Visualização, Simulação e Games

Bruno Feijó, Paulo Aristarco Pagliosa e Esteban Walter Gonzalez Clua

Abstract

This chapter presents the state of the art in technologies for visualization, artificial intelligence, and simulation of rigid body dynamics. It also describes how to use these technologies to develop applications that require intense real-time processing, in particular, 3D games. Besides giving an overview of basic techniques for computer graphics, graphics pipelines, OpenGL, GPU architectures, artificial intelligence, and physics simulation, the chapter also presents solutions for the integration of these basic techniques using models of 3D game engine architecture.

Resumo

Este capítulo apresenta o estado-da-arte em tecnologias para visualização, inteligência artificial e simulação dinâmica de corpos rígidos. Descreve também como empregar estas tecnologias no desenvolvimento de aplicações que requerem processamento em tempo real intenso, em particular jogos digitais 3D. Além de conter uma visão geral das técnicas básicas de computação gráfica, pipelines gráficos, OpenGL, arquiteturas de GPU, inteligência artificial e simulação física, o capítulo também apresenta soluções para a integração destas técnicas básicas usando modelos de arquitetura de motor de jogo 3D.

3.1. Introdução

O objetivo deste capítulo é apresentar o estado-da-arte nas tecnologias relacionadas à visualização, inteligência artificial e simulação dinâmica de corpos rígidos em tempo real, com enfoque à sua aplicação em jogos digitais, ou games, 3D[1]. Um dos motivos de interesse em pesquisa e desenvolvimento de games 3D é que este é atualmente considerado, em muitos países, setor estratégico dentro da indústria tecnológica. Isto se deve ao fato de que games podem ser desenvolvidos para servir como ferramentas interativas de visualização e simulação, usadas tanto para fins de entretenimento como para fins específicos (por exemplo, treinamento de pilotos em simuladores de vôo) em áreas diversas tais como engenharia (de estruturas, automobilística, aeroespacial), exploração de petróleo e medicina, entre outras aplicações.

[1] Os termos “jogo digital”, “jogo” e “game” são usados indistintamente ao longo do texto.

A complexidade do desenvolvimento de um game 3D deve-se à sua própria natureza multi e interdisciplinar e ao fato de que se espera que games atuais sejam capazes de prover, em tempo real, o maior grau de realismo possível, tanto no aspecto gráfico como no aspecto de simulação. O interesse pelo acréscimo de realismo físico em jogos digitais é resultado não somente do aumento de velocidade das CPUs, mas também da evolução das unidades de processamento gráfico (*graphics processing unit*s, ou GPUs), as quais implementam em hardware muitas das funções de visualização.

Como conseqüência desta folga da capacidade de processamento, modelos físicos e comportamentoe inteligentes mais realistas podem ser considerados sem prejudicar a resposta em tempo real. Além disso, já está disponível no mercado unidades de processamento de física (*physics processing units*, ou PPUs) tal como a AGEIA PhysX [AGEIA 2006], a qual, segundo o fabricante, foi projetada para permitir a aceleração de dinâmica de corpos rígidos, detecção de colisões, dinâmica de fluídos e dinâmica de certos corpos flexíveis, tais como roupas, entre outros.

Para lidar com a complexidade, há uma variedade de ferramentas disponíveis para desenvolvimento de jogos, tais como API´s, bibliotecas, frameworks e motores (*game engines*). Não cabe neste texto explorar a diferença entre estas diversas ferramentas, mas apenas salientar que os motores são aquelas que podem ser consideradas como as de mais alto nível na cadeia do desenvolvimento, pois: minimizam a necessidade de programação, abstraindo várias etapas do desenvolvimento através de editores e ferramentas gráficas; têm uma integração grande com ferramentas de desenvolvimento de artes, especialmente modeladores e editores de imagens; e permitem que grande parte do desenvolvimento customizado seja feita através de scripts.

Uma das vantagens dos motores de jogos em relação a outras ferramentas é que estes podem tornar mais simples o desenvolvimento de uma aplicação; por outro lado, possuem como desvantagem o fato de que as aplicações com eles construídas são mais específicas e, portanto, possuem um escopo mais limitado. Por isso, é comum encontrar motores gráficos, motores para jogos de primeira pessoa, motores para simulação em vista isométrica, etc. Existem também motores de propósito mais geral. Entretanto, são soluções geralmente mais caras e que, na prática, consistem em um conjunto de ferramentas distintas, adaptadas para cada situação.

Não há, entre desenvolvedores, uma definição consensual do que seja um motor de jogo digital. Assim, é mais produtivo listar os principais requisitos que estes devem ter:

- Encapsular da melhor forma possível códigos que podem ser re-utilizados para diversos projetos com alguma semelhança entre si;
- Permitir uma perfeita integração entre os recursos de arte (modelos, imagens, texturas, sons, etc.) com a programação;
- Tornar o desenvolvimento o mais independente possível de plataformas e tecnologias;

- Fazer com que a aplicação seja capaz de usar o máximo possível os recursos de hardware disponíveis (GPU, processamento distribuído, hardware de áudio, etc.);
- Permitir um gerenciamento de projeto adequado.

3.1.1. Arquitetura Básica de um Motor de Jogo Digital

A seguir propõe-se uma arquitetura de motores de jogos baseada em dois níveis de abstração, denominados de nível SDK (*software development toolkit*) e nível ferramental.

No nível SDK estão todas as bibliotecas de funções básicas, separadas pela sua funcionalidade. Apesar de haver uma interdependência grande entre elas, estão sob camadas de prioridades distintas, de modo que as de prioridade mais alta contêm funções mais elementares e que serão usadas pelos outros módulos. Neste nível encontram-se as bibliotecas de matemática, controladores de recursos, física, visualização, redes e áudio/vídeo. Módulos de IA são difíceis de ser padronizados em bibliotecas.

O nível ferramental é caracterizado não por ser formado por bibliotecas de funções, mas sim aplicativos que irão compor o motor como um todo e que usam o SDK para sua implementação. Componentes básicos da arquitetura, além do SDK, são: editores (de modelos, terrenos, fases, scripts), interpretadores de script, ambientes de testes, conversores e exportadores e otimizadores.

3.1.1.1. Arquitetura do SDK

A arquitetura do SDK é ilustrada no diagrama UML da Figura 3.1.

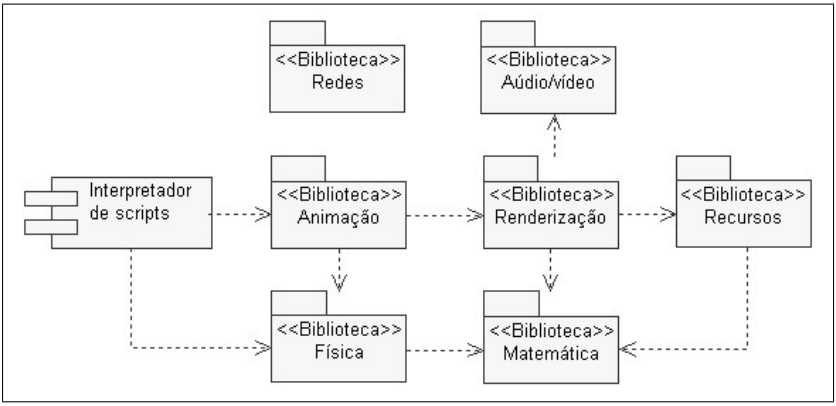


Figura 3.1. Arquitetura do SDK

O elemento básico do SDK é a biblioteca de matemática, amplamente usada pelos demais módulos. Esta consiste de funções matemáticas típicas, tais como operações com vetores, operações de matrizes, quaternions, etc. A fim de otimizar a eficiência dos cálculos, é interessante que as operações mais

básicas da biblioteca de matemática sejam implementadas em linguagem de montagem. Com isso, além de poder economizar algumas instruções matemáticas elementares (que em algumas situações são feitos milhões de vezes por frame), é possível lançar mão de otimizações de aceleração de operações matemáticas específicas de uma arquitetura. Exemplificando: a arquitetura SIMD (*single instruction*, *multiple data*), especificada pela Intel, possui registradores capazes de manipular mais de um número em ponto flutuante. Assim, é possível carregar um vetor inteiro em apenas um registrador e efetuar operações vetoriais com apenas um ciclo de máquina. A AMD, por sua vez, usa uma tecnologia semelhante, derivada da sua primeira proposta, chamada de 3DNOW!. Faz parte da biblioteca matemática detectar a arquitetura de CPU existente e acionar a biblioteca específica.

Juntamente com a biblioteca matemática pode haver uma segunda biblioteca, auxiliar, contendo funções implementadas em GPU. Há um ganho significativo em se fazer cálculos em GPU somente quando estes puderem ser processados em paralelo. Assim, para se fazer uma operação entre dois vetores uma única vez, o motor deve usar funções da biblioteca padrão, mas ao fazer operações vetoriais em vários pontos distintos (como por exemplo no tratamento de partículas), o motor pode carregar todos estes valores para o frame buffer da GPU e executar os cálculos requeridos. Em [Randima 2005] encontram-se descrições detalhadas sobre tipos de cálculos que são mais convenientes de ser feitos em CPU e GPU.

A biblioteca de renderização é responsável por abstrair a etapa de visualização. Nela está implementada todo o pipeline gráfico apresentado na Seção 3.2. Anexo a esta biblioteca pode haver uma biblioteca de *shaders*, contendo programas para definir diversos efeitos que podem ser acessados pelo desenvolvedor da aplicação final[2].

As funcionalidades da biblioteca de física (relacionadas à simulação dinâmica de corpos rígidos) são discutidas na Seção 3.3. As funções da biblioteca de física usam as funções da biblioteca de matemática. Na tentativa de realizar estes cálculos através de hardware, algumas funções podem lançar mão da biblioteca matemática em GPU.

O interpretador de scripts permite que o usuário possa ter um controle de recursos, objetos e cenário sem ter que utilizar o código fonte do SDK. Este interpretador possui como principal tarefa permitir acesso e algumas operações com os objetos e variáveis do sistema. Sua principal utilização se dá para implementação da inteligência artificial e da lógica da aplicação. Em alguns casos, permite acesso a algumas funcionalidades da biblioteca de física e tam-

[2] Um *shader* é um programa que executa em GPU, podendo ser de dois tipos: *vertex shader* e *pixel shader*. Um *vertex shader* é uma função de processamento gráfico usada para adicionar efeitos especiais a objetos de uma cena 3D através de operações matemáticas que transformam os dados dos vértices dos objetos, tais como posição, cor, textura, iluminação, etc. Um *pixel shader*, por sua vez, é uma função gráfica que calcula efeitos relativos à cor de pixels de um frame.

bém pode ser usado para a maioria dos controles relacionados a animações de objetos dinâmicos.

A biblioteca de áudio/vídeo e a biblioteca de recursos implementam funções para manipular diversos formatos de áudio (`mp3`, `wav`, `midi`, etc.), streamings de vídeo, malhas 3D e imagens.

Finalmente, a biblioteca de redes implementa funções para tráfego de mensagens. No caso de se tratar de um motor para plataformas massivas, deve haver uma biblioteca para processamento e tratamento cliente-servidor.

3.1.1.2. Arquitetura Ferramental

Os principais componentes da arquitetura ferramental de um motor são mostrados no diagrama UML da Figura 3.2 e comentados a seguir.

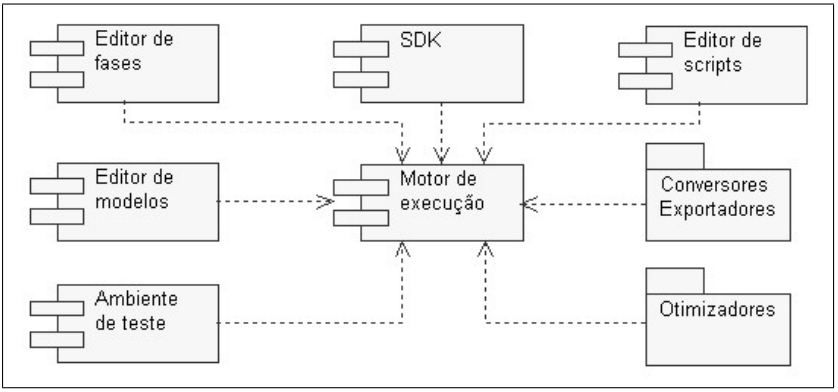


Figura 3.2. Componentes da arquitetura ferramental de um motor

- Editor de fases: ambiente onde são inseridos todos os modelos geométricos que compõem uma fase da aplicação. Estes modelos podem ser tanto os estáticos, os quais sofrem algum tipo de pré-processamento pelo módulo de renderização, como objetos dinâmicos, os quais podem possuir associados scripts que definem seu comportamento. Neste módulo também são inseridas as luzes estáticas e dinâmicas de uma fase.
- Editor de terrenos: em geral, esta ferramenta pode estar inserida dentro do editor de fase. Permite que o usuário gere e manipule os mapas de altura, bem como as suas camadas de texturas. Existem diversas ferramentas dedicadas exclusivamente a editar terrenos, tais como o Pandromeda Mojo World Generator, Vue D´Espirit ou o Terragen.
- Editor de modelos: um motor pode e deve permitir que o usuário crie os modelos dentro de ambientes especializados para isto, tais como 3DStudio MAX ou Maya. Entretanto, é comum que haja um editor para criar modelos mais simples ou adaptar os modelos criados em ferramentas de terceiros, dentro dos padrões do motor.

- Ambientes de teste: ambiente em que a aplicação é executada, permitindo no entanto que o usuário possa alterar variáveis definidas nos scripts em tempo de execução.
- Editor de scripts: apesar de um script poder ser escrito em qualquer editor de texto, um motor pode fornecer um ambiente de programação integrado, de forma que o usuário possa facilmente associar o código a elementos dinâmicos.
- Conversores e exportadores: é fundamental que o motor possua uma série de ferramentas que possibilitem ler arquivos desenvolvidos em programas de terceiros. Muitas vezes são *plug-ins* desenvolvidos em linguagens de scripts fornecidos por estes próprios ambientes.
- Otimizadores: como visto na Seção 3.2, é necessário que o motor execute uma série de etapas de pré-processamento para otimização do processo de visualização, tais como geração de *level of details*, BSP´s, *light maps*, portais, etc. Normalmente estas ferramentas podem estar dentro dos editores de fase.

Existe ainda o motor de execução, que consiste no programa que executa toda a aplicação e é distribuído juntamente com todos os recursos. Em última instância, este componente é o que se pode chamar de motor por essência, mas não chega a ser uma ferramenta propriamente dita.

A arquitetura ferramental pode variar bastante, dependendo do tipo de aplicação para qual um motor é desenvolvido. A arquitetura de SDK, contudo, é mais universal e é comum que possa ser usada em várias instâncias de motores diferentes. Neste capítulo abordam-se os conceitos envolvidos no desenvolvimento de algumas bibliotecas da arquitetura do SDK, enfatizando os aspectos de visualização e simulação de corpos rígidos.

3.1.2. Organização do Capítulo

O restante do capítulo é dividido em três seções organizadas como segue. A Seção 3.2 trata da visualização de cenas 3D. Inicialmente, apresentam-se as etapas de um processo, ou pipeline, de renderização em tempo real de modelos geométricos poligonais. Em seguida, introduz-se a OpenGL, API gráfica que pode ser utilizada na implementação do pipeline de renderização. Na seqüência, duas operações do pipeline muito importantes em aplicações tempo real, *culling* e recorte (*clipping*), são descritas com mais detalhes. A arquitetura de GPUs e os conceitos básicos de programação de GPUs também são abordados. Por fim, apresenta-se e discute-se a arquitetura de software padrão de um motor gráfico.

A Seção 3.3 é dedicada à simulação dinâmica de corpos rígidos. Os conceitos básicos da mecânica de uma partícula e de um sistema de partículas são inicialmente apresentados: posição, velocidade, aceleração, força, torque, momentos linear e angular, centro de massa, energias cinética e potencial, restrições e equação de movimento de Newton. A seguir, descreve-se um sistema para determinação em computador do comportamento de uma coleção

de corpos rígidos submetidos à ação de forças e torques, incluindo resposta a colisões. Na seqüência, descrevem-se a arquitetura de um motor de física e as ações executadas por este para simulação dinâmica de uma cena em um determinado instante de tempo. A seção finaliza com uma introdução ao PhysX SDK, um motor de física desenvolvido pela AGEIA.

A Seção 3.4 trata da aplicação de técnicas de IA em jogos digitais. Primeiramente, mostra-se a relação da IA de jogos com a IA clássica. Depois, as técnicas são classificadas como sendo do tipo consagradas, tendências e fronteira do desenvolvimento. Por fim, duas técnicas consagradas são explicadas: busca de caminho A* e máquinas de estado finito.

3.2. Visualização

Esta seção trata das etapas do processo de geração de imagens a partir de uma base de modelos geométricos e definições de cenário. Primeiramente é exposto o processo como um todo, que é normalmente denominado de *pipeline* gráfico. A seguir, são apresentados os conceitos de API gráficas, bem como uma proposta de arquitetura de um pipeline utilizando a OpenGL. Algoritmos de *culling* e recorte (*clipping*) para otimização da visualização também são discutidos. Finalmente, é apresentada uma arquitetura típica de um hardware gráfico dedicado, bem como uma breve introdução aos conceitos de programação de shaders.

3.2.1. Pipeline Gráfico

O termo *pipeline* pode ser traduzido como processo de fabricação ou construção. O pipeline de montagem de um carro, por exemplo, consistirá em todas as etapas da montagem de um automóvel. Cada uma destas etapas também pode ser denominada de estágio. O gargalo de um pipeline, por sua vez, consiste na etapa do processo que possui maior demora e que irá definir o tempo mínimo necessário para poder completar a fabricação. Otimizar um pipeline consiste em determinar qual é o gargalo do processo e tentar diminuí-lo. Dentro do conceito de um pipeline é possível dizer que processos podem ser executados em paralelo. Assim, numa linha de produção de automóveis, o estágio que coloca o motor no chassi não precisa esperar que um carro fique totalmente pronto para colocar outro motor. Assim que acabar de colocar em um carro, pode-se começar a colocar o motor em outro, enquanto o primeiro continua na linha de produção, em outros estágios de fabricação.

