**GLFW** – É uma biblioteca para necessidades básicas de renderização na tela, definindo parâmetros de window e processamento de inputs de teclados, mouse e gamepads.

**Pipeline gráfico** é dividiso em duas grandes partes, a primeira recebe coordenadas 3D e trasnfoma as mesmas em coordenadas 2D para tela. A segunda parte pega as coordenadas 2D e as transformam em pixels coloridos reais na tela.

**Coordenadas Normalizadas** serão então transformadas em coordenadas do espaço da tela através do transformação da janela de visualização usando dados fornecidos na função **glViewport**. As coordenadas de espaço de tela resultantes são então transformadas em fragmentos como entradas para o **Fragmente Shader**.

**Objetos buffer de vértice (VBO)** são utilizados para podermos enviar grandes lotes de dados de uma vez para a placa de vídeo e mantê-los lá se houver memória suficiente, sem ter que enviar dados um vértice de cada vez. ***Isso justifica criar um array de vértices.***

**Objetos de matriz de vértices (VAO)** são utilizados para vincularmos VBOs e quaisquer chamadas subsequentes desse ponto em diante serão armazenadas dentro do VAO. Isso tem a vantagem de que, ao configurar VBOs você só precisa fazer essas chamadas uma vez e sempre que quisermos desenhar o objeto podemos apenas vincular o VAO correspondente. ***Sempre deve haver pelo menos um VAO para podermos desenhar. Ele vai armazenas os VBOs.***

**Objetos buffer de elementos (EBO)** são utilizado para gerar índices, sequências de quais vértices serem ligados, evitando sobreposições de dois vértices. Um VAO armazena o glBindBuffer() quando o destino é GL\_ELEMENT\_ARRAY\_BUFFER . Isso também significa que ele armazena suas chamadas de desvinculação, portanto, certifique-se de não desvincular o EBO antes de desvincular seu VAO, caso contrário, ele não terá um EBO configurado.

**SHADERS**

**Shaders** são pequenos programas executados pelos núcleos de uma GPU.Sempre começam com uma declaração de versão**,** seguida por uma lista de variáveis ​​de in, out e uniform e abaixo o seu main(). O ponto de entrada de cada shader está em seu main(), onde processamos quaisquer variáveis ​​‘in’ e ‘uniform’ e emitimos os resultados em suas variáveis ​​‘out’.

#version version\_number

**in type in\_variable\_name;**

**in type in\_variable\_name;**

**out type out\_variable\_name;**

**uniform type uniform\_name;**

**void main()**

}

// process input(s) and do some weird graphics stuff

...

// output processed stuff to output variable

out\_variable\_name = weird\_stuff\_we\_processed;

}

Declarar uma **variável uniform** **que não é usado em nenhum lugar faz** com que o compilador remova ela silenciosamente da versão compilada, o que é a causa de vários erros frustrantes; mantenha isso em mente!

Vertex Shaders:

* Entradas são os atributos dos vértices. E são obrigatórias.
* Metadados ‘layout (location = \*)’ são obrigatórios nas entradas.
* Número máximo de 16 atributos de 4 componentes.

Fragment Shaders:

* Requer uma saída vec4 de cor.
* Para receber um entrada deve ser setado uma saída no vertex shader com **mesmo nome** e **mesmo tipo**.

Tipo de dado – Vetores

* vec\*: o vetor padrão de floats.
* bvec\*: um vetor de bool.
* ivec\*: um vetor de ints.
* uvec\*: um vetor de n unsigned int.
* dvec\*: um vetor de doubles.

\* representa o número de componentes (1 a 4)

vec.x ou vec.r ou vec.s - para acessar o primeiro componente  
vec.y ou vec.g ou vec.t - para acessar o segundo componente  
vec.z ou vec.b ou vec.p - para acessar o terceiro componente  
vec.w ou vec.a ou vec.q - para acessar o quarto componente

xyzw para quaisquer, rgba para cores e stpq para textura.

