UM AMBIENTE DE ANIMAÇÃO DINÂMICA DE CORPOS RÍGIDOS

Leonardo de Lima Oliveira

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 Aristarco Pagliosa.

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

Campo Grande - MS
2006

Ouse fazer, e o poder lhe será dado! (Lair Ribeiro)

Aos meus pais Lourival e Durcelina, e irmãos Bersony e Alan.

Agradecimentos

Poderíamos lembrar das nossas conquistas como cenas de animações que ganham progresso à medida que o relógio "vida" trabalha. Em todas as minhas conquistas jamais fui o único ator das cenas. Muitos outros atores contribuíram direta ou indiretamente, com palavras ou atitudes, para que o resultado final da animação pudesse ser o mais encantador possível. Agradeço eternamente a Deus, o verdadeiro e único autor da animação a qual todos nós somos atores; pela força e recompensa justa aos meus esforços em tentar fazer valer a pena cada segundo dedicado neste trabalho.

Agradeço ao meu orientador e amigo Paulo Pagliosa, exemplo de honestidade e competência. Fica registrada aqui a minha eterna gratidão pela atenção, paciência e a confiança depositada em mim na conclusão de um trabalho de dimensão considerável.

Agradeço aos professores e funcinonários do Departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul pela atenção e presteza que tiveram em toda a minha vida acadêmica.

Agradeço a CAPES pelo apoio financeiro.

Agradeço ao meu amigo Breno Ribeiro pelo apoio e exemplo de amizade e caráter. Pelas noites em claro em dedicação aos trabalhos da disciplina de compiladores (ainda na graduação) e que contribuíram consideravelmente no conhecimento adquirido para o desenvolvimento de parte deste trabalho.

Agradeço aos professores da Escola Rui Barbosa da cidade de Imperatriz/MA, os quais ensinaram-me com maestria o gosto e a importância dos estudos; além de muitos outros amigos e amigas daquela cidade que souberam provar que na amizade não há distância; sempre com palavras de apoio e carinho.

Resumo

Oliveira, L.L. *Um Ambiente de Animação Dinâmica de Corpos Rígidos.* Campo Grande, 2006. Dissertação de Mestrado — Universidade Federal de Mato Grosso do Sul.

O objetivo geral deste trabalho é o estudo dos fundamentos da animação por computador e o desenvolvimento orientado a objetos de um sistema de animação procedimental de cenas 3D para visualização de simulações dinâmicas em aplicações de ciência e engenharia. Uma animação é especificada através de uma linguagem de animação, derivada de uma linguagem de propósito geral chamada L, estendida com produções para descrição de roteiros de animação baseados em *scripts* e ações que modificam o estado dos objetos no tempo, além de uma API de animação. Os principais componentes do sistema são: compilador da linguagem de animação, máquina virtual de animação, renderizador, controlador de animação, ligador e visualizador de arquivos de animação, e um motor de física de corpos rígidos denominado PhysX, desenvolvido pela Ageia Technologies. Os objetivos específicos do trabalho estão divididos em duas etapas. A primeira é o estudo dos fundamentos matemáticos e computacionais necessários ao desenvolvimento das classes de objetos que compôs o sistema. Tal estudo inclui técnicas de animação e controle de movimentos, teoria de compiladores, máquinas virtuais, gerência de memória e coleta de lixo, bibliotecas nativas e conceitos da mecânica clássica. A segunda etapa consiste no desenvolvimento dos componentes do sistema, sua integração com o motor de física, e a implementação da API de animação.

Palavras-chave: *animação*, *linguagem de animação*, *dinâmica de corpos rígidos*, *programação orientada a objetos.*

Abstract

Oliveira, L.L. *Um Ambiente de Animação Dinâmica de Corpos Rígidos.* Campo Grande, 2006. MSc dissertation — Universidade Federal de Mato Grosso do Sul.

The general purpose of this work is the study of the fundamentals of the computer animation and the object-oriented development of a procedural animation system for visualization and dynamic simulations of 3D scenes in science and engineering applications. In such system an animation is specified by an animation language AL and an animation API. AL was derived from a general-purpose language called L, which was extended with productions for description of scripts and actions that modify the state of scene objects over time. The main components of the system are: animation language compiler, animation virtual machine, renderer, animation controller, animation file viewer and animation file linker, and a physics engine for dynamic simulation of rigid bodies called PhysX, developed by the Ageia Technologies. The specific purposes of the work are divided in two stages. The first one is the study of mathematical and computational principles and methods for developing the object classes that compose the animation system. Such study includes animation techniques and movements control, compilers theory, virtual machines, memory and garbage collection management, native interfaces and classic mechanics concepts. The second stage is the development of the system components, including the integration with the physics engine, and the animation API implementation.

Keywords: *animation*, *animation language*, *rigid bodies dynamics*, *object-oriented programming.*

Conteúdo

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

1.1 Motivação e Justicativas

O Grupo de Visualização, Simulação e Games (GVSG) do DCT/UFMS tem como um de seus objetivos o desenvolvimento de *aplicações de modelagem* destinadas à simulação e visualização de modelos computacionais de meios contínuos tridimensionais. As aplicações de modelagem desenvolvidas no GVSG têm sido implementadas a partir de um *framework* denominado OSW [31]. Os componentes do *framework* podem ser empregados, por exemplo, em aplicações de análise não-linear estática e dinâmica de sólidos pelo método dos elementos finitos ou método dos elementos de contorno, mas ainda não há recursos adequados para visualização ao longo do tempo de resultados da análise dinâmica.

Este trabalho é parte do projeto de pesquisa que pretende iniciar a expansão das capacidades de OSW visando seu emprego também no desenvolvimento de aplicações de modelagem dinâmica. Modelagem dinâmica significa a utilização de modelos computacionais para visualização e compreensão da estrutura e do comportamento de objetos ao longo do tempo. Como conseqüência, os métodos computacionais numéricos de análise destes modelos não fornecem somente um conjunto de resultados, como nos problemas estáticos, mas vários conjuntos de dados, um conjunto para cada instante de tempo do intervalo de tempo considerado. Devido ao caráter temporal dos resultados da modelagem dinâmica, seria conveniente a visualização dos resultados ao longo do tempo de análise, o que conduz naturalmente ao emprego de técnicas de animação.

Como exemplo de simulação dinâmica, considere uma esfera de material contínuo perfeitamente elástico que, sob ação de forças gravitacionais, cai em queda livre de uma determinada altura e choca-se com uma superfície plana, também contínua e elástica. Como resultado da simulação, seria interessante observar a trajetória percorrida pela esfera e as deformações por ela sofrida após os choques contra a superfície plana. Até mesmo para este problema, cujos resultados em termos visuais são intuitivamente vislumbráveis, a complexidade de implementação é considerável. Além da análise numérica e da computação gráfica em si, o sistema de animação da aplicação de modelagem deveria ser capaz de armazenar a animação resultante da simulação em meio persistente (usualmente disco rígido ou CD) e reproduzir a animação na velocidade adequada. Para isso, existem vários componentes e ferramentas disponíveis (inclusive gratuitas) que podem ser imediatamente empregadas ou convenientemente adaptadas. Porém, é mais complicado implementar os mecanismos através dos quais os componentes de uma animação, isto é, os atores, luzes, câmeras e ações envolvidos, são repre-

sentados, armazenados e manipulados pelo sistema de animação para geração da seqüência animada.

Animação é vista em vários países como setor estratégico dentro da indústria tecnológica. Muito disso deve-se ao fato de que esta pode ser desenvolvida para servir como ferramenta de visualização e simulação usada em diversas áreas: na indústria aeronáutica (com os simuladores de vôo), na indústria automobilística (com os simuladores de colisões de automóveis), na indústria de entretenimento (com os jogos interativos), bem como na robótica (com a dinâmica do movimento dos robôs). Em todas essas áreas o realismo de uma animação é um fator considerável e, dentre outras estratégias, as leis da física podem ser utilizadas para prover este realismo.

O interesse pelo acréscimo de realismo é resultado não somente do aumento da velocidade das CPUs, mas também da evolução das unidades de processamento gráfico (GPUs), as quais implementam em hardware as funções de renderização. Como conseqüência desta folga da capacidade de processamento, modelos físicos mais realistas podem ser usados e ainda fornecer respostas em tempo real; uma característica tipicamente aplicada na indústria de jogos digitais [11], por exemplo. Além disso, já está disponível no mercado a primeira unidade de processamento de física (PPU), a AGEIA PhysX SDK [1], cuja `API` de física é utilizada neste trabalho. O emprego de PPUs promete ser para simulação física o que o uso de GPUs é para gráficos.

1.2 Animação: Conceitos, Técnicas e Sistemas

Animação é, literalmente, a arte de "dar vida" a objetos. Embora se possa pensar em animação como sinônimo de deslocamento, o termo pode ser genericamente aplicado a todas as alterações do estado interno de um objeto que produzem, ao longo do tempo, variações de efeito visual. Uma animação pode incluir modificações da posição, forma, cor, transparência, estrutura e textura de um objeto, bem como modificações da iluminação, posição, orientação e foco de uma câmera, por exemplo. Animação pode ser definida como a apresentação de uma seqüência de imagens individuais, ou *quadros*, que, quando exibidas rapidamente, fornece a sensação de movimento [44]. Velocidades típicas de animação, definidas em termos do número de quadros exibidos por segundo (qps), são, por exemplo, 30 qps para o videotape e 24 qps para o filme fotográfico.

Ainda na década de 60, o computador começou a ser utilizado como ferramenta de apoio na confecção de filmes criados através do processo de animação tradicional (também conhecida como animação convencional) [19], o qual pode ser descrito, genericamente, pela seguinte seqüência de etapas [44]:

1. **Definição da estória**. O escritor escreve uma estória narrativa que descreve os cenários e os personagens da animação, suas aparências, diálogos e ações.

2. **Storyboard**. É uma sinopse gráfica da animação que ilustra sua aparência e o fluxo da estória. O número de ilustrações não é importante; o que realmente importa é que o *storyboard* deve representar os momentos principais da animação. Notemos que a animação é composta por um conjunto de seqüências que definem ações específicas. Cada seqüência, por sua vez, é composta por um conjunto de cenas definidas pelos atores que dela participam, além de luzes e câmeras. Uma cena é formada por um conjunto de quadros que correspondem às imagens individuais da animação.

3. **Roteiro**. O roteiro define o posicionamento e os movimentos precisos dos atores, luzes e câmeras da animação ao longo do tempo.

4. **Desenho**. Desenho dos personagens e das ações a serem executadas.

5. **Trilha sonora**. Na animação convencional, a trilha sonora deve preceder o processo de animação, visto que o movimento deve ser "casado" com os diálogos e com a música.

6. **Animação**. Nesta etapa, os animadores desenham os quadros-chave (*keyframes*) da animação e os quadros intermediários entre dois quadros-chave, denominados de *in-betweens*. Os quadros intermediários interpolam a animação entre os quadros-chave.

7. **Cópia e pintura**. Os quadros desenhados a lápis são transferidos para folhas de acetato e pintados, com paciência e precisão.

8. **Verificação e fotografia**. Os animadores verificam se as ações em suas respectivas cenas estão corretas, antes da fotogravação dos quadros em um filme colorido.

9. **Edição final**. É a etapa de pós-produção da animação.

Algumas etapas do processo de animação convencional podem ser automatizadas em computador. Uma animação é auxiliada por computador se as etapas de criação dos desenhos, criação dos movimentos, pintura, fotogravação e pós-produção são executadas com o auxílio de programas de computador. Uma animação é modelada por computador quando o animador determina os atores, o ambiente e as ações a serem executadas, e entrega o controle total da animação a um programa de computador. A Figura 1.1, adaptada de [15], mostra uma taxomia da animação.

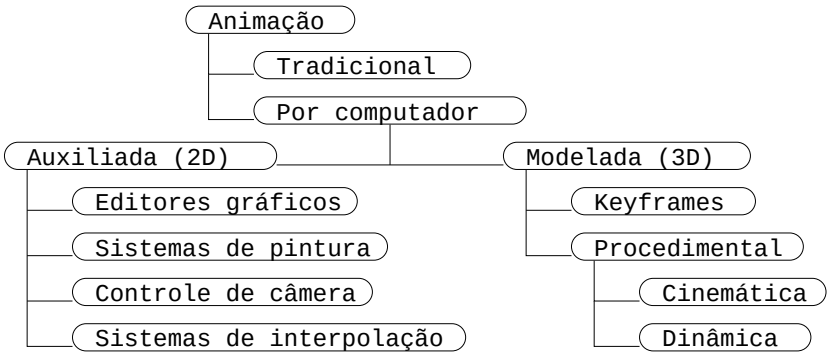


Figura 1.1: Taxonomia da animação.

Segundo a taxonomia da Figura 1.1, o sistema desenvolvido neste trabalho é de animação modelada por computador, procedimental (animação descrita por algoritmos através de uma linguagem) e dinâmica, onde a física é utilizada para prover realismo. O processo de criação de uma animação deste tipo pode ser caracterizado pelas seguintes etapas [36]:

- **Simulação**. As simulações fornecem, geralmente, os movimentos da animação para aplicações científicas. Os cientistas e engenheiros familiares com o domínio da aplicação são os responsáveis pela configuração e execução dessas simulações, as quais podem consumir, de forma intensiva, os recursos de processamento e armazenamento do computador.

- **Modelagem**. Cada ator da animação deve ter um modelo geométrico que descreve sua aparência. Durante o processo de renderização, a descrição matemática desse modelo é convertida para uma porção de imagem bidimensional.

- **Preview**. Antes da conclusão da seqüência da animação e da etapa de fotogravação, uma pré-visualização rápida da animação pode ajudar a estabelecer a aparência geral do produto final, e em quanto tempo será obtido.

- **Renderização**. O processo de renderização é o responsável pela produção de imagens dos modelos geométricos, considerando-se as interações das fontes de luz com os materiais que constituem suas superfícies.

Outras estratégias procedimentais, entretanto, podem ser utilizadas em animações, como por exemplo, animação baseada em máquinas de estados [7], em cinemática (veja Seção 1.2.1) e aquelas dirigidas por inteligência artificial. Contudo, um sistema de animação modelado por computador é, segundo Camargo [5], composto fundamentalmente pelos seguintes componentes:

- **Modelador geométrico**, para que se possa definir e modelar as características geométricas dos atores e cenários envolvidos na animação.

- **Controlador da animação**, para que se possa definir e executar o controle das interações entre os atores e entre os atores e o cenário.

- **Renderizador e visualizador**, para que se possa definir os atributos de cor e de textura dos objetos, além de visualizar cada quadro gerado durante o processo de animação.

A taxonomia da animação é descrita tomando como fundação as técnicas para o controle de movimentos. Estas técnicas, entretanto, são utilizadas como base para a classificação das técnicas de animação. Por esta razão, existe uma interseção entre técnicas de animação e técnicas para o controle de movimentos.

1.2.1 Técnicas para Controle de Movimentos

O controle de uma animação é baseado na manipulação de parâmetros dos objetos da cena ou da imagem (iluminação, câmera, etc.). A principal tarefa do animador é selecionar e controlar os parâmetros apropriados para a obtenção do efeito desejado [2]. O controle do movimento/interação dos atores é, de acordo com Camargo [5], o maior desafio da área de animação por computador. Embora existam tecnologias padronizadas para representação de objetos geométricos (b-rep, CSG, etc.) [27] e para a síntese de imagens (*scan-line*, *ray tracing*, etc.) [13, 40, 47], não existe um algoritmo genérico para o tratamento da movimentação/interação de objetos em uma animação. Definir o movimento para cenas tridimensionais complexas pode ser complicado e consumir muito tempo, devido, em grande parte, à quantidade de informação que deve ser manipulada [45, 50, 52].

Na tentativa de resolver o problema do fluxo de uma animação, Camargo [6] apresenta a divisão de um modelo que controla os movimentos numa animação em partes denominadas blocos de *controle local* e blocos de *controle global*. Em uma animação composta por diversos atores, por exemplo, atribui-se um bloco de controle local que conterá as "regras de comportamento de cada um". Uma vez que o sistema seja composto por muitos atores, podem ocorrer interações entre os atores e os animadores. O bloco de controle global sugere uma abordagem lógica para a determinação de um modelo para estas interações.

Nesta seção abordamos a animação por interpolação, animação cinemática e dinâmica. As duas últimas podem ainda ser divididas em cinemática direta, cinemática inversa, dinâmica direta e dinâmica inversa. Uma taxonomia das técnicas para o controle de movimento é ilustrada na Figura 1.2, adaptada de [52].

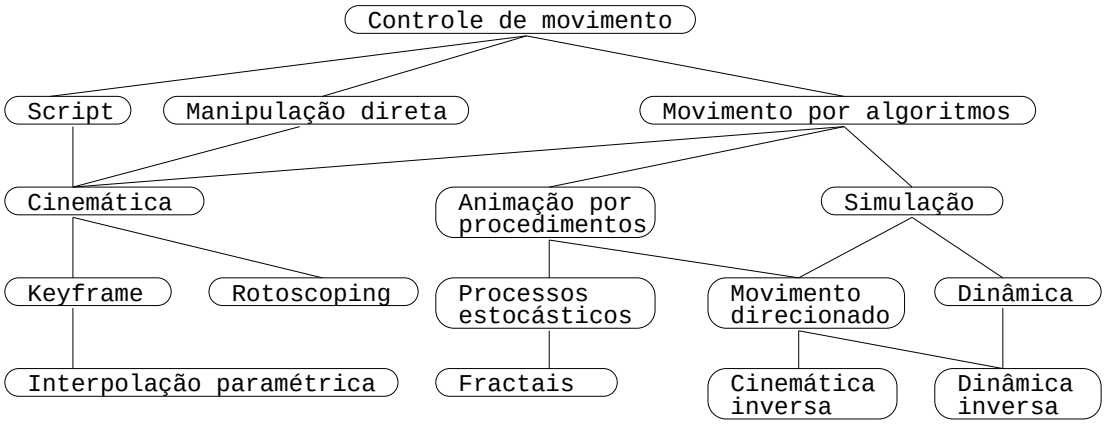


Figura 1.2: Taxonomia das técnicas para o controle de movimentos.

Animação por Interpolação

Na animação tradicional o animador desenha um ator em uma posição e depois faz outro desenho com o ator em uma segunda posição. Estes dois desenhos são normalmente passados para outro desenhista, que então desenhará os quadros intermediários (*inbetweens*). A animação por interpolação de *keyframes* é a transposição desta técnica para os computadores.

A essência da animação por interpolação de *keyframes* é a determinação de correspondência entre os quadros-chave, isto é, quais pontos de um quadro-chave deverão ser mapeados em quais pontos do quadro-chave seguinte. É exatamente nesse ponto que se encontram as maiores dificuldades na automatização deste processo, já que cada *keyframe* é uma projeção bidimensional de um objeto 3D e, portanto, informações são perdidas (informações estas que podem ser essenciais ao longo da interpolação). Na animação convencional este problema não existe, pois o artista conhece (ou pelo menos é capaz de imaginar) o modelo 3D do ambiente e dos objetos da cena.

Esse impasse pode ser resolvido se for usada a interpolação paramétrica [2, 52], onde são dados valores-chave para certos parâmetros da imagem, valores estes que serão interpolados nos *inbetweens*. Com a imagem definida a partir de parâmetros, não há mais o problema da perda de informações tridimensionais, já que elas estão embutidas nos parâmetros a serem interpolados.

Animação Cinemática

Ao invés de determinar posições-chave para um parâmetro e interpolá-las, o animador pode especificar uma posição (ou valor) inicial e uma função do tempo que descreva as modificações deste parâmetro. Por exemplo, é possível animar a queda livre de um objeto colocando-o em uma altura inicial e calculando sua posição em cada quadro a partir de sua equação de queda livre. De forma análoga, um objeto lançado com certa velocidade inicial em uma direção que forma com a horizontal um determinado ângulo, desprezada a resistência do ar, tal objeto ficará sob a ação exclusiva de seu peso e, portanto, sujeito apenas à aceleração da gravidade. A trajetória descrita, em relação à Terra, é uma parábola e pode ser determinada, nos quadros da animação, através de sua equação de lançamento oblíquo. Este método de controle é chamado cinemático porque os movimentos dos objetos são controlados por funções de posição, velocidade e aceleração.

Animação Dinâmica

Quando o objetivo de uma animação é a simulação de um processo físico, um modelo mais sofisticado que o cinemático deve ser utilizado para que o resultado final apresente um aspecto mais convincente.

O modelo dinâmico usa forças e torques aplicados aos objetos, além dos respectivos momentos e produtos de inércia para determinar os movimentos. Assim, no exemplo de queda livre, o movimento seria calculado a partir do peso do objeto, além de outras forças que poderiam estar atuando sobre o mesmo. A posição, velocidade, acelerações, deformações e tensões dos objetos da cena são determinadas, a partir de uma configuração inicial, pelas leis da dinâmica (usualmente Newtoniana).

A grande vantagem do modelo dinâmico é criar modelos fisicamente corretos. Entretanto, é exigido o conhecimento de todas as forças que atuam sobre os objetos, o que nem sempre é simples ou possível. Assim, o modelo dinâmico apresenta um grau de complexidade maior que o cinemático, com maior número de variáveis a serem controladas. De uma maneira geral, as simulações dinâmicas apresentam resultados mais realistas que as cinemáticas, mas o controle do movimento através de forças e torques não é tão natural ao animador. Por estas razões, a escolha entre a utilização do modelo cinemático ou dinâmico deve ser criteriosa. Em [5] é traçado um paralelo mais detalhado entre modelos cinemáticos e dinâmicos.

1.2.2 Técnicas de Animação

As técnicas de animação por computador podem ser categorizadas unindo tipo e natureza dos objetos que serão animados. Tipicamente, o movimento de objetos é descrito em termos de *graus de liberdade* (DOFs — *degrees of freedom*), ou seja, o número de coordenadas independentes necessárias para especificar as posições de todos os componentes do sistema [50]. A seguir, discutimos algumas dessas categorias, as quais não encerram, evidentemente, todas as técnicas de animação. Por exemplo, excluímos da discussão as técnicas que englobam o estudo de corpos frágeis [51] e animação de objetos deformáveis [17]. Maiores detalhes podem ser encontrados em [48].

- **Animação de corpos rígidos**. Na animação de corpos rígidos, produzimos seqüências animadas gerando imagens de uma cena com objetos em diferentes posições, ou então movendo a câmera, e gravando o resultado em quadros. É a categoria de animação fundamental, mais simples de implementar animação por computador e a mais utilizada em animações. Este tipo de animação foi criada a partir de extensões de programas que fazem desenhos de cenas tridimensionais. Tecnicamente, um corpo rígido é definido por um número de pontos que devem ser movimentados juntamente. Um ponto não pode se mover em relação ao outro, mas os pontos movem-se como um todo, como no caso de pontos que definem polígonos e superfícies de forma livre. O movimento de um corpo rígido é especificado por seis graus de liberdade, translação e rotação nos eixos x, y e z.

- **Animação de estruturas articuladas**. Estruturas articuladas em computação gráfica são modelos usados para simular, por exemplo, quadrúpedes e bípedes. A dificuldade de definir um roteiro de movimento de estruturas articuladas é uma função da complexidade dos objetos e da complexidade dos movimentos desejados. Camargo [5] provê um modelo cinemático para estruturas articuladas. Usualmente os animadores estão interessados em estruturas articuladas complexas (humanos e animais, por exemplo) e isto implica numa dificuldade maior no controle dos movimentos. Tecnicamente, corpos articulados são compostos por segmentos cujos movimentos em relação aos outros segmentos obedecem a certas restrições. O corpo humano, por exemplo, é freqüentemente

representado como segmentos rígidos unidos por articulações (juntas) que possuem três graus de liberdade. O número total de graus de liberdade que deve ser especificado é a soma dos números dos graus de liberdade de cada junta.

- **Simulação dinâmica**. Simulação dinâmica significa usar as leis da física para simular os movimentos. A motivação aqui é que estas leis produzem movimentos mais realistas e a possibilidade de simulação de uma grande variedade de fenômenos físicos. A desvantagem da simulação dinâmica é que ela tende a privar o controle artístico por parte do animador, uma vez que este deve dominar suficientemente tais leis para exprimir seu dom artístico.

- **Animação de partículas**. Animação de partículas significa animar individualmente uma grande quantidade de corpos diminutos para simular alguns fenômenos, visualizados como um movimento completo (cada partícula contribui para um todo) [38]. Explosão de fogos de artifícios é um exemplo. As partículas possuem cada uma um roteiro de animação próprio. Tecnicamente, uma partícula pode ser descrita por um ponto no espaço tridimensional (x, y, z). O posicionamento e movimento de determinados pontos são descritos por três variáveis, portanto, um ponto tem três graus de liberdade de movimento. A animação de um ponto requer uma tripla de números para cada quadro da animação, ou três funções descrevendo a variação das coordenadas em relação ao tempo.

- **Animação comportamental**. Animação comportamental ou por comportamento é aquela em que o animador descreve um conjunto de regras para a maneira como um ou mais objetos da cena reagirão com o ambiente e entre eles. A animação por comportamento é similar à animação de partículas, mas com roteiros independentes.

O tipo e a natureza dos objetos a serem animados, como mencionado, podem ser de uma variedade considerável, ou seja, podemos abstrair objetos cujas suas propriedades influenciam no seu comportamento no decorrer de uma animação; quando estes submetidos a condições diversas. As técnicas de animação correspondem a um ferramental fundamental quando se deseja animar tais objetos. No sistema de animação foco deste trabalho, optamos por oferecer em um único sistema as cinco técnicas de animação descritas. As quatro primeiras implementadas/controladas internamente, deixando livre o animador no que diz respeito aos detalhes de baixo nível, como por exemplo, o controle das restrições em estruturas articuladas, a aplicabilidade da física Newtoniana sobre os objetos, a garantia da rigidez dos corpos e etc. A última técnica é gestionada por meio de *scripts*, ações e eventos especificados em uma linguagem.

1.2.3 Sistemas de Animação

Os sistemas que lidam com a variação temporal de elementos de imagens geradas por computador são chamados sistemas de animação por computador [34]. De forma geral, os sistemas de animação podem ser classificados de diferentes maneiras. Alguns autores optam por classificar-los em *sistemas de animação procedimental* e *sistemas gráficos* [54]. Em um sistema gráfico não é exigido do animador o conhecimento prévio de alguma linguagem de programação específica. Todo o controle da animação é feito graficamente por intermédio de uma interface. Um exemplo conhecido é o Autodesk 3D Studio. Um sistema de animação procedimental, por outro lado, utiliza roteiros que determinam como os parâmetros da animação variarão no decorrer do tempo. A vantagem deste tipo de sistema sobre os sistemas gráficos é permitir um maior controle dos movimentos por parte do animador, facilitando a criação de movimentos complexos. O sistema deste trabalho é de animação *procedimental*. Thalmann [44] opta por classificar os sistemas de animação por níveis:

- **Nível 1**. São basicamente os editores gráficos utilizados unicamente para criar, pintar, armazenar, recuperar e modificar desenhos interativamente.
- **Nível 2**. São sistemas utilizados para calcular *inbetweens* e mover objetos ao longo de uma trajetória. Burtnyk [4] propõe uma técnica para sistemas desse nível.
- **Nível 3**. São sistemas que permitem ao animador operar sobre os objetos (rotações e translações, por exemplo) da animação além de operações sobre câmeras virtuais (*zoom*, *pan* ou *tilt*).
- **Nível 4**. São sistemas que provêem uma forma de definir ações próprias para os atores de uma animação.
- **Nível 5**. São os denominados sistemas "inteligentes". O computador é capaz de modelar atores capazes de aprender à medida que realizam suas tarefas. Este aprendizado pode consistir, por exemplo, na capacidade do ator alterar seu movimento em função da ocorrência de determinados eventos.

Os sistemas de animação auxiliada por computador geralmente encontram-se nos níveis 1 e 2. Por sua vez, os sistemas de animação modelada por computador geralmente situam-se nos níveis 3, 4 e 5.

Na literatura, encontramos uma variedade de ferramentas e sistemas de animação por computador. Magalhães e Raposo [37, 26] provêem uma ferramenta cinemática para animação. Sua principal característica é a presença de uma linguagem de roteiros (uma extensão da linguagem C) e uma interface para criação dos mesmos. Outro exemplo é o sistema de animação baseado em software livre Blender (www.blender.com.br). Tal sistema é voltado para modelagem, renderização 3D e a animação propriamente dita e já possui uma comunidade de usuários.

O sistema ASAS (*Actor/Scriptor Animation System*), proposto por Reynolds [39], é um exemplo clássico de sistema de animação procedimental. O ASAS é um sistema orientado a objetos, baseado em LISP. Os objetos básicos de um *script* em ASAS são os atores, que são vistos como entidades especiais dotadas de recursos para a troca de mensagens. Cada ator pode controlar um ou mais aspectos da animação. Enquanto ativo, cada ator será executado uma vez a cada quadro da animação. Um ator pode ser ativado e desativado pelo roteiro (que pode ser visto como a função *main* de um programa) ou por um outro ator. O ator também pode desativar a si mesmo. A linguagem do ASAS provê grande flexibilidade ao animador. O conceito de ator pode levar inclusive ao desenvolvimento de animações comportamentais, pois os atores podem ser definidos para saberem responder a determinados estímulos. O preço dessa flexibilidade é uma sintaxe complexa, um forte obstáculo ao animador/programador inexperiente.

Tentando criar um sistema acessível a artistas não programadores, Thalmann [45] apresentou o MIRANIM, composto de um sistema orientado ao animador (o ANIMEDIT) e uma sublinguagem de roteiros (a CINEMIRA-2). Com o ANIMEDIT, o animador pode especificar um roteiro de animação completo, sem nenhum tipo de programação. O animador, segundo Thalmann, pode criar atores com seus movimentos e transformações, bem como câmeras virtuais, com seus movimentos e características.

Zeleznik [53] desenvolveu outro sistema de modelagem e animação orientado a objetos que facilitou a integração de vários paradigmas de animação. Os objetos podem ser geométricos ou não-geométricos (câmeras, luzes, etc.) e trocam mensagens entre si. A lista de mensagens de um objeto determina seus parâmetros variantes no tempo e seu comportamento. Esta lista pode ser alterada pelo animador e por outros objetos (interação ator-ator e ator-animador). Este sistema também explora as noções de controle global e controle local [6], porque uma

mensagem enviada a um objetos é abstrata, cabendo ao próprio objeto definir como ela o afetará. A mensagem determina o que será feito, e o objeto determina como fazê-lo. O roteiro de animação deve ser escrito numa linguagem própria, capaz de descrever objetos e suas mensagens.

O CLOCKWORKS [8] é um sistema de animação orientado a objetos que engloba a modelagem geométrica de objetos (CSG), o controle de movimentos, a renderização e o pós-processamento de imagens (através da superposição de imagens, útil quando a ação ocorre em frente a um fundo fixo). Pode ser usado como ferramenta de projeto em engenharia através da criação e visualização de formas complexas, além de também ser utilizado como ferramenta de animação. Para o controle do movimento, CLOCKWORKS faz uso de interpolação da *keyframes* e/ou uma linguagem de roteiros. A técnica de *keyframes* é utilizada através de ferramentas interativas, embora sempre seja gerado automaticamente um roteiro a ser executado.

IMPROV [33] é um sistema para especificação de roteiros entre atores interativos. Ele consiste em dois subsistemas. O primeiro é um motor de animação que utiliza técnicas procedimentais para capacitar os animadores na criação de movimentos em camadas, contínuos e não repetitivos, além da suavização dos movimentos. O segundo subsistema é um motor comportamental que capacita o animador a criar sofisticadas regras que administram como os atores se comunicam, mudam e tomam decisões. A combinação dos subsistemas fornecem um conjunto integrado de ferramentas para associar as "mentes" e os "corpos" de atores interativos. Um ator do sistema pode realizar tarefas ao mesmo tempo. Estas atividades simultâneas podem atuar de diferentes formas, caracterizando o fluxo da animação.

1.3 Objetivos e Contribuições

O objetivo geral deste trabalho é o estudo dos fundamentos da animação por computador e o desenvolvimento de um sistema de animação dinâmica de cenas constituídas por corpos rígidos. Os objetivos específicos são:

- Estudo dos fundamentos matemáticos e computacionais necessários ao desenvolvimento das classes de objetos que comporão o sistema de animação. Tal estudo inclui técnicas de animação e controle de movimentos, teoria de compiladores, máquinas virtuais, gerência de memória e coleta de lixo, bibliotecas nativas e conceitos da mecânica clássica, conforme justificado na próxima seção;
- Desenvolvimento e/ou integração dos componentes do sistema de animação;
- Descrição das principais classes de objetos que comporão a implementação do sistema e da `API` de animação.

As contribuições pretendidas com o trabalho são:

- Criação de um ambiente favorável à aplicação de recursos adequados para visualização ao longo do tempo de resultados da análise dinâmica pelo GVSG;
- Prover uma fundação de componentes que possam ser utilizados como fonte de pesquisa e aprendizado para os alunos de graduação em ciência da computação da UFMS;
- Implementação do núcleo de um sistema que possa ser estendido e/ou adaptado ao escopo de jogos digitais e aplicações de tempo real.

1.4 Visão Geral do Sistema de Animação

O sistema de animação deste trabalho foi desenvolvido para atuar como ferramenta de visualização de simulações dinâmicas em aplicações de ciência e engenharia. Para torná-lo tão adequado quanto possível, foi criada uma linguagem de animação LA para descrever os objetos e roteiros de uma animação. Em adição, é disponibilizada uma interface para programação de aplicativos (API) a qual implementa classes que representam cenas e seus componentes (atores, luzes, câmera, etc.), *scripts*, ações e eventos. A partir dessas classes base, novas classes podem ser derivadas para aplicações específicas.

A *linguagem de animação* LA é derivada de uma linguagem de propósito geral chamada L, híbrida (ou seja, provê mecanismos para definição de funções globais e estruturas de dados bem como características de programação orientada a objetos, como C++) na qual foram adicionadas produções para facilitar a criação de componentes de cenas e especificação de *scripts*, ações e eventos. O sistema de animação tem como componentes principais um compilador da linguagem LA e uma máquina virtual de animação (MVA) que executa o *bytecode* compilado e controla o ciclo de atualização de uma animação. Eficiência pode ser alcançada com métodos nativos, ou seja, métodos cujos corpos são implementados em uma linguagem diferente da linguagem de animação (usualmente C++). Muitos métodos da API de animação do sistema são nativos.

Embora nós pudéssemos ter estendido uma linguagem de propósito geral já existente (tal como Java [21]) decidimos implementar uma linguagem de animação própria devido às facilidades de extensão da gramática da linguagem L, acrescentando novos recursos sintáticos para o controle de movimentos e para criação de roteiros, além da possibilidade de reunir em uma única linguagem características encontradas em outras (como sobrecarga de operadores, definições de propriedades, etc.). Além disso, a linguagem L e um protótipo rudimentar da sua máquina virtual MVL foram objetos de estudos em anos anteriores pelo autor deste trabalho. Pelo fato de LA ser derivada de L e a MVA derivada de MVL, foi facilitada a integração da máquina virtual com outros componentes do sistema; e em um futuro, prover facilidades para adicionar novas características na linguagem, tal como regras comportamentais para atores.

No sistema de animação, *scripts* e ações (genericamente conhecidos como *seqüenciadores*) podem ser usados para criar novos objetos e controlar no tempo quaisquer mudanças sobre o estado dos objetos em uma simulação (não se tratando somente de movimento, mas também da aparência, atuando forças e torques, etc.). Em adição, um dos componentes do sistema de animação é um motor de física responsável por computar os efeitos de restrições dinâmicas sobre os objetos. A versão deste trabalho utiliza o motor de física PhysX SDK [1] desenvolvido pela AGEIA Technologies para detecção de colisão e simulação dinâmica de corpos rígidos.

1.4.1 Arquitetura

A arquitetura do sistema de animação é definida pelos seguintes componentes: compilador da linguagem de animação, MVA, ligador de arquivos de animação, e visualizador de arquivos de animação.

O *compilador da linguagem de animação* (CLA) é derivado do compilador de L. Ele toma como entrada arquivos contendo especificações em LA de uma ou mais cenas a serem animadas (arquivos scn), e produz como saída o arquivo objeto correspondente (arquivo oaf). Neste arquivo contém a coleção de *bytecodes* que são carregados e interpretados pela MVA. Um código de animação típico cria cenas e iniciam *scripts* e ações que modificam o estado da cena no tempo. Cenas são animadas seqüencialmente pelo sistema no instante em que são iniciadas; a cena animada em um dado tempo é chamada de *cena corrente*.

Durante uma simulação, a MVA renderiza quadros da cena corrente que podem ser enpacotados pelo *ligador de arquivos de animação* para a produção de filmes em uma variedade de formatos (avi, mpeg, flic, dentre outros), o qual pode ser assistido com o *visualizador de arquivos de animação*. Neste trabalho, o ligador e visualizador são ferramentas de terceiros já desenvolvidas específicamente para tal e não serão discutidas. Na ligação dos quadros foi utilizado o Fast Movie Processor [46], de licensa *shareware* porém gratuito para uso não comercial. Para visualização dos arquivos de animação foi usado o Avi Preview [22], também de licensa gratuita.

O sistema de animação gira em torno da MVA, o componente mais importante. Ela é formada pelos subcomponentes ilustrados no diagrama UML da Figura 1.3.

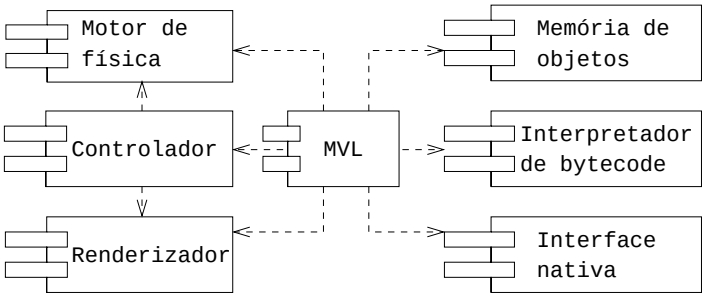


Figura 1.3: Arquitetura da MVA.

O núcleo da MVA é a máquina virtual de L. Esta é composta do interpretador de *bytecode*, da interface nativa e da memória de objetos.

O *interpretador de bytecode* é um processador virtual que executa o código objeto, isto é, as instruções representadas pelos *bytecodes* gerados pelo CLA. A coleção de *bytecodes* representam instruções para criar novos objetos, invocar métodos, tratar exceções, etc.

Métodos nativos são executados com o auxílio da *interface nativa* (IN). Quando um método nativo é invocado, a MVL empilha sobre a pilha nativa os argumentos passados para o método e também uma referência para o objeto IN, em seguinda, invoca a função nativa que implementa o método, empilhando o valor de retorno (se houver) sobre a pilha da MVL. Um código nativo pode utilizar o objeto IN para acessar muitas das funcionalidades da MVL, tal como criar novos objetos, invocar métodos, etc.

A *memória de objetos* é o local onde residem todos os objetos criados pela aplicação. Quando um objeto não puder ser alcançado pela MVL, um coletor de lixo automaticamente recupera a memória utilizada pelo objeto.

