CS380: Computer Graphics
Modeling Transformations

Sung-Eui Yoon
(윤성의)

Course URL:
http://sglab.kaist.ac.kr/~sungeui/CG/

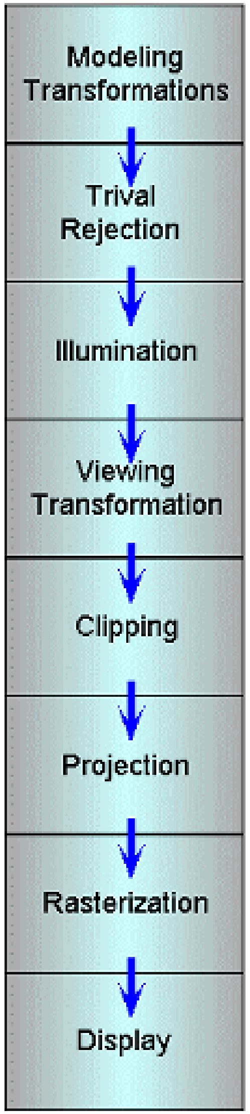
Class Objectives (Ch. 6)

- Know the classic data processing steps, rendering pipeline, for rendering primitives
- Understand 3D translations and rotations

Outline

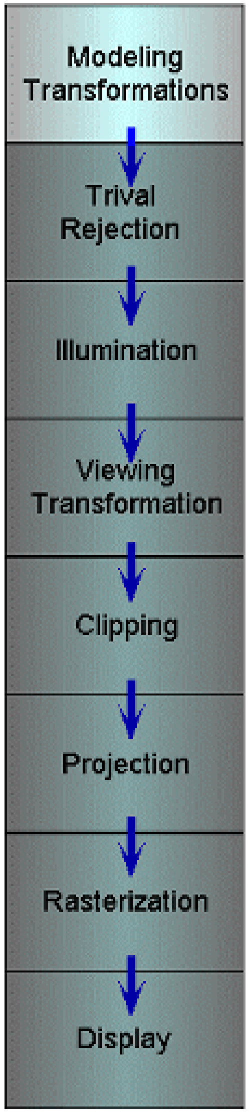
- **Where are we going?**
  - **Sneak peek at the rendering pipeline**
- **Vector algebra**
- **Modeling transformation**
- **Viewing transformation**
- **Projections**

The Classic Rendering Pipeline

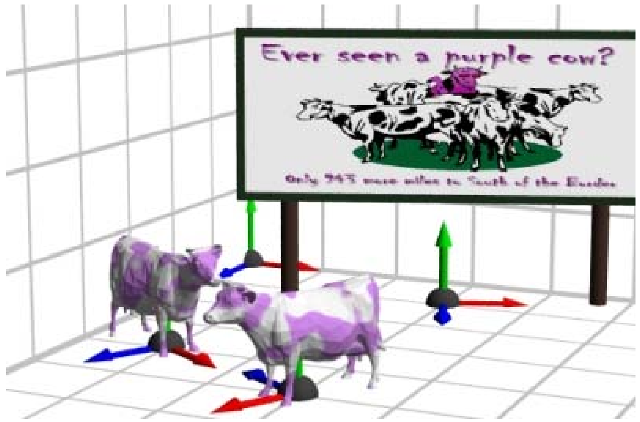


- Object primitives defined by vertices fed in at the top
- Pixels come out in the display at the bottom
- Commonly have multiple primitives in various stages of rendering

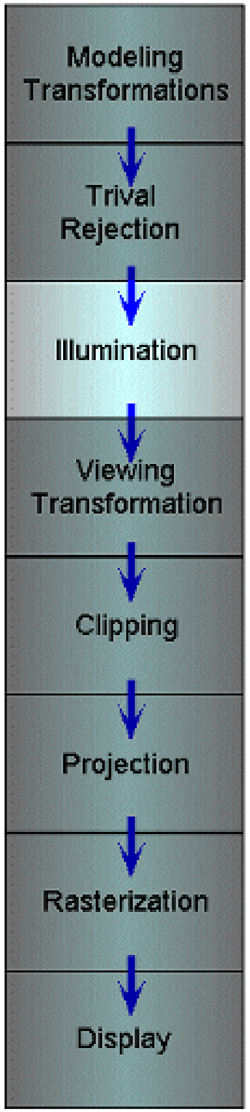
Modeling Transforms



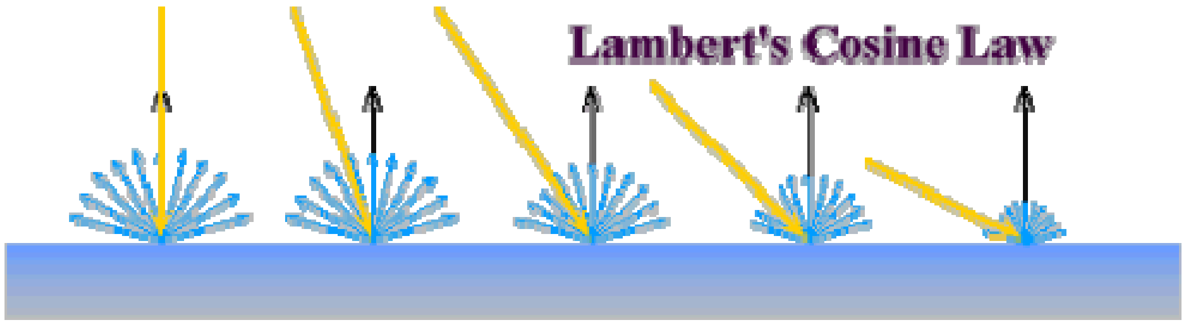
- **Start with 3D models defined in modeling spaces with their own modeling frames:** $\dot{m}_1^t, \dot{m}_2^t, \ldots, \dot{m}_n^t$
- **Modeling transformations orient models within a common coordinate frame called world space,** $\dot{w}^t$
  - **All objects, light sources, and the camera live in world space**
- **Trivial rejection attempts to eliminate objects that cannot possibly be seen**
  - **An optimization**



