CS380: Computer Graphics
Modeling Transformations

Sung-Eui Yoon
(윤성의)

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

Class Objectives (Ch. 3.5)

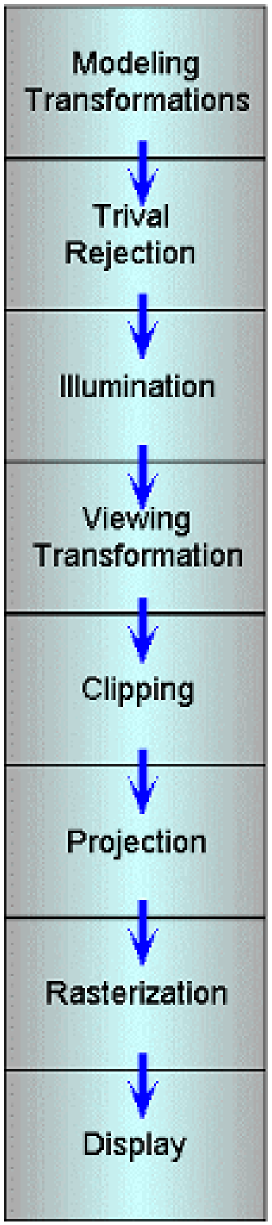
- **Know the classic data processing steps, rendering pipeline, for rendering primitives**
- **Understand 3D translations and rotations as parts of modeling transformation**

- **At the last class:**
  - **Diff. between points and vectors**
  - **Understand the frame**
  - **Diff. transformations in local and global frames**

Outline

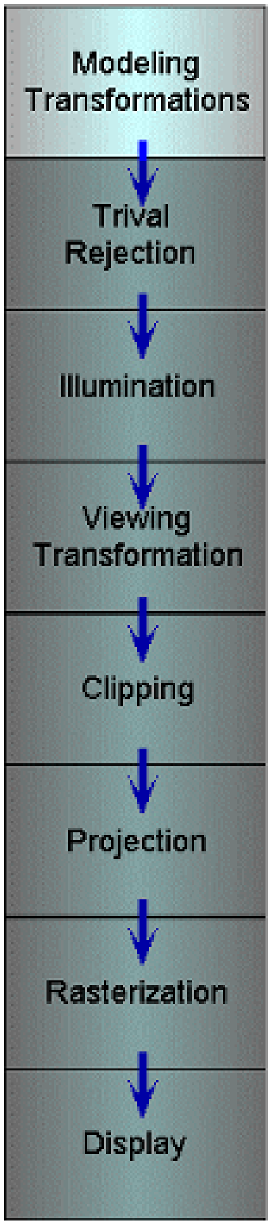
- **Where are we going?**
  - **Sneak peek at the rendering pipeline**
- **Vector algebra**
- **Modeling transformation**
- **Viewing transformation**
- **Projections**

The Classic Rendering Pipeline

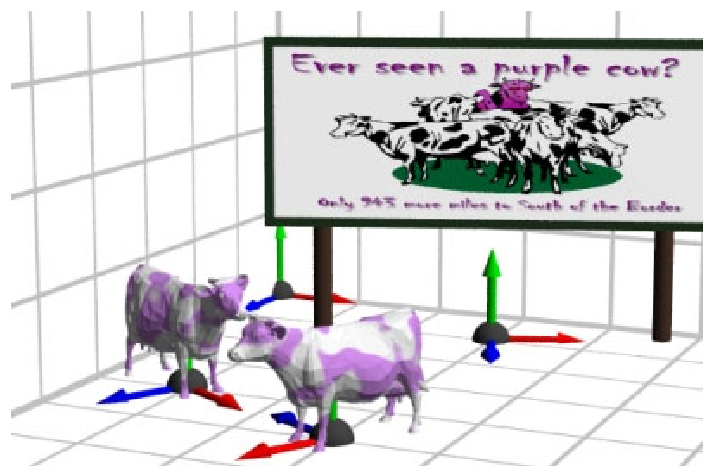


- Object primitives defined by vertices fed in at the top
- Pixels come out in the display at the bottom
- Commonly have multiple primitives in various stages of rendering

