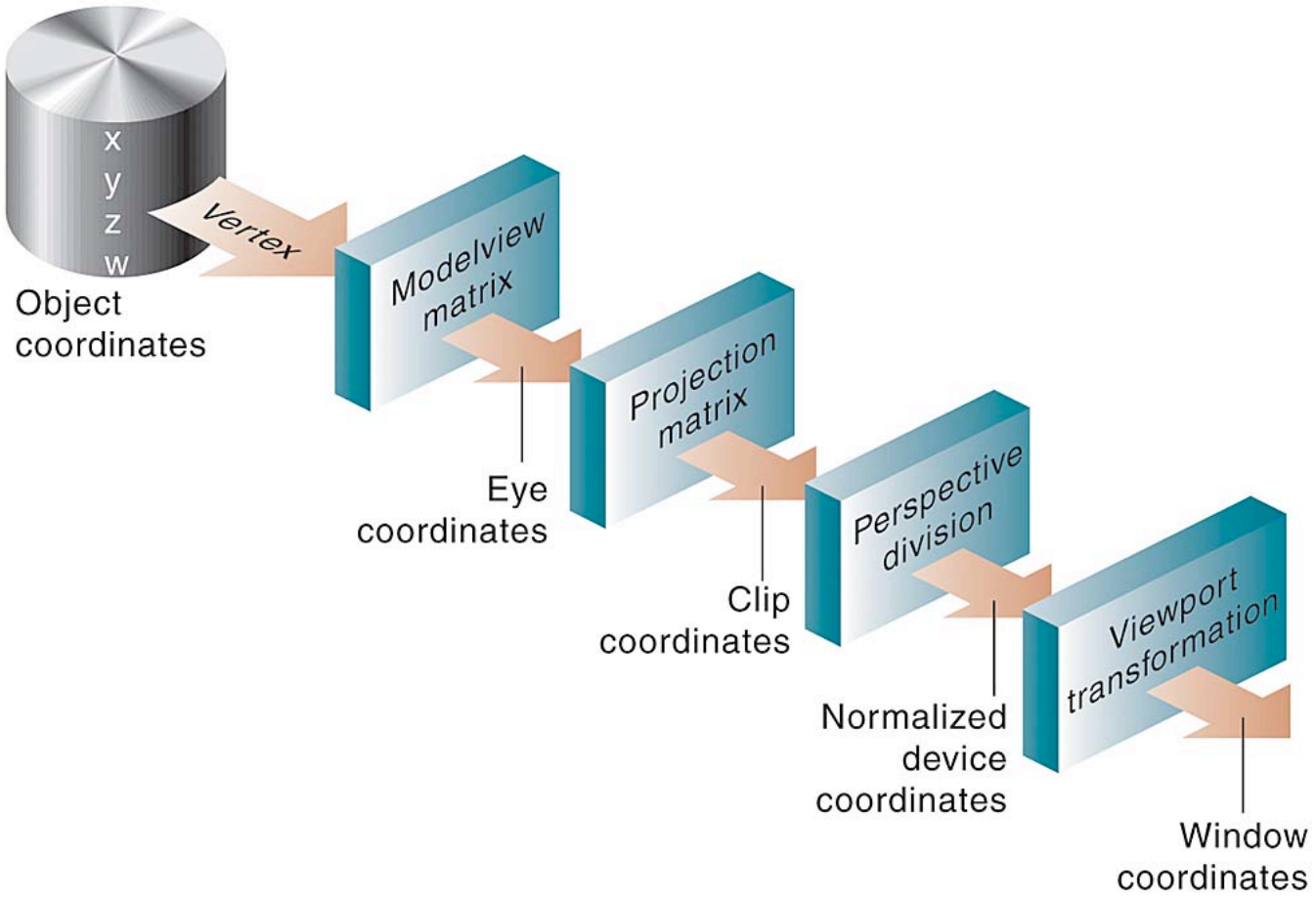
Overview

By end of the week:

- Know the basics of git
- Make sure we can all compile and run a C++/ OpenGL program
- Understand the OpenGL rendering pipeline
- Understand how matrices are used for geometric transformations
- Understand how the projection from 3D to 2D is encoded in a matrix

OpenGL – Vertex Transformation

Moving a point in 3D space to a 2D screen…



OpenGL – Coordinate systems

The Object or Local coordinate system is defined in terms of the Geometry itself. The origin is usually the center or the lower-left of the object.

The Model or World coordinate system defines the x, y, and z axes which serve as a basis for the 3D space. Where is the origin? Which way is up?

The Eye, Camera, or View coordinate system defines another set of x, y, and z axes which server as a different basis for the 3D space. The camera is always positioned at the origin of this coordinate system.

The Clip coordinate system describes the bounded view of the visible by the camera in terms of both the "lens" of the camera, its "depth of focus", and the aspect ratio of the screen bounds.

The Normalized Device coordinates is the same view normalized from -1 to +1 along each axis.

The Window coordinates are these x and y coordinates positioned within the screen bounds. The z is used for depth-testing and is bound between 0 and 1.

Matrix multiplication

$$(AB)_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j}$$ for each pair i and j with 1 ≤ i ≤ m and 1 ≤ j ≤ p

For example:

$$\begin{bmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 2 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \times 3 + 0 \times 2 + 2 \times 1 & 1 \times 1 + 0 \times 1 + 2 \times 0 \\ -1 \times 3 + 3 \times 2 + 1 \times 1 & -1 \times 1 + 3 \times 1 + 1 \times 0 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 4 & 2 \end{bmatrix}$$

The multiplication of a matrix by a vector results in a vector (which can be thought of as a matrix with a single column):

[1 0 ; -1 3] [3 ; 2] = [1 * 3 + 0 * 2 ; -1 * 3 + 3 * 2] = [3 ; 3]

Chirality of coordinate systems

In general, we intuitively think of defining 3D space with a "Left-handed" coordinate system, where the x axis goes from left to right, the y axis goes from bottom to top, and the z axis goes from you into the distance.

In a "Right-handed" system, one of these axes would be reversed. For example, if we thought of the x axis as going from left to right, the y axis as going from you into the distance, and the z axis as going from bottom to top. OpenGL uses a Right-hand system.

Why is it called Right-handed? Make an L with your thumb and your index finger. The thumb is the positive x axis, the index finger is the positive y axis. Make your middle finger orthogonal to the thumb and index finger. It represents the positive z axis heading toward from you. DirectX and Processing use a Left-hand system.

If you situate your thumb and index finger in the same way with your left hand, the z axis will be heading away you. There is no way to rotate a Left-hand system into a Right-hand system. However you can move between one and the other simply by scaling one of the axes by -1.

Modelview Matrix

The modelview encodes the information about the world coordinates and the camera coordinates and orientation. It is used to transform a point from object coordinates into eye coordinates.

The modelview matrix is an "affine" transformation matrix. It is composed of a 3x3 matrix representing any linear transformation (or combination of linear transformations) along with a vector representing a translation:

$$\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \vec{b} \\ 0, \ldots, 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix}$$

Where **A** is a 3x3 Matrix representing a linear transformation and the **b** vector represents a translation. **x** is the input vector and **y** is the transformed vector. Algebraically it is the same as this:

$$\vec{y} = A\vec{x} + \vec{b}.$$

The extra stuff isn't actually needed until we multiply by the projection matrix.

Linear transformations

The upper left 3x3 matrix encodes a linear transformation (or combination of linear transformations). Basically a linear transformation is some type of operation which preserves vector addition and scalar multiplication.

$$f(x+y) = f(x) + f(y)$$

$$f(ax) = af(x)$$

(You can think of the function *f* as the process of multiplying a matrix by a vector. In the above equations "a" is a scalar, **x** and **y** are vectors)

That is, all if I take two vectors and multiply them each by the matrix, and then add those vectors together or scale them by some number, it is the same as if I had taken those vectors and added them together or scaled them first and then multiplied the result by the matrix…

Linear transformations

Simple example in 2D:

if $A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ and we have the vectors (2, 3) and (6, 5)…

Then adding the vectors together = (8, 8)
and then multiplying by **A** = (8, -8)

Which is the same as multiplying them both by **A** = (2, -3) and (6, -5)
And then adding them together = (8, -8).

In 3D graphics the most common linear operations are scaling and rotation.

Geometrically, any linear operation keeps the origin in the same place.

Linear transformations

Each row of the upper left 3x3 corresponds to an axis:

x1, x2, x3

y1, y2, y3

z1, z2, z3

The simplest one is the "standard basis", which looks like the normal Cartesian graph.

x = (1, 0, 0)

y = (0, 1, 0)

z = (0, 0, 1)

Any point in 3D can be represented as linear combination of these three axes:

v = (v1, v2, v3) = (v1 * **x**, v2 * **y**, v3 * **z**)

Multiplying any vector **v** by this matrix **A** "places" it in the standard coordinate system.

Rotation

We can rotate the entire coordinate system by multiplying it by a transformation matrix. The following matrices encode a rotation by angle theta around the specified axis.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

If we want to rotate a point *p* that is pointed to by the vector **v** around the z axis we move the **x** and **y** basis vectors in a proportional circular motion.

Now we can refer to *p* by this new rotated coordinate system simply by multiplying the vector **v** by this matrix. This returns a new vector, **u**.

u points to the same location that **v** points to, that is, *p*. Now if we use **u** in our original coordinate system (by multiplying **u** by our original matrix) we have in effect rotated the point *p*.

For example, to rotate the point *p*=(2,2,0) 45° around the z-axis:

I = (1 0 0 ; 0 1 0 ; 0 0 1) and **R** = (.707 -.707 0 ; .707 .707 0; 0 0 1)

v = (2 ; 2 ; 0)

u = (**Rv**) = (2 * .707 + 2 * -.707 + 0 ; 2 * .707 + 2 * .707 + 0 ; 2 * 0 + 2 * 0 + 0 * 1)
= (0 ; 2.828 ; 0)

And (**Iu**) = (0 ; 2.828 ; 0) which points to our new rotated location.

Affine transformation

To simultaneously rotate and translate a point in 3D we need the extra information in the 4x4 affine transformation matrix, where the 4th column represents the translation.

We also need to refer to points in homogeneous coordinates. In matrix algebra you don't ever refer to points per se, you refer to vectors, and vectors only have an orientation and a magnitude– they don't have a location. The 4th component of an homogeneous vector indicates whether or not we are thinking of it as a point or not. In general, we use the modelview to transform points in space, and so in most cases the 4th component is 1.

Homogeneous coordinates are ubiquitous in computer graphics because they solve the problem of representing a translation and projection as a matrix operation.

Homogeneous Coordinates

A 3D point in homogeneous coordinates looks like this has four elements, x, y, z, and w.

To map a point (x, y, z, w) in homogeneous coordinates back to normal Euclidian coordinates we divide each component by w:

$(x', y', z') = (x/w, y/w, z/w) = (x, y, z, w)$

The idea is that anything along a line with the gradient defined by x, y, and z will be projected onto the same point on a specified plane, w = 1.

Thus, for instance, the following points are equivalent:

(5, -2, 3, .5) and (20, -8, 12, 2), since (x/w, y/w, z/w) both equal (10, -4, 6).

Modelview

The modelview matrix is created by multiplying the model matrix (which encodes the location and orientation of the world coordinates) with the view matrix (which encodes the location and orientation of the camera) .

Example:

We use the standard basis for our world coordinates, and then translate it 10 units to the right.

We then move the camera 10 units away along the z axis and rotate it 45° to the right.

Question: what does the modelview matrix look like?

Question: where does the point (5,5,2) appear in eye coordinates?

Modelview

The standard basis **A** = (1 0 0 ; 0 1 0 ; 0 0 1)

The translation vector **b** = (0 ; 0 ; -10)

so our model matrix **M** = (1 0 0 0 ; 0 1 0 0 ; 0 0 0 1 ; 0 0 -10 1)

Then we can update the **M** by multiplying by a rotation matrix:

The 45° rotated basis on the z axis

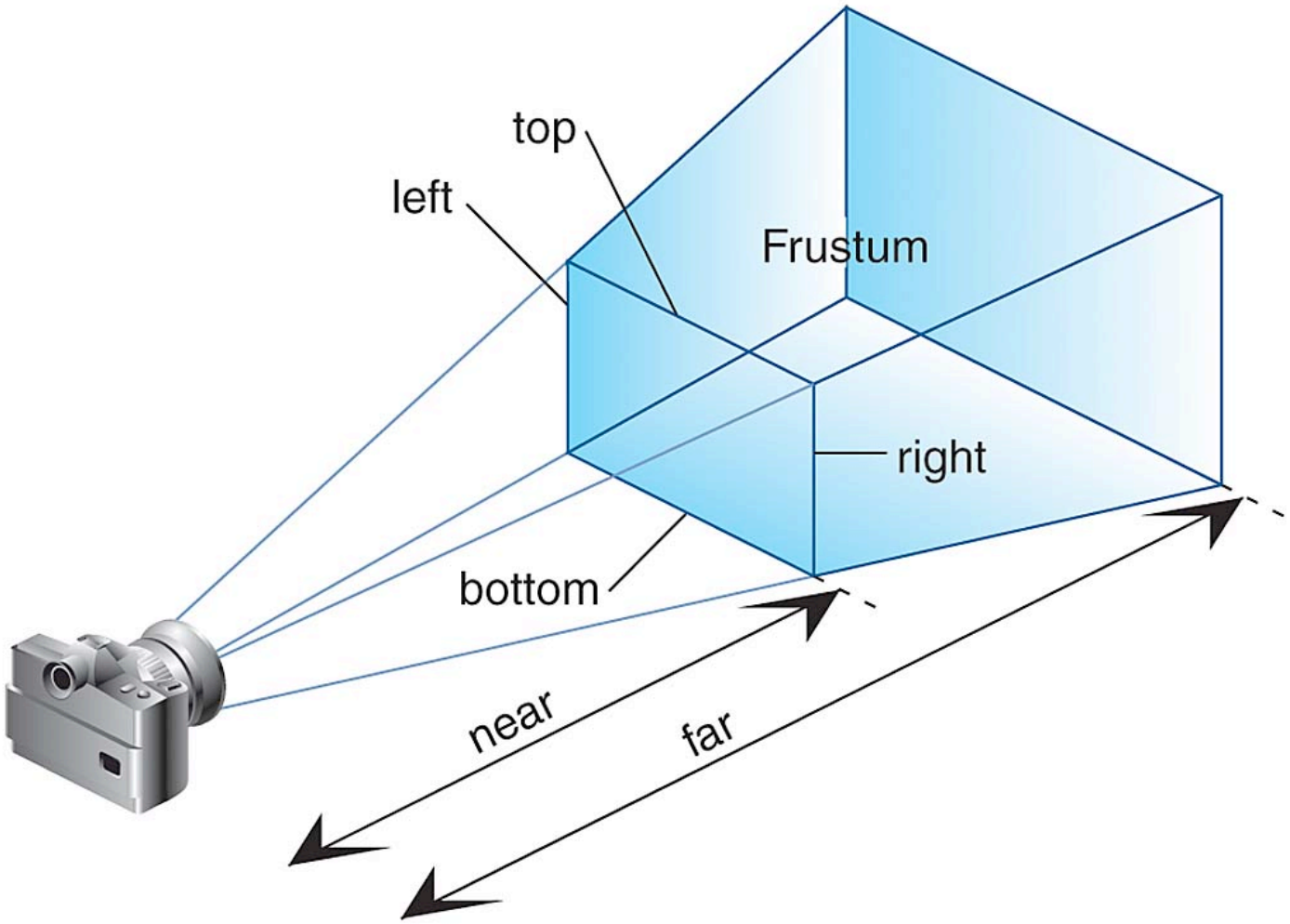
$\mathbf{R_z}$ = (.707 -.707 0 0 ; .707 .707 0 0; 0 0 1 1; 0 0 0 1)

so our modelview matrix

M = (.707 -.707 0 0 ; .707 .707 0 0 ; 0 0 1 0 ; 0 0 -10 1)

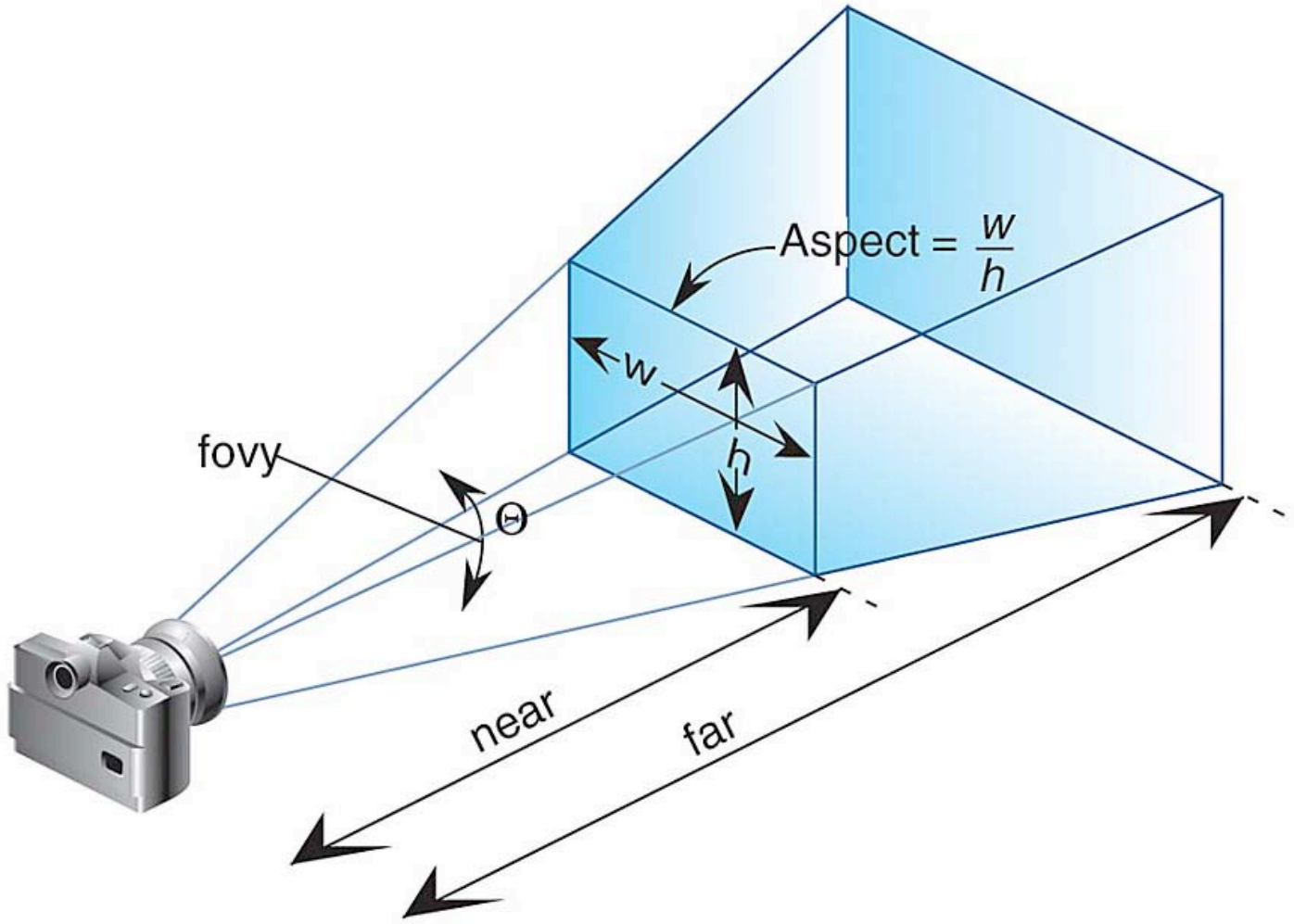
Clip Coordinates

The view frustum is defined from the point of view of the camera.



Clip Coordinates

Defining the view frustum using a perspective transformation.



Projection Matrix

The Projection Matrix defines how much of the world is seen by the camera. It encodes the following information:

The near plane and the far plane: The range of depth in the world that the camera can see.

The field of view angle that the camera sees in the y direction.

The aspect ratio of the screen which the world will be projected on.

Projection Matrix

The near plane and far plane define the distance along the z axis from the camera origin. The near plane needs to be a distance > 0 and the far plane needs to be < infinity. Common values are .1 and 100, but it depends on how you decide to position things in the world.

The field of view, or "fovy", defines the angle in the y direction

The aspect ratio (width/height) of the screen bounds thus defines the clipping in the x axis.

These values are used to define the view "frustum" in terms of 6 values, the left, right, top, bottom, near, and far bounds of the world.

The projection matrix transforms the view "frustum" into a unit cube.

Projection Matrix

The actual Projection Matrix looks likes this:

$$
\begin{pmatrix}
2n/(r-l) & 0 & -(r+l)/(r-l) & 0 \\
0 & 2n/(t-b) & (t+b)/(t-b) & 0 \\
0 & 0 & -(f+n)/(f-n) & -(2fn)/(f-n) \\
0 & 0 & 1 & 0
\end{pmatrix}
$$

Where n and f are the near and far planes,

t and b are defined by the fovy

And l and r are further defined by the aspect ratio

Example: Transforming a vertex

To transform our 3D point from object coordinates into 2D window coordinates we do the following operations:

Given a vertex $\mathbf{v}_o$ in object coordinates (x_o, y_o, z_o, w_o), where w_o is always 1.

Put the object point into eye coordinates by multiplying it by the MODELVIEW matrix **M** (which concatenates the transformation from object coordinates → world coordinates → eye coordinates)…

$\mathbf{v}_e = \mathbf{M}\mathbf{v}_o$

Put the vertex into clip coordinates by multiplying it by the PROJECTION matrix P

$\mathbf{v}_c = \mathbf{P}\mathbf{v}_e$

Put the vertex into normalized device coordinates by dividing by the w_c value of $\mathbf{v}_c$.

$\mathbf{v}_d = (x_c / w_c, y_c / w_c, z_c / w_c)$

Put the vertex into screen space by scaling x_c and y_c by the width and height of the screen.

$\mathbf{v}_p = (\text{width}/2 + (x_d * \text{width}/2), \text{height}/2 + (y_d * \text{height}/2))$