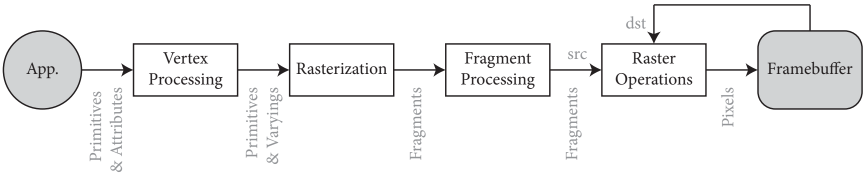
CSC 4356 / ME 4573 Interactive Computer Graphics

Transformation

The Basic 3D Graphics Pipeline



MVP

Model–View–Projection

The MVP is a *composition of transformations* that takes an object and puts it on the screen.

Model	Place an object within a scene.
View	Place a camera within a scene.
Projection	Set the camera zoom.

Let's introduce transformations…

Homogeneous Vectors

3D graphics uses a 4D vector space.

$$\boldsymbol{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}$$

Homogeneous Vectors

This is a *generalization* of the 3D vector, admitting

- Infinity
- Translation
- Perspective

$$\begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix} = \begin{bmatrix} v_x/v_w \\ v_y/v_w \\ v_z/v_w \\ 1 \end{bmatrix}$$

in the form of matrix multiplication.

Matrix notation

$$M = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

This abnormal element numbering coincides with the use of C language arrays by the OpenGL API.

Vector Transformation

$$\boldsymbol{w} = M \cdot \boldsymbol{v}$$

$$\begin{bmatrix} w_x \\ w_y \\ w_z \\ w_w \end{bmatrix} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}$$

$$\begin{aligned} w_x &= v_x \cdot m_0 + v_y \cdot m_4 + v_z \cdot m_8 + v_w \cdot m_{12} \\ w_y &= v_x \cdot m_1 + v_y \cdot m_5 + v_z \cdot m_9 + v_w \cdot m_{13} \\ w_z &= v_x \cdot m_2 + v_y \cdot m_6 + v_z \cdot m_{10} + v_w \cdot m_{14} \\ w_w &= v_x \cdot m_3 + v_y \cdot m_7 + v_z \cdot m_{11} + v_w \cdot m_{15} \end{aligned}$$

Identity

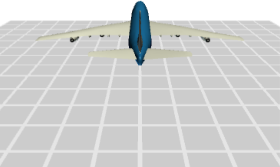
```
glLoadIdentity();
```

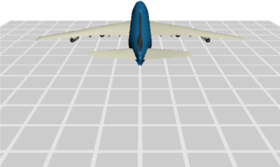
$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

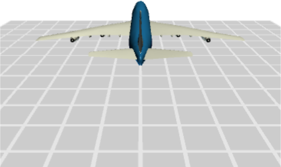
$$\begin{array}{ccccccccccc} w_x & = & v_x \cdot 1 & + & v_y \cdot 0 & + & v_z \cdot 0 & + & v_w \cdot 0 & = & v_x \\ w_y & = & v_x \cdot 0 & + & v_y \cdot 1 & + & v_z \cdot 0 & + & v_w \cdot 0 & = & v_y \\ w_z & = & v_x \cdot 0 & + & v_y \cdot 0 & + & v_z \cdot 1 & + & v_w \cdot 0 & = & v_z \\ w_w & = & v_x \cdot 0 & + & v_y \cdot 0 & + & v_z \cdot 0 & + & v_w \cdot 1 & = & v_w \end{array}$$

Scaling

```
glScalef(x, y, z);
```







x *y* *z*

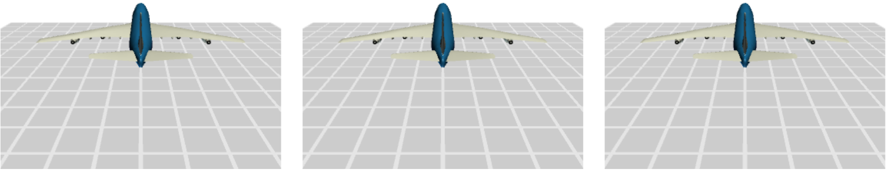
Scaling

$$S = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{array}{lllllllllll} w_x & = & v_x \cdot x & + & v_y \cdot 0 & + & v_z \cdot 0 & + & v_w \cdot 0 & = & v_x \cdot x \\ w_y & = & v_x \cdot 0 & + & v_y \cdot y & + & v_z \cdot 0 & + & v_w \cdot 0 & = & v_y \cdot y \\ w_z & = & v_x \cdot 0 & + & v_y \cdot 0 & + & v_z \cdot z & + & v_w \cdot 0 & = & v_z \cdot z \\ w_w & = & v_x \cdot 0 & + & v_y \cdot 0 & + & v_z \cdot 0 & + & v_w \cdot 1 & = & v_w \end{array}$$

Translation

```
glTranslatef(x, y, z);
```



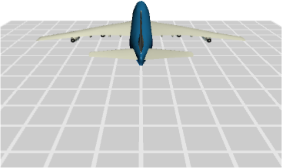
x *y* *z*

Translation

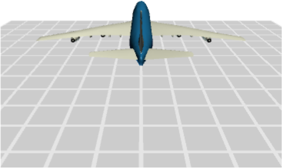
$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{array}{ccccccccccc} w_x & = & v_x \cdot 1 & + & v_y \cdot 0 & + & v_z \cdot 0 & + & v_w \cdot x & = & v_x + x \\ w_y & = & v_x \cdot 0 & + & v_y \cdot 1 & + & v_z \cdot 0 & + & v_w \cdot y & = & v_y + y \\ w_z & = & v_x \cdot 0 & + & v_y \cdot 0 & + & v_z \cdot 1 & + & v_w \cdot z & = & v_z + z \\ w_w & = & v_x \cdot 0 & + & v_y \cdot 0 & + & v_z \cdot 0 & + & v_w \cdot 1 & = & v_w \end{array}$$

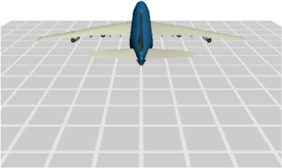
Rotation



x



y



z

The instructor sticks out his thumb.

Rotation

```
glRotatef(a, 1.0f, 0.0f, 0.0f);
glRotatef(a, 0.0f, 1.0f, 0.0f);
glRotatef(a, 0.0f, 0.0f, 1.0f);
```

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & t & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ t & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} c & t & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$c = \cos a \qquad s = \sin a \qquad t = -\sin a$$

Matrix multiplication

$$M = A \cdot B$$

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} = \begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix} \cdot \begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

$$m_0 = a_0 \cdot b_0 + a_4 \cdot b_1 + a_8 \cdot b_2 + a_{12} \cdot b_3$$

Matrix multiplication

Matrix multiplication is *not* commutative.

$$A \cdot B \neq B \cdot A$$

$$M = A \cdot B \quad \ldots \quad m_0 = a_0 \cdot b_0 + a_4 \cdot b_1 + a_8 \cdot b_2 + a_{12} \cdot b_3$$

$$M = B \cdot A \quad \ldots \quad m_0 = a_0 \cdot b_0 + a_1 \cdot b_4 + a_2 \cdot b_8 + a_3 \cdot b_{12}$$

Matrix multiplication

Matrix multiplication *is* associative.

$$(A \cdot (B \cdot C)) \quad = \quad ((A \cdot B) \cdot C)$$

The transformation "order of application" is *right-to-left*.

$$A \cdot B \cdot C \cdot \boldsymbol{v} \quad = \quad A \cdot (B \cdot (C \cdot \boldsymbol{v}))$$

$$A \cdot B \cdot C \cdot \boldsymbol{v} \quad \neq \quad C \cdot (B \cdot (A \cdot \boldsymbol{v}))$$

Transform composition

- We compose transforms by multiplying matrices.
- Because matrix multiplication is not commutative, transform composition order *matters*.
- Because the order of application is right-to-left, the *last* transform specified happens *first*.

OpenGL code seems upside-down

OpenGL transformation functions *compose* transformations, they do not *perform* transformations.

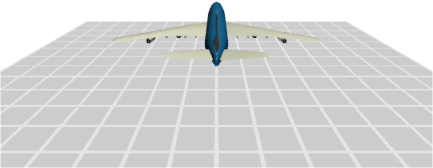
Rotate, then translate:

```
glTranslatef(...);
glRotatef(...);
glutSolidTeapot(1.0);
```

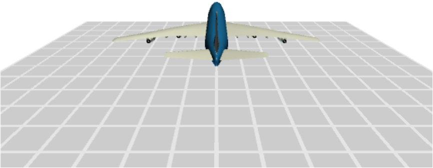
Translate, then rotate:

```
glRotatef(...);
glTranslatef(...);
glutSolidTeapot(1.0);
```

Composing Rotation and Translation

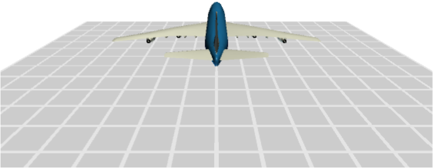


Translate–Rotate

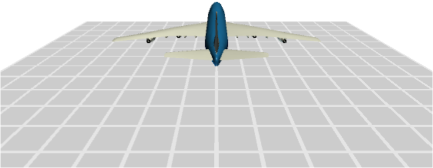


Rotate–Translate

Composing Scaling and Translation



Translate–Scale



Scale–Translate

Transform Composition

Remember that transformation occurs in the *world* coordinate system, not the object's coordinate system.

- All scaling is radially centered on the world origin.
- All rotation is about the world origin.
- All translation is along the world axes.

Matrix Inverse

The *inverse* undoes a transform.

$$M \cdot M^{-1} = I$$

Inverse matrices compose on the *left*.

$$(A \cdot B)^{-1} = B^{-1} \cdot A^{-1}$$

Matrix Transpose

The *transpose* is flipped.

$$M^T = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix} \qquad (M^T)^T = M$$

Transposes also compose on the left.

$$(A \cdot B)^T = B^T \cdot A^T$$

Normal Transformation is Different

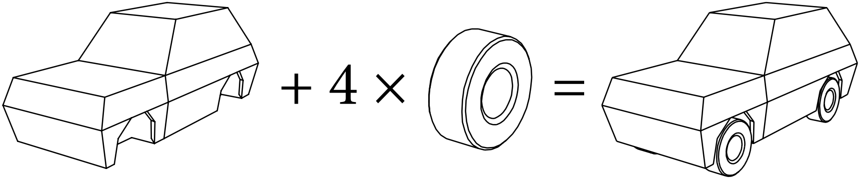
$$\boldsymbol{w} = M \cdot \boldsymbol{v}$$

$$\boldsymbol{m} = \left(M^{-1}\right)^T \cdot \boldsymbol{n}$$

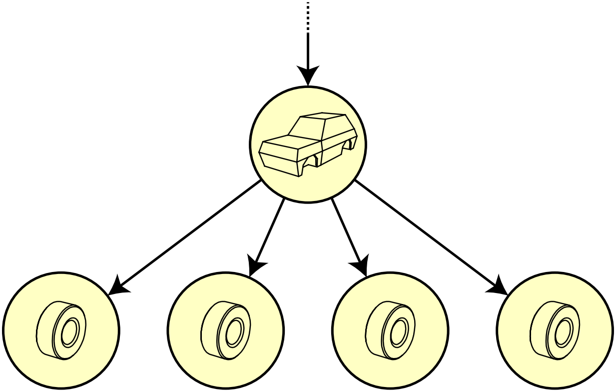
OpenGL composes the transposed inverse for you as you apply transformations.

One note: $\left(R^{-1}\right)^T = R$.

Object Instances



- A car's wheels are attached to its body.
- The wheels spin independently of the car.
- All four wheels are identical.



There is thus a *hierarchical* relationship: a tree structure.

The OpenGL Matrix Stack

The matrix stack lets you perform rendering hierarchically, getting matrix composition under control.

- `glPushMatrix();`
- `glPopMatrix();`

The OpenGL Matrix Stack

```
glTranslate(...);       // Translate relative to the world.
glRotate(...);          // Rotate about the car's axis.
draw_model(body);       // Draw the body of the car.

for (i = 0; i < 4; ++i)
{
    glPushMatrix();     // Remember the current transform.

    glTranslate(...);   // Translate relative to the car.
    glRotate(...);      // Rotate about the wheel's axis.
    draw_model(wheel);  // Draw one wheel.

    glPopMatrix();      // Recall the previous transform.
}
```

Transform Quiz

This is the default orientation of the 3D Example Cube.

Transform Quiz #1

What rotation has occurred?

Transform Quiz #1 Answer


```
glRotate(-90, 0, 1, 0)
```

Transform Quiz #2

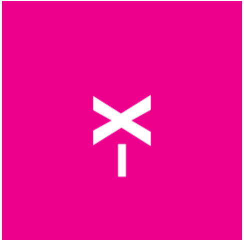
What rotation has occurred?

Transform Quiz #2 Answer


```
glRotate(+90, 1, 0, 0)
glRotate(-90, 0, 1, 0)
```

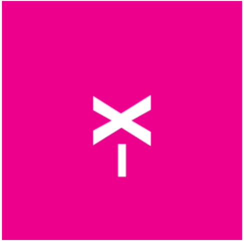
(not unique)

Transform Quiz #3



What rotation has occurred?

Transform Quiz #3 Answer



```
glRotate(-90, 1, 0, 0)
glRotate(+90, 0, 0, 1)
```

(not unique)

M is for Model

Through compositions of transformations in the form of multiplications of matrices, implemented using `glScale`, `glTranslate`, `glRotate`, controlled using the matrix stack, we construct the *Model* matrix.

This positions objects within the scene.

V is for View

Now we wish to *View* the scene, placing a “camera” within it in the same fashion as we placed objects.

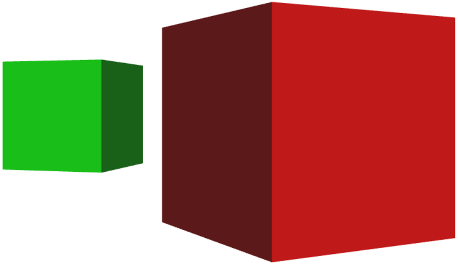
- But to move the camera to the right is to move the world to the left.
- Translate–Rotate is now Rotate–Translate.
- We must compose *inverse* transforms in *reverse*.

V is for View

It may be best for your sanity *not* to think about moving a camera around in the scene. Instead, imagine the camera is *always* at the origin, looking down $-z$, and move the scene into position to be viewed from there.

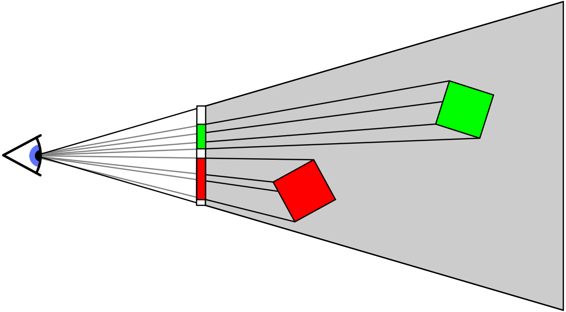
OpenGL behaves this way.

P is for Projection

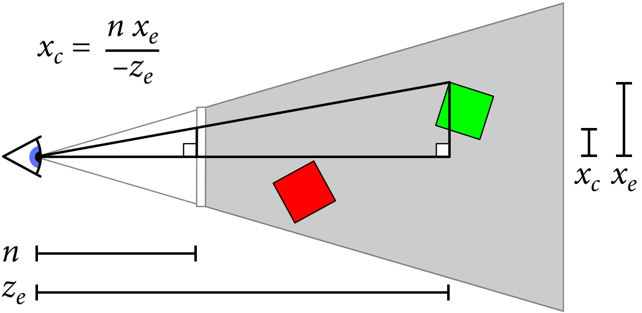


It maps 3D geometry onto the 2D plane of the screen.

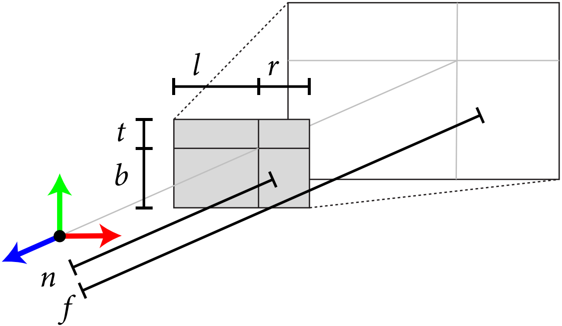
Perspective Projection from Above



Perspective Projection from Above



Perspective Projection in 3D



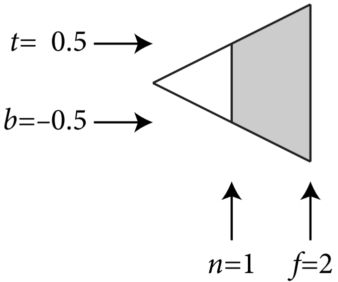

```
glFrustum(l, r, b, t, n, f);
```

General Perspective Matrix

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$l \ldots$ left
$r \ldots$ right
$t \ldots$ top
$b \ldots$ bottom
$n \ldots$ near
$f \ldots$ far

A simple example perspective

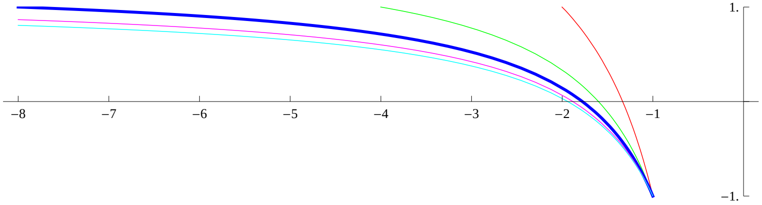


$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -3 & -4 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

$$x_c = x_e \quad w_c = -z_e$$

$$x_d = \frac{x_c}{w_c} = \frac{x_e}{-z_e} = \frac{n\, x_e}{-z_e}$$

What happened to Z?



$n = 1$

$f = 2$

$f = 4$

$f = 8$

$f = 16$

$f = 32$

$$z_d = \frac{z_e\left(f+n\right)+2fn}{z_e\left(f-n\right)}$$ is our *depth* value.

Normalized device coordinates

Normalized device coordinates are *normalized*. Really.

- The x, y, and z are all in $[-1, +1]$.
- The entire visible scene fits in the unit cube.
- This makes *clipping* easy.

Window transform

After all this, it's finally time to map vertices into the frame buffer using a simple scaling with translation.

$$W = \begin{bmatrix} w/2 & 0 & 0 & w/2 \\ 0 & h/2 & 0 & h/2 \\ 0 & 0 & d/2 & d/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For example: $w = 1024 \quad h = 768 \quad d = 65535.$

The path from model to screen

$$
\begin{array}{lcllr}
\boldsymbol{v}_o & & & & \text{Object coordinates} \\
\boldsymbol{v}_w & = & M & \cdot\, \boldsymbol{v}_o & \text{World coordinates} \\
\boldsymbol{v}_e & = & V & \cdot\, \boldsymbol{v}_w & \text{Eye coordinates} \\
\boldsymbol{v}_c & = & P & \cdot\, \boldsymbol{v}_e & \text{Clip coordinates} \\
\boldsymbol{v}_d & = & \boldsymbol{v}_c & /\, v_{cw} & \text{Normalized device coordinates} \\
\boldsymbol{v}_s & = & W & \cdot\, \boldsymbol{v}_d & \text{Window coordinates}
\end{array}
$$

Let's summarize some of the many ways you can inadvertently wind up thinking backwards:

1. OpenGL specification order versus application order.
2. Transforming normals with a vertex matrix, or vice versa.
3. View transformation versus model transformation.

Coming up…

Viewing. The many ways of computing V.