Projections and Hardware Rendering

Brian Curless
CSE 557
Fall 2014

Reading

Required:

- Shirley, Ch. 7, Sec. 8.2, Ch. 18

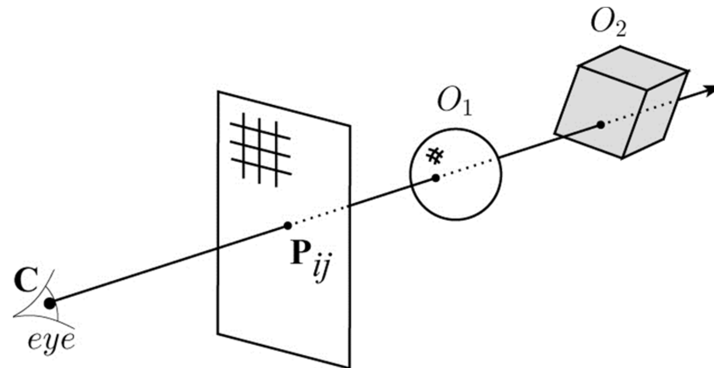
Further reading:

- Foley, et al, Chapter 5.6 and Chapter 6
- David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, 2nd Ed., McGraw-Hill, New York, 1990, Chapter 2.
- I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1-55, March 1974.

Going back to the pinhole camera…

Recall that the Trace project uses, by default, the pinhole camera model.

If we just consider finding out which surface point is visible at each image pixel, then we are **ray casting**.

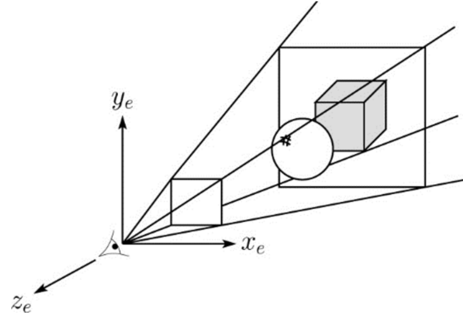


For each pixel center $\boldsymbol{P}_{ij}$

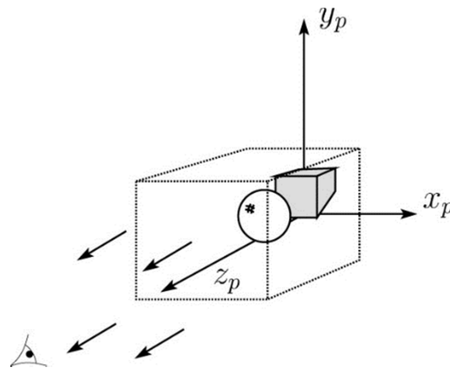
- Send ray from eye point (COP), **C**, through $\boldsymbol{P}_{ij}$ into scene.
- Intersect ray with each object.
- Select nearest intersection.

Warping space

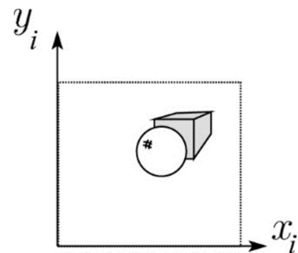
A very different approach is to take the imaging setup:



then warp all of space so that all the rays are parallel (and distant objects are smaller than closer objects):



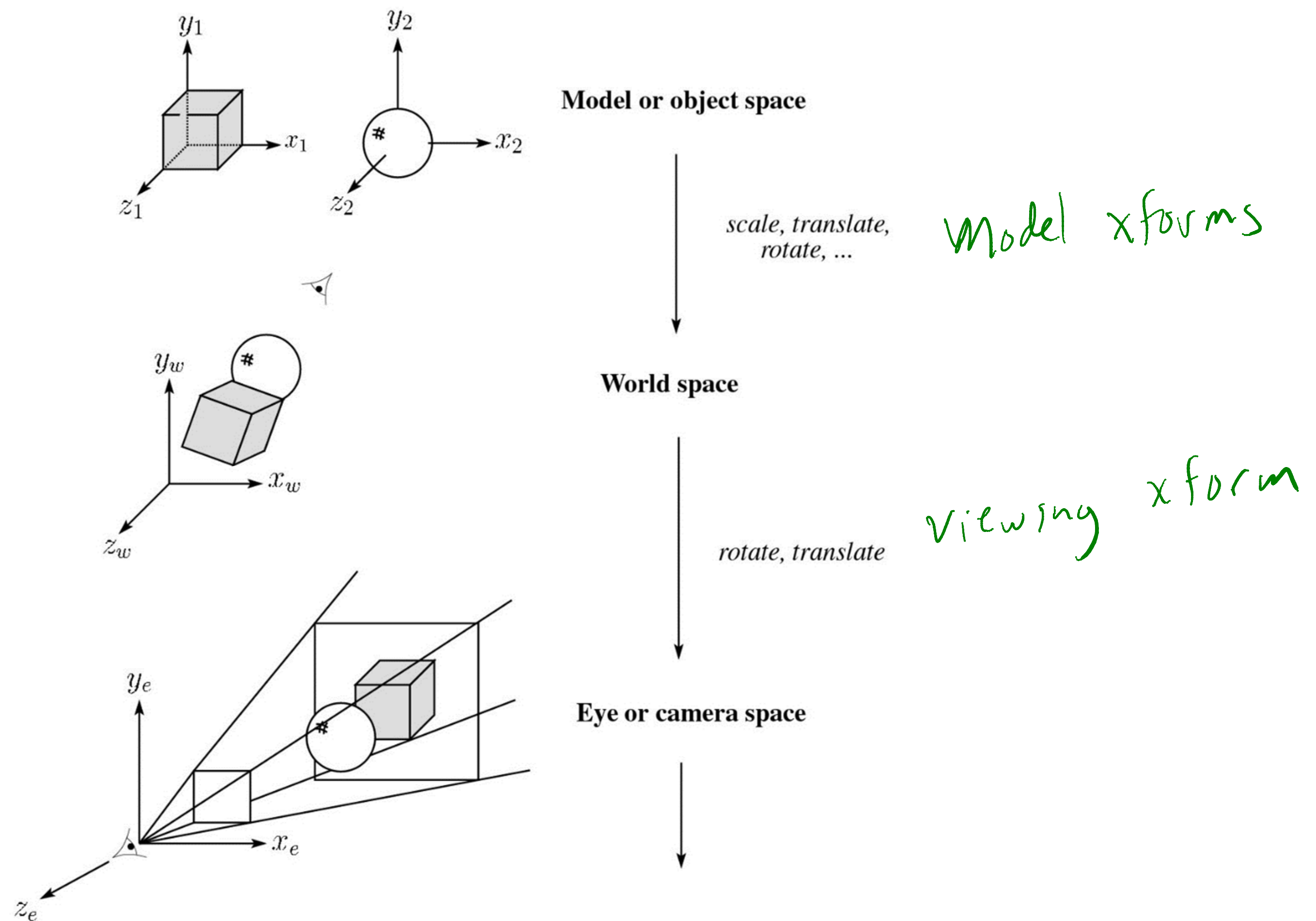
and then just draw everything onto the image plane, keeping track of what is in front:



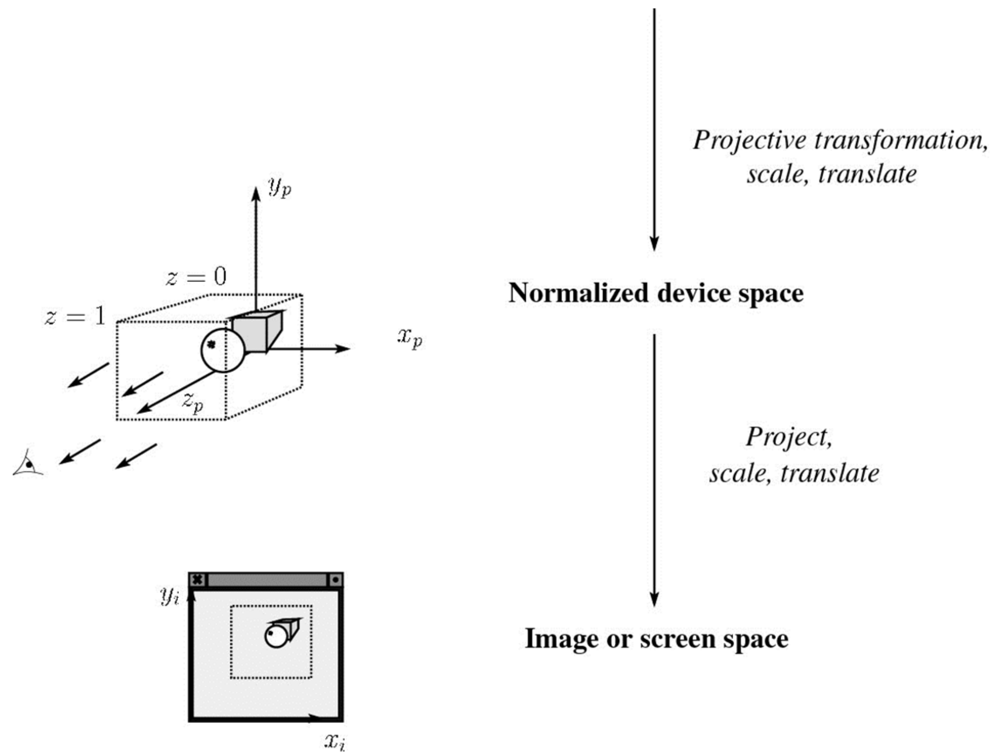
3D Geometry Pipeline

Graphics hardware follows the "warping space" approach.

Before being turned into pixels, a piece of geometry goes through a number of transformations...



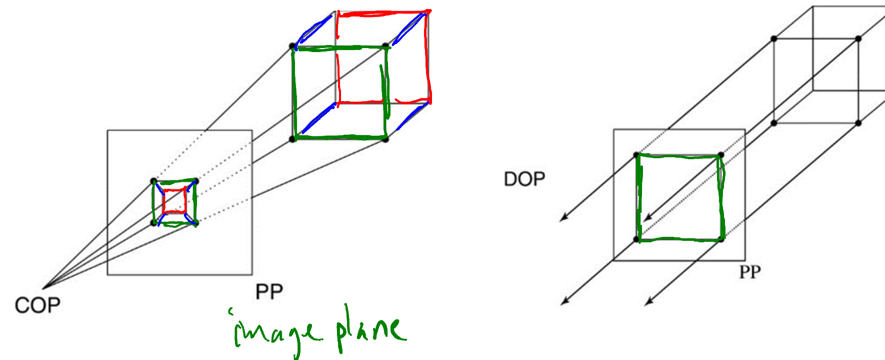
3D Geometry Pipeline (cont'd)



Projections

Projections transform points in *n*-space to *m*-space, where $m<n$.

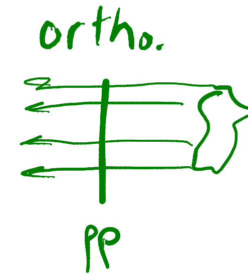
In 3-D, we map points from 3-space to the **projection plane** (PP) (a.k.a., image plane) along **projectors** (a.k.a., viewing rays) emanating from the center of projection (COP):



There are two basic types of projections:

- Perspective – distance from COP to PP finite
- Parallel – distance from COP to PP infinite

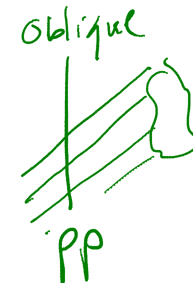
Parallel projections



For parallel projections, we specify a **direction of projection** (DOP) instead of a COP.

There are two types of parallel projections:

- **Orthographic projection** – DOP perpendicular to PP
- **Oblique projection** – DOP not perpendicular to PP



We can write orthographic projection onto the *z*=0 plane with a simple matrix.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Normally, we do not drop the z value right away. Why not?

Z-buffer

The **Z-buffer** or **depth buffer** algorithm [Catmull, 1974] can be used to determine which surface point is visible at each pixel.

