Frustum Culling

Rodrigo Setti, Fernando Biesdorf, Douglas Schmidt

Sumário

Resumo

Muitas aplicações baseadas em computação gráfica utilizam o conceito de desenho em tempo real. Mas tudo que deve ser feito no computador em uma grande velocidade, exige também um grande quantidade de processamento. Para minimizar a demanda de cálculos que o processador deve realizar, utilizam-se algumas técnicas, como por exemplo, a técnica de *Frustum Culling*. Esta por sua vez, utiliza-se de cálculos de distância entre planos e pontos para verificar o que deve ser desenhado no monitor do computador ou não. Verificar ponto a ponto também acaba se tornado um processo lento, portanto utilizamos estruturas envoltas aos objetos, como esferas, caixas e outros poliedros. Este trabalho apresenta a técnica de *Frustum Culling* aliado ao uso de *Bounding Spheres*.

Abstract

Many aplications based in graphics computing uses the concept of real time rendering. But every thing that must be made over a computer in high velocity, demands also a great quantity of processing. To mimimalize the calculum demands that the processor must do, we use some tecnincs, like for example, the Frustum Culling. That procedure uses plane to point distance calculum to evaluate what must be drawn over the computer screen and must not. Point by point evaluation ends to be a very slow process, so we use bounding structures over the objects, like spheres, box and other poliedra. This work introduces the Frustum Culling tecnic allied with bounding spheres.

Introdução

Apresentamos uma técnica de otimização aplicada para softwares 3D, chamada de *Frustum Culling* [01], que consiste basicamente em não processar objetos que estejam fora da área de visão da câmera virtual. Damos noções gerais de álgebra vetorial e transformações lineares aplicadas para ambientes tridimensionais, o funcionamento e operação de uma câmera virtual e como os objetos são projetados para serem desenhados corretamente no monitor.

Detalhamos a arquitetura do volume de visão, denominado *Frustum*, como é construído e como será realizado os testes para avaliar se um determinado vértice está inserido neste espaço.

Introduzimos o conceito de sólidos que englobam modelos complexos, mais especificamente o *Bouding Sphere* [02] para testar a intersecção de modelos com o volume do *Frustum* sem testar ponto a ponto.

Em seguida elucidamos os algoritmos e uma forma de otimização para serem inseridos em uma aplicação gráfica. Apresentamos os resultados de testes realizados em uma aplicação que implementamos em duas versões: Utilizando a técnica e não a utilizando. Realizamos comparações entre as versões e discutimos os resultados.

Estado da Arte

O estágio atual de desenvolvimento de técnicas de otimização gráfica tridimensional atualmente impressiona pela qualidade e quantidade de procedimentos para este fim. O *Frustum Culling*, que consiste em processar somente os objetos presentes na área de visão da câmera virtual é uma técnica considerada básica na indústria atual de softwares do ramo.

Entretanto otimizações no algoritmo deste método são sempre bem vindas, tendo em vista que é um procedimento básico e muito utilizado para boa performance dos softwares que o utilizam.

Existem pesquisas atuais que visam melhorar o desempenho da técnica e adicionar recursos como o *Adaptative Occusion Culling* [03] que é uma especialização da técnica que aqui apresentamos. Outro procedimento que podemos citar é o *BackFace Culling* [04] que visa evitar o processamento de objetos obstruídos da visão por outros mais próximos da câmera virtual, bem como outros métodos menos específicos, mas também voltados a área de otimização gráfica tridimensional como o *QuadTree* e *Octree* [05].

1 Introdução à Computação Gráfica

Computadores se tornaram ferramentas poderosas para a geração de imagens de alta qualidade de forma rápida e mais econômica. Com o computador podemos inclusive produzir cenas novas, impossíveis de serem feitas de outra forma. Esses são alguns motivos do sucesso da Computação Gráfica nas últimas décadas e especialmente nos últimos anos. As aplicações em engenharia e ciência têm crescido exponencialmente, a relação de interdependência das tecnologias nesses setores tem se mostrado evidente. Podem-se citar exemplos de áreas que têm se relacionado intimamente com a computação gráfica: medicina, indústria, governo, arte, propagandas, pesquisas espaciais, treinamento, educação e especialmente entretenimento.

As principais áreas da computação gráfica são:

• Aplicações CAD, onde o usuário tem o computador como ferramenta na produção de projetos detalhados de engenharia e construções;

• Representações gráficas, onde as informações podem ser visualizadas de várias formas, facilitando o entendimento do observador;

• Arte, onde várias técnicas são utilizadas para facilitar o trabalho de artistas, como utilização de ambientes virtuais que incentivam a criação;

• Entretenimento. A computação gráfica tem sido a peça chave para a produção de software de entretenimento e no cinema principalmente;

• Educação e Treinamento. O usuário pode ter contato visual com conceitos abstratos e a realidade virtual melhora e reduz os custos de treinamentos,

• Visualização. Facilita a identificação de soluções em diversas áreas, como o fluxo de vento;

• processamento de imagens, a edição automatizada é de grande utilidade no tratamento de imagens de satélite, por exemplo;

• Interfaces gráficas com o usuário. Facilitam muito a manipulação de dispositivos lógicos, tornando muito mais rápido e prático o processamento;

1.1 Vértices

Os vértices são as unidades fundamentais da computação gráfica, qualquer ponto da cena em questão é um vértice. Podemos abstrair um vértice como um vetor de coordenadas x, y e z que define a posição espacial de um ponto qualquer. Os vértices podem ser organizados para formar os mais diferentes modelos, desde formas primitivas como o triângulo até estruturas altamente detalhadas, o nível de detalhe de um modelo geométrico depende do número de vértices que o compõem.

1.2 Objetos

O mundo é composto por inúmeros objetos [06], eles podem e devem ser tratados separadamente para melhor controle dos processos. Muitas vezes precisamos modificar apenas partes da cena sem alterar toda a imagem, definindo cada objeto individualmente como um modelo geométrico fazemos isso mais facilmente. Pois podemos mudar a orientação e posição dos objetos da cena apenas modificando as suas coordenadas globais. Também podemos retirar e colocar objetos na cena sem termos que consertar lacunas na imagem final, assim como mudar o ponto de vista da câmera em busca da melhor perspectiva sem maiores dificuldades.

Ambientes tridimensionais criados por computador podem conter vários tipos de objetos e formados por diversos materiais: metal, plástico, árvore, flores, nuvens, madeira, pedras, papel, montanhas, líquidos etc. Já que estes objetos podem apresentar alto nível de complexidade, foram criadas várias técnicas para representar as peculiaridades de cada objeto da melhor forma possível. Essas técnicas giram em torno de usar modelos geométricos nas representações.

Na programação desses ambientes a representação virtual dos objetos do mundo real é armazenada em uma estrutura central. Cada objeto possui sua estrutura, que pode ser referenciada por um número ou um nome escolhido pelo programador. Essas estruturas contêm as coordenadas tridimensionais de cada vértice do objeto, relações entre as partes do objeto e outras informações necessárias para a aplicação de diversas técnicas de renderização. Como exemplos mais simples dessas informações temos: a cor, o tipo de linha que será utilizada, textura ou algum atributo de transparência.

