eRigidBody`, que são os dois corpos de uma junção; `body0` aponta para o primeiro corpo, nunca podendo ser nulo, e `body1` aponta para o segundo corpo.

Uma junção, dependendo de sua especialização, pode ter uma âncora e um eixo. A âncora refere-se ao ponto exato onde dá-se a junção entre os dois corpos. O eixo representa, no caso da junção de revolução, o eixo sobre o qual é permitido que os corpos rotacionem. A classe `Joint` armazena uma representação local de âncora e de eixo para cada corpo rígido. Além dos membros citados, `Joint` também possui atributos chamados `ERP` e `CFM` que representam o parâmetro de redução de erros e o *constraint force mixing*, respectivamente. Conta também com uma lista de estruturas do tipo `ConstraintInfo`, as quais contém informações necessárias sobre a junção para que seja possível montar as matrizes e vetores do PCL para encontrar as forças de restrições de junção. Uma luz, no motor 3D, serve apenas como iluminação para o material que representa o aspecto visual dos atores. A classe `Light`, objeto que representa uma fonte de luz, também deriva de `SceneComponent`; é possível, representar uma luz pontual. Como toda luz pontual, ela possui uma posição, que é representada na classe

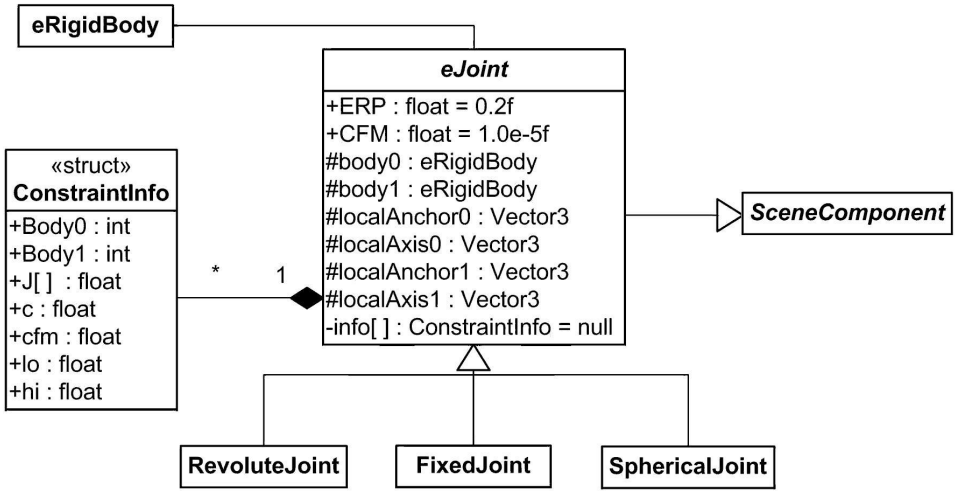


Figura 4.5: Representação estática das estruturas das classes de junção.

pelo atributo `Position`. Adicionalmente, a classe possui um atributo do tipo `Material`, que por sua vez tem triplas de valores para representar a iluminação difusa, especular e ambiente. Deste modo, atribuindo-se algum valor para o componente ambiente do material da luz, esta contribuição será tratada também como iluminação ambiente.

4.3.4 Seqüenciadores

Na API do sistema, há classes de objetos chamadas seqüenciadores. Estes sequenciadores são objetos sincronizáveis (podem ser executados cada um em uma threads própria), e em conjunto com o motor de física definem alterações no estado dos compoentes da cena, assim como também podem definir alterações em eventos e parâmetros do motor de renderização. Um sequenciador pode ser composto de qualquer conjunto de atividades que, quando executadas, podem modificar o estado de componentes da cena, inclusive em relação ao tempo de vida do sequenciador. Por exemplo, controlar o movimento e invocar quaisquer membros acessíveis de atores, luzes e câmeras; alterar atributos de um modelo tal como cores e densidade; criar e remover componentes da cena; aplicar forças e torques sobre os corpos rígidos; criar, iniciar ou esperar por outros sequenciadores; et cetera.

Sequênciadores são representados pela classe abstrata `Sequencer`. Um sequenciador pode ter cinco estados diferentes, indicados pelos valores do enumerador `Sequencer::State`:

- `Created` - Criado apenas, mas ainda sem ter sido iniciada sua execução;
- `Running` - Rodando, ou executando. O sequenciador já foi criado e no momento está sendo executado.
- `Suspended` - Suspenso. O sequenciador foi criado e iniciou sua execução, porém em determinado momento seu fluxo de execução foi pausado. Ao ser retomada a execução, o fluxo deverá continuar de onde havia parado.
- `Waiting` - Aguardando. O sequenciador foi criado e iniciou a execução, porém foi requisitado que antes de continuar o fluxo de execução, espere uma mensagem de algum objeto ou espere por determinado período; neste determinado momento o sequanciador está em espera, somente após a mensagem do objeto requisitado ou após o tempo determinado pela espera o sequanciador continuará seu fluxo de execução normalmente.

- `Terminated` - Terminado. O sequenciador foi criado e executado até atingir seu final, portanto após esta etapa ele pode enviar mensagens à outros sequanciadores que esperam por seu término e depois disso, ser excluído do sistema.

`Sequencer` possui dois métodos que devem ser obrigatóriamente sobrecarregados: `abstract void Start()` e `abstract void Exit()`. `Start()` é invocado sempre na criação de um sequenciador e `Exit()`, quando este atingir o estado `Terminated`. A representação estática da estrutura de sequenciadores em UML e exibida na Figura 4.6.

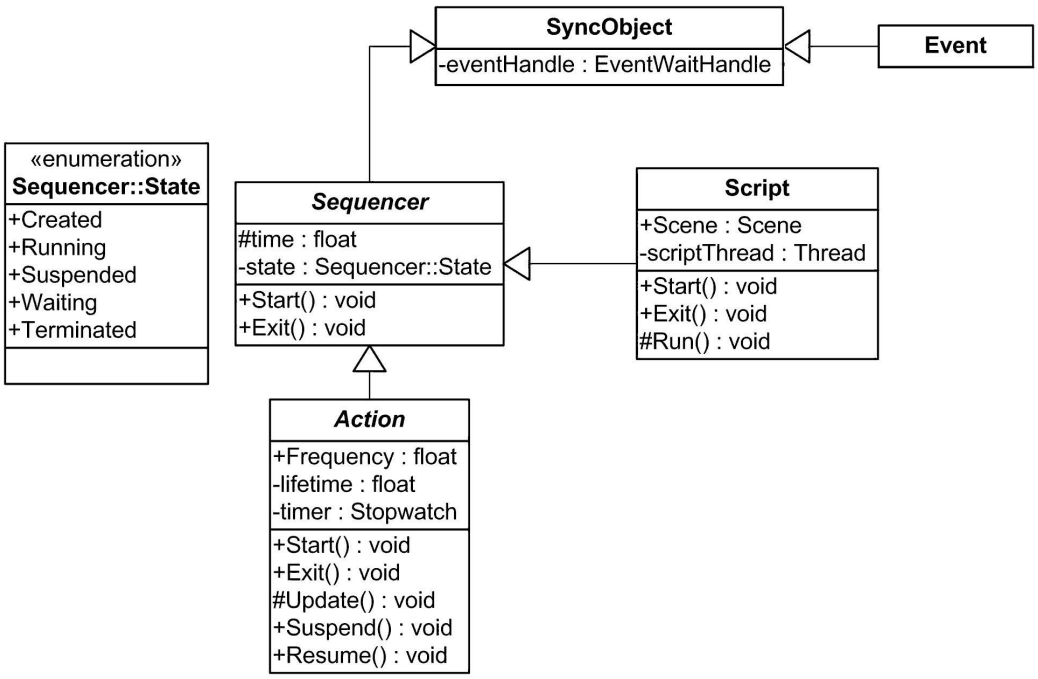


Figura 4.6: Representação estática da estrutura de Sequencer.

Uma ação é indicada por uma classe derivada de `Sequencer`, de nome `Action`. Possui um contador de tempo (atributo `timer`), um atributo chamado `lifetime` que vai acumulando seu tempo de vida conforme o tempo passa. Portanto, é possível dizer que certas ações podem ter seu tempo de vida determinado pelo programador. `Action` sobrecarrega os métodos `Sequencer::void Start()` e `Sequencer::void Exit()` que pedem ao motor que inicie ou termine o ciclo de vida da ação.

Além destes, há o método `Action::void Update()`, que é onde está alojado todo o código descritivo a ser executado pela ação. A cada passo de tempo, sua lista de ações ativas é percorrida e o método `Update()` de cada uma das ações é invocado; ou seja, cada método `Update()` de cada objeto do tipo `Action` que esteja ativo é executado inteiramente dentro de uma única iteração do laço principal. Por exemplo, suponha uma ação cujo método `Update()` realize mudanças nos atores de índice 1,2,4 e 5. Em toda iteração do laço principal, os atores 1,2,3,4 e 5 serão mudados, até que o ciclo de vida da ação termine (entre em estado `Terminated`). Lógivamente, se as ações tiverem uma carga excessiva de comandos dentro de `Update()`, estes poderão afetar a eficiência do motor 3D no sentido de diminuir a taxa de quadros por segundo, dependendo da velocidade do processador.

Para descrever os roteiros, devemos sobrecarregar a classe `Script` derivada de `Sequencer`, Figura 4.6. O usuário pode definir um roteiro por qualquer classe que derive de `Script`. O método `Script::void Run()`, que é invocado sempre no início da execução de um roteiro (não da criação, somente no momento que iniciar-se a execução da descrição do roteiro),

implementa as atividades que devem ser executadas por um roteiro específico. O estado de um roteiro que tem sua execução iniciada é definido como `Running`. Um roteiro termina quando CPPtextRun() retorna ou na invocação de `Exit()`; em ambos os casos, o estado do roteiro torna-se `Terminated`.

Um roteiro poderá conter, do mesmo modo que as ações, descrições textuais que indiquem a criação de novos objetos e também mudanças nos estados dos objetos já existentes. A partir do momento que o roteiro é iniciado (`Running`), seus comandos são executados até o final do mesmo, se possível, em apenas um passo de tempo. Ainda, um roteiro pode ser suspenso e esperar até que determinado evento ou ação ocorra ou até mesmo que outro roteiro seja executado até o final. Quando a condição de espera for satisfeita, o roteiro continua executando seus passos a partir do trecho onde havia parado. Ao esperar por determinado tempo ou por uma resposta de algum objeto, o roteiro entra em estado de `Waiting`.

É possível haver vários roteiros em uma única cena, assim como roteiros que disparem outros roteiros. Por causa disso, em certos momentos teremos alguns roteiros suspensos em `Waiting` e outros sendo executados do início ao fim. Por tratar-se de uma aplicação em tempo real, um roteiro que entra em `Waiting` não pode pausar a execução do programa, do mesmo modo que um roteiro longo não pode tomar uma fatia muito grande do tempo em cada quadro de animação, pois então teríamos um "atraso" na simulação.