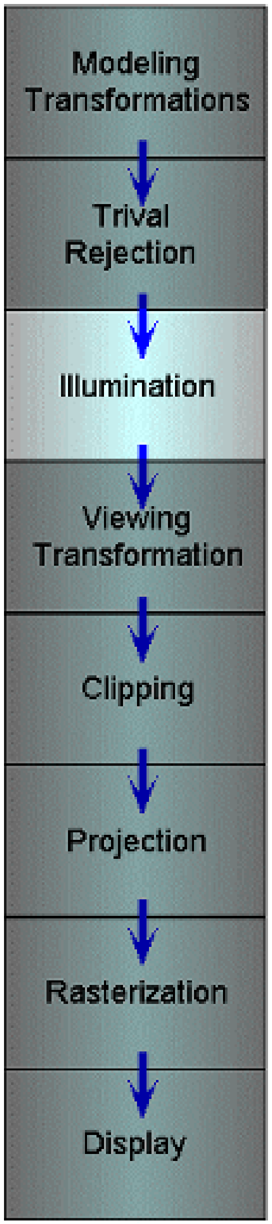
Modeling Transforms



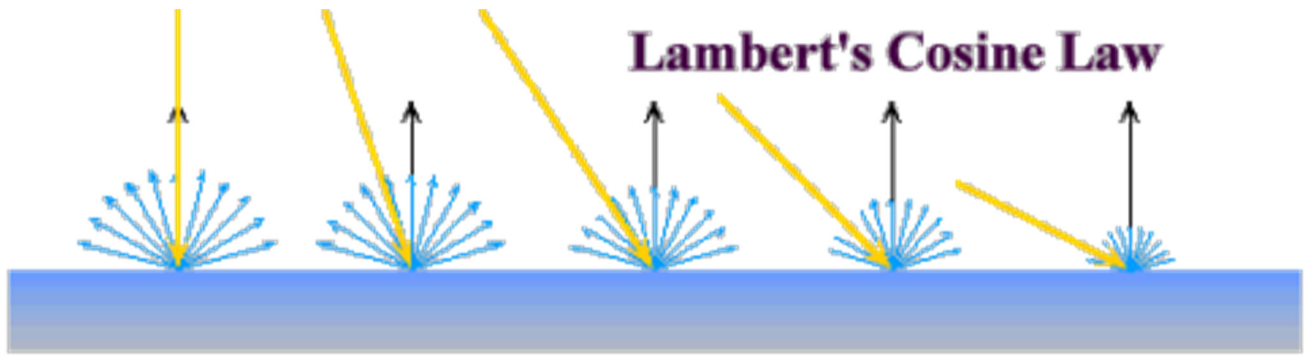
- Start with 3D models defined in **modeling spaces** with their own **modeling frames**: $\dot{m}_1^t, \dot{m}_2^t, \ldots, \dot{m}_n^t$
- Modeling transformations orient models within a common coordinate frame called **world space**, $\dot{w}^t$
  - All objects, light sources, and the camera live in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



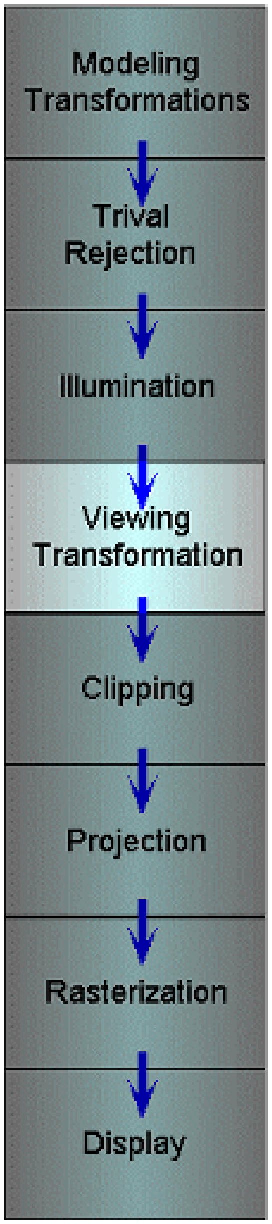
Illumination



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



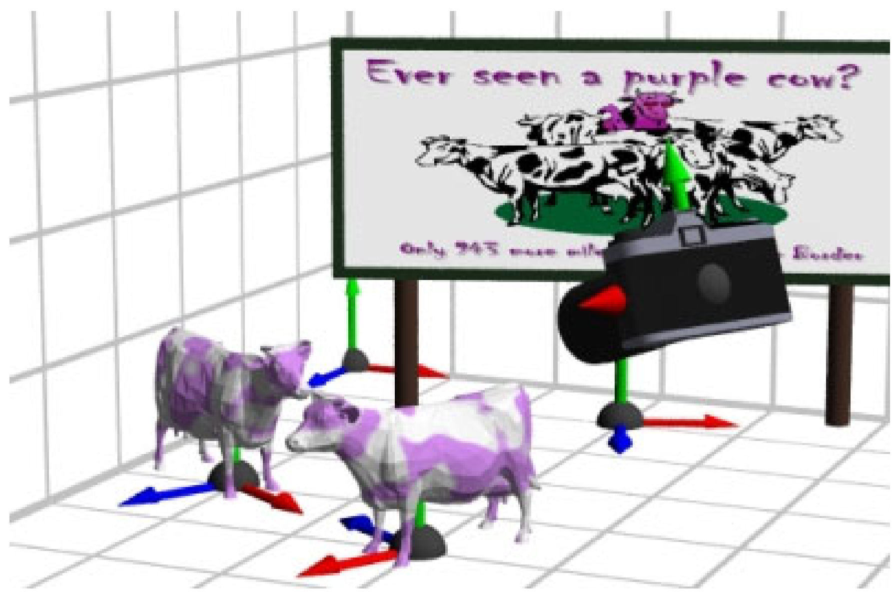
Viewing Transformations



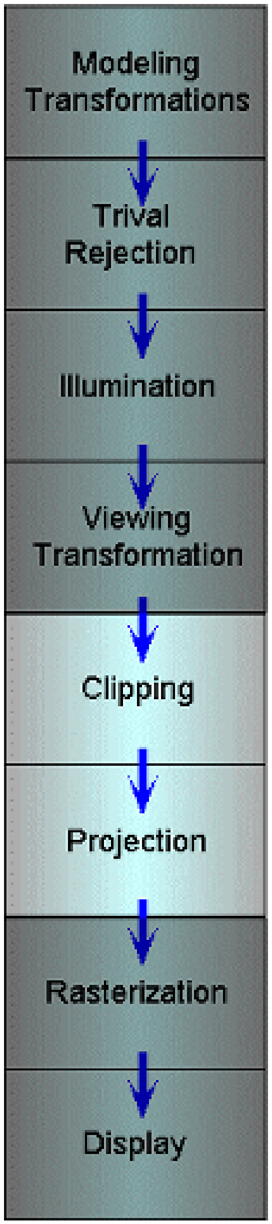
- **Maps points from world space to eye space:**

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

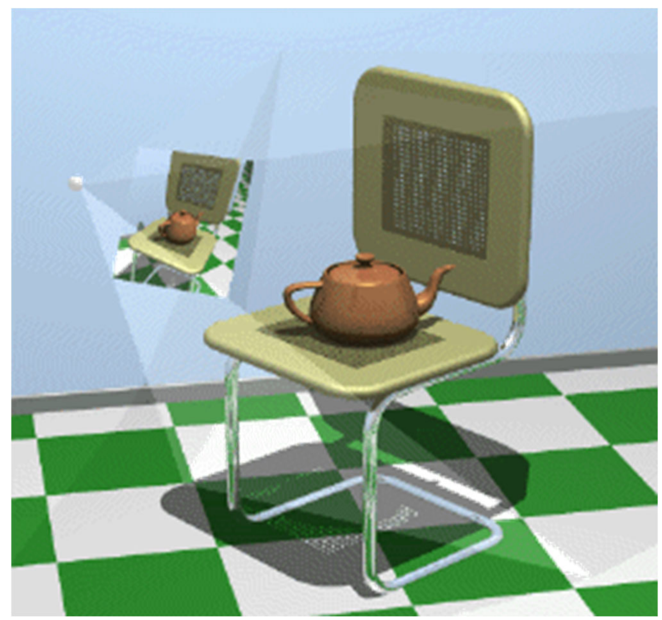
  - **Viewing position is transformed to the origin**
  - **Viewing direction is oriented along some axis**