Em computação gráfica, seja em tempo real ou não, o pipeline é definido normalmente por três estágios: *aplicação*, *geometria* e *rasterização*. O estágio de aplicação, como o nome diz, é a etapa que está implementada no programa ou no motor. Enquanto os outros dois estágios estarão parcial ou totalmente implementados pelo hardware, o que dificulta a interferência do processo por parte do desenvolvedor, este é uma estágio puramente implementado em software. Justamente por isto é nele que o desenvolvedor possui maior controle. Desta maneira, aqui estará implementada grande parte da lógica da aplicação, tal como a física, o controle de entrada de dados, a inteligência artificial, ani-

mação e grande parte dos algoritmos de *culling*. No término deste estágio, a geometria da cena será enviada para o estágio seguinte, que já não será capaz de alterar o cenário e os objetos, mas apenas tratar da sua visualização. Na biblioteca de renderização, o motor deverá permitir que o desenvolvedor tenha acesso completo a este estágio. Num primeiro momento, pode-se pensar que quanto menos polígonos o estágio de aplicação enviar para o estágio de geometria, mais eficiente será a visualização. Entretanto, pode ocorrer que uma suposta otimização leve mais tempo do que simplesmente tratar um polígono desnecessário para a imagem final. Ao implementar os métodos de *culling* esta é uma variável que deve ser levada em conta.

O estágio de geometria será responsável por tratar individualmente os polígonos e vértices no seu processo de visualização. Este estágio pode ser dividido em vários subestágios: *transformação*, *iluminação*, *projeção*, *recorte* e *transformação para coordenadas de tela*.

Na descrição de uma cena, a posição dos vértices de cada objeto costuma estar em coordenadas locais, isto é, em relação a um sistema de coordenadas local próprio de cada objeto. Além disso, cada objeto pode estar descrito no cenário através de coordenadas globais, de forma a posicioná-lo corretamente no espaço. Para eficiência de algumas das etapas do pipeline, convém que a câmera seja posicionada na origem e que todos os polígonos do cenário sejam descritos neste novo sistema de coordenadas. O subestágio de transformação consiste em transformar as coordenadas de todos os vértices para este novo sistema de coordenadas espaciais. Este processo é relativamente simples de se efetuar, pois basta multiplicar cada vértices por uma matriz de transformação.

Existem dois momentos do pipeline em que se pode tratar da iluminação: (1) no estágio de geometria, quando se calcula a iluminação para cada vértice; (2) no estágio de iluminação de pixel, quando se calcula a cor final de um pixel. No pipeline tradicional, o valor de iluminação para cada vértice é calculado usando um modelo de iluminação local. Depois, a cor final do pixel é calculada no estágio de rasterização, através de um processo de interpolação entre vértices. No caso das placas gráficas programáveis (Seção 3.2.4), estes momentos de iluminação ficam bastante flexíveis. Em qualquer caso, porém deve-se adotar um modelo de iluminação local, como, por exemplo, o modelo de Phong, o qual define que a iluminação total de um ponto sobre um objeto é dada por três componentes:

$$I_{\text{total}} = I_{\text{ambiente}} + I_{\text{difusa}} + I_{\text{especular}},$$

cuja fórmula simplificada, considerando-se apenas uma fonte pontual de luz, pode ser dada por

$$I_{\text{total}} = I_a K_a C_d + f_{\text{at}}\, I_{\text{luz}} [K_d C_d (\mathbf{N} \cdot \mathbf{L}) + C_s K_s (\mathbf{R} \cdot \mathbf{V})^{n_s}],$$

onde I_a é a intensidade de iluminação ambiente do cenário; I_{luz} é a intensidade de iluminação da fonte pontual de luz (e f_{at} é o seu fator de atenuação); C_d

e C_S são a cor difusa e a cor especular do objeto, respectivamente; K_a, K_d e K_s são os coeficientes de luz ambiente, difusa e especular, respectivamente; **N** é a normal da superfície do objeto no ponto; **L** é o vetor de luz que possui a direção do ponto para a luz; **V** é o vetor da posição da câmera ao ponto, **R** é o vetor de reflexão (igual a **L** espelhado em relação a **N**) e n_s é o expoente de reflexão especular do objeto.

Uma vez calculada a iluminação dos vértices, estes seguem para o estágio de projeção. Até este momento os vértices estavam descritos no espaço 3D. Nesta etapa os vértices serão levados para um sistema de coordenadas do plano de projeção da câmera, ou seja, uma das dimensões será "perdida". Em computação gráfica os tipos de projeção mais usualmente empregados são a projeção perspectiva e a projeção paralela ortográfica.

Após a projeção, algums polígonos podem estar no interior, interceptar ou estar no exterior da área de visão. Apenas devem prosseguir no pipeline os polígonos que são total ou parcialmente visíveis. Assim sendo, aqueles que estiverem no exterior da área de visão devem ser descartados. O estágio de recorte (*clipping*) irá determinar a parte visível dos polígonos que interceptarem a área de visão, criando novos polígonos que estarão totalmente em seu interior. Finalmente, o estágio de transformação para coordenadas de tela irá mapear todas as coordenadas 3D dos vértices, que já estão no plano de projeção da câmera, para as coordenadas de tela. Estas coordenadas devem ser números inteiros e não mais números reais.

Ao terminar o estágio de geometria, tem-se uma série de vértices, já coloridos e iluminados, sobre o plano de projeção da câmera. A rasterização, último estágio do pipeline, consiste em preencher o “interior” de cada polígono, pixel a pixel, realizando uma interpolação para que a mudança da cor seja gradativa. Pode haver polígonos sobrepostos. Para tanto, antes de “pintar” um pixel, deve-se fazer um teste para saber se a sua profundidade é maior do que a do pixel que já está pintado. Este estágio deve ser necessariamente feito num hardware dedicado, pois realizar esta interpolação através de software seria lento.

As placas de vídeo possuem uma área denominada de *frame buffer*. Tudo o que estiver nesta área será visto na tela. Assim, se a rasterização for efetuada sobre o frame buffer, além do usuário ver imagens sendo formadas aos poucos, irá perceber que alguns pixels são pintados de uma cor, logo depois por outra, já que quando um polígono está por cima de outro, o hardware gráfico pinta o novo por cima. Para tanto, é comum que a rasterização ocorra numa área de memória secundária, chamada de *back buffer*. Após o término da renderização de uma imagem, os dados do *back* buffer são colocados para o frame buffer principal.

3.2.2. APIs Gráficas e OpenGL

Para uma aplicação poder se valer dos recursos disponibilizados por uma determinada placa gráfica, seria necessário que esta fosse implementada utili-

zando as funções específicas da GPU em questão. Entretanto, dada a grande diversidade de modelos disponíveis, desenvolver uma aplicação desta forma se tornaria impossível. Para tal há APIs gráficas, as quais consistem em bibliotecas capazes de acessar os recursos do hardware, abstraindo-lhes qualquer tipo de dependência em baixo nível. Além disso, quando o programador acessar algum recurso que não está disponível numa placa específica, as APIs serão capazes de emular tais funcionalidades através de software.

As APIs mais utilizadas para programação gráfica são a OpenGL (*Open Graphics Library*) e o DirectX. Enquanto a primeira é multi-plataforma, o segundo é voltado apenas para plataforma Microsoft Windows. O DirectX também se caracteriza por ser uma coletânea de diversas APIs não apenas voltadas para gráfico, mas também para áudio, redes e dispositivos de entrada. Já a OpenGL é apenas voltada para a programação gráfica. Como a OpenGL não cuida do tratamento de janelas, entrada de dados ou menus, existem diversas bibliotecas que irão facilitar estas operações. Uma das mais usadas é a biblioteca GLUT (*OpenGL Utility Tool*), que também é multi-plataforma.

O funcionamento básico da OpenGL é descrito através de definição de estados: antes de pedir para a placa gráfica renderizar um conjunto de polígonos, acionam-se diversos estados que irão caracterizar esta renderização, tais como atributos do material, texturas, modelo de iluminação, etc. Estes estados funcionam de forma acumulativa, ou seja, ao ajustar um estado, outras definições anteriores não serão necessariamente descartadas.

Um programa que usa OpenGL deve começar criando o chamado *rendering context*, que consiste em associar uma janela do sistema operacional à OpenGL. Feito isto, o núcleo de uma aplicação de visualização em tempo real típica deve consistir de um laço que verifica constantemente se houve alguma mudança na janela (movimento ou redimensionamento, por exemplo). Se isto ocorreu, então a posição da janela deve ser atualizada e a cena redesenhada. Deve-se verificar também neste laço se houve alguma entrada de dados provinda do teclado, mouse ou joystick. Caso tenha ocorrido, devem-se atualizar dados da cena (como mover a câmera, aplicar um passo de física para um objeto, calcular um passo de um personagem, etc.) e logo em seguida redesenhar a cena. Finalmente, uma aplicação deve conter a função de renderização. Esta função será chamada em todos os momentos que for necessário redesenhar a cena. Basicamente, esta deve limpar a tela, ajustar os estados de renderização necessários para um conjunto de polígonos, desenhar estes polígonos, ajustar outros estados de renderização, desenhar outros polígonos e assim por diante até que terminem todos os polígonos que constituem uma cena. Como foi discutido anteriormente, para evitar que se veja a imagem sendo formada durante a renderização, esta é feita toda no *back* buffer. A última etapa da renderização consiste em transferir o *back* buffer para o *front* buffer.

Dentro da arquitetura de um motor, a biblioteca de SDK de renderização consiste na implementação básica da função de renderização, deixando claro o espaço para inserir a visualização de novos elementos e de novos recursos.

Além disso, o SDK poderá implementar as funções de inicialização e o laço principal. Deve-se salientar que este laço é o esqueleto de um motor, uma vez que é nele que serão chamadas as funções de atualização para física, IA, entrada de dados, etc. A estrutura básica de um programa em OpenGL é descrita a seguir.

- Inicialização. Consiste na criação da janela e do rendering context e inicialização de estados da OpenGL (tais como câmera, cor de fundo e iluminação). As operações de inicialização de outros componentes, como do motor de física, também são feitas aqui.
- Laço principal. O laço principal destina-se a capturar e tratar eventos (de janela, entrada de dados, etc.) e redesenhar a cena. O tratamento de um evento pode resultar na execução de um passo da simulação física, aplicação da IA, execução de script, finalização da aplicação, etc. O redesenho da cena envolve a configuração da câmera, limpeza da tela com a cor de fundo, ajuste dos estados de renderização, desenho dos polígonos que compõem o modelo geométrico dos atores e *swap* de buffers.
- Finalização. Consiste na liberação de todos os recursos criados e usados pelo motor. As operações de finalização de outros componentes também são feitas aqui.

Diversos efeitos podem ser inseridos no processo de renderização, como visto mais adiante na seção de programação de GPUs. Estes efeitos serão inseridos na configuração de estados da função de redesenho da cena. A seguir é apresentado um programa ilustrativo do uso da OpenGL. Apesar de muito elementar e sem detalhes, corresponde ao esqueleto de um motor. Para clareza de compreensão, não estão sendo colocados os parâmetros de funções, definições de variáveis e implementação de funções padrões.

```
// Função que inicializa o motor.
void InitEngine(...)
{
  // Inicializa a OpenGL.
  g_hWnd = window;
  context = wglCreateContext(...); // cria o rendering context
  SizeOpenGLScreen(width, height); // determina o tamanho da janela
  // Inicializa a física (veja Seção 3.3.3).
  InitPhysics();
  // Inicializa a IA.
  InitAI();
  ...
}

// Laço principal do motor.
// A função RenderScene() é constantemente chamada.
void MainLoop()
{
  for (;;) // o laço é infinito
```

```
    // Neste exemplo o sistema se baseia em mensagens para notificar a ocorrência
    // de eventos à aplicação.
    if (GetMessage(msg))
    {
      if (msg == QUIT) // mensagem para terminar programa
        break;
      // Se a mensagem é uma notificação que uma tecla válida foi pressionada,
      // haverá um tratamento específico para esta ação. O motor deverá logo
      // em seguida aplicar os passos de física e IA e renderizar a cena.
      if (msg == KEY_XXX)
        HandleKey_XXX(...);
      ...
      RunPhysics(); // (veja Seção 3.3.3)
      RunAI();
      RenderScene();
    }
}

// Função que renderiza uma cena.
void RenderScene()
{
  // Configura câmera.
  glLoadIdentity();
  gluLookAt(0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
  // Se houve transformação da câmera, esta ação pode ser aplicada aqui.
  SetupCamera();
  // Limpa a tela
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  // Configura estados de renderização para os polígonos que serão plotados em
  // seguida. Alguns estados podem ser ativados e outros poderão ser desativados.
  glEnable(GL_ATRIBUTO_XXX); glDisable(GL_ATRIBUTO_YYY);
  // Desenha um triângulo com vértices em (2,0,0),(0,2,0) e (0,0,2). A função
  // glColor3f() define a cor de um vértice. A OpenGL interpolará as cores ao
  // plotar o triângulo. Se houver textura no polígono, aqui também serão dadas suas
  // coordenadas de textura. A função glBegin() inicia o envio de geometria para a
  // OpenGL e glEnd() encerra. O parâmetro GL_TRIANGLES indica que a cada
  // três vértices tem-se um polígono.
  glBegin(GL_TRIANGLES);
  glColor3f(1.0, 0.0, 0.0);
  glVertex3f(2.0, 0.0, 0.0);
  glColor3f(0.0, 1.0, 0.0);
  glVertex3f(0.0, 2.0, 0.0);
  glColor3f(0.0, 0.0, 1.0);
  glVertex3f(0.0, 0.0, 2.0);
  glEnd();
  // Em cenas mais complexas, serão definidos novos estados e desenhados mais
  // polígonos (veja a função DrawActor() na Seção 3.3.3).
  ...
  // Depois de renderizar tudo realiza o swap de buffers.
  SwapBuffers();
}
```

```
// Função que finaliza o motor.
void ReleaseEngine(...)
{
  // Deleta o rendering context.
  wglCreateContext(...);
  // Finaliza a física (veja Seção 3.3.3).
  ReleasePhysics();
  // Finaliza a IA.
  ReleaseAI();
  ...
}

// Função principal
int main(...)
{
  // Chama função para criar janela no Windows.
  window = CreateMyWindow(...);
  // Inicializa o motor sobre a janela recém criada.
  InitEngine(...);
  // Chama o laço principal.
  MainLoop();
  // Finaliza o motor.
  ReleaseEngine(...);
}
```

Neste exemplo, na função `RenderScene()` os vértices que compõem um triângulo são listados no corpo da função. Entretanto, na arquitetura de um motor, convém implementar um método que automatiza esta operação, dando como entrada a malha tridimensional num formato gerado por algum modelador comercial. Além disso, esta função poderá também automatizar o processo de definição de estados da OpenGL, baseado na definição dos materiais que compõem este objeto. Aqui se deve prestar atenção também para o tratamento de *culling*. Assim, antes de enviar os triângulos para a OpenGL, pode-se chamar uma função específica para tratar a otimização para este tipo de objeto específico. Desta forma, é comum implementar funções separadas para renderizar objetos estáticos da cena, objetos dinâmicos, terrenos e panoramas, uma vez que possuem métodos de otimização diferentes.

Além disso, para desenhar uma malha composta por muitos polígonos, a OpenGL possui métodos mais eficientes para listar os vértices do que a função `glBegin()`. Um destes métodos consiste em usar *vertex arrays*. Após organizar corretamente todos os vértices de um objeto 3D dentro de um vetor, com apenas uma chamada de função, tais como `glDrawArrays()` ou `glMultiDrawArrays()`, a OpenGL irá desenhar todo o conjunto de polígonos correspondentes.

3.2.3. Recorte e Culling

Ao projetar polígonos sobre o plano de projeção da câmera, alguns polígonos cairão totalmente dentro da área da tela e outros cairão parcialmente

dentro, ou seja, apenas uma parte do polígono estará na tela de projeção. Para estes polígonos é necessário realizar o recorte, que consiste em criar novas arestas e vértices, de forma a não mandar para a rasterização as partes dos polígonos que estão fora da tela de projeção.

Na medida que a capacidade de processamento gráfico aumenta, modelos e cenários mais complexos vão sendo criados e elaborados. De certa forma, pode-se afirmar que sempre haverá modelagens e cenários suficientemente complexos, por melhor que seja a capacidade de processamento disponível. Desta forma, sempre serão necessárias e apreciadas as técnicas de aceleração, por melhor que seja o hardware gráfico disponível.

Diminuir o número de polígonos a serem tratados é uma das formas de atingir esta otimização. É justamente nisto que o *culling* consiste: jogar fora tudo que não interessa para um determinado instante (*cull* em inglês significa "refugo, escolher, selecionar de dentro de um grupo"). Assim, o que as técnicas de *culling* terão de fazer é saber escolher polígonos adequadamente, de forma que numa determinada situação, restem apenas polígonos que realmente importam para a visualização a partir do ponto em que a câmera se encontra. Pode-se pensar também da seguinte maneira: quais polígonos de uma cena devem ser enviados para o pipeline, pois possuem uma grande chance de serem vistos na imagem final?

Existem muitos algoritmos que farão este tipo de escolha, como se verá mais adiante. Em muitos casos a eficiência deste procedimento estará atrelada ao tipo de agrupamento e ordem de polígonos (um terreno possui uma distribuição de polígonos completamente diferente de um personagem ou de um labirinto).

O *culling* pode ser feito em qualquer estágio do pipeline gráfico. Entretanto, pode-se pensar que quanto antes um polígono for descartado, melhor. Desta maneira, o melhor momento para se realizar o descarte de polígonos indesejados é no estágio de aplicação. Ressalte-se que um método de *culling* não anula outro: podem-se ter os efeitos somados em muitos casos.

3.2.3.1. Backface Culling

O primeiro conjunto de polígonos triviais a serem descartados são aqueles que se encontram oclusos pelo próprio objeto, ou seja, atrás da própria geometria (*backface*). Num primeiro momento, em objetos "bem comportados", pode-se assumir que as faces visíveis de um objeto são apenas aquelas cujas normais apontam para o "lado" do observador.

Realizar este *culling* no estágio de geometria irá poupar o rasterizador de receber esta classe de polígonos indesejados. O algoritmo é simples: ao realizar a projeção dos polígonos com a matriz de projeção de câmera, a normal dos polígonos possuirá apenas duas direções possíveis: apontado para dentro da tela e apontado para fora. Em última instância, realizar o *culling* neste caso corresponderá apenas a um teste de sinal de um dos componentes do vetor normal do polígono projetado.

Outro algoritmo mais interessante permite a eliminação destes polígonos ainda no estágio de aplicação: cria-se um vetor dado pela direção de um ponto qualquer pertencente ao polígono (por exemplo, um dos vértices) à posição do observador. Se o ângulo formado entre este vetor e a normal do polígono for maior do que 90^o, então este polígono simplesmente é ignorado e não é mais enviado para o estágio de geometria.

Entretanto, nem sempre será conveniente descartar as *backfaces*. Se um cenário possui espelhos ou superfícies reflexivas, por exemplo, polígonos ocultos podem ser vistos. Da mesma forma, caso um objeto possua transparência, possibilitará a visibilidade de polígonos que estariam ocultos por ele de serem vistos. Para solucionar isto, pode-se criar uma flag que controla se uma face é visível “pelas costas” ou não.