Como um roteiros não tem um tempo definido de quantos ticks de tempo levará para ser executado, a priori não é possível escaloná-los satisfatóriamente ao desenrolar da simulação. A partir do momento que um roteiros é iniciado ele pode esperar por algum evento ou ação, ou ainda criar outros roteiros, o que fatalmente não permitiria que a aplicação fosse executada em tempo real. Por estes motivos, um roteiros terá além das funcionalidades já citadas, propriedades de uma *thread*. Cada roteiro será tratado como uma thread separada do laço principal. Desta maneira, se uma thread entrar em `Waiting`, o restante da simulação (demais roteiros e o laço principal) pode continuar sendo executado normalmente. Isso é utilizado também para poder limitar o tempo de processamento de um roteiro por ticks de tempo. Tratando o roteiros como uma thread permite que ele seja executado "concorrentemente" com o laço principal e com os outros roteiros ou ações, não sendo necessário escaloná-lo manualmente, o que de certa forma seria inviável já que não é possível determinar quanto tempo ou quantos ticks de tempo cada trecho do roteiros levará para ser executado.

A idéia principal é a seguinte: cada roteiros executa em sua própria thread. Além dos roteiros ainda há uma thread principal, que é o laço principal da simulação. A thread principal sempre é executada e jamais entra em hibernação (*sleep*), a não ser é claro quando na escalonação das tarefas das threads realizadas pelo SO. Já as threads responsáveis por cada roteiros podem executar concorrentemente suas ações, concorrendo inclusive com o laço principal. Caso um roteiro específico necessite entrar em estado `Waiting` a thread responsável por ele entra em sleep e somente será acordada pela mensagem esperada, que pode ser uma quantidade de tempo ou uma mensagem de algum objeto ou até mesmo outro roteiro. Vale ressaltar que, apesar da thread do laço principal não ser dependente das threads dos roteiros, o contrário pode ocorrer pois um roteiro pode entrar em `Waiting` até que algum evento ou ação específica do laço principal ocorra.

No exemplo da Figura 4.7, temos o laço principal divido em criação da cena com roteiro, atualização e renderização (conceitualmente poderíamos entender a entrada e atualização como uma única etapa) e quatro roteiros criados: Script0, Script1, Script2 e Script3. Onde Script0 é o roteiro de entrada da cena, ou seja, o que será iniciado (método `Script::Run()` invocado) ao invocarmos `Scene::Start()`.

Ao iniciar a cena `s`(Figura 4.7), Script0 aciona o início da execução do Script1 e também do Script2. O Script1 executa a *ação a* e *ação b*; concorrentemente o Script2 executa a *ação*

a2 e entra em estado de espera pelo término do Script1. Note que os próximos passos do Script2 são iniciar o Script3 e esperar pelo término do mesmo, o que pode levar algum tempo. Além disso, o Script2 está esperando pelo término do Script1, que por sua vez está esperando pelo término do Script3. Como cada roteiro é uma thread separada e o laço principal é uma outra thread independente, o laço principal continua sendo executado independentemente se algum roteiro está ou não em estado de espera.

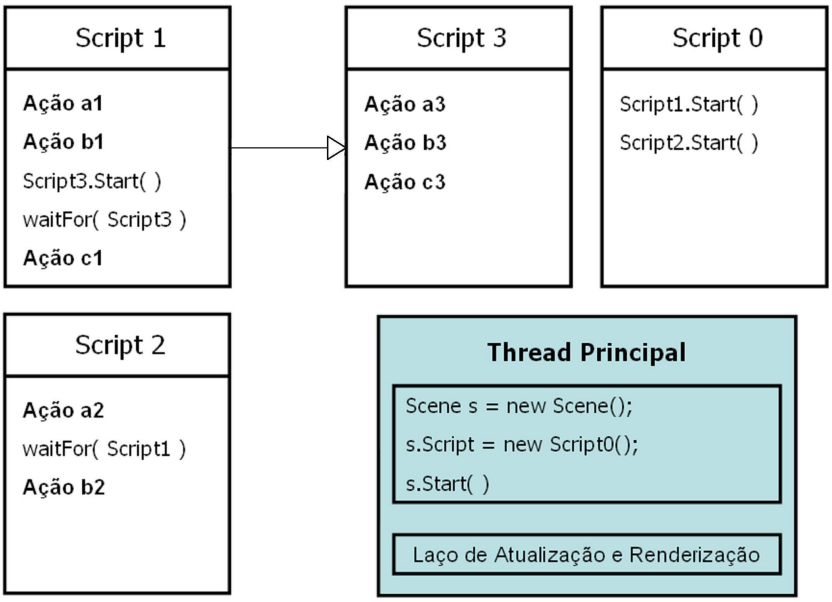


Figura 4.7: Threads da simulação de roteiros.

Quando o Script2 resolve esperar pelo término do Script1, a thread deste entra em uma estado de sleep até que seja acordada novamente pelo término do Script1. O Script1 por sua vez, ao resolver esperar pelo Script3, entra em estado de sleep também, até que o término do Script3 acorde-o. Enquanto isso o Script3 e o laço principal continuam sendo executados normalmente em suas respectivas threads. Quando no término de Script3, uma mensagem será enviada para a thread do Script1 acordando-o. Este continuará a executar seus comandos, neste caso a *ação c*. Após executar esta última ação, Script1 mandará uma mensagem à thread de Script2 acordando-o e este seguirá executando a *ação b2*. Não importa quanto tempo cada roteiro levará para chegar ao fim, concorrentemente o laço principal sempre está sendo executado, e o mesmo vale para os roteiros ativos. Desta forma evitamos atrasos na renderização e atualização.

Importante ressaltar que os trechos de modificações concorrentes lutam pela alteração da cena. Para sincronizar estas alterações, o desenvolvedor deve utilizar-se do bloqueador da cena, `Scene::Lock @lock`.

Seja `s` o objeto criado em LA que representa a cena. Quando algum roteiro for modificar estado de objetos da cena, o desenvolvedor deve sempre invocar o comando `Lock.Acquire()` da cena. Neste momento o poder de modificar a cena é passado à thread onde o comando foi invocado; se outro roteiro quiser também modificar `s`, ao invocar `s.Lock.Acquire()`, o bloqueador informará para a thread que a cena já possui alguma outra thread controlando-a. Ao receber esta informação, o roteiro entra em espera até que o controle da cena seja liberado. ao adquirir o controle da cena, o roteiro deve realizar então as operações desejadas e ao final, liberar a cena para que outras threads também possam utilizá-la; para isto o comando para a cena `s` seria `s.Lock.Release()`.

4.3.5 Eventos do Usuário

Neste contexto, um evento nada mais é do que um sinal enviado por algum objeto dizendo que algo esperado pela aplicação ocorreu no momento. Geralmente um evento é utilizado para ativar a execução de um ou mais métodos definidos pelo usuário ou pela aplicação. Há dois tipos de eventos que consideraremos, o primeiro é um evento de objeto na qual algum objeto criado pelo usuário notifica outro objeto ou a própria aplicação de que algo esperado ocorreu, como por exemplo o término do ciclo de vida do objeto.

O segundo tipo é um evento de dispositivo onde algum dispositivo de entrada, por exemplo o teclado ou mouse, é utilizado para enviar dados e interagir com o programa. Enquanto que no primeiro caso deve haver uma política definida pelo programador de como cada mensagem será capturada e tratada pelos objetos, no segundo caso a captura dos eventos é feita em primeira instância pelo sistema operacional. Certas linguagens de programação como por exemplo `C#` permitem que verifiquemos a ocorrência destes eventos e obtenhamos os dados recebidos pelo SO para que possamos dar-lhes o tratamento pretendido. Para tal geralmente a linguagem de programação permite declarar métodos ou funções e relacioná-los com cada evento, são as chamadas funções ou métodos de callback.

Na linguagem utilizada na implementação, `C#`, um *delegate* é um tipo por referência utilizado para referenciar um método [dCed]. Ao criar um tipo *delegate*, devemos definir para ele um tipo de retorno e os parâmetros que serão recebidos (os argumentos passados ao método). Qualquer método que satisfaça a assinatura do *delegate*, pode ser atribuído à uma instância do mesmo. A assinatura de um *delegate* não é tal como a assinatura de um método em `C++`; a assinatura é composta pelo tipo de retorno e parâmetros apenas, por isso podemos atribuir ao *delegate* métodos que satisfaçam apenas a condição de ter tipo de retorno e tipos de parâmetros iguais aos do *delegate*. *Delegates* funcionam analogamente a ponteiros em `C++` porém são orientados a objeto e tipos seguros. Mais ainda, *delegates* são encapsuladores de métodos; é possível atribuir diversos métodos diferentes ao mesmo *delegate*. Programaticamente podemos realizar mudanças nas chamadas dos métodos, inclusive incluindo um novo código chamado de dentro de algum tipo existente.

`C#` utiliza *delegates* para declarar eventos de um tipo. Para eventos do sistema operacional (eventos do teclado, mouse, tela, etc) são utilizados tipos especiais de *delegates* chamados *hooks* ou ganchos. Fora da classe que declarou o evento (*delegate*), um evento é como um campo do objeto, porém com acesso mais restrito. As únicas operações permitidas são compor um novo evento naquele *delegate* ou remover algum *delegate* adicionado anteriormente. Ao compor um evento naquele *delegate*, estamos conectando ou *"hookin up"* o evento. Em outras palavras, criando um gancho do nosso evento no *delegate* já existente.

Existe em `C#` um método chamado `setWindowsHookEx` responsável por ativar um procedimento gancho definido pela aplicação [Cora]. Este método, recebe um *delegate*, e um enumerador que define o tipo de evento do SO (mouse, teclado, tela, etc). O *delegate* recebido deve ter parâmetros compatíveis com o tipo de evento do SO indicado pelo enumerador. O método `setWindowsHookEx` ativa este *delegate* para receber notificações do SO do tipo de evento desejado, desta forma, sempre que o SO receber um evento daquele tipo, o *delegate* indicado será invocado automaticamente. Componentes de formulário em `C#` já têm diversos eventos conectados à eles; todos definidos por um método virtual. Para utilizá-los em um componente derivado, basta sobrecarregar o método equivalente e este será invocado dinâmicamente sempre que houver um evento daquele tipo.

Neste ambiente, o usuário deve ser capaz de definir ações para quando determinado evento do SO ocorrer. Por exemplo, ao pressionar uma tecla, caso esta tecla corresponder a uma tecla específica definida pelo usuário para gerar um novo ator de algum tipo, isto deve ocorrer sempre que aquela tecla for pressionada. O mesmo vale para eventos do mouse

definidos pelo usuário (manipulação da câmera por exemplo). Isto é obtido criando algum objeto derivado da classe `UserEventCallBack`. Esta classe possui métodos que recebem um nome de função como parâmetro e adicionam esta função na lista de callbacks a serem invocados ao ocorrer determinado evento. A classe `UserEventCallBack` possui um método `Add[TIPO_DE_EVENTO](EventArgs e)` para cada evento permitido que o usuário manipule, onde `[TIPO_DE_EVENTO]` pode ser um evento do tipo `OnKeyDown` (ao pressionar uma tecla), `OnMouseDown` (ao mover o mouse), `OnMouseClick` (ao clicar o mouse), `OnMouseDrag`(ao arrastar o mouse com algum botão pressionado). `UserEventCallBack` tem uma referência para o gerenciador de callbacks, `CallBackManager`, responsável por adicionar e retirar métodos de suas respectivas listas de callbacks. Além disso, na ocorrência de um evento, o componente capturador pede ao `CallBackManager` para invocar o container responsável por abrigar os métodos daquele tipo de evento. O container, por sua vez, trata de invocar todos os métodos nele contidos, passando o(s) argumento(s).