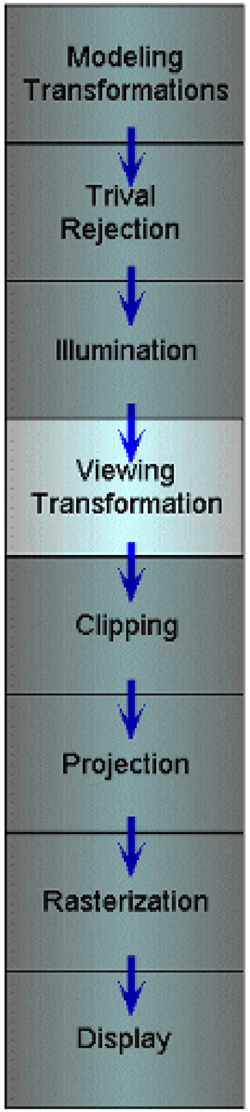
Illumination



- Illuminate potentially visible objects
- Final rendered color is determined by object's orientation, its material properties, and the light sources in the scene



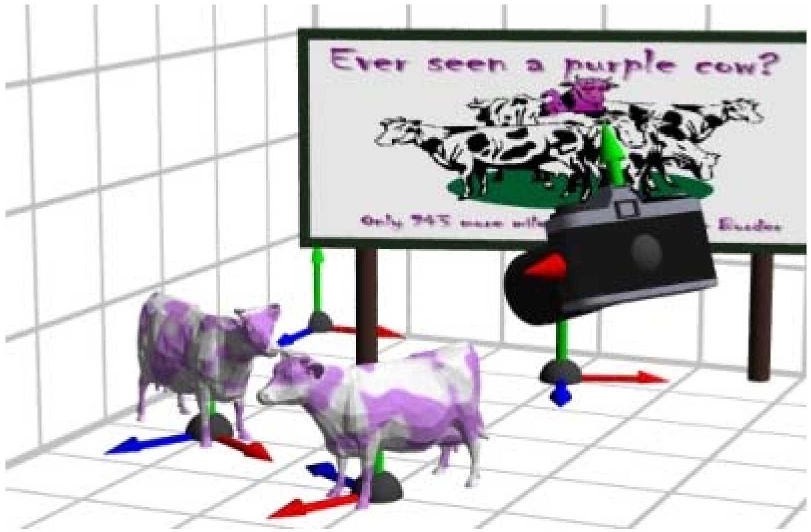
Viewing Transformations



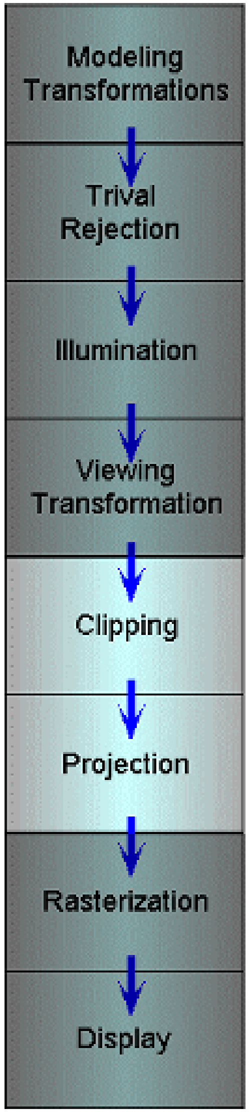
- **Maps points from world space to eye space:**

$$\dot{e}^t = \dot{w}^t \mathbf{V}$$

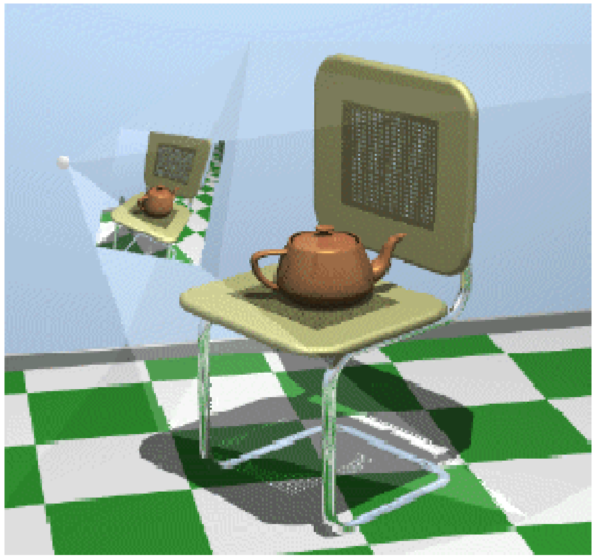
  - **Viewing position is transformed to the origin**
  - **Viewing direction is oriented along some axis**



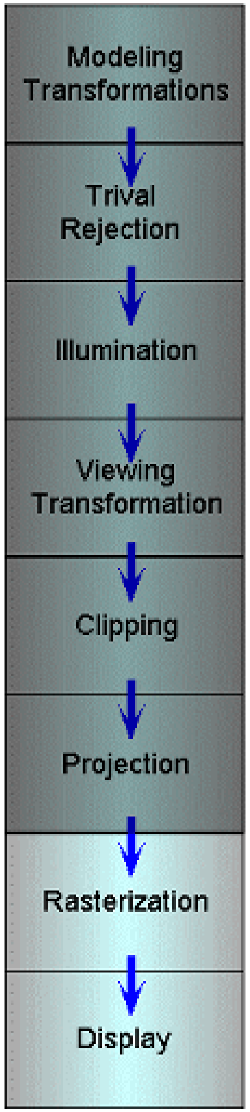
Clipping and Projection



- **We specify a volume called a *viewing frustum***
- **Map the view frustum to the unit cube**
- **Clip objects against the view volume, thereby eliminating geometry not visible in the image**
- **Project objects into two-dimensions**
- **Transform from eye space to normalized device coordinates**



