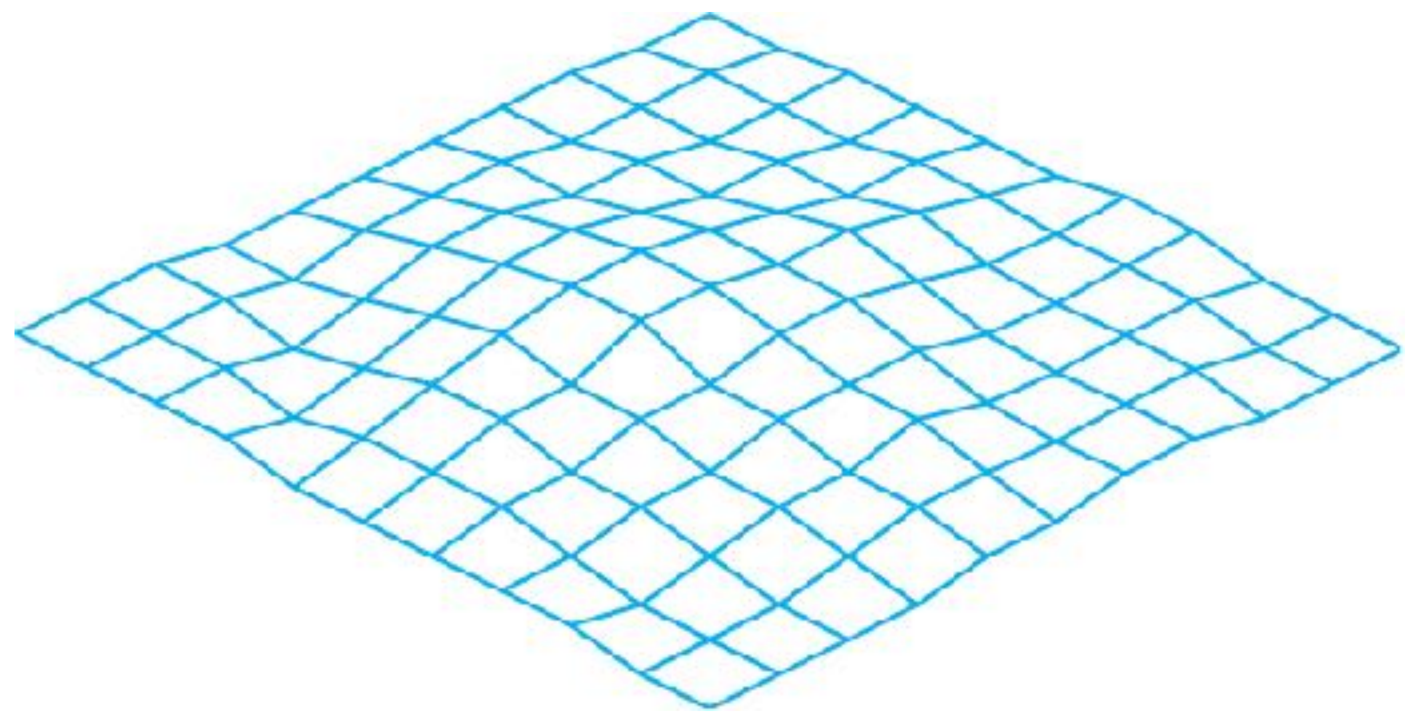
Shading Polygonal Geometry

Smooth surfaces are often approximated by polygons

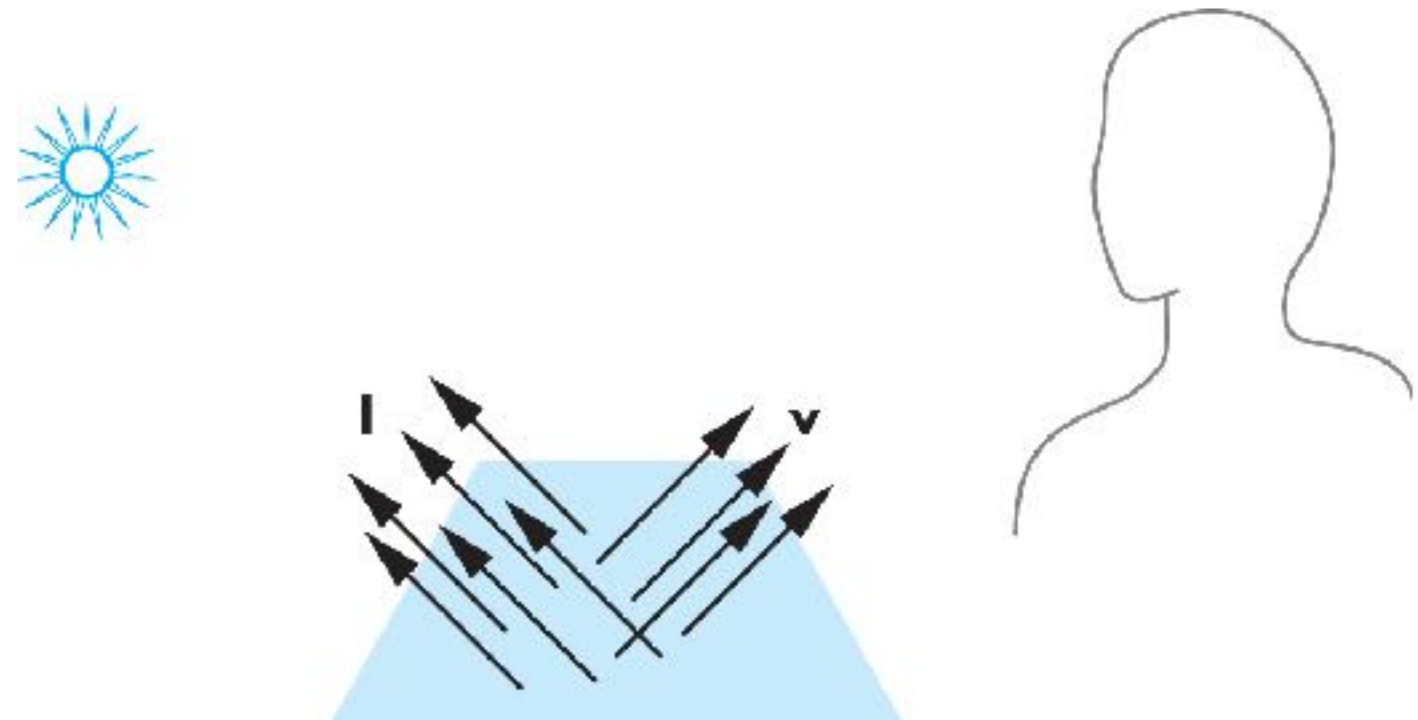
Shading approaches:

1. Flat
2. Smooth (Gouraud)
3. Phong



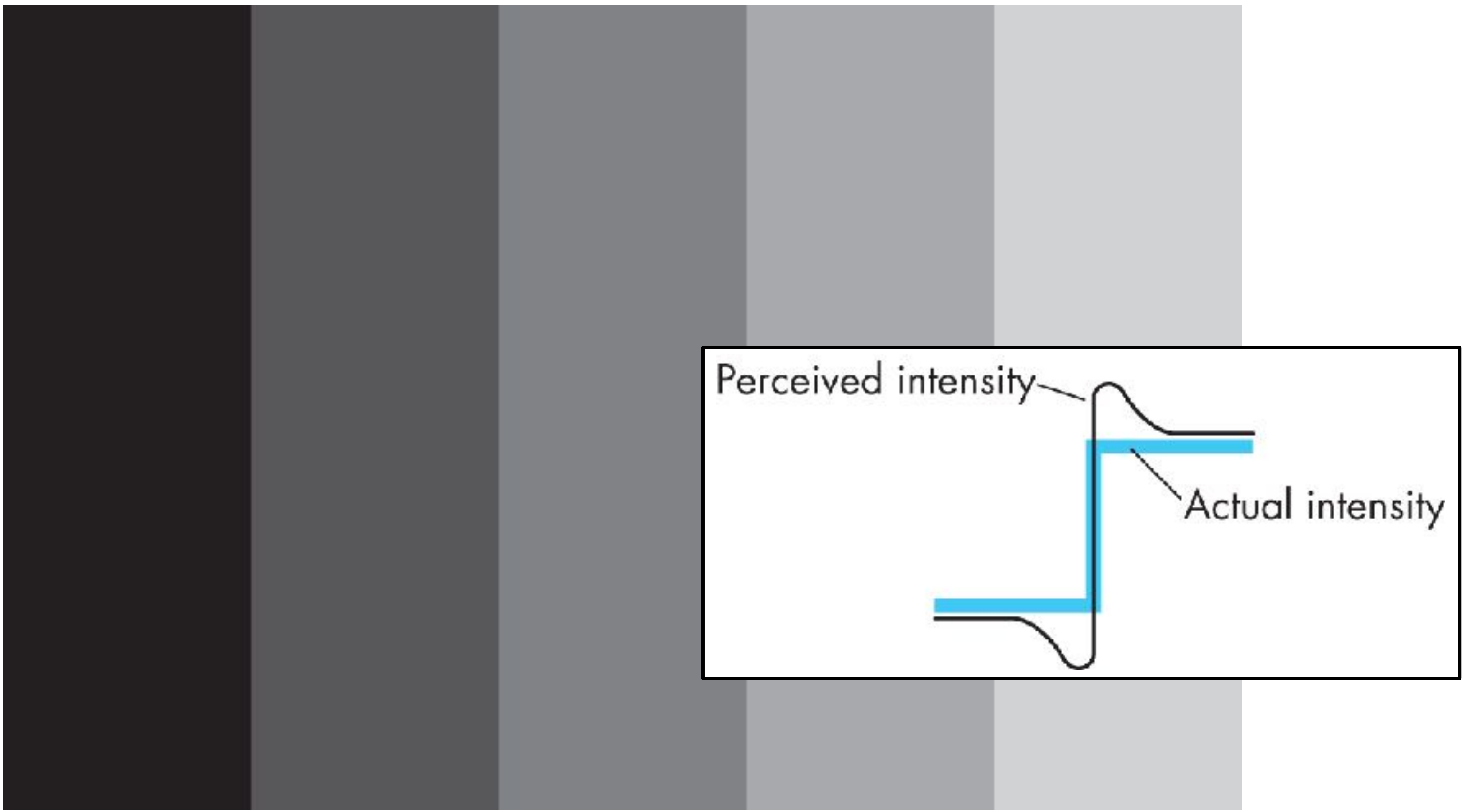
Flat Shading

do the shading calculation once per **polygon**



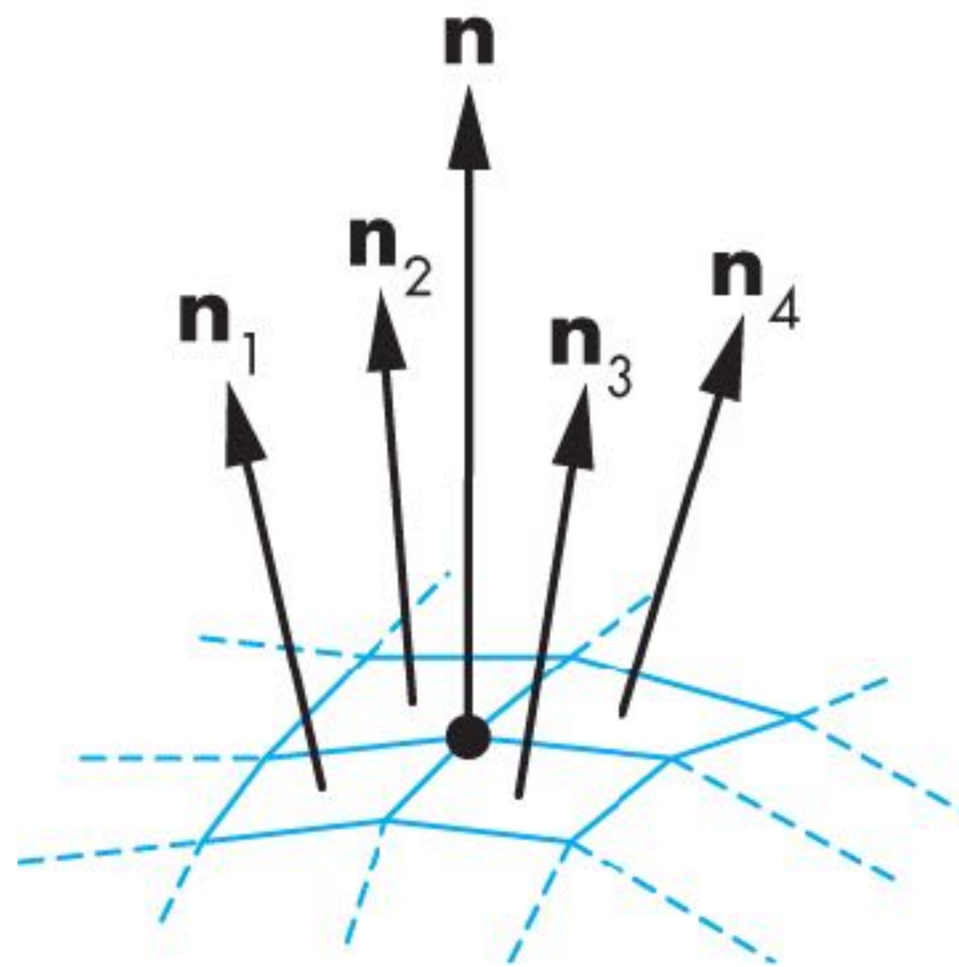
valid for light at ∞
and viewer at ∞
and faceted surfaces

Mach Band Effect



Smooth Shading

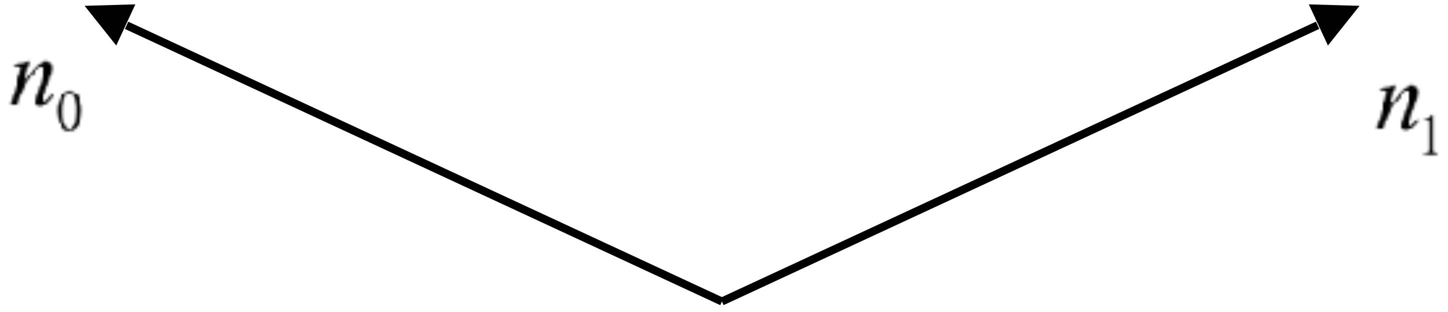
$$\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{||\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4||}$$