O parâmetro `EventArgs` é um argumento recebido pelo tipo de evento, pode ser um `KeyEvent` ou `MouseEvent`, que são objetos que contém informações sobre os dispositivos envolvidos no evento. É importante notar que, o método a ser passado como callback pode utilizar quaisquer membros de sua classe como um objeto comum, ou seja, continua sendo um método de algum objeto. No momento da ocorrência de determinado evento, o programa se encarrega de invocar todos os métodos referênciados na lista de callbacks (utilizamos *delegates* para este papel). Os valores possíveis que podem ser acessados pelo usuário através de `KeyEvent` são:

- `KeyCode` - Retorna código Unicode do teclado para a tecla a ser pressionada ou solta.
- `Alt` - Retorna um valor `true` caso a tecla ALT foi pressionada, senão retorna `false`.
- `Control` - Retorna um valor `true` caso a tecla CTRL foi pressionada, senão retorna `false`.
- `Modifiers` - Retorna uma flag com valores indicando os modificadores ativados. A flag indica qual combinação de CTRL, SHIFT, e ALT foi pressionada. A flag é uma combinação dos valores do enumerador chamado Keys, com possíveis valores CTRL, SHIFT, ou ALT. Por exemplo, se as teclas CTRL e SHIFT estiverem pressionadas, o valor da flag retornada será o valor de Keys.SHIFT & Keys.CTRL.
- `Shift` - Retorna `true` se a tecla SHIFT foi pressionada, caso contrário retorna `false`.

Para objetos do tipo `MouseEvent` pode-se acessar,

- `Button` - Botão do mouse que está sendo pressionado, caso houver. O retorno é na forma de algum valor do enumerador chamado MouseButton, com os possíveis valores:
  - MouseButton.Left - Botão esquerdo do mouse foi pressionado.
  - MouseButton.None - Nenhum botão do mouse foi pressionado.
  - MouseButton.Right - Botão esquerdo do mouse foi pressionado.
  - MouseButton.Middle - Botão do meio do mouse foi pressionado.
- `Location` - Retorna a localização do ponteiro do mouse. A localização é retornada em um objeto da classe chamada `Point`. `Point` possui apenas dois atributos: `int x`, `int y`, que indicam a posição do ponteiro em coordenadas de tela; possui somente um método, `boolean IsEmpty()`, que retorna verdadeiro se as coordenadas x e y do mouse estiverem na posição (0,0), caso contrário retorna o valor falso.

- `Clicks` - Retorna um inteiro indicando o número de vezes que o botão do mouse foi pressionado e solto.
- `Delta` - Retorna um inteiro indicando o número de *detents* da roda do mouse que rodaram. Um detent é um ponto da roda do mouse.

  `X` - Retorna a coordenada x do mouse, em relação a tela.

  `Y` - Retorna a coordenada y do mouse, em relação a tela.

Neste trabalho só é necessário interagir através do mouse e do teclado. Portanto,os únicos eventos do sistema operacional implementados que podem ser definidos pelo usuário são:

- Teclado:
  - Pressionar uma tecla. Para adicionar o método à lista de métodos que serão invocados ao pressionar uma tecla basta invocar o método `AddKeyDownCallBack(p)`, onde `p` é o nome do método que tratará este evento.
  - Soltar uma tecla. Para adicionar o método à lista de métodos que serão invocados ao soltar uma tecla antes pressionada de uma tecla basta invocar o método `AddKeyUpCallBack(p)`, onde `p` é o nome do método que tratará este evento.
  - Pressionar e soltar uma tecla. Para adicionar o método à lista de métodos que serão invocados ao pressionar e soltar uma tecla (o callback somente será invocado quando a tecla for solta) de uma tecla basta invocar `AddKeyPressCallBack(p)`, onde `p` é o nome do método que tratará este evento.
- Mouse:
  - Pressionar um botão do mouse. Para adicionar o método à lista de métodos que serão invocados ao pressionar um botão do mouse basta invocar o método `AddMouseClickCallBack(p)`, onde `p` é o nome do método que tratará este evento.
  - Arrastar o mouse com algum botão pressionado. Para adicionar o método à lista de métodos que serão invocados ao arrastar o mouse (com algum botão pressionado) uma tecla basta invocar o método `AddMouseDragCallBack(p)`, onde `p` é o nome do método que tratará este evento.
  - Pressionar e soltar algum botão do mouse. Para adicionar o método à lista de métodos que serão invocados ao pressionar e soltar de um botão do mouse basta invocar o método `AddMousePressCallBack(p)`, onde `p` é o nome do método que tratará este evento.

Apesar de tudo, não há necessidade do usuário ter conhecimento de todos estes conceitos sobre eventos e sua criação. Para facilitar a descrição das rotinas tratadoras de eventos do usuário, a linguagem de animação LA foi estendida, sendo inclusas as classes `UserEventCallBack` e `UserEventManager`, esta última não é acessível pelo desenvolvedor e faz parte apenas da implementação. Classes derivadas de `UserEventCallback` são "manipuladoras de eventos", porém o desenvolvedor só as vê como classes que têm métodos que podem processar teclas e ações do mouse. Basta criar uma classe qualquer, que derive de `UserEventCallBack`, criar um método com mesma assinatura do callback desejado e invocar o correspondente método `Add[TIPO_DE_EVENTO]` herdado da classe base, por exemplo,

```
class KeyListener: UserEventCallback
{
     public void onKeyPress(KeyEventArgs e){...}
     public override void Start()
     {
          AddKeyPressCallBack(onKeyPress);
     }
}

class MouseListener: UserEventCallback
{
     public void onMouseClick(MouseEventArgs e){...}
     public override void Start()
     {
          AddMouseClickCallBack(onMouseClick);
     }
}
```

Para a classe `KeyListener`, o atributo `KeyEventArgs` automaticamente terá sempre informações sobre a tecla pressionada (a aplicação cuidará de mudar este valor automaticamente sempre que houver algum evento de teclado). O usuário deve implementar seus próprios métodos responsáveis por tratar uma tecla ao pressioná-la, ao soltá-la, ou ao pressionar e soltar cujos corpos dos métodos devem conter código LA sobre como agir ao pressionar, soltar ou pressionar e soltar uma tecla respectivamente, e então adicioná-los à lista de eventos com seus respectivos métodos `Add[TIPO_DE_EVENTO]`.

Por outro lado, se desejarmos uma classe como `MouseListener`, responsável por fazer algo quando o mouse for clicado, Temos que declarar um método que recebe um argumento `MouseEventArgs`, que contém valores para cada botão e para o ponteiro do dispositivo. Analogamente ao manipulador de teclado, basta que o usuário implemente os métodos que representem as ações ao pressionar um botão, arrastar o mouse com algum botão pressionado ou clicar algum dos botões respectivamente.

Internamente, estas classes e métodos são convertidos para `C#` e atribuídos à um *delegate* que é o gancho responsável por cuidar daquela espécie de evento do SO. É importante notar que o usuário pode criar vários objetos destes que gerenciam os eventos (teclado e mouse). Como estamos trabalhando em cima de um encapsulador de métodos, podemos adicionar ou retirar estes gerenciadores de eventos a qualquer hora. Mas para isso, basta invocar o método `void Rem[TIPO_DE_EVENTO]`, já presente na classe base e passar como argumento o nome do método que foi adicionado ao container através de `Add[TIPO_DE_EVENTO]`.

4.3.6 Engine e Renderizador

`Engine` é a classe principal do motor 3D, e está sempre relacionada a uma cena. No motor `Engine` é um *singleton*, portanto há uma única instância desta classe em todo o motor 3D. Ela é responsável por gerenciar tanto a parte de modelos, luzes, câmera, e ações, e é esta classe que controla e inicia o laço principal. Possui diversos atributos de classe, dentre eles:

- `private static Scene currentScene` Atributo que armazena a cena corrente sendo processada.
- `private static Camera camera = new Camera()` - Contém uma referência para a câmera responsável pela visão do usuário , como só há uma saída para a exibição da

cena só há uma câmera, porém sempre há uma câmera. Por isso, na criação do atributo uma câmera default é providenciada pela classe `Camera`.

- `private static float timeStep = 1/60.0f` - Armazena qual o passo de tempo utilizado no motor (para realização de todas as rotinas). Tem como valor default 60 quadros por segundo.
- `private static Stopwatch timer = new Stopwatch()` Objeto relógio para controle de sincronização das invocações das rotinas de renderização e atualização, é instânciado na criação de `timer`.
- `private static MainWindow mainWindow` Como todo motor 3D, deve haver uma janela para exibição da cena; esta é representada por `MainWindow mainWindow`. O tipo `MainWindow` é também um formulário, pois é classe derivada de `System.Windows.form`, e declara um atributo do tipo `RenderWindow`, abstração de uma janela de renderização. Essa janela de renderização é quem possui o renderizador, representado pelo atributo `Renderer renderer` da mesma, como mostra o diagrama da Figura 4.8.
- `private static eEngine engine` - Instância única da classe `eEngine` para todo o motor 3D.

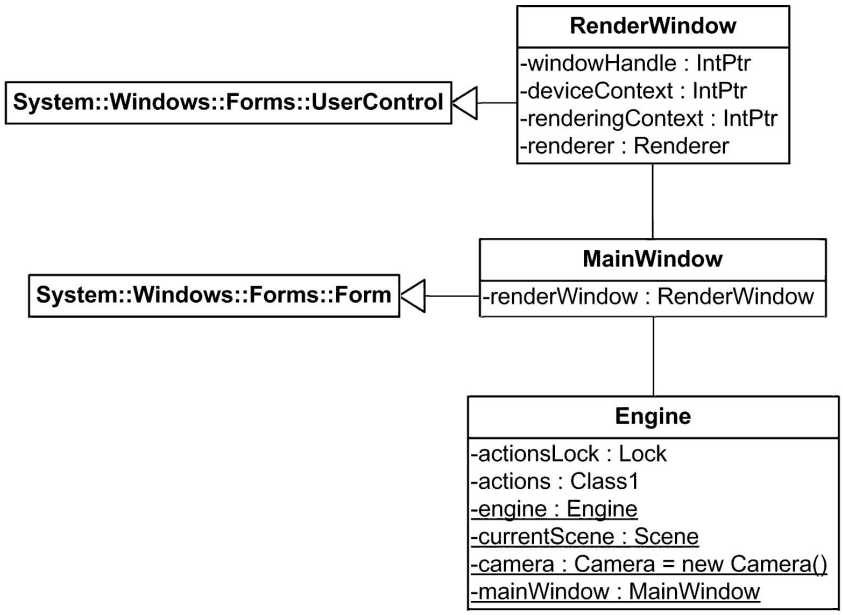


Figura 4.8: Classe eEngine.

Adicionalmente, há os seguintes atributos de instância declarados em `Engine`:

- `private ModelList models` - É uma lista de todos os modelos alocados no motor 3D.
- `private Queue<Action> actions = new Queue<Action>()` - A classe `Queue<T>` é uma classe paramétrica para representação de filas de um tipo `T`, neste caso de objetos `Action`. A fila de ações é atualizada a cada iteração do laço principal contendo todas as ações que estão ativas ou em estado de espera.
- `private Lock actionsLock` - Objeto sincronizador que bloqueia acesso a alguns objetos. Neste caso, é possível bloquear a lista e criação de ações. O usuário não precisa

e nem é capaz de fazer bloqueio de ações; isto é realizado apenas pelo motor 3D, mais especificamente `engine`, no momento que ele processa e atualiza as ações existentes. Isto evita inconsistências, como por exemplo um roteiro (que executa em uma outra thread) criar uma nova ação enquanto o motor 3D já estava processando a lista de ações.

