

Jeferson Vieira Ramos

*Técnicas de otimização para ambientes dinâmicos de
simulação*

Universidade Federal de Santa Catarina

Florianópolis, março de 2009

Jeferson Vieira Ramos

*Técnicas de otimização para ambientes dinâmicos de
simulação*

Trabalho de conclusão de curso
Semestre 2009/1

Orientador:

Dr. rer.nat. Aldo von Wangenheim

Co-orientador:

Diego D. B. Carvalho

Universidade Federal de Santa Catarina

Florianópolis, março de 2009

Trabalho de conclusão de curso defendido na Universidade Federal de Santa Catarina. Sob o título de “*Técnicas de otimização para ambientes dinâmicos de simulação*”, defendido por Jefer-son Vieira Ramos e aprovada em Florianópolis, março de 2009 pela banca examinadora constituída pelos professores e convidados:

Dr. rer.nat. Aldo von Wangenheim
Orientador

Diego D. B. Carvalho
Co-Orientador

Tiago de Holanda Cunha Nobrega
Universidade Federal de Santa Catarina

*Dedico este trabalho à minha família e a minha
namorada por seu apoio incondicional.
Em especial, agradeço aos meus pais,
Diomar e Marcia pelo seu incentivo
e cooperação.
Através dessas pessoas foi possível seguir em
frente e chegar ao fim dessa etapa
que esta sendo vencida.*

Resumo

Em um ambiente de simulação é fundamental que o usuário final tenha um resultado factível com o que ele espera. Por isso é muito importante que o dispositivo gráfico possa de forma rápida e confiável mostrar esses resultados. Através do estudo dos dispositivos gráficos é possível obter um bom entendimento de suas características e assim tomar boas decisões em um projeto de computação gráfica para a simulação de um ambiente dinâmico. Além disso é imprescindível analisar como tirar proveito de uma técnica que pode vir a prejudicar o desempenho do sistema, se efetuada de maneira errônea, como a colisão de objetos. O objetivo desse trabalho de conclusão de curso é estudar métodos de otimização de renderização, utilizando recursos oferecidos pelo OpenGL, e fornecer meios eficientes para o uso de técnicas de tratamento de colisão para objetos tridimensionais. Através desses estudos obter alta performance para um ambiente dinâmico de simulação. Para isso são, primeiramente, analisadas as vantagens e desvantagens de cada técnica para posteriormente obter conclusões que possam melhorar a utilização dos recursos citados no ambiente virtual.

Palavras-chave: Vertex Buffer Object (VBO), dispositivo gráfico, OpenGL, colisão, otimização, ambiente dinâmico de simulação.

Sumário

Lista de Figuras

Lista de Tabelas

1	Introdução	p. 11
1.1	Objetivo geral	p. 15
1.2	Objetivo específico	p. 15
2	OpenGL	p. 16
2.1	Histórico	p. 16
2.2	O que é OpenGL?	p. 17
2.3	Características	p. 18
2.4	Transmissão de informações geométricas	p. 20
2.4.1	Modo Imediato	p. 20
2.4.2	Display List	p. 21
2.4.3	Vertex Array	p. 22
2.4.4	Interleaved vertex array	p. 24
2.4.5	Vertex Buffer Object	p. 24
3	Vertex Buffer Object	p. 25
4	Sistemas de Computação Gráfica	p. 29

4.1	Aceleração	p. 29
5	Teste Gráfico	p. 33
5.1	ATI	p. 34
5.2	NVIDIA	p. 36
6	Detecção de Colisão	p. 37
6.1	Bounding Volume Hierarchies	p. 39
6.1.1	Axis Aligned Bounding Box	p. 42
6.1.2	Bounding Sphere	p. 44
6.1.3	Oriented Bounding Box	p. 45
6.2	Teste de Bounding Volume	p. 45
6.3	Tabelas comparativas	p. 47
6.4	Classificador de comportamento	p. 53
7	Conclusão	p. 57
	Referências Bibliográficas	p. 58

Lista de Figuras

2.1	Resultado do algoritmo 1	p. 18
2.2	Diagrama de blocos, simplificado, que mostra as fases do <i>pipeline</i> OpenGL (KUEHNE et al., 2005)	p. 19
2.3	Pipeline OpenGL mostrando os estados em que se encontra a Display Lista, assim como o processo que segue após sua criação (AHN, 2005)	p. 21
2.4	Renderização com VAR (MARTZ, 2006)	p. 23
3.1	Gráfico ilustrativo da tabela 3.1	p. 27
3.2	Figura que ilustra o modelo cliente servidor de VBO e como modos da VBO são usados. (NVIDIA, 2003)	p. 28
4.1	Estágios OpenGL até que a imagem final seja gerada e mostrada no monitor. (CASTELLÓ; RAMOS; CHOVER, 2005)	p. 30
5.1	Tabela que mostra as vendas de placas de vídeo distintas (PEDDIE, 2007).	p. 33
5.2	Essas figuras mostram a renderização do macaco fornecido pelo Blender. (a) 507 vértices usando modelo de arames. (b) 507 vértices sem usar modelo de arames. (c) 124242 vértices usando modelo de arames. (d) 124242 vértices usando modelo de arames. Todos criados através do aplicativo desenvolvido para a renderização.	p. 34
6.1	Objeto ou modelo, representado como um modelo de arames	p. 38
6.2	Figura que exemplifica o nível hierárquico utilizando esferas.	p. 40
6.3	Figura que exemplifica o nível hierárquico utilizando paralelepípedos alinhados.	p. 41
6.4	Figura que exemplifica o nível hierárquico utilizando paralelepípedos não alinhados.	p. 41

- 6.5 Representação de diferentes tipos de *bounding volumes*. A figura mostra as diferentes características. Quanto mais simples a forma (como caixa ou esfera), mais rápido é o teste de intersecção. Quanto mais complexa é a forma, menos testes de intersecção é necessário fazer. Formas mais elaboradas, também, se encaixam melhor nos objetos (GOTTSCHLICK, 2000). p. 42
- 6.6 Figura que demonstra como a má utilização de um AABB pode resultar em um encaixe imperfeito para figuras em uma posição não alinhada (*figura obtida através de um aplicativo desenvolvido para esse fim*). p. 43
- 6.7 Aqui é demonstrado como a *Bounding Sphere* pode ter um resultado ruim para objetos muito alongados. É importante perceber que independente da orientação do objeto a Sphere sempre terá o mesmo resultado. Isso é uma desvantagem, óbvia, se comparado ao AABB (*figura obtida através de um aplicativo desenvolvido para esse fim*). p. 44
- 6.8 Aqui é demonstrado como que o OBB pode, facilmente, se ajustar a uma figura geométrica, mesmo ela estando em diagonal. Ao contrário da Sphere e do AABB, o OBB consegue deixar um objetos inscrito a ele com menos áreas sobrando. Através dessa figura é possível perceber o por que do OBB conseguir efetuar a colisão com menos testes de intersecção (*figura obtida através de um aplicativo desenvolvido para esse fim*). p. 46
- 6.9 Aqui é mostrado cenas do aplicativo em execução. Na primeira figura é mostrado a colisão usando OBB. Na segunda figura é mostrado a colisão usando AABB. Na última é usado a *bounding sphere*. Pode-se perceber nas figuras partes amarelas onde os objetos estão se tocando. p. 48
- 6.10 Aqui é mostrada uma sequência de cenas do aplicativo desenvolvido em execução. Na figura é testada a colisão entre dois modelos que inicialmente estão parados e não se colidem, mas que com o passar do tempo ambos se aproximam através de translação. Como os modelos não rotacionam, o dispositivo desenvolvido modifica a estrutura de colisão para a caixa no momento da colisão automaticamente. Com o uso desse protótipo é possível utilizar mais de uma árvore hierárquica, visando a obtenção do tempo ótimo. p. 54
- 6.11 Aqui é mostrada passos do programa em execução utilizando os novos modelos. p. 55

Lista de Tabelas

- 3.1 Tabela (flag/modelo) registra o tempo, em segundos, do processamento. p. 26
- 4.1 Resultados obtidos com o modelo Cow, retirado dos estudos de Castelló. (CASTELLÓ; RAMOS; CHOVER, 2005) p. 31
- 5.1 Tabela comparativa com o resultado obtido a partir dos testes efetuados com o aplicativo funcionando em três diferentes modos de OpenGL: Modo imediato, DL e VBO para placa de vídeo ATI. p. 35
- 5.2 Tabela comparativa com o resultado obtido a partir dos testes efetuados com o aplicativo funcionando em três diferentes modos de OpenGL: Modo imediato, DL e VBO para placa de vídeo NVIDIA. p. 36
- 6.1 Fórmula que ilustra o cálculo do custo básico, para esfera e caixa orientada, da colisão em *Bounding Volume Hierarchies* (BRADSHAW; O'SULLIVAN, 2004) . . . p. 42
- 6.2 Tabela resultante dos testes efetuados com diferentes tipos de BVH, para três modelos criados com o auxílio do Blender (FOUNDATION, 2009). p. 49
- 6.3 Essa tabela mostra os testes efetuados com diferentes tipos de BVH, para 3 modelos criados com o auxílio do Blender (FOUNDATION, 2009). p. 50
- 6.4 Essa tabela demonstra a atuação das BVH para objetos alongados. p. 51
- 6.5 Aqui é mostrada a tabela de desempenho das atualizações efetuadas na árvore da BVH para diferentes comportamentos. p. 52
- 6.6 Tabela com o tempo de colisão para objetos que não estão colidindo. Essa tabela calcula o tempo que objetos levam para detectar que não estão em contato. p. 52
- 6.7 Tabela que mostra o tempo (em tempo por frame e em frame por segundo) da *bounding sphere*, AABB e do protótipo que atualiza a árvore em tempo de execução (genérico). Esse é o tempo para a execução do teste de intersecção e da atualização das árvores. p. 55

6.8 Tabela adquirida a partir dos teste efetuados em novos modelos. p. 56

1 *Introdução*

A representação gráfica em displays 2D é baseada em pixels, pontos que fazem com que imagens sejam sintetizadas visualmente num dispositivo gerador de imagens. Para o usuário a renderização é ansiosamente esperada quando se está produzindo uma cena. Há inúmeras dificuldades de se encontrar as configurações exatas para se obter uma renderização consideravelmente boa. A utilização dos recursos do dispositivo gerador de imagens nem sempre é algo trivial. Para não cometer erros e obter o máximo da placa gráfica é necessário um estudo mais aprofundado sobre suas funcionalidades e limitações.

Hoje é possível ver o processo de renderização em inúmeras aplicações. Desde um simples tocador de mídia à sofisticadas animações geradas por games ou filmes, cada um com suas características e técnicas.

Para o usuário de uma aplicação gráfica o importante é a imagem resultante, aquilo que pode ser visto na tela. O resultado, ou seja, aquilo que aparece no *display* acaba sendo uma representação 2D do objeto. Isso pode soar estranho uma vez que existe uma grande diferença visual entre objetos 2D e 3D. Para alcançar esse objetivo, transformar um objeto complexo 3D em um simples objeto 2D, é necessário que suas faces, vértices e arestas passem por sucessivas transformações matemáticas. Um conhecido método para essas transformações é a rasterização.

Rasterização - No andar final do pipeline de visualização, as primitivas gráficas são enviadas ao dispositivo físico onde são afixadas, depois de realizada a transformação das suas coordenadas para as coordenadas próprias do dispositivo. A afixação das primitivas em unidades do tipo vetorial não apresenta quaisquer problemas, pelo menos no caso de primitivas simples suportadas por essas unidades. Porém, no caso de dispositivos gráficos do tipo de quadrícula¹, há ainda que converter tais primitivas nas quadrículas dos dispositivos, em operações denominadas de rasterização ou, como também são conhecidas, conversão por varrimento. O objetivo desta operação é determinar quais os pixels que representarão as primitivas gráficas.(LOPES, 2008)

¹pixel

A computação gráfica até meados dos anos 80 era um campo muito pequeno e especializado, isso se devia em grande parte às limitações tecnológicas dos hardware. Com crescimento da computação gráfica, esta veio a incluir a criação, armazenamento e manipulação de modelos computacionais. Esses modelos surgiram de diversos campos como, física, matemática, engenharia, arquitetura e até mesmo em meteorologia.

Hoje até mesmo pessoas que não trabalham diretamente com computação gráfica a vê presente nas suas vidas em lugares como na televisão e em efeitos especiais no cinema.

Desde o desenvolvimento da fotografia e da televisão a computação gráfica tem se tornado uma grande invenção de destaque. Com a utilização da computação gráfica muitas vantagens tem sido conquistadas ao se utilizar um computador. Não só imagens concretas, do mundo real, são criadas, mas também imagens abstratas, como objetos sintetizados, e superfícies matemáticas em quatro dimensões (FOLEY et al., 1995).

Na representação de gráficos 3D, a renderização pode ser feita lentamente, em modelos pré renderizados ou em tempo real, utilizando recursos de aceleração de placas gráficas. Devido à impressionantes evoluções nas placas gráficas, estas que outrora eram utilizadas apenas para enviar sinais para o monitor hoje apresentam capacidade de processamento independente. Através dessas evoluções elas têm a capacidade de efetuar operações em velocidades superiores, incluindo cálculos de renderização, à unidade central de processamento (CPU) (HARADA; KOSHIZUKA; KAWAGUCHI, 2007).

A utilização da unidade gráfica de processamento (GPU) para propósito geral possui muitas vantagens (LUEBKE et al., 2004):

- Performance: A performance de GPUs modernas são muito superiores as CPUs atuais para várias aplicações .
 - Pentium 4, 3GHz: 6 GFLOPS, pico de 5.96 GB/seg.
 - GeForce FX 5900: 20 GFLOPS, pico de 25.3 GB/seg.
- Balanceamento de carga (“*load balancing*”): Um dos mais importantes meios para utilizar eficientemente o poder máximo de processamento de sistemas de grande escala. Ele consiste, basicamente, em utilizar a informação de localidade especificada pelo aplicativo. Nesse método, o software especifica a localidade da computação e forma uma heurística que permite que o poder computacional seja melhor distribuído (TAKEDA et al., 1988).

- Evolução: As placas de vídeo tem evoluído, na sua capacidade de processamento, muito mais rapidamente que as CPUs.

Infelizmente a unidade gráfica de processamento (GPU) ainda está longe de ter todas as funcionalidades e facilidades do uso da CPU. As principais dificuldade de uso são (LUEBKE et al., 2004):

- A depuração é complicada. Não existe comando de impressão em console e provavelmente nunca existirá. Por outro lado, ferramentas estão sendo desenvolvidas com a capacidade de melhorar a depuração (FERNANDES, 2008).
- As GPU são inerentemente paralelas. Isso pode tornar programas triviais mais complexos.
- Evoluem rapidamente (mesmo em características básicas).
- Não é possível, simplesmente, portar o código da CPU para a GPU.

O fato desse tipo de arquitetura estar intimamente relacionado ao desenvolvimento de games faz com que elas visem satisfazer esse tipo de necessidades inerentes a essa vertente. No entanto, atualmente a utilização da GPU pode ser facilitada graças ao aparecimento de algumas API's para placas gráficas específicas, como: CUDA (*Compute Unified device Architecture*) da NVIDIA e CTM (*Close to Metal*) da ATI (CARAPETO; FRANCISCO; MAURÍCIO, 2007).

CUDA - O CUDA é uma nova arquitetura de hardware e software que permite tratar de computação que utiliza paralelismo de dados, sem ter de a transformar para funcionar numa API gráfica. Funciona sobre o hardware gráfico da série 8 ou superior da GeForce da NVIDIA (CARAPETO; FRANCISCO; MAURÍCIO, 2007).