3.2.3.2. Culling Baseado numa Estrutura Hierárquica de Objetos

Entende-se por *bounding volume* como um volume capaz de envolver um objeto por completo. Os tipos mais comuns são *bounding sphere* (menor esfera que envolve um objeto) e o *bounding box* (menor caixa que envolve um objeto), que por sua vez pode ser de dois tipos: faces paralelas ou perpendiculares ao sistema de coordenadas globais (*axis-aligned bounding boxes*, ou AABB) ou faces não alinhadas ao sistema de coordenadas globais (*oriented bounding boxes*, ou OBB). Estes elementos, além de servirem para os algoritmos de *culling*, serão importantes em outras operações, tais como testes de colisão.

Criar uma estrutura hierárquica de *bounding volumes* para uma cena corresponde a agrupar um conjunto de *bounding volumes* por outros *bounding volumes*, fazendo com que cada nó filho seja um volume que se encontra no interior do volume do seu pai. Assim sendo uma árvore hierárquica será uma estrutura em que os nós correspondem a *bounding volumes* e as folhas a objetos geométricos. Pode-se chamar de grafo de cena a uma estrutura hierárquica deste tipo.

A primeira e mais intuitiva das idéias que os *bounding volumes* trazem para realizar o *culling* consiste em, ao invés de testar se um determinado objeto está dentro do campo de visão de uma câmera, realiza-se apenas o teste para o volume que o envolve. Isto pode significar uma simplificação enorme, pois por menor que seja o modelo geométrico, dificilmente possuirá menos polígonos do que o seu *bounding volume*. O algoritmo de *culling* neste caso pode ser reduzido a um teste de colisão. No caso do volume estar totalmente dentro do campo de visão da câmera (região esta também chamada de *view frustrum*), então toda a geometria que se encontra dentro do volume é passada para o rasterizador. Neste caso, pode-se inclusive desativar o recorte de polígonos, uma vez que todos os elementos se encontram totalmente dentro do campo de visão da câmera. Caso apenas uma parte do volume possua interseção com o campo de visão da câmera, realiza-se um novo cálculo de interseção com cada um dos volumes que são filhos imediatos do volume em questão. Caso os nós filhos sejam a própria geometria de um objeto, então todos os seus polígonos

são repassados ao rasterizador. Neste caso, entretanto, deve-se deixar ativo o recorte, pois haverá polígonos que possuem interseção com as bordas do *view frustrum* da câmera e precisarão ser recortados.

3.2.3.3. Binary Space Partition (BSP)

Os algoritmos de *octree* e BSP são métodos mais estruturados para *culling* hierárquico e utilizam o conceito de *bounding volumes*. Estas técnicas, no entanto, possuem como inconveniente um pré-processamento prévio (construção da estrutura hierárquica) que pode vir a ser custoso e demorado, dependendo da complexidade da cena. Isto torna estes métodos soluções pouco flexíveis para objetos dinâmicos, que sofrem transformações espaciais numa cena, mas por outro lado trazem uma otimização enorme para objetos estáticos. Desta maneira, é comum em aplicações que lançam mão de BSPs dividir a cena em duas categorias: a parte estática — da qual será criada uma estrutura de BSP através de um pré-processamento prévio — e a parte dinâmica, que não irá usufruir da aceleração deste método e que portanto não estará presente na estrutura hierárquica.

A idéia básica das BSP consistem, como o nome diz, em repartir recursivamente o espaço em duas partes. Existem duas variações deste algoritmo: *axis aligned* e *polygon aligned*. Será discutido apenas o método conhecido como *polygon aligned*, já que é mais comum na implementação de jogos 3D.

O algoritmo inicia-se escolhendo um polígono que será o nó raiz de toda a estrutura hierárquica. O plano onde se encontra o polígono dividirá o espaço em duas partes e toda a geometria que define uma cena estará apenas num dos dois subespaços resultantes. A seguir, para cada um dos subespaços criados escolhe-se um novo polígono que lhe pertença e se realiza uma nova subdivisão, restrita apenas ao subespaço em que se encontra este polígono. Isto será feito recursivamente, até que todos os polígonos estejam presentes na árvore BSP. Note-se que poderão existir várias BSP possíveis para uma mesma cena, definidas de acordo com o critério de escolha dos polígonos que servirão como divisores de espaço. As melhores árvores — e que portanto permitirão maior aceleração — deverão ser esparsas. Uma árvore que possuir como altura o mesmo número de elementos da cena será o pior caso que se pode ter.

É necessário criar uma convenção do que é frente e o que é atrás de um polígono para que a árvore seja construída com coerência. Sempre que um polígono estiver presente em dois subespaços, este deverá ser “quebrado” em duas partes, com a quebra exatamente na linha de interseção do mesmo com o plano divisor. A seguir, descreve-se passo a passo a criação da árvore BSP correspondente à cena composta por corredores da Figura 3.3, onde as linhas tracejadas indicam a extensão do plano que contém cada parede.

Escolhe-se arbitrariamente o polígono 1 para ser raiz de toda a BSP. Os polígonos 2, 3 e 4 estão do seu lado esquerdo e os polígonos 5, 6, 7, 8 e 9 do seu lado direito. Para o subespaço da esquerda de 1, escolhe-se para raiz o

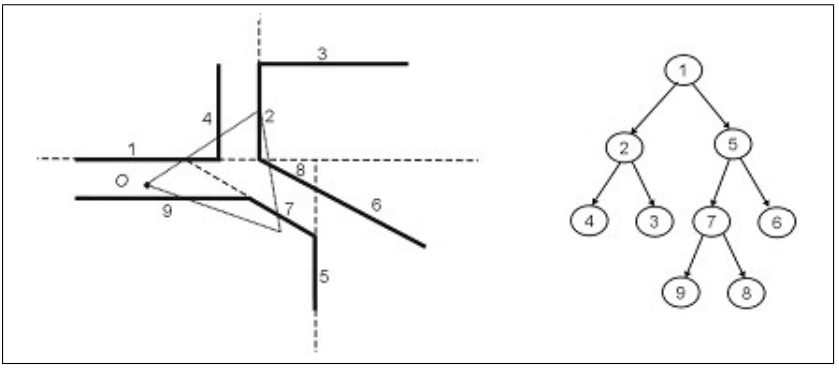


Figura 3.3. Esquerda: Vista de topo de uma cena. Direita: Uma BSP da cena

polígono 2, que terá como nós filhos 4, que está à sua esquerda e 3 que está à sua direita. Para o subespaço da direita de 1 escolhe-se 5 como raiz. Note que o plano que contém 5 intercepta um polígono, que será então dividido em dois: 8 e 6. Assim, 6 estará à direita de 5 e como não há mais nenhum polígono nos dois subespaços formados pela extensão do plano que contém 6, este é um nó terminal da árvore. Para o lado esquerdo de 5, escolhe-se o polígono 7 como raiz. Este terá 8 à sua direita e 9 à sua esquerda.

Para a visualização do cenário representado pela BSP deve-se percorrer a estrutura de trás para frente, isto é, primeiro determina-se de qual lado o observador O se encontra em relação ao plano do nó raiz. A seguir, o algoritmo irá mandar todos os polígonos que estão no subespaço oposto de O em relação ao plano, para serem desenhados. Logo em seguida é plotado o próprio polígono do nó raiz e em seguida manda-se desenhar todos os polígonos que estão no mesmo subespaço do observador. Plotar todos os polígonos do subespaço onde se encontra o observador ou o subespaço oposto, corresponde na verdade a uma chamada recursiva da função, pois se pode tratar o subespaço representado pela parte direita ou esquerda da árvore BSP como uma nova árvore BSP. Esta recursão tem fim quando a subárvore é apenas um nó terminal. O pseudocódigo abaixo mostra como pode ser implementado este processo.

```
Desenha_BSP(O, nó_Árvore_BSP)
  se nó_Árvore_BSP é folha
    Plota_Poligono(nó_Árvore_BSP)
  senão
    Testa de que lado O está em relação ao plano de nó_Árvore_BSP
    se O estiver à direita do plano
      Desenha_BSP(O, nó_Arvore_BSP.esquerda)
      Plota_Poligono(nó_Árvore_BSP)
      Desenha_BSP(O, nó_Arvore_BSP.direita)
    se O estiver à esquerda do plano
      Desenha_BSP(O, nó_Árvore_BSP.direita)
```

```
    Plota_Poligono(nó_Arvore_BSP)
    Desenha_BSP(O, nó_Arvore_BSP.esquerda)
```

Este algoritmo obedece à seguinte propriedade: todos os polígonos serão plotados na ordem de trás para frente, ou seja, não existe a preocupação de que num determinado momento um polígono deva ser plotado por trás de algum outro que já tenha sido plotado. Esta propriedade permite que seja utilizado o algoritmo do pintor: um polígono pode ser inteiramente desenhado por cima do anterior, sobrepondo-se completamente a este, possibilitando que o teste de Z-buffer seja totalmente dispensável.

Para um observador no ponto O da Figura 3.3 (o triângulo representa o *view frustrum*), o algoritmo se comportaria da seguinte maneira: como O está à direita do plano do polígono 1, o algoritmo desenha antes o subespaço da esquerda. Ao fazer a chamada recursiva, vê-se que O está à esquerda do plano de 2, portanto desenha-se o subespaço da direita de 2, que é apenas o polígono 3. Como o nó de 3 é uma folha, desenha-se este polígono. Logo em seguida desenha 2 e depois o subespaço da esquerda de 2, que é dado apenas por 4. Antes de chamar a recursão para o lado direito de 1, desenha-se o próprio polígono 1. Ao desenhar o subespaço da direita, vê-se que O está à esquerda de 5, portanto manda-se desenhar o subespaço da direita, que é apenas o polígono 6. Após desenhar o polígono 5, verifica-se que O está à esquerda de 7, portanto desenha-se o polígono 8, em seguida o polígono 7 e por último o polígono 9. Assim, a ordem dos polígonos desenhados será: 3, 2, 4, 1, 6, 5, 8, 7, 9.

Até agora, entretanto, a BSP não implica em nenhuma redução de polígonos para a visualização. Como uma BSP pode ser utilizada para realizar o *culling*? A idéia é relativamente simples: o *view frustrum* do observador pode ser representado por um conjunto de polígonos que definem um volume (levando em consideração o far plane da câmera este volume é de tamanho finito). Caso não haja interseção dos planos do *view frustrum* com o polígono do nó raiz da árvore BSP, todo o subespaço oposto ao observador pode ser desprezado, já que está fora do alcance de visibilidade, sendo por certo que apenas polígonos no mesmo subespaço são visíveis. O pseudocódigo apresentado anteriormente ficaria da seguinte forma:

```
Desenha_BSP(O, nó_Árvore_BSP)
  se nó_Árvore_BSP é folha
    Plota_Poligono(nó_Árvore_BSP)
  senão
    Testa de que lado O está em relação ao plano de nó_Árvore_BSP
    se O estiver atrás do plano
      Desenha_BSP(O, nó_Arvore_BSP.direita)
      se há interseção do view frustrum com nó_Árvore_BSP
        Desenha_BSP(O, nó_Arvore_BSP.esquerda)
      Plota_Poligono(nó_Árvore_BSP)
    se O estiver na frente do plano
      Desenha_BSP(O, nó_Árvore_BSP.esquerda)
```

```
se há interseção do view frustrum com nó_Árvore_BSP
  Desenha_BSP(O, nó_Arvore_BSP.direita)
Plota_Poligono(nó_Arvore_BSP)
```

A Figura 3.4 ilustra como é feito o *culling* para o exemplo de BSP das Figura 3.3. Verifica-se que o plano do polígono 1 não possui nenhuma interseção com o *view frustrum* de O. Neste caso, todo o subespaço da sua esquerda é desprezado por completo, incluindo-se o próprio polígono 1. Ao continuar percorrendo a árvore vê-se que 9 também está fora do alcance da câmera e portanto é desprezado da geometria. Perceba-se por outro lado que a árvore BSP não fornece a solução ótima. No exemplo, o polígono 5 não seria desprezado, embora ele acabe sendo totalmente obstruído por 7 e portanto sendo desnecessário para a posição em que O se encontra.

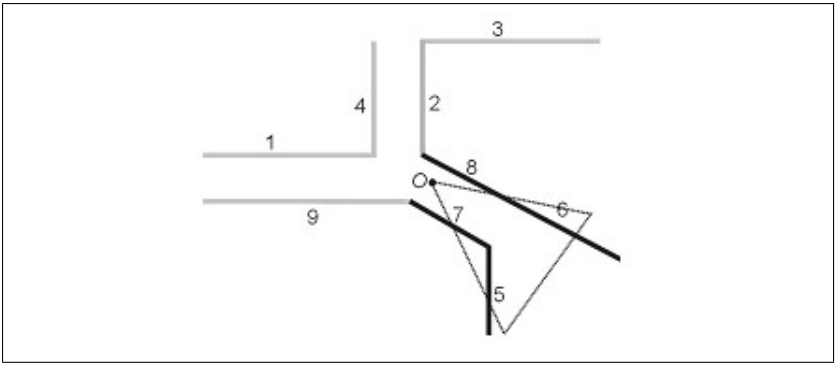


Figura 3.4. Exemplo de culling com árvore BSP

3.2.3.4. Portais

Cenários de ambientes fechados possuem uma característica importante: por mais extensa que seja a área modelada, em cada local onde o observador se encontra apenas um número relativamente pequeno de polígonos podem ser vistos. Isto ocorre porque as paredes funcionam como elementos que obstruem grande parte dos elementos presentes. Este fato é bastante tentador para o desenvolvedor tirar proveito através de alguma técnica de *culling*.

A técnica dos portais permite justamente eliminar polígonos que estejam sendo obstruídos por grandes polígonos, tipicamente paredes presentes em ambientes fechados. O pré-processamento dos portais consiste basicamente em dividir o cenário em células, como ilustrado na Figura 3.5. Várias implementações sugerem que estas células sejam convexas, uma vez que tornará o processo de recorte mais simples e rápido (uma célula é convexa quando, tomados dois pontos quaisquer do seu interior, a reta que os une não intercepta nenhum polígono das paredes da célula). Esta restrição, no entanto, pode acarretar no surgimento de um número grande de portais, mesmo para um cenário pequeno e com poucas salas.

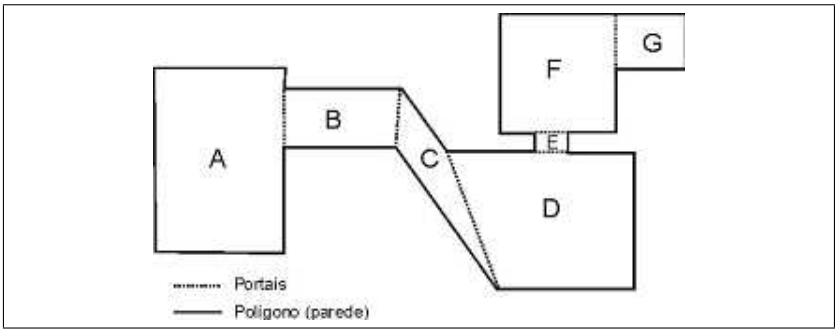


Figura 3.5. Portais podem ser vistos como polígonos especiais. Uma célula apenas pode ver outra adjacente a ela através de um portal

Criada a divisão de células, procura-se determinar os portais, que são regiões bidimensionais e invisíveis, comunicando uma célula com a sua célula vizinha. Estes podem ser também entendidos como polígonos especiais: enquanto as paredes são polígonos normais e ao serem renderizados calculam-se a iluminação e a textura deles, no caso dos portais, a visualização será tratada de forma especial. Normalmente, estes portais coincidirão com as aberturas de portas e janelas. É importante frisar que uma célula apenas poderá ver outra através de um portal.

Faz-se necessário uma estrutura de dados que seja capaz de armazenar estas informações: paredes e outros polígonos pertencentes a uma mesma célula devem estar agrupados, sendo que este grupo possui também a informação de quem são suas células adjacentes e os portais de comunicação. Esta estrutura de células pode ser adaptada à estrutura de uma BSP.

Os portais serão normalmente utilizados em conjunto com a técnica de PVS (*potentially visible set*), que consiste numa tabela que possui a informação, para cada uma das células do cenário, se é possível ou não ver cada uma das outra células existentes. O cálculo de construção da PVS costuma ser caro, dependendo especialmente da precisão que se deseja obter e do número de células do cenário. O PVS consiste numa tabela com valores booleanos, dizendo se a partir de alguma posição de uma célula X é possível ver alguma parte de uma célula Y. Existem várias formas de se realizar este cálculo, sendo os mais utilizados os algoritmos de *point sampling* e o de *shadow volume*.

O algoritmo de *point sampling* funciona da seguinte forma: discretiza-se uma série de pontos espaciais pertencentes a uma determinada célula A. Para cada um destes pontos traça-se uma reta para cada um dos pontos discretizados de cada uma das i células do cenário. Caso alguma reta possua uma interseção vazia com os polígonos da cena, então a célula A pode enxergar a célula i. Caso nenhuma reta possua interseção vazia, não é possível ver a célula i a partir da célula A. É justamente essa discretização um dos fatores que pode ou não encarecer este pré-processamento.

Uma das principais vantagens da utilização de portais consiste na facilidade e simplicidade da implementação da visualização. O algoritmo resumido para isto é o seguinte:

```
render(Célula* célula, Câmera* câmera, View* view)
  para todos os polígonos da célula onde está o observador faça
    se o polígono não é portal
      Plote o polígono recortadoo para a tela
    senão
      Crie um novo_view utilizando o portal e o view corrente
      render(célula_vizinha, câmera, novo_view)
```

Como a célula corrente é um elemento convexo, os polígonos que compõem esta célula podem ser plotados em qualquer ordem, sendo que não ocorrerá sobreposição. O único recorte que será necessário para os mesmos será com as bordas da tela.

O termo *view* define um plano que limita o alcance do campo de visão da câmera, Figura 3.6. O primeiro *view* da recursão consiste no próprio plano que define o alcance original máximo da câmera. A recursão dos portais consiste sobretudo em ir diminuindo o tamanho do *view frustrum*, fazendo com que a base do triângulo formado seja o novo *view*. Este plano é na verdade o próprio portal que se está tratando nesta etapa da recursão.