O *controlador* é o componente responsável por orquestrar a execução dos *scripts* e ações de uma animação (podendo ser comparado ao "diretor" do filme). O tempo de duração total de uma simulação é dividido em um número discreto de passos de tempo chamados *ticks*. Em cada *tick* de tempo, o controlador determina quais as partes de (*scripts* e ações) código devem ser executadas pela MVL para prover a atualização do estado da cena corrente. No final do ciclo de atualização, o controlador invoca o motor de física para dirigir a simulação de física.

Para um número positivo de *ticks* chamado *resolução*, o renderizador toma a cena corrente e renderiza um quadro da cena. A implementação atual do sistema utiliza um renderizador baseado em OpenGL (Open Graphics Library).

1.4.2 Trabalhos Relacionados

A principal característica do sistema deste trabalho que o classifica como um sistema procedimental, é a *linguagem de animação*. Tal como a ferramenta proposta por Magalhães [26], esta linguagem permite descrever roteiros de animação, não se tratando, entretanto, de uma linguagem puramente procedimental como C, mas uma linguagem híbrida, podendo-se especificar roteiros estruturadamente ou com uma visão orientada a objetos, oferecendo, desta forma, todas as características deste paradigma de programação[1].

O ASAS de Reynolds [39], como mencionado, aborda o fluxo de uma animação por meio de ativação, desativação e execução *de atores*. Esta idéia é interessante pois trata cada ator como uma forma independente no que diz respeito a seu comportamento individual, isto quer dizer que as regras que regem os aspectos *da animação* (que é uma abordagem global) só tornam-se evidentes por meio de efeitos decorrentes da troca de mensagens entre atores. No sistema de animação deste trabalho, existe similaridade no que diz respeito ao fluxo de uma animação (existe troca de mensagens entre atores), porém uma entidade mais abstrata foi adicionada (seqüenciadores), que é quem de fato controla os atores, isto é, o ator é independente mas é subordinado aos seqüenciadores, desta forma, são os seqüenciadores que podem ser ativados, desativados e executados, deixando o ator dedicado às tarefas que diz respeito a ele e delegar a responsabilidade aos seqüenciadores sobre as regras que regem os aspectos da animação. Como em ASAS, um ator responde também a estímulos, porém, adicionalmente tal resposta pode ser tomada consultando o estado físico do ator, uma vez que a técnica de simulação dinâmica é disponibilizada pelo sistema.

O sistema proposto por Zeleznik [53] assemelha-se ao sistema deste trabalho no que se refere à visão orientada a objetos sobre a qual uma animação ganha progresso (troca de mensagens por meio de uma linguagem orientada a objetos). Porém, muito mais do que simples trocas de mensagens, é o gerenciamento/controle de tais mensagens. No sistema deste trabalho, o controle pode ser de natureza local ou global, como propõe Camargo [6]. O controle local está nas mãos do animador, onde especifica sua animação por meio de seqüenciadores, já o controle global é feito internamente por um componente da arquitetura do sistema (o controlador), gerenciando aqueles seqüenciadores de natureza local definidos pelo animador. Com isso, atores podem responder a mensagens e/ou estímulos gerados não somente por outros atores, mas gerados em decorrência de eventos capturados pelo controlador.

O sistema CLOCKWORKS [8] oferece ferramentas interativas na especificação de uma animação, porém, internamente, o que é especificado pelo animador é reescrito em uma linguagem de roteiros. Isto faz do CLOCKWORKS também um sistema procedimental. Embora no sistema deste trabalho não exista uma ferramenta para esta metodologia interativa (não houve esforços neste sentido), é dado ao animador a possibilidade de criar animações com uma visão de um *framework*, isto é, foi desenvolvida uma `API` de animação que pode ser utilizada e/ou estendida segundo os propósitos do animador.

A simulação dinâmica é uma das principais característica do sistema de animação deste trabalho. A simulação é utilizada para prover realismo na interação entre os componentes de uma cena animada. A simulação utilizada é baseada em dinâmica de corpos rígidos articulados, com detecção e reação de colisões, amplamente utilizada também em jogos digitais [3].

1.5 Organização do Texto

O texto é organizado em um único volume constituído de cinco capítulos. Além do presente capítulo, o texto possui ainda mais quatro, além de dois apêndices, resumidos a seguir.

[1] Herança, encapsulamento e polimorfismo.

Capítulo 2
Simulação Dinâmica de Corpos Rígidos
No Capítulo 2 apresentamos os conceitos básicos relativos à simulação dinâmica de corpos rígidos. Feito isto, fazemos uma introdução à arquitetura do motor de física PhysX SDK a ser integrado à máquina virtual de animação, descrevendo as principais classes de objetos que definem a `API` de simulação de física.

Capítulo 3
Especificando uma Animação
No Capítulo 3 apresentamos os conceitos básicos da linguagem de animação. Primeiramente, descrevemos os aspectos que a caracterizam como uma linguagem de propósito geral. Em seguida, descrevemos as extensões da linguagem utilizadas para a especificação de roteiros de animações, as quais são baseadas nos conceitos de seqüenciadores e eventos. Também neste capítulo descrevemos as interfaces das principais classes da `API` de animação desenvolvida no trabalho.

Capítulo 4
Executando Animações
No Capítulo 4 introduzimos os componentes do sistema de animação proposto que, a partir de arquivos compilados da especificação de uma cena, são responsáveis pela execução e visualização da animação. O principal destes componentes é a `MVL`. São descritos a arquitetura e os principais componentes da `MVL`, com ênfase à estrutura e a funcionalidade da memória de objetos lixo-coletável, à estratégia de coleta de lixo e à integração com o motor de física PhysX SDK.

Capítulo 5
Conclusão
No Capítulo 5 finalizamos com nossas conclusões, dificuldades encontradas, sugestões e contribuições pretendidas com o desenvolvimento do trabalho.

Apêndice A
No Apêndice A apresentamos a gramática da linguagem de animação para a especificação de animações.

Apêndice B
No Apêndice B apresentamos as instruções da máquina virtual de animação.

CAPÍTULO 2

Simulação Dinâmica de Corpos Rígidos

2.1 Introdução

A simulação em computador de algum fenômeno consiste na implementação de um modelo que permite predizer o comportamento e/ou visualizar a estrutura dos objetos envolvidos no fenômeno. No contexto de uma aplicação de simulação, o modelo computacional de um objeto pode ser dividido em *modelo geométrico*, *modelo matemático* e *modelo de análise*. Um modelo geométrico é uma representação das características que definem as formas e dimensões do objeto. O modelo matemático é usualmente dado em termos de equações diferenciais que descrevem aproximadamente o comportamento do objeto. Dependendo da complexidade, uma solução do modelo matemático, para os casos gerais de geometria e condições iniciais, somente pode ser obtida através do emprego de métodos numéricos, tais como o método dos elementos finitos (MEF). Nestes casos, o modelo de análise é baseado em uma malha de elementos (finitos) — resultante de uma discretização do volume do objeto — em cujos vértices são determinados os valores incógnitos que representam a solução do modelo matemático.

Em ciências e engenharia, a precisão do modelo é quase sempre mais importante que o tempo de simulação; neste trabalho, o tema *tempo real* não ganhou tanta atenção, embora o desempenho geral do sistema seja considerável. Focamos a atenção na utilização de modelos "simplificados" oferecidos por um motor de física em particular. Destes, o mais comumente utilizado são os modelos dinâmicos de corpos rígidos. Corpos rígidos podem ser classificados de várias maneiras. Um corpo rígido *discreto* é um sistema de $n > 0$ partículas no qual a distância entre duas partículas quaisquer não varia ao longo do tempo, não obstante a resultante de forças atuando no sistema. Um corpo rígido *contínuo* é um sólido indeformável com $n \to \infty$ partículas, delimitadas por uma superfície fechada que define o contorno de uma região do espaço de volume V.

Este capítulo faz uma introdução à mecânica de corpos rígidos contínuos necessária à compreensão das classes de objetos sobre os quais opera o motor de física, responsável pela simulação (de corpos rígidos). O objetivo é apresentar os principais conceitos, além de introduzir o PhysX SDK, o *framework* para simulação dinâmica de corpos rígidos utilizado neste trabalho. Na Seção 2.2 abordamos os conceitos básicos da mecânica de Newton em um sistema de partículas e restrições de movimento. Na Seção 2.3 tratamos dos conceitos

da dinâmica de corpos rígidos e o papel fundamental de um motor de física. Na Seção 2.4 apresentamos a arquitetura do motor de física PhysX SDK, as principais classes de objetos da API de física, além de uma introdução à sua utilização.

2.2 Conceitos Básicos da Mecânica Newtoniana

Seja[1] uma partícula de massa m localizada, em um instante de tempo t, em um ponto cuja posição no espaço é definida pelo vetor $\mathbf{r} = \mathbf{r}(t)$ (Figura 2.1).

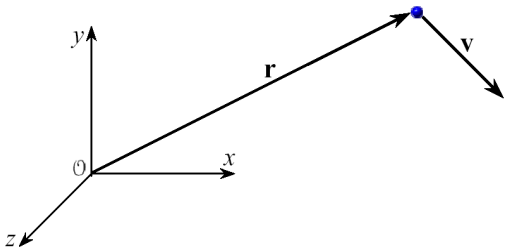


Figura 2.1: Posicao de uma partícula.

Será assumido que as coordenadas de $\mathbf{r}$ 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

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

e sua aceleração

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

O *momento linear* da partícula é definido como

$$\mathbf{p}(t) = m\mathbf{v}. \tag{2.3}$$

Seja $\mathbf{F} = \mathbf{F}(t)$ a resultante das forças (gravidade, atrito, etc.) que atuam sobre a partícula em um instante de tempo t (Figura 2.2).

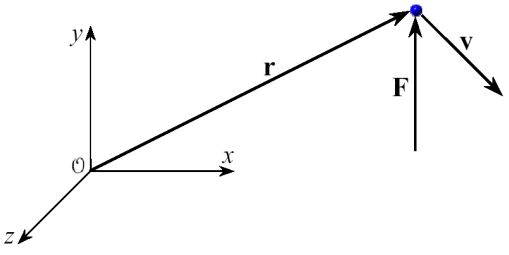


Figura 2.2: Resultante das forças que atuam numa partícula.

A *segunda lei de Newton* afirma que o movimento da partícula é governado pela equação diferencial

$$\mathbf{F}(t) = \dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = \frac{d}{dt}(m\mathbf{v}). \tag{2.4}$$

[1] O conteúdo desta seção é um resumo de [11].

Se a massa da partícula é constante:

$$\mathbf{F} = m\frac{d\mathbf{v}}{dt} = m\mathbf{a}. \tag{2.5}$$

Como conseqüência da segunda lei de Newton, se a resultante de forças que atuam na partícula é nula, então o momento linear da partícula é constante (teorema de conservação do momento linear).

O *momento angular* da partícula em relação à origem $\mathcal{O}$ do sistema global é definido como

$$\mathbf{L}(t) = \mathbf{r} \times \mathbf{p} = \mathbf{r} \times m\mathbf{v}. \tag{2.6}$$

Seja $\boldsymbol{\tau}$ o *momento* ou *torque* da resultante de forças $\mathbf{F}$, em relação à origem $\mathcal{O}$ do sistema global, aplicado à particula (Figura 2.3):

$$\boldsymbol{\tau}(t) = \mathbf{r} \times \mathbf{F}. \tag{2.7}$$

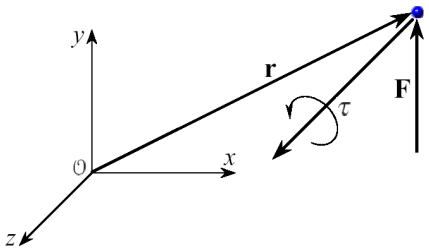


Figura 2.3: Torque sobre uma partícula.

Da mesma forma que, de acordo com a Equação (2.4), 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:

$$\dot{\mathbf{L}} = \frac{d\mathbf{L}}{dt} = \frac{d}{dt}(\mathbf{r} \times \mathbf{p}) = \mathbf{r} \times \frac{d\mathbf{p}}{dt} + \frac{d\mathbf{v}}{dt} \times \mathbf{p} = \mathbf{r} \times \mathbf{F} = \boldsymbol{\tau}. \tag{2.8}$$

Como conseqüência, se a resultante de forças que atuam na partícula é nula, o momento angular é constante (teorema da conservação do momento angular).

O *trabalho* realizado pela força $\mathbf{F}$ sobre a partícula quando esta se move ao longo de uma curva do ponto $\mathcal{P}_1$ ao ponto $\mathcal{P}_2$ é definido pela integral de linha

$$W_{12} = \int_{\mathbf{r}_1}^{\mathbf{r}_2} \mathbf{F} \cdot d\mathbf{r}, \tag{2.9}$$

onde $\mathbf{r}_1$ e $\mathbf{r}_2$ são as posições de $\mathcal{P}_1$ e $\mathcal{P}_2$, respectivamente. Como $d\mathbf{r} = \mathbf{v}dt$, a equação acima pode ser escrita, para massa constante, como

$$W_{12} = m\int_{t_1}^{t_2} \mathbf{F} \cdot \mathbf{v}dt = m\int_{t_1}^{t_2} \frac{d\mathbf{v}}{dt} \cdot \mathbf{v}dt = \frac{m}{2}\int_{t_1}^{t_2} \frac{d}{dt}(v^2)dt = \frac{m}{2}(v_2^2 - v_1^2). \tag{2.10}$$

A quantidade escalar $mv^2/2$ é chamada *energia cinética* da partícula e denotada por K. Portanto, o trabalho é igual à variação da energia cinética

$$W_{12} = K_2 - K_1. \tag{2.11}$$

Em um sistema *conservativo*, o campo de força é tal que W_{12} é independente do caminho entre os pontos $\mathcal{P}_1$ e $\mathcal{P}_2$. Uma condição necessária e suficiente para que isso ocorra é que $\mathbf{F}$ seja o gradiente de uma função escalar da posição

$$\mathbf{F} = -\nabla P(\mathbf{r}(t)) = -\left(\frac{\partial P}{\partial x}, \frac{\partial P}{\partial y}, \frac{\partial P}{\partial z}\right), \tag{2.12}$$

onde P é chamada *energia potencial*. Em um sistema conservativo

$$W_{12} = P_1 - P_2. \tag{2.13}$$

Combinando-se a equação acima com a Equação (2.11), obtém-se

$$K_1 + P_1 = K_2 + P_2, \tag{2.14}$$

ou seja: se as forças atuantes sobre uma partícula são conservativas, então a energia total do sistema, $E = K + P$, é constante (teorema da conservação da energia).

2.2.1 Mecânica de um Sistema de Partículas

Seja 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 é

$$\frac{d\mathbf{p}_i}{dt} = m_i \mathbf{v}_i = \mathbf{F}_i^e + \sum_j \mathbf{F}_{ji}, \tag{2.15}$$

onde $\mathbf{p}_i$, m_i e $\mathbf{v}_i$ são o momento linear, massa e 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

$$\frac{d^2}{dt^2} \sum_i m_i \mathbf{r}_i = \sum_i \mathbf{F}_i^e + \sum_{i,j} \mathbf{F}_{ji}. \tag{2.16}$$

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 $\overline{\mathbf{r}}$ igual à média das posições das partículas, ponderada em proporção a suas massas:

$$\overline{\mathbf{r}}(t) = \frac{\sum m_i \mathbf{r}_i}{\sum m_i} = \frac{\sum m_i \mathbf{r}_i}{M}, \tag{2.17}$$

onde M é a massa total. O vetor $\overline{\mathbf{r}}$ define um ponto $\mathcal{C}$ chamado *centro de massa* (Figura 2.4) do sistema.

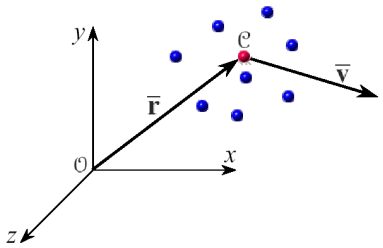


Figura 2.4: Centro de massa.

Com esta definição, a Equação (2.15) reduz-se a

$$M \frac{d^2 \overline{\mathbf{r}}}{dt^2} = \sum_i \mathbf{F}_i^e = \mathbf{F}, \tag{2.18}$$

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.

O momento linear total do sistema,

$$\mathbf{P}(t) = \sum_i m_i \frac{d\mathbf{r}_i}{dt} = M\frac{d\overline{\mathbf{r}}}{dt} = M\overline{\mathbf{v}}, \tag{2.19}$$

é a massa total vezes a velocidade $\overline{\mathbf{v}} = \dot{\overline{\mathbf{r}}}$ 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

$$\mathbf{L}(t) = \sum_i \mathbf{r}_i \times \mathbf{p}_i = \overline{\mathbf{r}} \times M\overline{\mathbf{v}} + \sum_i \mathbf{r}'_i \times \mathbf{p}'_i, \tag{2.20}$$

onde $\mathbf{r}'_i = \mathbf{r}_i - \overline{\mathbf{r}}$ 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 (Figura 2.5). Ou seja,

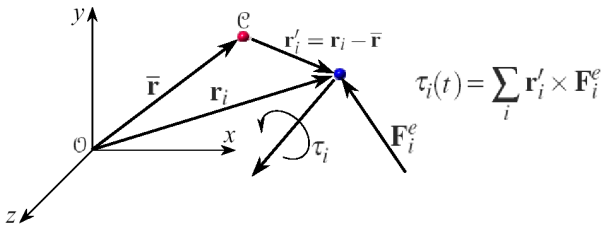


Figura 2.5: Torque externo em uma partícula em relação ao centro de massa.

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,

$$\dot{\mathbf{L}} = \boldsymbol{\tau} = \sum_i \mathbf{r}_i \times \mathbf{F}^e_i, \tag{2.21}$$

é igual ao torque da força externa total em relação a $\mathcal{O}$ (Figura 2.6). Como conseqüência, $\mathbf{L}$ é constante no tempo se o torque externo total é nulo.

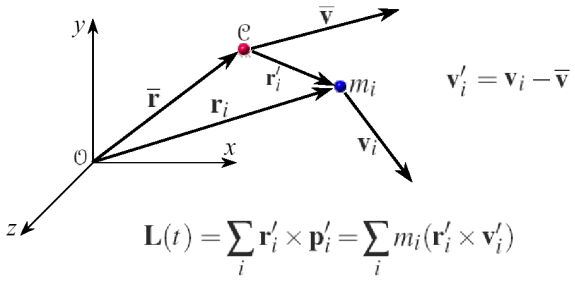


Figura 2.6: Momento angular total em relação ao centro de massa.

Da mesma forma que foi feito para uma partícula, pode-se demonstrar que, se as forças externas e internas forem derivadas de uma função escalar de energia potencial, então a energia total $E = K + P$ de um sistema de partículas é constante [16].

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* $\rho = \rho(\mathbf{r}(t))$, tal que uma partícula na posição $\mathbf{r}$ concentra uma massa $dm = \rho dV$. Em particular, a posição do centro de massa $\mathcal{C}$ fica definida como

$$\overline{\mathbf{r}}(t) = \frac{\int_V \mathbf{r}\, dm}{\int_V dm} = \frac{\int_V \rho\mathbf{r}\, dV}{M}, \tag{2.22}$$

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

2.2.2 Restrições de Movimento

A *configuração* de um sistema de n partículas em um instante de tempo t é o conjunto das posições $\mathbf{r}_i$, $1 \leq i \leq n$, 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 um número de configurações sejam válidas, isto é, nem toda configuração do sistema pode ser atingida, mesmo com tempo e energia suficientes para tal. Um exemplo de restrição é a imposição que o movimento de determinada partícula do sistema ocorra sobre uma determinada supefície. Outro exemplo é que não ocorra interpenetração no choque de dois ou mais sólidos indeformáveis.

São consideradas neste texto somente restrições de movimento que podem ser descritas por uma ou mais condições expressas em função das posições das partículas do sistema e do tempo (ou seja, independem das velocidades e/ou acelerações das partículas). Se uma condição é definida por uma equação algébrica da forma

$$h(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n, t) = 0 \tag{2.23}$$

é chamada *vínculo holonômico*. Uma restrição é holonômica se seus vínculos forem holonômicos. Um vínculo holonômico determina, em um instante t, uma superfície no espaço de configurações; o espaço de configurações válidas é a intersecção de todas as superfícies de vínculo.

Um sistema com n partículas possui $3n$ *graus de liberdade*, ou DOFs, uma vez que o movimento de partícula no espaço pode ser expresso como uma combinação de translações nas direções de cada um dos três eixos de um sistema de coordenadas Cartesianas. De modo geral, um vínculo holonômico elimina um grau de liberdade do sistema. Um exemplo de restrição holonômica é dada pelos vínculos

$$r_{ij} - c_{ij} = 0, \quad 1 \leq i, j \leq n, \tag{2.24}$$

onde r_{ij} é a distância entre as partículas i e j e c_{ij} é uma constante positiva. Como visto no início desta seção, um sistema de partículas sujeito a tal restrição é um corpo rígido discreto.

2.3 Dinâmica de Corpos Rígidos

Os vínculos da Equação (2.24) não são todos independentes (se fossem, estes eliminariam $n(n-1)/2$ DOFs, número que, para valores grandes de n, excede os $3n$ DOFs 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 DOFs, portanto, não pode ser maior que nove. Estes três pontos de referência não são, contudo, independentes, mas sujeitos aos vínculos

$$r_{12} = c_{12}, \quad r_{23} = c_{23} \quad \text{e} \quad r_{13} = c_{13}, \tag{2.25}$$

o que reduz o número de DOFs para seis.

Embora as equações de movimento tenham sido escritas até aqui em termos de coordenadas Cartesianas, as coordenadas dos graus de liberdade de um corpo rígido não serão 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 $\mathbf{u}(t) = (u_x, u_y, u_z)$, $\mathbf{v}(t) = (v_x, v_y, v_z)$ e $\mathbf{n}(t) = (n_x, n_y, n_z)$, com coordenadas tomadas em relação ao sistema global. Este sistema é chamado *sistema local* do corpo rígido, Figura 2.7. Três das coordenadas do corpo rígido em t serão as coordenadas globais da posição do centro de massa $\overline{\mathbf{r}}(t)$, Equação (2.22); as três restantes serão a orientação do sistema local em relação ao sistema global.

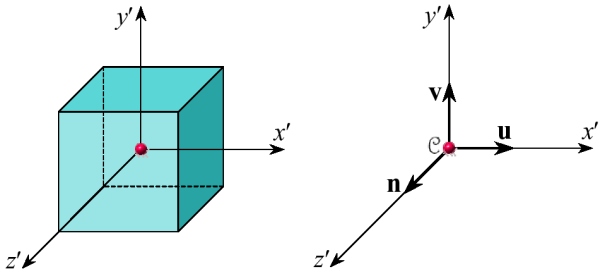


Figura 2.7: Sistema local de coordenadas de um corpo rígido.

Uma das maneiras de se representar a orientação do sistema local em um instante t é através de uma matriz de rotação de um ponto do corpo em torno de seu centro de massa:

$$\mathbf{R}(t) = \begin{bmatrix}\mathbf{u}(t) & \mathbf{v}(t) & \mathbf{n}(t)\end{bmatrix} = \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix}, \tag{2.26}$$

onde as coordenadas dos versores $\mathbf{u}$, $\mathbf{v}$ e $\mathbf{n}$ formam as colunas da matriz, Figura 2.8 (apesar de nove elementos, estes não são todos independentes e representam de fato as três coordenadas restantes de orientação do corpo).

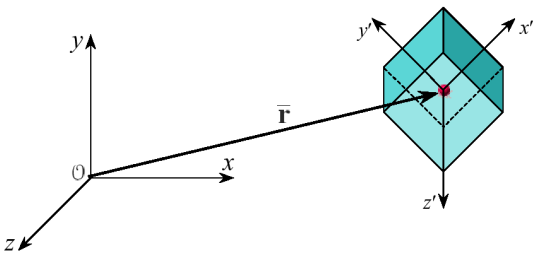


Figura 2.8: Orientação do sistema local em relação ao sistema global.

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

$$\mathbf{r}(t) = \overline{\mathbf{r}}(t) + \mathbf{R}(t)\mathbf{r}_0, \tag{2.27}$$

onde $\mathbf{r}_0$ é a posição de $\mathcal{P}$ em relação ao sistema local. A posição $\overline{\mathbf{r}}$ e a orientação $\mathbf{R}$, as quais definem totalmente a configuração (de qualquer partícula do) corpo em t, são chamadas *variáveis espaciais* do corpo rígido.

Durante uma simulação, não apenas as variáveis espaciais, mas também as velocidades dos corpos são mantidas e calculadas pelo motor de física. A velocidade de translação ou *velocidade linear* de (qualquer ponto de) um corpo rígido é a velocidade $\overline{\mathbf{v}}(t)$ 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 $\boldsymbol{\omega}(t)$, Figura 2.9. A direção de $\boldsymbol{\omega}(t)$ define a direção do eixo de rotação e $\|\boldsymbol{\omega}(t)\|$ o ângulo percorrido por um ponto em torno deste eixo no instante t.

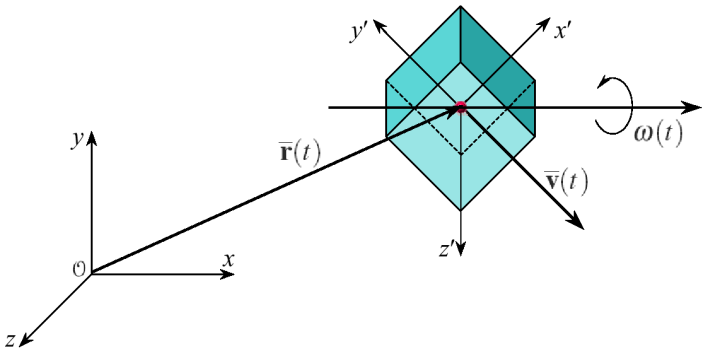


Figura 2.9: Velocidade linear e angular de um corpo rígido.

Pode-se estabelecer uma relação entre $\dot{\mathbf{R}}$ e a velocidade angular $\boldsymbol{\omega}$, do mesmo modo que há relação uma entre $\dot{\overline{\mathbf{r}}}$ e a velocidade linear $\overline{\mathbf{v}}$. 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 [3].

$$\dot{\mathbf{r}} = \boldsymbol{\omega} \times \mathbf{r}. \tag{2.28}$$

Agora, aplica-se a equação acima a cada uma das colunas de $\mathbf{R}$ na Equação (2.26), nominalmente os versores $\mathbf{u}$, $\mathbf{v}$ e $\mathbf{n}$, obtendo-se

$$\dot{\mathbf{R}} = \begin{bmatrix} \boldsymbol{\omega} \times \mathbf{u} & \boldsymbol{\omega} \times \mathbf{v} & \boldsymbol{\omega} \times \mathbf{n} \end{bmatrix}. \tag{2.29}$$

A expressão acima pode ser simplificada notando-se que, se $\mathbf{a}$ e $\mathbf{b}$ são vetores 3D, então o produto vetorial entre eles pode ser escrito como

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} a_y b_z - b_y c_z \\ a_z b_x - a_x b_y \\ a_x b_y - b_x a_y \end{bmatrix} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \widehat{\mathbf{a}}\mathbf{b}, \tag{2.30}$$

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

$$\widehat{\mathbf{a}} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}. \tag{2.31}$$

A relação procurada entre $\dot{\mathbf{R}}$ e $\boldsymbol{\omega}$ é obtida escrevendo-se os produtos vetoriais da Equação (2.29) como a multiplicação da matriz $\widehat{\boldsymbol{\omega}}$ pelos versores $\mathbf{u}$, $\mathbf{v}$ e $\mathbf{n}$, resultando

$$\dot{\mathbf{R}}(t) = \widehat{\boldsymbol{\omega}}(t)\,\mathbf{R}(t). \tag{2.32}$$

A partir desta relação, pode-se derivar a Equação (2.27) e escrever a velocidade em coordenadas globais de um ponto $\mathcal{P}$ de um corpo rígido em um instante t (Figura 2.10) como sendo

$$\dot{\mathbf{r}}(t) = \overline{\mathbf{v}}(t) + \widehat{\boldsymbol{\omega}}(t)\mathbf{R}(t)\mathbf{r}_0 + = \overline{\mathbf{v}}(t) + \boldsymbol{\omega}(t) \times (\mathbf{r}(t) - \overline{\mathbf{r}}(t)). \tag{2.33}$$

O conjunto das variáveis espaciais e das velocidades linear e angular define o *estado* de um corpo rígido. É mais conveniente, contudo, expressar as velocidades em termos dos momentos linear e angular. A Equação (2.19) estabelece que

$$\overline{\mathbf{v}}(t) = \frac{\mathbf{P}(t)}{M}. \tag{2.34}$$

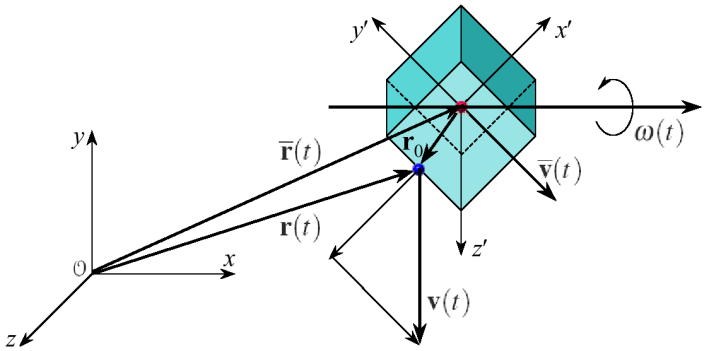


Figura 2.10: Velocidade de uma partícula de um corpo rígido.

Da mesma forma, pode-se relacionar o momento angular em relação ao centro de massa $\mathcal{C}$ e a velocidade angular através da seguinte transformação linear:

$$\mathbf{L}(t) = \mathbf{I}(t)\,\boldsymbol{\omega}(t), \tag{2.35}$$

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

$$I_{ij}(t) = \int_V \rho(\mathbf{r}')(r'^2\delta_{ij} - x'_i x'_j)\,dV, \quad i,j = 1,2,3, \tag{2.36}$$

onde $\mathbf{r}' = \mathbf{r}(t) - \overline{\mathbf{r}}(t) = (x'_1, x'_2, x'_3)$ é o vetor do centro de massa à posição $\mathbf{r}$ de um ponto do corpo, em coordenadas globais, e δ_{ij} é o *delta de Kronecker*, definido como

$$\delta_{ij} = \begin{cases} 0, \text{se } i \neq j, \\ 1, \text{se } i = j. \end{cases} \tag{2.37}$$

Se o tensor de inércia de um corpo rígido tivesse que ser calculado através da Equação (2.36) em cada instante t em que $\mathbf{R}(t)$ variasse, o tempo de processamento para fazê-lo, durante a simulação, poderia ser custoso. Ao invés disto, o tensor de inércia é calculado, para qualquer orientação $\mathbf{R}(t)$, em termos de integrais computadas em relação ao sistema local antes do corpo rígido entrar em cena e, portanto, constantes ao longo da simulação. Seja $\mathbf{I}_0$ este tensor de inércia. Pode-se mostrar que [3].

$$\mathbf{I}(t) = \mathbf{R}(t)\,\mathbf{I}_0\,\mathbf{R}(t)^T. \tag{2.38}$$

Finalmente, o estado de um corpo rígido em t pode ser definido como

$$\mathbf{X}(t) = \begin{bmatrix} \overline{\mathbf{r}}(t) \\ \mathbf{R}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{bmatrix}. \tag{2.39}$$

A massa M e o tensor de inércia local $\mathbf{I}_0$ (determinado antes do corpo entrar em cena) são constantes. Em resumo, em qualquer tempo t, as grandezas $\overline{\mathbf{v}}(t)$, $\mathbf{I}(t)$ e $\boldsymbol{\omega}(t)$ podem ser calculadas por

$$\overline{\mathbf{v}}(t) = \frac{\mathbf{P}(t)}{M}, \quad \mathbf{I}(t) = \mathbf{R}(t)\mathbf{I}_0\mathbf{R}(t)^T \quad \text{e} \quad \boldsymbol{\omega}(t) = \mathbf{I}(t)^{-1}\mathbf{L}(t). \tag{2.40}$$

O papel fundamental de um motor de física é, durante a simulação de uma cena com vários corpos rígidos, conhecidos os estados $\mathbf{X}_i(t)$ de cada corpo no tempo t, determinar os estados $\mathbf{X}_i(t + \Delta t)$ no tempo $t + \Delta t$, onde Δt é um *passo de tempo*. Para um sistema sem restrições de movimento, esta determinação pode ser efetuada por qualquer método numérico de resolução de equações diferenciais de primeira ordem, como o método de Runge-Kutta de quarta ordem. O componente do motor responsável por isto é chamado ODE *solver*. Basicamente, um ODE *solver* 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

$$\frac{d}{dt}\mathbf{X}(t) = \begin{bmatrix} \overline{\mathbf{v}}(t) \\ \widehat{\boldsymbol{\omega}}(t)\mathbf{R}(t) \\ \mathbf{F}(t) \\ \boldsymbol{\tau}(t) \end{bmatrix} \tag{2.41}$$

do estado de cada corpo, e (3) os valores de t e Δt, e computa o estado $\mathbf{X}_i(t + \Delta t)$ de cada corpo rígido. Note que todas as grandezas na Equação (2.41) 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 determinados pela aplicação.

Restrições de Contato

Em simulação dinâmica de corpos rígidos fundamentalmente são tratados dois tipos de restrições: (1) aquelas impostas por *junções* entre (normalmente dois) corpos, e (2) resultante do *contato* entre corpos. Uma junção entre dois corpos força que o movimento de um seja relativo ao do outro de alguma maneira que depende do tipo da junção. Alguns exemplos são ilustrados na Figura 2.11.

Figura 2.11: Exemplo de junções: esférica, de revolução e cilíndrica.

Uma junção esférica força que dois pontos sobre dois corpos diferentes sejam coincidentes, removendo três DOFs de cada corpo. Uma junção de revolução pode ser usada para representar uma dobradiça entre dois corpos: cinco DOFs 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 DOFs de cada corpo. Se as translações e rotações permitidas por estes tipos de junções não são limitadas, então as restrições correspondentes podem ser definidas por vínculos holonômicos.

Restrições de contato, por sua vez, podem envolver condições expressas também por inequações, ou seja, não holonômicas, o que implica que métodos distintos daqueles empregados no tratamento de restrições de junções devem ser considerados. Embora seja possível um tratamento unificado de ambos os tipos de restrições, abordamos somente as restrições decorrentes do contato entre corpos.

Para corpos rígidos cuja geometria é definida por poliedros, são considerados dois casos não degenerados de contato. Um contato *vertice/face* ocorre quando, em um instante t_c, um

vértice v_A de um pliedro A está em contato com uma fade f_B de um poliedro B em um ponto $P_c = v_A$. A direção normal ao ponto de contato $\mathbf{N}(t_c)$ é igual à normal à face f_B, suposta apontar para fora do poliedro B. Um contato *aresta/aresta* ocorre quando duas arestas não colineares e_A e e_B de dois poliedros A e B se interceptam em um ponto P_c. Neste caso, a direção normal ao ponto de contato é definida como

$$\mathbf{N}(t_c) = e_A(t_c) \times e_B(t_c), \tag{2.42}$$

onde e_A e e_B são versores nas direções de e_A e e_B no instante t_c, respectivamente. Assume-se que o motor de física possa contar com um componente responsável pela *detecção de colisões*, o qual, num determinado instante t_c, em que se pressupõe não haver interpenetração entre quaisquer corpos, seja capaz de determinar quantos e quais são todos os pontos de contato entre os corpos da simulação.

2.4 Arquitetura do Motor de Física PhysX

O PhysX SDK é um componente de software capaz de realizar cálculos para simular efeitos de gravidade, forças e torques em cenas cujos atores podem ser considerados modelos de corpos rígidos articulados. PhysX é visto pelo desenvolvedor como um conjunto de bibliotecas de ligação estática e dinâmica e um conjunto de arquivos de cabeçalho C++ nos quais são declaradas as interfaces das classes do SDK. Estas classes são implementadas em dois pacotes principais: Foundation SDK e Physics SDK. No primeiro encontram-se as classes de objetos utilizadas pelo segundo, tais como as que representam vetores 3D, matrizes de transformação geométrica, quaternions, além de várias definições de tipos e funções matemáticas. No pacote Physics SDK são implementadas internamente as classes concretas que representam os objetos utilizados na simulação dinâmica de corpos rígidos. As interfaces com as funcionalidades de tais classes são definidas pelas classes abstratas ilustradas no diagrama `UML` da Figura 2.12 e comentadas a seguir.

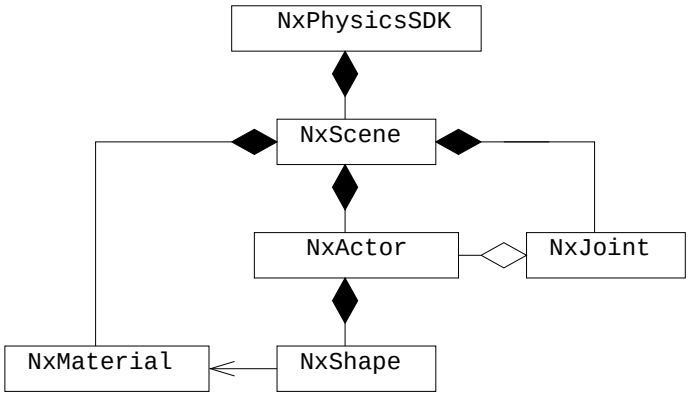


Figura 2.12: Principais classes do PhysX SDK.

2.4.1 Instanciação do Motor e Criação de uma Cena

O ponto de partida para a utilização do motor é a criação de um objeto da classe (interna derivada da classe abstrata) `NxPhysicsSDK`, a qual representa o motor propriamente dito. A classe define métodos para criação e destruição de cenas e para ajuste de vários parâmetros usados em uma simulação. Sua interface é apresentada na Tabela 2.1.

Classe `NxPhysics`	
Métodos públicos	
`virtual void release()=0`	Destrói o objeto.
`virtual NxScene* createScene(const NxSceneDesc&)=0`	Cria uma cena a partir de um descritor.
`virtual void releaseScene(NxScene&)=0`	Destrói uma cena.
`virtual NxScene* getScene(NxU32 index)=0`	Retorna a index-ésima cena da simulação.
`virtual NxU32 getNbScenes()const=0`	Retorna o número de cenas criadas.
`virtual int addMaterial(const NxMaterial&)=0`	Adiciona um novo material.
`virtual NxMaterial* getMaterialFromIndex(int)=0`	Retorna o material num dado índice.
`virtual NxU32 getNbMaterials()=0`	Retorna o quantidade de materiais.
`virtual bool setParameter(NxParam p, NxReal v)=0`	Ajusta o parâmetro `p` para o valor `v`.
`virtual NxReal getParameter(NxParam p)const=0`	Retorna o valor do parâmetro `p`.

Tabela 2.1: Classe `NxPhysics`.