CTM - O CTM foi desenhado para expor as capacidades dos processadores de ponto flutuante existentes no hardware gráfico da ATI. É controlado por um conjunto de comandos, residentes em memória, que inserem parâmetros, especificam endereços de memória e formatos, invalidam e executam a limpeza de caches ou iniciam programas (CARAPETO; FRANCISCO; MAURÍCIO, 2007).

Com CUDA, por exemplo, não é necessário usar uma linguagem incomum, como de “*shader*” ou GLSL. É possível implementar um aplicativo utilizando linguagens de mais alto nível e mais familiar, como C, sem a necessidade de pensar muito na arquitetura da GPU. Posteriormente esses aplicativos podem ser executados em alta performance através dos dispositivos gráficos (GeForce 8 ou superior) compatíveis com este (NVIDIA, 2009).

Infelizmente a utilização de CUDA ou CTM restringe a aplicação a rodar somente em um tipo de hardware, NVIDIA ou ATI, respectivamente.

O presente processo de pesquisa desenvolvido discute técnicas, utilizando a unidade de processamento gráfico, para atingir alta performance na renderização, de objetos estáticos e dinâmicos.

Uma vez que tenha ficado claro as vantagens de usar a GPU, ao invés da CPU para o desenvolvimento de aplicações gráficas, é inimaginável falar de renderização sem mencionar os mecanismos citados anteriormente. Esse estudo tem como objetivo identificar procedimentos que possibilitem a utilização da placa gráfica com eficiência. Por não se tratar de um processo trivial, o uso indiscriminado pode levar a quedas de desempenho e até mesmo a erros de lógica no software. Por esse motivo é essencial identificar, a partir da literatura, técnicas sofisticadas para obtenção de maior desempenho. Para atingir tal objetivo foi utilizada a API OpenGL, para interfacear o hardware gráfico. Essa escolha se deve por ser uma biblioteca de rotina gráficas livre, extremamente portátil e rápida.

Embora a otimização da renderização seja um importante fator para a construção de um ambiente interativo, um ambiente virtual, ou seja, aquele que tem como objetivo a simulação do mundo onde vivemos, contém outros fatores que podem ser analisados para a obtenção de performance. Um desses fatores, talvez o principal, seja o tratamento da interação entre os objetos que estão presentes no espaço. Essa interação é muito importante em diversas áreas como robótica, simulação de fluidos, e até mesmo em softwares para o auxílio à medicina (XIE; YANG; ZHU, 2006). Para evitar que objetos em contato consigam se transpassar é necessário ter um sistema robusto de tratamento de colisão.

Muitos são os modos como a literatura trata o sistema de colisão, desde o uso de Octree até o uso de hierarquia de volumes. As hierarquias de volume, conhecidas como “*Bounding Volume Hierarchies*” (BVH) foram tratadas com atenção especial nesse trabalho, pois são conhecidas como as técnicas mais rápidas para o tratamento de colisões para objetos genéricos (SU, 2007). Através de uma estrutura de dados contendo volumes especiais é realizada a intersecção de todos os polígonos de um modelo de forma rápida e concisa.

O caminho que o pesquisador percorre para atender os objetivos de pesquisa no presente projeto envolve a utilização de artigos científicos, impressos e online, capítulo de livros, testes, dentre outros.

1.1 Objetivo geral

O presente projeto tem por objetivo a execução de testes para técnicas de renderização, vislumbrando como objetivo estudar os distintos comportamentos, assim estabelecer uma metodologia para otimização de renderização de um ambiente dinâmico.

1.2 Objetivo específico

Especificamente esse documento tem por objetivo:

- Avaliar diferentes técnicas de renderização em objetos de diferentes naturezas, como: Estáticos, móveis entre outros, sempre buscando o maior desempenho possível.
- Analisar os dados obtidos nos testes buscando encontrar heurísticas que garantam que para cada comportamento exista uma técnica condizente com a situação.
- Aplicar essas heurísticas em um ambiente dinâmico, onde existam múltiplos objetos com diferentes comportamentos e que possam interagir.
- Criar um guia, estabelecendo diretrizes para que se possa fazer um levantamento de requisitos necessários para a renderização de múltiplos objetos.

2 *OpenGL*

OpenGL é uma API livre para a renderização de primitivas gráficas, implementada, por alguns desenvolvedores, diretamente no hardware gráfico. Essa interface que iniciou com aproximadamente 150 comandos distintos, atualmente podendo chegar a 250 comando que podem ser usados para a produção interativa de aplicações em três dimensões (OPENGL et al., 2005)

2.1 Histórico

Na década de 80 cada fabricante de hardware tinha seu próprio conjunto de instruções para processamento gráfico 2D e 3D. Construir aplicações para tais tecnologias era um problema. As aplicações demandavam muito esforço, e por serem proprietários, as empresas menores, com baixo capital, acabavam ficando com pouco espaço no mercado. Após a criação do padrão *Programmer's Hierarchical Graphics System* (PHIGS), que não obteve sucesso devido a sua complexidade, a *Silicon Graphics Inc.* (SGI), criou o padrão IRIS GL (sigla para *Integrated Raster Imaging System Graphics Library*). Considerado de fácil uso, a API tornou-se um padrão na indústria.

Mesmo com um sistema mais fácil no mercado algumas empresas grandes, como a Sun Microsystems, ainda utilizavam o padrão PHIGS. Para tentar conquistar essas empresas a SGI tornou sua API um padrão público. Como a IRIS continha muito software proprietário (impedindo-a de ser pública) e lidava com questões fora do escopo relacionado ao desenho 2D e 3D (gerenciamento de janelas, controle de teclado e mouse) foi criado o OpenGL.

OpenGL é um padrão mantido, a muito tempo, por grandes empresas como 3DLabs, ATI, Dell, Evans&Sutherland, HP, IBM, Intel, Matrox, nVidia, SUN e Silicon Graphics (OPENGL et al., 2005).

2.2 O que é OpenGL?

OpenGL, de acordo com os autores supracitados, é uma interface para auxiliar a utilização do hardware gráfico. Ela é modelada como uma máquina de estados, independente de hardware, ou seja, pode ser usada em diferentes plataformas. Em algumas implementações, OpenGL é modelada para funcionar mesmo se o computador que mostrará o gráfico não for o mesmo que processará o programa.

Sua interface consiste, hoje, de aproximadamente 250 diferentes comandos que são usados para especificar objetos e operações sobre esses objetos criando um aplicativo interativo tridimensional (SEGAL; AKELEY, 2008).

Como dito anteriormente, OpenGL funciona como uma máquina de estado, desse modo é possível colocá-la em um estado (ou modo) e então perpetuar esse efeito até que um novo estado seja criado.

Alg. 1 Código OpenGL que exemplifica o funcionamento da máquina de estados através do algoritmo. (OPENGL et al., 2005)

```

1 #include <whateverYouNeed.h>
2
3 main(){
4     InicializaAWindowPlease();
5
6     glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
7     glClear(GL_COLOR_BUFFER_BIT);
8     glColor3f(1.0f, 1.0f, 1.0f);
9     glOrtho(0.0f, 1.0f, 0.0f);
10    glBegin(GL_POLYGON);
11        glVertex3f(0.25f, 0.25f, 0.0f);
12        glVertex3f(0.75f, 0.25f, 0.0f);
13        glVertex3f(0.75f, 0.75f, 0.0f);
14        glVertex3f(0.25f, 0.75f, 0.0f);
15    glEnd();
16    glFlush();
17
18    UpdateTheWindowAndCheckForEvents();
19 }
```

OpenGL possui dois objetivos principais:

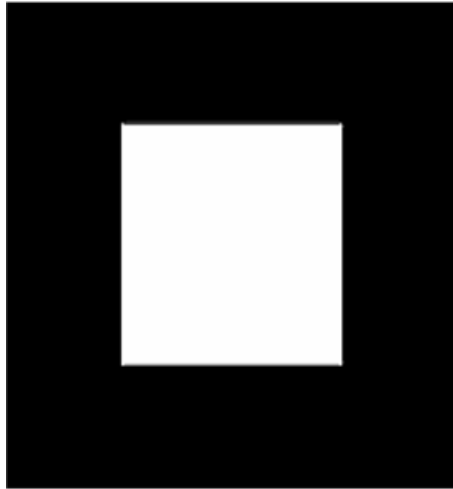


Figura 2.1: Resultado do algoritmo 1

- Esconder a complexidade de interfaceamento dos diferentes aceleradores 3D, fornecendo o programador com uma API simples e uniforme.
- Ocultar as diferentes capacidades das plataformas de hardware, fazendo com que todas as implementações suportem as ferramentas do OpenGL, mesmo que seja necessário emular.

OpenGL é historicamente influenciado pelo desenvolvimento de aceleradores gráficos, provendo uma base de funcionalidades nos mais comuns hardware (OPENGL et al., 2005):

- Pipeline de transformações e iluminação.
- Mapeamento de textura.

O algoritmo 1 mostra um código exemplo de OpenGL e seu resultado. Como é possível perceber, a partir do momento que o estado de cor é setado (no exemplo, como branco) todos os objetos que forem criados terão a cor branca até que uma nova cor seja inserida na máquina de estados.

2.3 Características

Algumas seções de uma aplicação podem vir a ser mais lentas que o resto do programa. Essas seções mais lentas podem provir de controle de arquivos, cálculo de formulas matemáticas complexas ou renderização de gráficos. Essas parte são conhecidas como “gargalos” da aplicação. Em

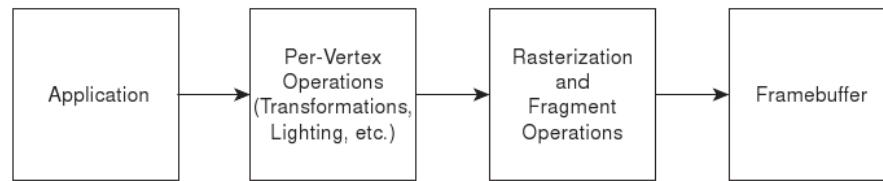


Figura 2.2: Diagrama de blocos, simplificado, que mostra as fases do *pipeline* OpenGL (KUEHNE et al., 2005)

muitas dessas aplicação esses limitadores de velocidade podem ser analisados e pensados para que a aplicação se torne mais rápida.

O pipeline de renderização do OpenGL recebe objetos geométricos e imagens primitivas da aplicação e as *rasteriza* para enviar para o “*framebuffer*”. Basicamente o pipeline pode ser dividido em duas seções. Fase de transformação e fase de rasterização como mostrado na figura 2.2.

Na fase de transformação, são processados primitivas como vértices, luz, dentre outros. O resultado dessa fase é passado para a rasterização.

A rasterização, por sua vez, responsabiliza-se por colorir os pixels apropriadamente no “*framebuffer*” (processo conhecido como “*shading*”). É nessa etapa, também, que são realizados operações como, teste de profundidade, “*alpha blending*” (transparência), mapeamento de textura, entre outros.

Entender como o OpenGL funciona é importante para o entendimento dos problemas de performance que podem ser gerados por áreas críticas existentes na aplicação. Em OpenGL, existe três possibilidade principais de gargalos que podem ser encontrados e evitados (KUEHNE et al., 2005):

- A aplicação não desenha todos os pixels na “*window*” em tempo útil. Essa situação é conhecida, na literatura, como “*fill limited*”.
- A aplicação não pode processar todos os vértices que são requeridos para renderizar a forma do objeto na cena. Essa situação é conhecida, na literatura, como “*vertex (ou transform) limited*”.
- A aplicação não pode enviar os comandos de renderização do OpenGL rápido o suficiente para manter o renderizador do OpenGL funcionando. A essa situação da-se o nome de “*application limited*”.

Esse trabalho dará ênfase a técnicas para a prevenção do “*application limited*”, buscando uma taxa factível e visualmente agradável para se efetuar a renderização em um ambiente dinâmico de simulação.

2.4 Transmissão de informações geométricas

OpenGL possui cinco modos para transmitir um dado geométrico para o seu pipeline:

- Modo imediato.
- *Display lists*.
- *Vertex Arrays*.
- *Interleaved Array*.
- *Vertex Buffer Object*.

Embora todos os cinco comandos sejam perfeitamente funcionais, eles variam grandemente de performance (KUEHNE et al., 2005).

2.4.1 Modo Imediato

O modo imediato é o modo mais simples, dentre os cinco, e o mais usado por desenvolvedores de aplicações. Contudo a simplicidade traz duas grandes desvantagens para esse modo:

- As formas geométricas em modo imediato requerem um grande número de chamadas de função. Portanto, o modo imediato não é recomendado para arquiteturas de computadores onde chamadas de função requerem muito tempo de processamento.
- Modo imediato transmite todos os vértices de um objeto geométrico na aplicação pelo barramento. Com o barramento é possível a interligação de vários dispositivos. Por estar ligando dispositivos distintos, este pode vir a limitar a largura de banda e tornar um empecilho para a velocidade da aplicação. Placas gráficas modernas podem processar vértices em grande velocidade sem que elas precisem ser perturbadas pelos limites impostos pelo barramento (KUEHNE et al., 2005).

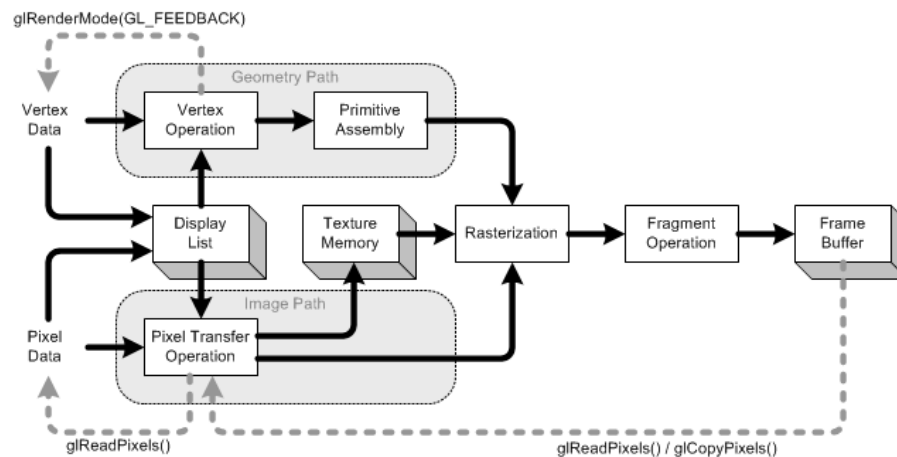


Figura 2.3: Pipeline OpenGL mostrando os estados em que se encontra a Display Lista, assim como o processo que se segue após sua criação (AHN, 2005)

2.4.2 Display List

Display List (DL), introduzida desde a versão 1.0 de OpenGL e “*deprecated*” na versão do OpenGL 3.0 (SEGAL; AKELEY, 2008), é um meio de buscar mais performance, deixando, assim os aplicativos que usam OpenGL mais rápidos. Ela é frequentemente utilizada quando se é planejado desenhar a mesma forma geométrica múltiplas vezes.

Display List é um mecanismo do OpenGL que agrupa um grupo de comandos e os armazena para uma posterior execução. Ao invocar a DL os comandos que estão nela contidos, são executados na ordem em que foram armazenados (OPENGL et al., 2005). A figura 2.3 mostra o pipeline do OpenGL com seus vários estágios e onde é formada a DL.

A primeira vista DL parece uma função como outra qualquer do grupo de funções de OpenGL. Uma vez definida os comandos a ser armazenados, ela pode ser chamada quantas vezes for necessário. Além disso essas implementações permitem que a GPU pegue dados diretamente do sistema de memória com transferência via DMA. Embora seja possível transferir um vértice individual por transferência DMA, os benefícios da redução de ciclos da CPU e do tráfego “*front-side bus*” são mais do que compensados pelos custos estruturais envolvidos. Exibir DL permite que mais dados e/ou comandos possam ser transportados em uma única transferência.