do the shading calculation once per **vertex**

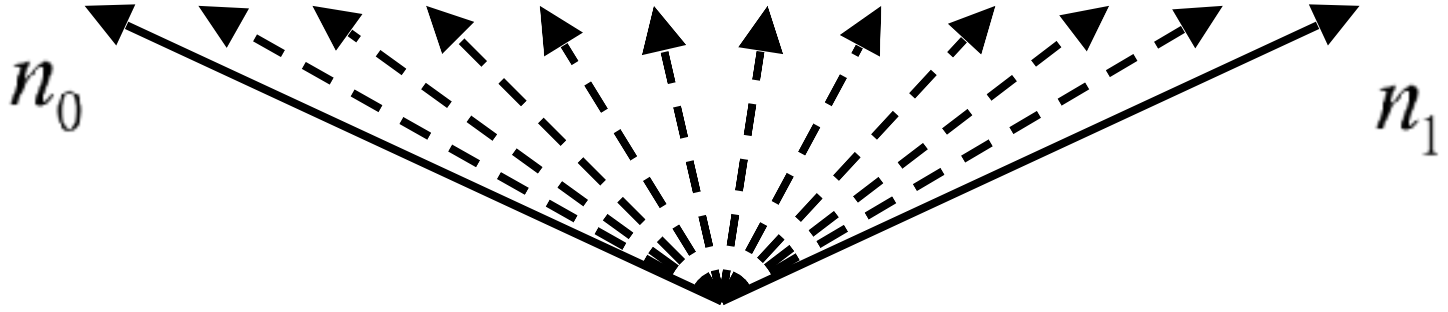
Interpolating Normals

- Must renormalize



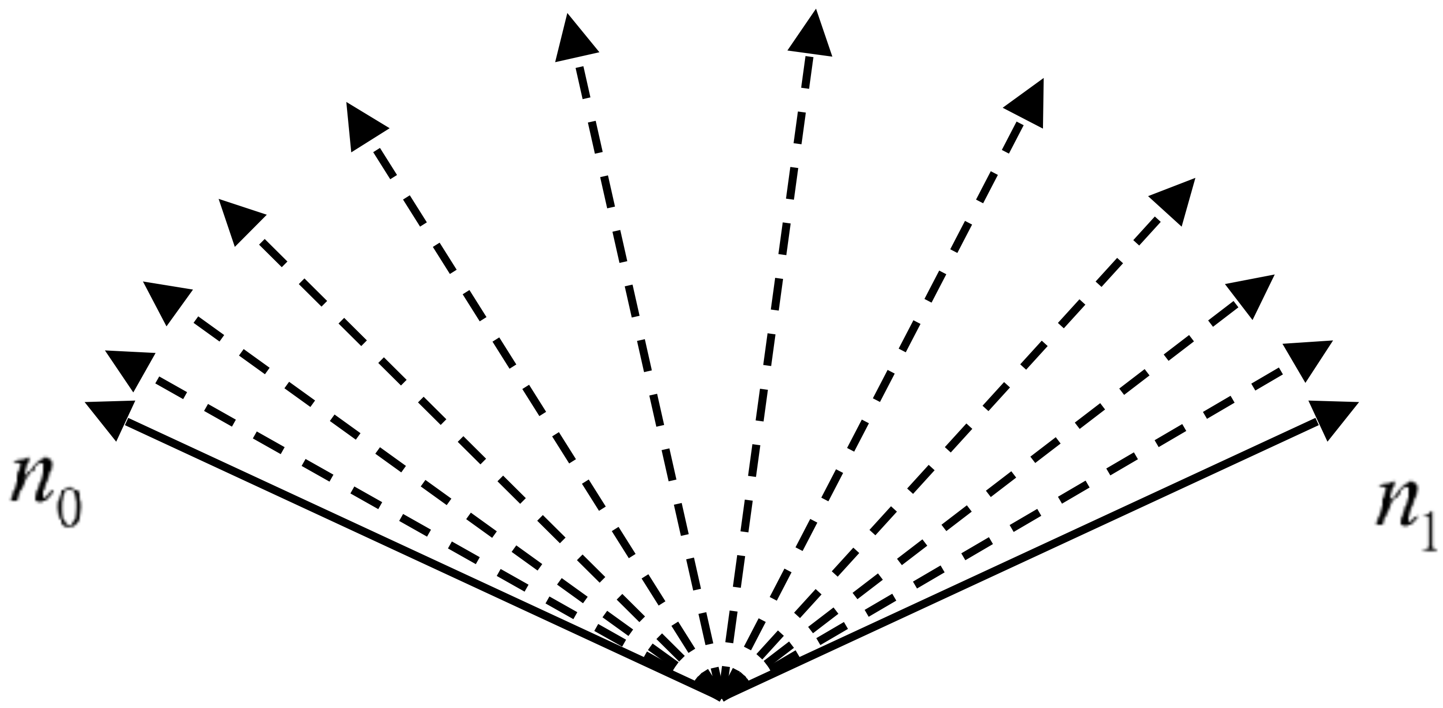
Interpolating Normals

- Must renormalize

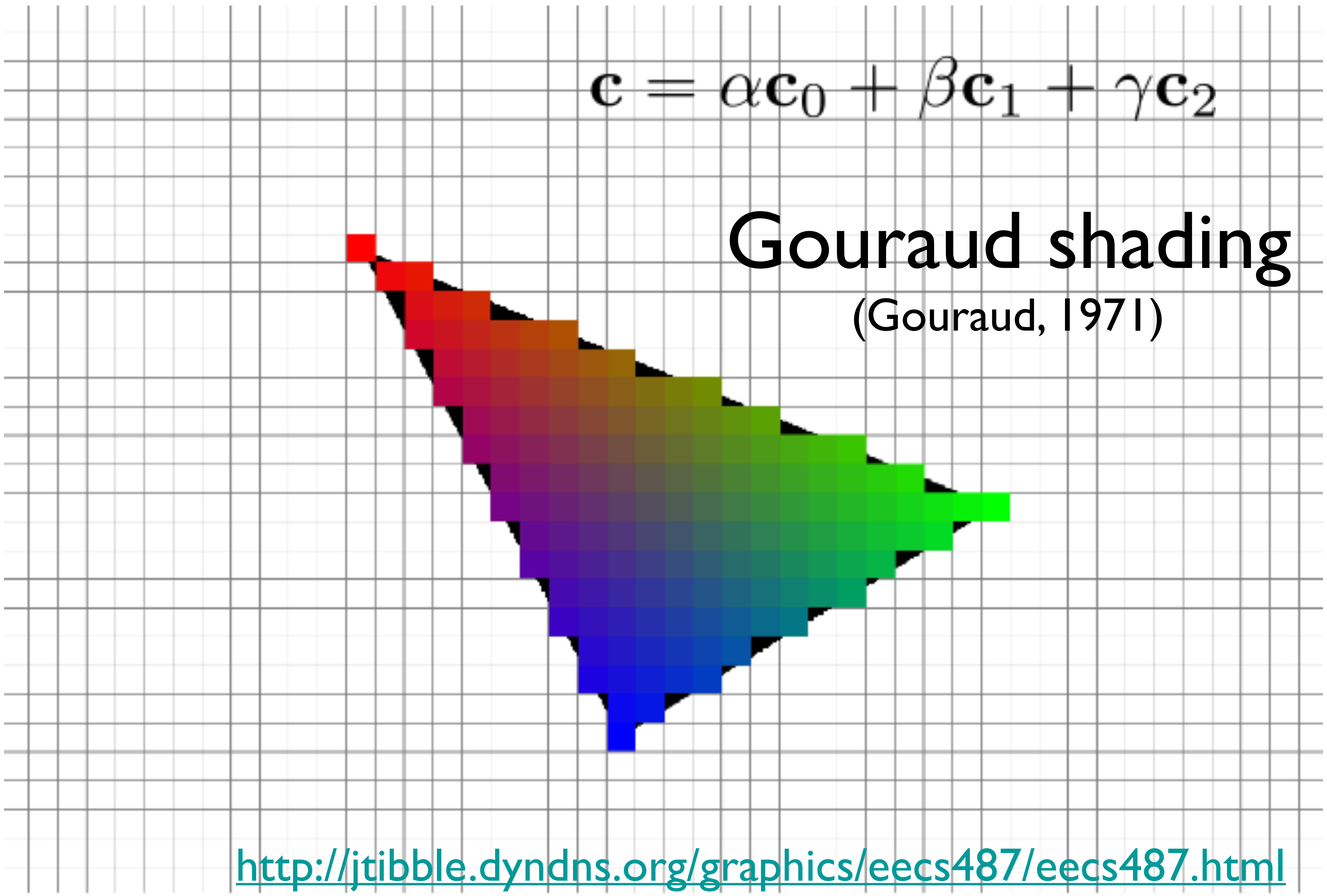


Interpolating Normals

- Must renormalize