Instanciado o motor, pode-se criar uma ou mais cenas. Uma cena no PhysX SDK é um objeto cuja classe é derivada da classe abstrata `NxScene`. Uma cena mantém coleções de *materiais*, *atores* e *junções*. É possível criar várias cenas e simulá-las concorrente ou paralelamente. As simulações em duas ou mais cenas são completamente disjuntas, ou seja, objetos em uma determinada cena não influenciam sobre objetos de outras cenas. Uma cena não possui nenhuma restrição espacial em relação ao posicionamento de seus objetos e provê funcionalidades relacionadas à física, tais como campo gravitacional uniforme atuando sobre seus objetos, detecção de colisões, entre outras. A classe `NxScene` define a interface (Tabela 2.2) de métodos de gerenciamento das coleções de objetos mantidas em uma cena (atores, junções, etc.), de ajuste da gravidade e, o mais importante, para disparar a simulação da cena em um determinado instante de tempo.

Classe `NxScene`	
Métodos públicos	
`virtual void setGravity(const NxVec3&)=0`	Ajusta a gravidade na cena.
`virtual NxActor* createActor(NxActorDescBase&)=0`	Cria um ator.
`virtual void releaseActor(NxActor&)=0`	Destrói um ator.
`virtual NxJoint* createJoint(const NxJointDesc&)=0`	Cria uma junção.
`virtual void releaseJoint(NxJoint&)=0`	Destrói uma junção.
`virtual NxU32 getNbActors()const=0`	Retorna o número de atores.
`virtual NxActor** getActors()=0`	Retorna os atores da cena.
`virtual NxU32 getNbJoints()const=0`	Retorna o numero de junções.
`virtual void resetJointIterator()=0`	Prepara um iterador de junções.
`virtual NxJoint* getNextJoint()=0`	Retorna a próxima junção numa iteração.
`virtual void simulate(NxReal)=0`	Dá inicio à simulação.
`virtual void flushStream()=0`	Prepara os resultados da simulação.
`virtual bool fetchResults(NxSimulationStatus)=0`	Atualiza os resultados da simulação.

Tabela 2.2: Classe `NxScene`.

Para criação de uma cena utiliza-se um descritor de cena. No PhysX SDK, um descritor de objeto de um determinado tipo é um objeto temporário passado como argumento para um método de criação de objetos daquele tipo. Um destes descritores de objeto é o descritor de cena, objeto da classe `NxSceneDesc`. O trecho de código a seguir ilustra a instanciação do motor e a criação de uma cena em uma aplicação Windows. O descritor `sceneDesc` define que a cena terá gravidade de $9.8m/s^2$ e detecção de colisão ativada. `NX_MIN_SEPARATION_FOR_PENALTY` é um parâmetro que indica ao detector de colisões a distância mínima em que dois corpos são considerados em contato. `NX_BROADPHASE_COHERENT` representa um dos possíveis algoritmos de detecção de colisões do SDK. Um ponteiro para a cena a ser criada é obtido enviando-se a mensagem `createScene(sceneDesc)` à instância do motor endereçada por `gPhysicsSDK`:

```
#include "NxPhysics.h"
NxPhysicsSDK* gPhysicsSDK = NULL;
NxScene* gScene = NULL;
void InitPhysics()//Instancia o motor e cria uma cena
{
  //Instancia o Physics SDK.
  gPhysicsSDK = NxCreatePhysicsSDK(NX_PHYSICS_SDK_VERSION);
  if(gPhysicsSDK == NULL)
    return;
  gPhysicsSDK->setParameter(NX_MIN_SEPARATION_FOR_PENALTY, -0.1);
  //Cria a cena.
  NxSceneDesc sceneDesc;
  sceneDesc.gravity = NxVec3(0,-9.8,0);
  sceneDesc.broadPhase = NX_BROADPHASE_COHERENT;
  sceneDesc.collisionDetection = true;
  gScene = gPhysicsSDK->createScene(sceneDesc);
}
```

2.4.2 Criação de Atores

Atores são os corpos rígidos protagonistas de uma simulação. No PhysX, atores podem ser objetos estáticos ou corpos rígidos dinâmicos. Atores estáticos são objetos fixos em relação a um sistema de coordenadas de referência. Atores dinâmicos, por sua vez, têm as propriedades de corpo rígido vistas anteriormente (velocidade, momento, etc.) alteradas ao longo do tempo como resultado de forças, torques e contatos. Um ator é um objeto de uma classe derivada da classe abstrata `NxActor`. Entre outros, a classe define métodos de ajuste da massa, posição, orientação, velocidades e momentos lineares e angular, e para aplicação de forças e torques em um ator. A Tabela 2.3 descreve os principais métodos da interface de `NxActor`.

A geometria de um ator é definida por uma ou mais *formas*. Uma forma é um objeto de uma classe derivada da classe abstrada `NxShape`. Os principais tipos de formas disponíveis no SDK são: bloco, esfera, malha de triângulos (para modelos poligonais), cápsula e plano. Como vimos, um corpo rígido pode ser perfeitamente representado por uma único ponto, localizado em seu centro de massa, e um tensor de inércia. Esta é a representação utilizada pelo PhysX. Formas servem basicamente para dois propósitos:

- *Cálculo do centro de massa e inêrcia de um ator.* A toda forma podemos atribuir uma massa ou densidade em função da qual o PhysX pode calcular automaticamente seu centro de massa $\bar{\mathbf{r}}$ e tensor de inércia $\mathbf{I}_0$, embora estes também podem ser explicitamente definidos pela aplicação através de métodos próprios declarados na classe `NxShape` (Tabela 2.4). O centro de massa e inércia de um ator podem ser computados a partir das

Classe `NxActor`	
Métodos públicos	
`virtual NxShape* createShape(const NxMat33&)=0`	Cria uma forma para o ator.
`virtual void releaseShape(NxShape&)=0`	Destrói uma forma.
`virtual NxU32 getNbShapes()const=0`	Retorna o número de formas do ator.
`virtual NxShape** getShapes()const=0`	Retorna as formas do ator.
`virtual bool isDynamic()const=0`	Retorna se o ator é dinâmico ou não.
`virtual void setMass(NxReal)=0`	Ajusta a massa do ator.
`virtual NxReal getMass()const=0`	Retorna a massa do ator.
`virtual void setLinearVelocity(const NxVec3&)=0`	Ajusta a velocidade linear.
`virtual void setAngularVelocity(const NxVec3&)=0`	Ajusta a velocidade angular.
`virtual void setMaxAngularVelocity(NxReal)=0`	Ajusta a máxima velocidade angular.
`virtual void setLinearMomentum(const NxVec3&)=0`	Ajusta o momento linear.
`virtual void setAngularMomentum(const NxVec3&)const=0`	Ajusta o momento angular.
`virtual void setLinearDamping(NxReal)=0`	Ajusta o amortecimento linear.
`virtual void addForce(const NxVec3&)=0`	Aplica uma força no centro de massa.
`virtual void addTorque(const NxVec3&, const NxVec3&)=0`	Aplica um torque sobre um eixo.
`virtual NxReal computerKineticEnergy()const=0`	Calcula a energia cinética do ator.
`virtual void setCMassLPose(const NxMat34&)=0`	Ajusta a pose do centro de massa.
`virtual void setGlobalPose(const NxMat34&)=0`	Ajusta a pose do ator.

Tabela 2.3: Classe `NxActor`.

contribuições da massa e inércia das formas que o compõe (ou definidas pela aplicação através de métodos da classe `NxActor`).

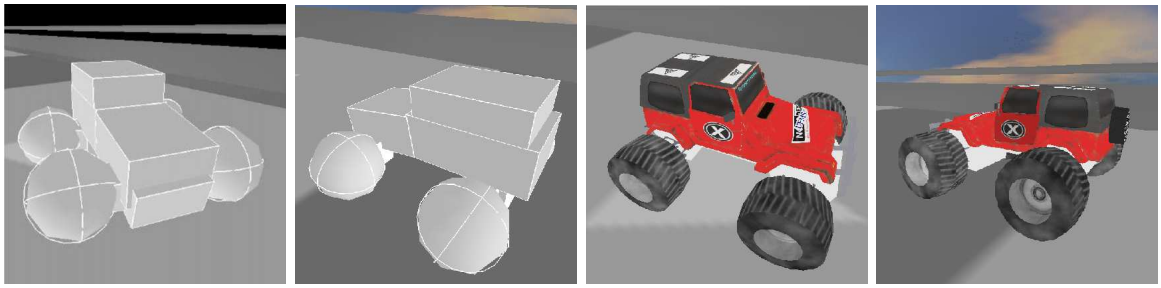
- *Detecção de colisões.* As formas definem, do ponto de vista dos métodos de detecção de colisões do PhysX, a geometria a partir da qual serão determinados os pontos de contato entre os atores de uma cena. De modo geral, as formas *não* definem, do ponto de vista do motor gráfico (renderizador), a aparência de um ator, devendo para isto ser empregado um modelo geométrico adequado. (O PhysX é independente do motor gráfico utilizado e, prometem seus desenvolvedores, pode ser acoplado a qualquer um). O mecanismo de detecção de colisão do PhysX assegura que uma dada forma geométrica de um ator não é capaz de atravessar outras formas. Como exemplo, a Figura 2.13 mostra, à esquerda ((a) e (b)), formas mais simples do SDK usadas na modelagem de um "carrinho". O objeto é definido por onze atores (um corpo, quatro rodas, quatro barras de suspensão e duas esferas cada (objetos do tipo `NxSphereShape` e o corpo, representado por seis blocos (objetos do tipo `NxBoxShape`)). Em comparação às malhas de triângulos, estas formas mais simples permitem uma detecção de colisões mais eficiente. À direita ((c) e (d)), uma imagem do modelo gráfico do carrinho.

2.4.3 Criação de Junções

Sem junções ou formas os atores fazem pouco mais que flutuar pelo espaço. As junções fornecem uma maneira persistente de conectar dois atores. Requerem que ambos atores movam-se sempre com um movimento relativo em relação ao outro. A maneira específica que limita o movimento é determinada pelo tipo da junção. Atualmente o PhysX SDK pode oferecer onze tipos de junções para simulações [1], que correspondem às classes derivadas da classe abstrata `NxJoint`, cuja interface simplificada é exibida na Tabela 2.5.

Classe `NxShape`	
Métodos públicos	
`virtual NxActor& getActor()const=0`	Retorna o ator associado à forma.
`virtual void setLocalPose(const NxMat34&)=0`	Ajusta a pose do ator.
`virtual void setLocalPosition(const NxVec3&)=0`	Ajusta a posição local do ator.
`virtual void setLocalOrientation(const NxMat33&)=0`	Ajusta a orientação local.
`virtual NxMat34 getLocalPose()const=0`	Retorna a pose local.
`virtual NxVec3 getLocalPosition()const=0`	Retorna a posição local.
`virtual NxMat33 getLocalOrientation()const=0`	Retorna a orientação local.
`virtual void setGlobalPose(const NxMat34&)=0`	Ajusta a pose global.
`virtual void setGlobalPosition(const NxVec3&)=0`	Ajusta a posição global.
`virtual void setGlobalOrientation(const NxMat33&)=0`	Ajusta a orientação global.
`virtual NxMat34 getGlobalPose()const=0`	Retorna a pose global.
`virtual NxVec3 getGlobalPosition()const=0`	Retorna a posição global.
`virtual NxMat33 getGlobalOrientation()const=0`	Retorna a orientação global.
`virtual void setMaterial(NxMaterialIndex)=0`	Ajusta o material que é constituído.
`virtual NxMaterialIndex getMaterial()const=0`	Retorna o material.
`virtual NxShapeType getType()const=0`	Retorna o tipo da forma.
`virtual NxPlaneShape* isPlane()=0`	Retorna se a forma é um plano.
`virtual NxSphereShape* isSphere()=0`	Retorna se a forma é uma esfera.
`virtual NxBoxShape* isBox()=0`	Retorna se a forma é uma caixa.
`virtual NxCapsuleShape* isCapsule()=0`	Retorna se a forma é uma cápsula.

Tabela 2.4: Classe `NxShape`.



(a) (b) (c) (d)

Figura 2.13: Formas geométrica de um ator: colisão e gráfico.

Junções são também chamadas de restrições por restrigirem o movimento de um corpo. O PhysX SDK provê somente restrições aos pares, ou seja, cada junção é exatamente entre dois atores, entretanto, um destes atores pode ser um "ator abstrato", o qual representa uma referência global imóvel no espaço. Para especificar uma junção entre um ator qualquer e o "ator abstrato", passa-se o valor `NULL` (que representa o "ator abstrato") como um dos ponteiros para atores quando da criação de uma junção. O ator que não é definido como `NULL` deve ser dinâmico (obviamente não faz sentido conectar dois atores estáticos por uma junção).

Junções são criadas enviando-se a mensagem `createJoint(jointDesc)` à instancia da cena. Esta mensagem toma como parâmetro um descritor de junção de um determinado tipo, que contém todas as informações necessárias para a criação e uma junção daquele tipo. A seguir um exemplo de criação de uma junção esférica:

Classe `NxJoint`	
Métodos públicos	
`virtual NxScene& getScene()=0`	Retorna a cena da junção.
`virtual void getActors(NxActor** a1, NxActor** a2)=0`	Retorna os atores envolvidos.
`virtual void setGlobalAnchor(const NxVec3& vec)=0`	Ajusta o ponto de conexão.
`virtual void setGlobalAxis(const NxVec3& vec)=0`	Ajusta a direção do eixo primário.
`virtual NxVec3 getGlobalAnchor()const=0`	Retorna o ponto de conexão.
`virtual NxVec3 getGlobalAxis()const=0`	Retorna o eixo primário da junção.
`virtual NxJointType getType()const=0`	Retorna o tipo específico da junção.
`virtual void setBreakable(NxReal mF, NxReal mT)=0`	Ajusta a força/torque máxima(o) capaz de quebrar a junção.
`virtual void getBreakAble(NxReal& mF, NxReal& mT)const=0`	Retorna a força/torque máxima(o) de uma junção quebrável.

Tabela 2.5: Classe `NxJoint`.

```
NxSphericalJoint* CreateSphericalJoint(NxActor* a0, NxActor* a1,
     const NxVec3& globalAnchor, const NxVec3& globalAxis)
{
   //Um descritor para uma junção do tipo esférica.
   NxSphericalJointDesc sphericalDesc;
   sphericalDesc.actor[0] = a0;
   sphericalDesc.actor[1] = a1;

   /*Ajusta o ponto onde os dois atores são conectados,
   especificado em coordenadas globais.*/
   sphericalDesc.setGlobalAnchor(globalAnchor);

   /*Ajusta a direção do eixo primário da junção,
   especificada também em coordenadas globais.*/
   sphericalDesc.setGlobalAxis(globalAxis);

   NxSphericalJoint* sphereJoint;
   sphereJoint = (NxSphericalJoint*)gScene->createJoint(sphericalDesc);
   return sphereJoint;
}
```

2.4.4 Materiais

As propriedades de superfícies e de colisão de objetos de uma cena são definidas pelo material do qual são feitos, ou seja, a forma pela qual os objetos deslizam, saltam e rolam sobre os outros objetos é influenciada pela constituição do seu material. No PhysX, um material é abstraído pela classe `NxMaterial`, brevemente descrita na Tabela 2.6.

O *coeficiente de atrito estático* determina a maior força de atrito quando da iminência do movimento, enquanto que o *coeficiente de atrito dinâmico* determina a força de atrito após o movimento. O *coeficiente de restituição* mede a elasticidade de um impacto. Outra propriedade de um material é a direção de anisotropia, isto é, a direção do movimento de um corpo em que suas propriedades materiais comportam-se de maneiras distintas.

2.4.5 Execução da Simulação

O PhysX efetua os cálculos de simulação de uma cena em uma *thread* criada especificamente para tal. Um mecanismo de leitura e escrita protegidas é utilizado para prevenir que o

Classe `NxMaterial`	
Atributos públicos	
`NxReal dynamicFriction`	Coeficiente de atrito dinânico.
`NxReal staticFriction`	Coeficiênte de atrito estático.
`NxReal restitution`	Coeficiente de restituição.
`NxVec3 dirOfAnisotropy`	Direção anisotrópica.
Métodos públicos	
`void` `setToDefault()`	Ajusta os atributos para valores padrão.
`bool` `isValid()` **`const`**	Retorna **`true`** se os atributos possuem um valor válido.

Tabela 2.6: Classe `NxMaterial`.

programador altere os dados que definem os estados, ou seja, as informações sobre a posição e velocidade, dos atores envolvidos em um passo da simulação, enquanto esta estiver em progresso. Simulações são feitas em um passo de cada vez, tipicamente usando-se um intervalo fixo entre 1/100 e 1/50 segundos, ou seja, a cada passo de tempo, os dados resultantes dos cálculos da dinâmica de corpos rígidos são atualizados pela *thread* de simulação e, através do mecanismo de leitura e escrita, entregues para processamento à *thread* da aplicação que está sob controle do programador. Os estados dos atores são atualizados invocando-se a seguinte seqüência de operações: (1) dá-se início à simulação da cena em um passo de tempo; (2) assegura-se que todos os dados estejam prontos para serem enviados para a *thread* de simulação; e (3) verifica-se se a simulação já foi realizada; em caso afirmativo, atualizam-se os dados dos estados dos atores que foram modificados pela *thread* de simulação, preparando-os para o próximo passo de tempo.

Esta seqüência de operações é implementada a seguir na função `RunPhysics()`. A função começa invocando `UpdateTime()` a fim de obter o tempo transcorrido desde a execução da ultima simulação. Em seguida `simulate()` é invocada para executar a simulação para o passo de tempo e `flushStream()` e `fetchResults()` para terminar a simulação. O método `flushStream()` prepara todos os comandos da cena eventualmente protegidos de modo que sejam executados e que prossigam até a sua conclusão. O método `fetchResults()` é bloqueante, e o argumento `NX_RIGID_BODY_FINISHED` indica que este não retornará até que a *thread* de simulação conclua todos os cálculos que envolvam a dinâmica de corpos rígidos.

```
void RunPhysics()
{
   //Atualiza o passo de tempo.
   NxReal deltaTime = UpdateTime();

   //Executa colisão e dinâmica para o lapso de tempo desde a última
   // simulação.
   gScene->simulate(deltaTime);
   gScene->flushStream();
   gScene->fetchResults(NX_RIGID_BODY_FINISHED, true);
}
```

2.5 Considerações Finais

Um corpo rígido *discreto* é um sistema de $n > 0$ 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.

O PhysX SDK é um motor de física capaz de simular efeitos de gravidade, forças e torques

aplicados a modelos de corpos rígidos articulados. A API PhysX SDK é implementada em dois pacotes: FoundationSDK e PhysicsSDK. O pacote PhysicsSDK baseia-se no FoundationSDK.

Uma cena PhysX SDK é uma coleção de atores e junções. Um ator representa um protagonista de uma simulação. No PhysX, atores podem ser estáticos ou dinâmicos. Atores estáticos são objetos fixos em relação ao sistema de referência global e não sofrem influência de qualquer efeito dinâmico, diferentemente dos atores dinâmicos. Uma das propriedades importante de um ator é sua pose, que define sua posição e orientação em relação a um sistema de referência global.

Um ator é definido por formas. Uma forma é uma objeto que representa parte da geometria de um ator. Formas servem para o cálculo do centro de massa do ator e para a determinação dos pontos de contatos quando da colisão entre atores. De forma geral, as formas não definem, do ponto de vista do motor gráfico, a aparência do ator, devendo para isto ser empregado um modelo geométrico adequado. Formas podem ser primitivas (caixa, esfera, cápsula e plano) ou uma malha de triângulos.

Um material é um objeto que define as propriedades materiais de cada uma das formas que compõe um ator. Tais propriedades incluem as de superfície e de detecção de colisão.

Uma junção representa um conjunto de possibilidades de movimento ou graus de liberdade. No PhysX, junções fornecem uma forma persistente de conectar dois atores. A API PhysX oferece uma variedade de junções, tais como, esférica, prismática, cilíndrica, de revolução dentre outras.

CAPÍTULO 3

Especificando Animações

3.1 Introdução

Na literatura pode-se encontrar uma variedade de linguagens para descrever animações [12, 14, 24]. Segundo Foley [13], as linguagens podem ser divididas em três categorias:

- **Notação de listas lineares**. Na notação de listas lineares, cada evento na animação é descrito por um número inicial e final de quadro e uma ação a ser tomada. As ações tipicamente requerem parâmetros, tais como o exemplo abaixo:

  ```
  42, 53, B ROTATE "PALM", 1, 30
  ```

  que significa: "entre os quadros 42 e 53, rotacione o objeto chamado PALM em torno do eixo 1 de 30 graus, determinando a quantidade de rotação em cada quadro de acordo com a tabela B". Outras linguagens baseadas na notação de listas lineares incorporam características de linguagem de programação de alto nível, com variáveis e fluxo de controle [18, 42].

- **Linguagens de propósito geral**. As animações podem ser descritas através de extensões de linguagens de propósito geral. Dessa forma, estruturas de dados e fluxo de controle podem ser utilizadas em rotinas para gerar animações. As animações, portanto, são efeitos colaterais de simulações geradas através da linguagem. Estas linguagens possuem grande potencial, embora exijam conhecimento de programação por parte do animador. O desenvolvimento das linguagens de programação tem levado a novos conceitos fundamentais no controle do movimento e de eventos temporais [44].

- **Linguagens gráfica**. Um dos problemas com linguagens de propósito geral é a dificuldade encontrada pelo animador em visualizar a animação observando apenas o texto. Linguagens gráficas descrevem a animação de uma forma mais visual. Tais linguagens substituem o paradigma textual por um paradigma visual, ou seja, ao invés de escrever descrições das ações, o animador fornece um esboço da ação.

Em adição à classificação de Foley existem as linguagens de *script*. Esta categoria de linguagem constitui um recurso considerável para dirigir uma animação. Características como gerenciamento automático de memória, coleta de lixo, tipagem dinâmica e apoio à construção dinâmica de estruturas de dados, entre outras, fazem das linguagens de *scripts*

uma ferramenta que pode ser utilizada além do escopo de animação por computador. Um exemplo é Lua [20], a qual combina uma sintaxe procedimental simples com construções de descrições de dados baseadas em *arrays* associativos com semântica extensível. Lua tem sido atualmente usada no desenvolvimento de jogos.

Neste capítulo apresentamos as características da linguagem utilizada para especificação de animações, além da forma de compilação da linguagem. A linguagem de animação LA é derivada de uma linguagem de propósito geral, híbrida como C++, similar em alguns aspectos a Java, com produções para tratamento de exceções, sobrecarga de operadores e definição de propriedades. A esta linguagem de propósito geral, chamada L, foram acrescidas produções específicas para definição de roteiros de animação, dentre outras características. Na Seção 3.2 apresentamos as características de propósito geral da linguagem: inclusão de arquivos, declaração de variáveis, definição de funções, declaração de classes e tipos genéricos, sentenças e manipulação de erros. Na Seção 3.3 descrevemos animações, para isso, apresentamos em paralelo a API de animação. A linguagem de animação LA é abordada na Seção 3.4. Exemplos simples de utilização da linguagem são abordados na Seção 3.5. As características do compilador e a forma de compilação são apresentadas na Seção 3.6.

3.2 A Linguagem de Propósito Geral L

Nesta seção apresentamos os principais recursos da linguagem L, a linguagem de propósito geral da qual a linguagem de animação LA foi estendida.

3.2.1 Inclusão de Arquivos

A inclusão de arquivos, definida pela palavra reservada **include** seguida por uma seqüência de caracteres entre aspas, insere o conteúdo de outro arquivo no código fonte. A seqüência de caracteres define o caminho do arquivo a ser incluído. Por exemplo, seja o seguinte trecho de código dado no arquivo `colors.scn`:

```
const Color red   = new Color(1,0,0);
const Color green = new Color(0,1,0);
const Color blue  = new Color(0,0,1);
```

Se desejamos utilizar as cores definidas em `colors.scn` em outro arquivo de descrição de cena, escrevemos:

```
include "colors.scn";
```

3.2.2 Declaração de Variáveis

A linguagem L é fortemente tipada. Isto significa que toda variável e toda expressão tem um tipo que é conhecido em tempo de compilação. Tipos limitam os valores que uma variável pode guardar ou que uma expressão pode produzir para limitar as operações suportadas naqueles valores e determinar o significado da operação. Uma linguagem fortemente tipada ajuda a detectar erros em tempo de compilação.

Os tipos da linguagem L são divididos em duas categorias: os tipos primitivos e os tipos de classe (ou referência). Os tipos primitivos são os tipos booleano (**bool**) e os tipos numéricos. Os tipos numéricos são **int**, **char** e **float**. Os tipos referência são tipos de classe, **null** ou vetores de quaisquer dos tipos. A Tabela 3.1 descreve os tipos da linguagem.

Um objeto é uma instância de um tipo de classe ou vetor criada sempre dinamicamente. A uma variável do tipo `t`, podemos atribuir valores do tipo `t` ou de um tipo que possa

Tipo	Descrição	Tamanho	Valores
bool	Booleano	32 bits	**true**/**false**
char	Caracter	16 bits	UNICODE
float	Número real	64 bits	2,23 10e-308 a 1,79 10e+308
int	Inteiro	32 bits	-2.147.483.648 a 2.147.483.647
Referência	Tipo de classe ou vetor	32 bits	endereço

Tabela 3.1: Tipos da linguagem.

ser convertido para `t`. Declaramos uma variável escrevendo seus modificadores (opcionais) seguidos de seu tipo, nome e inicializador. Por exemplo:

```
//cria uma variável de tipo primitivo (sem inicialização).
bool flag;

//cria e inicializa variáveis de tipo primitivo.
float d = 2.56;
char c1 = 'A', c2 = 'B';

//declara e inicializa uma variável constante.
const float PI = 3.1415;

//cria uma referência e instancia um objeto de tipo de classe.
Color cor;
cor = new Color(1,0,0);

//cria uma referência e instancia um vetor de tipo primitivo.
int[] vector1;
vector1 = new int[5];

//cria uma referência, instancia e inicializa o vetor.
float[] vector2 = {1.0, 2.0, 3.0, 4.0};
```

3.2.3 Definição de Funções

Uma função é definida por um tipo de retorno, seu nome, uma lista de parâmetros formais e modificadores (opcionais). Adicionalmente uma função pode lançar exceções, neste caso, cláusulas de lançamento de exceção são incluídas em sua assinatura (abordamos exceções na Seção 3.2.10). Como exemplo de declaração de função, considere:

```
int area(int largura, int altura)
{
  return largura * altura;
}
```

A passagem de parâmetros para funções é sempre por valor, ou seja, uma cópia dos argumentos é passada à função. Funções admitem protótipos, ou seja, não há a necessidade de se escrever o corpo da função no instante da sua definição, entretanto, toda função deve possuir um corpo, com exceção das *funções nativas*. Uma função é nativa se for definida com o modificador **`native`**. Este modificador indica que as instruções da função encontrar-se-ão definidas em uma biblioteca de ligação dinâmica (.dll, **d**ynamic **l**ink **l**ibrary, no caso do Windows.) Por exemplo: **`int`** `funcao_nativa()` **`native`**;.

3.2.4 Declaração de Classes

Uma classe é uma declaração de tipo em termos de *atributos*, *métodos* e *propriedades*, chamados genericamente de *membros*.

- **Atributos**. Um atributo é uma instância de uma variável que é parte de um objeto. Como os campos de um registro, um atributo de instância representa dados que existem em cada instância da classe. Atributos de instância definidos em uma classe indicam o estado dos objetos daquela classe. Um atributo de classe, ao contrário, é uma variável pertencente à própria classe e compartilhada por todos os objetos da classe. Um atributo de classe é definido com o modificador **`static`** aplicado ao atributo.
- **Métodos**. Um método de instância é um procedimento ou função que opera sobre os atributos de instância e define parte do comportamento dos objetos da classe. Um método de classe pertence à própria classe e opera somente sobre atributos de classe. Como um atributo de classe, um método de classe é definido com o modificador **`static`** aplicado ao método.
- **Propriedade**. Uma propriedade é uma interface para dados associados a um objeto (geralmente armazenados em um atributo). Propriedades possuem especificadores que determinam como seus dados são lidos e/ou modificados.

Tecnicamente, objetos são blocos de memória alocados dinamicamente cuja estrutura é determinada pela sua classe. Cada objeto tem uma única cópia de todos os atributos de instância definidos na classe, porém, todas as instâncias de uma classe compartilham as mesmas instruções que definem seus métodos.

Visibilidade dos Membros da Classe

Cada membro de uma classe possui uma característica chamada *visibilidade*. A visibilidade de um membro é definida por uma das palavras reservadas **`private`**, **`protected`** ou **`public`**. Estas palavras reservadas também são conhecidas como especificadores de visibilidade e são responsáveis pela forma de encapsular os membros da classe. No exemplo a seguir declaramos um atributo privado chamado `_radius`:

```
class Sphere
{
  private:
    float _radius;
  ...
}
```

A visibilidade determina as regras de acesso a um membro de uma classe. Um membro é dito privado se for definido na seção privada da classe, com o especificador de visibilidade **`private`**. Um membro privado só pode ser acessado por métodos da classe na qual foi declarado. Similarmente, um membro é dito protegido se for definido na seção protegida da classe (com o especificador **`protected`**). Um membro protegido é como um membro privado, porém, pode ser acessado diretamente por métodos definidos nas classes *derivadas* da classe na qual foi declarado. Um membro definido na seção pública da classe, com o especificador de visibilidade **`public`**, é visível por todo o programa (possui acesso irrestrito).

3.2.5 Declaração de Métodos

A sintaxe para a declaração de métodos é similar à sintaxe para a declaração de funções. Em métodos, porém, podemos indicar sua visibilidade colocando-os em uma das três possíveis

seções da classe[1]. Os métodos podem ser implementados dentro ou fora do corpo da classe. A implementação de um método fora da classe deve ser qualificado com o nome da sua classe, como a seguir:

```
class Sphere
{
   public:
      void setRadius(float radius);
   ...
}

void Sphere::setRadius(float radius)
{
   ...
}
```

A criação de uma instância de classe executa um método especial da classe chamado *construtor* cuja finalidade é inicializar os atributos de um objeto. O construtor deve ser definido como um método sem tipo de retorno e cujo nome é definido pela palavra reservada **constructor**. Um construtor, assim como um método, pode ou não aceitar argumentos. Além de um construtor, a linguagem permite a definição de destrutores. Na linguagem L, um *destrutor* é um método especial que é executado um momento antes da destruição/coleta do objeto, quando este passar a ser inutilizável/inalcançável. O nome de um método destrutor é definido pela palavra reservada **destructor**. Como um construtor, não se define um tipo de retorno para um destrutor, porém, um destrutor não deve tomar quaisquer argumentos. Por exemplo:

```
class Sphere
{
   public:
      constructor(float radius)
      {
      }
      destructor()
      {
      }
      ...
}
```

3.2.6 Declaração de Atributos

Os atributos têm a mesma sintaxe de declaração de variáveis, estando sujeitos à sua seção de visibilidade. Contudo, os atributos podem ser inicializados no instante de sua declaração somente se forem definidos com os modificadores **const** e **static** simultaneamente:

```
class Sphere
{
   public:
      const static int defaultRadius = 1.0;
}
```

Declaração de Propriedades

Uma propriedade, como um atributo, define uma característica de um objeto. Enquanto um atributo é meramente um local de armazenamento, cujo conteúdo pode ser examinado e

[1]Se não for especificado explicitamente a seção da classe, esta por padrão é uma seção privada.

modificado, uma propriedade associa formas específicas para a leitura e escrita de seus dados através dos especificadores de propriedade **read** e **write**, respectivamente. A declaração de uma propriedade especifica um nome, um tipo e pelo menos um especificador de propriedade. Por exemplo:

```
class Sphere
{
  private:
    float _radius;
  public:
    void setRadius(float radius);

    constructor(float radius)
    {
      this.setRadius(radius);
    }

    property float radius
    {
      read  = _radius;
      write = setRadius
    };
  ...
}

void Sphere::setRadius(float radius)
{
  if (radius > 0)
    this._radius = radius;
  else
    throw new Exception("Raio deve ser positivo.");
}
```

Quando uma propriedade é um *rvalue* (lado direito de uma atribuição), seu valor é o valor do atributo ou o resultado da invocação do método indicado pelo especificador **read**. Quando uma propriedade é um *lvalue* (lado esquerdo de uma atribuição) seu valor é escrito no atributo ou ajustado com o argumento passado na invocação do método indicado pelo especificador **write**. Por exemplo:

```
Sphere e = new Sphere(4); // esfera com raio 4.
e.radius = 3;             // invoca implicitamente o método setRaio().
float tmp = e.radius;     // acessa implicitamente o atributo _radius.
```

Herança

A herança permite definir uma nova classe com base em uma já existente. A classe criada (subclasse ou classe derivada) herda todas as variáveis e métodos da classe já existente (superclasse ou classe base). O mecanismo de herança permite ainda que uma subclasse inclua novos atributos ou sobrecarregue métodos da superclasse. Como exemplo de utilização de herança, considere o trecho de código a seguir:

```
1    class GeometricShape
2    {
3      public:
4        virtual void draw()
5        {
```

```
6           ...
7         }
8       ...
9     }
10
11    class Sphere:  GeometricShape
12    {
13      public:
14        void draw()// Sobrecarga do método GeometricShape::draw().
15        {
16          ...
17        }
18      ...
19    }
```

A classe `Sphere` deriva da classe `GeometricShape` (linha 11). Em `GeometricShape`, o método `draw()` foi definido com o modificador **`virtual`**; tal como C++ (e diferentemente de Java), o acoplamento posterior de mensagens é seletivo. Este modificador indica que o método é virtual, podendo, portanto, ser sobrecarregado em classes derivadas. Métodos virtuais puros podem ser especificados unicamente com o modificador **`abstract`**.

Além de herança simples a linguagem `L` admite herança múltipla. A seguir um exemplo de definição de uma classe que herda as características de duas outras:

```
1     //modela um animal terrestre.
2     class Earthly
3     {
4       ...
5     }
6
7     //modela um animal aquático
8     class Aquatic
9     {
10      ...
11    }
12
13    //modela um animal anfíbio
14    class Amphibious:  Earthly, Aquatic
15    {
16      constructor(...):
17        Earthly(...),
18        Aquatic(...)
19      {}
20      ...
21    }
```

3.2.7 Classes Genéricas

Uma classe é genérica se declara um ou mais tipos variáveis. Esses tipos variáveis são conhecidos como tipos parametrizados da classe. A seção de tipo parametrizado segue o nome da classe e é delimitado pelos símbolos $<$ e $>$. Isto define um ou mais tipos variáveis que atuam como parâmtros na classe. A declaração de classe genérica define um conjunto de tipos parametrizados para cada possível referência à seção. Como exemplo, considere:

```
1     class Pair<X, Y>
2     {
3       private:
```

```
4         X a;
5         Y b;
6
7       public:
8         constructor(X a, Y b)
9         {
10          this.a = a;
11          this.b = b;
12        }
13
14        X getFirst() const
15        {
16          return a;
17        }
18
19        Y getSecond() const
20        {
21          return b;
22        }
23    }
```

O exemplo (linha 1) define uma classe genérica para representar um par de dados. Os tipos passados à seção de parâmetros devem ser quaisquer tipos *não primitivos*. A criação de objetos da classe é tal como segue:

```
1   void main()
2   {
3     String a = "test";
4     Integer b = new Integer(3);
5     Pair<String, Integer> pair = new Pair<String, Integer>(a, b);
6     //...
7     String c = pair.getFirst();
8     Integer d = pair.getSecond();
9     //...
10    Char e = new Char('$');
11    Bool f = new Bool(true);
12    Pair<Char, Bool> pair = new Pair<Char, Bool>(e, f);
13    //...
14    Char g = pair.getFirst();
15    Bool h = pair.getSecond();
16    //...
17  }
```

No exemplo foram definidos quais tipos não primitivos de dados atuam na classe `Pair`. Na linha 5 o parâmetro `X` da classe corresponde ao tipo `String`, enquanto o parâmetro `Y` corresponde ao tipo `Integer`. O mesmo vale para as classes `Char` e `Bool` respectivamente, linha 12.

3.2.8 Sobrecarga de Operadores

A sobrecarga de operadores é um tipo especial de polimorfismo que possibilita o programador a definição de novas semânticas para alguns operadores, tais como `+`, `-`, `*`, `/`, `=`, `+=`, `!=`, etc. No exemplo a seguir, uma versão simplificada da classe `String`:

```
1   class String
2   {
```

```
3       public:
4         constructor(char c) {...}
5         ...
6         String operator + (char c)   {...}
7         String operator = (char c)   {...}
8         String operator = (String s) {...}
9         String operator += (char c)
10        {
11          return this.operator=(this.operator+(c));
12        }
13    }
```

Nas linhas 6-9 foram sobrecarregados os operadores +, = e +=. A utilização desses operadores é exemplificado no trecho de código a seguir:

```
String s = new String('a');
s = 'b';  // o mesmo que s.operator=('b')
s += 'c'; // o mesmo que s.operator+=('c')
```

3.2.9 Sentenças

A linguagem permite o uso de sentenças comuns às linguagens C++ e Java: sentenças de repetição (**while**, **do**...**while**, **for**); sentenças de seleção (**if**, **if**...**else**, **switch**); sentenças de transferência de controle (**return**, **break**, **continue**); sentenças compostas (blocos); manipulação de erros (**throw**, **try**...**catch**); e expressões.

3.2.10 Manipulação de Erros

Um tipo especial de sentença é capaz de tratar exceções em um programa. Uma exceção representa uma condição excepcional que altera o fluxo normal de um programa. Quando um evento deste tipo ocorre, uma exceção é lançada e a execução das instruções é transferida para um bloco de código responsável por tratar esta exceção. É possível, através deste mecanismo, isolar o código responsável pelo tratamento do erro (a própria exceção) em blocos separados, deixando o código principal mais legível. Em adição, é possível a transferência do tratamento de uma exceção para outras(os) funções/métodos da pilha de execução. Por exemplo:

```
1     void funcao1()
2     {
3       try
4       {
5         funcao2();
6       }
7       catch (X e1)
8       {
9         // Trata-se neste bloco uma exceção do tipo X.
10      }
11    }
12    void funcao2() throws (X)
13    {
14      try
15      {
16        if (...)
17          throw new X();
18      }
19      catch (Y e2)
```

```
20      {
21         // Trata-se neste bloco uma exceção do tipo Y.
22      }
23      catch (Z e3)
24      {
25         // Trata-se neste bloco uma exceção do tipo Z.
26      }
27   }
```

No exemplo, `X`, `Y` e `Z` são classes de objetos que representam os tipos de exceções que potencialmente ocorrerão quando da execução do programa. Na linha 17 uma condição qualquer provocou o lançamento de uma exceção do tipo `X`. Nas linhas 19 e 23 são definidos blocos para o tratamento de exceções de tipos `Y` e `Z`, não cabendo, portanto, o tratamento daquela exceção por estes blocos. A função `funcao2()` é definida com o modificador **`throws`**, linha 12, o qual indica que esta função é capaz de lançar uma exceção do tipo `X`[2]. Justifica-se então a necessidade de um bloco de tratamento para uma exceção do tipo `X`, definido a partir da linha 7.

3.3 Descrevendo uma Animação

Até então descrevemos as características de propósito geral de `L`. Neste seção apresentamos os componentes de uma cena a ser animada. Estes componentes são representados por classes de objetos da `API` de animação; muitas delas escritas em `L`. Adicionalmente, nesta seção abordamos os conceitos necessários para a definição das entidades responsáveis pelo fluxo de uma animação: seqüênciadores e eventos.