É por meio dessas estruturas que podemos controlar os objetos do ambiente tridimensional. Esse controle é feito por funções criadas usando linguagens de programação, com essas funções podemos criar, editar ou apagar estruturas integralmente ou em partes.

1.3 Translação

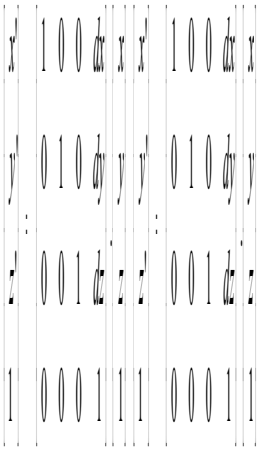
Os vértices de um objeto são relativos ao seu centro de posição, e ele pode mudar, isso representa um movimento de translação [07]. A partir da matriz que representa as coordenadas tridimensionais de cada vértice de um objeto é possível move-lo no espaço virtual. Nesse processo o objetivo é obter uma matriz modificada, com as novas coordenadas do objeto, dessa forma saberemos a nova posição mesmo. Para obtermos a nova matriz de posição multiplicamos a matriz da posição original do objeto por uma outra matriz que especifica, nas três dimensões, a movimentação desejada. O nome da matriz que contém as especificações para a nova posição é matriz transformação. A maneira de obter a matriz transformação será detalhada em um tópico específico sobre operações de matrizes adiante.

1.4 Rotação

Um objeto pode sofrer modificações em sua orientação [08], isso significa que ele pode ser girado em qualquer eixo do espaço. Para girarmos o objeto precisamos mudar as coordenadas de cada vértice do modelo, mantendo a relação geométrica entre todos eles, do contrário o objeto será deformado. Para se efetuar o giro é necessário definir o eixo, que em ambientes tridimensionais pode ser qualquer reta do espaço, e um ângulo. Por convenção, se o valor do ângulo for positivo a rotação será no sentido anti-horário e se for negativo será no sentido horário. Para executar o procedimento o cálculo é similar ao da translação, utilizando a matriz transformação. Utilizamos três matrizes de transformação, elas representam as rotações nos eixos x, y e z. No entanto, como os eixos disponíveis são infinitos, para executar rotações fora dos eixos x, y e z pode-se fazer operações de translação conjuntamente com as operações de rotação. Essas operações requerem muitas considerações, e como o objetivo principal do artigo tem outro enfoque, a rotação em eixos aleatórios não será exposta. Detalhes sobre estes processos podem ser encontrados em alguns títulos pesquisados [08].

1.5 Álgebra com Matrizes e Vetores

Todo o ambiente virtual está sustentado por matrizes e vetores, que contém as coordenadas de todos os vértices e conseqüentemente os modelos. Os algoritmos que controlam o ambiente fazem uso maciço de álgebra linear para executar as diversas transformações necessárias. A translação [09] é um exemplo de álgebra com matrizes e vetores:



Onde x', y' e y' são as novas coordenadas de qualquer vértice situado nas coordenadas x, y e z. Esta é a fórmula pra a translação de um vértice no espaço tridimensional. Os valores de dx, dy e dz representam o deslocamento nos eixos x, y e z respectivamente, se esses valores forem declarados negativos o movimento muda de sentido. Para movimentar um modelo por completo basta realizar esta operação em todos os vértices que o compõem.

Para girar um objeto devemos escolher um eixo e um ângulo θ. Para cada eixo x, y ou z existe uma matriz transformação que controla a rotação [10]. Se o objetivo é girar o objeto em torno do eixo z, executa se a seguinte operação em todos os vértices que o compõem:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\operatorname{sen}\theta & 0 & 0 \\ \operatorname{sen}\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Onde x', y' e y' são as novas coordenadas de qualquer vértice situado nas coordenadas x, y e z. Para girar modelos nos eixos x e y a operação é a mesma, porém a matriz transformação é particular para cada caso:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -sen\theta & 0 \\ 0 & sen\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ Matriz transformação para o eixo x.

$$\begin{bmatrix} \cos\theta & 0 & sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ Matriz transformação para o eixo y.

É necessário operar todos os vértices do modelo para não deformar a forma do mesmo.

Introduzimos neste sub-tópico, alguns procedimentos básicos de modificação de vértices, permitindo a criação de matrizes de mudança de rotação e translação a serem aplicadas nos objetos do mundo virtual.

1.6 Câmera Virtual

Os movimentos da câmera na verdade são modificações em todos os vértices do mundo, ordenados para que seja capturada uma determinada região do mundo virtual.

A câmera virtual possui as mesmas características de uma câmera fotográfica convencional, exceto por ser virtual. Como nas máquina fotográfica convencionais, existe uma abertura de lente, que é responsável pelo ângulo de abertura da imagem, tanto na máquina fotográfica quanto na câmera virtual de um ambiente tridimensional. O filme da máquina seria o plano de projeção, que é paralelo aos eixos x e y.

Quando queremos obter uma nova posição na imagem da câmera, movemos ou rotacionamos todos os modelos da cena, ou seja, a câmera virtual não pode ser movimentada.

A movimentação a partir deste conceito acaba se tornando um pouco abstrata, mas compreensível, visto que a câmera virtual não é um objeto da cena, e sim um orientador de projeção dos modelos. Embutido a este conceito está o de *Loop* de software, mas este será melhor explicado adiante. Por convenção e facilidade, algumas vezes no texto falaremos de “posição da câmera virtual”, mas deve-se subentender que a câmera não se move.

A câmera virtual possui sua visão delimitada por um poliedro que varia de acordo com o tipo de projeção utilizado.

1.7 Projeções

Após a cena ter sido criada com os modelos tridimensionais é necessário converter as coordenadas do ambiente para as coordenadas do plano de projeção [11], ou tela do monitor.

Antes de obtermos as coordenadas de visualização para o monitor precisamos definir a posição, orientação e ângulo de abertura da câmera. A posição da câmera passará a ser o ponto de referência para a projeção. Existem dois tipos básicos de projeção, a paralela e a perspectiva.

Projeção Paralela

A projeção paralela [12] respeita as proporções dos objetos mostrados. Retas partem de cada vértice do objeto paralelamente, a intersecção dessas retas com o plano de projeção é mostrada na saída bidimensional. As retas podem interceptar o plano de projeção de forma perpendicular ou oblíqua. Apesar de a visão não ser realista esse método de projeção é importantíssimo em várias áreas do conhecimento, como na produção de peças, projetos arquitetônicos e outras aplicações CAD.

Projeção Perspectiva

Na projeção perspectiva [13] as linhas de projeção convergem para um ponto, chamado ponto de referência da projeção, esse ponto é onde a câmera está situada. A imagem bidimensional é formada a partir dos pontos de intersecção das linhas de projeção com o plano de projeção. Como o processo de formação da imagem é muito semelhante ao da percepção do olho humano, a imagem nos parece bem realista. Os objetos que estão longe do ponto de convergência aparecem menores e os objetos mais próximos aparecem maiores, essa diferença é diretamente proporcional ao ângulo de abertura da câmera.