Com isso é possível imaginar, por exemplo, a modelagem de um carro, definindo cada roda como uma única DL e chamando-a quatro vezes com as transformações apropriadas (rotações e translações para que cada uma das rodas fiquem nas posições certas).

Embora possa parecer que DL só possua vantagens, ainda é possível citar algumas desvantagens. Em algumas situações os dados geométricos criados em uma DL precisam ser modificados. Isso é impossível. Seria necessário criar uma nova DL com os objetos modificados. Dependendo com que frequência esses objetos são modificados e da quantidade de funções que são armazenadas nela, o potencial avanço de performance adquirido outrora pode ser perdido, além de complicar o controle dela tendo que manusear constantes criações e deleções de DL.

Display list criada por um programa é usada no lado do cliente OpenGL. No final das contas, esses dados são processados pela GPU de uma cópia guardada pelo lado servidor OpenGL. Isso cria uma duplicata dos dados, quando comparado ao modo imediato, o que pode se tornar um problema quando o espaço é constricto. Dependendo do hardware ou do driver usado, DL pode funcionar como um mero alocador de memória. Se a arquitetura do hardware usado for preparada pra receber a DL, assim como os drivers, então a display list pode ser armazenada no dispositivo gráfico. (WILLIAMS, 2005; MOLOFEE, 1999; KUEHNE et al., 2005).

Aplicação

Um exemplo que ilustra o desempenho obtido por um programa que utiliza a Display List é apresentado no Lighthouse3d (FERNANDES, 2008). Em seu aplicativo ele simula a renderização de 36 bonecos de neve, sem textura ou iluminação, rodando no mesmo computador.

- Com Display List: 175.65 fps.
- Sem Display List: 55.45 fps.

É importante salientar que esses valores podem sofrer alguma mudança com diferentes tipos de arquiteturas de hardware ou até mesmo dos softwares. Também é importante notar que colocar todo o código dentro de uma DL pode ser prejudicial ao desempenho. A DL com excesso de código pode se tornar excessivamente grande e as transferências para a memória podem degradar o desempenho.

2.4.3 Vertex Array

Como uma alternativa a *Display List*, OpenGL implementa o *Vertex Array* (VAR), desde sua versão 1.1 e também “*deprecated*” na versão 3.0 (SEGAL; AKELEY, 2008). Ele permite que

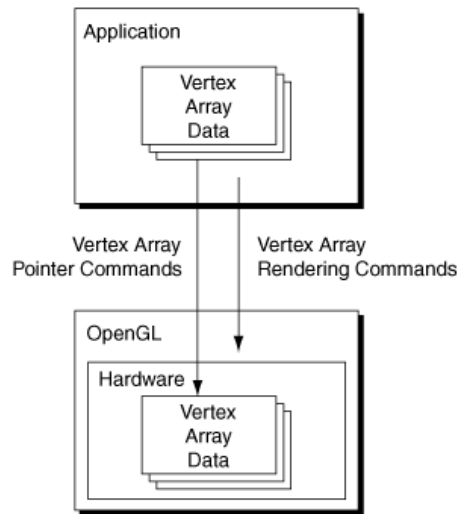


Figura 2.4: Renderização com VAR (MARTZ, 2006)

os dados característicos do objeto (como vértices e arestas) sejam agrupados e tratados como um bloco único, que provoca na performance da transferência de dados alguns ganhos proporcionados pela exibição de listas. VAR, permite, também, que dados como formas geométricas e cores sejam entrelaçados, o que pode ser útil para a criação e referênciação.

Infelizmente, VAR assume que qualquer dado individual nele não serão alterados. Isso resulta que, quando chamado um objeto usando VAR, os dados no array devem ser validados. Isso adiciona *overhead* para a transferência de dados. *Vertex array* não sofre, no entanto, a limitação de conter réplicas de dados na memória.

Alguns problemas:

- Ele quebra o paradigma cliente/servidor (o cliente toma o controle do gerenciamento da memória).
- Ele não prove um gerente interno de memória. A única coisa que o VAR faz é prover um grande gancho de memória na memória do servidor.

(WILLIAMS, 2005)

Como mostra na figura 2.4, OpenGL deve copiar o VAR toda vez que a aplicação usar as primitivas para a renderização (MARTZ, 2006).

2.4.4 Interleaved vertex array

Interleaved vertex array é uma variante do VAR, também “*deprecated*” no OpenGL 3.0, com a característica de que os dados são guardados em um bloco contínuo de memória. Em algumas arquiteturas este pode vir a ser um ótimo formato pra especificar formas geométricas (KUEHNE et al., 2005).

2.4.5 Vertex Buffer Object

Vertex Buffer Object (VBO), assim como a *Display List*, é um bom meio de ganhar desempenho para objetos que sofrem transformação no conjunto dos seus vértices, ou seja, que podem ser efetuados através da pilha de transformação de OpenGL sem mudar a sua estrutura interna, como translação, rotação e escalonamento.

VBO é uma poderosa ferramenta que permite armazenar dados em memória de alta performance, no lado do servidor OpenGL.

Ao ser criada ela prove um “gancho” da memória (buffer) que será usado através de um identificador (assim como em DL). VBO fornece, ao programador, um mecanismo de “dicas” de como esses dados serão usados. Essa “dicas” facilitam a tomada de decisão do gerente de memória do OpenGL sobre como deveria ser armazenado esses dados. Esse gerente de memória pode escolher o melhor tipo de memória (sistema, vídeo) que será alocada, dependendo do modo como o buffer será usado (NVIDIA, 2003).

VBO, também, fornece à aplicação flexibilidade. Com ela é possível modificar dados internos sem causar “overhead” na transferência causada pela validação dos dados (WILLIAMS, 2005).

Resumidamente, a VBO permite facilmente acessar dados da GPU sem a necessidade de passar pela CPU, o que pode acarretar num avanço em áreas que necessitam de objetos mutáveis com alto desempenho como, simulação de fluidos, tecidos, entre outros.

3 *Vertex Buffer Object*

VBO, extensão desenvolvida a partir da especificação de OpenGL 1.4, foi projetada para permitir que o *Vertex Array* possa ser criado diretamente no dispositivo de memória gráfica. Ou seja, através dessa abordagem pode-se evitar o excesso de transferência no barramento. Quando o *buffer object* for usado para armazenar dados de pixels, ele é chamado de *Pixel Buffer Object* (PBO).

VBO prove algumas funções de acesso especificadas através das funções de acessos normais ao VAR, como *glVertexPointer*, *glNormalPointer*, *glTexCoordPointer*, etc. Ao invés de providenciar um ponteiro, como em, *glVertexPointer()*, a aplicação especifica, primeiramente, um buffer com *glBindBuffer()* e então prove um caminho dentro do buffer para *glVertexPointer()*, no lugar de um ponteiro para a memória (KUEHNE et al., 2005).

Para criar e inicializar os dados que ficaram armazenados dentro do buffer é usado o método *BufferData()*, que além de receber os dados, recebe informações de que tipo de dados são e como esses dados serão usados para uma posterior otimização.

Para informar os tipos de dados que serão armazenadas, são usados as seguintes flags (SEGAL; AKELEY, 2008):

- *ARRAY_BUFFER* - elementos (vértices, arestas, entre outros).
- *ELEMENT_ARRAY_BUFFER* - índices.
- *PIXEL_UNPACK_BUFFER* e *PIXEL_PACK_BUFFER* - pixel (usado para leitura e escrita, respectivamente).

Para informar como os dados serão usados, são usados as seguintes flags:

STREAM_DRAW, *STREAM_READ*, *STREAM_COPY*, *STATIC_DRAW*, *STATIC_READ*, *STATIC_COPY*, *DYNAMIC_DRAW*, *DYNAMIC_READ* e *DYNAMIC_COPY*.

	STREAM	STATIC	DYNAMIC
Estático	2.740741	2.467198	2.786732
Dinâmico	3.308582	3.474968	3.310108

Tabela 3.1: Tabela (flag/modelo) registra o tempo, em segundos, do processamento.

Dividindo as palavras é possível ter um entendimento melhor de suas utilidades na aplicação (WILLIAMS, 2005).

- *STREAM* - Os dados serão especificados uma vez na aplicação e usado poucas vezes dessa maneira. Assume-se que ele terá uma atualização por acesso (seja para desenho ou não), ou seja, terá seus dados constantemente alterados.
- *STATIC* - Assume que os dados serão especificados uma vez na aplicação e não serão modificados ao longo do programa.
- *DYNAMIC* - Assume que os dados terão muita mudanças, mas também serão usados múltiplas vezes.
- *READ* - Especifica para VBO que esses dados serão usados para a leitura.
- *DRAW* - Especifica para VBO que o buffer será usado para enviar dados a GPU.
- *COPY* - Especifica o uso conjunto de READ e DRAW.

Essas combinações de uso de memória ajudam o gerente de memória da aplicação a escolher entre as várias espécies de memórias. Como cada memória tem suas diferentes características de acesso, essas “dicas” dadas a VBO permitem a esta escolher de forma eficiente qual o melhor tipo de memória a ser usado. Isso não significa para o usuário restrições, e que deva tomar decisões prévias de onde seus dados serão armazenados. Ou seja, são apenas dicas. Uma aplicação que utilize VBO pode ter seus atributos modificados mesmo após ter sido inicializada com a flag *STATIC*. No entanto o uso das flags erradas são fortemente desencorajadas, uma vez que pode haver queda (WILLIAMS, 2005). Para verificar ter resultados mais precisos de quanto pode ser a queda de desempenho um dispositivo foi desenvolvido. Ele registra o tempo de mil renderizações usando as diferentes flags. A tabela 3.1 mostra o aplicativo executando em um computador com uma placa gráfica *ATI Technologies Inc RS482 [Radeon Xpress 200]*. A tabela 3.1 foi criada a partir de dois objetos idênticos, porém para o modelo dinâmico o objeto tem cem de seus vértices alterados, en-

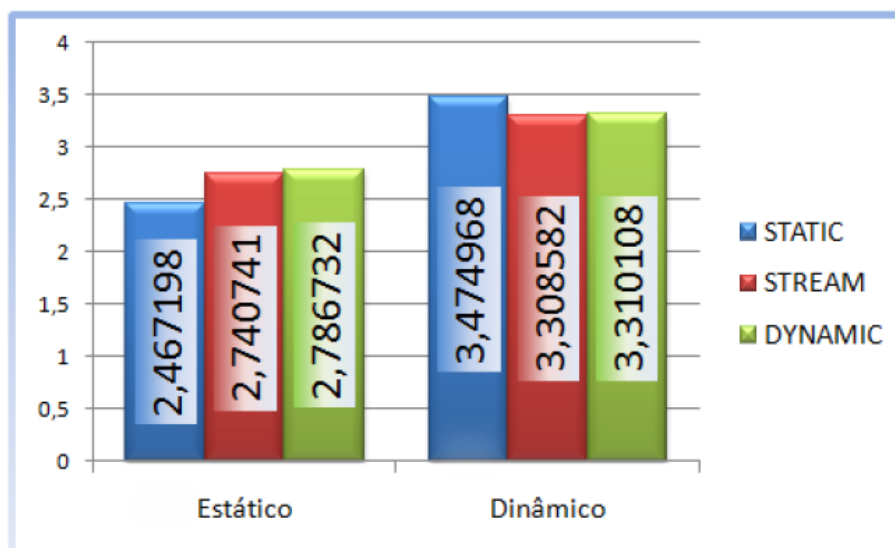


Figura 3.1: Gráfico ilustrativo da tabela 3.1

quanto que no modelo estático não existe alteração na estrutura do objetos. Ao final é registrado o tempo de processamento.

Com uma simples análise da tabela 3.1 já possível perceber que para um objeto que não sofre transformação em sua estrutura a melhor flag será a STATIC. Por outro lado para um objeto deformáveis a estrutura recomendada será a STREAM ou STATIC. As flags STREAM e STATIC obtiveram resultados muito parecidos devido a sua forma de uso que, também, é muito parecido.

Se no momento em que a VBO é gerada o OpenGL for incapaz de reservar a quantidade de memória requerida é gerado um erro do tipo `OUT_OF_MEMORY`.

Como nesse trabalho será usado essencialmente a VBO para desenhar, muitas vezes essas flags serão referidas apenas com *STREAM*, *STATIC* e *DYNAMIC*, pré-supondo o entendimento que são as flags terminadas em *DRAW*.

Na VBO blocos de arrays de índices podem ser guardados no buffer object. Para isso é necessária a utilização das funções de desenho `DrawElements` ou `DrawRangeElements` que renderizam um objeto baseando seus vértices a partir do índice. Com essa opção, é possível armazenar um objeto complexo, diminuindo drasticamente a quantidade de vértices repetidos que devem ser passados para a aplicação (SEGAL; AKELEY, 2008). A figura 3.2 ilustra os modos da VBO.

Como esse trabalho visa o estudo da renderização, com ênfase a objetos dinâmicos, VBO será o foco da atenção, ao longo deste documento.

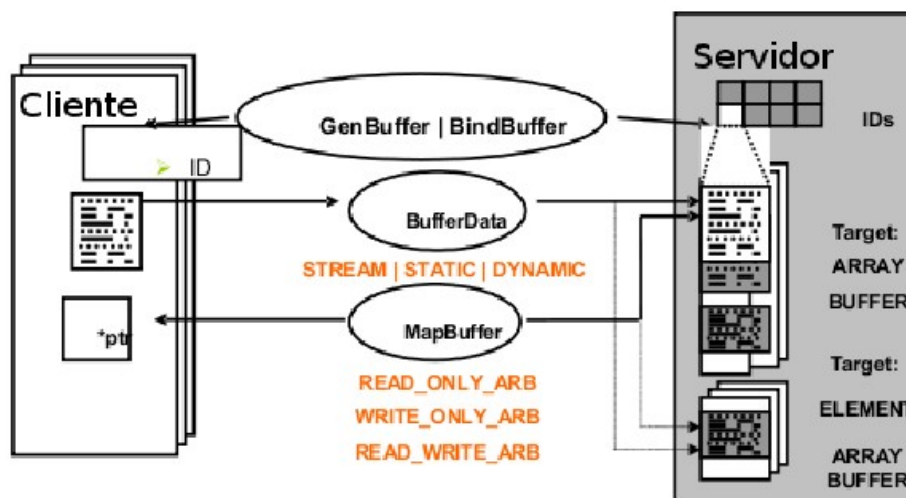


Figura 3.2: Figura que ilustra o modelo cliente servidor de VBO e como modos da VBO são usados. (NVIDIA, 2003)

Além disso, de acordo com o documento “The OpenGL Graphics System: A Specification”, Display List, assim como suas respectivas funções, terão seu modelo deprecado. Ou seja, é esperado que sejam completamente removidas em futuras versões do OpenGL (SEGAL; AKELEY, 2008; RICCIO; GUINOT, 2007).

4 *Sistemas de Computação Gráfica*

Os sistemas de computação gráfica apresentam uma arquitetura de pipeline no qual os dados 3D numa cena são mostrados cruzando-se diferentes estágios. Dependendo da aplicação ou do tipo de dados, é possível que alguns estágios tornem-se gargalos, assim reduzindo drasticamente a performance dos dispositivos gráficos. É essencial que se localize esses gargalos para que os dispositivos gráficos possam trabalhar adequadamente.