We can interpolate attributes using barycentric coordinates



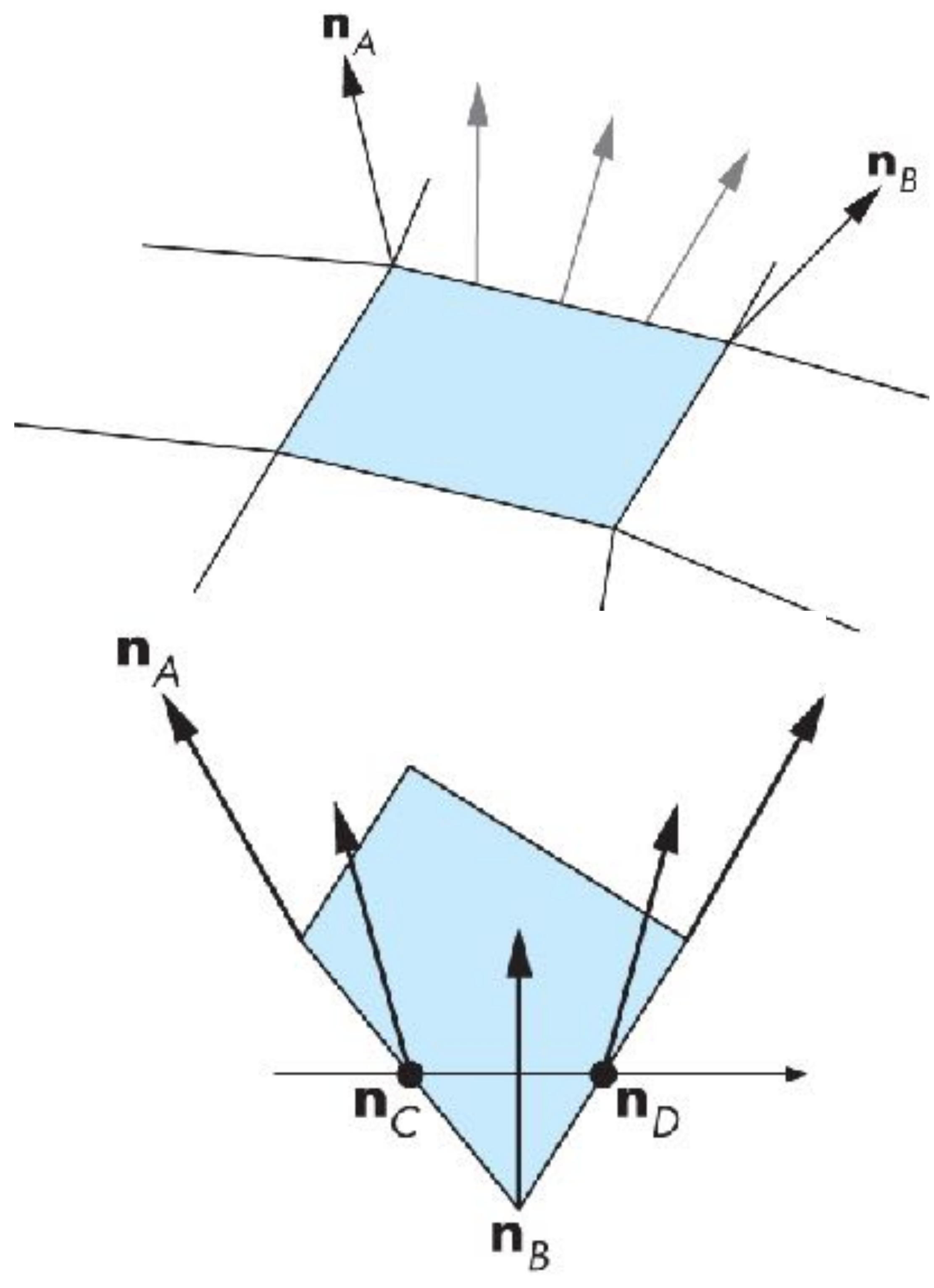
$$\mathbf{c} = \alpha \mathbf{c}_0 + \beta \mathbf{c}_1 + \gamma \mathbf{c}_2$$

Gouraud shading

(Gouraud, 1971)

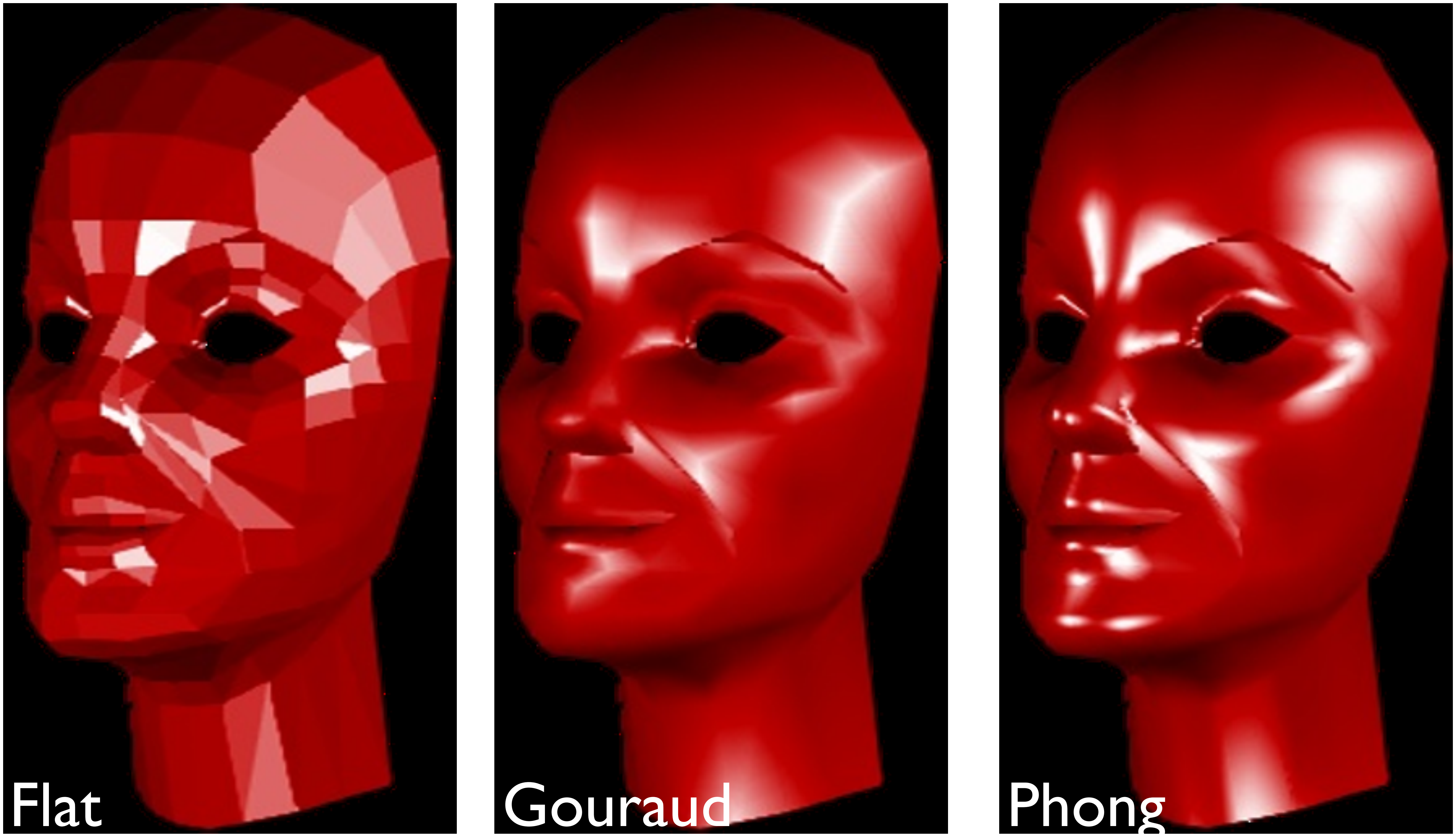
http://jtibble.dyndns.org/graphics/eecs487/eecs487.html