1.8 *Rendering*

Quando trabalhamos com ambientes tridimensionais, diversos modelos são desenhados na tela a partir de vértices e linhas. Transformar uma matriz tridimensional com diversos atributos adjuntos ao objeto em uma matriz bidimensional (como tudo é mostrado no monitor) é um processo comumente chamado de *Rendering*.

Diversos softwares de desenho computacional utilizam o processo de *Rendering*, já que é impossível atualmente mostrar no monitor bidimensional uma matriz tridimensional. Modeladores de objetos virtuais tem a facilidade de mostrar ao usuário em tempo real o que ele está desenhando, como os softwares proprietários 3D Studio Max e Auto CAD 3D. O processo de *Rendering* que não é em tempo real, geralmente utilizado em filmes e imagens fotorealisticas, tem um alto custo de processamento e portanto o resultado é agrupado quadro a quadro em um arquivo de vídeo.

1.9 *Real Time Rendering*

Quando utilizamos o processo de *Rendering* em cenas dinâmicas, o mesmo passa a se chamar *Real Time Rendering*, que basicamente significa que o processo de *Rendering* acima descrito é realizado em tempo real.

Mas como temos de realizar um processo geralmente muito custoso em termos computacionais em um tempo ótimo de até 60 vezes por segundo ou mais, algumas técnicas de otimização devem ser utilizadas.

Relativo a parte de otimização, podemos utilizar um conceito simples que não é feito automaticamente durante o processo de *Real Time Rendering*, o corte ou C*ulling* de alguns modelos da cena que não serão vistos na versão final mostrada no monitor do computador.

2 *Frustum*

Em determinados ambientes virtuais tridimensionais, um grande volume de informação deve ser processado para que seja formada a imagem na tela do computador, geralmente devido pelo fato destes ambientes serem extensos ou conterem uma grande quantidade de objetos.

Para que haja algum tipo de limitação, geralmente usa-se um poliedro qualquer para delimitar o limite de visão da câmera virtual e baseando-se no modelo de visão em perspectiva do homem, o poliedro mais usado é um tronco de pirâmide, usualmente chamado de *Frustum* ou *Volume de Frustum,* termos de certa forma equivocados, já que a palavra *Frustum* vem do Latim "Volume", portanto o melhor termo seria apenas *Frustum* ou volume de visão.

Quando usado um tronco de pirâmide para delimitar a quantidade de informações a serem processadas, podemos dizer que estamos usando a técnica de *Frustum Culling*, que será melhor elucidada nos capítulos seguintes.

Usualmente o poliedro delimitador é formado por seis planos, um para delimitar a distância máxima de observação, um para delimitar a distância mínima de observação, dois para delimitarem as partes laterais que não aparecem no monitor e dois para delimitarem as partes superior e inferior que também não aparecem no desenho final no monitor do computador. Mas algumas técnicas utilizam um número maior de planos [14], para por exemplo, subdividir em níveis de distância os diferentes tipos de objetos da ambiente virtual.

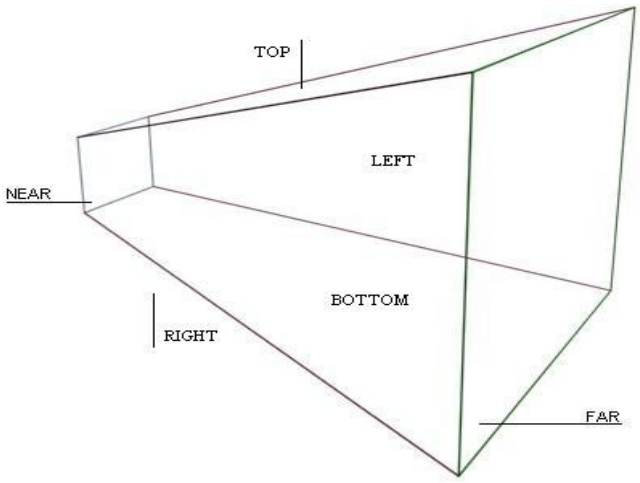


Fig 01. Frustum

3 Intersecção de Modelos no *Frustum*

Cada plano do *Frustum* pode ser representado por uma expressão numérica simples. A partir desta expressão numérica e da posição do nosso objeto no espaço tridimensional, podemos verificar se o objeto está ou não dentro do volume de visão [15].

Todo triângulo ou plano na computação gráfica, deve conter uma orientação especial que diz para que lado o mesmo está virado, essa orientação leva o nome de normal. Se a normal de um dos planos do *Frustum* estiver apontando para dentro do volume de visão, todas devem estar.

Quando é feita a verificação, vemos se a distância central do objeto verificado em relação a normal de cada plano do *Frustum* é positiva ou não. Se esta distância for positiva o objeto está dentro do plano, então é feito o teste para os outros planos, sucessivamente. Caso a distância seja negativa, o objeto certamente está fora do volume de visão e então o mesmo pode ser descartado no processo de desenho.

Algumas técnicas verificam cada ponto ou cada triângulo dos objetos em realação aos planos e retorna com precisão se o modelo está ou não dentro do *Frustum*, mas com um custo computacional maior. Outras técnicas se utilizam de estruturas envoltas, perdendo um pouco da precisão, mas ganhando muito em relação ao tempo de checagem.

As técnicas abordadas por este trabalho se resumem as de *Bounding Mesh, Bounding Box* e *Bounding Sphere,* técnicas muito usadas nos softwares atuais.

3.1 Cálculo de *Bounding Mesh*

Na construção de um mundo virtual, os modelos computacionais que serão usados neste ambiente, são construído a partir de uma forma simples, o triângulo, já que quase toda forma pode ser desenhada utilizando-se triângulos.

Mas um modelo grande, ou com um nível alto de detalhes pode consumir uma boa parte do poder de processamento de um computador. Para solucionar este problema, pode-se facilmente excluir do processo de *rendering* os triângulos que não estão dentro da área de visão da câmera virtual.

Como dito anteriormente, o volume de visão é representado por seis planos. Cada um destes planos tem sua posição no espaço tridimensional bem definida, assim como cada um dos triângulos que formam os objetos da cena. Portanto, através de verificações podemos checar se cada um destes triângulos fazem ou não parte do nosso volume de visão ou *Frustum*.

O problema é que geralmente, em um ambiente grande, têm-se milhões de triângulos formando diversos modelos, logo essa verificação se torna lenta e dependendo dos casos, pode piorar o custo de processamento.

É lógico que se alguns ou vários destes triângulos pudessem ser agrupados e serem verificados de forma conjunta este processo seria extremamente mais eficiente. Deste fato tiramos o conceito de *Bounding Mesh*, que nada mais é que o uso de objetos maiores para "englobar" vários polígonos pequenos. Diversos tipos de estruturas podem ser usadas para esta verificação, como por exemplo esferas e caixas. Outras formas podem ser usadas, mas convém que cubos e esferas e caixas têm um número menor de atributos para serem verificados.

