Implementing Phong Lighting

Gouraud Shading

Phong Shading

Some slides modified from: David Kabala
Others from: Andries Van Damm, Brown Univ.

Lighting and Shading

- ***Shading*** is the process of ***interpolation* of color** at points in-between those with known lighting,
  typically vertices of triangles or quads in a mesh
  - crucial to real time graphics applications (e.g., games) because calculating illumination at a point is usually expensive.
  - Slow but good: ray-tracing computes lighting for all pixel samples
    Each pixel combines one or more sub-pixel sample for antialiasing,
    but no shading!
- On the GPU systems we use with WebGL,
  the ***vertex shader*** usually computes lighting for each vertex,
  the ***fragment shader*** computes shading for eacy pixel

Why? Flat *vs* Gouraud shading

Flat

Gouraud

Goraud vs Phong Shading

Phong Lighting

- Emissive +
- Ambient +
- Diffuse +
- Specular = →→

Computed per VERTEX

Computed Per PIXEL

Lighting and Shading

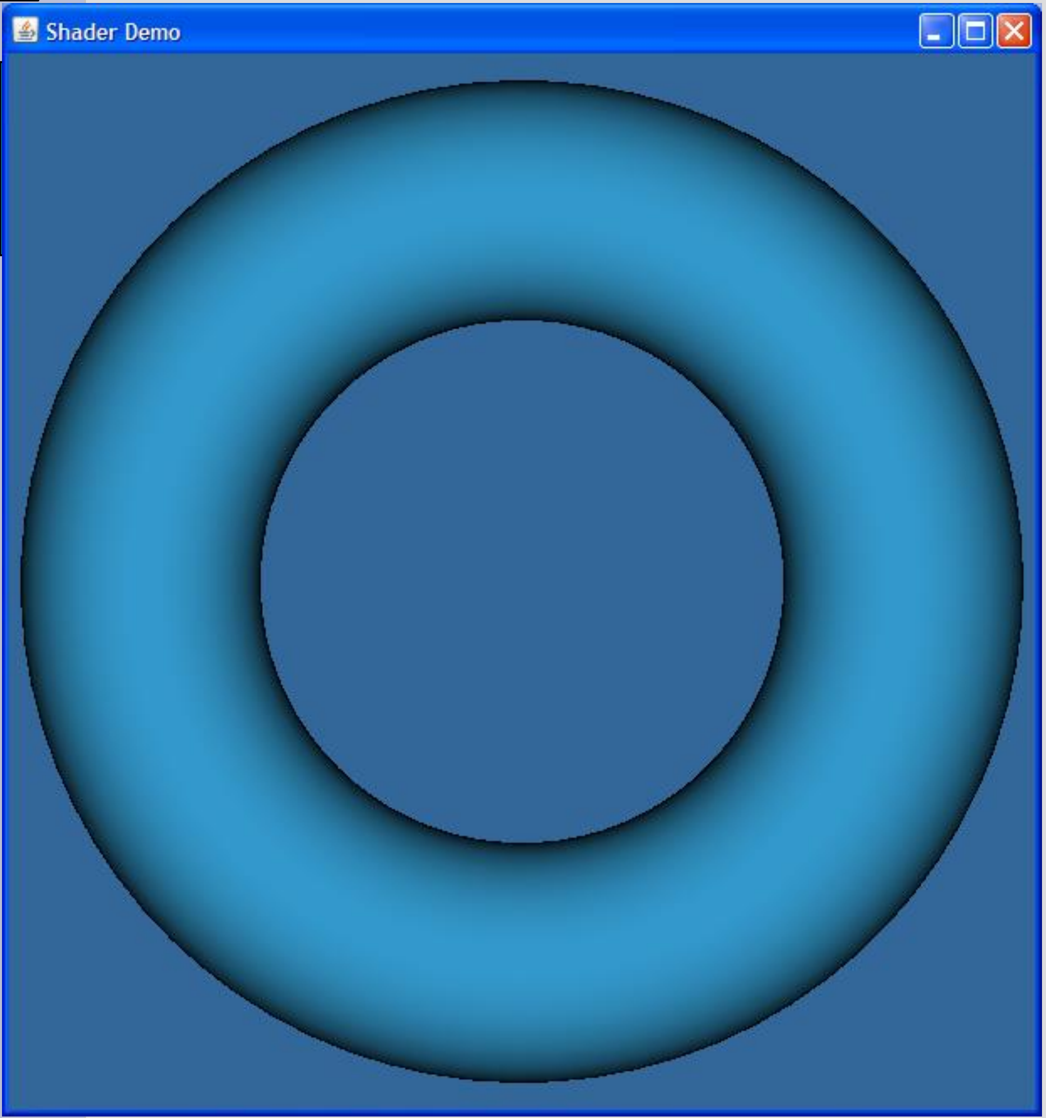
- ***Lighting***, or ***illumination***, is the process of computing the intensity and color of a sample point in a scene as seen by a viewer