A figura 4.1 mostra o diagrama de estágios, simplificado, do pipeline gráfico do *OpenGL*:

Alguns gargalos podem ser percebidos e analisados ao se observar essa arquitetura (figura 4.1) (CASTELLÓ; RAMOS; CHOVER, 2005):

- AGP/PCI Express: Excesso de dados no barramento.
- Geometria: Excesso de vértice ou muitos cálculos por vértice.
- Frame buffer: Excesso de leitura/escrita no frame buffer.

Além de outros que não serão enfatizados nesse trabalho como problemas com drivers, texturas muito grandes.

4.1 **Aceleração**

Diferentes metodologias podem ser usadas para remover esses gargalos existentes nos sistemas gráficos. O principal problema pode ser encontrar o motivo por trás desses gargalos. Para problemas conhecidos, *OpenGL* prove métodos específicos para o tratamento de cada um. Desde os métodos imediatos (os menos recomendados, por serem, em geral lentos) até métodos mais avançados como os introduzidos anteriormente, *Display List* (recomendada para objetos geométricos com

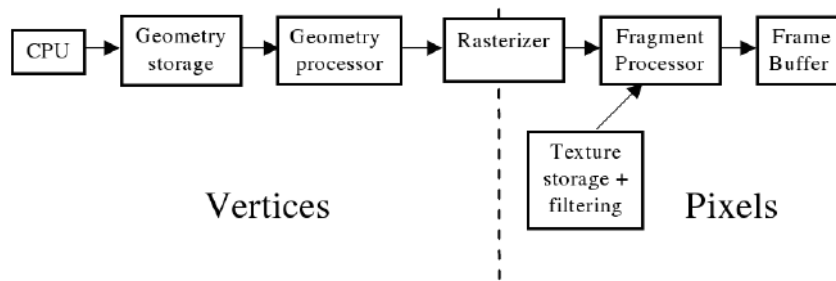


Figura 4.1: Estágios OpenGL até que a imagem final seja gerada e mostrada no monitor. (CASTELLÓ; RAMOS; CHOVER, 2005)

estrutura estática) e *Vertex Buffer Object* (usado normalmente para objetos que sofrem constantes transformações em suas estrutura de vértices).

Os modos imediatos e o VAR transmitem os seus dados através do barramento a cada quadro. VAR foi projetado, primeiramente, para reduzir o tempo em uma chamada de função, e a semântica dele permite que a aplicação modifique os dados dentro do array quando for necessário. Já o VBO foi desenhado para permitir que o vertex array fosse criado diretamente na memória dos dispositivos gráficos (KUEHNE et al., 2005).

Através dos estudos mostrados a seguir, por Castelló, Ramos e Chover na tabela 4.1, VBO mostrou uma característica importante a ser observada. A tabela mostra um estudo comparativo entre as diferentes técnicas comentadas até o presente momento. O experimento foi feito usando diferentes computadores, com diferentes processadores, placas de vídeo e sistema operacional. Como principal meio de mensuração foi usado o frame por segundo. Para a implementação do aplicativo foi usada a linguagem C++ usando a API do OpenGL.

Afim de evitar o envio de vértices repetidos ao sistema gráfico estruturas como “*triangle strips*” ou “*triangle fan*” são altamente recomendados. No “*triangle strip*” o primeiro triângulo é desenhado e posteriormente apenas um vértice é adicionado para formar cada novo triângulo subsequente. Já no “*triangle fan*” todos os triângulos compartilham os vértices que tenham em comum. Conseqüentemente a visualização de imagens formadas com o uso de “*triangle strip*” ou “*fan*” proporciona grandes ganhos, uma vez que menos informações são enviadas para a placa gráfica (CASTELLÓ; RAMOS; CHOVER, 2005).

Na tabela 4.1 são usados dois algoritmos que permitem o uso da representação baseado em “strip”:

- Stripe (EVANS; SKIENA; VARSHNEY, 1996).
- NvTriStrip (LIBRARY, 2002).

Esse estudo visa testar diferentes técnicas OpenGL (modo imediato, DL, VAR, VBO) com os seguintes modos de comparação (CASTELLÓ; RAMOS; CHOVER, 2005):

- Modelo original de triângulos.
- Modelo de triângulos otimizado pela NVIDIA.
- Um “strip” com modelo NVIDIA.
- Vários “strip” com modelo NVIDIA.
- Vários “strip” com modelo Stripe.

Cow	GPU	V. cache	t / s	Kb	IM	DL	VA	VBO
<i>Triangles: original model</i>	<i>FX5700</i>	no	5804	136	397.60	1315.68	707.68	1050.95
	<i>Ge6600</i>				850.00	1319.00	1259,70	1268.00
	<i>FX5900</i>				816.19	1951.05	1421.58	1913.09
<i>Triangles: NvTriStrip</i>	<i>FX5700</i>	yes	5804	136	371.63	1288.71	384.62	1031.97
	<i>Ge6600</i>				796,20	1488,50	1420,60	1430,60
	<i>FX5900</i>				774.23	1958.04	1385.61	1888.11
<i>One strip: NvTriStrip</i>	<i>FX5700</i>	yes	1	102	543.90	1408.59	831.17	1239.76
	<i>Ge6600</i>				1084,90	1494,50	1436,60	1434,60
	<i>FX5900</i>				1089.91	2154.88	1860.20	2081.92
<i>Several strips: NvTriStrip</i>	<i>FX5700</i>	yes	551	98	427.72	1368.63	701.30	1105.89
	<i>Ge6600</i>				1015.00	1430,60	1441,60	1427,60
	<i>FX5900</i>				981.02	2010.99	1550.45	1674.33
<i>Several strips: STRIPE</i>	<i>FX5700</i>	no	101	98	622.38	1592.41	833.17	1259.74
	<i>Ge6600</i>				1009.00	1520,50	1440,60	1450,60
	<i>FX5900</i>				966.03	2350.65	1995.00	2283.72

Tabela 4.1: Resultados obtidos com o modelo Cow, retirado dos estudos de Castelló. (CASTELLÓ; RAMOS; CHOVER, 2005)

Ao observar a tabela 4.1 percebe-se que após a utilização de diferentes técnicas de renderização, do conjunto de funções do OpenGL, a pior opção a ser adotada seria o uso do modo imediato, seguido do Vertex Array. Display List lidera com cerca de 800% de ganho de velocidade em cima do modo imediato.

Mesmo com um desempenho menor, como esperado, que a *Display List*, VBO se mostrou muito eficiente. Ou seja, para objetos genéricos, que não se conheça seu comportamento durante toda sua existência, é recomendado a utilização da VBO. Isso já ocorre em estudos em que o tempo de renderização é levado em consideração como no caso do artigo, “*Realtime 3D computed tomographic reconstruction using commodity graphics hardware*” em que a VBO e a PBO são usados em conjuntos com o barramento PCIexpress para acrescentar aumento de performance (XU; MUELLER, 2007).

5 *Teste Gráfico*

Nesse capítulo será efetuado o estudo comparativo entre VBO, DL, e o modo Imediato. Para esse estudo serão aplicados, para a VBO, suas diferentes flags, *STATIC*, *DYNAMIC* e *STREAM*. No estudo do efeito da atualização dos vértices na VBO será usado o pior caso, ou seja, atualização de todos os vértices.

Tais testes serão efetuados em dois computadores diferentes com configurações distintas. Nesses computadores destacam-se a utilização de marcas distintas de placas de vídeo, ATI e NVIDIA. Tais placas foram escolhidas por representarem placas gráficas com alta fatia do mercado. Como pode ser observado na figura 5.1, a placa de vídeo mais vendida é da Intel, porém, é importante ressaltar que, nem todas as placas gráficas da Intel são destinadas a computação de alto desempenho, muitas dessas são vendidas em conjunto com placas-mãe e computadores portáteis para baratear os custos, motivo esse que levou a escolha da NVIDIA e ATI (TECHARP, 2009).

Os resultados dos testes são obtidos através de um protótipo desenvolvido especificamente para esse fim. Nesse protótipo aparece a figura da cabeça de um macaco, criado através de um mesh de triângulos. Esse modelo foi retirado de um dos modelos prontos oferecidos pelo software de

Overall PC Graphics Shipments, Q3 2007					
Data by Jon Peddie Research					
Rank	Name	Q3 2007		Q2 2007	
		Shipments* (thousands)	Share	Shipments* (thousands)	Share
1	Intel	37183	38.00%	30576.32	37.60%
3	Nvidia	33171.15	33.90%	26510.32	32.60%
2	ATI/AMD	18689.35	19.10%	15857.4	19.50%
4	Via Technologies	6487.455	6.63%	6261.64	7.70%
5	SiS Corp.	2054.85	2.10%	2033	2.50%
6	Matrox Graphics	97.85	0.10%	162.64	0.20%
7	Others	n/a	<1%	n/a	<1%
	Total	97850	100.00%	81320	100.00%
	* Approximate number of units				

Figura 5.1: Tabela que mostra as vendas de placas de vídeo distintas (PEDDIE, 2007).

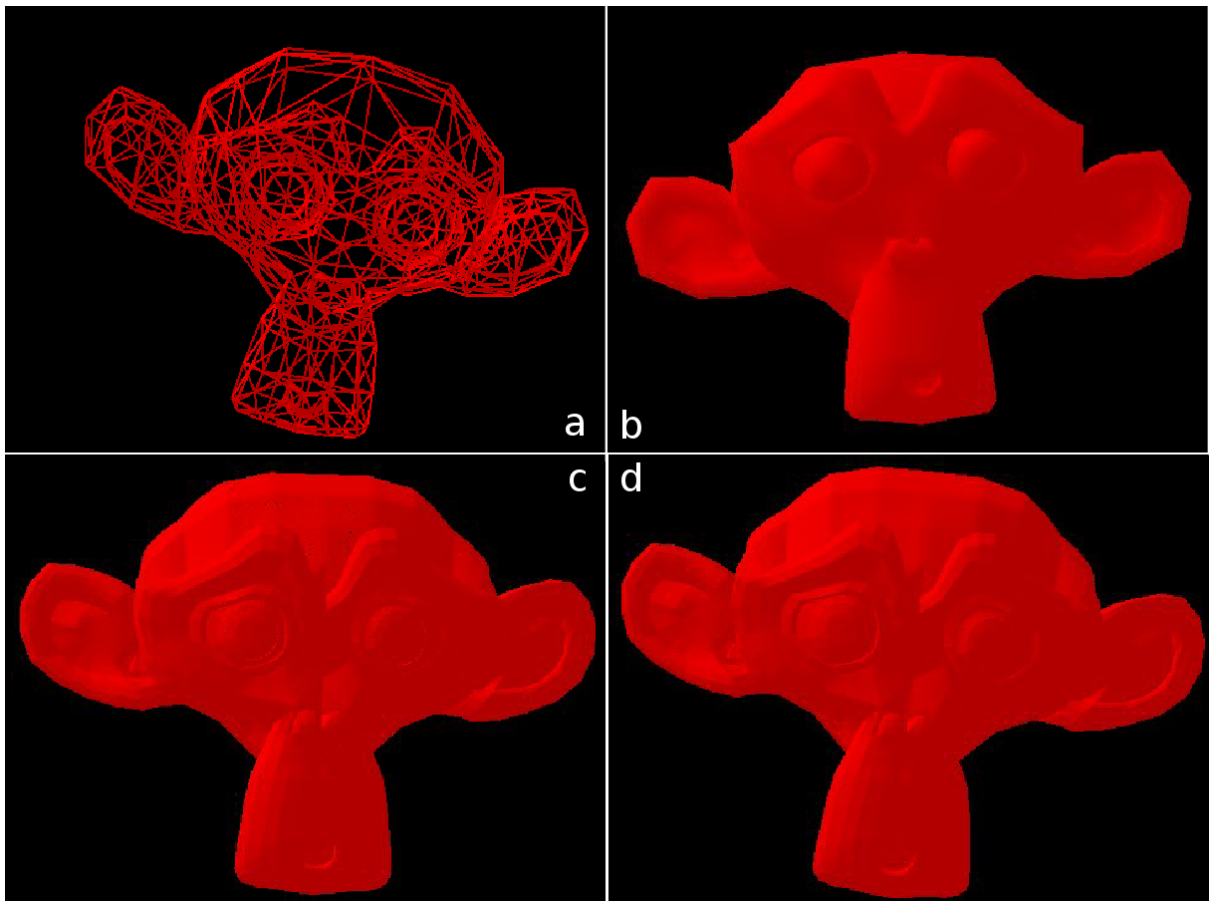


Figura 5.2: Essas figuras mostram a renderização do macaco fornecido pelo Blender. (a) 507 vértices usando modelo de arames. (b) 507 vértices sem usar modelo de arames. (c) 124242 vértices usando modelo de arames. (d) 124242 vértices usando modelo de arames. Todos criados através do aplicativo desenvolvido para a renderização.

modelagem em três dimensões, o Blender (FOUNDATION, 2009). O modelo é carregado e após ficar alguns segundos estático na tela é efetuado o teste de performance. As figuras 5.2a, b, c e d mostram imagens do protótipo em funcionamento para quatro diferentes tipos de situações.

5.1 ATI

O computador usado para os testes que seguem possui a seguinte configuração:

- Placa de Vídeo: *ATI Technologies Inc RS482 [Radeon Xpress 200]*.
- Processador: *AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ processors*.

- Memória RAM: 1931032 KB.
- Memória de vídeo: 128 MB (compartilhada).

O modo imediato na Placa de Vídeo descrita, para o polígono com 124242 vértices, se mostrou proibitivo. O tempo de espera para realizar a renderização foi muito grande, inviabilizando o uso desse modo para a renderização desse tipo de mesh de triângulo. Por essa razão a tabela mostra '***' no lugar do tempo em FPS.

Quantidade de vértices	Modo		Wireframe	Fps
507	Imediato		Sim	2.35831
			Não	2.40554
	VBO	STATIC_DRAW	Sim	256.27
			Não	315.76
		STREAM_DRAW	Sim	257.756
			Não	314.705
		DYNAMIC_DRAW	Sim	256.126
			Não	315.316
	Display List		Sim	251.281
			Não	317.362
124242	Imediato		Sim	***
			Não	***
	VBO	STATIC_DRAW	Sim	5.69205
			Não	6.55219
		STREAM_DRAW	Sim	5.69674
			Não	6.56185
		DYNAMIC_DRAW	Sim	5.70677
			Não	6.59073
	Display List		Sim	18.0255
			Não	39.5942

Tabela 5.1: Tabela comparativa com o resultado obtido a partir dos testes efetuados com o aplicativo funcionando em três diferentes modos de OpenGL: Modo imediato, DL e VBO para placa de vídeo ATI.

5.2 NVIDIA

O computador usado para os testes que seguem possui a seguinte configuração:

- Placa de Vídeo: GeForce 8600 GT.
- Processador: AMD Athlon(tm) 64 X2 Dual Core Processor 4400+.
- Memória RAM: 1931032 KB.
- Memória de vídeo: 256 MB.

Quantidade de vértices	Modo		Wireframe	Fps
507	Imediato		Sim	481.277
			Não	454.402
	VBO	STATIC_DRAW	Sim	924.772
			Não	1482.3
		STREAM_DRAW	Sim	904.495
			Não	1548.36
		DYNAMIC_DRAW	Sim	905.079
			Não	1573.22
	Display List		Sim	978.373
			Não	2124.81
124242	Imediato		Sim	1.76376
			Não	1.76971
	VBO	STATIC_DRAW	Sim	26.174
			Não	98.38
		STREAM_DRAW	Sim	42.1023
			Não	109.726
		DYNAMIC_DRAW	Sim	28.5337
			Não	97.6158
	Display List		Sim	29.0233
			Não	183.322

Tabela 5.2: Tabela comparativa com o resultado obtido a partir dos testes efetuados com o aplicativo funcionando em três diferentes modos de OpenGL: Modo imediato, DL e VBO para placa de vídeo NVIDIA.