O algoritmo usado neste trabalho utiliza-se da técnica de *Bounding Sphere,*já que esferas são extremamente mais fáceis de serem verificadas e consomem certa de um quarto do tempo em relação ao uso de caixas para cada um dos seis planos do volume de visão.

3.2 Bounding Box

O uso de caixas ou cubos pode ser de grande valia em objetos que não tem uma boa distribuição de massa, por exemplo um lápis. Se utilizarmos uma esfera para encapsular um lápis, corremos o risco do mesmo ser desenhado muito antes de realmente estar dentro do volume de visão, já que o mesmo tem um formato estreito.

Muitos softwares utilizam a técnica de *Frustum Culling* aliada ao uso de *Bounding Box* [16], por exemplo, softwares demonstrativos arquitetônicos, onde praticamente todos os objetos a serem desenhados são retangulares.

Para que a caixa envolta ao objeto atinja o menor tamanho possível, deve-se levar em consideração o vértice mais distante em x, y e z do objeto em relação ao centro do mesmo e o mais próximo, a partir destes valores a caixa que irá "conter" o objeto é gerada.

Com a *Bounding Box* formada e pronto para o uso, o próximo passo é ver se a mesmo está ou não dentro do *Frustum*, para tanto, fazemos uso do conceito de distancia entre um plano e um ponto. Após feita a verificação para todos os oito vértices da *Bounding Box*, um valor de retorno indicará se a mesmo deve ou não ser desenhado.

A verificação não demora a ser feita, mas poderia ser mais rápida se um número menor de pontos tivesse de ser testados. Outra grande desvantagem é em relação à rotatividade do objeto a ser verificado, já que caso ocorra, a caixa envolta também deve rotacionar e para que isso ocorra, a *Bounding Box* deve ser recalculada gerando mais um cálculo que poderia ser evitado caso a estrutura envolta fosse por exemplo, uma esfera.

3.3 *Bounding Sphere*

Não há motivos para se usar uma técnica de encapsulamento de objetos como a do *Bounding Box* em objetos com um número pequeno de triângulos, já que o tempo para checar se cada vértice da caixa está ou não dentro do volume de visão pode ser maior que desenhar o objeto mesmo este não estando perfeitamente dentro do *Frustum.*

Neste caso, a técnica a ser utilizada é a de *Bonding Sphere*, que ao invés de caixas ou cubos, faz uso da esfera para envolver determinado objeto.

Verificar se a esfera está dentro do volume de visão é simples, já que ela atende apenas a dois parâmetros, posição do centro e raio. Logo, se a distância entre a normal de cada plano do *Frustum* e o centro da esfera menos o raio for igual ou positiva, a função verificadora deve retornar para o programa que o objeto deve ser desenhado, se for essa distância for negativa, quer dizer que o objeto não dentro do volume de visão e não precisa ser desenhado. A figura abaixo demonstra este conceito. Vale salientar que alguma das seis verificações a serem feitas retornar uma distância negativa, não é necessário checar o restante dos planos, já que certamente a distância não pode ser positiva para alguns planos e negativa para outros.

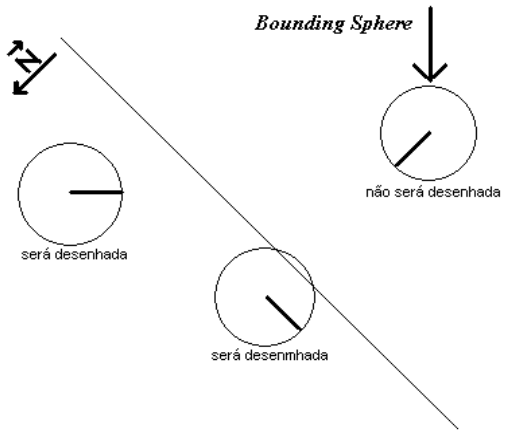


Fig 02 . Bounding Spheres

Um plano simples deve seguir uma equação, representado por:

$$Ax + By + Cz + D = 0;$$

Temos que A, B e C são as componentes normais do plano normal e D é definido usando-se A, B, C e um ponto no plano.

Digamos que o centro da esfera que envolve determinado objeto está posicionada em um ponto qualquer (Scx, Scy, Scz) e o que tem seu raio R definido, a verificação de distância, considerando o plano acima descrito é feita por:

$$A . Scx + B . Scy + C . Scz + D < -R;$$

Esta desigualdade chamada por uma condição retornará um resultado booleano de verdadeiro ou falso, sendo que se o retorno for verdadeiro o objeto está sim, dentro do *Frustum* [02].

A parte de algoritmo será melhor tratada nos tópicos seguintes.

4 O Algoritmo *Frustum Culling*

O algoritmo deve ser inserido corretamente no interior do código da aplicação que utilize o *Frustum Culling*, não é necessário implementar nenhuma classe, apenas algumas funções, entretanto a implementação na forma orientada será muito útil caso se queira trabalhar com mais câmeras e facilmente verificar os objetos que uma ou outra vê.

Nossa estrutura de dados que representa o *Frustum* é um conjunto de seis fórmulas que descrevem os seis planos, a formula do plano – como se sabe – pode ser representada pelos seus coeficientes, que são quatro.

Logo, iremos contruir uma matriz 6 x 4, significando seis planos com quatro valores para representar sua formula.

Os planos do *Frustum* não são facilmente obtidos, ele devem ser uma combinação da forma projeção com as transformações de rotação, translação e escala efetuados no mundo.

4.1 API Gráfica e Linguagem de programação

Os algorimos que escrevemos neste trabalho são genéricos, entretanto dependemos de uma plataforma gráfica para testar o sistema. Escolhemos o *OpenGL* [17] (*Open Graphics Library*), uma biblioteca gráfica poderosa, livre e de código aberto.

O *OpenGL* possui muitas vantagens por ser código aberto, pois um código escrito em sua plataforma pode ser recompilado em diversos sistemas operacionais e funcionar, além de que existe muita documentação e exemplos disponiveis na internet.

A linguagem de programação escolhida, apesar de ser indiferente à estrutura lógica do algoritmo é C++ [18], uma das razões é a sua fácil interface com o *OpenGL* além de ser uma linguagem eficiente para aplicações que necessitam de desempenho como as gráficas.

4.2 Algoritmo de Teste

Em *OpenGL[19]* obtemos facilmente as matrizes de projeção e transformação que necessitamos para calcular o frustum com os comandos:

```
float proj[16];
```

```
  float   modl[16];

  glGetFloatv( GL_PROJECTION_MATRIX, proj );
  glGetFloatv( GL_MODELVIEW_MATRIX, modl );
```

"*glGetFloatv*" é uma função que busca uma variável do sistema tridimensional, no caso, passamos por parâmetro as constantes "*GL_PROJECTION_MATRIX*" e "*GL_MODELVIEW_MATRIX*" que representam a matriz de projeção e a matriz de transformação, respectivamente, e também respectivamente são salvas nos vetores que declaramos "proj" e "modl". Note que por questões de otimização a matriz é transformada em vetor, entretanto, nada que altere o real propósito de nosso algoritmo.