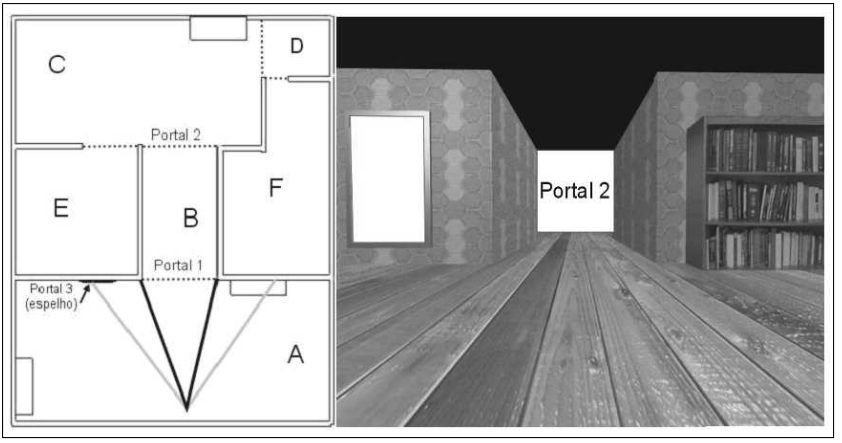


Figura 3.6. Esquerda: Portais criados para um ambiente fechado. Direita: parte do processo de visualização utilizando o algoritmo de portais

Criar um novo *view* utilizando o portal como referência pode ser feito de duas maneiras distintas:

- Utilizando polígonos 2D. Cria-se a partir da interseção do polígono que define o portal (já projetado sobre o plano de projeção da câmera, e portanto já com a correção de perspectiva), com o *view* corrente. O cálculo de interseção de dois polígonos 2D é muito simples, especialmente

neste caso, onde um polígono (no caso, o novo *view*) estará sempre dentro do outro. O único que deverá ser feito é realizar o recorte da parte do polígono do portal que eventualmente estiver na parte de fora do *view* original. Assim, caso a interseção seja vazia, então não é possível ver a célula referenciada pelo portal, a partir da posição em que o observador se encontre.

- Utilizando *view frustrum*. Neste caso, o método consiste em criar um novo *view frustrum* (formado por planos 3D), utilizando como base a coordenada espacial da posição da câmera e os vértices que definem o polígono do portal.

3.2.4. Arquitetura e Programação de GPUs

As arquiteturas de GPUs são muito variadas e cada vez mais diversos modelos surgem. Além disso, talvez hoje os hardwares que mais avançam na indústria sejam as GPUs. Desta forma, é difícil apresentar uma arquitetura geral. Entretanto, há elementos comuns entre todas as placas aceleradoras gráficas atuais e são estes que serão discutidos neste texto. Conhecer estes elementos é fundamental para que se possa explorar ao máximo a performance de uma aplicação. Além disso, como estes hardwares são programáveis, ao desenvolver programas para as GPUs é importante conhecer sua estrutura interna.

O componente mais fundamental de uma GPU é a sua memória. Aliás, os primeiros dispositivos gráficos eram basicamente uma memória de vídeo com alguns poucos recursos de aceleração. Esta memória será usada para várias finalidades e é dividida em várias partes. A primeira delas é o *frame buffer*, que é uma região da memória onde serão escritos os valores dos pixels que serão mostrados na tela. Pode-se assumir que a tela é um espelho do frame buffer: tudo o que for escrito nesta memória será mostrado no monitor. Diretamente conectado ao frame buffer está a controladora de vídeo, que irá converter o sinal digital presente nesta memória para o sinal analógico que será enviado ao monitor.

Ao renderizar uma imagem, os polígonos são plotados seqüencialmente. Assim, se estes forem desenhados diretamente no frame buffer a aplicação gráfica apresentará imagens que vão sendo formadas aos poucos. Além disso, muitos dos polígonos serão logo em seguida sobrepostos por outros que estão à sua frente, ocorrendo o desaparecimento dos primeiros. Desta forma, se o desenho for efetuado no frame buffer o usuário irá perceber polígnos "piscando", ou seja, aparecendo e desaparecendo muito rapidamente. Assim sendo, outro componente da GPU é o *back buffer*. Toda a renderização será feita nesta área de memória. Apenas ao terminar uma imagem por completo é que o conteúdo desta será transferido para o frame buffer principal, também conhecido como *front buffer*.

Ao plotar polígonos no *back* buffer, pode ocorrer de que um polígono se sobre-escreva a outro, pois na descrição da cena, o segundo está na frente do primeiro. Assim, é necessário que haja uma memória que armazene a pro-

fundidade do último polígono plotado em cada pixel. Antes de plotar um novo polígono, será feito um teste para ver se sua profundidade é maior ou menor que o valor escrito nesta memória. Caso seja maior, este polígono será totalmente descartado do pipeline. Esta memória de profundidade é denominada *Z-buffer*.

O *stencil buffer* é uma área da memória usada para operações de máscara: serão ajustados alguns valores desta memória e posteriormente, ao desenhar no *back* buffer, algumas áreas poderão ser escritas, outras não, dependendo dos valores ajustados no *stencil* buffer.

Finalmente, há mais uma área de memória chamada de *accumulation buffer*. Esta permite que várias imagens sejam desenhadas ao mesmo tempo, possibilitando que haja sobreposição entre elas. Assim, nesta área serão compostas imagens formadas a partir de duas ou mais, fundamental para criar *motion blur* ou *depth of field*.

Além da memória, o hardware gráfico é dividido em duas regiões: o *bloco de geometria* e o *bloco de rasterização*. Grosso modo, o primeiro bloco irá tratar os vértices e o segundo cuidará dos fragmentos, que são os pixels que ainda não foram mostrados na tela. Uma das principais razões da eficácia das GPUs consiste na sua arquitetura de processadores paralelos. Vários vértices podem ser tratados simultaneamente, bem como vários pixels também o são.

A arquitetura básica de uma GPU é ilustrada na Figura 3.7 e comentada a seguir.

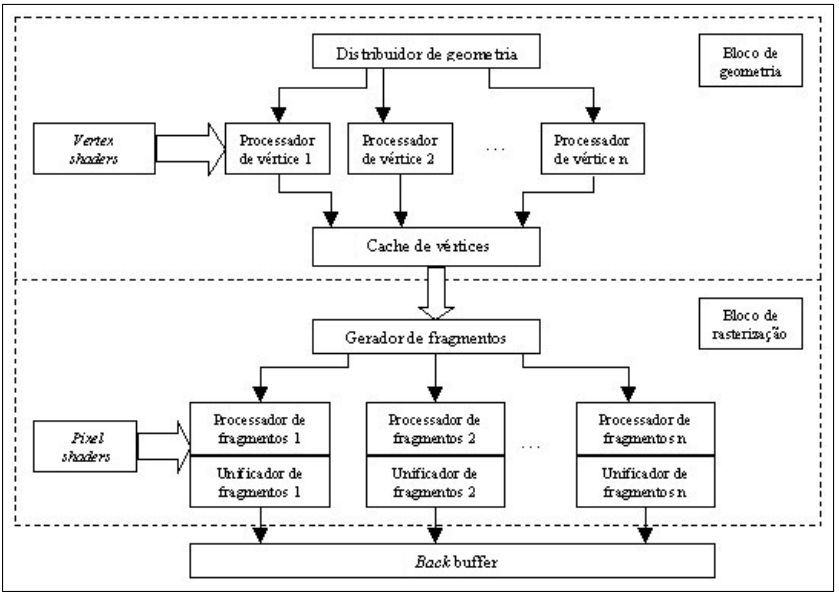


Figura 3.7. Arquitetura padrão de uma GPU

O *distribuidor de geometria* se encarregará de distribuir os vértices para cada um dos processadores de vértices disponíveis. O *processador de vértice* fará, em operações de hardware, as operações do pipeline gráfico corresponte ao estágio de geometria: transformação, iluminação de vértice, projeção, recorte e transformação para coordenadas de tela. Através dos *vertex shaders* o programador pode escrever um pequeno programa que alterará as operações implementadas no hardware. O *cache de vértices* é uma memória que recebe os vértices já processados, devidamente colocados no plano de projeção. O *gerador de fragmentos* irá distribuir polígonos a serem preenchidos e rasterizados. Isto será efetuado por outros processadores, denominados de *processadores de fragmentos*. Opcionalmente pode-se chamar um programa, chamado de *pixel shader*, que alterará o processo de rasterização padrão do hardware, permitindo que o programador insira uma série de efeitos. O *unificador de fragmentos* irá fazer um teste antes de escrever o candidato a pixel no *front* buffer, verificando através do Z-buffer, se este fragmento está escondido por outro pixel pertencente a um polígono que está na frente do polígono que deu origem ao fragmento.

No pipeline apresentado no início da seção, as etapas de processamento de vértice e rasterização de fragmentos eram pré-estipuladas pela arquitetura do hardware. De fato, durante os primeiros anos do advento dos hardwares gráficos estes passos foram pré-determinados pelo fabricante. Atualmente, as etapas de processamento de vértice e processamento de fragmento podem ser alteradas pelo desenvolvedor várias vezes na sintese de uma mesma imagem, permitindo que diferentes tratamentos e efeitos sejam dados durante a renderização. Para desenvolver estes programas há diversas linguagens de alto nível, sendo as mais conhecidas a Cg (*C for Graphics*), a HLSL (*High Level Shader Language*) e a *OpenGL Shader Language* [Rost 2004] (adotada neste texto).

3.2.4.1. Vertex Shaders

O processador de vértices é responsável por efetuar principalmente as seguintes operações: transformação da posição do vértice, geração de coordenadas de textura para a posição do vértice, iluminação sobre o vértice, operações para determinar o material a ser aplicado ao vértice. Os *vertex shaders* serão programas que irão interferir e alterar de alguma maneira todos ou algumas destas tarefas. Assim sendo, todo *vertex shader* possui como entrada um vértice e alguns de seus atributos e produz como saída este mesmo vértice com os atributos modificados (até mesmo a posição do vértice pode ser alterada). Dentro deste programa, frequentemente serão utilizados a normal do vértice, a coordenada de textura, a iluminação referente ao vértice e algumas variáveis globais configuradas previamente. Efeitos que tipicamente exigem o uso de *vertex shaders* são geração de texturas procedurais, efeitos de iluminação *per-vertex*, animação procedural em vértices, *displacement mapping*, etc.

Algumas variáveis já são pré-definidas pelo *OpenGL Shader Language*, em especial aquelas que serão amplamente usadas num programa. Assim, para

referir-se às coordenadas do vértice pode-se usar a variável `gl_Vertex` e para se referir à normal do mesmo usa-se a variável pré-definida `gl_Normal`. Da mesma forma, `gl_ProjectionMatrix` dará acesso a matriz de projeção e a variável `gl_ModelViewMatrix` permitirá acessar a matriz de transformação para coordenadas de câmera. Algumas variáveis serão definidas especificamente para o tipo de efeito que se deseja programar. Estas variáveis poderão herdar valores provindos do programa principal de OpenGL.

A seguir, é mostrado um *vertex shader* padrão, comentando-se detalhadamente sua estrutura. Apesar do programa mostrado fazer exatamente o que um pipeline não programável faz, perceba-se que o cálculo é totalmente executado de forma customizada. Este código serve de molde para que programas mais sofisticados possam ser efetuados.

```
// Definição de variáveis a serem usadas pelo vertex shader.
// As GPUs possuem registradores internos para vetores, além dos registradores de
// ponto flutuante. Assim, ao declarar variáveis do tipo vec4, vec3 e vec2 são
// alocados registradores específicos deste tipo. Uma variável uniform é provinda
// do programa principal OpenGL.
uniform vec3 LightPosition;
// Para o cálculo de iluminação per-vertex que será feito neste programa deverão ser
// ajustados algumas constantes com os devidos parâmetros.
const float CEspecular = 0.25;
const float CDifusa = 1.0 - CEspecular;
// Como saída deste shader, serão gerados as coordenadas projetadas do vértice e
// sua cor, os quais serão passados para o fragment shader. As variáveis seguintes
// armazenarão este resultado. O termo varying indica as variáveis repassadas
// para o fragment shader.
varying float IntensidadeLuz;
varying vec2 PosicaoMC;
// Início do vertex shader.
void main(void)
{
  // A variável PosicaoCC refere-se à posicão do vértice nas coordenadas da
  // camêra. Este valor é obtido multiplicando a matriz de transformação a seguir com
  // as coordenadas do vértice sendo tratado.
  vec3 PosicaoCC = vec3(gl_ModelViewMatrix * gl_Vertex);
  // Assim como o vértice foi convertido de sistema de coordenadas, também deve-se
  // aplicar esta conversão à sua normal. Ao mesmo tempo que isto é feito, aplica-se
  // uma função para normalizar o vetor normal resultante.
  vec3 normT = normalize(gl_NormalMatrix * gl_Normal);
  // Neste exemplo será aplicado um modelo de iluminação padrão ao vértice. Para
  // este cálculo é necessário obter um vetor de iluminação, que tem a direção dada
  // pelo vértice e a posição da luz. Aqui este vetor será calculado já no sistema de
  // coordenadas da câmera.
  vec3 vetorLuz = normalize(LightPosition - PosicaoCC);
  // Para o cálculo de iluminação a ser efetuado será necessário o vetor de reflexo
  // da luz. A função reflect() da OpenGL Shader Language fará esta operação.
  vec3 VetorReflexo = reflect(-VetorLuz, normT);
  // Para o cálculo da especular é necessário obter o vetor que vai da câmera ao
  // vértice. Como o vértice já foi transformado para o sistema de coordenadas da
```

```
  // camêra e a origem da câmera é (0,0,0), este vetor é obtido através da operação
  // (0,0,0) - PosicaoCC.
  vec3 vetorCamera = normalize(-PosicaoCC);
  // A contribuição difusa corresponde ao cosseno do ângulo formado entre o vetor
  // da luz e a normal. Se os dois vetores estão normalizados, este cosseno é dado
  // pelo produto escalar de ambos. Caso este valor seja menor que zero, a luz está
  // atrás do objeto e portanto não deve haver iluminação. A função max trunca o
  // valor para zero neste caso.
  float difuso = max(dot(VetorLuz, normT), 0.0);
  // O componente especular da iluminação é calculada da seguinte forma:
  // (VetorCamera.VetorReflexo)^coef. O fator coef define a "concentração"
  // da área especular. Quanto maior for este valor, mais pontual esta será.
  float coef = 20;
  float especular = max(dot(VetorReflexo, VetorCamera), 0.0);
  especular = pow(especular, coef);
  // Finalmente, o resultado da iluminação será a mescla dos valores difuso e
  // especular, usando os coeficientes de contribuição previamente definidos.
  // Este resultado será passado para o fragment shader através da variável
  // IntensidadeLuz.
  IntensidadeLuz = CDifusa * difuso + CEspecular * especular;
  // Além de passar para o fragment shader a cor resultante, pode-se passar outros
  // atributos. Neste exemplo se passam também as coordenadas do vértice.
  PosicaoMC = gl_Vertex;
  // Finalmente, quase sempre será necessário nos vertex shaders projetar as
  // coordenadas do vértice para o plano de projeção. Isto é feito pela
  // função pré-definida ftransform().
  gl_Position = ftransform();
}
```

3.2.4.2. Pixel Shaders

A principal função dos *pixel shaders* consiste em computar a cor de um fragmento. Como já foi discutido, um fragmento é um candidato a pixel: a GPU irá pintar um fragmento na tela, mas este ainda pode ser sobre-escrito por algum outro fragmento tratado posteriormente e que pertença a um polígono que está na frente daquele que deu origem ao primeiro. Um *pixel shader* não pode alterar as coordenadas na tela do fragmento sendo tratado.

Foi visto que no *vertex shader* era devolvido o valor `PosicaoMC`, que corresponde ao vértice devidamente projetado. De igual forma, o valor da variável `IntensidadeLuz` corresponde à iluminação calculada para este vértice. A entrada de um *pixel shader*, por outro lado, não é um vértice, mas sim um pixel. Assim, quando um *pixel shader* recebe a *PosicaoMC* não significa que esteja recebendo as coordenadas de um vértice, mas sim o valor interpolado correspondente à posição geométrica da parte do polígono ao qual o pixel pertence. Isto ocorre também com as demais variáveis provindas do *vertex shader*. As coordenadas do fragmento na tela pode ser acessada através da variável pré-definida `gl_FragCoord`.

O programa que se segue é bastante elementar e apenas aplica uma cor azul mesclada com a iluminação provinda do *vertex shader* anteriormente apre-

sentado. Apesar da simplicidade, este programa corresponde ao modelo de um *pixel shader* qualquer.

```
// Apenas uma variável será definida, contendo a cor do material a ser aplicado.
uniform vec3 material = (0.0, 0.0, 1.0);
// As duas variáveis que provêm do vertex shader são a posição e a cor resultante da
// iluminação, ambas interpoladas.
varying vec2 PosicaoMC;
varying float IntensidadeLuz;
// Início do pixel shader.
void main(void)
{
  // A variável cor acumulará o resultado da iluminação com o material.
  vec3 cor = material * IntensidadeLuz;
  // A variável pré-definida gl_FragColor deve receber no final a cor com que o
  // fragmento será pintado.
  gl_FragColor = vec4(cor, 1.0);
}
```

3.3. Simulação Dinâmica de Corpos Rígidos

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 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. Em games, ao contrário, a simulação é interativa e em tempo real. Por isso, mesmo com o advento das PPUs (unidades de processamento de física) e a possibilidade de utilização de GPUs no desenvolvimento de aplicações de simulação, apenas modelos mais “simplificados” têm sido implementados em games. Destes, os mais comuns são os modelos dinâmicos de corpos rígidos.

Corpos rígidos pode ser classificados de várias maneiras. 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. Um corpo rígido *continuo* 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. Esta seção faz uma introdução à mecânica de corpos rígidos contínuos necessária à compreensão do funcionamento do componente de um motor de jogo digital responsável pela simulação física (de corpos rígidos), chamado *motor de física*. O objetivo é apresentar os principais conceitos e dificuldades envolvidos na implementação de um motor de física (apesar da "simplicidade" do modelo), bem como sua arquitetura e funcionalidades, além de introduzir o PhysX SDK, um framework para simulação dinâmica de corpos rígidos desenvolvido pela AGEIA, a fabricante da primeira PPU do mercado.

