UNIVERSIDADE DO EXTREMO SUL CATARINENSE - UNESC

CURSO DE CIÊNCIA DA COMPUTAÇÃO

DANIEL MILAK NATAL

REUSO, EXTENSIBILIDADE E SIMULAÇÃO DE FÍSICA EM MOTORES PARA JOGOS DIGITAIS

CRICIÚMA, JULHO DE 2010

DANIEL MILAK NATAL

REUSO, EXTENSIBILIDADE E SIMULAÇÃO DE FÍSICA EM MOTORES PARA JOGOS DIGITAIS

Trabalho de Conclusão de Curso apresentado para obtenção do Grau de Bacharel em Ciência da Computação da Universidade do Extremo Sul Catarinense.

Orientadora: Profª. MSc. Leila Laís Gonçalves

CRICIÚMA, JULHO DE 2010

DANIEL MILAK NATAL

REUSO, EXTENSIBILIDADE E SIMULAÇÃO DE FÍSICA EM MOTORES PARA JOGOS DIGITAIS

Submetido ao corpo docente do Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense como um dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Profa. MSc. Ana Claudia Garcia Barbosa
Coordenadora do Curso de Ciência da Computação

Banca Examinadora:

Profa. MSc. Leila Laís Gonçalves (UNESC)
Orientadora

Profa. MSc. Ana Claudia Garcia Barbosa (UNESC)

Prof. MEng. Evânio Ramos Nicoleit (UNESC)

RESUMO

O realismo dos jogos digitais vem evoluindo a cada dia, seguindo a evolução do *hardware*. Com isso seu processo de desenvolvimento se torna mais complexo e longo, o reuso de *software* no desenvolvimento de jogos digitais vem para facilitar, fazendo com que o desenvolvedor se preocupe apenas com o domínio da aplicação, proporcionando também maior produtividade, redução dos custos e esforços. Os comportamentos dos elementos presentes nos jogos são baseados nas leis da física, esta simulação é responsabilidade do motor de física. Este trabalho visa avaliar a simulação física, reuso e extensibilidade de alguns dos motores disponíveis atualmente (Bullet, ODE e PhysX), para isso foi feito um estudo detalhado destes itens de cada um dos motores, para então, poder realizar testes, comparações e avaliações entres estes motores. Com a análise dos resultados pôde-se notar que nenhum dos motores é melhor em todos os aspectos avaliados, porém, pode-se indicar Bullet como a melhor alternativa de código livre, tendo em vista seus resultados e funcionalidades.

Palavras-chave: **Jogos Digitais, Reuso, Extensibilidade, Simulação de Física, Motores de Física.**

ABSTRACT

The realism of digital games has been evolving every day, following the hardware's evolution. Thus, their process of development become more complex and longer, the reuse of software in the development of digital games comes to make things easier, so that developer can focus just on the application domains, also providing more produtivity, lower costs and efforts. The elements' behaviors present in games are based on laws of physics, this simulation is responsability of the physics engine. This study aims to evaluate the physical simulation, reuse and extensibility of some of currently available engines (Bullet, ODE e PhysX). For this, a detailed study of these items of each engine was done, so that tests, comparisons and evaluations among these engines could be done. With the result's analysis could be noticed that no one of the engines is better in any evaluated aspects, however, Bullet can be pointed out as the better open source choice, in view of its results and features.

Keywords: **Digital Games, Reuse, Extensibility, Physics' Simulation, Physics Engines.**

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

2D – Bidimensional

3D – Tridimensional

8-DOPs – *Eight-Direction Discrete Orientation Polytope*

AABB – *Axis Aligned Bounding Box*

API – *Application Programing Interface*

AWT – *Abstract Window Toolkit*

BV – *Bounding Volume*

BVH – *Bounding Volume Hierarchy*

CBD – *Component-Based Development*

CBSE – *Component-Based Software Engineering*

CORBA – *Common ORB Architecture*

CPU – *Computer Processor Unit*

DLL – *Dynamic-link library*

DOO – Desenvolvimento Orientado a Objetos

GPU – *Graphics Processor Unit*

IA – Inteligência Artificial

IDL – *Interface Definition Language*

NPC – *Non-player Character*

OBB – *Oriented Bounding Boxes*

OO – Orientação a Objetos

ORB – *Object Request Broker*

PPU – *Physics Processing Unit*

RMI – *Remote Method Invocation*

SAP – *Sweep and Prune*

SDK – *Software Development Kit*

XML – *Extensible Markup Language*

SUMARIO

1 INTRODUÇÃO

Um jogo por computador pode ser definido como um sistema composto de três partes básicas: enredo, motor e interface interativa (BATTAIOLA et al, 2001). O enredo define o tema, a trama, o(s) objetivo(s) do jogo, o qual por meio de uma série de passos o usuário deve se esforçar para atingir. A interface interativa controla a comunicação entre o motor e o usuário, reportando graficamente um novo estado do jogo. O motor do jogo (*game engine*) é o seu sistema de controle, o mecanismo que controla a reação do jogo em função de uma ação do usuário.

O conceito de motores de jogos surgiu com o lançamento do jogo Doom da companhia norte-americana iD Software no final de 1994. De acordo com Bishop et al (1998, tradução nossa), um motor de jogos é um *framework*, ou seja, uma estrutura pré-implementada responsável por todos os aspectos independentes de um jogo como: gráficos, sons, inteligência artificial, física, rede, controle, entre outros.

A implementação do motor envolve diversos aspectos computacionais, tais como: a escolha apropriada da linguagem de programação em função de sua facilidade de uso e portabilidade, o desenvolvimento de algoritmos específicos, o tipo de interface com o usuário, etc (HECKER, apud MEIRELES et al, 2006). Outros conceitos envolvidos são modularidade, extensibilidade e reutilização. Toda essa gama de requisitos torna o motor um artefato de *software* complexo e de alto custo.

A utilização de motores de jogos já desenvolvidos possibilita acelerar a produção de novos jogos, sendo que a maioria dos motores permite que os utilizadores possam modificar cenários, modelos, sons e outros componentes do jogo, a partir de modificações (*mods*) ou adições (*adds*), ampliando as possibilidades no desenvolvimento (NAKAMURA

apud MEIRELES et al, 2006). O motor deve ser independente da linguagem de programação, facilmente integrável a um jogo e atualizável.

A utilização de motores no desenvolvimento de jogos envolve estudo e análise da sua arquitetura de alto-nível para compreender e melhor utilizar seus componentes, interfaces de comunicação, desempenho, abstrações dos componentes no nível de modelo para a componentização no nível de código, e demais questões pertinentes. É importante compreender a arquitetura e composição dos motores dos jogos que de acordo com Bittencourt e Osório (2006), um motor é composto por diversos “submotores”, sendo cada um responsável por tratar um tipo de atuação nos jogos. Os principais componentes são de interação, visualização, som, interconexão em rede, física e inteligência artificial.

Alguns dos problemas relatados na bibliografia com relação à utilização de motores de jogos são: a integração dos submotores na geração do código fonte; componentização das funcionalidades envolvendo encapsulamento e abstração; desempenho e qualidade da gerencia, apresentação, manipulação e reação realística dos objetos do jogo obedecendo às leis da física; formalização e padronização da arquitetura dos jogos bem como a documentação sobre seus componentes; e os critérios de escolha dos motores para o desenvolvimento de jogos.

A falta de formalização da arquitetura de jogos digitais é uma das barreiras mais significativas para o reuso de componentes prejudicando também a portabilidade de jogos para diferentes ambientes. Esforços no sentido de buscar uma arquitetura padrão, bem como associar aos motores técnicas de engenharia de *software*, são discutidos por Falstein (1997), Joselli (2007), Madeira, Ramalho e Ferraz (2001), Siebra, Ramalho e Frery (2000), Zamith (2007), entre outros autores com o objetivo de prover níveis satisfatórios de desempenho, de robustez, de modularidade, de abstração, de reusabilidade de código, de padronização, etc.

Um dos problemas na escolha do motor de jogo mais adequado é a dificuldade em fazer uma comparação direta entre as diversas funcionalidades apresentadas pelos motores, uma vez que quase todos apresentam as características esperadas numa aplicação deste tipo.

1.1 OBJETIVO GERAL

Avaliar motores de física para jogos digitais, visando o reuso, extensibilidade e simulação de física no contexto de jogos digitais.

1.2 OBJETIVOS ESPECIFICOS

a) contextualizar o reuso e a extensibilidade no desenvolvimento de aplicações;
b) compreender os motores de jogos, em especial, os submotores de física;
c) identificar situações e possibilidades de reuso e extensibilidade em submotores de física;
d) entender os conceitos e funcionalidades compreendidos na simulação de física nos motores;
e) realizar testes, validações e comparações entre submotores de física envolvendo simulação de física, reuso e extensibilidade para avaliação de desempenho, realismo dos resultados, abstração, modularidade, produtividade e integração;
f) documentar a arquitetura e o interfaceamento entre os componentes dos submotores de física.

1.3 JUSTIFICATIVA

A investigação sobre os submotores de física busca melhorar a eficiência e eficácia do processo de desenvolvimento de jogos digitais. Inicialmente, para implementar física, cada efeito necessário era programado diretamente no código-fonte do jogo aplicado apenas para aquela circunstância. Se um jogo necessitasse reproduzir a trajetória de uma flecha, então era programada uma equação específica apenas para calcular a trajetória da flecha naquele jogo, esta equação seria inútil para simular qualquer outra coisa, mas para simular a trajetória de uma flecha seria perfeita. Este modelo de desenvolvimento de jogos é possível ser aplicado em jogos simples, com pouco código-fonte e uso limitado da física. Porém, com o aumento da complexidade dos jogos e da qualidade gráfica ficou mais difícil conseguir uma boa aparência física nos comportamentos dos elementos dos jogos, isso encorajou os desenvolvedores a buscar soluções mais genéricas e reutilizáveis para o desenvolvimento da simulação física nos jogos digitais. A solução encontrada foi a modularidade, ou seja, dividir o projeto do jogo em motores com funcionalidades distintas que possam ser desenvolvidos independentemente e reutilizados em outros jogos. Um motor de jogo deve disponibilizar um conjunto de funcionalidades para que se possa simular algum comportamento do jogo, sem a necessidade do programador saber como o motor está implementado. Como Millington (2007, tradução nossa) sugere, um motor de física é basicamente uma grande calculadora, ele tem a matemática necessária para simular a física, porém ela não sabe o que precisa ser simulada, ele necessita de dados específicos dos elementos do jogo para então poder simular seus comportamentos físicos.

Para conhecer melhor as necessidades e problemas encontrados no reuso de motores no desenvolvimento de jogos é necessário investigar questões relacionadas à engenharia dos jogos e comportamento de seus componentes. A partir da engenharia dos

componentes pode-se obter, entre outras informações, como os motores implementam essas funcionalidades; como se dá a integração dos componentes responsáveis por cada função; como se dá o interfaceamento entre os submotores; como é a arquitetura de um jogo digital e qual a sua composição mínima para geração de um jogo; quais as possibilidades de integração entre os motores e submotores de jogos; como é realizado o reuso de componentes no desenvolvimento de jogos. Com testes, validações e comparações é possível conhecer como os componentes se comportam e quais os resultados de sua aplicação.

A escolha pela avaliação específica do submotor física se dá pelo fato de se encontrar pouca bibliografia sobre o assunto sendo que o motor gráfico é o mais abordado. Outro motivo é a importância do tema no desenvolvimento de jogos digitais, pois está diretamente relacionado à interatividade do jogo, que é uma das principais características que atraem e prendem o jogador ao jogo, portanto tem que ser muito bem estudada e implementada, pois cada erro ou inconsistência pode torná-lo menos atrativo. A aplicação das leis da física nos jogos digitais tem como objetivo aumentar o realismo dos mesmos sendo responsável por todo o comportamento físico dos elementos do jogo. Por meio da simulação cinemática pode-se reproduzir a movimentação de carros, aviões ou outros objetos; por meio da simulação da dinâmica podem-se reproduzir efeitos como gravidade, atrito, aceleração, desaceleração, colisão; a simulação da aplicação de forças em corpos rígidos ou deformáveis e também as reações as colisões entre os corpos; a simulação de efeitos naturais como água, fogo, vento e explosões (BITTENCOURT, 2006).

Há muitos motores de física disponíveis tanto comercialmente quanto de forma gratuita (*open source* ou *freeware*), entre eles, Havok Physics, PhysX, Newton, Tokamak, Bullet, ODE e OpenTissue. A seleção dos motores para comparação levou em conta suas funcionalidades, sua validação em jogos comerciais, sua documentação, e sua disponibilização, sendo selecionados os motores Bullet, ODE e PhysX.

Com este trabalho, buscou-se contribuir com o estudo sobre engenharia de jogos digitais; na formalização da arquitetura e na ampliação de documentação e bibliografia sobre submotores de física; no reuso de componentes na elaboração de jogos; e no crescimento da cultura de desenvolvimento de jogos digitais no curso de Ciência da Computação da UNESC e do Laboratório de Informática Aplicada com mais uma área de pesquisa.

1.4 ESTRUTURA DO TRABALHO

O foco deste trabalho foram os motores de física dos jogos digitais cujo principal objetivo é caracterizar o reuso e a extensibilidade no desenvolvimento de jogos com o uso destes motores e avaliar a simulação de física no comportamento dos objetos presentes na cena do jogo.

No Capítulo 2 apresenta a caracterização do reuso e da extensibilidade a partir do levantamento bibliográfico das principais abordagens de desenvolvimento de aplicações com estes fins, tratando da orientação a objetos, componentes e frameworks.

O estudo da arquitetura geral e funcionalidades dos motores de jogos, em especial o de física a partir da análise de exemplos e da literatura sob o assunto, está descrito no Capítulo 3. Neste capítulo também foi abordada a simulação de física nos motores e identificadas funcionalidades que tratam dos fenômenos de física nos jogos, em especial a detecção de colisões e dinâmica de corpos rígidos.

Os trabalhos correlatos são apresentados no capítulo 4 abordando os temas simulação física e detecção de colisão.

O Capítulo 5 trata da avaliação dos submotores de física Bullet, ODE e PhysX, abordando o reuso, extensibilidade e simulação de física. Foram aplicados testes, validações e comparações entre os submotores de física para avaliação de desempenho, realismo dos

resultados, abstração, modularidade, produtividade e integração. Neste capítulo são apresentadas a arquitetura e o interfaceamento entre os componentes dos submotores de física avaliados e discutidos os resultados.

2 O REUSO E EXTENSIBILIDADE NO DESENVOLVIMENTO DE APLICAÇÕES

Este capítulo apresenta a caracterização do reuso e da extensibilidade a partir do levantamento bibliográfico das principais abordagens de desenvolvimento de aplicações com estes fins, tratando da orientação a objetos, componentes e frameworks.

2.1 O REUSO NO DESENVOLVIMENTO DE APLICAÇÕES

Diante do aumento da demanda e da complexidade no desenvolvimento de *software* e de exigências como produtividade, qualidade, redução de custos e esforços, novas perspectivas e esforços vêm sendo aplicados buscando atender estes quesitos. Pesquisas em Engenharia de *Software* visam buscar diferentes abordagens para melhorar a qualidade dos artefatos de *software* e reduzir o tempo e o esforço necessários para produzi-los (FAIRLEY, 1986).

A reutilização vem sendo indicada como ponto chave para dar suporte a essas, e ainda outras, necessidades e exigências de desenvolvimento de *software* sendo uma tendência crescente neste processo. Dentre os objetivos da reutilização estão aumento da produtividade, facilitação na implantação e integração de *softwares* ou processos, visto que os *softwares* estão cada vez maiores e mais complexos. Com isso, cresce paralelamente a necessidade de maior confiabilidade, usabilidade, robustez, flexibilidade e adaptabilidade. Entre os grandes desafios dos desenvolvedores estão trabalhar com esta complexidade, diminuir custos e tempo de produção e aumentar sua qualidade (CRNKOVIC; LARSSON, 2002, tradução nossa).

Sommerville (2007, tradução nossa) afirma que a reutilização é a única maneira de lidar com a complexidade dos sistemas de *software* de hoje proporcionando mais qualidade e rapidez no desenvolvimento. Esta afirmação considera a perspectiva de que o reuso de

artefatos de *software* já desenvolvidos e depurados possa reduzir o tempo de desenvolvimento e de testes além de minimizar a inserção de erros na produção de novos artefatos.

O reuso não é uma idéia nova, vem sendo utilizada desde o inicio da computação com o aproveitamento de trechos de código, idéias, processos e abstrações, porém, para estes problemas é necessária uma abordagem mais sistemática (PRESSMAN, 2006). Em muitas abordagens o reuso não está de forma inerente ao desenvolvimento, no qual não são definidos o que se pode ou não reutilizar, como também não são formalizadas as possíveis alterações, tornando assim, a reutilização mais difícil e problemática.

Para D'Souza e Wills (1999), um artefato reutilizável é uma parte do trabalho que pode ser utilizado em mais de um projeto incluindo código-fonte, código compilado, casos de teste, modelos, interfaces para usuário, planos e estratégias, entre outros.

A reutilização de artefatos de *software* pode ser realizada em nível de código, de análise e projeto. Na reutilização de código utilizam-se trechos de código já desenvolvidos em novos programas sendo a forma mais elementar de reuso. Como exemplos de sistematizações utilizadas nesta prática podem-se citar o uso de bibliotecas (por exemplo, DLL[1]), interfaces (por exemplo, API[2]) e *frameworks*. A reutilização de análise e projeto consiste no reaproveitamento de concepções arquitetônicas anteriormente planejadas de um novo artefato de *software* e, de acordo com Deutsch (1989), a reutilização dos produtos destas etapas é mais significativa que a reutilização de trechos de código.

A granularidade do artefato de *software* é um fator relevante no reuso. A reutilização de rotinas (por exemplo, uso de bibliotecas de funções) corresponde a um nível de granularidade inferior ao reuso de módulo que corresponde ao reuso de classes em orientação a objetos (MEYER, 1988).

---

[1]DLL - *Dynamic-link library* (ou Biblioteca de ligação dinâmica) é a implementação para o conceito de bibliotecas compartilhadas nos sistemas operacionais Microsoft Windows e OS/2.

[2]API - *Application Programing Interface* (ou Interface de Programação de Aplicativos) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por programas aplicativos que não querem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços.

Como conceitos básicos relevantes para o reuso efetivo citam-se (KRUEGER, 1992): abstração, seleção, adaptação e integração.

A abstração é a característica que possibilita ao desenvolvedor subtrair detalhes do artefato relacionado à implementação do código e visualizar a parte abstrata que contém as informações mais conceituais necessárias à reutilização. No reuso é fundamental o processo de seleção dos artefatos de *software* que irão atender às necessidades da aplicação. Por princípio, qualquer artefato pode ser reutilizado em um contexto diferente ao qual foi inicialmente projetado inicialmente podendo ocorrer adaptações para o novo contexto (por exemplo, parametrização, configuração ou pequenas modificações). A integração dos artefatos, para geração da arquitetura da aplicação final, inclui desafios como manutenção da consistência e padronização, detecção de necessidades de adaptação ou modificação e realização de testes para validar funcionalidades e desempenho da aplicação final.

Dentre as abordagens que sugerem o reuso de forma extensiva citam-se o desenvolvimento baseado em objetos, desenvolvimento baseado em componentes (CBD – *Component-Based Development)* e a abordagem de *frameworks* discutidas a seguir.

2.1.1 Desenvolvimento Baseado em Objetos e o Reuso

O reuso constitui um dos fatores mais atrativos do desenvolvimento orientado a objetos (DOO) na busca de produtividade no processo de desenvolvimento. No DOO a reutilização pode se dar no nível de código, em específico o reuso das classes. A reutilização de classes pode ocorrer sob enfoques diferentes (MEYER, 1988):

a) composição: com o reuso de classes disponíveis em bibliotecas para gerar novos objetos da implementação;

b) herança: aproveitamento da concepção de classes disponíveis em bibliotecas, no desenvolvimento de outras, a partir de herança;
c) polimorfismo: possibilidade de um objeto assumir diferentes formas a partir da implementação diferenciada de uma operação em classes diferentes.

Na reutilização de classes, cabe ao desenvolvedor estabelecer a arquitetura da aplicação de forma a organizar os elementos que a compõem e o fluxo de controle entre eles. Além disto, há em comum o fato da reutilização ser de artefatos isolados, cabendo ao desenvolvedor estabelecer sua conexão ao sistema em desenvolvimento.

Em DOO é possível reutilizar requisitos, análise, projeto, planejamento de testes, interfaces de usuários e arquiteturas, podendo ser encapsulados como objetos reusáveis praticamente todos os componentes do ciclo de vida da engenharia de software (YOURDON 1999).

Sommerville (2003) apresenta alguns motivos pelos quais a orientação a objetos não atingiu seu potencial máximo apontando a granularidade dos objetos, o detalhamento e especificidade das classes de objetos como obstáculos ao reuso extensivo. Muitas vezes se torna necessário o conhecimento aprofundado das classes e a disponibilização de seu código-fonte para viabilizar sua utilização. Outra questão está na composição, exigem que as classes estejam associadas ao projeto, obrigando nova compilação a cada atualização das classes de objeto que compõem a aplicação.

2.1.2 Desenvolvimento Baseado em Componentes e o Reuso

A Engenharia de *Software* Baseada em Componentes, também chamada de *Component-Based Software Engineering* (CBSE), surge no final da década de 90 e Sommerville (2007) a define como “o processo de definição, implementação e integração ou

composição de componentes independentes fracamente acoplados ao sistema" (p. 440, tradução nossa). O autor afirma ainda que a frustração causada pela abordagem orientada a objetos quanto ao reuso acabou motivando o surgimento de uma nova abordagem baseada em componentes (CBD – *Component-Based Development).* Ainda segundo o autor, os componentes são mais abstratos do que as classes de objetos e podem ser considerados como provedores de serviço *stand-alone*, ou seja, o componente presta serviços quando solicitado pelo sistema, independente de onde este componente está ou da linguagem de programação utilizada em seu desenvolvimento.

A abordagem da CBSE restabelece a idéia de reutilização, onde os componentes são desenvolvidos e preparados especificamente para a construção de *softwares* (CRNKOVIC; LARSSON, 2002, tradução nossa).

O foco é deslocado da programação para a composição do software, considerando que não há a necessidade de re-implementar o que pode-se reutilizar, como justifica Clements (1995, p. 02, tradução nossa):

> CBSE está mudando o modo pelo qual grandes sistemas de *software* são desenvolvidos. CBSE incorpora a filosofia "comprar, em vez de construir" abraçada por Fred Brooks e outros. Do mesmo modo que as primeiras sub-rotinas liberaram o programador de pensar sobre detalhes, CBSE desloca a ênfase da programação de *software* para a composição de sistemas de *software*. A implementação deu lugar a integração como foco. Na sua fundamentação está a pressuposição de que há muitos pontos em comum, em vários sistemas grandes de *software*, para justificar o desenvolvimento de componentes reusáveis para explorar e satisfazer esses pontos em comum.

Como elementos da CBSE, temos componentes independentes, padrões de componentes em modelos, *middleware* e um processo de desenvolvimento (SOMMERVILLE, 2007, tradução nossa). Componentes independentes são completamente especificados por suas interfaces onde a implementação do componente pode ser alterada sem alterar o sistema. O uso de padrões de componentes em modelos visa definir como as interfaces devem ser especificadas e como os componentes se comunicam, facilitam a integração entre eles. A possibilidade de integração entre componentes independentes e

distribuídos se dá pelo uso de *middleware*. Para aproveitar o máximo de seu potencial, é necessário um processo de desenvolvimento que se adapte a Engenharia de *Software* Baseada em Componentes.

Os principais objetivos da CBSE são os seguintes (CRNKOVIC; LARSSON, 2002, tradução nossa):

a) fornecer suporte ao desenvolvimento de sistemas como conjuntos de componentes;
b) apoiar o desenvolvimento de componentes reutilizáveis;
c) facilitar a manutenção e aprimoramento de sistemas por meio de customizações ou substituição de componentes.

A CBSE apresenta a seguintes vantagens (CLEMENTS, 1995, tradução nossa; SOMMERVILLE, 2003):

a) redução nos custos gerais de desenvolvimento, não será necessário especificar, projetar, implementar e validar todos os componentes utilizados no sistema de software;
b) maior confiabilidade, componentes já reutilizados devem ser mais confiáveis do que componentes novos, pois já foram experimentados e testados em diferentes ambientes, assim, provavelmente já tiveram suas falhas corrigidas;
c) redução dos riscos de processo, as incertezas sobre os custos relacionados com o reuso de um componente são menores do que sobre os custos de desenvolvimento das funcionalidades necessárias;
d) desenvolvimento acelerado, com o reuso de componentes o tempo de desenvolvimento e validação são reduzidos, pressupondo-se que a busca por componentes adequados não consuma muito tempo;

e) maior flexibilidade, o reuso de componente dispensa o conhecimento detalhado da implementação do componente, qualquer componente que satisfaça os requisitos do sistema pode ser utilizado, o que gera mais competitividade por parte dos desenvolvedores de componentes e mais opções de escolha para os desenvolvedores de sistemas de *software*.

Embora CBSE esteja se desenvolvendo rapidamente em integrar uma abordagem de desenvolvimento de *software*, existe uma série de problemas associados com o reuso, que podem inibir a introdução desse método e significar que as reduções no custo total de desenvolvimento, com reuso, serão menores do que as previstas. Os principais são problemas apontados por Sommerville (2003; 2007, tradução nossa) são:

a) aumento nos custos de manutenção, se o código-fonte do componente não estiver disponível os custos de manutenção poderão aumentar já que as mudanças no sistema podem trazer incompatibilidades aos componentes;
b) síndrome do “não-foi-criado-aqui”, por falta de confiança nos componentes reutilizáveis, engenheiros de *software* as vezes preferem reescrever todo o componente, pois acreditam que farão melhor que o componente já disponível;
c) criação e manutenção de uma biblioteca de componentes, pode ser caro criar uma biblioteca de componentes reutilizáveis e assegurar que os desenvolvedores podem utilizá-la, pois as técnicas atuais de classificação, catalogação e recuperação de componentes ainda são imaturas;
d) encontrar e adaptar componentes reutilizáveis, os engenheiros precisam ter uma razoável certeza de que poderão encontrar um componente antes de incluírem a rotina de busca de componente como parte do processo de desenvolvimento;

e) confiabilidade dos componentes, quando os componentes são caixa-preta e o código-fonte pode não estar disponível ao usuário, neste caso, como saber se o componente é confiável?

f) certificação de componentes, uma questão relacionada a confiabilidade dos componentes, foi proposto que avaliadores independentes devem certificar os componentes a fim de garantir sua confiabilidade, porém, ainda não se sabe como isso será feito;

g) equilíbrio de requisitos, deve-se encontrar um equilíbrio entre os requisitos ideais e os componentes disponíveis nos processo de especificação e desenvolvimento do sistema. Para isso, hoje é usado um processo intuitivo, por isso se necessita de um processo mais sistemático e estruturado para auxiliar na seleção e configuração dos componentes.