Phong Shading

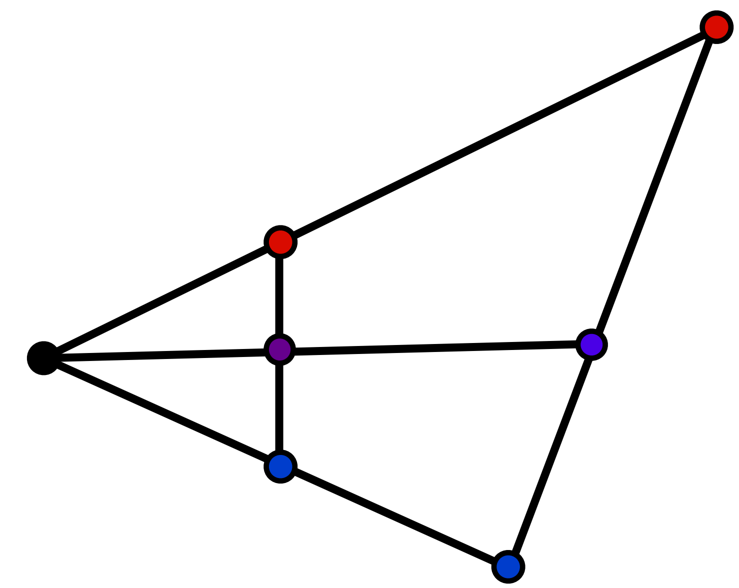


do the shading calculation once per **fragment**

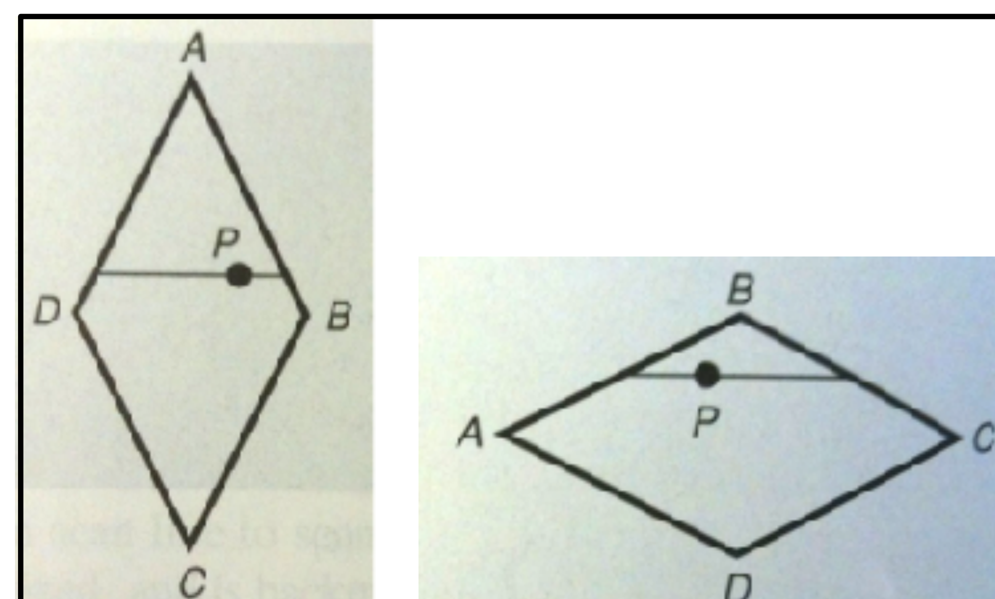
Comparison

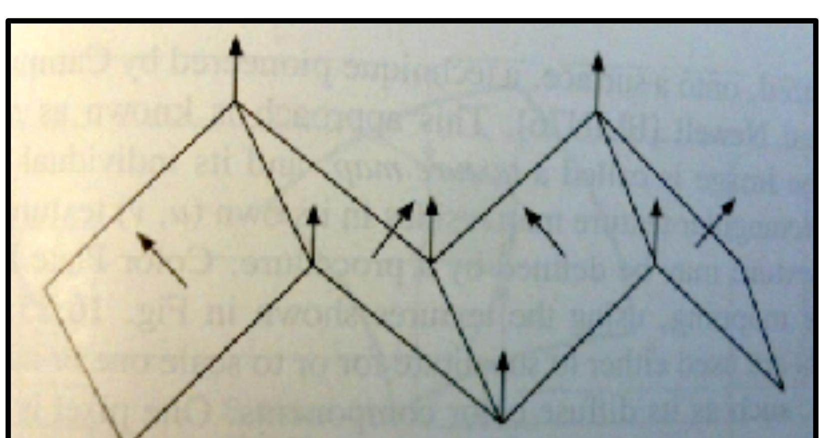


Problems with Interpolated Shading



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

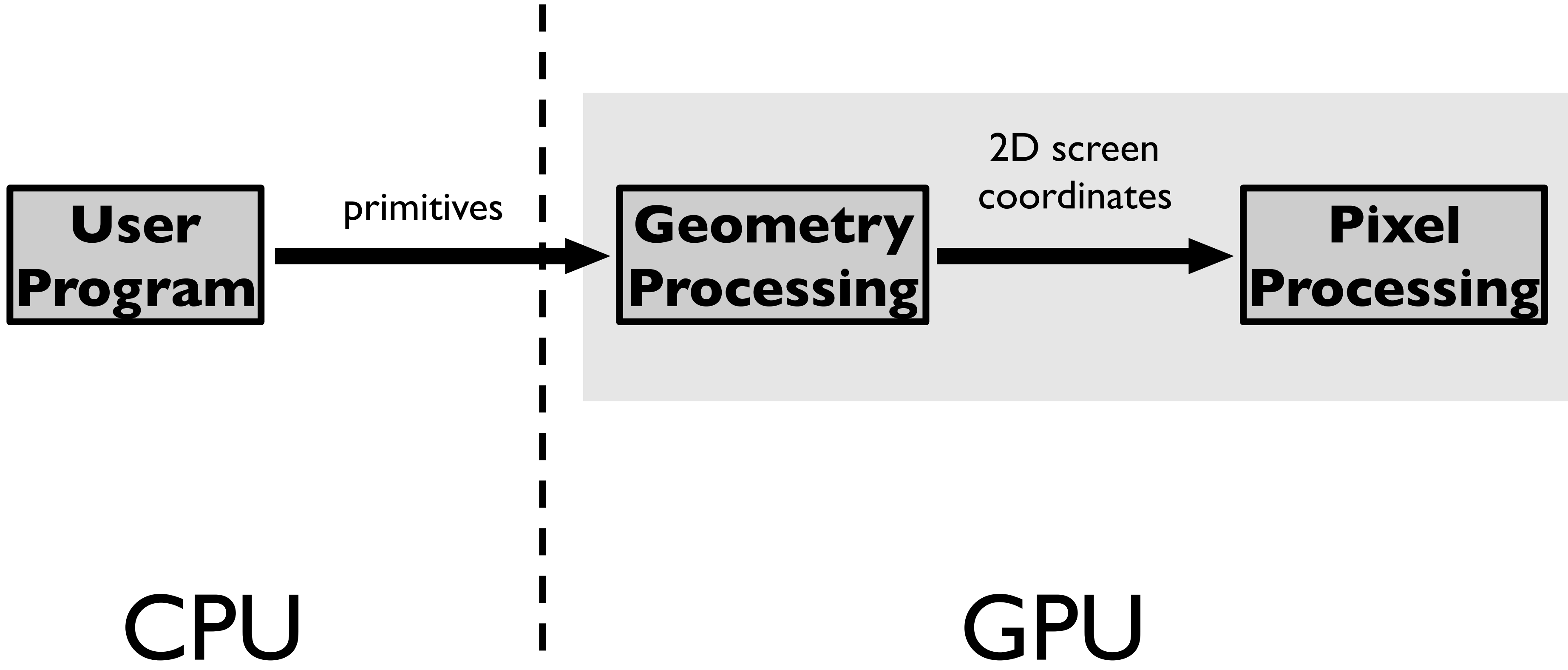




[Foley, van Dam, Feiner, Hughes]