3.3.1. Conceitos Básicos da Mecânica Newtoniana

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

$$\mathbf{v}(t) = \dot{\mathbf{r}} = \frac{d\mathbf{r}}{dt} \tag{3.1}$$

e sua aceleração

$$\mathbf{a}(t) = \dot{\mathbf{v}} = \frac{d\mathbf{v}}{dt} = \ddot{\mathbf{r}} = \frac{d^2\mathbf{r}}{dt^2}. \tag{3.2}$$

O *momento linear* da partícula é definido como

$$\mathbf{p}(t) = m\mathbf{v}. \tag{3.3}$$

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

$$\mathbf{F}(t) = \dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = \frac{d}{dt}(m\mathbf{v}). \tag{3.4}$$

Se a massa da partícula é constante:

$$\mathbf{F} = m\frac{d\mathbf{v}}{dt} = m\mathbf{a}. \tag{3.5}$$

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

$$\mathbf{L}(t) = \mathbf{r} \times \mathbf{p} = \mathbf{r} \times m\mathbf{v}. \tag{3.6}$$

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

$$\tau(t) = \mathbf{r} \times \mathbf{F}. \tag{3.7}$$

Da mesma forma que, de acordo com a Equação (3.4), 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:

$$\dot{\mathbf{L}} = \frac{d\mathbf{L}}{dt} = \frac{d}{dt}(\mathbf{r} \times \mathbf{p}) = \mathbf{r} \times \frac{d\mathbf{p}}{dt} + \frac{d\mathbf{v}}{dt} \times \mathbf{p} = \mathbf{r} \times \mathbf{F} = \tau. \tag{3.8}$$

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

$$W_{12} = \int_{\mathbf{r}_1}^{\mathbf{r}_2} \mathbf{F} \cdot d\mathbf{r}, \tag{3.9}$$

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

$$W_{12} = m\int_{t_1}^{t_2} \mathbf{F} \cdot \mathbf{v}dt = m\int_{t_1}^{t_2} \frac{d\mathbf{v}}{dt} \cdot \mathbf{v}dt = \frac{m}{2}\int_{t_1}^{t_2} \frac{d}{dt}(v^2)dt = \frac{m}{2}(v_2^2 - v_1^2). \tag{3.10}$$

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

$$W_{12} = K_2 - K_1. \tag{3.11}$$

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

$$\mathbf{F} = -\nabla P(\mathbf{r}(t)) = -\left(\frac{\partial P}{\partial x}, \frac{\partial P}{\partial y}, \frac{\partial P}{\partial z}\right), \tag{3.12}$$

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

$$W_{12} = P_1 - P_2. \tag{3.13}$$

Combinando-se a equação acima com a Equação (3.11), obtém-se

$$K_1 + P_1 = K_2 + P_2, \tag{3.14}$$

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).

3.3.1.1. 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

$$\frac{d\mathbf{p}_i}{dt} = m_i\mathbf{v}_i = \mathbf{F}_i^e + \sum_j \mathbf{F}_{ji}, \tag{3.15}$$

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

$$\frac{d^2}{dt^2}\sum_i m_i\mathbf{r}_i = \sum_i \mathbf{F}_i^e + \sum_{i,j}\mathbf{F}_{ji}. \tag{3.16}$$

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

$$\bar{\mathbf{r}}(t) = \frac{\sum m_i\mathbf{r}_i}{\sum m_i} = \frac{\sum m_i\mathbf{r}_i}{M}, \tag{3.17}$$

onde M é a massa total. O vetor $\bar{\mathbf{r}}$ define um ponto $\mathcal{C}$ chamado *centro de massa* do sistema. Com esta definição, a Equação (3.15) reduz-se a

$$M\frac{d^2\bar{\mathbf{r}}}{dt^2} = \sum_i \mathbf{F}_i^e = \mathbf{F}, \tag{3.18}$$

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,

$$\mathbf{P}(t) = \sum_i m_i\frac{d\mathbf{r}_i}{dt} = M\frac{d\bar{\mathbf{r}}}{dt} = M\bar{\mathbf{v}}, \tag{3.19}$$

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

$$\mathbf{L}(t) = \sum_i \mathbf{r}_i\times\mathbf{p}_i = \bar{\mathbf{r}}\times M\bar{\mathbf{v}} + \sum_i \mathbf{r}_i'\times\mathbf{p}_i', \tag{3.20}$$

onde $\mathbf{r}_i' = \mathbf{r}_i - \bar{\mathbf{r}}$ 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. Ou seja, o momento angular total é o momento angular do sistema concentrado no centro de massa mais o momento angular do movimento em torno do centro de massa. A taxa de variação do momento angular total,

$$\dot{\mathbf{L}} = \tau = \sum_i \mathbf{r}_i \times \mathbf{F}_i^e, \tag{3.21}$$

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

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 [Goldstein 1980].

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

$$\bar{\mathbf{r}}(t) = \frac{\int_V \mathbf{r}\, dm}{\int_V dm} = \frac{\int_V \rho \mathbf{r}\, dV}{M}, \tag{3.22}$$

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

3.3.1.2. 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

$$h(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_n, t) = 0 \tag{3.23}$$

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. (Posteriormente, nesta seção, são vistas condições expressas por *inequações* envolvendo posições e o tempo, não holonômicas portanto.)

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

$$r_{ij} - c_{ij} = 0, \quad 1 \leq i, j \leq n, \tag{3.24}$$

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.

3.3.2. Dinâmica de Corpos Rígidos

Os vínculos da Equação (3.24) 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

$$r_{12} = c_{12}, \quad r_{23} = c_{23} \quad \text{e} \quad r_{13} = c_{13}, \tag{3.25}$$

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. Três das coordenadas do corpo rígido em t serão as coordenadas globais da posição do centro de massa $\bar{\mathbf{r}}(t)$, Equação (3.22); as três restantes serão a orientação do sistema local em relação ao sistema global.

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:

$$\mathbf{R}(t) = \begin{bmatrix} \mathbf{u}(t) & \mathbf{v}(t) & \mathbf{n}(t) \end{bmatrix} = \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix}, \tag{3.26}$$

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

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

$$\mathbf{r}(t) = \bar{\mathbf{r}}(t) + \mathbf{R}(t)\mathbf{r}_0, \tag{3.27}$$

onde $\mathbf{r}_0$ é a posição de $\mathcal{P}$ em relação ao sistema local. A posição $\bar{\mathbf{r}}$ 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 $\bar{\mathbf{v}}(t)$ 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 $\omega(t)$. A direção de $\omega(t)$ define a direção do eixo de rotação e $\|\omega(t)\|$ o ângulo percorrido por um ponto em torno deste eixo no instante t.

Pode-se estabelecer uma relação entre $\dot{\mathbf{R}}$ e a velocidade angular ω, do mesmo modo que há relação uma entre $\dot{\bar{\mathbf{r}}}$ e a velocidade linear $\bar{\mathbf{v}}$. 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 [Baraff 2001]

$$\dot{\mathbf{r}} = \omega \times \mathbf{r}. \tag{3.28}$$

Agora, aplica-se a equação acima a cada uma das colunas de $\mathbf{R}$ na Equação (3.26), nominalmente os versores $\mathbf{u}$, $\mathbf{v}$ e $\mathbf{n}$, obtendo-se

$$\dot{\mathbf{R}} = \begin{bmatrix} \omega \times \mathbf{u} & \omega \times \mathbf{v} & \omega \times \mathbf{n} \end{bmatrix}. \tag{3.29}$$

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

$$\mathbf{a} \times \mathbf{b} = \begin{bmatrix} a_y b_z - b_y c_z \\ a_z b_x - a_x b_y \\ a_x b_y - b_x a_y \end{bmatrix} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \widehat{\mathbf{a}}\mathbf{b}, \tag{3.30}$$

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

$$\widehat{\mathbf{a}} = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}. \tag{3.31}$$

A relação procurada entre $\dot{\mathbf{R}}$ e ω é obtida escrevendo-se os produtos vetoriais da Equação (3.29) como a multiplicação da matriz $\widehat{\omega}$ pelos versores $\mathbf{u}$, $\mathbf{v}$ e $\mathbf{n}$, resultando

$$\dot{\mathbf{R}}(t) = \widehat{\omega}(t)\mathbf{R}(t). \tag{3.32}$$

A partir desta relação, pode-se derivar a Equação (3.27) e escrever a velocidade em coordenadas globais de um ponto $\mathcal{P}$ de um corpo rígido em um instante t como sendo

$$\dot{\mathbf{r}}(t) = \bar{\mathbf{v}}(t) + \widehat{\omega}(t)\mathbf{R}(t)\mathbf{r}_0 + = \bar{\mathbf{v}}(t) + \omega(t) \times (\mathbf{r}(t) - \bar{\mathbf{r}}(t)). \tag{3.33}$$

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 (3.19) estabelece que

$$\bar{\mathbf{v}}(t) = \frac{\mathbf{P}(t)}{M}. \tag{3.34}$$

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:

$$\mathbf{L}(t) = \mathbf{I}(t)\,\omega(t), \tag{3.35}$$

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

$$I_{ij}(t) = \int_V \rho(\mathbf{r}')(r'^2\delta_{ij} - x'_i x'_j)\,dV, \quad i,j = 1,2,3, \tag{3.36}$$

onde $\mathbf{r}' = \mathbf{r}(t) - \bar{\mathbf{r}}(t) = (x'_1, x'_2, x'_3)$ 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

$$\delta_{ij} = \begin{cases} 0, \text{se } i \neq j, \\ 1, \text{se } i = j. \end{cases} \tag{3.37}$$

Se o tensor de inércia de um corpo rígido tivesse que ser calculado através da Equação (3.36) em cada instante t em que $\mathbf{R}(t)$ variasse, o tempo de processamento para fazê-lo, durante a simulação, poderia ser proibitivamente 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 [Baraff 2001]

$$\mathbf{I}(t) = \mathbf{R}(t)\,\mathbf{I}_0\,\mathbf{R}(t)^T. \tag{3.38}$$

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

$$\mathbf{X}(t) = \begin{bmatrix} \bar{\mathbf{r}}(t) \\ \mathbf{R}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{bmatrix}. \tag{3.39}$$

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 $\bar{\mathbf{v}}(t)$, $\mathbf{I}(t)$ e $\omega(t)$ podem ser calculadas por

$$\bar{\mathbf{v}}(t) = \frac{\mathbf{P}(t)}{M}, \quad \mathbf{I}(t) = \mathbf{R}(t)\mathbf{I}_0\mathbf{R}(t)^T \quad \text{e} \quad \omega(t) = \mathbf{I}(t)^{-1}\mathbf{L}(t). \tag{3.40}$$

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

$$\frac{d}{dt}\mathbf{X}(t) = \begin{bmatrix} \bar{\mathbf{v}}(t) \\ \widehat{\omega}(t)\mathbf{R}(t) \\ \mathbf{F}(t) \\ \tau(t) \end{bmatrix} \tag{3.41}$$

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 (3.41) são conhecidas no tempo t, sendo a força $\mathbf{F}$ e o torque τ em relação ao centro de massa $\mathcal{C}$ de cada corpo rígido determinados pela aplicação.

3.3.2.1. Quaternions

Em simulação dinâmica é preferível usar *quaternions* unitários a matrizes de rotação para representar a orientação de corpos rígidos. O principal motivo é que os erros numéricos acumulados nos nove coeficientes de $\mathbf{R}(t)$, à medida que a matriz é atualizada ao longo do tempo da simulação, faz com que esta não seja precisamente uma matriz de rotação.

Um quaternion $\mathbf{q}$ é uma estrutura algébrica constituída de duas partes: um escalar s e um vetor $\mathbf{v} = (v_x, v_y, v_z)$, ou $\mathbf{q} = [s, \mathbf{v}]$. A multiplicação de dois quaternions $\mathbf{q}_1$ e $\mathbf{q}_2$ é definida como

$$\mathbf{q}_1\mathbf{q}_2 = [s_1, \mathbf{v}_1][s_2, \mathbf{v}_2] = [s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2]. \tag{3.42}$$

Um quaternion unitário é um quaternion onde $s^2 + v_x^2 + v_y^2 + v_z^2 = 1$. Uma rotação de um ângulo θ em torno do versor $\mathbf{u}$ é representada pelo quaternion unitário

$$\mathbf{q} = [s, \mathbf{v}] = [\cos(\theta/2), \operatorname{sen}(\theta/2)\,\mathbf{u}]. \tag{3.43}$$

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

produto

$$\mathbf{p}' = [0,(x',y',z')] = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}, \tag{3.44}$$

onde $\mathcal{P}'(x',y',z')$ é o ponto $\mathcal{P}$ rotacionado. A matriz de rotação 3×3 correspondente, necessária para o cálculo do tensor de inércia, é

$$\mathbf{R} = \begin{bmatrix} 1-2(v_y^2+v_z^2) & 2(v_xv_y-sv_z) & 2(v_xv_z+sv_y) \\ 2(v_xv_y+sv_z) & 1-2(v_x^2+v_z^2) & 2(v_yv_z-sv_x) \\ 2(v_xv_z-sv_y) & 2(v_yv_z-sv_x) & 1-2(v_x^2+v_y^2) \end{bmatrix}. \tag{3.45}$$

Se $\mathbf{q}(t)$ é um quaternion unitário que representa a orientação de um corpo rígido no instante t, então pode-se deduzir [Eberly 2004]

$$\dot{\mathbf{q}}(t) = \frac{1}{2}\mathbf{w}(t)\mathbf{q}(t), \tag{3.46}$$

onde $\mathbf{w}(t) = [0,\omega(t)]$. O estado de um corpo rígido e sua derivada podem então ser escritos respectivamente como

$$\mathbf{X}(t) = \begin{bmatrix} \overline{\mathbf{r}}(t) \\ \mathbf{q}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{bmatrix} \quad \text{e} \quad \frac{d}{dt}\mathbf{X}(t) = \begin{bmatrix} \overline{\mathbf{v}}(t) \\ \frac{1}{2}\mathbf{w}(t)\mathbf{q}(t) \\ \mathbf{F}(t) \\ \tau(t) \end{bmatrix}. \tag{3.47}$$

3.3.2.2. Dinâmica de Corpos Rígidos com Restriçõs 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) resultantes 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 de junção. Alguns exemplos são ilustrados na Figura 3.8.

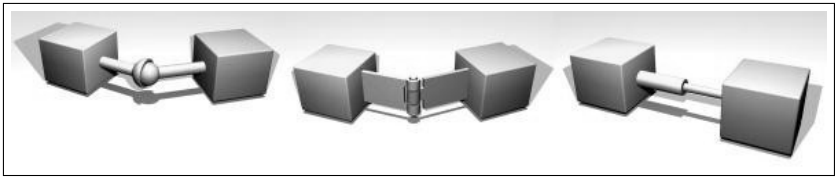


Figura 3.8. Exemplos de junções: esférica, de revolução e cilíndrica

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, neste texto são abordadas 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 *vértice/face* ocorre quando, em um instante t_c, um vértice v_A de um poliedro A está em contato com uma face f_B de um poliedro B em um ponto $\mathcal{P}_c = v_A$. 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 $\mathcal{P}_c$. Neste caso, a direção normal ao ponto de contato é definida como

$$\mathbf{N}(t_c) = \mathbf{e}_A(t_c) \times \mathbf{e}_B(t_c), \tag{3.48}$$

onde $\mathbf{e}_A$ e $\mathbf{e}_B$ são versores nas direções de e_A e e_B no intante 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 os n_c pontos de contato entre os corpos da simulação.

As restrições de contato podem ser caracterizadas como *de colisão* e *de repouso*, dependendo da velocidade relativa entre os corpos no momento do contato. Para o i-ésimo contato, $1 \leq i \leq n_c$, sejam $\mathcal{P}_{A_i}$ e $\mathcal{P}_{B_i}$ os pontos sobre os corpos A_i e B_i em que estes se tocam no instante t_c, isto é, $\mathcal{P}_{A_i} = \mathcal{P}_{B_i} = \mathcal{P}_{c_i}$. A distância relativa entre $\mathcal{P}_{A_i}$ e $\mathcal{P}_{B_i}$, medida ao longo da normal ao ponto de contato em um instante $t < t_c$ é

$$d_i(t) = \mathbf{N}_i(t) \cdot (\mathbf{r}_{A_i}(t) - \mathbf{r}_{B_i}(t)). \tag{3.49}$$

Naturalmente, quando $t = t_c$, $d_i(t_c) = 0$ e $\mathbf{r}_{A_i} = \mathbf{r}_{B_i}$, mas as velocidades dos pontos $\mathcal{P}_{A_i}$ e $\mathcal{P}_{B_i}$ podem ser diferentes. Estas velocidades são, de acordo com a Equação (3.33),

$$\dot{\mathbf{r}}_{A_i}(t_c) = \overline{\mathbf{v}}_{A_i}(t_c) + \omega_{A_i}(t_c) \times \mathbf{r}'_{A_i}(t_c), \tag{3.50}$$

$$\dot{\mathbf{r}}_{B_i}(t_c) = \overline{\mathbf{v}}_{B_i}(t_c) + \omega_{B_i}(t_c) \times \mathbf{r}'_{B_i}(t_c). \tag{3.51}$$

Nas equações acima, $\mathbf{r}'_{A_i} = \mathbf{r}_{A_i} - \overline{\mathbf{r}}_{A_i}$, $\mathbf{r}'_{B_i} = \mathbf{r}_{B_i} - \overline{\mathbf{r}}_{B_i}$ e $\overline{\mathbf{r}}_\chi$, $\overline{\mathbf{v}}_\chi$ e ω_χ são, respectivamente, a posição, velocidade linear e velocidade angular do corpo χ, onde $\chi = A_i, B_i$.

Derivando-se a Equação (3.49) em t, obtém-se a velocidade relativa entre $\mathcal{P}_{A_i}$ e $\mathcal{P}_{B_i}$ no ponto de contato:

$$\dot{d}_i(t_c) = \mathbf{N}_i(t_c) \cdot (\dot{\mathbf{r}}_{A_i}(t_c) - \dot{\mathbf{r}}_{B_i}(t_c)). \tag{3.52}$$

Se $\dot{d}_i(t_c) > 0$, os corpos estão se afastando e não haverá o contato em $\mathcal{P}_{c_i}$ em $t > t_c$. Se $\dot{d}_i(t_c) = 0$, então os corpos permanecerão em contato de repouso em $t > t_c$. Este tipo de contato é tratado mais adiante. A condição $\dot{d}_i(t_c) < 0$ caracteriza uma colisão entre A_i e B_i no instante t_c. Nesta situação, haverá interpenetração dos corpos se as velocidades $\dot{\mathbf{r}}_{A_i}$ e $\dot{\mathbf{r}}_{B_i}$ não forem imediatamente modificadas. A Figura 3.9 ilustra.

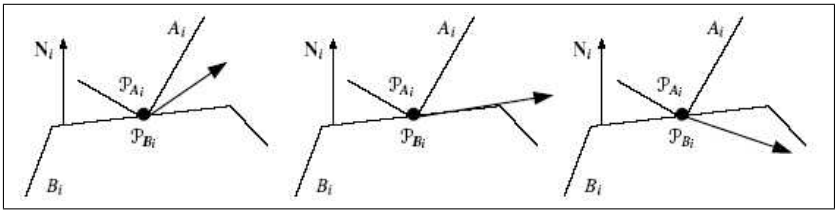


Figura 3.9. Velocidade $\dot{\mathbf{r}}_{A_i}(t_c) - \dot{\mathbf{r}}_{B_i}(t_c)$ no ponto de contato i

Para modificar instantaneamente a velocidade, deve-se aplicar um *impulso* aos corpos A_i e B_i no ponto de contato de colisão. O impulso de uma força $\mathbf{F}$ atuando em tempo infinitesimal Δt é definido como

$$\mathbf{F}_\mathrm{I} = \int_{\Delta t} \mathbf{F}\, dt. \tag{3.53}$$

Aplicado a um corpo rígido, o impulso causa uma variação instantânea do momento linear $\Delta\mathbf{P} = \mathbf{F}_\mathrm{I}$ e, em conseqüência, da velocidade $\Delta\bar{\mathbf{v}} = \mathbf{F}_\mathrm{I}/M$. O torque devido ao impulso, $\tau_\mathrm{I} = (\mathbf{r}_c - \bar{\mathbf{r}}) \times \mathbf{F}_\mathrm{I}$, onde $\mathbf{r}_c$ é a posição do ponto $\mathcal{P}_c$ de aplicação do impulso, causa uma variação do momento angular $\Delta\mathbf{L} = \tau_\mathrm{I}$.