2.1.2.1 Componente

De maneira geral, um componente pode ser entendido como uma unidade independente de *software* que pode ser composta com outros componentes para criar um sistema de *software* (SOMMERVILLE, 2007, tradução nossa). Porém, ainda não há um consenso geral sobre um conceito mais especifico e claro, diversos autores apresentam conceitos diferentes sobre o que é e como deve ser especificado um componente.

Uma das primeiras definições de componente de *software* na literatura apareceu em 1987 (SZYPERSKI, 2002, tradução nossa). Mostra a idéia de reuso definindo um componente de *software* como uma parte de *software* reutilizável, encapsulada e fracamente acoplada (BOOCH, 1987, tradução nossa).

Council and Heinemann (2001, p. 40, tradução nossa) enfocam em sua definição a conformidade dos componentes com o modelo de componente, “é um elemento de *software* que está em conformidade com um modelo de componente, pode ser implantado independentemente e composto sem modificações de acordo com um padrão de composição”. Também mencionando que um padrão de composição define como os componentes podem ser compostos; que um modelo de componentes define como os componentes devem ser construídos e como podem se comunicar com outros componentes; e que é necessário uma implementação de modelo de componente para dar suporte a execução de componentes em um modelo de componentes.

Szyperski (2002, p. 41, tradução nossa) aborda em sua definição aspectos sobre composição, interfaces, independência, implantação e a possibilidade de comercialização – “Um componente de *software* é uma unidade de composição com interfaces contratualmente especificadas e com dependências de contexto explícitas apenas. Um componente de *software* pode ser implantado de forma independente e está sujeito a composição por terceiros”. Dessa forma, os componentes podem ser desenvolvidos de forma independente da tecnologia de implementação da aplicação. Por exemplo, pode ser utilizado o paradigma de orientação a objetos aplicando estruturas de classes interrelacionadas, com visibilidade externa limitada (KRAMER & CRAWFORD, 1996).

As definições encontradas nas literaturas abordam estas e outras características, porém, são bastante abstratas e não definem claramente o que é uma componente. Para Sommerville (2007, tradução nossa), as características essenciais de um componente utilizado em CBSE são:

a) padronização, componentes devem estar em conformidade com algum modelo de componentes. Modelos de componentes podem definir as interfaces, metadados, composição e implantação dos componentes;

b) independência, componentes devem ser independentes, ou seja, deve ser possível compor e implantar componentes sem a necessidade de outros componentes específicos;
c) composição, todas as interações externas devem ocorrer publicamente através de interfaces para que componentes possam ser compostos;
d) implantação, componentes devem ser auto-contidos e capaz de operar de forma autônoma em uma plataforma que implementa o modelo de componentes;
e) documentação, para que os potenciais usuários possam decidir quanto a utilização de um componente, este componente deve estar totalmente documentado. Todas as interfaces devem estar devidamente documentadas.

Quanto ao nível de abstração, ou seja, a visibilidade da implementação do componente por de trás de sua interface, os componentes podem ser classificados em caixa-branca (*whitebox*), caixa-preta (*blackbox*) e caixa-cinza (*graybox*) (SZYPERSKI, 2002, tradução nossa). De acordo com o autor, na abstração caixa-branca a implementação dos componentes é totalmente visível aos usuários (desenvolvedores). O reuso é feito através de interfaces, porém o acesso a implementação possibilita seu estudo a fim de melhor entender o funcionamento do componente. Já na abstração caixa-preta, os usuários não podem visualizar a implementação do componente. Os componentes são reutilizados apenas em base de suas interfaces e especificações. A abstração caixa-cinza é semelhante à abstração caixa-preta, porém, uma parte da implementação é visível aos usuários de forma controlada, que pode ser analisada como uma especificação ou alterada.

2.1.2.2 Modelos de Componentes

Modelos de componentes são essenciais na CBSE, pois definem padrões utilizados por desenvolvedores para implementar, documentar e implantar componentes a fim de garantir principalmente a interoperabilidade entre eles, e também outros atributos de qualidade como evolução e manutenção (GAO; TSAO; WU, 2003, tradução nossa).

Segundo Weinreich e Sametinger (2001, tradução nossa) os principais elementos de um modelo de componente ideal são: interfaces; convenção de nomes; metadados; interoperabilidade; customização; composição; suporte a evolução; e empacotamento e implantação descritos a seguir.

Interface é um dos elementos centrais de um modelo de componentes. O modelo de componentes define como os serviços oferecidos pelo componente devem ser descritos por meio de interfaces, definindo também os elementos que podem constituir a interface, como o nome dos serviços, seus parâmetros e os tipos válidos de parâmetros. Para descrever estas interfaces e elementos muitos modelos de componentes têm uma Linguagem de Definição de Interfaces ou IDL (*Interface Definition Language*). O modelo de componentes também pode definir interfaces especificas que devem ser implementadas pelos componentes, que são utilizadas pela implementação do modelo de componentes para prover serviços esperados por outros componentes que estão em conformidade com o modelo.

Um modelo de componente deve seguir um padrão de nomeação para seus componentes e interfaces a fim de garantir que sejam unicamente identificáveis globalmente. As duas principais abordagens de convenção de nomes são os identificadores únicos e os espaços de nomes hierárquicos. Identificadores únicos são gerados por ferramentas dedicadas usando uma combinação de dados específicos para garantir que todos os identificadores gerados sejam únicos. Em espaços de nomes hierárquicos, um item sempre recebe como parte

de seu identificador, o identificador do item de seu nível superior, garantindo que o identificador seja único desde que o nível mais alto seja globalmente único.

Os metadados são as informações sobre os componentes, interfaces e seus relacionamentos. O modelo de componentes especifica como os metadados devem ser descritos e como podem ser recuperados. A recuperação é feita por meio de serviços providos pela implementação do modelo de componentes.

A interoperabilidade deve garantir que componentes de diferentes fornecedores possam se conectar e trocar informações por um canal de comunicação bem definido. As convenções de chamadas devem ser padronizadas pelo modelo de componente a nível binário quando componentes são implementados em diferentes linguagens.

A customização é definida como a capacidade do componente ser adaptado pelo consumidor antes de ser instalado ou usado. A customização permite modificação e inclusão de propriedades simples ou comportamentos complexos.

Dois ou mais componentes podem se combinar a fim de gerar uma nova funcionalidade. O padrão de composição de componente pode criar uma estrutura maior, chamada de infra-estrutura de componentes ou *framework* de componentes, inserindo e conectando componentes em uma estrutura já definida. O modelo de componentes deve definir como as interfaces de componentes devem suportar essas composições.

É importante que modelos de componentes definam regras e padrões para a evolução dos componentes. Componentes podem ter suas funcionalidades melhoradas e outras incluídas, necessitando incluir ou alterar suas interfaces, o ideal, é que usuários já existentes não sofram nenhum efeito em razão destas evoluções. Às vezes é necessário que diferentes versões de interfaces ou componentes co-existam no mesmo sistema.

Um modelo de componente deve descrever como componentes são empacotados para que possam ser implantados independentemente. A descrição de implantação prove as

informações necessárias para o processo implantação e o conteúdo do pacote. O pacote deve conter tudo que o fornecedor do componente espera que possa não existir em uma infra-estrutura de componente.

2.1.2.3 Interface de Componentes

Componentes oferecem serviços por meio de pontos de acesso, cada serviço prestado possui um ponto de acesso, a interface de componentes pode ser definida como a especificação destes pontos de acesso. A interface de um componente nomeia a coleção de serviços do componente e prove as descrições e protocolos destes serviços, assim separando o cliente da implementação dos serviços do componente. Isso torna possível a substituição da implementação sem alterar a interface e a adição de novas interfaces sem alterar a implementação já existente (CRNKOVIC; LARSSON, 2002, tradução nossa).

Uma interface não é uma parte constituinte do componente, mas serve como um contrato entre o componente e seus clientes. O cliente solicita serviços de um componente, e este, prove a implementação destes serviços. Sendo que a interface pode definir restrições de uso que devem ser consideradas pelo componente e cliente (WEINREICH, 2001, tradução nossa).

Componentes geralmente têm dois tipos de interfaces que se relacionam, como mostra a Figura 1.

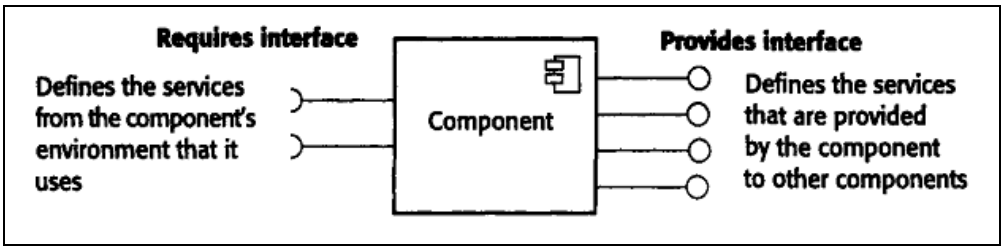

Figura 1. Tipos de interfaces de componentes
Fonte: adaptado de Sommerville (2007)

A interface provedora, que descreve os serviços prestados pelo componente para outros componentes ou para o ambiente, fundamentalmente, é a API do componente. Esta interface define os métodos que podem ser chamados pelos usuários do componente; e a interface requerida, que especificam os serviços que devem ser providos ao componente por outro componente ou pelo ambiente. O componente funcionará apenas se estes serviços estiverem disponíveis, isto não compromete a independência e implantação do componente, pois ele não determina que um componente específico deva ser utilizado. Na Figura 1 foi utilizada para representar as interfaces provedoras uma linha e circulo, já para as interfaces requeridas uma linha e semi-circulo, denotando o relacionamento entre as interfaces provedoras e as interfaces requeridas. (SOMMERVILLE, 2007, tradução nossa).

2.1.3 Framework

Segundo os autores Buschmann et al (1996), Johnson & Foote (1988), Pinto (2000) e Pree (1995) um *framework* é definido como um *software* incompleto projetado para ser instanciado. O *framework* é composto de um conjunto de recursos e conceitos correlacionados que criam uma plataforma de desenvolvimento que explora um modelo ou arquitetura de solução computacional. Esta plataforma de desenvolvimento define uma arquitetura abstrata que une códigos com características similares a determinado contexto de *software* provendo funcionalidades comuns que podem ser utilizadas no desenvolvimento de uma nova aplicação.

Os *frameworks* têm sido utilizados para o auxílio no desenvolvimento de aplicações complexas cujo objetivo é possibilitar o reuso de código e de projeto através do estabelecimento de um modelo de construção de aplicações pertencentes a um determinado

domínio. Sendo assim, o uso de um *framework* visa amenizar o custo de desenvolvimento de aplicações deste domínio (FAYAD et al, 1999).

O reuso na abordagem de *framework* se dá no nível de código e projeto, que pode diminuir o tempo e o esforço exigidos na produção de *software*. Em contrapartida, é complexo desenvolver *frameworks*, bem como aprender a usá-los.

Assim como nos componentes, o nível de abstração dos *frameworks* pode ser classificado, de acordo com Fayad et al (1997), em caixa-branca, caixa-preta e caixa-cinza. No nível de abstração caixa-branca é necessário, para uso do *framework*, o conhecimento aprofundado da estrutura das classes e o seu fluxo de informações e seu reuso é por meio de composição ou definição de interfaces para os artefatos. Já no *framework* com nível de abstração caixa-preta não é necessário conhecer o funcionamento de sua estrutura interna para utilizá-lo e o reuso pode ser efetuado por composição ou definição de interfaces para os artefatos. O nível de abstração caixa-cinza é um híbrido dos níveis caixa-branca e caixa-preta e o reuso se dá por herança, associação dinâmica e definição de interfaces.

Os *frameworks* podem ser classificados em dois grupos principais: *frameworks* de aplicação orientado a objetos (FAYAD et al, 1999) e *framework* de componentes (SZYPERSKI, 2002, tradução nossa).

2.1.3.1 Frameworks de Aplicação Orientados a Objetos

De acordo com Fayad et al (1999) e Johnson & Foote (1988), um *framework* de aplicação orientado a objetos é um conjunto de classes que constitui um projeto abstrato para a solução de uma família de problemas.

Johnson (1991) e Gamma et al (1995) definem este tipo de *framework* como um conjunto de objetos que colaboram com o objetivo de atender a um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação.

Quanto à estrutura de um *framework* de aplicação orientado a objetos, Mattsson (1996, 2000) identifica-o como uma arquitetura desenvolvida com o objetivo atingir máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização.

O foco deste tipo de *framework* é gerar famílias de aplicações orientadas a objetos. Seus pontos de extensão são definidos como classes abstratas ou interfaces, que são estendidas ou implementadas por cada instância da família de aplicações.

Fayad et al (1999) classificam *frameworks* de aplicação orientado a objetos quanto ao seu escopo definindo-os como: *frameworks* de integração de *middleware*, *frameworks* de infra-estrutura de sistemas e *frameworks* de aplicações corporativas.

Frameworks de integração de *middleware* são usados para integrar aplicações e componentes distribuídos em uma mesma arquitetura. Exemplos de *frameworks* de *middleware* são o CORBA (*Common ORB*[3] *Architecture*) e o RMI (*Remote Method Invocation*).

Os *frameworks* de infra-estrutura de sistema simplificam o desenvolvimento tratando de questões de projeto como sistemas operacionais, comunicação, interfaces gráficas e linguagens de programação. Um exemplo de *framework* de infra-estrutura é o AWT (*Abstract Window Toolkit*).

Já os *frameworks* de aplicação são desenvolvidos para um domínio de aplicação específico abordando questões de projeto. Existem diferentes domínios de aplicação contemplados com soluções de *framework* de aplicação como telecomunicações, finanças,

[3] ORB (*Object Request Broker*) é um componente de software cuja função é facilitar a comunicação entre objetos.

produção, educação e jogos digitais. *Frameworks* de aplicações possibilitam que uma família de produtos seja gerada a partir de uma única estrutura (PINTO, 2000) que encapsula o conhecimento sobre os conceitos mais gerais do domínio (FAYAD et al, 1999).

2.1.3.2 Framework de Componentes

O *framework* de componentes tem por objetivo, por meio de um conjunto de componentes que podem interagir entre si, fornecer diversas funcionalidades como numa biblioteca do tipo *toolbox* (SZYPERSKI, 2002, tradução nossa).

Szyperski (2002, p. 425, tradução nossa) define um *framework* de componentes como:

> Um *framework* de componentes é uma entidade de *software* que dá suporte a componentes que estão em conformidade com certos padrões e permite que instâncias destes componentes se "pluguem" ao *framework* de componentes. O *framework* de componentes estabelece condições de ambiente para instancias de componentes e regula a interação entre estas instancias de componentes.

O *framework* de componentes aceita a inserção dinâmica de instancias de componentes em tempo de execução, assim, podendo criar uma ilha para determinados componentes que cooperam entre si ou com outros *frameworks*.

O objetivo principal dos *frameworks* de componentes é dar suporte ao processo de composição de sistemas de *software* a fim de torná-lo mais eficiente e preciso. Os desenvolvedores, tanto de componentes como de sistemas, não necessitam conhecer os detalhes de funcionamento do *framework*. Desenvolvedores de componentes devem conhecer os padrões e formatos requeridos pelo *framework* para desenvolvimento e especificação de componentes, enquanto os desenvolvedores de sistemas deverão saber seu funcionamento básico para usar suas funcionalidades (CRNKOVIC; LARSSON, 2002, tradução nossa).

Os conceitos de modelos de componentes e *frameworks* de componentes podem ser interligados. O modelo de componentes define os padrões e convenções para o desenvolvimento de componentes, o *framework* de componentes é infra-estrutura que apóia o modelo de componentes, se componentes são desenvolvidos conforme um determinado modelo de componentes eles podem ser integrados em um *framework*. Bachman et al (2000, tradução nossa) descreve as relações entre os conceitos de componentes de *software* na Figura 2.

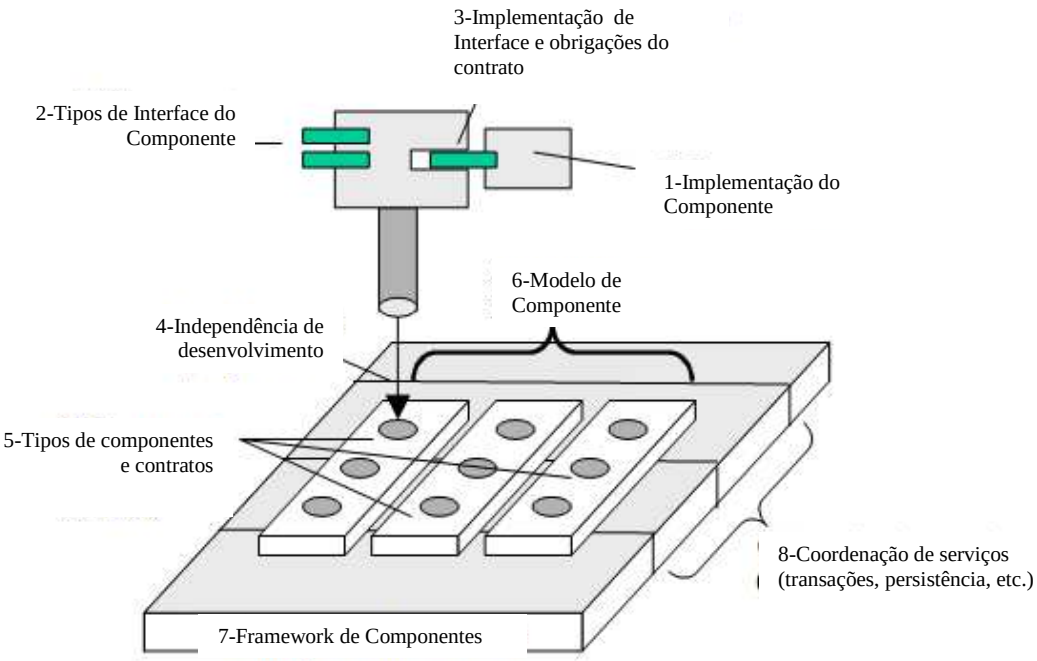

Figura 2. Elementos de um Framework de Componentes
Fonte: adaptado de Bachman et al (2000)

Como mostra a Figura 2, um componente (item 1) é uma implementação de *software* que é executada quando necessário e implementa uma ou mais interfaces (item 2), para isso o componente deve satisfazer certas obrigações por contratos (item 3). Estes contratos garantem que a interação entre componentes independentes (item 4). Um sistema baseado em componentes baseia-se em diferentes tipos de componentes (item 5), cada um desempenha um determinado papel no sistema e é descrito por uma interface. Um modelo de componente

(item 6) é um conjunto de tipos de componentes com suas interfaces e uma especificação de padrões que controlam a interação entre os tipos de componentes. Um *framework* de componentes (item 7) prove e coordena serviços (item 8) em tempo de execução que apóia e faz cumprir o modelo de componentes (BACHMAN et al, 2000, tradução nossa). Este modelo é bastante encontrado em tecnologias de componentes de *softwares* comerciais, como Sun Microsystems' Enterprise JavaBeans™ e Microsoft's COM+.

2.2 EXTENSIBILIDADE NO DESENVOLVIMENTO DE APLICAÇÕES

Uma das características de artefatos reutilizáveis é sua generalidade quanto a conceitos e funcionalidades do domínio tratado contando que possa ser especializado com o mínimo esforço possível para a obtenção de aplicações específicas maximizando o reuso no desenvolvimento de *software* (SILVA & PRICE, 1998). As especializações (modificações ou extensões) que podem ou não ser realizadas em um artefato reutilizável depende de suas características de flexibilidade e extensibilidade conceituadas por Fayad & Cline (1996) e Meyer (1988). De acordo com os autores, a flexibilidade se refere à possibilidade de alterar funcionalidades do artefato e a extensibilidade permite ampliar suas funcionalidade, ambas sem conseqüências imprevistas sobre o conjunto da estrutura.

As modificações e/ou extensões nos artefatos de *software* para reuso podem ocorrer diretamente em nível de código, por meio de especializações (herança, polimorfismo, combinação, hot spots), ou ainda customização (interfaces) (BELLUR, 1998). É importante o uso de padrões de projeto (*Design Patterns*) como os definidos por Gamma et al (1994), para possibilitar redução de tamanho e complexidade do código.

Artefatos com baixo nível de abstração as adaptações podem ser feitas por meio de alterações diretamente em seu código. Os artefatos reutilizáveis a partir de bibliotecas de

classes (por exemplo, DLL) e com nível de abstração caixa-branca são estendidos a partir de herança de classes do *framework* e ligação dinâmica (*dynamic binding*) (SZYPERSKI, 2002, tradução nossa).

Em *frameworks*, seja orientado a objetos ou de componentes, projeto de classes ou de componentes devem prever em que pontos o *framework* deve ser estendido ou modificado para que as aplicações provenientes dele respondam às particularidades dos problemas do domínio. Estes pontos são denominados de *hot spots*, pontos de variação ou ainda pontos de extensão. Estes pontos são os locais em que o *framework* é flexível nos quais as adaptações para um funcionamento específico podem ser realizadas. Os pontos fixos são chamados de *frozen spots* que definem a arquitetura geral da aplicação com seus componentes básicos e os relacionamentos entre eles permanecendo inalterados em todas as instanciações do *framework* de aplicação.

A extensão da arquitetura de *frameworks* caixa-preta é realizada a partir de interfaces definidas (conhecidas como API) e integração de componentes em um *framework* que utiliza padrões de projeto. Assim como a customização de componentes é feita por meio de interfaces, pois são as únicas partes visíveis do componente. O ideal seria que a semântica de cada serviço do componente fosse especificada, pois é importante para a implementação do componente e para os clientes que o usam. Porém, na maioria dos modelos de componentes, é definido apenas os a sintaxe da interface (parâmetros de entrada e saída) e uma descrição básica do que serviço faz (CRNKOVIC; LARSSON, 2002, tradução nossa).

2.3 CONSIDERAÇÕES SOBRE REUSO E EXTENSIBILIDADE

A reutilização de artefatos de *software* pode ser realizada nos níveis de código, análise e projeto e sua efetividade está diretamente relacionada à forma de produção e aplicação em

um domínio específico. Dentre as características mais importantes na produção de artefatos de *softwares* reutilizáveis destaca-se a modularidade, granularidade, nível de abstração e interfaceamento.

Segundo Meyer (1988), a modularidade deve visar à criação de módulos com interfaces pequenas, explícitas e em pequena quantidade, além de aplicar o princípio da ocultação de informação (abstração). Como estruturas modulares, considerando a produção de artefatos reutilizáveis, têm-se as bibliotecas de classes, *frameworks* e os componentes. Neighbors (1989) indica como problema no uso de bibliotecas de artefatos de *software*, a dificuldade de seleção de um artefato adequado à aplicação em desenvolvimento, sendo que, em alguns casos, é mais fácil produzir um novo artefato do que buscar um em uma biblioteca.

A granularidade do artefato está associada ao aumento de produtividade na medida em que se alcança um maior nível no reuso. O nível de granularidade aumenta de acordo como o artefato reutilizado e do seu nível de complexidade (do menor para maior nível): bibliotecas de funções (reutilização de rotinas), módulos (reutilização de classes), *framework* e componentes (reutilização de código e projeto). Sendo assim, reutilizar classes, artefato de *software* mais complexo e de maior nível de granularidade, tende a ser mais eficiente que reutilizar rotinas isoladas. A reutilização de *frameworks* e componentes tem um nível de granularidade superior à reutilização de classes, por proverem reuso de classes ou conjunto de componentes interligados, ao invés de isolados, além de possibilitarem o reuso de código e projeto o que lhes conferem um nível de granularidade superior às demais abordagens citadas.

A abstração tem como objetivo desconsiderar os aspectos não relevantes na efetivação do reuso do artefato de *software*. O nível de abstração de cada artefato está associado à quantidade de informações necessárias para sua manipulação. Quanto mais global é a informação, mais alto é seu grau de abstração e quanto mais detalhada ela for, menor é seu grau de abstração. A abstração em OO é realizada por meio de classes que representam um

objeto complexo e o encapsulamento esconde detalhes de utilização deste objeto (métodos e atributos). Já em *frameworks* e componentes o nível de abstração da implementação é relacionado ao grau de conhecimento da estrutura interna de funcionamento necessário á sua utilização.

O interfaceamento entre artefatos de *software* reutilizáveis pode ser mais ou menos especializado de acordo com a abstração e a arquitetura destes artefatos possibilitando reuso, comunicação, interconexão e transferência de dados dentro da aplicação. No reuso de rotinas e de bibliotecas de classes, por utilizar artefatos isolados, é o desenvolvedor que estabelece a arquitetura da aplicação definindo a organização dos elementos, o fluxo de controle e a conexão entre os artefatos.

A extensibilidade do artefato reutilizável depende de suas possibilidades de ampliação. Quanto maior for o nível de abstração do artefato mais especializado é o método de extensão e mais eficaz se dá o seu reuso.

Quanto maior o nível de abstração do artefato de *software* (objetos, biblioteca de classes, *frameworks*, componentes) a ser reutilizado, maior a sua granularidade, melhor o potencial de reuso (código, análise e projeto), maior é a produtividade no desenvolvimento e mais especializados são os recursos para a extensão da aplicação (código, herança, customização, interfaces), como mostra a Figura 3.

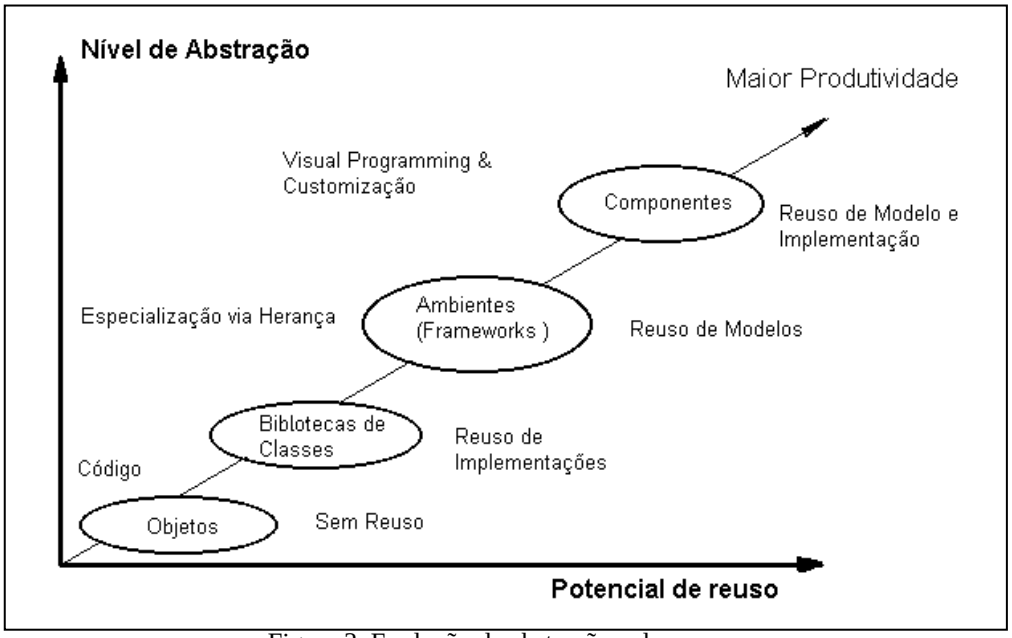

Figura 3. Evolução da abstração e do reuso.
Fonte: Bellur (1998)

3 MOTORES DE JOGOS DIGITAIS

Um jogo digital (ou videojogo ou jogo eletrônicos) é uma expressão genérica que se refere a jogos eletrônicos desenvolvidos para serem jogados num computador, num console ou outro dispositivo tecnológico, envolvendo interação entre ser humano e computador, recorrendo ao uso de tecnologia (GEE, 2003).