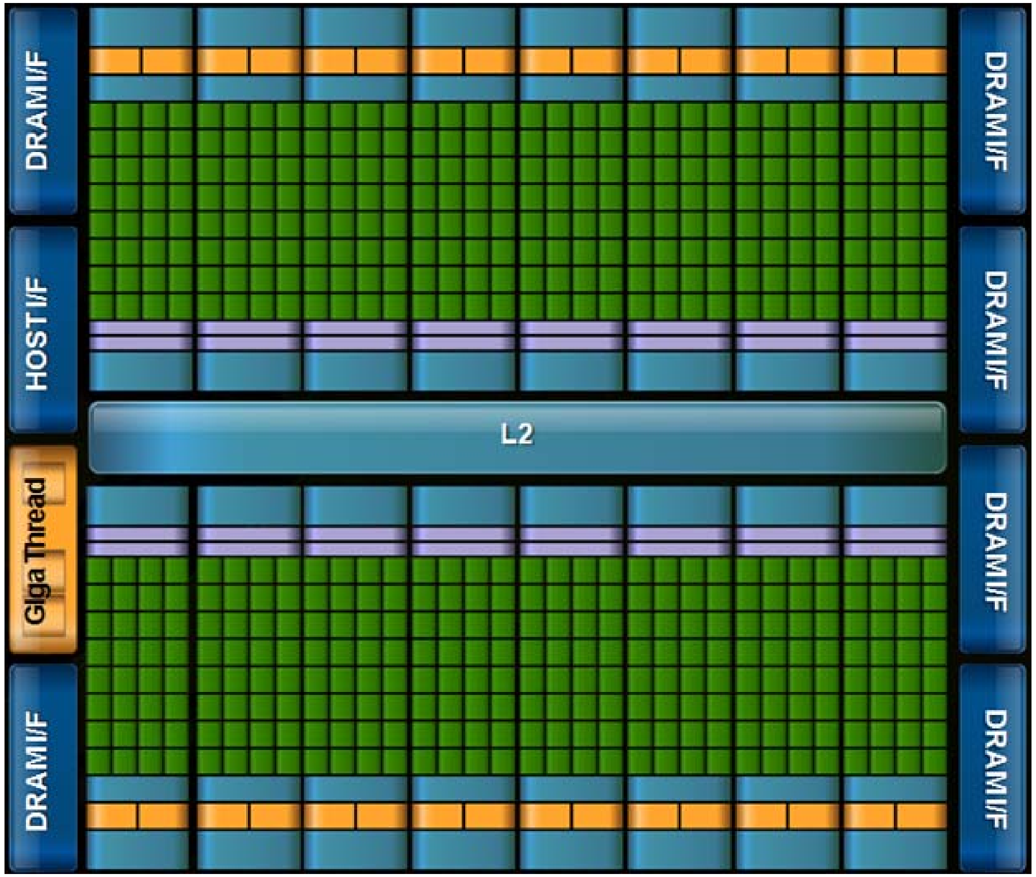
Rasterization and Display



- **Transform normalized device coordinates to screen space**
- **Rasterization converts objects pixels**

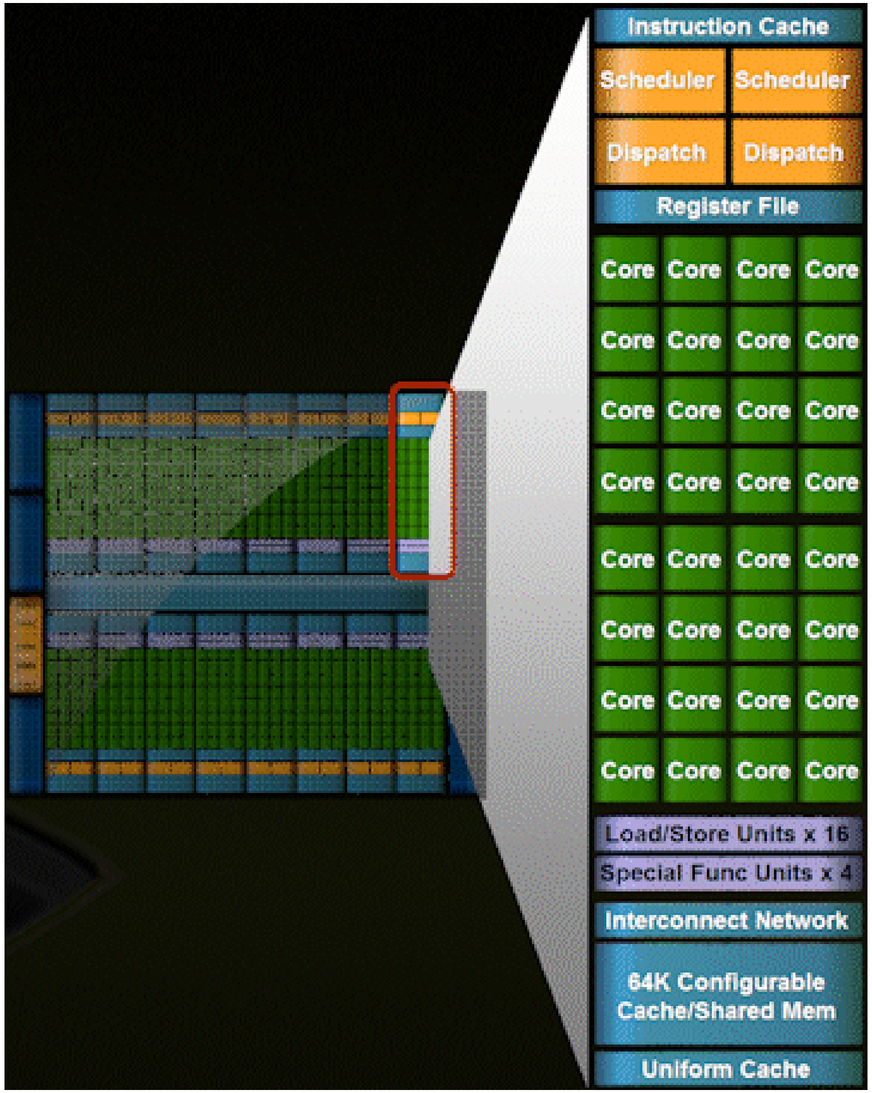
- Almost every step in the rendering pipeline involves a change of coordinate systems!
- Transformations are central to understanding 3D computer graphics