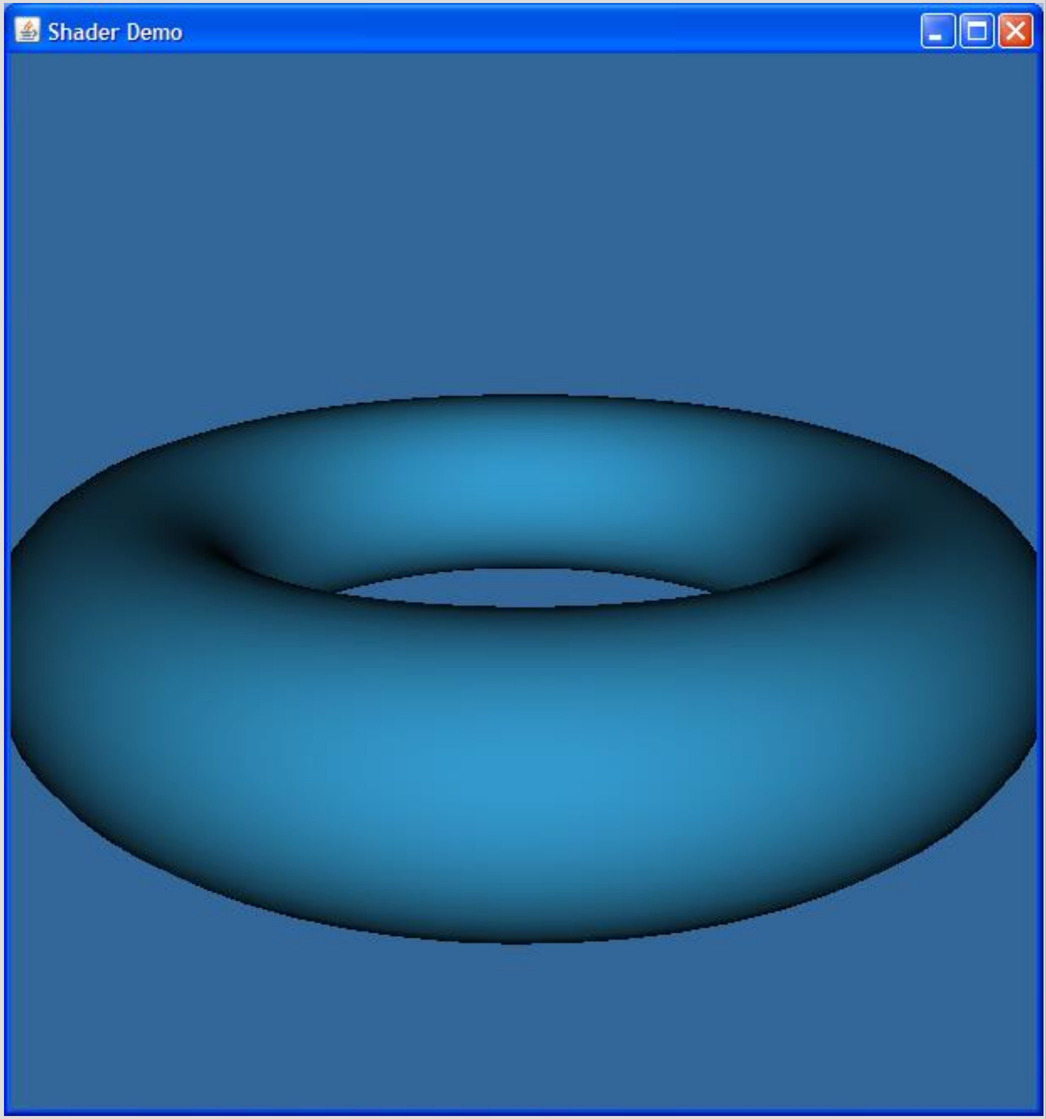
- ***lighting depends on the geometry of the scene***
- the models, the lights and the camera positions & orientations
- the surface orientations and the surface material properties

“diffuse lighting”: N·L

Shown: Light Source at Eye-point)

Project B: light-source overhead

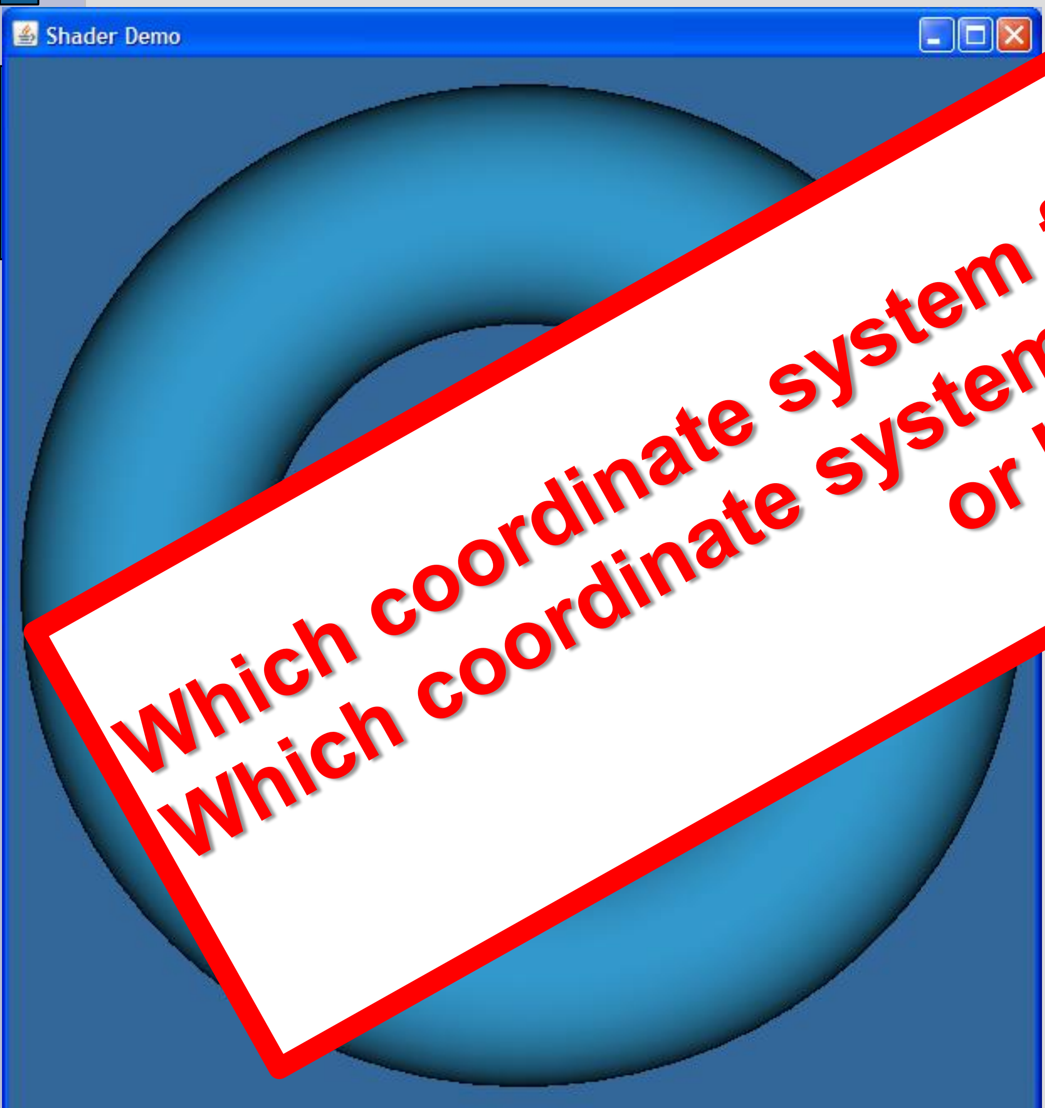


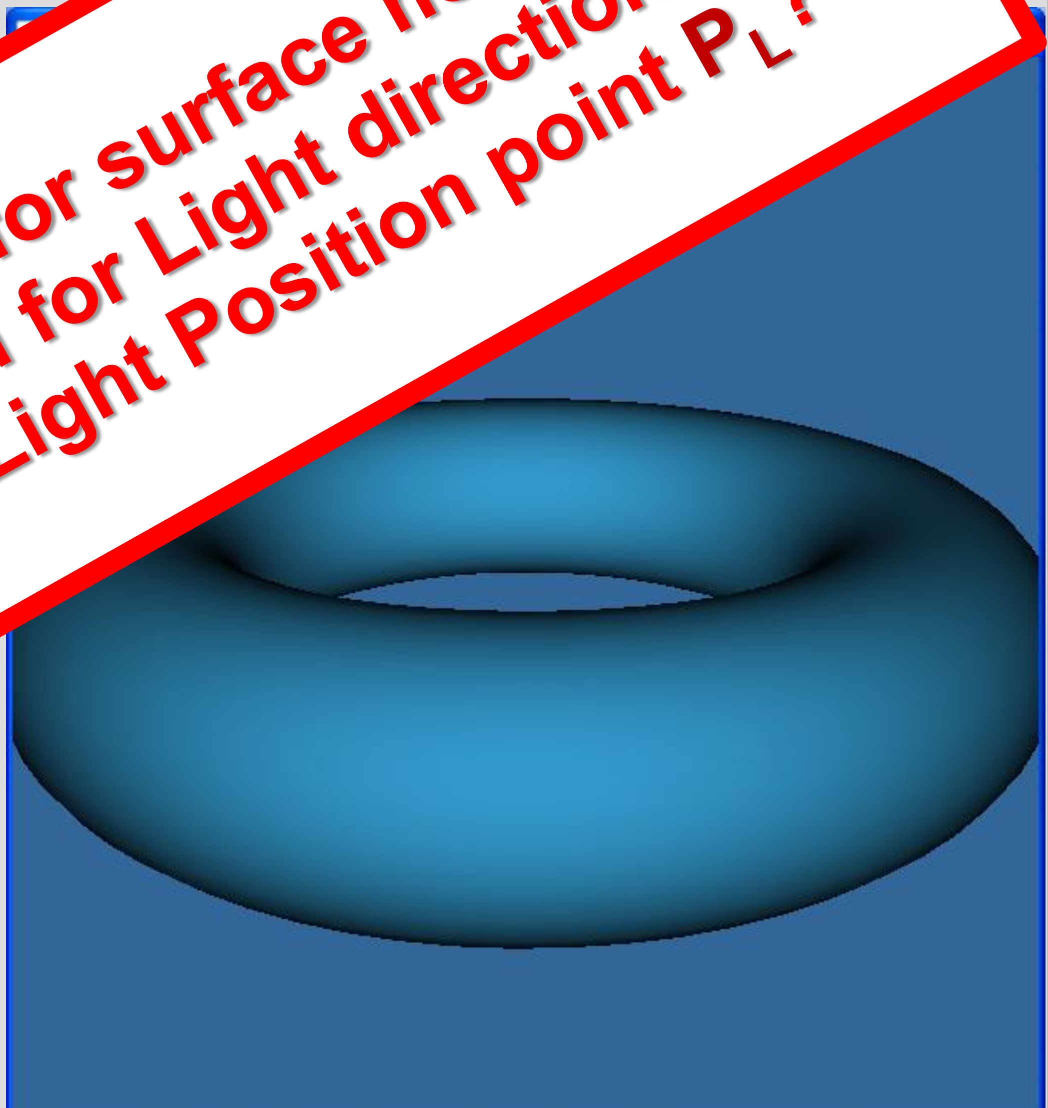


“diffuse lighting”: N.L

Shown: Light Source at Eye

Project B: light-source





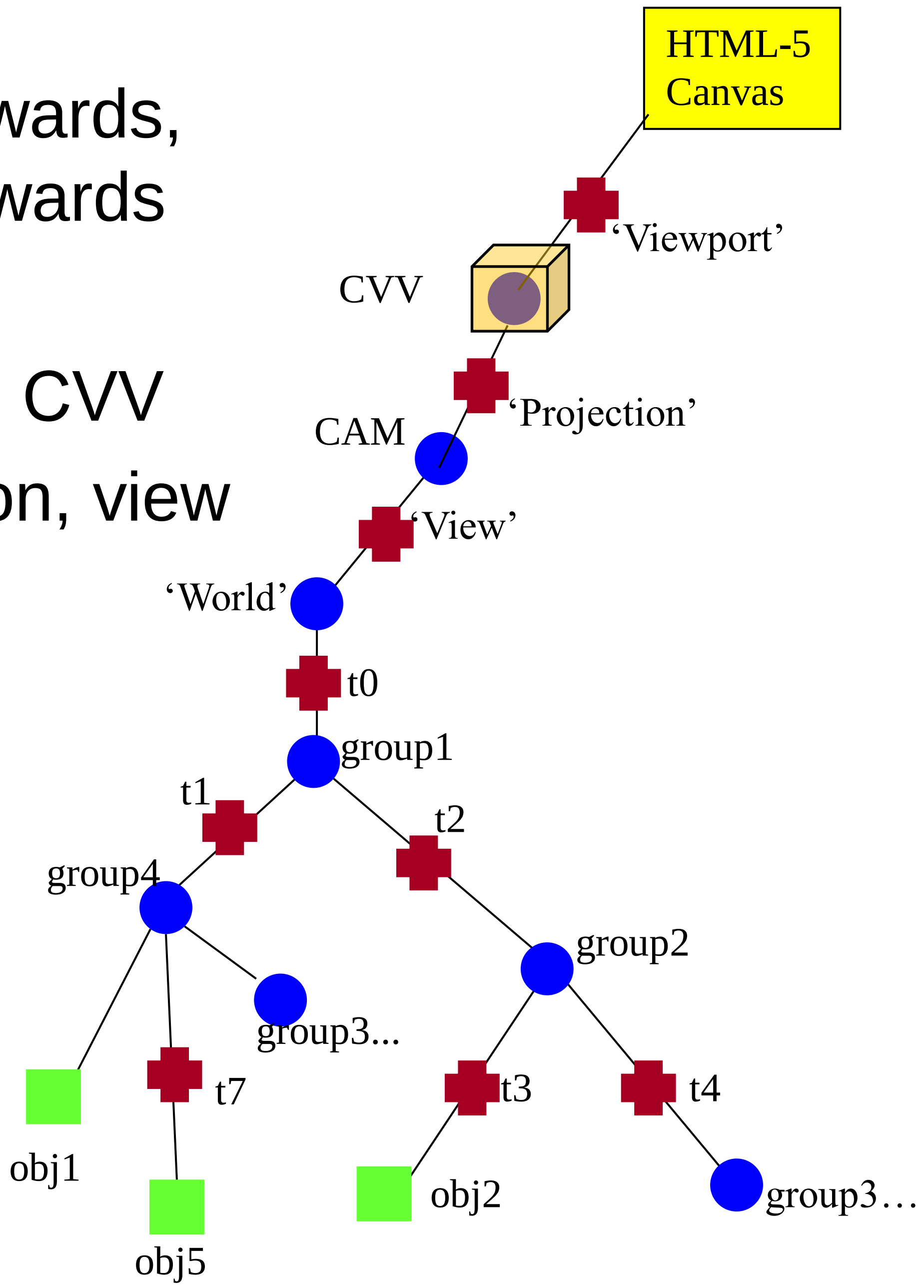
Which coordinate system for surface normal N?
Which coordinate system for Light direction L?
or Light Position point P_L?

WebGL: Vertex Position Pipeline

RECALL: in scene graphs,
Vertices & values move upwards,
transform calls move downwards

Project A: Draw 'world' directly in CVV

Project B: Add viewport, projection, view



WebGL: Vertex Position Pipeline

RECALL: in scene graphs,

Vertices & values move upwards,

transform calls move downwards

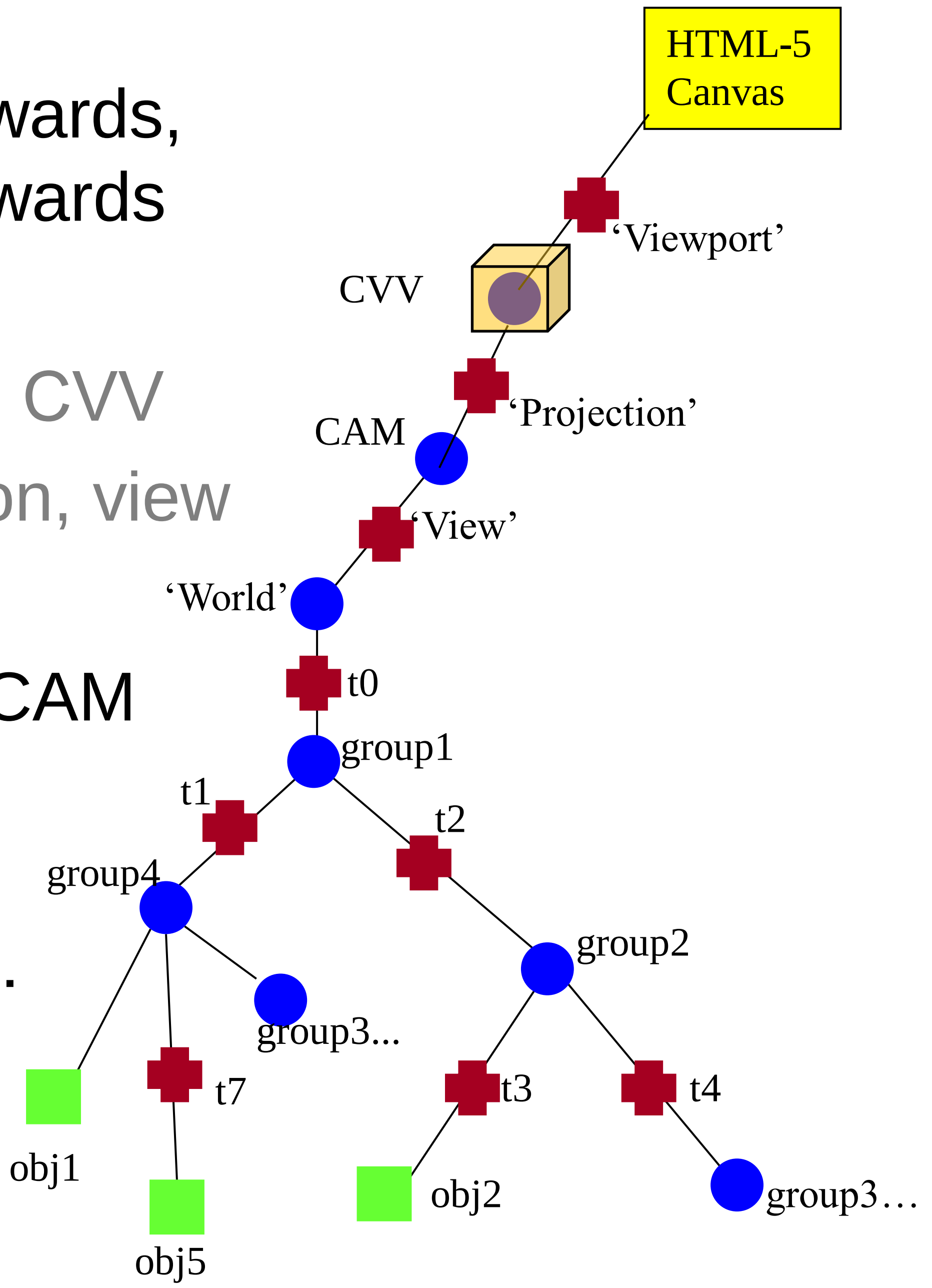
Project A: Draw 'world' directly in CVV

Project B: Add viewport, projection, view

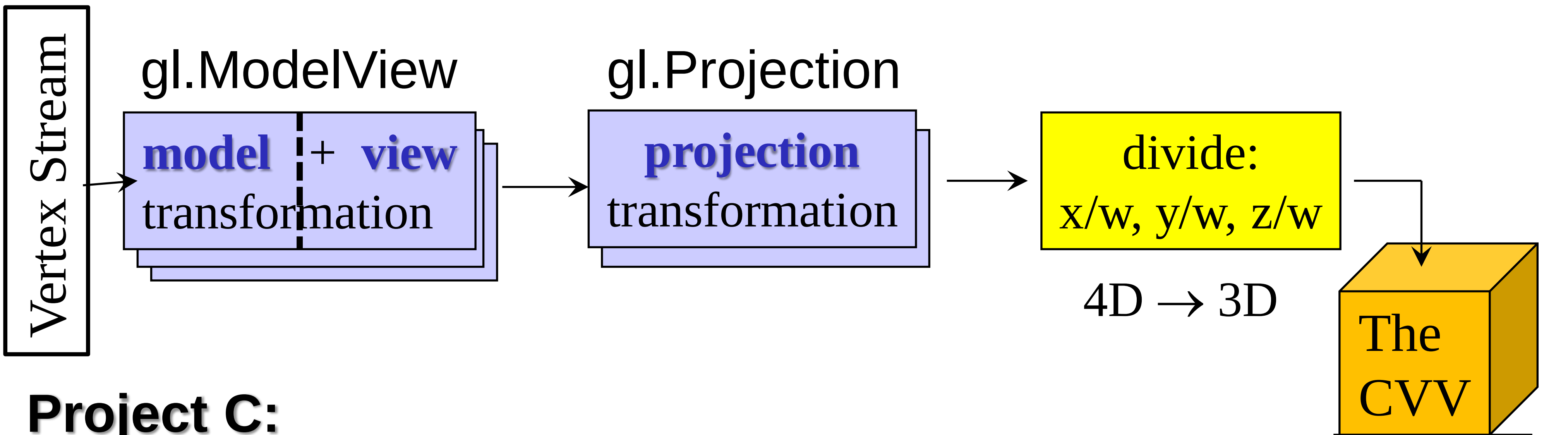
Project C:

- Lights: transform from World to CAM (How? apply 'View' matrix)
- Normals: from Object to CAM normal matrix == $modelView^{-T}$ …
- Compute lighting in

  CAM coordinate system

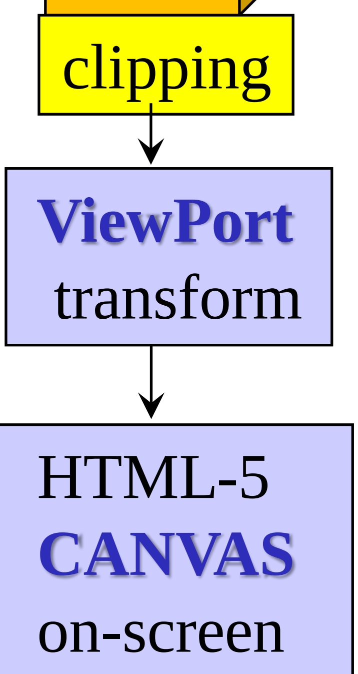


WebGL: Vertex Position Pipeline



Project C:

- Lights: transform from World to CAM
  (How? apply ‘View’ matrix in modelView)
- Normals: from Object to CAM
  normal matrix == modelView^{-T} …
- Compute lighting in CAM coordinate system
  (e.g. find N·L, find R vector, etc).



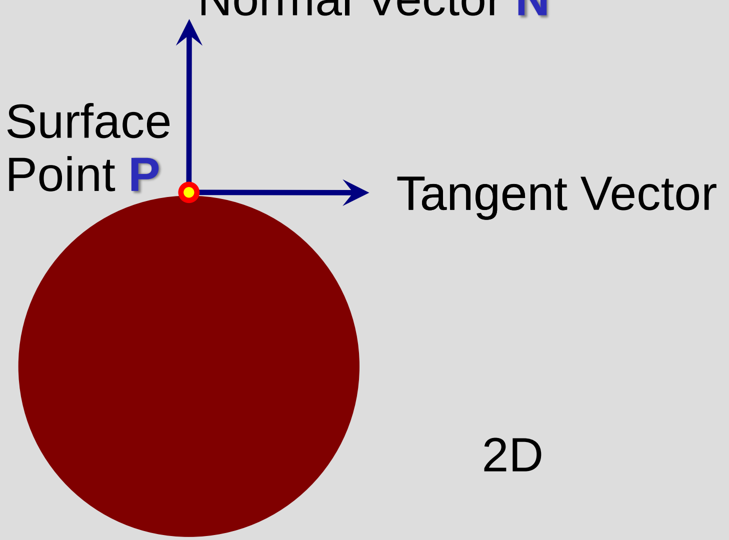
“Normal” == Surface Orientation

Perpendicular Vector at any Surface Point

More formally:

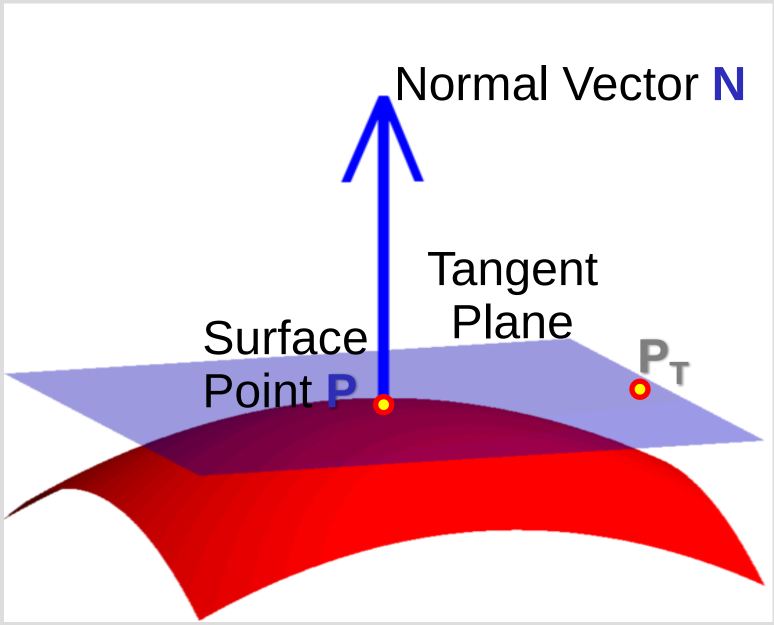
- Unit-length vector **N** at surface point P
- Vector **N** defines surface tangent plane at P:

  (For all points P_T in tangent plane, $(P_T - P) \cdot N = 0$)



2D

3D



Transforming Normals

We KNOW how to transform vertex positions.

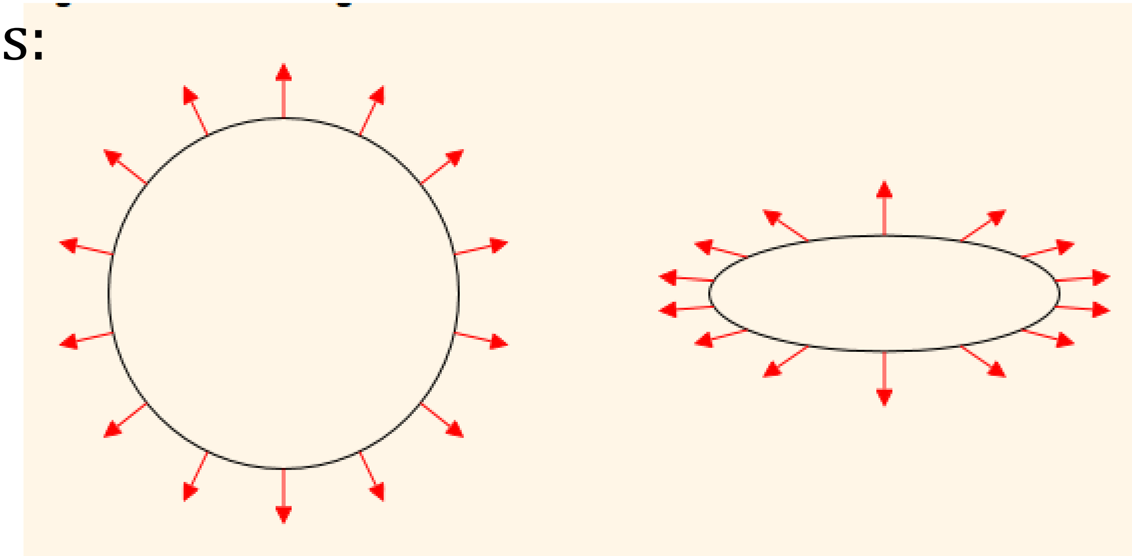
Can we transform Normal Vectors with the same matrix?

ALMOST always yes,

but <u>not always:</u> → **thus *the answer is*** $\boxed{\textbf{NO.}}$

we need a special **'normal transform' matrix**

because non-uniform scaling of shapes (stretched robot arm, etc) distorts these normals:



Transforming Normals

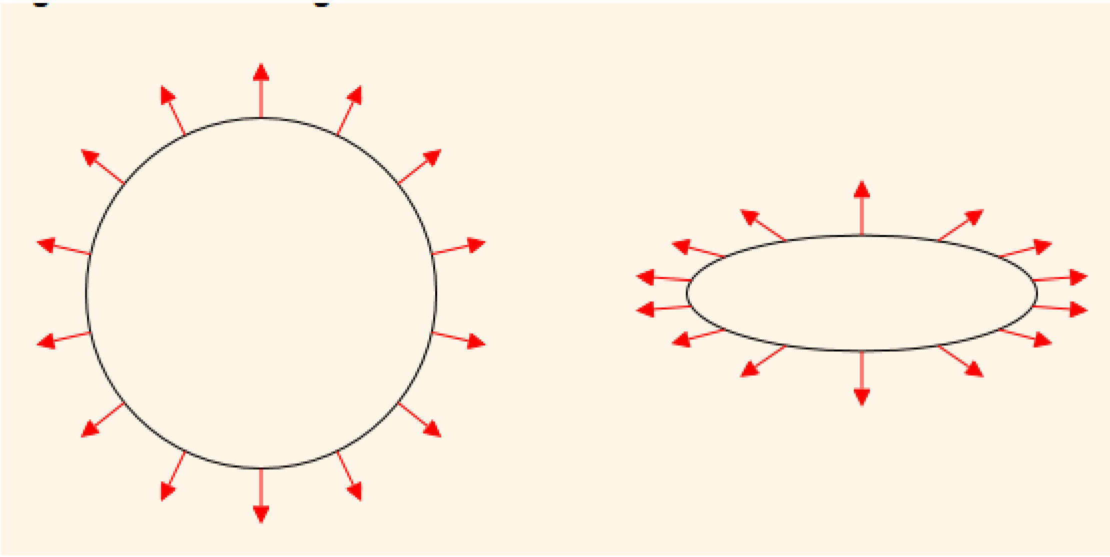
***SOLUTION*: use inverse-transpose:**

$$\boxed{\text{Normal Matrix} == (\text{Model Matrix})^{-T}}$$

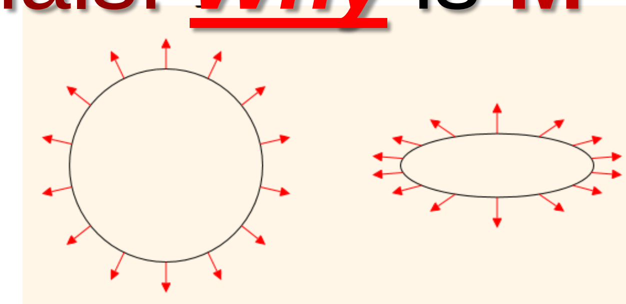
Why? see:

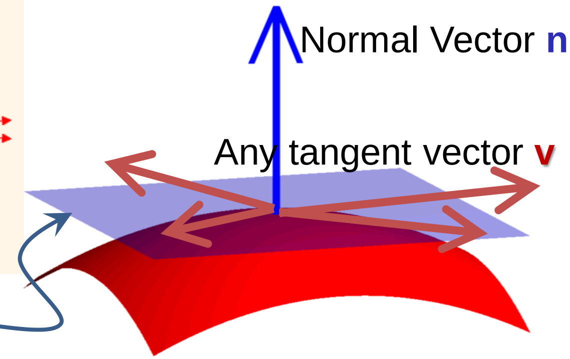
[***http://www.arcsynthesis.org/gltut/Illumination/Tut09%20Normal%20Transformation.html***](http://www.arcsynthesis.org/gltut/Illumination/Tut09%20Normal%20Transformation.html)

How? cuon-matrix-quat.js functions …



Transforming Normals: ***Why*** is $\mathbf{M}^{-T}$ correct?

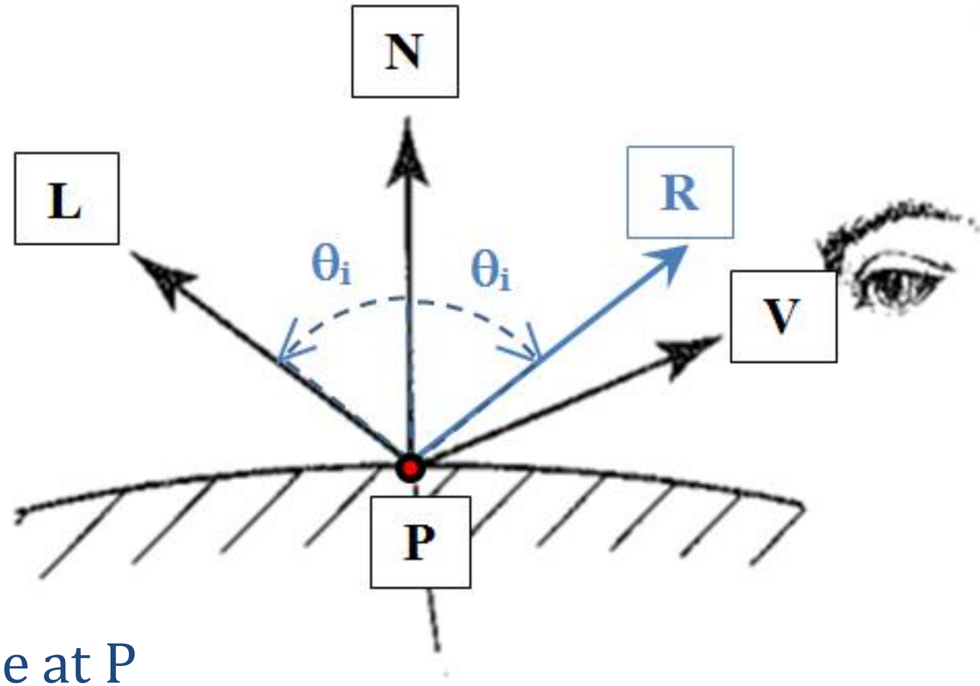




- **n**ormal vector == perpendicular (⊥)to surface tangent plane
- Any transform matrix **M** applied to the surface applies to the tangent plane too.
- Any vector **v** in the tangent plane is ⊥ to **n**, thus $\mathbf{n}\cdot\mathbf{v} = 0$, or equivalently: $\mathbf{n}^T\mathbf{v} = 0$
- REVIEW:
  - For any non-singular matrix **M** we can find an inverse $\mathbf{M}^{-1}$ that cancels it: $\mathbf{M}^{-1}\,\mathbf{M} = \mathbf{I}$
  - Transpose lets us multiply column vector **v** and matrix **A** in either order: $\mathbf{A}\mathbf{v} = \mathbf{v}^T\mathbf{A}^T$
- Expand $\mathbf{n}^T\mathbf{v} = 0$ with the 'do-nothing' identity matrix: $\mathbf{n}^T\mathbf{M}^{-1}\mathbf{M}\mathbf{v} = 0$
- Associate each matrix with its neighbor: $(\mathbf{n}^T\mathbf{M}^{-1})(\mathbf{M}\mathbf{v}) = 0$ and then look closely:
  - $(\mathbf{M}\mathbf{v})$ == Any and all transformed tangent-plane vectors
  - $(\mathbf{n}^T\mathbf{M}^{-1})$ == The transformed normal vector *guaranteed* ⊥ to all the tangent-plane vectors
- Rearrange $\boxed{\textbf{transformed normal vector}}$ using transpose: $(\mathbf{n}^T\mathbf{M}^{-1}) = (\mathbf{M}^{-1})^T\mathbf{n} = \boxed{\mathbf{M}^{-T}\mathbf{n}}$

