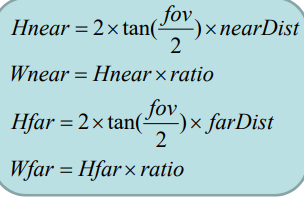
# View Frustum

Para calcular o View Frustum da câmara optamos por utilizar a formula geométrica. Para esta fórmula precisamos de diferentes parâmetros da câmara: o valor do near, do far, o rácio do espaço (largura <-> altura) e o valor do ângulo de visão da mesma.

Estes diferentes campos são importantes visto que o frustum da câmara é parecido ao nosso, ou seja é defeinido por uma pirâmide consoante o ângulo de visão (a abertura da pirâmide é dependente do ângulo). No entanto não é bem uma pirâmide visto que o plano mais distante é o que corresponde ao valor do far e o plano mais aproximado correspondente ao valor do near, sendo uma pirâmide truncada começando em “near” e não em 0. Por isto precisamos destes valores para definir os 8 pontos que vão definir os 6 planos do frustum.

De seguida apresentamos os valores das alturas e larguras para os planos correspondentes ao near e far da câmara (serão planos do frustum).

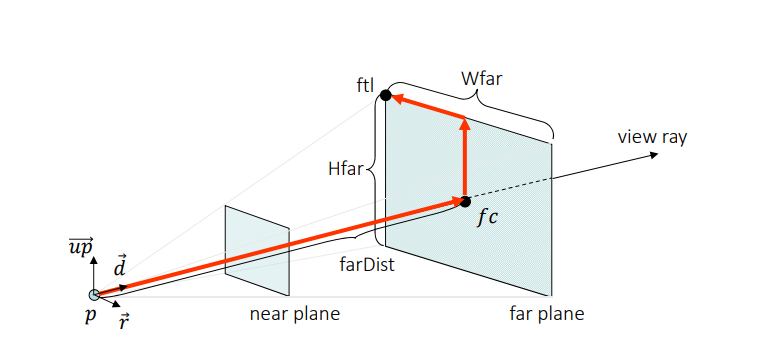
fov corresponde ao ângulo de visão, nearDist e farDist aos valores do near e do far e ratio ao rácio do “ecrã”.



Para calcuar o Frustum é muito fácil perceber que este é dependente da câmara, logo dependerá dos eixos da câmara, pelo que antes de calcular o mesmo tivemos de calcular os eixos da câmara. Utilizamos os seguintes cálculos:

* Z = (Pos – L) <- normalizado
* X = up \* Z <- normalizado, com up = (0,1,0)
* Y = Z \* X <- normalizado
* D = -Z

De referir que como são vetores as operação correspondem a produtos vetoriais.



Depois de definidas os eixos da câmara apenas temos de definir os pontos de controlo pois estes são suficientes para definir os 6 planos.

Analisando a imagem podemos verificar que:

* fc =
* ftl = fc + =

Na imagem up corresponde ao y e r ao x.

Os restantes pontos são: (f->far/n->near) (t->top/b->bottom) (l->left(r->right)

* ftl =
* ftr =
* fbl =
* fbr =
* ntl =
* ntr =
* nbl =
* nbr =

Para calcular cada plano do Frustum precisamos do vetor normal ao mesmo e de um ponto, visto que estes definem o plano. Para esta ser normalizada, basta que o vetor esteja normalizado.

A equação do plano é:

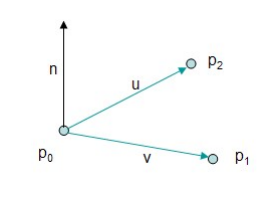
* Ax + By + Cz + D = 0. O vetor normal ao mesmo é (A,B,C).

Para calcular o valor de D apenas necessitamos de um ponto que pertença ao plano e substituir as suas coordenadas dando o resultado:

* p0 -> (pX,pY,pZ)
* -D = A \* pX + B \* pY + C \* pZ = • p0 (produto escalar entre 2 vetores pois o ponto pode ser considerado como um vetor)

Para calcular a normal ao plano, são necessários 3 pontos do mesmo, sendo que esta corresponde ao produto vetorial entre os vetores:

* = p1 – p0
* =p2 - p0
* = \*



Logo para calcular cada plano do frustum apenas precisamos de 3 pontos que pertençam a esse plano. Dos 8 pontos calculados conseguimos criar conjuntos de 3 pontos que vão definir os 6 planos. Cada conjunto de 3 pontos que define o plano são:

* Topo -> (ntr,ntl,ftl)
* Baixo -> (nbl,nbr,fbr)
* Esquerdo -> (ntl,nbl,fbl)
* Direito -> (nbr,ntr,fbr)
* Próximo -> (ntl,ntr,nbr)
* Afastado -> (ftr,ftl,fbl)

De referir que cada um dos planos definidos anteriormente têm a sua normal a apontar para “dentro” do frustum pelo que a distância de um ponto a cada plano será superior a 0 se o ponto estiver “dentro” do frustum na perspetiva desse plano.

A distância de um ponto ao plano, visto que este está normalizado, é apenas o produto escalar entre o ponto e a normal do plano somando a componente d do plano.

* P = (x0,y0,z0)
* dist(p) = • (x0,y0,z0) + d

FALAR SOBRE OS TESTES DA ESFERA E DA CAIXA!