**TEXTURAS**

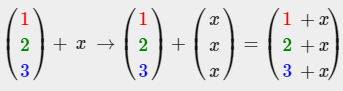
OpenGL deve ter um mínimo de 16 unidades de textura para você usar, as quais você pode ativar usando GL\_TEXTURE0 a GL\_TEXTURE15 . Eles são definidos em ordem para que também possamos obter GL\_TEXTURE8 via GL\_TEXTURE0 + 8 por exemplo, o que é útil quando teríamos que fazer um loop em várias unidades de textura.

**VETORES**

Em sua definição mais básica, vetores são direções e nada mais. Um vetor tem uma direção e uma magnitude(também conhecido como força ou comprimento). Como os vetores representam direções, a origem do vetor não muda seu valor.

**OPERAÇÕES DE VETOR ESCALAR**

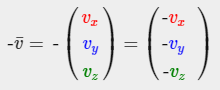
Uma **escalar** é um único digito.



Onde **+** pode ser +, - , ⋅ ou ÷.

**NEGAÇÃO DE VETOR**

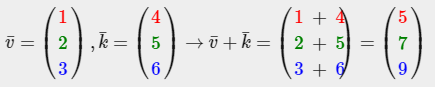
Negar um vetor resulta em um vetor na direção reversa.



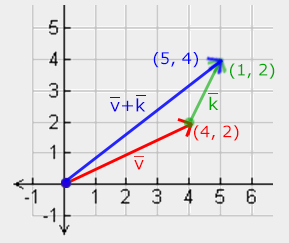
Uma multiplicação de vetor escalar com um valor escalar de -1.

**ADIÇÃO E SUBTRAÇÃO DE VETORES**

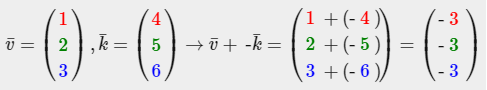
Cada componente de um vetor é adicionado ao mesmo componente do outro vetor.



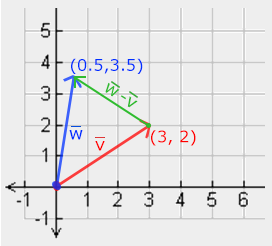
Onde **o segundo vetor é adicionado** **no topo da extremidade do primeiro vetor para encontrar o ponto final do vetor resultante** (método cabeça-a-cauda):



A subtração vetorial é o mesmo que a adição com um segundo vetor negado:

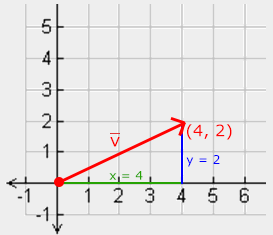


**Subtrair** dois vetores um do outro **resulta em um vetor que é a diferença das posições para as quais os dois vetores estão apontando.** Isso é útil em certos casos em que precisamos recuperar um vetor que é a diferença entre dois pontos.



**COMPRIMENTO**

Para recuperar o comprimento / magnitude de um vetor, usamos o **Teorema de Pitágoras.** Um vetor forma um triângulo quando você visualiza **seu componente x e seu componente y como dois lados de um triângulo.** O comprimento dos dois lados (x, y) é conhecido e queremos saber o comprimento do lado inclinado (v).



Onde ||v|| é o **comprimento do vetor *‘v’***.

**VETOR UNITÁRIO OU NORMAL**

**Seu comprimento é exatamente 1**. Podemos calcular um vetor unitário *‘n’* de qualquer vetor dividindo cada um dos componentes do vetor por seu comprimento:



Esse processo é conhecido como **normalização de vertor**.

*Obs: Facilita muito o trabalho pois não precisamos lidar com o comprimento do vetor restando só sua direção.*

**MULTIPLICAÇÃO DE VETOR-VETOR**

Temos dois casos específicos que podemos escolher ao multiplicar: um é o **produto escalar** denotado como *‘****v’ ⋅ ‘k’*** e o outro é o **produto cruzado** denotado como ***‘v’ × ‘k’***.

