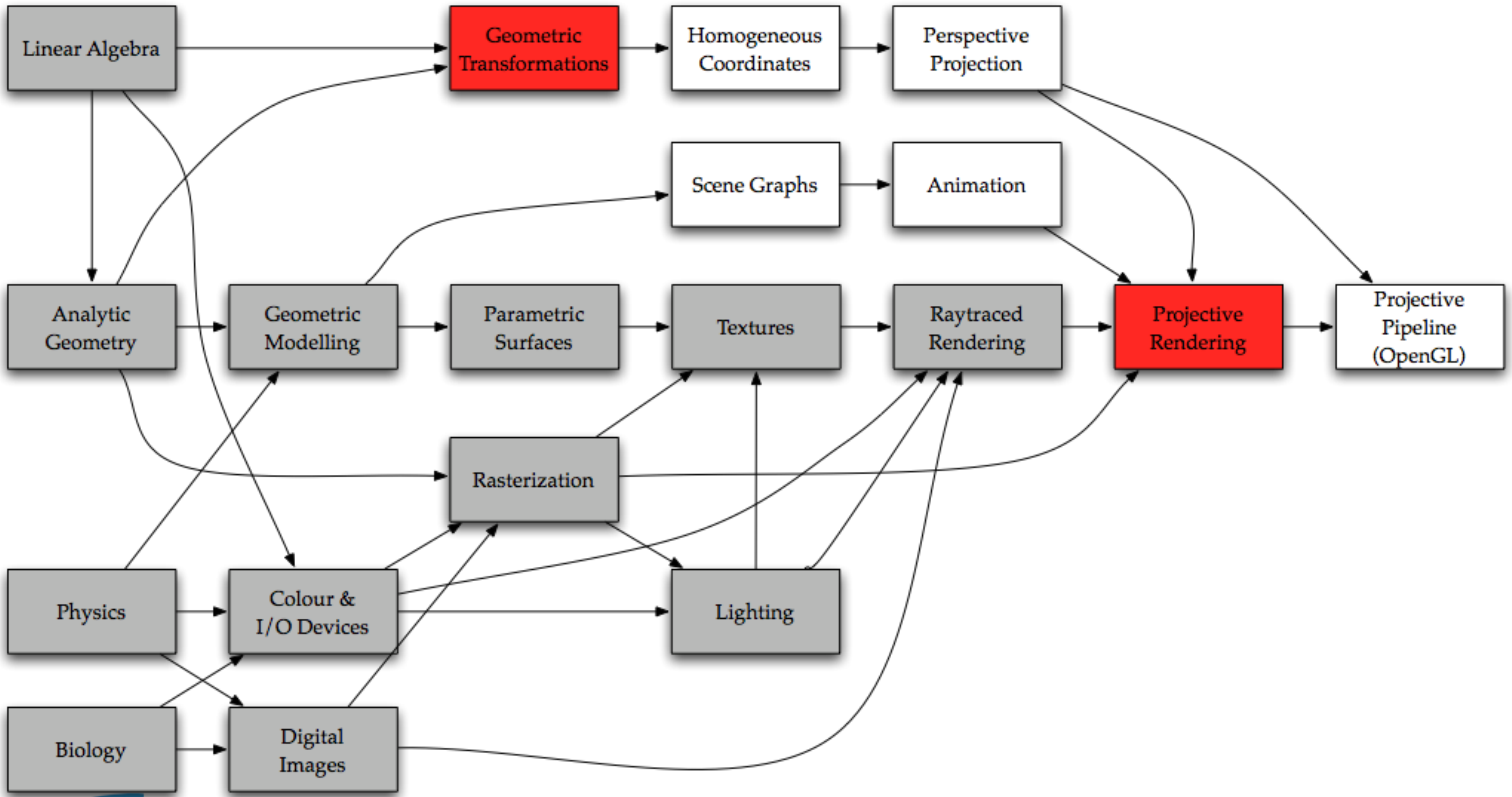
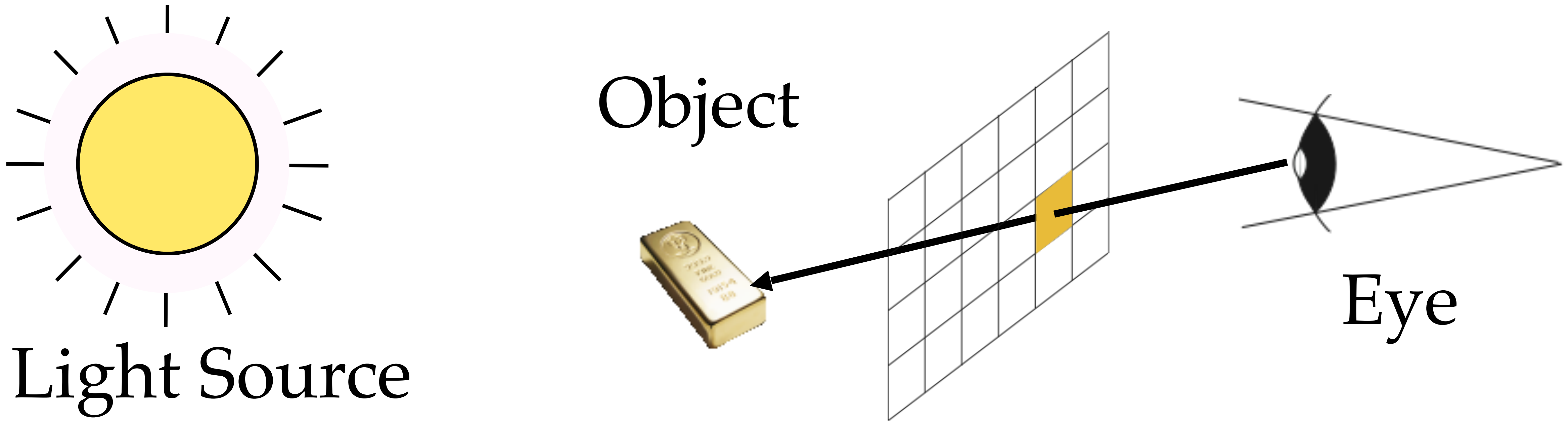
Projective Rendering & Transformations

Where we Are


Linear Algebra
Geometric Transformations
Homogeneous Coordinates
Perspective Projection
Scene Graphs
Animation
Analytic Geometry
Geometric Modelling
Parametric Surfaces
Textures
Raytraced Rendering
Projective Rendering
Projective Pipeline (OpenGL)
Rasterization
Physics
Colour & I/O Devices
Lighting
Biology
Digital Images