Depois de calculado o frustum, temos que testar se uma determinada esfera(como visto em Bounding Spheres) está interseptando o frustum. Temos, que as normais de todos os planos do frustum apontam para a região do centro do volume, então, se qualquer ponto testado para qualquer formula de um dos planos resultar em um número negativo, temos certeza que este ponto não está contido no frustum.

Isto facilita nosso trabalho, e permite a elaboração de um algoritmo simples;

```
bool PointoNoFrustum( float x, float y, float z, frustum[6][4])
{
  int p;

  for( p = 0; p < 6; p++ )
    if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <= 0 )
      return false;
  return true;
}
```

Esta função verifica se o ponto dado no parametro esta dentro do *Frustum* dado no parametro. Como descrito, nosso frustum está construido em uma matriz 6 x 4, seis planos com quatro números que são os coeficientes da formula do plano.

No laço, contamos os seis plano e testamos os valores do vértice do ponto com os coeficientes das formulas dos planos, e se em algum teste este valor for negativo, temos certeza que este ponto está fora do frustum e retornamos falso a função. Caso o ponto passe em todos os testes, temos então convicção que o ponto está contido no volume.

Para testar uma esfera basta fazer pequenas alterações no algoritmo, adicionando somente o raio, que é somado à formula:

```
bool EsferaNoFrustum( float x, float y, float z, float raio, frustum[6][4] )
{
  int p;

  for( p = 0; p < 6; p++ )
    if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <= -raio )
      return false;
  return true;
}
```

4.2 Extração do *Frustum*

Como já elucidamos, o *Frustum* é um volume, cuja obtenção deve ser feita a partir de uma combinação das matrizes de projeção e de transformação. A biblioteca gráfica *OpenGL* já fornece funções fáceis para obtenção destas matrizes. O *Frustum* é a matriz de projeção transformada de acordo com a matriz de transformação, logo o que temos que fazer é multiplicar a matriz de projeção pela matriz de transformação para obtermos o *Frustum* que queremos. Este será armazenado em nossa estrutura de dados, que é uma matriz 6 x 4.

Esta operação, é claro, não precisa ser feita sempre que um objeto deva ser testado, pois podemos salvar nosso *Frustum* na memória e depois utilizarmos um algoritmo para testar a intersecção de determinado modelo no volume do *Frustum*.

Entretanto é importante frisar que antes de testar determinado modelo devemos ter em mãos as coordenadas dos planos do *Frustum* corretas, ou seja, atualizadas. Isto significa que se uma das matrizes, projeção ou transformação for modificada, é necessário recalcular o *Frustum.*

Geralmente a matriz projeção não é transformada ao longo do jogo, e, se for, é uma boa prática carregar a matriz identidade para as matrizes antes de aplicar as transformações. Transformações acumuladas podem trazer problemas de clareza e entendimento do código.

4.3 Otimização Probabilística

O algoritmo de teste que escrevemos pode ser utilizado tranqüilamente em qualquer aplicação gráfica, muito embora seja possível uma otimização particular para a aplicação especificamente. Em certos casos é muito mais provável que um objeto seja eliminado antes pelo plano *FAR* do *Frustum* do que os planos *UP* e *DOWN,* num ambiente gráfico em que a câmera esteja limitada à um plano horizontal aonde os objetos estejam próximos(um jogo de tiro em primeira pessoa), é mais provável um objeto ser eliminado pelos planos *FAR* e *NEA*R do que os planos *UP* e *DOWN*, cuja a maioria dos objetos estará contida entre eles. Logo, podemos construir um algoritmo otimizado sem utilizar um laço for;

```
bool PointoNoFrustum( float x, float y, float z, frustum[6][4])
{
  int p;

    if( frustum[0][0] * x + frustum[0][1] * y + frustum[0][2] * z + frustum[0][3] <= 0 )
      return false;
    if( frustum[1][0] * x + frustum[1][1] * y + frustum[1][2] * z + frustum[1][3] <= 0 )
      return false;
    if( frustum[2][0] * x + frustum[2][1] * y + frustum[2][2] * z + frustum[2][3] <= 0 )
      return false;
    if( frustum[3][0] * x + frustum[3][1] * y + frustum[3][2] * z + frustum[3][3] <= 0 )
      return false;
    if( frustum[4][0] * x + frustum[4][1] * y + frustum[4][2] * z + frustum[4][3] <= 0 )
      return false;
    if( frustum[5][0] * x + frustum[5][1] * y + frustum[5][2] * z + frustum[5][3] <= 0 )
      return false;

  return true;
}
```

As condicionais podem ser mudadas de ordem para se maximizar a chance de a função retornar falso o quanto antes, ou seja, minimizar o número de condicionais testadas para ganhar tempo de processamento.

5 Contexto em Aplicações

Já possuímos neste ponto os dois tipos de algoritmos necessários para a aplicação do *Frustum Culling* em uma aplicação: o cálculo do *Frustum* em si e a verificação de um ponto ou esfera no *Frustum.* Temos que inserir as chamadas para estas funções no código da aplicação de modo a funcionar corretamente e minimizar a execução dos algoritmos para poupar processamento.

É importante notar, como já foi dito, que a cada *Rendering* de um *Frame* o cálculo do *Frustum* deverá ser feito algumas vezes ou apenas uma vez, e, a verificação de modelos no *Frustum* deverá ser feita para cada objeto, entretanto, cada objeto deverá ter uma esfera pré calculada para ser usada como *Bounding Mesh*.

5.1 Modelos com *Bounding*

Explicamos aqui que todos os modelos devem conter o atributo de sua "*Bounding Mesh*", pois ela deverá ser calculada (ou recalculada, se necessário) para cada modelo empregado na aplicação.

Uma forma elegante de tratar objetos renderizáveis num mundo 3D é como instâncias de classes que seguem o padrão da seguinte interface:

```
class OBJETO
{
        –posição
        –esfera
        –Desenhar()
}
```

O membro “esfera” deve conter dois sub-membros: raio e centro, aonde centro é um vértice. Observe que nem sempre o centro da *Bounding Sphere* ótima(de menor raio) corresponderá exatamente ao centro de renderização modelo.

Se o modelo for estático a esfera pode ser calculada somente uma vez, caso contrário ela poderá ser recalculada a cada alteração no modelo, ou então calculada uma vez de forma que de liberdade para qualquer modificação possivel no modelo, geralmente isso significa um raio maior.

5.2 Loop da Aplicação

Observe o pseudo-código tipo C de um loop de desenho de uma aplicação arbitraria que utilize *Frustum Culling*:

```
Loop de Desenho da aplicação
{
        Carregar identidade para a matriz projeção
        Carregar identidade para a matriz transformação

        Aplica transformações na matriz projeção
        Aplica transformações na matriz transformação

        Calcula Frustum

        Loop de desenho dos objetos
        {
                Se (objeto no Frustum)
                        Desenhar objeto
        }
}
```

É importante ressaltar que somente em casos especiais haverá transformações dinâmicas na matriz projeção, podendo ser eliminada do esquema dado, e também, obviamente antes de iniciar o loop de desenho as esferas dos objetos devem estar calculadas.

