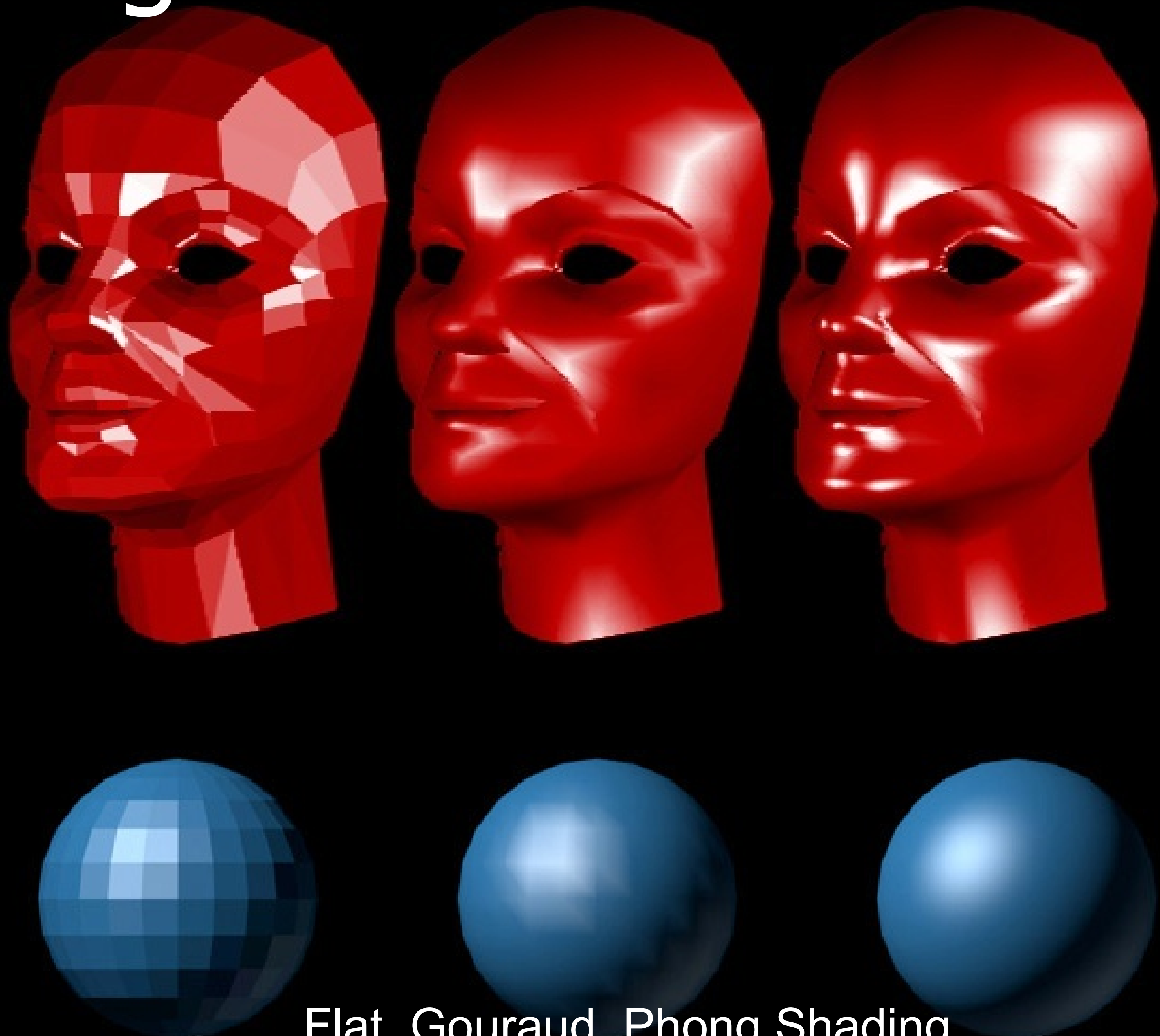
Shading and Lighting

- **Shading models**
  - Flat
  - Gouraud (smooth)
  - Phong
- **Lighting models**
  - Phong
  - Blinn-Phong
    - per-vertex implementation
    - per-pixel implementation

Shading



Flat, Gouraud, Phong Shading

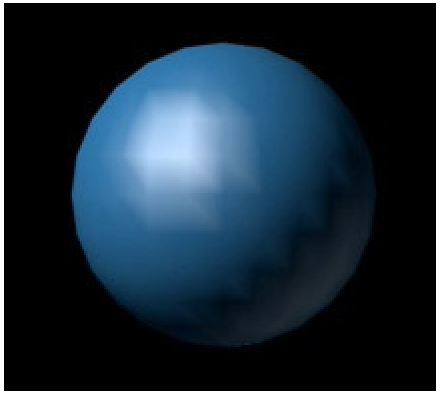
Shading

- **Flat shading**
  - Use triangle face normals for lighting
  - Solid color per triangle
  - Can't use shared normals

.