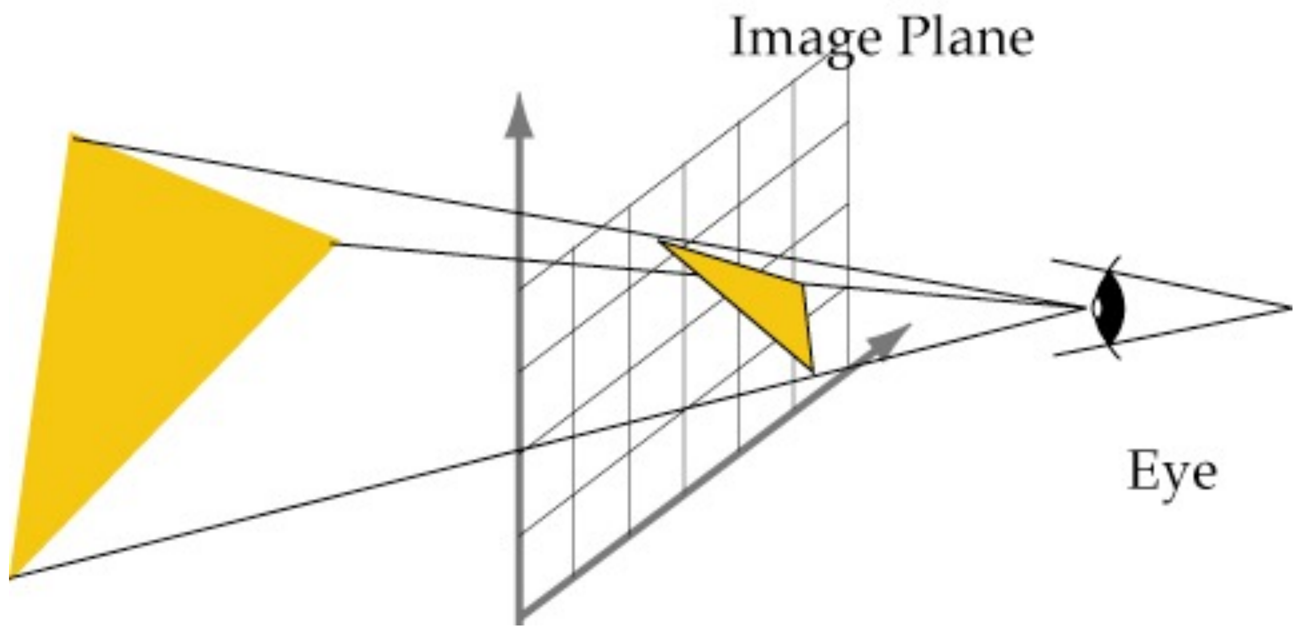
Raytracing

- For each pixel
    - Start at eye
    - Trace a ray through image plane
    - Compute colour of object it hits



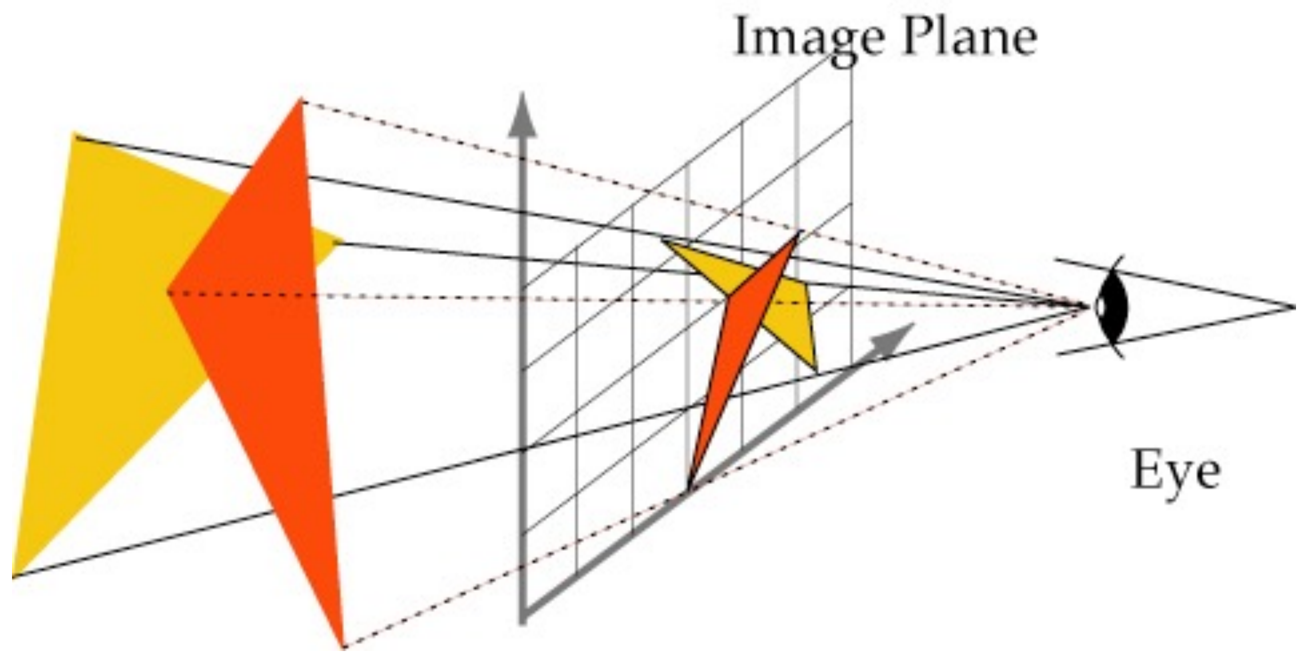
Projective Rendering

- Ray-tracing computes one pixel at a time
- Instead, we compute multiple pixels
    - For each triangle, compute it’s image
    - i.e. project it to the image plane



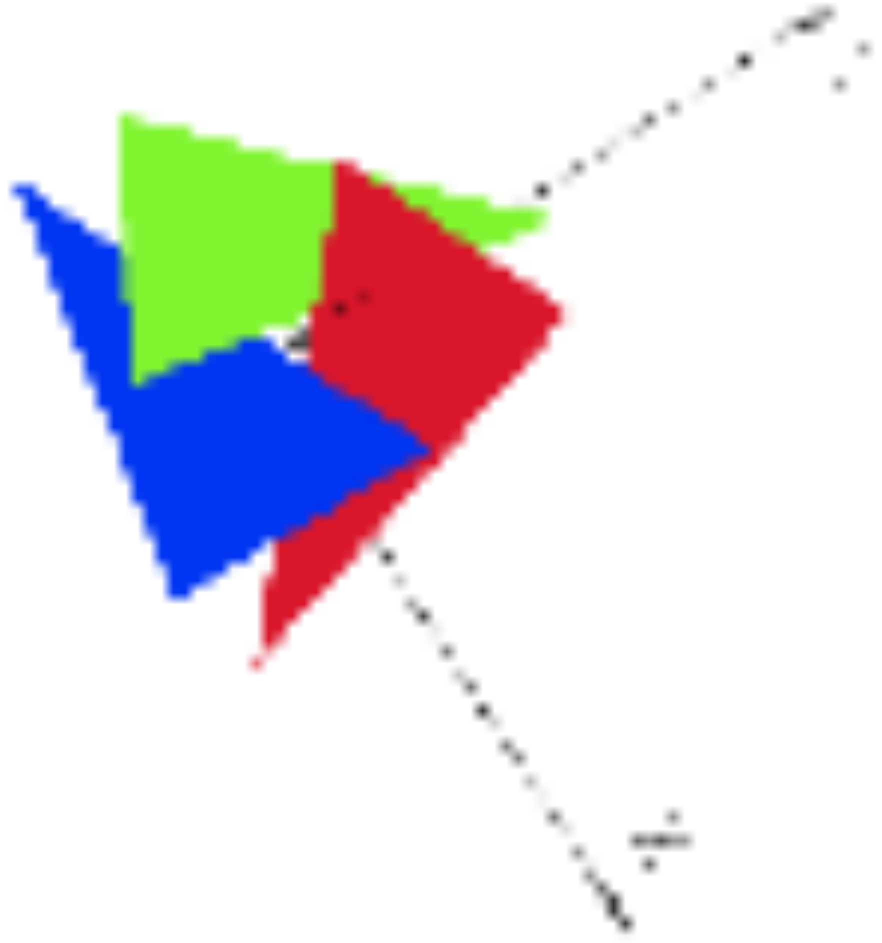
Painter’s Algorithm

- If we draw objects from back to front
  - the back objects will be *occluded*
    - i.e. we will see only the front object
- We have to *sort* all objects for each image



Worst Case

- Three triangles
  - Red overlaps Green
  - Green overlaps Blue
  - Blue overlaps Red
- Painter's Algorithm fails!