3.3.1 Cena e Seus Componentes

Os componentes de uma animação são objetos de classes `L` os quais são agrupados na `API` de animação do sistema. As classes que representam uma cena e seus componentes principais são descritos no diagrama `UML` da Figura 3.1 e comentadas a seguir.

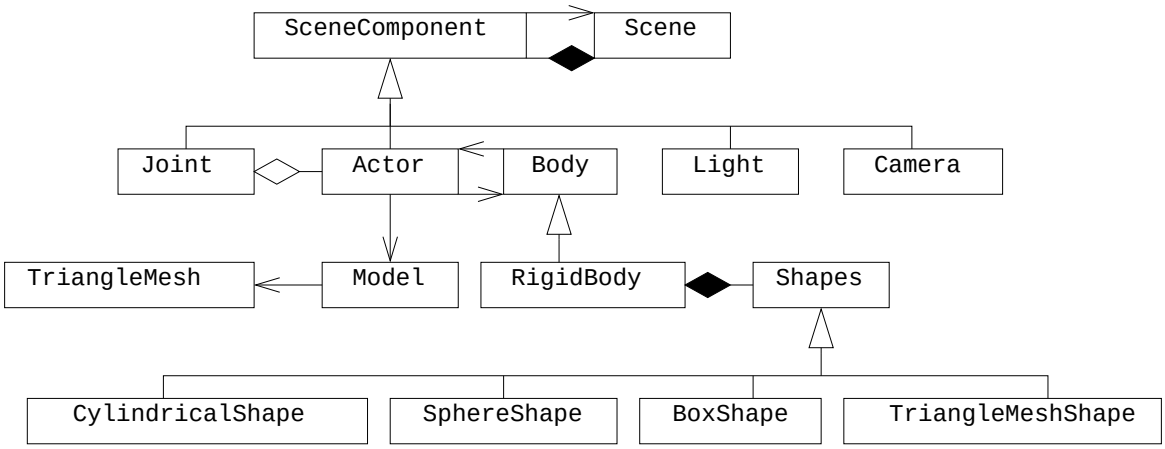


Figura 3.1: Classes cena e componentes de cena.

Um objeto da classe `Scene` é um container de atores, junções, luzes e câmeras. Um objeto da classe `Actor` é definido por um modelo geométrico e um corpo. Um modelo geométrico é

[2]Um método/função pode lançar mais de um tipo de exceção; no exemplo, poderiamos declarar **`void`** funcao2() **`throws`** (X, Y, Z).

uma instância da classe `Model` e descreve a pose, formas e dimensões do ator. Este modelo é usado pelo renderizador para sintetizar a aparência do ator e é definido por uma malha de trinângulos — objeto da classe `TriangleMesh`.

O corpo do ator é um objeto da classe derivada da classe abstrata `Body` e define as propriedades físicas de um ator. A classe `RigidBody` é uma classe concreta que encapsula as propriedades específicas e métodos usados pelo motor de física para simulação dinâmica de corpos rígidos. A geometria de um corpo rígido é definido pela coleção de objetos da classe derivada da classe abstrata `Shape`, tal como esferas (`SphereShape`), caixas (`BoxShape`), cilindros (`CylindricalShape`) e malhas de triângulos (`TriangleMeshShape`), os quais são usados pelo PhysX para computar o centro de massa e os pontos de contato entre atores. Em geral, a geometria de um corpo é mais simples que a geometria do modelo (Figura 2.13), embora possa ser a mesma. Atores podem ser unidos por junções, que são representadas por instâncias especializadas da classe abstrata `Joint`.

Para a compreensão dos exemplos de animações, somente algumas das interfaces das classes de objetos usadas são listadas. Por comodidade visual, a seqüência de três pontos indica que detalhes são omitidos.

```
class SceneComponent
{
  private:
    Scene myScene;
  public:
    property Scene scene { read = myScene };
    ...
}

class Actor:  SceneComponent
{
  public:
    property String name {...};
    property Body body {...};
    property Model model {...};
    ...
}

class RigidBody:  Body
{
  public:
    property Vector3D globalPosition {...};
    property Quaternium globalOrientation {...};
    property Vector3D localPosition {...};
    property Quaternium localOrientation {...};
    property Vector3D linearVelocity {...};
    property Vector3D angularVelocity {...};
    property Vector3D linearMomentum {...};
    property Vector3D angularMomentum {...};
    property float kineticEnergy {...};
    property float mass {...};
    property Tensor3D inertia {...};
    property Vector3D centerOfMass {...};
    property List <Shapes> shapes {...};
    ...
    void addForce(Vector3D force);
    void addTorque(Vector3D torque, Vector3D axis);
}
```

```
class SphereShape:  Shape
{
  public:
    property float radius {...}
    property Vector3D globalPosition {...};
    property Quaternium globalOrientation {...};
    property Vector3D localPosition {...};
    property Quaternium localOrientation {...};
    ...
}

class BoxShape:  Shape
{
  public:
    Vector3D dimensions {...};
    ...
}

class Camera:  SceneComponent
{
  public:
    property String name {...};
    property float aspectRatio {...};
    property float distance {...};
    property float viewAngle {...};
    property int projection {...};
    property Vector3D position {...};

    void azimuth(double angle);
    void zoom(double z);
    ...
}
```

3.3.2 Seqüenciadores e Eventos

Na API do sistema há classes de objetos chamados de *seqüenciadores* que, em conjunto com o motor de física, são responsáveis por definir as alterações na cena durante uma animação. Um *seqüenciador* é qualquer conjunto de atividades que, quando executadas seqüencialmente ou em paralelo, podem modificar o estado de um ou mais objetos de uma cena no tempo. Um seqüenciador pode controlar o movimento de atores, luzes e câmeras; alterar atributos de um modelo tal como cores e texturas; criar novos componentes de cena; aplicar forças e torques sobre os corpos rígidos; iniciar outros seqüenciadores; etc. Um seqüenciador é um objeto da classe derivada da classe abstrata `Sequencer`, exibida no diagrama de classes da Figura 3.2. A interface de um seqüenciador é mostrada a seguir.

```
abstract class Sequencer:  SyncObject
{
  public:
    abstract void start();
    abstract void exit();

    float getTime();
    property float time { read = getTime };
}
```

O método `start()` inicia a execução das atividades de um seqüenciador. Todos os seqüenciadores iniciados serão executados em paralelo pela `MVA`. O método `exit()` termina a execução de um seqüenciador. A propriedade `time` (somente de leitura), bem como o método `getTime()`, indicam o tempo em milisegundos desde quando o seqüenciador foi iniciado.

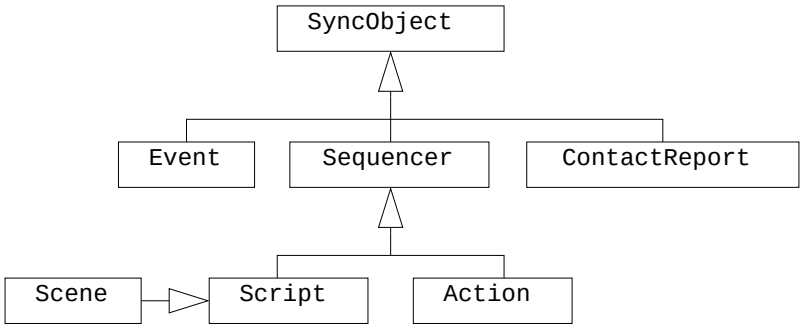


Figura 3.2: Classes de seqüenciadores e evento.

Um seqüenciador pode ser ou um *script* ou uma ação. Um *script* é um objeto da classe derivada da classe abstrata `Script`:

```
abstract class Script:  Sequencer
{
   public:
      void start();
      void exit();

      void waitFor(float);
      void waitFor(SyncObject, float = -1);
      void waitFor(SyncObject[], float = -1);

   protected:
      abstract void run();
}
```

A classe `Script` sobrecarrega os métodos `start()` e `exit()` herdados da classe `Sequencer`. Quando um *script* é iniciado, o controlador solicita à `MVL` a invocar o método `run()`, o qual deve ser sobrecarregado em classes derivadas. O método implementa as atividades que devem ser executadas por um *script* específico. O estado de um *script* que tem sua execução iniciada é definido como `RUNNING`. Um *script* termina quando `run()` retorna ou quando `exit()` é invocado; em ambos os casos, o estado do *script* torna-se `TERMINATED`.

Para um *script* no estado `RUNNING`, o código de `run()` será inteiramente executado pela `MVL` em exatamente um *tick*, isto é, durante um ciclo de atualização (em termos do tempo de animação, isto significa *instantaneamente*). Entretanto, se um dos métodos `waitFor()` é invocado a partir do método `run()`, então a execução do *script* é suspendida pelo controlador até que a condição especificada pelos argumentos passados ao método seja satisfeta. O estado de um *script* suspendido é definido como `WAITING`. Um *script* pode aguardar por um número de ciclos de atualização até que: ou seu tempo de espera (*timeout*) em milisegundos expire, ou um ou mais objetos de sincronização tornarem-se sinalizados. Assim que a condição seja verificada, o controlador instrue à `MVL` a retomar a execução do método `run()` na próxima instrução depois da invocação do método `waitFor()`. O *script*, então, retorna para o estado `RUNNING`. A Figura 3.3 ilustra um autômato que resume dos estados que um *script* pode permanecer.

Um objeto de sincronização é uma instância de uma classe derivada da classe abstrata `SyncObject`. Ele representa um sinal esperado por um *script* que esteja no estado `WAITING`

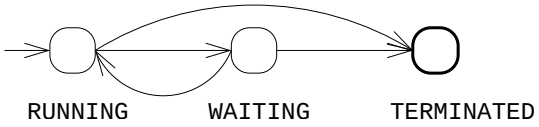


Figura 3.3: Estados de um *script*.

para ter sua execução retomada pela MVL (voltando ao estado RUNNING). Um objeto de sincronização pode ganhar um dos dois estados a qualquer instante: *sinalizado* e *não sinalizado*. Como mostra a Figura 3.2, seqüenciadores são objetos de sincronização. Quando um seqüenciador é criado e executado (passando ao estado RUNNING), ele torna-se *não sinalizado*. Assim que um seqüenciador termina sua execução, ele torna-se *sinalizado*. Desta forma, um *script* pode aguardar por outros *scripts* (e ações) concluirem sua execução antes de proceder com a sua.

Além daqueles, há outros dois tipos de objetos de sincronização: *eventos* e *notificadores de contato*. Um evento genérico é uma instância da classe `Event`:

```
class Event:  SyncObject
{
  public:
    void setSignaled();
}
```

O método `setSignaled()` é invocado para ajustar manualmente o estado de um evento como *sinalizado*.

Um *notificador de contato* é um evento interno sinalizado pela MVA sempre que um contato entre quaisquer dois atores é detectado. O evento é uma instância de `ContactReport`:

```
class ContactReport:  SyncObject
{
  public:
    static ContactReport getInstance();
    property List<Contact> contacts {...};
}
```

Um código de animação não deveria criar um objeto da classe `ContactReport`. A única instância que deveria ser usada é mantida pela MVA cuja referência é obtida por meio da invocação do método estático `getInstance()`. O objeto mantém uma lista com os pontos de contato detectados no *tick* corrente. Cada ponto de contato, entretanto, é representado por uma instância da classe `Contact`:

```
class Contact
{
  public:
    bool isBetween(Actor a1, Actor a2) const;
    property List<Vector3D> points {...};
    ...
}
```

O método `isBetween()` da classe `Contact` retorna **`true`** se o contato é entre os atores especificados como argumentos do método. A propriedade `points` indica a lista de pontos de contato entre os atores envolvidos.

Uma ação é um seqüenciador instância de uma classe derivada da classe abstrata `Action`:

```
abstract class Action:  Sequencer
{
  public:
    void start();
    void exit();
    void abort();

    constructor(float = -1);
    property float lifetime{...};

    virtual void init();
    virtual void update();
    virtual void finalize();
}
```

Uma ação tem como uma propriedade o seu tempo de vida em milisegundos, o qual é passado como argumento para o construtor (um número negativo é considerado infinito). A classe `Action` sobrecarrega os métodos `start()` e `exit()` herdados da classe `Sequencer`. As atividades de uma ação são codificadas nos métodos `init()`, `update()` e `finalize()`, os quais podem ser sobrecarregados em classes derivadas. O método `init()` implementa um código de inicialização para a ação e é executado apenas uma vez pela `MVL` assim que a ação é iniciada. Logo depois da invocação do método `start()` mas antes da invocação do método `init()`, o estado da ação é ajustado para `INITIATED`. (Note que o código de inicialização pode ser escrito no construtor, mas neste caso será executado logo depois da criação do objeto, o que nem sempre é desejável). Uma vez iniciada, o estado da ação é ajustado para `RUNNING`.

Em cada *tick* de tempo o controlador decrementa o valor da propriedade `lifetime` de uma ação no estado `RUNNING`. Se o resultado é um número positivo, então o controlador solicita à `MVL` a invocação do método `update()` da ação. Uma ação termina quando seu tempo de vida é zero ou quando o método `exit()` é invocado. Neste caso, o estado da ação passa para `TERMINATED` e o método `finalize()` é invocado. Se ao invés, `abort()` é invocado, a ação passa ao estado `TERMINATED` porém sem executar o método `finalize()`. A Figura 3.4 ilustra um autômato que resume dos estados que uma ação pode permanecer.

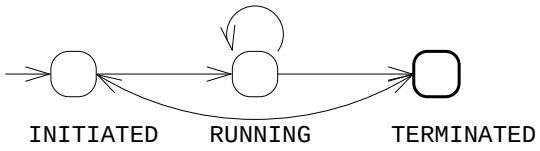


Figura 3.4: Estados de uma ação.

Ações são usadas para definir atividades que *continuamente* variam no tempo e devam ser executadas a cada ciclo de atualização da animação, enquanto *scripts* são usados para definir uma seqüência linear de atividades sincronizadas que devam ser executadas apenas uma vez, em um período de tempo conhecido antecipadamente.

Uma cena é também um *script*, no sentido de que ela pode ser iniciada e definir uma seqüência de atividades que podem ser executadas em conjunto com seus componentes. A interface da classe `Scene` é:

```
abstract class Scene:  Script
{
  public:
    static Scene getCurrent();
    constructor(float = -1);
```

```
    property float totalTime {...};
    property List<Actor> actors {...};
    property Camera camera {...};
    //...

  protected:
    void run();
}
```

3.4 A Linguagem de Animação LA

É possível usar a linguagem L juntamente com a API para especificar completamente uma animação. Todavia, é melhor fazer isto com a linguagem de animação. LA possui extensões que torna fácil a criação de cenas, adicionar componentes em uma cena, e definir seqüenciadores. As principais características da linguagem LA são discutidas nesta seção.

LA introduz blocos de propriedade. Como um exemplo, considere o seguinte trecho de código escrito em L:

```
Actor actor = new Actor();
actor.name = "actor1";

RigidBody body = new RigidBody();

body.position = <0,0,0>;
body.orientation = new Quaternion(0,<1,1,1>);
body.mass = 50;
body.centerOfMass = <0,0,1>;
...
actor.body = body;
```

O código cria um novo corpo rígido e ajusta sua posição, orientação, massa e centro de massa. O corpo rígido é atribuído ao novo ator de nome "actor1". Note que um vetor 3D pode ser definido por uma expressão <x,y,z>, onde x, y e z são expressões do tipo **float**. Usando blocos de propriedades, o código escrito pode ser reescrito em LA como se segue:

```
Actor actor = new Actor()
{
  name = "actor1";
  body = new RigidBody()
  {
    position     = <0,0,0>;
    orientation  = new Quaternion(0,<1,1,1>);
    mass         = 50;
    centerOfMass = <0,0,1>;
  };
};
```

Um bloco de propriedade é uma expressão definida como uma expressão seguida de um bloco que contém uma lista de expressões de atribuição. A produção da gramática é:

```
PropertyBlock:
  Expression "{"(AssignmentExpression;)* "}";
```

O valor de `Expression` deve ser uma referência não nula para um objeto `O`, e o *lvalue* de cada expressão de atribuição no bloco deve ser uma propriedade ou um atributo de `O`. O valor da expressão bloco de propriedade é uma referência para `O`.

Uma variante do bloco de propriedade pode ser aplicada para adicionar elementos dentro de uma coleção. Para a `MVA`, uma coleção é qualquer objeto instância de classes derivadas da classe abstrata `Collection`, tais como `Vector` e `List`, que representam coleções implementadas como vetor e lista encadeada, respectivamente. Seja o trecho de código:

```
Scene scene = new Scene();
Actor actor;

//cria "actor1" e adiciona na cena.
actor = new Actor();
actor.name = "actor1";
actor.body = new RigidBody();
//...
scene.actors.add(actor);
//cria "actor2" e adiciona na cena.
actor = new Actor();
actor.name = "actor2";
actor.body = new RigidBody();
//...
scene.actors.add(actor);
```

Usando blocos de propriedades com sua variante que adiciona em coleções, o código escrito pode ser reescrito em `LA` como segue:

```
Scene scene = new Scene()
{
  actors
  {
    //cria "actor1" e adiciona na cena.
    new Actor()
    {
      name = "actor1";
      body = new RigidBody()
      {
        ...
      };
    };
    //cria "actor2" e adiciona na cena.
    new Actor()
    {
      name = "actor2";
      body = new RigidBody();
      {
        ...
      };
    };
  };
};
```

A variante de bloco de propriedades que adiciona em coleções é definida como uma expressão seguida por um bloco contendo uma lista de expressões. A produção da gramática é:

```
AddIntoCollectionVariant:
  Expression "{" (Expression;)* "}";
```

O valor da primeira Expression deve ser uma referência não nula para um objeto O o qual espera-se ser uma coleção de objetos de um tipo T. O valor de cada expressão em um bloco deve ser uma referência para um objeto do tipo T. O valor da expressão *bloco de propriedade variante* é a própria referência para o objeto O.

Bloco de propriedades com sua variante oferece uma aparência clara para partes descritivas do código de animação, tal como PSCL [35].

Outra extensão em LA é a definição de classes anônimas. Por exemplo, suponha que o animador queira declarar uma nova classe derivada de Scene, sobrecarregando o método run() da classe Script, e iniciando uma única instância (isto é, um *singleton*) da classe. O código em L é:

```
class MyScene:  Scene
{
   protected:
      void run()
      {
         // cria alguns atores.
         // inicia alguns scripts e/ou ações.
         //...
      }
}

(new MyScene()).start();
```

Se somente uma instância de MyScene é criada, então a declaração explícita da classe pode ser evitada. O seguinte código em LA cria e inicia um *singleton* de uma classe anônima derivada de Scene:

```
new Scene() class
{
   protected:
      void run()
      {
         // cria alguns atores.
         // cria alguns scripts e/ou ações.
         //...
      }
}.start();
```

No exemplo, a palavra reservada **class** depois da expressão **new** denota que uma nova instância de uma classe derivada da classe Scene será criada. O corpo da classe anônima segue depois da palavra reservada **class**. A mensagem start() é então enviada para o *singleton*. A sintaxe para definição de uma classe anônima é:

```
AnonymousNewExpression:
   new Name ( ExpressionList? ) class ClassBody
```

Em adição, LA declara as seguintes palavras reservadas:

- **run**: quando usada no corpo de uma classe derivada da classe Script, detona o cabeçalho para o método run().
- **init**, **update** e **finalize**: quando usados no corpo de uma classe derivada da classe Action, denotam cabeçalhos para os métodos init(), udpate() e finalize(), respectivamente.

Se uma ação deve executar algumas atividades em um dado instante do seu tempo de vida, a sentença **switch_time** pode ser utilizada no interior do bloco **update**, tal como apresentado a seguir:

```
class MyAction:  Action
{
   constructor(float lifetime);

   init
   {
      // código de iniciação aqui.
   }

   update
   {
      switch_time(time)
      {
         from 0 to 2000:
            //faça qualquer coisa.
         at 5000:
            //faça qualquer coisa.
         from 1000:
            //faça qualquer coisa.
         to 7000:
            //faça qualquer coisa.
         from 1000 for 6000:
            //faça qualquer coisa.
         for lifetime:
            //faça qualquer coisa.
      }
   }

   finalize
   {
      // código de finalização aqui.
   }
}
```

A sentença **switch_time** toma como argumento uma expressão do tipo **float**, tipicamente o tempo local da ação ou o tempo total da cena. No bloco de sentença, códigos podem ser associados a intervalos de tempo os quais são especificados por condições **at**, **from-to**, **from**, **to**, **from-for** e **for**. A MVA executa o código para todas as condições que são satisfeitas no ciclo de atualização corrente.

Para criar e iniciar uma instância escreve-se:

```
/* O construtor é invocado.  O argumento é o tempo de vida da ação.
Action action = new MyAction(10000);
//O bloco init é invocado e o bloco update é executado a cada tick.
action.start();
```

Como uma alternativa, pode-se usar uma expressão **start** para criar e imediatamente dar início a um seqüenciador:

```
Action action = start MyAction(10000)
```

3.5 Exemplos

Nesta seção apresentamos dois exemplos que ilustram como criar cenas simples, *scripts* e ações. Os parâmetros que definem as dimensões de cada imagem (*frame*) da animação, o intervalo entre quadros, dentre outras são encapsulados em um objeto da classe `RenderingContext` (Tabela 3.2), obtido com invocação do método `getInstance()` da classe.

Classe `RenderingContext`	
Métodos públicos	
static `RenderingContext getInstance()`	Retorna o *singleton* de `RenderingContext`.
Propriedades públicas	
property int `imageHeight`	Altura da imagem.
property int `imageWidth`	Largura da imagem.
property int `resolution`	Resolução da imagem
property bool `generateFrame`	Habilita ou não a geração de *frames*.
property `String fileName`	Nome do arquivo de imagem.
property `Renderer renderer`	O renderizador das cenas.

Tabela 3.2: Classe `RenderingContext`.

O primeiro exemplo é de uma cena cujo estado inicial é definido por uma pilha formada por caixas (`BoxShape`) sobre um plano (`PlaneShape`), além de uma esfera (`SphereShape`) posicionada acima da pilha (Figura 3.5(a)). Quando a cena é iniciada, a gravidade atua e a esfera cai sobre a pilha (Figura 3.5(b)). Posteriormente, um *script* atira, a cada dois segundos, vinte novas esferas na direção de projeção (Figura 3.5(c,d)). Em paralelo, uma ação é iniciada para rotacionar a câmera ao redor da cena e para criar uma nova pilha de caixas em um dado instante (Figura 3.5(e)). O trecho de código de animação é apresentado e comentado a seguir:

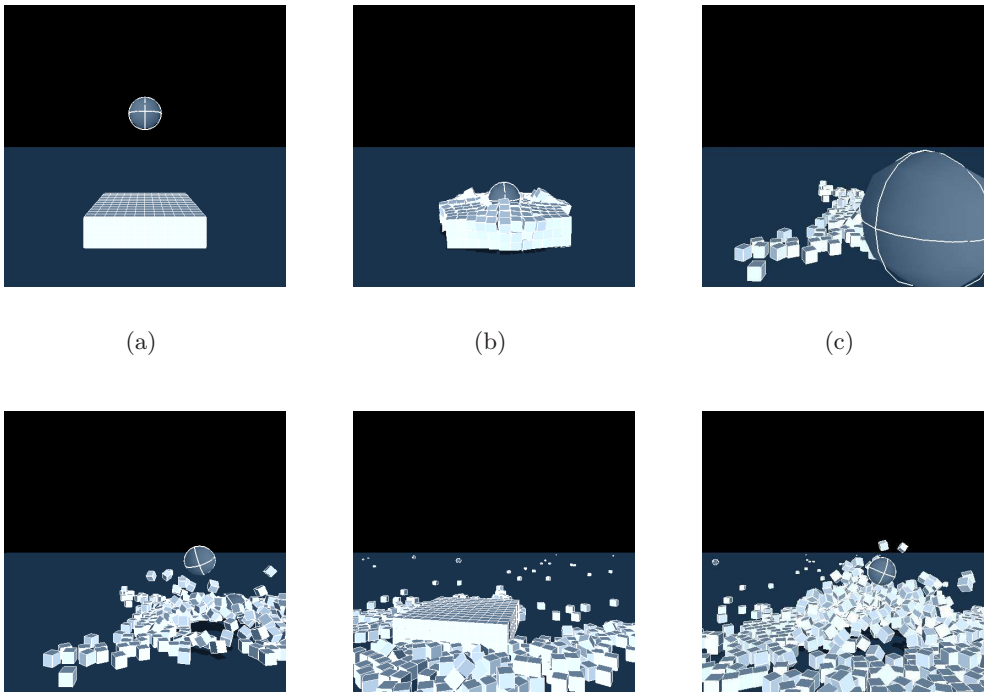
```
//Esta é a função principal da animação.
void main()
{
  //Inicializa parâmetros de renderização.
  RenderingContext rContext = tRenderingContext::getInstance();

  rContext.imageWidth = 512;
  rContext.imageHeight = 512;
  rContext.resolution = 1;
  rContext.ticksPerSecond = 30;
  rContext.generateFrame = true;
  rContext.renderer = new OpenGlRenderer();

  start Scene() class //Cria uma cena e inicia seu script.
  {
    //adiciona um plano à cena.
    void createGroundPlane();

    //adiciona uma pilha de caixas à cena.
    void createBoxStack();

    /* Cria um ator cujo corpo é uma esfera.
    * Uma vez que o ator não possui um modelo,
    * seu corpo é usado para renderização.*/
    Actor createSphere(Vector3D position)
    {
```



(a) (b) (c)

(d) (e) (f)

Figura 3.5: Quadros do exemplo 1.

```
   Actor sphere = new Actor()
   {

      body = new RigidBody()
      {
         shapes
         {
            new SphereShape()
            {
               globalPosition = position;
               radius = 1;
            };
         };
      };
   };

   actors.add(sphere);
   return sphere;
}

// Este é o script.
run
{
   //Cria os primeiros atores.
   this.createGroundPlane();
   this.createBoxStack();
   this.createSphere(<0,6,0>);
```

```
        //Inicia uma ação para mover a câmera.
        start Action(this, 42000) class
        {
          Scene s;

          constructor(Scene s, float lifetime):
            Action(lifetime)
          {
            this.s = s;
          }

          update
          {
            // Continuamente rotaciona a câmera.
            s.camera.azimuth(0.5);
            /*
            * A sentença switch_time abaixo
            * de fato não é necessaria e pode
            * ser trocada por uma sentença if.
            */
            switch_time(time)
            {
              /*
              * Cria uma nova pilha no tempo 32s
              * (Figura 3.5(e)).
              */
              at 32000:
                s.createBoxStack();
            }
          }
        };


        //arremessa esferas a cada 2 segundos
        for(int ball = 0; ball < 20; ++ball)
        {
          waitFor(2000);
          createSphere(s.camera.position)
          {
            linearVelocity = s.camera.DOP * 60;
          };
        }
      }
    };
  }
```

O próximo exemplo demonstra como usar o evento notificador de contato (objeto da classe `ContactReport`). A classe `CheckContact` abaixo define um *script* que, uma vez iniciado, aguarda até que ocorra um contato entre os atores `a1` e `a2`:

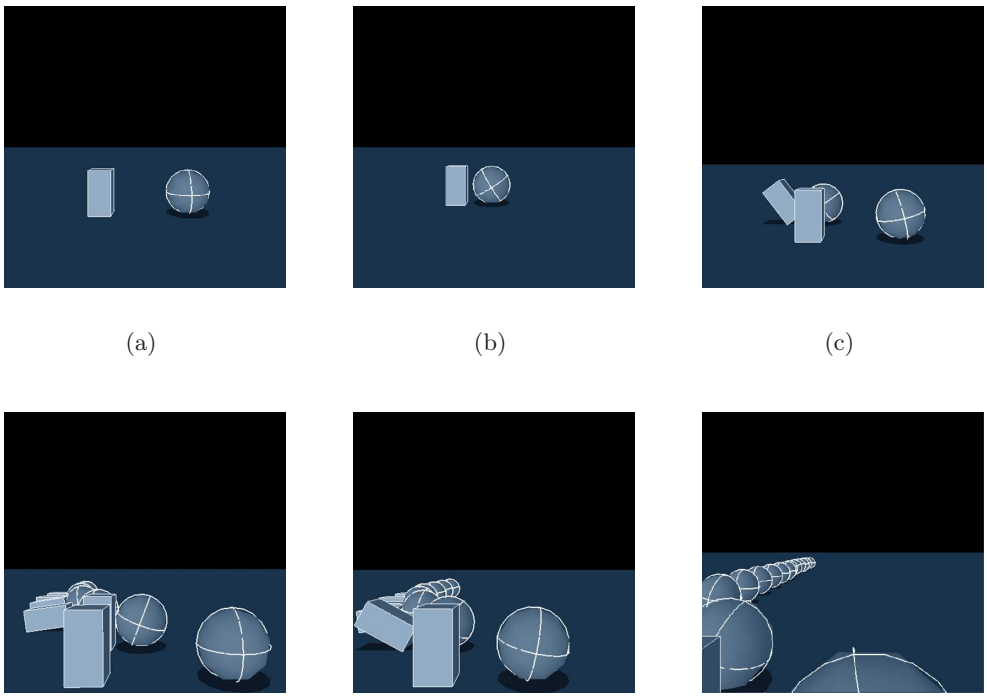
```
class CheckContact:  Script
{
  private:
    Actor a1;
    Actor a2;

  public:
```

```
    constructor(Actor a1, Actor a2)
    {
      this.a1 = a1; this.a2 = a2;
    }
    run
    {
      ContactReport r;

      for(r = ContactReport::getInstance();;)
      {
        waitFor(r);
        for (Contact c:  r.contacts)
          if (c.isBetween(a1, a2))
            return;
      }
    }
}
```

O *script* da cena é bem simples. Ele cria quarenta vezes uma esfera e uma caixa, aplica uma força sobre a esfera para fazê-la entrar em rota de colisão com a caixa e aguarda por um contato. Em paralelo, uma ação move a cãmera na direção do eixo positivo z do sistema global. A Figura 3.6 ilustra alguns quadros da animação.



(a) (b) (c)

(d) (e) (f)

Figura 3.6: Quadros do exemplo 2.

```
void main()// Esta é a função principal da animação.
{
  //Inicializa parâmetros de renderização.
  RenderingContext rContext = tRenderingContext::getInstance();
```

```
rContext.imageWidth = 512;
rContext.imageHeight = 512;
rContext.resolution = 1;
rContext.ticksPerSecond = 30;
rContext.generateFrame = true;
rContext.renderer = new OpenGlRenderer();

// Cria uma cena e inicia seu script.  Seu tempo de vida é de 42s
start Scene(42000) class
{
  constructor(float totalTime):
    Scene(totalTime)
  {}

  //adiciona um plano à cena.
  void createGroundPlane()
  {
    ...
  }
  // Adiciona uma esfera na cena.
  Actor createSphere(Vector3D position)
  {
    ...
  }
  // Adiciona uma caixa na cena.
  Actor createBox(Vector3D position)
  {
    ...
  }

  run // Este é o script.
  {
    // Cria o plano da cena.
    this.createGroundPlane();

    // Inicia a ação para mover a câmera.
    start Action(camera) class
    {
      Camera c;

      constructor(Camera c)
      {
        this.c = c;
      }

      update
      {
        c.pan(<0,0,10>);
      }
    };

    for(int i = 0, z = 0; i <= 40; i++, z += 3)
    {
      // Cria uma esfera e uma caixa.
      Actor s = createSphere(<2,0,z>);
      Actor b = createBox(<-2,0,z>);
```

```
            // Aplica uma força na esfera.
            s.body.addForce(<-550,0,0>);

            //Aguarda por um contato entre a esfera e a caixa e repete
            waitFor(start CheckContact(s, b));
        }
      }
    };
}
```

3.6 Compilando uma Animação

O compilador da linguagem é um componente do sistema que toma como entrada um arquivo texto contendo a descrição de uma animação (`.scn`, **sc***e***n***e description*), e produz como saída um arquivo contendo o código objeto da animação (`.oaf`, **o***bject* **a***nimation* **f***ile*). Nesta seção mostramos como são implementadas as fases de análise léxica, sintática, semântica e geração de código, além de apresentar as principais classes de objetos que compõe a `API` de compilação. O compilador é um dos componentes mais importantes do sistema, porém, em virtude de limitação de espaço, este componente será descrito apenas em linhas gerais.

3.6.1 Principais Classes do Compilador

Os relacionamentos das principais classes de objetos do compilador é tal como ilustra o diagrama UML da Figura 3.7 e são descritas a seguir.

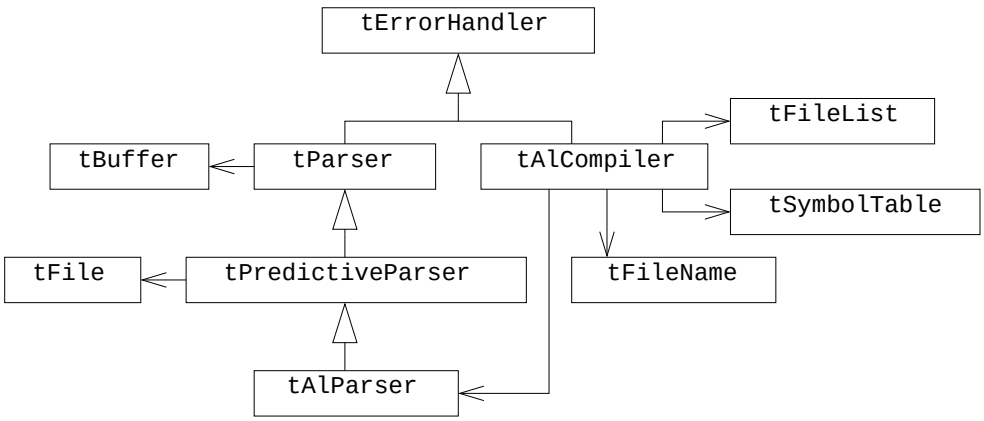


Figura 3.7: As principais classes do compilador.

- A classe `tBuffer` encapsula a implementação de um algoritmo de bufferização para os arquivos de entrada de modo a evitar excessivos acessos a disco. A classe `tFile` encapsula as principais funções de manipulação de arquivos tais como abertura, fechamento, leitura, escrita, etc.

- A classe `tErrorHandler` define os métodos para manipulação de erros efetuada pelo compilador. A classe possui métodos e atributos para manusear a ocorrência de um erro, formatar a mensagem de erro correspondente, tomar os argumentos da mensagem e imprimir a mensagem em um dispositivo de saída.

- A classe `tParser` encapsula a implementação de um *parser* genérico. Esta classe é usada como classe base para a implementação do parser específico para a linguagem de animação, um objeto da classe `tAlParser`.
- A classe `tPredictiveParser` é uma classe abstrata e encapsula as principais funcionalidades características de um analisador sintático do tipo descendente recursivo.
- A classe `tFileList` encapsula a funcionalidade de uma lista de arquivos a serem compilados. Ela possue métodos para manutenção da lista e verificação de elementos reincidentes, evitando-se, portanto, a compilação de um mesmo arquivo repetidas vezes.

3.6.2 A Tabela de Símbolos

A tabela de símbolos é uma das estruturas de dados fundamentais do compilador da linguagem de animação. A tabela de símbolos armazena todos os símbolos encontrados durante a compilação. Estes símbolos são as variáveis locais e globais, atributos de classe e de instância, funções, métodos, propriedades e classes. A tabela de símbolos é um objeto da classe `tSymbolTable` e seus símbolos são objetos de uma classe derivada da classe abstrata `tSymbol`. A Figura 3.8 mostra as principais classes da tabela de símbolos.

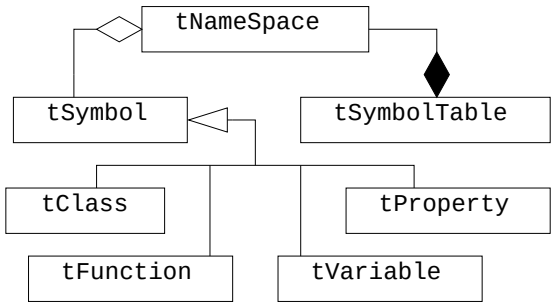


Figura 3.8: Classes da tabela de símbolos.

A classe `tSymbolTable` possui uma coleção de objetos da classe `tNameSpace`, e esta, por sua vez, possui uma coleção de objetos da classe `tSymbol`. Estes símbolos são armazenados numa árvore binária balanceada na qual os símbolos mais à esquerda de um nó da árvore são lexicograficamente anteriores aos símbolos do lado direito do nó.

A classe `tNameSpace` representa um espaço de nomes. Um espaço de nomes é criado quando o compilador encontra o início de um bloco de sentença ou uma declaração de classe. (Em um mesmo espaço de nomes podem existir dois ou mais símbolos do tipo função, pois a linguagem de animação admite funções polimórficas.) Quando a tabela de símbolos cria um espaço de nomes, o espaço de nomes criado passa a ser o espaço de nomes corrente. Desta forma, todos os símbolos encontrados durante a compilação são inseridos no espaço de nomes corrente da tabela de símbolos. A linguagem de animação permite a declaração de variáveis e constantes globais; por esta razão, a tabela de símbolos, quando criada, já possui pelo menos um *espaço de nomes global* onde são armazenados estes símbolos.

Podemos observar, na Figura 3.8, as classes `tClass`, `tVariable`, `tFunction` e `tProperty`, derivadas da classe `tSymbol`. Na verdade não inserimos objetos da classe `tSymbol` na tabela de símbolos, mas somente objetos das classes derivadas da classe `tSymbol`. A classe `tSymbol` possui atributos e métodos comuns a todo símbolo: nome, os modificadores e a classe onde o símbolo foi definido, entre outros. As classes derivadas declaram atributos e métodos específicos para cada tipo de símbolo.

A classe `tClass` representa uma classe ou tipo primitivo da linguagem. Esta classe é diferente das demais pois é um símbolo e um espaço de nomes ao mesmo tempo. Ao inserirmos um objeto da classe `tClass` na tabela de símbolos, estamos inserindo um símbolo e também criando um novo espaço de nomes. Este espaço de nomes se torna corrente e todos os membros da classe são inseridos neste objeto. A classe `tClass` é a principal classe do sistema de tipos do compilador.

A classe `tVariable` representa uma variável global, local ou atributo de uma classe. A classe `tVariable` possui atributos e métodos que definem o tipo da variável e o seu deslocamento na área em que está armazenada.

A classe `tFunction` representa uma função global ou método declarado em uma classe. `tFunction` possui atributos e métodos que definem o tipo de retorno da função ou método e uma lista de parâmetros formais. A lista de parâmetros formais define o tipo e o nome de cada um dos argumentos que a função recebe como entrada. Uma função também possui um objeto da classe `tCode`, o qual armazena todo o código objeto gerado para a função como resultado do processo de compilação.

A classe `tProperty` representa uma propriedade. Uma propriedade só pode ser membro de classe e portanto sempre será armazenada no espaço de nomes de uma classe. Uma propriedade possui atributos e métodos que definem o seu tipo e os símbolos que representam os atributos ou métodos associados à leitura e escrita.