O desenvolvimento de um jogo digital pode ser realizado em etapas sendo a primeira a etapa de projeto, onde é criado um documento chamado *Design Document*. Nesta etapa inicial são especificados os componentes do jogo, suas funcionalidade e interfaces, entre outros. Também devem ser selecionadas as ferramentas para cada atividade envolvida no desenvolvimento (BITTENCOURT; OSÓRIO, 2006):

- ferramentas de gerência de projetos;
- editores de gráficos 2D;
- ferramentas de modelagem e animação 3D;
- editores de áudio para efeito sonoro e trilha sonora;
- ferramentas de edição de níveis de jogos;
- ferramentas de suporte ao desenvolvimento do jogo.

Existem muitas alternativas de ferramentas de suporte ao desenvolvimento de jogos digitais das quais se podem citar: código de máquina (Assembly), bibliotecas[4] (como OpenGL/DirectX/SDL), *toolkits*[5] (Crystal Space), SDKs[6] (Dark Game SDK) ou motores de jogos (ODE, Havok, Half live).

Para Lewis & Jacobson (2002, p. 28, tradução nossa), motor de jogo (ou *game engine*) é uma "coleção de módulos de simulação que não especifica diretamente o

[4] Bibliotecas são designadas por um conjunto de funções para desempenhar alguma tarefa.
[5] Toolkits são coleções de classes que oferecem um conjunto de serviços.
[6] SDK (*Software Development Kit* ou Kit de Desenvolvimento de Software) é composto por um conjunto de softwares e demais artefatos usados para programar outro software.

comportamento do jogo (lógica do jogo) e o ambiente do jogo (mapa)" para não perder o objetivo da reusabilidade. Segundo os autores, os motores incluem módulos para capturar eventos de entrada (ação do usuário), gerar saída gráfica e de áudio (resposta ao usuário) e gerenciar a dinâmica do mundo de jogo (interação entre motor e interface visual) conforme Figura 4.

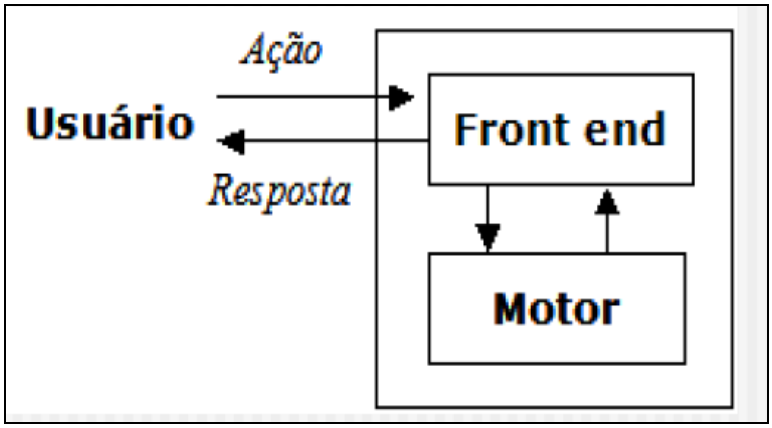

Figura 4. Esquema genérico de um motor de jogo
Fonte: adaptado de Bittencourt e Osório (2006)

O motor do jogo é o componente responsável pelo sistema de controle, o mecanismo que controla a reação do jogo em função de uma ação do usuário. A implementação do motor envolve aspectos computacionais, tais como, a escolha apropriada da linguagem de programação em função de sua facilidade de uso e portatilidade, o desenvolvimento de algoritmos específicos, o tipo de interface com o usuário, operações como detecção de colisão, animação de *sprites*, renderização, que são necessárias a todo jogo (BATTAIOLA et al, 2001).

Inicialmente, todas as funcionalidades necessárias em um jogo digital eram desenvolvidas no projeto daquele jogo de forma especifica para aquele jogo. Todo o gerenciamento gráfico, controle, interação com o usuário e outros aspectos, quando necessários, estavam no código-fonte do jogo, programado especificamente para as

necessidades daquele jogo, ou seja, não poderia ser utilizado em outro jogo a não ser que tinham exatamente a mesma necessidade. Este modelo de desenvolvimento pode ser aplicado a jogos simples, com pouco código-fonte e uso limitado dos recursos de *hardware* disponível. Porém, a evolução do *hardware* tornou possível a criação de jogos mais complexos, de melhor qualidade gráfica, ficando mais difícil ter domínio de todas as funcionalidades necessárias e gerenciar todos os comportamentos de um jogo. Isso encorajou os desenvolvedores a buscar melhores soluções para o desenvolvimento de jogos digitais. A solução encontrada foi a modularidade, ou seja, dividir o projeto de desenvolvimento de um jogo em motores com funcionalidades distintas que podem ser desenvolvidos independentemente e reutilizados em outros jogos.

Dentro do desenvolvimento de *softwares*, os motores de jogos são *frameworks* que funcionam como uma estrutura que dará suporte ao desenvolvimento de diferentes aplicações por meio do reuso de bibliotecas de códigos, linguagens de *script* e outros *softwares* que dêem à sustentação necessária para a elaboração e funcionamento da aplicação. A estrutura pré-implementada (*framework*) do motor é responsável por controlar todos os aspectos independentes de um jogo como: gráfico, áudio, inteligência artificial, física, rede, controle, entre outros (BISHOP et al, 1998, tradução nossa). O termo “motor” é utilizado para se referir ao motor de jogo num todo, ele contém vários “submotores”, que controlam cada um destes aspectos do jogo, também são chamados de motor gráfico, motor de áudio, motor de inteligência artificial, motor de física, etc. (ZERBST; DÜVEL, 2004, tradução nossa). Ainda para os autores, cada submotor, ou módulo, deve gerenciar todos os dados em sua área de responsabilidade; computar todos os dados de acordo com as tarefas de sua área; quando necessário, passar dados para outras instancias e receber dados de outras instancias para gerenciar e calcular (Figura 5).

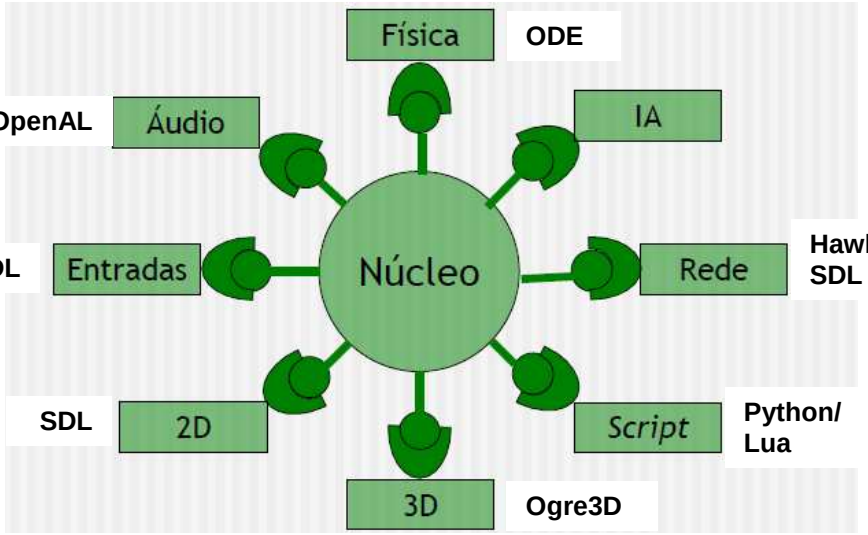

Figura 5. Módulos de motores de jogos
Fonte: adaptado de Bittencourt e Osório (2006)

Dois objetivos concorrentes de motores de jogos são simplicidade e flexibilidade. A perfeição seria que um motor seja 100% flexível e com maior facilidade de uso possível, porém isso não é possível, quanto mais flexibilidade é adicionada ao motor, menos simplicidade de uso ele terá, pois necessitará de mais funções e parâmetros; e quanto mais simples o uso, mais inflexível é o motor. O ideal, é que o desenvolvedor encontre um equilíbrio adequado entre a flexibilidade e a simplicidade de uso (ZERBST; DÜVEL, 2004, tradução nossa).

Os motores de jogos (*game engines*) proporcionam uma redução significativa no ciclo de desenvolvimento deste tipo de aplicação. De acordo com Furtado e Santos (2008), o uso de ferramentas do tipo *engines* permite tornar o desenvolvimento de jogos um processo automatizado, padronizado e rápido oferecendo um conjunto de funcionalidades ao programador como especificação, em alto nível, do comportamento do jogo, explicitação da detecção de colisão, gerenciamento do mapa de fundo.

De acordo com Millington (2007, tradução nossa), existem duas grandes vantagens de se utilizar um motor de física em jogos:

a) economia de tempo devido a utilização de solução de integração de motor de física na aplicação, não despendendo de tempo para desenvolver uma solução própria;
b) qualidade no resultado obtida com o uso de soluções pré-existentes especializadas em simulação física.

Existem disponíveis algumas dezenas de motores, alguns gratuitos e de código aberto, outros proprietários e pagos, entre eles cita-se: Antiryad Gx, Microsoft XNA Game Studio, jGame, Knight Free 3D Suite, Nvidia PerfKit, AGS, Blitzbasic, ODE, Guff, JFrog, Crystal Space, Amphibian.

É importante observar que a escolha do motor de jogo está diretamente relacionada às demais ferramentas de suporte ao desenvolvimento, pois geralmente motores de jogos têm restrições quanto a funcionalidades, modelos e formatos de arquivos, sendo que as demais ferramentas devem ser compatíveis com as especificações do motor escolhido.

Este capítulo aborda ainda a arquitetura de motores de jogos, detalhando os submotores, gráficos, de inteligência artificial e, com maior enfoque, nos submotores de física.

3.1 ARQUITETURA DOS MOTORES DE JOGOS DIGITAIS

Um motor de jogo possui uma arquitetura de *software* que possibilita a interligação de uma serie de componentes a fim de propiciar uma experiência gráfica interativa em tempo real para os usuários. O conhecimento sobre técnicas de engenharia de *software*, como, *frameworks*, padrões de projeto e componentização são essenciais para o desenvolvimento de um motor de jogo. Além de enfocar, em sua arquitetura, o nível de generalização de uso do motor. Quanto maior o nível de abstração, maior é o impacto no

desempenho da aplicação, ou seja, para um motor de jogo poder atender vários gêneros de jogos digitais e diversas plataformas é necessária uma arquitetura com muitas abstrações capazes de generalizar as diferentes especificidades de gêneros e plataformas, o que provavelmente tornará seu uso mais complexo (BITTENCOURT; OSÓRIO, 2006).

Há uma pré-definição de concepção genérica da arquitetura do motor de um jogo citada por Filion (appud BATTAIOLA et al, 2001) composta pelos gerenciadores de entrada, gráfico, de som, de inteligência artificial, de múltiplos jogadores, de objetos, do mundo, principal; editor de cenários e objeto do jogo. Apesar desta concepção genérica, a formalização dos componentes do motor e sua especificação para determinadas aplicações ainda é um campo em aberto (CORLEY, 1998). Esta falta de formalização prejudica a portabilidade, ou seja, o uso de jogos em diferentes ambientes, bem como o uso de módulos específicos. Esforços têm sido realizados no sentido de definir uma arquitetura padrão, bem como associar aos motores técnicas de engenharia de *software*, de forma a garantir níveis satisfatórios de desempenho, de robustez, de modularidade, de reusabilidade de código, de padronização, etc (MADEIRA, RAMALHO e FERRAZ, 2001).

Existem varias propostas de arquiteturas para motores de jogos, porém nenhuma levada como padrão para criação de motores. Baseando-se na modularização, Bittencourt (2004 apud BITTENCOURT; OSÓRIO, 2006) define uma estrutura genérica para motores de jogos, contendo os componentes básicos de um motor de jogo (interação, comunicação, controle e visualização), como mostra a Figura 6.

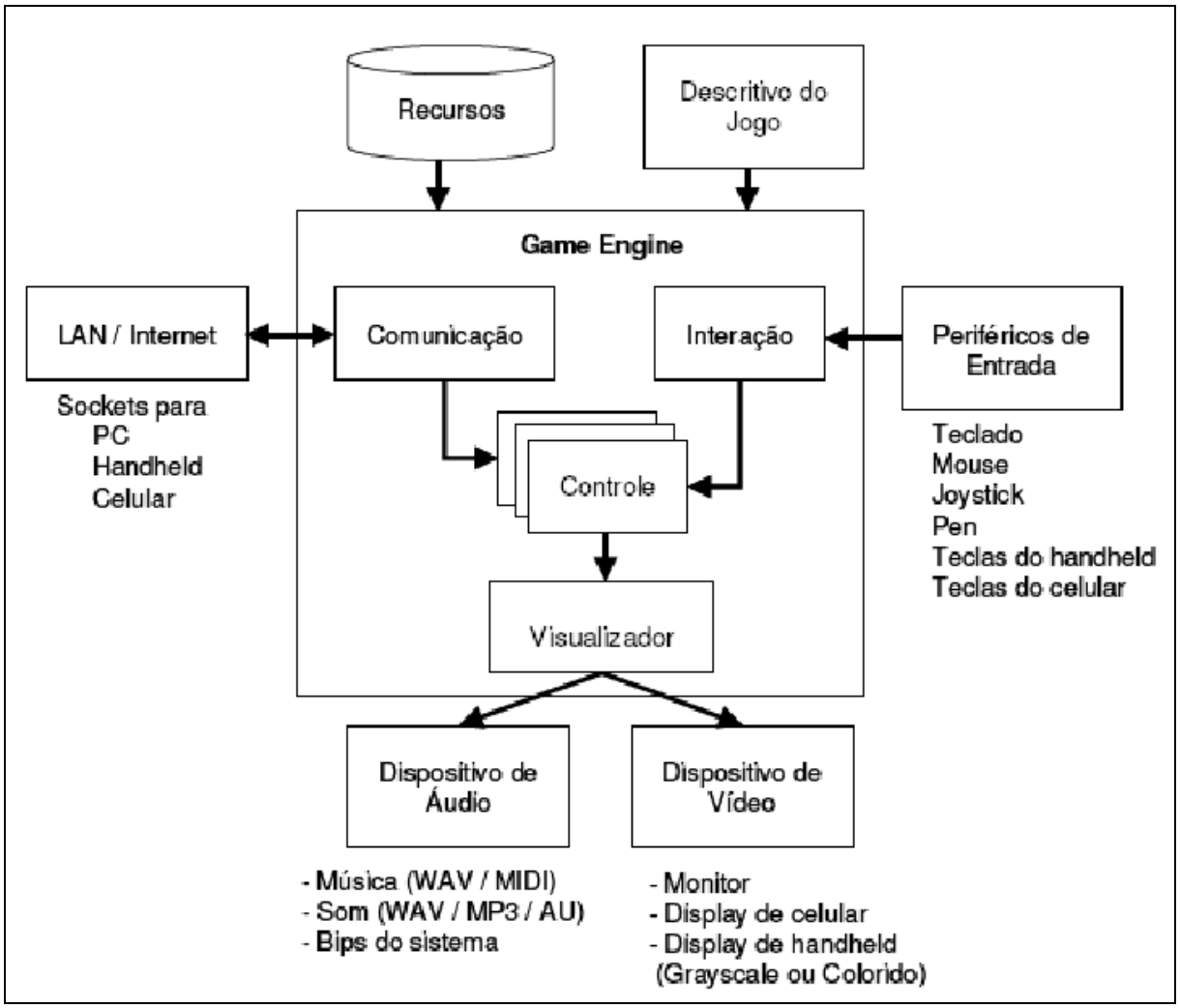

Figura 6. Estrutura genérica de um motor de jogo
Fonte: adaptado de Bittencourt e Osório (2006)

Os componentes Interação e Comunicação são responsáveis por fornecer as entradas para o motor do jogo. O componente Interação faz o tratamento de eventos gerados pelos periféricos de entrada, como teclado, *mouse*, *joystick*, quando ocorre alguma interação entre o usuário e jogo. E o componente Comunicação é responsável pela comunicação entre jogadores da rede, quando permitido este modo de jogo (BITTENCOURT; OSÓRIO, 2006).

Os componentes de Controle são responsáveis por manter a lógica do jogo e manipulação dos objetos do jogo. Cada componente de Controle é responsável por algum aspecto do jogo, por exemplo, a simulação física dos objetos e a inteligência artificial do jogo. (BITTENCOURT; OSÓRIO, 2006)

O Visualizador cria a imagem que representa o estado atual do jogo e apresenta esta imagem a algum dispositivo de saída, como o monitor de um computador ou *display* de um telefone celular (BITTENCOURT; OSÓRIO, 2006).

3.2 SUBMOTOR GRÁFICO

O submotor gráfico é o modulo responsável por mostrar o resultado final do processamento da lógica do jogo para jogador, como define LaMothe (2003, p. 495, tradução nossa), “é o *software* que processa as estruturas de dados do mundo 3D, incluindo todas as luzes, ações, e estado geral das informações, e renderiza o mundo do ponto de vista do jogador ou câmera”.

Um motor gráfico geralmente utiliza uma API gráfica, como OpenGL ou Direct3D, na implementação de seu renderizador (*renderer*). Estas APIs gráficas trabalham diretamente com o *hardware* de aceleração gráfica (GPU – *Graphics Processor Unit*), fornecem funções para desenhar corretamente os objetos. Quando não se tem um *hardware* de aceleração gráfica ou uma API gráfica disponível o motor tem deve ser implementado para utilizar a CPU (*Computer Processor Unit*) para desenhar os objetos (EBERLY, 2005, tradução nossa).

O renderizador faz a apresentação em 2D de uma cena 3D, dois aspectos são essenciais para isto, a visualização e a projeção. A visualização se refere à posição e orientação do visualizador no espaço 3D, a partir desta posição, pode se ter o *viewport* , um retângulo que representa a projeção de uma cena. A projeção se refere a ação de projetar os dados de um espaço 3D para um espaço 2D (ZERBST; DÜVEL, 2004, tradução nossa).

3.3 SUBMOTOR DE INTELIGÊNCIA ARTIFICAL

O submotor de inteligência artificial fornece funcionalidades relacionadas à aplicação de técnicas da Inteligência Artiticial (IA) que podem ser utilizadas para diversos fins em jogos digitais. Como exemplos, inteligência do computador em jogos de tabuleiro; controle de personagens não-jogaveis (NPC – *Non-player Character*); busca de caminhos no mundo virtual (*path finding*); entre outros (BITTENCOURT; OSÓRIO, 2006).

A IA aplicada a jogos digitais é chamada de *Game IA*, este termo foi adotado para diferenciar a IA acadêmica e a IA para jogos. A principal diferença entre estes dois termos é o objetivo que cada um busca, no primeiro, os pesquisadores buscam soluções para problemas extremamente difíceis, procuram saber como tal inteligência age para entender melhor seu funcionamento (KISHIMOTO, 2004).

Já na IA para jogos, os desenvolvedores estão interessados principalmente na sua engenharia, como construir os algoritmos e aplicá-los em seu jogo, se baseiam em pesquisas acadêmicas, mas não buscam entender profundamente como tudo ocorre (MILLINGTON, 2006, tradução nossa).

O objetivo dos motores de IA não é simplesmente encontrar uma solução para uma determinada situação, e sim, decidir qual é a melhor solução entre muitas outras. Para isso emprega o uso de varias técnicas da IA, para serem usadas no jogo de acordo com o objetivo e situação do problema. É importante ter conhecimento de que nem todas as técnicas de IA são apropriadas para uso em jogos digitais, devido seu alto custo computacional, às vezes se mostrando ineficiente para execução em tempo real. Como exemplo, Algoritmos Genéticos e Redes Neurais, que não são muito utilizadas em jogos digitais (SCHWAB, 2004, tradução nossa).

Millington (2006, tradução nossa) define os elementos básicos de um motor de IA como: estratégia; tomada de decisão; e movimentação; como mostrado no modelo da Figura 7. Nem todos os jogos necessitam destes três elementos, alguns podem usar apenas o nível de estratégia, outros apenas o nível de tomada de decisão, isto depende do gênero de jogo e os comportamentos necessários pelos elementos controlados pelo computador.

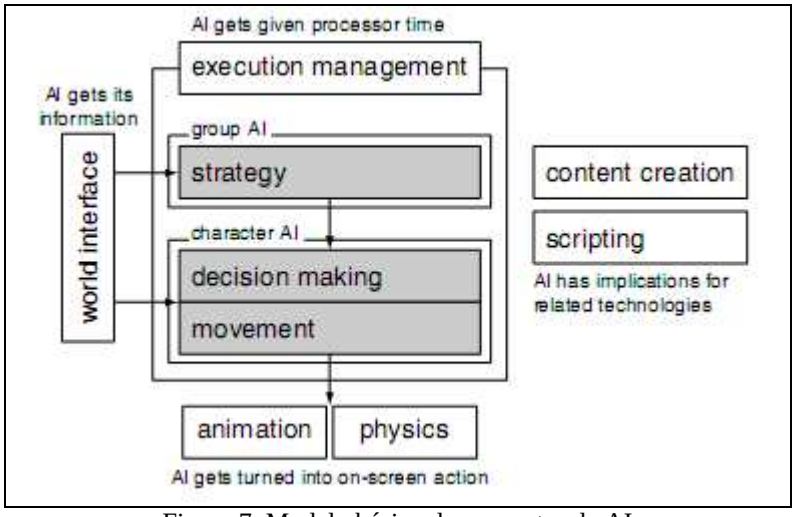

Figura 7. Modelo básico de um motor de AI
Fonte: adaptado de Millington (2006)

A tomada de decisão envolve os objetivos de um personagem não-jogavel e a decisão de suas ações futuras. Geralmente, cada personagem tem um conjunto de ações possíveis de realizar (atacar, esconder-se, explorar, patrulhar, etc.), o nível de tomada de decisão deve planejar as ações mais adequadas em cada momento do jogo de acordo com dados do ambiente virtual em que se encontra (MILLINGTON, 2006, tradução nossa).

A movimentação se refere aos algoritmos que transformam as decisões e algum tipo de movimento. Algumas tomadas de decisão necessitam do nível de movimentação para realizar suas ações, por exemplo, quando o sistema de tomada de decisão determina que um personagem deva atacar o jogador, o nível de movimentação deverá levá-lo até o jogador para que possa realizar sua ação (MILLINGTON, 2006, tradução nossa).

Já o nível de estratégia é responsável pela estratégia global, quando necessária no jogo, usada por um grupo de personagens. Cada personagem tem suas tomadas de decisão e movimentação, porém suas tomadas de decisões são influenciadas por um grupo de estratégia (MILLINGTON, 2006, tradução nossa).

Além destes componentes que se referem propriamente a IA do jogo, o motor de IA necessita de um conjunto de infra-estruturas adicionais. Um sistema de interface com o mundo, que faz a comunicação entre o ambiente de jogo e o motor de IA, pois ele necessita de informações do jogo para tomar suas decisões; Uma interface para movimentação, onde o motor de IA faz requisições de ações ao jogo, que pode utilizar animações ou simulações físicas; Um sistema de gerenciamento de execução, para gerir a forma correta do motor de IA utilizar os recursos de processamento e memória (MILLINGTON, 2006, tradução nossa).

3.4 SUBMOTOR DE FÍSICA

O submotor de física, também conhecido como motor de física ou *physics engine* é um componente de *software* responsável pela simulação dos comportamentos físicos de um jogo considerando variáveis como massa, atrito, velocidade, forças e volume para definir novas posições, velocidades e transformações dos diversos objetos de uma cena (GOMES, 2007; MILLINGTON, 2007; WOULFE, 2005). Millington (2007, p. 3, tradução nossa) define o motor de física como “basicamente uma grande calculadora, ele faz a matemática para simular a física. Mas ele não sabe o que precisa ser simulado. Além do motor, precisamos de dados específicos que representam o ambiente do jogo”. A definição aborda também a generalização das funcionalidades dos motores e sua independência quanto a um determinado jogo.

O submotor de física é um dos módulos mais importantes de um motor de jogo, porque ele está diretamente relacionado à interatividade do jogo, que é uma das principais características que atraem e prendem o jogador ao jogo, portanto tem de ser muito bem estudado e implementado, pois cada erro ou inconsistência encontrado no jogo pode torná-lo menos atrativo.

Os motores de física aplicam em sua implementação as leis da física, de forma matematicamente exata ou aproximada, com o objetivo aumentar o realismo dos mesmos sendo responsável por todo o comportamento físico dos elementos do jogo. Por meio da simulação cinemática pode-se reproduzir a movimentação de carros, aviões ou outros objetos; por meio da simulação da dinâmica podem-se reproduzir efeitos como gravidade, atrito, aceleração, desaceleração, colisão; a simulação da aplicação de forças em corpos rígidos ou deformáveis e também as reações as colisões entre os corpos; a simulação de efeitos naturais como água, fogo, vento e explosões (BITTENCOURT; OSÓRIO, 2006).

Embora o objetivo do motor de física seja, através das leis da física, aumentar o realismo dos jogos digitais, o objetivo real não é copiar exatamente os comportamentos do mundo real para o mundo virtual do jogo, o real objetivo é uma simulação consistente da física, sem erros que possam desapontar os jogadores, como justifica Hecker (2000, p. 35, tradução nossa):

> Realidade é chata, escapar da realidade é o motivo, em primeiro lugar, das pessoas jogarem jogos de computador. Consistência, por outro lado, é crucial para manter o jogador imerso no ambiente do jogo. Ele pode ser capaz de correr 100 milhas por hora e saltar 20 pés – enquanto o resto do mundo reage normalmente. Na verdade, cada jogo de computador faz um contrato com seus jogadores quando começam a jogar; qualquer inconsistência viola este contrato, lembrando o jogador de que ‘isto é só um jogo

3.4.1 Simulação Física em Jogos

O objetivo da simulação física em jogos digitais é tornar os efeitos mais reais para o observador aumentando o nível de imersão e de credibilidade das ações e objetos fundamentais na experiência do jogar. O uso de mecanismos de física visa simular o comportamento de objetos no mundo real aplicando realismo na forma como os objetos se movimentam, interagem e reagem ao ambiente que os cerca. Sem o uso da física, a maior parte das ações em jogos se limitaria a animações pré-fabricadas, acionadas por eventos do próprio jogo.