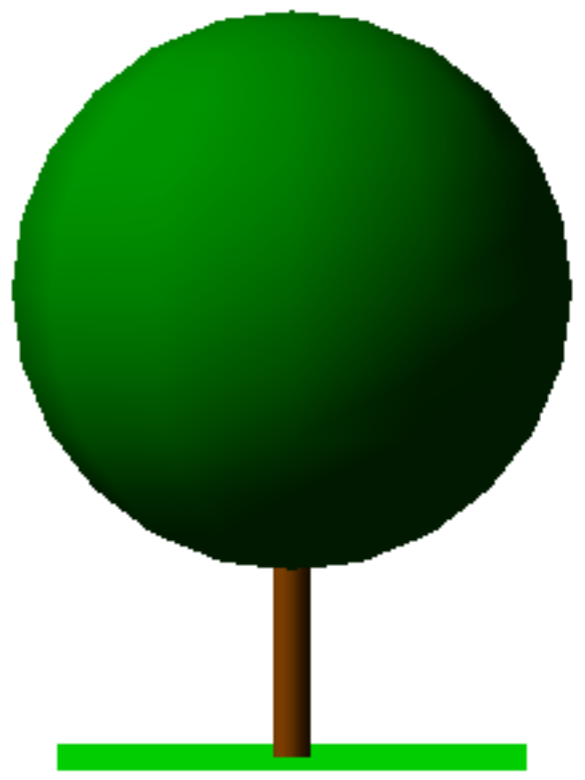


Z- (Depth) Buffering

- Store *z* coordinate along with RGB
- When drawing a pixel, compute *z* value
- Check previous *z* value first
  - only draw pixel if new *z* is larger
  - discard pixel if new z is smaller

Description

- Draw a sphere
  - radius 10 m
  - centred 7 m above ground
  - colour light green
- Draw a cylinder
  - radius 2 m, height 15 m
  - bottom face on ground
  - colour brown



Composite Modelling

- We build objects from *primitives*
  - e.g. points, lines, triangles
- Can also use *bigger* primitives
  - e.g. spheres, cones, cylinders
- Specify location, &c. with *transformations*

Object Description

- Objects have:
  - shape (what type of primitive)
  - size
  - location

Spheres

- Spheres are easy to move / resize:

A sphere of radius 1 at the origin:

$$x^2 + y^2 + z^2 = 1$$

A sphere of radius r at (x_0, y_0, z_0):

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$$

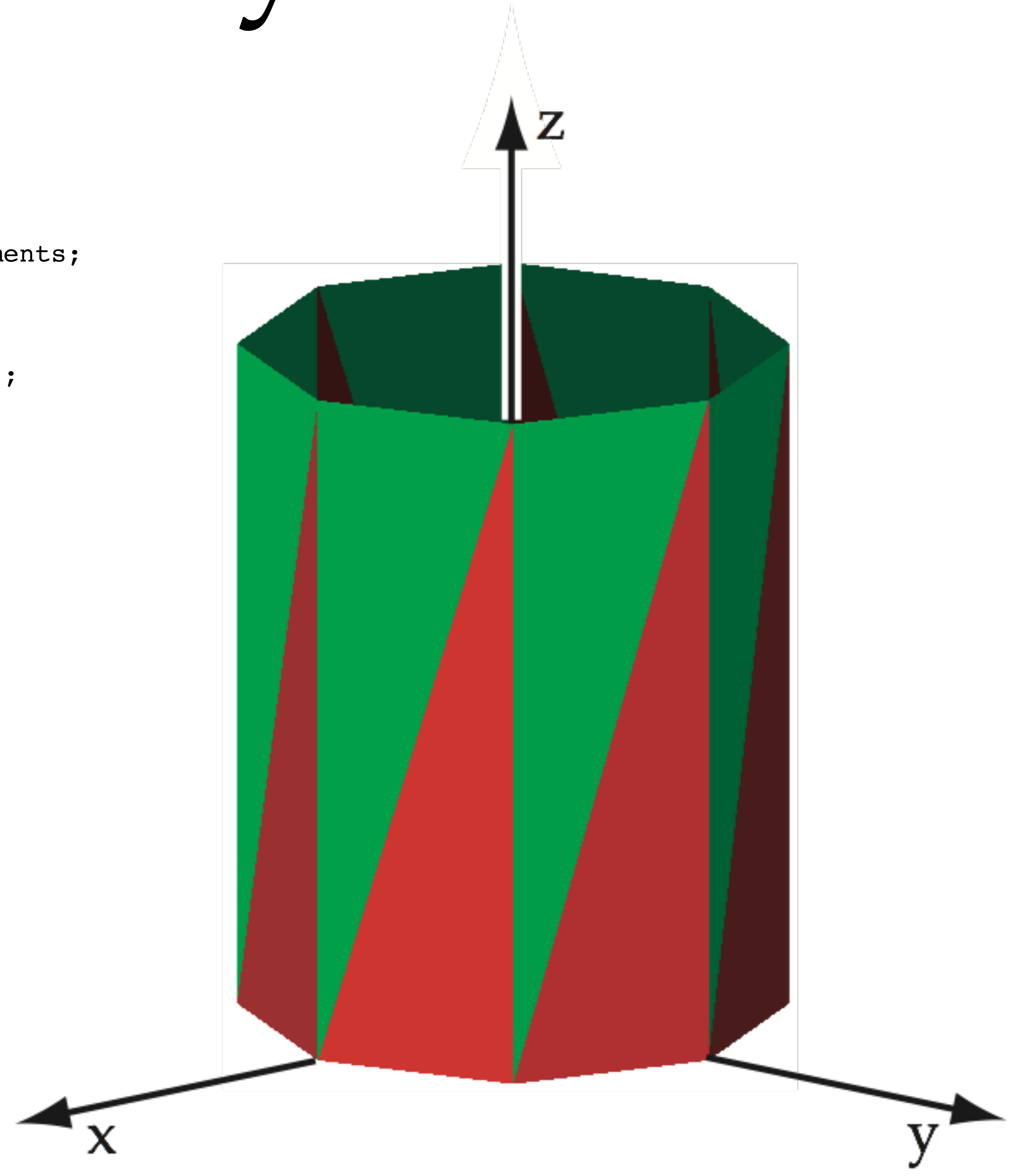
- What about orientation?

Spheres

- Spheres are *symmetric*
  - they are the same in every orientation
  - so we don't have to worry
- But what about *cylinders?*
  - Moving / scaling isn't hard
  - Orientation (rotation) is hard