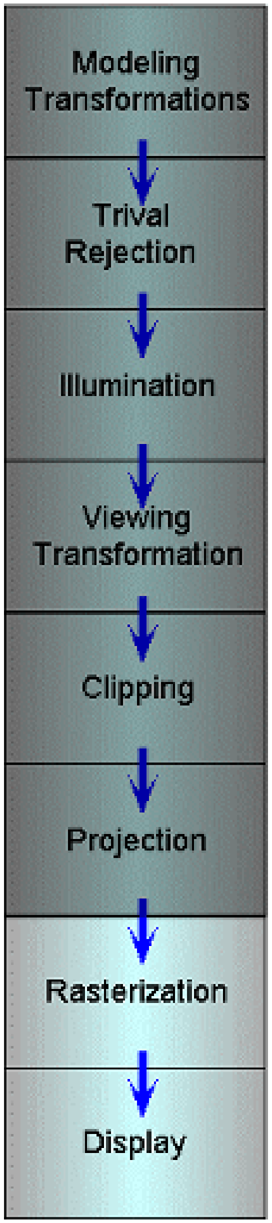
Clipping and Projection



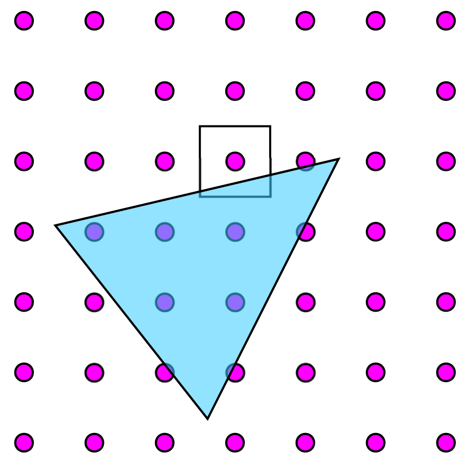
- We specify a volume called a *viewing frustum*
- Clip objects against the view volume, thereby eliminating geometry not visible in the image
- Project objects into two-dimensions