6 *Detecção de Colisão*

A detecção de colisão entre modelos geométricos em um ambiente dinâmico é uma questão importante em muitas áreas da computação gráfica. Áreas como games, CAD, geometria computacional, modelagem de sólidos, aplicativos de simulação e até mesmo robótica tem estudado soluções para desenvolver bons sistemas de interação (GOTTSCHALK; LIN; MANOCHA, 1996). Conseguir alcançar a velocidade requerida em um ambiente virtual pode vir a ser um grande desafio. Por exemplo, um sistema de reação ao tato (“*haptic force-feedback*”) pode requerir aproximadamente 1000 interações por segundo (KLOSOWSKI et al., 1998).

Detectar a colisão é perceber o instante exato em que dois ou mais objetos estão perto o suficiente para que possa ocorrer a sobreposição entre eles, mas sem que essa realmente ocorra. A detecção de colisão em um ambiente virtual é importante para que se possa evitar a interpenetração dos objetos.

Para o propósito deste trabalho será definido um objeto ou modelo como sendo uma coleção de polígonos, aqui triângulos, agrupados formando uma figura espacial tridimensional, como mostrado na figura 6.1. Sendo assim, a colisão acontecerá, de forma simplificada, pela intersecção de cada polígono de um par de objetos. Como é possível perceber, fazer esse tipo de teste é extremamente demorado, inviabilizando a prática de uma simulação em tempo real. Por esse motivo é importante estudar uma estrutura que consiga simplificar a complexidade (quantidade muito grande de polígonos) e a representação estrutural (formas abstratas com muita concavidade) dessa interação entre objetos genéricos sem perder a precisão para determinar os contatos.

Abaixo segue algumas das técnicas encontradas na literatura que abordam esse assunto:

- Técnica de decomposição espacial: Octree (MOORE; WILHELMS, 1988), k-d tree (HELD; KLOSOWSKI; MITCHELL, 1996), brep-indices (JR., 1991) e regular grid (HELD; KLOSOWSKI; MITCHELL, 1996) são exemplos da técnica de decomposição espacial (*Spatial decomposi-*

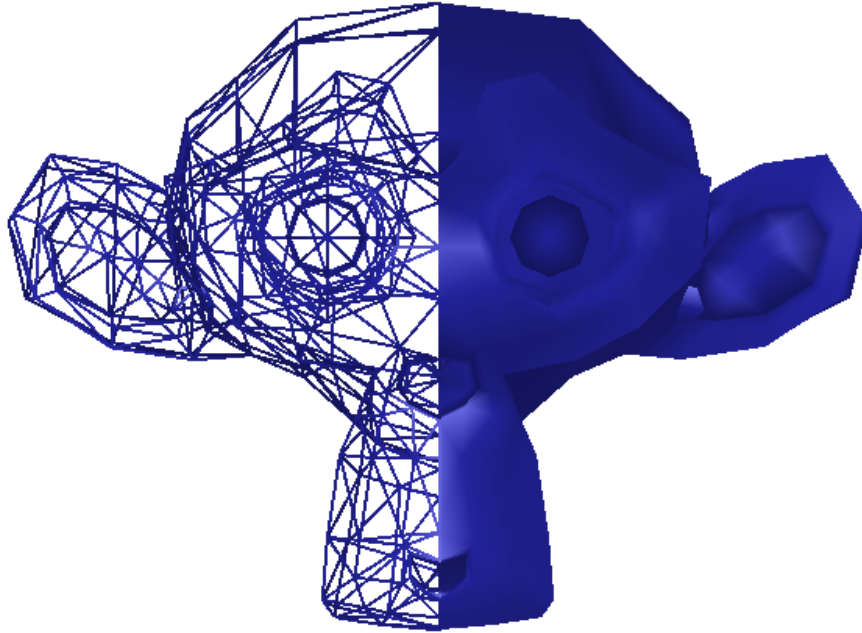


Figura 6.1: Objeto ou modelo, representado como um modelo de arames

tion techniques - SST). Após dividir o espaço ocupado pelos objetos, apenas uma checagem é necessária para verificar a colisão entre um par de objeto (ou pares de objetos) que estão na mesma célula (área subdividida) ou entre células próximas.

- Técnica baseada em distância: Coleção de trabalhos baseados no diagrama de Voronoi (COHEN et al., 1995; LIN; MANOCHA, 1995). Se a distância é menor que um limiar, pré estabelecido, então a colisão é declarada (LIN; MANOCHA, 1995). Um sistema popular de colisão é o I-COLLIDE (COHEN et al., 1995), que é usado em conjunto com a técnica “sweep-and-prune”(COHEN et al., 1995; PONAMGI; MANOCHA; LIN, 1995) para reduzir a quantidade de pares com colisão em potencial.
- Técnica de hierarquias de volume: Essa é uma das técnicas mais rápidas de colisão para propósito geral para modelos de polígonos (SU, 2007). Com essa técnica é criada uma árvore de volumes simples que são testados. Esse processo exclui polígonos que não se colidem. Ao final é feita a intersecção entre os polígonos restantes. Essa técnica será mais aprofundada em capítulos posteriores.

6.1 Bounding Volume Hierarchies

A representação hierárquica de objetos geométricos é uma importante otimização na área de tratamento de colisão. Diferentes formas primitivas (caixas, esferas) podem ser usadas, para tratar a colisão, fornecendo uma intersecção de áreas de forma rápida e eficiente.

O tratamento computacional de colisões é um ponto crítico importante que deve ser estudado cuidadosamente. Assegurar que objetos interajam de forma robusta é um processo computacional extremamente pesado. Muitas pesquisas nessa área dividem a colisão em fases, assim diminuindo os cálculos mais complexos. Na fase inicial, *broad phase*, os algoritmos tentam descartar aqueles objetos que certamente não interagirão. Várias são as técnicas que podem ser usadas para efetuar esse descarte. Entre elas estão, “*Sweep & Prune*” (COHEN et al., 1995; PONAMGI; MANOCHA; LIN, 1995), “*global bounding volume tables*” (PALMER; GRIMSDALE, 1995) e “*overlap table*” (WILSON et al., 1999).

Tendo determinado qual algoritmo será usado é hora de determinar que tipo de volume primitivo será usado para redução de detecção. Essa segunda fase consiste em utilizar objetos de menor complexidade para representar um objeto mais complexo. Chamada de “*narrow phase*” essa fase tipicamente usa um dos objetos das figuras 6.2, 6.3, 6.4, criadas para ilustrar esse trabalho, para delimitar a região de interesse. Posteriormente essa região será usada para efetivar a intersecção. Para isso é criada uma árvore de volumes à partir dos pontos do objeto. Essa fase reduz o esforço computacional que é exigido para uma detecção com exatidão, como é descrito pelos autores supracitados, Palmer e Grimsdale (1995) e Moore e Wilhelms (1988). A árvore é construída por um método “top-down” (de cima para baixo), através de sucessivas divisões recursivas. A cada passo de recursão um novo volume, menor, é gerado até que exista, ao final, apenas uma primitiva a ser computada.

As figuras 6.2, 6.3 e 6.4, mostram alguns dos diferentes tipos de “*Bounding Volume Hierarchies*” (BHV) que são usados na literatura, para o processamento da “*narrow phase*”. Nas figuras é possível observar três níveis de hierarquias. Os volumes mostrados revestem um objeto no formato de um pequeno coelho. Ao final existe um volume para cada polígono.

Outros tipos podem ser usados, tipos com complexidades bem maior e que se adaptem melhor a estrutura do objeto que será usado. Porém objetos muito complicados podem vir a dificultar a interação. Abaixo uma pequena equação que pode ajudar a decidir que volume usar (GOTTSCHALK; LIN; MANOCHA, 1996; BERGEN, 1997b; KLOSOWSKI et al., 1998).

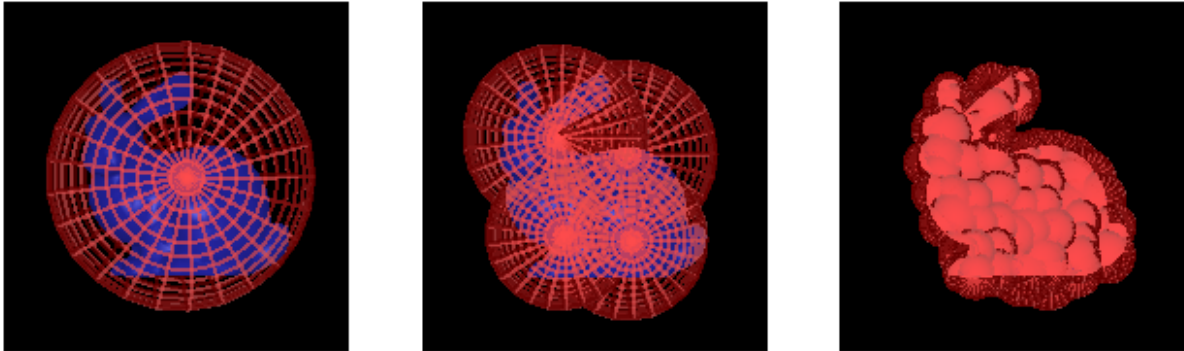


Figura 6.2: Figura que exemplifica o nível hierárquico utilizando esferas.

$$T = N_u * C_u + N_v * C_v$$

Onde:

- T: custo total da detecção de colisão entre dois objetos.
- N_u : é o número de testes de intersecção dos pares de bounding volumes.
- C_u : é o custo para testar um par de bounding volume durante a intersecção.
- N_v : é o número de pares de primitivas testadas por contato.
- C_v : é o custo para testar um par de primitivas durante o contato.

Essa fórmula mostra claramente que a performance depende principalmente de dois fatores: ao encaixe do “*bounding volume*” e da simplicidade do teste de intersecção para o par de modelos. Esse primeiro fator representado pelos N_v e C_u , e o segundo pelos C_v .

Primitivas simples, como caixas ou esferas, possuem um baixo valor de C_u e C_v , porém eles podem não se encaixar muito bem com o objeto em questão. As esferas, principalmente, por poder ter sua rotação ignorada, simplificam a fórmula para translações. Por outro lado, elipsóides e caixas orientados podem se ajustar melhor ao objeto, mas possuem a checagem de sobreposição relativamente custosa.

O exemplo na tabela 6.1 ilustra melhor a fórmula. Nela r é o custo de rotação para um ponto ou vetor tridimensional e t é o custo para aplicar a rotação e a translação a um ponto. Então, r e t , serão 18 e 20 operações em ponto flutuante, respectivamente. Na equação é usada a esfera e o paralelepípedo orientado para mostrar a diferença entre um volume simples e um complexo. A

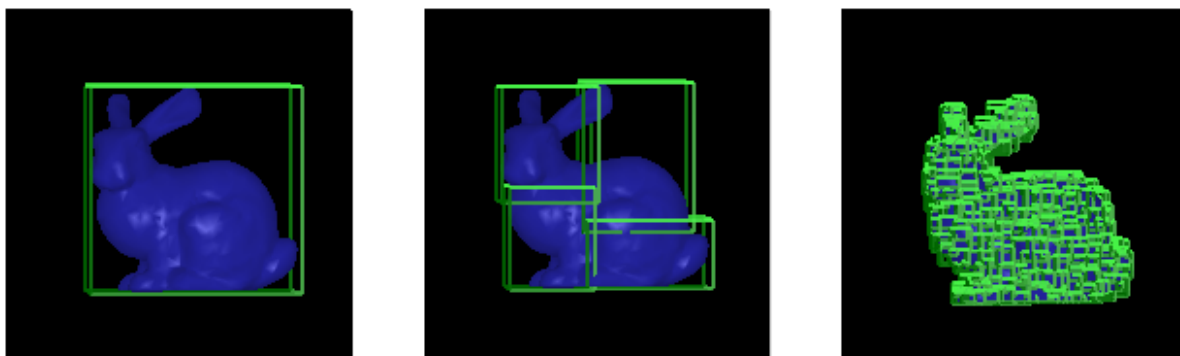


Figura 6.3: Figura que exemplifica o nível hierárquico utilizando paralelepípedos alinhados.

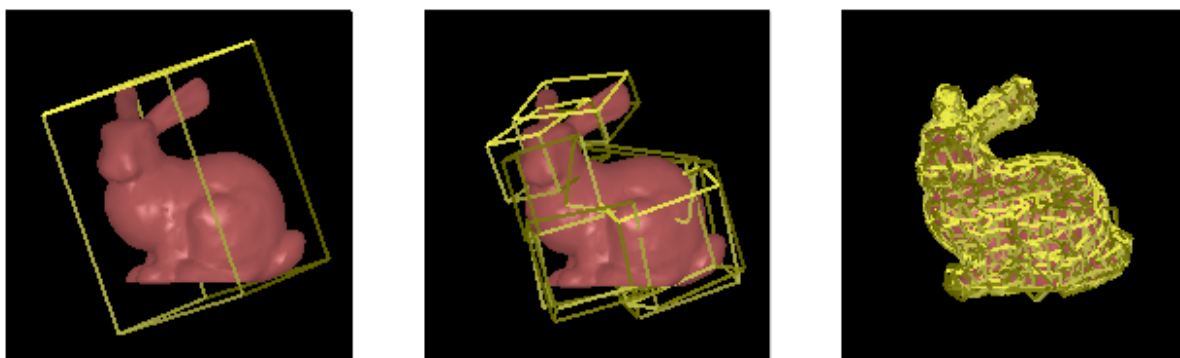


Figura 6.4: Figura que exemplifica o nível hierárquico utilizando paralelepípedos não alinhados.

Primitivas	C_u	C_v
Esfera	$21 = 1 * t$	10
Caixa orientada	$75 = 3 * r + 1 * t$	80-200

Tabela 6.1: Fórmula que ilustra o cálculo do custo básico, para esfera e caixa orientada, da colisão em *Bounding Volume Hierarchies* (BRADSHAW; O’SULLIVAN, 2004)

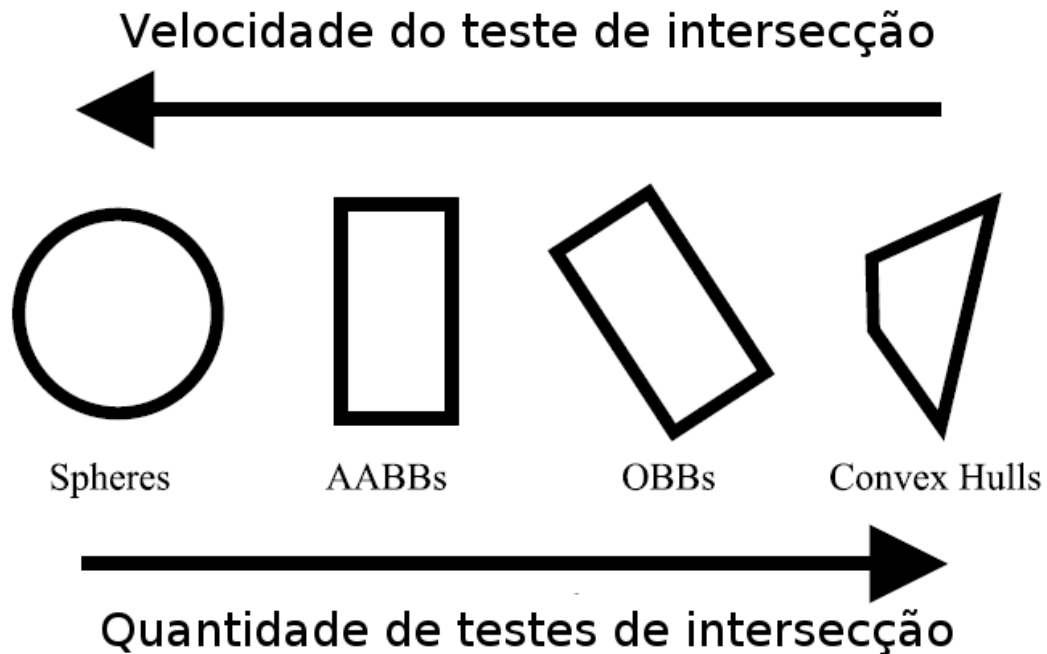


Figura 6.5: Representação de diferentes tipos de *bounding volumes*. A figura mostra as diferentes características. Quanto mais simples a forma (como caixa ou esfera), mais rápido é o teste de intersecção. Quanto mais complexa é a forma, menos testes de intersecção é necessário fazer. Formas mais elaboradas, também, se encaixam melhor nos objetos (GOTTSCHLK, 2000).

figura 6.5 ilustra a quantidade de testes de intersecção em termo da velocidade para se efetuar essa detecção.