Porém, no mundo virtual dos jogos a percepção do jogador deve ser adaptada, visto a complexidade do mundo real e a dificuldade de imitar perfeitamente a totalidade das interações entre jogador e cenário de um jogo. Normalmente, a simulação física é apenas uma aproximação estreita com a física real sendo que a computação é realizada utilizando os valores discretos. A simulação de física pode ser utilizada, entre outras aplicações, para (PALMER, 2005):

a) prover realismo nas ações e objetos;
b) modelar a trajetória de objetos (de um projétil, de uma bola, etc.);
c) calcular a força de um objeto (de uma bola caindo, de um carro colidindo, etc.);
d) calcular a posição dos objetos no jogo;
e) detectar colisões de objetos em uma cena do jogo;
f) simular comportamento e interação entre objetos.

A maioria dessas simulações consome muitos recursos de processamento requerendo cálculos bastante detalhados em função dos altos níveis de precisão exigidos nos resultados. Dependendo da configuração de *hardware* e das tecnologias de implementação

utilizadas, este processamento pode se dar em processadores como CPU[7] (*Central Processing Unit*), GPU[8] (*Graphics Processing Unit*) e PPU[9] (*Physics Processing Unit*), ou com o uso de aceleradores gráficos 3D[10]. As implementações ou motores de física têm buscado soluções diferenciadas para tratar do problema de desempenho nas simulações. Por exemplo, o motor Havok executa a simulação em um número muito limitado de ciclos de CPU.

De maneira geral, a física nos jogos digitais é tratada e aplicada considerando-se as seguintes áreas (GOMES, 2007):

a) dinâmica de corpos rígidos e/ou flexíveis (*rigid/soft body dynamics*);

b) detecção de colisão (*collision detection*);

c) dinâmica de fluidos (*fluid dynamics*);

d) simulação de comportamento de objetos especiais (personagens, veículos, cabelo, roupas, fragmentação de objetos, etc.).

3.4.2 Dinâmica de corpos rígidos e flexíveis

Quanto à abordagem de desenvolvimento categorizada pelo tipo de objeto, os motores podem simular corpos rígidos ou corpos de massa agregada ou flexíveis. Um corpo rígido é um objeto sólido que não sofre deformação, ou seja, sua forma e dimensões permanecem constantes durante toda a simulação física, independente das forças que são aplicadas ao corpo. Nos motores de corpos rígidos, tanto no movimento quanto na rotação, os objetos são tratados com um todo (MILLINGTON, 2007, tradução nossa).

---

[7] A Unidade Central de Processamento tem como função principal unificar todo o sistema, controlar as funções realizadas por cada unidade funcional, e é também responsável pela execução de todos os programas do sistema, que deverão estar armazenados na memória principal.
[8] A Unidade Gráfica de processamento é um microprocessador especializado em processar gráficos.
[9] A Unidade Física de processamento é um processador especializado em realizar cálculos de física complexa dos jogos, especialmente para os que usam cenários em 3 dimensões.
[10] Hardware ou software com a função de processamento gráfico.

Gomes (2007, p. 9) indica que um corpo rígido é caracterizado pelos elementos: massa, posição do centro de massa, orientação, velocidade linear, velocidade angular e volume.

Grande parte da dificuldade em implementar um motor de física para corpos rígidos é a simulação de contato, ou seja, os locais onde dois objetos se tocam ou se conectam. Existem algumas abordagens que para trabalhar com a simulação de contato, uma delas é a abordagem iterativa (*iterative approach*), suas vantagens são a fácil implementação e a velocidade de resolução, pois ela trata uma colisão por vez. Porém, como desvantagem de se tratar cada contato isoladamente é que cada resolução de contato pode interferir em outra. Uma opção mais realista para a resolução de contatos é a abordagem baseada em Jacobian (*Jacobian-based),* porém mais custosa ao processador, pois trata de todos os contatos ao mesmo tempo. Requer uma matemática mais complexa, uma única resolução de contato pode envolver milhões de cálculos e nem sempre tem uma resposta valida (MILLINGTON, 2007, tradução nossa).

Já a simulação abordando massa agregada ou corpo flexível trata os objetos como lotes de pequenas massas. Cada unidade de massa está localizada em um ponto único. Por meio de equações de movimento linear pode ser feita a movimentação e rotação dos objetos (MILLINGTON, 2007, tradução nossa). Um corpo flexível é um sólido mole que sofre deformações, alteração no formato e volume, devido à ação de forças externas. Dessa forma, a simulação de física de corpos flexíveis é mais trabalhosa que a simulação de corpos rígidos, porém é uma característica desejável, uma vez que é usada para simular a física de cabelos, roupas, bandeiras e objetos secundários agregados ao principal, emprestando mais realismo às aplicações.

Gomes (2007, tradução nossa) aponta três aspectos importantes que devem ser considerados na simulação de corpos flexíveis:

a) detecção de colisões entre objetos deformantes;

b) cálculo das forças de impacto quando os corpos colidem;

c) identificação das forças de deformação ou deformação de contato dos corpos.

3.4.3 Dinâmica de fluidos

A simulação física da dinâmica de fluídos visa modelar o comportamento de substâncias líquidas, gasosas ou plasmas que não possuem formato definido, e se deformam indefinidamente quando submetido a diversas forças (como água, o ar, o sangue e o vidro). Simulação de fluidos consome bastante processamento e apresenta bastantes erros, tornando seu uso bastante dificultado, apesar de ser um diferencial em jogos (CRUZ e SANTOS, 2009).

3.4.4 Simulação de comportamento

Objetos especiais, e mais complexos, como personagens, veículos, cabelo, roupas, fragmentação de objetos, necessitam de uma simulação física mais apurada, que leve em consideração elementos específicos de sua composição, por exemplo, o torque do motor de um carro, a movimentação de uma pessoa, destruição de objetos, etc.

Várias técnicas podem ser utilizadas, entre elas a *ragdoll* a qual utiliza uma coleção de corpos rígidos ligados, respeitando limitações que definem a forma como cada objeto pode se mover em relação a outro (por exemplo, os ossos na locomoção). Uma técnica para simular a destruição de um objeto é substituir estes objetos por outros objetos menores que juntos têm a mesma forma e volume que o original (GOMES, 2007).

3.4.5 Detecção de colisão

Uma das principais funcionalidades dos motores de física para jogos é a detecção de colisão. Esta funcionalidade é responsável por verificar, durante o andamento do jogo, quais dos objetos do mundo, ou de um determinado conjunto de objetos, estão colidindo. Sua outra função é gerar a lista de pontos de contatos entre os objetos que estão em colisão. Estas informações sobre posicionamento e colisão podem ser utilizadas posteriormente pela simulação dinâmica dos corpos ou na lógica do jogo (MILLINGTON, 2007, tradução nossa).

Geralmente, a detecção de colisão leva em consideração apenas a forma geométrica do objeto, seu tamanho e sua posição, deixando de lado seus detalhes relacionados à simulação dinâmica que caracteriza seu comportamento, como forças, velocidade, direção, etc. Porém, alguns sistemas de detecção de colisão podem considerar também o comportamento atual do objeto para tentar prever colisões futuras (MILLINGTON, 2007, tradução nossa).

A detecção de colisão entre os objetos de um mundo é feita de par em par. A medida que são adicionados novos objetos ao mundo, a execução dos algoritmos que verificam a intersecção geométrica passa a consumir maior tempo de processamento, sendo que a cada inserção, o número de pares de objetos se multiplica (complexidade $O(n^2)$). Como geralmente os ambientes dos jogos são grandes contendo muitos objetos, o sistema de detecção de colisões, para verificar todos os pares de objetos, pode reduzir o desempenho de execução do jogo (MILLINGTON, 2007, tradução nossa; ERICSSON, 2005, tradução nossa).

Uma solução para diminuir este custo é dividir o processo de detecção de colisão em duas etapas. Na primeira etapa, chamada de “detecção de colisão grosseira” ou *broadphase*, são selecionados os pares de objetos que provavelmente estão em contato, mas sem se preocupar se realmente estão em contato, por isso pode-se ser usado um algoritmo

menos custoso computacionalmente, o objetivo desta etapa é eliminar, de forma rápida, grande quantidade dos pares que não estão colidindo e gerar uma lista de possíveis colisões (ERICSSON, 2005, tradução nossa).

A segunda etapa, chamada de “detecção de colisão fina” ou *narrowphase*, é que efetivamente realiza a detecção de colisão, são aplicados os algoritmos de detecção de colisão em cada par selecionado na etapa anterior, assim identificando quais objetos realmente estão colidindo e gerando a lista de contatos entre os objetos que estão colidindo (ERICSSON, 2005, tradução nossa).

Porém, nem todos os motores tratam estas etapas separadamente. O processo de listagem e verificação de colisão pode ser realizado de forma unificada, ou seja, à medida que encontra os pares de objetos que estão possivelmente colidindo (*broadphase*) já são realizadas as verificações da *narrowphase* e gerada uma lista com pares de objetos que realmente estão colidindo.

As principais características da *broadphase* são sua legitimidade e seu tamanho. Todos os pares de objetos devem ser verificados, destes podem ser selecionados pares que não estejam realmente colidindo (falsos positivos), porém não deve deixar de selecionar pares que estão colidindo (falsos negativos), ou seja, deve garantir que todos os pares que estejam realmente colidindo sejam selecionados para a *narrowphase* (MILLINGTON, 2007, tradução nossa).

A lista de pares selecionados pela *broadphase* deve ser menor quanto possível sem tornar seu custo computacional inviável para a aplicação. A menor lista possível é a que contém apenas os pares que estão realmente colidindo, porém, na prática são gerados muitos falsos positivos (MILLINGTON, 2007, tradução nossa).

As abordagens mais utilizadas para a seleção dos pares de objetos na *broadphase* são volumes limitantes (BV - *Bounding Volume*) e estruturas de dados espaciais.

3.4.5.1 Bounding Volumes

Um BV é simplesmente uma área de espaço que encapsula um ou mais objetos de geometria mais complexa. A idéia desta abordagem é utilizar geometrias simples, como caixas ou esferas, que têm algoritmos de sobreposição menos custosos que abordar objetos de geometrias mais complexas, permitindo assim, na *broadphase*, rejeição rápida de pares que não estão colidindo (MILLINGTON, 2007, tradução nossa; ERICSSON, 2005, tradução nossa).

Um exemplo básico desta idéia pode ser visto na Figura 8. Os elementos A, B, C e D representam os BVs de seus objetos contidos. O par de A e B não estão sobrepostos, logo pode ser rejeitado pela *broadphase*, pois não há possibilidade de seus objetos estarem colidindo. Já o par de C e D estão sobrepostos, então os objetos contidos neles podem estar colidindo. O par é passado para *narrowphase* para verificação com suas formas geométricas reais, no caso se trata de um falso positivo, pois, como podemos ver, seus BVs estão sobrepondo, mas os objetos contidos neles não (ERICSSON, 2005, tradução nossa).

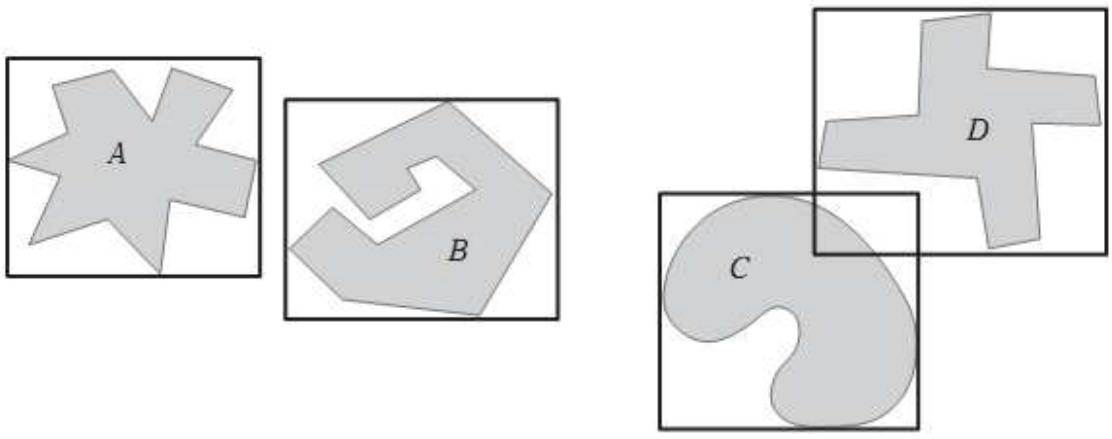

Figura 8. Bounding Volumes
Fonte: Ericsson (2005)

O exemplo da Figura 8 utiliza caixas limitantes alinhadas aos eixos (AABB - *Axis Aligned Bounding Box*) como BV, o que geralmente resulta em grandes espaços vazios dentro da BV, causando grande quantidade de falsos positivos. Para diminuir os falsos positivos na *broadphase* podem ser usadas outras orientações e formas geométricas para os BVs, de forma que diminua o espaço vazio dentro da BV. Na Figura 9 são apresentados os BVs geralmente utilizados, da esquerda para direita aumentando o custo computacional, mas fazendo uma melhor filtragem; e da direita para esquerda aumentando a rapidez dos testes, porém gerando maior número de falsos positivos (ERICSSON, 2005, tradução nossa).

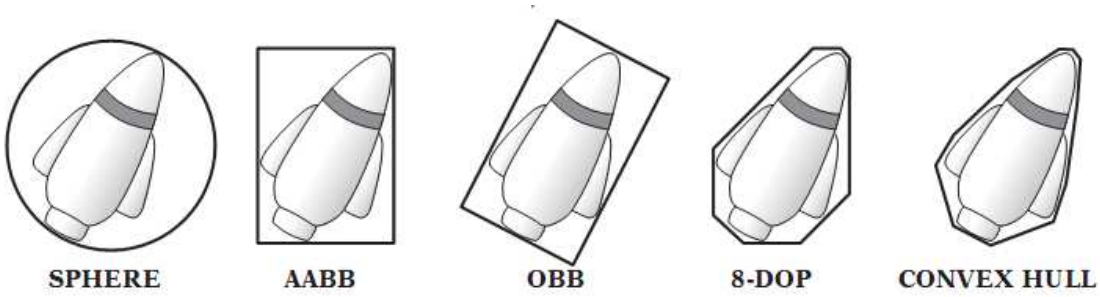

Figura 9. Formas geométricas utilizadas como BV
Fonte: Ericsson (2005)

O encapsulamento AABB é o mais comuns em aplicações de detecção de colisão. A geometria do AABB tem a forma de uma caixa retangular de seis lados (3D), sendo que todas as faces estão em paralelo com os eixos do sistema de coordenadas, ou seja, independente da forma geométrica ou rotação do objeto de colisão. A orientação espacial da AABB estará sempre na mesma rotação alinhada com os eixos do sistema de coordenada, isso pode ocasionar grandes áreas vazias dentro dela, mas seus testes de sobreposição são muito rápidos, pois envolve apenas a comparação dos valores de coordenadas individuais (ERICSSON, 2005, tradução nossa).

A esfera também é um BV bastante comum e seus testes de sobreposição também são bem rápidos. Tem a forma de uma esfera, sua vantagem é que é rotacionalmente

invariável, o que significa que sua transformação é trivial, simplesmente deve ser atualizada para a nova posição do objeto de colisão. O cálculo da melhor esfera para um objeto não é tão simples quanto a AABB, porém, em termos de memória utilizada, é a mais eficiente (ERICSSON, 2005, tradução nossa).

Caixas limitantes orientadas (OBB - *Oriented Bounding Boxes*) são blocos retangulares, como as AABBs, porém são orientadas dependendo da forma geométrica e orientação do objeto de colisão a fim de ter menos espaço vazio do que uma AABB. A quantidade de memória necessária para armazenamento de uma OBB é maior e seu teste de sobreposição é mais complexo, porém é possível aperfeiçoar para se tornar eficiente para a *broadphase* (ERICSSON, 2005, tradução nossa).

Além destas, podem ser usadas outras formas geométricas como BV, como 8-DOPs (*Eight-Direction Discrete Orientation Polytope*) e cascos convexos, que podem mostrar melhores resultados na filtragem dos pares, mas nãos são muito utilizadas pois suas estruturas consomem muita memória e processamento, nem sempre trazendo melhoria no desempenho ao processo de detecção de colisão (ERICSSON, 2005, tradução nossa).

A utilização de BVs na *broadphase* evita boa parte das verificações de detecção de colisão desnecessárias na *narrowphase*, para isso é feito a verificação de detecção de colisão de todos os pares de BVs do mundo. É possível aperfeiçoar ainda mais o processo da *broadphase* organizando os BVs em hierarquias, a fim de diminuir grande parte destas verificações, esta abordagem é chamada de “hierarquia de volumes limitantes” (BVH - *Bounding Volume Hierarchy*) (MILLINGTON, 2007, tradução nossa).

BVHs utilizam árvores como estrutura de dados para organizar os BVs. Cada objeto de colisão é mantido em um BV, cada BV é ligado com algum nó pai, sendo que para representar esta ligação é utilizado um BV que engloba todos os BVs filhos. Todos os nós são

organizados desta forma até que um BV de nó pai englobe todos os objetos de colisão do mundo, como mostrado na Figura 10 (MILLINGTON, 2007, tradução nossa).

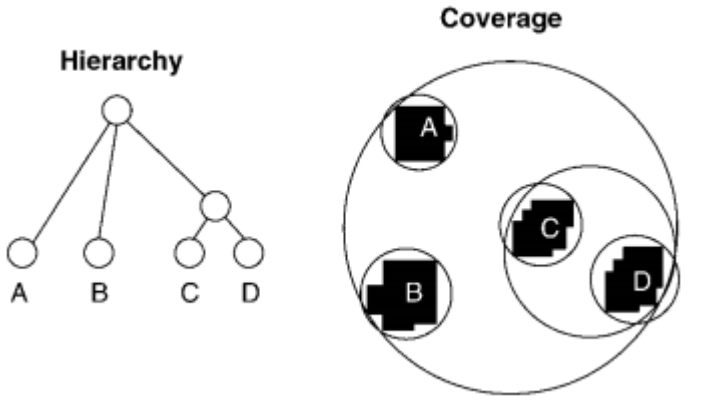

Figura 10. BVH utilizando esferas
Fonte: Millington (2007)

Varias abordagens de implementações de BVH são encontradas na literatura, diferindo quanto à forma geométrica dos BVs; o tipo de árvore utilizada para organizas os BVs; as regras para construção e manutenção da estrutura de dados; e a busca e verificação de sobreposição dos BVs. De maneira geral, a implementação de um BVH deve organizar os BVs em hierarquias, de forma que, a não sobreposição de duas BVs de um mesmo nível garanta que nenhum de seus BVs descendentes estejam colidindo.

3.4.5.2 Narrowphase

Finalizado o processo da *broadphase*, ou durante sua execução, a *narrowphase* verifica a colisão entre os pares de objetos de colisão selecionados, retornando seus pontos de contato. Alguns motores retornam apenas a profundidade máxima de interpenetração entre os dois objetos, porém este não é o ideal, pode ser satisfatório para alguns tipos de objetos (esfera e superfície plana), porém para objetos de geometria mais complexa (automóvel e

superfície plana) é necessário mais do que apenas um ponto de contato para representar a colisão (MILLINGTON, 2007, tradução nossa).

A geração da lista de contatos é mais complexa e custosa computacionalmente do que a simples verificação da colisão entre os dois objetos. O processo da *narrwophase* pode ser dividido em duas etapas, na primeira é feita uma detecção de colisão fina entre os dois objetos para saber se estão realmente colidindo, caso sim, na segunda etapa é gerado a lista de contatos entre os objetos. Como a geração de todos os contatos pode ser muito custosa computacionalmente, alguns motores determinam, ou permitem o desenvolvedor determinar, um número máximo de pontos de contatos a serem gerados em cada par de objetos (MILLINGTON, 2007, tradução nossa).

Os sistemas de detecção de colisão utilizam diferentes algoritmos de colisão dependendo da formas geométricas dos objetos envolvidos nos testes, possibilitando otimização para cada combinação de par de objetos.

3.4.5.3 Formas Geométricas

Os sistemas de detecção de colisão não utilizam os mesmo objetos criados para a renderização visual, pois estes, geralmente têm geometria muito detalhista, o que tornaria o teste de colisão e geração de pontos de contato muito custosos computacionalmente para serem executados em tempo real. A fim de diminuir estes custos, assim possibilitando execução em tempo real, os motores de física utilizam objetos de geometria simplificada, criados especialmente para a detecção de colisão, ainda assim, resultando em uma precisão de detecção de colisão satisfatória para visualização final (MILLINGTON, 2007, tradução nossa).

Estes objetos podem ser representados utilizando diferentes técnicas da computação gráfica, como por exemplo, malhas de triângulo; objetos primitivos ou uma combinação deles. Sendo que a técnica utilizada pode influenciar no desempenho da aplicação e na precisão da detecção de colisão (EBERLE, 2004, tradução nossa).

Objetos primitivos são formas geométricas simples, pré-definidas pelos motores de física, que geralmente têm algoritmos de colisão simples e rápidos. Os objetos primitivos geralmente disponibilizados pelos motores de física são: esfera, caixa, superfície plana, cilindro e cone. Podem ser utilizados para representar objetos de mesma forma geométrica ou aproximada, também podendo ser combinados com outros objetos primitivos, a fim de representar formas geométricas mais complexas, como visto na Figura 11, uma combinação de cones e esferas (EBERLE, 2004, tradução nossa).

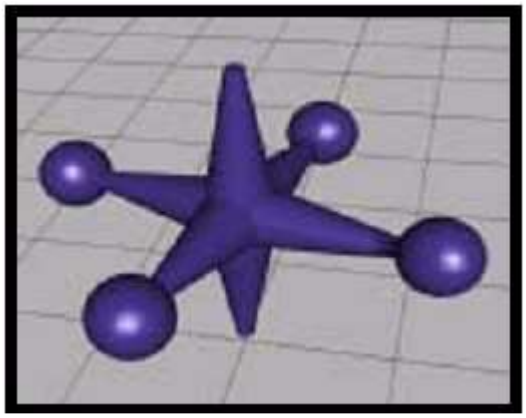

Figura 11. Forma geométrica composta
Fonte: Eberle (2004)

O uso destas formas primitivas pode causar imperfeições no resultado final em tela, devido à simplificação geométrica, por isso além das formas primitivas, geralmente os motores possibilitam a criação de malhas de triângulos.

3.5 REUSO, EXPANSIBILIDADE E INTERFACEAMENTO NOS MOTORES DE FÍSICA

Millington (2007, tradução nossa) aponta características (reusabilidade) e abordagens (tipo de objeto, resolução de contato, força e impulso) para o desenvolvimento de motores de física. Para o autor, é essencial para a reusabilidade que um motor de física seja genérico a ponto de poder simular um mesmo comportamento para mais de um elemento com propriedades físicas diferentes, ou seja, um simulador de balística deve ser capaz de simular tanto uma flecha, quanto um bala. A implementação do simulador deve ter apenas as propriedades que os elementos têm em comum, as propriedades físicas especificas de cada elemento deve ser fornecidas pelo jogo.

Para Gomes (2007), o caráter genérico dos motores de física possibilita o reuso requerendo, na maioria dos casos do desenvolvimento do jogo, apenas a integração do motor escolhido na aplicação. Fatores que precisam ser considerados no reuso de motores são: customização de funcionalidades, interoperabilidade de dados sobre objetos do jogo entre motores e aplicações, uso unificado de diferentes motores de física para o desenvolvimento de um jogo, troca do motor utilizado (seja por mudança de versão ou por substituição).

Esta solução, porém, é limitada em termos de expansibilidade da aplicação, visto a utilização específica e dependente de recursos de interfaceamento (como por exemplo, API de programação do motor) do motor físico selecionado. Isto dificulta a mudança de motor físico e também a troca dos dados de cenas e objetos entre diferentes aplicativos e demais motores de jogos (IA, gráfico, etc.).

Os problemas de dependência na expansibilidade, que impossibilita o uso concomitante de mais de um motor, assim como as dificuldades na troca e carregamento de informações dos objetos da cena física podem ser resolvidos com a implementação de uma camada de abstração de física intermediária entre a aplicação e os diversos motores de física

existentes. A camada de abstração de *software* permite o uso de diferentes motores físicos através de um conjunto único de interfaces (API). Deste modo, é possível trocar de motor de física sem alterar o código fonte, ou até mesmo utilizar mais de um motor de física na mesma aplicação de forma mais fácil e prática.

Para o desenvolvimento desta camada de abstração física podem ser utilizadas as abordagens de *wrappers* ou de arquivos físicos intercambiáveis, representado pelo padrão COLLADA Physics[11] (CRUZ e SANTOS, 2009).

Os sistemas de abstração por *wrappers* atuam de acordo com o padrão de projeto (estrutural) modificando o comportamento de uma classe (alterações na interface, extensão do comportamento ou restrição de acesso). Os *wrappers* são geralmente uma fina camada de abstração de *software* que fica sobre a classe encapsulada a ser modificada realizando a intermediação entre duas classes incompatíveis, traduzindo as chamadas entre as interfaces. Isto permite que dois trechos de programa que não tenham sido projetados ou escritos juntos, e então sejam levemente incompatíveis, possam ser utilizados em conjunto em qualquer situação.

Os *wrappers* podem ser: adaptadores (*adapters*), decoradores (*decorators*) e *proxies*. Os adaptadores alteram a interface de uma classe sem modificar as suas funcionalidades básicas. Um *wrapper* decorador estende a classe adicionando funcionalidades mantendo a mesma interface fazendo com que os objetos da classe resultante se comportem exatamente como os originais, mas também façam alguma coisa extra. O *proxy* é um *wrapper* que possui a mesma interface e a mesma funcionalidade da classe que ele encapsula, porém pode ser utilizado para controlar o acesso a outros objetos principalmente quando estes objetos devem ser acessados de uma maneira estilizada. Os *wrappers* mais conhecidos para aplicação em motores de física para jogos digitais são o GangstaWrapper, PAL e OPAL.

[11] COLLADA physics é um subconjunto de elementos de física do COLLADA (COLLAborative Design Activity): padrão, baseado em XML, de exportação e importação de arquivos interativos 3D criado pela Sony que suporta modelos detalhados, animações e iluminação.

O PAL[12] (*Physics Abstraction Layer* ou Camada de Abstração de Física) fornece uma interface unificada para um conjunto de motores de física, dentre eles Bullet, Havok, Bullet, ODE, Newton e PhysX, possibilitando o uso de múltiplos motores no desenvolvimento de uma aplicação. Esta solução de abstração oferece um *plug-in* extensível de arquitetura para o sistema de física. Possui também a funcionalidade estendida de componentes para simulação comum tais como simulação de diferentes dispositivos ou carga de configurações de física em XML (*Extensible Markup Language*) com o padrão COLLADA permite usar os mesmos dados da cena física em diversos motores de física.

A camada de abstração de física PAL possui um conjunto de oito grupos de interfaces, cada qual desenvolvida para suportar variados níveis de capacidades de simulação, que são (BOEING; BRÄUNL, 2007):

a) interfaces para o núcleo do motor de física, responsáveis pelo acesso e definições sobre a simulação física (por exemplo, valor da gravidade);
b) interfaces para sólidos, responsável pela criação dos sólidos e suas propriedades;
c) interfaces para geometrias, responsável pela criação dos formatos dos sólidos;
d) interfaces para materiais, responsável pela definição das propriedades materiais das geometrias (dureza, fricção, densidade, etc);
e) interfaces para ligações, responsável pela criação de ligações (conexões entre dois corpos, onde o movimento relativo deles é limitado);
f) interfaces para sensores, objetos que adquirem informações sobre uma simulação em andamento;
g) interfaces para motores, objetos responsáveis por aplicar forças e torque a corpos e/ou ligações;

[12] Site oficial da PAL: http://www.adrianboeing.com/pal/index.html

h) interfaces para representação de terrenos, obstáculos que limitam a movimentação de outros objetos simulados.

