# Homogeneous Coordinates

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

# Homogeneous Coordinates: Adding a 4th Value to an XYZ Triple

We usually think of a 3D point as being represented by a triple: **(x,y,z)**.

Using homogeneous coordinates, we add a 4th number: **(x,y,z,w)**

A graphics system, by convention, performs transformations and clipping using (x,y,z,w) and then divides x, y, and z by w before it uses them.

$$X = \frac{x}{w}, Y = \frac{y}{w}, Z = \frac{z}{w}$$

Thus (1,2,3,1) , (2,4,6,2) , (-1,-2,-3,-1) all represent the same 3D point.

When you write:

**glVertex3f( x, y, z );**

OpenGL really calls:

**glVertex4f( x, y, z, 1. );**

# This Seems Awkward – Why Do It?

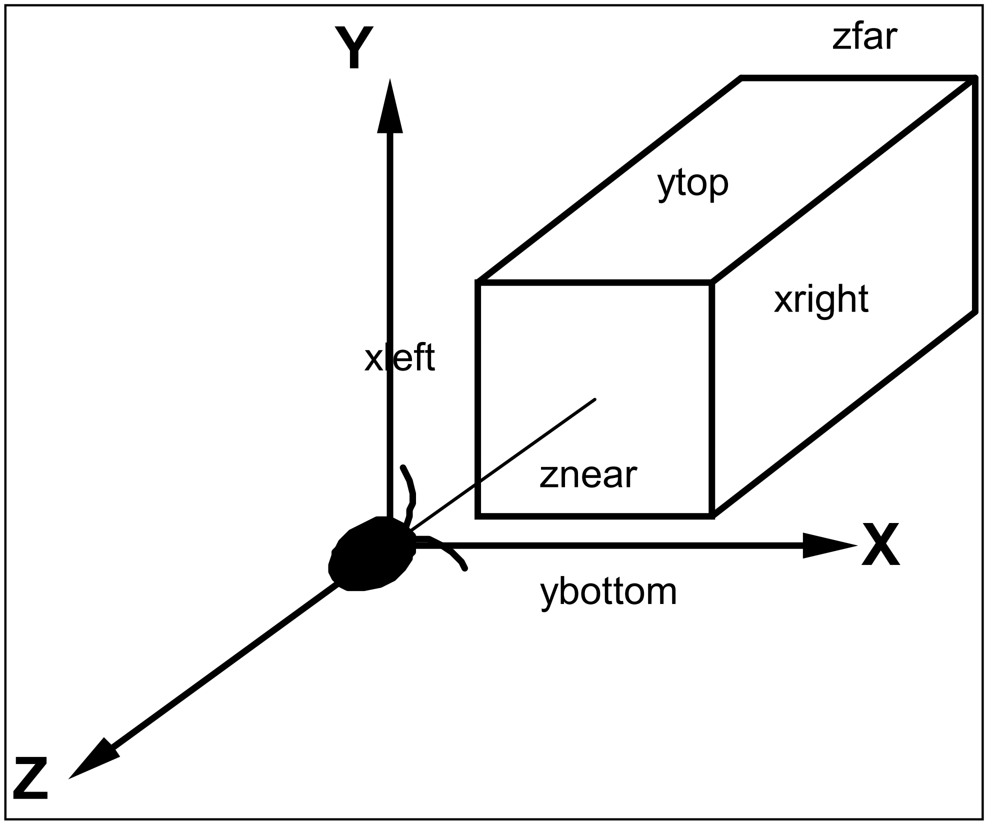
One reason is that it allows for perspective division within the matrix way of doing things. The OpenGL call *glFrustum( left, right, bottom, top, near, far )* creates this matrix:

$$\begin{Bmatrix} x' \\ y' \\ z' \\ w' \end{Bmatrix} = \begin{bmatrix} \frac{2 \cdot near}{right - left} & 0 & \frac{right + left}{right - left} & 0 \\ 0 & \frac{2 \cdot near}{top - bottom} & \frac{top + bottom}{top - bottom} & 0 \\ 0 & 0 & \frac{-(far + near)}{far - near} & \frac{-2 \cdot far \cdot near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{Bmatrix} x \\ y \\ z \\ 1 \end{Bmatrix}$$

This gives w' = -z, which is the necessary divisor for perspective.