A janela de exibição `mainWindow` é criada pela aplicação, porém como dissemos, seu atributo renderWindow (Figura 4.8) é quem será o objeto onde ocorrerá a renderização. `RenderWindow`, derivada de `UserControl`, contém um índice para manipulação da janela, um contexto de renderização (necessário para que seja possível ordenar à uma API gráfica que renderize uma imagem em determinado local) e um objeto `Renderer`, representado pelo atributo `renderer`.

`Renderer`, Figura 4.9, é a classe a qual contém todos os métodos necessários para exibir polígonos, linhas, pontos e mudar parâmetros de renderização tais como estados de renderização. Dentro do `Renderer` encontra-se a câmera e o volume de vista, responsáveis por definir e delimitar a janela de vista, posição e direção do olhar do observador.

O tamanho da janela de exibição é definido pelas constantes `DFL_IMAGE_H` e `DFL_IMAGE_W`, consequentemente também definida a razão de aspecto (`DFL_IMAGE_WDFL_IMAGE_H`). Há ainda um atributo do tipo `Material` chamado de `material`, que serve para setar determinado material antes de iniciar a renderização; Ao saber do material do ator corrente a ser desenhado, seu material é atribuído à `material`, e no momento de renderizá-lo, o material ativado é sempre o que está referenciado por `material`. Há 3 tipos de enumeradores utilizados para

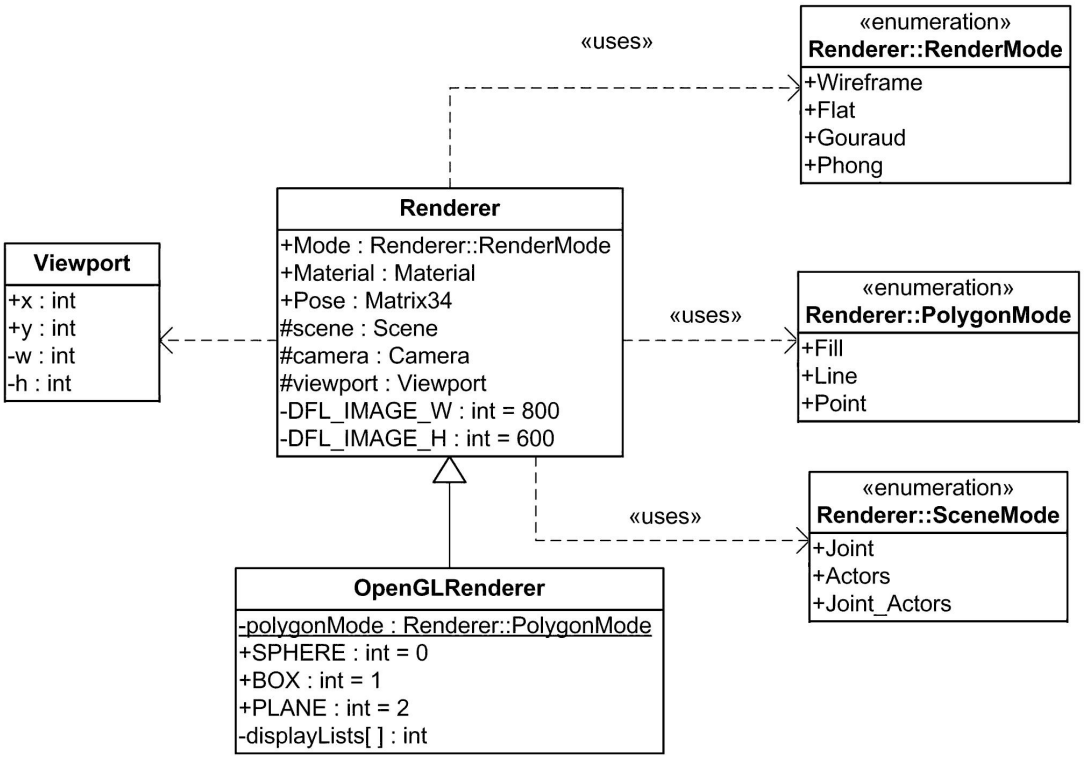


Figura 4.9: Classes Renderer e OpenGLRenderer.

definir estado de renderização e modos de exibição da cena: `RenderMode`,`PolygonMode` e `SceneMode`. O primeiro lista os possíveis modos de tonalização da imagem, pode ser tonalização de Gouraud, Phong, Flat ou WireFrame. `PolygonMode` por sua vez, contém os modos possíveis de exibição de polígonos: `Fill` para polígonos totalmente preenchidos, `Line` para desenhar apenas as arestas dos polígonos e `Point` para exibir apenas os vértices. Por fim,

`SceneMode` define se na cena serão exibidos somente os modelos dos atores, apenas as conexões e eixos das junções, ou ambos.

Os principais métodos declarados em `Renderer` são,

- `public abstract void RenderLights(LightList lights)` - Ativa as fontes de luz da cena. As fontes são recebidas pelo renderizador como uma lista de objetos `Light`.
- `public abstract void RenderSphere(Matrix34 pose, float r)` - Desenha uma esfera. Onde, o primeiro argumento é a pose da esfera (rotação e translação) em relação ao sistema global de coordenadas, `r` é o raio da esfera a ser desenahda.
- `public abstract void RenderJoint( )` - Recebe uma junção como parâmetro e renderiza o ponto âncora como uma esfera, e linhas ligando o centro de cada ator até esta esfera. É mais utilizado com o propósito de ajudar no processo de depuração e desing da cena.
- `public abstract void RenderBox(Matrix34 pose, Vector3 s)` - Desenha uma caixa. Onde, o primeiro argumento é a pose da caixa em relação ao sistema global de coordenadas, `s` é um vetor de 3 posições onde cada posição corresponde a comprimento,largura e altura da caixa.
- `public abstract void RenderCapsule(Matrix34 p, float r, float h)` - Renderiza uma cápsula. Onde, o primeiro argumento é a pose da capsula em relação ao sistema global de coordenadas, `r` é o raio do cilindro que faz parte do corpo da capsula, `h` é a altura da capsula.
- `public abstract void RenderPlane(Matrix34 pose, Plane plane)` - Desenha uma plano. Onde, o primeiro argumento é a pose do plano em relação ao sistema global de coordenadas, `plane` é o objeto a ser desenhado. Ele é passado como parâmetro porque, não é possível nem desejável renderizar planos sempre infinitos, então o objeto com medidas pré-definidas é passado.
- `protected abstract void StartRender()` - Deve conter inicializações necessárias da API antes de iniciar-se a renderização.
- `protected abstract void EndRender()` - Deve conter liberação de recursos e finalizações da API ao término da renderização.
- `public static bool DoRenderActors()` - Retorna verdadeiro se o modo de renderização da cena permitir a renderização de atores (atributo `sceneMode`). Retorna falso caso contrário.
- `public static bool DoRenderJoints()` - Retorna verdadeiro se o modo de renderização da cena permitir a renderização das junções (atributo `sceneMode`). Retorna falso caso contrário.
- `public static void ChangeSceneRenderMode()` - Alterna circularmente entre os modos de renderização da cena: somente atores, somente junções ou junções e atores.

A especialização utilizada para renderizar os modelos neste projeto utiliza OpenGL, e é implementada pela classe `OpenGLRenderer`, Figura 4.9. Nesta classe estão definidas as *display lists* das formas primitivas e sobrecarregados os métodos de `Renderer` para que utilizando OpenGL como API gráfica sejam capazes de exibir a imagem desejada.

4.4 Comentários Finais

Neste capítulo, foi exibida a arquitetura, principais componentes e classes responsáveis pelo funcionamento do motor. Mostramos como dá-se a componentização entre a aplicação, motor de física e motor de renderizaçao. Foi apresentada a classe `Scene` que representa uma cena, sua principal função e a lista de componentes que integram esta cena. Dentre estes componentes, derivados da classe paramétrica (`SceneComponent`), estão atores e luzes. Para atores, foi exibida como relacionamos um objeto do tipo `Actor` - composto de formas representados por objetos `eShape` - com um corpo rígido, cuja abstração é feita pela classe `eRigidBody`. Adicionalmente, são mostradas as classes que tem o papel de representar os corpos rígidos e suas propriedades mais importantes pertinentes ao usuário.

Definimos os conceitos e comandos para manipular sequenciadores e eventos. Descrevemos o funcionamento e como criar ações e roteiros. Apresentamos também a sequência de execução presente no laço principal, incluindo ai a renderização dos atores, invocação do motor de física e tratamento da lista de ações criadas pelo usuário. O tratamento implementado consistiu em um sistema de travamento da lista quando no processamento da mesma pelo laço principal; isto foi proposto porque existem threads independentes da principal que podem alterar a lista de ações concorrentemente.

Descrevemos ainda, como os roteiros, que executam em threads próprias, são sincronizados para evitar que ocorra inconsistência nos dados. A decisão de manter uma thread para cada roteiro deu-se pelo fato de que roteiros podem esperar por respostas de outros objetos ou mesmo por determinada fatia de tempo; como estas tarefas não podem parar ou atrasar a exibição e simulação, optamos por escaloná-las na forma de threads, já que a priori não é possível prever exatamente o tempo que cada roteiro exige.

Mostramos que atores possuem formas e também um modelo gráfico atribuído. Apesar disso, devido ao tempo exigido para criar modelos gráficos para os atores, optamos por não fazê-lo neste trabalho, pois além do tempo não contribui para alcançar os principais objetivos do desenvolvimento do projeto. Em vez disso, o motor permite que sejam exibidas apenas as formas dos atores. No que diz respeito ao material das superfícies, apresentamos os objetos que indicam as características físicas e visuais dos atores e formas. Para exibição dos atores, mostramos onde localiza-se a janela de renderização dentro do motor de renderização; vimos que esta janela possui um objeto renderizador (`Renderer`) que tem a capacidade de utilizar uma API gráfica para renderização dos modelos, neste caso a API OpenGL, definida pela especialização de `Renderer` chamada `OpenGLRenderer`. Apesar de conter apenas uma API gráfica e poucos algoritmos que manipulam a cena, no que diz respeito ao aspecto gráfico, o motor e o motor de renderização foram moldados de forma a permitir, sem grandes dificuldades, sua extensão para inclusão de novas APIs e algoritmos.

CAPÍTULO 5

Exemplos

5.1 Introdução

Neste capítulo apresentamos exemplos que demonstram algumas das funcionalidades do motor 3D. Em primeiro lugar, são apresentadas descrições de cenas mais simples, apenas para definir aspectos importantes da utilização da linguagem LA, como criação de atores, luzes, corpos rígidos e aplicação de forças sobre eles, todos utilizando-se de roteiros para descrição da cena. Em um segundo momento, apresentamos casos um pouco mais complexos. Estes demos, por sua vez, têm seu funcionamento descrito, porém seus respectivos códigos fonte em LA não estão inclusos neste capítulo por tratar-se apenas de um cápítulo demonstrativo. Neles, descrevemos como criar uma ação e executá-la, também exemplificamos a criação de um evento de usuário que controla a câmera do motor de renderização através do mouse e do teclado.