Here is pseudocode for the Z-buffer hidden surface algorithm:

```
for each pixel (i,j) do
    Z-buffer [i,j] ← FAR
    Framebuffer[i,j] ← <background color>
end for
for each triangle A do
    for each pixel in A do
        Compute depth z of A at (i,j)
        if z > Z-buffer [i,j] then
            Z-buffer [i,j] ← z
            Framebuffer[i,j] ← color of A
        end if
    end for
end for
```

Q: What should FAR be set to? $-\infty$

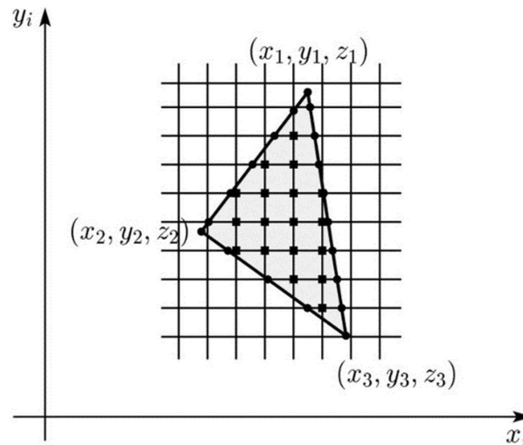
– FAR_CLIPPING_PLANE

– BIG_NUMBER

Rasterization

The process of filling in the pixels inside of a polygon is called **rasterization**.

During rasterization, the *z* value can be computed incrementally (fast!).



Curious fact:

- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for huge memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.

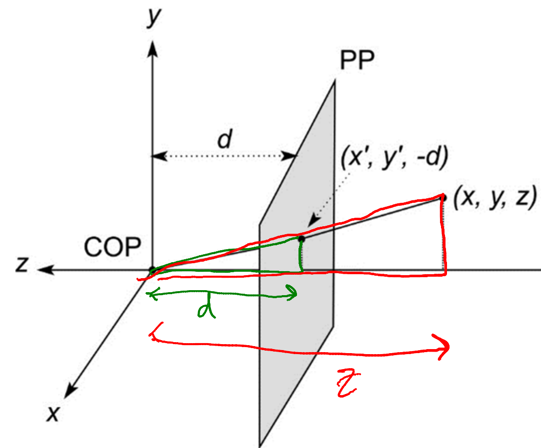
Properties of parallel projection

Properties of parallel projection:

- Not realistic looking
- Good for exact measurements
- Are actually a kind of affine transformation
  - Parallel lines remain parallel
  - Ratios are preserved
  - Angles not (in general) preserved
- Most often used in CAD, architectural drawings, etc., where taking exact measurement is important

Derivation of perspective projection

Consider the projection of a point onto the projection plane:



By similar triangles, we can compute how much the *x* and *y* coordinates are scaled:

$$\frac{d}{-z} = \frac{y'}{y}$$

$$y' = \frac{-d}{z} \cdot y$$

$$x' = \frac{-d}{z} x$$

Homogeneous coordinates revisited

Remember how we said that affine transformations work with the last coordinate always set to one.

What happens if the coordinate is not one?

We divide all the coordinates by w:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix}$$

If $w = 1$, then nothing changes.

Sometimes we call this division step the “perspective divide.”

Homogeneous coordinates and perspective projection

Now we can re-write the perspective projection as a matrix equation:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ -z/d \end{bmatrix}$$

$\div -z/d$

$(\cdot -d/z)$

After division by *w*, we get:

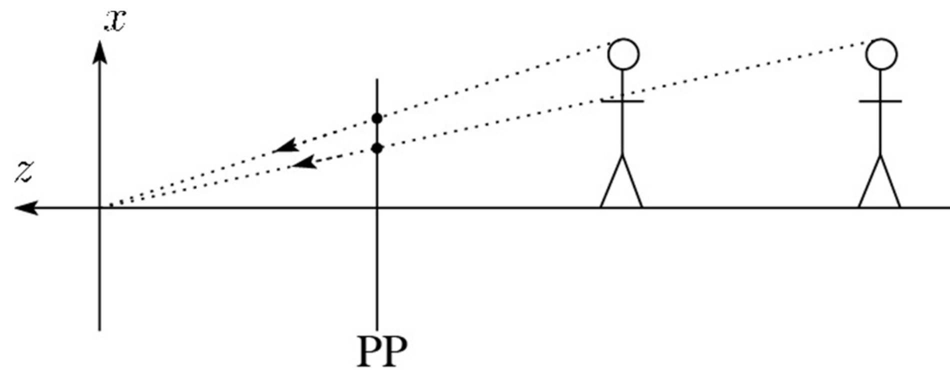
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{d}{z}x \\ -\frac{d}{z}y \\ 1 \end{bmatrix}$$

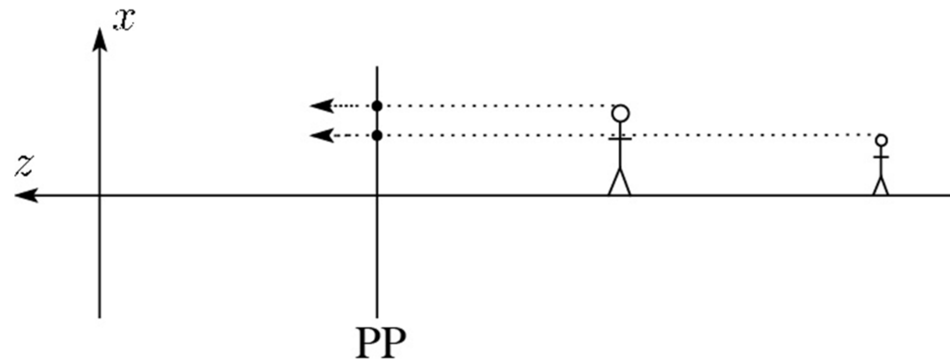
Again, projection implies dropping the *z* coordinate to give a 2D image, but we usually keep it around a little while longer.

Projective normalization

After applying the perspective transformation and dividing by *w*, we are free to do a simple parallel projection to get the 2D image.

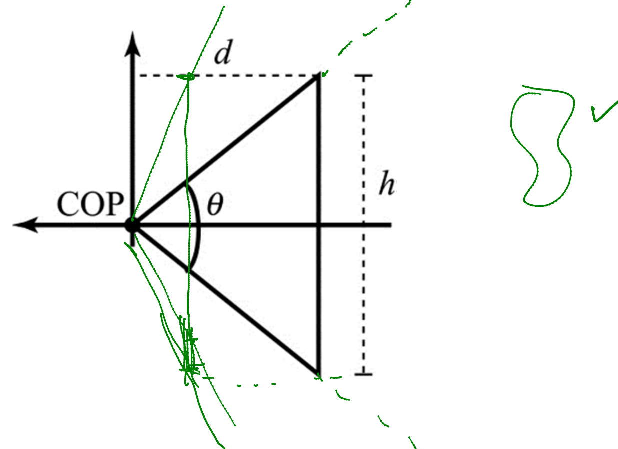
What does this imply about the shape of things after the perspective transformation + divide?





Viewing angle

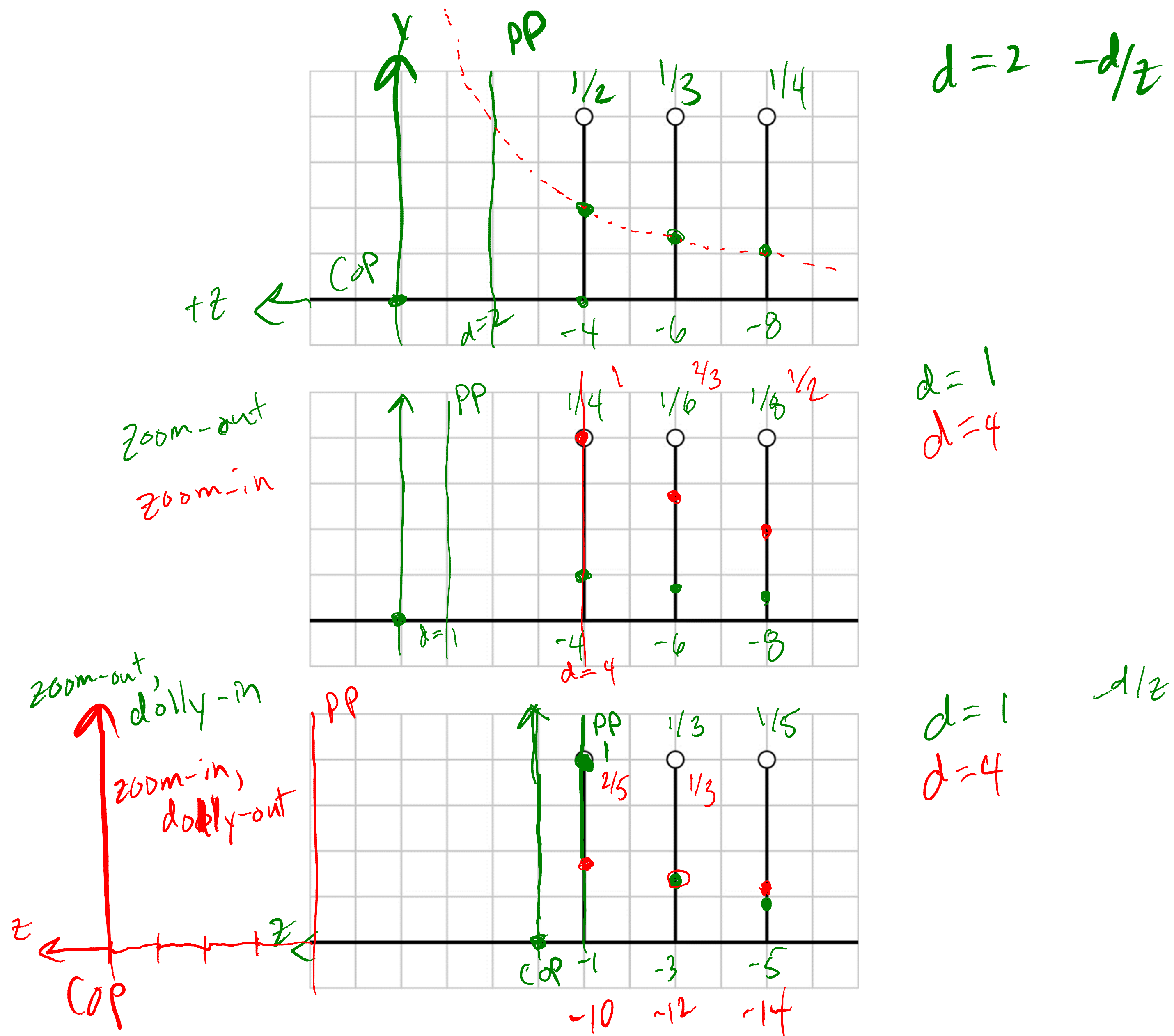
An alternative to specifying the distance from COP to PP is to specify a viewing angle:



Given the height of the image h and θ, what is d?

What happens to d as θ increases (while h is constant)? d decreases

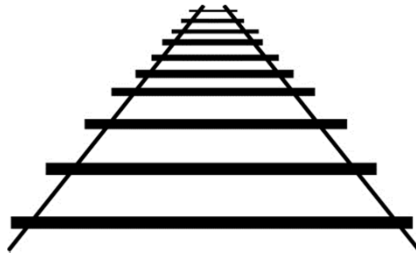
Zoom and dolly



Vanishing points

What happens to two parallel lines that are not parallel to the projection plane?

Think of train tracks receding into the horizon...



The equation for a line is:

$$\mathbf{l} = \mathbf{p} + t\mathbf{v} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} + t \begin{bmatrix} v_x \\ v_y \\ v_z \\ 0 \end{bmatrix}$$

After perspective transformation we get:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} p_x + tv_x \\ p_y + tv_y \\ -(p_z + tv_z)/d \end{bmatrix}$$

Vanishing points (cont'd)

Dividing by w:

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} -\frac{p_x + tv_x}{p_z + tv_z} d \\ -\frac{p_y + tv_y}{p_z + tv_z} d \\ 1 \end{bmatrix}$$

Letting t go to infinity:

We get a point!

What happens to the line $\mathbf{l} = \mathbf{q} + t\mathbf{v}$?

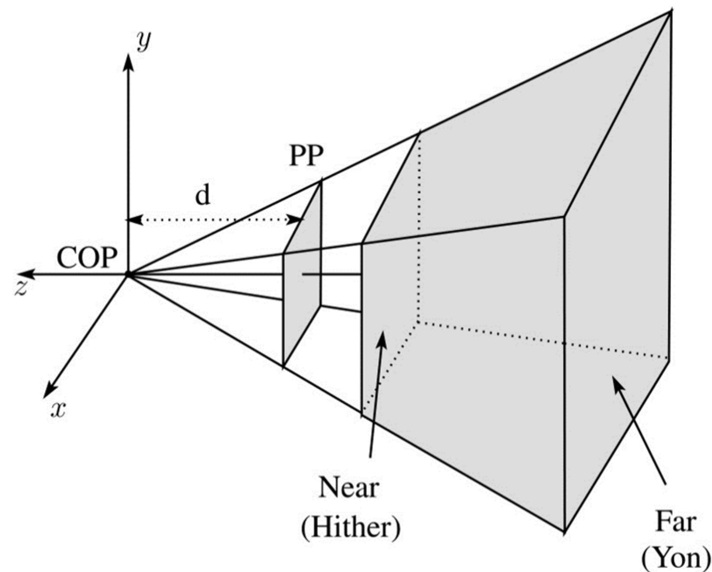
Each set of parallel lines intersect at a **vanishing point** on the PP.

Q: How many vanishing points are there?

Clipping and the viewing frustum

The center of projection and the portion of the projection plane that map to the final image form an infinite pyramid. The sides of the pyramid are **clipping planes**.

Frequently, additional clipping planes are inserted to restrict the range of depths. These clipping planes are called the **near** and **far** or the **hither** and **yon** clipping planes.

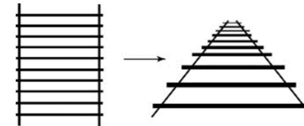


All of the clipping planes bound the the **viewing frustum**.

Properties of perspective projections

The perspective projection is an example of a **projective transformation**.

Here are some properties of projective transformations:



- Lines map to lines
- Parallel lines do <u>not</u> necessarily remain parallel
- Ratios are <u>not</u> preserved

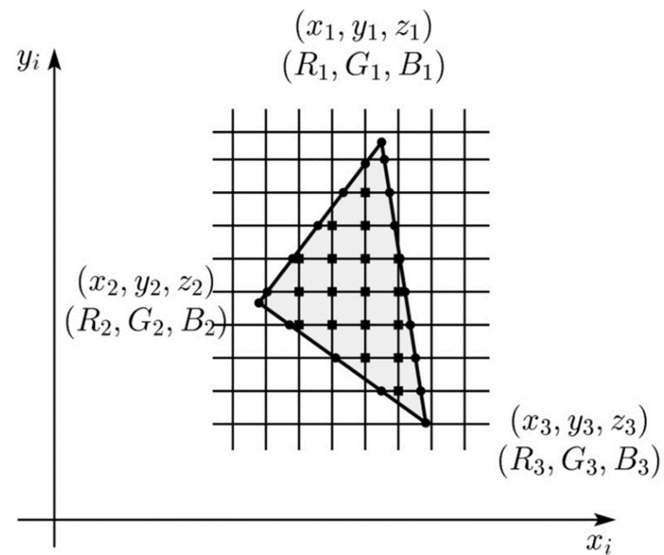
One of the advantages of perspective projection is that size varies inversely with distance – looks realistic.

A disadvantage is that we can't judge distances as exactly as we can with parallel projections.

Rasterization with color

Recall that the z-buffer works by interpolating z-values across a triangle that has been projected into image space, a process called rasterization.

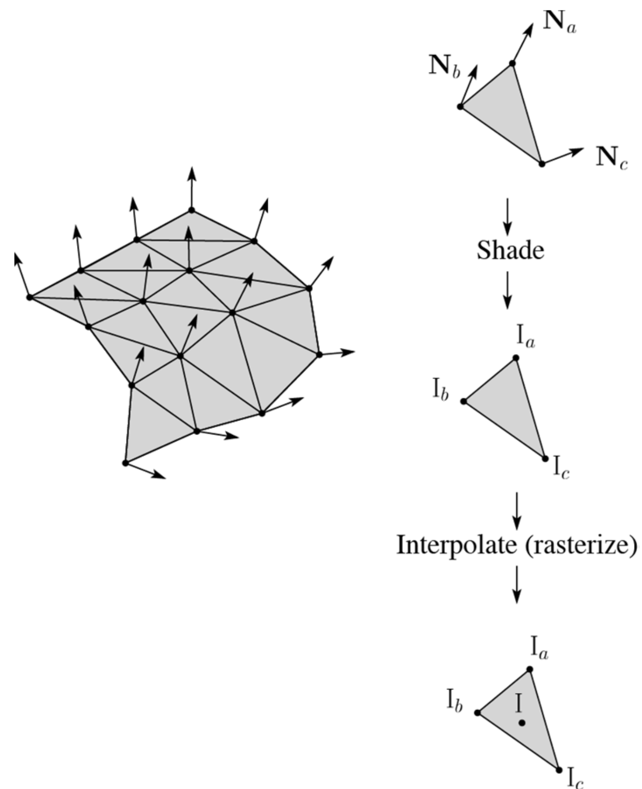
During rasterization, colors can be smeared across a triangle as well:



Gouraud interpolation

Recall from the shading lecture, rendering with per triangle normals leads to faceted appearance. An improvement is to compute per-vertex normals and use graphics hardware to do **Gouraud interpolation**:

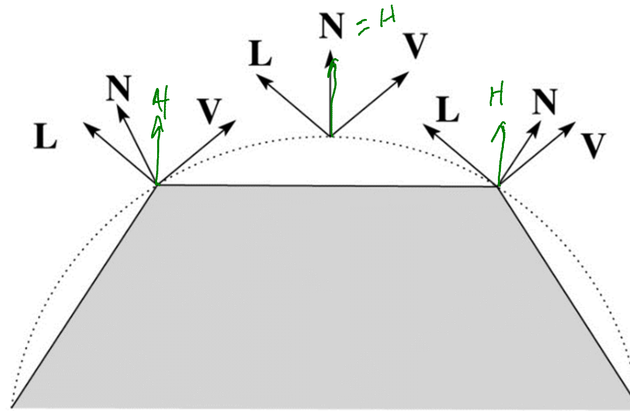
1. Compute normals at the vertices.
2. Shade only the vertices.
3. Interpolate the resulting vertex colors.

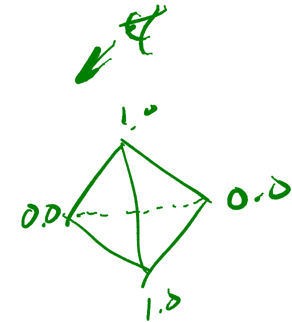


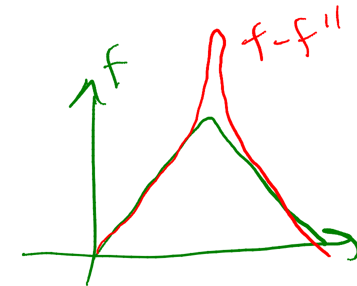
Gouraud interpolation artifacts

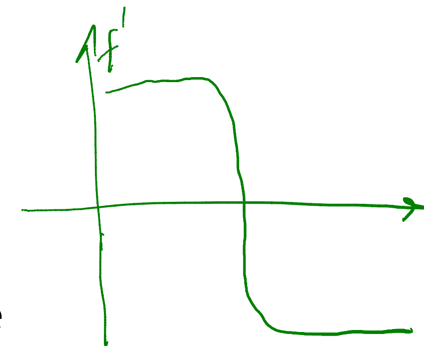
Gouraud interpolation has significant limitations.

1. If the polygonal approximation is too coarse, we can miss specular highlights.





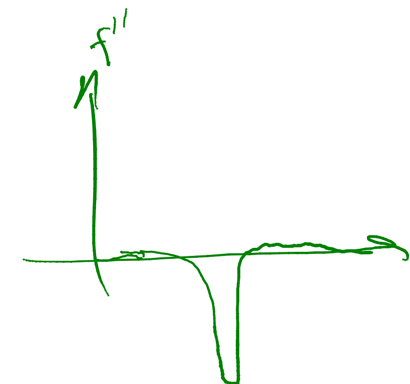




2. We will encounter **Mach banding** (derivative discontinuity enhanced by human eye).

This is what graphics hardware does by default.

A substantial improvement is to do…

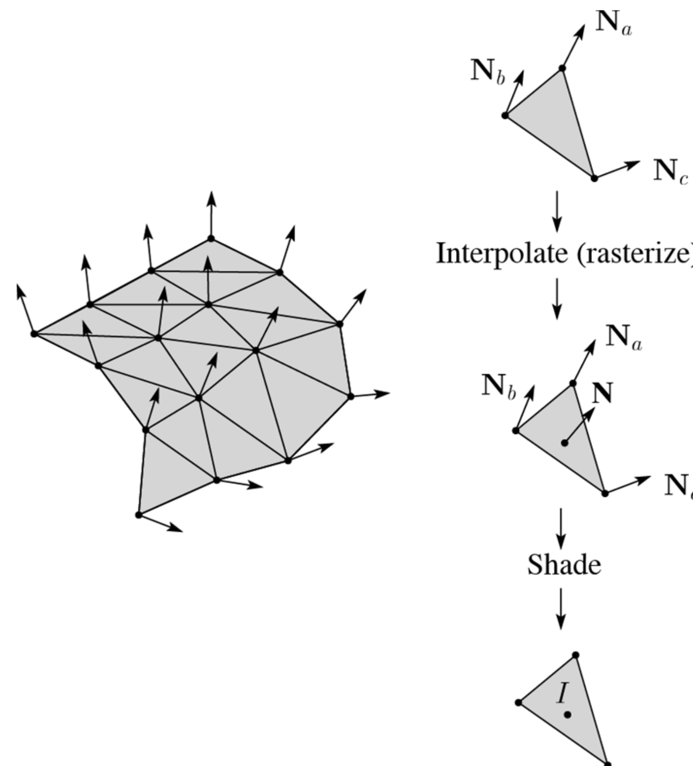


Phong interpolation

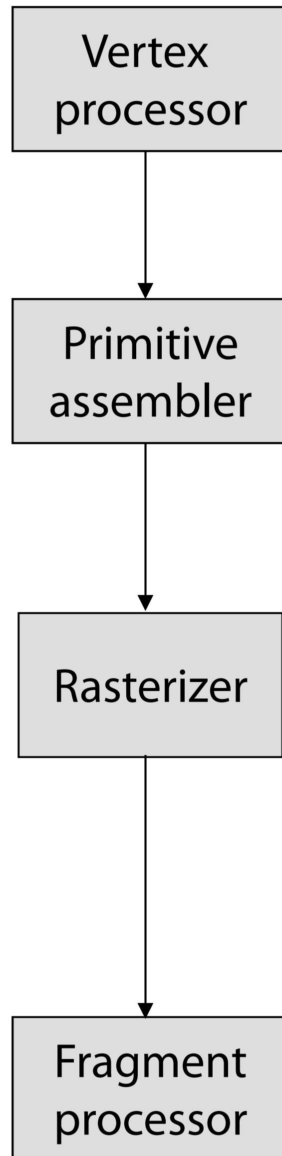
To get an even smoother result with fewer artifacts, we can perform **Phong *interpolation***.

Here's how it works:

1. Compute normals at the vertices.
2. Interpolate normals and normalize.
3. Shade using the interpolated normals.



Default pipeline: Gouraud interpolation

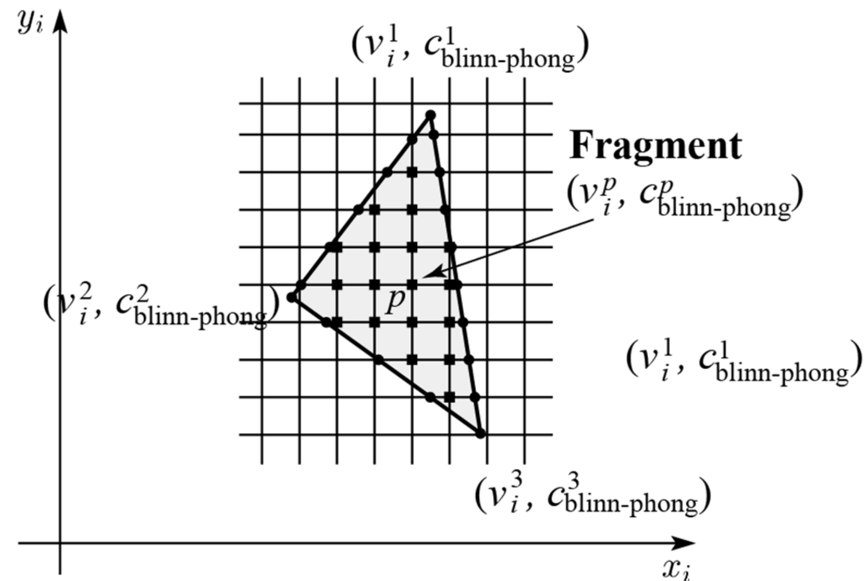


Default vertex processing:

$L \leftarrow$ determine lighting direction
$V \leftarrow$ determine viewing direction
$N \leftarrow \text{normalize}(n_e)$
$c_{\text{blinn-phong}} \leftarrow$ shade with L, V, N, k_d, k_s, n_s

attach $c_{\text{blinn-phong}}$ to vertex as "varying"
$v_i \leftarrow$ project v to image

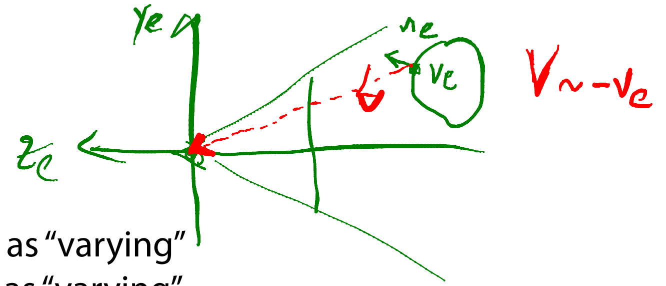
$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle

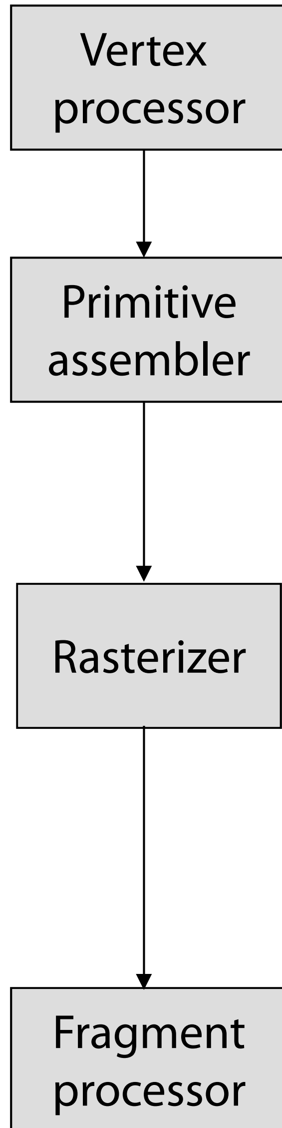


Default fragment processing:

color $\leftarrow c^p_{\text{blinn-phong}}$

Programmable pipeline: Phong-interpolated normals!



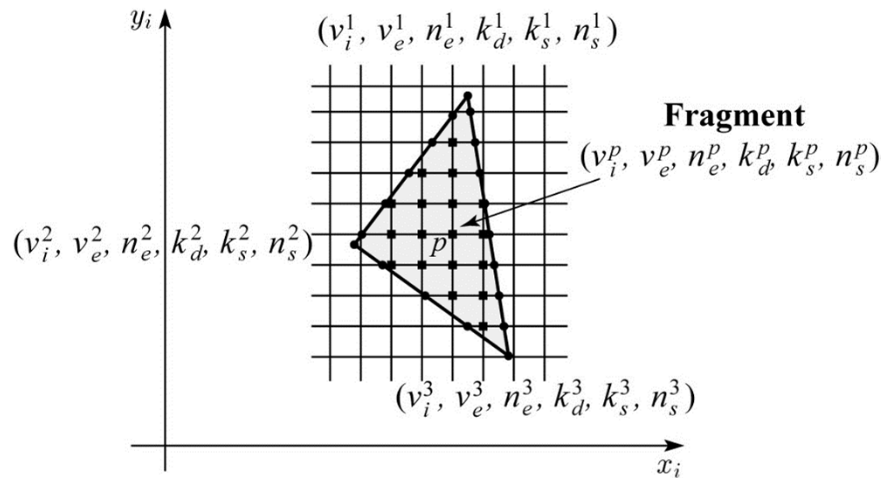


Vertex shader:

attach n_e to vertex as "varying"
attach v_e to vertex as "varying"
$v_i \leftarrow$ project v to image

(attach kd, ks, ns ...)