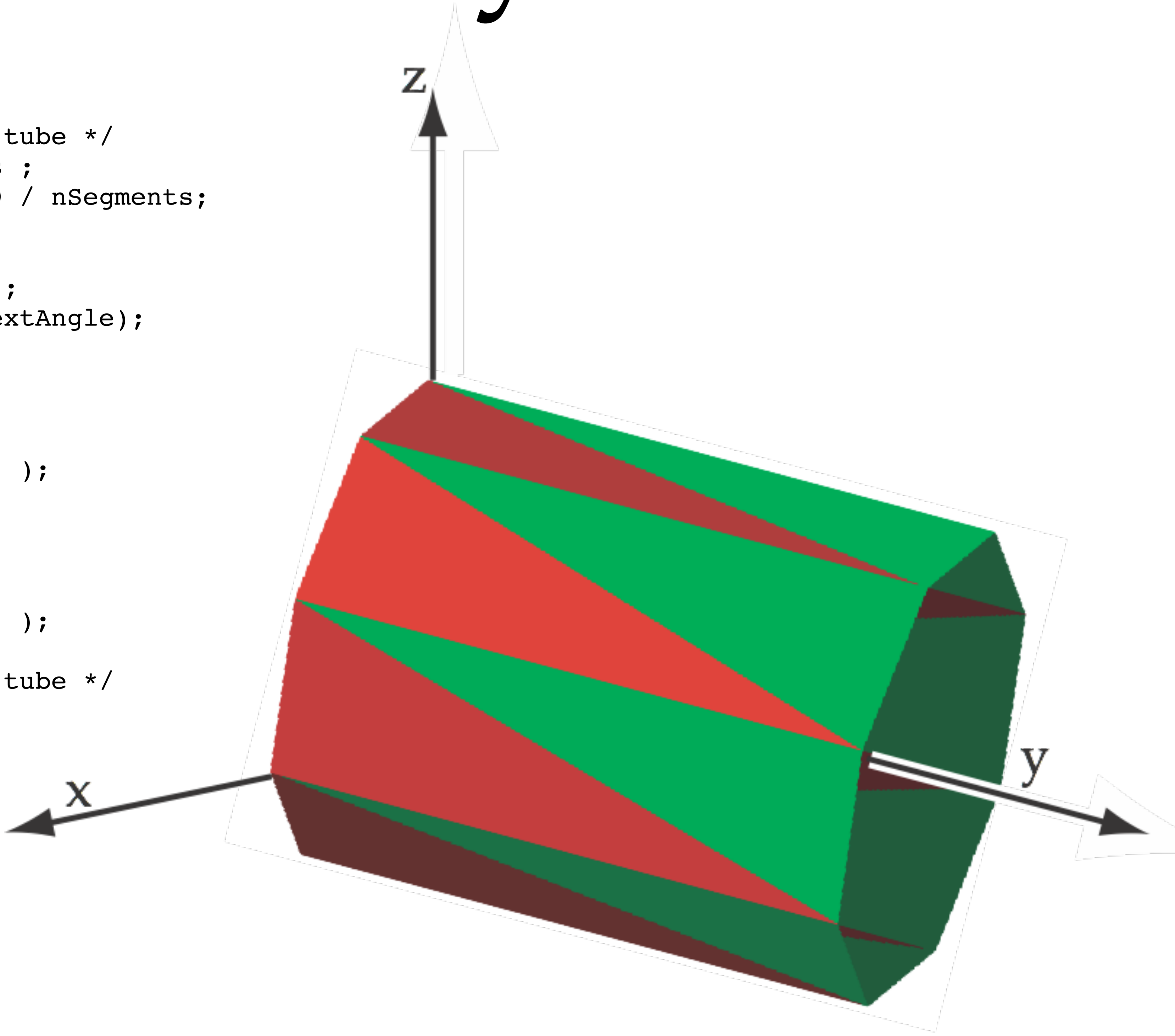
A Vertical Cylinder

```
for (float i = 0.0; i < nSegments; i += 1.0)
       { /* a loop around circumference of a tube */
       float angle = PI * i * 2.0 / nSegments ;
       float nextAngle = PI * (i + 1.0) * 2.0 / nSegments;

       /* compute sin & cosine */
       float x1 = sin(angle), y1 = cos(angle);
       float x2 = sin(nextAngle), y2 = cos(nextAngle);

       /* draw top triangle */
      drawTriangle(   x1, y1, 0.0,
                               x1, y1, 1.0,
                               x2, y2, 1.0    );

       /* draw bottom triangle */
      drawTriangle(   x1, y1, 0.0,
                               x1, y1, 1.0,
                               x2, y2, 1.0    );

       } /* a loop around circumference of a tube */
```



A Horizontal Cylinder

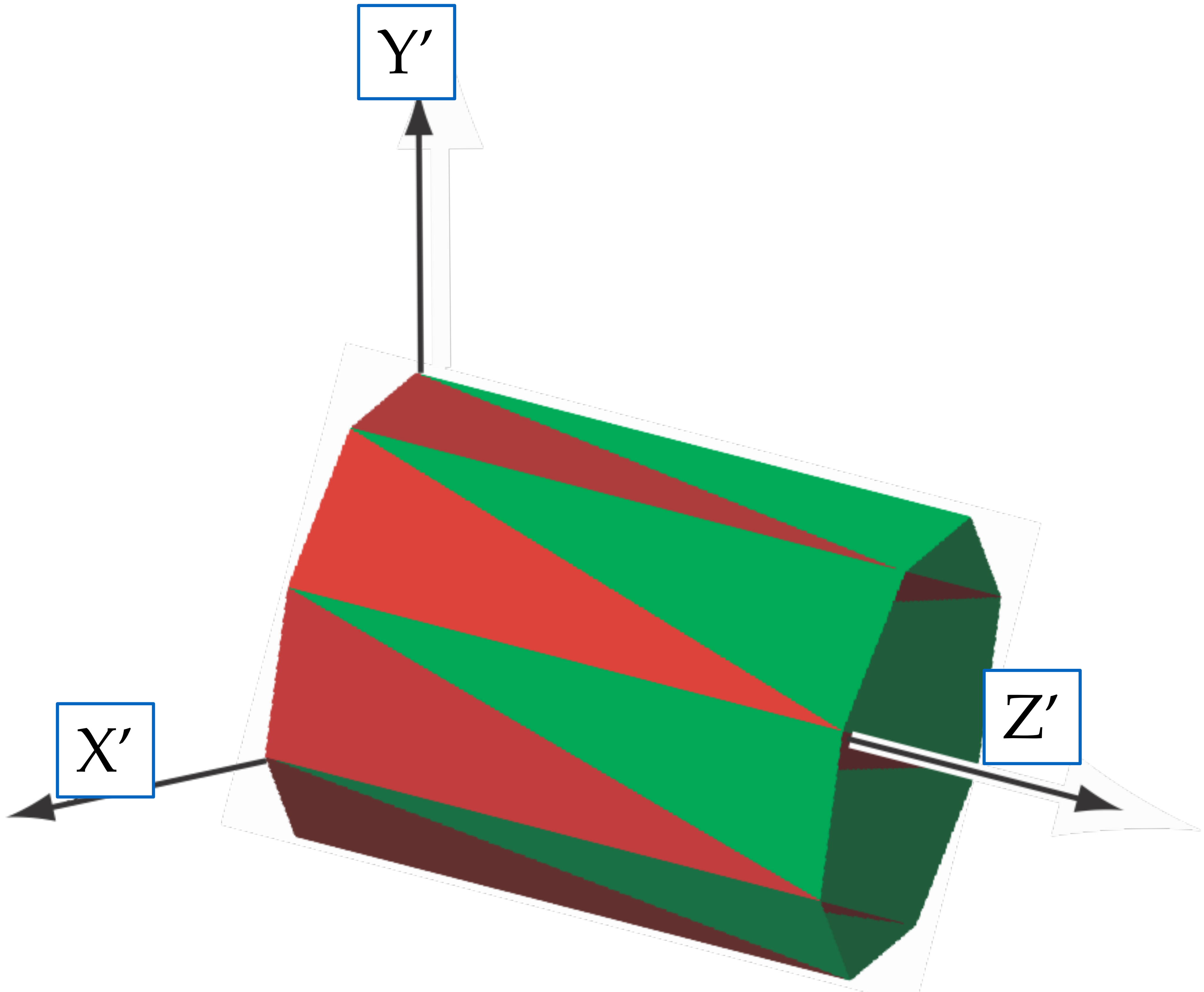
```
for (float i = 0.0; i < nSegments; i += 1.0)
       { /* a loop around circumference of a tube */
       float angle = PI * i * 2.0 / nSegments ;
       float nextAngle = PI * (i + 1.0) * 2.0 / nSegments;

       /* compute sin & cosine */
       float y1 = sin(angle), z1 = cos(angle);
       float y2 = sin(nextAngle), z2 = cos(nextAngle);

       /* draw top triangle */
      drawTriangle(   0.0, y1, z1,
                               1.0, y1, z1,
                               1.0, y2, z2    );

       /* draw bottom triangle */
      drawTriangle(   0.0, y1, z1,
                               1.0, y1, z1,
                               1.0, y2, z2    );

       } /* a loop around circumference of a tube */
```



Drawbacks

- We need code for *each* cylinder
    - although they're essentially the *same*
- So how can we *reuse* code?
    - draw a standard cylinder
    - and move it around easily

A Different Viewpoint



New Axes for Old

- Tilt your head to the right
- Now “up” in your vision is changed
- The cylinder’s coordinates haven’t
- It’s in a different *coordinate system*
- Described by a new *basis*
    - consisting of axes x′, y′, z′