But, this is a architectural overview of a recent GPU (Fermi)

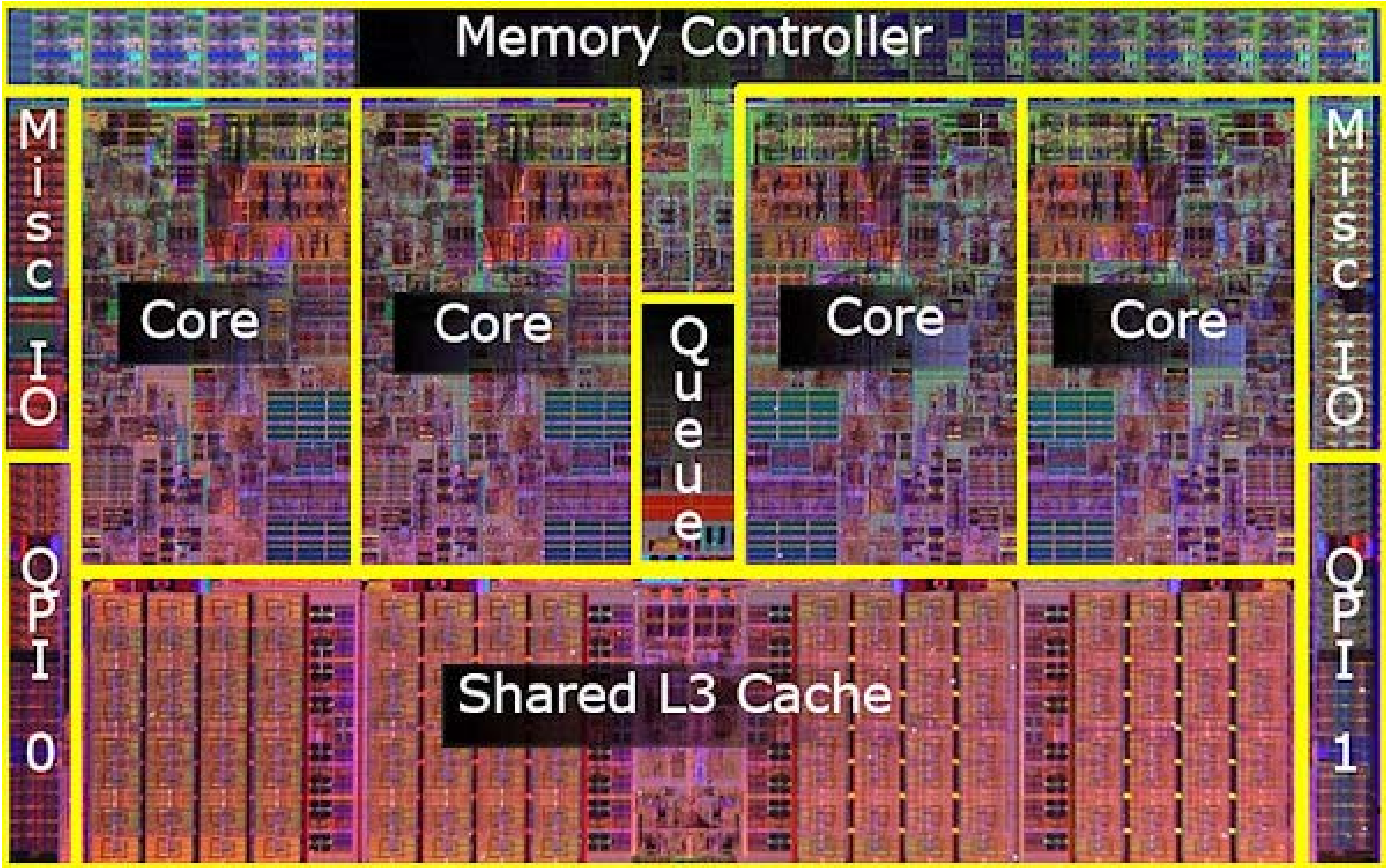


- Unified architecture
- Highly parallel
- Support CUDA (general language)
- Wide memory bandwidth

But, this is a architectural overview of a recent GPU



Recent CPU Chips (Intel's Core i7 processors)



KAIST

Vector Algebra

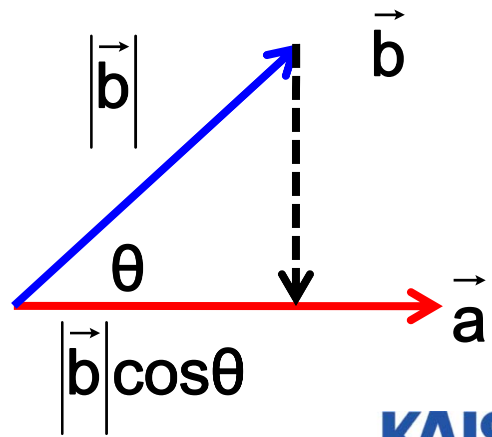
- We already saw vector addition and multiplications by a scalar
- Will study three kinds of vector multiplications
  - Dot product ($\cdot$) - returns a scalar
  - Cross product ($\times$) - returns a vector
  - Tensor product ($\otimes$) - returns a matrix

KAIST

Dot Product (·)

$$\vec{a}\cdot\vec{b} \equiv \vec{a}^T\vec{b} = \begin{bmatrix} a_x & a_y & a_z & 0 \end{bmatrix}\begin{bmatrix} b_x \\ b_y \\ b_z \\ 0 \end{bmatrix} = s, \qquad \vec{a}\cdot\dot{b} \equiv \vec{a}^T\dot{b} = \begin{bmatrix} a_x & a_y & a_z & 0 \end{bmatrix}\begin{bmatrix} b_x \\ b_y \\ b_z \\ 1 \end{bmatrix} = s$$