Rasterization and Display

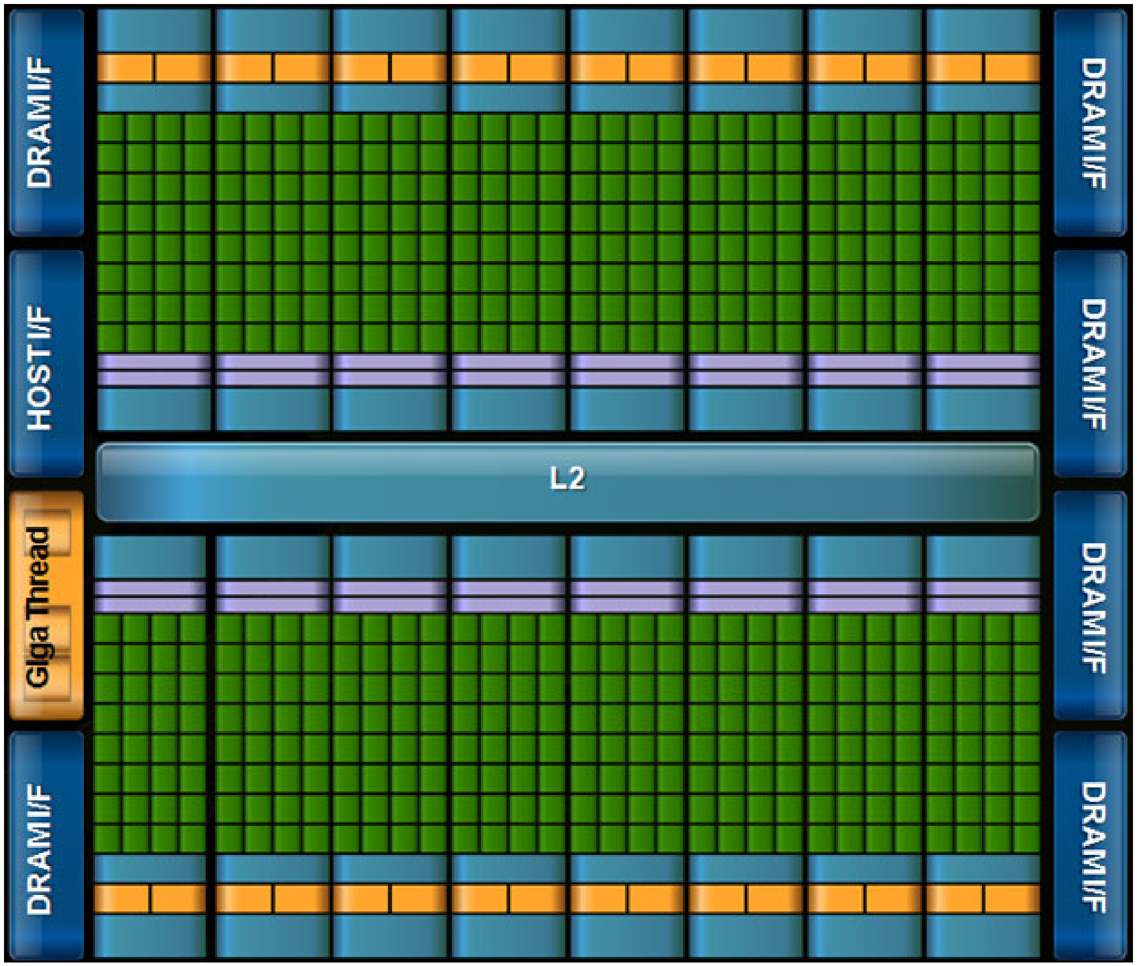


- **Rasterization converts objects pixels**



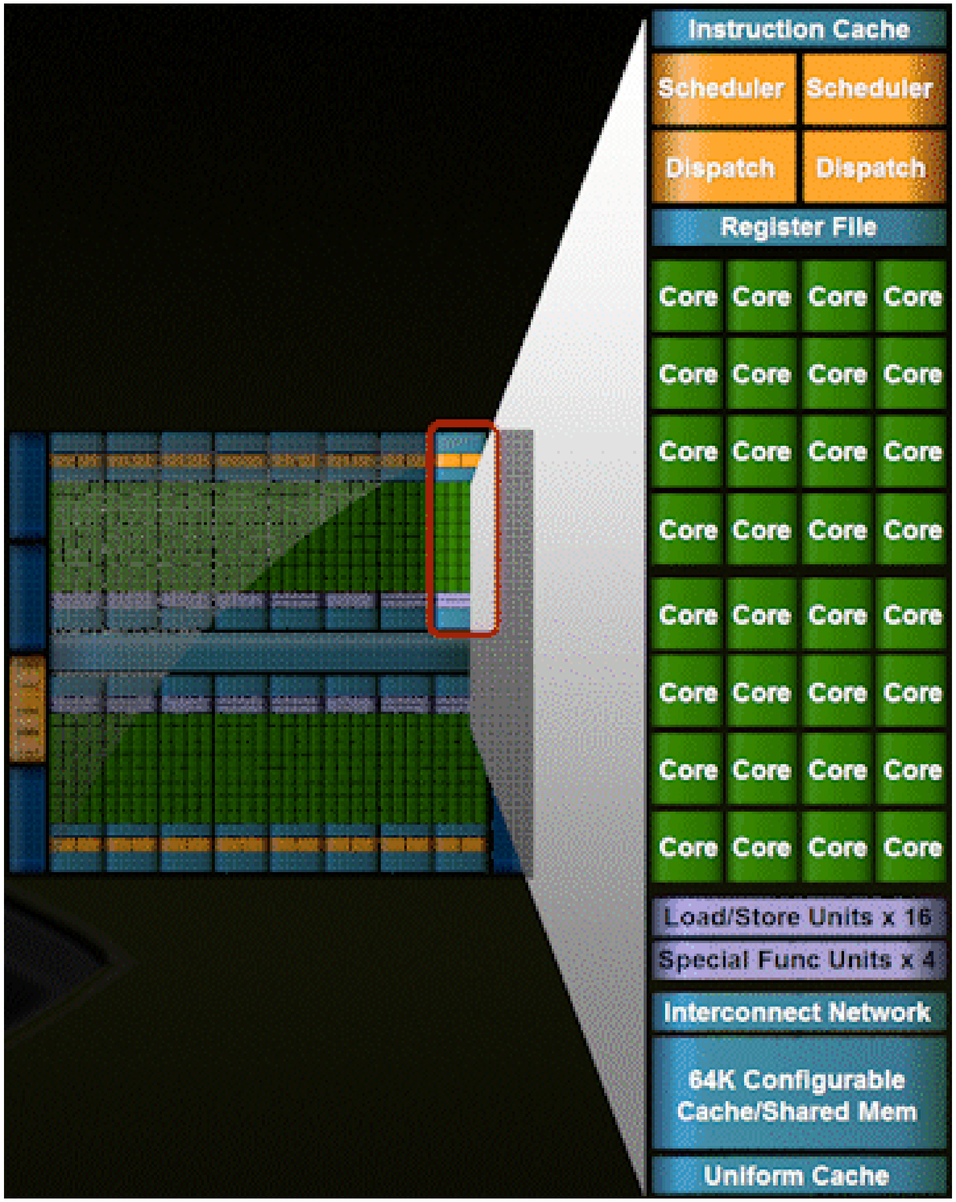
- Almost every step in the rendering pipeline involves a change of coordinate systems!
- Transformations are central to understanding 3D computer graphics