**Produto escalar** de dois vetores é igual ao produto escalar de seus comprimentos multiplicado pelo cosseno do ângulo entre eles.



O ângulo entre eles é representado como teta (θ) Por que isso é interessante? Bem, imagine se *‘****v’*** e *‘****k’*** são vetores unitários, então seu comprimento seria igual a 1. Isso efetivamente reduziria a fórmula para:



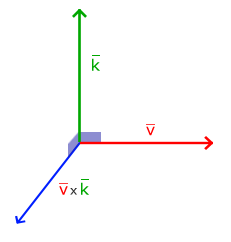
O produto escalar define apenas o ângulo entre os dois vetores. **A função cosseno torna-se 0 quando o ângulo é de 90 graus ou 1 quando o ângulo é 0**. Isso nos permite testar facilmente se os dois vetores são ortogonal ou paralelo uns aos outros usando o produto escalar. O produto escalar é uma multiplicação de componentes em que somamos os resultados. É assim com dois vetores unitários.



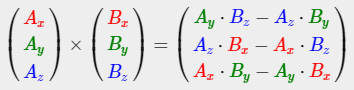
Para calcular o grau entre esses dois vetores unitários, usamos o inverso da função cosseno   
*(cos-1)* e isso resulta em 143.1 graus. Agora calculamos efetivamente o ângulo entre esses dois vetores. O produto escalar se mostra muito útil ao fazer cálculos de iluminação posteriormente.

**PRODUTO CRUZADO**

O **produto vetorial** é definido **apenas no espaço 3D** e leva **dois vetores não paralelos** como entrada e **produz um terceiro vetor que é ortogonal a ambos os vetores** de entrada. Se ambos os vetores de entrada também forem ortogonais entre si, um produto vetorial resultaria em 3 vetores ortogonais.



Produto vetorial entre dois vetores ortogonais A e B:



É necessário entender de **matrizes.**

**MATRIZES**

**| 1 | 2 | 3 |**

**| 4 | 5 | 6 |**

As matrizes são indexadas por (i , j) onde **i está a linha** e **j é a coluna**, por isso a matriz acima é chamada de matriz 2x3 (3 colunas e 2 linhas, também conhecida como dimensões da matriz).

**ADIÇÃO E SUBTRAÇÃO**

A adição e subtração de matrizes entre duas matrizes são feitas por elemento. As mesmas regras se aplicam à subtração da matriz.





**PRODUTOS DE MATRIZ ESCALAR**

Um produto escalar-matriz multiplica cada elemento da matriz por um escalar.



Um escalar basicamente **dimensiona** todos os elementos da matriz por seu valor. No exemplo anterior, todos os elementos foram **escalados por 2**.

**MULTIPLICAÇÃO MATRIZ-MATRIZ**

Você **só pode multiplicar** duas matrizes **se o número de colunas na matriz do lado esquerdo for igual ao número de linhas na matriz do lado direito**. A multiplicação de matrizes não é comutativo isso é **A ⋅ B ≠ B ⋅ A**.

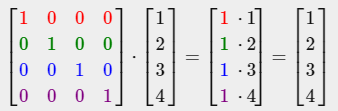


**MULTIPLICAÇÃO MATRIZ-VETOR**

Agora podemos dizer que **um vetor** é basicamente uma matriz ***n* x 1** onde ***n*****é o número de componentes do vetor**. Os vetores são como matrizes um *array* de números, mas com apenas 1 coluna. Há muitas transformações 2D / 3D interessantes que podemos colocar dentro de uma matriz, e multiplicar essa matriz por um vetor transforma esse vetor.

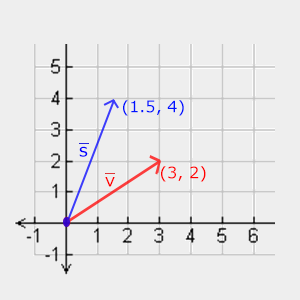
**MATRIZ IDENTIDADE**

A matriz de transformação mais simples que podemos pensar é a **matriz de identidade**. A matriz de identidade é uma *NxN* matriz com apenas 0s, exceto em sua diagonal.