6.1.1 Axis Aligned Bounding Box

O “*Axis Aligned Bounding Box*” (AABB) é o “*bounding volume*” mais comum. Ele é criado como uma caixa, formada por 6 faces retangulares, sempre alinhado ao eixo de coordenadas do sistema, como na figura 6.3. Por possuir uma forma mais simplificada, o uso do AABB proporciona testes de intersecção com maior velocidade, envolvendo apenas comparações diretas entre seus valores de coordenadas. Em termos de armazenamento em memória, para sua inteligibilidade,

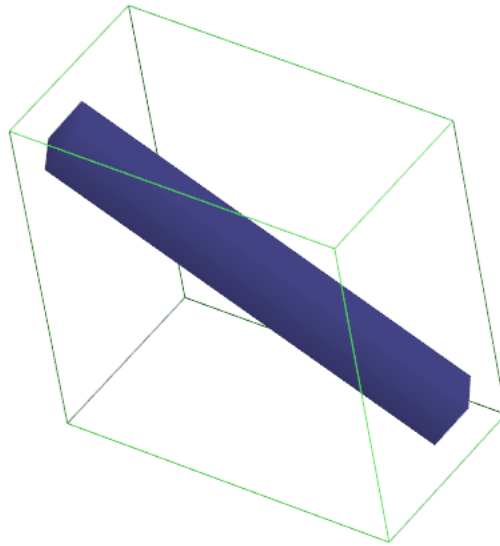


Figura 6.6: Figura que demonstra como a má utilização de um AABB pode resultar em um encaixe imperfeito para figuras em uma posição não alinhada (*figura obtida através de um aplicativo desenvolvido para esse fim*).

pode-se utilizar apenas 6 grandezas escalares (3 para posição e 3 para extensão) (ERICSON, 2004).

Diferente de outras árvores de volumes, a *AABB tree* (árvore criada com volumes AABB) podem ser atualizada em alta velocidade, após a deformação do modelo, como mostrado pelo autor Gino van den Bergen. Como reconstruir totalmente a árvore é um processo demorado, a atualização pode vir a ser um bom método para ganhar velocidade.

As árvores criadas com AABB são formadas por caixas alinhadas ao eixo de coordenadas do sistema cartesiano. Ou seja, todas as caixas nessa árvore são orientadas (BERGEN, 1997a).

Por estar preso sempre a mesma orientação as árvores AABB possuem algumas desvantagens. Sempre que um objeto mudar a sua orientação uma nova AABB deve ser criada. Um simples rotação pode fazer com que o AABB mude de tamanho. Outro detalhe que deve ser percebido é que objetos que fiquem na diagonal em relação a coordenada do sistema, fazem com que sobre muito espaço vazio dentro da caixa, como observado na figura 6.6. Vale lembrar que árvores em geral podem ser orientadas a dois sistemas de coordenadas. Uma requer que as AABBs sejam refeitas, orientadas as coordenadas do mundo. A outra, orientada ao sistema de coordenadas do objeto, onde a rotação é guardada e a AABB está alinhada ao sistema local.

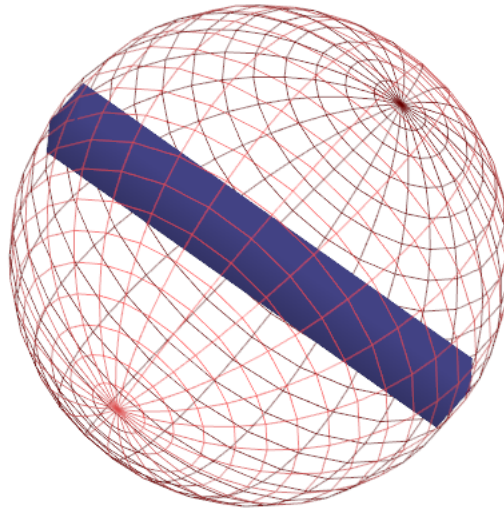


Figura 6.7: Aqui é demonstrado como a *Bounding Sphere* pode ter um resultado ruim para objetos muito alongados. É importante perceber que independente da orientação do objeto a Sphere sempre terá o mesmo resultado. Isso é uma desvantagem, óbvia, se comparado ao AABB (*figura obtida através de um aplicativo desenvolvido para esse fim*).

6.1.2 Bounding Sphere

A Bounding Sphere é outra estrutura de “*bounding volumes*” muito popular. Sua utilização pode ser, até mesmo, comparada com o AABB. Esse tipo de estrutura proporciona muitas vantagens. As características da esfera tornam a rotação muito rápida, tendo a necessidade de efetuar a rotação nas árvores, mas não no volume. No quesito armazenamento, ela pode ser considerada uma das mais econômicas, uma vez que é preciso guardar apenas o seu centro e raio. Uma outra vantagem muito importante é o seu teste de intersecção que tem uma performance muito alta (ERICSON, 2004).

Apesar de todos os benefícios que o uso da *Bounding Sphere* traz ela ainda possui algumas desvantagens. Computar a menor esfera possível não é um trabalho tão trivial quanto calcular a menor *Bounding Box*. Existem muitos algoritmos na literatura, como descritas em (GLASSNER, 1995) e (KIRK, 1994) que devem ser analisados para verificar aquele que é mais performático e que crie uma esfera mínima verdadeira.

Outro problema é semelhante ao encontrado no AABB. No AABB figuras não alinhadas ao eixo como a mostrada na figura 6.6 fazem com que o *Bounding Volume* não se adapte ao objeto. Para a esfera mesmo as alinhadas ao eixo podem trazer problemas como na figura 6.7.

6.1.3 Oriented Bounding Box

Um “*Oriented Bounding Box*” (OBB), é como uma caixa AABB, mas tem orientação arbitrária, como visto na figura 6.4. Um dos modos de representar o OBB é armazenando uma AABB com orientação, assim ela possuiria 15 grandezas escalares (uma matriz 3x3 para guardar a orientação, 3 para a posição e 3 para a extensão) (BERGEN, 1997a). Uma forma de representação, mais compacta seria armazenar a orientação como um ângulo Euleriano (GLASSNER, 1995) ou como um quaternion (GLASSNER, 1995). Infelizmente, apesar de ser mais compacto, o teste de colisão entre OBB-OBB usa a matriz da orientação destas, isso iria requerer que se efetuasse uma conversão. Esse processo de conversão do ângulo para a matriz tornaria a operação de colisão muito lenta e inviabilizaria o uso da árvore com caixas não alinhadas. Independente da forma como a OBB é armazenada ela, inevitavelmente, ocupará mais memória que a “*Bounding Box*” ou a “*Bounding Sphere*” (ERICSON, 2004). Ao tentar usar a OBB para objetos dinâmicos deformáveis, esta poderá apresentar uma perda de performance grande uma vez que a atualização da árvore OBB é mais complexa (BERGEN, 1997a).

Apesar de possuir algumas desvantagens, como a utilização de mais memória e a complexidade de atualização, a OBB diminui a quantidade de testes que se deve efetuar para se verificar a intersecção entre as formas. Por possuir um formato mais livre, ele pode se ajustar ao formato do objeto com mais facilidade. A figura 6.8 mostra que, ao contrário da “*Sphere*” e da AABB, a utilização dessa abordagem diminui o problema das figuras diagonais mostrados anteriormente.

6.2 Teste de Bounding Volume

“*Bounding Volume Hierarchies*” (BVH) são amplamente utilizados em algoritmos de detecção de colisão. Para facilitar o teste de colisão entre objetos, formas geométricas mais simples são circunscritas em modelos mais complexos. Entre as formas mais utilizadas duas se destacam devido ao seu desempenho e sua facilidade de implementação, as “*bounding sphere*” (utilizando esferas) e as *bounding box* (utilizando paralelepípedos) (ERICSON, 2004), como mostrados em seções anteriores. Os diferentes tipos de bounding volumes possuem diferentes tipos de características. Apesar da facilidade de implementação a utilização de formas simples perdem grande parte de seu ganho quando são usadas para testar a intersecção de objetos com muitas concavidades. As formas mais simples como as das caixas ou das esferas tendem a não ficar bem ajustados a formas que contenham muita complexidade. Para amortecer esse problema é possível usar um outro tipo de

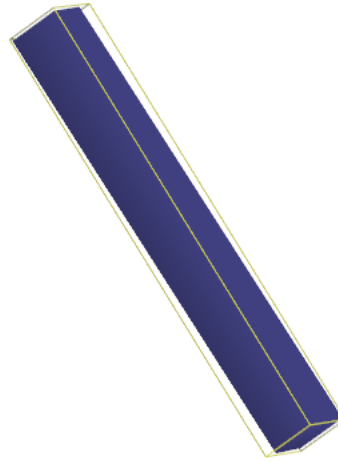


Figura 6.8: Aqui é demonstrado como que o OBB pode, facilmente, se ajustar a uma figura geométrica, mesmo ela estando em diagonal. Ao contrário da Sphere e do AABB, o OBB consegue deixar um objetos inscrito a ele com menos áreas sobrando. Através dessa figura é possível perceber o por que do OBB conseguir efetuar a colisão com menos testes de intersecção (*figura obtida através de um aplicativo desenvolvido para esse fim*).

abordagem, como o uso de “*oriented bounding box*” (OBB), elipsóides ou em casos mais extremos “*convex hull*” (menor polígono convexo contendo a menor área que envolve uma figuras geométricas). Como a OBB consiste de um paralelepípedo livre em sua orientação, esse pode tentar tomar uma posição onde melhor se encaixe com o objeto que sofrerá o teste de sobreposição. O “*convex hull*”, ao contrário dos outros volumes apresentados, já apresenta uma estrutura não trivial, uma vez que ele pode ser representado como uma simplificação (com uma quantidade de polígonos muito reduzida) do objeto que ele inscreve. Aumentar a quantidade de polígonos nos volumes da hierarquia pode significar um aumento na dificuldade de implementação do algoritmo de detecção de colisão e possivelmente um aumento no tempo de execução.

Para tirar proveito do desempenho no tratamento de colisão dos “*bounding*” mais simples, com caixas ou esferas, e ajustar essas formas mais simples em objetos de diferentes complexidades, normalmente é criado uma estrutura de dados no formato de árvore, que contém “*bounding volume*” que representam apenas um pedaço do objeto. Para isso, normalmente, o objeto é subdividido igualmente até que nas folhas cada “*bounding*” contenha apenas um único polígono.

Para tirar conclusões mais precisas sobre o assunto foi efetuado testes de desempenho em três diferentes tipos de BVH:

- Axis aligned bounding box (AABB).
- Oriented bounding box (OBB).
- Bounding Sphere.

6.3 Tabelas comparativas

Para realizar tais testes foi desenvolvido um aplicativo onde diferentes tipos de colisões são testadas. Como a VBO se mostrou uma grande aliada para a renderização de objetos estáticos e dinâmicos os protótipos criados a partir desse ponto terão sua renderização realizada com o auxílio da VBO. A imagem 6.3 mostra algumas cenas com o aplicativo em execução. É possível perceber o momento de colisão, onde os triângulos do modelos são realçados com a cor amarela. Ao final da simulação é obtido o tempo em que cada teste foi efetuado. O aplicativo usa algumas técnicas apresentadas no (ERICSON, 2004).

No primeiro protótipo desenvolvido foi percebido que ao montar a árvore hierárquica, as folhas não possuíam o menor polígono (no protótipo representado por um triângulo), fazendo com que a hierarquia não alcançasse um molde ideal. Ao corrigir esse erro, foi possível obter resultados mais confiáveis e além disso, tirar outras conclusões.

Para não poluir a imagem das tabelas a maneira como os objetos se comportam, foi simplificada:

- P: Objeto parado.
- 2M: Dois objetos se movendo (transladando).
- 3M: Três objetos se movendo (transladando).
- 2MeR: Dois objetos se movendo e rotacionando.

Primeiramente é importante deixar claro que para se efetuar as devidas comparações é importante perceber que estas devem ser feitas entre objetos com comportamentos idênticos. Cada um dos comportamentos listado nas tabelas 6.2, 6.3, 6.4 e 6.5 testam uma característica. Assim ao se comparar o comportamento da *Bounding Sphere* com AABB, será subentendido que será um teste entre os comportamentos da *Bounding Sphere* com os respectivos comportamentos da AABB.

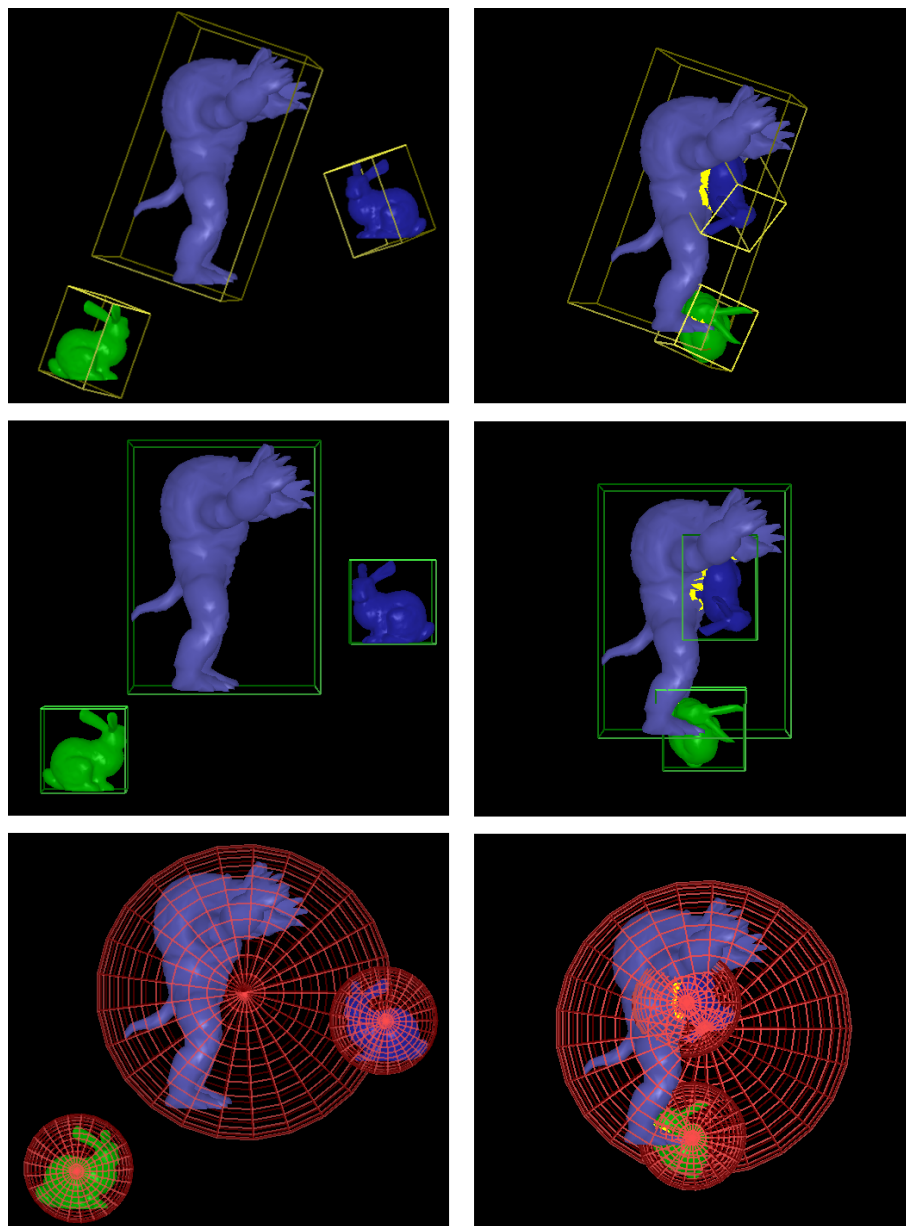


Figura 6.9: Aqui é mostrado cenas do aplicativo em execução. Na primeira figura é mostrado a colisão usando OBB. Na segunda figura é mostrado a colisão usando AABB. Na última é usado a *bounding sphere*. Pode-se perceber nas figuras partes amarelas onde os objetos estão se tocando.