Quando não há atrito entre os corpos A_i e B_i no ponto de contato $\mathcal{P}_{c_i}$, a direção da força de contato para evitar a interpenetração será a mesma da normal $\mathbf{N}_i$. Assim,

$$\mathbf{F}_{\mathrm{I}i}(t_c) = \lambda_{c_i} \mathbf{N}_i(t_c), \tag{3.54}$$

onde o escalar λ_{c_i}, o qual define a magnitude do impulso, é determinado através da seguinte lei empírica para colisões:

$$\dot{d}_i^+(t_c) = -\varepsilon_i \dot{d}_i^-(t_c). \tag{3.55}$$

Na equação acima, os índices $-$ e $+$ denotam imediatamente antes e imediatamente após à aplicação do impulso, respectivamente, e $0 \leq \varepsilon_i \leq 1$ é o *coeficiente de restituição* da colisão. Se $\varepsilon_i = 1$, então nenhuma energia cinética é perdida na colisão e o choque é perfeitamente elástico; se $\varepsilon_i = 0$, toda a energia cinética é perdida e os corpos permanecerão em contato de repouso após

a colisão. O coeficiente de restituição é uma propriedade usualmente associada ao material de um corpo rígido. Neste caso, o valor de ε_i pode ser tomado como uma combinação (valor máximo, mínimo, média, etc.) dos coeficientes de restituição dos corpos em contato.

A partir das equações (3.52) e (3.55), pode-se deduzir [Baraff 2001]:

$$\lambda_{c_i} = \frac{-(1+\varepsilon_i)\,\dot{d}_i^-}{M_{A_i}^{-1} + M_{B_i}^{-1} + \mathbf{N}_i \cdot (\mathbf{I}_{A_i}^{-1}(\mathbf{r}'_{A_i} \times \mathbf{N}_i)) \times \mathbf{r}'_{A_i} + \mathbf{N}_i \cdot (\mathbf{I}_{B_i}^{-1}(\mathbf{r}'_{B_i} \times \mathbf{N}_i)) \times \mathbf{r}'_{B_i}}, \quad (3.56)$$

onde $\mathbf{I}_\chi$ é o momento de inércia do corpo $\chi = A_i, B_i$. Conhecido o valor λ_{c_i}, aplicam-se forças impulsivas $+\lambda_{c_i}\mathbf{N}_i$ no ponto $\mathcal{P}_{A_i}$ e $-\lambda_{c_i}\mathbf{N}_i$ no ponto $\mathcal{P}_{B_i}$ a fim de se evitar a interpenetração de A_i e B_i.

Um pseudocódigo C++ ilustrativo para tratamento de colisões é apresentado a seguir. Inicialmente, seja a definição dos tipos `RigidBody` (atributos de um corpo rígido) e `Contact` (informações de contato). Os tipos `Vector3D` e `Matrix3D` representam vetores 3D e matrizes 3×3, respectivamente.

```
struct RigidBody
{
  double epsilon; // coeficiente de restituição ε
  double massInv; // inversa da massa 1/M
  Matrix3D IBodyInv; // inversa do tensor de inércia local I_0^-1
  Vector3D x; // posição do centro de massa x̄
  Matrix3D R; // orientação do sistema local R
  Vector3D P; // momento linear P
  Vector3D L; // momento angular L
  Vector3D v; // velocidade linear v̄
  Vector3D w; // velocidade angular ω
  Matrix3D IInv; // inversa do tensor de inércia global RI_0^-1 R^T
  Vector3D force; // força F
  Vector3D torque; // torque τ
}; // RigidBody

struct Contact
{
  RigidBody* A; // corpo rígido A
  RigidBody* B; // corpo rígido B
  Vector3D P; // ponto de contato P_c
  Vector3D N; // normal N
}; // Contact
```

A função `handleCollision()` a seguir calcula e aplica uma força impulsiva nos corpos em contato de colisão. A função toma como argumento uma referência para o `contact` a ser tratado. O valor de λ_{c_i}, dado pela Equação (3.56), é computado pela função `computeLambda()`. A função `cross()` calcula o produto vetorial de dois `Vector3D`s. Ambas não são listadas aqui.

```
void handleCollision(Contact& contact)
{
```

```
  // Calcula λ_ci e a força de impulso
  Vector3D force = computeLambda(contact) * contact.N;
  // Aplica a força e o torque de impulso nos corpos
  contact.A->P += force;
  contact.B->P -= force;
  contact.A->P += cross(contact.P - contact.A->x, force);
  contact.B->P -= cross(contact.P - contact.B->x, force);
  // Altera as velocidades dos corpos
  contact.A->v = contact.A->massInv * contact.A->P;
  contact.B->v = contact.B->massInv * contact.B->P;
  contact.A->w = contact.A->IInv * contact.A->L;
  contact.B->w = contact.B->IInv * contact.B->L;
}
```

A função C++ `HandleAllCollisions()` abaixo resolve colisões em múltiplos pontos de contato. Os argumentos da função são o número `nc` e um vetor `contacts` com `nc` contatos. A função `isColliding()`, não apresentada, toma como argumento uma referência a um contato e verifica se a velocidade relativa no ponto de contato, dada pela Equação (3.52), é negativa, ou seja, se o contato é de colisão. Em caso afirmativo, a função `handleCollision()` é chamada.

```
void handleAllCollisions(Contact contacts[], int nc)
{
  bool hadCollision;

  do
  {
    hadCollision = false;
    for (int i = 0; i < nc; i++)
      if (isColliding(contacts[i]))
      {
        handleCollision(contacts[i]);
        hadCollision = true;
      }
  }
  while (hadCollision == true);
}
```

Note, em `handleAllCollisions()`, que toda vez que um contato de colisão é identificado e tratado por `handleCollision()`, o vetor de contatos é percorrido novamente. Isto é necessário porque a aplicação de um força de impulso pode fazer com que corpos que estavam em repouso não o estejam mais, ou que novos contatos de colisão venham a ocorrer. Esta técnica de tratamento de colisões, um ponto de contato por vez, é conhecida como *propagação*. Há situações em que o resultado da simulação pode variar, dependendo da ordem em que as colisões são processadas. A computação *simultânea* dos impulsos em múltiplos pontos de contato, portanto, é desejável, embora mais complicada de implementar, conforme visto a seguir para o caso de contato de repouso.

Seja uma configuração com n_c pontos de contato, supostos em repouso, isto é, $\dot{d}_i(t_c) = 0,\ 1 \leq i \leq n_c$. Em cada ponto de atua uma força de contato $\mathbf{F}_{c_i}(t_c) = \lambda_{r_i}\mathbf{N}_i(t_c)$, onde λ_{r_i} é um escalar a ser determinado tal que

1. $\mathbf{F}_{c_i}$ deve prevenir a interpenetração dos corpos;
2. $\mathbf{F}_{c_i}$ deve ser uma força repulsiva; e
3. $\mathbf{F}_{c_i}$ anula-se quando os corpos se separam.

Para satisfazer a primeira condição, deve-se analisar a aceleração $\ddot{d}_i(t_c)$, a qual mede como os dois corpos estão acelerando um em relação ao outro no ponto de contato $\mathcal{P}_{c_i}$ (uma vez que $d_i(t_c) = \dot{d}_i(t_c) = 0$). Se $\ddot{d}_i(t_c) > 0$, o contato entre os corpos será quebrado em $t > t_c$; se $\ddot{d}_i(t_c) = 0$, estes permanecerão em contato de repouso; se $\ddot{d}_i(t_c) < 0$, os corpos estão acelerando um na direção do outro e haverá interpenetração em $t > t_c$, o que deve ser evitado. Portanto, a primeira condição impõe a restrição

$$\ddot{d}_i(t_c) \geq 0 \tag{3.57}$$

para cada ponto de contato. A expressão de $\ddot{d}_i(t_c)$ é obtida derivando-se a Equação (3.52):

$$\ddot{d}_i(t_c) = \mathbf{N}_i(t_c)\cdot(\ddot{\mathbf{r}}_{A_i}(t_c) - \ddot{\mathbf{r}}_{B_i}(t_c)) + 2\dot{\mathbf{N}}_i(t_c)\cdot(\dot{\mathbf{r}}_{A_i}(t_c) - \dot{\mathbf{r}}_{B_i}(t_c)), \tag{3.58}$$

onde a derivada da normal em $\mathcal{P}_{c_i}$, para contato vértice/face, é

$$\dot{\mathbf{N}}_i(t) = \omega_{B_i}(t) \times \mathbf{N}_i(t_c) \tag{3.59}$$

e para contato aresta/aresta

$$\dot{\mathbf{N}}_i(t) = \frac{\mathbf{U}(t) - (\mathbf{U}(t)\cdot\mathbf{N}_i(t_c))\mathbf{N}_i(t)}{\|\mathbf{e}_{A_i} \times \mathbf{e}_{B_i}\|}, \tag{3.60}$$

$$\mathbf{U}(t) = \mathbf{e}_{A_i} \times (\omega_{B_i} \times \mathbf{e}_{B_i}) + (\omega_{A_i} \times \mathbf{e}_{A_i}) \times \mathbf{e}_{B_i}. \tag{3.61}$$

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

$$\lambda_{r_i} \geq 0 \tag{3.62}$$

(pois $+\lambda_{r_i}\mathbf{N}_i$ atua sobre o corpo A_i) e

$$\lambda_{r_i}\ddot{d}_i(t_c) = 0. \tag{3.63}$$

A fim de se determinar as forças de contato simultaneamente e de tal forma a satisfazer as condições (3.57), (3.62) e (3.63), cada $\ddot{d}_i(t_c)$ é escrito em função das incógnitas λ_{r_i}:

$$\ddot{d}_i(t_c) = \sum_{i=1}^{n_c} a_{ij}\,\lambda_{r_i} + b_i. \tag{3.64}$$

Eberly (2004) deduz as expressões a seguir para a_{ij} e b_i. Para o par de corpos (α, β):

$$\begin{aligned} a_{ij} =& s_\alpha^{ij}(M_\alpha^{-1}\mathbf{N}_i\cdot\mathbf{N}_j + (\mathbf{r'}^i{}_\alpha \times \mathbf{N}_i)^T\mathbf{I}_\alpha^{-1}(\mathbf{r'}^j{}_\alpha \times \mathbf{N}_j)) - \\ & s_\beta^{ij}(M_\beta^{-1}\mathbf{N}_i\cdot\mathbf{N}_j + (\mathbf{r'}^i{}_\beta \times \mathbf{N}_i)^T\mathbf{I}_\beta^{-1}(\mathbf{r'}^j{}_\beta \times \mathbf{N}_j)), \end{aligned} \tag{3.65}$$

onde s_χ^{ij} é $+1$ quando $\mathcal{P}_j$ está no corpo χ e a direção da força é $+\mathbf{N}_j$, -1 quando $\mathcal{P}_j$ está no corpo χ e a direção da força é $-\mathbf{N}_j$, ou 0 quando quando $\mathcal{P}_j$ não está no corpo χ, e

$$b_i = e_i + \mathbf{N}_i \cdot \left[(M_\alpha^{-1}\mathbf{F}_\alpha + (\mathbf{I}_\alpha^{-1}\tau_\alpha) \times \mathbf{r'}^i_\alpha) - (M_\beta^{-1}\mathbf{F}_\beta + (\mathbf{I}_\beta^{-1}\tau_\beta) \times \mathbf{r'}^i_\beta) \right], \quad (3.66)$$

onde

$$\begin{aligned} e_i = & \mathbf{N}_i \cdot \Big[((\mathbf{I}_\alpha^{-1}(\mathbf{L}_\alpha \times \omega_\alpha)) \times \mathbf{r'}_\alpha + \omega_\alpha \times (\omega_\alpha \times \mathbf{r'}^i_\alpha)) - \\ & ((\mathbf{I}_\beta^{-1}(\mathbf{L}_\beta \times \omega_\beta)) \times \mathbf{r'}_\beta + \omega_\beta \times (\omega_\beta \times \mathbf{r'}^i_\beta)) \Big] + \\ & 2\dot{\mathbf{N}}_i \cdot \left[(\mathbf{v}_\alpha + \omega_\alpha \times \mathbf{r'}^i_\alpha) - (\mathbf{v}_\beta + \omega_\beta \times \mathbf{r'}^i_\beta) \right], \end{aligned} \quad (3.67)$$

sendo $\mathbf{F}_\chi$, τ_χ e $\mathbf{L}_\chi = \mathbf{I}_\chi \omega_\chi$ a força externa, torque externo e momento angular do corpo χ, respectivamente.

Em notação matricial, sejam os vetores $n_c \times 1$ $\lambda_r = [\lambda_{r_i}]$, $\ddot{\mathbf{d}} = [\ddot{d}_i(t_c)]$ e $\mathbf{b} = [b_i]$, e a matriz $n_c \times n_c$ $\mathbf{A} = [a_{ij}]$. As condições (3.57), (3.62) e (3.63), para $1 \leq i \leq n_c$, podem ser sucintamente escritas como $\ddot{\mathbf{d}} \geq \mathbf{0}$, $\lambda_r \geq \mathbf{0}$ e $\ddot{\mathbf{d}} \circ \lambda_r = \mathbf{0}$. O problema de contato de repouso consiste, então, na determinação de λ_r (e, portanto, das n_c forças de contato) satisfazendo $\ddot{\mathbf{d}} = \mathbf{A}\lambda_r + \mathbf{b}$, sujeito às restrições $\ddot{\mathbf{d}} \geq \mathbf{0}$ e $\lambda_r \geq \mathbf{0}$ e à condição de complementaridade $\ddot{\mathbf{d}} \circ \lambda_r = \mathbf{0}$. Este é um *problema de complementaridade linear*, ou LCP, cuja solução pode ser obtida pelo algoritmo de Lenke [Eberly 2004]. O componente do motor de física responsável por isto é chamado LCP *solver*.

Uma vez determinadas as forças e torques de contato de repouso, estas são adicionadas às forças e torques atuantes nos respectivos corpos para determinação do estado do sistema no passo de tempo seguinte da simulação. Como ilustração, a função `computeContactForces()` a seguir calcula e aplica forças e torques de contato. Os argumentos são o número `nc` e o vetor de `contacts` (de repouso). A função invoca `computeA()` e `computeB()` para calcular a matriz $\mathbf{A}$ e o vetor $\mathbf{b}$, de acordo com as equações (3.65) e Equação (3.66), respectivamente. Assume-se a existência dos tipos `Matrix` e `Vector` para representar matrizes e vetores de números reais de qualquer dimensão. O vetor λ_r é calculado pelo LCP *solver*, representado pela função `LCPSolver()`. Uma vez obtido, as forças e torque de contato são determinadas e aplicadas aos corpos rígidos.

```
void computeContactForces(Contact contacts[], int nc)
{
  // Calcula matriz A e vetor b
  Matrix A = computeA(contacts, nc);
  Vector b = computeB(contacts, nc);
  // Invoca o LCP solver para calcular λr
  Vector lambda = LCPSolver(a, b);
```

```
  // Calcula e aplica forças e torques de contato
  for (int i = 0; i < nc; i++)
  {
    RigidBody* A = contacts[i].A;
    RigidBody* B = contacts[i].B;
    Vector3D force = lambda[i] * contacts[i];
    // Aplica +λ_{r_i} N_i no corpo A_i
    A->force += force;
    A->torque += cross(contacts[i].P - A->x, force);
    // Aplica −λ_{r_i} N_i no corpo B_i
    B->force -= force;
    B->torque -= cross(contacts[i].P - B->x, force);
  }
}
```

O algoritmo a seguir sumariza o modelo de resposta a contatos de colisões (por propagação) e de repouso visto nesta seção. Para uma simulação com n_a corpos rígidos, a entrada do algoritmo é o estado $\mathbf{X}_j(t)$, $1 \leq j \leq n_a$, de cada corpo rígido no tempo t. Neste momento, as forças (gravidade, vento, etc.) e torques externos já foram aplicadas a cada corpo. A saída da algoritmo é o estado $\mathbf{X}_j(t+\Delta t)$ de cada corpo no tempo $t+\Delta t$. Os passos são:

Passo 1 Invoca-se o detector de colisões para determinação de todos os nc `contacts` entre os corpos da simulação no instante t.

Passo 2 Se nc > 0, invoca-se a função `handleAllCollisions()` com os argumentos `contacts` e nc, responsável pela tratamento de contatos de colisão, se houver. (`contacts[i]` é um contato de colisão se $\dot{d}_i(t) < 0$.) Ao retorno da função, cada contato com $\dot{d}_i(t) > 0$ é removido de `contacts` e nc decrementado. Os nc contatos restantes são de repouso.

Passo 3 Se nc > 0, invoca-se a função `computeContactForces()` com os argumentos `contacts` e nc, responsável pelo tratamento de contatos de repouso.

Passo 4 **I**nvoca-se o ODE *solver* para determinação do estado de cada corpo rígido em $t+\Delta t$.

3.3.3. Arquitetura de um Motor de Física: Introdução ao PhysX

O PhysX SDK é o motor de física da AGEIA (2006). Este 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 e 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 3.10 e comentadas a seguir.

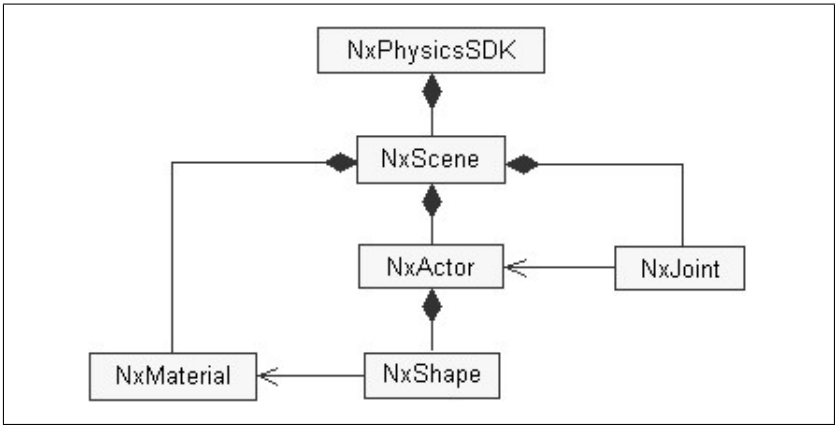


Figura 3.10. Principais classes do Physics SDK

3.3.3.1. 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.