5.2 Carro contra Bloco de Esferas

Neste exemplo, temos um conjunto de atores que representam um "automóvel". Antes de iniciar esta sequência de criação de atores e aplicação de forças, devemos bloquear a cena para que outras threads não a alterem enquanto a estivermos utilizando. Ao final do roteiro, a cena é liberada. Este é composto de uma carroceria formada por um ator com diversas formas; na realidade é um ator composto por 3 caixas. Além da carroceria, o carro possui eixos sobre os quais rotacionam suas rodas, que possuem formas representadas por esferas. As rodas têm uma junção do tipo junção de revolução, onde a âncora situa-se no centro da esfera que representa a roda e o eixo da junção é paralelo ao eixo ao qual a roda pertence. Desta maneira, é permitido que a roda seja rotacionada apenas em torno de seu eixo (por causa das restrições das junção de revolução). Neste exemplo, ao criarmos a cena, aplicamos uma força linear sobre a carroceria do automóvel. Em seguida, criamos uma pilha de 360 esferas de material definido pelo desenvolvedor pelo nome de `Glass`. A gravidade aqui é de -0.9 na direção vertical, eixo Y das coordenadas mundiais. A Figura 5.1 mostra 4 quadros sequênciais desta simulação. A imagem de fundo da cena é um background do tipo `BackGround::Type::ENVIRONMENT_MAP`(Capítulo 4). O material do plano é reflexivo (para o fundo da cena) e também levemente transparente. O fundo da cena é definido sempre quando na criação da cena, sendo passado como argumento. Além do automóvel e da pilha

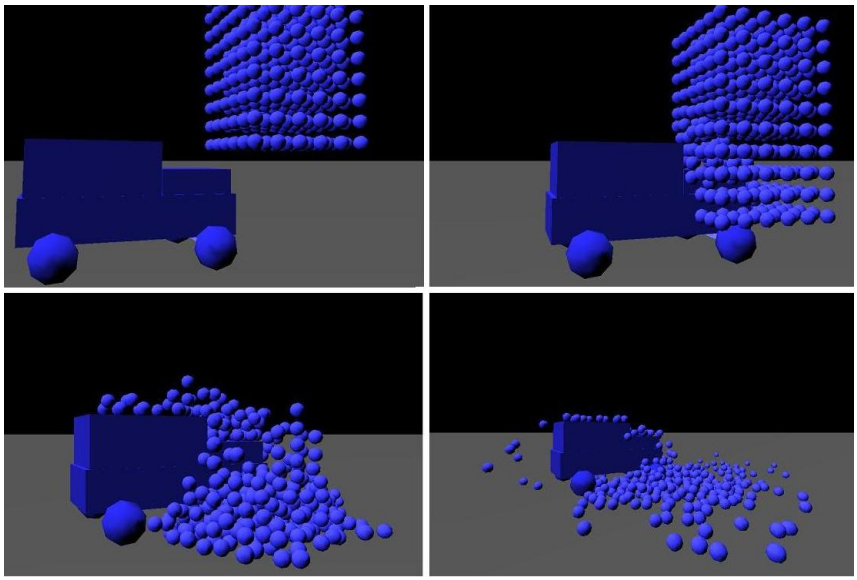


Figura 5.1: Carro contra bloco de esferas.

de esferas, temos ainda o plano sobre o qual os atores caem; é um ator estático, ou seja, não movimenta-se independente das forças aplicadas. Para criação deste, seguimos a mesma sequência de passos de um ator não estático: criamos o ator, criamos uma forma (neste caso plano) e a adicionamos ao ator.

5.3 Esfera Arremessada

Seguindo o mesmo princípio de bloquear a cena para alterações, nesta, criamos um palco, formado por um bloco, e dizemos que este é um ator estático. Em seguida, adicionamos o palco à cena. Criamos então, dois bonecos "Pep" (Pep é o nome que foi dado ao tipo do boneco); um boneco é composto de uma caixa que compõem seu tronco, 2 outras caixas que representam os braços e ligam-se ao corpo por uma junção esférica. Cada um dos membros inferiores são formados por duas caixas ligadas por uma junção de revolução para simular um "joelho" e conectados ao tronco por uma junção esférica. A cabeça é um ator formado por uma esfera com uma outra esfera menor representando o nariz (que faz um contra-peso para a cabeça), ligada ao tronco por uma junção de revolução.

Ao iniciarmos a cena, é criada uma esfera e aplicado nela uma velocidade linear, diagonalmente em relação ao sistema global de coordenadas, (0, 8.5, -8.5). Em seguida, posicionamos a câmera e criamos um bloco de caixas atrás do palco e dos bonecos. Ao inicio efetivo da cena, a esfera será lançada contra o peito de um dos bonecos Pep, que devido ao impacto colidirá com o segundo Pep, fazendo com que ambos sejam arremessados contra o bloco de caixas, conforme mostra a Figura 5.2.

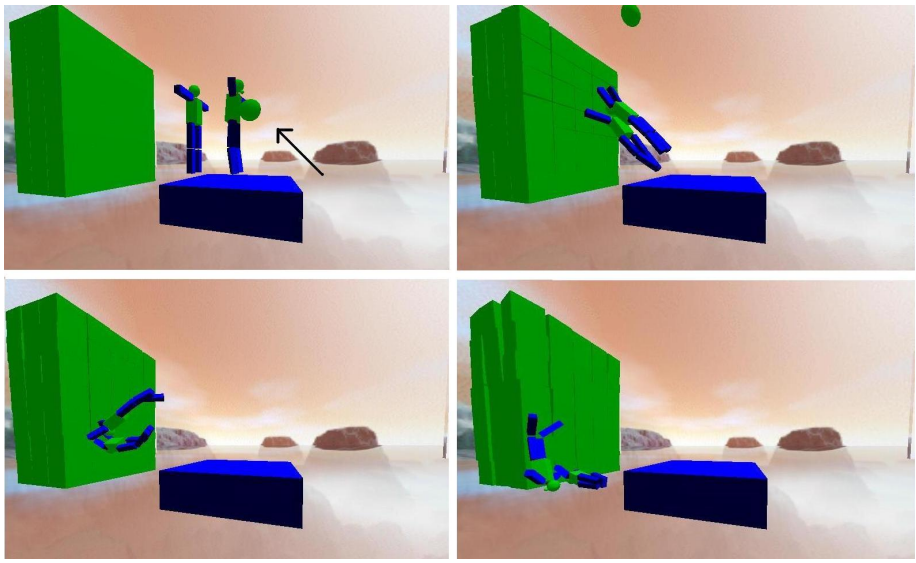


Figura 5.2: Esfera lançada contra boneco Pep.

5.4 Atropelamento de Moto

Nesta, simulamos um boneco Pep sendo atropelado por outro, que pilota uma "moto". Iniciamos esta demonstração invocando um método que cria a moto nos retorna um vetor com os seguintes componentes da cena, na ordem: roda dianteira da moto, roda traseira da moto, corpo do piloto, eixo da direção da moto. A moto é basicamente composta de esferas, caixas com uma junção de revolução para o guidon e junções esféricas para as rodas. Além disso, um boneco Pep é criado logo acima da moto, e quando começar a cair devido a ação da gravidade, ficará "sentado" sobre a moto, Figura 5.3. Criamos então o boneco Pep que será

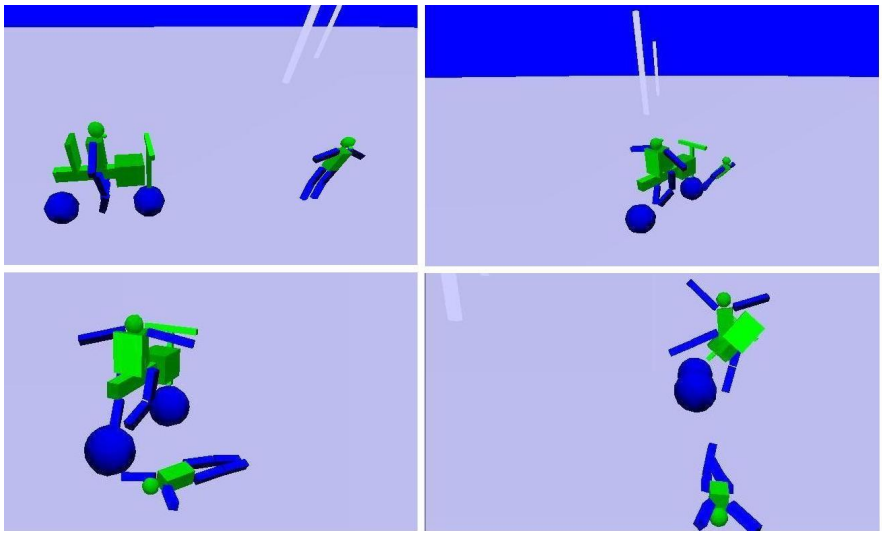


Figura 5.3: Moto com câmera automática.

atropelado. Setamos apenas a posição da câmera, e a gravidade da cena como -0.9 na direção Y (no caso, para baixo). Criamos uma ação, que é uma ação de câmera. Este tipo de ação recebe em seu construtor uma referência a algum ator, e a cada atualização, aponta a câmera na direção ao centro de massa deste ator, portanto ele será seguido pela câmera enquanto esta ação existir.

Ainda, criamos uma outra ação, porém do tipo que adiciona velocidade angular à um ator. Este objeto recebe um vetor de atores, o número de atores a serem modificados e uma velocidade angular que será aplicada à eles. A cada atualização, a ação percorre o número de atores passados no vetor argumento recebido, e aplica a velocidade angular. Repare que nesta cena, o fundo é default, onde há apenas uma cor RGB atribuída.