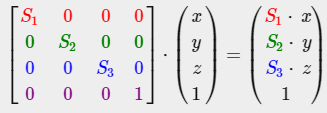
**DIMENSIONAMENTO**

Ao dimensionar um vetor, aumentamos o comprimento da seta na quantidade que gostaríamos de dimensionar, mantendo a mesma direção. Vamos tentar dimensionar o vetor ‘**v’** = ( 3 , 2 ). Vamos dimensionar o vetor ao longo do eixo x em 0.5, tornando-o duas vezes mais estreito; e escalaremos o vetor ao longo do eixo y em 2, tornando-o duas vezes mais alto. Vamos ver como ficaria se escalarmos o vetor ( 0.5, 2) como **‘s’**:



Lembre-se de que o **OpenGL** geralmente **opera no espaço 3D**, portanto, **para** este caso **2D**, **podemos definir a escala do eixo z como 1, deixando-o ileso.**

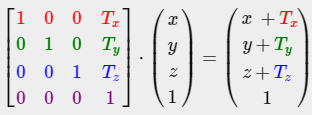
**Uma matriz de transformação que faça o dimensionamento**. Na matriz de identidade cada um dos elementos diagonais foi multiplicado com seu elemento vetorial correspondente. E se mudássemos o 1s na matriz de identidade para 3s? Nesse caso, estaríamos multiplicando cada um dos elementos do vetor por um valor de 3 e, assim, escalar efetivamente o vetor de maneira uniforme por 3. Se representarmos as variáveis ​​de escala como *( S1 , S2 , S3 )* podemos definir uma matriz de escala em qualquer vetor *( x , y , z )* Como:



**O 4º valor de escala 1**. O componente *w* é usado para outros fins.

**TRANSLAÇÃO**

**Translação** é o processo de adicionar outro vetor no topo do vetor original para retornar um novo vetor com uma posição diferente, **movendo** assim o vetor com base em um vetor de translação. Assim como a matriz de escala, existem vários componentes em uma matriz 4 por 4 para realizar certas operações e para translação os 3 valores da 4ª coluna são os utilizados. Se representarmos o vetor de translação como *( Tx , Ty , Tz )* podemos definir a matriz de translação por:



Isso funciona porque todos os valores de translação são multiplicados pela coluna *w* do vetor e adicionados aos valores originais do vetor (regras de multiplicação de matrizes). **Isso não teria sido possível com uma matriz 3 por 3.**

O **componente w** de um vetor também é conhecido como **coordenada homogênea**. Para obter o vetor 3D a partir de um vetor homogêneo, dividimos as coordenadas x, y, z pela sua coordenada w. Normalmente não notamos isso, pois o componente w está é 1. Usar coordenadas homogêneas tem várias vantagens: nos permite fazer translações de matrizes em vetores 3D (sem um componente w não podemos transladar vetores) e também é usado para criar a perspectiva 3D. Além disso, sempre que a coordenada homogênea é igual a 0, o vetor é especificamente conhecido como um vetor de direção já que um vetor com uma coordenada w em 0 não pode ser traduzido.

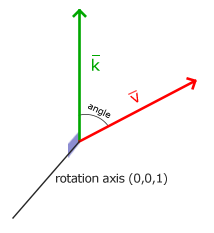
**ROTAÇÃO**

Uma **rotação** em 2D ou 3D é representada por um **ângulo** onde um círculo inteiro tem 360 graus ou 2 radianos PI. A maioria das funções de rotação requer um ângulo em radianos, mas felizmente os graus são facilmente convertidos em radianos:

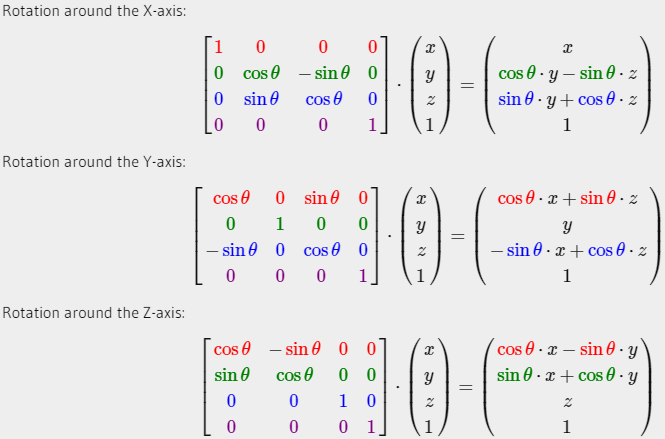
angle in degrees = angle in radians \* (180 / PI)

angle in radians = angle in degrees \* (PI / 180)

Girar meio círculo nos gira 360/2 = 180 graus e girar 1/5 para a direita significa que giramos 360/5 = 72 graus para a direita. Isso é demonstrado para um vetor 2D básico, onde ***‘v’*** é girado 72 graus para a direita, ou no sentido horário, de ***‘k’***:



As rotações em 3D são especificadas com um ângulo e um eixo de rotação. É possível transformar vetores em “vetores recém-girados” dado um ângulo. Isso geralmente é feito por meio de uma combinação inteligente das funções de seno e cosseno. Uma matriz de rotação é definida para cada eixo de unidade no espaço 3D onde o ângulo é representado como o símbolo teta θ.



(Existe a rotação em torno de um eixo 3D arbitrário ---- estudo mais aprofundado)

**MATRIZES DE COMBINAÇÃO**

O verdadeiro poder do uso de matrizes para transformações é que podemos combinar várias transformações em uma única matriz graças à multiplicação matriz-matriz.



Observe que primeiro fazemos uma tradução e, em seguida, uma transformação de escala ao multiplicar as matrizes. **A multiplicação da matriz não é comutativa, o que significa que sua ordem é importante.** Ao multiplicar matrizes, a matriz mais à direita é primeiro multiplicada pelo vetor, portanto, você deve ler as multiplicações da direita para a esquerda.

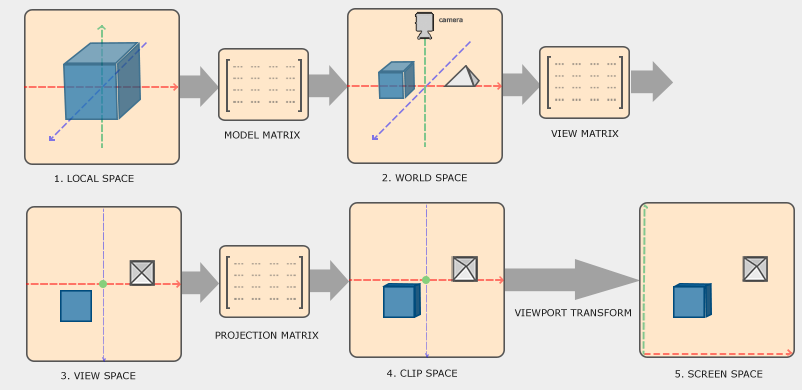
É aconselhável fazer primeiro as operações de escalonamento, depois as rotações e, por último, as translações ao combinar matrizes, caso contrário, elas podem afetar (negativamente) umas às outras.

**SISTEMA DE COORDENADAS**

Há um total de 5 sistemas de coordenadas diferentes que são importantes para nós:

* Espaço local (ou espaço do objeto)
* Espaço mundial
* Espaço visual (ou espaço dos olhos)
* Espaço do clipe
* Espaço da tela

Para transformar as coordenadas de um espaço para o outro é necessário elas passarem por algumas matrizes, sendo elas, **model matrix, view matrix,** e **projection matrix**.



**Local Space –** coordenadas do objeto em relação à sua origem local; são as coordenadas nas quais o objeto começa.

**World Space –** coordenadas relativas a alguma origem global do mundo, junto com muitos outros objetos também colocados em relação à origem deste mundo.