- **Returns a scalar s**
- **Geometric interpretations s:**
  - $\vec{a}\cdot\vec{b} = |\vec{a}||\vec{b}|\cos\theta$
  - **Length of** $\vec{b}$ **projected onto and** $\vec{a}$ **or vice versa**
  - **Distance of** $\dot{b}$ **from the origin in the direction of** $\vec{a}$



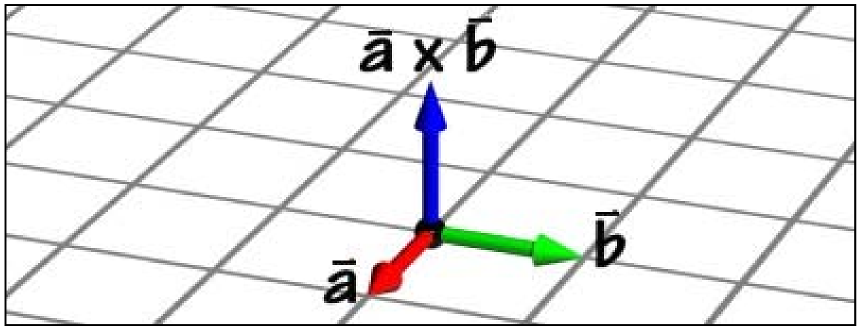
KAIST

Cross Product (×)

$$\vec{a} \times \vec{b} \equiv \begin{bmatrix} 0 & -a_z & a_y & 0 \\ a_z & 0 & -a_x & 0 \\ -a_y & a_x & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \\ 0 \end{bmatrix} = \vec{c} \qquad \begin{matrix} \vec{a} \cdot \vec{c} = 0 \\ \vec{b} \cdot \vec{c} = 0 \end{matrix}$$

$$\vec{c} = \begin{bmatrix} a_y b_z - a_z b_y & a_z b_x - a_x b_z & a_x b_y - a_y b_x \end{bmatrix}$$

- **Return a vector $\vec{c}$ that is perpendicular to both $\vec{a}$ and $\vec{b}$, oriented according to the right-hand rule**
- **The matrix is called the skew-symmetric matrix of $\vec{a}$**



Cross Product (×)

- **A mnemonic device for remembering the cross-product**

$$\vec{a} \times \vec{b} \equiv det \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$

$$= (a_y b_z - a_z b_y)\vec{i} + (a_z b_x - a_x b_z)\vec{j} + (a_x b_y - a_y b_x)\vec{k}$$

$$\vec{i} = [1 \quad 0 \quad 0]$$

$$\vec{j} = [0 \quad 1 \quad 0]$$

$$\vec{k} = [0 \quad 0 \quad 1]$$

Modeling Transformations

- **Vast majority of transformations are modeling transforms**
- **Generally fall into one of two classes**
  - **Transforms that move parts within the model**

$$\dot{m}_1^t \boldsymbol{c} \Rightarrow \dot{m}_1^t \boldsymbol{M}\boldsymbol{c} = \dot{m}_1^t \boldsymbol{c}'$$

  - **Transforms that relate a local model's frame to the scene's world frame**

$$\dot{m}_1^t \boldsymbol{c} \Rightarrow \dot{m}_1^t \boldsymbol{M}\boldsymbol{c} = \dot{w}^t \boldsymbol{c}$$

- **Usually, Euclidean transforms, 3D rigid-body transforms, are needed**

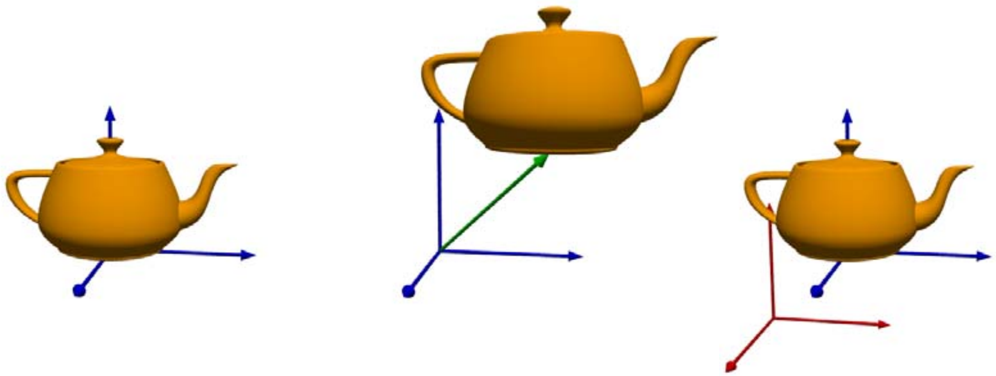
Translations

- **Translate points by adding offsets to their coordinates**

$$\dot{m}^t c \Rightarrow \dot{m}^t T c = \dot{m}^t c'$$
$$\dot{m}^t c \Rightarrow \dot{m}^t T c = \dot{w}^t c$$

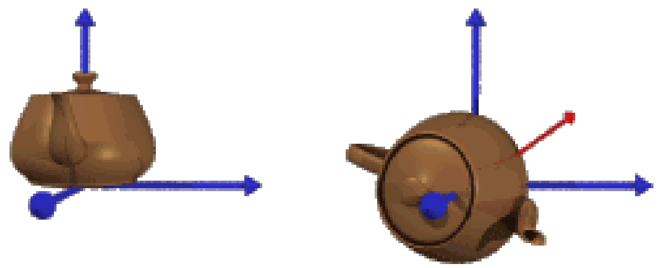
where $T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- **The effect of this translation:**



