gaze vector $\vec{g} = \vec{l} - \vec{e} = (-1,-1,-1)$

$\vec{g} \times \vec{t} = (1, 0, -1)$

Attributes : Vertex position, color, normal
tex. coordinate
Uniform : (mat4) Model View Proj transform
Varying : Per-fragment color/normal/tex coordinate
In local illumination models (e.g. the Phong model we discussed in

→ Eye location $\underline{e}$

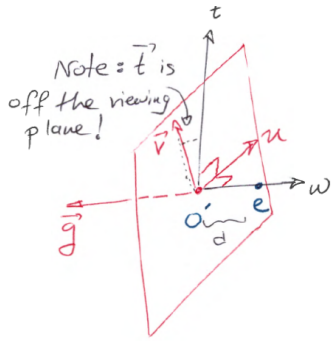
Invert & normalize $\vec{g}$ to get $\vec{w} = -\frac{1}{\|\vec{g}\|}\vec{g}$

Construct $\vec{u}$ as mutually orthogonal to $\vec{t}$ & $\vec{w}$

$\vec{u} = \frac{\vec{t} \times \vec{w}}{\|\vec{t} \times \vec{w}\|}$

Build $\vec{v}$ as mutually orthogonal to $\vec{w}$, $\vec{u}$

$\vec{v} = \vec{w} \times \vec{u}$ (No need to normalize



(a) The \gaze" vector g points from e to l, and is oriented exactly in

the opposite direction of the coordinate axis vector w.

The **Vertex Shader** is the programmable Shader stage in the rendering pipeline that handles the processing of individual vertices. Vertex shaders are fed Vertex Attribute data, as specified from a vertex array object by a drawing command.

Compute di_use shading.
Can be done in either shader, the result will be higher quality if done in the fragment shader. However, if the 3 per-vertex normals of every triangle are exactly equal, the result would be
the same regardless of whether the di_use color was computed in
vertex or fragment shader (di_use color depends on normal and
light direction).

Note that each instance of the vertex program will be given the information pertaining to a specific vertex; it will not have the information to combine several vertices
Only the vertex shader can actually modify the positions of vertices being drawn
A **Fragment Shader** is the Shader stage that will process a Fragment generated by the Rasterization into a set of colors and a single depth value.

It is possible, and generally preferable to do texture lookups in the fragment shader, since doing that in the vertex shader would only interpolate the textured color from the vertices of each triangle

main purpose of textures, which is to create appearance variation at a granularity finer than that of individual triangles
If the lookup is in vertex shader, we will need to reference it somewhere else, and also because of the limits on graphic cards, the performance will be slow.

The fragment shader can access uniform variables, which are set by the host program, and also receive information indirectly via the vertex shader
Z-buffer: The painter's algorithm requires performing a *topological sort* on the polygons, which takes quadratic time;

Z-buffering does not handle transparency well;

Z-buffering can suffer from precision issues.

Varying variables are a mechanism for passing information from the vertex shader to the fragment shader, by interpolating vertex data to interior locations of triangles being rasterize

The most common case is when the normal vector (passed as a vertex attribute) needs to be transformed within the shaders. The matrix that transforms the normals could be computed from the Model-View matrix (although it would have been much more e_cient to just pass it in as a separate matrix), but not from the aggregate MVP matrix that includes the projection as well. Also, in certain instances we may want to pass the ModelView (MV) matrix separately to the shaders for transforming point locations to camera coordinates, e.g. for computing specular lighting.

Computing barycentric coordinates is a very regular operation (does not need branching/conditionals) and is very parallel (every fragment can be done independently). The performance gains due to these properties on modern, massively parallel graphics hardware outweigh any "wasted" computation

The design of the graphics pipeline is meant to facilitate both **stream parallelism** (applying the same computation on many instances of data), as well as **pipeline parallelism** (concurrently using different stages of the hardware to execute separate parts of the computation undertaken by individual triangles

Yes. Specular reflection could produce different results if done on a per-fragment basis.

Cubic curves provide the minimum-curvature interpolants to a set of points. That is, if you have a set of $n + 3$ points and define the "smoothest" curve that passes through them (that is the curve that has the minimum curvature over its length), this curve can be represented as a piecewise cubic with n segments.

Cubic polynomials have a nice symmetry where position and derivative can be specified at the beginning and end.

Cubic polynomials have a nice tradeoff between the numerical issues in computation and the smoothness.

Hermite cubics are convenient because they provide local control over the shape, and provide $C1$ continuity

parametric curves are much easier to draw, because we can sample the free parameter

The advantage of the canonical form (Equation (15.4)) is that it works for these more complicated curves, just by letting n be a larger number

One advantage to using a piecewise representation is that it allows us to make a tradeoff between

1. how well our represented curve approximates the real shape we are trying to represent;
2. how complicated the pieces that we use are;
3. how many pieces we use.

A disadvantage of natural cubic splines is that they are not local. Any change in any segment may require the entire curve to change

Some of the disadvantages of the interpolating polynomial are:

The interpolating polynomial tends to overshoot the points

Control of the interpolating polynomial is not local

Evaluation of the interpolating polynomial is not local

A cardinal spline has the disadvantage that it does not interpolate the first or last point, which can be easily fixed by adding an extra point

mipmap:

The result could be quite different in the boundaries between two neighboring triangles. Blurring the entire image would have the effect of smearing those triangle boundaries, while mip-mapping would generally preserve such sharp transitions.

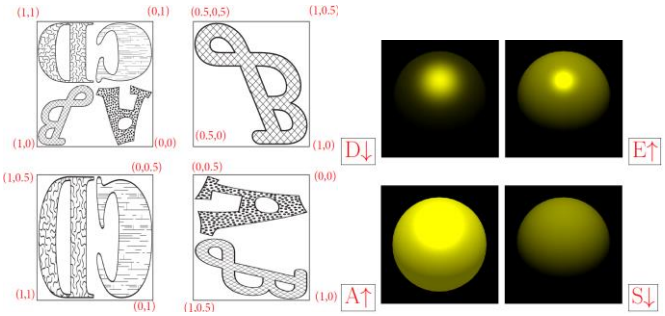
Blurring the entire image would unnecessarily smooth out detail even in parts of the rendered image where we do not have risks of aliasing

When, during rasterization and texturing, the size of a *fragment* (display pixel) is significantly **larger** than the size of a feature of the texture image, as projected on the display.

When aliasing artifacts appear on textured surfaces, typically on surfaces that are farther away from the camera, and/or oriented at an angle relative to the viewing direction.

Texture:

The convention that texture coordinates



range between zero and one is typically just related to how we do lookups when image files are used for texturing. A procedural texture does not need to abide by this convention, and could have texture coordinates assume any range that the designer of the texture might have chosen.