Changing Systems

- Let $\vec{x}', \vec{y}', \vec{z}'$ be axes
- P is a weighted sum
- Express as a matrix

$$p' = p_x \vec{x}' + p_y \vec{y}' + p_z \vec{z}'$$

$$= p_x \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + p_y \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} + p_z \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ \underbrace{0}_{\vec{x}'} & \underbrace{-1}_{\vec{y}'} & \underbrace{0}_{\vec{z}'} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

Basis Vectors

- Axes can be any set of 3 *independent* vectors
    - Call this set a *basis*

Orthonormal Basis

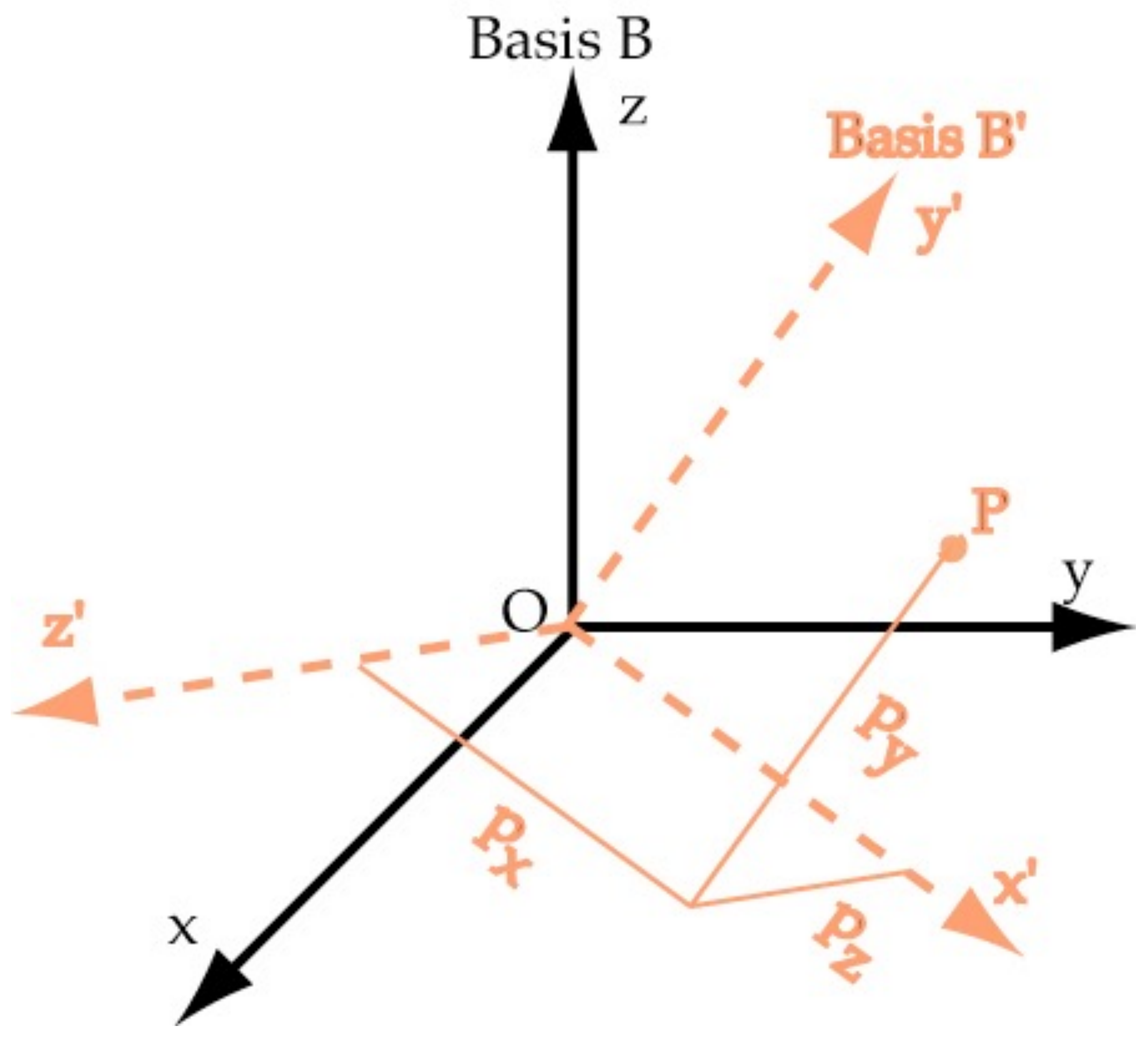
- The “best” bases are *orthonormal*
  - vectors are mutually perpendicular
  - length 1
- Cartesian coordinates are orthonormal
  - that’s why they’re so useful

Changing Bases

- Assume P is in B′
- Where is P in B?
- Find x′, y′, z′ in B
- Multiply by

$$M = \begin{bmatrix} \vec{x}' & \vec{y}' & \vec{z}' \end{bmatrix}$$

- To reverse, use M^{-1}



Transformations

- Changing basis is a *transformation*
- An operation on vectors, points, &c.
- An *affine* transformation preserves lines
  - Lines before are still lines after
  - Angles and lengths may change
- Cartesian matrices are *linear* transformations
  - and are always *affine*

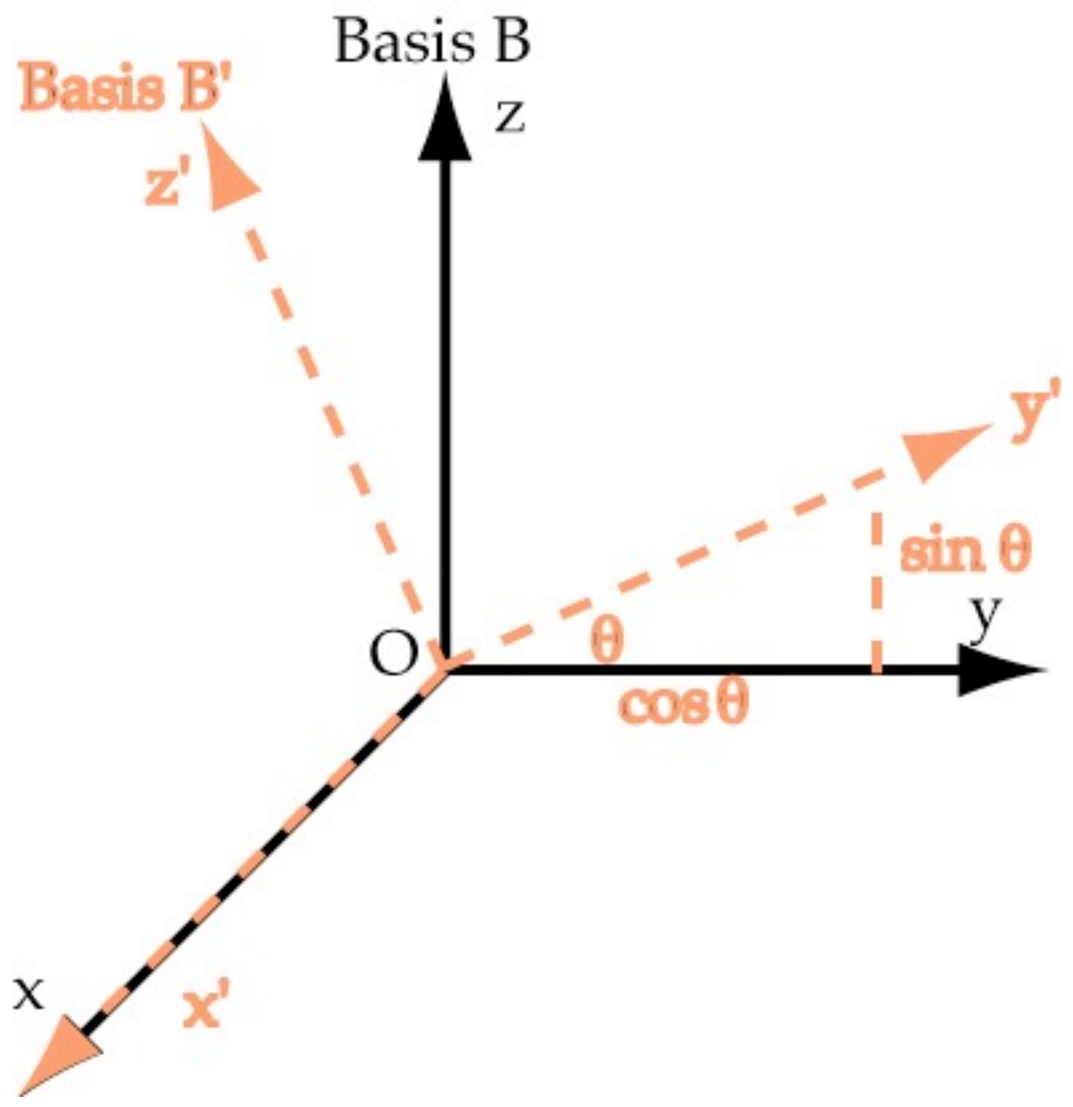
What can Bases Do?