**View Space –** As coordenadas mundiais passam para coordenadas do espaço de visualização, de forma que cada coordenada seja vista da câmera ou do ponto de vista do observador. É o que as pessoas geralmente chamam de Câmera de OpenGL (às vezes também é conhecido como espaço da câmera ou espaço ocular)

**Clip Space –** Coordenadas do clipe são processados para *gama* de -1.0 a 1.0 e determinam quais os vértices vão aparecer na tela. A projeção para as coordenadas de *clip space* pode adicionar perspectiva se usar a *perspective projection*. O que fica de fora do espaço entre -1.0 a 10 é cortado.

**Screen Space** **–** As coordenadas de tela executam um processo chamado transformação da janela de visualização que transforma as coordenadas de -1.0 e 1.0 para o intervalo de coordenadas definido por glViewport. As coordenadas resultantes são enviadas ao rasterizador para transformá-las em fragmentos.

**Model Matrix -** É a matriz de transformação que translada, dimensiona e / ou gira o objeto para colocá-lo no mundo em um local / orientação a que pertencem.

**View Matrix -** Transforma as coordenadas mundiais para visualizar o espaço. É realizado com uma combinação de translações e rotações para transladar / girar a cena de modo que certos itens sejam transformados para a frente da câmera.

**Projection Matrix -** Especifica um intervalo de coordenadas, por exemplo, -1000 e 1000 em cada dimensão, então transforma as coordenadas dentro desse intervalo especificado em coordenadas normalizadas do dispositivo ( -1.0, 1.0 ).

*OBS.: Se apenas uma parte de uma primitiva, por exemplo um triângulo, está fora do volume de corte, o OpenGL irá reconstruir o triângulo como um ou mais triângulos para caber dentro do intervalo de recorte.*

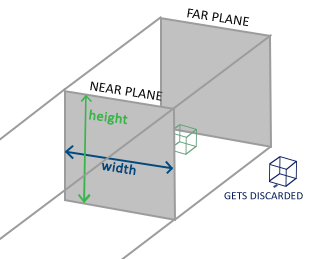
Uma vez que todos os vértices são transformados na *Clip View*, uma operação final é chamada, a *perspective division,* que realiza a divisão dos componentes *(x, y, z)* dos vetores de posição pelo componente *(w)*, a coordenada homogênea do vetor. A *perspective division* é o que transforma as coordenadas 4D do *clip space* em coordenadas 3D de dispositivo normalizado (normais). Esta etapa é executada automaticamente no final da etapa do *vertex shader*.

Após esta fase, as coordenadas resultantes são mapeadas para as coordenadas da tela (usando as configurações de glViewport) e se transformadas em fragmentos.

A *projection matrix* para transformar as coordenadas do *view space* em coordenadas de *clip space* geralmente assumem **duas formas diferentes**, onde cada forma define seu próprio *frustum*. Podemos criar uma **matriz de projeção ortográfica** ou uma **matriz de projeção perspectiva**.

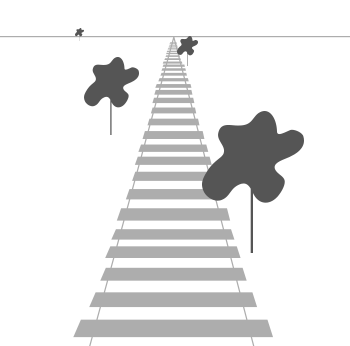
**PROJEÇÃO ORTOGRÁFICA**

Uma **matriz de projeção ortográfica** define uma caixa de *frustum* semelhante a um cubo que define que o *clip space* de cada vértice fora desta caixa é recortado. Ao criar uma matriz de projeção ortográfica é especificado uma *width*, uma *height* e um *near* e *far plane* visíveis do *frustum*. Todas as coordenadas dentro do *frustum* irão ficar dentro do intervalo NDC após serem transformadas pela matriz e, portanto, não serão cortadas. O *frustum* se parece um pouco com um recipiente.