But, this is a architectural overview of a recent GPU (Fermi) around 2010



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

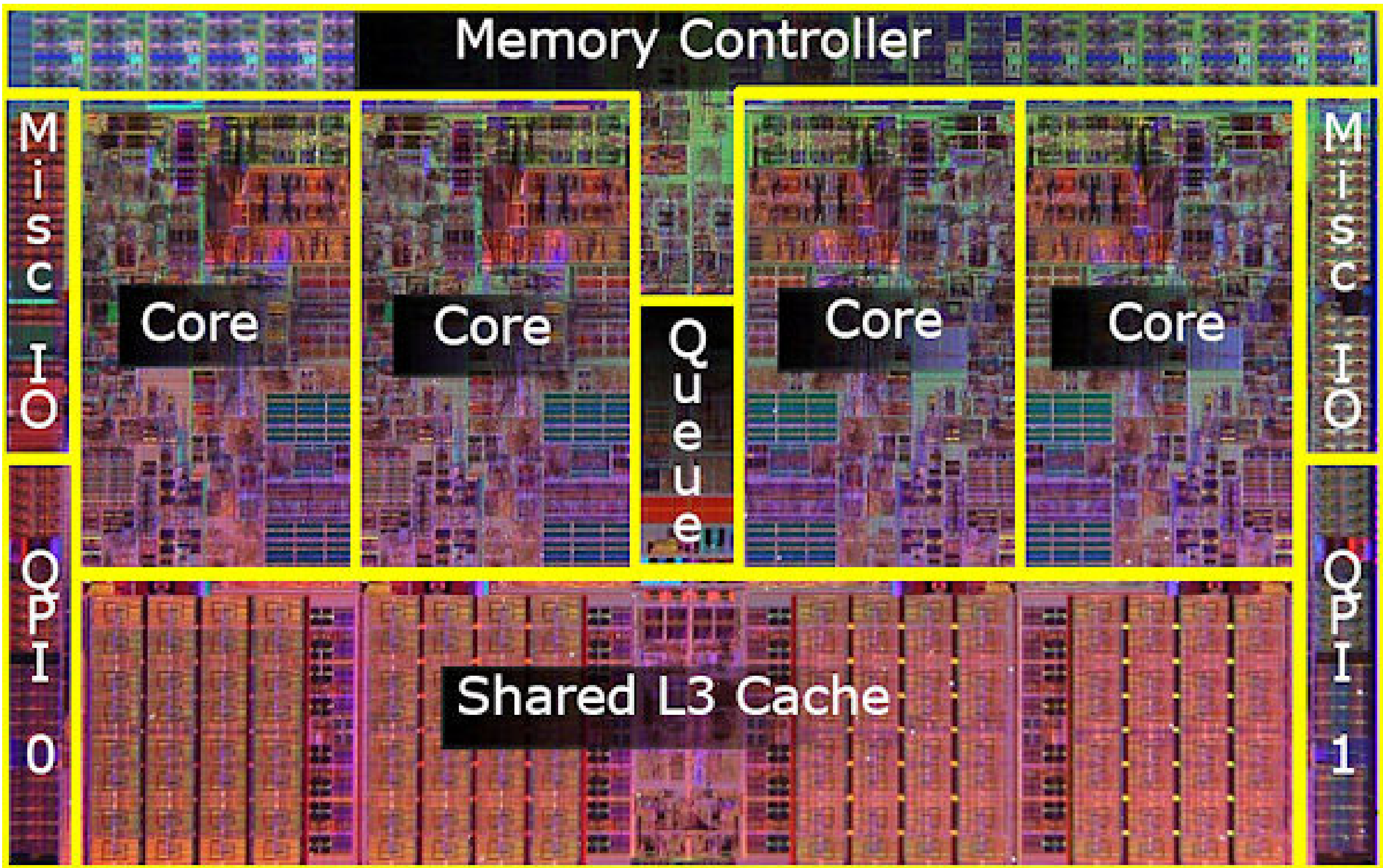
But, this is a architectural overview of a recent GPU



KAIST

Nvidia Hopper Architecture (18K FP32 cores)

Recent CPU Chips (Intel's Core i7 processors) around 2020



Intel Core i9-13900K (USD 650 == 800 Korean won)

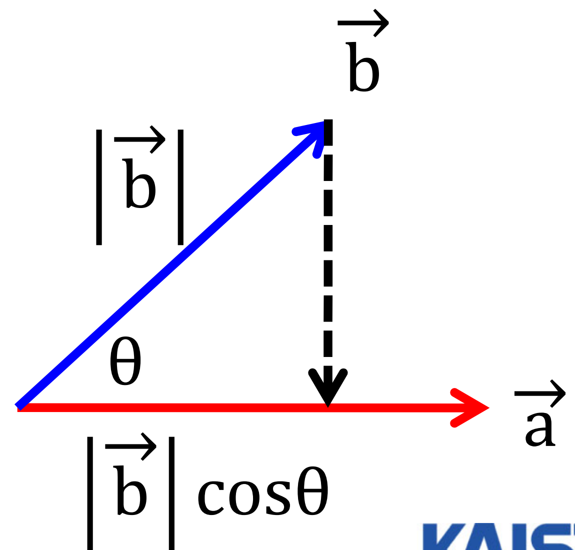
Vector Algebra

- Already saw vector addition and multiplications by a scalar
- Discuss two kinds of vector multiplications
  - Dot product ($\cdot$) - returns a scalar
  - Cross product ($\times$) - returns a vector

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

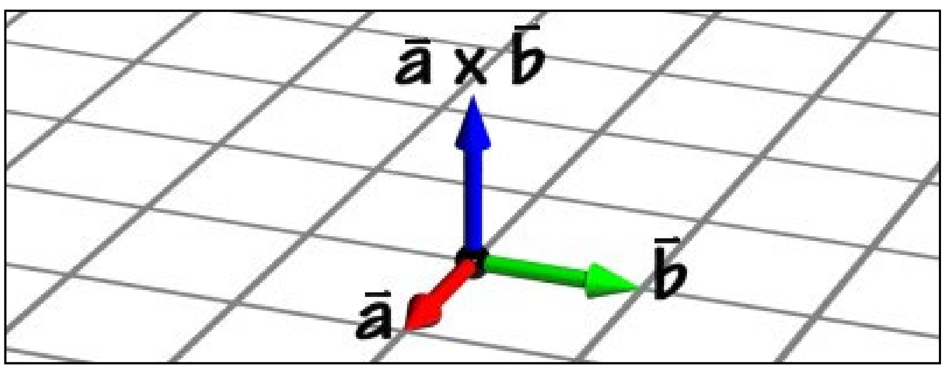


Cross Product (×)

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

$$\vec{c} = [a_y b_z - a_z b_y \quad a_z b_x - a_x b_z \quad a_x b_y - a_y b_x]$$

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



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**

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

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

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

Translations

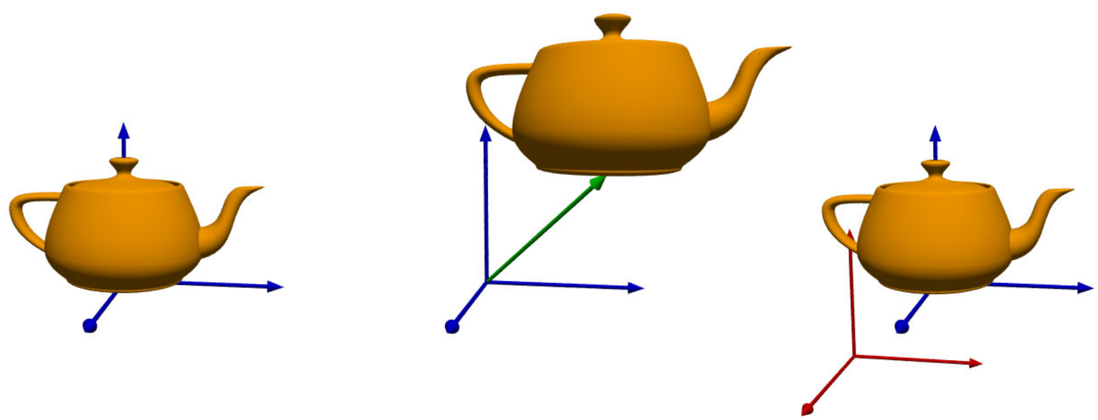
- **Translate points by adding offsets to their coordinates**