A camada de abstração de física aberta OPAL[13] (*Open Physics Abstraction Layer*) é uma API de código aberto para prover abstração na integração entre motores de física semelhante ao PAL, suportado apenas pelo motor ODE. A troca e armazenamento de dados são baseados em XML para objetos complexos.

3.5.1.1 Bullet Physics

Bullet Physics é um motor de física código aberto e livre para uso comercial sob licença ZLib. Escrito em C++, seu principal autor é Erwin Coumans, que trabalhou anteriormente para Havok.

Como principais características estão: detecção de colisão discreta e continua; dinâmica de corpos rígidos, flexíveis e veículos; suporte a varias formas primitivas, também a formas compostas e malhas de triângulos; suporte a COLLADA Physics; integração com Blender e Maya.

Disponível para as plataformas PLAYSTATION 3, XBox 360, Wii, PC, Linux, Mac OSX and iPhone. Seu processamento ocorre na CPU, porém com PLAYSTATION 3 tem suporte ao SPU para desenvolvedores licenciados.

Possui jogos de categoria AAA desenvolvidos pela Sony Computer Entertainment, Disney e Rockstar, porém em muitos casos esta informação não pode ser compartilhada por questões contratuais. Algumas partes do motor foram alteradas e otimizadas pela equipe da Rockstar e integrada seu motor de jogo Rage, utilizada nos jogos “Midnight Club: Los Angeles” e “Grand Theft Auto 4”.

[13] Site Oficial da OPAL: http://opal.sourceforge.net/

Bullet é um motor de física para detecção de colisão 3D e dinâmica de corpos rígidos e flexíveis. A principal tarefa do motor é realizar a detecção de colisão, resolver as colisões e *constraints* e fornecer um ambiente atualizado com a transformação de todos seus objetos.

O motor foi projetado para ser modular, como pode ser visto na Figura 12, assim possibilitando que o desenvolvedor não dependa exclusivamente do Bullet para a física do jogo. O desenvolvedor pode personalizar o motor de acordo com algumas opções pré-estabelecidas ou alterar o motor da forma que deseja, já que possui código aberto. O motor, ou parte dele, também pode ser integrado a outros motores (moto gráfico, motor de IA) no desenvolvimento de jogos ou até mesmo como parte de outros motores de física.

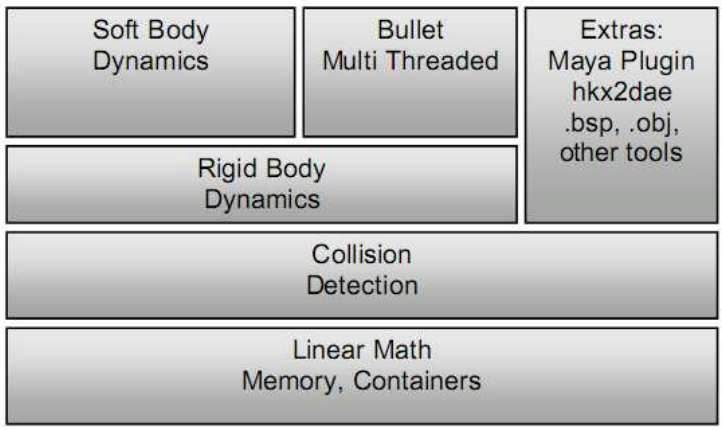

Figura 12. Módulos do Bullet

O módulo mais a baixo contém as estruturas básicas de dados (ponto flutuante, posição e rotação 3D, etc), *containers* e alocadores padrão de memória e outras funcionalidades extras. Este módulo é utilizado por todos os outros módulos do Bullet e também pela aplicação.

O modulo base de simulação é a detecção de colisão, que implementa todas as estruturas e funcionalidades relacionadas a detecção de colisão, como o mundo de colisão,

objetos de colisão, testes de colisão na *broadphase* e *narrowphase*, algoritmos de colisão, filtragem de colisão.

A partir do mundo de colisão, por meio de herança, é implementado o mundo dinâmico de corpos rígidos com detecção de colisão discreta, que então é extendido para o mundo o de corpos rígidos com detecção de colisão contínua.

O mundo de dinâmica de corpos flexíveis é implementado a partir da herança do mundo de dinâmica de corpos rígidos com detecção de colisão discreta. Cada módulo também implementa outras estruturas e funcionalidades necessárias para sua tarefa principal.

Desta forma, o Bullet possibilita que seja utilizado apenas o sistema de detecção de colisão ou o sistema de detecção de colisão e um dos sistemas de simulação dinâmica disponíveis.

A linguagem utilizada em seu desenvolvimento é o C++, baseado na programação orientada a objetos, cada módulo possui um projeto, o qual gera uma biblioteca estática para distribuição.

Sua utilização na aplicação é feita pela referencia as bibliotecas de cada módulo, ou mesmo a adição do código fonte dos módulos ao projeto. As interfaces para acesso as funcionalidades do motor são as declarações das classes abstratas (.h) utilizadas na implementação do motor. Como são utilizadas bibliotecas estáticas para a distribuição do Bullet, ao compilar a aplicação que o utiliza, a bibliotecas são inseridas no executável.

3.5.1.1.1 Detecção de Colisão

É o módulo responsável por todas as funcionalidades relativas à detecção de colisão do Bullet, como, testes de colisão, filtragem de colisão, objetos de colisão, algoritmos de colisão, entre outras.

A estrutura principal do sistema de colisão é o mundo de colisão, é ele quem armazena todos os objetos de colisão e fornece a interface para os métodos relacionados à detecção de colisão, como a verificação das colisões entre os objetos do mundo, geração dos pontos de contatos, distancia e profundidade de penetração entre um par de objetos de colisão.

Um objeto de colisão é composto de uma forma de colisão e uma transformação de mundo, a forma de colisão descreve a forma geométrica do objeto (esfera, caixa, malha de triângulos) e seu tamanho, e a transformação de mundo é a combinação da posição do objeto no mundo e sua rotação.

A verificação das colisões entre os objetos do mundo é divida em duas etapas, a *broadphase* e a *narrowphase*.

Na primeira etapa são atualizadas as AABBs dos objetos de colisão e então são verificados quais destas AABBs estão sobrepostas, de acordo com a técnica de colisão foi selecionada. A *broadphase* mantém um cachê com os pares de objetos que tem suas AABBs sobrepostas, a cada verificação de colisão, a *broadphase* atualiza este cachê.

Na segunda etapa, a *narrowphase*, são verificados quais dos pares selecionados estão realmente colidindo, nesta etapa o teste de colisão é feito com as formas de colisão de cada par do cachê de pares da *broadphase*. Ao ser detectado uma colisão entre um par de objetos de colisão, é gerado um cachê com as informações de pontos de contato entre os dois objetos, estes pontos de contatos podem ser utilizados posteriormente pelo sistema de dinâmica do motor para resolver a colisão e na lógica do jogo.

O Bullet disponibiliza três técnicas de *broadphase*, mas também pode ser implementadas outras técnicas, sendo que deve ter a interface padrão das técnicas do Bullet. As técnicas disponibilizadas pelo motor são:

a) hierarquia dinâmica de volumes limitantes: técnica baseada na abordagem de arvores de AABBs, utiliza duas arvores para controles dos corpos, uma para os

dinâmicos e outra para os estáticos ou que não estão se movendo, quando necessário, os corpos pode ser movidas de uma arvore para outra. Sua estrutura se adapta de acordo com as dimensões do mundo e de seus corpos. Indicada para uso geral;

b) *Sweep and Prune* (SAP): esta técnica armazena as dimensões das AABBs de forma ordenada em *arrays*, um para cada eixo, e então os varre verificando as sobreposições. Utiliza números inteiros de 16 *bits*, ao invés de ponto flutuante, para determinar as dimensões das AABBs, a fim de melhorar seu desempenho. Tem a limitação de ter um mundo de tamanho fixo, determinado na sua construção. É indicado para ambientes onde os corpos têm pouco ou nenhum movimento;

c) *Sweep and Prune 32bit*: otimização do método SAP utilizando inteiro de 32 *bits* para as dimensões das AABBs, a fim de obter melhor precisão na verificação das sobreposições;

Para a detecção de colisão na *narrowphase* o motor tem algoritmos específicos para cada tipo de par de formas de colisão, a fim de otimizar a detecção para cada um destes pares. Para cada par selecionado pela *broadphase* é verificado a forma de colisão dos dois objetos do par, então é executado o algoritmo apropriado para verificação. Há também a possibilidade de criar novos algoritmos para outras formas de colisão ou para substituir os algoritmos padrões.

3.5.1.1.2 Formas Geométricas

O Bullet suporta uma grande variedade de formas de colisão, sendo que é possível também adicionar outras formas.

As formas primitivas disponibilizadas pelo motor são: caixa, esfera, cápsula, cilindro, cone e casco convexo de múltiplas esferas. Para a cápsula, cilindro e cone são disponibilizados três tipos formas, uma para cada eixo, sem a necessidade de usar rotação. O casco convexo de múltiplas esferas cria uma serie de esperas de acordo com posições pré-definidas, pode ser usado pra criar cápsulas especiais, ou efeitos de deformação.

Varias formas primitivas podem ser combinadas formando uma composição de formas, neste caso, cada forma da composição é chamado de forma filha e tem seu próprio posicionamento e rotação, relativo à composição.

O Bullet também suporta a utilização de malhas de triângulos para representar seus objetos de colisão, que podem ser criadas a partir de *arrays* de vértices, vale lembrar, que o ideal é utilizar o mínimo possível de vértices numa malha de triângulos, pois afetar significativamente o desempenho da aplicação.

Diante da variedade de formas de colisão disponíveis no Bullet, é importante saber escolher as formas corretas de acordo com suas finalidades, como a escolha do algoritmo de detecção de colisão depende das formas de colisões envolvidas, escolher a forma de colisão errada pode significar perda de desempenho ou qualidade.

3.5.1.2 Havok Physics

É um motor de física comercial, porém gratuito para desenvolvimento e distribuição de jogos não comerciais para plataforma PC. Havok Physics é desenvolvido pela empresa Havok, que atualmente é da Intel Corporation. A linguagem utilizada em seu desenvolvimento é C++.

Entre as principais funcionalidades estão: detecção de colisão discreta e continua; dinâmica de corpos rígidos, flexíveis e veículos; suporte a varias primitivas, objetos

compostos e malhas de triângulos; compactação de malhas grandes; depurador visual, entre outros.

Disponível para as plataformas Wii, PLAYSTATION 3, PLAYSTATION 2, PSP, Xbox, Xbox 360, GameCube, e PC. Tem como diferencial, o uso da GPU para otimizar o processamento de parte de seus cálculos.

Havok Physics é utilizado em mais de 200 jogos comerciais, alguns dos jogos AAA já desenvolvidos são: “Alone in the Dark”, “Assassin's Creed”, “Spore”, “Half-Life 2: The Orange Box”, “Battlefield: Bad Company”. E atualmente sendo utilizado no desenvolvimento de “Diablo III”, “Starcraft II”, “Indiana Jones”, entre outros.

3.5.1.3 PhysX

O motor de física PhysX tem como principais funções a dinâmica de corpos rígidos e a detecção de colisão, mas vem acompanhado de outras funcionalidades, como: simulação de roupas, fluidos, corpos flexíveis e personagens; e pré-processamento de malhas de triângulos; distribuídas nos 5 componentes: Physics, Cooking, Foundation, Character e PhysXLoader.

O componente principal, PhysicsSDK, é composto pelos módulos: Physics, onde ocorre a simulação dinâmica de corpos rígidos e a detecção de colisão; Cloth, para a simulação de roupas; Fluids, para a simulação de fluidos; SoftBody, para a simulação de corpos flexíveis; Foundation, composto pelo tipos básicos de dados, matemática e outras utilidade utilizadas pelos outros módulos; e PhysXLoader, usado para carregar a versão apropriada do motor para a aplicação.

O motor é desenvolvido na linguagem C++, como uma hierarquia de classes. Sua interface é também desenvolvida em C++, são as classes abstratas (.h) das classes utilizadas

na implementação do motor, porém os objetos não podem ser instanciados da forma tradicional (utilizado o operador *new* para invocar o construtor da classe), as interfaces fornecem métodos que instanciam outros tipos de objetos e então retornam a aplicação.

O PhysX é distribuído em bibliotecas dinâmicas, sendo que os componentes principais (PhysXCore e NxCooking) não devem ser distribuídos junto a aplicação, pois são fornecidos por meio de um instalador disponibilizado pela fabricante, nVidia, com a versão atual e versões anteriores. Por meio do PhysXLoader é especificado qual versão do PhysX será utilizado e este se responsabiliza de carregar as DLLs apropriadas. Assim tem-se todas as versões do PhysX disponíveis para uso, tornando mais fácil alterar as versões do PhysX na aplicação.

Para utilizar o PhysX em uma aplicação basta referenciar as bibliotecas estáticas necessárias e utilizar a as interfaces para acessar as funcionalidades.

3.5.1.3.1 Detecção de Colisão

O sistema de detecção de colisão do PhysX foi implementado junto ao sistema de dinâmica de corpos, no módulo *Physics*, é responsável pela detecção de colisão (discreta e contínua), geração de pontos de contato, filtragem de contatos e *triggers*.

O mundo de colisão, no PhysX chamado de cena, abriga todos os elementos da simulação física, também fornecendo uma interface para a execução da simulação, configuração do mundo e consultas de colisão.

Os elementos do mundo são chamados de atores, que são compostos por uma forma geométrica, com as propriedades para colisão, e um corpo com as propriedades para dinâmica. Caso seja necessário um objeto apenas para colisão, deve se criar um ator, sem um corpo, assim o ator tem comportamento de um objeto estático.

A detecção de colisão é dividida em duas etapas, *broadphase* e *narrowphase*, e no caso de trabalhar com malhas de triangulo, uma etapa entre as duas, a *midphase*.

Na *broadphase* ocorre atualização das BVs dos objetos de colisão e a verificação de colisão entre as BVs, Na *narrowphase* é executada a verificação de contatos dos pares selecionados pela *broadphase*, nesta etapa, pode-se acessar os pontos de contato identificados e também altera-los, por meio de funções *callback*.

O sistema de detecção de colisão do PhysX disponibiliza para a *broadphase* apenas a técnica SAP e não permite que sejam implementadas outras técnicas. É possível definir o tipo de SAP, simples ou múltipla. Na SAP múltipla o mundo é dividido em forma de grade, o número de células é definido na criação do mundo. Cada célula tem sua própria estrutura para armazenar as dimensões das AABBs e verifica a sobreposição apenas dos corpos posicionados em suas dimensões.

A detecção de colisão na *broadphase* utiliza algoritmos específicos para cada combinação de forma geométrica a fim de otimizar o processo.

3.5.1.3.2 Formas Geométricas

O PhysX suporta a descrição de objetos de colisão com formas primitivas, combinações ou malhas de triângulos.

As formas primitivas disponibilizadas são: caixa, esfera, cápsula e plano. Alem destas formas primitivas, o PhysX disponibiliza outras formas de uso mais especifico, como a roda para carro e *heightfield*.

3.5.1.4 Open Dynamics Engine (ODE)

ODE é um motor de física gratuito para uso sob licença GNU e de código livre. É escrito em C++ e sua interface em C. Foi criado por Russell Smith e conta com vários contribuidores.

Entre as principais funcionalidades estão: dinâmica de corpos rígidos e com articulações; detecção de colisão discreta, suporta a grande variedade de formas primitivas, também a compostas e malhas de triângulos.

Disponível para as plataformas PLAYSTATION 2, Xbox, PC, Linux e Mac OSX. O motor de física tem otimizações únicas para cada plataforma.

Entre os jogos comerciais criados utilizando o motor de física ODE estão: “Xpand Rally”, “BloodRayne 2”, “Resident Evil: The Umbrella Chronicles”.

ODE é um motor de física para detecção de colisão e dinâmica de corpos rígidos, incluindo corpos com articulações. ODE prioriza a rapidez e estabilidade sobre a precisão física.

O sistema de detecção de colisão e o sistema de dinâmica de corpos rígidos são independentes, possibilitando a utilização de apenas um deles, ou os dois em conjunto. Também é possível a integração com outros motores de jogos ou ao sistema de detecção de colisão ou dinâmica de outros motores de física.

O motor é desenvolvido na linguagem C++, porém sua interface é escrita em funções C, a fim de tornar sua utilização mais simples.

Ao compilar o ODE se tem a opção de gerar uma biblioteca estática (.lib), ou uma biblioteca dinâmica. A vantagem no uso de bibliotecas dinâmicas é que o componente se torna independente da aplicação, assim facilitando na atualização do componente, sem a necessidade de recompilar a aplicação.

Para utilizar o ODE em uma aplicação basta adicionar a biblioteca estática (completa ou complementar da biblioteca dinâmica) e utilizar a interface disponibilizada. Caso seja usada a opção de biblioteca dinâmica, a DLL gerada deve estar no mesmo diretório da aplicação.

3.5.1.4.1 Detecção de colisão

As funções principais do módulo de detecção de colisão do ODE é fornecer as formas de colisão, verificar quais objetos estão em contato e passar estas informações para o usuário.

Para que o ODE execute a detecção de colisão é necessário um mundo de colisão, no ODE este mundo se chama *space*, no *space* são adicionados todos os objetos de colisão ou outros *spaces*. Um objeto de colisão é uma forma geométrica (esfera, caixa, cilindro) com uma posição e orientação. Outros *spaces* podem ser adicionados um mundo de colisão, sendo que estes não têm posição nem orientação.

No ODE, as etapas *broadphase* e *narrowphase* são executadas em conjunto, a medida que os pares são selecionados na *broadphase*, são imediatamente verificados pela *narrowphase*, e então gerado os pontos de contatos, caso estejam realmente colidindo.

O desenvolvedor é obrigado a implementar a função que executa a *narrowphase*, que e chamada por *callback*, a partir do algoritmo da *broadphase*. Esta função deve fazer a verificação da colisão, por meio de funções disponibilizadas pelo ode, e também resolver a colisão.

As técnicas de *broadphase* são definidas pelo tipo de mundos que armazenam os objetos de colisão, sendo que todos implementam uma mesma interface. São disponibilizadas

três técnicas, mas também podem ser implementadas outras, as técnicas disponibilizadas são as seguintes:

a) simples: técnica que verifica a sobreposição de AABBs de todos os pares de corpos contidos no mundo, indicado apenas para mundos com poucos objetos;
b) *quadtree*: é uma abordagem BVH que utiliza *quadtrees* para armazenar as AABBs dos corpos, ou seja, o mundo é dividido em quatro áreas, e cada uma das áreas são divididas em outras quarto, assim recursivamente, organizando a árvore de forma que os corpos fiquem nas folhas. A verificação de colisão é feita em todos os pares de cada nó e seus filhos. A profundidade máxima da arvore pode ser definida na criação do mundo de colisão. Tem a limitação de ser necessário definir as dimensões máximas para o mundo;
c) *hash table*: está técnica utilize tabelas *hash* para armazenar os objetos de colisão, os objetos são mapeados nas tabelas de acordo com sua posição, dependendo do tamanho do volume limitante, ele pode ser divido em partes menores que são armazenadas separadamente. Uma colisão é identificada quando mais de uma AABB ou parte dela se encontram na mesma posição na tabela.

3.5.1.4.2 Formas Geométricas

O ODE suporta a descrição de objetos de colisão com formas primitivas, combinações ou malhas de triângulos.

As formas primitivas disponibilizadas são: caixa, esfera, cápsula, cilindro e plano. Alem destas formas primitivas, também são disponibilizadas outras formas de uso mais especifico, como o *heightfield*.

4 TRABALHOS CORRELATOS

Neste capítulo serão apresentados alguns trabalhos descritos na literatura, relacionados ao uso e desenvolvimento de motores de física ou de ambientes para sua integração, com finalidade de auxiliar no estudo sobre os motores de jogos e submores de física.

4.1 UMA ARQUITETURA DE MOTOR DE FÍSICA PARA GAMES 3D COM PROCESSAMENTO HÍBRIDO ENTRE CPU E GPU E DISTRIBUIÇÃO DINÂMINA DE CARGA

Joselli (2007) apresenta o GDE (GPU *Dynamics Engine*), um motor de física para jogos digitais 3D com processamento híbrido entre CPU e GPU e distribuição dinâmica de carga. A dissertação também apresenta a utilização do motor GDE juntamente com o *framework* GUFF (Games UFF).

Os motores de física requerem muito processamento computacional, pois necessitam de muitos cálculos matemáticos para desempenhar seu papel em um jogo digital. Geralmente esse processamento ocorre na CPU, que em determinados casos não mostra bom desempenho, com o avanço das GPUs programáveis esses cálculos podem ser direcionados para serem processados nestas placas de aceleração gráfica. A proposta do GDE é a distribuição automática de sua carga de processamento entre a CPU e a GPU (JOSELLI, 2007).

O conceito desta possibilidade de utilização de GPUs para outros fins é o GPGPU (*General-Purpose computation on* GPU), que é a utilização destas GPUs programaveis para processamento genérico, sendo que elas têm uma estrutura altamente paralela e voltada para

cálculos matemáticos. O processamento de cálculos matemáticos é altamente otimizado para grandes números de cálculos, por isso nem sempre é vantajoso redirecionar todo o processamento para a GPU, em casos de pequeno numero de corpos a CPU mostra melhor desempenho (JOSELLI, 2007).

O GDE tem como base o motor de física ODE, foi selecionado dentre outros vários motores disponíveis, sendo que os critérios de escolha foram: ser gratuito; ter código aberto; ter sido utilizado em jogos comerciais; ser atualização periódica; possuir boa documentação; possuir grande comunidade de desenvolvedores (JOSELLI, 2007).

Foram realizados testes com o motor de física GDE comparando os métodos implementados em GPU e CPU. Os resultados mostraram que este motor tem uma otimização em GPU, se comparado com a implementação em CPU, para grandes números de corpos rígidos, comprovando que com o uso da GPU, a simulação pode ter uma maior numero de corpos com um tempo de resposta mais baixo (JOSELLI, 2007).

Foram implementados os métodos para a distribuição entre GPU e CPU: definida pelo desenvolvedor; definida automaticamente por inteligência artificial; definida por heurísticas de decisão. Segundo os testes o modo automático se mostrou melhor, pois tira proveito de ambos os processadores, evita que a escolha seja feita pelo desenvolvedor e realiza a distribuição de acordo com a carga de cada processador (JOSELLI, 2007).

4.2 UM AMBIENTE DE ANIMAÇÃO DINÂMICA DE CORPOS RÍGIDOS

Oliveira (2006) propõe um ambiente de animação dinâmica de corpos rígidos, onde utiliza o motor de física para corpos rígidos PhysX. O objetivo do trabalho é o estudo dos fundamentos da animação por computador e o desenvolvimento de um ambiente de

animação de cenas 3D para visualização de simulações dinâmicas em aplicações de ciência e engenharia.

Os componentes que compõe o sistema de animação são: compilador da linguagem de animação; maquina virtual de animação; renderizador; controlador de animação; ligador e visualizador de arquivos de animação; e motor de física de corpos rígidos PhysX. O trabalho visa também a integração dos componentes do sistema com o motor de física utilizado (OLIVEIRA, 2006).

O desenvolvimento deste ambiente visa a expansão do framework OSW (*Object Structural Workbench*) desenvolvido na UFMS (Universidade Federal de Mato Grosso do Sul). No contexto da OSW, este sistema é um programa gráfico de analise numérica e de visualização de modelos sólidos baseados em física (OLIVEIRA, 2006).

O motor de física PhysX foi adotado no trabalho por se apresentar mais apto as necessidades do sistema quanto ao realismo, este motor tem como diferencial, o processamento de seus cálculos em uma unidade unicamente para este propósito, a PPU (*Physics Processor Unit*). O physX é desenvolvido pela Ageia Technologies, também fabricante da unidade de aceleração física utilizado pelo motor (OLIVEIRA, 2006).

Todos os objetivos do trabalho foram atingidos. Quanto ao motor de física PhysX e a sua integração com o sistema, é importante destacar que foi possível que o sistema foi desenvolvido de forma que não dependesse deste motor de física em particular; e que se mostrou atender as necessidade esperadas do sistema quanto a simulação dinâmica de corpos rígidos (OLIVEIRA, 2006).

5 TESTES, VALIDAÇÕES E COMPARAÇÕES ENTRE MOTORES DE FÍSICA

Neste capitulo será apresentada a avaliação dos submotores de física Bullet, ODE e PhysX, abordando o reuso, extensibilidade e simulação de física. Foram aplicados testes, validações e comparações entre os submotores de física para avaliação de desempenho, realismo dos resultados, abstração, modularidade, produtividade e integração.

5.1 METODOLOGIA

Após o levantamento bibliográfico a cerca de reuso, extensibilidade e simulação de física em jogos digitais, a fim de compreender melhor o domínio do trabalho proposto, foram selecionados três entre alguns dos motores disponíveis atualmente.

Tendo os três motores selecionados, foi feito um estudo mais detalhado de cada um deles, buscando compreender melhor sua arquitetura, características, funcionalidades e interfaceamento, para então poder realizar testes, validações e comparações entre os motores selecionados a fim de avaliar questões como desempenho, realismo dos resultados, abstração, modularidade, produtividade e integração.

5.1.1 Seleção dos motores de física

Diante do grande número de motores de física disponíveis e não sendo possível avaliar todos, o trabalho foi limitado a avaliar três motores de física.

Como pré-requisito, os motores deverão estar disponíveis para uso, não importando se para uso comercial ou não comercial, ou se possui código aberto, porém, deve ser selecionado no mínimo um motor de código livre.

Outros itens foram levados em consideração na seleção dos motores, como o seu uso em jogos comerciais, documentação, atualização, comunidade de desenvolvedores.

A princípio, os três motores selecionados foram PhysX, Havok e Bullet, porém por uma clausula nos termos de uso do Havok, que diz que a licença gratuita não pode ser utilizada para testes e *benchmarks* públicos, ele não pode ser usado.

Os motores selecionados foram: Bullet, ODE e PhysX.

5.1.2 Testes, validações e comparações entre motores de física

A seguir os testes, avaliações e comparações entre os motores selecionados.

5.1.2.1 Desempenho da Broadphase

A implementação de *broadphase* de melhor desempenho é aquela que seleciona o menor número de pares possível, ou seja, a quantidade de pares que estão realmente colidindo, num menor período de tempo.

Esta simulação visa comparar as técnicas de *broadphase* de cada motor. Para que os resultados não sofram influencia da simulação dinâmica dos corpos, pois cada motor apresenta simulações diferentes, foi elaborado um ambiente de estresse para testar a *broadphase* (Figura 13).