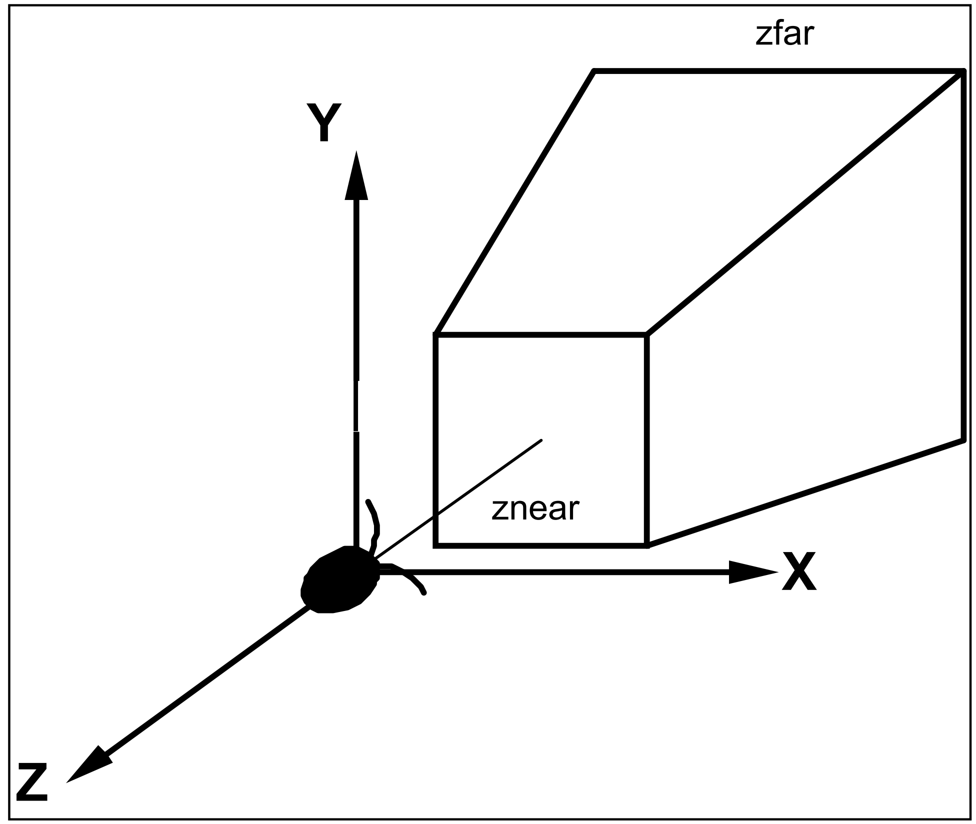
# How the Viewing Volumes Look from the Outside

**glOrtho( xl, xr, yb, yt, zn, zf );**




Parallel/Orthographic

**glFrustum( xl, xr, yb, yt, zn, zf );**




Perspective

OpenGL treats the eye as being at the origin looking in **-Z**

# The Effect of the Perspective Projection Matrix

**glFrustum( left, right,   bottom, top,   near, far );**

$$
\begin{Bmatrix} x' \\ y' \\ z' \\ w' \end{Bmatrix} =
\begin{bmatrix}
\frac{2 \cdot near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\
0 & \frac{2 \cdot near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\
0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2 \cdot far \cdot near}{far-near} \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{Bmatrix} 0. \\ 0. \\ -near \\ 1 \end{Bmatrix}
=
\begin{Bmatrix} 0. \\ 0. \\ -near \\ near \end{Bmatrix}
=
\begin{Bmatrix} 0. \\ 0. \\ -1. \end{Bmatrix}
$$

$$
\begin{Bmatrix} x' \\ y' \\ z' \\ w' \end{Bmatrix} =
\begin{bmatrix}
\frac{2 \cdot near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\
0 & \frac{2 \cdot near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\
0 & 0 & \frac{-(far+near)}{far-near} & \frac{-2 \cdot far \cdot near}{far-near} \\
0 & 0 & -1 & 0
\end{bmatrix}
\begin{Bmatrix} 0. \\ 0. \\ -far \\ 1 \end{Bmatrix}
=
\begin{Bmatrix} 0. \\ 0. \\ far \\ far \end{Bmatrix}
=
\begin{Bmatrix} 0. \\ 0. \\ +1. \end{Bmatrix}
$$

# While We're At It: The Effect of the Orthographic Projection Matrix



**glOrtho( left, right, bottom, top, near, far );**

$$\begin{Bmatrix} x' \\ y' \\ z' \\ w' \end{Bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} 0. \\ 0. \\ -near \\ 1 \end{Bmatrix} = \begin{Bmatrix} 0. \\ 0. \\ -1. \\ 1. \end{Bmatrix} = \begin{Bmatrix} 0. \\ 0. \\ -1. \end{Bmatrix}$$

$$\begin{Bmatrix} x' \\ y' \\ z' \\ w' \end{Bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} 0. \\ 0. \\ -far \\ 1 \end{Bmatrix} = \begin{Bmatrix} 0. \\ 0. \\ 1. \\ 1. \end{Bmatrix} = \begin{Bmatrix} 0. \\ 0. \\ +1. \end{Bmatrix}$$