- *Rotation* (any)
- *Scaling*, including
  - *Reflection*
  - *Perpendicular Projection*
- *Shearing*
- BUT not *Translation* or *Perspective Projection*

Rotation Matrices

- Rotate CW around x-axis by angle θ:
  - from B to B′

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix}$$



More Rotations

- Stand at the end of the axis of rotation
    - face in, and rotation is CCW
- Find the matrix for a rotation around y
- Find the matrix for a rotation around z
- Any orthonormal basis is a rotation
    - How can we find the axis?

Inverse Rotations

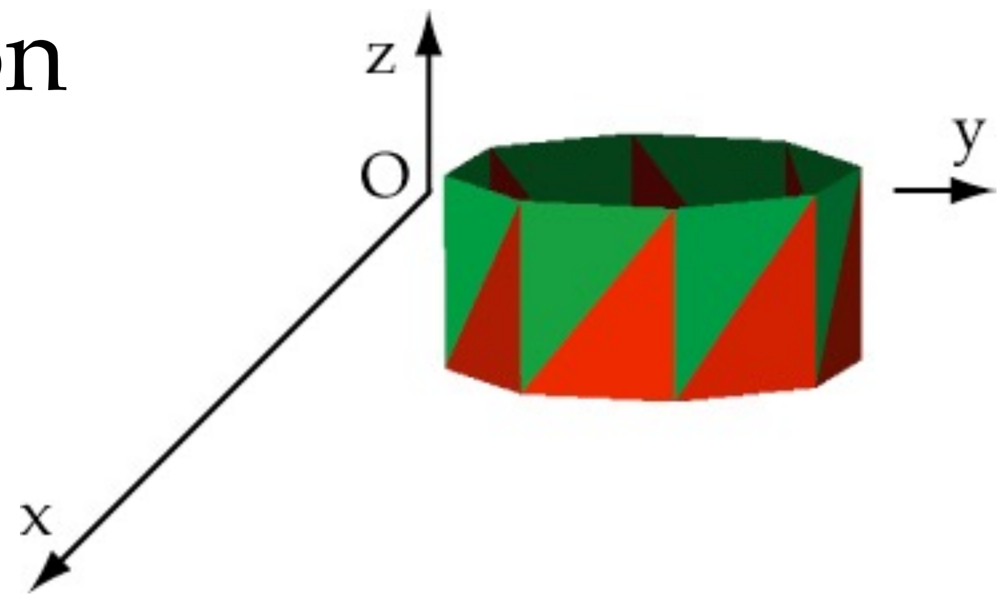
- *Inverse* rotation given by *transpose*
  - only works for (orthonormal) rotations

$$
\begin{aligned}
RR^T &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos^2\theta + \sin^2\theta & -\cos\theta\sin\theta + \sin\theta\cos\theta \\ 0 & -\cos\theta\sin\theta + \sin\theta\cos\theta & \sin^2\theta + \cos^2\theta \end{bmatrix} \\
&= I \\
R^{-1} &= R^T
\end{aligned}
$$

Scaling

- Shrink or grow one coordinate, but not the others
- Negative scale is reflection

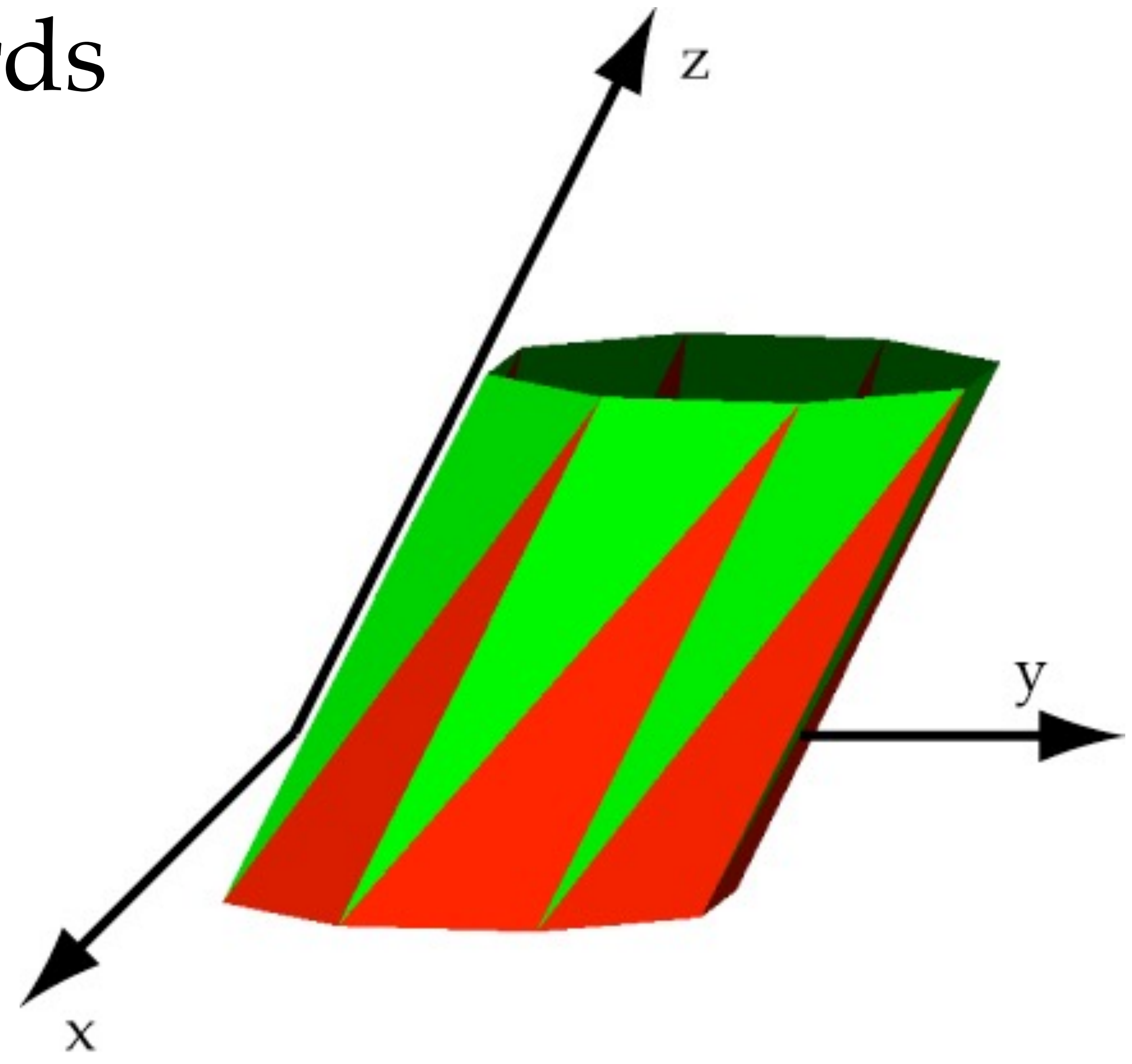
$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.25 \end{bmatrix}$$



Shearing

- Slide the top sideways
  - adds 0.5z to y coords

$$H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$



Translation

- Translation *moves* an object
  - in the direction given by a vector
  - add the vector to each vertex

$$p' = p + \vec{v}$$

- Can't do it with Cartesian matrix multiplication
  - We'll see a way around this next class
  - For now, *assume* that there is a matrix

Normals and Transformations

Normals (the perpendicular vectors to surfaces) play a crucial role in determining how light interacts with objects, which affects shading, reflections, and overall appearance. When transforming objects using rotation, translation, scaling, or shearing, it's important to ensure that the normals remain consistent with the geometry.