6 Relatório de Desempenho

O desempenho de nossa aplicação tridimensional "Bosque" foi medido a partir do número de *Frames* por segundo renderizados.

O número de *Frames* renderizados por segundo foi deixado "livre" no programa, para que ele renderize o máximo possível no quanto dispuser de tempo para isso.

Foi utilizado um outro programa auxiliar para medir a taxa de FPS renderizados, fizemos isso para que o sistema de medida de FPS não interfira na própria aplicação ou vice-versa. O software que utilizamos é o *Fraps*[19]. Este programa é executado em segundo plano e captura a taxa de FPS da aplicação corrente, grava vídeos e imagens estáticas.

Sobre o aplicativo 3D de teste:

O aplicativo que implementamos e utilizamos para o teste de desempenho é um mundo virtual, especificamente um "bosque" com objetos diversos dentre árvores, gramas, pedras e pinheiros. O usuário pode navegar com a câmera no ambiente livremente, o mundo é circunavegável, isto significa que o jogador nunca encontrará o fim do cenário.

Hardware utilizado:

Rodamos o teste em um desktop com processador AMD K-7 3000+, com memória DIMM-DDR de 512 megabytes e uma placa de vídeo aceleradora 3D MSI Gforce 128 megabytes FX 5600 XT.

Resultados:

O aplicativo foi rodado em dois módulos: Utilizando a técnica de *Frustum Culling* e não a utilizando.

Temos que:

Utilizando *Frustum Culling*	FPS: 59
Não utilizado *Frustum Culling*	FPS: 08

Constatamos que houve um considerável incremento na taxa de FPS com a utilização da técnica de *Frustum Culling* em nossa aplicação, de fato houve um aumento de 73,75% na taxa de FPS.

7 Considerações Finais

Durante a pesquisa do projeto em encontrar definições básicas muito utilizadas em computação gráfica e que julgamos necessária uma explanação detalhada. A dificuldade de encontrar tais definições se deve à grande popularidade dos termos já consagrados na área.

Durante a implementação dos algoritmos nos deparamos com dificuldades técnicas relativas ao simples uso da biblioteca gráfica *OpenGL*, criação de modelos tridimensionais. Para isso tivemos que realizar pesquisas complementares para organizar e implementar com segurança os programas que utilizamos para teste e para o seminário.

8 Conclusões

Depois de uma pesquisa sobre nosso objeto de estudo que é a técnica de *Frustum Culling*, verificamos através de testes a importância de seu uso para a otimização de aplicativos gráficos tridimensionais.

Possíveis futuros projetos relacionados:

***Backface Culling* [4]:**

Em nosso trabalho percebemos uma possível otimização adicional. Em muitos casos existem objetos que estão dentro do *Frustum*, entretanto não são desenhados no monitor porque estão ocultos da câmera por outro objeto. Esta técnica já é utilizada em alguns softwares, propomos realizar uma pesquisa sobre o assunto, implementar programas e realizar testes.

9 Referências

[1] SÁNCHEZ, Daniel e DALMAU, Crespo. *Core tecnics and algorithms in game programming*. pg 359. Indiana : NRG, 2004.

[2] SÁNCHEZ, Daniel e DALMAU, Crespo. *Core tecnics and algorithms in game programming*. pg 364. Indiana : NRG, 2004.

[3] MIDIA, R. Charle. *Graphics Programming methods*. pg. 398. 1 ed. Jeff Lander, 2003.

[4] MIDIA, R. Charle. *Graphics Programming methods*. cap. 2. 1 ed. Jeff Lander, 2003.

[5] SÁNCHEZ, Daniel e DALMAU, Crespo. *Core tecnics and algorithms in game programming*. pg 388. Indiana : NRG, 2004.

[6] BAKER M, Pauline e HEARN, Donald. *Computer graphics - C Version. cap 10.* 2.ed. New Jersey: Prentice Hall, 1997.

[7] BAKER M, Pauline e HEARN, Donald. *Computer graphics - C Version. Cap 11.1.* 2.ed. New Jersey: Prentice Hall, 1997.

[8] BAKER M, Pauline e HEARN, Donald. *Computer graphics - C Version. Cap 11.2.* 2.ed. New Jersey: Prentice Hall, 1997.

[9] STAHLER, Wendy. *Beginning Math and Phisics for Game Programming*. Cap 6. Indiana : NRG, 2004.

[10] STAHLER, Wendy. *Beginning Math and Phisics for Game Programming*. Cap 6, cap 14. Indiana : NRG, 2004.

[11] BAKER M, Pauline e HEARN, Donald. *Computer graphics - C Version. Cap 12.3.* 2.ed. New Jersey: Prentice Hall, 1997.

[12] BAKER M, Pauline e HEARN, Donald. *Computer graphics - C Version. Pag 439.* 2.ed. New Jersey: Prentice Hall, 1997.

[13] BAKER M, Pauline e HEARN, Donald. *Computer graphics - C Version. Pag 443.* 2.ed. New Jersey: Prentice Hall, 1997.

[14] BORMANN, *Karsten. An adaptive occlusion culling algorithm for use in large* ves. Denmark Technical University University of Nottingham, Graphical Communication, IFP Communications Research Group, CS. 2003.

[15] STAHLER, Wendy. *Beginning Math and Phisics for Game Programming*. Cap 2. Indiana : NRG, 2004.

[16] IONES, A.,ZHUKOV, S., KRUPKIN A. *On Optimality of OBBs for Visibility Tests for Frustum Culling, Ray Shooting and Collision Detection*. St.-Petersburg State Technical University (Russia), Applied Mathematics Department, CREAT Studio (USA, SF).

[17] SHEREINER, Dave; WOO, Mason; NEIDER, Jackie; DAVIS, Tom. *OpenGL programming guide: The official guide to learning opengl, Version 1.4.* 4 ed. Addison-Wesley, 2003.

[18] KUNDALIA, *Nishant. Basis of c++ programming*. Fire Wall Media 2002.

[19] Fraps – Software. Disponível em http://www.fraps.com/, *acessado em 21/11/2005.*

10 Glossário

Algoritmo:

Um algoritmo é uma seqüência finita e não ambígua de instruções para solucionar um problema computável. Algoritmos podem ser implementados por programas de computadores. (A palavra *algoritmo* tem origem no nome do matemático persa *Al-Khwarizmi*). O conceito de algoritmo é freqüentemente ilustrado pelo exemplo de uma receita, embora muitos algoritmos sejam mais complexos. Eles podem repetir passos (fazer interações) ou necessitar de decisões, tais como comparações ou lógica, até que a tarefa seja completada. Um algoritmo corretamente executado não irá resolver um problema se o algoritmo estiver incorreto ou não for apropriado ao problema.

***API*:**

Application Programming Interface (Aplicação de Interface de Programação) ou simplesmente API é um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades. De modo geral, a API é composta por uma série de funções acessíveis somente por programação, e que permitem utilizar características do software menos evidentes ao usuário tradicional.