# The Effect of the Projection Matrices

Both projection matrices are designed to take:

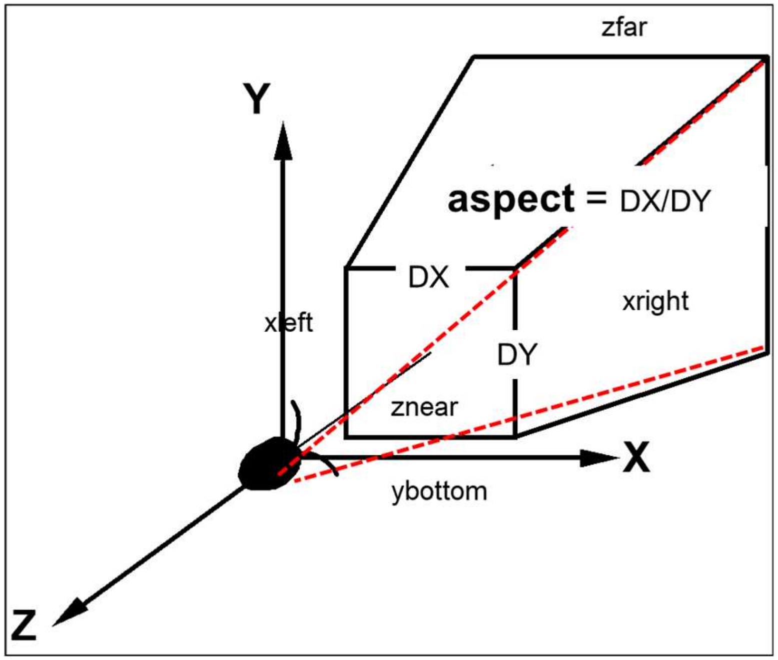
- The range of ***left*** **≤ x ≤** ***right*** and map it to **-1. ≤ x' ≤ +1.**
- The range of ***bottom*** **≤ y ≤** ***top*** and map it to **-1. ≤ y' ≤ +1.**
- The range of ***-near*** **≤ z ≤** ***-far*** and map it to **-1. ≤ z' ≤ +1.**

So, the effect of each OpenGL projection matrix is to project and to scrunch the scale of the scene into a box of size (-1.,-1.,-1.) to (+1.,+1.,+1.).

This is called **Normalized Device Coordinates**.

# Wait -- where does gluPerspective( ) come into all of this?






```
void
gluPerspective( float fovy, float aspect, float near, float far )
{
        // tangent of the y field-of-view angle:

        float tanfovy = tan( fovy * (M_PI / 180.) / 2. );

        // the top and bottom boundaries come from near:

        float top       = near * tanfovy;
        float bottom = -top;

        // the left and right boundaries come from the x/y aspect ratio:

        float right =  aspect * top;
        float left    =  aspect * bottom;

        // ask for a viewing volume in terms of glFrustum:

        glFrustum( left, right,   bottom, top,   near, far );
}
```

# Another Reason to have Homogeneous Coordinates is to be able to represent Points at Infinity

This is useful to be able specify a **parallel light source** by placing the light source location at infinity.

The point (1,2,3,1) represents the 3D point (1,2,3)

The point (1,2,3,.5) represents the 3D point (2,4,6)

The point (1,2,3,.01) represents the point (100,200,300)

So, (1,2,3,0) represents a point at infinity, but along the ray from the origin through (1,2,3)

Points-at-infinity are used for parallel light sources and some shadow algorithms

# However, when Using Homogeneous Coordinates, You Sometimes Just Need to be able to get a Vector Between Two Points

To get a vector between two homogeneous points, we subtract them:

$$(x_b, y_b, z_b, w_b) - (x_a, y_a, z_a, w_a) = \frac{(x_b, y_b, z_b)}{w_b} - \frac{(x_a, y_a, z_a)}{w_a}$$

$$= \frac{(w_a x_b, w_a y_b, w_a z_b) - (w_b x_a, w_b y_a, w_b z_a)}{w_a w_b}$$

Fortunately, most of the time that we do this, we only want a ***unit vector*** in that direction, not the full vector. So, we can ignore the denominator, and just say:

$$\hat{v} = normalize(w_a x_b - w_b x_a, w_a y_b - w_b y_a, w_a z_b - w_b z_a);$$

```
vec3
VectorBetween( vec4 a, vec4 b )
{
   return normalize(  vec3( a.w*b.x – b.w*a.x  ,  a.w*b.y – b.w*a.y  ,  a.w*b.z – b.w*a.z )  );
}
```

However, to save space in the sample code, these notes will assume that w = 1.