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

$$\dot{m}^t c \Rightarrow \dot{m}^t T c = \dot{w}^t c$$

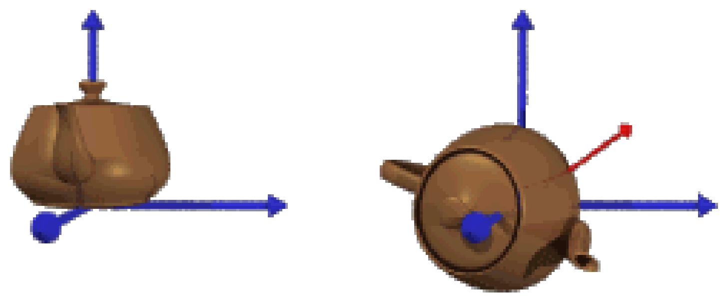
where $T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- **The effect of this translation:**



3D Rotations

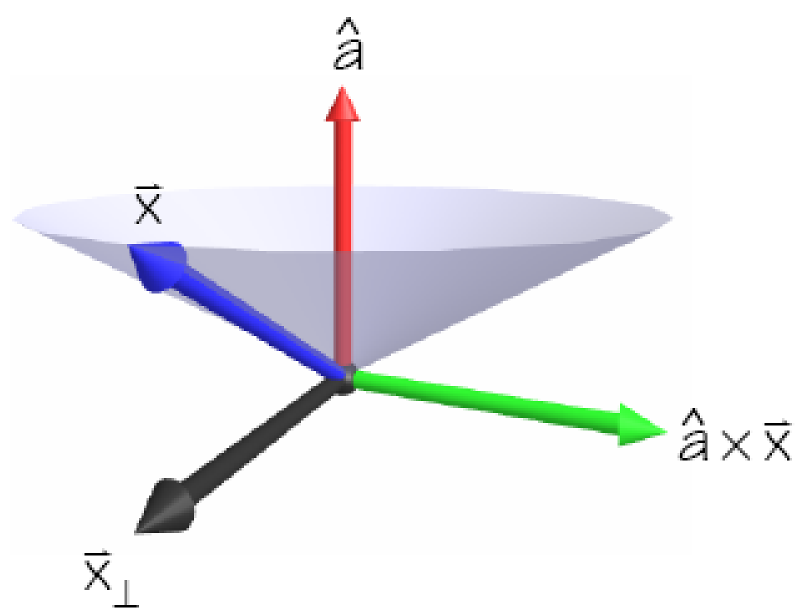
- More complicated than 2D rotations
  - Rotate objects along a rotation axis



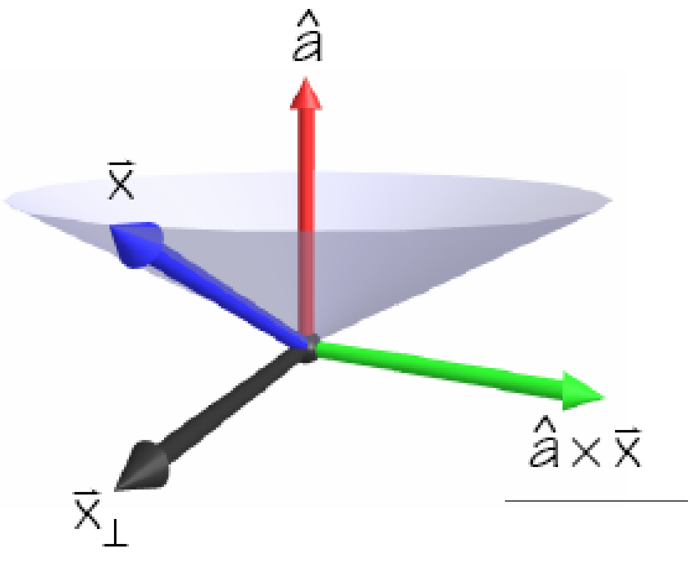
- Several approaches
  - Compose three canonical rotations about the axes
  - Quaternions