5.5 Fonte de Corpos Rígidos e Martelo

Começamos a cena com uma espera de 5000 milisegundos antes de criarmos os atores, ou seja, o roteiro sobre o qual a cena está executando pausa sua thread por 5000 ms. Criamos então o martelo. O método responsável por tal façanha nos retorna uma referência ao braço do martelo (onde podemos aplicar uma força). Uma ação do tipo que adiciona torque é instânciada, e à ela passada o braço do motor recém criado. Junto com o braço do motor, passamos um vetor que indica a força e direção que deve ser aplicada ao ator, Figura 5.4. Neste caso, o martelo fará movimentos circulares em volta do eixo Z das coordenadas globais. Feito isso, entramos em um laço que só será finalizado quando tivermos 20 milisegundos de

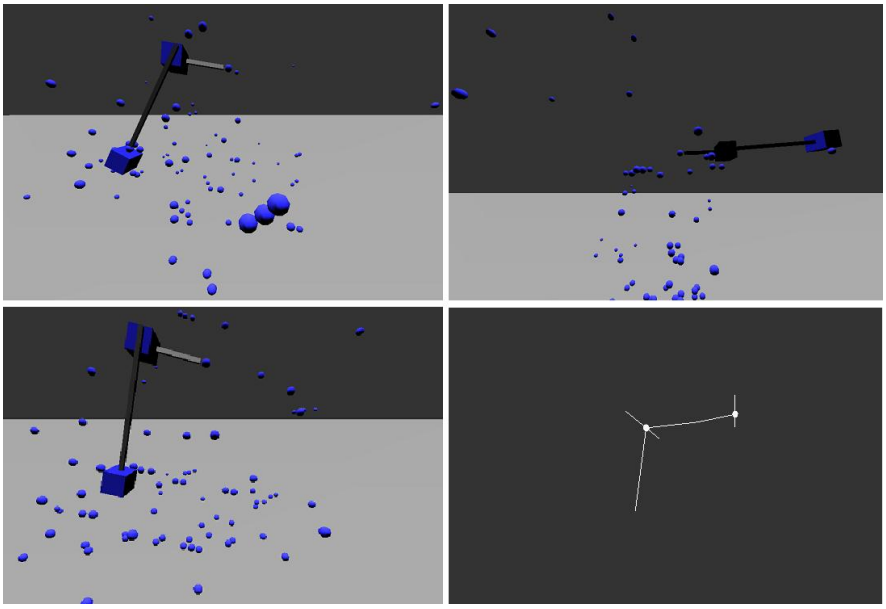


Figura 5.4: Martelo sobre fonte de corpos rígidos.

chamadas às ações que criam as esferas. O laço faz basicamente o seguinte: Se o número de ações criadas e acumuladas for menor do que o número máximo definido (no exemplo, 30), é criada uma ação que instância um determinado número de esferas, na origem da fonte de corpos rígidos, seu método `Update()` aplica uma força aleatória na direção Y positiva do sistema de coordenadas globais e também lateralmente e aleatóriamente em X ou Z. Estas ações são armazenadas em um vetor de ações do roteiro. Depois disso, a ação é pausada (pois já foi aplicada a força, não a aplicamos a todo quadro de animação). Se o número de ações criadas já ultrapassar o limite máximo estabelecido, percorremos o vetor de ações já criadas, voltando os atores a origem da fonte e despausando a ação, pausando-a logo em seguida. Estes passos vão, continuamente, reaproveitando as esferas que a muito tempo já foram acionadas.

5.6 Ponte Articulada

Esta cena simula uma ponte articulada presa a dois blocos fixos (estáticos), distantes um do outro. Sobre esta ponte estão dois bonecos Pep e um automóvel passa sobre aponte colidindo com os Peps. Após a criação dos blocos, invocamos o método responsável pela criação da

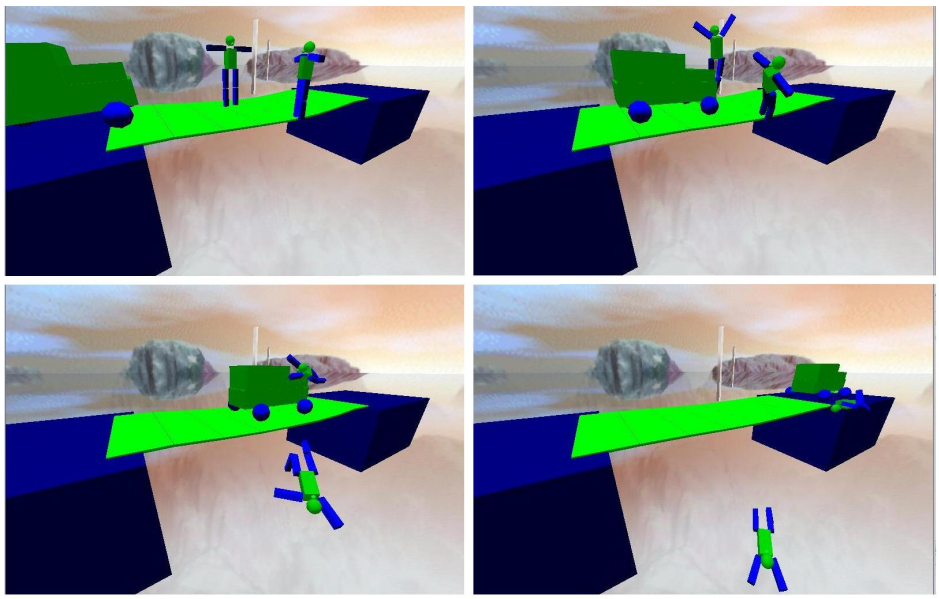


Figura 5.5: Carro sobre ponte.

ponte que, na realidade, cria várias caixas de altura bem pequena (como se fossem tábuas da ponte) e as liga por junções de revolução, de modo que formem uma articulação entre as tábuas da ponte. Criamos o automável, como no Exemplo 1, e imediatamente aplicamos uma força no centro de massa do ator que representa o "corpo" do carro. esta força tem como sentido do carro para a ponte, isto fará com que o carro movimente-se, atravessando a ponte. Criamos dois Peps mudando apenas os parâmetros, ou seja, poses e pesos (é possível também alterar a escala de todos os atores). Por fim, setamos a câmera, aceleração da gravidade e iniciamos a cena, Figura 5.5.

5.7 Caminhão Dirigível

Esta cena tem uma implementação na qual não entraremos em detalhes sobre o código, pois é extenso e não há necessidade de descrevê-lo neste capítulo. Vamos adiantar que, na criação do objeto roteiro de entrada desta cena, criamos uma rampa estática (caixa inclinada e fixa). Declaramos um vetor que será a velocidade angular aplicada nas rodas do caminhão (segue uma idéia parecida com a do primeiro exemplo, onde aplicávamos uma rotações nas rodas). A seguir, criamos uma ação de câmera. Este objeto(ação) é derivado de `UserEventCallBack` (Capítulo 4), portanto, uma classe que define eventos do usuário. Neste caso, a classe foi descrita de modo a controlar a câmera da cena através do pressionamento de teclas do teclado e do eventos do mouse (clique e movimentação do ponteiro). É possível mover e aplicar transformações na câmera e modos de renderização da cena e dos atores apenas pressionando as teclas definidas. Logo em seguida temos a instânciação de outro *callback* que cuida de permitir que o caminhão seja movimentado. Este por sua vez, tem o papel de permitir que forças sejam aplicadas nas rodas do caminhão ao pressionarmos as devidas teclas, permitindo que o caminhão ande para a frente, de marcha-a-ré e consiga girar sua direção (movendo as rodas, é claro). O caminhão é formado com uma base ou cavalo, uma carroceria, onde há uma junção de revolução entre elas, e óbviamente suas diversas rodas com junções esféricas. Ao fim do roteiro, esperamos um determinado tempo, fizemos isso porque criamos o caminhão a uma altura considerável acima do plano, então damos tempo para ele cair; porém isso não é obrigatório.

Na cena da Figura 5.7 é mostrado no primeiro quadro a forma fio de arame dos atores juntamente com os eixos e ligações das junções presentes neste modelo. No quadro seguinte,

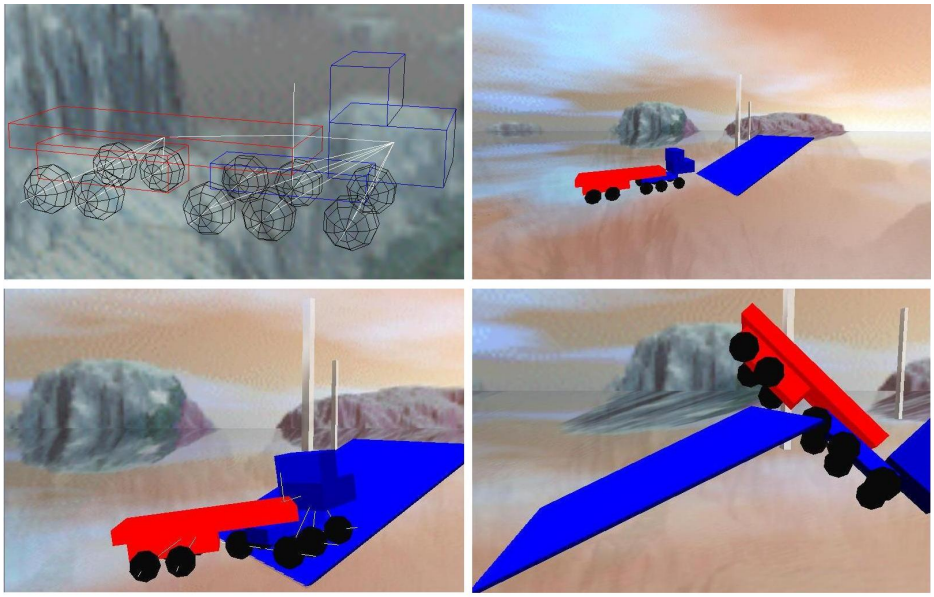


Figura 5.6: Caminhão.

avançamos com o caminhão, acelerando-o através do pressionamento de uma tecla; no terceiro quadro, exibimos os eixos das junções apenas para mostrar que as rodas dianteiras podem ser giradas. O último quadro apenas mostra o estado do caminhão quando ultrapassa a rampa.

5.8 Esferas Coloridas

Neste roteiro, centenas de esferas são jogadas dentro de uma caixa, estática, de material definido como transparente pelo desenvolvedor. Existe nesta cena um evento de teclado do usuário na qual é possível direcionar uma esfera de cor branca dentro da caixa; há uma tecla para cada sentido do sistema globla de coordenadas: X,-X, Y,-Y, Z e -Z. Ao pressionar a tecla correspondente, uma força é aplicada no cento de massa da esfera branca, produzindo assim uma aceleração linear na direção desejada de movimento. O coeficiente de atrito utilizado foi

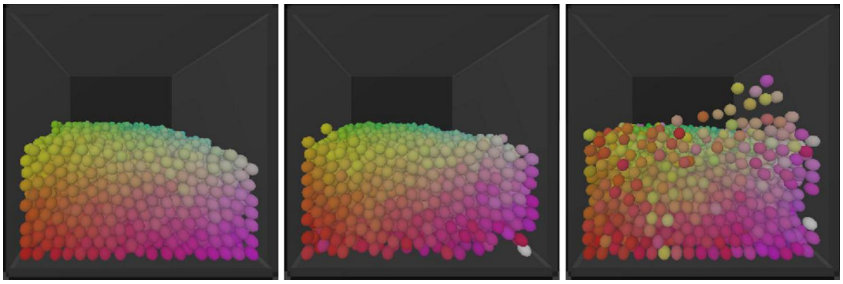


Figura 5.7: Imagem da simulação com 4000 esferas.

de 0,5 e a gravidade de (0,-0.9,0). A Tabela 5.1 a seguir mostra o desempenho para 50, 200, 500, 1000, 2000, 3000, 4000, 8000, 12000 e 16000 esferas dentro da caixa. A coluna *Restrições*

		Tempo CPU(ms)			
Atores	Restrições	Det. Colisão.	PCL	EDO	Total
50	156	0,21	0,36	0,01	0,58
200	948	0,66	0,95	0,04	1,65
500	3819	2,02	5,95	0,11	8,08
1000	8151	4,55	11,93	0,24	16,72
2000	15276	9,86	25,12	0,47	35,45
3000	23805	15,68	40,23	0,75	56,66
4000	32370	19,50	52,63	0,96	73,09
8000	66507	44,08	114,79	1,96	160,83
12000	103023	68,55	178,45	3,09	250,09
16000	139092	92,11	236,33	4,38	332,82

Tabela 5.1: Tempos de execução da aplicação das esferas coloridas.

indica a quantidade de restrições de contato para o número de esferas indicadas pela coluna *Atores*. *Det. Colisão* é o tempo que foi gasto com a detecção de colisão — lembrando que não foram feitas otimizações na detecção de colisão e nem levado em conta o tipo de cena para escolha da técnica de detecção de colisão utilizada —, *PCL* o tempo gasto solucionando-se o PCL utilizando o algorítmo SOR LCP; *EDO* é o tempo total para solucionar as equações diferenciais ordinárias (*ODEs*).