$v_i^1, v_i^2, v_i^3 \rightarrow$ triangle



Fragment shader:

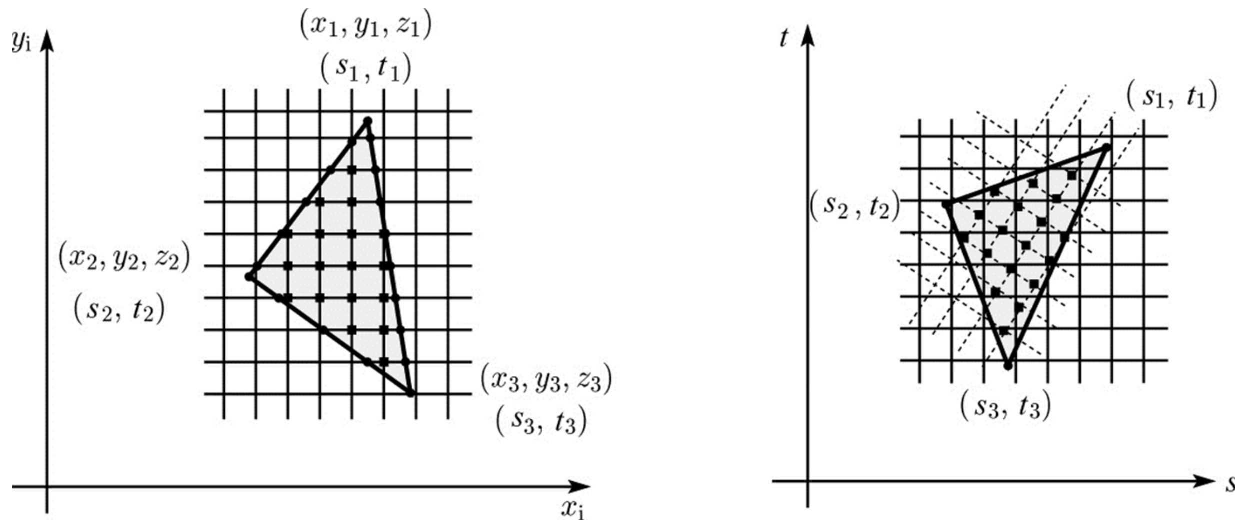
$L \leftarrow$ determine lighting direction
$V \leftarrow$ determine viewing direction
$N \leftarrow$ normalize(n_e^p)
color $\leftarrow$ shade with $L, V, N, k_d^p, k_s^p, n_s^p$

Texture mapping and the z-buffer

Texture-mapping can also be handled in z-buffer algorithms.

Method:

- Scan conversion is done in screen space, as usual
- Each pixel is colored according to the texture
- Texture coordinates are found by Gouraud-style interpolation



Note: Mapping is more complicated to handle perspective correctly!

Shading in OpenGL

The OpenGL lighting model allows you to associate different lighting colors according to material properties they will influence.

Thus, our original shading equation:

$$I = k_e + k_a I_{La} + \sum_j \frac{1}{a_j + b_j r_j + c_j r_j^2} I_{L,j} B_j \left[k_d \left(\mathbf{N} \cdot \mathbf{L}_j \right) + k_s \left(\mathbf{N} \cdot \mathbf{H}_j \right)_+^{n_s} \right]$$

becomes:

$$I = k_e + k_a I_{La} + \sum_j \frac{1}{a_j + b_j r_j + c_j r_j^2} \left[k_a I_{La,j} + B_j \left\{ k_d I_{Ld,j} (\mathbf{N} \cdot \mathbf{L}_j) + k_s I_{Ls,j} (\mathbf{N} \cdot \mathbf{H}_j)_+^{n_s} \right\} \right]$$

where you can have a global ambient light with intensity I_{La} in addition to having an ambient light intensity $I_{La,j}$ associated with each individual light, as well as separate diffuse and specular intensities, $I_{Ld,j}$ and $I_{Ls,j}$, repectively.

Materials in OpenGL

The OpenGL code to specify the surface shading properties is fairly straightforward. For example:

```
GLfloat ke[] = { 0.1, 0.15, 0.05, 1.0 };
GLfloat ka[] = { 0.1, 0.15, 0.1, 1.0 };
GLfloat kd[] = { 0.3, 0.3, 0.2, 1.0 };
GLfloat ks[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat ns[] = { 50.0 };
glMaterialfv(GL_FRONT, GL_EMISSION, ke);
glMaterialfv(GL_FRONT, GL_AMBIENT, ka);
glMaterialfv(GL_FRONT, GL_DIFFUSE, kd);
glMaterialfv(GL_FRONT, GL_SPECULAR, ks);
glMaterialfv(GL_FRONT, GL_SHININESS, ns);
```

Notes:

- The `GL_FRONT` parameter tells OpenGL that we are specifiying the materials for the front of the surface.
- Only the alpha value of the diffuse color is used for blending. It’s usually set to 1.

Shading in OpenGL, cont'd

In OpenGL this equation, for one light source (the 0th) is specified something like:

```
GLfloat La[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat La0[] = { 0.1, 0.1, 0.1, 1.0 };
GLfloat Ld0[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat Ls0[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat pos0[] = { 1.0, 1.0, 1.0, 0.0 };
GLfloat a0[] = { 1.0 };
GLfloat b0[] = { 0.5 };
GLfloat c0[] = { 0.25 };
GLfloat S0[] = { -1.0, -1.0, 0.0 };
GLfloat beta0[] = { 45 };
GLfloat e0[] = { 2 };

glLightModelfv(GL_LIGHT_MODEL_AMBIENT, La);
glLightfv(GL_LIGHT0, GL_AMBIENT, La0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, Ld0);
glLightfv(GL_LIGHT0, GL_SPECULAR, Ls0);
glLightfv(GL_LIGHT0, GL_POSITION, pos0);
glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, a0);
glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION, b0);
glLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, c0);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, S0);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, beta0);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, e0);
```

Shading in OpenGL, cont'd

Notes:

You can have as many as GL_MAX_LIGHTS lights in a scene. This number is system-dependent.

For directional lights, you specify a light direction, not position, and the attenuation and spotlight terms are ignored.

The directions of directional lights and spotlights are specified in the coordinate systems *of the lights*, not the surface points as we've been doing in lecture.