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, podendo ser estendida ao infinito. Uma cena define também propriedades e funcionalidades relacionadas à física, tais como campo gravitacional uniforme atuando sobre seus objetos, detecção de colisões, entre outras. A classe `NxScene` define a interface 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.

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. A função `InitPhysics()` é chamada de dentro da função `InitEngine()`, introduzida na Seção 3.2. O descritor `sceneDesc` define que a cena terá gravidade de $9.8m/s^2$ e detecção de colisões 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 instancia 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);
}
```

3.3.3.2. Criação de Atores

Criada uma cena, podem-se criar materiais, atores e junções. Um material define as propriedades do interior e da superfície de um corpo rígido, respectivamente coeficientes de restituição (visto anteriormente) e coeficientes de atrito estático e dinâmico. Outra propriedade de um material é e a direção de anisotropia, isto é, a direção do movimento de um corpo em que suas propriedades materiais comportam-se de maneira distintas. No PhysX, um material é um objeto de uma classe derivada da classe abstrata `NxMaterial`.

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 linear e angular, e para aplicação de forças e torques em um ator.

A geometria de um ator é definida por uma ou mais *formas*. Uma forma é um objeto de uma classe derivada da classe abstrata `NxShape`. Os principais

tipos de formas disponíveis no SDK são: bloco, esfera, malha de triângulos (para modelos poligonais), cápsula e plano. Formas servem basicamente para dois propósitos:

- Cálculo do centro de massa e inércia de um ator. A toda forma pode-se 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 $\mathbf{I}_0$. (Estes também podem ser explicitamente definidos pela aplicação através de métodos próprios declarados na classe `NxShape`). O centro de massa e inércia de um ator podem ser computados a partir das 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, 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). Como exemplo, a Figura 3.11 mostra, à esquerda, 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 rodas-motrizes). Na figura são visíveis as rodas, representadas por uma esfera 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, uma imagem do modelo gráfico do carrinho.

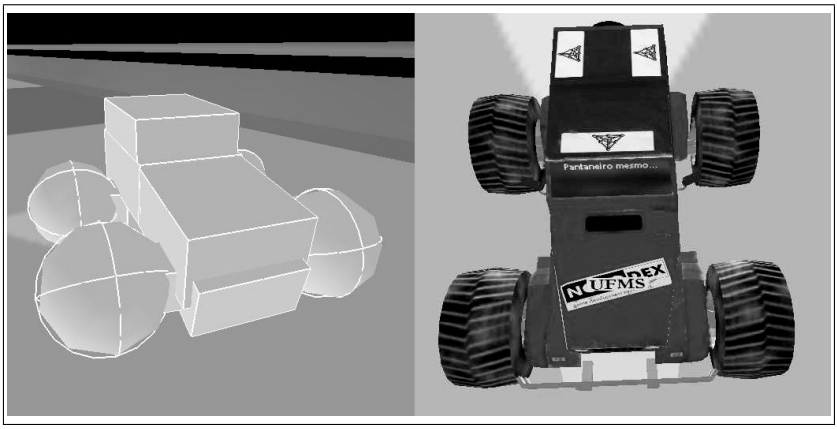


Figura 3.11. Formas geométricas de um ator: colisão e gráfico

A função `CreateCube()` a seguir ilustra a criação de ator cuja posição, orientação e tamanho são passados como argumentos. (No PhysX, posição e orientação são juntos chamados de *pose*). Note o uso de descritores de objetos para criação do ator e de sua forma. Para criar o ator, é necessário passar como argumento para o método `NxScene::createActor()` o descritor de ator `actorDesc`. A este é associado um único descritor de forma `boxDesc`, o qual descreve um cubo com s metros de lado. A posição global do ator a ser criado é definida pelo vetor p, e sua orientação dada pelo quaternion q, o qual é transformado na matriz 3×3 correspondente.

```
NxActor* CreateCube(const NxVec3& p, const NxQuat& q, NxReal s)
{
  NxActorDesc actorDesc;
  NxBodyDesc bodyDesc;
  // A única forma do ator é uma caixa
  NxBoxShapeDesc boxDesc;
  boxDesc.dimensions.set(s/2,s/2,s/2);
  actorDesc.shapes.pushBack(&boxDesc);
  actorDesc.body = &bodyDesc;
  actorDesc.globalPose.t = p;
  actorDesc.globalPose.M = NxMat33(q);
  return gScene->createActor(actorDesc);
}
```

Dois atores podem ser articulados através de junções. Uma junção é um objeto de uma classe derivada da classe abstrata `NxJoint`. Há vários tipos de junções disponíveis no PhysX: esférica, de revolução, cilíndrica (todas ilustradas anteriormente), prismática (similar à junção cilíndrica, mas onde a rotação não é permitida), ponto preso a um plano, plano preso a uma linha, entre outras. Há também um tipo genérico de junção, definida pela classe `NxD6Joint`, que permite ao desenvolvedor restringir individualmente cada um dos graus de liberdade (translações e rotações) dos corpos articulados pela junção. Os atores do modelo da Figura 3.11, por exemplo, são conectados por seis junções de revolução (duas ligando as rodas-motrizes às rodas dianteiras, duas ligando as rodas-motrizes às barras de suspensão dianteiras e duas ligando as rodas traseiras às barras de suspensão traseiras) e quatro junções prismáticas (ligando cada uma das quatro barras de suspensão ao corpo do carrinho), objetos cujas classes derivam das classes abstratas `NxRevoluteJoint` e `NxPrismaticJoint`, respectivamente.

3.3.3.3. 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 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 um passo de cada vez, tipicamente usando-se um in-

tervalo 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 atualizado 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 ser 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()`. (Esta é chamada de dentro do laço principal do motor gráfico, conforme ilustrado pela função `MainLoop()` apresentada na Seção 3.2). A função começa invocando `UpdateTime()` a fim de obter o tempo transcorrido desde a execução do último frame (idealmente deseja-se executar a simulação em passos de tempo constantes, mas pode ser que, entre um e outro, a aplicação tenha gasto mais tempo na renderização ou no processamento da IA, por exemplo). 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 lapso de tempo desde o último frame
  gScene->simulate(deltaTime);
  gScene->flushStream();
  gScene->fetchResults(NX_RIGID_BODY_FINISHED, true);
}
```

3.3.3.4. Renderização da Cena com o Motor Gráfico

Após a simulação, qualquer processamento necessário à renderização da cena pode ser invocado na função `RenderScene()` introduzida na Seção 3.2. Na listagem parcial abaixo, destaca-se a invocação de `RenderActors()`, responsável pela renderização de todos os atores da cena. Esta, por sua vez, invoca a função `DrawActor()` para cada ator da cena, passando como argumento uma referência para o ator a ser renderizado.

```
void RenderScene()
{
```

```
  ...
  RenderActors();
  ...
}
void RenderActors()
{
  int nbActors = gScene->getNbActors();
  NxActor** actors = gScene->getActors();
  // Renderiza todos os atores da cena
  while (nbActors-)
    DrawActor(*actors++);
}
```

A implementação de `DrawActor()` depende da forma geométrica de cada ator, do ponto de vista da renderização. Deve-se lembrar, conforme dito anteriormente, que as formas de um ator definem a geometria do ponto de vista da detecção de colisões, e que estas não correspondem, necessariamente, à geometria para renderização. Quando for este o caso, um ponteiro para o objeto ou estrutura de dados do modelo geométrico para renderização de um ator pode ser a este acoplado através do atributo `NxActor::userData`. Em `DrawActor()`, o modelo pode ser recuperado e manipulado pelo motor gráfico. Seja, então, uma classe `Model` que representa um modelo geométrico para renderização e uma função `DrawModel()` capaz de renderizá-lo. `DrawActor()` poderia então ser escrita como:

```
void DrawActor(NxActor& actor)
{
  // pose é a posição e orientação do ator
  NxMat34 pose = actor->getGlobalPose();
  // Prepara transformação para coordenadas globais
  SetupGLMatrix(pose.t, pose.M);
  // Renderiza geometria
  DrawModel((Model*)actor->userData);
}
```

A função `SetupGLMatrix()` acima combina a matriz de visualização de modelos corrente da OpenGL com a pose (posição e orientação) do ator resultante da simulação dinâmica. Supondo que o modelo geométrico foi definido em termos do sistema local do ator, isto é necessário para que este seja transformado para o sistema global. A implementação de `SetupGLMatrix()` é a seguinte:

```
void SetupGLMatrix(const NxVec3& pos, const NxMat33& orient)
{
  float glmat[16];
  // Converte pose para matriz da OpenGL
  orient.getColumnMajorStride4(&(glmat[0]));
  pos.get(&(glmat[12]));
  glmat[3] = glmat[7] = glmat[11] = 0.0f;
  glmat[15] = 1.0f;
  glMultMatrixf(&(glmat[0]));
}
```

3.3.3.5. Finalização do PhysX

A finalização do motor e destruição da cena e de todos seus objetos é efetuada pela função `ReleasePhysics()` a seguir (esta é invocada pela função `ReleaseEngine()` da Seção 3.2).

```
void ReleasePhysics()
{
  if (gScene)
    gPhysicsSDK->releaseScene(*gScene);
  if (gPhysicsSDK)
    gPhysicsSDK->release(); // THE END
}
```

3.4. Inteligência Artificial em Jogos

Atualmente, os gráficos em jogos atingiram um patamar previsível de evolução baseado na capacidade de processamento das placas gráficas. Um novo paradigma gráfico ainda está por vir, muito provavelmente na forma de composição em tempo real de vídeos com objetos 3D sintéticos e dinâmicos, uma área ainda em pesquisa. A evolução visual na nova geração de consoles está na simulação física, principalmente aquela baseada nas PPUs. Entretanto, no presente momento da indústria de jogos, o foco não está na melhoria de gráficos e sons (que começam a ser pouco notados), mas na jogabilidade (*gameplay*) e no comportamento (onde personagens têm que ser tão bons quanto oponentes humanos online). Este novo foco está no cerne da questão de inteligência artificial (IA). A IA usada em jogos representa a última fronteira na área de desenvolvimento de jogos e enfrenta desafios gigantescos, começando pelo fato de que não consegue ser facilmente encapsulada em um motor (como os motores gráficos) nem embarcada em unidades dedicadas de hardware (como as PPUs para física). Até mesmo a questão de um middleware de IA continua polêmica, apesar dos esforços do grupo AIISC (*AI Interface Standards Committee*) do IGDA (http://www.igda.org/ai).

Na década de 2000, os estúdios de jogos começaram a aumentar as suas equipes de IA, influenciados pelo enorme sucesso do The SIMS. Este jogo, criado por Will Wright, demonstrou que a arquitetura de IA influencia toda a engenharia de software do jogo e conduz a características essenciais, tais como flexibilidade, extensibilidade e modularidade (os sucessivos módulos de expansão do The SIMS ilustram este ponto).

A IA usada em jogos segue princípios que a destacam da chamada IA clássica: tempo-real, simplicidade e reatividade. Estes princípios são tão fortes que caracterizam uma nova área chamada de IA de jogos (*game AI*). Uma introdução à IA de jogos deveria primeiramente estabelecer os fundamentos dos quatro tratamentos clássicos de IA[3]: sistemas agindo como humanos; sis-

[3] O projetista de IA, em um jogo, está sempre transgredindo os limites do que é correto, em favor do desempenho e da jogabilidade. Por esta razão, este deve ter a noção do quanto está se afastando do que é correto e completo em IA. Os projetistas de IA devem,

temas pensando como humanos (ciência cognitiva); sistemas pensando racionalmente (lógica); e sistemas agindo racionalmente (agentes racionais). A IA de jogos deve considerar elementos de todas estas abordagens e, neste sentido, trata de modelos híbridos. Entretanto, a abordagem de agentes racionais é a que melhor representa os princípios da IA de jogos[4]: (1) agir racionalmente significa o NPC (*non-player character*) agir para alcançar suas metas, considerando as suas crenças; (2) NPCs são agentes que percebem e agem; (3) fazer inferências corretas é apenas parte de ser um NPC racional, é apenas um mecanismo útil para atingir racionalidade; (4) alcançar racionalidade perfeita — sempre fazendo a coisa certa — não é possível em ambientes complexos, mas entender o que é tomada de decisão perfeita é um bom começo — para depois simplificar; (5) NPCs têm racionalidade limitada, isto é: agem adequadamente quando não existe tempo para fazer todas as computações que gostariam de fazer. Estes fundamentos não são tratados aqui, devido ao objetivo de apresentar as técnicas e abordagens práticas mais consagradas da IA de jogos.

No nível mais alto de abstração, as técnicas e abordagens em IA de jogos podem ser classificadas em técnicas usuais e consagradas, técnicas que são tendências (mas não estão amplamente difundidas) e as técnicas que são a fronteira do desenvolvimento (em geral, ainda em pesquisa e sem atender a todos os requisitos da IA de jogos — em particular, os de tempo-real). As *técnicas consagradas* são: busca de caminho com A* (A* *path finding*); máquina de estado finito (FSM); sistemas de gatilhos (*trigger systems*); e previsão de trajetória (e.g. em jogos de esporte). As *tendências*, por sua vez, são: comportamento emergente (aquele que não é explicitamente programado e que emerge da interação de unidades simples)[5] ; simulação de multidões (um caso especial de comportamento emergente)[6] ; IA de time (inicialmente denominado *squad AI*, trata de grupos de NPCs sem programa centralizador)[7]; LOD-AI (*level-of-detail*:

portanto, dominar a técnica clássica para depois começar a criar distorções interessantes, tal como um pintor que se torna pós-modernista somente após dominar e entender o clássico.

[4] A lista de características aqui apresentada é uma adaptação direta da melhor referência geral sobre IA no momento (Russell e Novig, 2002)

[5] É equivalente ao conceito de agentes sem mente (*mindless agents*) de Minsky (1985), de arquiteturas reativas em robótica (Brooks, 1991) e de agentes em animação comportamental (Costa e Feijó, 1996). Neste conceito, um personagem isolado é extremamente simples, mas que, devolvido ao ambiente, revela inteligência. Seguindo as idéias de Brooks (1991): a inteligência está no olho de quem vê (no caso, o jogador). Este tema, junto com o de *Team AI*, relaciona-se com as áreas de IA distribuída e de multi-agentes.

[6] Magistralmente explorada em Senhor dos Anéis, I Robot, King Kong e Narnia, através do software Massive da Weta Digital (http://www.massivesoftware.com/index.html).

[7] Os NPCs apenas publicam intenções e observações. A forma mais simples é a hierarquia de comando, típica de jogos de assalto. O *Team AI* apresenta uma forma de comportamento emergente.

expandir ou colapsar processamento de IA automaticamente)[8]; *feedback* entre NPCs e jogador (que representa uma solução para o paradoxo de não se saber se a cena inteligente não foi notada porque está muito bem feita ou se o jogador não está prestando atenção)[9]; raciocínio de terreno (onde o terreno não é mera topologia e geometria); lógica Fuzzy; aprendizado de máquina na foma mais simples (árvores de decisão e perceptrons); FSM avançadas (FSM hierárquicas com polimorfismo, FSM com arquitetura de subordinação — *subsumption*, FSM Fuzzy, FSM com LOD, múltiplas FSM coordenadas por *blackboard*). Na fronteira do desenvolvimento estão: incerteza matemática (redes Bayesianas e N-Gram, como principais candidatos); computação evolucionária (algoritmos genéticos, porém com sérias restrições de tempo real); vida artificial (*A-Life*, também com sérios problemas de tempo real); aprendizado de máquina (de reforço e redes neurais); planejamento; lógica clássica associada a arquiteturas reativas (modelos híbridos); modelagem de jogador. Quanto à classe de jogos, o prognóstico é o de haver investimentos pesados em jogos online multijogador em massa (*massive online multiplayer games*), que são mundos virtuais de dimensões planetárias, com assinantes fiéis, imunes à pirataria, gerando negociações extrajogo e envolvendo inúmeras áreas de ponta em computação.

3.4.1. Busca de Caminho com A (Path Finding)*

Um problema de busca é definido como o conjunto {*estado inicial*, *operadores*, *teste de meta*, *função de custo de trajetória*}, onde um operador é uma ação que leva um estado a outro e o custo de uma trajetória é a soma dos custos das ações individuais ao longo da trajetória. A solução deste problema é uma trajetória que vai do estado inicial a um estado que satisfaz o teste de meta. Buscam-se, sempre que possível, as soluções de menor custo. No problema de descoberta de caminhos (*path finding*) em um jogo, o mapa do terreno é geralmente representado por uma malha de pequenos quadrados com os seguintes elementos: os estados são as coordenadas de uma posição no mapa; o estado inicial é o ponto de partida; e os operadores são os oito movimentos possíveis ao redor de uma posição. Em *path finding*, as oito posições ao redor de um quadradinho são chamadas de vizinhos.

Encontrar uma solução para o problema de busca é descobrir um caminho na *árvore de busca*, onde cada nó da árvore se expande através da aplicação de operadores, (Figura 3.12). Cada nó na árvore de busca representa um estado. Deve-se observar que, a partir de cada nó na árvore, abrem-se b alternativas (este parâmetro é chamado de *fator de ramificação*). Dependendo das restrições, nem todos os operadores são legais. O nó que contém o estado inicial é chamado de *raiz*. A *profundidade* de um nó é o número de nós na

[8] Semelhante ao LOD gráfico. O termo LOD-AI termo é atribuído a Demis Hassabis, fundador do Elixir Studio e do jogo Republic: the Revolution (2003).

[9] Implementar este *feedback* influencia a jogabilidade, altera o projeto do jogo e exige LOD-AI.

trajetória que vai da raiz até este nó. Uma solução é uma trajetória que vai do estado inicial até o estado final (meta).