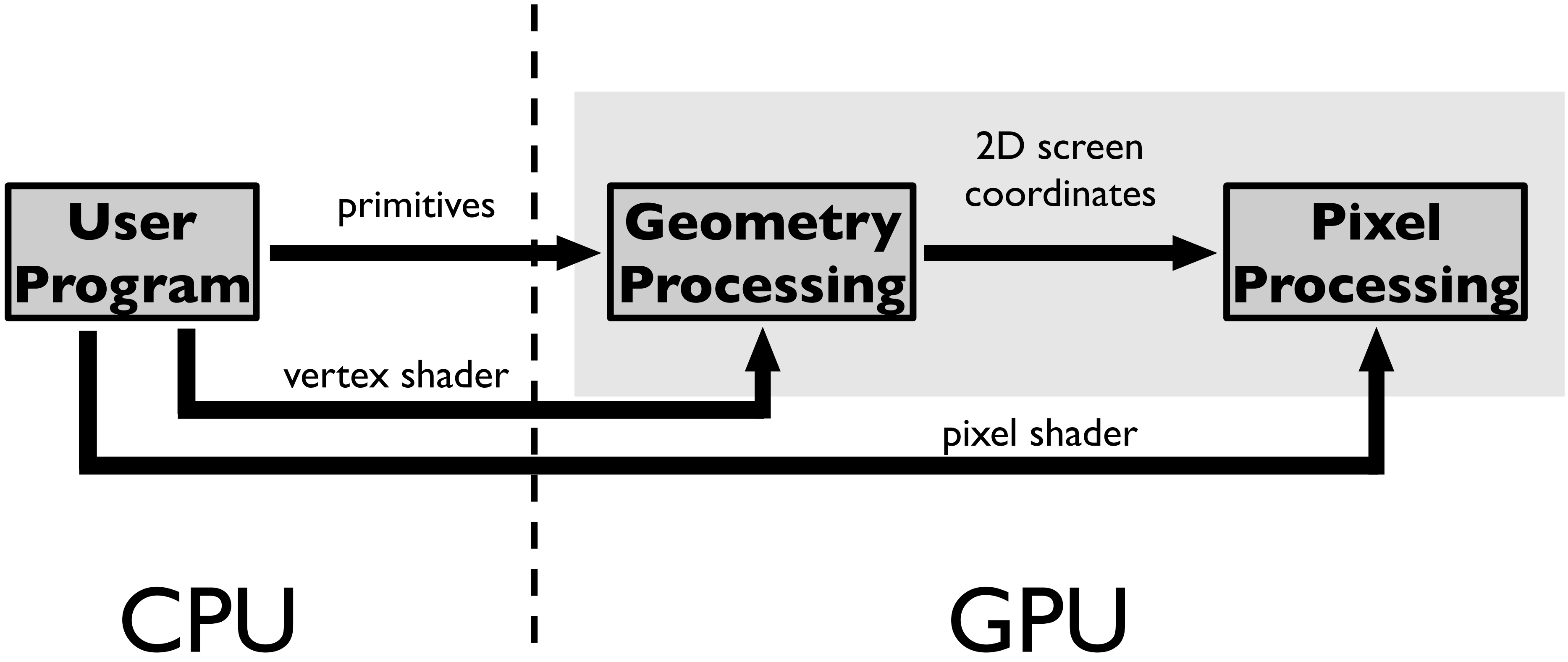
Programmable Shading

Fixed-Function Pipeline



Control pipeline through GL state variables

Programmable Pipeline



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

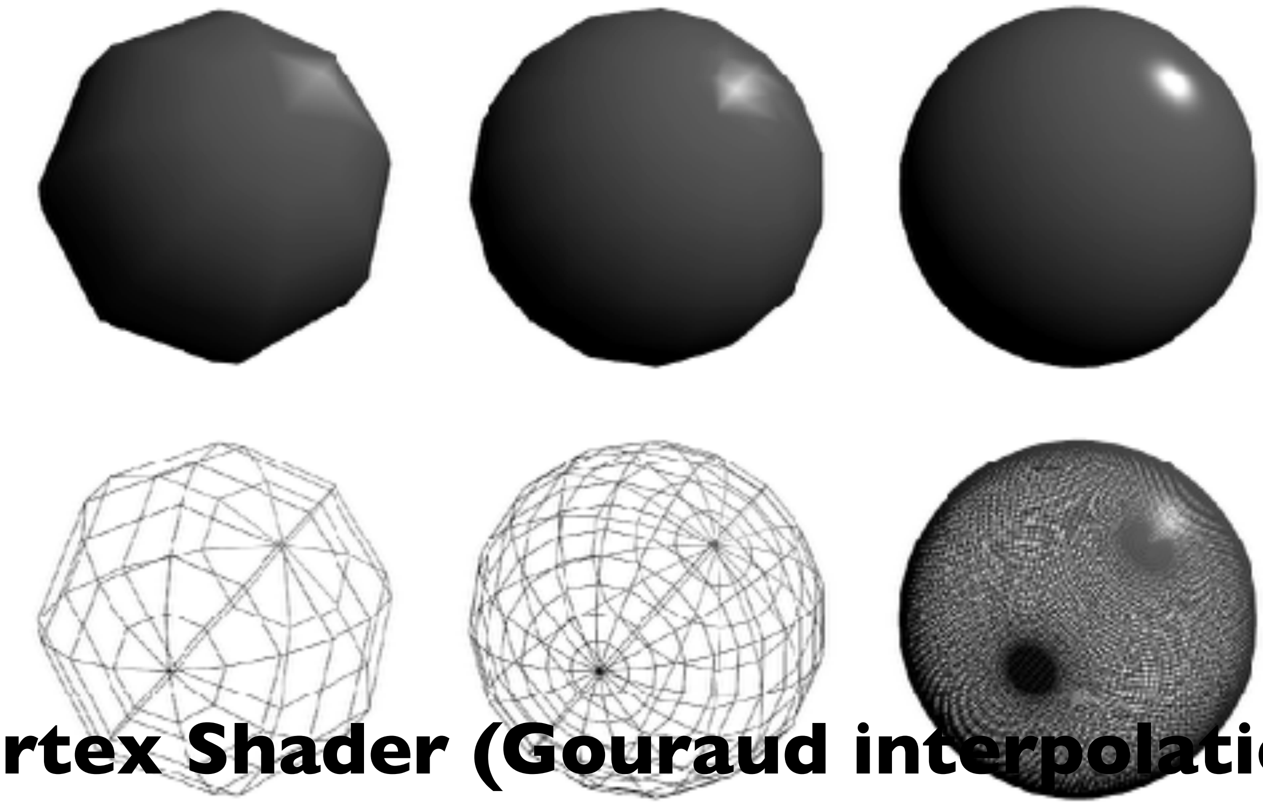
Phong reflectance in vertex and pixel shaders using GLSL

```
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();
}
```



Vertex Shader (Gouraud interpolation)

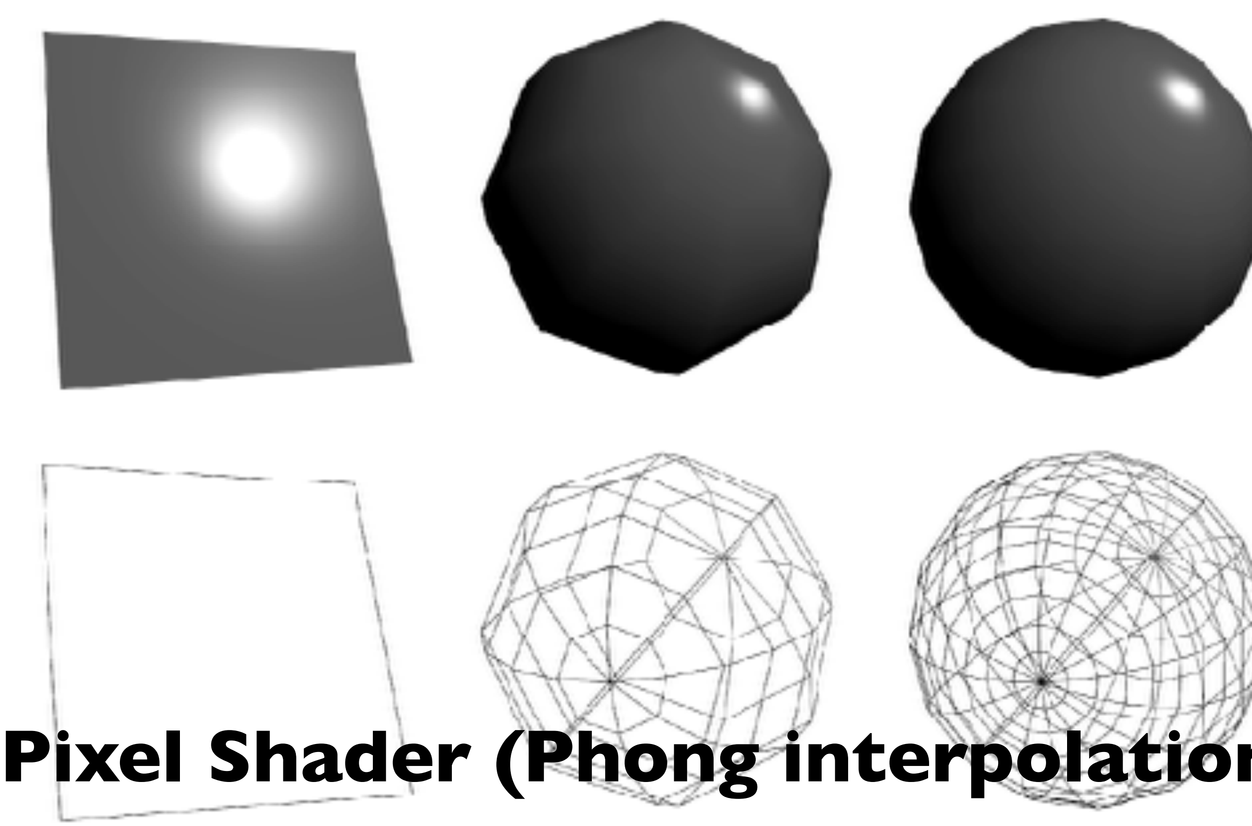
```
varying vec4 v;
varying vec3 n;

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;
}
```