O ambiente é delimitado por seis paredes (caixas estáticas) onde um número determinado de corpos se movimentam em direções aleatórias e velocidade constante, para isso foi necessário atribuir gravidade zero ao mundo, restituição total das forças nas colisões e correção da velocidade de todos os corpos a cada *frame*.

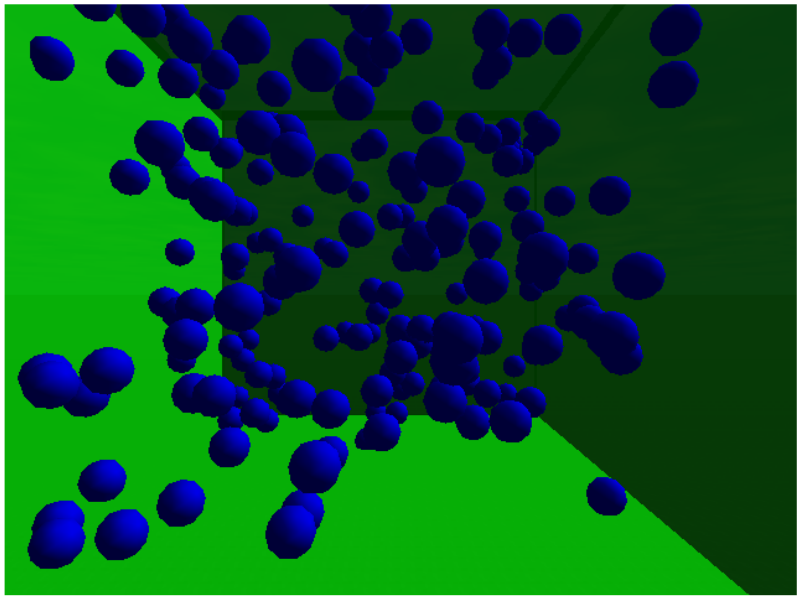

Figura 13. Ambiente do teste de desempenho da broadphase

Para cada técnica de *broaphase* testada foi utilizado o mesmo numero de corpos, mesma velocidade e mesmo *step time* (0.01s). A cada 100 *frames* simulados, um *frame* foi utilizado como amostra, deste, foram coletadas os seguintes dados: número de pares selecionados pela *broaphase*; número de pares que estão realmente colidindo; e o tempo de execução da etapa de *broaphase*.

Os testes foram executados em diferentes situações, diferindo quanto à velocidade dos corpos e ao número de corpos no ambiente, a fim de observar como cada situação afeta o desempenho das técnicas testadas.

Foram utilizadas as seguintes configurações: 50 corpos e velocidade de 100m/s (Figura 15); 200 corpos e velocidade de 100m/s (Figura 14); 800 corpos e velocidade de 100m/s (Figura 16); 200 corpos e velocidade de 2m/s (Figura 17).

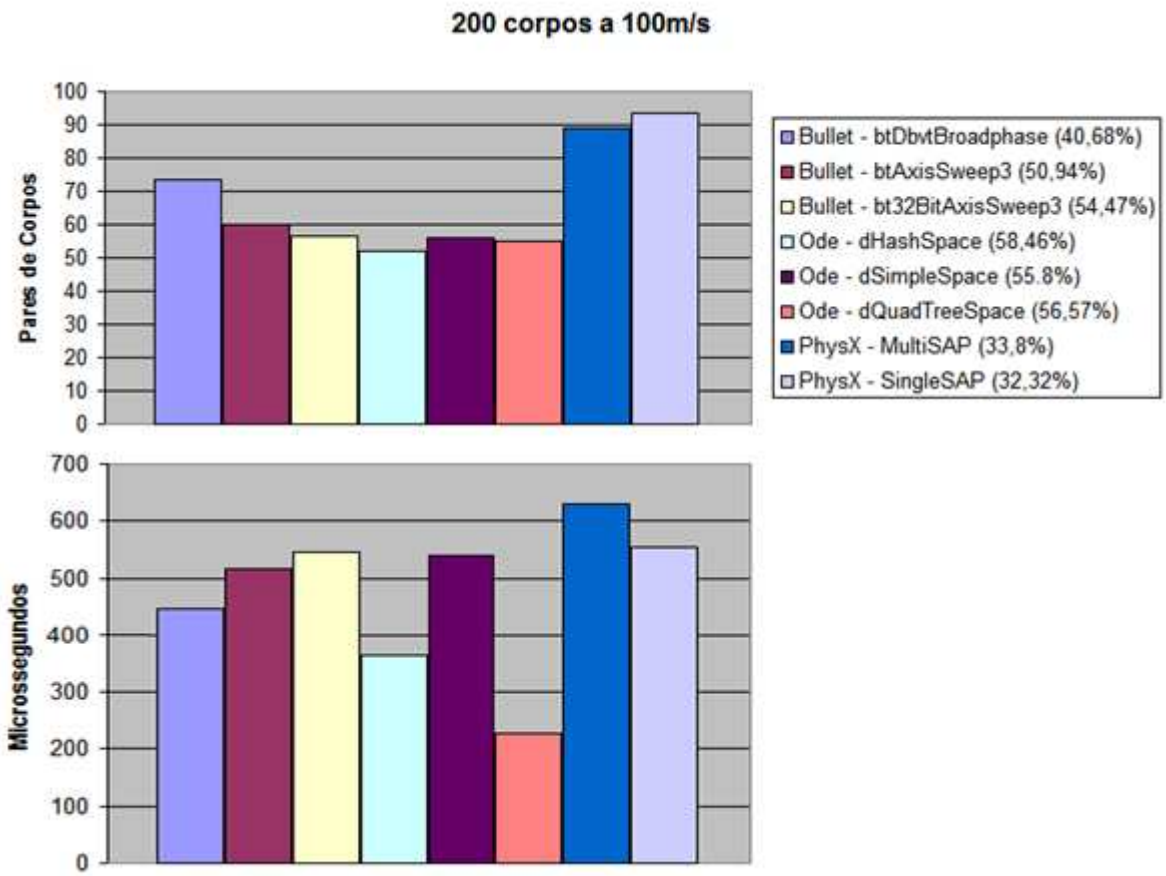

Figura 14. Resultados do teste de broadphase de 200 corpos a 100m/s

Na simulação de 200 corpos à velocidade de 100m/s, as técnicas SAP, SAP32, hash, simple e quad tiveram o número de pares selecionados na *broadphase* semelhantes, entre 50 e 60 pares por *frame*, sendo em media 30 pares realmente colidindo. Destes, destaque para o quad, que teve o melhor tempo, devido seu particionamento espacial, diminuindo o número de verificações de sobreposição.

SAP, SAP32, Single-SAP e Multi-SAP tiveram um dos maiores tempos devido a alta velocidade do corpos, que gera muitas atualizações na listas de armazenamento dos volumes limitantes, que consome mais tempo para reordenação. Quanto ao SAP32 em relação ao SAP, pode-se observar a diminuição do número de pares selecionados, devido à utilização de inteiro 32 *bits*, porém aumento no tempo de *broadphase*. Situação semelhante também ocorre com o Single-SAP e Multi-SAP, porém, devido à divisão do mundo em grade.

A alta velocidade dos corpos também influenciou nos resultados do DBVT, pois as arvores AABBs necessitam ser constantemente atualizadas.

Apesar de técnica simples verificar todos os pares de objetos no mundo, não apresentou um tempo tão alto em relação as outras, pois o número de objetos é pequeno.

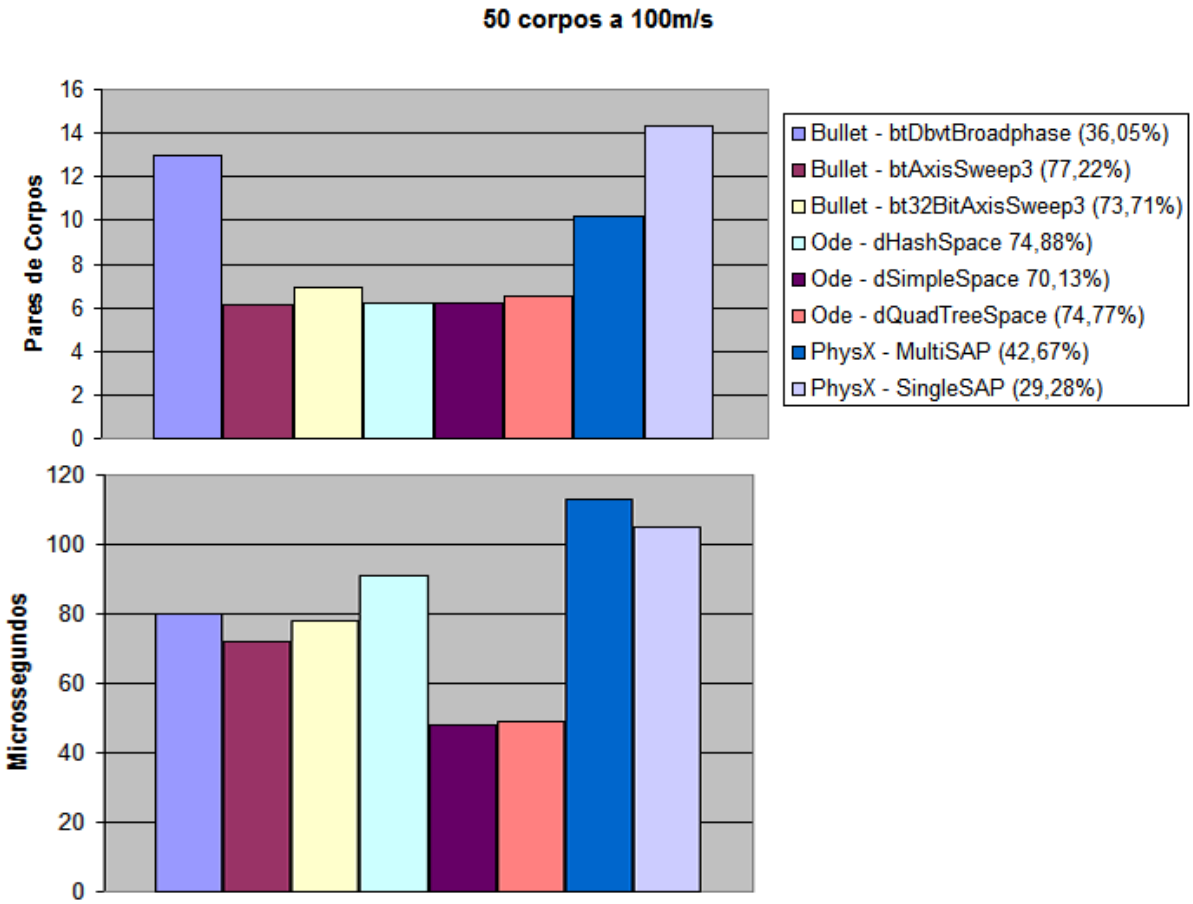

Figura 15. Resultados do teste de broadphase de 50 corpos a 100m/s

Na simulação de 50 corpos à 100m/s o melhor desempenho foi do simples, devido ao número pequeno de corpos, que necessita de poucas verificações de sobreposição para selecionar os pares.

A alta velocidade dos corpos e o número baixo de pares realmente colidindo, em media 4 pares por *frame*, fazem com que os corpos se movimentem por todo o ambiente sem

interrupções, este comportamento é o que causou os altos tempos do DBVT, SAPs e *Hash Table*, pois aumenta o custo de suas atualizações estruturais.

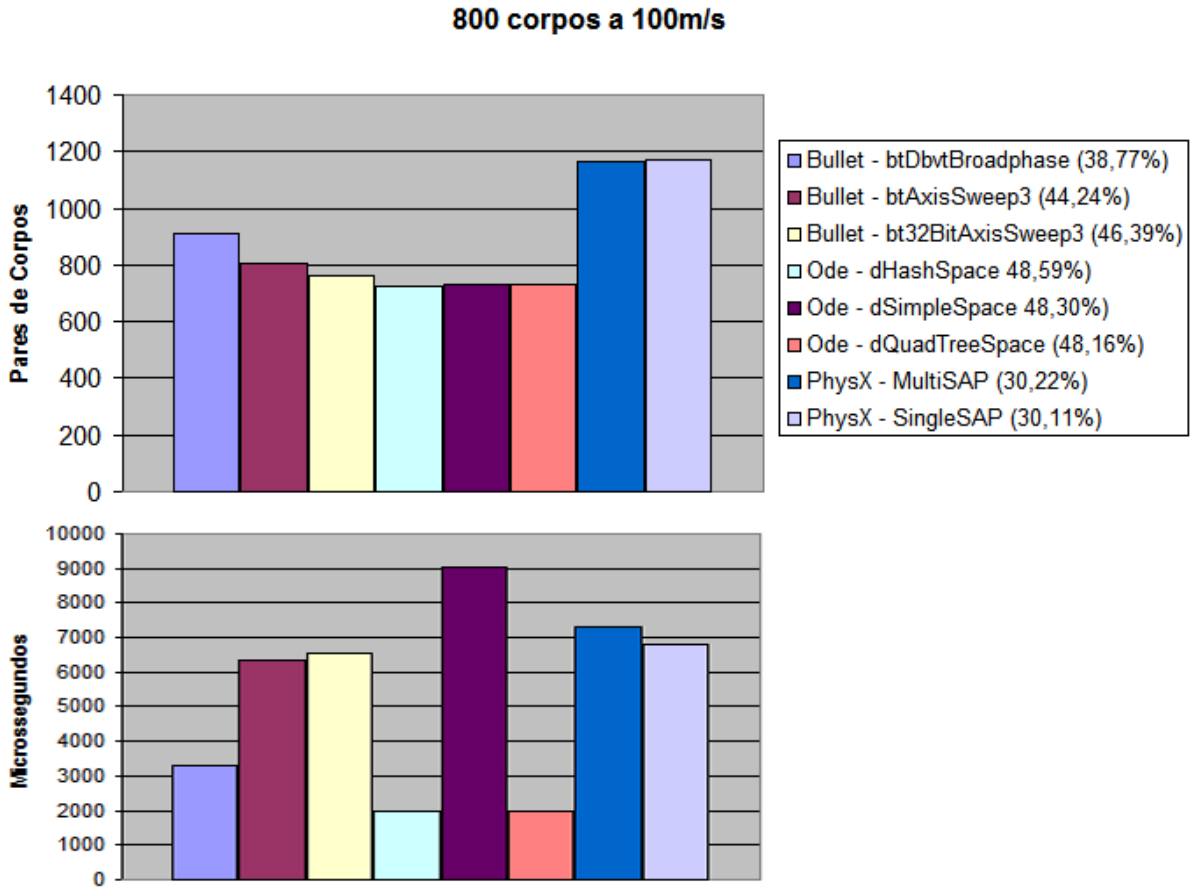

Figura 16. Resultados do teste de broadphase de 800 corpos a 100m/s

Na simulação de 800 corpos à 100m/s, com aumento significativo no número de corpos em relação às outras simulações, fica mais visível o problema dos testes de colisão de ordem quadrática, como na técnica simples.

Ocorre uma situação inversa da simulação anterior, os 800 corpos da simulação ocupam boa parte do ambiente, e o número de colisões aumenta bastante, em média 360 por *frame*, fazendo com que os corpos se movam pouco, por falta de espaço. Está situação foi favorável para o DBVT, *Hash Table* e *QuadTree*, pois suas atualizações estruturais são menores.

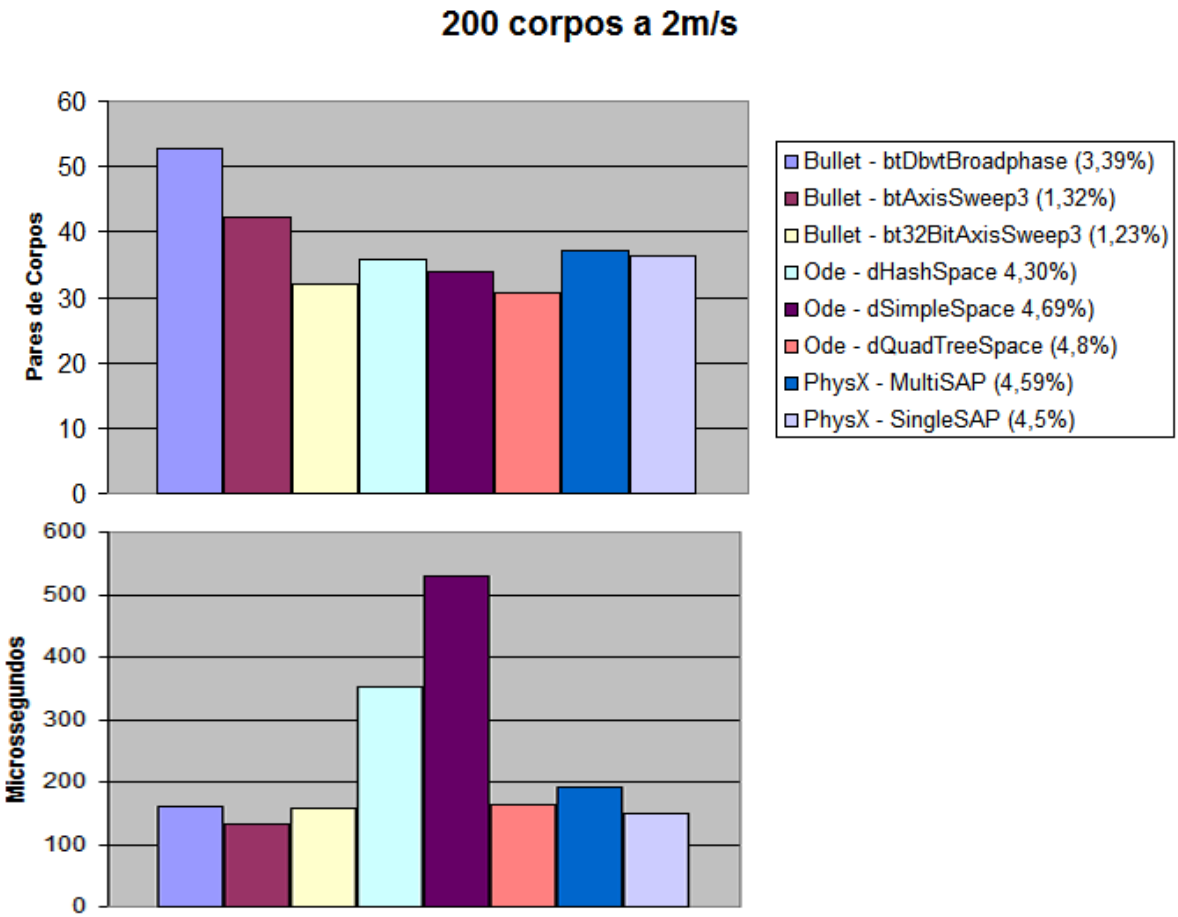

Figura 17. Resultados do teste de broadphase de 200 corpos a 2m/s

Na simulação de 200 corpos à velocidade de 2m/s o destaque é para DBVT e SAPs, que tiveram um melhor tempo em relação aos testes anteriores, o que favoreceu isto foi a baixa velocidade dos corpos. No caso do DBVT, a estrutura das arvores sofrem menos atualizações e no caso das SAPs a ordenação das listas que armazenam as dimensões das AABBs têm custo menor.

5.1.2.2 Incremental com colisão

Nesta simulação, são comparados os tempos de execução da *broadphase* de acordo com o número de corpos colidindo no mundo.

A simulação dinâmica não é utilizada neste teste, os corpos são estáticos, ao serem adicionados ao mundo são posicionados em forma de pilha, sempre colidindo com o corpo abaixo.

A cada *step* é adicionado um novo corpo no mundo, assim pode-se observar o aumento do tempo de *broadphase* e total da colisão de acordo com número de corpos no ambiente.

Na Figura 18 são apresentados os resultados dos testes.

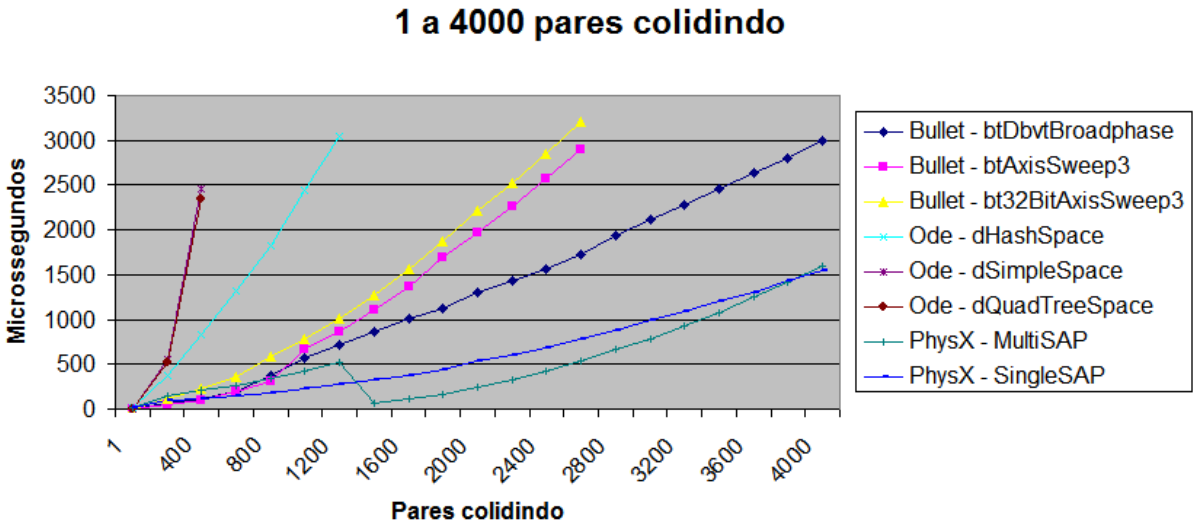

Figura 18. Resultados do teste incremental com colisão

Para melhor visualização do gráfico, os tempos acima de 3200 microssegundos foram omitidos.

O DBVT e SAPs tiveram resultados semelhantes até cerca de 600 corpos.

O Single-SAP e Multi-SAP, do PhysX, tiveram melhor desempenho, destaque para o Multi-SAP, que permite dividir o mundo em forma de grade, e processar cada uma das células individualmente, por isto ocorreu a diminuição de tempo por volta dos 1400 corpos.

No caso do *QuadTree* e *Hash Table*, o desempenho foi afetado pela forma em que os corpos foram dispostos no mundo, em pilha, não favorecendo no uso de suas estruturas. No

caso do *QuadTree*, fez com que as verificações de sobreposição ocorressem como no simples, em ordem quadrática. É interessante observar também que no *Hash Table*, a partir de 1800 corpos no mundo, ocorre um erro durante a *broadphase* e a aplicação é finalizada.

Pode ser notado também, a perda de desempenho do SAP32 em relação ao SAP, devido a utilização de inteiro de 32 *bits*, ao invés de inteiro 16 *bit*s.

5.1.2.3 Narrowphase

Esta avaliação busca comparar os tipos de formas geométricas disponíveis por cada motor de física e também a matriz de colisão entre as formas geométricas.

Na Tabela 1 seguir são apresentadas as formas primitivas suportas por cada motor.

	Caixa	Esfera	Capsula	Cilindro	Cone	Multi-Esfera	Plano
Bullet	x	x	x	x	x	x	x
ODE	x	x	x	x			x
PhysX	x	x	x				x

Tabela 1. Formas primitivas suportadas pelos motores

A detecção de colisão na *narrowphase* utiliza algoritmos diferentes para cada combinação de formas envolvidas na colisão, nas Tabelas 2, 3 e 4 são apresentados as combinações suportadas pelos motores ODE, PhysX e Bullet respectivamente.

	Ray	Plano	Plano	Caixa	Cáps.	Cilind.	Malha triang.	Convexas	*Heightfield*
Ray		x	x	x	x	x	x	x	x
Plano	-		x	x	x	x	x	x	
Plano	-	-	x	x	x	x	x	x	x
Caixa	-	-	-	x	x	x	x		x
Cápsula	-	-	-	-	x		x		x
Cilindro	-	-	-	-	-		x		x
Malha de triangulo	-	-	-	-	-	-	x		x
Convexas	-	-	-	-	-	-	-	x	x
Heightfield	-	-	-	-	-	-	-	-	

Tabela 2. Combinações de colisões suportadas pelo ODE

	Esfera	Caixa	Capsula	Plano	Convex	*Height.*	Pneu	Malha de Triang.	*Heightfield* (malha)
Esfera	x	x	x	x	x	x	x	x	x
Caixa	-	x	x	x	x	x	x	x	x
Cápsula	-	-	x	x	x	x	x	x	x
Plano	-	-	-		x		x	x	
Convexa	-	-	-	-	x	x	x	x	x
Heightfield	-	-	-	-	-		x		
Pneu	-	-	-	-	-	-		x	x
Malha de triângulos	-	-	-	-	-	-	-		
Heightfield (Malha)	-	-	-	-	-	-	-	-	

Tabela 3. Combinações de colisões suportadas pelo PhysX

	Caixa	Esf.	Conv.	Cil.	Cone	Caps.	Mult.	Comp.	Malha triang.	Plano	*Heigh.*
Caixa	x	x	x	x	x	x	x	x	x	x	x
Esfera	-	x	x	x	x	x	x	x	x	x	x
Convexa	-	-	x	x	x	x	x	x	x	x	x
Cilindro	-	-	-	x	x	x	x	x	x	x	x
Cone	-	-	-	-	x	x	x	x	x	x	x
Capsula	-	-	-	-	-	x	x	x	x	x	x
Multi-Esfera	-	-	-	-	-	-	x	x	x	x	x
Composição	-	-	-	-	-	-	-	x	x	x	x
Malha de triangulo	-	-	-	-	-	-	-	-	x	x	
Plano	-	-	-	-	-	-	-	-	-		
Heightfield	-	-	-	-	-	-	-	-	-	-	

Tabela 4. Combinações de colisões suportadas pelo Bullet

5.1.2.4 Extensibilidade (Wrappers)

Este teste verifica a funcionamento de *wrappers* para motores de física, visando o desempenho, realidade visual e identificação de problemas.

No teste foi utilizado o PAL, que suporta um grande número de motores de física. Foram testados com os motores Bullet, ODE e PhysX.

Após problemas na compilação do PAL, foi feito uma simulação simples e verificado funcionamento com os três motores. A simulação teve mau funcionamento com o motor ODE, ocorreu um erro na inicialização e o programa é finalizado, não foi identificado a causa do problema.

Com Bullet e PhysX o PAL teve funcionamento normal, não foi identificado nenhum problema de dinâmica, detecção de colisão ou de visualização.

Quanto ao desempenho, foi uma pequena perda, porém nada significativo, sendo que a simulação era simples, uma esfera caindo até o chão.

5.1.2.5 Integração dos sistemas de detecção de colisão e dinâmica

Há a possibilidade de integrar um motor de física a um motor de jogo ou outro motor de física, a fim de adicionar funcionalidades a um deles, ou substituir funcionalidades em busca de melhor desempenho, estabilidade, etc.