3.7 Sistema de Tipos

O sistema de tipos implementa a representação interna dos tipos de dados oferecidos pela linguagem de animação. Um tipo de dado é representado por um objeto da classe `tClass`. Na Figura 3.9 temos uma visão da hierarquia de classes do sistema de tipos do compilador.

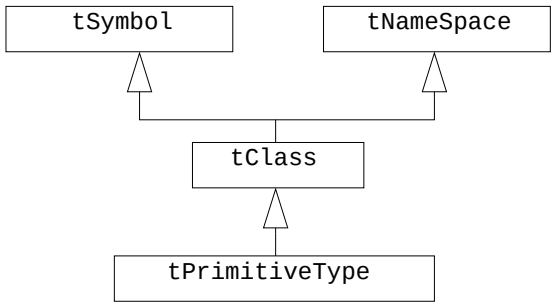


Figura 3.9: Classes do sistema de tipos.

Os tipos de dados podem ser divididos em duas categorias, os tipos primitivos e os tipos de classe. Os tipos primitivos são os tipos **`void`**, **`bool`**, **`char`**, **`int`**, **`float`** e fazem parte da especificação da linguagem, tal como abordado na Seção 3.2.2. Os tipos primitivos são representados pela classe `tPrimitiveType`, derivada da classe `tClass`.

3.7.1 Acomplamento de Mensagens

A linguagem do sistema de animação, além de implementar herança, encapsulamento e polimorfismo, possibilita ao animador a invocação de métodos virtuais. Métodos virtuais estão estreitamente associados ao conceito de polimorfismo. Métodos virtuais definidos em uma classe base podem ser sobrecarregados em classes derivadas. O acoplamento de mensagens é seletivo (estático e/ou dinâmico).

Toda classe da linguagem possui uma *tabela de ponteiros para métodos virtuais* (TPMV). Cada entrada dessa tabela é um ponteiro de 32 bits que armazena um endereço de um método virtual declarado na própria classe ou herdado das classes bases. Cada método virtual, portanto, possui como atributo um índice correspondente ao número de sua entrada na TPMV da classe.

Uma classe derivada contém inicialmente uma TPMV que é uma cópia da TPMV de sua classe base. Quando um método é sobrecarregado, o endereço da TPMV correspondente é alterado para o endereço do método sobrecarregado. Métodos virtuais novos, por sua vez, tem seus endereços adicionados em extensões da TPMV da classe, conforme o exemplo a seguir:

```
class X
{
  virtual void mx();
}
```

(a) Classe X.

TPMV da classe X	
0	&X::mx

(b) TPMV de X.

```
class Y: X
{
  void mx();
  virtual void my1();
  virtual void my2();
}
```

(c) Classe Y.

TPMV da classe Y	
0	&Y::mx
1	&Y::my1
2	&Y::my2

(d) TPMV de Y.

```
class Z: Y
{
  void my2();
}
```

(e) Classe Z.

TPMVda classe Z	
0	&Y::mx
1	&Y::my1
2	&Z::my2

(f) TPMV de Z.

Figura 3.10: Tabela de ponteiro para métodos virtuais.

No acoplamento de mensagens, portanto, a TPMV da classe do objeto mantém informações para identificar, em tempo de execução, qual método correto deve ser executado.

A TPMV de todas as classes de objetos definidas na linguagem de animação são informações mantidas no código objeto gerado pelo compilador.

3.7.2 Análise e Geração de Código

A análise e geração de código têm início com a invocação do método Run() da classe tAlCompiler. Neste procedimento, o caminho do arquivo de entrada é processado e inserido na lista de arquivos incluídos do compilador. Em seguida um objeto da classe tAlParser é criado. Este objeto será responsável pela tradução dirigida pela sintaxe do arquivo de entrada. Sempre que um novo arquivo for incluído, um novo objeto da classe tAlParser será instanciado para traduzir o novo arquivo.

A análise léxica do arquivo de entrada é realizada pelo método NextToken() da classe tAlParser, o qual retorna o código correspondente ao *token* encontrado. Um *token* é um

segmento de texto ou símbolos (conjunto de caracteres) que podem ser manipulados pelos parser e que possui um significado coletivo. Para cada *token* associamos um número inteiro que o identifica. Se o *token* for um caracter ASCII, o código será o próprio código ASCII do caracter. Para os demais *tokens* atribuímos um código sempre maior ou igual a 256. Além de retornar o *token*, o método `NextToken()` retorna os valores dos atributos sintetizados do item léxico reconhecido. Um *token* possui pelo menos um atributo sintetizado chamado *lexeme*. O *lexeme* de um *token* corresponde à cadeia de caracteres de entrada que define o *token*. *Tokens* podem ter outros atributos. Um **`float`**, por exemplo, possui como atributo o valor real obtido pela conversão do *lexeme* em ponto flutuante. `NextToken()` utiliza o atributo `lvalue` da classe `tAlParser` para retornar os valores dos atributos do *token* corrente. Caso um ou mais caracteres inválidos sejam encontrados no arquivo fonte, `NextToken()` emite um erro léxico e um método de manipulação de erros é invocado.

As análises sintática e semântica são feitas em conjunto. O analisador sintático-semântico descendente recursivo está implementado na classe `tAlParser`, a qual declara um método para cada não terminal da gramática da linguagem de animação. O procedimento de um não terminal é responsável por escolher a alternativa correta para a próxima derivação. Uma vez escolhida a alternativa, o método analisa cada símbolo da alternativa da seguinte forma:

- Se o símbolo for um não terminal, o procedimento associado a esse não terminal é invocado.
- Se o símbolo for terminal, verifica-se se o *token* corrente coincide com o terminal. Em caso afirmativo, o próximo *token* é processado e análise sintática continua. Caso contrário, o analisador emite um erro.

Terminada a tradução de um método ou função, seu código é mantido no atributo `Code` da instância de `tFunction`. Finalmente, o arquivo de código objeto é gerado.

3.8 Métodos Nativos

Como vimos na Seção 3.2.3, é possível o animador definir funções que estão implementadas em outra linguagem. Funções/métodos desse tipo são chamados de *métodos nativos*. Nesta seção abordamos uma característica do compilador no que diz respeito à utilização de métodos nativos, para isto, faz-se necessário a definição de *mangled name*.

3.8.1 Mangled Name

Em compiladores, *name mangling* consiste em uma técnica usada pare solucionar alguns problemas causados pela necessidade de se manter nomes únicos para entidades de um programa. Esta técnica fornece uma forma de embutir informações adicionais sobre o nome de uma função, classe, propriedades, etc.

No compilador do sistema, um símbolo da tabela de símbolos possui um nome que o identifica de maneira única. Este nome é o que chamamos de *mangled name*. No compilador do sistema, o *mangled name* de um símbolo é definido recursivamente a partir do seu nome em conjunto com o nome do escopo (espaço de nomes) em que foi definido. Por exemplo, considere o seguinte trecho de código:

```
class ClsA
{
  class ClsB
  {
```

```
      private:
         int valor;
      public:
         void metodo(float, ClsB);
         void metodo(char, bool);
   }
   int computa(int);
   void computa(float);
}
```

Nele definimos sete símbolos: duas classes, um atributo, e quatro métodos. O símbolo `ClsB` tem como nome a cadeia de caracteres formada pelas letras `'C'`, `'l'`, `'s'` e `'B'`. Tal símbolo é definido no escopo da classe `ClsA` (como vimos, toda classe também é um espaço de nomes). Dizemos então que o nome qualificado do símbolo `ClsB` é `ClsA::ClsB` e seu *mangled name* é formado pelos mesmos caracteres do seu nome qualificado, trocando-se apenas a seqüência `'::'` pelo caracter `'_'`, ou seja, o *mangled name* de `ClsB` é `ClsA_ClsB`. É fácil perceber, portanto, que o *mangled name* do símbolo `valor` é `ClsA_ClsB_valor`, uma vez que `valor` é definido no escopo `ClsB` que por sua vez é definido no escopo `ClsA`. Pelo fato do símbolo `ClsA` estar definido no escopo global, seu *mangled name* coincide com o seu nome, ou seja, `ClsA`.

Mangled Name de Métodos

Como mencionado, a linguagem de animação admite a definição de funções polimórficas, ou seja, funções com o mesmo nome mas com implementações diferentes. Por esta razão, o *mangled name* para métodos/funções obedece uma regra diferenciada, uma vez que não seria possível identificar tais funções de forma únivoca, como no caso dos métodos **int** `computa(`**int**`)` e **void** `computa(`**float**`)` da classe `ClsA`.

O *mangled name* de um método é tal como *mangled name* de qualquer outro símbolo, porém, a seqüência de caracteres que o define é formada pelos caracteres que representam o seu nome, seguido dos caracteres que definem os tipos de seus argumentos. No caso do método **void** `computa(`**float**`)`, seu *mangled name* é `ClsA_computa__f`, enquanto que **int** `computa(`**int**`)` é `ClsA_computa__i`. No caso dos tipos primitivos da linguagem, ao invés de usar a seqüência de caracteres `'f'`, `'l'`, `'o'`, `'a'` e `'t'` para o tipo **float**, por exemplo, fazemos uso apenas do primeiro caracter (no caso, `'f'`). No trecho de código, os *mangled names* para os métodos da classe `ClsB` são `ClsA_ClsB_metodo__f_ClsB` e `ClsA_ClsB_metodo__c_b`.

3.8.2 Implementando um Método Nativo

Além do arquivo objeto de animação (`.oaf`) gerado pelo compilador, este também é capaz de gerar arquivos de cabeçalho (`.h`) e arquivos fontes (`.cpp`). Nos arquivos de cabeçalho estão definidos protótipos de funções C++ sobre as quais o animador deverá escrever o corpo dos métodos nativos definidos na linguagem de animação. A implementação de um método nativo segue os seguintes passos:

Passo 1 O animador cria sua animação (`anima.scn`), opcionalmente adicionando métodos nativos. A definição de um método nativo, como vimos, é feita adicionando-se o modificador **native** ao final da definição do protótipo do método/função:

```
class X
{
   public:
      void metodo(float, int) native;
```

```
    ...
}
```

Passo 2 O arquivo de animação é compilado e como resultado deste processo, é gerado, além do arquivo objeto de animação (`anima.oaf`), um arquivo de cabeçalho (`anima.h`) e um arquivo fonte C++ (`anima.cpp`). No exemplo considerado, o conteúdo de `anima.h` seria:

```
1   #if !defined(__nativeinterface_h)
2   #include "nativeinterface.h"
3   #endif
4
5   extern "C" __declspec(dllexport)
6   void __stdcall X_metodo__f_i(IN* in, void* stack);
7   ...
```

Note que, na linha 6, definiu-se um protótipo de uma função cujo nome corresponde ao *mangled name* do respectivo método definido na linguagem de animação no passo 1. Como argumento, a função recebe um ponteiro para o componente `IN`, e um ponteiro para a pilha `MVL` da função. Estes parâmetros, bem como o arquivo de cabeçalho incluido na linha 2, são abordados em maiores detalhes no Capítulo 4.

Definido o protótipo da função, o corpo desta é escrito pelo animador. O conteúdo de `anima.cpp` seria:

```
1   #include "anima.h"
2
3   void __stdcall X_metodo__f_i(IN** in, void* stack)
4   {
5   //O código nativo é escrito aqui.
6   }
```

Passo 3 De posse dos arquivos de cabeçalho e fonte, cabe ao animador a tarefa de implementar o corpo do método nativo. Feito isto, cria-se uma biblioteca de ligação dinâmica utilizando qualquer ferramenta especifica para tal.

No exemplo considerado, vemos que o método `X::metodo` recebe dois parâmetros (um do tipo **`float`** e outro do tipo **`int`**). De que forma o animador utiliza tais parâmetros é um assunto abordado no Capítulo 4, quando da descrição do componente `IN` (**I***nterface* **N***ativa*).

3.9 Considerações Finais

A linguagem `LA` do sistema de animação é uma linguagem orientada a objetos que foi extendida a partir de uma linguagem de propósito geral chamada `L`. Os recursos de propósito geral de L conta com inclusão de arquivos, declaração de funções, variáveis, sobrecarga de operadores, manipulação de erros, classes (genéricas ou não), etc.

Uma cena animada é um container de luzes, câmeras, junções e atores. A geometria de uma ator é definida pelo seu modelo geométrico, uma instância da classe `Model` que define a pose, formas e dimensões do ator. Este modelo é utilizado pelo renderizador para sintetizar a aparência do ator. A representação das propriedades físicas de um ator é abstraída por um objeto da classe `RigidBody`. Este objeto é utilizado pelo motor de física para computar as restrições dinâmicas do ator. Um corpo rígido (objeto da classe `RigidBody`) mantém uma coleção de formas que correspondem a objetos de classes que derivam da classe abstrata

`Shape`. A forma de um corpo rígido é o meio pelo qual o motor computa o centro de massa e os pontos de contato entre atores.

Junções fornecem uma forma persistente de conectar dois atores. Uma junção é um objeto da classe derivada da classe abstrata `Joint`.

`LA` provê recursos sintáticos para definição e manipulação de componentes de uma cena animada, além do controle do fluxo de execução de uma animação por meio de seqüenciadores, sejam ações ou *scripts*. Ações são usadas para definir atividades que *continuamente* variam no tempo e devam ser executadas a cada ciclo de atualização da animação, enquanto *scripts* são usados para definir uma seqüência linear de atividades sincronizadas que devam ser executadas uma única vez, em um período de tempo conhecido. O sincronismo entre seqüenciadores pode ser feito por qualquer instância de classe que derive da classe abstrata `SyncObject`.

O compilador da linguagem implementa um *parser* do tipo descendente recursivo. Toma como entrada um arquivo de descrição de cena (`.scn`) e gera como saída um arquivo objeto de animação (`.oaf`), onde residem, dentre outras informações, as instruções que correspondem à linguagem alvo da máquina virtual.

A tabela de símbolos do compilador mantém todos os símbolos definidos na `LA`. Ela é implementada como uma floresta de árvores binárias balanceadas, onde cada elemento da árvore corresponde a um objeto de classe derivada da classe `tSymbol`.

O acoplamento de mensagens é seletivo. O compilador mantém uma tabela de ponteiros para métodos virtuais (`TPMV`) que mantém o endereço do método a ser executado em tempo de execução. Métodos podem ser nativos. Um método nativo é especificado por meio do modificador **`native`** aplicado à assinatura do método. Símbolos podem ser identificados por meio do seu *mangled name*. O *mangled name* de um método corresponde ao nome do respectivo método implementado na linguagem nativa.

CAPÍTULO 4

Executando uma Animação

4.1 Introdução

Neste capítulo tratamos da forma como uma animação é executada. Executar uma animação significa carregar em memória o arquivo objeto de animação, criar instâncias dos componentes do sistema e interpretar as intruções que definem a linguagem alvo do compilador. Como vimos na Seção 1.4, a MVL é o centro das atividades do sistema. Como uma máquina de computação real, possui um conjunto de instruções e manipula diferentes áreas da memória em tempo de execução.

A MVL deste trabalho foi criada tendo como fundação os conceitos da máquina virtual Java[24]. Acoplado a ela, porém, existe um componente capaz de controlar *scripts* e ações de uma animação, além disso, pode contar com operações/cálculos que envolvam dinâmica de corpos rígidos por meio de um motor de física integrado a ela. A arquitetura da MVL é abordada na Seção 4.2, onde vemos os componentes que representam em software uma máquina de computação real. Na Seção 4.3 introduzimos suas instruções e o mecanismo de execução de funções/métodos. Em seguida, na Seção 4.4, tratamos da memória de objetos lixo-coletável e dos mecanismos de alocação e liberação de objetos. A estratégia adotada na coleta de lixo é vista na Seção 4.5. A integrãção do motor de física à MVL é abordada na Seção 4.6, onde comentamos também a biblioteca nativa de animação, necessária à compreensão da forma como o motor foi integrado à MVL. Na Seção 4.7, abordamos a gerência da execução de *scripts* e ações.

4.2 A Arquitetura da MVL

Uma característica da MVL é que sua arquitetura é independente da linguagem de animação (alto nível), ou seja, modificações na linguagem L e suas extensões não influenciam na maneira como uma animação é executada, porém, é dependente da linguagem alvo (instruções, baixo nível). Os componentes que definem esta arquitetura relacionam-se segundo o diagrama UML da Figura 4.1.

A máquina virtual de L é uma instância de tMachine. Esta instância mantém uma referência para um interpretador de instruções, representado por uma instância da classe tProcessor, que em parceria com o controlador (instância de tController), gerencia o escalonamento e execução de *scripts* e ações. Para isto, o interpretador faz uso de outros

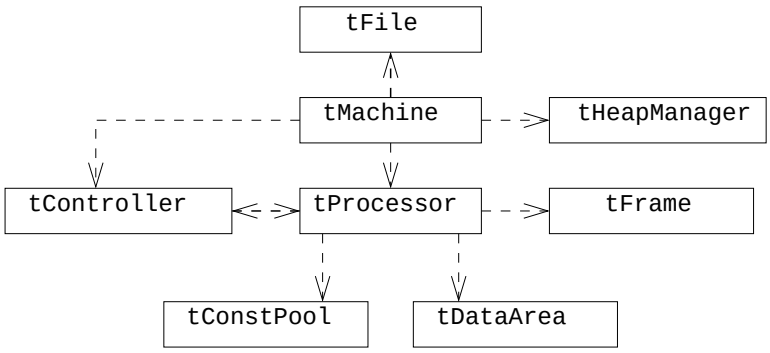


Figura 4.1: Arquitetura da MVL.

componentes, instâncias de tConstPool, tDataArea e tFrame, que são abordados ao longo desta seção. O controlador, em particular, é descrito mais detalhadamente na Seção 4.7.

Os componentes cujo interpretador faz uso mantém uma estreita relação com os tipos de dados representados na MVL.

Representação dos Tipos

Como na linguagem de animação, a MVL opera sobre duas categorias de tipos: os tipos primitivos e os tipos referência. Elas correspondem aos dois tipos de valores que podem ser armazenados em variáveis, passados como argumento ou retornados por métodos/funções. Um tipo de dado é representado por um objeto da classe tVMClass, e os tipos primitivos são representados por instâncias especializadas de tPrimitiveType, derivada de tVMClass. A Figura 4.2 ilustra o diagrama UML das classes de tipos.

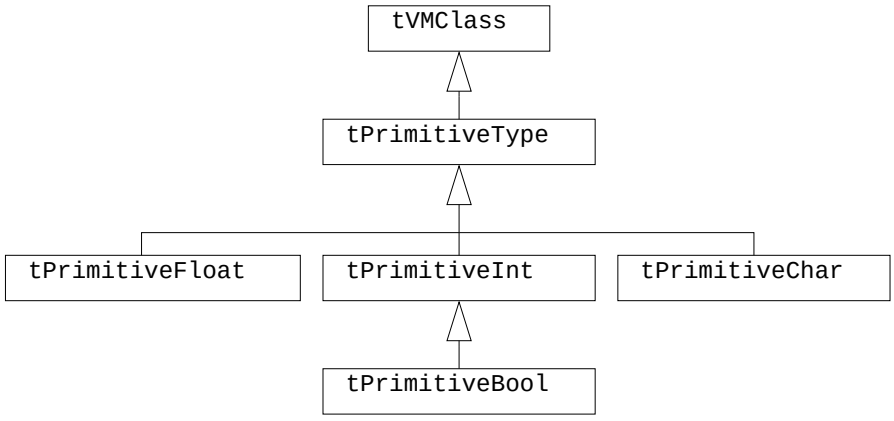


Figura 4.2: Representação dos tipos da MVL.

O tipo *caracter* representa um valor de 16 bits (UNICODE), uma instância da classe tPrimitiveChar. O tipo *inteiro* representa um número inteiro com sinal na faixa de -2^{31} a 2^{31}, instância de tPrimitiveInt. O tipo *boleano* (**bool**) definido na linguagem fonte é representado por um inteiro (ocupando 4 bytes) igual a 0 (**false**) ou 1 (**true**), instância de tPrimitiveBool, derivada de tPrimitiveInt. O tipo *real* representa um número em ponto flutuante de precisão dupla padrão IEEE, ocupando 8 bytes, instância de tPrimitiveFloat. O tipo *endereço* representa um número de 4 bytes sem sinal que armazena a referência para um objeto mantido na memória de objetos — um tipo de dado, instância de tVMClass.

4.2.1 Área de Dados

Como vimos no Capítulo 3, a linguagem de animação nos permite definir variáveis globais e variáveis estáticas de classe (atributos de classe). Do ponto de vista da MVL, não existe distinção entre variáveis globais e estáticas definidas no programa fonte. A arquitetura simplesmente mantém uma única região de memória específica para manter tais variáveis. Esta região é *área de dados* e é representada por um objeto da classe tDataArea.

Como qualquer variável do compilador (instância de tVariable), a representação de uma variável na MVL mantém como principal atributo o seu deslocamento em relação ao início da região de memória onde reside. Este deslocamento, chamado offset, calculado pelo compilador, faz parte das informações mantidas no arquivo objeto de animação (instância de tFile) e depende do tipo da variável em questão (o tipo determina o tamanho). A quantidade de memória destinada à área de dados, portanto, é igual à soma dos tamanhos dos tipos das variáveis globais e estáticas definidas no programa fonte.

4.2.2 Pool de Constantes

Em tempo de execução, como veremos, far-se-á necessário conhecer informações dos símbolos definidos na linguagem de animação. Por exemplo, inforamções a respeito dos nomes globais e estáticos, o número e tipo dos argumentos de uma função/método, sua assinatura, o offset de uma variável, dentre outras.

O *pool de constantes* é um contêiner de símbolos onde cada entrada guarda um símbolo compacto proveniente da tabela de símbolos do compilador[1]. Diferentemente daquela tabela de símbolos (representada por uma floresta de árvores binárias balanceadas), porém, o *pool* de constantes é um arranjo unidimensional de tamanho fixo. A Figura 4.3 ilustra o diagrama UML das classes de objetos que se relacionam com este componente.

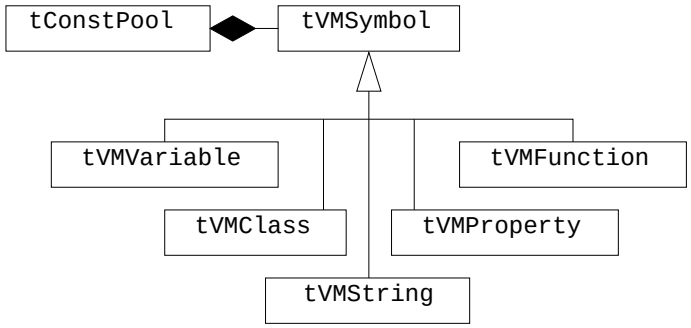


Figura 4.3: Classes do *pool* de constantes.

O *pool de constantes* é representado por uma instância de tConstPool. Esta instância conhece o formato do arquivo objeto de animação e é a responsável por sua carga pela MVL. Durante o processo de carga, as informações compactas referentes aos símbolos da linguagem são usadas para instanciar novos símbolos. Instanciados tais símbolos, estes são mantidos no contêiner e ganham, cada um, um número de 32 bits de forma a identificá-lo de maneira única. Ao final da carga, este número coincidirá com o índice da sua entrada no contêiner; como veremos, isto é útil no mecanismo de RTTI (**R**un**T**ime **T**ype **I**nformation) da MVL.

Um símbolo genérico mantido no *pool* de constantes é representado pela classe tVMSymbol.

[1] O termo "símbolo compacto"empregado é devido ao fato de que nem todas as informações inerentes a um símbolo, proveniente da tabela de símbolos do compilador, estarão presentes no respectivo símbolo instanciado pela MVL.

No contêiner, no entanto, são mantidos apenas instâncias especializadas desta classe. Um símbolo é capaz de responder qual o seu nome, modificadores e a classe na qual foi declarado.

Uma classe, como vimos na Seção 4.2, representa um tipo de dado. Uma instância de `tVMClass` é capaz de responder quais os seus atributos, métodos e propriedades definidos no programa fonte.

A representação de uma variável ou atributo é feita por uma instância de `tVMVariable` e mantém como informações o seu deslocamento `offset` e o seu tipo.

A classe `tVMFunction` representa uma função ou método a ser executada(o). Objetos desta classe possuem `code` como principal atributo (instância de `tCode`). O atributo `code` representa os *bytecodes* da função ou método a serem interpretados. Os *bytecodes* definem as instruções geradas pelo compilador; um byte por instrução.

A classe `tVMProperty` representa uma propriedade. Os principais atributos desta classe são os símbolos que definem os especificadores de leitura e escrita (**`read`** e **`write`**).

A classe `tVMString` representa uma cadeia de caracteres definida no programa fonte. Instâncias dessa classe representam também as *strings* literais.

4.2.3 Pilha de Frames

Frames de execução são as estruturas de dados (objetos da classe `tFrame`) da `MVL` utilizadas para armazenar o estado interno atual de um método ou função invocado(a) durante a execução de uma animação. A Figura 4.4 ilustra os tipos de *frames* instanciados na `MVL`[2].

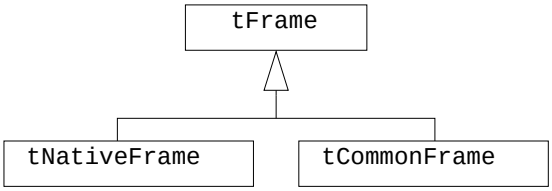


Figura 4.4: Tipos de *frame*.

Um *frame comum* é uma instância da classe `tCommonFrame` — o *frame* destinado à execução de métodos/funções cujas instruções correspondem às instruções da `MVL`. Um *frame nativo* é uma instância de `tNativeFrame` e corresponde ao *frame* decorrente da invocação de um método/função nativo(a), cujas instruções estão definidas numa biblioteca de ligação dinâmica.

O formato de um *frame comum* é mostrado na Figura 4.5 e é composto de:

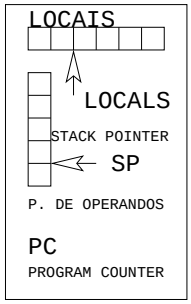


Figura 4.5: O formato de um *frame* comum.

[2] A `MVL` instancia apenas especializações da classe `tFrame`.

- **Área de variáveis locais**. É um arranjo unidimensional contendo os argumentos e as variáveis locais da função. Cada elemento do arranjo possui o tamanho de um número inteiro da MVL, ou seja, 32 bits. Valores do tipo inteiro, endereço e caracter ocupam um elemento do arranjo, enquanto valores do tipo **`float`** ocupam duas entradas consecutivas. Esse arranjo será denominado de LOCAIS. Cada entrada de LOCAIS é identificada por um número a partir de 0 (índice do arranjo). O número máximo de variáveis locais de um *frame* é 256. O número de entradas de LOCAIS é determinado em tempo de compilação e faz parte das informações mantidas no arquivo objeto.

- **Pilha de operandos**. É um arranjo unidimensional. Como LOCAIS, cada entrada da pilha de operandos armazena um inteiro da MVL. O número de entradas da pilha também é determinado em tempo de compilação. A pilha de operandos mantém os resultados temporários de todas as expressões efetuadas no código da função ou método corrententemente executada(o). Durante a execução das instruções de uma animação, os operandos e os valores de retorno dessas instruções são passados a esta pilha. A pilha de operandos é acessada através de duas instruções:

  - `push`: Esta instrução empilha um valor de 32 bits passado como parâmetro.
  - `pop`: Esta instrução desempilha e retorna, na variável passada como parâmetro, um valor de 32 bits.

- **Contador de programa**. O contador de programa é um inteiro de 32 bits que indexa, em relação ao início da área de código da função associada ao *frame*, a próxima instrução a ser executada pela MVL.

O gerenciamento de *frames* ocorre por meio de uma pilha de *frames*. Esta pilha corresponde à pilha de referências para os *frames* da cadeia de execução das funções/métodos da animação. Chamaremos de *frame corrente* o *frame* cuja referência encontra-se no topo da pilha de *frames*.

A execução de *frames* pela MVL é mostrada na Figura 4.6. Executar um *frame* significa interpretar as instruções da função a qual ele representa.

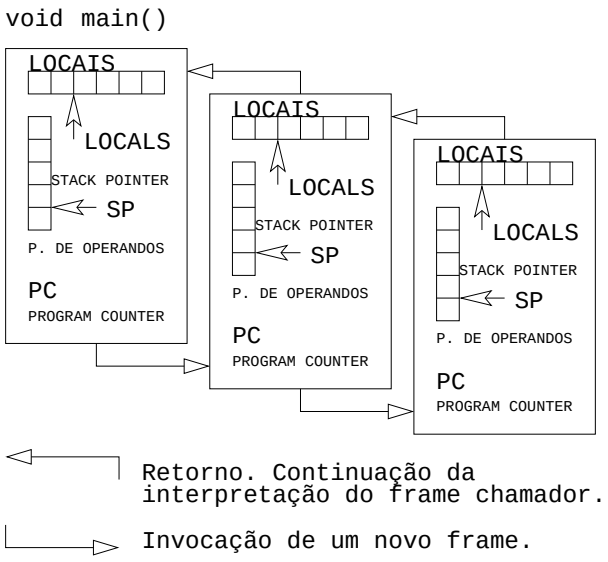


Figura 4.6: Execução de *frame*.

Durante a execução de uma animação, sempre que uma função ou método for executada(o), é criado um novo *frame*. Após a criação do *frame*, a MVL salva o *contexto* corrente

do processador virtual (o interpretador de *bytecode*) e o *frame* criado passa a ser o *frame* corrente, sendo colocado no topo da pilha de *frames*. O processador passa então a executar o *frame* corrente até que a última instrução deste seja interpretada. Em seguida, o contexto anteriormente salvo é restaurado e o *frame* corrente é retirado do topo da pilha de *frames* (logo, o *frame* anterior passa a ser o corrente). Toda animação possui pelo menos um *frame* que representa a sua função principal; a `main()` da especificação da animação. A animação termina quando a execução deste *frame* termina.

A representação de um *frame* nativo pela `MVL` difere daquela representação de um *frame* comum. A região de variáveis locais e a pilha de operandos de um *frame* nativo correspondem às variáveis locais e pilhas implícitas existentes quando da execução de uma função/método escrita(o) em uma linguagem nativa, tal como C/C++. Apesar disso, a classe `tNativeFrame` mantém métodos que ocultam o fato da(o) função/método que o *frame* representa não possuir instruções da `MVL`.

4.2.4 Processador e Registradores

O processador virtual da `MVL` é abstraído por uma instância da classe `tProcessor`. Esta instância é quem de fato interpreta cada uma das instruções definidas na linguagem alvo do compilador. Para cada instrução é associado um método de `tProcessor` que a representa. O processador virtual é utilizado pelo controlador como uma ferramenta de execução de *scripts* e ações. Além do papel de interpretador, o processador virtual, como veremos, é o responsável pela indicação do momento oportuno em que se deve buscar por objetos não utilizados (coleta de lixo). A coleta de lixo, todavia, é feita por outro componente do sistema (o gerenciador de memória, abordado na Seção 4.4) mas que faz uso do processador virtual quando da finalização de objetos. O processador da `MVL` conta com os seguintes registradores:

- **PC**. É o registrador que mantém o endereço da próxima instrução a ser executada. Sempre que um novo *frame* é criado pela `MVL`, o valor de `PC` é salvo no *frame* corrente e recuperado quando a função associada ao novo *frame* retorna; se esta for não nativa.
- **SP**. É o registrador que mantém um ponteiro de 32 bits que aponta para o topo da pilha de operandos do *frame* corrente; *frame* este associado a uma função não nativa. Sempre que um novo *frame* é criado pela `MVL`, o valor de `SP` é salvo no *frame* corrente e recuperado quando a função associada ao novo *frame* retorna.
- **LOCALS**. É o registrador que mantém um ponteiro de 32 bits que aponta para a área de variáveis locais do *frame* corrente.
- **FP**. É o registrador que mantém um ponteiro que guarda o endereço do *frame* correspondente à função corrente, sendo esta nativa ou não.
- **DATA**. É o registrador que mantém o endereço da área de dados do programa a ser executado.

4.3 Instruções

Cada instrução da `MVL` é definida por um código de oito bits chamado *bytecode*. Além do *bytecode*, uma instrução pode conter parâmetros de 8, 16, 32, ou 64 bits, os quais sucedem o *bytecode* no código objeto. O processamento de uma instrução começa com a obtenção, no código objeto da função corrente, do *bytecode* da próxima instrução. O processador, então, decodifica o *bytecode* e executa um método de `tProcessor` que implementa a instrução, o

qual é responsável pela obtenção, no código objeto, de seus parâmetros (se existirem). As instruções da MVL podem ser distribuídas nos seguintes grupos:

- Instruções aritméticas (iadd, isub, fmul).
- Instruções lógicas (iand, ior).
- Instruções de desvio condicional (ifeq, ifgt, fcmp).
- Instruções de conversão de dados (i2f, f2i).
- Instruções de transferência de dados (istore, fload, ipush, fpop).
- Instruções de desvio incondicional (invokevirtual, freturn, goto).
- Instruções de criação de objetos (new, start).

A descrição completa do conjunto de instruções da MVL, bem como a configuração da pilha de operandos para cada uma das instruções é feita no Apêndice B.

4.3.1 Invocação de Métodos

As informações contidas no *pool* de constantes da MVL relativas a uma classe inclue a TPMV da classe. Um objeto que reside na memória de objetos da MVL possui espaço para armazenamento de todos os atributos que definem seu estado (atributos declarados e herdados), além de um ponteiro que mantém o endereço da classe do objeto, situada em alguma entrada do *pool* de constantes. Em outras palavras, todo objeto da MVL "conhece" sua classe. Isto é útil para o mecanismo de RTTI para invocação de métodos virtuais. Na invocação de métodos, duas instruções em especial merecem atenção. São elas:

- invoke. Esta instrução possui um argumento arg de 4 bytes que promove a chamada de métodos de instância. Este argumento é o índice (ou deslocamento da entrada do *pool* de constantes) que contém as informações a respeito do método a ser invocado (instância de tVMFunction). A instrução efetua os seguintes passos:

  Passo 1 As informações relativas ao número de argumentos, tamanho de LOCAIS e o tamanho da pilha de operandos do método a ser chamado são recuparadas da entrada indexada por arg no *pool* de constantes.

  Passo 2 Estas informações são utilizadas na criação de um novo *frame* para o método a ser invocado.

  Passo 3 Os argumentos do método a ser invocado são retirados da pilha de operandos do método invocador e armazenados nas primeiras entradas de LOCALS do novo *frame* criado.

  Passo 4 O registrador FP do processador é armazenado no novo *frame* criado.

  Passo 5 O registrador PC do processador é armazenado no novo *frame* criado.

  Passo 6 O novo *frame* recebe o contexto do processador (contexto pai, ou contexto do *frame* invocador que representa a função/método chamador).

  Passo 7 O *frame* é executado.

- invokevirtual. Esta instrução possui como argumento arg um número de 2 bytes igual ao índice da entrada da TPMV do método a ser invocado. A TPMV contendo o endereço do método é a TPMV da classe de **this** (o ponteiro implícito de um objeto), o qual deve estar no topo da pilha de operandos. A instrução invokevirtual efetua as seguintes operações:

Passo 1 Desempilha o endereço de **this** do topo da pilha de operandos do método chamador. A partir do objeto **this**, as informações da classe de **this** são recuperadas do *pool* de constantes da MVL. Isto é possível porque, como mencionado, todo objeto mantém um ponteiro para sua classe. As informações da classe de **this** que endereçam para invokevirtual consistem da TPMV da classe.

Passo 2 Uma vez obtido o endereço da TPMV da classe de **this**, o índice da entrada do *pool* de constantes contendo as informações do método a ser invocado é obtido da arg-ésima entrada da TPMV.

Passo 3 Obtidas as informações a respeito do método a ser invocado, são executados os passos da instrução invoke.

4.4 Memória de Objetos

Nesta seção descrevemos a memória de objetos lixo-coletável da MVL. A memória de objetos é a região de memória onde residirão todos os objetos instanciados pelo sistema de animação. O gerenciamento de memória pela MVL é feito por uma instância da classe tHeapManager. Inicialmente abordamos alguns conceitos básicos para compreensão da memória de objetos, que inclui páginas de memória, o *heap* do sistema e blocos de alocação.

Página de Memória

Uma página de memória é a menor porção de dados que pode ser lido ou escrito na memória de objetos. O tamanho de uma página de memória influencia consideravelmente no desempenho da gerência de memória, determinando o número e a velocidade de acessos de leitura e escrita. Na MVL, o tamanho das páginas de memória é 8KB (algumas implementações da máquina virtual Java utilizam este valor). Nos referiremos a este tamanho utilizando o rótulo PageSize.

Heap

O *heap* é a coleção de páginas de memória alocadas na memória principal do computador e que armazenam tanto objetos da LA como objetos que representam componentes internos do sistema. Criado na inicialização da MVL, o tamanho inicial do *heap* é 5MB, ou seja, 512 páginas. Não necessariamente contínuo, o *heap* pode ser expandido e chegar até o limite de 64MB. A expansão do *heap* é feita a cada 1MB[3]. A destruição do *heap*, porém, ocorre somente na finalização da MVL. As principais variáveis que controlam o *heap* são:

- HeapBase. É o apontador que referencia o menor endereço de página do *heap*.
- HeapRange. É o apontador que referencia o maior endereço de página do *heap*.
- HeapInitialSize. Mantém a quantidade inicial de memória do *heap* (5MB).
- HeapIncrement. Mantém a quantidade de memória de crescimento do *heap* (1MB).
- HeapLimit. Mantém a quantidade máxima de memória que o *heap* pode suportar.

[3]O tamanho inicial, limite e incremento do *heap* são parametrizados na MVA.

Blocos de Alocação

O gerenciamento do *heap* da `MVL` dá-se pelo mapeamento das páginas de memória em blocos denominados *blocos de alocação*. Um *bloco de alocação* mantém referências para objetos que residem na página de memória por ele mapeada. O mapeamento das páginas de memória baseia-se em dois tipos de blocos:

- **Blocos de objetos pequenos.** São blocos de alocação que referenciam objetos de tamanho menor que um determinado limite (4KB foi assumido). Estes blocos mapeiam `PageSize` bytes[4], portanto, o número de objetos referenciados por blocos deste tipo varia de acordo com o tamanho destes objetos.
- **Blocos de objetos grandes.** São blocos de alocação que armazenam objetos com tamanho maior que o limite. Estes blocos, todavia, referenciam um único objeto apenas. O tamanho deste objeto corresponde ao primeiro múltiplo de `PageSize` maior ou igual ao tamanho do objeto armazenado.

4.4.1 Mapeamento de Blocos

Nesta seção tratamos do mapeamento de blocos. Inicialmente considere as principais informações mantidas em um bloco de alocação:

- `SizeObj`. Mantém o tamanho dos objetos referenciados pelo bloco de alocação.
- `Avail`. Mantém o número de referências a objetos disponíveis para alocação pelo bloco.
- `Data`. Mantém a referência para o primeiro objeto do conjunto de referências do bloco.
- `NextBlock`. Mantém um endereço para o próximo bloco quando mantido em uma lista de blocos.