O importante é notar, na coluna *Total* que, pela grande quantidade de atores interagindo simultâneamente na cena, o motor 3D, e consequentemtente o motor de física, são capazes de alcançar um desempenho satisfatório em relação a outros motores e aplicações de tempo crítico. Mesmo com o motor de física e o detector de colisões mesmo sem grandes otimizações,

5.9 Comentários Finais

Neste capítulo foram apresentados exemplos de como podemos criar cenas utilizando LA. Ainda, mostramos como é possível que o usuário defina suas próprias rotinas de eventos. A partir dos exemplos mostrados, é possível ao desenvolvedor ter uma idéia das possibilidades de criação de cenas a partir dos exemplos descritos. É óbvio que há mais cenas e comandos de LA, apesar de não termos apresentarmos o código fonte completo. Por fim, apresentamos uma tabela de desempenho de algumas cenas em termos de quadros por segundo, tempo para cálculo de física e número de restrições simultâneas. O objetivo não foi mostrar que o motor 3D desenvolvido tem desempenho superior aos outros motores disponíveis, mas sim, mostrar que seu desempenho é equivalente aos mesmos, sem grandes perdas mesmo que o código fonte não tenha muitas otimizações.

CAPÍTULO 6

Conclusão

6.1 Discussão dos Resultados Obtidos

Este projeto foi desenvolvido com o intuito de servir como uma ferramenta de visualização de simulações dinâmicas de corpos rígidos, aplicado ao desenvolvimento de aplicações interativas e em tempo real, incluindo aí o desenvolvimento de jogos digitais. O trabalho foi parte do projeto de pesquisa que deu continuidade à expansão das capacidades do ambiente de um sistema de animação denominado AS, destinado à simulação dinâmica de cenas com corpos rígidos [Oli06], desenvolvido pelo Grupo de Visualização, Simulação e Games (GVSG) do DCT/UFMS. Neste novo sistema ou motor 3D, uma linguagem de programação híbrida chamada LA estendida é usada para descrever os objetos de uma cena a ser animada, bem como os roteiros, ações e eventos do usuário que modificam o estado desta cena ao longo do tempo. Esta linguagem utilizada é uma derivação da linguagem LA [Oli06], que por sua vez foi estendida a partir de uma linguagem de propósito geral chamada L.

O sistema utiliza animação dinâmica como principal técnica de animação, a qual pode ser escrita através de uma combinação da linguagem LA e do controle do fluxo de execução definido pelos seqüenciadores, todos estes controlados pelo motor 3D, em seu laço principal. O sistema total é constituído de componentes responsáveis por compilar e executar uma simulação exibindo os resultados em uma janela de visualização.

Para *softwares* em tempo real, a arquitetura da MVA não era adequada, ou seja, não atendia aos requisitos de performance exigidos neste novo projeto. Portanto, optamos por utilizar um novo compilador da linguagem de animação. Este compilador, gera código em MSLI, a linguagem de montagem independente de plataforma do framework `.NET`. Este código objeto é convertido em código nativo pelo framework antes de ser executado pela primeira vez. Encontram-se disponíveis compiladores que geram código em MSLI, como por exemplo o *Microsoft Visual Studio* ®. Entretanto, não é possível compilar LA diretamente para MSLI. A solução encontrada foi, partindo do código fonte LA, gerar código fonte `C#` o qual pode ser compilado imediatamente pelo Visual Studio. Esta decisão foi também baseada na robustez e semelhança da linguagem `C#` com LA. A maioria dos comandos de LA podem ser convertidos quase que diretamente para código `C#`. Além do que, `C#` substitui com eficiência as característica principais da MVA do AS original, como por exemplo, coleta de lixo, tipos seguros, tratamento de exceções, et cetera.

Além da mudança do compilador do projeto, o restante da MVA também foi eliminado. A

MVA efetuava somente as etapas de atualização e renderização de cada frame, sem levar em consideração o tempo de computação. Porém, aplicações interativas em tempo real devem exibir informações dentro de um passo de tempo determinado, para permitir que a interação seja contínua. Daí surgiu a necessidade de reestruturar a organização do laço principal para que substituisse a tarefa da MVA de, continuamente, alimentar o sistema com novos dados e exibir a cena. Ainda, foi preciso incluir uma nova funcionalidade, que é a de permitir que o usuário descreva eventos do SO.

O objetivo geral deste trabalho foi desenvolver um motor 3D para simulações dinâmicas de cenas tridimensionais constituídas de corpos rígidos. Entre os objetivos específicos destaca-se a implementação de um motor de física de corpos rígidos próprio para o projeto. O motor de física desenvolvido teve em seu desenvolvimento a preocupação de modelar e implementar apenas os componentes e algorítmos necessários ao propósito da simulação, além de manter uma abordagem orientada a objetos o mais organizada e didática possível. Isto foi importante porque além de prover conhecimento adicional ao (GVSG) do DCT/UFMS sobre simulação dinâmica, constitui uma biblioteca estensível para a realização de novos trabalhos, como por exemplo simulação de corpos flexíveis. Diferentemente de outros motores de física, há preocupação em manter o código fonte livre, e de acordo com os estudos realizados pelo GVSG, tornando sua utilização ou alteração mais fácil do que seria com um motor composto por dezenas de bibliotecas as quais nem sempre contam com documentação satisfatória. Um outro importante objetivo específico é a utilização da linguagem LA como linguagem descritiva. Esta linguagem, originalmente desenvolvida pelo (GVSG) do DCT/UFMS, foi estendida e serve tanto para definição dos componentes da cena, como para descrição de roteiros e eventos do usuário. É possível agora que o próprio desenvolvedor, seja de jogos digitais ou simulações, crie seus próprios objetos, ações, roteiros ou mesmo eventos utilizando esta linguagem.

Podemos resumir os objetivos específicos almejados como:

- Uma API eficiente para o motor 3D que seja capaz de acoplar motores de física, tanto de física de corpos rígidos ou motores já existentes, como também motores de física de corpos deformáveis; de fato, o motor 3D interage com o motor de física através da API do mesmo, sem grandes mudanças. Inclusive, o motor de física, descrito em `C++`, poderia ser utilizado em qualquer outro motor 3D sem grandes adaptações.

- Adaptação da linguagem de animação LA, presente no trabalho de [Oli06], para ser utilizada como linguagem descritiva do motor 3D. O trabalho realizado anteriormente foi estendido e aproveitado como liguagem descritiva não apenas da cena, roteiros e ações, mas também de eventos do usuário.

- Tornar AS uma aplicação interativa em tempo real, reestruturando sua arquitetura e laço principal, permitindo assim a execução dos roteiros e eventos do usuário em tempo real.

- Desenvolvimento de um motor de física para corpos rígidos, utilizado no motor 3D, com código aberto; todas as funcionalidades essenciais ao funcionamento de um motor de física de corpos rígidos foi desenvolvido e documentado nesta dissertação.

As principais etapas do desenvolvimento do projeto podem ser resumidas em:

- Estudo detalhado de como funciona um motor 3D para simulações. Foram consultadas diversas fontes da literatura e internet, não só sobre motores 3D em geral, mas também sobre bibliotecas auxiliares e motores de física. Primeiramente, foi necessário obter conhecimento de como é possível organizar, escalonar e sincronizar as etapas de atualização, simulação, renderização e interrupções (como por exemplo, eventos do SO)

de modo que a aplicação ainda mantivesse características de uma aplicação de tempo crítico.

Definidas as idéias, fez-se um estudo sobre quais os principais componentes de um motor 3D. Este estudo foi realizado baseando-se na literatura e em diversos motores de código aberto disponíveis na internet, como [ead, eaa, eab, Geb, Lab, Sul, McG, Val05]. A partir disso foi possível definir uma arquitetura, que embora não muito específica, separa os componentes de um motor 3D modularmente.

Pelo fato de existirem diversos módulos em um motor 3D, optamos por estudar os que realmente fossem úteis ao propósito de simulação. Dividimos conceitualmente o motor 3D em duas partes principais: rotinas relacionadas à simulação da cena (física), e resposta à eventos do usuário, pertencentes à mesma tarefa que é atualizar o mundo virtual. A segunda parte consistiria no motor de renderização, que trata de exibir o mundo virtual ao usuário.

Pertencente a área relacionada ao motor de renderização, foram pesquisados diversos algorítmos de organização de mundos virtuais, tal como técnicas de *culling* e *clipping*, amplamente utilizados em jogos digitais modernos. Porém, o desenvolvimento de tais algorítmos não foi considerada uma contribuição efetiva ao projeto, pois como dito anteriormente, já existem diversos algorítmos implementados disponíveis nos motores de código aberto, cada um adequado a um tipo específico de jogo ou simulação.

Relativo as tarefas de atualização, decidimos por dar continuidade a parte do trabalho de [Oli06], utilizando a linguagem LA como linguagem descritiva. Como uma aplicação interativa, fez-se necessária a inclusão de eventos do usuário no projeto. Inicialmente, o objetivo era utilizar bibliotecas de motores de física disponíveis gratuitamente. Contudo, os motores disponíveis ou não permitiam acesso irrestrito ao código fonte ou exigiam um estudo detalhado de todos seus componentes se quiséssemos estendê-los. Daí, a decisão de, com o conhecimento adquirido através destes estudos, desenvolver um motor de física próprio, mais modular, facilitando seu entendimento e adequando-o ao propósito da simulação.

- Iniciamos a implementação de diversas bibliotecas e funcionalidades de um motor de física, como por exemplo algorítmos PCL e integradores de equações de movimento. Em alguns casos, como no algorítmo utilizado para solucionar o PCL, foi realizada a implementação de mais de um tipo de algorítmo. Isto ocorreu porque não foi possível prever, a priori, o impacto de alguns fatores apenas com os estudos da dinâmica — como por exemplo o nível de precisão numérica na solução de um sistema linear ou ainda do desempenho não tão eficiente como nos outros motores de física — levaram a busca de alternativas que contornassem os problemas que surgiam ao desenvolver da implementação. Algumas vezes, estas dificuldades surgiram como entrave ao objetivo do projeto; por exemplo, desejávamos fornecer um algorítmo que calculasse a solução do PCL mas permitisse que estudos futuros paralelizassem o problema, mas em determinada etapa da implementação, percebemos que o algorítmo inicialmente proposto não permitiria tal façanha por depender muito de comunicação e precisão numérica; então foi exigido o estudo e implementação de outro tipo de algorítmo. Determinadas as bibliotecas, foi desenvolvida a estrutura do motor de física, a qual está descrita no Capítulo 2.

- Definida a parte dinâmica, partiu-se para a implementação do motor 3D e do motor de renderização. O motor 3D, descrito em detalhes no Capítulo 4, foi engenhado para encapsular tanto a parte gráfica como a parte dinâmica e interativa. Quanto ao motor de renderização, definimos os objetos que seriam responsáveis pelos componentes da

cena: atores, luzes, formas, corpos rígidos, entre outros. Integrado ao motor 3D, está a janela de exibição e captura de eventos do sistema operacional. Após estas etapas, foi possível, mesmo que não de maneira definitiva, visualizar como encontrava-se o desempenho e funcionalidades do motor de física. Isto foi importante antes da etapa final por permitir que erros fossem detectados mais facilmente.