- We will transform:
  - vertices
  - normal vectors
- Non-uniform scaling distorts normals
  - as does shearing
  - uniform scaling also causes problems

Rotation and Translation (No Problem for Normals)

- **Rotation**: When you rotate an object, its surface normals rotate correspondingly with the geometry. This is because rotation is a linear transformation, and it preserves the angle and directionality of the normals relative to the surface.

- **Translation**: Translating an object (shifting it in space) does not affect the normals at all, because the normals are based on the relative orientation of the surface, not its absolute position in space.

Thus, **rotation** and **translation** leave the normals unchanged or correctly transformed, and therefore, they don't cause problems in shading or lighting.

Scaling and Shearing (Big Problem for Normals):

- **Scaling**: When scaling, especially non-uniform scaling (where the scale factors differ along different axes), the object's geometry gets distorted. If you directly apply the scaling transformation to the normals, they can end up either lengthened or skewed, leading to incorrect lighting calculations.
  For instance, if you scale an object along the x-axis but not along the y-axis, the surface normal that should be perpendicular to the surface might no longer be so after scaling, distorting how light interacts with it. The normal vector will also no longer have a unit length, which is required for proper lighting calculations.

- **Shearing**: Shearing also distorts the geometry, where surfaces are "stretched" or "slanted" unevenly, and similarly distorts the normals. Like scaling, shearing can produce normals that no longer correctly represent the surface orientation, leading to incorrect lighting and shading.

Distorted Normals

- Not a problem for rotation / translation
  - BIG problem for scaling / shearing

$$n \cdot p - c = 0$$

$$\left(\begin{bmatrix}2 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}n_x\\n_y\\n_z\end{bmatrix}\right)\cdot\left(\begin{bmatrix}2 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}p_x\\p_y\\p_z\end{bmatrix}\right) - c = \begin{bmatrix}2n_x\\n_y\\n_z\end{bmatrix}\cdot\begin{bmatrix}2p_x\\p_y\\p_z\end{bmatrix} - c$$

$$= 4n_xp_x + n_yp_y + n_zp_z - c$$

$$= 3n_xp_x + \left(n_xp_x + n_yp_y + n_zp_z\right) - c$$

$$= 3n_xp_x + n \cdot p - c$$

$$= 3n_xp_x$$

Solutions

- Use the **normal matrix** (inverse transpose of the 3x3 upper-left part of the transformation matrix) is the standard solution used in graphics to ensure normals are transformed correctly when scaling or shearing is applied.

- Calculate normals *after* scaling / shearing
  - more work for the programmer
  - but often done in modelling software

Or avoid scaling and shearing

  - e.g. specify cylinder *height* and *radius*

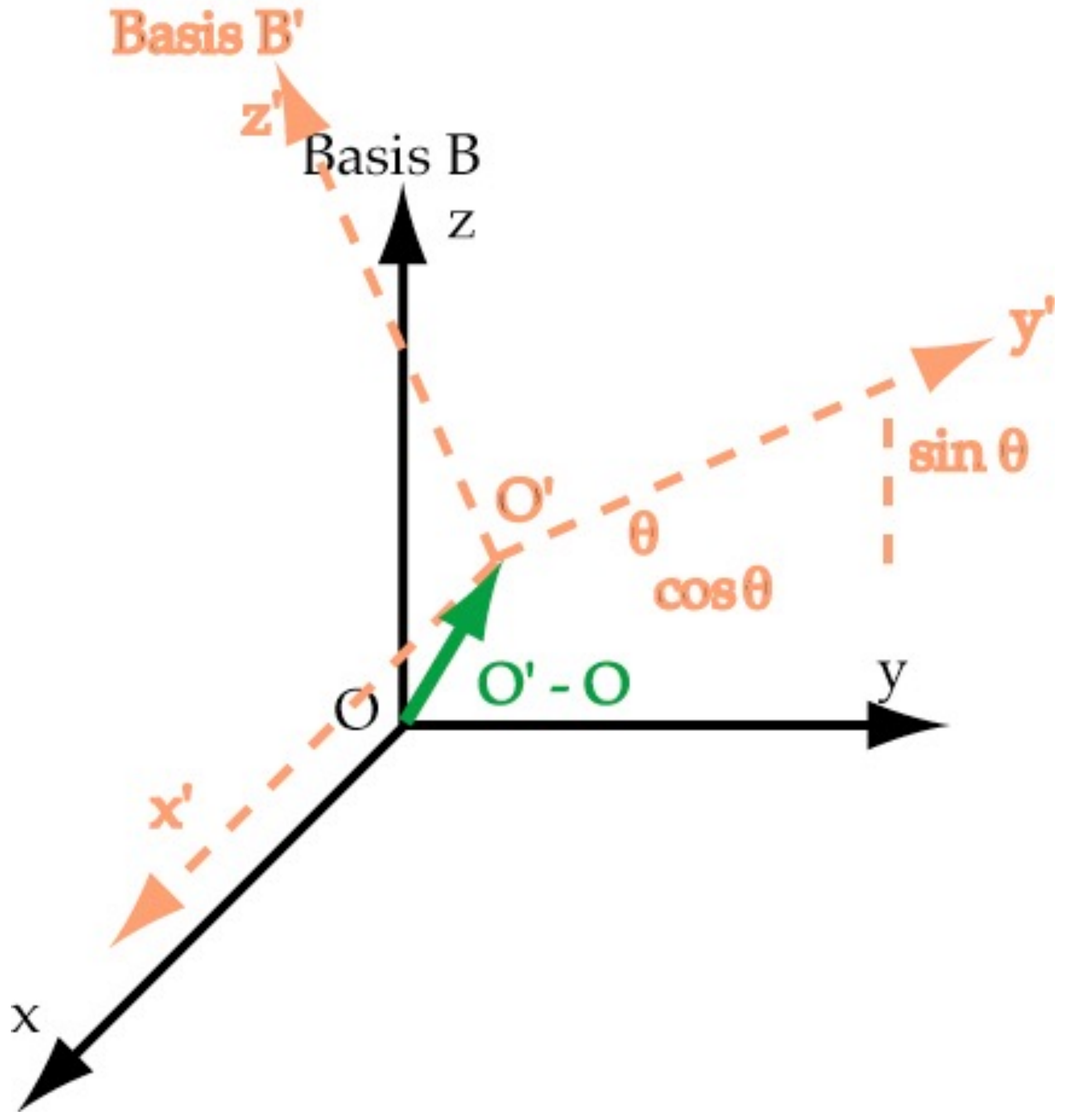
Meaning of Transformations

Two possible interpretations:

- transformation is *applied* to an object
  - used when *animating* an object
- transformation *resets* working coordinates
  - i.e. specifies new *basis* for drawing
  - used when *modelling* an object

Arbitrary Rotation

- Translate by (O - O′)
- Rotate at O
- Translate by (O′ - O)