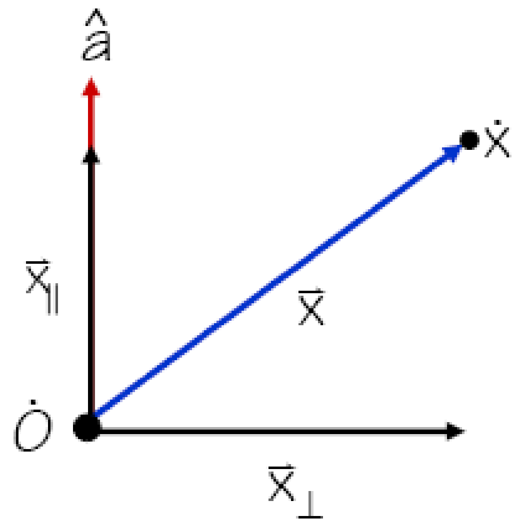
Geometry of a Rotation

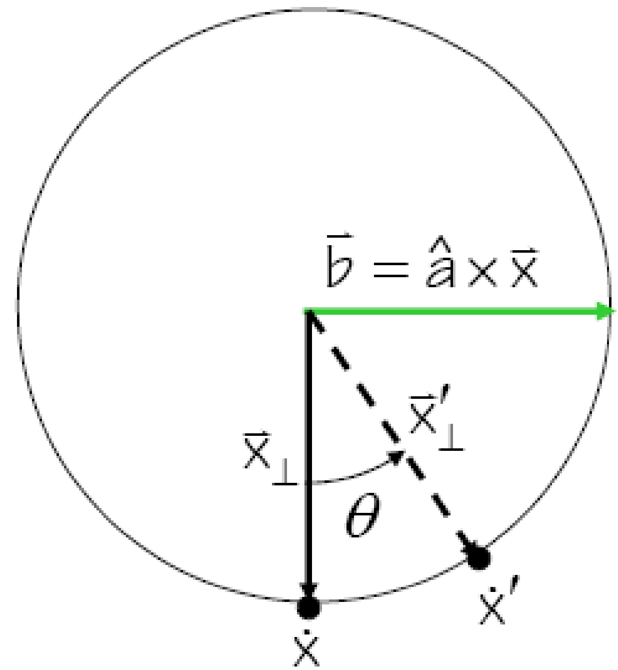
- **Natural basis for rotation of a vector about a specified axis:**
  - $\hat{a}$ - rotation axis (normalized)
  - $\hat{a} \times \vec{x}$ - vector perpendicular to
  - $\vec{x}_\perp$ - perpendicular component of $\vec{x}$ relative to $\hat{a}$



Geometry of a Rotation







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

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

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

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

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

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

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

Tensor Product ($\otimes$)

$$\vec{a}\otimes\vec{b}\equiv\vec{a}\vec{b}^t=\begin{bmatrix}a_x\\a_y\\a_z\\0\end{bmatrix}\begin{bmatrix}b_x & b_y & b_z & 0\end{bmatrix}=\begin{bmatrix}a_xb_x & a_xb_y & a_xb_z & 0\\a_yb_x & a_yb_y & a_yb_z & 0\\a_zb_x & a_zb_y & a_zb_z & 0\\0 & 0 & 0 & 0\end{bmatrix}$$

$$(\vec{a}\otimes\vec{b})\vec{c}=\begin{bmatrix}(b_xc_x+b_yc_y+b_zc_z)a_x\\(b_xc_x+b_yc_y+b_zc_z)a_y\\(b_xc_x+b_yc_y+b_zc_z)a_z\end{bmatrix}=\vec{a}(\vec{b}\cdot\vec{c})$$

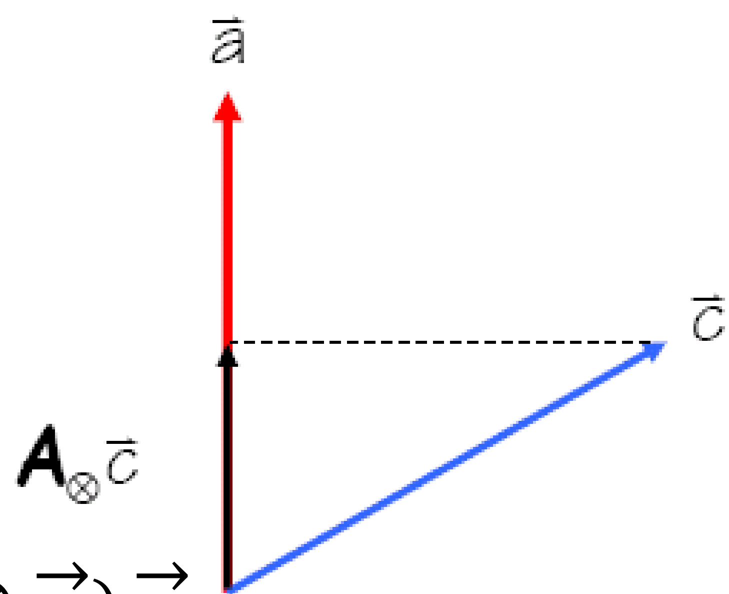
- **Creates a matrix that when applied to a vector $\vec{c}$ return $\vec{a}$ scaled by the project of $\vec{c}$ onto $\vec{b}$**

KAIST

Tensor Product ($\otimes$)

- **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 $A_{\otimes}$**

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



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

Sanity Check

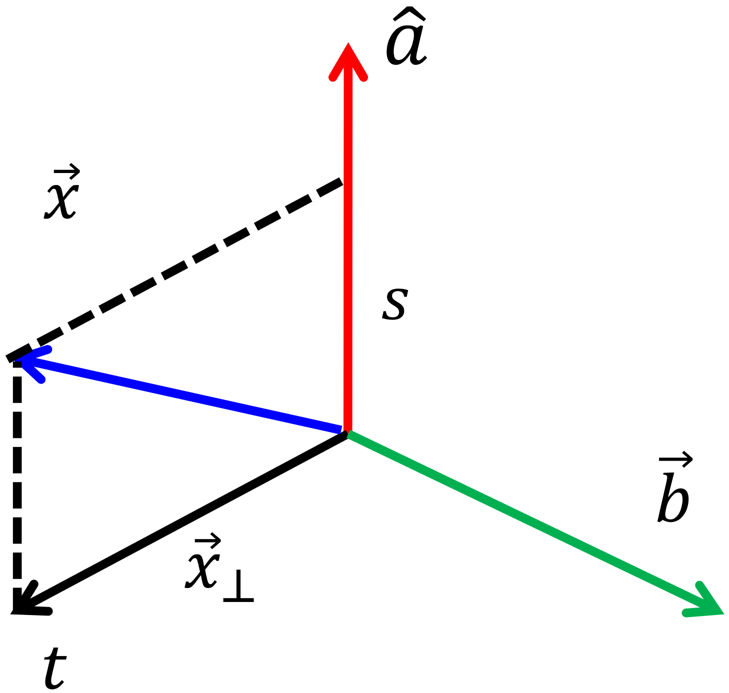
- **Consider a rotation by about the x-axis**

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

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

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

Rotation using Affine Transformation



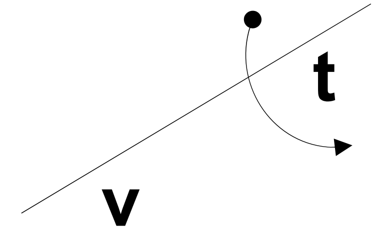
Assume that these basis vectors are normalized