Shading

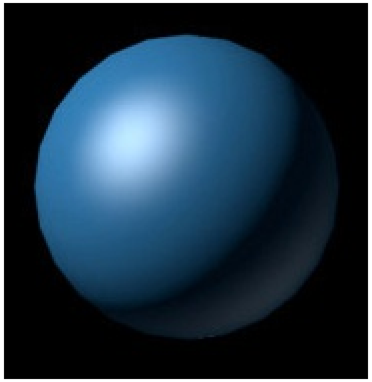
- **Gouraud**
  - Use vertex normal for lighting
  - Color computed for each vertex, interpolated over triangle
    - Compute color in vertex shader
    - Output as varying variable to fragment shader



Shading

- **Phong**
  - Interpolate parameters over triangle, compute color in fragment shader
  - Color computed per fragment
    - Output normal, light, view vectors from vertex shader as varyings
    - Compute lighting in fragment shader

  .

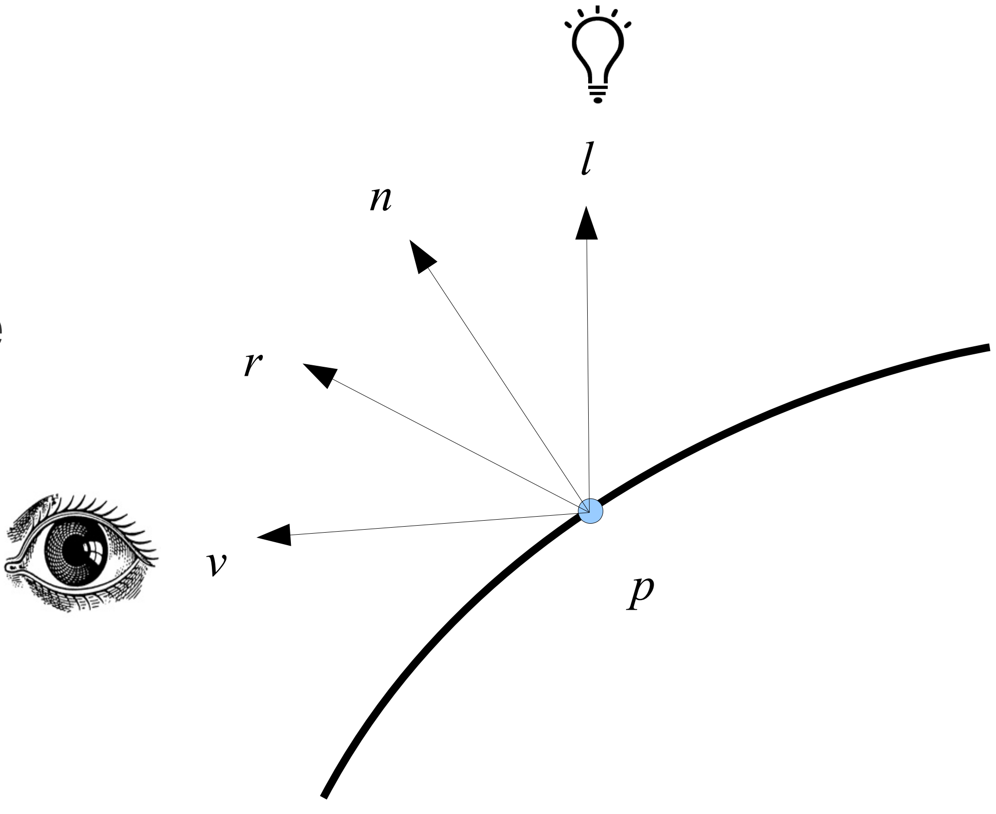


Bui Tuong Phong

- December 14, 1942 – July 1975
- Ph.D. University of Utah, 1973
- Joined Stanford faculty, 1975
- Developed Phong (specular) lighting model
- Developed Phong shading model

