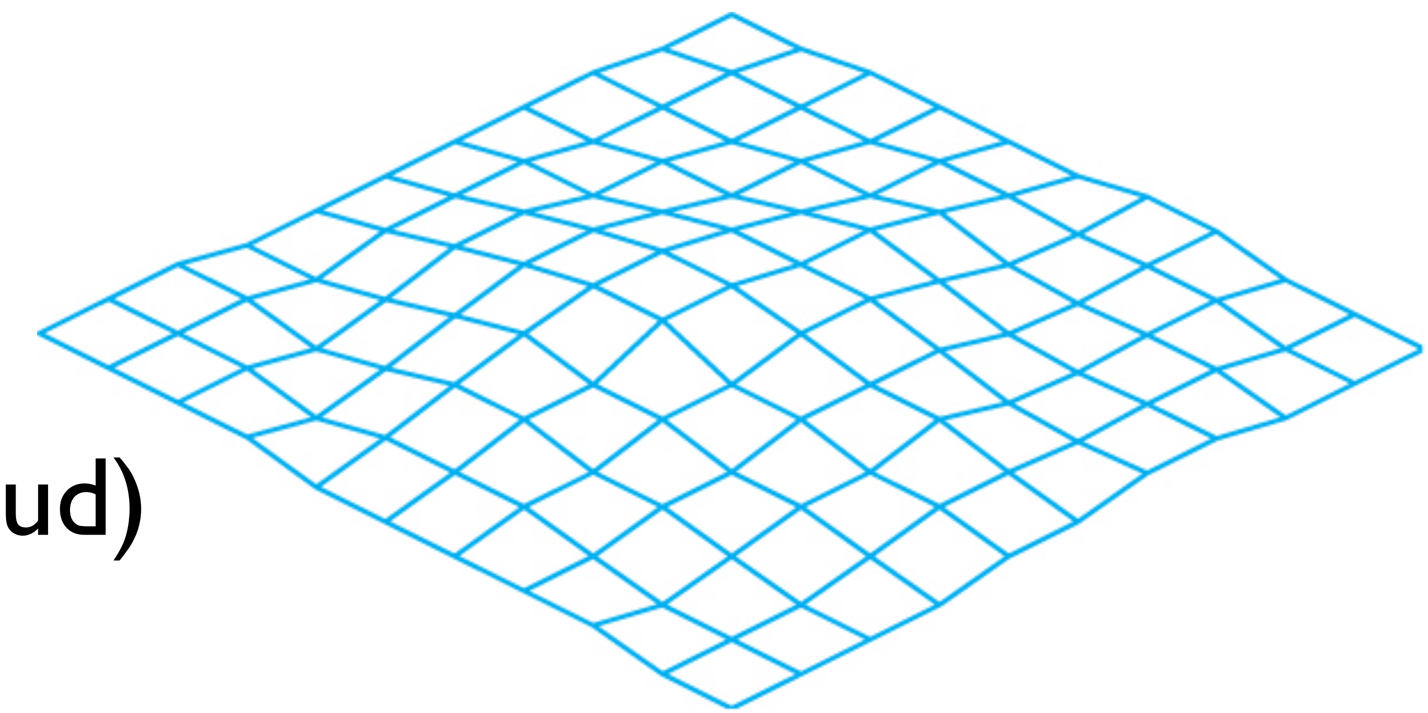
# Shading Polygonal Geometry

# Smooth surfaces are often approximated by polygons

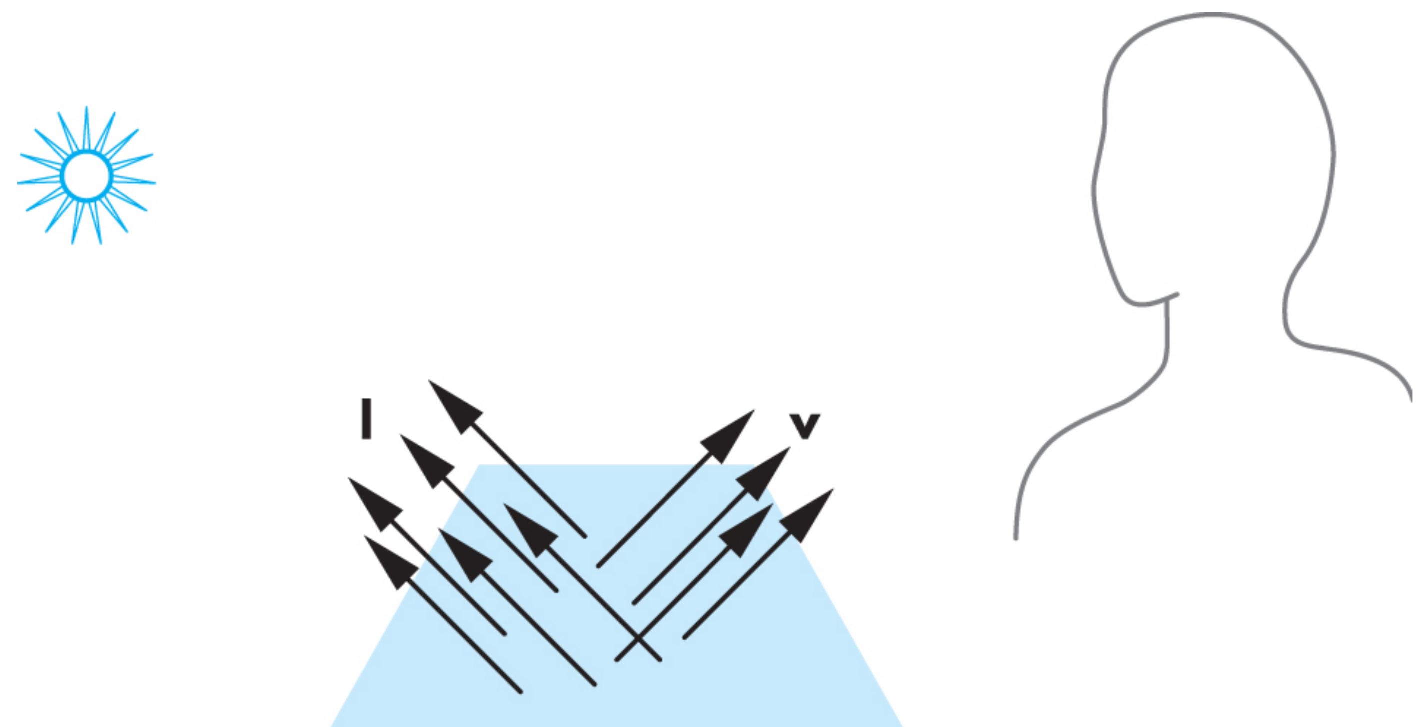
Shading approaches:

1. flat
2. per-vertex (Gouraud)
3. per-fragment



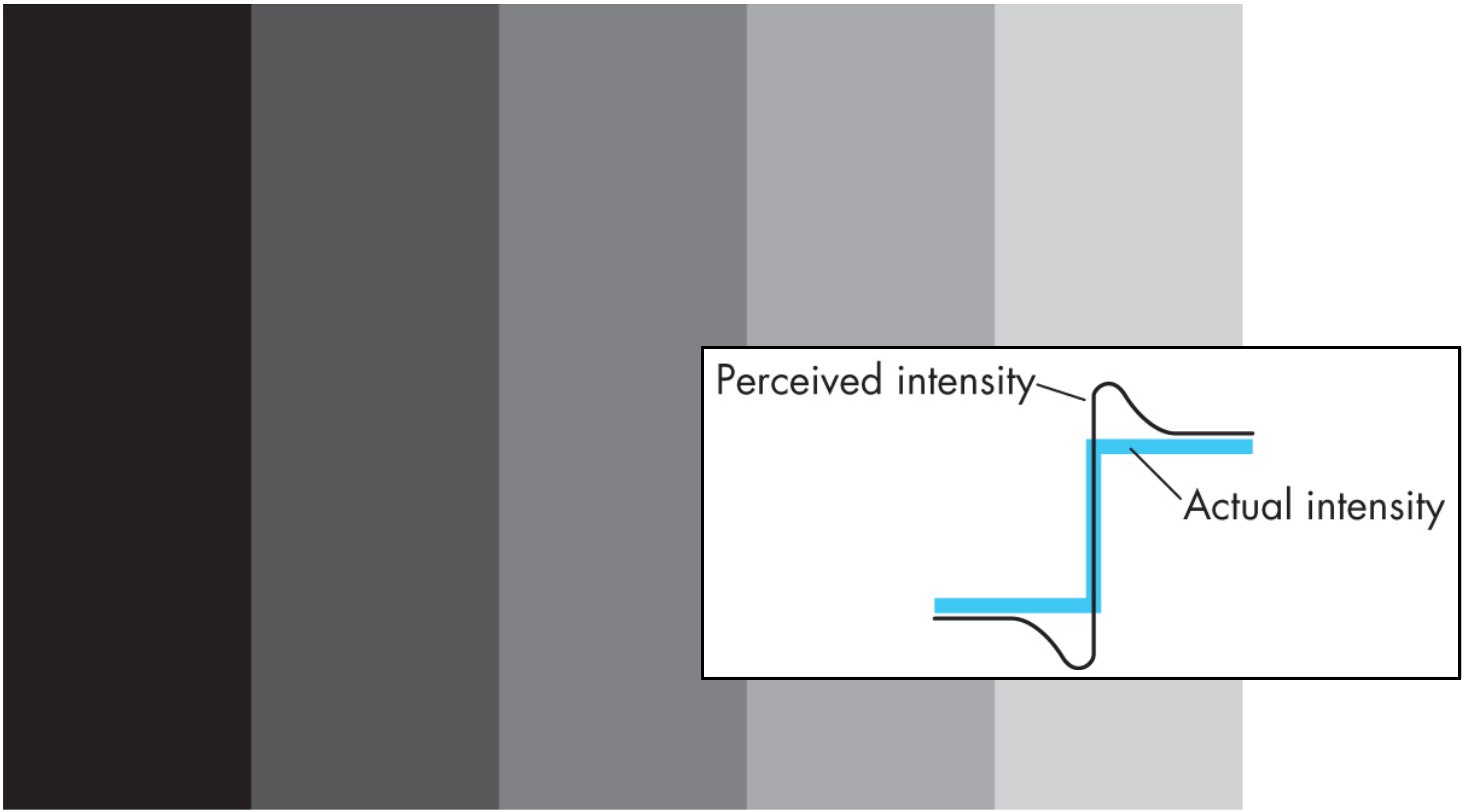
# Flat Shading

do the shading
calculation once
per **polygon**



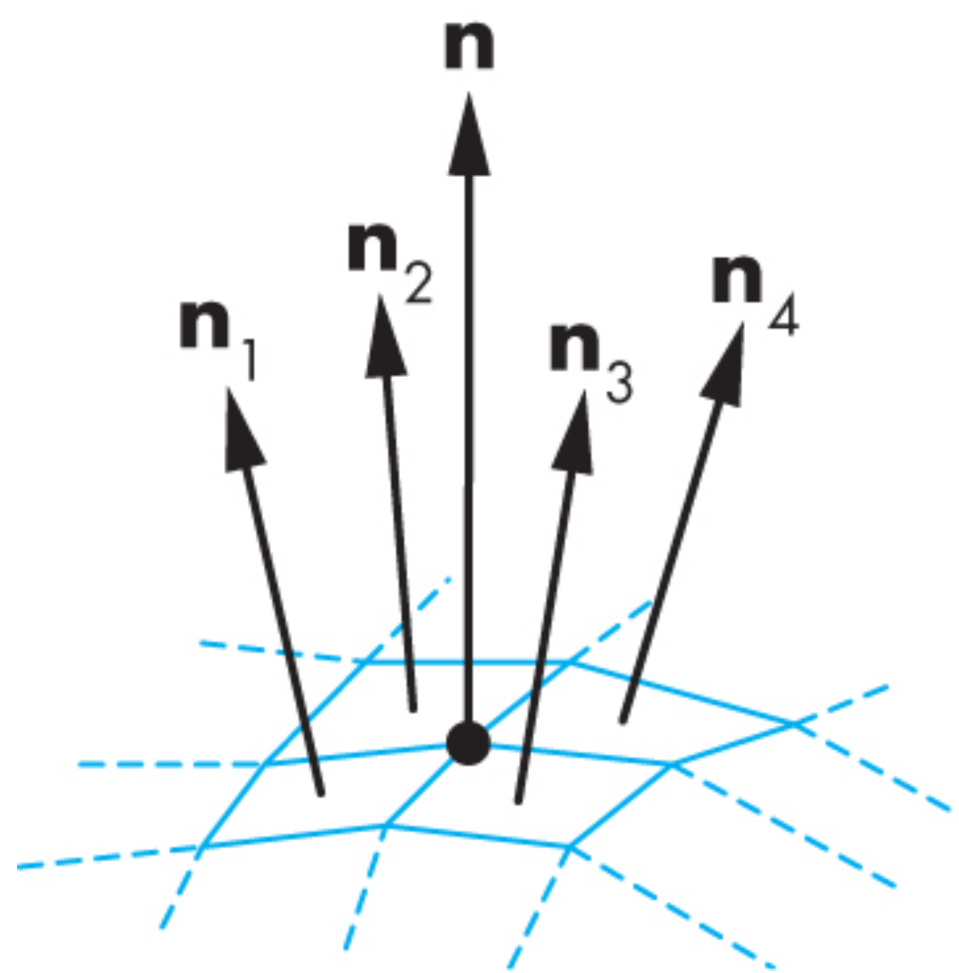

valid for light at $\infty$
and viewer at $\infty$
and faceted surfaces

# Mach Band Effect



# Per-Vertex Shading

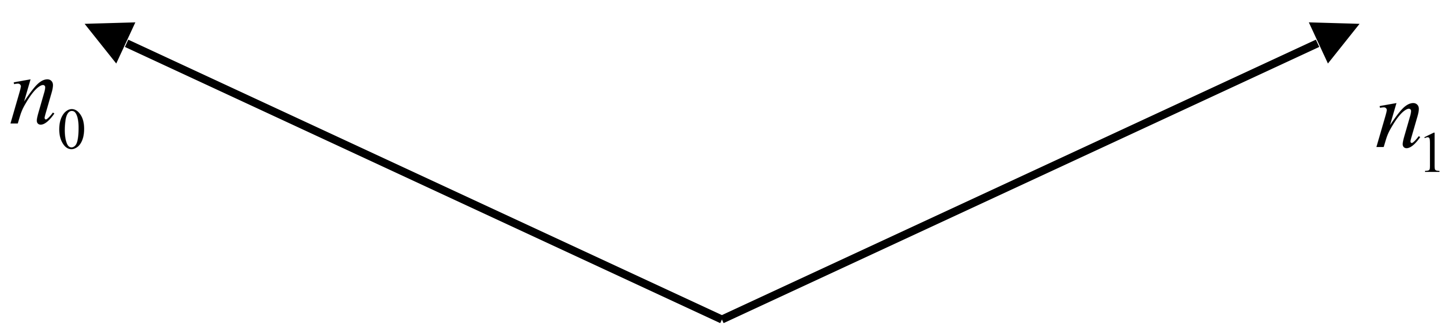
$$\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{||\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4||}$$




do the shading calculation once per **vertex**

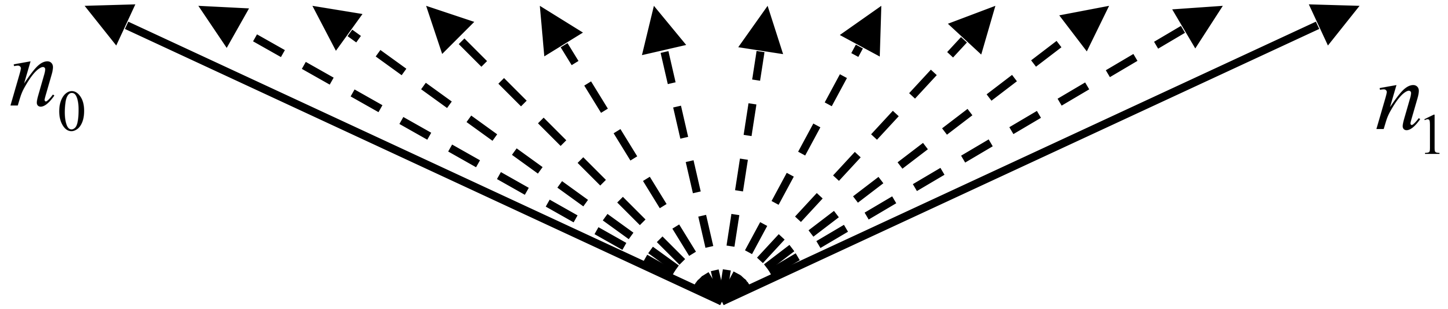
# Interpolating Normals

- Must renormalize



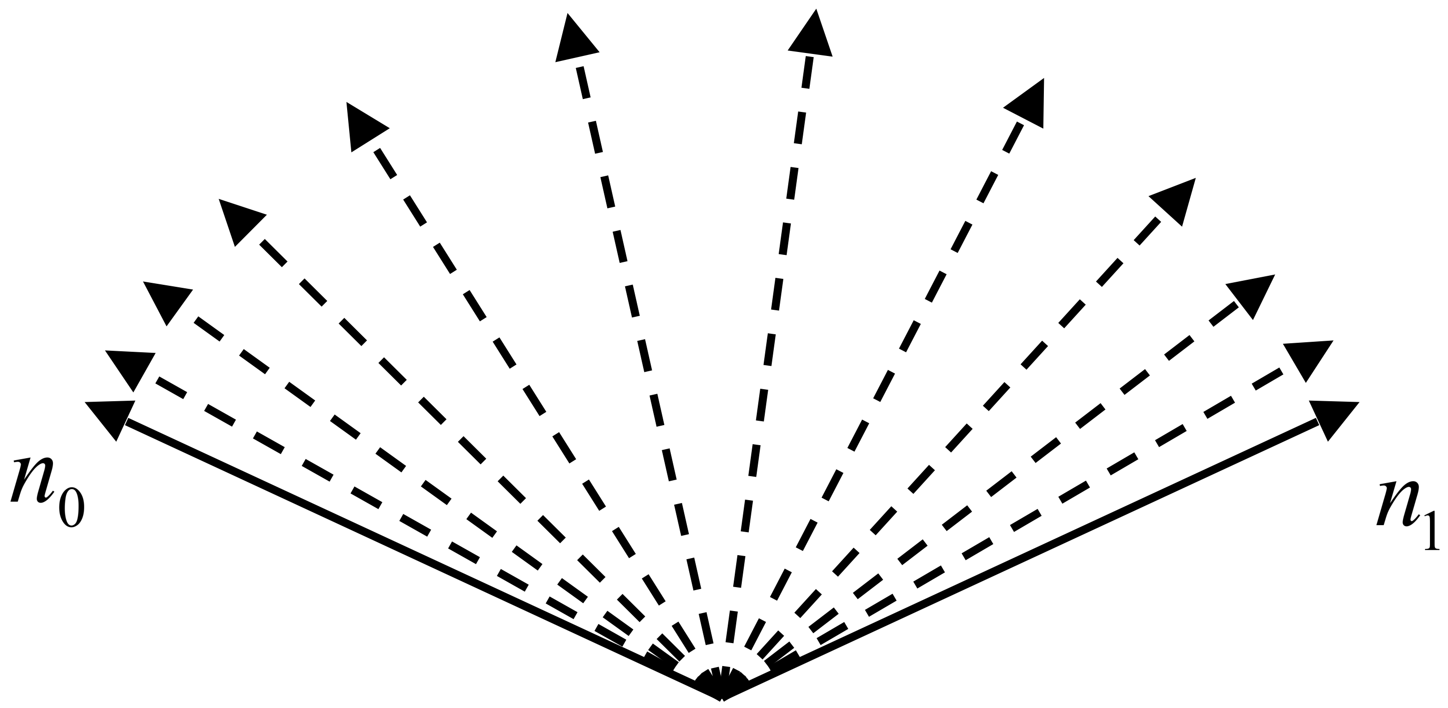
# Interpolating Normals

- Must renormalize

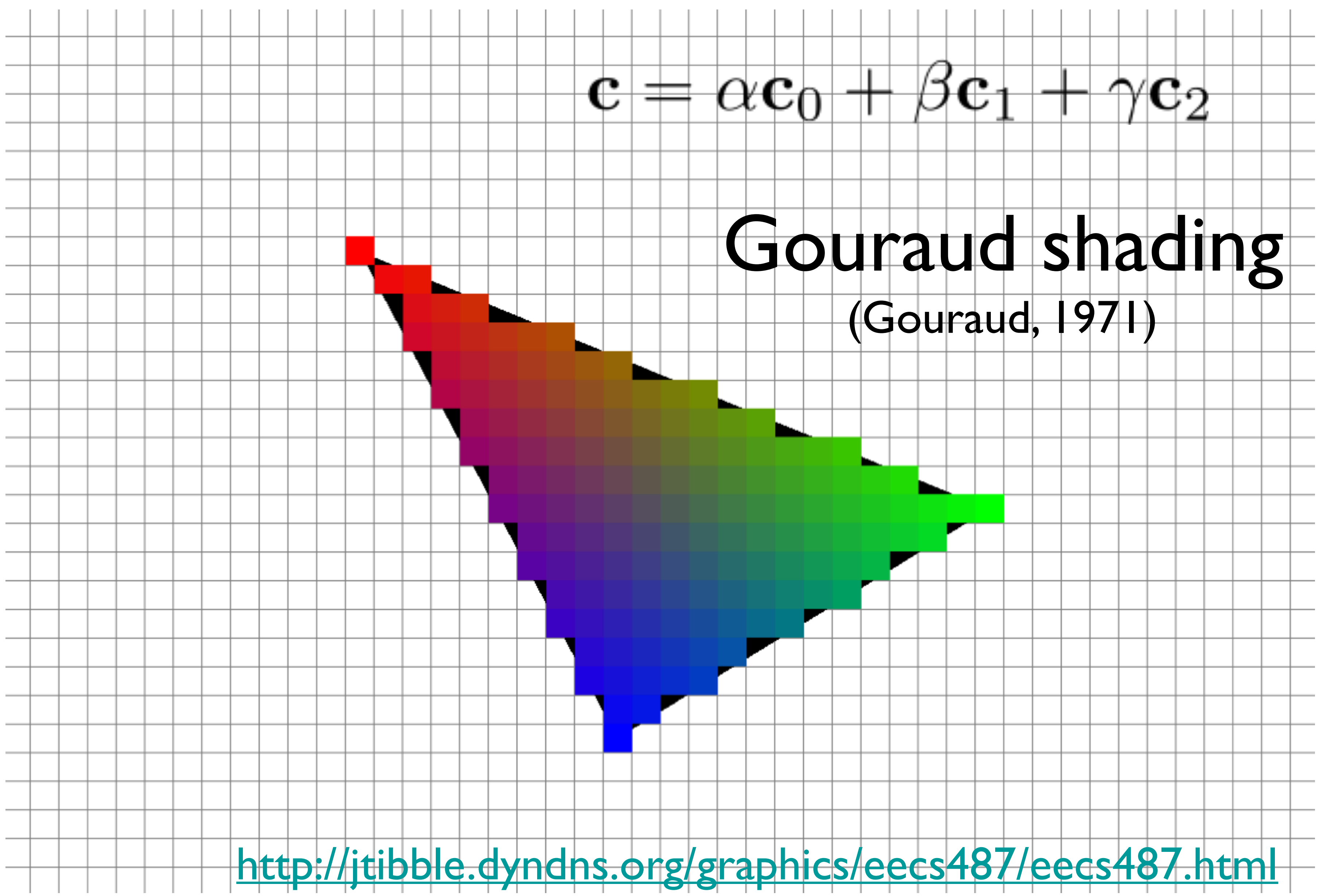


# Interpolating Normals

- Must renormalize



# We can interpolate attributes using barycentric coordinates



$$\mathbf{c} = \alpha \mathbf{c}_0 + \beta \mathbf{c}_1 + \gamma \mathbf{c}_2$$

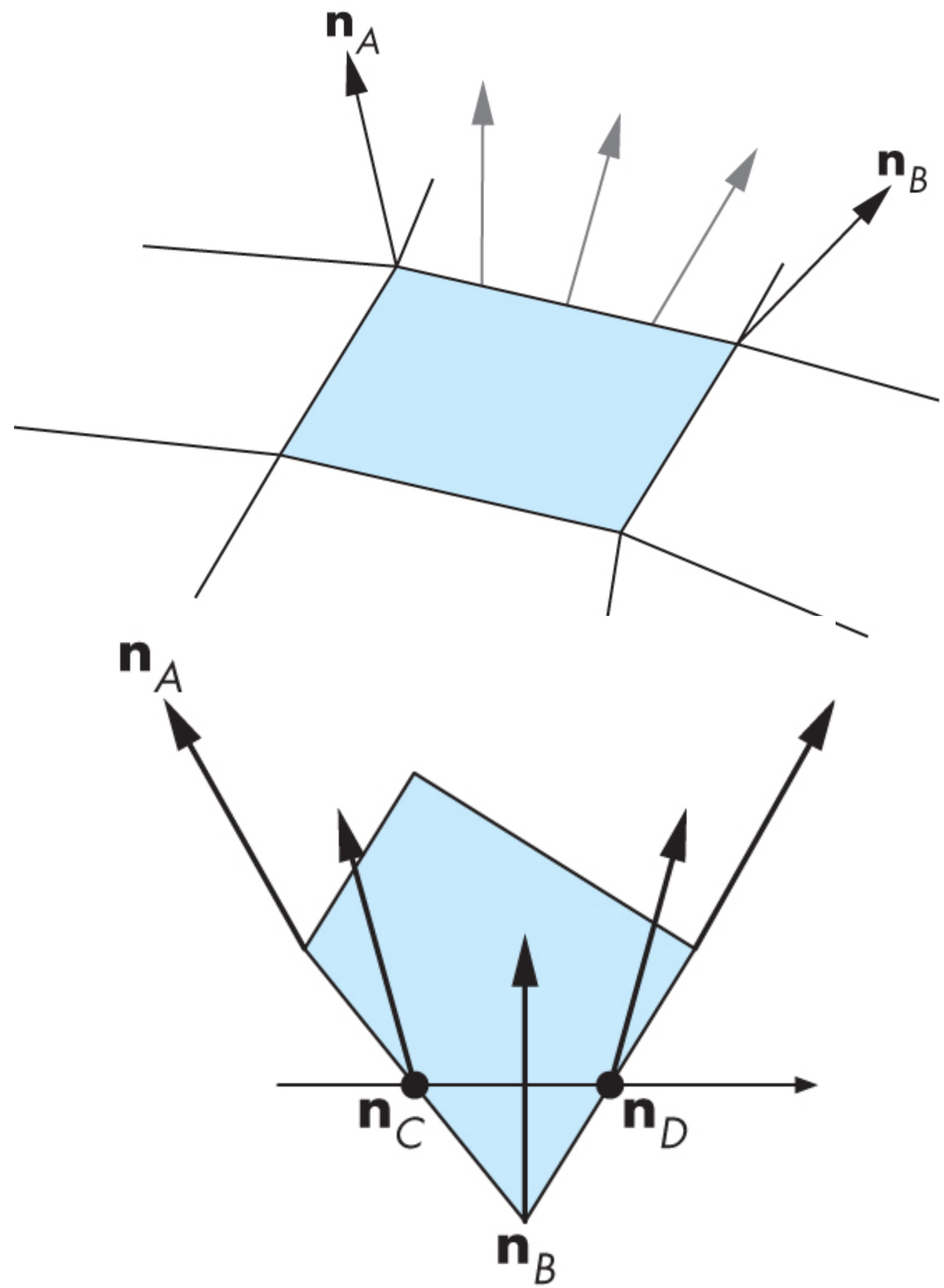
Gouraud shading

(Gouraud, 1971)

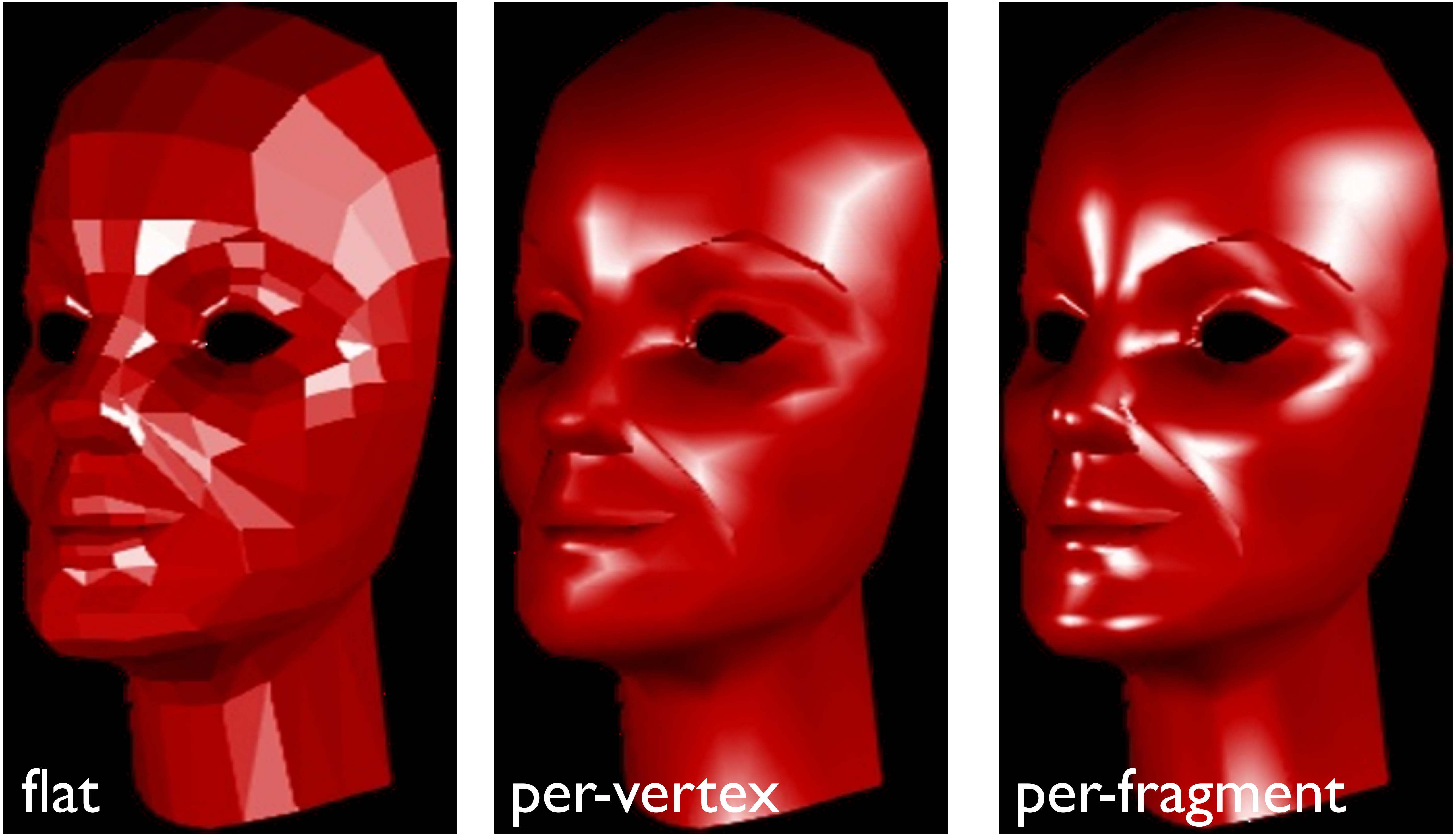
http://jtibble.dyndns.org/graphics/eecs487/eecs487.html

# Per-Fragment Shading

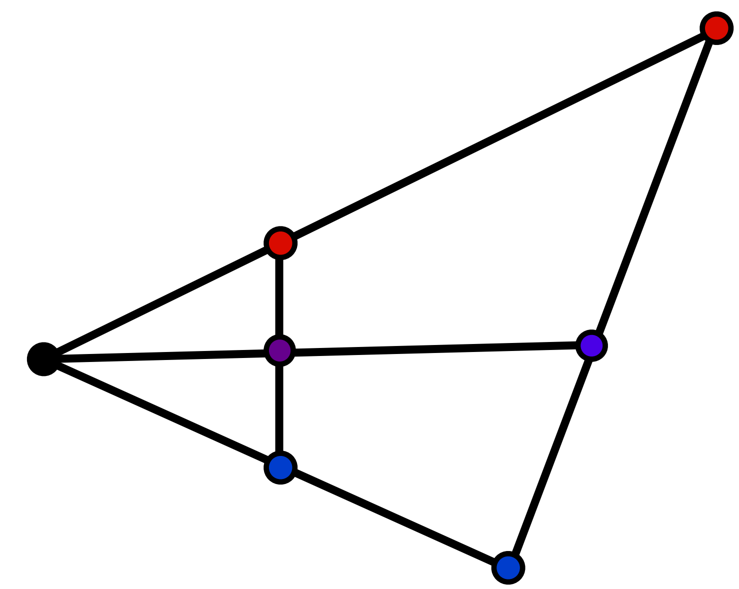
do the shading calculation once per **fragment**



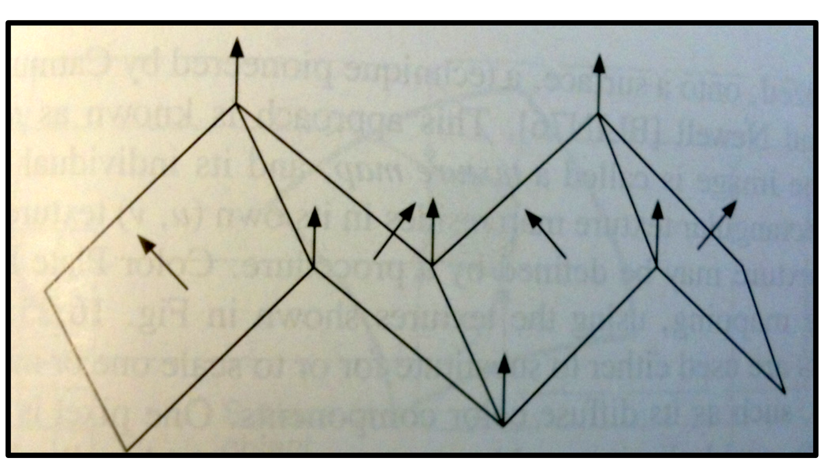
# Comparison

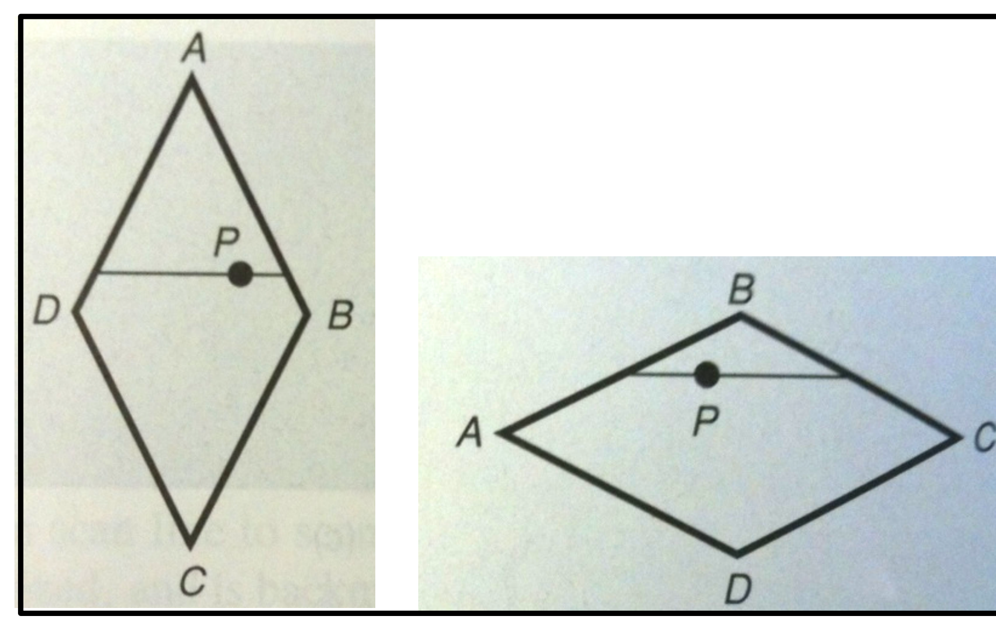


# Problems with Interpolated Shading



- Polygonal silhouette
- Perspective distortion
- Orientation dependence
- Unrepresentative surface normals

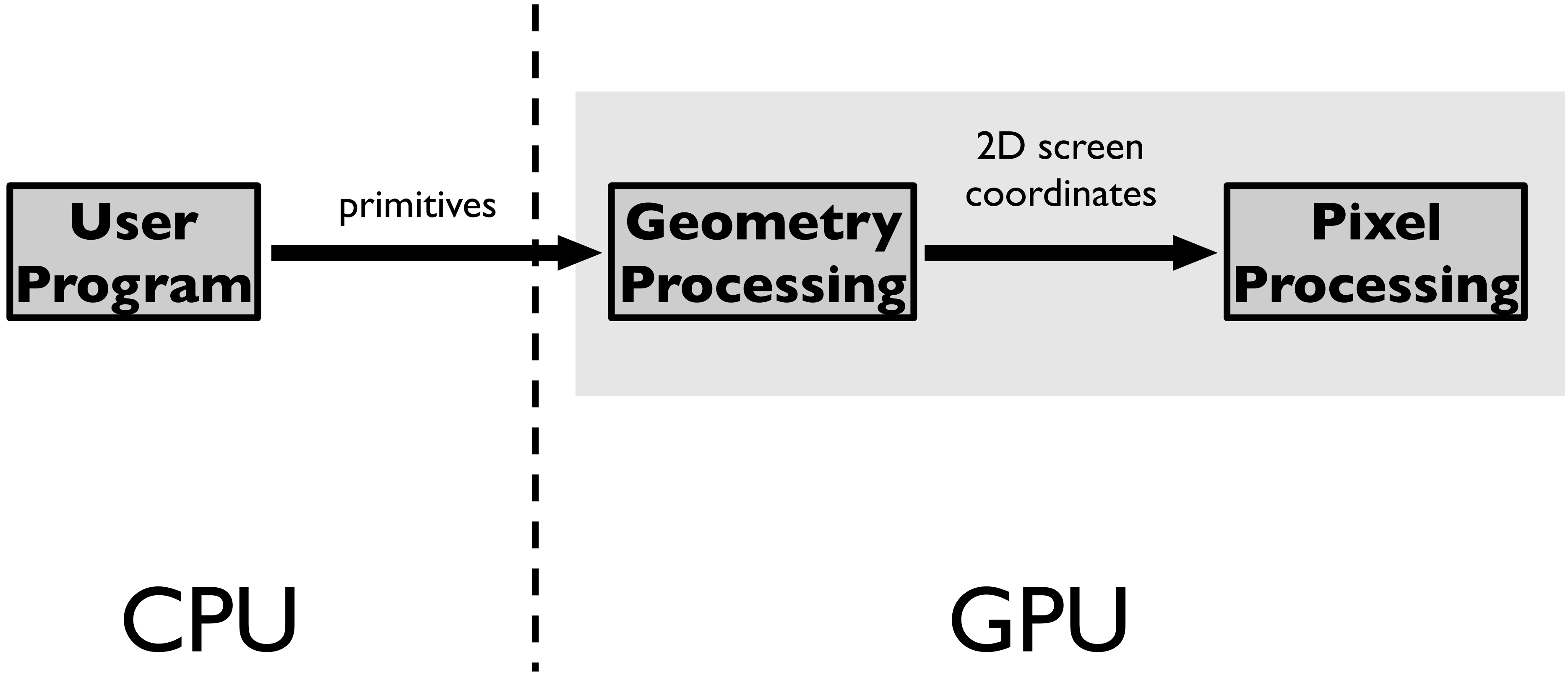





[Foley, van Dam, Feiner, Hughes]

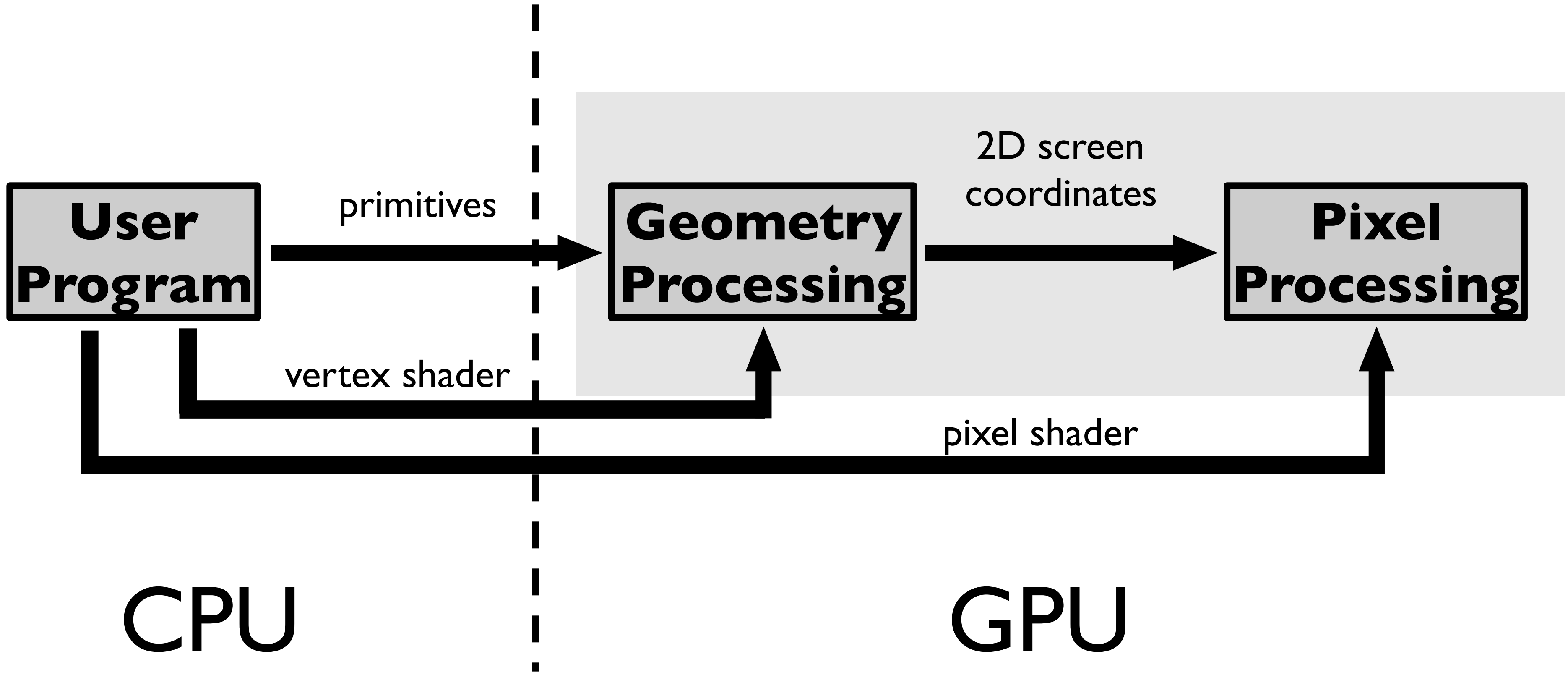
# Programmable Shading

# Fixed-Function Pipeline




Control pipeline through GL state variables

# Programmable Pipeline




Supply shader programs to be executed on GPU as part of pipeline

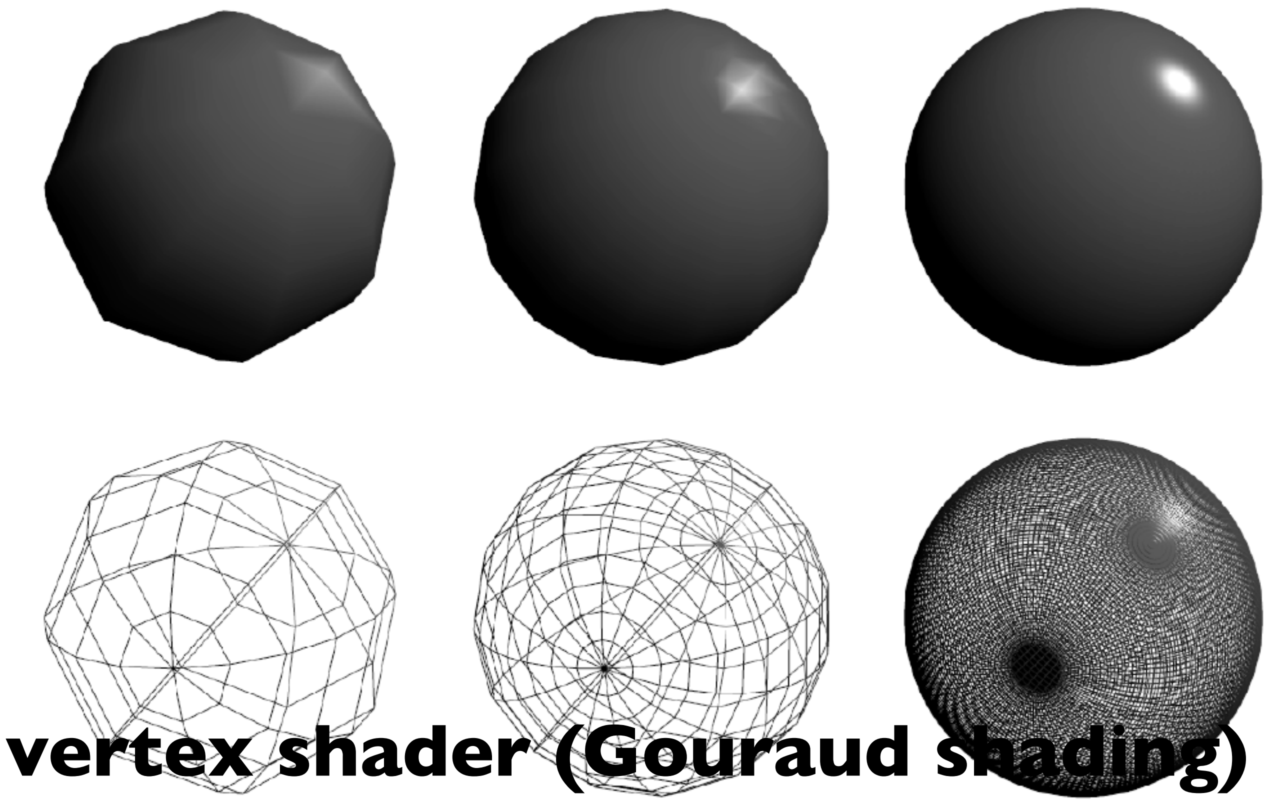
# Phong reflectance in vertex and pixel shaders using GLSL

```
void main(void)
{
    vec4 v = gl_modelView_Matrix * gl_Vertex;
    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 l = normalize(gl_lightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = fl_LightSource[0].diffuse;
    vec4 ca - vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(,n,l)))
            + cl* pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(,n,l)));

    gl_FrontColor = color;
    gl_Position = ftransform();
}
```



**per-vertex shader (Gouraud shading)**

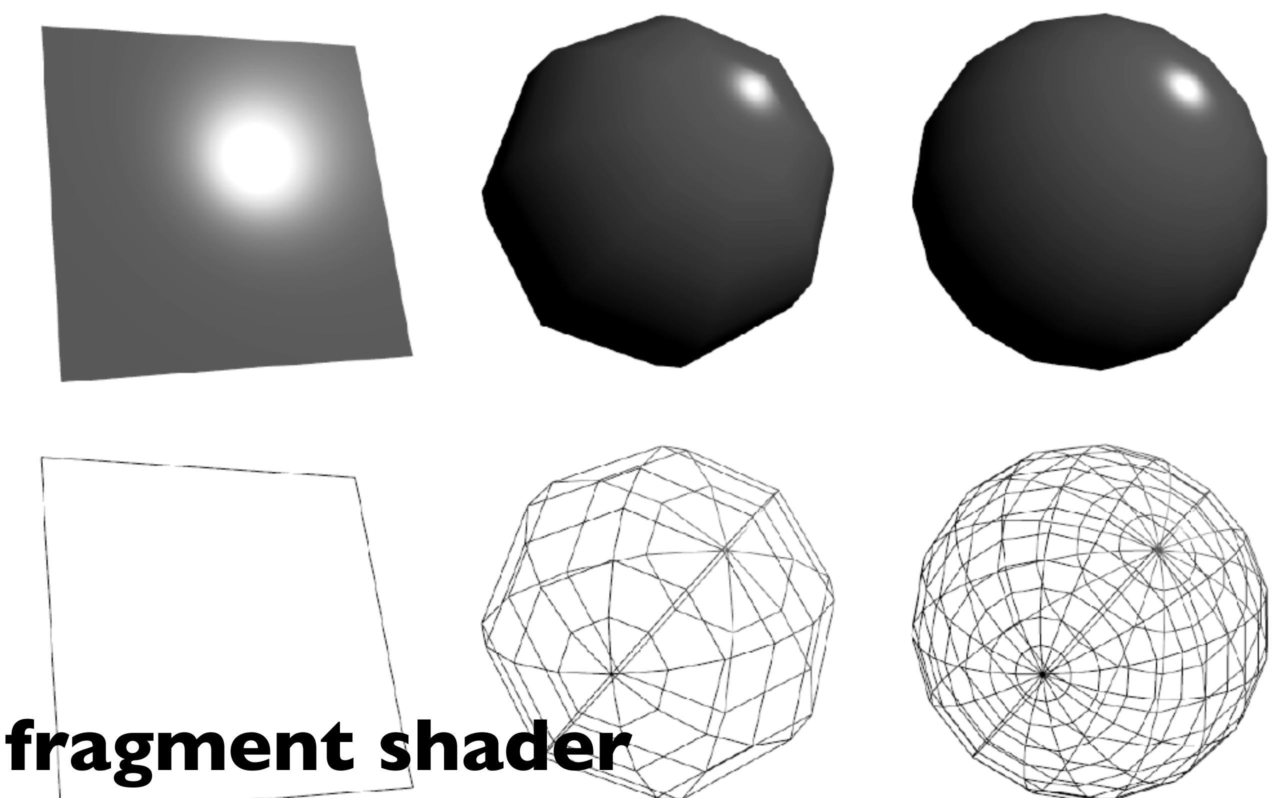
```
varying vec4 v;
varying vec3 n;

void main(void)
{
    vec3 l = normalize(gl_lightSource[0].position - v);
    vec3 h = normalize(l - normalize(v));

    float p = 16;
    vec4 cr = gl_FrontMaterial.diffuse;
    vec4 cl = fl_LightSource[0].diffuse;
    vec4 ca - vec4(0.2, 0.2, 0.2, 1.0);

    vec4 color;
    if (dot(h,n) > 0)
        color = cr * (ca + cl * max(0,dot(,n,l)))
            + cl* pow(dot(h,n), p);
    else
        color = cr * (ca + cl * max(0,dot(,n,l)));

    gl_FragColor = color;
}
```

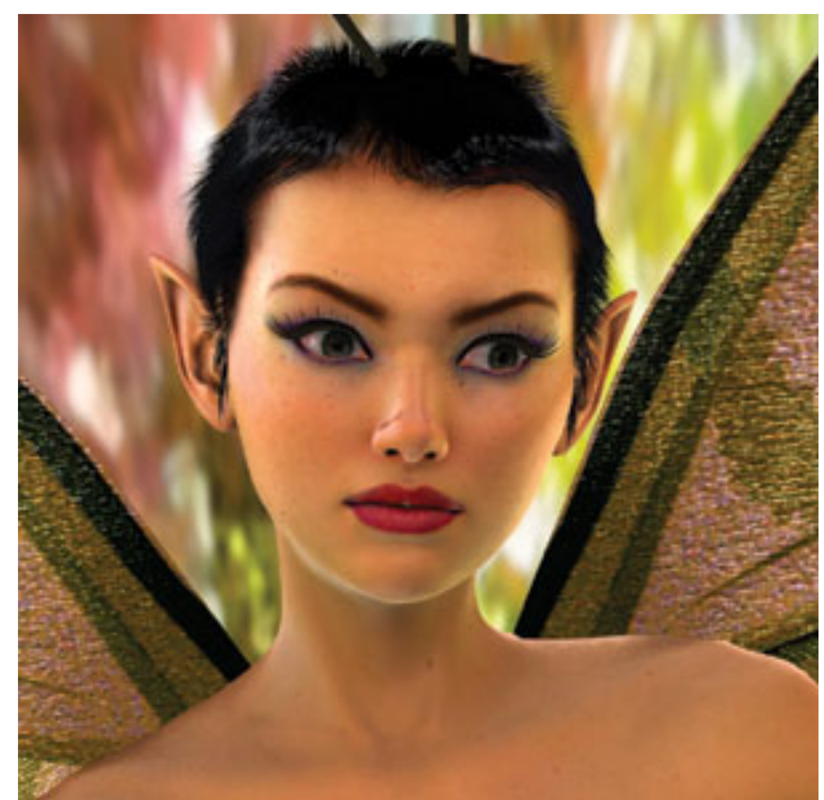


**per-fragment shader**

[Shirley and Marschner]

Rusty car shader, NVIDIA

*Call of Juarez* DX10 Benchmark, ATI

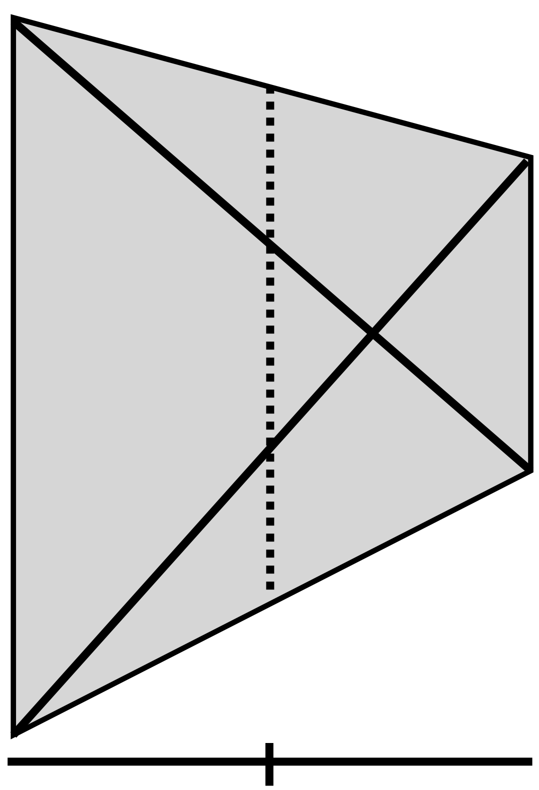


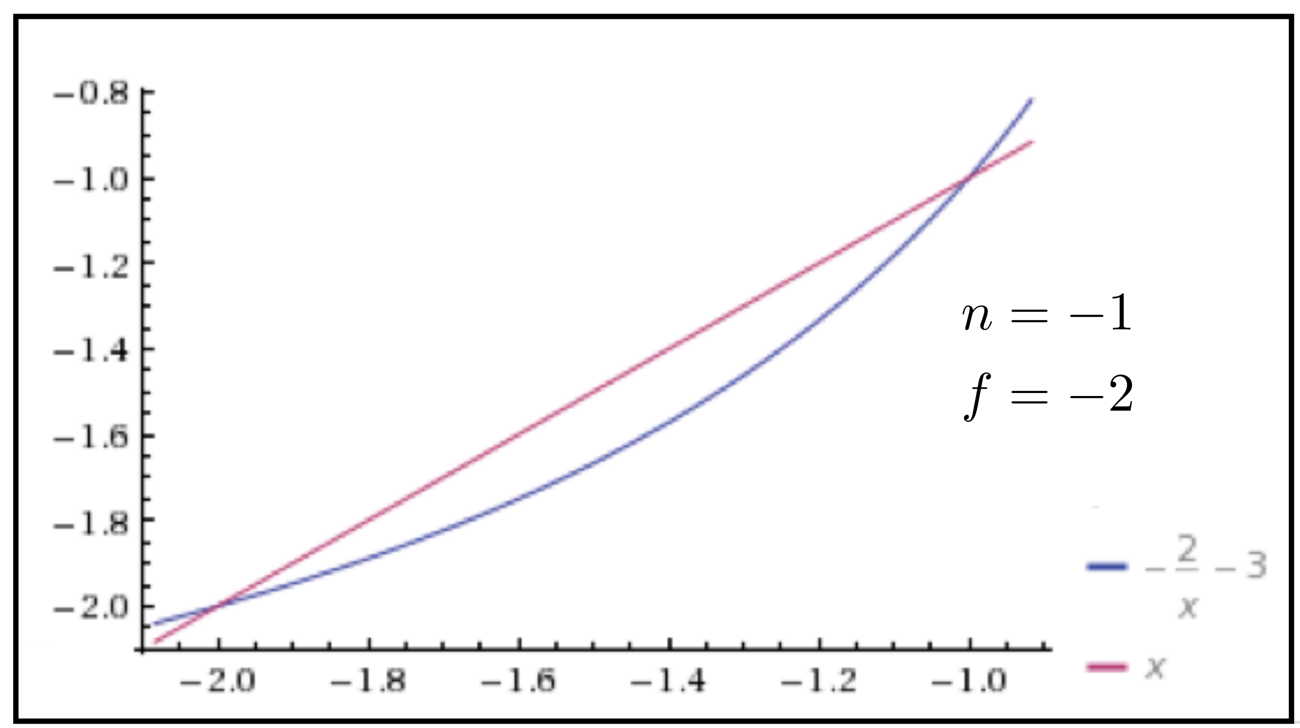
*Dawn,* NVIDIA

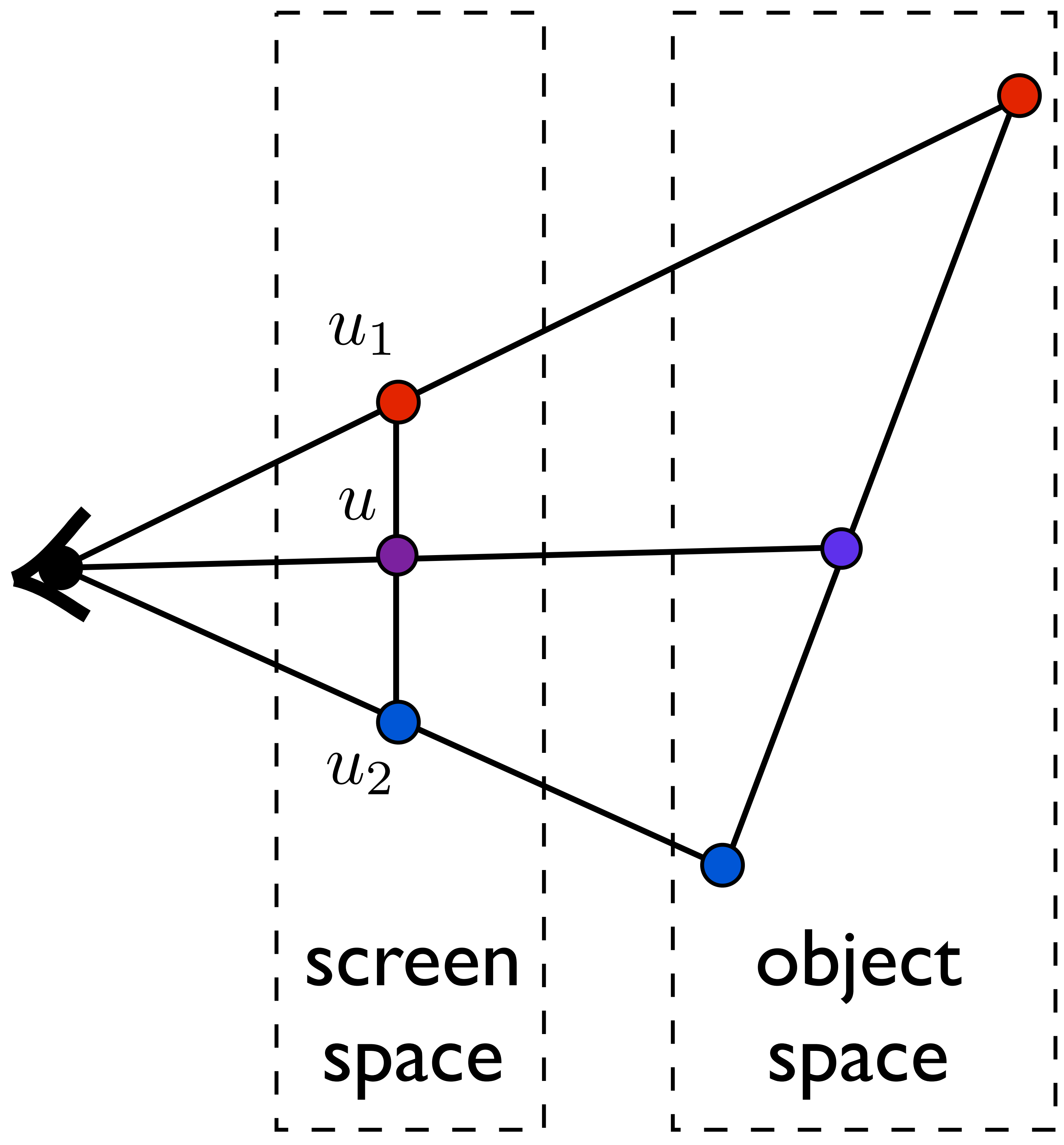
# Perspective correct interpolation

# Perspective correct interpolation

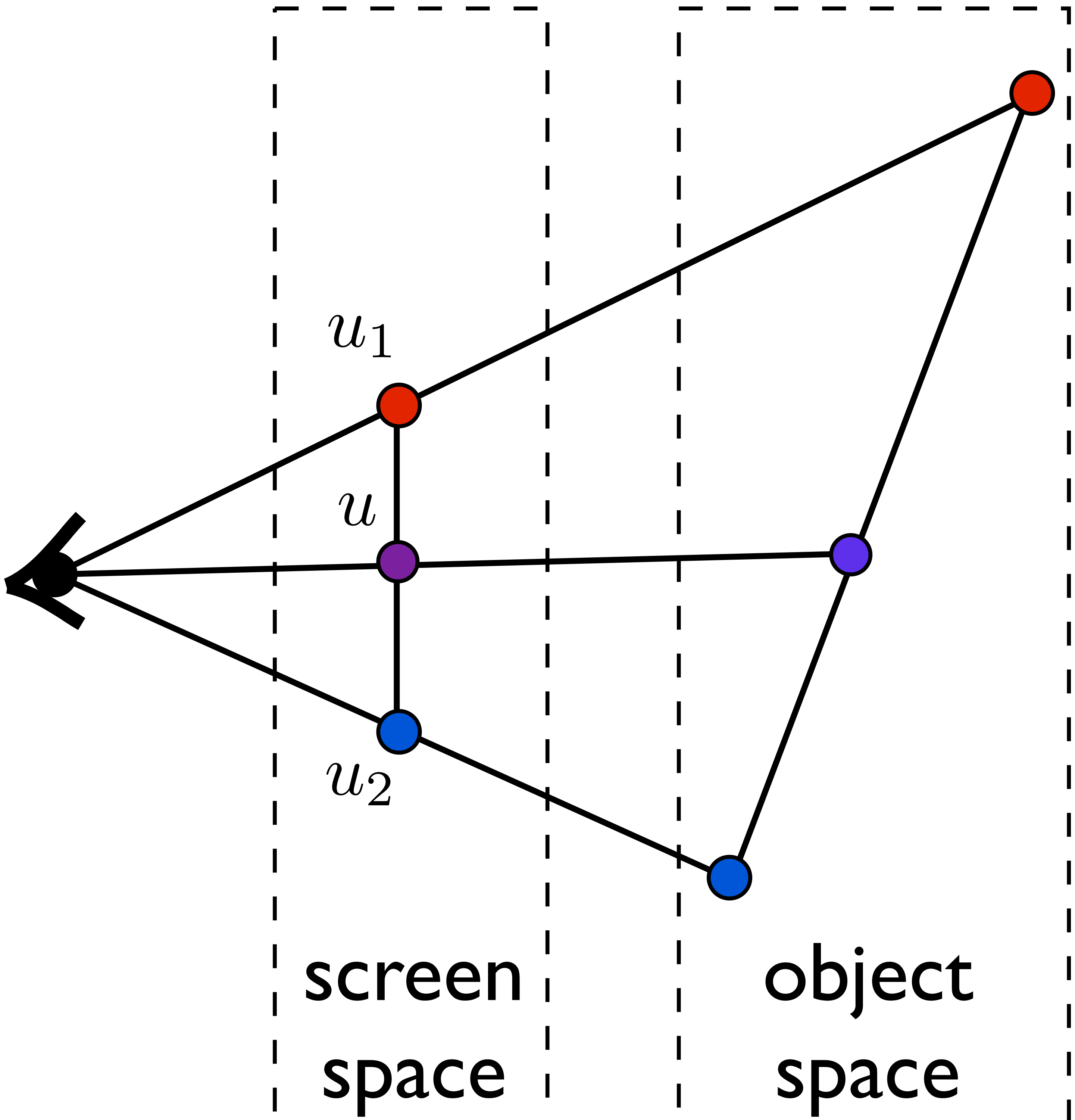
- In pipeline, we find barycentric coordinates in 2D screen space
  - but not the correct object space barycentric coords
  - these coordinates are okay for z-buffer test



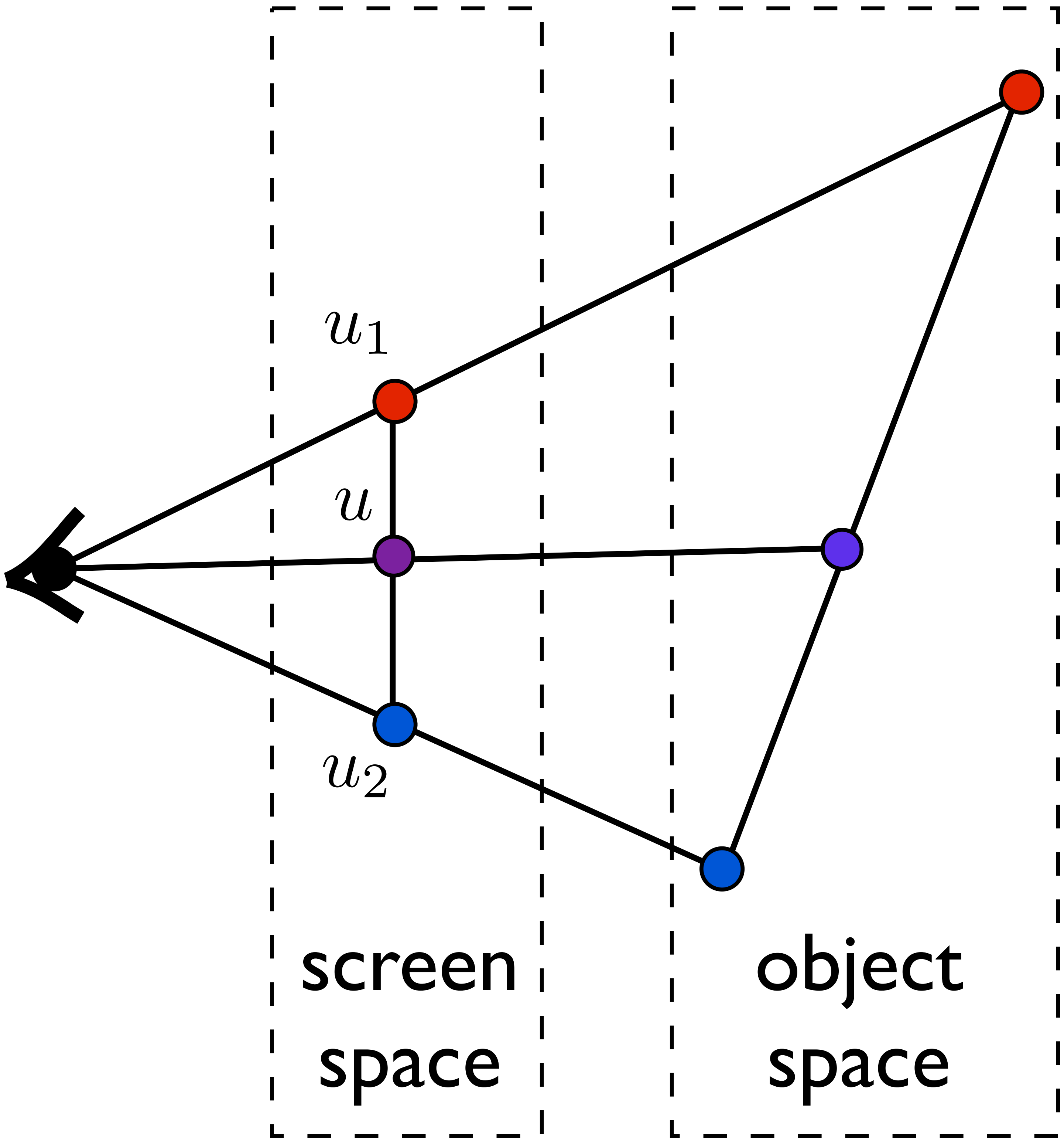





$$u = \frac{1}{2}u_1 + \frac{1}{2}u_2$$

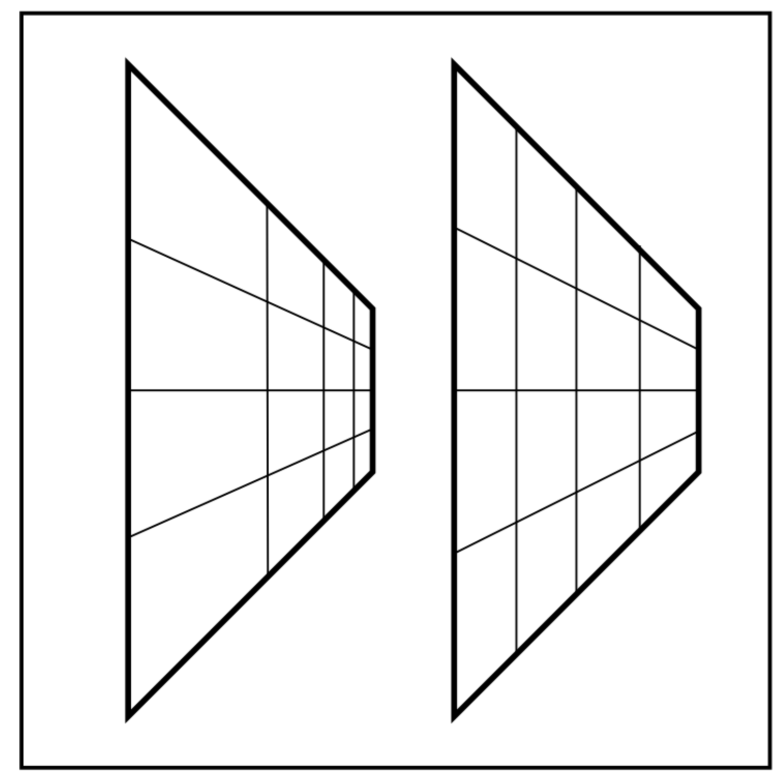



$$\cancel{u = \frac{1}{2}u_1 + \frac{1}{2}u_2}$$




# Interpolation with screen space weights is incorrect

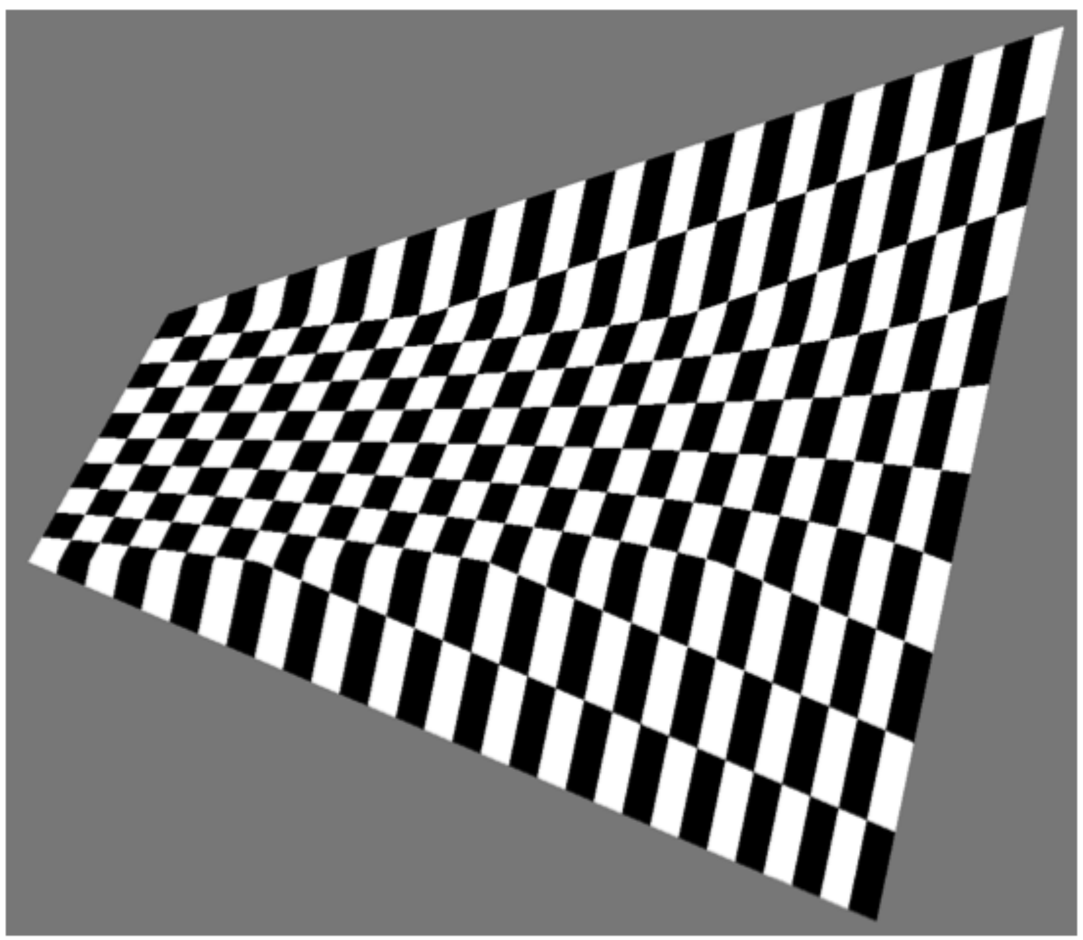
$$\cancel{u = \frac{1}{2}u_1 + \frac{1}{2}u_2}$$

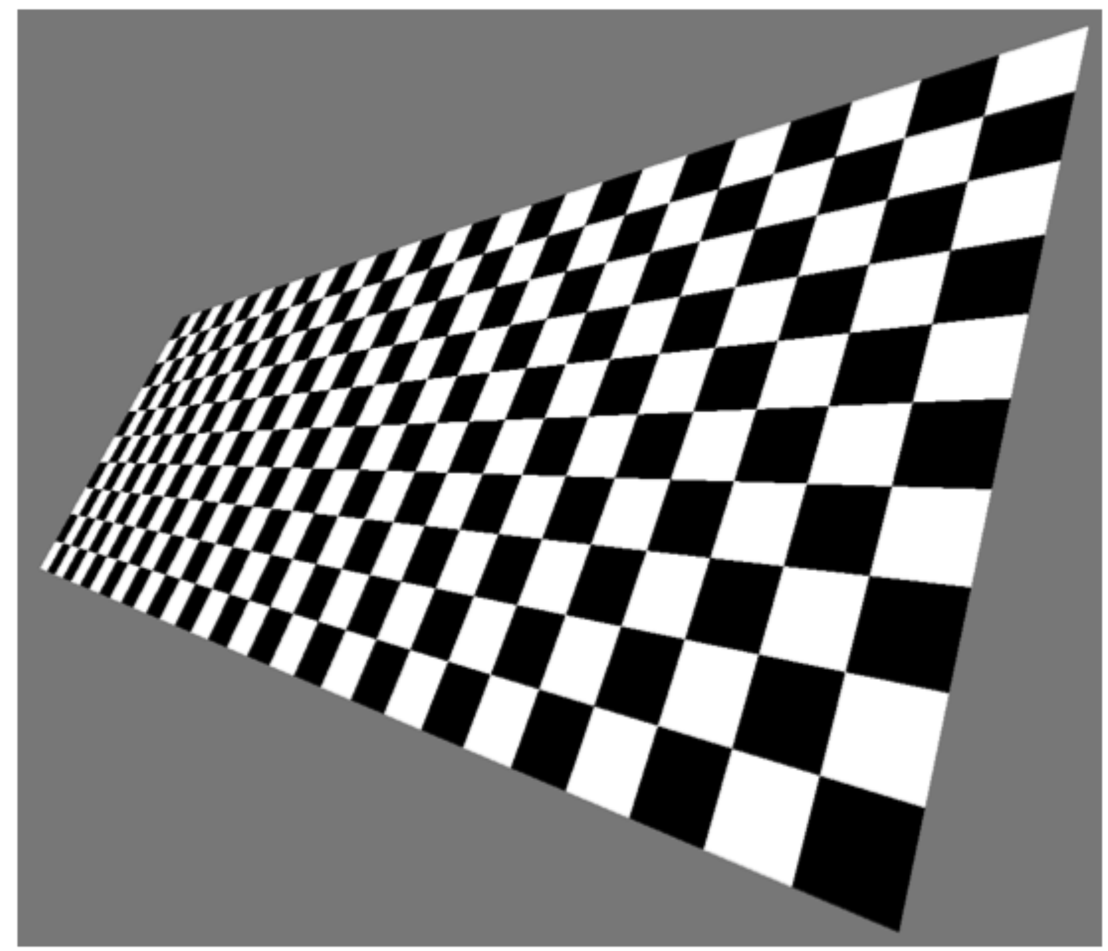


correct distorted

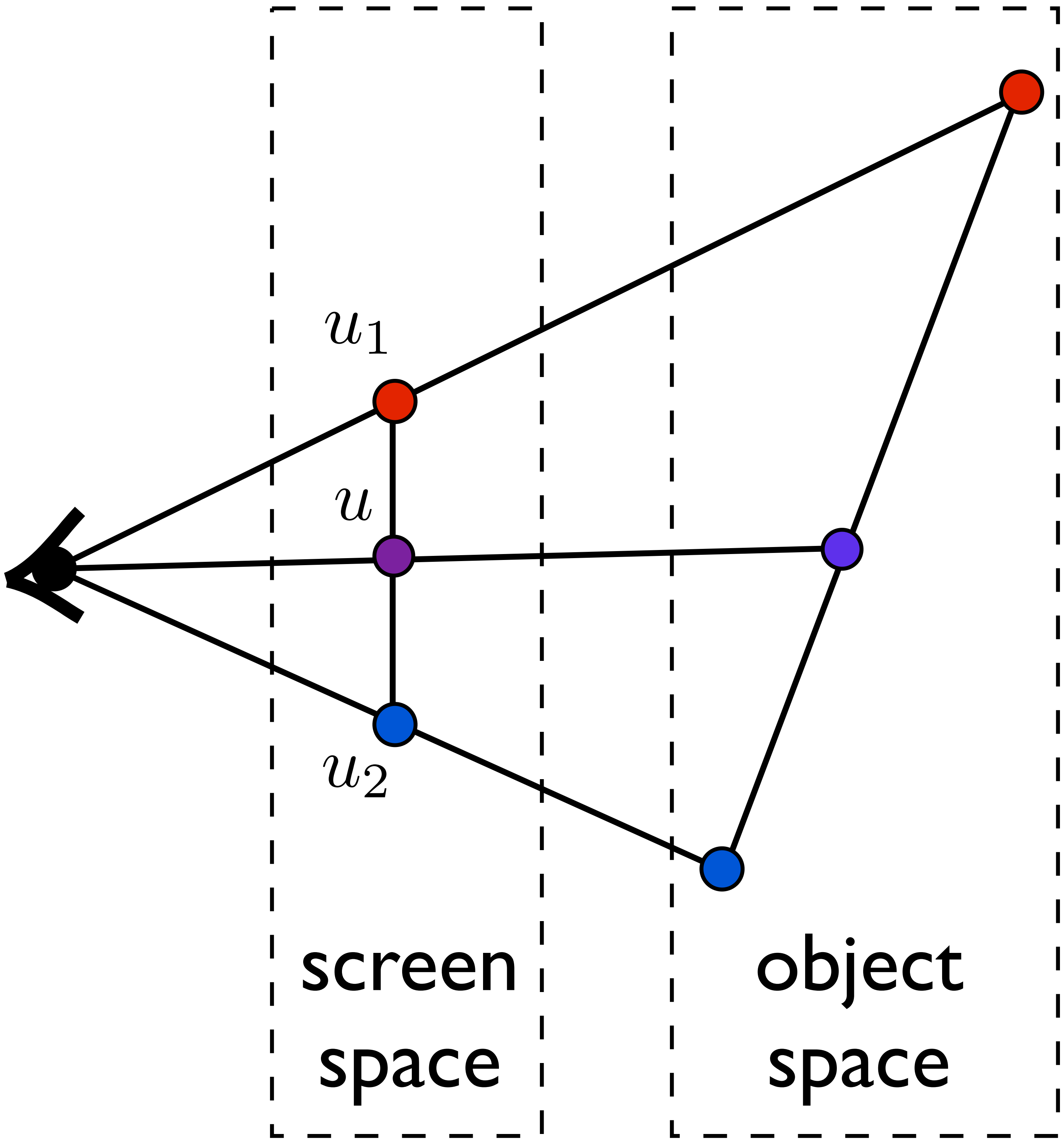
# Perspective correct interpolation

Using screen space weights looks wrong for textures





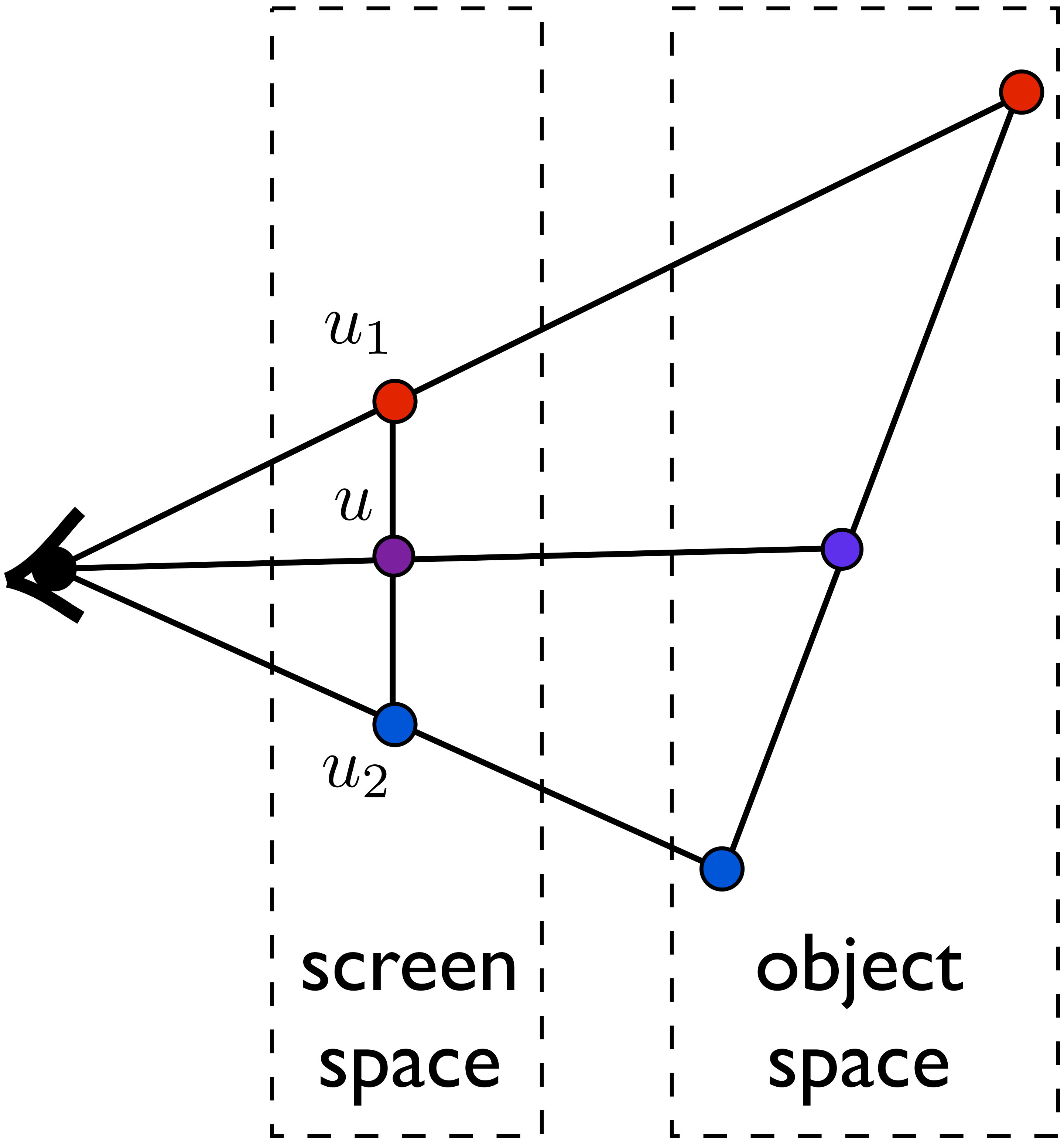
[Heckbert and Morton, 1990]




$$\cancel{u = \frac{1}{2}u_1 + \frac{1}{2}u_2}$$

Do we need to transform back to object space?

$$\mathbf{v}_{\mathrm{sc}} = M_{\mathrm{vp}} M_{\mathrm{pers}} M_{\mathrm{cam}} \mathbf{v}$$




$$u = \cancel{\frac{1}{2}u_1 + \frac{1}{2}u_2}$$

Do we need to transform back to object space?

NO!

<whiteboard>