A quantidade de referências para objetos mantidas em um bloco de alocação classifica-o da seguinte forma:

- **Em uso.** São blocos de alocação que possuem pelo menos uma referência a um objeto interno já alocado.
- **Fora de uso.** São blocos de alocação que não possuem referências a objetos internos alocados.
- **Livre.** São blocos de alocação que possuem referências a objetos internos não alocados.
- **Cheio.** São blocos de alocação que já atingiram o número máximo de referências a objetos internos alocados.
- **Não alocado.** São blocos de alocação que não mapeia página alguma do *heap*.

A Figura 4.7 ilustra o mapeamento das páginas de memória em blocos de alocação. A coleção de blocos de alocação é representada por uma tabela chamada `Tabela de Alocação`. Cada entrada da tabela, portanto, mantém um bloco de alocação que mapeia uma página de memória.

Note que a tabela de alocação indexa todo o intervalo de páginas do *heap*. Desta maneira, podemos assumir que o gerenciador atua sobre um *heap* contínuo com regiões inacessíveis que

[4]Dizemos que o mapeamento é feito em um bloco por página.

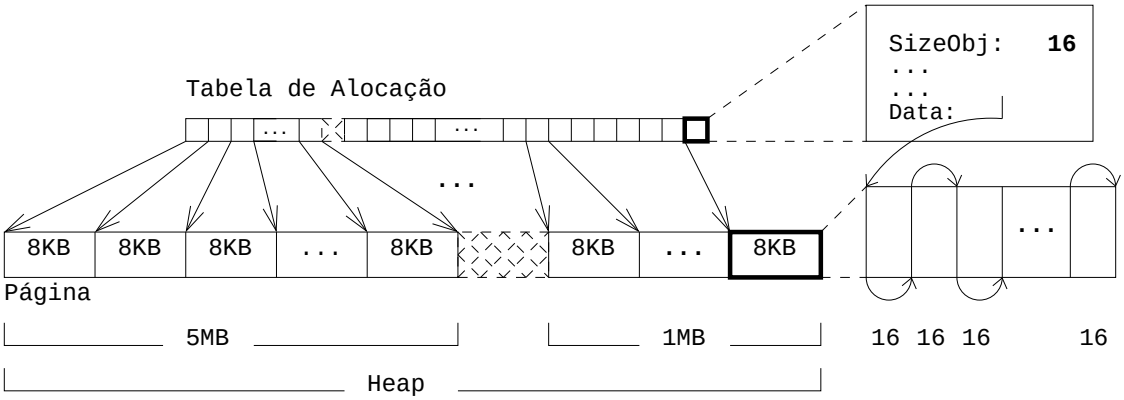


Figura 4.7: Mapeamento das páginas em blocos.

correspondem às regiões da memória principal que não foram alocadas para o *heap* mas que possuem entradas na tabela de alocação onde os blocos que as mapeiam são classificados como "não alocado".

O gerenciador do *heap* mantém uma tabela de espalhamento, denominada `FreeList` (Figura 4.8), que mantém os blocos de alocação livres. Em adição, mantém uma lista de blocos de alocação fora de uso, denominada `PrimitiveFreeList`, (Figura 4.9(a)).

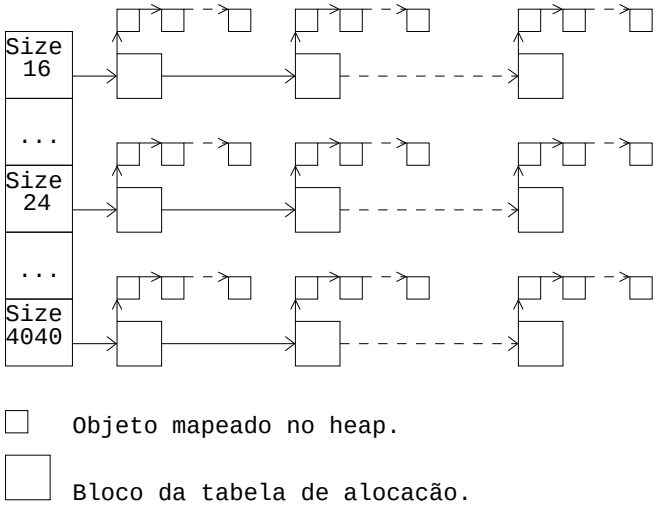


Figura 4.8: Tabela de espalhamento `FreeList`.

É possível determinar, no momento da alocação de um objeto de tamanho t, qual o bloco de alocação correto para manter este objeto, ou se será preciso criar um novo bloco de alocacação para objetos de tamanho t, ou ainda, se será necessário que ocorra uma expansão do *heap* para acomodar o novo objeto. O gerenciamento de blocos consiste em operações de inserções e remoções naquelas estruturas de dados, preparando blocos de `PrimitiveFreeList` para serem mantidos em `FreeList` e vice-versa. Estas operações são a base para o mecanismo de alocação e liberação de objetos.

4.4.2 Alocação de Objetos

Invocada toda vez que um novo objeto é criado durante a execução de uma animação, a atividade de alocação de objetos consiste em armazenar na memória de objetos, se possível, um novo objeto. Gerenciada pela instância de `tHeapManager`, a alocação segue os passos:

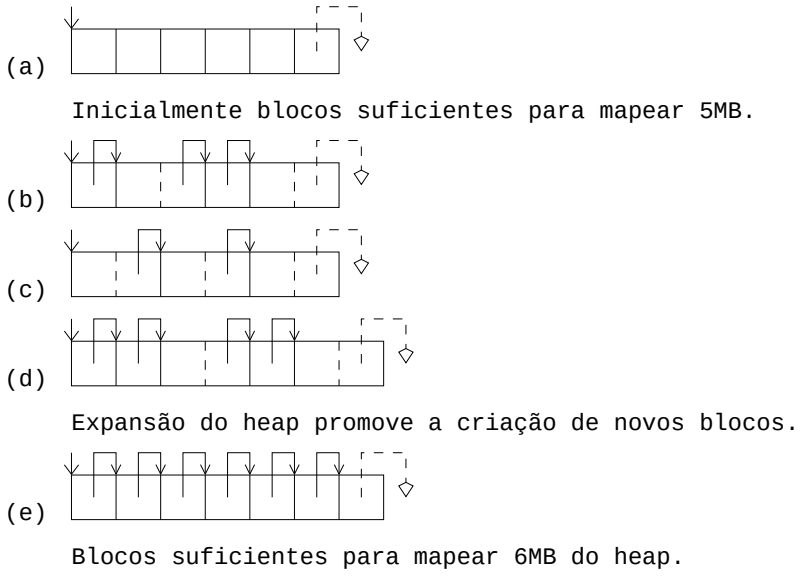


Figura 4.9: Lista de blocos fora de uso.

Passo 1 Após determinado o tipo, e conseqüentemente o tamanho t do objeto a ser alocado, faz-se uma busca em `FreeList` por um bloco de alocação que possua `SizeObj` igual a t (ou o menor tamanho maior ou igual a t). Encontrando-se este bloco, a memória destinada ao objeto é recuperada e o estado interno do bloco é atualizado. Não encontrado o bloco, prossegue-se.

Passo 2 Não encontrado o bloco de alocação na `FreeList`, faz-se uma busca na lista de blocos fora de uso (`PrimitiveFreeList`) pelo primeiro bloco (*first-fit*) de tamanho maior ou igual a t. Encontrado este bloco, é transferido da `PrimitiveFreeList` para a `FreeList`, seu estado interno é atualizado (bloco "livre") e retorna-se ao primeiro passo. Não encontrado um bloco na `PrimitiveFreeList`, prossegue-se.

Passo 3 Caso o gerenciador não puder fornecer nenhum tipo de bloco para alocação, este requisita a expansão do *heap*, Figura 4.9(d), que consiste em solicitar mais 1MB (128 páginas) ao sistema operacional. Se bem sucedido, retorna-se ao segundo passo, uma vez que novos blocos foram inseridos na `PrimitiveFreeList`. Se a expansão do *heap* não puder ser feita, por ter atingido seu tamanho limite ou por não haver memória livre, prossegue-se no passo seguinte.

Passo 4 Neste momento invoca-se a atividade de coleta de lixo com o objetivo de liberar o maior número possível de objetos desnecessários à execução da animação. Se durante a coleta ocorrer alguma liberação, retorna-se ao primeiro passo; caso contrário, é lançada uma exceção do tipo `OutOfMemory` (falta de memória).

4.4.3 Liberação de Objetos

A atividade de liberação de objetos consiste em devolver ao *heap* um objeto antes "em uso", aumentando o espaço livre no *heap*. A liberação de objetos segue os seguintes passos:

Passo 1 Após determinado o endereço do objeto e do bloco de alocação que o contém, o objeto passa a ser "livre", diminuindo, portanto, o número de objetos "em uso" do bloco.

Passo 2 Verifica-se o número de objetos "em uso" do bloco; caso este seja zero, o bloco de alocação é marcado como "fora de uso" e tem o seu endereço inserido na lista de blocos fora de uso (`PrimitiveFreeList`). Se existirem blocos adjacentes, anteriores e/ou posteriores, esses blocos são unidos com o propósito de reduzir a fragmentação da memória de objetos, Figura 4.9 (b) e (c), e sua entrada passa a ser única em `PrimitiveFreeList`. No caso em que houver apenas um objeto "livre", o bloco de alocação, antes "cheio", é adicionado em `FreeList`.

A liberação de objetos não reduz o tamanho do *heap*, apenas transformando objetos "em uso" em objetos "livres" (segundo a classificação dos blocos) sem realizar qualquer liberação de memória pelo sistema operacional.

4.5 Coleta de Lixo

Durante a alocação de objetos podemos nos deparar com a falta de memória. Neste instante, a atividade de coleta de lixo é invocada com o propósito de devolver à memória de objetos todo objeto inacessível durante o progresso de uma animação. A forma de como ocorre a coleta de lixo envolve os conceitos de *objetos raiz* e um *esquema de cores*.

Objetos Raiz

A determinação da acessibilidade de um objeto é feita buscando-se por referências feitas ao seu endereço. A este processo de busca dá-se o nome de *marcação de objetos*. Não encontrado uma referência a um objeto, dizemos que este deve ser finalizado e em seguida *coletado*. Finalizar um objeto significa executar as instruções definidas no método **`destructor`** da classe do objeto.

Uma referência a um objeto pode ocorrer na pilha de operandos, e/ou na área de variáveis locais de algum *frame* da cadeia de execução, e/ou na área de variáveis estáticas e globais (área de dados). Estas referências apontam para os chamados *objetos raiz*.

Esquema de Cores

A partir dos objetos raiz tem-se de fato o início da coleta de lixo. Esta coleta faz uso da estratégia *mark and sweep* [41] que utiliza um esquema de cores representadas por listas circulares que mantêm referências para objetos criados quando da execução de uma animação. São elas:

- **Lista de objetos brancos**. Mantém os objetos assim que criados.
- **Lista de objetos cinzas**. Mantém os objetos alcançados pela busca, diretamente como um objeto raiz, ou por referência indireta (como atributo de um outro objeto). Basicamente, objetos cinzas correspondem aos objetos seguramente utilizados, ou seja, objetos que não podem ser coletados naquele instante.
- **Lista de objetos pretos**. Mantém os objetos completamente atravessados, isto é, alcançados pela busca e com todas as suas referências para outros objetos (atributos) alcançadas.

Colorir/Pintar um objeto significa ajustar uma *flag* de cor do objeto, removê-lo da lista onde se encontra e colocá-lo na lista de objetos que representa a cor desejada. Além das listas de objetos coloridos, duas outras listas são utilizadas:

- **Lista de objetos finais**. Mantém os objetos que aguardam o momento de serem finalizados.
- **Lista de objetos prontos**. Mantém os objetos que estão prontos para serem devolvidos ao *heap* da MVL.

4.5.1 O Algoritmo

O algoritmo de coleta de lixo consiste basicamente em alternar os objetos entre as listas do esquema de cores. Ele segue os seguintes passos:

Passo 1 Iteraja na lista de objetos finais. Se um objeto da lista tem a cor branca, pinte-o com a cor cinza. (Ou seja, objetos só podem ser coletados se antes forem finalizados. Ganham, portanto, a cor cinza indicando que naquele instante não podem ser devolvidos ao *heap*).

Passo 2 Atravesse os *frames* da cadeia de execução (pilhas de operandos e variáveis locais) e a área de dados, colorindo com a cor cinza os objetos encontrados.

Passo 3 Os passos anteriores podem ter promovido a inserção de objetos na lista de objetos cinzas. Até este ponto, objetos desta cor correspondem aos objetos raiz. Neste passo, pinte estes objetos com a cor preta e atravesse (varredura do objeto) tais objetos em busca de referências indiretas (objetos como atributos). Se encontrado novos objetos, recursivamente pinte-os com a cor cinza e repita este passo até que não haja objetos nesta lista (todos os objetos varridos/atravessados).

Passo 4 Neste ponto os objetos completamente atravessados terão a cor preta. Note que os objetos brancos, se existirem, podem ser objetos que precisam ser finalizados. Se este for o caso, sinalize-os com o estado INFINALIZE e pinte-os com a cor cinza.

Passo 5 Execute o passo 3 e prossiga no passo 6. (Uma vez que novos objetos podem ter sido coloridos com a cor cinza no passo anterior, serão, então, varridos novamente).

Passo 6 Neste ponto, qualquer objeto branco é inalcançável, ou seja, são objetos que podem ser devolvidos ao *heap*. Neste passo, mova os objetos brancos (lista de objetos brancos) para a lista de objetos prontos.

Passo 7 Iteraja na lista de objetos pretos. Se um objeto da lista é sinalizado com o estado INFINALIZE, insira-o na lista de objetos finais, caso contrário, pinte-o com a cor branca.

Passo 8 Iteraja na lista de objetos finais colorindo-os com a cor branca e sinalizando-os com o estado FINALIZED, além disso, execute o método **`destructor`** da classe do objeto e devolva-o ao *heap* da MVL. Note que neste ponto fecha-se o ciclo de cores, ou seja, objetos que inicialmente eram brancos, voltaram a ganhar a cor branca.

No processo de marcação de objetos podemos nos deparar com valores de 32 bits que residem na pilha, na área de variáveis locais e/ou na área de dados que casualmente coincidam com valores que representam endereços do *heap*, indicando, desta forma, a possibilidade de alcançar uma "instância de classe" que na realidade não corresponde a um objetos alocado. Isto pode acontecer em decorrência da possibilidade de se realizar uma infinidade de operações matemáticas feitas sobre a pilha, cujos resultados sejam mantidos nela, na área de variáveis locais ou área de dados. Identificar se aqueles valores são de fato endereços de objetos alocados é computacionalmente inviável, porém, o algoritmo garante que nenhum objeto correntemente utilizado seja devolvido ao *heap*.

A Decisão de Coletar

Como mencionado, a instância de `tHeapManager` representa o gerenciador da memória de objetos e da coleta de lixo. Quando da solicitação de memória ao gerenciador, este primeiramente checa se existe alguma memória livre no *heap* cujo tamanho corresponde à quantidade solicitada. Se houver, esta memória é retornada. Isto significa que o gerenciador nem sempre invocará a coleta de lixo até que utilize mais memória que o tamanho inicial do *heap*, cujo padrão é 5MB. Desta forma, a coleta de lixo pode ser evitada se uma animação não requisitar mais memória que o tamanho inicial do *heap* durante todo o seu tempo de vida.

Se não houver qualquer memória livre no *heap*, uma tentativa de alocação fará com que o gerenciador ative a coleta de lixo. O gerenciador utiliza uma eurística para decidir se deve executar a coleta ou se deve adiá-la. Adiar uma coleta de lixo significa optar pela requisição de mais memória ao sistema operacional (crescendo o *heap* da `MVL`) ao invés de buscar por memória não mais utilizada. Toda vez que ele decide pelo adiamento da coleta de lixo, o gerenciador requisitará por uma quantidade de memória igual a 1MB. Obviamente, o adiamento da coleta é uma opção se o tamanho corrente do *heap* é menor que o tamanho máximo permitido (64MB). Se o tamanho máximo já tiver sido alcançado, a coleta de lixo é executada, caso contrário, cabe ao gerenciador a tarefa de decidir se deve aumentar o *heap* ou se deve ativar a coleta.

Esta decisão trata-se de um problema clássico de concessão entre espaço e tempo. Consideremos os dois casos extremos: primeiramente, suponhamos que o gerenciador adie as coletas até que a animação utilize a quantidade máxima de memória do *heap*. Isto significa que coletas ocorrerão menos freqüentemente, além do que, a maioria das animações de tempo de vida longo realmente necessitarão de toda a quantidade de memória disponibilizada pelo *heap*. O adiamento das coletas, neste caso, é visto como uma forma a utilizar a memória em favor da velocidade. Por outro lado, se o gerenciador ativar a coleta de lixo sempre que requisitado a fornecer memória, coletas ocorrerão mais vezes, porém, nem sempre a utilização de memória ultrapassará a quantidade de memória destinada aos dados de vida longa de uma animação. O não adiamento das coletas, neste caso, é visto como uma forma de abrir mão da velocidade em favor da quantidade de memória utilizada.

Em um exemplo hipotético, suponha que uma animação utilize 16MB de dados de vida longa e produza 160MB de dados de vida curta. Suponha também que o tamanho inicial do *heap* seja 5MB, o tamanho máximo seja de 64MB e que o *heap* cresça de 1MB. Se o gerenciador sempre ativar a coleta de lixo, então esta animação nem sempre usará mais que 17MB, porém executará 160/(17-16) = 160 coletas. Por outro lado, se disponibilizássemos toda a memória disponível, então a animação usaria 64MB, mas executaria somente 160/(64-16) = 4 coletas.

Poderíamos imaginar que o custo de cada uma das 160 coletas seria menor que o custo das 4 coletas do segundo caso, porém, isto não é verdade. O custo do algorítmo que implementa a estratégia *mark and sweep* é a soma dos custos de marcar objetos do *heap* e varrê-los. O custo da varredura é linear no número total de objetos liberados — o qual é o mesmo, não importando o quão freqüentemente o gerenciador coleta. De qualquer forma, o custo de marcar objetos depende da quantidade de dados correntemente em uso quando da ativação da coleta, a qual é aproximadamente equivalente à quantidade de memória dos dados de vida longa. Portanto, este custo é o mesmo em ambos os casos, mas uma vez que este custo deva ser pago por coleta, poucas coletas favorecem uma execução rápida da animação.

Na tentativa de manter um equilíbrio entre o número de coletas realizadas e a quantidade de memória utilizada, a eurística implementada no gerenciador observa a quantidade de memória que tem sido alocada desde a última vez que ocorreu uma coleta. Se esta quantidade de memória é menor que 1/3 do total de memória correntemente em uso, então a coleta é adiada. No exemplo hipotético, o gerenciador usaria em média 24MB (uma vez que cresceria

até 24MB em incrementos de 1MB). Toda vez que $\frac{1}{3}$.24=8MB de dados de vida curta fossem criados de acordo com os 16MB ocupados pelos dados de vida longa, o gerenciador coletaria 8MB. Portanto, executaria 160/(24-16) = 20 coletas ao invés de 160 ou 4.

A decisão de se utilizar 1/3 do total de memória é simplesmente pelo fato de que, assintoticamente, este número significa que o *heap* terá aproximadamente 33% de memória livre depois de uma coleta.

4.6 A Integração com o Motor de Física

Nesta seção tratamos da forma como o motor de física PhysX SDK é integrado ao sistema de animação.

Como vimos no Capítulo 2, uma instância do motor de física é obtida invocando-se a função `NxCreatePhysicsSDK()`. Esta função toma como argumento (*default*) uma instância especializada da classe `NxUserAllocator`. Passar este argumento à função significa especificar ao motor de física a existência de um objeto capaz de prover esquemas próprios de gerência de memória, dispensando o motor desta tarefa.

Na integração do motor de física com o sistema, uma instância especializada da classe `NxUserAllocator` é passada como argumento à `NxCreatePhysicsSDK()`. No sistema de animação, esta instância corresponde àquele objeto da classe `tHeapManager`, que provê o gerenciamento da memória de objetos em blocos tal como visto na Seção 4.4. Isto significa que todos os objetos instanciados pelo motor de física (excluindo-se alguns objetos internos ao motor utilizados para detecção de colisão) residem fisicamente na memória de objetos da `MVA`.

Do ponto de vista da integração do PhysX, objetos que residem na memória de objetos são classificados como *imunes* e *não imunes* à coleta de lixo. Obviamente, o coletor de lixo do sistema não pode atuar diretamente sobre objetos instanciados pelo motor de física. Estes objetos são classificados como imunes mas podem ser destruídos explicitamente pelo animador; diferentemente dos objetos não imunes. Esta classificação é a base para a definição de objetos locais e objetos remotos.

Objetos Locais e Remotos

Um objeto local é qualquer objeto instanciado pelo animador por meio da instrução **`new`** da especificação da `LA`. Isto significa que objetos locais são aqueles não imunes à coleta de lixo e cuja `MVA` conhece sua estrutura interna, ou seja, a estrutura definida pela classe do objeto (classe esta presente no *pool* de constantes do sistema). Um objeto remoto, por outro lado, é qualquer objeto cuja `MVA` desconhece sua estrutura interna. Instâncias do motor de física, por exemplo, são compreendidas pela `MVA` como um objeto remoto, embora ocorram fisicamente na memória de objetos da máquina virtual.

Fundamentalmente, a integração do motor de física com o sistema de animação ocorre por meio de troca de mensagens entre objetos locais e objetos remotos. A possibilidade de troca de mensagens é oferecida por meio da Interface Nativa — um componente da arquitetura do sistema.

4.6.1 Interface Nativa

A Interface Nativa `IN` é uma interface de programação nativa que permite o código da `LA` (*bytecodes* interpretados pela `MVL`) interoperar com aplicações e bibliotecas escritas em outras linguagens de programação, tal como C++ e assembly.

Há situações onde a LA apenas não oferece recursos para sanar algumas necessidades do animador. Para contornar esses problemas, animadores podem fazer uso da IN para escrever métodos nativos e assegurar que aquelas situações possam ser solucionadas utilizando-se recursos de uma outra linguagem (conhecida como linguagem nativa). A seguir exemplificamos alguns casos em que se faz necessária a utilização de métodos nativos:

- A biblioteca padrão de classes da LA não suporta as características dependentes de plataforma que podem ser necessárias pela aplicação de animação[5];
- A utilização de uma biblioteca já desenvolvida (código legado), escrita em outra linguagem e pronta para ser utilizada quando do desejo de fazê-la acessível ao código da LA (a integração do motor com o sistema encaixa-se neste caso);
- A necessidade de executar uma pequena parcela de código que exige tempo crítico. A utilização de uma linguagem de baixo nível, tal como assembly, pode ser de grande valia.

Escrevendo Código Nativo

Na escrita de códigos em uma animação, pode-se utilizar métodos nativos para criar, ler e atualizar o estado dos objetos da MVA (objetos locais, incluindo vetores e *strings*), invocar métodos da linguagem, além de lançar exceções. Todas essas funcionalidades são encapsuladas em métodos da IN. Como exemplo simples de utilização da IN, considere o seguinte trecho de código escrito no arquivo anima.scn:

```
void nativePrintMax(int a, int b) native;
void nonNativePrint(int value)
{
  Console::writeString("max:  " + value);
}
void main()
{
  nativePrintMax(2,3);
}
```

A função nativePrintMax() deverá computar e imprimir na saída padrão o maior valor dentre os parâmetros inteiros a e b. Esta função é nativa, ou seja, suas instruções encontrar-se-ão em uma biblioteca de ligação dinâmica a ser criada pelo animador. Considere também que a impressão propriamente dita do maior valor dentre os parâmetros deva ser feita pela função nonNativePrint(), que toma como argumento o valor inteiro a ser impresso. Isto quer dizer que a função nativePrintMax(), a partir do seu código nativo, deverá invocar a função *não* nativa nonNativePrint().

Uma vez compilado o arquivo anima.scn, o CLA criará o arquivo anima.h com o seguinte conteúdo:

```
1   #if !defined(__nativeinterface_h)
2   #include "nativeinterface.h"
3   #endif
4
5   extern "C"__declspec (dllexport)
6   void __stdcall nativePrintMax__i_i(IN* in, void* stack);
```

[5]Embora esta versão do sistema de animação execute apenas sobre a plataforma Windows, existe a possibilidade de estendê-lo para outras plataformas.

O nome da função C++ `nativePrintMax__i_i()`, na linha 6, corresponde ao *mangled name* da função `nativePrintMax(int,int)` definida na `LA`. A função toma como um dos argumentos um ponteiro para o objeto `IN` da `MVA`. É por meio deste objeto que `nativePrintMax()` invocará a função não nativa `nonNativePrint()` para impressão do maior valor. A definição da `IN` (classe do objeto) encontra-se no arquivo `nativeinterface.h`, justificando, portanto, sua inclusão na linha 2.

Além do arquivo de cabeçalho `anima.h`, o `CLA` criará também o respectivo arquivo `anima.cpp` no qual o animador deverá implementar o corpo de `nativePrintMax__i_i()`. Embora a interface da classe `IN` não tenha sido apresentada ainda, é fácil compreender a implementação no corpo da função em `anima.cpp`:

```
1   #include "anima.h"
2
3   void __stdcall nativePrintMax__i_i(IN* in, void* stack)
4   {
5      //captura os parametros na ordem
6      nint a = in->FetchInt(stack);
7      nint b = in->FetchInt(stack);
8
9      //processa
10     nint max = a > b ?  a :  b;
11
12     //invoca uma função definida em LA
13     static methodId mid = in->getGlobalMethodId("nonNativePrint__i");
14     in->voidCall(0, mid, max);
15  }
```

Note que, nas linhas 6 e 7, o objeto apontado por `in` é responsável pela captura dos parâmetros da função. Este objeto, bem como o parâmetro `stack`, são automaticamente empilhados pela `MVL` e disponibilizados ao animador para cada função definida como nativa. O argumento `stack` corresponde à uma referência à pilha da `MVL`, e a captura dos parâmetros deve obedecer a ordem imposta quando da invocação da função na `LA` (da esquerda para a direita). A classe `IN` provê uma série de métodos `FetchX` para captura de parâmetros, onde `X` corresponde aos tipos primitivos e não primitivos da linguagem.

Na linha 10 ocorre a computação propriamente dita do maior valor a ser impresso. Em seguida, as linhas 13 e 14 correspondem ao trecho de código nativo responsável pela invocação da função não nativa `nonNativePrint()` para impressão do maior valor computado.

Qualquer símbolo definido na `LA` (residente no *pool* de constantes do sistema) possue um identificador único. Este identificador é obtido referenciando-se o símbolo por meio do seu *mangled name*. A classe `IN` provê métodos para obtenção de identificadores dos símbolos da `LA`; sejam variáveis, funções globais, métodos de instância e de classe, atributos, etc. Na linha 13, especificamente, foi obtido o identificador para a função `nonNativePrint(int)` a partir do seu *mangled name* `nonNativePrint__i`. De posse deste identificador, a função é invocada na linha seguinte passando como argumento o identificador da função e o maior valor a ser impresso.

O método `voidCall()` da `IN` executa uma função ou método cujo tipo de retorno seja **`void`** (analogamente, existem métodos da `IN` para tipos de retorno primitivos e não primitivos). Sua assinatura é:

```
void voidCall(nobject obj, methodId mid, ...);,
```

onde `obj` é uma referência para o objeto o qual deseja-se enviar a mensagem, `mid` é o identificador para a mensagem (o método/função), e a seqüência de três pontos representa os argu-

mentos que o método pode receber (parâmetros C++ variados). Como `nonNativePrint()` é uma função global, o parâmetro `obj` é nulo (linha 14).

O passo seguinte é a criação da biblioteca de ligação dinâmica a partir dos arquivos `anima.h`, `anima.cpp` e `nativeinterface.h`. Feito isto, tal biblioteca deve ser mantida no `PATH` do sistema. A execução do programa é feita por linha de comando parametrizando a máquina virtual com a biblioteca. Por exemplo, suponha que a biblioteca criada tenha o nome `anima.dll`:

```
C:\>mva -l anima.dll anima.oaf [ENTER]
max: 3
```

A Interface da Classe IN

Nesta seção abordamos o conteúdo resumido do arquivo de cabeçalho `nativeinterface.h`, onde se encontra definida a classe `IN`.

Os tipos primitivos da linguagem de animação são mapeados para representações na linguagem nativa. Os tipos **`char`**, **`int`**, **`float`** e **`bool`** da `LA` são representados em C++ como `nchar`, `nint`, `nfloat` e `nbool`, respectivamente:

```
1   typedef unsigned short nchar;
2   typedef signed long nint;
3   typedef double nfloat;
4   typedef nint nbool;
```

Para a implementação de métodos nativos faz-se necessária uma referenciação a metadados, tal como atributos, métodos/funções e classes. Estas referências correspondem aos identificadores, sendo representados nativamente pelos tipos `classId` (para classes), `fieldId` (para atributos e variáveis), e `methodId` (para métodos e funções):

```
5   typedef class _nclass  {}* classId;
6   typedef class _nfield  {}* fieldId;
7   typedef class _nmethod {}* methodId;
8   typedef class _nobject {}* nobject;
```

Um objeto local da `MVA` é representado nativamente pelo tipo `nobject`. Note no trecho de código que, tanto os tipos de metadados quanto o tipo `nobject`, são classes C++ com corpos vazios. Isto é feito devido ao fato de que não se pode manipular diretamente instâncias dessas classes sem que seja por intermédio da `IN`.

Os tipos vetores da especificação da linguagem também ganham uma representação nativa. Vetores podem ser tanto de tipos primitivos quanto de tipos não primitivos:

```
9   typedef class _nstring:      public _nobject {}* nstring;
10  typedef class _narray:       public _nobject {}* narray;
11  typedef class _nboolArray:   public _narray  {}* nboolArray;
12  typedef class _ncharArray:   public _narray  {}* ncharArray;
13  typedef class _nintArray:    public _narray  {}* nintArray;
14  typedef class _nfloatArray:  public _narray  {}* nfloatArray;
15  typedef class _nobjectArray: public _narray  {}* nobjectArray;
```

Note que vetores de tipos primitivos, vetores de objetos genéricos, e objetos *strings* são representados por classes de objetos C++ que derivam direta ou indiretamente da classe `nobject`.

Métodos da `IN` fazem uso da representação nativa dos tipos da linguagem de animação. Devido às limitações de espaço, são apresentados somente os principais métodos da interface.

```
16  class IN
17  {
18    public:
19      //Captura de parâmetros.  Um método para cada tipo da LA.
20       nint    FetchInt   (void*& stack);
21       nfloat  FetchFloat (void*& stack);
22       nchar   FetchChar  (void*& stack);
23       nbool   FetchBool  (void*& stack);
24       nobject FetchObject(void*& stack);
25       nstring FetchString(void*& stack);
26
27      //Obtenção de identificadores para classes de objetos.
28      //O parâmetro 'sig' representa o mangled name da classe.
29       classId findClass (const char* sig) const;
30       classId findInnerClass (classId cls, const char* sig) const;
31       classId getClass(nobject obj) const;
32       classId getSuperClass(classId clazz) const;
33
34      //Obtenção de identificadores de métodos e funções.
35       methodId getMethodId(classId clazz, const char* sig) const;
36       methodId getGlobalMethodId(const char* sig) const;
37
38      //Executa funções e métodos com base no tipo de retorno.
39       void voidCall(nobject obj, methodId mid, ...)  const;
40       nint intCall(nobject obj, methodId mid, ...)  const;
41       nchar charCall(nobject obj, methodId mid, ...)  const;
42       nfloat floatCall(nobject obj, methodId mid, ...)  const;
43       nobject objCall(nobject obj, methodId mid, ...)  const;
44
45      //Obtenção de identificadores de atributos e variáveis.
46       fieldId getFieldId(classId clazz, const char* sig) const;
47       fieldId getFieldId(nobject obj, const char* sig) const;
48
49      //Ajuste de atributos de classe
50       void intSetStaticField(fieldId fid, nint value) const;
51       void charSetStaticField(fieldId fid, nchar value) const;
52       void floatSetStaticField(fieldId fid, nfloat value) const;
53       void objectSetStaticField(fieldId fid, nobject value) const;
54
55      //Ajuste de atributos de instância.
56       void intSetField(nobject o, fieldId f, nint v) const;
57       void charSetField(nobject o, fieldId f, nchar v) const;
58       void floatSetField(nobject o, fieldId f, nfloat v) const;
59       void objectSetField(nobject o, fieldId f, nobject v) const;
60
61      //Obtenção do valor de atributos de classe.
62       nint intGetStaticField(fieldId fid) const;
63       nchar charGetStaticField(fieldId fid) const;
64       nfloat floatGetStaticField(fieldId fid) const;
65       nobject objectGetStaticField(fieldId fid) const;
66
67      //Obtenção do valor de atributos de instância.
68       nint intGetField(nobject obj, fieldId fid) const;
69       nchar charGetField(nobject obj, fieldId fid) const;
70       nfloat floatGetField(nobject obj, fieldId fid) const;
71       nobject objectGetField(nobject obj, fieldId fid) const;
72
73      //Criação de objetos.  O parâmetro 'mid'
```

```
74        //é o identificador para o construtor da classe do objeto.
75         nobject newObject(classId clazz, methodId mid, ...)  const;
76         nstring newString(const char* str) const;
77
78        //Conversão de objetos strings em uma representação C++.
79         const char* getPtrChar(nstring obj) const;
80
81        //Criação de vetores de tipos primitivos e não primitivos.
82        //O parâmetro 'sz' é a dimensão do vetor.
83         nboolArray newboolArray(size_t sz) const;
84         ncharArray newcharArray(size_t sz) const;
85         nintArray newintArray(size_t sz) const;
86         nfloatArray newfloatArray(size_t sz) const;
87         nobjectArray newobjectArray(classId clazz, size_t sz) const;
88
89        //Lançamento de exceções.
90         void throwInt(nint value) const;
91         void throwChar(nchar value) const;
92         void throwBool(nbool value) const;
93         void throwObject(nobject value) const;
94
95        //Erro fatal (aborta a execução.)
96         void fatalError(const char* message) const;
97
98        //Destruição de referências locais para objetos.
99        //(explicação na seção seguinte).
100        void deleteRef(nobject local) const;
101    ...
102 };
```

4.6.2 Referenciando Nativamente Objetos da LA

Na utilização de métodos nativos, valores de tipos primitivos da linguagem são copiados entre LA e o código nativo. Objetos locais da LA, diferentemente, são passados por referência. A MVA deve manter-se a par de todos os objetos que são passados ao código nativo de maneira que estes objetos não possam ser liberados pelo coletor de lixo. Isto ocorre porque a MVA, quando da execução de um método nativo, perde temporariamente o controle sobre a pilha de execução de funções criada nativamente. O código nativo, por sua vez, deve ter uma maneira de informar à MVA que já não necessita de objetos. Além disso, o coletor de lixo deve ser capaz de alcançar todos os objetos referenciados no código nativo.

Referências Locais

A MVA provê um mecanismo de monitoração de objetos locais por meio de *referências locais*. Neste contexto, uma referência local é um objeto *imune* criado na memória de objetos da MVA para referenciar qualquer objeto local passado como argumento a um método nativo, como parâmetro em métodos da IN, ou retornados por métodos nativos.

A presença de uma referência local para um objeto local é a estratégia da MVA para alcançar objetos locais quando estes entregues ao código nativo. Desta forma, quando da execução do método nativo, objetos locais são vistos pelo coletor como uma referência indireta alcançada por meio da referência local (que é imune à coleta enquanto o código nativo é executado).

Na maioria dos casos o animador deve confiar à MVA a liberarção de todas as referências locais depois que um método nativo retorna. Entretanto, há situações em que o animador

deveria explicitamente liberar uma referência local. Considere, por exemplo, as seguintes situações:

- Um método nativo acessa um objeto da LA de tamanho considerável (por exemplo, um objeto grande — local da MVA). O método nativo então executa uma computação adicional antes de retornar ao método chamador. A referência local criada para o objeto grande irá prevenir da coleta o tal objeto, mesmo se este objeto não for mais utilizado no restante da computação do método nativo;
- O método nativo cria um grande número de objetos locais (obrigando a MVA criar também um grande número de referências locais para estes objetos), porém, nem todos os objetos são utilizados ao mesmo tempo. Uma vez que a MVA precisa de uma certa quantidade de memória para a criação de referências locais, criando-se muitos objetos locais no método nativo poderia induzir no sistema uma falta de memória. Por exemplo, considere um método nativo que iterage sobre um grande vetor de objetos, recuperando e realizando operações em um objeto por vez. Depois de cada iteração, o animador pode não mais fazer uso do objeto, embora ainda exista uma referência local para aquele objeto em particular.

A IN permite ao animador manualmente liberar uma referência local em qualquer ponto do código nativo. Para assegurar que o animador possa explicitamente liberar uma referência local, IN não permite criar explicitamente uma referência local (esta tarefa é da MVA). O método `deleteRef(nobject local)` da IN libera a referência local criada para o objeto local tomado como argumento.

4.6.3 Biblioteca Nativa de Animação

Na API de animação, como vimos, existem classes de objetos que abstraem entidades comumente utilizadas numa animação, tal como cenas, atores, luzes, etc. Algumas dessas entidades também possuem sua respectiva representação no motor de física integrado ao sistema. Uma cena da LA, por exemplo, é abstraída pela classe `Scene` (que é um seqüenciador, especificamente um *script*), já uma cena do PhysX SDK, por outro lado, é uma instância da classe `NxScene`. Do ponto de vista do animador, a integração do PhysX ao sistema deve ser transparente no sentido de que uma única entidade de cena, obviamente, deva ser disponibilizada na API de animação.

A biblioteca nativa do sistema corresponde àquelas classes de entidades que possuem uma representação tanto na LA quanto no PhysX. Objetos LA da biblioteca nativa correspondem aos objetos locais, instanciados por meio da instrução **new** da especificação da linguagem. Objetos do motor de física, diferentemente, são compreendidos pelo sistema como objetos remotos. A biblioteca nativa de animação é o meio pelo qual o animador, a partir de objetos locais, pode fazer uso do potencial realístico oferecido pelo motor de física. Para isto, mensagens enviadas a objetos locais são repassadas aos respectivos objetos remotos.

Ao animador não é dada a possibilidade de instanciar objetos remotos (esta tarefa é feita implicitamente no sistema), porém, uma representação local à LA de um objeto remoto é dada pela classe `RemoteObject`:

```
final class RemoteObject
{
  private:
    constructor(){}
}
```

Note que a classe é definida com o modificador **final**, ou seja, especifica que a classe `RemoteObject` não pode ser derivada. Em adição definiu-se um construtor na seção privada da classe, o que impede em tempo de compilação a criação de uma instância de `RemoteObject`.