No motor de física ODE, o sistema de detecção de colisão é independente do sistema de dinâmica de corpos rígidos e este é independente do sistema de detecção de colisão, portanto, permite que seja utilizado apenas um deles, e podem ser integrados a outros motores.

Já no Bullet o sistema de detecção de colisão é independente dos sistemas de dinâmica, porém, os sistemas de dinâmica dependem do sistema de detecção de colisão, neste caso não seria possível utilizar apenas o sistema de dinâmica do Bullet.

No PhysX o sistema de física engloba a detecção de colisão e dinâmica, portanto não é possível utilizar nem integrar apenas um deles.

Neste teste será integrado o sistema de detecção de colisão do Bullet ao sistema de dinâmica do ODE, a fim de comparar a o realismo visual e desempenho em relação a mesma simulação utilizando apenas o ODE para a detecção de colisão e dinâmica.

A integração ocorrera na aplicação, da seguinte forma: o mundo de dinâmica do ODE e o mundo de colisão do Bullet são criados, assim como as estruturas necessárias; ao ser adicionado um corpo no sistema de dinâmica, deve-se adicionar um objeto de colisão ao sistema de detecção de colisão; no andamento do aplicação, a cada *step*, deve-se atualizar as posições dos objetos de colisão de acordo com os corpos dinâmicos; executar a detecção de colisão; verificar as colisões e pontos de contato, converter para a estrutura do ODE para então resolver as colisões; e então executado o *step* do mundo dinâmico.

A simulação consiste em um plano e seis esferas caindo e quicando sobre o plano, como apresentado na figura 19.

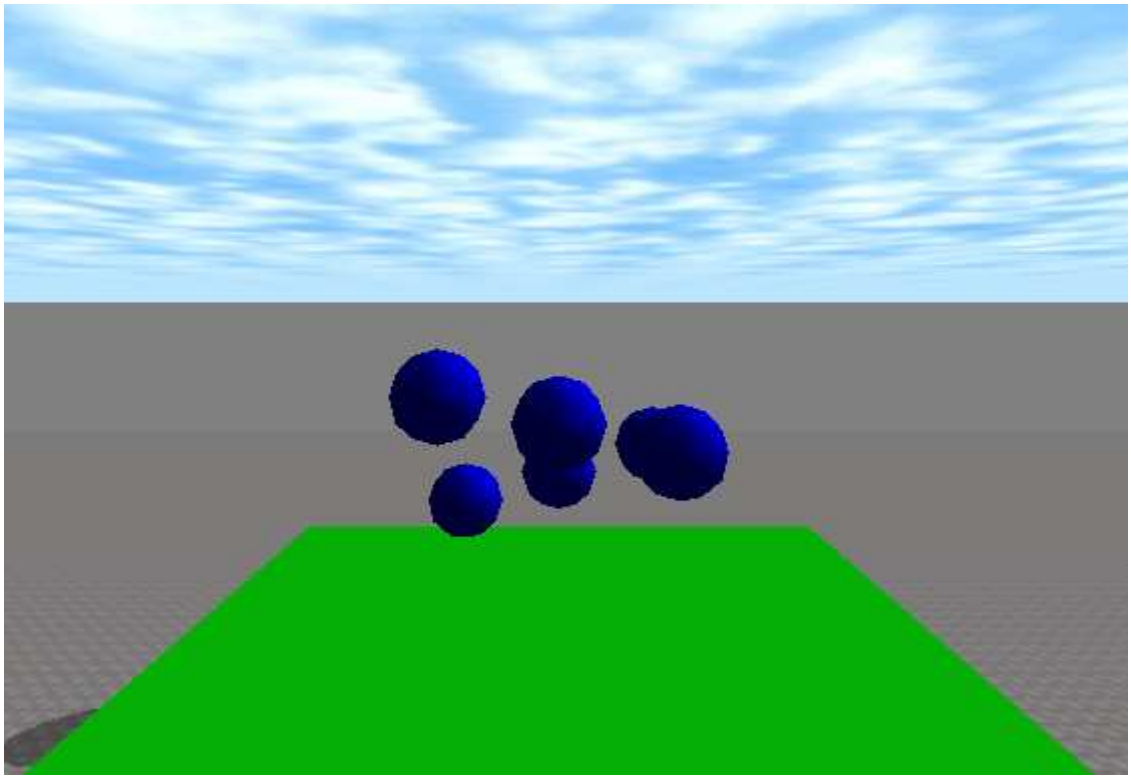

Figura 19. Ambiente do teste de integração entre motores

Quanto ao desempenho, foi coletado o tempo da detecção de colisão e simulação dinâmica, em dois momentos, quando não há nenhum corpo em colisão, e quando todos estão colidindo com o plano, nos dois casos não se teve diferença significativa, em media 20 microssegundos e 95 microssegundos respectivamente.

Quanto à realidade visual da simulação, não foi identificado nenhuma diferença ou falha. Porém, no Bullet os objetos de colisão têm uma margem de colisão, identificando como colisão quando algum objetos está dentro desta margem, no ODE não existe esta margem de colisão, por isso, as colisões sem penetração devem ser ignoradas, caso contrario podem ocorrer falhas visuais no momento da colisão.

É importante ter conhecimento também que os dois motores trabalham com sistema de coordenadas diferentes, sendo necessária conversão quando forem utilizadas.

5.2 RESULTADOS OBTIDOS

A partir da descrição detalhada de cada um dos motores, dos resultados dos testes realizados e também da implementação necessária para os testes foram observados os seguintes resultados e também elaborado a Tabela 5, com comparações quanto ao uso, desenvolvimento, funcionalidades e detecção de colisão dos motores.

	Bullet	**ODE**	**PhysX**
Uso			
Gratuito para jogos comerciais	Sim	Sim	Sim
Código Livre	Sim	Sim	Não
Licença de uso	ZLib	GNU	–
Utilizado em jogos comerciais AAA	Sim	Sim	Sim
Plataformas suportadas	Windows, Linux, Mac OSX, PlayStation 2, PlayStation 3, XBox, XBox 360, Wii, iPhone	Windows, Linux, Mac OSX, PlayStation 2, PlayStation 3, XBox, XBox 360, PSP	Windows, Linux, PlayStation 2, PlayStation 3, XBox, XBox 360, Wii,

Desenvolvimento			
Linguagem de implementação	C++	C++	C++
Abordagem de desenvolvimento	Desenvolvimento baseado em objetos	Desenvolvimento baseado em objetos	Desenvolvimento baseado em objetos
Distribuição	Bibliotecas estáticas	Bibliotecas estáticas ou dinâmicas	Bibliotecas dinâmicas
Interfaces	Classes abstratas C++	Classes abstratas C++	Funções C
Funcionalidades			
Dinâmica de corpos rígidos	Sim	Sim	Sim
Dinâmica de corpos flexíveis	Sim	Não	Sim
Detecção de colisão discreta	Sim	Sim	Sim
Detecção de colisão contínua	Sim	Não	Sim
Dinâmica de fluídos	Não	Não	Sim
Outras funcionalidades	Suporte a COLLADA Physics, Integração com Blender e Maya	–	Simulação de roupas, simulação de personagens, pré-processamento de malhas de triângulos
Detecção de colisão			
Formas primitivas	Sim	Sim	Sim
Composições	Sim	Sim	Sim
Malhas de triângulo	Sim	Sim	Sim
Abordagens de *broaphase*	Hierarquia dinâmica de volumes limitantes, SAP, SAP 32 *bit*	Simples, *Quadtree*, *Hash table*	SAP, Multi-SAP

Tabela 5. Quadro comparativo entre os motores avaliados

Os três motores de física avaliados, Bullet, ODE e PhysX, têm como funcionalidades principais, a dinâmica de corpos rígidos e a detecção de colisão discreta, porém têm outras funcionalidades que extendem ou auxiliam as funções principais do motor, que também podem ser levadas em consideração na escolha de um motor de física para uso. Por exemplo, o Bullet possui detecção de colisão contínua, dinâmica de corpos flexíveis, integração com Blender e Maya, entre outros; o PhysX possui suporte a processamento da na GPU, simulação dinâmica para roupas, fluidos, corpos flexíveis e personagens, detecção de

colisão contínua, ferramenta para pré-processamento de malhas de triângulo, etc.; quanto ao ODE, apesar de ser um dos primeiros motores de física de código aberto, não possui outras funcionalidades que se possa destacar das principais.

Os testes realizados focaram mais na detecção de colisão. Nos testes de *broadphase*, as técnicas mostraram resultados diferentes para situações diferentes, por exemplo, para ambientes onde os corpos não têm muito movimento, as técnicas SAP tiveram melhor resultado, porém para ambientes com muito movimento dos corpos, estavam entre os piores resultados. Para uso geral, as técnicas de *broadphase* que tiveram melhores resultados foram a DBVT do Bullet e QUADTREE do ODE. Os testes com as técnicas SAP do PhysX não mostraram os melhores resultados, porém, com PhysX, há a possibilidade de processar a *broadphase* na GPU, que melhora bastante seu desempenho.

Quanto às formas geométricas, todos os motores disponibilizam as primitivas básicas, porém o Bullet suporta outras, de uso mais especifico, como cone, cilindro e multi-esfera. Alem das primitivas, todos os motores suportam composição de formas geométricas e malhas de triângulos, então pode-se construir outras forma de colisão necessárias, porém deve-se saber que a detecção de colisão com malhas de triangulo é mais complexa e custosa computacionalmente.

Na *narrowphase,* o Bullet é o motor que suporta detecção de colisão entre mais combinações de formas geométricas, não suportando apenas a colisão entre *heightfield* e malha de triângulos. O PhysX não suporta a detecção de colisão entre malhas de triangulo, e o ODE entre objetos convexo contra caixa, cápsula, cilindro e malha de triangulo; e cilindro contra cápsula e cilindro. Nestes casos podem ser utilizadas outras primitivas ou composição de primitivas.

Outro ponto do processo de detecção de colisão a ser observado é a execução do da *narrowphase* e resolução dos contatos, o ODE obriga a implementação de como será

executado os testes de sobreposição na *narrowphase* e resolução dos contatos por meio de uma função *callback*, isto pode ser bom para o caso de necessitar fazer customizações. No caso do Bullet e PhysX, isso é feito pelo próprio motor, porém disponibilizam formas para customizar isto caso necessário.

O desenvolvimento de aplicações utilizando *wrappers* para motores de física, ao invés utilizar diretamente o motor de física, pode ser proveitoso caso necessite alterar o motor que esta sendo utilizado sem necessidade de alteração de código fonte da aplicação, porém deve ser levado em consideração que a utilização de *wrappers* pode resultar em perda de desempenho e funcionalidades, uma vez que o *wrapper* precisa abstrair as especificações de cada um dos motores fornecendo apenas uma interface geral para todos os motores, por este motivo o *wrapper* necessita fazer um processamento adicional para comunicação com o motor de física, causando perda de desempenho.

A integração entre motores também é um tópico interessante, pode-se integrar partes de motores na aplicação, a fim de melhor desempenho em algum aspecto ou mesmo adição de funcionalidades. É possível fazer a integração de partes que tenham independência do resto do motor, dos motores avaliados, o Bullet permite a integração do sistema de detecção de colisão, o ODE permite a integração do sistema de detecção de colisão e dinâmica de corpos, já o PhysX não permite nenhuma integração parcial, pois seu sistema de detecção de colisão dependendo do sistema de dinâmica, e o de dinâmica depende da detecção de colisão.

A integração realizada nos testes, com o sistema de detecção de colisão do Bullet e dinâmica de corpos do ODE, não apresentou falhas de dinâmica, nem de detecção de colisão. Como a simulação executada foi simples, não mostrou nenhum ganho, porém, também não mostrou perda de desempenho, devido ao processamento adicional para a integração.

Quanto ao desenvolvimento dos motores, os três utilizam a linguagem C++, utilizando orientação a objetos. O Bullet é organizado por módulos, cada módulo gera um arquivo de distribuição em formato de biblioteca estática. Já o ODE, tem apenas um projeto, que pode gerar para distribuição uma biblioteca estática ou dinâmica. O PhysX é organizado em módulos e os arquivos de distribuição são bibliotecas dinâmicas, porém para a execução de uma aplicação que utiliza PhysX é necessário a instalação de um pacote de bibliotecas da fabricante, que contem todas as versões do PhysX.

A vantagem de ser utilizar bibliotecas dinâmicas é que o motor fica independente do aplicativo que o utiliza, assim pode ser atualizada, sem a necessidade de recompilação da aplicação; e também o tamanho do executável do aplicativo fica menor, pois não tem todo o código referente ao motor, que fica na biblioteca dinâmica.

Quanto a interface dos motores, o Bullet e PhysX disponibilizam as classes abstratas (.h) utilizada em seu desenvolvimento, com um detalhe que no PhysX os objetos não são instanciados da forma tradicional invocando o construtor, são disponibilizadas na interface, métodos que instanciam o objeto e retornam a aplicação. O ODE disponibiliza uma interface C com funções, que torna a utilização mais simples.

CONCLUSÃO

Atualmente o reuso é totalmente necessário no desenvolvimento de jogos digitais, devido à alta complexidade dos jogos atuais possibilitada pela evolução constante do *hardware*, hoje seria quase impossível criar um jogo sem reutilizar nenhum recurso, implementando toda a interação com diferentes especificações de *hardware* e plataforma, gerenciamento da parte gráfica, física, sonora, interativa, etc. Na área dos jogos digitais o reuso é mais encontrado em forma de motores, que fornecem a abstração de *hardware*, plataforma e cálculos necessários para um determinado objetivo, como o gerenciamento gráfico, simulação física, gerenciamento de IA, etc., fazendo com que o desenvolvedor necessite do conhecimento apenas das interfaces dos motores e do domínio da aplicação para desenvolver um jogo. Além de facilitar o desenvolvimento, os motores proporcionam maior produtividade, redução dos custos e esforços durante o desenvolvimento e também maior qualidade no resultado final. O motor de simulação física é um dos mais importantes, pois é ele que prove o comportamento realístico dos objetos através das funcionalidades de detecção de colisão e dinâmica de corpos.

Após o levantamento bibliográfico acerca de reuso, extensibilidade e simulação de física em jogos digitais, a fim de compreender melhor o domínio do trabalho proposto, foram selecionados três entre alguns dos motores disponíveis atualmente, a partir do conhecimento superficial das características e funcionalidades de cada um. Tendo os três motores selecionados, foi feito um estudo mais detalhado de cada um deles, buscando compreender melhor sua arquitetura, características, funcionalidades e interfaceamento, para então poder realizar testes, validações e comparações entre os motores selecionados a fim de avaliar questões como desempenho, realismo dos resultados, abstração, modularidade, produtividade e integração.

Os resultados dos testes, validações e comparações mostraram que nenhum dos motores é melhor em todos os aspectos, cada motor tem pontos fortes em alguns aspectos, porém em outros não mostram resultados tão bons. Ainda assim é possível indicar o Bullet como o melhor dos motores avaliados, tendo em vista seus resultados nas simulações, funcionalidades além da dinâmica de corpos rígidos e detecção de colisão discreta, possibilidades de extensibilidade e ter código aberto. Porém, na possibilidade de aquisição de um motor de física comercial, o PhysX deve ser levado em consideração, diante de seus resultados, possibilidade de processamento na GPU, documentação e suporte. Um dos aspectos mais problemáticos em projetos de código aberto são a documentação e o suporte, a documentação geralmente não completa ou às vezes inexistente; e o suporte a ferramenta, geralmente prestado pela comunidade de desenvolvedores do projeto e utilizadores, pode ser demorado ou não atendido. Quanto ao reuso, foi observado que todos utilizam a abordagem do desenvolvimento baseado em objetos, fazendo necessário o conhecimento especifico de cada classe reutilizada, e que as abordagens da engenharia de *software* poderiam ser melhores aplicadas a fim de conseguir um melhor nível de reuso.

Como trabalhos futuros, podem ser citados, a extensão das avaliações para outros aspectos não focados neste trabalho, como a dinâmica de corpos flexíveis, detecção de colisão contínua, dinâmica de fluidos; a integração de motores de física, como utilizando a detecção de colisão do Bullet e dinâmica do ODE, possibilidade comentada diversas vezes na comunidade Bullet e ODE; Estudo ou desenvolvimento de *wrappers* para motores de física; e também o desenvolvimento de um motor, ou extensão de um já existente, seguindo algum padrão de CBSE.

REFERÊNCIAS

BACHMANN, Felix et al. **Volume II: Technical Concepts of Component-Based Software Engineering.** 2. ed. Pittsburgh: Carnegie Mellon University, 2000. 52 p. Disponível em: <http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr008.pdf>. Acesso em: 24 jun. 2009.

BATTAIOLA, André Luiz et al.. Desenvolvimento de Jogos em Computadores e Celulares. **Revista de Informática Teórica e Aplicada**, Edição Especial: Computação Gráfica e Processamento de Imagens, Porto Alegre, Vol. VIII, N. 2, Outubro de 2001.

BISHOP, Lars et al. Designing a PC Game Engine. **Ieee Computer Graphics And Applications**, v. 18, n. 1, p.46-53, jan. 1998.

BITTENCOURT, João Ricardo; OSÓRIO, Fernando Santos. Motores de Jogos para Criação de Jogos Digitais - Gráficos, Áudio, Interface, Rede, Inteligência Artificial e Física. In: V ESCOLA REGIONAL DE INFORMÁTICA DE MINAS GERAIS (ERI-MG 2006), 2006, Belo Horizonte. **Anais da V ERI-MG SBC.** Belo Horizonte: PUC Minas, 2006. v. 1, p. 1 – 36.

BOOCH, Grady. **Software Components with Ada:** Structures, Tools, and Subsystems. Redwood City: Benjamin-cummings, 1987.

BUSCHMANN, F., Meunier, R., Rohnert, H., Sommerland, P. & Stal, M, **Pattern-Oriented Software Architectur A System of Patterns**, John Wiley & Sons, 1996

CLEMENTS, Paul C**. From Subroutines to Subsystems:** Component Based Software Development. The American Programmer, v. 8, n. 11, 1995.

CRNKOVIC, Ivica; LARSSON, Magnus (Ed.). **Building Reliable Component-Based Software Systems**. Norwood: Artech House, 2002. 413 p.

CRUZ, Heron Sampaio da; SANTOS, Igor Edington Anselmo. **PhysicsXML**: Esquema XML para descrição de cenas em motores de física 2D. Monografia, Curso de Informática da Universidade Católica do Salvador, 2009.

D'SOUZA, D.; WILLS, A. **Objects, Components and Frameworks with UML**: The Catalysis Approach. [S.l.]: Addison-Wesley, 1999. (Object Technology Series).

DEUTSCH, P. Frameworks and reuse in the smalltalk 80 system. In: BIGGERSTAFF, T. **Software reusability**. New York: ACM Press, 1989. v.1, p. 57-71.

EBERLY, David H.. **3D Game Engine Architecture:** Engineering Real-Time Applications with Wild Magic. San Francisco: Elsevier Inc., 2005. 736 p. (The Morgan Kaufmann Series in Interactive 3D Technology).

FAIRLEY, R. **Software Engineering concepts**. [S.l.]: McGraw-Hill, 1986.

FALSTEIN, N. Game Design: **How Platform Choices Dictates Design**, Game Developer. Maio, 1997.

FAYAD, Mohamed C.; SCHMIDT, Douglas C.; JOHNSON, Ralph E. **Building Appiication Frameworks** - Object-Oriented foundations of frameworks design. Estados Unidos: Wiley, 1999.

GAO, Jerry Zeyu; TSAO, H.-s. Jacob; WU, Ye. **Testing and Quality Assurance for Component-Based Software.** Norwood: Artech House, 2003. 466 p.

GEE, James Paul. **What video games have to teach us about learning and literacy**. New York, Palgrave Macmillan, 2003.

GOMES, Carlos Miguel Pimenta. **Generic Physics**: a conceptual interface for rigid body physics engines. 2007. 82f. Dissertação (Mestrado em Engenharia Informática e de Computadores) - Instituto Superior Técnico, Universidade Técnica de Lisboa, Lisboa, Portugal. 2007.

HECKER, Chris. Physics in computer games. **Communications Of The Acm**, New York, v. 43, n. 7, p.34-39, jul. 2000.

JOHNSON, Ralph E. FOOTE, Brian. Designing Reusable Classes, Journal of Object. Oriemed Programming, August/September 1988.

JOSELLI, Mark Eirik Scortegagna. **Uma Arquitetura de Motor de Física para Games 3D com Processamento Híbrido entre CPU e GPU e Distribuição Dinâmina de Carga.** 2007. 63 f. Dissertação (Mestrado) - Curso de Computação, Universidade Federal Fluminense, Niterói, 2007.

KISHIMOTO, André. **Inteligência Artificial em Jogos Eletrônicos**. 2004. Disponível em <http://www.programadoresdejogos.com/trab_academicos/andre_kishimoto.pdf>, acessado em novembro de 2008.

KRAMER, A. e Crawford, B., "**The Amber project**", APM.1686.00.1, ANSA Phase III (draft) Request for Comments, 1996.

KRUEGER, C. W. **Software Reuse**, ACM Computing Surveys, Vol. 24, No. 02, June, 1992, pp. 131-183.

LAMOTHE, André. **Tricks of the 3D Game Programming Gurus:** Advanced 3D Graphics and Rasterization. Indianapolis: Sams Publishing, 2003. 1728 p.

LEWIS, Michael; JACOBSON, Jeffrey. Game Engines In Scientific Research. **Communications Of The Acm**, New York, v. 45, n. 1, p.27-31, jan. 2002.

MADEIRA, Charles Andryé. **Forge V8: um Framework para o Desenvolvimento de Jogos de Computador e Aplicações Multimídia**. Dissertação (Mestrado em Ciências da Computação) - Universidade Federal de Pernambuco, julho, 2001. Disponível em <http://www.poleia.lip6.fr/~madeira/publications/MadeiraMasterThesis2001.pdf>. Acesso em 03 nov. de 2008.

MEIRELES, Luis Otoni Ribeiro; TIMM Maria Isabel; ZARO, Milton Antonio. **Modificações em Jogos Digitais e seu uso potencial como Tecnologia Educacional para o ensino de Engenharia**. Revista Renote V. 4 Nº 1, Julho, 2006. Disponível em <http://www.cinted.ufrgs.br/renote/jul2006/artigosrenote/a36_21203.pdf>. Acesso em 03 nov. de 2008.

MEYER, B. **Object-oriented software construction**. Englewood Cliffs: Prentice Hall, 1988.

MILLINGTON, Ian. **Artificial Intelligence for Games.** San Francisco: Elsevier Inc., 2006. 856 p. (The Morgan Kaufmann Series in Interactive 3D Technology).

MILLINGTON, Ian. **Game Physics Engine Development.** San Francisco: Elsevier Inc., 2007. 455 p. (The Morgan Kaufmann Series in Interactive 3D Technology).

NEIGHBORS, James M., Draco: A Method for Engineering Reusable Softwar e Systems, Chapter 12 of Software Reusability, Volume 1: **Concepts and Models, Bigger staff**, T. J, and Perlis, A.J., eds., ACM Press Frontier Series, pp. 295-319, Addison-We sley, 1989.

OLIVEIRA, Leonardo de Lima. **Um ambiente de animação dinâmica de corpos rígidos.** 2006. 122 f. Dissertação (Mestrado) - Curso de Ciência da Computação, Departamento de Computação e Estatística, Universidade Federal de Mato Grosso do Sul, Campo Grande, 2006.

PALMER, grant. **Physics for Game Programmers. Apress**, Berkeley, CA - Estados Unidos, 2005.

PINTO, S. C. C. S. (2000): **Composição em WebFrameworks, tese de doutorado**, Departamento de Informática PUC-Rio.

PREE, W. **Design patterns for object oriented software development**. Reading: Addison-Wesley, 1995.

PRESSMAN, Roger S.. **Engenharia de Software.** 6. ed. São Paulo: Mcgraw-hill, 2006. 720 p.

SCHWAB, Brian. **AI Game Engine Programming.** Hingham: Charles River Media, 2004. 593 p. (Game Development Series).

SIEBRA, C.; RAMALHO, G.; Frery A. **Uma Arquitetura para Suporte de Atores Sintéticos em Ambientes Virtuais** - Uma Aplicação em Jogos de Estratégia. Dissertação de Mestrado - UFPE, 2000.

SILVA, Ricardo P. e, PRICE, R. T. A busca de generalidade, flexibilidade e extensibilidade no processo de desenvolvimento de frameworks orientados a objetos. In: **Proceedings of Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software** (IDEAS'98). Torres: apr. 1998. v.2, p.298-309.

SOMMERVILLE, Ian. **Engenharia de software.** 6. ed. São Paulo: Addison-wesley, 2003. 592 p.

SOMMERVILLE, Ian. **Software Engineering.** 8. ed. Addison-Wesley, 2007. 840 p.

SZYPERSKI, Clemens (Ed.). **Component Software:** Beyond Object-Oriented Programming. 2. ed. São Paulo: Addison-wesley, 2002. 589 p. (Component Software Series).

WEINREICH Rainer; SAMETINGER, Johannes. Component Models and Component Services: Concepts and Principles. In: HEINEMAN, George; COUNCILL, William T. **Component-Based Software Engineering**: putting the pieces together. Ma: Addison-wesley, 2001. Cap. 3, p. 33-48.

YOURDON, E. **Projetos virtualmente impossíveis: Guia Completo do Desenvolvedor de Software para Sobreviver as Projetos Virtualmente Impossíveis**. São Paulo: MAKRON Books, 1999.

ZAMITH, Marcelo P. **Uma Arquitetura de Distribuição Dinâmica de Tarefas Genéricas entre CPU e GPU em Jogos Digitais**. Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2007.

ZERBST, Stefan; DÜVEL, Oliver. **3D Game Engine Programming.** Boston: Thomson Course Technology, 2004. 896 p. (Game Development Series).

APÊNDICE A – Artigo científico: Reuso, Extensibilidade e Simulação de Física para Jogos Digitais

Reuso, Extensibilidade e Simulação de Física para Jogos Digitais

Daniel M. Natal[1], Leila Laís Gonçalves[1]

[1]Curso de Ciência da Computação - Unidade Acadêmica de Ciências, Engenharias e Tecnologias - Universidade do Extremo Sul Catarinense (UNESC)
Caixa Postal 3.167 – 88.806-000 – Criciúma – SC – Brasil

danielmn1986@hotmail.com, llg@unesc.net

Abstract. *With the evolution of realism in digital games, their development process became more complex and longer. The reuse of software in the development of digital games comes to make things easier and also providing more produtivity, lower costs and efforts. This study aims to evaluate the physical simulation, reuse and extensibility of some of currently available engines (Bullet, ODE e PhysX), for this, a detailed study of these items was done, so that tests, comparisons and evaluations among these engines. With the analysis of the results could be noticed that no one of the engines is better in any evaluated aspects, however, Bullet can be pointed out as the better open source choice, in view of its results and features*

Resumo. *Com a evolução do realismo dos jogos digitais, seu processo de desenvolvimento se tornou mais complexo e longo. O reuso de software no desenvolvimento de jogos tem como objetivo facilitar este processo, proporcionando também maior produtividade, redução dos custos e esforços. Este trabalho visa a avaliação da simulação física, reuso e extensibilidade dos motores Bullet, ODE e PhysX, para isso foi feito um estudo detalhado destes itens, para então, poder realizar testes, comparações e avaliações entres os motores. Pôde-se notar que nenhum dos motores é melhor em todos os aspectos avaliados, porém, pode-se indicar Bullet como a melhor alternativa de código livre, tendo em vista seus resultados e funcionalidades.*

1. Introdução

Com o crescente desenvolvimento de jogos digitais atualmente, aumenta também a necessidade por agilidade, produtividade e qualidade no processo de desenvolvimento, para atender estas necessidades emprega-se o uso de abordagens da engenharia de *software* no desenvolvimento de jogos, como o reuso e extensibilidade.