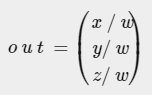


Uma **matriz de projeção ortográfica** mapeia diretamente as coordenadas para o plano 2D que é a tela, mas produz resultados irreais pois esse tipo de projeção não leva perspectiva em consideração.

**PROJEÇÃO PERSPECTIVA**



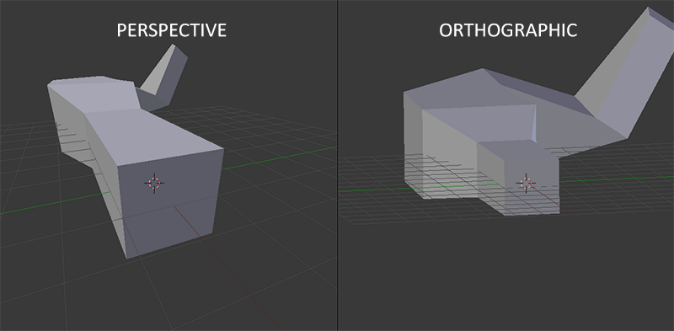
Devido à perspectiva, as linhas parecem coincidir a uma distância suficiente. Este é exatamente o efeito que a projeção da perspectiva tenta imitar e faz isso usando um **matriz de projeção perspectiva.**  A matriz de projeção perspectiva não só mapeia o intervalo de corte do *frustum* para o *clip space*, como também manipula o componente *(w)* de cada coordenada do vértice de tal forma que quanto mais longe uma coordenada de vértice estiver do observador, maior será o valor do componente *(w)*. Neste tipo de projeção, as coordenada ficam no intervalo do *frustum* de *-w* a *w* e tudo que está fora deste intervalo é cortado ao passar e então, em seguida é feita a normalização das coordenadas para o intervalo de -1.0 a 1.0 exigidos pelo OpenGL pela *perspective division* para passar para *clip space.*



Um *frustum* de perspectiva pode ser visualizado como uma caixa de formato não uniforme onde cada coordenada dentro desta caixa será mapeada para um ponto no *clip space*. Especifica-se um *fov* que define o quão grande será o campo de visão(normalmente 45 graus para resultados mais realistas), também se define a proporção da imagem dividindo largura pela altura além de valores para *near* e *far* *plane* (normalmente definidos em 0.1 e 100.0).

*OBS.: Sempre que o valor near da matriz de perspectiva é definido muito alto (como 10.0), o OpenGL irá cortar todas as coordenadas perto da câmera (entre 0.0e 10.0), o que pode dar um resultado visual onde pode se ver através de certos objetos ao se mover desconfortavelmente perto deles.*

COMPARATIVO

Com a **projeção perspectiva** os vértices mais distantes e parecem muito menores (w > 1), enquanto na **projeção ortográfica** cada vértice tem a mesma distância para o usuário (w = 1) .

**JUNTANDO TUDO SOBRE SISTEMAS DE COORDENADAS**

Criamos uma matriz de transformação para cada uma das e uma coordenada de vértice é então transformada em coordenadas de *clip view* da seguinte forma:

***Vclip = Mprojection ⋅ Mview ⋅ Mmodel ⋅ Vlocal***

*OBS.: A ordem da multiplicação da matriz é invertida (precisamos ler a multiplicação da matriz da direita para a esquerda). O vértice resultante deve então ser atribuído a gl\_Position no vertex shader e o OpenGL executará automaticamente a perspective division e o corte para clip view. A saída do vertex shader requer coordenadas do clip space, que é o que acabamos de fazer com as matrizes de transformação. O OpenGL então realiza a perpective division nas coordenadas do clip space para transformá-las em coordenadas de dispositivo normalizadas . OpenGL então usa os parâmetros de glViewPort para mapear as coordenadas do dispositivo normalizado para as coordenadas da tela, onde cada coordenada corresponde a um ponto na tela (screen space). Esse processo é chamado de transformação da janela de visualização.*