Mathematics of Per-Pixel Lighting

Cass Everitt
NVIDIA Corporation
cass@nvidia.com

Overview
• Why Per-Pixel Lighting?
• Review
  - OpenGL Transforms and Spaces
  - OpenGL Per-vertex Lighting
  - Object Space Per-vertex Lighting
• Surface-local Space?
  - Other names
  - Why is this necessary?
  - Surface-local Space Per-Vertex Lighting
• Per-Pixel Lighting
  - In Surface-local Space
  - In other spaces?

Why Per-Pixel Lighting?
• Because it looks better than per-vertex lighting
• Because it’s hardware accelerated
• Because everyone else is doing it
  - Don’t be the last on your block

OpenGL Transformations
- OpenGL operation transforms
  - object space
  - MODELVIEW matrix
  - eye space
  - PROJECTION matrix
  - clip space
  - Perspective Divide
  - normalized device coordinates
  - viewport/depthrange scale & bias
  - window space

This is do-it-yourself lighting
• You get total control, but this means you have to do it all
  - No glShadeModel(GL_PHONG)
  - No glEnable(GL_BUMP_MAPPING)
• If you don’t know how to implement per-vertex lighting, learn how to do that first
• Per-pixel shading is an extension of per-vertex shading (for the most part)

Example Scene -- world space

Note: world space is not an explicit space in OpenGL
Example Scene -- *eye space*

Object Space

Each object has its own origin, orientation, and scale

OpenGL Per-Vertex Lighting

- For OpenGL Per-Vertex Lighting, all calculations happen in *eye space*
- Not essential, but convenient
- For each OpenGL per-vertex light, the illumination is computed as (assuming separate specular)

$$C_{pel} = (\text{spot} \times \text{att}) \cdot (n \cdot l) + (n \cdot l) \cdot (n \cdot d)$$

$$C_{née} = (\text{spot} \times \text{att}) \cdot (n \times h)^* - s_{n\alpha}s_{d\lambda}$$

Lighting in *eye space*

The vectors...

Transforming Normals

- To evaluate the lighting equation in *eye space*, normals must be transformed from *object space* into *eye space*
- Normals are not simply transformed by the modelview matrix like position
- You may know from the Red Book or various other sources that “normals are transformed by the inverse-transpose of the modelview matrix”, but let’s consider why...
- The following slides should help provide some intuition about the transforming of normals
Transforming Normals (2)

• **Translation** of position does not affect **normals**

<br>

Transforming Normals (3)

• **Rotation** is applied to **normals** just like it is to position

<br>

Transforming Normals (4)

• **Uniform scaling** of position does not affect the direction of **normals**

<br>

Transforming Normals (5)

• **Non-uniform scaling** of position does affect the direction of **normals**!

  • **Opposite** of the way position is affected – or the **inverse** of the scaling matrix that’s applied to position

<br>

Transforming Normals (6)

• To summarize, these are the basic position transformations and the corresponding normal transformation:

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Position</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>T</td>
<td>I</td>
</tr>
<tr>
<td>Rotation</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>Scaling</td>
<td>S</td>
<td>$S^{-1}$</td>
</tr>
</tbody>
</table>

• Note that any sort of scaling applies inversely to the normal – we treat all scales (uniform and non-uniform) the same
• This is why we need GL_NORMALIZE and GL_RESCALE_NORMAL for OpenGL lighting
• We have to deal with it in per-pixel lighting as well

<br>

Transforming Normals (7)

• How does this match what OpenGL does?

  \[ n_v = M^{-1} n_i \]

  • For simplicity, consider $M$, the modelview matrix, is composed of a scale and a rotation
    • **Inverse-transpose is distributive**
    • For rotation (orthonormal) matrices $R^T = R^{-1}$, and $R^{-1} = R$
    • For scaling (diagonal) matrices $S^{-1} S = I$

  \[
  M^{-1} = (RS)^{-T} = R^{-T} S^{-T} = RS^{-1}
  \]

  This matches our ad hoc result!
Object Space Per-Vertex Lighting

- Nothing in the lighting equation requires evaluation in eye space - consider lighting in object space instead
  - Non-uniform scaling in the modeling matrix would complicate things, so we will ignore that for now...
- If the modeling matrix is simply a rigid body transform, then this is easy...
  - Need to transform the light into object space from eye space
    \[ L_{\text{obj}} = M^T L_{\text{eye}} \]
    - local light source
    \[ L_{\text{obj}} = M^T L_{\text{inf}} \]
    - infinite light source
  - No need to transform each normal now (cheaper)

Example Scene -- object space for

The vectors...