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

$$\downarrow$$

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

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
- Example: rotate by degree a along x axis:

  $q_x = [\cos(a/2), \sin(a/2)\,(1, 0, 0)]$

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**
  - $q^{-1} = q^* / N(q)$

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 with q**
- **Then, the rotated v′**
  - **$v' = q\ r\ q^{-1}$, where $r = [0, v]$)**

- **Compositing rotations**
  - **R = R2 R1 (rotation R1 followed by rotation R2)**

Quaternion to Rotation Matrix

- $Q = w + xi + yj + zk$
- $R_m = \begin{vmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{vmatrix}$

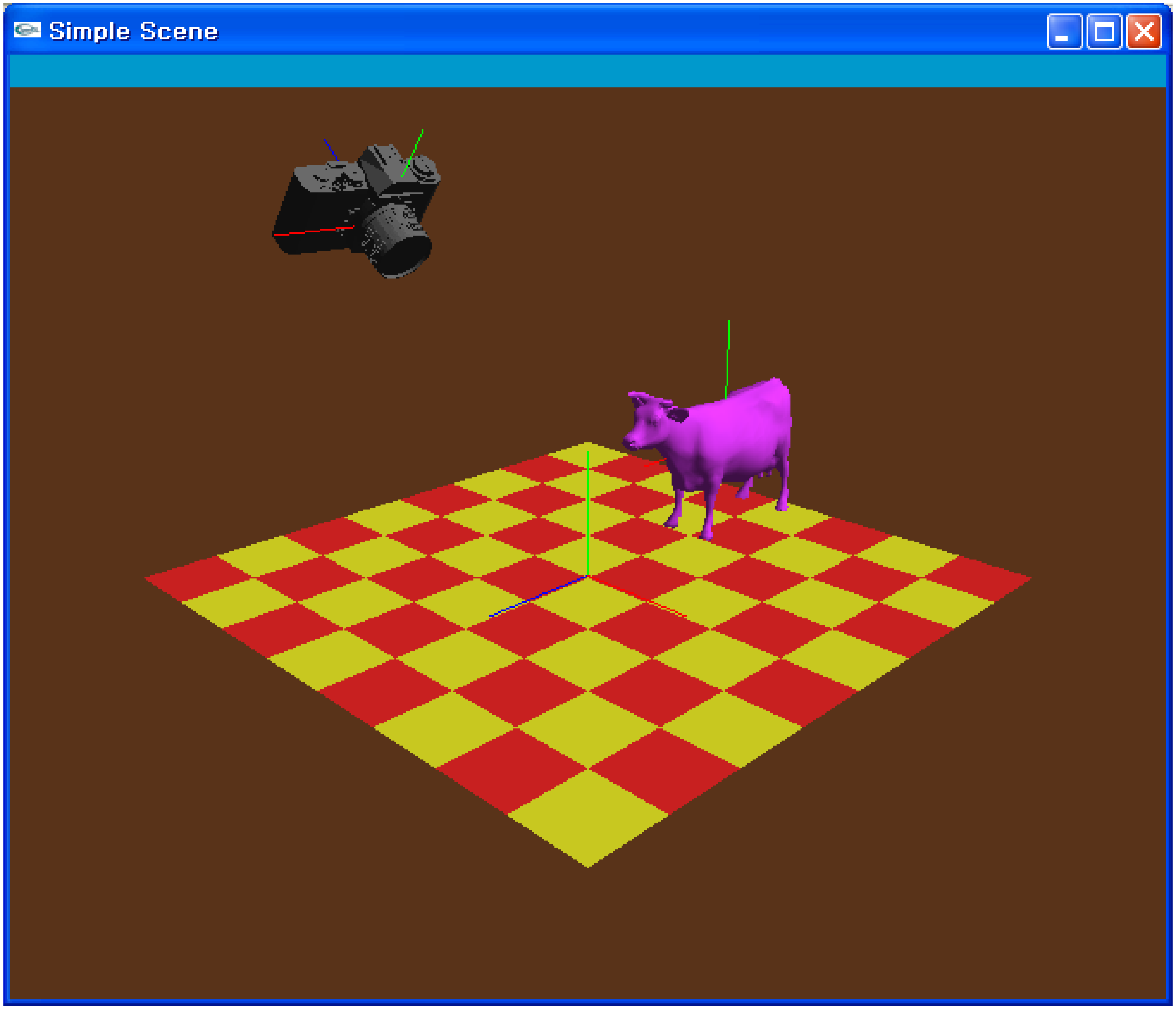
- We can also convert a rotation matrix to a quaternion

KAIST

Advantage of Quaternions

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

PA2: Simple Animation & Transformation



KAIST

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
  - Ver. 4.3: Vertex Array Object is much better
- 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

```
void createTriangle() {
    // Define vertices of the triangle
    float vertices[] = {
        -0.5f, -0.5f, 0.0f,  // Bottom-left
         0.5f, -0.5f, 0.0f,  // Bottom-right
         0.0f,  0.5f, 0.0f   // Top     };

    // Generate VBO and VAO
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    // Bind VAO
    glBindVertexArray(VAO);

    // Bind VBO and set vertex data
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);

    // Specify vertex attribute pointers
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);

    // Unbind VBO and VAO
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
}
```

```
void renderScene() {
    // Clear the color buffer
    glClear(GL_COLOR_BUFFER_BIT);

    // Use the shader program
    glUseProgram(shaderProgram);

    // Bind the VAO
    glBindVertexArray(VAO);

    // Draw the triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // Unbind the VAO
    glBindVertexArray(0);
}
```

Some corresponding recent codes

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());
..
}
```

Class Objectives (Ch. 3.5) were:

- **Know the classic data processing steps, rendering pipeline, for rendering primitives**
- **Understand 3D translations and rotations as parts of modeling transformation**

Homework

- Watch SIGGRAPH Videos
- Go over the next lecture slides
- Try to come up with a question

Next Time

- **Viewing transformations**