Local lighting model

- v, view vector
- n, normal vector
- l, light vector
- r, reflection vector
- All are unit vectors
- Pointing away from surface

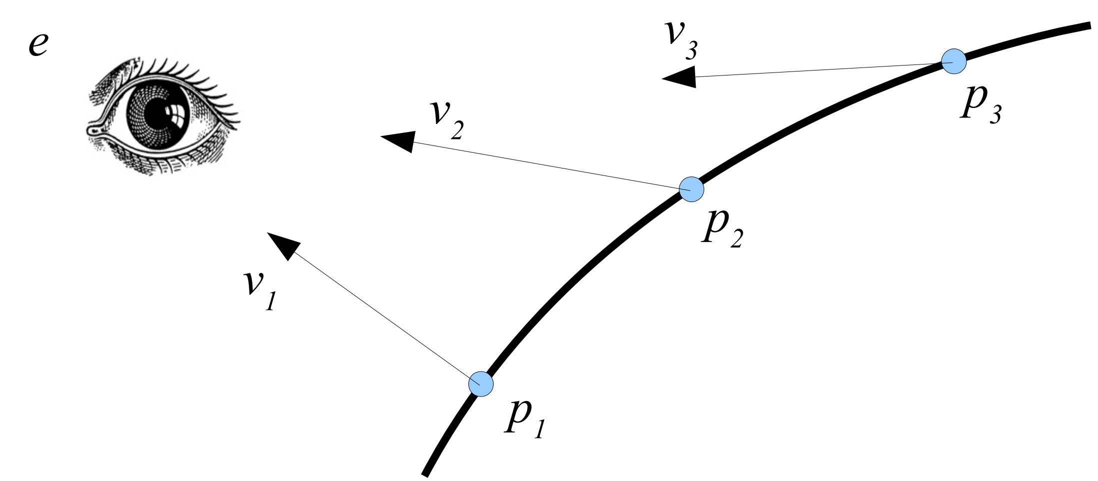


Viewer models

How to compute v?

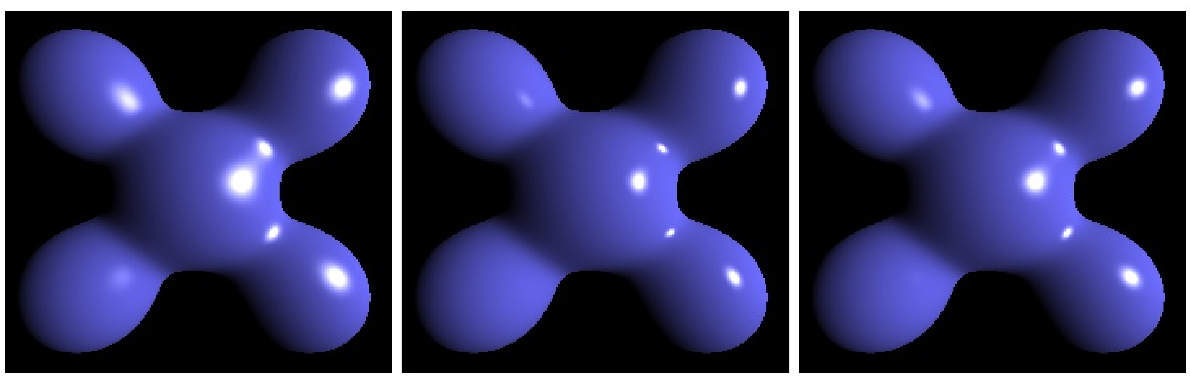
- **Infinite viewer**
  - $v = -look$
  - Faster since view vector is the same for all vertices

- **Local viewer**
  - e = eye position, p = point on surface
  - $v = (e-p)/||e-p||$



Blinn-Phong

- Jim Blinn modified the Phong lighting model so that the specular term is **estimated** quickly
- Less precise, but faster
  - Halfway vector: $h = \frac{1}{2}(l+v)$
  - *Replace* $(r \cdot v)^{\alpha}$ *with* $(h \cdot n)^{\alpha'}$