3D Rotations

- More complicated than 2D rotations
  - Rotate objects along a rotation axis

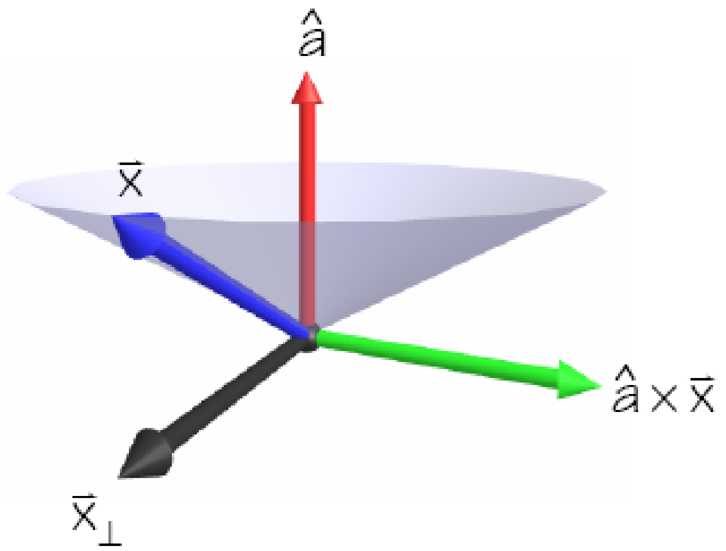


- Several approaches
  - Compose three canonical rotations about the axes
  - Quaternions

KAIST

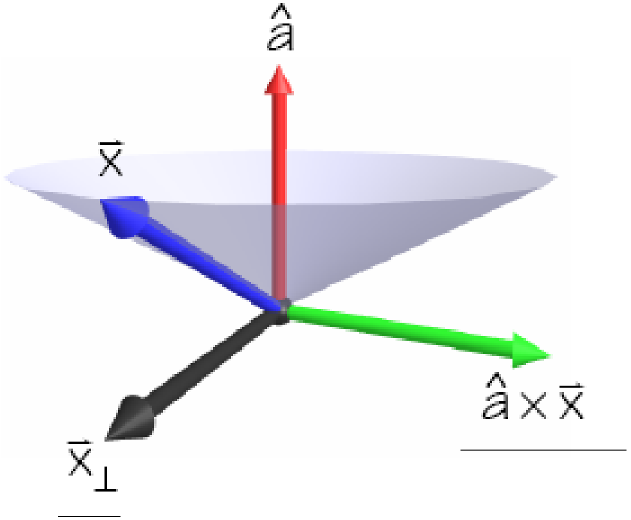
Geometry of a Rotation

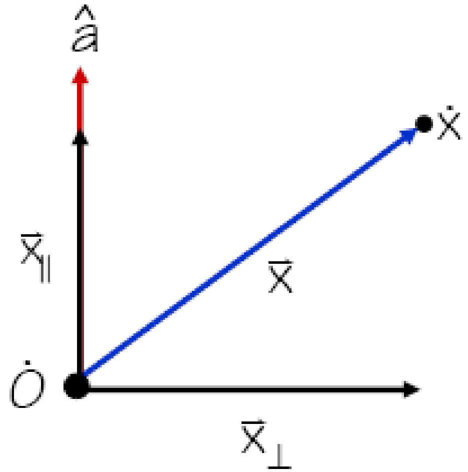
- **Natural basis for rotation of a vector about a specified axis:**
  - $\hat{a}$ - rotation axis (normalized)
  - $\hat{a} \times \vec{x}$ - vector perpendicular to
  - $\vec{x}_\perp$ - perpendicular component of $\vec{x}$ relative to $\hat{a}$

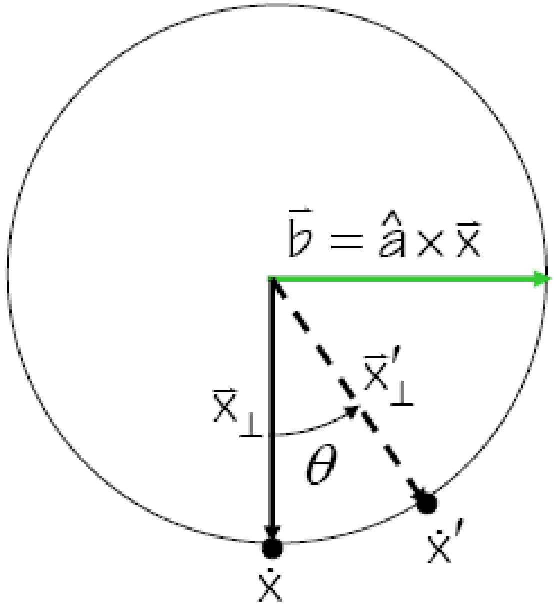


KAIST

Geometry of a Rotation







$$\dot{x}' = \dot{O} + x_{\|} + \vec{x}'_{\perp}$$

$$\vec{x}'_{\perp} = \cos\theta\,\vec{x}_{\perp} + \sin\theta\,\vec{b}$$

$$\vec{x}_{\|} = \hat{a}(\hat{a}\cdot\vec{x})$$

$$\vec{x}_{\perp} = \vec{x} - \vec{x}_{\|}$$

$$\dot{x}' = \dot{O} + \cos\theta\,\vec{x} + (1-\cos\theta)(\hat{a}(\hat{a}\cdot\vec{x})) + \sin\theta(\hat{a}\times\vec{x})$$

$$\mathbf{c}_{\dot{x}'} = \mathbf{M}\mathbf{c}_{\dot{x}}$$

$$\mathbf{M} = diag(\dot{O}) + \cos\theta\, diag([1 \quad 1 \quad 1 \quad 0]^t) + (1-\cos\theta)\mathbf{A}_{\otimes} + \sin\theta\,\mathbf{A}_{\times}$$

Tensor Product (⊗)

$$\vec{a} \otimes \vec{b} \equiv \vec{a}\vec{b}^t = \begin{bmatrix} a_x \\ a_y \\ a_z \\ 0 \end{bmatrix} \begin{bmatrix} b_x & b_y & b_z & 0 \end{bmatrix} = \begin{bmatrix} a_x b_x & a_x b_y & a_x b_z & 0 \\ a_y b_x & a_y b_y & a_y b_z & 0 \\ a_z b_x & a_z b_y & a_z b_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