Tipo de árvore	Objetos	Tempo	TPF	FPS
Bounding Sphere	P x MeR	17.89190	0.17891900	5.589
	2M	17.35050	017350500	5.764
	2MeR	14.79230	0.14792300	6.760
	2MeR x P	16.29630	1.16296300	6.136
	3M	34.93030	0.34930300	2.863
AABB	P x MeR	9.541960	0.09541960	10.480
	2M	8.619200	0.08619200	11.602
	2MeR	9.637530	0.09637530	10.376
	2MeR x P	8.932270	0.08932270	11.195
	3M	24.47270	0.24472700	4.086
OBB	P x MeR	5.438790	0.05438790	18.386
	2M	6.440900	0.06440900	15.526
	2MeR	4.702610	0.04702610	21.265
	2MeR x P	5.348130	0.05348130	18.698
	3M	12.86240	0.12862400	7.775

Tabela 6.2: Tabela resultante dos testes efetuados com diferentes tipos de BVH, para três modelos criados com o auxílio do Blender (FOUNDATION, 2009).

Para as tabelas 6.2, 6.3, 6.4 e 6.5 é testado a interação entre objetos mostrados na figura . Para isso é usado dois tipos de objetos. Dois com o formato de um coelho com cerca de três mil vértices. E um segundo objeto com um formato de um tatu com aproximadamente mil e quinhentos vértices. O aplicativo executa situações diferentes onde é possível ocorrer colisão. Em uma das situações um dos objetos fica parado enquanto o outro translada e rotaciona na direção do primeiro até que a colisão ocorra (P x MeR). Ao final das cem interações que ocorrem durante a simulação é obtido o tempo em segundos, o tempo por frame (TPF) e o frame por segundo (FPS) de cada situação. Ao final são usados outros objetos para garantir a integridade dos testes.

Como mostrado na tabela 6.2 a BVH deve ser criada completamente para que se tenha um resultado ótimo. Ao comparar as tabelas 6.2, 6.3 é visível o ganho de performance. Uma vez que as folhas não continham apenas um polígono, efetuar a intersecção entre as várias faces se torna um processo demorado. Ou seja, com a utilização da BHV é possível reduzir muito a quantidade de intersecções custosas através de testes de contato simples e rápidos.

Com base nos experimentos realizados conclui-se que as formas mais simples (caixa e esfera), usadas indiscriminadamente, tem um menor desempenho. A comparação entra as tabelas 6.3 e a 6.4 mostrou que os volumes que possuem muitos espaços vazios para os objetos que se diferenciam muito delas degradam o desempenho. Através do uso das estruturas de dados adequadas pode-se reduzir esse fato, ajustando assim as formas simples a objetos complexos, e até mesmo objetos com

Tipo de árvore	Objetos	Tempo	TPF	FPS
Bounding Sphere	P x MeR	0.884771	0.00884771	113.024
	2M	0.953547	0.00953547	104.872
	2MeR	1.187370	0.01187370	84.220
	2MeR x P	1.290120	0.01290120	77.512
	3M	2.657690	0.02657690	37.627
AABB	P x MeR	0.380773	0.00380773	262.624
	2M	0.364937	0.00364937	274.020
	2MeR	0.548463	0.00548463	182.328
	2MeR x P	0.498351	0.00498351	200.662
	3M	1.431560	0.01431560	69.854
OBB	P x MeR	0.869080	0.00869080	115.064
	2M	0.908204	0.00908204	110.107
	2MeR	1.086220	0.01086220	92.063
	2MeR x P	1.240920	0.01240920	80.585
	3M	2.694290	0.02694290	37.116

Tabela 6.3: Essa tabela mostra os testes efetuados com diferentes tipos de BVH, para 3 modelos criados com o auxílio do Blender (FOUNDATION, 2009).

concavidades.

Para a detecção de colisão o AABB mostrou o melhor resultado, em muitos casos chegando a ter ganhos superiores a 100% em relação aos outros métodos, como demonstrado na 6.3. Ao comparar o OBB e a *bounding sphere*, eles obtiveram resultados semelhantes (não chegando a ultrapassar 2% entre os tempos) variando entre mais ou menos rápido dependendo do tipo de teste.

Para testes efetuados nas estruturas, verificando o desempenho nas atualizações das árvores, o resultado parece se inverter. A AABB que possui os melhores resultados nos testes de colisão, possui desempenho inferior as outras duas. Nesse caso o destaque vai para a esfera, que devido a simplicidade em efetuar rotações, obteve os melhores números, mesmo não sendo muito superiores a OBB. Esses resultados inversos se devem, em grande parte, ao fato de a AABB ter que recalcular o tamanho de suas caixas toda vez que o objeto é rotacionado para melhor ajustar a figura que se encontra em seu interior. O que não acontece no caso da OBB e da *bounding sphere* que não importando a orientação da figura sempre estarão com o melhor ajuste. Isso, possivelmente, não aconteceria se uma deformação ocorresse. Caso ocorresse deformações dos objetos os volumes na árvore tendem a trocar seu tamanho, isso causaria grandes prejuízos a *bounding sphere*.

Finalmente, como último teste, foi analisado como as hierarquias reagiriam diante de um ambiente sem colisão. Nesse caso os objetos foram mantidos afastados o suficiente para que seus

Tipo de árvore	Objetos	Tempo	TPF	FPS
Bounding Sphere	P x MeR	0.938159	0.00938159	106.592
	2M	1.076610	0.01076610	92.884
	2MeR	1.238140	0.01238140	80.766
	2MeR x P	1.401670	0.01401670	71.343
	3M	3.416150	0.03416150	29.273
AABB	P x MeR	0.418097	0.00418097	239.179
	2M	0.441248	0.00441248	226.630
	2MeR	0.675903	0.00675903	147.950
	2MeR x P	0.603785	0.00603785	165.622
	3M	1.814850	0.01814850	55.101
OBB	P x MeR	0.955510	0.00955510	104.656
	2M	1.033860	0.01033860	96.725
	2MeR	1.285340	0.01285340	77.800
	2MeR x P	1.410740	0.01410740	70.885
	3M	3.538890	0.03538890	28.257

Tabela 6.4: Essa tabela demonstra a atuação das BVH para objetos alongados.

BVH não se tocassem. Mesmo sabendo que eles nunca iriam se colidir os testes de colisão, ainda assim, foram feitos para o estudo. Como esperado dos teste onde foi verificado somente a colisão, tanto dos objetos parados como em movimentação, obtiveram resultados muito parecidos. Apesar da hierarquia com esfera obter o melhor resultados nos testes, este não teve uma diferença muito significativa, cerca de 5%, se comparado com a AABB. Os testes para objetos que não se chocam, com o modelo usado, tornam-se triviais, a medida que são simplificados por seus *bounding*. Esses ganhos puderam ser obtidos graças as simplificações impostas pelo uso da BVH.

Tipo de árvore	Objetos	Tempo*	TPF	FPS
Bounding Sphere	P x MR	1.139440	0.01139440	87.762
	2M	0578866	0.00578866	172.752
	2MeR	1.808970	0.01808970	55.280
	2MeR x P	2.045310	0.02045310	48.892
	3M	2.911300	0.02911300	34.349
AABB	P x MR	2.553870	0.02553870	39.156
	2M	0.637578	0.00637578	156.844
	2MeR	4.238550	0.04238550	23.593
	2MeR x P	4.074680	0.04074680	24.542
	3M	5.715530	0.05715530	17.496
OBB	P x MR	1.374300	0.01374300	72.764
	2M	1.397790	0.01397790	71.541
	2MeR	2.245640	0.02245640	44.531
	2MeR x P	2.249920	0.02249920	44.446
	3M	3.250950	0.03250950	30.760

Tabela 6.5: Aqui é mostrada a tabela de desempenho das atualizações efetuadas na árvore da BVH para diferentes comportamentos.

Tipo de Árvore	Tempo*	TPF	FPS
Bounding Sphere	0.000147	0.00000147	680272.00
AABB	0.000154	0.00000154	649351.00
OBB	0.000248	0.00000248	403226.00

Tabela 6.6: Tabela com o tempo de colisão para objetos que não estão colidindo. Essa tabela calcula o tempo que objetos levam para detectar que não estão em contato.

Em suma, ao se analisar o desempenho de colisões, e extraindo os desempenho de ambos, atualização das árvores, e do teste de colisão, é possível perceber que para objetos que se movem por um ambiente dinâmico sem sofrer rotação (e deformação) a melhor escolha seria o uso de um hierarquia em forma de árvore usando o caixa alinhado, ou seja, o AABB. Para objetos que além de movimentar-se, também rotacionam em torno de um vetor arbitrário, é possível adquirir um melhor resultado utilizando a hierarquia de esferas, ou seja a *bounding sphere*. No caso da ausência do choque entre objetos a hierarquia de esfera merece destaque. Em todos os casos, foram usados

apenas volumes simples, que demonstraram poder representar objetos complexos sem complicar o código ou ter que criar métodos novos para cada situação.

6.4 Classificador de comportamento

Através do estudo dos comportamentos de diferentes BVH foi possível obter informações importantes sobre o desempenho de cada tipo de estrutura em uma determinada situação. Tomando como base os resultados obtidos pode-se encontrar heurísticas para desenvolver um classificador de comportamento. Esse classificador deve ser capaz de extrair as vantagens de cada estrutura de forma rápida e confiável.

Na seção anterior foi visto que a OBB e a *bounding sphere* obtiveram resultados muito próximos nos seus testes de intersecção. Já nos testes de atualização a *bounding sphere* obteve grandes vantagens, quando comparado a atualização das estruturas após comportamentos de rotação. Ou seja, efetuar vários testes rápidos de colisão em um volume simples se mostrou mais eficiente que efetuar poucos testes (mais lentos) como observado na figura 6.5. Devido a esses motivos não há mais a necessidade de permanecer utilizando a árvore OBB. Este só traria mais complexidade e talvez degradaria os tempos de execução.

Com base nos dados obtidos nos testes efetuados, as seguintes características foram notadas:

- Para objetos que não se colidem, independente do tipo de movimento que ele tiver, ou que trajetória ele segue os melhores resultados foram obtidos com a utilização da *bounding sphere*.
- Para objetos que se colidem, em movimento de translação (ou seja, sem rotacionar), independente da trajetória dele, os melhores resultados foram obtidos com a utilização da AABB.
- Para objetos que se colidem, em movimento de rotação (transladando ou não), os melhores resultados foram obtidos com a utilização da *bounding sphere*.

Devido a simplicidade dos algoritmos de sobreposição de estruturas do mesmo tipo esse modo de interação possui as melhores velocidades. Com isso, no ambiente de colisão, dois ou mais objetos só poderão interagir se esses tiverem o mesmo tipo de árvore BVH. Como para o desenvolvimento de um ambiente virtual é fundamental que todos os objetos possam interagir é necessário uma maneira de que todos os objetos tenham a mesma estrutura colisiva. A medida que a quantidade de modelos no ambiente aumente, mais difícil é manter os ganhos de performance dos estudos

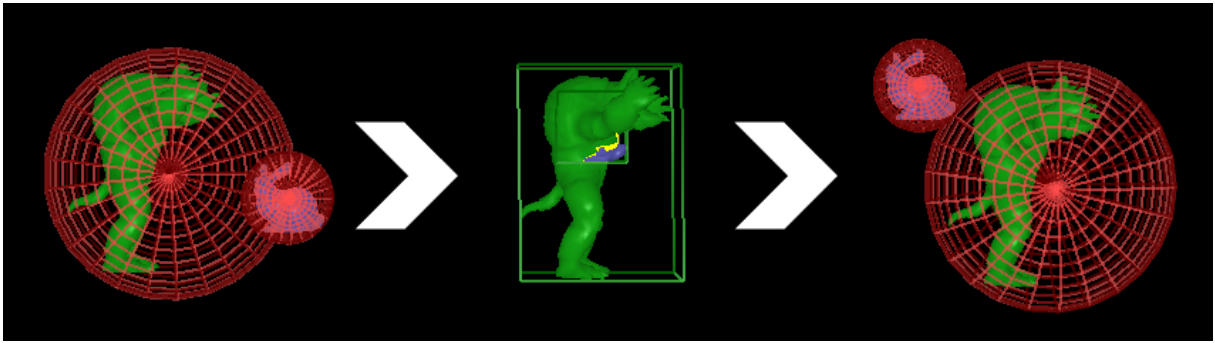


Figura 6.10: Aqui é mostrada uma sequência de cenas do aplicativo desenvolvido em execução. Na figura é testada a colisão entre dois modelos que inicialmente estão parados e não se colidem, mas que com o passar do tempo ambos se aproximam através de translação. Como os modelos não rotacionam, o dispositivo desenvolvido modifica a estrutura de colisão para a caixa no momento da colisão automaticamente. Com o uso desse protótipo é possível utilizar mais de uma árvore hierárquica, visando a obtenção do tempo ótimo.

anteriores, logo é inviável que todos tenham sempre a mesma estrutura hierárquica. Para que esse problema seja contornado e as otimizações obtidas com o uso das árvores não sejam perdidas um outro tipo de abordagem deve ser seguida.

O tratamento de colisão, no ambiente dinâmico, é efetuado aos pares. Ou seja, dado um grupo de objetos colisivos, cada objeto só efetua a colisão com um único outro objeto, um a um, até que ao final todos tenham sido verificados. Um vez que as colisão ocorrem aos pares, dois objetos que estejam preste a colidir adotarão a mesma estrutura até que um outro objeto, ou tipo de colisão, o force a trocar. Sendo assim, os modelos sempre colidirão usando árvores hierárquicas idênticas. Com essa abordagem é viável a obtenção de performance através de estruturas hierárquicas dinâmicas.

Outro fato que pode dificultar esse tipo de abordagem é a velocidade de criação das árvores. A criação delas não é um processo com velocidades constantes, dependendo da quantidade de polígonos que um modelo tem a criação pode ser mais rápida ou mais lenta. Para evitar o desperdício de tempo de criar uma nova árvore sempre que elas são trocada, no momento em que o programa inicia elas são criadas e armazenadas. Para manter essas árvores condizentes com o modelo sempre que este sofre transformação todas as BVH (aqui AABB e *bounding sphere*) devem ser atualizadas.