Blinn-Phong Phong Blinn-Phong with higher exponent

Coordinate system for lighting

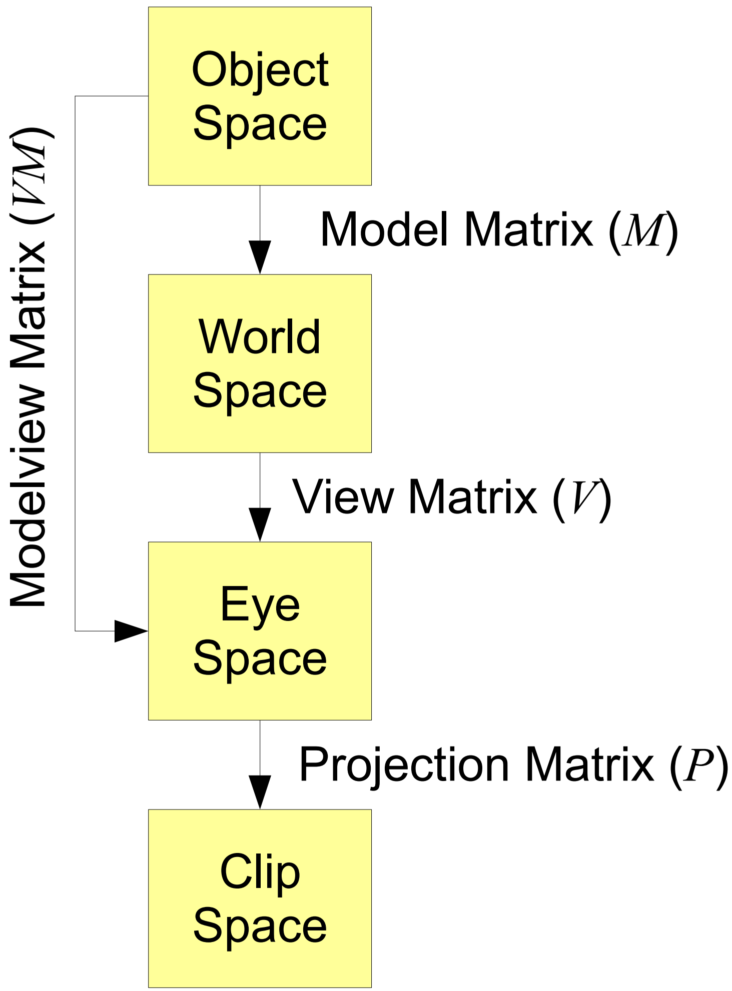
- We need all vectors to be in the <u>same coordinate system</u> in order to compute lighting
- Some choices:
  - Object local coordinate system
  - World coordinate system
  - Eye coordinate system
  - Tangent space (see CGT 521)

Coordinate systems

Recall :

- If matrix T transforms points from space A to space B then T^{-1} transforms points from B to A
- If T transforms points from A to B and S transforms points from B to C then ST transforms points from A to C
- If M transforms points then M^{-T} transforms normal vectors
  - $M^{-T} = (M^{-1})^T$