Bounding Box:

Pela tradução direta do inglês para o português, significa "caixa envolta", essa por sua vez é usada para englobar objetos mais complexos, como por exemplo, uma árvore virtual. Esta técnica é usada principalmente para verificar colisões e geralmente é muito eficiente, pois uma caixa ou um cubo possuem poucas retas ou planos para serem verificados.

Bounding Mesh:

Também traduzido diretamente do inglês significa "polígono envolto", mas é também um termo mais genérico, pois até um cubo ou caixa é um polígono, portanto deve-se utilizar o termo *bounding mesh* quando o tipo de estrutura que irá envolver o objeto não foi definido, por exemplo, podemos utilizar uma esfera ou um cubo para envolver uma árvore, depende da necessidade.

Bounding Sphere:

Do mesmo modo que podemos utilizar caixas ou cubos para envolver um objeto virtual, pode-se ser usado círculos ou esferas. O termo *Bounding Sphere* que significa "esfera envolta" na nossa língua, consiste numa das melhores técnicas para verificação em ambientes de realidade virtual, por ser mais simples e ter um menor custo de processamento.

CAD:

Computer Aided Design (Desenho Auxiliado por Computador), é o nome genérico de sistemas computacionais utilizados pela engenharia, geologia, arquitetura e *design* para facilitar o projeto e desenho técnicos.

Código Fonte:

Um conjunto de palavras escritas de forma ordenada, contendo instruções em uma das linguagens de programação existentes no mercado, de maneira lógica. Depois de compilado,

transforma-se em software, ou seja, programas executáveis. Este conjunto de palavras, que formam linhas de comandos, deverá estar dentro da padronização da linguagem escolhida, obedecendo a critérios de execução. Atualmente com a diversificação de linguagens, o código pode ser escrito de forma totalmente modular, podendo um mesmo conjunto de códigos ser compartilhado por diversos programas, e até mesmo linguagens.

Frame:

Pode ser traduzido como "quadro" em nossa língua. Para cada nova imagem desenhada na tela ou para cada imagem que é atualizada pelo *loop* do programa temos um frame ou quadro. Uma união de *frames* pode compor uma cena dinâmica.

Frame Rate:

Quando unimos diversos *frames* em uma única cena temos uma cena dinâmica, e o Frame Rate, que em nossa língua significa "taxa de quadros", mede de certa forma a quantidade de quadros por segundo. Portanto se temos uma taxa de atualização de 12 quadros por segundo seu Frame Rate será de 12fps (fps vem da abreviação do inglês *frame per second).* Existem diversos padrões de taxas de atualizações, como no cinema, onde o Frame Rate deve se manter a razoáveis 32fps.

***Free Software Foundation*:**

A Free Software Foundation (FSF, Fundação para o Software Livre) é uma organização sem fins lucrativos, fundada em 1985 por Richard Stallman e que se dedicada à eliminação de restrições sobre a cópia, redistribuição, entendimento e modificação de programas de computadores – bandeiras do movimento do software livre, em essência.

Freeware:

Alguns programas de computador, talvez por serem simples, para atenderem à algum tipo de trabalho acadêmico ou por questões filosóficas, são de distribuição gratuita e de comercialização proibida. A maioria destes são de código fechado, mas uma nova tendência no mundo da informática é a união da filosofia de programas *freeware* com a de programas *Open Source* ou de código aberto.

Frustum:

Termo em latim para volume. Na computação gráfica, este volume representa um tronco de pirâmide, que é utilizado pra demarcar o campo de visão de uma suposta câmera virtual. Este poliedro possui quatro faces que convergem para um ponto que anteriormente é limitado por um outro plano chamado de *near* ou plano próximo. A distância máxima é delimitada por um sexto plano chamado de *far* ou plano distante.

Frustum Culling:

Consiste em uma técnica de otimização de performance para desenho de imagens computacionais calculadas em tempo real. A otimização é feita tendo como princípio o fato de que objetos fora do volume do polígono não precisam ser calculados para o desenho, já que não tem necessidade de aparecerem.

GPL:

GPL, sigla de General Public License (Licença Pública Geral), é uma licença de software livre publicada pelo Projeto GNU com a intenção de permitir que software possa ser distribuído de maneira livre, utilizando a filosofia de "deixar copiar".

Loop:

O termo *loop* pode ser traduzido do ingles como "volta". Quase todo tipo de programa de computador possui um mecanismo de *loop*, pois é necessário que o mesmo continue após que todos as linhas de comando forem executadas, seja para voltar a tela inicial ou para desenhar alguma nova imagem. Um *loop* de jogo por exemplo, é fundamental para que a cada *frame* ou quadro de imagem os objetos sejam redesenhados na tela para atualizar possíveis modificações.

Modelo:

Na computação gráfica, modelo é um conjunto de vértices e linhas que formam um objeto virtual. Ao modelo podem ser agrupadas informações de textura e posicionamento espacial.

Open Source:

Termo definido pela Free Software Foundation para um programa que possui código aberto, ou seja, público. Porém, não estabelece certas restrições estabelecidas pela GPL (General Public License).

OpenGL:

A *OpenGL* (*Open Graphics Library*) é uma especificação definindo uma API multi-plataforma e multi-linguagem para a escrita de aplicações capazes de produzir gráficos computacionais 3D (bem como gráficos computacionais 2D). A interface consiste de cerca de 250 funções diferentes que podem ser utilizadas para desenhar cenas tridimensionais complexas. A *OpenGL* é bastante popular na indústria dos videogames e compete diretamente com o *Direct3D* (no *Microsoft Windows*). O *OpenGL* é bastante utilizado em ferramentas CAD, realidade virtual, simulações e visualizações científicas e desenvolvimento no campo dos softwares de entretenimento.

Real time rendering:

O *rendering* em *real time* (tempo real) é o processo de *rendering*, porém feito em tempo real. Útil para animações, jogos, e outras aplicações que trabalhem com mudança constante de objetos e câmera.

Rendering:

Processo de desenhar em um meio bidimensional, como o monitor de um computador, modelos matemáticos usualmente tridimensionais.

Software:

Conjunto de instruções passadas para um computador para que ele execute uma determinada tarefa. É a parte não tangível de um equipamento de processamento de dados. O termo surgiu como gíria no contexto da informática. Já que os equipamentos (computadores e periféricos) ganharam o apelido de "ferragens" ("*hardware*"), os

programas que rodam dentro das máquinas chegaram a ser chamados de "*software*" (jogo de palavras: *hard*/*soft* = duro/mole).