Classes da biblioteca nativa correspondem às interfaces para manipulação de objetos de classes remotas. Esta manipulação é provida pela interface nativa `IN`. A Figura 4.10 ilustra o relacionamento entre as classes de objetos da `LA` e do PhysX SDK:

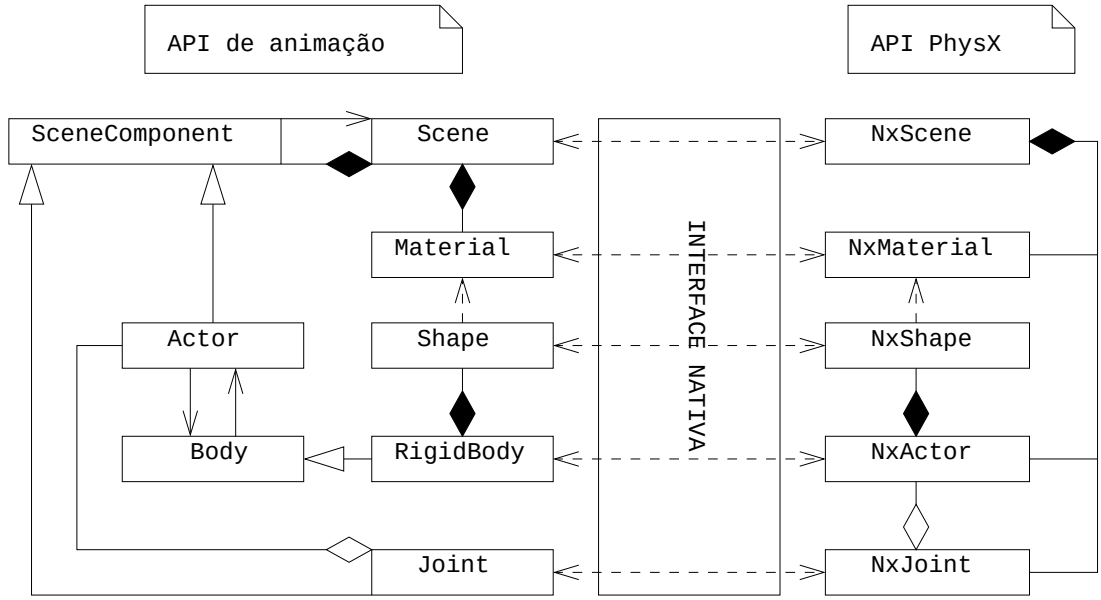


Figura 4.10: Biblioteca nativa.

Como podemos ver na figura, a `IN` é o elo de ligação entre classes de objetos da `LA` e classes do motor de física. Dizemos que uma classe da biblioteca nativa (local ao sistema) conhece sua respectiva classe do motor (remota ao sistema). Em outras palavras, objetos locais mantêm uma referência para os seus respectivos objetos remotos (e vice-versa). A troca de mensagens entre objetos locais e remotos ocorre por meio dessas referências.

Troca de Mensagens Entre Objetos Locais e Remotos

Para a compreensão de como é feita a troca de mensagens entre objetos locais e remotos, exemplificamos usando o par de classes `RigidBody - NxActor`. Considere inicialmente a interface simplificada da classe `Body`:

```
1   class Body
2   {
3     private:
4       Actor myActor;
5     ...
6     public:
7       constructor(Actor actor):
8         myActor(actor)
9       {}
10    ...
11      property Actor actor {read = myActor};
12  }
```

Um corpo mantém uma referência para seu respectivo ator (linha 4). Este ator é passado como argumento no construtor de `Body` (linha 7), ou seja, um corpo "conhece" seu ator.

A seguir a interface simplificada da classe `RigidBody`:

```
13  class RigidBody:  Body
14  {
15    private:
16      RemoteObject remoteObject;
17    ...
18    public:
19      constructor(Actor actor) native;
20    ...
21      float getMass() const native;
22      void setMass(float mass) native;
23    ...
24      property float mass { read= getMass; write= setMass };
25
26      virtual destructor() native;
27  }
```

Como vemos na linha 16, um corpo rígido (objeto local, instância de `RigidBody`) mantém uma referência pra seu respectivo objeto remoto. Este objeto remoto, como veremos, é criado nativamente e atribuído à `remoteObject` a partir do construtor nativo (linha 19). Além disso, note a existência de métodos nativos na definição da classe (linhas 21 e 22). Parte do código nativo associado à classe `RigidBody` é exibida e comentada a seguir. Inicialmente, considere as seguintes funções C++:

```
static fieldId cacheRemFieldId = 0;
static fieldId getRemoteRigidBodyFieldId(IN* in, nobject lBody)
{
  if (cacheRemFieldId == 0)
    cacheRemFieldId = in->getFieldId(lBody, "RigidBody_remoteObject");
  return cacheRemFieldId;
}

static fieldId cacheRemScnFieldId = 0;
static fieldId getRemoteSceneFieldId(IN* in, nobject lScene)
{
  if (cacheRemScnFieldId == 0)
    cacheRemScnFieldId = in->getFieldId(lScene, "Scene_remoteObject");
  return cacheRemScnFieldId;
}
```

Estas funções, dentre muitas outras, fazem parte de um conjunto de rotinas que implementam um mecanismo de *cache* para a obtenção, via interface nativa, dos identificadores de metadados, como atributos (fieldId), classes (classId) e métodos (methodId), que são referenciados freqüentemente no código nativo. Este mecanismo alivia a `IN` melhorando consideravelmente o desempenho global da aplicação. Ambas as funções retornam identificadores do atributo `remoteObject` das instâncias de `RigidBody` e `Scene`, respectivamente.

Além das funções de *cache*, considere as funções:

```
NxActor& RigidBodyL2R(IN* in, nobject localBody)
{
  fieldId rFieldId = getRemoteRigidBodyFieldId(in, localBody);
  return *(NxActor*)in->objectGetField(localBody, rFieldId);
}
```

```
NxScene& SceneL2R(IN* in, nobject localScene)
{
  fieldId sFieldId = getRemoteSceneFieldId(in, localScene);
  return *(NxScene*)in->objectGetField(localScene,sFieldId);
}
```

Ambas tomam como argumentos um ponteiro para a instância da interface nativa e a representação nativa de um objeto local. A função `RigidBodyL2R()` retorna, com auxílio da `IN`, o objeto remoto (que corresponde à instância de `NxActor`) de um corpo rígido local (`L2R` = **L**ocal to **R**emote). Similarmente, a função `SceneL2R()` retorna o objeto remoto (que corresponde à instância de `NxScene`) de uma cena local.

A escrita do código nativo do construtor de `RigidBody` (e de outras classes que compõe a biblioteca nativa) obedece 3 passos básicos: (1) a invocação do construtor da classe base; (2) a criação do objeto remoto relativo ao objeto local; (3) a atualização do atributo `remoteObject` do objeto local:

```
1   void __stdcall RigidBody_constructor__Actor(IN* in, void* stack)
2   {
3     nobject thiz       = in->FetchObject(stack);
4     nobject localActor = in->FetchObject(stack);
5
6     //Invocação do construtor da classe base.  Passo (1)
7     static nclass cacheClass
8       = in->getSuperClass(in->getClass(thiz));
9     static methodId cacheConstructorMethodId
10      = in->getMethodId(cacheClass, "Body_constructor__Actor");
11    //Um construtor, nativamente, é visto com retorno void.
12    in->voidCall(thiz, cacheConstructorMethodId, localActor);
13
14    //criação de uma instância de NxActor.  Passo (2)
15    NxActor* remoteObject;
16    static fieldId cacheSceneFieldId
17      = in->getFieldId(localActor, "SceneComponent_myScene");
18    nobject localScene
19      = in->objectGetField(localActor, cacheSceneFieldId);
20    NxActorDesc actorDesc;
21    NxBodyDesc body;
22    actorDesc.body = &body;
23    body.mass = 1;
24    body.massSpaceInertia = NxVec3(0,-1,0);
25    remoteObject = SceneL2R(in, localScene).createActor(actorDesc);
26    remoteObject->userData = thiz; //objeto remoto conhece seu local
27
28    //atribuição da instância remota à instância local.  Passo (3)
29    fieldId rFieldId = getRemoteRigidBodyFieldId(in, thiz); //cache
30    in->objectSetField(thiz, rFieldId, remoteObject);
31  }
```

Nas linhas 3 e 4 ocorre a obtenção dos parâmetros do construtor. Como qualquer método de instância, o primeiro argumento é o **`this`** da especificação da linguagem (este parâmetro é empilhado implicitamente pela `MVL`), enquanto que o segundo é o ator local definido na assinatura do construtor na `LA`.

As linhas 7-12 correspondem ao trecho de código nativo responsável pela invocação do construtor da classe base (neste trecho desconsideramos a utilização de funções de *cache* para detalhar a compreensão do código). As linhas 14-26 definem o trecho de código para a criação

de um objeto remoto (instância de NxActor). Note que, especificamente entre as linhas 20 e 26, definiu-se propriedades iniciais para a criação de um corpo rígido remoto. No sistema de animação, por padrão, todo corpo rígido dinâmico é criado definindo-o com aquela massa inicial e tensor de inércia (linhas 23 e 24, respectivamente), dispensando, portanto, a criação prévia de descritores de formas (instâncias especializadas de NxShapeDesc) naquele instante.

As linhas 29 e 30 correspondem ao trecho de código responsável pela atribuição do objeto remoto ao atributo remoteObject do objeto local (thiz).

Os métodos nativos getMass() e setMass() da classe RigidBody, dentre outros, correspondem às mensagens enviadas a objetos locais. A seguir é apresentada a implementação nativa destes métodos, os quais provêem o repasse das mensagens do objeto local para seu respectivo objeto remoto:

```
33  nfloat __stdcall RigidBody_getMass(IN* in, void* stack)
34  {
35    nobject thiz = in->FetchObject(stack);
36    //repasse da mensagem
37    return (nfloat)RigidBodyL2R(in, thiz).getMass();
38  }
39
40  void __stdcall RigidBody_setMass__f(IN* in, void* stack)
41  {
42    nobject thiz = in->FetchObject(stack);
43    nfloat mass = in->FetchFloat(stack);
44    //repasse da mensagem
45    RigidBodyL2R(in, thiz).setMass(NxReal(mass));
46  }
```

A integração do motor de física ao sistema não obrigou a destruição de objetos remotos pelo animador. Esta destruição continua sendo provida implicitamente pela MVA, embora classes de objetos da biblioteca nativa mantenham métodos para a destruição explícita de objetos. Similarmente à construção de um objeto remoto a partir de um construtor nativo, a destruição de um objeto remoto é feita no destrutor virtual nativo de cada classe de objetos da biblioteca nativa.

4.7 Controlador de Animação

O fluxo de uma animação envolve a colaboração de todos os componentes da MVA. O controlador é o componente responsável por orquestrar os passos de execução; fundamentalmente, gerencia a execução das partes dos seqüenciadores instanciados pelo animador. Para isto, ele mantém um conjunto de filas de contexto de *script* e filas de ações.

Um elemento de uma *fila de contexto de script* contém uma referência para um *script* ativo (ou seja, no estado RUNNING ou WAITING) e um objeto de contexto referente ao método run() do *script*. Um contexto é uma estrutura que mantém todas as informações necessárias para a MVL ser capaz de retomar a execução de um(a) método/função a partir de um ponto específico no código objeto. Isto inclue o endereço da próxima instrução a ser executada e uma rerefência para o respectivo *frame* da pilha de *frames* da cadeia de execução, dentre outras. O controlador possui duas filas de contexto de *script*, são elas:

- RSQ, a qual mantém um elemento para cada *script* iniciado em uma animação (*scripts* no estado RUNNING);
- WSQ, a qual mantém um elemento para cada *script* no estado WAITING. Um elemento de WSQ, adicinoalmente, mantém o tempo restante que um *script* aguarda para retomar

sua execução (*timeout*, no caso de o tempo expirar), além de uma lista de referências para objetos de sincronização no qual o *script* aguarda por sinalizações.

Um elemento de uma *fila de ação* contém uma referência para uma ação. O controlador possui três filas de ações: `IAQ`, `UAQ` e `FAQ`, as quais tem um elemento para cada ação no estado `INITIATED`, `RUNNING` e `TERMINATED`, respectivamente. Em adição, o controlador possui uma lista `SEL` de referências para objetos de sincronização sinalizados no *tick* corrente.

O gerenciamento dos passos de uma animação pelo controlador consiste em alternar os *scripts* e ações entre as filas de seqüenciadores, de sinalizar e desfazer a sinalização de eventos, solicitar ao motor de física operações de dinâmica de corpos rígidos, além de invocar o renderizador para criação de um quadro da animação. A execução de uma animação e sua gerência pelo controlador segue os passos:

Passo 1 A `MVA` carrega o arquivo objeto de animação e busca pela função principal. Se esta não é encontrada, a `MVA` lança uma exceção que aborta a execução do programa. Caso contrário, a `MVA` solicita à `MVL` a execução do *bytecode* da função principal.

Passo 2 A execução procede até que o método `start()` da cena seja invocado. A primeira instrução gerada pelo `CLA` para qualquer método que dá início a um *script* é `halt`, a qual é interpretada pela `MVL` como uma ordem para suspender a execução da função corrente. Uma vez que uma cena é um *script*, a `MVL` é interrompida. Em seguida, o controlador cria um novo elemento de contexto de *script* contendo uma referência para a cena iniciada e o contexto corrente da `MVL`, e o coloca em `RSQ`. A cena torna-se a cena corrente e o *tick* corrente é ajustado para zero. Se um seqüenciador de tipo diferente de `Scene` é iniciado, então a `MVA` lança uma exceção que aborta a execução do programa. Pelo menos uma cena deveria ser iniciada a partir da função principal.

Passo 3 Enquanto o tempo total da cena corrente não é zero e as filas de contexto de *script* e ações não estão vazias, o controlador realiza o ciclo de atualização para o *tick* corrente: passos 4 a 10.

Passo 4 Para cada elemento `S` de `RSQ`, o controlador solicita à `MVL` para retomar sua execução a partir do contexto de *script* correspondente. Como conseqüência, a `MVL` executa o método `run()` do *script*, o qual pode iniciar outros *scripts* e ações além de sinalizar eventos. Se uma nova ação é iniciada, o método `start()` da classe `Action` coloca em `IAQ` um novo elemento contendo a referência para esta ação. Se um evento é sinalizado, o método `setSignaled()` da classe `Event` adiciona em `SEL` um novo elemento contendo uma referência para o evento. A `MVL` continua a execução de `run()` até que:

- O método retorne ou `exit()` ou `abort()` seja invocado.
- O método `waitFor()` seja invocado. Como no caso do método `start()`, a primeira instrução de `waitFor()` interrompe a `MVL`. Em seguida, o controlador remove `S` de `RSQ` e coloca em `WSQ` (estado `WAITING`) um novo elemento contendo a referência para o *script*, o contexto corrente, e, no caso de argumentos passados para `waitFor()`, o tempo de espera por eventos e a lista de referências para os objetos de sincronização pelos quais o *script* aguardará sinalização.

Passo 5 Para cada elemento `S` de `WSQ`, o controlador verifica se o tempo de espera (*timeout*) é zero ou se os objetos de sincronização para o qual o *script* correspondente está aguardando está em `SEL`. Se assim for, então `S` é removido de `WSQ` e um novo elemento para o *script* é colocado em `RSQ`, ou seja, o *script* "acorda" e retorna ao estado `RUNNING`. Caso contrário, o tempo de espera é decrementado.

Passo 6 Para cada elemento `A` de `IAQ`, o controlador solicita à `MVL` a execução do método `init()` para a ação correspondente. `A` é removido de `IAQ` para `UAQ`.

Passo 7 Para cada elemento `A` de `UAQ`, o controlador verifica se o tempo de vida (propriedade `lifetime` de classe `Action`) da ação correspondente não é zero. Se assim for, o controlador solicita à `MVL` a execução do método `update()` para esta ação, além de atualizar as propriedades `time` e `lifetime`. Caso contrário, ou se `exit()` foi invocado a partir de `update()`, `A` é movido de `UAQ` para `FAQ`. Se `abort()` é invocado, a ação é removida de `UAQ` sem que tenha a parte `finalize()` executada no passo seguinte.

Passo 8 Para cada elemento `A` de `FAQ`, o controlador solicita à `MVL` a execução do método `finalize()` para a ação correspondente. Em seguida, `A` é removido de `FAQ`.

Passo 9 O controlador remove todos os elementos de `SEL` e solicita ao motor de física a executar a simulação para o passo de tempo correspondente ao *tick* corrente. Se qualquer contato entre os corpos rígidos é detectado, o controlador ajusta o estado de `ContactReport` como sinalizado e adiciona um novo elemento para o evento em `SEL`.

Passo 10 O tempo total da cena corrente é atualizado e o *tick* corrente é incrementado. Se este *tick* é multiplo de `resolução`, o controlador solicita ao renderizador a criação de um quadro da cena, o qual, dependendo da aplicação, pode ser imediatamente exibido ou posteriormente encaminhado para o ligador de arquivos de animação.

4.8 Considerações Finais

A execução de uma animação é feita pela `MVA`. Como uma máquina de comutação real, possui um conjunto de instruções e manipula diferentes áreas da memória em tempo de execução.

Na árquitetura da máquina virtual encontram-se definidos a área de dados, o *pool* de constantes, a pilha de frames além de um processador virtual.

A área de dados corresponde à região de memória onde residem todas as variáveis globais e estáticas de classe. O *pool* de constantes é uma representação compacta da tabela de símbolos criada pelo `CLA` e responsável pela obtenção de todas as informações de cada símbolo em tempo de execução. Para cada método invocado, é mantida na máquina virtual uma estrutura de dados que representa o estado da do método em execução. A pilha de *frames* corresponde à cadeia de execução de cada método invocado. O processador virtual é o componente responsável pela interpretação propriamente dita de cada uma das instruções que definem a linguagem alvo do compilador.

Qualquer objeto instanciado pela aplicação reside na memória de objetos lixo-coletável do sistema. A gerência de memória faz uso de um mecanismo de páginas, mapeadas por blocos de alocação. Cada bloco de alocação mantém uma coleção de objetos passíveis de alocação na página mapeada por ele. A coleta de lixo utiliza a estratégia mark-and-sweep de coleta de lixo. Para isto, o gerenciador da memória, instância de `HeapManager`, utiliza um esquema de cores representadas por listas encadeadas que mantém cada objeto alocado na memória de objetos. A decisão de coletar é baseada em uma eurística. Se a quatidade de memória que tem sido alocada desde a última vez que ocorreu uma coleta é menor que 1/3 do total de memória correntemente em uso, então o gerenciador opta pelo adiamento da coleta.

A integração do motor de física à `MVA` é baseada nos conceitos de objetos locais e objetos remotos. Um objeto local é um objeto cuja estrutura (definida por sua classe que reside no *pool de constantes*) é conhecida pela `MVA`, diferentemente de um objeto remoto. Instancias

do motor de física são compreendidos pela MVA como um objeto remoto, embora residam fisicamente na memória de objetos da máquina virtual.

A interface nativa da MVA corresponde ao componente que implementa o elo de ligação entre código nativo e código não nativo. Cada mensagem enviada a objetos locais do sistema é repassada aos respectivos objetos do motor de física. Para isto, a cada método nativo invocado é passado como argumento uma referência para uma instância da interface nativa IN.

Algumas entidades da API de animação comumente utilizadas possuem também uma representação no motor de física integrado ao sistema. A biblioteca nativa do sistema corresponde àquelas classes de objetos que possuem uma representação tanto na LA quanto no PhysX. Objetos LA correspondem aos objetos locais, e do motor aos objetos remotos. Cada objeto local da biblioteca nativa possui uma referência aos seu respectivo objeto remoto. É por meio desta referência que ocorre a troca de mensagens entre estes objetos.

Na construção da biblioteca nativa foi desenvolvido um conjunto de rotina que implementam um mecanismo de *cache* para manipulação de metadados no código nativo, aliviando a utilização da interface nativa e melhorando consideravelmente o desempenho global da aplicação.

O controlador do sistema é o componente responsável por orquestrar a execução dos seqüenciadores instanciados pelo animador. Para isto faz uso de um conjunto de filas de seqüenciadores e ações que representam o estado em que se encontra cada seqüenciador (RUNNING, WAITING, INITIATED e TERMINATED).

CAPÍTULO 5

Conclusão

5.1 Discussão dos Resultados Obtidos

O sistema de animação deste trabalho foi desenvolvido para atuar como uma ferramenta de visualização de simulações dinâmicas em aplicações de ciência e engenharia. O trabalho foi parte do projeto de pesquisa que pretendeu dar continuidade à expansão das capacidades de um *framework* denominado OSW — *Object Structural Workbench* — visando seu emprego também no desenvolvimento de aplicações de modelagem dinâmica. No contexto OSW, uma aplicação de modelagem é um programa gráfico de análise numérica e de visualização de modelos de sólidos baseados em física. Devido ao caráter temporal dos resultados da modelagem dinâmica, tornou-se conveniente a visualização destes resultados ao longo do tempo de análise, conduzindo naturalmente ao emprego de técnicas de animação.

Animação é uma área de pesquisa ampla e promissora. Os estudos realizados nos levaram a conhecer vários trabalhos que, de alguma forma, nos auxiliaram a alcançar os resultados obtidos. A pesquisa foi direcionada no desenvolvimento orientado a objetos de um sistema de computação que pudesse oferecer recursos para descrição de uma cena a ser animada e o emprego de algumas das técnidas de animação encontradas na literatura. Para a descrição foi criada uma linguagem de animação `LA`, estendida a partir de uma linguagem de propósito geral chamada `L`. Neste trabalho, as técnicas de animação foram providas ao animador como uma combinação entre os recursos da linguagem `LA` e o controle do fluxo da animação pelos componentes da arquitetura do sistema.

A primeira decisão a ser tomada foi definir o escopo do sistema no que se refere a quais técnicas de animação estariam disponíveis ao animador, de maneira que pudéssemos estender `L` direcionando-a não só à aplicabilidade de algumas daquelas técnicas de animação mencionadas no Capítulo 1, mas também na facilidade do controle da animação pelo animador.

Em nossas pesquisas percebemos que o controle do movimento/interação dos atores (o que caracteriza em grande parte o fluxo de uma cena animada) é de fato um dos maiores desafios na área de animação por computador. Muitos autores propõem diferentes estratégias para tratar esta questão [6, 9, 33, 26, 49]. Um ponto em comum entre elas é considerar o realismo da cena, ou seja, o quão convincente ela seria se ocorresse em vida real. Dentre as diferentes técnicas de animação, a simulação dinâmica mostrou-se como aquela que oferece o maior grau de realismo. Acreditamos, então, na conveniência em prover simulação dinâmica no sistema de animação desenvolvido.

Simulação dinâmica é um assunto amplamente explorado também na área de jogos digitais [11, 32]. Encontramos muitos motores de física (*physics engines*) capazes de realizar simulação dinâmica provendo o realismo [1, 28, 29]. No sistema deste trabalho utilizou-se o motor de física PhysX (desenvolvido pela Ageia Technologies — a fabricante da primeira unidade de processamento de física (PPU) do mercado), como um componente do sistema.

Adicionalmente foram desenvolvidos neste trabalho componentes responsáveis por compilar uma cena descrita na LA, orquestrar os componentes da cena (atores, luzes, câmeras, junções, etc.), renderizá-la, gerenciar a memória de objetos lixo coletável, além de integrar apropriadamente o motor de física ao sistema. As etapas do desenvolvimento do projeto consistiram em:

- *A especificação da linguagem de animação.* Nesta etapa estudou-se os recursos sintáticos da gramática da linguagem L afim de estendê-la. Uma vez estendida, nasceu LA, que passou então a contar com blocos de propriedades que facilitaram a parte descritiva de uma cena, além disso, com uma variante capaz de adicionar elementos em objetos que representam coleções, tal como vetores e listas encadeadas. Em adição LA ganhou recursos sintáticos que facilitaram a definição de classes de objetos que representam seqüenciadores. Concebemos seqüenciadores como uma abstração de entidades capazes de controlar o fluxo da animação por intermédio da linguagem, seja baseado em um fluxo seqüencial ou contínuo (*scripts* e ações, respectivamente). A essência dos seqüenciadores consistiu na sobrecarga de métodos abstratos definidos na classe base `Sequencer`. LA, então, foi capaz de oferecer maneiras de sobrecarregar tais métodos de forma trivial, tal como os blocos **`init`**, **`update`** (com auxílio da sentença **`switch_time`**) e **`finalize`** de uma ação, por exemplo, bem como o bloco **`run`** da classe `Script`. Não somente quanto à definição de métodos, LA também facilitou a criação de *singletons* por meio da definição de classes anônimas.

  De forma geral, nossa pretensão quanto à especificação da LA foi em manter todos os recursos de propósito geral de L (reforçando ainda mais a característica procedimental do sistema) mas acrescentando outros recursos que minimizasse a rigidez sintática de L que, de certa forma, deixava trabalhosa a escrita de códigos pelo animador. De toda maneira, a especificação de uma animação pela LA não foi direcionada aos artistas não programadores, ou seja, não nos preocupamos, por exemplo, com metodologias interativas na construção de animações.

- *A especificação da arquitetura e instruções da* MVA. Nesta etapa não nos preocupamos com detalhes de implementação da MVA. Nossos esforços concentraram-se em estudar e definir a arquitetura da máquina virtual (a área de dados, o *pool* de constantes, a pilha de *frames*, o processador virtual e seus registradores, etc.) e seu conjunto de instruções (a linguagem alvo para o qual o compilador, que seria implementado no passo seguinte, geraria código).

  Em nossas pesquisas adotamos a especificação da máquina virtual Java (JVM) [25] como um guia para a tomada de decisões nesta etapa. Todavia não precisamos adotar, obviamente, todos os recursos da JVM. A MVA deste trabalho, contudo, seria original sobre muitos aspectos, dentre eles: a existência de um controlador para conduzir o progresso de uma animação; um renderizador; e o fato de ter acoplada a ela um componente de física para cálculos de restrições dinâmica sobre objetos de uma cena.

  Nesta etapa definiu-se também o formato do arquivo de código objeto. Diferentemente da JVM, que cria vários arquivos de código objeto (`.class`) e os carregam por demanda segundo um mecanismo de *class loaders* [25], a MVA foi simplificada no sentido de criar um único arquivo de código objeto e carregá-lo por completo uma única vez.

As instruções da MVA, entretanto, foram inspiradas nas instruções da JVM. Algumas foram descartadas e outras criadas com o propósito específico para o controle do fluxo de uma animação (**halt** e **start**), para sincronismo (**wait_for**), para renderização (**render**), dentre outras.

- *A implementação do compilador.* Uma vez especificadas a linguagem de animação e a linguagem alvo que seria interpretada pela MVA, passou-se então à implementação do compilador da LA. Nesta etapa foi utilizada como fundação uma API para desenvolvimento de compiladores criada em anos anteriores pelo GVSG para ministrar a disciplina de compiladores aos alunos da graduação da UFMS. Entretanto, ao longo deste trabalho, esta API sofreu extensões e adaptações consideráveis para acomodar características específicas da linguagem de animação (algumas decorrentes das extensões de L), bem como pequenas otimizações.

  O compilador do sistema implementa um *parser* do tipo descendente recursivo. Durante o *parsing* de um arquivo de cena, é criada a representação do programa em uma árvore de derivação. A árvore de derivação criada, todavia, é uma árvore implícita decorrente das chamadas recursivas de cada procedimento que implementa uma produção da gramática da LA. Embora esta estratégia de compilação tenha se mostrada mais simples de implementar, a criação explícita em memória da árvore de derivação nos permitiria melhor otimização do código objeto gerado, uma vez que, depois de criada, poderíamos navegar nesta árvore convenientemente em busca daquelas otimizações, contribuindo ainda mais na eficiência global da aplicação de animação.

  Para tratar a eficiência, entretanto, o CLA pôde prover facilidades para a criação de métodos nativos. Um método nativo é um método escrito em outra linguagem de programação (usualmente C++) cujas instruções são definidas em uma biblioteca de ligação dinâmica; instruções estas nativas da plataforma corrente sobre a qual executa a máquina virtual (cuja desempenho de execução é considerável em comparação ao processo de interpretação de instruções da MVA). Esta característica do compilador mostrou-se adequada não só pelo própósito da eficiência, mas também por sua grande influência na integração do motor de física à MVA.

  De fato, o compilador da LA é o componente fundamental do sistema, uma vez que todos os outros componentes operam direta ou indiretamente sobre o resultado da especificação de uma cena compilada. Entretanto, reconhecemos (devido às limitações de espaço) que neste trabalho omitimos alguns detalhes importantes da sua construção, mas nos concentramos naqueles que contribuiriam mais intensamente nas características do sistema: a especificação de uma animação por meio da LA e sua execução pela MVA.

- *Desenvolvimento da* MVA. Construído o compilador (que passou a gerar o código a ser interpretado), passamos então ao desenvolvimento da máquina virtual de animação. Da mesma forma que LA foi estendida a partir de L, criaríamos a MVL para sua posterior extensão, nascendo a MVA. Em outras palavras, o núcleo da MVA seria a MVL.

  O primeiro componente criado da MVL foi a instância de `tHeapManager`, que correspondeu ao objeto que provê todo o gerenciamento de memória da aplicação. Para a criação deste componente estudou-se algumas estratégias de gerenciamento de memória comumente utilizadas em sistemas operacionais [43], por exemplo. A nossa abordagem de gerência de memória, porém, deveria tratar do controle de uma memória lixo-coletável, o que naturalmente nos levou a estudar algoritmos de coleta de lixo [10, 41]. Nestes estudos incluiu uma implementação da JVM Kaffe [23], de código aberto e facilmente encontrada em muitas distribuições Linux. As nossas decisões quanto à estratégia de gerência de memória e coleta de lixo na construção deste componente, portanto,

baseou-se não apenas na literatura, mas também em implementações práticas do que aprendemos nela.

Outro componente da MVL tratava-se do interpretador de *bytecode*. Este componente correspondeu ao objeto instância de tProcessor que abstraiu o processador da máquina virtual, responsável pela interpretação propriamente dita das instruções da MVL. O desenvolvimento deste componente tornou-se fácil devida à especificação minuciosa da arquitetura da MVA em um passo anterior, o que incluia também a especificação das instruções. Desta forma, trivialmente atribuímos um método de tProcessor para cada instrução a ser interpretada pela máquina virtual.

Para acomodar aquela característica do CLA quanto às facilidades na implementação de métodos nativos, a MVL contou com um componente capaz de fornecer recursos a um código nativo para acessar muitas das funcionalidades da máquina virtual — a interface nativa IN. A IN foi inspirada na JNI (Java Native Interface) — uma tecnologia para execução de códigos nativos a partir da linguagem Java. Embora um código nativo contribua significativamente no desempenho da aplicação, existe um custo em sua utilização com a IN (e com a JNI, no caso de Java). Este custo corresponde à manipulação de metadados (obtenção de identificadores), bem como a referenciação nativa a um objeto local da LA (a criação de referências locais implicitamente pela MVL). Diferentemente da JNI, porém, a IN contou com um conjunto de rotinas que implementam um mecanismo de *cache* de metadados para otimizar e aliviar consideravelmente a prática da IN.

Além da memória de objetos, do interpretador de *bytecodes* e da IN, obviamente, muitos outros componentes foram somados à máquina virtual (área de dados, *pool de constantes*, pilha de *frames*, etc.) de tal maneira que a MVL pudesse ser capaz de executar um programa de propósito geral, ou seja, um programa escrito em L apenas, sem a geração de instruções específicas para animação pelo CLA.

Uma vez criada a MVL, passaríamos então à sua extensão para a MVA. Esta extensão consistiu nas capacidades da MVL em controlar e executar os seqüenciadores instanciados pelo animador (o que dependia da geração de código específico para animação pelo CLA), bem como simular os efeitos de física sobre os objetos de uma cena animada. O controle da execução dos seqüenciadores foi tarefa do controlador, acoplado então à MVL, orquestrando uma animação por meio da execução e sincronização de cada uma das partes de seqüenciadores (**init**, **update**, **finalize** e **run**), provendo uma abordagem global e local de controle de movimentos dos atores, tal como mencionado no Capítulo 1.

A simulação de física foi tarefa do motor de física PhysX SDK, também acoplado à MVL nesta etapa. Antes da integração propriamente dita deste motor à MVL, entretanto, foi necessário o estudo da mecânica clássica relativa à dinâmica de corpos rígidos articulados, na qual toda a API de física do PhysX é baseada. Concluídos estes estudos, passamos então a estudar tal API afim de conhecermos a arquitetura do PhysX e adotarmos uma estratégia para sua integração à MVA. Desde a especificação da máquina virtual, entretanto, nos preocupamos em criar um sistema que não fosse dependente deste motor de física em particular. Nossos esforços também foram no sentido de criar um sistema que fosse adaptado com o PhysX porém não nascido em função dele.

Um ponto importante que mostrou-se interessante e adequando é o fato do PhysX possuir embutido de forma transparente um tipo de controle de movimentos. O mecanismo de simulação de física implementado com *threads* e baseado em passos de tempo foi de grande valia na orquestra da animação pelo controlador. Esta característica simplificou consideravelmente alguns pontos, tal como em que momento seria oportuna a execução dos seqüenciadores para a atualização do estado da cena, bem como em que momento seria oportuna a execução da simulação dinâmica (afinal, estas questões eram de res-

ponsabilidade do controlador). Além disso, a colaboração do controlador na gerência dos eventos capturados pelo PhysX e repassados a ele (controle da sinalização de objetos — lista `SEL`, por exemplo) foi interessante para facilitar o controle da animação também por parte do animador (considere, por exemplo, a obtenção dos pontos de contatos entre atores por meio do *singleton* instância de `ContactReport`).

Para o resultado final de uma animação (quadros que seriam agrupados em um filme) foi então criado o renderizador da `MVA`. Este renderizador foi baseado em OpenGl, contudo, escrito de forma tal a minimizar o impactor em uma possível mudança da tecnologia utilizada para este fim, tal como DirectX. Assim como o compilador, não nos concentramos na descrição das funcionalidades do renderizador, embora também tenha sido importante na conclusão de uma animação. Porém, a documentação do sistema (especificamente da `MVA`) também cobre o renderizador e pode ser consultada facilmente.

- *Desenvolvimento da* `API` *de animação.* A possibilidade de definir roteiros de animações por meio da `LA` e sua execução pela `MVA` possibilitou a implementação de uma `API` de animação. Esta `API` passou a ser composta por uma biblioteca nativa de animação que pôde ser prontamente utilizada e estendida pelo animador para fazer uso do potencial realístico oferecido pelo motor de física.

O objetivo geral deste trabalho foi o estudo dos fundamentos da animação por computador e o desenvolvimento orientado a objetos de um sistema de animação procedimental de cenas constituídas por corpos rígidos. Os objetivos específicos foram:

- *Estudo dos fundamentos matemáticos e computacionais necessários ao desenvolvimento das classes de objetos que compõe o sistema de animação.* No Capítulo 1 apresentamos conceitos básicos relativos ao tema "animação" bem como técnicas de animação e de controle de movimentos. Abordamos alguns sistemas de animação procedimental encontrados na literatura e comparamos as características do sistema de animação proposto neste trabalho com aqueles mencionados. Os fundamentos matemáticos e computacionais relativos à dinâmica de corpos rígidos foram apresentados no Capítulo 2, os quais deram a base para a compreensão da `API` de física do PhysX e a construção da `API` de animação, incluindo a biblioteca nativa do sistema.

- *Desenvolvimento e/ou integração dos componentes do sistema de animação.* Neste trabalho foram desenvolvidos o compilador `CLA` e a máquina virtual de animação `MVA`, que contou com componentes para controle da animação, renderização, gerência de memória-lixo coletável, processamento/interpretação de instruções, simulação dinâmica, e integração com código nativo.

- *Descrição das principais classes de objetos que compõe a implementação do sistema e da* `API` *de animação.* No Capítulo 3 apresentamos a linguagem de propósito geral `L`, bem como sua extensão na `LA`, onde as principais classes de objetos foram descritas.

As contribuições pretendidas com o trabalho foram:

- *A criação de um ambiente favorável à aplicação de recursos adequados para visualização ao longo do tempo de resultados de análise dinâmica pelo GVSG.* De fato, o sistema desenvolvido no trabalho oferece tal ambiente.

- *Prover uma fundação de componentes que possam ser utilizados como fonte de pesquisa e aprendizado aos alunos da graduação em ciência da computação da UFMS.* O compilador da `LA` de fato pode ser encaminhado como um estudo de caso na disciplina

de compiladores aos alunos da graduação. Na disciplina de sistemas operacionais, por exemplo, o estudo da gerência da memória de objetos lixo-coletável. Na disciplina de algoritmos e estrutura de dados, a aplicabilidade das mais diversas estruturas de dados e algoritmos utilizados no sistema (árvore binária balanceada que implementa a tabela de símbolos do compilador, tabelas de espalhamento, listas e filas na gerência da memória de objetos, pilhas na execução de *frames*, etc.);

- *Implementação do núcleo de um sistema que possa ser estendido e/ou adaptado ao escopo de jogos digitais e aplicações de tempo real.* Acreditamos que os conceitos de seqüenciadores (*scripts e ações*) para dirigir animações possam ser aplicados também no controle de jogos interativos, haja vista o caráter temporal no controle no progresso de uma animação por parte do animador.

Em adição àquelas contribuições, este trabalho resultou no artigo [30].

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

5.2 Trabalhos Futuros

Como trabalhos futuros sugerimos:

- A melhoria na estratégia de compilação. Como já mencionado, a criação explícita da árvore de derivação permitiria melhoras consideráveis na otimização do código objeto correspondente às instruções da `MVA`;
- A tradução *just-in-time* de parte do *bytecode* de uma animação em código nativo da plataforma sobre a qual executa a máquina virtual, melhorando ainda mais o desempenho da aplicação;
- A mesclagem de outras estratégias de coleta de lixo diferentes da *mark-and-sweep*, comparando os resultados em busca de novas otimizações;
- Uma interface humano-computador para descrição gráfica de animações;
- A extensão da gramática da linguagem de animação para acomodar definições sintáticas de regras comportamentais para atores, baseando-se, por exemplo, em inteligência artificial;
- A utilização de um motor de física capaz de computar restrições dinâmicas não só sobre corpos rígidos mas também sobre corpos deformáveis.

Referências Bibliográficas

[1] AGEIA Thechnologies. *PhysX SDK documentation.* Disponível em `http://www.ageia.com/pdf/PhysicsSDK.pdf`. Acessado em 15 de marco de 2006.

[2] BADLER, N.I. *Computer Animation Techniques.* In: BAILEY, M. at al, eds. *Introduction to Computer Graphics*, Course Notes for SIGGRAPH, 1995.

[3] BARAFF, D. *Physically based modeling: Rigid body simulation.* Disponível em `http://www.pixar.com/companyinfo/research/pbm2001/notes.pdf`. Acessado em 15 de março de 2006.

[4] BURTNYK, N.; WEIN, M. *Interactive Skeleton Thecniques for Enhancing Motion Dynamics in Keyframe Animation.* In: BEATTY, J.C.; BOOTH, K.S., eds. Computer Graphics, second edition, IEEE Computer Society Press, p.516-512, 1982. *Tutorial.*

[5] CAMARGO, J.T.F.; MAGALHÃES, L.P.; RAPOSO, A.B. *Fundamentos da Animação Modelada por Computador.* In: Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, 1995. *Tutorial.*

[6] CAMARGO, J.T.F. *Animação Modelada por Computador: Técnicas de Controle de Movimento.* Campinas, 1995. Dissertação (Doutorado) – Universidade Estadual de Campinas.