Lighting in object space

Note that the dot products are the same for all vectors in the same space.

Surface-local Space

- This gets called a lot of things...
  - surface-local space
  - tangent space
  - texture space
- A surface-local space is a class of spaces defined for every point on a surface
  - Tangent space and texture space are surface-local spaces that give specific definitions to the basis vectors
- Consider one additional transform from surface-local space to object space

Surface-local Space (2)

- The classes of surface-local space we use are defined for every point on a surface such that the point is at the origin, and the geometric surface normal is along the positive z axis
  - Note that for per-pixel lighting the geometric surface normal is generally not what we use in the lighting equation
- The x and y axes are orthogonal and in the tangent plane of the surface
- Now the entire scene can be defined relative to any point on any surface in the scene – not just relative to any object
Lighting in surface-local space

The vectors...

Surface-local matrix

- If we specified vertices in surface-local space, they'd all be the same!
  - glNormal3f(0,0,1); glVertex3f(0,0,0);
- The surface-local matrix, $S_l$, would provide the object space position and the object space normal orientation, and it would vary per-vertex:

$$
\begin{bmatrix}
T_x & B_x & N_x & P_x \\
T_y & B_y & N_y & P_y \\
T_z & B_z & N_z & P_z \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

- $T$, $B$, $N$, and $P$ are vectors. $T$ = tangent vector, $B$ = binormal vector, $N$ = object space vertex normal, $P$ = object space vertex position

- More on the tangent and binormal ($\Gamma$ and $\beta$) vectors later...

Per-Vertex Lighting in surface-local space

- As with lighting in eye space or object space, surface-local space is a perfectly valid coordinate frame to evaluate the lighting equation
- We simply transform the light and eye into surface-local space – the normal is known by definition, so it doesn't need to be transformed
- Compare eye space and surface-local space lighting:
  - Eye space lighting: the light vector or eye vector are "free", but you must transform each normal into eye space
  - Surface-local space lighting: the normal is free, but you must transform the light and eye vectors into surface-local space

Per-Pixel Lighting

- Getting back to the original point...
- We really want to evaluate the lighting equation per-pixel
- Rather than passing in normals per-vertex, we’ll fetch them from a texture map
  - We simulate surface features with illumination only

Merely fetching the normals from the texture map (per-pixel normals) is not sufficient because it doesn't account for the surface orientation at each pixel. This is where the tangent and binormal vectors come into play. They provide a way to align the lighting calculation with the local surface orientation at each pixel, which is crucial for achieving realistic lighting effects.

Per-Pixel Lighting (2)

- The texture map containing normals (normal map) clearly uses normals that are not aligned with the +z axis in surface-local space
  - This makes the tangent and binormal vectors important (see discussion later)
- GPUs certainly have enough horsepower to evaluate the illumination equation at each pixel – but it is more expensive in eye space!
  - That would require transforming each normal into eye space (after fetching it from the texture map)
Per-Pixel Lighting (2)

- The better solution is to light in surface-local space
  - Fetched normals are already in the correct space
  - Light and eye vector interpolate nicely as long as the tangent and binormal are “well behaved”
- All remaining arithmetic can be evaluated with register combiners (not important these days)
- Minor limitation: as with object space per-vertex lighting, you can’t have a non-uniform scale without requiring a per-normal transform and renormalize
  - don’t do lots of non-uniform scaling – it won’t behave correctly

Tangent and Binormal

- Whether we implement per-pixel lighting in surface-local space or eye space, the tangent and binormal vectors need to be well-behaved from vertex to vertex
- Specifically, \( \| \text{perp}(T_1, T_2) \| = 1 \) and \( \| \text{perp}(B_1, B_2) \| = 1 \)

Tangent and Binormal (2)

- Another way to look at the problem case:
  - The vectors we interpolate over the polygon are: 
  - very denormalized

Tangent and Binormal (3)

- In the previous case, we considered transforming the light into the surface-local space of each vertex and interpolating it for the per-pixel light vector – this is what we would do for GeForce2 (old GPUs)
- With modern GPUs, we can interpolate the 3x3 matrix over the surface and transform the normals by it – for this case if the tangent and binormal are not well-behaved, other anomalous behavior will result
  - Normal “twisting”
  - Incorrect bump scale/smoothing
  - The interpolated matrix should be “nearly orthonormal”

Questions?

Cass Everitt
cass@nvidia.com
www.nvidia.com/Developer