11 Apêndice

Códigos Fonte

```
void CalcularFrustum(float frustum[6][4])
{
  float   proj[16];
  float   modl[16];
  float   clip[16];
  float   t;

  /* pega a matriz PROJEÇÃO do OpenGL */
  glGetFloatv( GL_PROJECTION_MATRIX, proj );

  /* pega a matriz TRANSFORMAÇÃO do OpenGL */
  glGetFloatv( GL_MODELVIEW_MATRIX, modl );

  /* Combina as duas matrizes (multiplica as matrizes projeção e transformação) */
  clip[ 0] = modl[ 0] * proj[ 0] + modl[ 1] * proj[ 4] + modl[ 2] * proj[ 8] + modl[ 3] *
proj[12];
  clip[ 1] = modl[ 0] * proj[ 1] + modl[ 1] * proj[ 5] + modl[ 2] * proj[ 9] + modl[ 3] *
proj[13];
  clip[ 2] = modl[ 0] * proj[ 2] + modl[ 1] * proj[ 6] + modl[ 2] * proj[10] + modl[ 3]
* proj[14];
  clip[ 3] = modl[ 0] * proj[ 3] + modl[ 1] * proj[ 7] + modl[ 2] * proj[11] + modl[ 3]
* proj[15];

  clip[ 4] = modl[ 4] * proj[ 0] + modl[ 5] * proj[ 4] + modl[ 6] * proj[ 8] + modl[ 7] *
proj[12];
  clip[ 5] = modl[ 4] * proj[ 1] + modl[ 5] * proj[ 5] + modl[ 6] * proj[ 9] + modl[ 7] *
proj[13];
```

```
  clip[ 6] = modl[ 4] * proj[ 2] + modl[ 5] * proj[ 6] + modl[ 6] * proj[10] + modl[ 7] * proj[14];
  clip[ 7] = modl[ 4] * proj[ 3] + modl[ 5] * proj[ 7] + modl[ 6] * proj[11] + modl[ 7] * proj[15];

  clip[ 8] = modl[ 8] * proj[ 0] + modl[ 9] * proj[ 4] + modl[10] * proj[ 8] + modl[11] * proj[12];
  clip[ 9] = modl[ 8] * proj[ 1] + modl[ 9] * proj[ 5] + modl[10] * proj[ 9] + modl[11] * proj[13];
   clip[10] = modl[ 8] * proj[ 2] + modl[ 9] * proj[ 6] + modl[10] * proj[10] + modl[11] * proj[14];
   clip[11] = modl[ 8] * proj[ 3] + modl[ 9] * proj[ 7] + modl[10] * proj[11] + modl[11] * proj[15];

   clip[12] = modl[12] * proj[ 0] + modl[13] * proj[ 4] + modl[14] * proj[ 8] + modl[15] * proj[12];
   clip[13] = modl[12] * proj[ 1] + modl[13] * proj[ 5] + modl[14] * proj[ 9] + modl[15] * proj[13];
   clip[14] = modl[12] * proj[ 2] + modl[13] * proj[ 6] + modl[14] * proj[10] + modl[15] * proj[14];
   clip[15] = modl[12] * proj[ 3] + modl[13] * proj[ 7] + modl[14] * proj[11] + modl[15] * proj[15];

  /* extrai números para o plano RIGHT*/
  frustum[0][0] = clip[ 3] - clip[ 0];
  frustum[0][1] = clip[ 7] - clip[ 4];
  frustum[0][2] = clip[11] - clip[ 8];
  frustum[0][3] = clip[15] - clip[12];

  /* Normaliza resultado*/
```

```
   t = sqrt( frustum[0][0] * frustum[0][0] + frustum[0][1] * frustum[0][1] + frustum[0]
[2] * frustum[0][2] );
   frustum[0][0] /= t;
   frustum[0][1] /= t;
   frustum[0][2] /= t;
   frustum[0][3] /= t;

   /* extrai números para o plano  LEFT */
   frustum[1][0] = clip[ 3] + clip[ 0];
   frustum[1][1] = clip[ 7] + clip[ 4];
   frustum[1][2] = clip[11] + clip[ 8];
   frustum[1][3] = clip[15] + clip[12];

   /* Normaliza resultado */
   t = sqrt( frustum[1][0] * frustum[1][0] + frustum[1][1] * frustum[1][1] + frustum[1]
[2] * frustum[1][2] );
   frustum[1][0] /= t;
   frustum[1][1] /= t;
   frustum[1][2] /= t;
   frustum[1][3] /= t;

   /*extrai números para o plano BOTTOM */
   frustum[2][0] = clip[ 3] + clip[ 1];
   frustum[2][1] = clip[ 7] + clip[ 5];
   frustum[2][2] = clip[11] + clip[ 9];
   frustum[2][3] = clip[15] + clip[13];

   /* Normaliza resultado */
   t = sqrt( frustum[2][0] * frustum[2][0] + frustum[2][1] * frustum[2][1] + frustum[2]
[2] * frustum[2][2] );
   frustum[2][0] /= t;
```

```
  frustum[2][1] /= t;
  frustum[2][2] /= t;
  frustum[2][3] /= t;

  /* extrai números para o plano TOP*/
  frustum[3][0] = clip[ 3] - clip[ 1];
  frustum[3][1] = clip[ 7] - clip[ 5];
  frustum[3][2] = clip[11] - clip[ 9];
  frustum[3][3] = clip[15] - clip[13];

  /* Normaliza resultado */
  t = sqrt( frustum[3][0] * frustum[3][0] + frustum[3][1] * frustum[3][1] + frustum[3]
[2] * frustum[3][2] );
  frustum[3][0] /= t;
  frustum[3][1] /= t;
  frustum[3][2] /= t;
  frustum[3][3] /= t;

  /* extrai números para o plano FAR */
  frustum[4][0] = clip[ 3] - clip[ 2];
  frustum[4][1] = clip[ 7] - clip[ 6];
  frustum[4][2] = clip[11] - clip[10];
  frustum[4][3] = clip[15] - clip[14];

  /* Normaliza resultado */
  t = sqrt( frustum[4][0] * frustum[4][0] + frustum[4][1] * frustum[4][1] + frustum[4]
[2] * frustum[4][2] );
  frustum[4][0] /= t;
  frustum[4][1] /= t;
  frustum[4][2] /= t;
  frustum[4][3] /= t;
```

```
  /* extrai números para o plano  NEAR */
  frustum[5][0] = clip[ 3] + clip[ 2];
  frustum[5][1] = clip[ 7] + clip[ 6];
  frustum[5][2] = clip[11] + clip[10];
  frustum[5][3] = clip[15] + clip[14];

  /* Normaliza resultado*/
  t = sqrt( frustum[5][0] * frustum[5][0] + frustum[5][1] * frustum[5][1] + frustum[5]
[2] * frustum[5][2] );
  frustum[5][0] /= t;
  frustum[5][1] /= t;
  frustum[5][2] /= t;
  frustum[5][3] /= t;
}

bool PontoNoFrustum( float x, float y, float z, float frustum[6][4])
{
  int p;

  for( p = 0; p < 6; p++ )
    if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <= 0 )
      return false;
  return true;
}

bool EsferaNoFrustum( float x, float y, float z, float raio, frustum[6][4] )
{
  int p;

  for( p = 0; p < 6; p++ )
```

```
        if( frustum[p][0] * x + frustum[p][1] * y + frustum[p][2] * z + frustum[p][3] <=
-raio )
            return false;
    return true;
}
```