- Com o motor 3D e de renderização em funcionamento, iniciamos o desenvolvimento das classes que definiriam seqüenciadores: roteiros e ações. Daí a importância da primeira etapa do projeto, onde analizamos os modelos de laços em tempo real, definimos que roteiros não poderiam executar sequêncialmente com as etapas de renderização e atualização do laço principal. Definidas as classes responsáveis pelos roteiros, foi montada uma política de sincronização, acessível ao desenvolvedor objetivando evitar problemas de concorrência entre roteiros e laço principal. Com ações e roteiros podendo ser descritos pelo desenvolvedor, o último passo foi mapear as implementações de eventos descritos pelo próprio desenvolvedor para a linguagem no qual foi implementado o motor 3D (`C#`); com isto é permitido descrever o tratamento de eventos do mouse e teclado, e o motor 3D automaticamente cuida de invocá-los quando o SO indicar que um evento aconteceu.

- Foi finalizado o ambiente de desenvolvimento, baseado no ambiente de AS. O motor e a linguagem LA foram integrados ao ambiente, composto por um compilador de LA, interface gráfica e bibliotecas de ligação. Por fim, vários exemplos foram desenvolvidos para teste do motor 3D.

Em virtude dos resultados obtidos, podemos enfim afirmar que todos os objetivos inicialmente propostos no trabalho foram plenamente alcançados.

6.2 Trabalhos Futuros

A grande interdisciplinaridade de um motor 3D, nos remete à possibilidade de diversos trabalhos em outras áreas da computação. Podemos sugerir como possíveis trabalhos futuros:

- Estudo e desenvolvimento de novas técnicas de subdivisão espacial e *culling* para o motor de renderização, como por exemplo árvores de subdivisão espacial, grafos de cena, entre outras estruturas de dados. O motor 3D implementado não conta com algorítmos sofisticados de subdivisão espacial. Caso tivessemos utilizando *shaders* ou mesmo uma cena muito extensa, isso poderia afetar a performance consideravelmente. Com o *framework* desenvolvido, é possível propor inclusive novas técnicas para gerenciamento da cena.

- Desenvolvimento de módulos ou submotores de rede e som. Um submotor de rede ou mesmo de som, além de completar o motor 3D como um *framewok* para jogos, abriria novas possibilidades de estudo na área de redes, por exemplo.

- Desenvolver e integrar o motor 3D à um motor de física de corpos flexíveis.

- Diversas otimizações podem ser realizadas no código fonte, entre elas a paralelização de algumas tarefas. isto pode ser realizado tanto em GPU como também aproveitando-se de processadores que ofereçam paralelismo.

- Estender a linguagem LA para que acomode descrições sintáticas de regras comportamentais para atores, o que poderia ser uma entrada para que o desenvolvedor defina, utilizando alguma técnica de IA, o comportamento inteligente de atores.

- Atualização do ambiente de desenvolvimento para que permita a inclusão de atores não só textualmente, mas também através de alguma interface gráfica.

Referências Bibliográficas

[AP97] M. Anitescu e F. Potra. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics*, 14:231–247, 1997.

[Bar89] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics*, 23(3):223–232, 1989.

[Bar92] David Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. Tese de Doutoramento, Cornell University, USA, 1992.

[Bar94a] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *In Proceedings of SIGGRAPH 1994*, 28:23–34, 1994.

[Bar94b] David Baraff. Linear-time dynamics using lagrange multipliers. *SIGGRAPH 94*, páginas 137–146, 1994.

[Bar01] David Baraff. Physical based modeling: Rigid body simulation. *SIGGRAPH 2001 Course Notes. Pixar Animation Studios*, 2001.

[Cli99] Michael Bradley Cline. *Rigid Body Simulation with Contact and Constraints*. Dissertação de Mestrado, B.S., The University of Texas at Austin, 1999.

[Cora] Microsoft Corporation. About hooks. Disponível em http://msdn.microsoft.com/library/en-us/winui/WinUI/WindowsUserInterface/Windowing/Hooks/AboutHooks.asp, último acesso em 27/11/2007.

[Corb] Microsoft Corporation. Directx develop center. Disponível em http://msdn.microsoft.com/directx/, último acesso em 27/04/2007.

[Cor08a] AGEIA Corporation. The ageia physx sdk. Disponível em http://developer.download.nvidia.com/PhysX/2.8.1/PhysX_2.8.1_SDK_Core.msi, último acesso em 15/07/2008, 2008.

[Cor08b] Microsoft Corporation. .net framework conceptual overview. Disponível em http://msdn.microsoft.com/en-us/library/zw4w595w(VS.85).aspx, último acesso em 12/07/2008, 2008.

[Cor08c] NVIDIA Corporation. Nvidia cuda programming guide 2.0. Disponível em http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, último acesso em 03/03/2008, jun 2008.

[CRY] CRYTEK. Cryengine specifications. Disponível em http://www.crytek.com/technology/index.php?sx=cryengine, último acesso em 22/03/2006.

[Dal03] Daniel Sánchez-Crespo Dalmau. *Core Techniques and Algorithms in Game Programming*. New Riders, 2003.

[dCed] Rogério Moraes de Carvalho. Métodos anônimos em c# 2.0. In *MSDN Magazine*. DevMedia, 2007. Ano 03 - 32ª ed.

[Dev] DevMaster.net. Devmaster's game and graphics engines database. Disponível em http://www.devmaster.net/engines/, último acesso em 22/03/2006.

[eaa] Jorrit Tyberghein et. al. Crystal space manual. Disponível em http://www.crystalspace3d.org/docs/download/manual/, último acesso em 22/03/2006.

[eab] Nikolaus Gebhardt et. al. Irrlicht: Características, notícias, tutoriais e documentação. Disponível em http://irrlicht.sourceforge.net/index.html, último acesso em 20/08/2006.

[eac] Russell Smith et. al. Open dynamics engine: v0.5. Disponível em http://ode.org/ode-latest-userguide.pdf, último acesso em 15/09/2005.

[ead] Steve Streeting et. al. Ogre manual v1.0.4. Disponível em http://www.ogre3d.org/docs/manual, último acesso em 15/09/2005.

[Ebe00] D. H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann, 2000.

[Ebe04] D. H. Eberly. *Game Physics*. Morgan Kaufmann, 2004.

[Erl01] Kenny Erleben. *En introducerende lærebog i dynamisk simulation af stive legemer*. Dissertação de Mestrado, The Department of Computer Science, University of Copenhagen, Denmark, may 2001.

[Erl04] Kenny Erleben. *Stable, Robust, and Versatile Multibody Dynamics Animation*. Tese de Doutoramento, The Department of Computer Science, University of Copenhagen, Denmark, nov 2004.

[Fea87] Roy Featherstone. *Robot Dynamics Algorithm*. Kluwer Academic Publishers, Norwell, MA, USA, 1987. ISBN 0898382300. Manufactured By-Kluwer Academic Publishers.

[Gama] Epic Games. Unreal technology. Disponível em http://www.unrealtechnology.com/html/homefold/home.shtml, último acesso em 22/03/2006.

[Gamb] Garage Games. Torque game engine (commercial license). Disponível em http://www.garagegames.com/products/31, último acesso em 22/03/2006.

[Geb] Nikolaus Gebhardt. Irrlicht engine 0.14.0 api documentation. Disponível em http://irrlicht.sourceforge.net/docu/index.html, último acesso em 22/03/2006.

[Gol80] H. Goldstein. *Classical Mechanics*. Addison Wesley, second edition, 1980.

[Gro] Gold Standard Group. Opengl overview. Disponível em http://www.opengl.org/about/overview/, último acesso em 22/03/2006.

[GV03] Jonas Gomes e Luiz Velho. *Fundamentos da Computacao Grafica*. IMPA, 2003.

[IdFC06] R. Ierusalimschy, L. H. de Figueiredo, e W. Celes. aug 2006.

[KZZ02] Konrad-Zuse-Zentrum. Rigid body simulation course. (course no. 178), 2002.

[Lab] Radon Labs. Nebula 2 documentation. Disponível em http://nebuladevice.cubik.org/documentation/nebula2/, último acesso em 22/03/2006.

[Lam03] André Lamothe. *Tricks of the 3D Game Programming Gurus Advanced 3D Graphics*. Editora SAMS, 2003.

[Lun03] Frank D. Luna. *Introduction to 3D Game Programming with DirectX® 9.0*. Wordware Inc., 2003.

[MB98] Tom McReynolds e David Blythe. Advanced graphics programming techniques usingopengl. In *SIGGRAPH 98 Course*. 1998.

[McG] Morgan McGuire. G3d manual. Disponível em http://g3d-cpp.sourceforge.net/html/index.html, último acesso em 22/03/2006.

[Oli06] L. L. Oliveira. *Um sistema de animação baseado em dinâmica de corpos rígidos articulados*. Dissertação de Mestrado, UFMS, Campo Grande - MS, 2006. In Portuguese.

[Per08] Márcio A. Peres. *Motor de Física de Corpos Rígids em GPU com Arquitetura CUDA*. Dissertação de Mestrado, Universidade Federal de Mato Grosso do Sul, Campo Grande - MS, 2008. In Portuguese.

[RE01] Samuel Ranta-Eskola. *Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering*. Dissertação de Mestrado, Information Technology Computing Science Department Uppsala University, Sweden, jan 2001.

[RS04] Subhankar Ray e J. Shamanna. Virtual displacement in lagrangian dynamics. Disponível em http://arxive.org/abs/physics/0410123, último acesso em 01/03/2008, 2004.

[Sha01] Ahmed A. Shabana. *Computational Dynamics - Second Edition*. Wiley, 2001.

[Sha05] Ahmed A. Shabana. *Dynamics of Multibody Systems*. Cambridge University Press, 2005. ISBN 0521850118, 9780521850117.

[Sof] Id Software. id software´s new technology licensing program. Disponível em http://www.idsoftware.com/business/technology/printdoc.html, último acesso em 22/03/2006.

[Stu] Artificial Studios. Reality engine overview. Disponível em http://artificialstudios.com/product_re.php, último acesso em 22/03/2006.

[Sul] J. Sullivan. Open scene graph tutorials. Disponível em http://openscenegraph.com/documentation/NPSTutorials, último acesso em 15/09/2005.

[TPSL97] J.C. Trinkle, Jong-Shi Pang, Sandra Sudarsk, e Grace Lo. Dynamic multi-rigid-body contact problems with coulomb friction. *Zeitschrift fur Angewandte Mathematik und Mechanik*, 77(4):267–279, 1997.

[Val05] Luis Valente. *Guff: A System for Game Development*. Dissertação de Mestrado, Universidade Federal Fluminense, Rio de Janeiro - RJ, 2005. In Portuguese.

[VCF05] Luis Valente, Aura Conci, e Bruno Feijó. Real time game loop models for single-player computer games. *WJOGOS 2005 Conference Proceedings*, páginas 89–99, 2005.

[WS99] Richard S. Wright e Michael Sweet. *OpenGL SuperBible*. Waite Group Press, 2 edição, dec 1999.

[ZD04] Stefan Zerbst e Oliver Duvel. *3D game Engine Programming*. Premier Press, 2004.