Coordinate systems



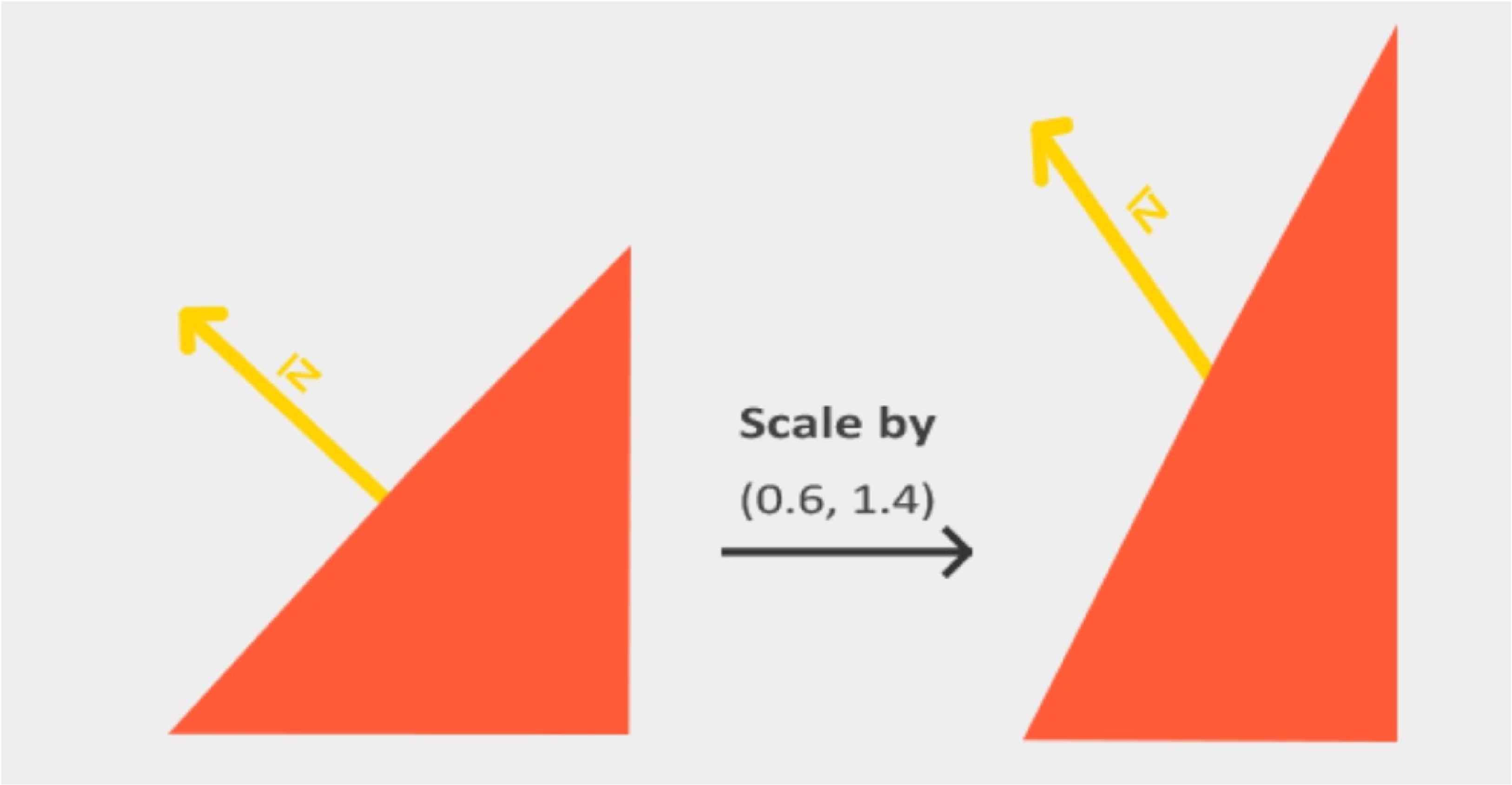
- What matrix transforms points from object space to eye space?
- VM : the product of view and model matrix

Normals

- Normal vectors also get transformed to move from one coordinate system to another
- Keep in mind:
  - M may include uniform scaling, so normalize n in the shader so that it is a unit vector
  - If M includes nonuniform scale then we have another problem...

Nonuniform scaling

- Look at what applying a nonuniform scaling transformation does to normal vectors...



- The opposite thing should have happened, the normal should have gotten shorter in the y-direction, not longer.

Transforming normals

- Consider the problem or transforming object-space normals into world-space

- Instead of multiplying with M, multiply with the **upper 3x3 part** of $(M^{-1})^T$
- This matrix, M inverse transposed, is sometimes called the *normal matrix*
- Sometimes written as M^{-T}
- Why does this work?

Normal matrix

- Let M = TRS
  - This idea works for any product, but let's demonstrate with this example
  - $M^{-T} = ((TRS)^{-1})^T$
  - $M^{-T} = (S^{-1}R^{-1}T^{-1})^T$
  - $M^{-T} = T^{-T}R^{-T}S^{-T}$
    - $T^{-T} = I$ since we are only considering the upper 3x3 part of M
    - $R^{-T} = R$ since rotation matrices are orthogonal ($R^{-1} = R^T$)
    - $S^{-T} = S^{-1}$ since $S^T = S$
    - So, $M^{-T} = RS^{-1}$