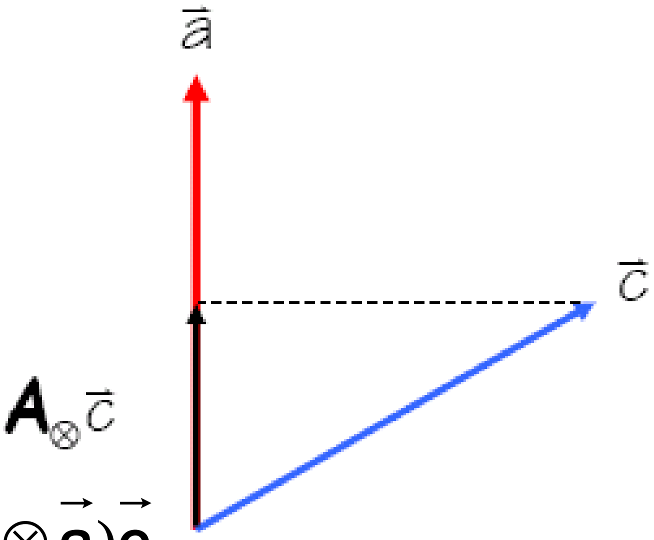
$$(\vec{a} \otimes \vec{b})\vec{c} = \begin{bmatrix} (b_x c_x + b_y c_y + b_z c_z) a_x \\ (b_x c_x + b_y c_y + b_z c_z) a_y \\ (b_x c_x + b_y c_y + b_z c_z) a_z \end{bmatrix} = \vec{a}(\vec{b} \cdot \vec{c})$$

- **Creates a matrix that when applied to a vector $\vec{c}$ return $\vec{a}$ scaled by the project of $\vec{c}$ onto $\vec{b}$**

Tensor Product (⊗)

- **Useful when** $\vec{b} = \vec{a}$
- **The matrix** $\vec{a} \otimes \vec{a}$ **is called the symmetric matrix of** $\vec{a}$
  - **We shall denote this** $A_{\otimes}$

$$A_{\otimes} = \vec{a} \otimes \vec{a} = \begin{bmatrix} a_x a_x & a_x a_y & a_x a_z & 0 \\ a_y a_x & a_y a_y & a_y a_z & 0 \\ a_z a_x & a_z a_y & a_z a_z & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



$$= (\vec{a} \otimes \vec{a})\vec{c}$$

$$= \vec{a}(\vec{a} \cdot \vec{c})$$

KAIST

Sanity Check

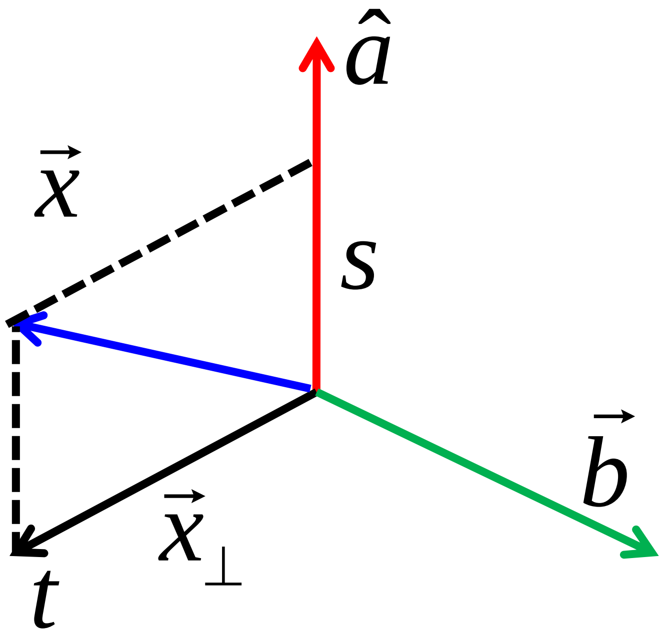
- **Consider a rotation by about the x-axis**

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

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

- **You can check it in any computer graphics book, but you don't need to memorize it**

Rotation using Affine Transformation



Assume that these basis vectors are normalized

$$\begin{bmatrix}\hat{a} & \vec{x}_\perp & \vec{b} & \dot{o}\end{bmatrix}\begin{bmatrix}s\\t\\0\\1\end{bmatrix}$$

$$\downarrow$$

$$\begin{bmatrix}\hat{a} & \vec{x}_\perp & \vec{b} & \dot{o}\end{bmatrix}R_x^{\theta}\begin{bmatrix}s\\t\\0\\1\end{bmatrix}$$

Quaternion

- Developed by W. Hamilton in 1843
  - Based on complex numbers

- Two popular notations for a quaternion, q
  - w + xi + yj + zk, where $i^2 = j^2 = k^2 = ijk = -1$
  - [w, v], where w is a scalar and v is a vector

- Conversion from the axis, v, and angle, t
  - q = [cos (t/2), sin (t/2) v]
  - Can represent rotation

KAIST

Basic Quaternion Operations

- Addition
  - $q + q' = [w + w', v + v']$
- Multiplication
  - $qq' = [ww' - v \cdot v', v \times v' + wv' + w'v]$
- Conjugate
  - $q^* = [w, -v]$
- Norm
  - $N(q) = w^2 + x^2 + y^2 + z^2$
- Inverse
  - $q^{-1} = q^* / N(q)$

KAIST

Basic Quaternion Operations

- q is a unit quaternion if N(q)= 1
  - Then $q^{-1} = q^*$
- Identity
  - [1, (0, 0, 0)] for multiplication
  - [0, (0, 0, 0)] for addition

Rotations using Quaternions

- Suppose that you want to rotate a vector/point v
- Then, the rotated v'
  - $v' = q\, r\, q^{-1}$, where r = [0, v])
- But, what is q?
  - Notice that q is a unit quaternion
- Compositing rotations
  - R = R2 R1 (rotation R1 followed by rotation R2)

KAIST

Example

- **Rotate by degree a along x axis:**
  $q_x = [\cos(a/2), \sin(a/2)\ (1, 0, 0)]$