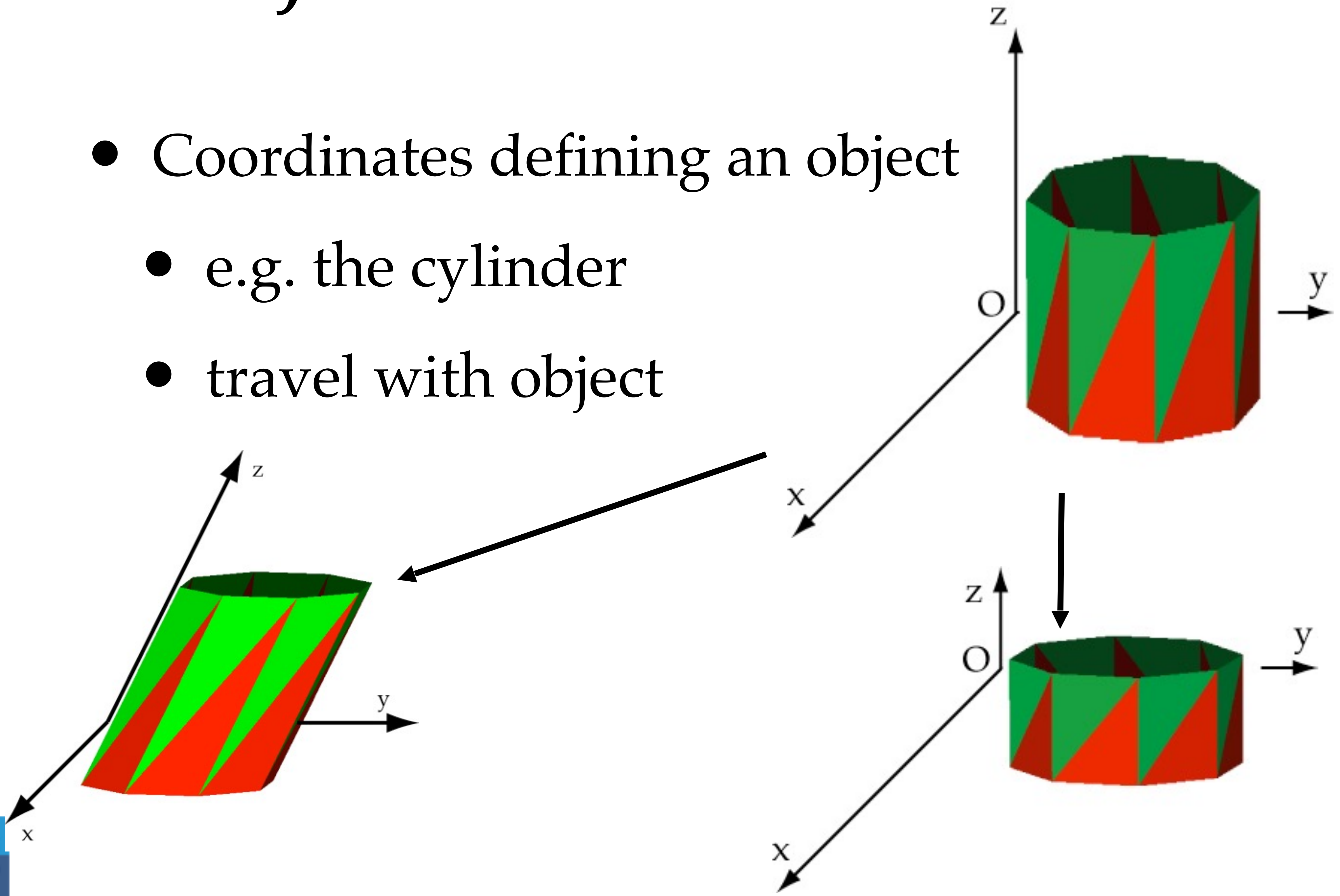


Some Standard Coordinate Systems we have seen

- *OCS* is the *Object Coordinate System*
- *WCS* is the *World Coordinate System*
- *VCS* is the *View Coordinate System*

Object Coordinates

- Coordinates defining an object
  - e.g. the cylinder
  - travel with object



World Coordinates

- Arbitrary coordinate system
- Where is the origin?
    - The Earth?
    - The Sun?
    - Greenwich meridian?
- Used to keep track of other systems

View Coordinates

- Belong to the *eye* or *camera*
- Tilt your head 90 degrees right
  - "Up" (y) is now to the right
  - "Backward" (z) hasn't changed
  - We can use cross-product to get x

Applying Transformations

- Rotate a cylinder, then translate it:

$$p' = \boxed{Rp + v}$$

- Translate a cylinder, then rotate it:

$$p' = \boxed{R(p + v)}$$

- Specify transformations in *reverse order*

Applying Transformations

Matrix Multiplication Order:

When we apply a transformation to a point in space, you multiply the transformation matrix by the point's position vector. If multiple transformations are applied, you multiply the matrices together first and then apply them to the point. The **last transformation matrix to be multiplied is applied first**.

So, if you want to:

1. **Scale** an object,
2. **Rotate** it,
3. **Translate** it,

You need to **multiply the matrices in reverse order** (i.e., **Translate** → **Rotate** → **Scale**):

T·R·S·p

T

·

R

·

S

·

p

\mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S} \cdot \mathbf{p}

More Information on Transformations

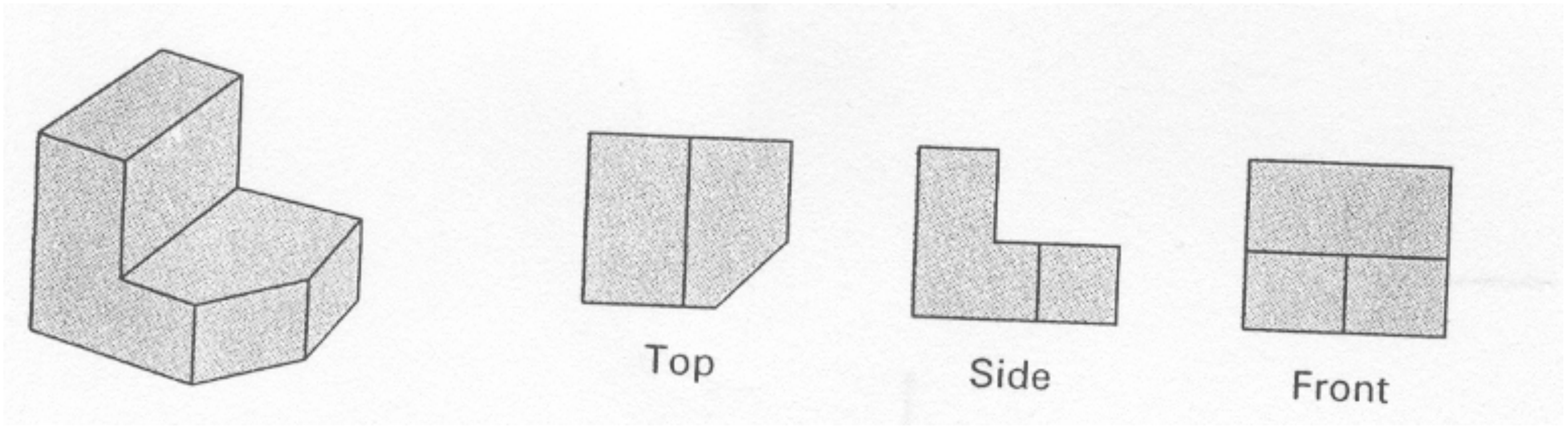
- You may read Chapter 5 in Red book for more information on transformations and Projective transformations

Parallel Projection

- Often used for engineering
  - parallel lines in the world stay parallel
  - distances can be measured
- This is called *parallel* projection
  - *Orthographic* or *Orthogonal*
  - *Oblique* (a slanted view)

Orthographic Projection

- Projection perpendicular to view plane
- Common in science & engineering


Top
Side
Front