Para verificar que uma figura se situava dentro do frustum ou não, decidimos guardar uma stack de matrizes, que correspondem às matrizes anteriores à atual para que quando se realize glPopMatrix() a matriz atual se atualize como a matriz sem as transformações geométricas desde o glPushMatrix(). Para além desta stack temos também a matriz atual que corresponde à matriz com o espaço global (as transformações realizadas até ao momento). Quando realizamos uma operação do tipo glPushMatrix() criamos uma nova matriz igual à atual e adicionamos essa matriz à stack. Quando realizamos o inverso (popMatrix) retiramos a matriz do topo da stack e atualizamos a matriz atual como essa matriz. A matriz inicial tem de ser a matriz identidade, pois é essa da qual partimos.

Por forma a utilizar estas matrizes, e visto que já existem bibliotecas que implementam as transformações geométricas auxiliamo-nos na biblioteca **glm** que realiza as operações que desejamos (rotações, translações e escalas) tendo apenas atenção que esta nos dá a matriz transposta à que queremos, sendo que quando queremos calcular a matriz atual para verificar onde se encontra um ponto (por forma a verificar se se encontra no frustum ou se a câmara não colidiu com este) temos de obter a transposta desta, bastando para isso apenas inverter os índices (i,j) para (j,i).

Quando testamos se uma Figura pode ser ou não desenhada passamos a matriz atual, pelo que com os limites definidos pelo Frustum fazemos as seguintes operaçãoes:

* caso seja uma esfera então atualizamos o ponto onde se encontra o centro a partir do ponto (0,0,0) que “é o centro por defeito” e utilizamos a função de teste de uma esfera
* caso seja uma caixa atualizamos cada um dos vetores de pontos de controlo correspondentes aos vértices da caixa. De seguida atualizamos um vetor com os mínimos e máximos de cada coordenada (x,y e z) pois serão úteis para a função que calcular a distância de uma caixa a um plano e depois chamamos essa função.
* A atualização de cada ponto é feita multiplicando a matriz do espaço global pela coordenada do ponto!

Inicialmente pensamos em ter um frustum com partições, ou seja um frustum não só para as figuras, mas também para os grupos. Este frustum seria sempre representado por uma Caixa (por facilidade), e começaria com os 8 pontos de controlo iguais e correspondentes ao ponto (0,0,0). Depois sempre que adicionamos uma figura realizamos as operações do grupo (para que a matriz fique com os valores correspondentes ao espaço do grupo) à matriz do grupo, a partir da matriz identidade. Depois atualizamos casa ponto analisando as coordenadas máximas e mínimas da figura da seguinte forma:

* Calculamos as coordenadas com a análise dos valores (se algum é superior para as máximas e se algum é inferior para as mínimas)
* Atualizamos os 8 pontos a partir das novas coordenadas, com os respetivos:
  + (xmin,ymin,zmin)
  + (xmin,ymin,zmax)
  + (xmin,ymax,zmin)
  + (xmin,ymax,zmax)
  + (xmax,ymin,zmin)
  + (xmax,ymin,zmax)
  + (xmax,ymax,zmin)
  + (xmax,ymax,zmax)

Depois seria potencialmente mais eficiente calcular o teste para cada grupo, visto que se um grupo não fosse visível então as figuras que estavam “dentro” do mesmo também não seriam. No entanto não optamos por esta opção por 2 razões:

* A maior parte dos nossos grupos tem poucos figuras e os que têm mais teriam uma caixa de volume muito elevado
* Exige um pré-processamento muito elevado

# Colisões

Para além de termos implementado o View Frustum decidimos também implementar colisões, sendo que para este caso foi mais simples. O mecanismo de colisões que utilizamos foi também para cada figura e neste caso definimos o seu volume como uma esfera em todo o caso. Para testar se existe uma colisão apenas temos de verificar se a nova posição da câmara não se encontra a uma distância do centro da esfera que define o volume da Figura inferior ou igual ao valor do raio da esfera.

No entanto, este algoritmo tem um pequeno problema, que é: se o passo da câmara for demasiado grande, esta pode “atravessar” a esfera e isso não deveria acontecer.

Para evitar isto poderíamos aplicar um no qual passamos a posição anterior da câmara, a vetor que corresponde à direção por onde esta se moveu e o valor do passo. Depois apenas tínhamos de calcular a interseção da esfera com o segmento de reta correspondente ao movimento da câmara, resolvendo o seguinte sistema:

Na qual:

* O ponto (x0,y0,z0) é o centro da esfera
* r é o raio da esfera
* O vetor (a1,a2,a3) é o vetor na qual a câmara se moveu
* O ponto (x1,y1,z1) é a posição da câmara antes de se mover
* n é o valor do passo da câmara

Da mesma forma que para o frustum inicialmente pensamos em aplicar um algoritmo no qual detetávamos colisões com os grupos também, pelo que em cada grupo se o teste desse falso, ou seja a câmara não se pudesse mover teríamos de verificar nos elementos dentro desse grupo o teste. No entanto caso o teste desse verdadeiro já não precisávamos de verificar os elementos do grupo pois estes estávamos contidos no espaço do grupo e não seriam “atingidos”. Pelas mesmas razões não aplicamos este método.