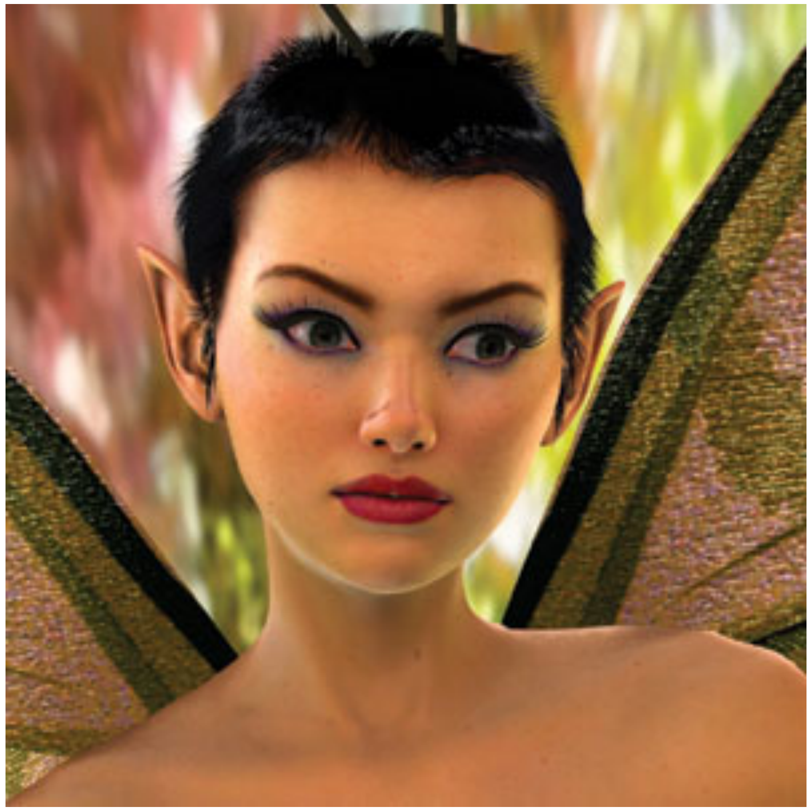


Pixel Shader (Phong interpolation)

[Shirley and Marschner]