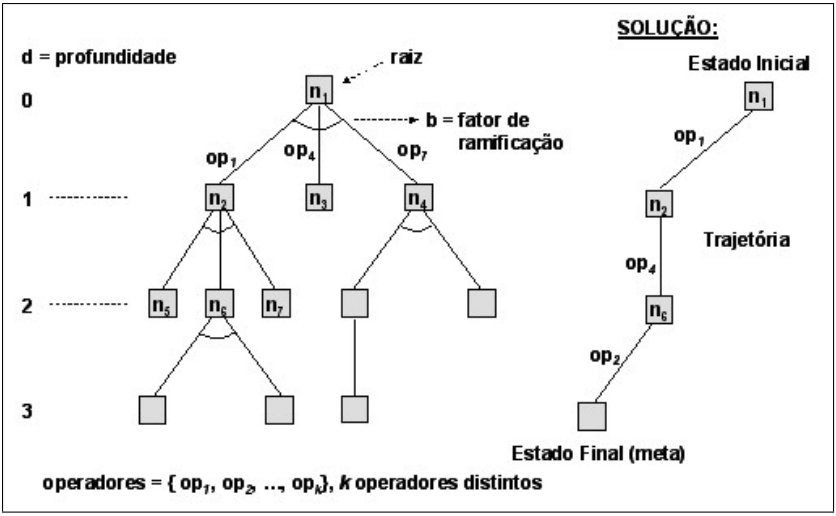


Figura 3.12. Árvore de busca e solução do problema

O pseudocódigo a seguir apresenta o algoritmo geral de busca, onde a notação `[nó...]` representa a estrutura `Nó={Estado,Operador,Nó_Pai, Custo_Trajetória}` de um nó específico da árvore, `{[n1...],[n2...], [n3...]}` representa uma trajetória e o ponto (`.`) indica um componente da estrutura (por exemplo, `Teste_Meta.p` significa o teste de meta do problema p). A `Função_Enfileiramento` insere a lista de novos candidatos na fila e define a estratégia de busca (BL: busca em largura, quando insere no final da fila; BP: busca em profundidade, quando insere na frente da fila; e busca *best-first* quando esta função ordena os novos candidatos de acordo com as suas chances de chegar à meta). BL e BP são buscas cegas, enquanto *best-first* é uma busca heurística (i.e. estima um provável custo da escolha, conforme uma função de avaliação heurística).

```
function busca(Função_Enfileiramento) return trajetória
n é uma estrutura Nó; lista e fila são uma lista de estruturas Nó
  n = nó inicial do problema // i.e. uma estrutura, e.g. [n1...] (Figura 3.12)
  lista = GeraListCandidatos(n) // expansão de n ({[n2...],[n3...],[n4...]})
  fila = lista
  loop
    if fila vazia then return vazio // i.e. trajetória vazia -> falha
    n = nó do topo de fila // e.g. [n2...]
    if Testa_Meta.problema(n) sucesso then return trajetória(n)
    fila = remove(n, fila) // remove n da fila, p. ex. {[n3...],[n4...]}
    lista = GeraListaCandidatos(n) // calcula custo, checa repetição e legalidade
    fila = Função_Enfileiramento(lista, fila) // insere lista na fila
    // fila = {[n3...],[n4...],[n5...],[n6...],[n7...]}
  end
```

A busca heurística A* (pronuncia-se “A estrela”) é uma *best-first* cuja função de avaliação é dada por $f(n) = g(n) + h(n)$, onde $g(n)$ é o custo da trajetória até n e $h(n)$ é uma estimativa do custo chamada de heurística. A Figura 3.13 mostra g e h para o problema de *path finding*.

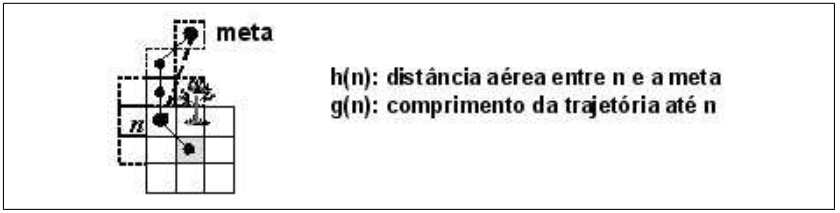
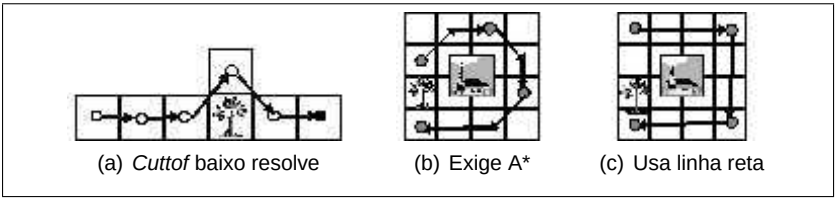


Figura 3.13. Busca heurística A*, onde $h(n)$ é uma reta ligando n à meta

Na prática, os jogos consideram as seguintes estratégias: (1) implementar as listas CLOSED e OPEN de nós; (2) estabelecer um *cutoff* para a árvore de busca determinado pelos seguintes valores: um custo $g + h$ (e.g. distância máxima que um NPC pode andar ou um custo limite de ações executadas no terreno); número de iterações do loop A* central; e máxima quantidade de memória permitida; (3) começar com *cutoff* baixo, pois muitas vezes existe uma trajetória quase linha reta, Figura 3.14(a); (4) adicionar pequeno atraso antes da próxima tentativa (para permitir a ocorrência de mudanças, como uma ponte destruída); (5) aceitar trajetórias parciais, como uma série de A* com *cutoffs* baixos (as vantagens são: isto torna os NPCs mais reativos, alertas e receptivos, devido aos movimentos imediatos; as eventuais paradas dão um aspecto mais humano ao NPC; isto também dá a impressão de que o NPC está explorando o ambiente desconhecido); (6) fechar os nós bloqueados imediatamente (adicionar imediatamente os filhos bloqueados à lista CLOSED mantêm a lista OPEN bem objetiva); (7) manter cache da trajetória que falhou (as observações são: isto evita a repetição de chamadas e os NPCs “empacados”; o cache deve ser limpo quando os objetos móveis mudarem de posição e quando se iniciar uma nova chamada do A*); (8) antes de disparar um A*, verificar se um algoritmo mais simples (tipo “teste de trajetória em linha reta”) é possível (grande parte dos movimentos é em linha reta); (9) definir *waypoints* (marcações-chave do terreno) que tiram vantagem do algoritmo tipo “teste de trajetória em linha reta”, Figura 3.14(b) e (c) ; (10) evitar algoritmos de *path finding* totalmente (e.g. se os pontos de início e fim não são visíveis para o jogador humano, deve-se transportar o NPC instantaneamente, respeitando, obviamente, as condições do jogo)[10].

[10] Para aprofundamento da prática com A*, recomenda-se a seqüência de artigos de Dan Higgins, o programador de IA do jogo de RTS (*real-time strategy*) chamado Empire Earth (Stainless Steel Studios) [Higgins 2002a, 2002b, 2002c].



(a) *Cuttof* baixo resolve (b) Exige A* (c) Usa linha reta

Figura 3.14. Situações de trajetórias em linha reta

3.4.2. Máquina de Estado Finito (FSM)

Uma máquina abstrata (ou autômato) é um modelo abstrato de uma máquina física que serve para modelar circuitos, sistemas lógicos e funções recursivas. A mais fundamental e simples máquina abstrata á a máquina de estado finito (*finite state machine*, ou FSM). As propriedades de uma FSM são: é finita; opera em instantes discretos; ações são seqüenciais; é determinística (i.e. a saída é função do estado inicial e do estímulo). Numa FSM, um *estado* s é uma condição da máquina em um tempo específico e uma *transição* T é a mudança de um estado em outro: $s_1 \longrightarrow s_2$.

Formalmente, FSM $= \langle \mathbf{S}, s_0, \mathbf{I}, T(s,i) \rangle$, onde $\mathbf{S}$ é um conjunto de estados, s_0 é o estado inicial, $\mathbf{I}$ é um alfabeto de entrada e $T(s,i)$ é uma função de transição que mapeia um estado e um símbolo de entrada em um próximo estado. Numa FSM, as ações podem ser associadas com as transições (máquina de Mealy, criada em 1955) ou com os estados (máquina de Moore, criada em 1956). Visualiza-se uma FSM através de um grafo direcionado (chamado de *diagrama de estado*), onde as arestas são transições. Na máquina de Mealy, a aresta é rotulada com símbolo-de-entrada/símbolo-de-saída. Na máquina de Moore, a aresta é rotulada com o símbolo-de-entrada apenas.

A máquina de Moore é uma 6-tupla M $= \langle \mathbf{S}, \mathbf{I}, \Delta, T(s,i), \lambda, s_0, \mathbf{A} \rangle$, contendo um conjunto (finito) de estados $\mathbf{S}$, um alfabeto de entrada $\mathbf{I}$, um alfabeto de saída Δ, uma função de transição $T : \mathbf{S} \times \mathbf{I} \rightarrow \mathbf{S}$, uma função de saída $\lambda : \mathbf{S} \times \Delta \rightarrow \Delta$, um estado inicial s_0, e um conjunto opcional de estados aceitáveis (chamado de reconhecedor) $\mathbf{A} \subseteq \mathbf{S}$. Numa autotransição, a máquina de Moore permanece no estado atual, isto é: $T(s,i) = s$. A máquina de Moore é a preferida em jogos.

A FSM usada na IA de jogos se afasta do formalismo original em uma série de pontos: permite estados dentro de estados; permite aleatoriedade em transições de estado; estado tem código e representa um comportamento; código é executado a cada *game tick* dentro de um estado; cada estado "conhece" as condições de transição; não há noção de estados aceitáveis (há o fim de execução da FSM); a entrada continua indefinidamente até que a FSM não é mais necessária ou o jogo termina. Na prática, uma FSM é uma descrição não-linear e concisa de como um objeto pode mudar seu estado ao longo do tempo, em resposta a eventos no seu ambiente. A Figura 3.15 ilustra um diagrama de transição de estados para um NPC que é um predador em um jogo.

Os três modos mais usuais de se implementar uma FSM são: (1) modo direto com código contendo uma longa seqüência de **`if`** e **`case`**; (2) linguagem

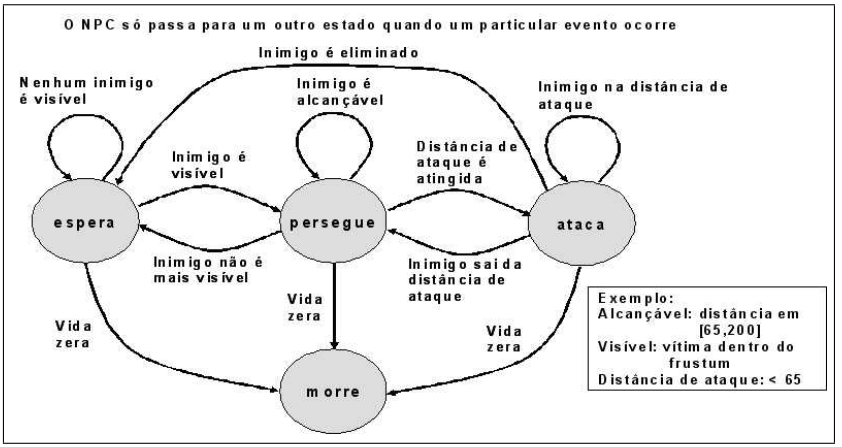


Figura 3.15. Máquina de estado para um NPC predador

assistida por macros; (3) linguagem script customizada (usando a linguagem Lua [Ierusalimschy 2006], por exemplo). Os problemas com o modo direto são: as seqüências de **`if`**s e **`case`**s não são adequadas para depuração; estados são do tipo **`int`** e seriam mais robustos e depuráveis como **`enum`**; a omissão de um simples **`break`** causa erros difíceis de serem encontrados; a lógica é difícil de ser organizada e concisa; não há como dizer que um estado foi acionado pela primeira vez (para, por exemplo, acionar códigos de inicialização); não há como monitorar ou registrar (log) a maneira como a máquina de estado se comporta ao longo do tempo; falta estrutura. Uma versão melhorada do modo direto é a que provê uma lógica de transição através de uma entrada que realiza a transição. Trata-se de uma situação próxima à máquina de Mealy (ações nas transições). Nesta versão, as ações de estado e a lógica de transição estão desacopladas. Os problemas nesta versão são: os códigos separados dificultam uma visão geral; pode haver cálculo de entrada que não é relevante. As funções abaixo ilustram, respectivamente, o modo direto e modo direto com lógica de transição desacoplada.

```
void RunStates(int* state) // modo direto
{
  switch (*state)
  {
    case 0:  // Espera
      espera();
      if (detecta_inimigo()) *state = 1;
      if (vida_acaba()) *state = 3;
      break;
    case 1:  // Persegue
      ...
  }
}
```

```
void RunStates(FSM* fsm) // com lógica de transição desacoplada
{
 int entrada = 0;
 switch (fsm->GetStateID())
 {
  case 0:  // Espera
   espera();
   if (detecta_inimigo()) entrada = DETECTA_INIMIGO;
   if (vida_acaba()) entrada = MORTO;
   break;
  ...
 }
 fsm->StateTransition(entrada);
}
```

As linguagens assistidas por macros são a forma mais avançada de se implementar uma FSM. Além de não ter os problemas do modo direto, este modo permite a consideração eficiente de mecanismos de mensagens. Em um jogo, os objetos são dirigidos a eventos (e não por mecanismos de percepção-ação que imitam o mundo real). Isto requer um sistema de mensagens com log e mecanismos de momento de entrega, como na seguinte estrutura: {nome: dano; de: dragão; para: cavaleiro; momento_de_entrega: 123.15; dados: 10}, onde dados é a quantidade de dano.

O código a seguir apresenta a proposta de Rabin (2000), onde os elementos da linguagem são: `BeginStateMachine`: começa a definição da FSM; `State(NameOfState)`: designa o começo de um particular estado; `OnEnter`: responde a um estado que entra e permite a inicialização; `OnExit`: responde ao estado sendo terminado e permite a limpeza; `OnUpdate`: responde à atualização do game *tick*; `OnMsg(NameOfMessage)`: responde a uma mensagem; `SetState(NameOfState)`: muda estados; `SendMsg()`: envia mensagem para qualquer objeto de jogo (`gameObj`); `SendDelayedMsg()`: envia uma mensagem com atraso para qualquer; `EndStateMachine`: termina a definição da FSM.

```
BeginStateMachine
 OnEnter
  SetState(STATE_espera)
 OnMsg(MSG_morto)
  destruir(gameObj);
 State(STATE_espera)
  OnEnter
   // Inicialização
  OnUpdate
   if (vida_acaba(gameObj))
    SetState(STATE_morre)
   if (detecta_inimigo(gameObj))
    SetState(STATE_persegue)
  OnExit
   // Finalização
```

```
  State(STATE_persegue)
    OnUpdate
      // calcula direção p/onde gameObj deve mover
      // movimenta gameObj um passo na direção calculada
      // executa animação de "caminha"
      if (distancia_inimigo(gameObj) < dist_min)
        SetState(STATE_ataque)
      if (distancia_inimigo(gameObj) > dist_ataque)
        SetState(STATE_espera)
  State ...
  ...
EndStateMachine
```

As definições dos macros são as seguintes (os **`return`**s ajudam a registrar se a mensagem foi entregue ou não; nomes de estados e de mensagens são tipos enumerados):

```
#define BeginStateMachine if(state < 0){
#define State(a) if(0) {return(true);}} else if(a == state){ if(0){
#define OnEnter return(true);} else if(MSG_RESERVED_Enter == msg->name){
#define OnExit return(true);} else if(MSG_RESERVED_Exit == msg->name){
#define OnUpdate return(true);} else if(MSG_RESERVED_Update == msg->name){
#define OnMsg(a) return(true);} else if(a == msg->name){
#define SetState(a) SetStateInGameObject(gameObj, (int)a);
#define EndStateMachine return(true);}} else {assert(!"Invalid State");\
  return(false);} return(false);

bool ProcessStateMachine(GameObject* gameObj, int state, MsgObject* msg)
{
  // Coloque aqui a FSM
}
```

3.5. Comentários Finais

Grande parte dos avanços e da queda de preço das tecnologias de hardware se deve à indústria de jogos digitais. Estes avanços facilitaram o desenvolvimento de algoritmos e técnicas para computação gráfica, inteligência artificial e simulação física em tempo real que são usados em muitas aplicações que vão além da indústria de entretenimento. Ademais, os jogos digitais contribuem fortemente para o desenvolvimento de arquiteturas capazes de utilizar todos estes recursos em ambientes integrados, sobretudo através dos conceitos de motor de jogo (*game engine*). Este capítulo apresenta uma sugestão para uma arquitetura deste tipo, indicando como desenvolver os principais componentes que devem obedecer aos requisitos de tempo real.

Referências

[AGEIA 2006] AGEIA (2006). Physx SDK documentation. Disponível em `www.ageia.com/pdf/PhysicsSDK.pdf`, último acesso em 15/03/2006.

[Baraff 2001] Baraff, D. (2001). Physically based modeling: Rigid body simulation. Disponível em `www.pixar.com/companyinfo/research/pbm2001/notes.pdf`, último acesso em 15/03/2006.

[Brooks 1991] Brooks, R. A. (1991). Intelligence without reason. A.I. Memo 1293, Artificial Intelligence Laboratory, MIT.

[Costa and Feijó 1996] Costa, M. and Feijó, B. (1996). Agents with emotions in behavioral animation. *Computer & Graphics*, 20(3):377–384.

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

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

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

[Higgins 2002a] Higgins, D. (2002a). Generic A* pathfinding. In Rabin, S., editor, *AI Game Programming Wisdom*, pages 114–121. Charles River Media, Hingham, Massachusetts.

[Higgins 2002b] Higgins, D. (2002b). How to achieve lightning-fast A*. In Rabin, S., editor, *AI Game Programming Wisdom*, pages 133–145. Charles River Media, Hingham, Massachusetts.

[Higgins 2002c] Higgins, D. (2002c). Pathfinding design architecture. In Rabin, S., editor, *AI Game Programming Wisdom*, pages 122–132. Charles River Media, Hingham, Massachusetts.

[Ierusalimschy 2006] Ierusalimschy, R. (2006). *Programming in Lua*. Lua.org.

[LaMothe 2003] LaMothe, A. (2003). *Tricks of the 3D Game Programming Gurus: Advanced 3D Graphics and Rasterization*. Pearson Education.

[Minsky 1985] Minsky, M. (1985). *The Society of Mind*. Simon & Schuster.

[Rabin 2000] Rabin, S. (2000). Designing a general robust ai engine. In DeLoura, M. A., editor, *Game Programming Gems*, pages 221–236. Charles River Media, Rockland, Massachusetts.

[Randima 2004] Randima, F. (2004). *GPU GEMS: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison Wesley.

[Randima 2005] Randima, F. (2005). *GPU GEMS II: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison Wesley.

[Rollings and Morris 2004] Rollings, A. and Morris, D. (2004). *Game Architecture and Design: A New Edition*. New Riders Publishers.

[Rost 2004] Rost, R. J. (2004). *OpenGL(R) Shading Language*. Addison Wesley.

[Russell and Novig 2002] Russell, S. and Novig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, second edition.

[Schwab 2004] Schwab, B. (2004). *AI Game Engine Programming*. Game Development Series. Charles River Media.

[Zerbest 2005] Zerbest, S. (2005). *3D Game Engine Programming*. XTreme Games LCC. Premier Press.