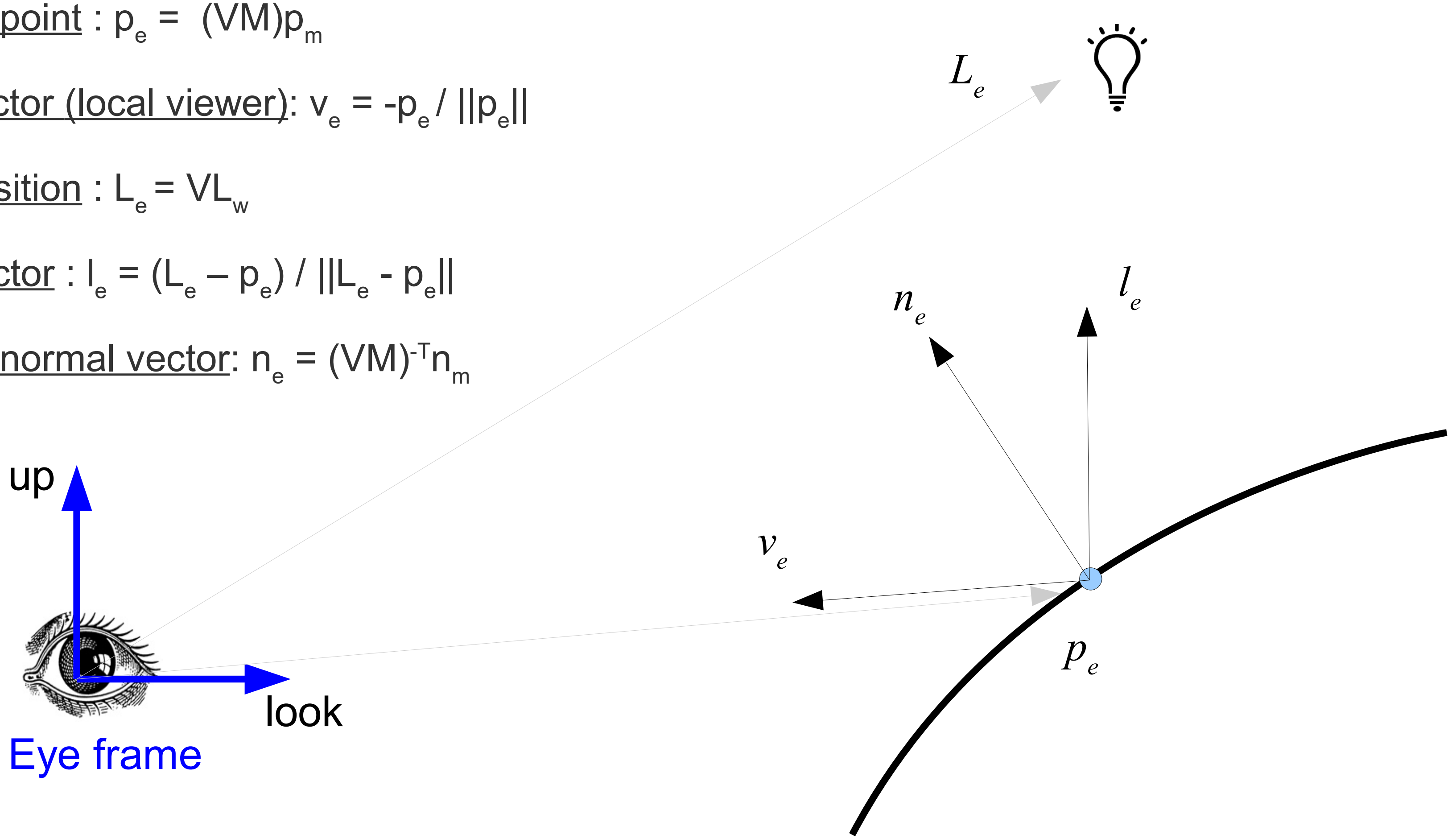
Normal matrix

- If $M = TRS$ then $M^{-T} = RS^{-1}$
    - Note that
        - S^{-1} applies geometric scaling to normals in the correct way
        - The rotation and scaling are in the correct order, so there is no issue with the noncommutativity of matrix multiplication
- In glm:
    - `mat3 N = glm::inverseTranspose(glm::mat3(M));`