[7] CREMER, J.; KEARNEY,J.; WILLEMSEM.P; *Directable behavior models for virtual driving scenarios. Transactions of the Society for Comuter Simulation International*, 14(2), June 1997.

[8] DAVID, E. B.; PHYLLIP, H.G.; ANTHONY, A. A. *The ClockWorks: An Object-Oriented Computer Animation System.* Center of Interactive Computer Graphics, Rensselaer Polytechnic Institute, Troy, NY. EUROGRAPHICS, 1987.

[9] DEVILLERS, F; ONIKIAN, S. *A scenario language to orchestrate virtual world evolution.* In: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, San Diego. Eurographics Association, 265-275, 2003.

[10] DIJKSTRA, E. W.; LAMPORT, L.; MARIN, A.; et al. *On-the-fly Garbage Collection: An Exercise in Cooperation*, Communications of the ACM, 21,11 (November 1978), p. 966-975.

[11] FEIJÓ, B; PAGLIOSA P.A; Clua, E.W.G. *Visualização, Simulação e Games.* In: Atualizações em Informática, p. 127, 2006.

[12] FIUME, E.; TSICHRITZIS, D.; DAMI, L. *A Temporal Scripting Language for Object-oriented Animation*. In: Eurographics, Amsterdam, Netherlands, p.283-294, august 1987. *Proceedings*.

[13] FOLEY, J. et al. *Computer Graphics: Principles and Practice*, second edition. Addison-Wesley, 1992.

[14] FORMELLA, A.; KIEFER, P.P. *AniLan: An Animation Language*. In: Computer Animation, IEEE Computer Society Press, p.184-189, 1996. *Proceedings*.

[15] GIAMBRUNO, M. *3D Graphics & Animation*. New Riders, 2002.

[16] GOLDSTEIN, H. *Classical Mechanics*, second edition. Addison-Wesley, 1980.

[17] GREMINGER, M.A; NELSON, B.J. *Deformable Object Tracking Using the Boundary Element Method*. In: Computer Vision and Pattern Recognition, IEEE Computer Society Conference, 2003. *Proceedings*.

[18] HACKATHORN, R. J. Anima II: *A 3D Color Animation System*. In: FREEMAN, H., ed. *Tutorial and Selected Reading in Interactive Computer Graphics*, IEEE Computer Society Press, p.364-374, 1980.

[19] HUNTER, G. M. *Computer Animation Survey*. Computer and Graphics, Oxford, v.2, p.255-229, 1977.

[20] IERUSALIMSCHY, R. *Programming in Lua*. Lua.org, 2006.

[21] JAVALAN, SUN MICROSYSTEMS, INC. *Java Language Specification*, third edition, 1996-2005.

[22] JEFREMOV, A. AVI Preview. Disponível em `http://www.softpedia.com/get/Multimedia/Video/Encoders-Converter-DIVX-Related/AVI-Preview.shtml`. Acessado em 15 de agosto de 2006.

[23] KAFFE ORGANIZATION. *Kaffe Java Virtual Machine — Source Code*. Disponível em `ftp://ftp.kaffe.org/pub/kaffe/v1.1.x-development/kaffe-1.1.4.tar.gz`. Acessado em 29 agosto de 2006.

[24] LEE, G.S. *A general specification for scene animation*. In: SIBGRAPI, Rio de Janeiro, Brasil, october 1998. *Proceedings*.

[25] LINDHOLM, T.; YELLIN, F. *The Java Virtual Machine Specification*, second edition. Disponível em `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`. Acessado em 15 de julho de 2003.

[26] MAGALHÃES, L.P.; RAPOSO, A.B. *TOOKIMA: Uma Ferramenta Cinemática para Animação*. Relatório Técnico. Universidade Estadual de Campinas, DCA-FEE, março 1997.

[27] MÄNTYLÄ, M. *An Introduction to Solid Modeling*. Computer Science Press, 1988.

[28] NEWTON ENGINE. *Newtown Game Dynamics Documentation*. Dísponível em `http://www.newtondynamics.com/downloads/7765.pdf`. Acessado em 20 de julho de 2006.

[29] ODE ENGINE. *Open Dynamics Engine*. Disponível em `http://ode.org`. Acessado em 10 de junho de 2006.

[30] OLIVEIRA, L.L; PAGLIOSA, P.A. *A New Programming Environment for Dynamics-based Animation*. V Brazilian Symposium On Computer Games, 2006.

[31] PAGLIOSA, P.A. *Um Sistema de Modelagem Estrutural Orientado a Objetos*. São Carlos, 1998. Tese (Doutorado) – Escola de Engenharia de São Carlos – USP.

[32] PAGLIOSA, P.A. *Motor 3D para Visualização e Simulação Dinâmica de Corpos Elásticos*. Projeto de Pesquisa. Fundect N. 02/2005 — Edital Universal. 2005.

[33] PERLIN, K; GOLDBERG, A. *Improv: A system for scripting interactive actors in virtual worlds*. In: Proceedings of SIGGRAPH 96. ACM Press, 205-216, 1996

[34] PUEYO, X.; TOST, D. *Scanline: Um Sistema para Visualização de Imagens Foto-Realisticas*. Campinas, 1992. Dissertação (Mestrado) – Universidade Estadual de Campinas.

[35] PSCL. *The physics script language*. Disponível em `http://www.physicstools.org/docs/pscl`. Acessado em julho de 2006.

[36] RAMBAUGH, J. et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[37] RAPOSO, A.B. *Um Sistema Interativo de Animação no Contexto ProSim*. Campinas, 1996. Dissertação (Mestrado) – Universidade Estadual de Campinas.

[38] REEVES, W.T. *Particle Systems: A Thecnique for Modeling a Class of Fuzzy Objects*. Computer Graphics, v.17, n.3, p.359-376, 1983.

[39] REYNOLDS, C.W. *Computer Animation with Scripts and Actors*. Computer Graphics, v.16, n.3, p.289-296, 1982.

[40] ROGERS, D.F. *Procedural Elements for Computer Graphics*. McGraw-Hill, 1988.

[41] SIEBERT, F. *Hard Real-Time Garbage Collection in the Jamaica Virtual Machine*. In: The Sixth International Conference on Real-Time Computing Systems and Applications, Communications of the ACM, 1999. *Proceedings*.

[42] TALBOT, P.A. et al. *Animator: An On-line Two-dimensional Film Animation System*. In: FREEMAN, H., ed. *Tutorial and Selected Reading in Interactive Computer Graphics*, IEEE Computer Society Press, p.348-356, 1980.

[43] TANENBAUM, A.S. *Modderning Operating Systems*. Prentice-Hall, 1992.

[44] THALMANN, N.M. et al. *Computer Animation: Theory and Practice*. In: KUNII, T.L., ed. *Computer Science Workbench*, Springer-Verlag, Tokyo, 1985.

[45] THALMANN, N.M. *Three-dimencional Computer Animation: More an Evolution than a Motion Problem*. IEEE Computer Graphics and Applications, v.5, n.10, p.47-57, october 1985.

[46] TIBLJAS, R.; NIKOLIC, Z. *Fast Movie Processor*. Disponível em `http://www.winsite.com/bin/Info?500000009888`. Acessado em 10 de julho de 2006.

[47] WATKINS, C.D. et al. *Photorealism and Ray Tracing in C*. Prentice-Hall, 1992.

[48] WATT, A. *3D Computer Graphics*, second edition. Addison-Wesley, 2000.

[49] WAVISH, P; CONNAH, D. *Virtual actors that can perform scripts and improvise roles.* IEEE Computer Graphics and Applications, 1997.

[50] WILHELMS, J. *Toward Automatic Motion Control.* IEEE Computer Graphics and Applications, vol.7, n.4, p.11-22, 1987.

[51] WILHELMS, J. *Animals with Anatomy.* IEEE Computer Graphics and Applications, vol. 17, n. 3, p.22-30, 1997.

[52] WYVILL, B. *A Computer Animation Tutorial.* In: ROGERS, D.F; EARNSHAW, R.A., eds. *Computer Graphics Techniques: Theory and Pratice*, Springer-Verlag, New York, 1990.

[53] ZELEZNIK, R.C. *An Object-Oriented Framework for the Integration of Interactive Animation Techniques. Computer Graphics*, v.25, n.4, p.105-111, 1991.

[54] ZELTZER, D. *Towards an Integrated View of 3D Computer Animation. The Visual Computer*, v.1, n.4, pp.249-259, 1985.

Apêndice A - A Gramática da Linguagem de Animação

A seguir a sintaxe da linguagem de animação. Os símbolos *, +, e ? denotam zero ou mais, um ou mais, e optional, respectivamente.

Palavras reservadas

```
TOKENS
<CLASS: "class">
  | <MODIFIER: "public"
  | "protected"
  | "private"
  | "abstract"
  | "virtual"
  | "static"
  | "final"
   >
  | <NATIVE: "native">
  | <PRIMITIVE_TYPE_NAME: "int" | "float" | "char" | "bool" | "void">
  | <CONSTRUCTOR: "constructor">
  | <DESTRUCTOR: "destructor">
  | <INIT: "init">
  | <UPDATE: "update">
  | <FINALZIE: "finalize">
  | <RUN: "run">
  | <SUPER: "super">
  | <IF: "if">
  | <ELSE: "else">
  | <WHILE: "while">
  | <DO: "do">
  | <FOR: "for">
  | <RETURN: "return">
  | <BREAK: "break">
  | <CONTINUE: "continue">
  | <THIS: "this">
  | <NEW: "new" | "start">
  | <READ: "read">
  | <WRITE: "write">
  | <NULL: "null">
```

```
   | <BOOLEAN: "true" | "false">
   | <THROW: "throw">
   | <THROWS: "throws">
   | <TRY: "try">
   | <CATCH: "catch">
```

Literais

```
TOKEN
<INTEGER: <DECIMAL>  | <HEX>  | <OCTAL>>
    | <DECIMAL: ["1-"9"] (["0-"9"])*>
    | <HEX: "0"["x","X"] (["0-"9","a-"f","A-"F"])+>
    | <OCTAL: "0"(["0-"7"])*>
    | <FLOAT: (["0-"9"])+ ("."(["0-"9"])* (<EXP>)?   | <EXP>)  |
      "."(["0-"9"])+ (<EXP>)?
    >
    | <EXP: ["e","E"] (["+",-"])?  (["0-"9"])+>
    | <CHARACTER: "'"( ["'","/","n"]  | "/"["'",""","/","n"]) "'">
    | <STRING: """( [""","/","n"]  | "/"["'",""","/","n"])* """>
    | <NULL: "null">
```

Identificadores

```
TOKEN
    <IDENTIFIER: ("_" | <ALPHA>) ("_" | <ALNUM>)*>
    | <ALPHA: ["a-"z","A-"Z"]>
    | <DIGIT: ["0-"9"]>
    | <ALNUM: <ALPHA>  | <DIGIT>>
```

Operadores

```
<SCOPE_OP: "::">
   | <ASSIGN_OP: "+=" | -=" | "*=" | "/¯ | "%=" | "|=" | "&=" | "≙">
   | <LOGICAL_OR_OP: "||">
   | <LOGICAL_AND_OP: "&&">
   | <EQUAL_OP: "==" | "!=">
   | <REL_OP: "<" | "<=" | ">" | ">=">
   | <INC_DEC_OP: "++" | --">
   | <UNARY_OP: "!" | " ">
```

Produções

```
CompilationUnit:
    (ImportDeclaration)*
    | (TypeDeclaration)+

IncludeDeclaration:
    <Include> <STRING> ";"

TypeDeclaration:
```

```
    Modifiers ClassDeclaration

Modifiers:
    (<MODIFIER>)*

ClassDeclaration:
    <CLASS> <IDENTIFIER>
    ("<"TypeParameterList ">")?  (SuperclassSpecifier)?  ClassBody

TypeParameterList:
    Name ("," Name)*

SuperclassSpecifier:
    :  ClassName ("," ClassName)*

ClassName:
    Name

Name:
    <IDENTIFIER> (<SCOPE_OP> <IDENTIFIER>)* (< TypeParameterList >)?

ClassBody:
    "{" (MemberDeclaration)* "}"

MemberDeclaration:
    Modifiers
    (
      MemberTypeDeclaration
      | LOOKAHEAD(Type <IDENTIFIER> "(")
      | MethodDeclaration
      | ConstructorDeclaration
      | DestructorDeclaration
      | PropertyDeclaration
      | Declaration
    )

MemberTypeDeclaration:
    ClassDeclaration

Type:
    TypeName ("[]")?

TypeName:
    <PRIMITIVE_TYPE_NAME>
    | ClassName

MethodDeclaration:
    Type <IDENTIFIER> "("FormalParameters ")"
    (<NATIVE>)?  (MethodBody  | ";")
    | <INIT>(MethodBody  | ";")
    | <UPDATE>(MethodBody  | ";")
    | <FINALIZE>(MethodBody  | ";")
    | <RUN>(MethodBody  | ";")

FormalParameters:
    (FormalParameter ("," FormalParameter)*)?

FormalParemeter:
```

```
    Type <IDENTIFIER>

MethodBody:
    Block

ConstructorDeclaration:
    <CONSTRUCTOR> "("FormalParameters ")"(<NATIVE>)?  CtorBody

DestructorDeclaration:
    <DESTRUCTOR> "()"(<NATIVE>)?

CtorBody:
    (SuperclassInitializer)?
    MethodBody

SuperClassInitializer:
    ":"ClassName "("ExpressionList ")"

PropertyDeclaration:
    <PROPERTY> Type Name (<READ> = Name)?  (<WRITE> = Name)?

Declaration:
    Type Declarator (","Declarator)* ";"

Declarator:
    <IDENTIFIER> ("="Initializer)?

Initializer:
    ArrayInitializer
    | Expression

ArrayInitializer:
    "{"ExpressionList "}"

Block:
    "{"(Statement)* "}"

Statement:
    LOOKAHEAD(Type <IDENTIFIER>)
    LocalDeclaration
    | ExpressionList ";"
    | Block
    | PropertyBlock
    | AddIntoCollectionVariant
    | AnonymousNewExpression
    | IfStatement
    | WhileStatement
    | DoStatement
    | ForStatement
    | ReturnStatement
    | BreakStatament
    | ContinueStatement
    | ThrowStatement
    | TryStatement

LocalDeclaration:
   Declaration
```

```
ExpressionList :
    (Expression (","Expression)*)?

AddIntoCollectionVariant:
    Expression "{"(Expression;)* "}"

AnonymousNewExpression:
    <NEW> Name (ExpressionList?)  <CLASS> ClassBody

PropertyBlock:
    Expression "{"(AssignmentExpression;)* "}"

IfStatement:
    <IF> "("Expression ")"
      Statement
    (LOOKAHEAD(1) <ELSE> Statement)?

WhileStatement:
    <WHILE> "("Expression ")"
      Statement

DoStatement:
    <DO> Statement <WHILE> "("Expression ");"

ForStatement:
    <FOR> "("ForInit (Expression)?  ";"ForExpressions ")"Statement

ForInit:
    LOOKAHEAD(Type <IDENTIFIER>)
    LocalDeclaration
    | ForExpressions ";"

ForExpressions:
    ExpressionList

ReturnStatement:
    <RETURN> (Expression)?  ";"

BreakStatement:
    <BREAK> ";"

ContinueStatement:
    <CONTINUE> ";"

ThrowStatement:
    <THROW> Expression ";"

TryStatement:
    <TRY> Block
    (<CATCH> "("FormalParameter ")"Block)*
    (<FINALLY> Block)?

Expression:
    LogicalOrExpression (("=" | <ASSIGN_OP>) Expression)?

AssignmentExpression:
    LogicalOrExpression ("=" | <ASSIGN_OP>)?  Expression
```

```
LogicalOrExpression:
   LogicalAndExpression (<LOGICAL_OR_OP> LogicalAndExpression)*

LogicalAndExpression:
   InclusiveOrExpression (<LOGICAL_AND_OP> InclusiveOrExpression)*

InclusiveOrExpression:
   ExclusiveOrExpression (" |"ExclusiveOrExpression)*

ExclusiveOrExpression:
   AndExpression ("^ AndExpression)*

AndExpression:
   EqualityExpression ("&"EqualityExpression)*

EqualityExpression:
   RelationalExpression (<EQUAL_OP> RelationalExpression)*

RelationalExpression:
   AdditiveExpression (<REL_OP> AdditiveExpression)*

AdditiveExpression:
   MultiplicativeExpression (("+" | -") MultiplicativeExpression)*

MultiplicativeExpression:
   CastExpression (("*" | "/" | "%") CastExpression)*

CastExpression:
   UnaryExpression
   | <CAST> "<"ClassName ">("CastExpression ")"

UnaryExpression:
   PostfixExpression
   | ("+" | -" | <UNARY_OP>) UnaryExpression
   | <INC_DEC_OP> PrimaryExpression

PostfixExpression:
   PrimaryExpression (<INC_DEC_OP>)?

PrimaryExpression:
   PrimaryPrefix (PrimarySuffix)*

PrimaryPrefix:
   Literal
   | <THIS>
   | <SUPER> "."<IDENTIFIER>
   | "("Expression ")"
   | Name
   | NewExpression

PrimarySuffix:
   "."<IDENTIFIER>
   | "("ExpressionList ")"
   | "["Expression "]"

NewExpression:
   <NEW> TypeName ("("ExpressionList ")" | "["Expression "]")
```

```
Literal:
   <INTEGER>
    | <FLOAT>
    | <CHARACTER>
    | <STRING>
    | <BOOLEAN>
    | <NULL>
```

Apêndice B - As Instruções da MVA

Os valores `(a)` representam endereços de objetos, `(i)` inteiros, `(c)` caracter e `(f)` valores em ponto flutuantes.

A

aaload

`aaload`

Carrega na pilha de operandos uma referência para o objeto que reside em um vetor `v` indexado por `id` que reside no topo da pilha de operandos.
Configuração da pilha: `...,(a)v,(i)id ⇒ ..., (a)v[id].`

aastore

`aastore`

Armazena em um vetor `v` indexado por `id` um objeto `o` que reside no topo da pilha de operandos.
Configuração da pilha: `...,(a)v,(i)id,(a)o ⇒ ...; (a)v[id] = (a)o.`

aconst_null

`aconst_null`

Carrega na pilha de operandos a constante **`null`**
Configuração da pilha: `... ⇒ ..., (a)0.`

aload

`aload arg`

Carrega na pilha de operandos um objeto mantido na `arg`-ésima entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[arg].`

aload_0

`aload_0`

Carrega na pilha de operandos um objeto mantido na primeira entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[0].`

aload_1
`aload_1`
Carrega na pilha de operandos um objeto mantido na segunda entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[1].`

aload_2
`aload_2`
Carrega na pilha de operandos um objeto mantido na terceira entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[2].`

aload_3
`aload_3`
Carrega na pilha de operandos um objeto mantido na quarta entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[3].`

anewarray
`anewarray arg`
Carrega na pilha de operandos uma referência para um novo vetor `v` de referências para objetos da classe que reside na `arg`-ésima entrada do *pool* de constantes. A dimensão `s` de `v` encontra-se no topo da pilha.
Configuração da pilha: `...,(i)s ⇒ ...,(a)v.`

areturn
`areturn`
Retorna a referência para um objeto `o` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ....`

arraylength
`arraylength`
Carrega na pilha de operandos a dimensão do vetor `v` que se encontra no topo da pilha.
Configuração da pilha: `...,(a)v ⇒ ...,(i)size{v}.`

astore
`astore arg`
Armazena na `arg`-ésima entrada de `LOCALS` do *frame* corrente o objeto `o` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (a)LOCALS[arg] = (a)o.`

astore_0
`astore_0`
Armazena na primeira entrada de `LOCALS` o objeto `o` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (a)LOCALS[0] = (a)o.`

astore_1
`astore_1`
Armazena na segunda entrada de `LOCALS` o objeto `o` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (a)LOCALS[1] = (a)o.`

astore_2
`astore_2`
Armazena na terceira entrada de `LOCALS` o objeto `o` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (a)LOCALS[2] = (a)o.`

astore_3
`astore_3`
Armazena na quarta entrada de `LOCALS` o objeto `o` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (a)LOCALS[3] = (a)o.`

C

caload
`caload arg`
Carrega na pilha de operandos um caracter que reside em um vetor `v` indexado por `id` que reside no topo da pilha de operandos.
Configuração da pilha: `...,(a)v,(i)id ⇒ ..., (c)v[id].`

castore
`castore`
Armazena em um vetor `v` indexado por `id` um caracter `c` que reside no topo da pilha de operandos.
Configuração da pilha: `...,(a)v,(i)id,(c)c ⇒ ...; (c)v[id] = (c)o.`

D

dup
`dup`
Duplica uma palavra do topo da pilha de operandos.
Configuração da pilha: `...,(i)v ⇒ ...,(i)v,(i)v.`

dup2
`dup2`
Duplica duas palavras `v1` e `v2` do topo da pilha de operandos.
Configuração da pilha: `...,(i)v1,(i)v2 ⇒ ...,(i)v1,(i)v2,(i)v1,(i)v2.`

dup_x1
`dup_x1`
Duplica uma palavra `v1` abaixo do topo da pilha de operandos.
Configuração da pilha: `...,(i)v1,(i)v2 ⇒ ...,(i)v1,(i)v2,(i)v1.`

dup_x2
`dup_x2`
Duplica a palavra dupla que esta uma palavra abaixo do topo da pilha.
Configuração da pilha: `...,(f)v1, (i)v2 ⇒ ...,(f)v1,(i)v2,(f)v1.`

F

f2i

`f2i`

Converte o número real `v` que se encontra no topo da pilha de operandos em um número inteiro mantendo o resultado na própria pilha.
Configuração da pilha: `...,(f)v` ⇒ `...,(i)v`.

fadd

`fadd`

Soma dois números reais `v1` e `v2` que se encontram na pilha de operandos, mantendo o resultado na própria pilha.
Configuração da pilha: `...,(f)v1,(f)v2` ⇒ `...,(f)v1+v2`.

faload

`faload`

Carrega na pilha de operandos o número real que reside em um vetor `v` indexado por `id` mantido no topo da pilha de operandos.
Configuração da pilha: `...,(a)v,(i)id` ⇒ `..., (f)v[id]`.

fastore

`fastore`

Armazena em um vetor `v` indexado por `id` um número real `n` que reside no topo da pilha de operandos.
Configuração da pilha: `...,(a)v,(i)id,(f)n` ⇒ `...; (a)v[id] = (f)n`.

fcmp

`fcmp`

Compara os valores dos dois números reais `v1` e `v2` que se encontram no topo da pilha. Empilha o valor -1 se o segundo número real for maior que o primeiro, empilha o valor 0 se eles forem iguais ou empilha o valor 1 se o primeiro valor é maior que o segundo.
Configuração da pilha: `...,(f)v1,(f)v2` ⇒ `...,(i)sign(val1-val2)`.

fconst_0

`fconst_0`

Carrega na pilha o valor constante 0.0.
Configuração da pilha: `...` ⇒ `...,(f)0.0`.

fconst_1

`fconst_1`

Carrega na pilha o valor constante 1.0.
Configuração da pilha: `...` ⇒ `...,(f)1.0`.

fconst_2

`fconst_2`

Carrega na pilha o valor constante 2.0.
Configuração da pilha: `...` ⇒ `...,(f)2.0`.

fconst_m1

`fconst_m1`

Carrega na pilha o valor constante -1.0.
Configuração da pilha: `...` ⇒ `...,(f)-1.0`.

fdiv
`fdiv`
Divide dois números reais que se encontram na pilha de operandos e empilha o resultado.
Configuração da pilha: `...,(f)val1,(f)val2 ⇒ ...,(f)val1/val2`.

fload
`fload arg`
Carrega na pilha de operandos o valor real mantido na `arg`-ésima entrada de `LOCALS` do frame corrente.
Configuração da pilha: `... ⇒ ...,(f)LOCALS[arg]`.

fload_0
`fload_0`
Carrega na pilha de operandos o valor real mantido na primeira entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (f)LOCALS[0]`.

fload_1
`fload_1`
Carrega na pilha de operandos o valor real mantido na segunda entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[1]`.

fload_2
`fload_2`
Carrega na pilha de operandos o valor real mantido na terceira entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[2]`.

fload_3
`fload_3`
Carrega na pilha de operandos o valor real mantido na quarta entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (a)LOCALS[3]`.

fmul
`fmul`
Divide dois números reais `v1` e `v2` que se encontram na pilha de operandos, empilhando o resultado.
Configuração da pilha: `...,(f)v1,(f)v2 ⇒ ...,(f)v1*v2`.

fneg
`fneg`
Nega o valor do número real `v` que se encontra no topo da pilha de operandos, empilhando o resultado.
Configuração da pilha: `...,(f)v ⇒ ...,(f)~(i)v)`.

fpush
`fpush arg`
Carrega na pilha de operandos o valor real mantido na `arg`-ésima entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ...,(f)LOCALS[arg]`.

frem
`frem`
Carrega na pilha de operandos o resto da divisão entre dois números reais `v1` e `v2` mantidos no topo da pilha de operandos.
Configuração da pilha: `...,(f)v1,(f)v2 ⇒ ...,(f)v1%v2`.

freturn
`freturn`
Retorna um número real `v` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(f)v ⇒ ...`.

fstore
`fstore arg`
Armazena na `arg`-ésima entrada de `LOCALS` do *frame* corrente o valor real `v` mantido no topo da pilha de operandos.
Configuração da pilha: `...,(f)v ⇒ ...; (f)LOCALS[arg] = (f)v`.

fstore_0
`fstore_0`
Armazena na primeira entrada de `LOCALS` do *frame* corrente o valor real `v` mantido no topo da pilha de operandos.
Configuração da pilha: `...,(f)v ⇒ ...; (f)LOCALS[0] = (f)v`.

fstore_1
`fstore_1`
Armazena na segunda entrada de `LOCALS` do *frame* corrente o valor real `v` mantido no topo da pilha de operandos.
Configuração da pilha: `...,(f)v ⇒ ...; (f)LOCALS[1] = (f)v`.

fstore_2
`fstore_2`
Armazena na terceira entrada de `LOCALS` do *frame* corrente o valor real `v` mantido no topo da pilha de operandos.
Configuração da pilha: `...,(f)v ⇒ ...; (f)LOCALS[2] = (f)v`.

fstore_3
`fstore_3`
Armazena na quarta entrada de `LOCALS` do *frame* corrente o valor real `v` mantido no topo da pilha de operandos.
Configuração da pilha: `...,(f)v ⇒ ...; (f)LOCALS[3] = (f)v`.

fsub
`fsub`
Subtrai dois números reais `v1` e `v2` que se encontram na pilha de operandos, empilhando o resultado.
Configuração da pilha: `...,(f)v1,(f)v2 ⇒ ...,(f)(v1-v2)`.

G

getfield
`getfield arg`
Empilha na pilha de operandos o valor do atributo cujo símbolo reside na `arg`-ésima entrada do *pool* de constantes e cuja referência ao objeto encontra-se no topo da pilha.
Configuração da pilha: `...,(a)obj ⇒ ...,(a|i|c|f)obj[offset].`

getglobal
`getglobal arg`
Carrega na pilha de operandos a variável global ou estática de classe cujo símbolo reside na `arg`-ésima entrada do *pool* de constantes.
Configuração da pilha: `... ⇒ ..., (a|i|c|f)DATA[pool[arg].offset].`

goto
`goto arg`
Atualiza o contador de programa para o endereço *arg* do código objeto da função/método corrente.

H

halt
`halt`
Suspende a execução da função corrente.

I

i2f
`i2f`
Converte um valor inteiro `v` mantido no topo da pilha de operandos para uma representação em ponto flutuante.
Configuração da pilha: `...,(i)v ⇒ ...,(f)v.`

iadd
`iadd`
Soma dois valores inteiros `v1` e `v2` e mantidos no topo da pilha de operandos, empilhando o resultado na própria pilha.
Configuração da pilha: `...,(i)v1,(i)v2 ⇒ ...,(i)(v1+v2).`

iaload
`iaload arg`
Carrega na pilha de operandos uma valor inteiro mantido em um vetor `v` indexado por `id` que reside no topo da pilha de operandos.
Configuração da pilha: `...,(a)v,(i)id ⇒ ..., (i)v[id].`

iand
`iand`
Realiza o "e"lógico entre dois valores inteiros `v1` e `v2` que se encontram no topo da pilha de operandos.
Configuração da pilha: `...,(i)v1,(i)v2 ⇒ ...,(i)(v1 & v2).`

iastore
`iastore`
Armazena em um vetor `v` indexado por `id` um número inteiro `n` que reside no topo da pilha

de operandos.
Configuração da pilha: `...,(a)v,(i)id,(i)n ⇒ ...; (i)v[id] = (i)n`.

iconst_0
`iconst_0`
Carrega na pilha o valor constante 0.
Configuração da pilha: `... ⇒ ..., (i)0`.

iconst_1
`iconst_1`
Carrega na pilha o valor constante 1.
Configuração da pilha: `... ⇒ ..., (i)1`.

iconst_2
`iconst_2`
Carrega na pilha o valor constante 2.
Configuração da pilha: `... ⇒ ..., (i)2`.

iconst_3
`iconst_3`
Carrega na pilha de operandos o valor constante 3.
Configuração da pilha: `... ⇒ ..., (i)3`.

iconst_4
`iconst_4`
Carrega na pilha o valor constante 4.
Configuração da pilha: `... ⇒ ..., (i)4`.

iconst_m1
`iconst_m1`
Carrega na pilha o valor constante -1.
Configuração da pilha: `... ⇒ ..., (i)-1`.

idiv
`idiv`
Divide dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos, empilhando o resultado.
Configuração da pilha: `...,(i)v1,(i)v2 ⇒ ..., (i)(v1/v2)`.

if_acmpeq
`if_acmpeq arg`
Compara as duas referências `v1` e `v2` que se encontram no topo da pilha. Caso a primeira seja igual a segunda o contador de programa é direcionado para o deslocamento `arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v1,(a)v2 ⇒ ...`.

if_acmpne
`if_acmpne arg`
Compara as duas referências `v1` e `v2` que se encontram no topo da pilha. Caso a primeira seja diferente da segunda o contador de programa é direcionado para o deslocamento `arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v1,(a)v2 ⇒ ...`.

if_icmpeq
`if_icmpeq arg`
Compara os dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos. Caso o primeiro seja igual ao segundo o contador de programa é direcionado para o deslocamento `arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v1,(a)v2 ⇒ ...`.

if_icmpge
`if_icmpge arg`
Compara os dois números inteiros `v1` e `v2` que se encontram no topo da pilha. Caso o primeiro seja maior ou igual ao segundo o contador de programa é direcionado para o deslocamento `arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v1,(a)v2 ⇒ ...`.

if_icmpgt
`if_icmpgt arg`
Compara os dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos. Caso o primeiro seja maior que o segundo o contador de programa é direcionado para o deslocamento `arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v1,(a)v2 ⇒ ...`.

if_icmple
`if_icmple arg`
Compara os dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos. Caso o primeiro seja menor ou igual ao segundo o contador de programa é direcionado para o deslocamento `arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v1,(a)v2 ⇒ ...`.

if_icmplt
`if_icmplt arg`
Compara os dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos. Caso o primeiro seja menor que o segundo o contador de programa é direcionado para o deslocamento `arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v1,(a)v2 ⇒ ...`.

ifeq
`ifeq arg`
Analisa o valor `v` do topo da pilha. Caso seja 0, o contador de programa é atualizado para o `offset arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a|i|f)v ⇒ ...`.

ifge
`ifge arg`
Analisa o valor `v` do topo da pilha. Caso seja maior ou igual a 0 o contador de programa é atualizado para o `offset arg` do código objeto da função/método corrente
Configuração da pilha: `...,(a|i|f)v ⇒ ...`.

ifgt
`ifgt arg`
Analisa o valor `v` do topo da pilha. Caso seja maior que 0 o contador de programa é atualizado para o `offset arg` do código objeto da função/método corrente
Configuração da pilha: `...,(a|i|f)v ⇒ ...`.

ifle
`ifle arg`
Analisa o valor `v` do topo da pilha. Caso seja menor que 0 o contador de programa é atualizado para o `offset arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a|i|f)v ⇒ ...`.

iflt
`iflt arg`
Analisa o valor do topo da Analisa o valor `v` do topo da pilha. Caso seja menor ou igual a 0 o contador de programa é atualizado para o `offset arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a|i|f)v ⇒ ...`.

ifne
`ifne arg`
Analisa o valor `v` do topo da pilha. Caso seja diferente de 0 o contador de programa é atualizado para o `offset arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a|i|f)v ⇒ ...`.

ifnull
`ifnull arg`
Analisa o valor do topo da pilha. Caso seja igual à constante **`null`**, o contador de programa é atualizado para o `offset arg` do código objeto da função/método corrente.
Configuração da pilha: `...,(a)v2 ⇒ ...`.

iinc
`iinc arg`
Incrementa o valor da `arg`-ésima variável local inteira `v` da função/método corrente.

iload
`iload arg`
Carrega na pilha de operandos um inteiro mantido na `arg`-ésima entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (i)LOCALS[arg]`.

iload_0
`iload_0`
Carrega na pilha de operandos um inteiro mantido na primeira entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (i)LOCALS[0]`.

iload_1
`iload_1`
Carrega na pilha de operandos um inteiro mantido na segunda entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (i)LOCALS[1]`.

iload_2
`iload_2`
Carrega na pilha de operandos um inteiro mantido na terceira entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (i)LOCALS[2]`.

iload_3
`iload_3`
Carrega na pilha de operandos um inteiro mantido na quarta entrada de `LOCALS` do *frame* corrente.
Configuração da pilha: `... ⇒ ..., (i)LOCALS[3]`.

imul
`imul`
Divide dois números inteiros que `v1` e `v2` se encontram no topo da pilha de operandos do *frame* corrente, empilhando o resultado.
Configuração da pilha: `...,(i)v1,(i)v2 ⇒ ..., (i)v1*v2`.

ineg
`ineg`
Nega o valor do número inteiro `v` que se encontra no topo da pilha de operandos do *frame* corrente, empilhando o resultado.
Configuração da pilha: `...,(i)v⇒ ...,(i)(-v)`.

invokestatic
`invokestatic arg`
Invoca o método estático cujo símbolo é mantido na `arg`-ésima entrada do *pool* de constantes.

invokevirtual
`invokevirtual arg`
Invoca o método virtual mantido na `arg`-ésima entrada da `TPMV` da classe de **`this`**, residente no topo da pilha de operandos.
Configuração da pilha: `...,`**`this`** `⇒ ....`

ior
`ior`
Realiza a operação `ou`-lógico entre dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos, empilhando resultado lógico.
Configuração dapilha: `...,(i)v1,(i)v2 ⇒ ...,(i)(v1|v2)`.

ipush
`ipush arg`
Empilha na pilha de operandos do *frame* corrente o valor inteiro `arg`
Configuração da pilha: `... ⇒ ...,(i)arg`.

irem
`irem`
Empilha na pilha de operandos o resto da divisão de dois valores inteiros `v1` e `v2` que se encontram no topo da pilha de operandos do *frame* corrente.
Configuração da pilha: `...,(i)v1,(i)v2 ⇒ ...,(i)(v1%v2)`.

ireturn
`ireturn`
Retorna um valor inteiro `v` que se encontra no topo da pilha de operandos do *frame* corrente.
Configuração da pilha: `...,(i)v ⇒ ....`

istore
`istore arg`
Armazena na `arg`-ésima entrada de `LOCALS` do *frame* corrente o valor inteiro `v` que se encontra

no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (i)LOCALS[arg] = (i)v`.

istore_0
`istore_0`
Armazena na primeira entrada de `LOCALS` do *frame* corrente o valor inteiro `v` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (i)LOCALS[0] = (i)v`.

istore_1
`istore_1`
Armazena na segunda entrada de `LOCALS` do *frame* corrente o valor inteiro `v` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (i)LOCALS[1] = (i)v`.

istore_2
`istore_2`
Armazena na terceira entrada de `LOCALS` do *frame* corrente o valor inteiro `v` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (i)LOCALS[2] = (i)v`.

istore_3
`istore_3`
Armazena na quarta entrada de `LOCALS` do *frame* corrente o valor inteiro `v` que se encontra no topo da pilha de operandos.
Configuração da pilha: `...,(a)o ⇒ ...; (i)LOCALS[3] = (i)v`.

isub
`isub`
Subtrai dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos do *frame* corrente, empilhando o resultado.
Configuração da pilha: `..., (i)v1,(i)v2 ⇒ ...,(i)(v1-v2)`.

ixor
`ixor`
Realiza a operação `xor`-lógico entre dois números inteiros `v1` e `v2` que se encontram no topo da pilha de operandos, empilhando resultado lógico.
Configuração dapilha: `...,(i)v1,(i)v2 ⇒ ...,(i)(v1v̂2)`.

N

new
`new arg`
Carrega na pilha de operandos uma referência `o` para um novo objeto instância da classe cujo símbolo é mantido na `arg`-ésima entrada do *pool* de constantes.
Configuração da pilha: `... ⇒ ...,(a)o`.

newarray
`newarray arg`
Carrega na pilha de operandos uma referência `o` para um novo vetor de objetos `o` cujo tipo é representado pelo símbolo mantido na `arg`-ésima entrada do *pool* de constantes.
Configuração da pilha: `... ⇒ ...,(a)o`

P

pop
`pop`
Remove uma palavra `p` do topo da pilha de operandos do frame corrente.
Configuração da pilha: `...,(i)v ⇒ ...`.

pop2
`pop2`
Remove duas palavras `p1` e `p2` do topo da pilha de operandos do *frame* corrente.
Configuração da pilha: `...,(i)p1,(i)p2 ⇒ ...`.

putfield
`putfield arg`
Armazena o valor `v` em um atributo cujo símbolo é mantido na `arg`-ésima entrada do *pool* de constantes, relativo a um objeto `o`.
Configuração da pilha: `...,(a)o,(i|c|f|a)v` *Rightarrow* `...`.

putglobal
`putglobal arg`
Armazena na variável global ou estática de classe cujo símbolo é mantido na `arg`-ésima entrada do *pool* de constantes um valor `v` que reside no topo da pilha de operandos do *frame* corrente.
Configuração da pilha: `...,(c|i|f|a)v ⇒ ...; data[pool[arg].offset] = (c|i|f|a)v`.

R

return
`return`
Interrompe a execução das instruções do *frame* corrente.

render
`render`
Renderiza uma cena representada por um objeto cuja referência encontra-se o topo da pilha de operandos.

S

start
`start`
Cria e inicia a execução um seqüenciador `o`.
Configuração da pilha: `...` *Rightarrow* `...,(a)o`.

swap
`swap`
Troca duas palavras `p1` e `p2` mantidas no topo da pilha de operandos do *frame* corrente.
Configuração da pilha: `...,(i)p1,(i)p2 ⇒ ...,(i)p2,(i)p1`.

swap2
`swap2`
Troca palavra dupla mantidas no topo da pilha de operandos do *frame* corrente.
Configuração da pilha: `...,(f)p1,(f)p2` *Rightarrow* `...,(f)p2,(f)p1`.

wait_for

`wait_for`

Mantém uma ação, cuja referência encontra-se no topo da pilha de operandos, no estado `WAITING`.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo