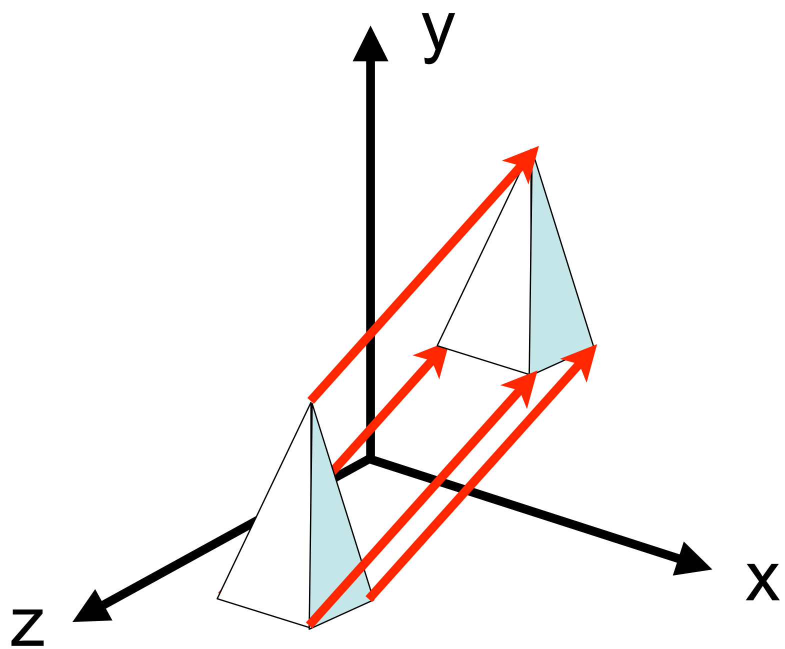
# Chapter 3 - Basic Mathematics for 3D Computer Graphics

- Three-Dimensional Geometric Transformations
- Affine Transformations and Homogeneous Coordinates
- OpenGL Matrix Logic

# Translation

- Add a vector *t*
- Geometrical meaning: Shifting
- Inverse operation?
- Neutral operation?

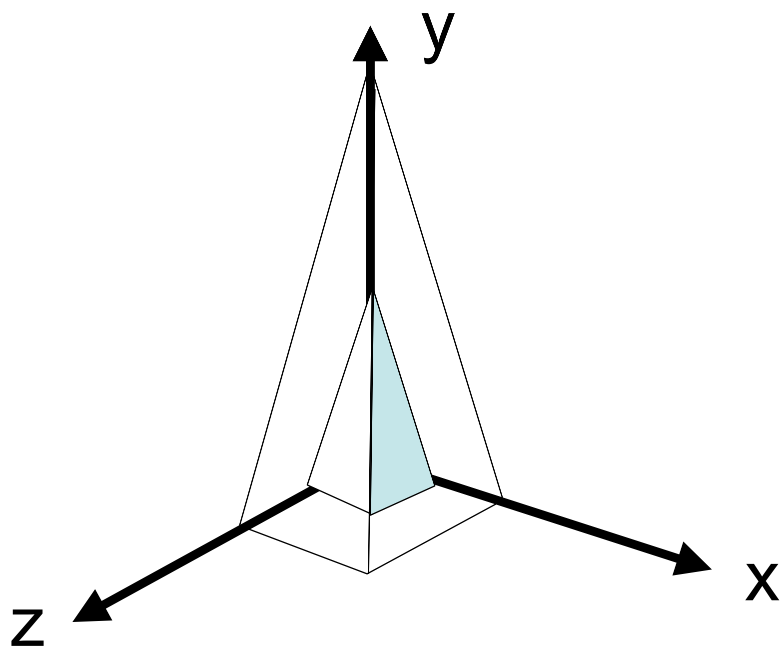
$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = \begin{pmatrix} p_1 + t_1 \\ p_2 + t_2 \\ p_3 + t_3 \end{pmatrix}$$



# Uniform Scaling

$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \cdot s = \begin{pmatrix} p_1 \cdot s \\ p_2 \cdot s \\ p_3 \cdot s \end{pmatrix}$$

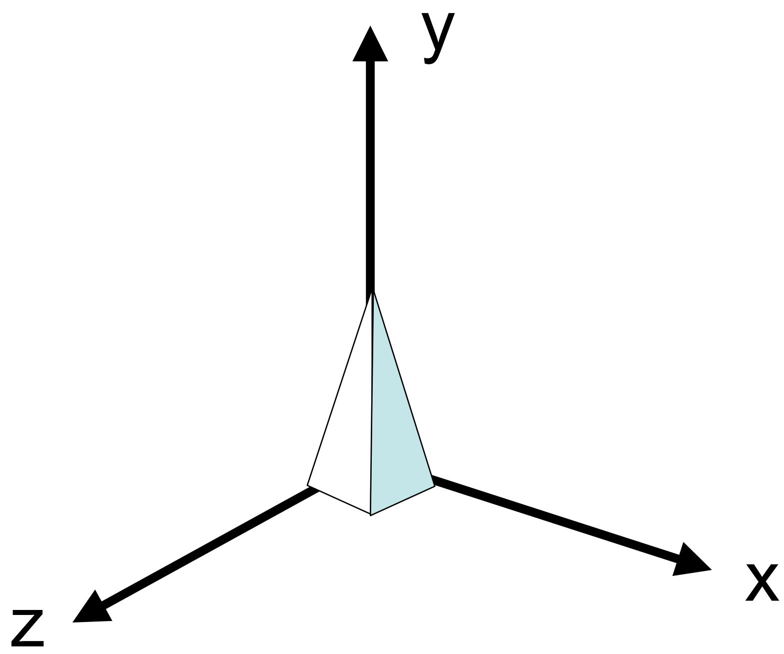
- Multiply with a scalar *s*
- Geometrical meaning: Changing size of object
- How to scale objects which are not at the origin?



# Non-Uniform Scaling

- Multiply with three scalars
- One for each dimension
- Geometrical meaning?

$$\begin{pmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} p_1 \cdot s_1 \\ p_2 \cdot s_2 \\ p_3 \cdot s_3 \end{pmatrix}$$



# Reflection (Mirroring)

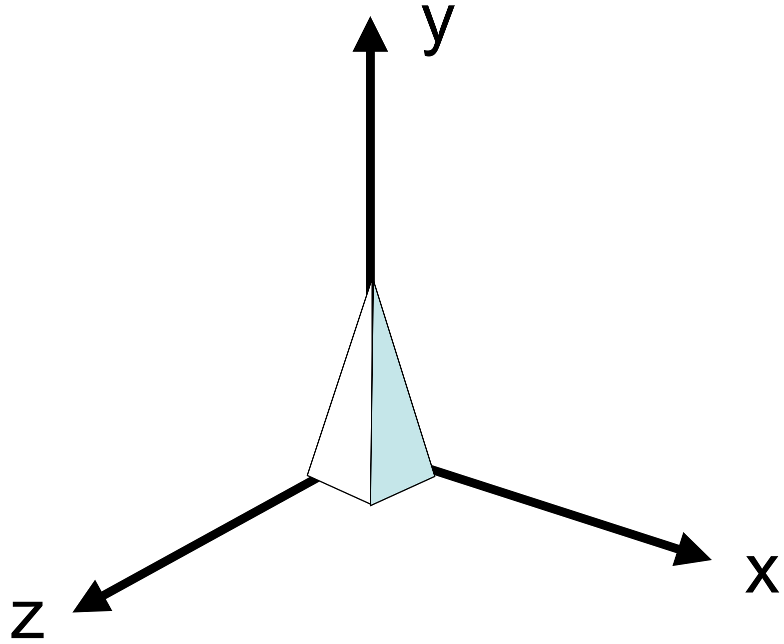
$$\begin{pmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} p_1 \cdot s_1 \\ p_2 \cdot s_2 \\ p_3 \cdot s_3 \end{pmatrix}$$

- Special case of scaling

$$s_1 \cdot s_2 \cdot s_3 < 0$$

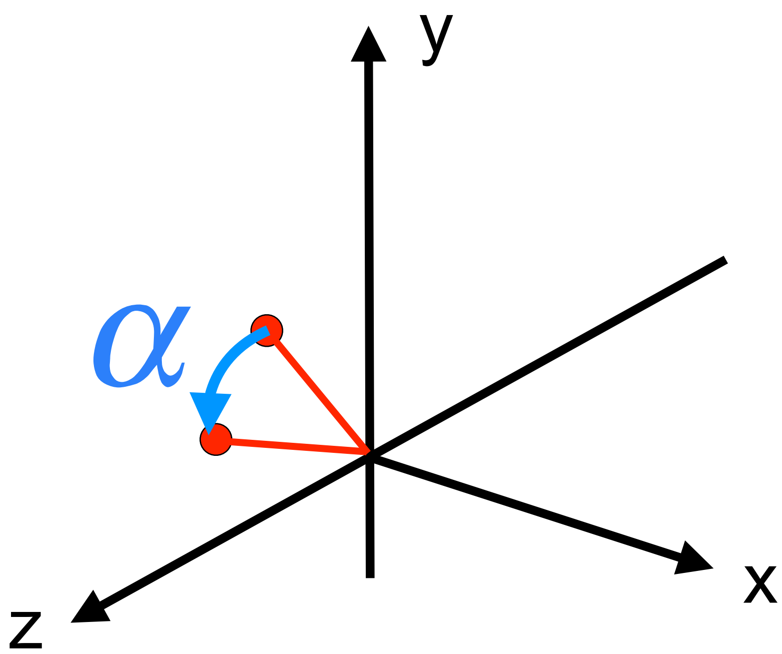
- Example:

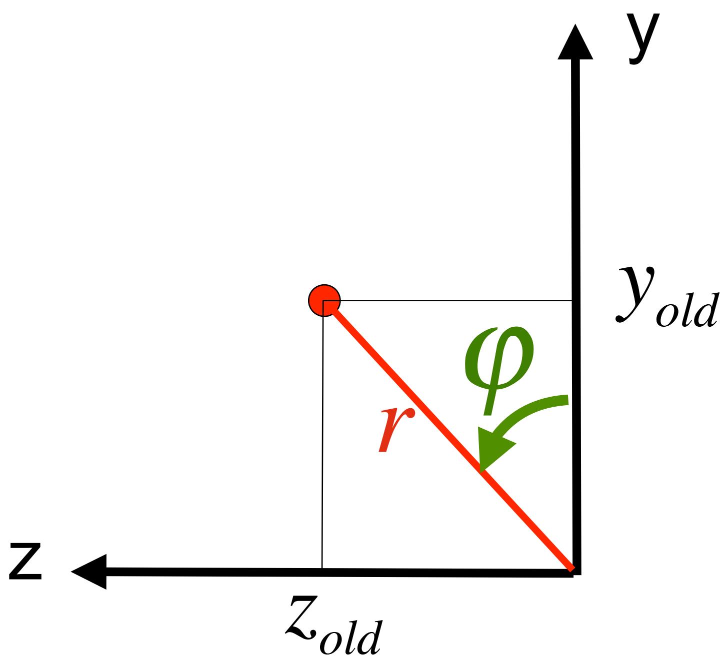
$$s_1 = 1,\ s_2 = -1,\ s_3 = 1$$



# Rotation about X Axis (1)

- x coordinate value remains constant
- Rotation takes place in y/z-plane (2D)
- How to compute new x and z coordinates from old ones?

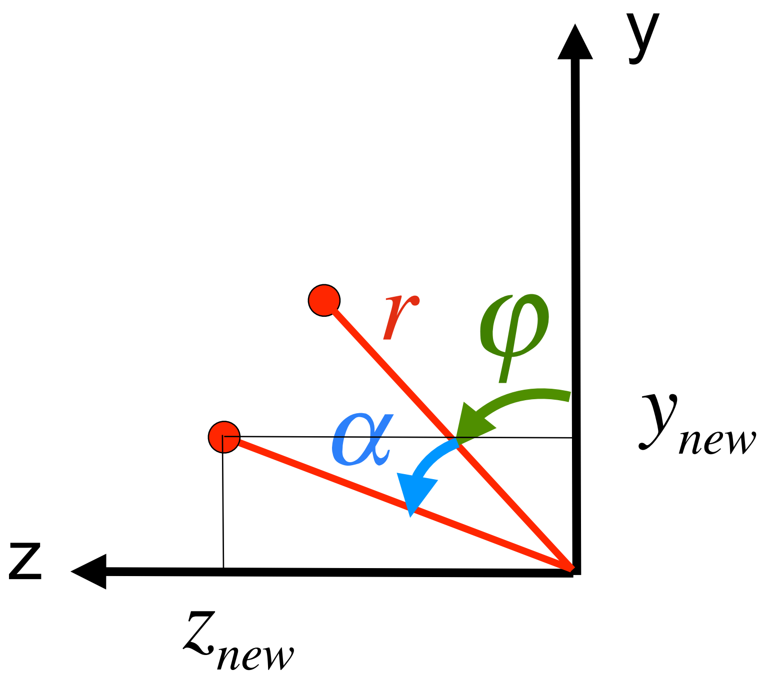



$$\sin\varphi = \frac{z_{old}}{r} \qquad z_{old} = r \cdot \sin\varphi$$

$$\cos\varphi = \frac{y_{old}}{r} \qquad y_{old} = r \cdot \cos\varphi$$

# Rotation about X Axis (2)




$$y_{old} = r \cdot \cos\varphi$$

$$z_{old} = r \cdot \sin\varphi$$

$$\cos(\alpha + \varphi) = \frac{y_{new}}{r}$$

$$y_{new} = r \cdot \cos(\alpha + \varphi)$$

$$= r \cdot \cos a \cdot \cos\varphi - r \cdot \sin\alpha \cdot \sin\varphi$$

$$= \cos\alpha \cdot y_{old} - \sin\alpha \cdot z_{old}$$

$$\sin(\alpha + \varphi) = \frac{z_{new}}{r}$$

$$z_{new} = r \cdot \sin(\alpha + \varphi)$$

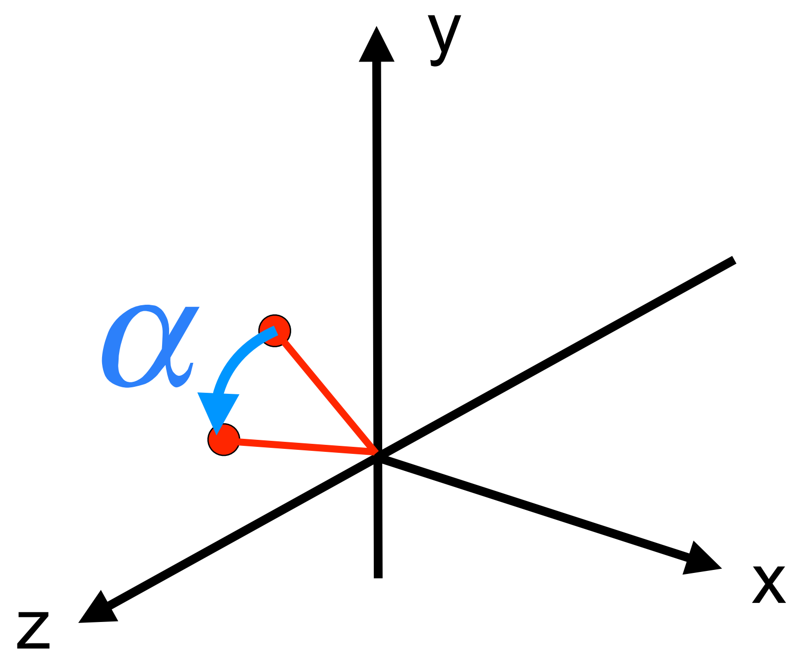
$$= r \cdot \sin a \cdot \cos\varphi + r \cdot \cos\alpha \cdot \sin\varphi$$

$$= \sin\alpha \cdot y_{old} + \cos\alpha \cdot z_{old}$$

# Rotation about X Axis (3)

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} p_1 \\ \cos\alpha \cdot p_2 - \sin\alpha \cdot p_3 \\ \sin\alpha \cdot p_2 + \cos\alpha \cdot p_3 \end{pmatrix}$$

- Special cases,
  e.g. 90 degrees, 180 degrees?
- How to rotate about other axes?



# Elementary rotations

- Combine to express arbitrary rotation
- This is not always intuitive
- Order matters (a lot!)
- Likely source of bugs!

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} p_1 \\ \cos\alpha \cdot p_2 - \sin\alpha \cdot p_3 \\ \sin\alpha \cdot p_2 + \cos\alpha \cdot p_3 \end{pmatrix}$$

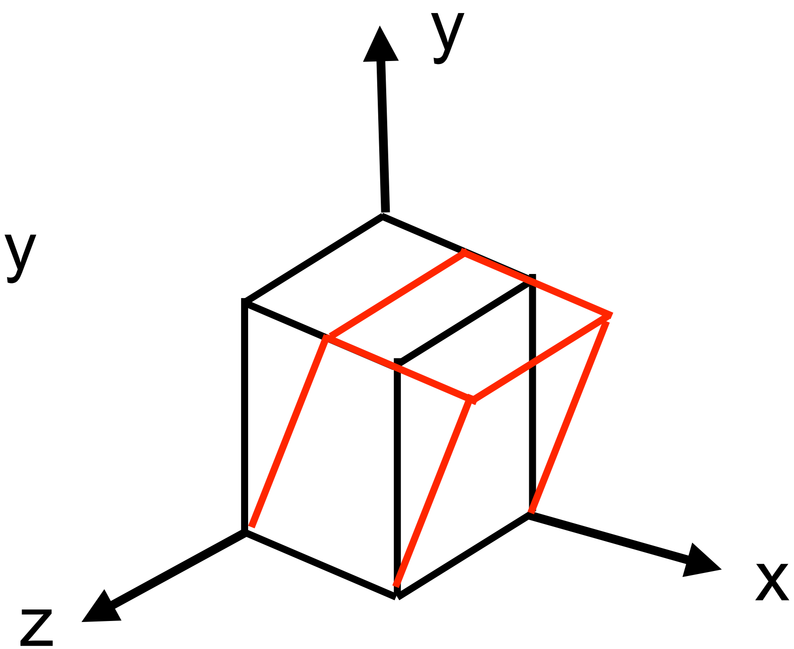
$$\begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} \cos\beta \cdot p_1 + \sin\beta \cdot p_3 \\ p_2 \\ \cos\beta \cdot p_3 - \sin\beta \cdot p_1 \end{pmatrix}$$

$$\begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} \cos\chi \cdot p_1 - \sin\gamma \cdot p_2 \\ \sin\gamma \cdot p_1 + \cos\gamma \cdot p_2 \\ p_3 \end{pmatrix}$$

# Shearing along X Axis

$$\begin{pmatrix} p_1 + m \cdot p_2 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} 1 & m & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix}$$

- Only x coordinate values are modified
- Modification depends linearly on y coordinate value
- Areas and volume remain the same
- Generalization to other axes and arbitrary axis?



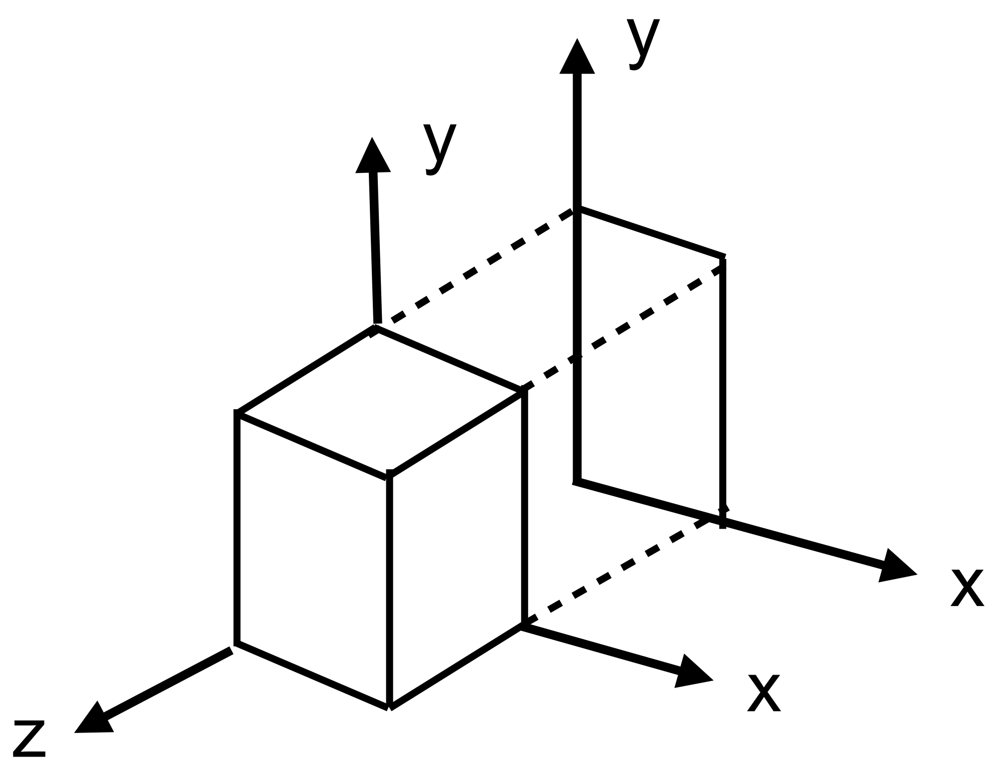
# Transformation of Coordinate Systems

- Applying a geometric transformation...
  - ...to all points of a single object: Transforming the object within its “world”
  - ...to all points of all objects of the “world”: Transforming the reference coordinates
- Geometric transformations can be used to…
  - …modify an object
  - ...place an object within a reference coordinate system
  - ...switch to different reference coordinates

# Transformation from 3D to 2D: Projection

- Many different projections exist (see later)
- Projection onto x/y plane:
  - "Forget" the z coordinate value
- Other projections?
- Other viewpoints?

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix}$$



# Chapter 3 - Basic Mathematics for 3D Computer Graphics

- Three-Dimensional Geometric Transformations
- Affine Transformations and Homogeneous Coordinates
- OpenGL Matrix Logic

# Affine Transformation

- Mathematically: A transformation preserving collinearity
  - Points lying on a line before are on a line after transformation
  - Ratios of distances are preserved (e.g. midpoint of a line segment)
  - Parallel lines remain parallel
  - Angles and lengths are *not* preserved!
- Basic transformations: translation, rotation, scaling and shearing
  - All combinations of these are affine transformations again
  - Combination is associative, but not commutative
- General form of computation:
  - New coordinate values are defined by linear function of the old values

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = Ap + t$$

# Combining Multiple Transformations

- Rotation, scaling and shearing are expressed as matrices
  - Associative, hence can all be combined into one matrix
  - Many of these operations can also be combined into one matrix

- Translation is expressed by adding a vector
  - Adding vectors is also associative
  - Many translations can be combined into a single vector

- Combination of Translation with other operations?
  - Series of matrix multiplications and vector additions, difficult to combine
  - How about using a matrix multiplication to express translation ?!?
  -
  -
  -

# Homogeneous Coordinates

- Usage of a representation of coordinate-positions with an extra dimension
  - Extra value is a *scaling factor*
- 3D position ($x$, $y$, $z$) is represented by ($x_h$, $y_h$, $z_h$, $h$) such that

$$x = \frac{x_h}{h}, \quad y = \frac{y_h}{h}, \quad z = \frac{z_h}{h}$$

- Simple choice for scaling factor $h$ is the value 1
  - In special cases other values can be used
- 3D position ($x$, $y$, $z$) is represented by ($x$, $y$, $z$, 1)

# Translation Expressed in Homogeneous Coordinates

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = \begin{pmatrix} p_1 + t_1 \\ p_2 + t_2 \\ p_3 + t_3 \end{pmatrix}$$

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 + t_1 \\ p_2 + t_2 \\ p_3 + t_3 \\ 1 \end{pmatrix}$$

# Scaling Expressed in Homogeneous Coordinates

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = \begin{pmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} s_1 p_1 \\ s_2 p_2 \\ s_3 p_3 \end{pmatrix}$$

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{pmatrix} = \begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = \begin{pmatrix} s_1 p_1 \\ s_2 p_2 \\ s_3 p_3 \\ 1 \end{pmatrix}$$

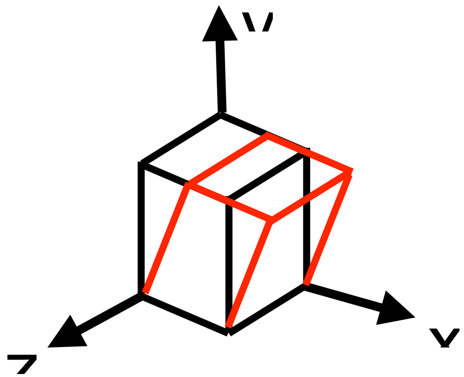
# Rotation Expressed in Homogeneous Coordinates

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} p_1 \\ \cos\alpha \cdot p_2 - \sin\alpha \cdot p_3 \\ \sin\alpha \cdot p_2 + \cos\alpha \cdot p_3 \end{pmatrix}$$

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 \\ \cos\alpha \cdot p_2 - \sin\alpha \cdot p_3 \\ \sin\alpha \cdot p_2 + \cos\alpha \cdot p_3 \\ 1 \end{pmatrix}$$

# Shearing Expressed in Homogeneous Coordinates

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = \begin{pmatrix} 1 & m & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} p_1 + m \cdot p_2 \\ p_2 \\ p_3 \end{pmatrix}$$



$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & m & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} = \begin{pmatrix} p_1 + m \cdot p_2 \\ p_2 \\ p_3 \\ 1 \end{pmatrix}$$

# Shearing: General Case

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & m_{12} & m_{13} & 0 \\ m_{21} & 1 & m_{23} & 0 \\ m_{31} & m_{32} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} p_1 + m_{12} \cdot p_2 + m_{13} \cdot p_2 \\ p_2 + m_{21} \cdot p_1 + m_{23} \cdot p_3 \\ p_3 + m_{31} \cdot p_1 + m_{32} \cdot p_2 \\ 1 \end{pmatrix}$$

# Computational Complexity for 3D Transformations

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & t_1 \\ a_{21} & a_{22} & a_{23} & t_2 \\ a_{31} & a_{32} & a_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} \cdot p_1 + a_{12} \cdot p_2 + a_{13} \cdot p_3 + t_1 \\ a_{21} \cdot p_1 + a_{22} \cdot p_2 + a_{23} \cdot p_3 + t_2 \\ a_{31} \cdot p_1 + a_{32} \cdot p_2 + a_{33} \cdot p_3 + t_3 \\ 1 \end{pmatrix}$$

- Operations needed:
  - 9 multiplications
  - 9 additions
- … for an arbitrarily complex affine 3D transformation
- Runtime complexity improved by pre-calculation of composed transformation matrices
  - Hardware implementations in graphics processors

# Chapter 3 - Basic Mathematics for 3D Computer Graphics

- Three-Dimensional Geometric Transformations
- Affine Transformations and Homogeneous Coordinates
- OpenGL Matrix Logic

# OpenGL Matrix Modes

- OpenGL maintains a storage of matrices for runtime computation of object properties
  - Identical operations used for all matrices
  - Selection of current "matrix mode" by
    `void GL2.glMatrixMode (int mode);`
- Four matrix modes are supported:
  - Modelview (`GL2.GL_MODELVIEW`)
  - Projection (`GL2.GL_PROJECTION`)
  - Texture
  - Color
- Always switch to the right mode before applying a matrix operation!

# OpenGL Matrix Stacks

- Matrices are frequently re-used in graphics programs
- Copying a whole matrix is often supported by hardware
- OpenGL supports a *stack* of matrices, one per matrix mode
  - Stack depth at least 2 in all implementations
  - Stack depth for modelview al least 32 in all implementations
- "Current matrix" is top of the stack
- Initializing a matrix with identity matrix:
  - `glLoadIdentity()`
- Stack operations:
  - `glPushMatrix()`
  - `glPopMatrix()`

# OpenGL Affine Transformations

- Basic affine transformations are available as methods
  - `glTranslate*(x, y, z)`
  - `glRotate*(alpha, x, y, z)`
  - `glScale*(sx, sy, sz)`
- Other transformations can be created using general matrix operations
  - `glLoadMatrix*(…)`
  - `glMultMatrix*(…)`
  - Parameter is matrix, represented as 16-value array (matrix as vector)
- All transformation matrices are right-multiplied onto current matrix

# OpenGL Camera Viewpoint as Transformation

- "Modelview" matrix is a hybrid between
  - "model transformations", e.g. for transforming objects
  - "view transformation": The transformation resulting from the viewpoint of the camera
- Camera viewpoint specification in JOGL
  - GLU utility function
    `lookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ)`
  - Results in an affine transformation (translation/rotation)
  - Effectively multiplies a matrix onto current modelview matrix

# Investigating JOGL's Internal Modelview Matrix

## JOGL code to print out the current modelview matrix

(adapted from http://www.cs.rutgers.edu/~decarlo/428/jogl.html)

```
public void printMVMatrix(GL2 gl) {

   double[] curmat = new double[16];
   // Get the current matrix on the MODELVIEW stack
   gl.glGetDoublev(GL2.GL_MODELVIEW_MATRIX, curmat, 0);
   // Print out the contents of this matrix in OpenGL format
   for (int row = 0; row < 4; row++)
     for (int col = 0; col < 4; col++)
       System.out.format("%7.3f%c", curmat[row+col*4], col==3 ? '\n':' ');
}
```

After `glLoadIdentity()`:

```
1.000   0.000   0.000   0.000
0.000   1.000   0.000   0.000
0.000   0.000   1.000   0.000
0.000   0.000   0.000   1.000
```

# View Matrix Generated by Camera Position (1)

```
gl.glLoadIdentity();
glu.gluLookAt(0, 0, 3, 0, 0, 0, 0, 1, 0);
System.out.println("View matrix (lookAt)");
printMVMatrix(gl);
```

```
View matrix (lookAt)
  1.000   0.000   0.000   0.000
  0.000   1.000   0.000   0.000
  0.000   0.000   1.000  -3.000
  0.000   0.000   0.000   1.000
```

Which viewpoint is taken here?
What is the default camera position in OpenGL?
Which kind of affine transformation dows this result to?

# View Matrix Generated by Camera Position (2)

```
gl.glLoadIdentity();
glu.gluLookAt(0, 0, -3, 0, 0, 0, 0, 1, 0);
System.out.println("View matrix (lookAt)");
printMVMatrix(gl);
```

```
View matrix (lookAt)
 -1.000   0.000   0.000   0.000
  0.000   1.000   0.000   0.000
  0.000   0.000  -1.000  -3.000
  0.000   0.000   0.000   1.000
```

Which camera viewpoint is taken here?
Which kind of affine transformation does this result to?
Where are the differences to the preceding case?

# Simple Translation in JOGL

```
gl.glLoadIdentity();
glu.gluLookAt(0, 0, 3, 0, 0, 0, 0, 1, 0);
gl.glTranslated(0.5, 0.5, -0.5);
System.out.println("After translate");
printMVMatrix(gl);
```

```
After translate
   1.000   0.000   0.000   0.500
   0.000   1.000   0.000   0.500
   0.000   0.000   1.000  -3.500
   0.000   0.000   0.000   1.000
```

What has happened here mathematically?

# Multiplication of Modelview Matrices in JOGL

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & -0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & -3.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Viewpoint matrix　　　　Translation matrix

Pre-multiplication is right-multiplication of matrices

Application order when applied to a point: $(A_1 \cdot A_2) \cdot p = A_1 \cdot (A_2 \cdot p)$

Transformations are applied for rendering in ***inverse order*** as computed in OpenGL program code!

Example: Translation is applied first, then viewpoint transformation

# JOGL Example for Modelview Matrix Stack (1)

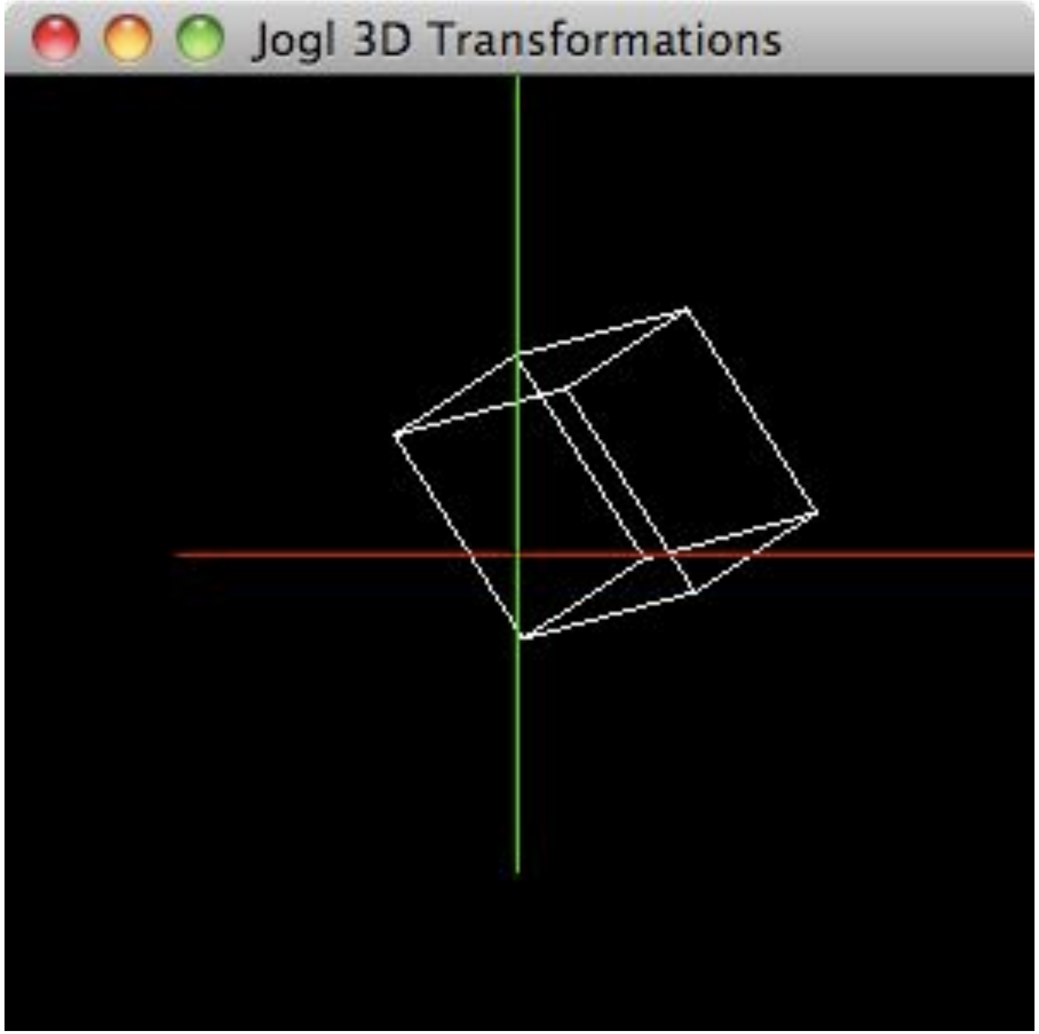
- Assume:
  - We want to display an object which was transformed by affine transformations
  - We also want to display the coordinate axes
    - of course in “original” (non-transformed) view

```
public void display(GLAutoDrawable drawable) {
  GL2 gl = drawable.getGL().getGL2(); ...
  GLU glu = new GLU(); // utility library object
  gl.glMatrixMode(GL2.GL_MODELVIEW);
  gl.glLoadIdentity();
  glu.gluLookAt(0, 0, 3, 0, 0, 0, 0, 1, 0);
  gl.glPushMatrix();
  gl.glTranslated(0.5, 0.5, -0.5);
  gl.glRotated(45, 0, 1, 1);
  gl.glScaled(0.5, 0.75, 1.0);

  gl.glBegin(GL2.GL_LINE_LOOP); // draw front side
    gl.glVertex3d(-1, -1, 1); ...
```

# JOGL Example for Modelview Matrix Stack (2)

```
        gl.glBegin(GL2.GL_LINE_LOOP);
          // draw ...
        gl.glEnd();

        gl.glPopMatrix();

        gl.glColor3d(1, 0, 0); //draw in red
        gl.glBegin(GL2.GL_LINES); // show x axis in red
          gl.glVertex3d(3, 0, 0); gl.glVertex3d(-2, 0, 0);
        gl.glEnd();

        ...

    }
```

# JOGL Example for Modelview Matrix Stack (3)




We are drawing also the z axis:

```
gl.glColor3d(0, 0, 1); //draw in blue
gl.glBegin(GL2.GL_LINES);
   // show z axis in blue
  gl.glVertex3d(0, 0, 3);
  gl.glVertex3d(0, 0, -2);
gl.glEnd();
```

Why isn't there a blue line?
How can we make it visible?