O processo de atualização dessas estruturas consiste em atualizar nodo a nodo até que toda árvore esteja coerente. Esse processo pode se tornar um gargalo durante o processo de simulação do ambiente. Através de um estudo no modelo de colisão pode-se perceber que uma vez que um tipo de hierarquia é adotada a outra pode ser ignorada até o momento da sua utilização. Visto isso,

Tipo de árvore	Objeto	TPF	FPS
Bounding sphere	P x MR	0.0208056	48.064
	2M	0.0247551	40.396
	2MeR	0.0309716	32.288
	2MeR x P	0.0272090	36.753
	3M	0.0506223	19.754
AABB	P x MR	0.0214073	46.713
	2M	0.0098139	101.896
	2MeR	0.0474066	21.094
	2MeR x P	0.0427409	23.397
	3M	0.0735857	13.590
Genérico	P x MR	0.0191064	52.339
	2M	0.093238	107.252
	2MeR	0.0309934	32.265
	2MeR x P	0.0272720	36.668
	3M	0.0484878	20.624

Tabela 6.7: Tabela que mostra o tempo (em tempo por frame e em frame por segundo) da *bounding sphere*, AABB e do protótipo que atualiza a árvore em tempo de execução (genérico). Esse é o tempo para a execução do teste de intersecção e da atualização das árvores.

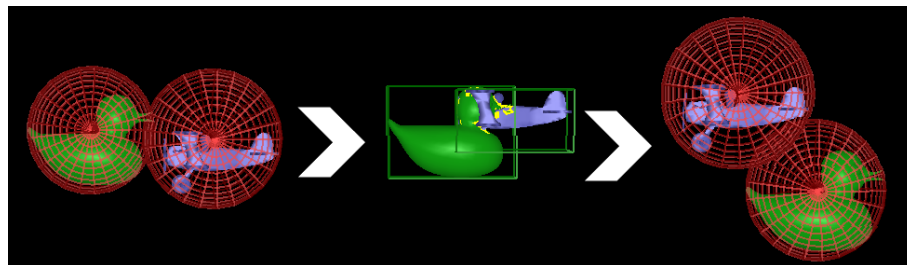


Figura 6.11: Aqui é mostrada passos do programa em execução utilizando os novos modelos.

no desenvolvimento do dispositivo de simulação apenas as estruturas úteis são atualizadas. Para não criar inconsistência entre as árvores usadas e as não usadas, toda transformação é armazenada dentro de uma pilha de matrizes de transformações. Através da pilha de transformações é possível guardar todas as modificações efetuadas no modelo, e com apenas uma matriz (produto de todas as matrizes empilhadas) restaurar a estrutura que não estava sendo usada. Com essa abordagem as transformações da estrutura são só aplicadas quando esta é requerida como é mostrado na figura 6.10. Nessa figura é mostrado passos do aplicativo em execução. Os objetos se aproximam e quando colidem sua estrutura de colisão muda para uma caixa. Quando não há mais colisão as estruturas voltam a ser esferas.

Através dos testes efetuados no novo modelo de colisão (atualizando as árvores em tempo

Tipo de árvore	Objeto	TPF	FPS
Bounding sphere	P x MR	0.0285599	35.0141
	2M	0.0156634	63.8432
	2MeR	0.0290779	34.3904
	2MeR x P	0.0567786	17.6123
	3M	0.0408506	24.4794
AABB	P x MR	0.029673	33.7007
	2M	0.0081278	123.034
	2MeR	0.0418165	23.914
	2MeR x P	0.0731133	13.6774
	3M	0.0538555	18.5682
Genérico	P x MR	0.0244036	40.9775
	2M	0.0081233	123.102
	2MeR	0.0274508	36.4288
	2MeR x P	0.0470974	21.2326
	3M	0.0341611	29.2731

Tabela 6.8: Tabela adquirida a partir dos teste efetuados em novos modelos.

de execução), refletidos na tabela 6.7, pode-se observar que este não obteve grandes ganhos de performance se comparado aos modelos hierárquicos. Entretanto, quando se analisa o resultado dos teste de maneira geral, é possível perceber que o modelo genérico conseguiu condensar os melhores tempos de cada tipo de interação. Ou seja, mesmo não conseguindo obter resultados melhores que o maior resultado da tabela, ele conseguiu ter seus resultados, no mínimo, igual ao melhor resultado de cada tipo de hierarquia.

Para garantir que os resultados dos testes efetuados não estão atrelados unicamente aos modelos utilizados até então, uma outra tabela é criada. Nessa nova tabela, todos os testes são reavaliados utilizando modelos diferentes, ilustrados na figura 6.11. A tabela 6.8 mostra os novos valores para os testes feitos anteriormente. Como é possível observar, apesar de tempos diferentes, a lógica não mudou, confirmando as heurísticas criadas.

Com esses resultados um ambiente virtual pode utiliza mais de um tipo de árvore, extraindo o melhor de cada estrutura. Para isso o usuário da aplicação deve se certificar que possui memória disponível para acomodar as árvores e assim poder extrair o máximo de desempenho da aplicação.

7 *Conclusão*

Através dos experimentos retirados da literatura e efetuados ao longo desse trabalho é possível perceber que a VBO traz algumas vantagens em sua utilização para a renderização de objetos tridimensionais. Seu comportamento nos testes demonstrou que mesmo que não tenha tido o mesmo desempenho que a *Display List*, ainda sim é equiparável a esta, com a vantagem de poder ser modificada em tempo de execução com mais rapidez. O que torna a inicialização e a sua mutação muito vantajosa.

Além disso, é importante observar que, uma vez que os programas desenvolvidos a partir desses estudos estarão atualizados com relação ao OpenGL 3.0 (SEGAL; AKELEY, 2008), já que DL, mais usado anteriormente, está prestes a desaparecer.

Espera-se que num futuro próximo, com o lançamento do *OpenGL 3.0* e a *Display List deprecated*, que as novas placas de vídeo procurem otimizar a VBO no próprio hardware.

Os resultados foram animadores e permitiram que se pudesse avançar para o segundo passo.

Através dos estudos e testes efetuados foi possível desenvolver um protótipo capaz de atualizar a estrutura da BVH automaticamente. Para isso ele utiliza as heurísticas obtidas nos testes para escolher qual a melhor estrutura para cada momento de simulação do ambiente tridimensional.

Um possível trabalho futuro seria analisar o comportamento da BVH para o tratamento de colisão de objetos deformáveis. Isso se torna relativamente simples na parte de renderização uma vez que a VBO mostrou ótimos resultado em sua atualização, mas é importante pensar em um modo de transmitir essas deformações para a BVH de modo que prejudique, o mínimo possível o desempenho. Com a utilização desses recursos poderá ser possível a simulação de objetos como tecidos, evitando-se possíveis gargalos da aplicação.

Referências Bibliográficas

AHN, S. H. *OpenGL Rendering Pipeline*. 2005.

BERGEN, G. van den. Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools*, A. K. Peters, Ltd., Natick, MA, USA, v. 2, n. 4, p. 1–13, April 1997. ISSN 1086-7651. Disponível em: <<http://portal.acm.org/citation.cfm?id=763346>>.

BERGEN, V. den. *Efficient collision detection of complex deformable models using AABB tree*. 1997.

BRADSHAW, G.; O’SULLIVAN, C. Adaptive medial-axis approximation for sphere-tree construction. *ACM Transactions on Graphics*, ACM Press, New York, NY, USA, v. 23, n. 1, p. 1–26, January 2004. ISSN 0730-0301. Disponível em: <<http://dx.doi.org/10.1145/966131.966132>>.

CARAPETO, R. R.; FRANCISCO, S. M.; MAURÍCIO, J. G. *Programação em GPU: CUDA, CTM*. nov 2007. Disponível em: <<http://algos.inescid.pt/jcm/cpd/papers/4a4/GPUProgramming.pdf>>.

CASTELLÓ, P.; RAMOS, J. F.; CHOVER, M. *A Comparative Study of Acceleration Techniques for Geometric Visualization*. 2005. Department of Computer Languages and Systems. Disponível em: <<http://www.springerlink.com/content/lnapg20p6un7b2ay/>>.

COHEN, J. D. et al. I-collide: an interactive and exact collision detection system for large-scale environments. In: *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*. New York, NY, USA: ACM, 1995. p. 189–ff. ISBN 0-89791-736-7.

ERICSON, C. *Real-Time Collision Detection*. Morgan Kaufmann, 2004. Hardcover. ISBN 1558607323. Disponível em: <<http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/1558607323>>.

EVANS, F.; SKIENA, S.; VARSHNEY, A. *Optimizing Triangle Strips for Fast Rendering*. 1996. IEEE Computer Society. Disponível em: <<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00568125>>.

FERNANDES, A. R. *Display Lists Tutorial*. mai 2008. Lighthouse3d. Disponível em: <<http://www.lighthouse3d.com/opengl/displaylists/index/php?1>>.

FOLEY, J. D. et al. *Computer Graphics: Principles and Practice in C*. Second. Addison-Wesley Professional, 1995. Hardcover. ISBN 0201848406. Disponível em: <<http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0201848406>>.

FOUNDATION, B. *Blender*. 2009. Blender Foundation. Disponível em: <<http://www.blender.com.br/>>.

GLASSNER, A. S. *Graphics Gems*. [S.l.: s.n.], 1995. AP Professional.

GOTTSCHALK, S.; LIN, M. C.; MANOCHA, D. Obbtree: A hierarchical structure for rapid interference detection. *Computer Graphics*, v. 30, n. Annual Conference Series, p. 171–180, 1996. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.7796>>.

GOTTSCHALK, S. *Collision Queries using Oriented Bounding Boxes*. 2000. The University of North Carolina at Chapel Hill.

HARADA, T.; KOSHIZUKA, S.; KAWAGUCHI, Y. *Smoothed Particle Hydrodynamics on GPUs. The Visual Computer manuscript*. 2007. Disponível em: <<http://www.iii.utokyo.ac.jp/takahiroharada/projects/sph.html>>.

HELD, M.; KLOSOWSKI, J. T.; MITCHELL, J. S. B. Real-time collision detection for motion simulation within complex environments. In: *SIGGRAPH '96: ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96*. New York, NY, USA: ACM, 1996. p. 151. ISBN 0-89791-784-7.

JR., G. V. *A Data Structure for Analyzing Collisions of Moving Objects*. Purdue University, 1991. Department of Computer Science. Disponível em: <<http://ftp.cs.purdue.edu/research/technical-reports/1990/TR-2090-986.pdf>>.

KIRK, D. *Graphics Gems III (IBM Version): Ibm Version (Graphics Gems - IBM)*. Har/dsk. Morgan Kaufmann, 1994. Hardcover. ISBN 0124096735. Disponível em: <<http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0124096735>>.

KLOSOWSKI, J. T. et al. Efficient collision detection using bounding volume hierarchies of k -DOPs. *IEEE Transactions on Visualization and Computer Graphics*, v. 4, n. 1, p. 21–36, /1998. Disponível em: <<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.7707>>.

KUEHNE, B. et al. Performance opengl: platform independent techniques. In: *SIGGRAPH 05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA: ACM, 2005. p. 1.

LIBRARY, N. *Available on the Internet at the following*. 2002. NVIDIA Corporation.

LIN, M. C.; MANOCHA, D. *Fast interference detection between geometric models*. 1995. The Visual Computer.

LOPES, J. M. B. *Computação Gráfica*. mai 2008. Disponível em: <<http://disciplinas.ist.utl.pt/leic-cg/programa/livro/Rasterizacao.pdf>>.

LUEBKE, D. et al. Gpgpu: general purpose computation on graphics hardware. In: *SIGGRAPH 04: ACM SIGGRAPH 2004 Course Notes*. New York, NY, USA: ACM, 2004. p. 33.

MARTZ, P. *Drawing Primitives in OpenGL*. may 2006. Disponível em: <<http://www.informit.com/articles/article.aspx?p=461848>>.

MOLOFEE, J. *Display Lista: Lesson 12*. 1999. NeHe. Disponível em: <<http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=12>>.

MOORE, M.; WILHELMS, J. Collision detection and response for computer animation³. In: *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1988. p. 289–298. ISBN 0-89791-275-6. Disponível em: <<http://dx.doi.org/10.1145/54852.378528>>.

NVIDIA. *Using Vertex Buffer Object (VBO)*. out 2003. NVidia. Disponível em: <<http://developer.nvidia.com/attach/6427>>.

NVIDIA. *What is CUDA*. 2009. NVidia. Disponível em: <http://www.nvidia.com/object/cuda_what_is.html>.

OPENGL et al. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, 2005. Paperback. ISBN 0321335732. Disponível em: <<http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321335732>>.

PALMER, I. J.; GRIMSDALE, R. L. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, v. 14, n. 2, p. 105–116, 1995. Disponível em: <<http://dx.doi.org/10.1111/1467-8659.1420105>>.

PEDDIE, J. *NVIDIA Continues to Gain Graphics Market Share, AMD Keeps on Downfall*. 2007. JPR. Disponível em: <<http://www.xbitlabs.com/news/video/display/20071029062106.html>>.

PONAMGI, M.; MANOCHA, D.; LIN, M. C. Incremental algorithms for collision detection between solid models. In: *SMA '95: Proceedings of the third ACM symposium on Solid modeling and applications*. New York, NY, USA: ACM, 1995. p. 293–304. ISBN 0-89791-672-7.

RICCIO, C.; GUINOT, J. *OpenGL Vertex Buffer Objects*. jan 2007. Ozono3d. Disponível em: <http://www.ozono3d.net/tutorials/opengl_vbo_p1.php>.

SEGAL, M.; AKELEY, K. *The OpenGL Graphics System: A Specification*. ago 2008. The Khronos Group Inc. Disponível em: <<http://www.opengl.org/registry/doc/glspec30.20080811.pdf>>.

SU, C. J. *Enhancing Sense of Reality by Efficient and Precise Collision Detection in Virtual Environments*. 2007. Department of Industrial Engineering e management - Yuan Ze University.

TAKEDA, Y. et al. *A Load Balancing mechanism for large multiprocessor systems and its implementation*. 1988. Proceedings of the International Conference on Fifth Generation Computer Systems. Disponível em: <<http://www.icot.or.jp/ARCHIVE/Museum/FGCS/FGCS88en-proc3/88ePIM-4.pdf>>.

TECHARP. *The Desktop Graphics Card Comparison Guide*. 2009. Techarp. Disponível em: <<http://www.techarp.com/showarticle.aspx?artno=88>>.

WILLIAMS, I. *Efficient rendering of geometric data using OpenGL VBOs in SPECviewperf*. set 2005. SPEC. Disponível em: <http://www.spec.org/gwpg/gpc.static/vbo_whitepaper.html>.

- WILSON, A. et al. Impact: Partitioning and handling massive models for interactive collision detection. In: *of Eurographics*. [S.l.]: Blackwell Publishers, 1999. p. 319–329.
- XIE, K.; YANG, J.; ZHU, Y. M. *Efficient and Accurate Collision Detection Based on Surgery Simulation*. Shanghai, China: Inst. of Image Processing E Pattern Recognition, 2006. Disponível em: <<http://www.springerlink.com/content/h53181vw028j1297/fulltext.pdf>>.
- XU, F.; MUELLER, K. *Realtime 3D computed tomographic reconstruction using commodity graphic hardware*. mai 2007. *Physics in Medicine and Biology*. Disponível em: <<http://www.iop.org/EJ/article/00319155/52/12/006/pmb712006.pdf>>.