Quaternion to Rotation Matrix

- $Q = w + xi + yj + zk$
- $$R_m = \begin{vmatrix} 1-2y^2-2z^2 & 2yz+2wx & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{vmatrix}$$
- We can also convert a rotation matrix to a quaternion

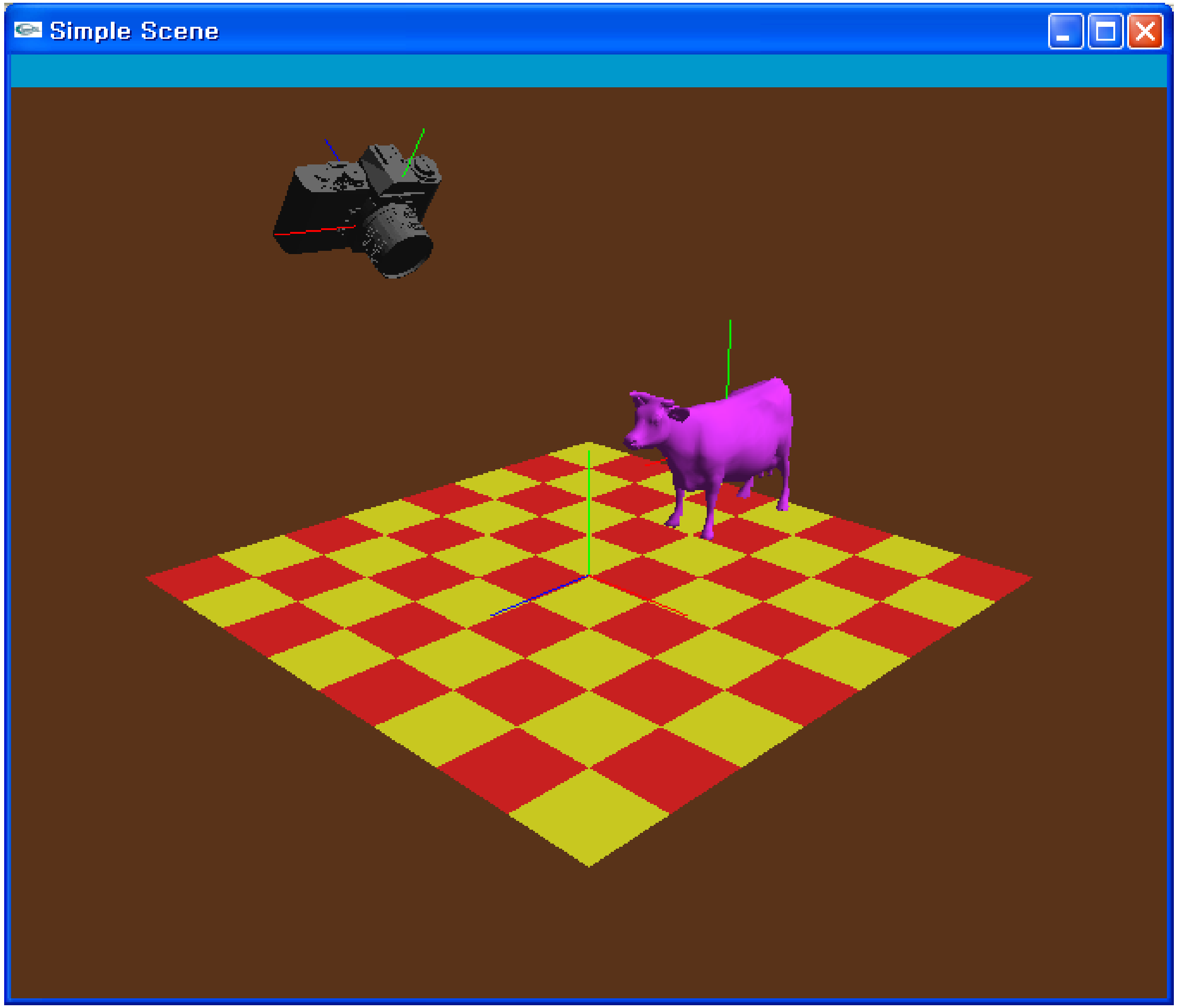
Advantage of Quaternions

- More efficient way to generate arbitrary rotations
- Less storage than 4 x 4 matrix
- Easier for smooth rotation
- Numerically more stable than 4x4 matrix (e.g., no drifting issue)
- More readable

Class Objectives were:

- Know the classic data processing steps, rendering pipeline, for rendering primitives
- Understand 3D translations and rotations

PA2: Simple Animation & Transformation



OpenGL: Display Lists

- Display lists
  - A group of OpenGL commands stored for later executions
  - Can be optimized in the graphics hardware
  - Thus, can show higher performance

- Immediate mode
  - Causes commands to be executed immediately

An Example

```
void drawCow()
{
 if (frame == 0)
 {
  cow = new WaveFrontOBJ( "cow.obj" );
  cowID = glGenLists(1);
  glNewList(cowID, GL_COMPILE);
  cow->Draw();
  glEndList();
 }

 ..
 glCallList(cowID);
 ..
}
```

API for Display Lists

Gluint glGenLists (range)

- generate a continuous set of empty display lists

void glNewList (list, mode) & glEndList ()

: specify the beginning and end of a display list

void glCallLists (list)

: execute the specified display list

OpenGL: Getting Information from OpenGL

```
void main( int argc, char* argv[] )
{
  …
  int rv,gv,bv;
  glGetIntegerv(GL_RED_BITS,&rv);
  glGetIntegerv(GL_GREEN_BITS,&gv);
  glGetIntegerv(GL_BLUE_BITS,&bv);
  printf( "Pixel colors = %d : %d : %d\n", rv, gv, bv );
  ….
}

void display () {
..
glGetDoublev(GL_MODELVIEW_MATRIX, cow2wld.matrix());
..
}
```

Homework

- **Read:**
  - **Ch. 7: Viewing**

- **Watch SIGGRAPH Videos**
- **Go over the next lecture slides**

Next Time

- Viewing transformations