Eye space lighting

Subscripts denote coord frame : m = model, w=world, e = eye

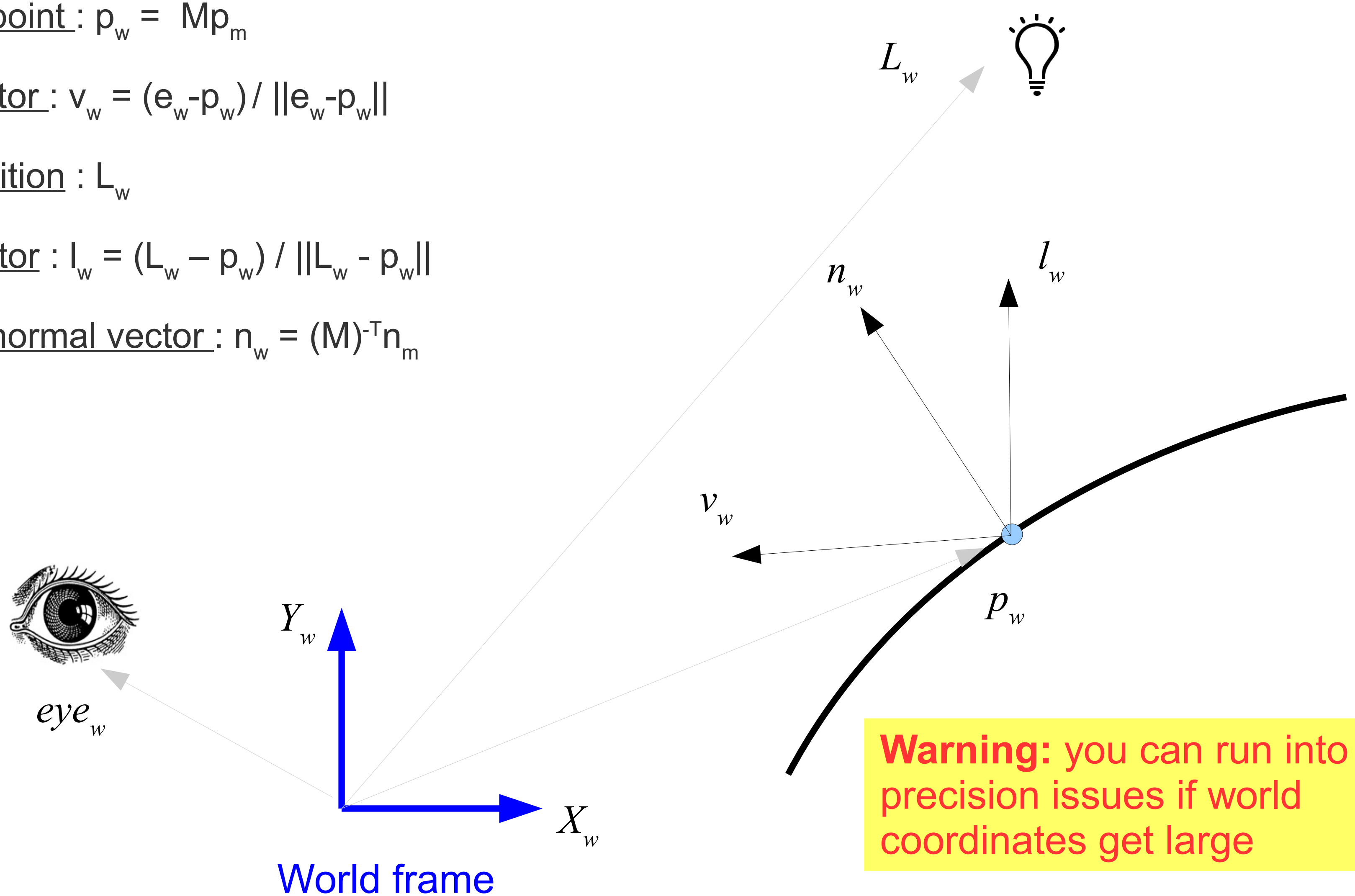
- Eye position : $[0,0,0,1]^T$
- Surface point : $p_e = (VM)p_m$
- View vector (local viewer): $v_e = -p_e / ||p_e||$
- Light position : $L_e = VL_w$
- Light vector : $l_e = (L_e - p_e) / ||L_e - p_e||$
- Surface normal vector: $n_e = (VM)^{-T}n_m$



World space lighting

Subscripts denote coord frame : m = model, w=world, e = eye

- Eye position : $e_w = V^{-1} [0,0,0,1]^T$
- Surface point : $p_w = Mp_m$
- View vector : $v_w = (e_w - p_w) / ||e_w - p_w||$
- Light position : L_w
- Light vector : $l_w = (L_w - p_w) / ||L_w - p_w||$
- Surface normal vector : $n_w = (M)^{-T} n_m$