Phong Lighting
Step 1: Find Scene Vectors

To find ***On-Screen RGB Color*** 
at point ***P (start):***



1) Find all 3D scene vectors first:

a) Light Vector **L**:
unit vector towards light source

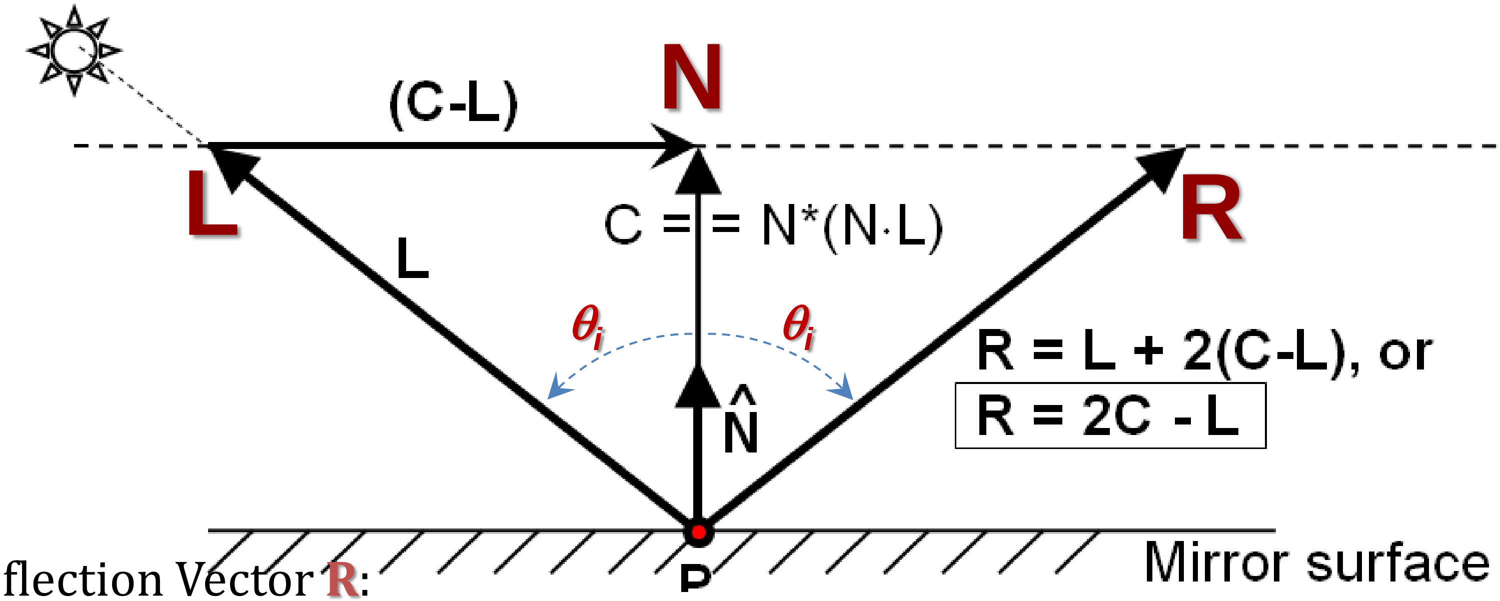
b) Normal Vector **N**:
unit vector perpendicular to surface at P

c) View Vector **V**:
unit vector towards camera eye-point

On to step 2: how do we find the Reflected-light Vector **R**?

Phong Lighting
Step 2: Find reflection Vector R

To find ***On-Screen RGB Color***
at point ***P (cont'd):***



2) COMPUTE the Light Reflection Vector **R**:

- Given unit light vector **L**, find lengthened normal **C**
  C = N (L·N)

- In diagram, if we add vector 2*(C-L) to L vector we get R vector. Simplify:
  R = 2C – L

- **Result:** unit-length **R** vector GLSL-ES→ See built-in '`reflect()`' function
  (If **N** is a unit-length vector, then **R** vector length matches **L** vector length)

Phong Lighting
Step 3: Gather Light & Material Data

To find ***On-Screen RGB Color***
at point ***P (cont'd):***

3) For each light source, gather:

- RGB triplet for **Ambient** Illumination **Ia** $0 \leq Iar, Iag, Iab \leq 1$
- RGB triplet for **Diffuse** Illumination **Id** $0 \leq Idr, Idg, Idb \leq 1$
- RGB triplet for **Specular** Illumination **Is** $0 \leq Isr, Isg, Isb \leq 1$

For each surface material, gather:

- RGB triplet for **Ambient** Reflectance **Ka** $0 \leq Kar, Kag, Kab \leq 1$
- RGB triplet for **Diffuse** Reflectance **Kd** $0 \leq Kdr, Kdg, Kdb \leq 1$
- RGB triplet for **Specular** Reflectance **Ks** $0 \leq Kar, Kag, Kab \leq 1$
- RGB triplet for **Emissive** term(often zero) **Ke** $0 \leq Ker, Keg, Keb \leq 1$
- Scalar 'shinyness' or 'specular exponent' term **Se** $1 \leq Se \leq \sim 100$

Phong Lighting
Step 4: Sum of Light Amounts

To find ***On-Screen RGB Color***
at point ***P (cont'd):***

sum of each kind of light at ***P***:
Phong Lighting = Ambient + Diffuse + Specular + Emissive
SUMMED for all light sources

4) For the i-th light source, find:

RGB= **Ke** + // 'emissive' material; it glows!
Ia*__Ka__ + // ambient light * ambient reflectance
Id*__Kd__*__Att__*max(0,(**N**·**L**)) // diffuse light * diffuse reflectance
Is*__Ks__*__Att__*(max(0,**R**·**V**))^{Se}, // specular light * specular reflectance

- Distance Attenuation scalar: $0 \leq$ **Att** ≤ 1
  - Fast, OK-looking default value: **Att =** 1.0
  - Physically correct value: **Att(d)** = 1/(**d**istance to light)^2 (too dark too fast!)
  - Faster, Nice-looking 'Hack': **Att(d)** = 1/(**d**istance to light)
  - OpenGL compromise: **Att(d) = min(1, 1/(c1 + c2*d + c3*d²))**
- 'Shinyness' or 'specular exponent' $1 \leq Se \leq \sim 100$ (large for sharp, small highlights)