Na área dos jogos digitais o reuso é mais encontrado em forma de motores, que fornecem a abstração de *hardware*, plataforma e cálculos necessários para um determinado objetivo, como o gerenciamento gráfico, simulação física, gerenciamento de IA, etc., fazendo com que o desenvolvedor necessite do conhecimento apenas das interfaces dos motores e do domínio da aplicação para desenvolver um jogo. Além de facilitar o desenvolvimento, os motores proporcionam maior produtividade, redução dos custos e esforços durante o desenvolvimento e também maior qualidade no resultado final. O motor de simulação física é um dos mais importantes, pois é ele que prove o comportamento realístico dos objetos através das funcionalidades de detecção de colisão e dinâmica de corpos.

A falta de formalização da arquitetura de jogos digitais é uma das barreiras mais significativas para o reuso de componentes prejudicando também a portabilidade de jogos para diferentes ambientes. Esforços no sentido de buscar uma arquitetura padrão, bem como associar aos motores técnicas de engenharia de software, são discutidos por Falstein (1997),

Joselli (2007), Madeira, Ramalho e Ferraz (2001), Siebra, Ramalho e Frery (2000), Zamith (2007), entre outros autores com o objetivo de prover níveis satisfatórios de desempenho, de robustez, de modularidade, de abstração, de reusabilidade de código, de padronização, etc.

Neste contexto o objetivo deste trabalho é avaliar os motores de física para jogos digitais visando o reuso, extensibilidade e simulação de física no contexto de jogos digitais. Para isso será necessário contextualizar o reuso e extensibilidade no desenvolvimento de aplicações; compreender os motores de jogos, submotores de física e seus funcionamentos; identificar situações e possibilidades de reuso e extensibilidade nos motores de física; realizar testes, validações e comparações entre submotores de física envolvendo simulação de física, reuso e extensibilidade.

2. Reuso e Extensibilidade no Desenvolvimento de Aplicações

Diante do aumento da demanda e da complexidade no desenvolvimento de *software* e de exigências como produtividade, qualidade, redução de custos e esforços, novas perspectivas e esforços vêm sendo aplicados buscando atender estes quesitos. Pesquisas em Engenharia de *Software* visam buscar diferentes abordagens para melhorar a qualidade dos artefatos de *software* e reduzir o tempo e o esforço necessários para produzi-los (FAIRLEY, 1986).

A reutilização vem sendo indicada como ponto chave para dar suporte a essas, e ainda outras, necessidades e exigências de desenvolvimento de *software* sendo uma tendência crescente neste processo. Dentre os objetivos da reutilização estão aumento da produtividade, facilitação na implantação e integração de *softwares* ou processos, visto que os *softwares* estão cada vez maiores e mais complexos (CRNKOVIC; LARSSON, 2002, tradução nossa).

Um artefato reutilizável é uma parte do trabalho que pode ser utilizado em mais de um projeto incluindo código-fonte, código compilado, casos de teste, modelos, interfaces para usuário, planos e estratégias, entre outros (D'SOUZA; WILLS, 1999).

A granularidade do artefato de *software* é um fator relevante no reuso. A reutilização de rotinas (por exemplo, uso de bibliotecas de funções) corresponde a um nível de granularidade inferior ao reuso de módulo que corresponde ao reuso de classes em orientação a objetos (MEYER, 1988).

A abstração é a característica que possibilita ao desenvolvedor subtrair detalhes do artefato relacionado à implementação do código e visualizar a parte abstrata que contém as informações mais conceituais necessárias à reutilização (KRUEGER, 1992).

Dentre as abordagens que sugerem o reuso de forma extensiva citam-se o desenvolvimento baseado em objetos, desenvolvimento baseado em componentes (CBD – *Component-Based Development)* e a abordagem de *frameworks*.

No desenvolvimento baseado em componentes a reutilização pode se dar no nível de código, em específico o reuso das classes, por meio da composição, reuso de classes disponíveis em bibliotecas para gerar novos objetos da implementação; herança, criação de novas classes herdando as características de uma outra classe; polimorfismo, possibilidade de um objeto assumir diferentes formas a partir da implementação diferenciada de uma operação em classes diferentes (MEYER, 1988).

Sommerville (2003) apresenta alguns motivos pelos quais a orientação a objetos não atingiu seu potencial máximo apontando a granularidade dos objetos, o detalhamento e especificidade das classes de objetos como obstáculos ao reuso extensivo. Muitas vezes se torna necessário o conhecimento aprofundado das classes e a disponibilização de seu código-fonte para viabilizar sua utilização.

Deste e outros problemas da orientação a objetos quanto ao reuso, surge o desenvolvimento baseado em componentes, que vem com a intenção de restabelecer o conceito de reuso, onde os componentes são desenvolvidos e preparados especificamente para a construção de *softwares* (CRNKOVIC; LARSSON, 2002, tradução nossa).

Um componente pode ser definido como uma grande unidade de software, que pode ser integrado com outros componentes a fim de criar um sistema de software, ou seja, o foco é deslocado da programação de software para a composição de software (CLEMENTS, 1995, tradução nossa). Componentes são mais abstratos que classes e têm alto nível de granularidade, o que dispensa a necessidade de conhecimento detalhado de suas implementações, tornando mais simples e eficaz sua reutilização (SOMMERVILLE, 2007).

Um *framework* é um conjunto de recursos e conceitos, que juntos, provem funcionalidade comuns a um determinado domínio de aplicação para desenvolvimento de novos softwares (BUSCHMANN et al, 1996). Os *frameworks* podem ser classificados em dois grupos principais: *frameworks* de aplicação orientados a objetos, constituído por classes de objetos; e *framework* de componentes, constituído de componentes reutilizáveis (FAYAD et al, 1999; SZYPERSKI, 2002, tradução nossa).

Outros conceitos interessantes no reuso, são a extensibilidade e flexibilidade. A flexibilidade se refere à possibilidade de alterar funcionalidades do artefato e a extensibilidade permitem ampliar suas funcionalidade, ambas sem conseqüências imprevistas sobre o conjunto da estrutura (FAYAD & CLINE, 1996).

No caso do desenvolvimento baseado em objetos, as modificações e extensões podem ocorrer diretamente em nível de código, por meio de especializações (herança, polimorfismo, combinação, hot spots), ou ainda customização (interfaces). Enquanto no desenvolvimento baseado em componentes são feitas por meio das interfaces ou integração entre componentes. (BELLUR, 1998; CRNKOVIC; LARSSON, 2002, tradução nossa).

3. Motores de Jogos Digitais

Um jogo digital (ou videojogo ou jogo eletrônicos) é uma expressão genérica que se refere a jogos eletrônicos desenvolvidos para serem jogados num computador, num console ou outro dispositivo tecnológico, envolvendo interação entre ser humano e computador, recorrendo ao uso de tecnologia (GEE, 2003).

De forma básica um jogo é composto por uma aplicação que contêm a lógica do jogo e seus objetivos, algum tipo de interface para que o usuário consiga interagir com o jogo, e o motor de jogo, que é responsável pelo sistema de controle, que controla as reações do jogo de acordo com as ações do usuário, resultando em um novo estado de jogo (LEWIS & JACOBSON, 2002, tradução nossa).

Contextualizando com engenharia de software, os motores de jogos são *frameworks* que funcionam como uma estrutura que dará suporte ao desenvolvimento de diferentes aplicações por meio do reuso de bibliotecas de códigos, linguagens de *script* e outros *softwares* que dêem à sustentação necessária para a elaboração e funcionamento da aplicação.

A estrutura pré-implementada (*framework*) do motor é responsável por controlar todos os aspectos independentes de um jogo como: gráfico, áudio, inteligência artificial, física, rede, controle, entre outros. O objetivo na reutilização de motores é facilitar o desenvolvimento de jogos digitais, assim aumentando a produtividade e também a qualidade no desenvolvimento e produto final (BISHOP et al, 1998, tradução nossa).

O termo “motor” é utilizado para se referir ao motor de jogo num todo, ele contém vários “submotores”, que controlam cada um destes aspectos do jogo, também são chamados de motor gráfico, motor de áudio, motor de inteligência artificial, motor de física, etc. (ZERBST; DÜVEL, 2004, tradução nossa).

O submotor de física, também conhecido como motor de física ou *physics engine* é um componente de *software* responsável pela simulação dos comportamentos físicos de um jogo considerando variáveis como massa, atrito, velocidade, forças e volume para definir novas posições, velocidades e transformações dos diversos objetos de uma cena (GOMES, 2007; MILLINGTON, 2007; WOULFE, 2005).

Os motores de física aplicam em sua implementação as leis da física, de forma matematicamente exata ou aproximada, com o objetivo aumentar o realismo dos mesmos sendo responsável por todo o comportamento físico dos elementos do jogo. Por meio da simulação cinemática pode-se reproduzir a movimentação de carros, aviões ou outros objetos; por meio da simulação da dinâmica podem-se reproduzir efeitos como gravidade, atrito, aceleração, desaceleração, colisão; a simulação da aplicação de forças em corpos rígidos ou deformáveis e também as reações as colisões entre os corpos; a simulação de efeitos naturais como água, fogo, vento e explosões (BITTENCOURT; OSÓRIO, 2006).

As funcionalidades pertinentes ao motor de física são a dinâmica de corpos, rígidos ou flexíveis, responsável pela simulação dos movimentos dos elementos do jogo; a detecção de colisão, responsável pela identificação dos elementos que estão colidindo em determinado momento; a dinâmica de fluidos, que visa simular o comportamento físico de líquidos e gasosos, como água e ar; e a simulação de outros objetos especiais, como roupas, cabelo, personagens articulados, veículos.

A funcionalidade mais aprofundada neste trabalho foi a detecção de colisão, pois não seria possível abordas todas, devido as suas complexidades.

A detecção de colisão é responsável por verificar, durante o andamento do jogo, quais dos objetos do mundo, ou de um determinado conjunto de objetos, estão colidindo e gerar uma lista de pontos de contatos entre os objetos que estão em colisão, que posteriormente são utilizadas pela simulação dinâmica dos corpos, para resolução das colisões, ou na lógica do jogo (MILLINGTON, 2007, tradução nossa).

O processo de detecção de colisão ocorre em duas etapas, na primeira, chamada de *broadphase*, são selecionados os pares de objetos que provavelmente estão em contato, mas sem se preocupar se realmente estão em contato, o objetivo desta etapa é eliminar, de forma rápida, grande quantidade dos pares que não estão colidindo e gerar uma lista de possíveis colisões (ERICSSON, 2005, tradução nossa).

A segunda etapa, chamada de *narrowphase*, é que efetivamente realiza a detecção de colisão, são aplicados os algoritmos de detecção de colisão em cada par selecionado na etapa anterior, assim identificando quais objetos realmente estão colidindo e gerando a lista de contatos (ERICSSON, 2005, tradução nossa).

Os sistemas de detecção de colisão utilizam diferentes algoritmos de colisão dependendo da formas geométricas dos objetos envolvidos nos testes, possibilitando otimização para cada combinação de par de objetos.

Os sistemas de detecção de colisão não utilizam os mesmo objetos criados para a renderização visual, pois estes, geralmente têm geometria muito detalhista, o que tornaria a detecção de colisão muito custosa computacionalmente para serem executados em tempo real. A fim de diminuir estes custos, os motores de física utilizam objetos de geometria

simplificada, criados especialmente para a detecção de colisão, ainda assim, resultando em uma precisão de detecção de colisão satisfatória para visualização final (MILLINGTON, 2007, tradução nossa).

Estes objetos podem ser representados utilizando diferentes técnicas da computação gráfica, como por exemplo, malhas de triângulo; objetos primitivos ou uma combinação deles. Sendo que a técnica utilizada pode influenciar no desempenho da aplicação e na precisão da detecção de colisão (EBERLE, 2004, tradução nossa).

4. Testes, Validações e Comparações Entre Motores de Física

O objetivo do trabalho é a avaliação dos submotores de física Bullet, ODE e PhysX, abordando o reuso, extensibilidade e simulação de física. Foram aplicados testes, validações e comparações entre os motores seleicionados a fim de avaliar o desempenho, realismo dos resultados, abstração, modularidade, produtividade e integração de cada um dos motores.

2 4.1. Metodologia

Após o levantamento bibliográfico a cerca de reuso, extensibilidade e simulação de física em jogos digitais, a fim de compreender melhor o domínio do trabalho proposto, foram selecionados três entre alguns dos motores disponíveis atualmente.

Tendo os três motores selecionados, foi feito um estudo mais detalhado de cada um deles, buscando compreender melhor sua arquitetura, características, funcionalidades e interfaceamento, para então poder realizar testes, validações e comparações entre os motores selecionados a fim de avaliar questões como desempenho, realismo dos resultados, abstração, modularidade, produtividade e integração.

3 4.1.1 Seleção dos Motores de Física

Diante do grande número de motores de física disponíveis e não sendo possível avaliar todos, o trabalho foi limitado a avaliar três motores de física.

Como pré-requisito os motores deverão estar disponíveis para uso, não importando se para uso comercial ou não comercial, ou se possui código aberto, porém, deve ser selecionado no mínimo um motor de código livre.

Outros itens foram levados em consideração na seleção dos motores, como o seu uso em jogos comerciais, documentação, atualização, comunidade de desenvolvedores.

A princípio, os três motores selecionados foram PhysX, Havok e Bullet, porém por uma clausula nos termos de uso do Havok, que diz que a licença gratuita não pode ser utilizada para testes e *benchmarks* públicos, ele não pode ser usado. Os testes foram validados somente para os motores selecionados Bullet, ODE e PhysX.

4 4.1.2 Testes, Validações e Comparações Entre Motores de Física

Foram elaborados testes de desempenho de *broadphase*, colisões na *narrowphase*, utilização de *wrappers* e integração entre motores.

O teste de desempenho de *broadphase* visa comparar as técnicas de *broadphase* de cada motor. Para que os resultados não sofram influencia da simulação dinâmica dos corpos, pois cada motor apresenta simulações diferentes, foi elaborado um ambiente de estresse para testar a *broadphase*, ele é delimitado por seis paredes (caixas estáticas) onde um número determinado de corpos se movimentam em direções aleatórias e velocidade constante.

Os testes foram executados em diferentes situações, diferindo quanto à velocidade dos corpos e ao número de corpos no ambiente, a fim de observar como cada situação afeta o desempenho das técnicas testadas.

Também foi elaborado um teste de desempenho de *broadphase*, onde é observado o aumento de tempo de execução de broadphase, de acordo com o número de corpos colidindo.

Quanto ao teste de utilização de *wrappers*, tem o objetivo de verificar o funcionamento de *wrappers* para motores de física, visando o desempenho, realidade visual e identificação de problema. No teste foi utilizado o PAL, que suporta um grande número de motores de física. Foram testados com os motores Bullet, ODE e PhysX.

No teste de integração entre motores de física, foi integrado o sistema de detecção do Bullet ao sistema de dinâmica de corpos rígidos do ODE, a fim de verificar a possibilidade de integração, realismo visual e desempenho em relação a mesma simulação utilizando apenas o ODE para a detecção de colisão e dinâmica.

5 4.2 Resultados Obtidos

A partir da descrição detalhada de cada um dos motores, dos resultados dos testes realizados e também da implementação necessária para os testes foram observados os seguintes resutados.

Os três motores de física avaliados, Bullet, ODE e PhysX, têm como funcionalidades principais, a dinâmica de corpos rígidos e a detecção de colisão discreta, porém têm outras funcionalidades que extendem ou auxiliam as funções principais do motor, que também podem ser levadas em consideração na escolha de um motor de física para uso. Por exemplo, o Bullet possui detecção de colisão contínua, dinâmica de corpos flexíveis, integração com Blender e Maya, entre outros; o PhysX possui suporte a processamento da na GPU, simulação dinâmica para roupas, fluidos, corpos flexíveis e personagens, detecção de colisão contínua, ferramenta para pré-processamento de malhas de triângulo, etc.; quanto ao ODE, apesar de ser um dos primeiros motores de física de código aberto, não possui outras funcionalidades que se possa destacar das principais.

Os testes realizados focaram mais na detecção de colisão. Nos testes de *broadphase*, as técnicas mostraram resultados diferentes para situações diferentes, por exemplo, para ambientes onde os corpos não têm muito movimento, as técnicas SAP tiveram melhor resultado, porém para ambientes com muito movimento dos corpos, estavam entre os piores resultados. Para uso geral, as técnicas de *broadphase* que tiveram melhores resultados foram a DBVT do Bullet e QUADTREE do ODE. Os testes com as técnicas SAP do PhysX não mostraram os melhores resultados, porém, com PhysX, há a possibilidade de processar a *broadphase* na GPU, que melhora bastante seu desempenho.

Quanto às formas geométricas, todos os motores disponibilizam as primitivas básicas, porém o Bullet suporta outras, de uso mais especifico, como cone, cilindro e multi-esfera. Alem das primitivas, todos os motores suportam composição de formas geométricas e malhas de triângulos, então pode-se construir outras forma de colisão necessárias, porém deve-se saber que a detecção de colisão com malhas de triangulo é mais complexa e custosa computacionalmente.

Na *narrowphase*, o Bullet é o motor que suporta detecção de colisão entre mais combinações de formas geométricas, não suportando apenas a colisão entre *heightfield* e malha de triângulos. O PhysX não suporta a detecção de colisão entre malhas de triangulo, e o ODE entre objetos convexo contra caixa, cápsula, cilindro e malha de triangulo; e cilindro contra cápsula e cilindro. Nestes casos podem ser utilizadas outras primitivas ou composição de primitivas.

Outro ponto do processo de detecção de colisão a ser observado é a execução do da *narrowphase* e resolução dos contatos, o ODE obriga a implementação de como será executado os testes de sobreposição na *narrowphase* e resolução dos contatos por meio de uma função *callback*, isto pode ser bom para o caso de necessitar fazer customizações. No caso do Bullet e PhysX, isso é feito pelo próprio motor, porém disponibilizam formas para customizar isto caso necessário.

O desenvolvimento de aplicações utilizando *wrappers* para motores de física, ao invés utilizar diretamente o motor de física, pode ser proveitoso caso necessite alterar o motor que esta sendo utilizado sem necessidade de alteração de código fonte da aplicação, porém deve ser levado em consideração que a utilização de *wrappers* pode resultar em perda de desempenho e funcionalidades, uma vez que o *wrapper* precisa abstrair as especificações de cada um dos motores fornecendo apenas uma interface geral para todos os motores, por este motivo o *wrapper* necessita fazer um processamento adicional para comunicação com o motor de física, causando perda de desempenho.

A integração entre motores também é um tópico interessante, pode-se integrar partes de motores na aplicação, a fim de melhor desempenho em algum aspecto ou mesmo adição de funcionalidades. É possível fazer a integração de partes que tenham independência do resto do motor, dos motores avaliados, o Bullet permite a integração do sistema de detecção de colisão, o ODE permite a integração do sistema de detecção de colisão e dinâmica de corpos, já o PhysX não permite nenhuma integração parcial, pois seu sistema de detecção de colisão dependendo do sistema de dinâmica, e o de dinâmica depende da detecção de colisão.

A integração realizada nos testes, com o sistema de detecção de colisão do Bullet e dinâmica de corpos do ODE, não apresentou falhas de dinâmica, nem de detecção de colisão. Como a simulação executada foi simples, não mostrou nenhum ganho, porém, também não mostrou perda de desempenho, devido ao processamento adicional para a integração.

Quanto ao desenvolvimento dos motores, os três utilizam a linguagem C++, utilizando orientação a objetos. O Bullet é organizado por módulos, cada módulo gera um arquivo de distribuição em formato de biblioteca estática. Já o ODE, tem apenas um projeto, que pode gerar para distribuição uma biblioteca estática ou dinâmica. O PhysX é organizado em módulos e os arquivos de distribuição são bibliotecas dinâmicas, porém para a execução de uma aplicação que utiliza PhysX é necessário a instalação de um pacote de bibliotecas da fabricante, que contem todas as versões do PhyX.

A vantagem de ser utilizar bibliotecas dinâmicas é que o motor fica independente do aplicativo que o utiliza, assim pode ser atualizada, sem a necessidade de recompilação da aplicação; e também o tamanho do executável do aplicativo fica menor, pois não tem todo o código referente ao motor, que fica na biblioteca dinâmica.

Quanto a interface dos motores, o Bullet e PhysX disponibilizam as classes abstratas (.h) utilizada em seu desenvolvimento, com um detalhe que no PhysX os objetos não são instanciados da forma tradicional invocando o construtor, são disponibilizadas na interface, métodos que instanciam o objeto e retornam a aplicação. O ODE disponibiliza uma interface C com funções, que torna a utilização mais simples.

5. Conclusão

Os resultados dos testes, validações e comparações mostraram que nenhum dos motores é melhor em todos os aspectos, cada motor tem pontos fortes em alguns aspectos, porém em outros não mostram resultados tão bons. Ainda assim é possível indicar o Bullet como o melhor dos motores avaliados, tendo em vista seus resultados nas simulações, funcionalidades

além da dinâmica de corpos rígidos e detecção de colisão discreta, possibilidades de extensibilidade e ter código aberto.

Na possibilidade de aquisição de um motor de física comercial, o PhysX deve ser levado em consideração, diante de seus resultados, possibilidade de processamento na GPU, documentação e suporte. Um dos aspectos mais problemáticos em projetos de código aberto são a documentação e o suporte, a documentação geralmente não completa ou às vezes inexistente; e o suporte a ferramenta, geralmente prestado pela comunidade de desenvolvedores do projeto e utilizadores, pode ser demorado ou não atendido.

Por utilizarem abordagem de desenvolvimento baseado em objetos, o reuso não se da ao melhor nível, devido a baixa granularidade e abstração das classes, que faz necessário o conhecimento detalhado das funções de cada uma para poder reutilizá-la.

Quanto à extensibilidade, é possível, porém os fabricantes dos motores não identificam muitos pontos que podem ser extendidos, e todos são em nível de código. Notou-se também que no caso do PhysX, por ser código proprietário, apresenta menos pontos que podem ser extendidos.

Como trabalhos futuros, podem ser citados, a extensão das avaliações para outros aspectos não focados neste trabalho, como a dinâmica de corpos flexíveis, detecção de colisão contínua, dinâmica de fluidos; a integração de motores de física, como utilizando a detecção de colisão do Bullet e dinâmica do ODE, possibilidade comentada diversas vezes na comunidade Bullet e ODE; Estudo ou desenvolvimento de *wrappers* para motores de física; e também o desenvolvimento de um motor, ou extensão de um já existente, seguindo algum padrão de desenvolvimento baseado em componentes.

Referências

Bishop, Lars et al (1998), Designing a PC Game Engine. “Ieee Computer Graphics And Applications”, v. 18, n. 1, p.46-53.

Bittencourt, João Ricardo; Osório, Fernando Santos (2006), Motores de Jogos para Criação de Jogos Digitais - Gráficos, Áudio, Interface, Rede, Inteligência Artificial e Física. In: V Escola Regional de Informatica de Minas Gerais (ERI-MG 2006), Belo Horizonte. “Anais da V ERI-MG SBC”. Belo Horizonte: PUC Minas, v. 1, p. 1 – 36.

Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. & Stal, M (1996), “Pattern-Oriented Software Architectur A System of Patterns”, John Wiley & Sons.

Clements, Paul C. (1995), “From Subroutines to Subsystems”: Component Based Software Development. The American Programmer, v. 8, n. 11.

Crnkovic, Ivica; Larsson, Magnus (Ed.) (2002), “Building Reliable Component-Based Software Systems”. Norwood: Artech House, 413 p.

D’Souza, D.; Wills, A. (1999), “Objects, Components and Frameworks with UML”: The Catalysis Approach. [S.l.]: Addison-Wesley, Object Technology Series.

Fairley, R. (1986), “Software Engineering concepts”. [S.l.]: McGraw-Hill.

Falstein, N. (1997), Game Design: “How Platform Choices Dictates Design”, Game Developer.

Fayad, Mohamed C.; Schmidt, Douglas C.; Johnson, Ralph E (1999), “Building Appiication Frameworks” - Object-Oriented foundations of frameworks design. Estados Unidos: Wiley.

Gee, James Paul (2003), “What video games have to teach us about learning and literacy”. New York, Palgrave Macmillan.

Joselli, Mark Eirik Scortegagna (2007), “Uma Arquitetura de Motor de Física para Games 3D com Processamento Híbrido entre CPU e GPU e Distribuição Dinâmina de Carga”. 63 f. Dissertação (Mestrado) - Curso de Computação, Universidade Federal Fluminense, Niterói.

Krueger, C. W (1992), “Software Reuse”, ACM Computing Surveys, Vol. 24, No. 02, June, pp. 131-183.

Lewis, Michael; Jacobson, Jeffrey (2002), Game Engines In Scientific Research. “Communications Of The Acm”, New York, v. 45, n. 1, p.27-31.

Madeira, Charles Andryé (2008), “Forge V8: um Framework para o Desenvolvimento de Jogos de Computador e Aplicações Multimídia”. Dissertação (Mestrado em Ciências da Computação) - Universidade Federal de Pernambuco. Disponível em <http://www.poleia.lip6.fr/~madeira/publications/MadeiraMasterThesis2001.pdf>. Acesso em 03 nov. de 2008.

Meyer, B (1988), “Object-oriented software construction”. Englewood Cliffs: Prentice Hall.

Millington, Ian (2007), “Game Physics Engine Development”. San Francisco: Elsevier Inc., 455 p. (The Morgan Kaufmann Series in Interactive 3D Technology).

Siebra, C.; Ramalho, G.; Frery A (2000), “Uma Arquitetura para Suporte de Atores Sintéticos em Ambientes Virtuais” - Uma Aplicação em Jogos de Estratégia. Dissertação de Mestrado - UFPE.

Sommerville, Ian (2007), “Software Engineering”. 8. ed. Addison-Wesley. 840 p.

Szyperski, Clemens (Ed.) (2002), “Component Software”: Beyond Object-Oriented Programming. São Paulo: Addison-wesley, 589 p., Component Software Series.

Zamith, Marcelo P. (2007), “Uma Arquitetura de Distribuição Dinâmica de Tarefas Genéricas entre CPU e GPU em Jogos Digitais”. Dissertação (Mestrado em Computação) - Universidade Federal Fluminense.

Zerbst, Stefan; Düvel, Oliver (2004), “3D Game Engine Programming”. Boston: Thomson Course Technology, 896 p., Game Development Series.