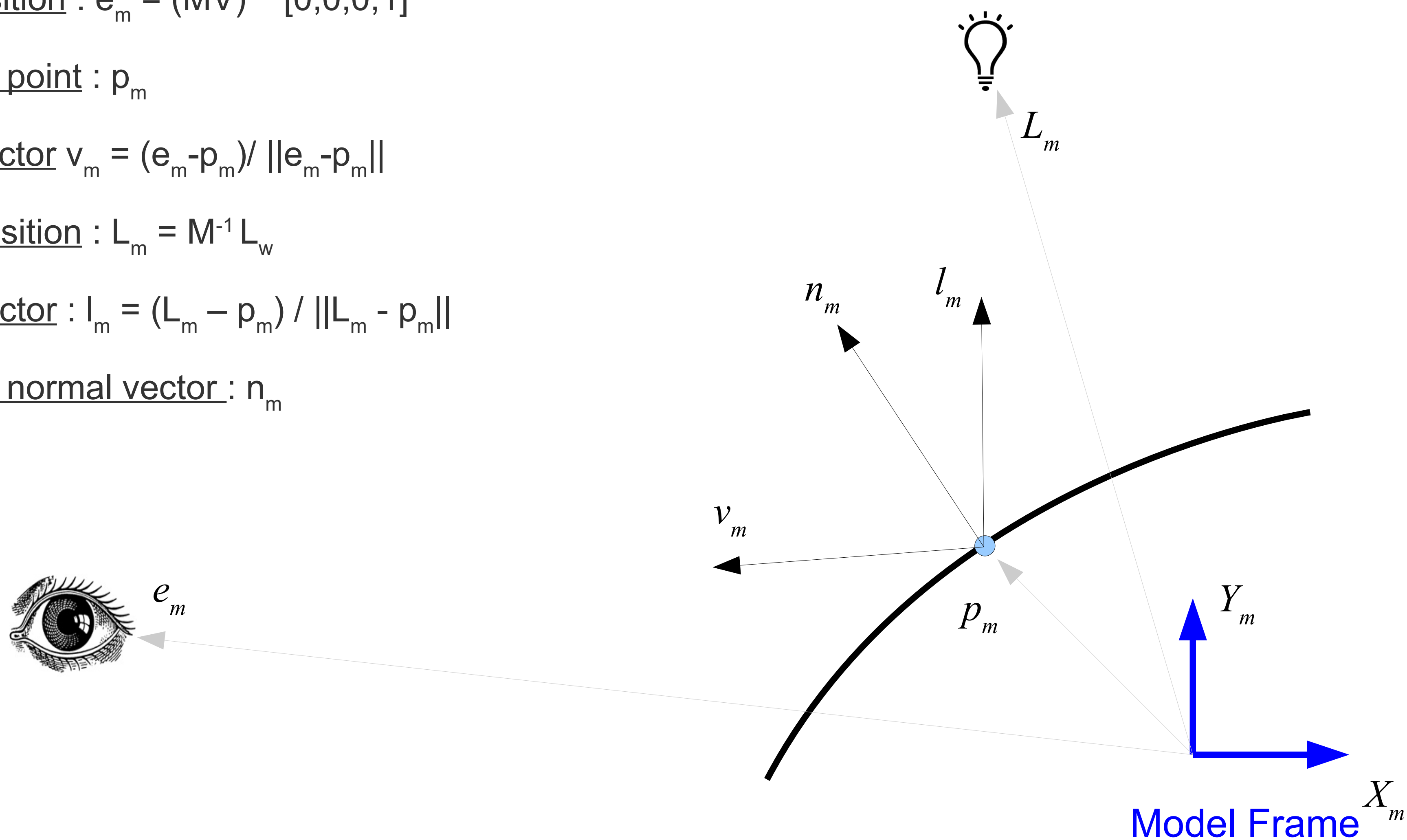


Warning: you can run into precision issues if world coordinates get large

Object space lighting

Subscripts denote coord frame : m = model, w=world, e = eye

- Eye position : $e_m = (MV)^{-1} [0,0,0,1]^T$
- Surface point : p_m
- View vector $v_m = (e_m - p_m) / ||e_m - p_m||$
- Light position : $L_m = M^{-1} L_w$
- Light vector : $l_m = (L_m - p_m) / ||L_m - p_m||$
- Surface normal vector : n_m



Clip space lighting

- **DON'T DO IT**
- Projection matrix, P, is not affine, will distort angles between objects

Which space to pick?

- Which space results in the fewest matrix-vector multiplications?
- The space your lights are in can help you decide
  - Lights can be positioned in the world
    - streetlights

  - They can be attached to objects
    - headlights
  - They can be attached to the camera
    - flashlight