Rusty car shader, NVIDIA

Call of Juarez DX10 Benchmark, ATI

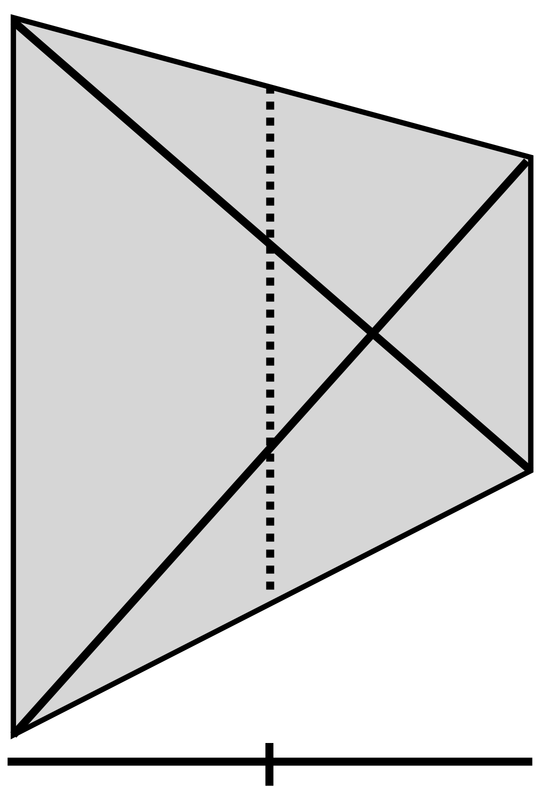


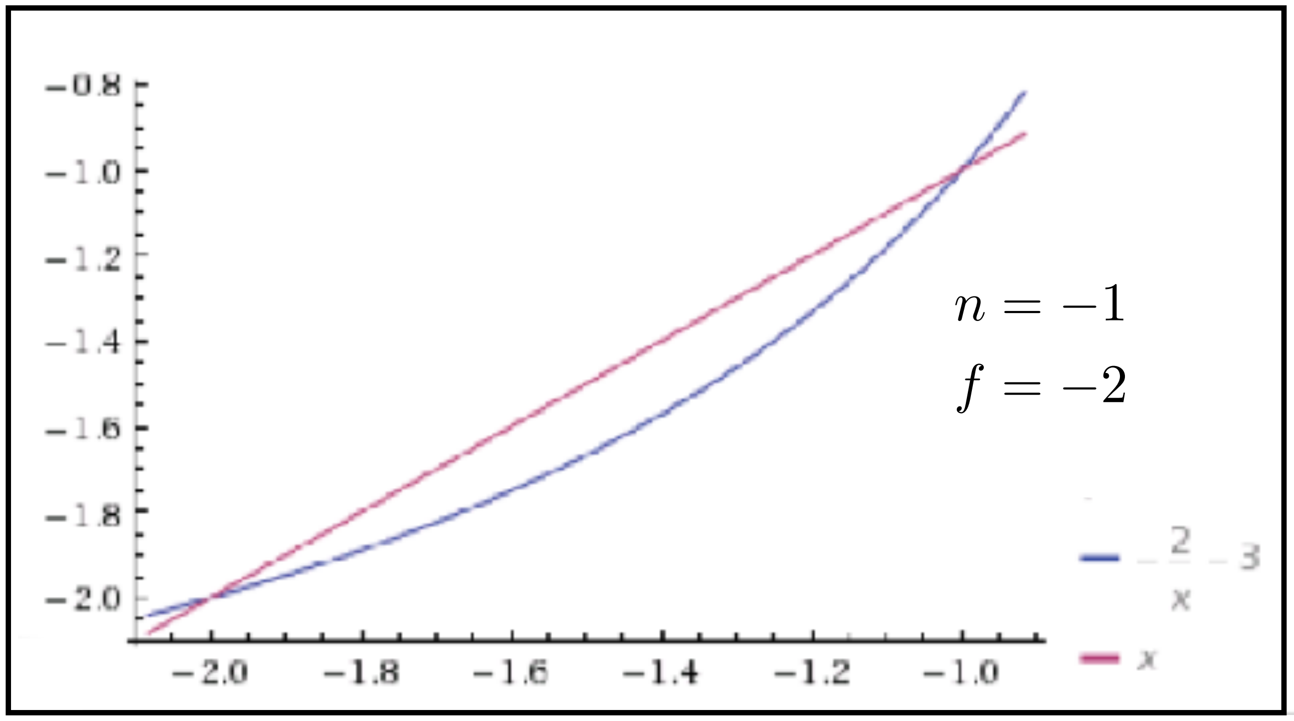
Dawn, NVIDIA

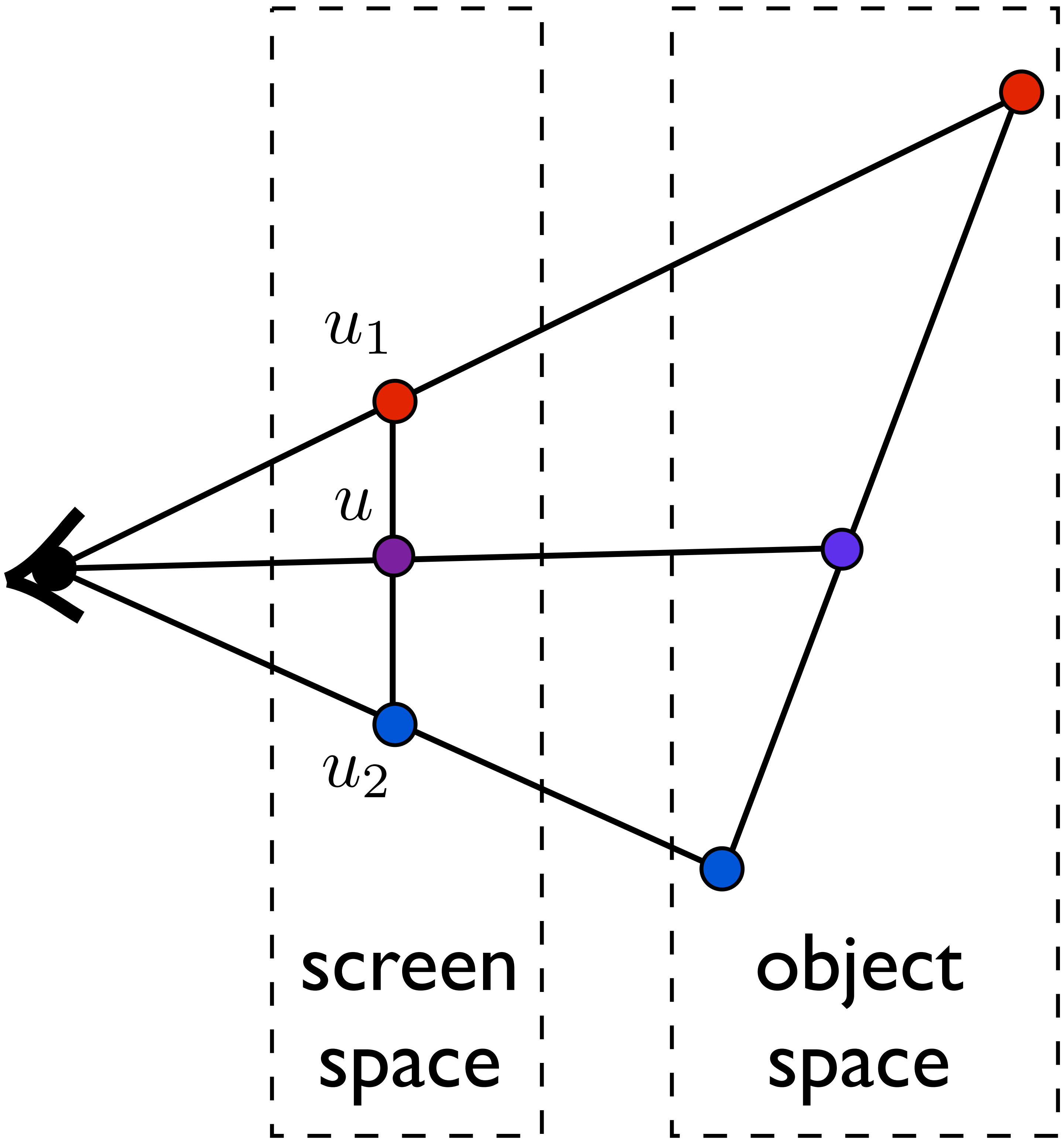
Perspective correct interpolation

Perspective correct interpolation

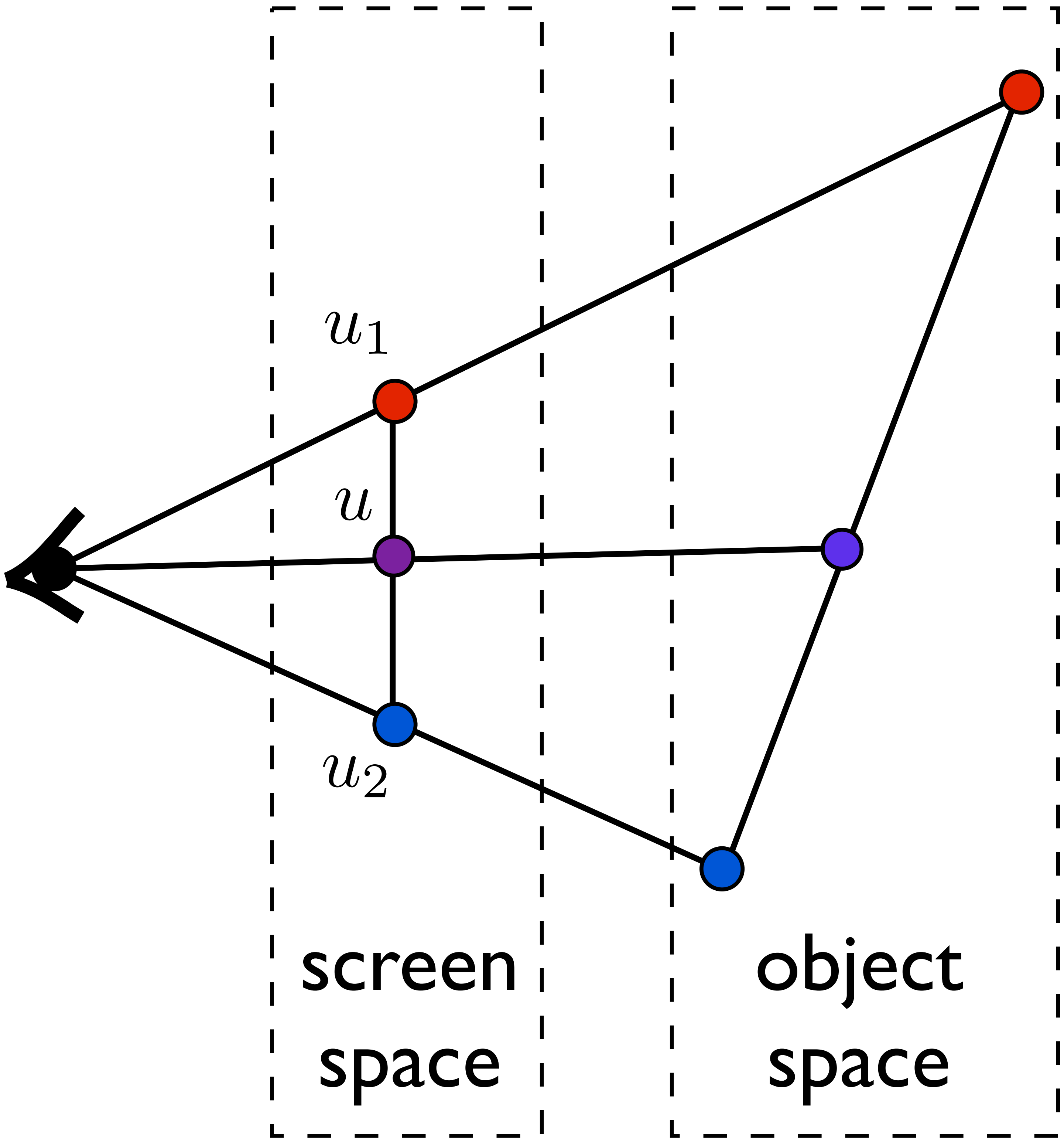
- In minigl pipeline, we find barycentric coordinates in 2D screen space
  - but not the correct object space barycentric coords
  - these coordinates are okay for z-buffer test





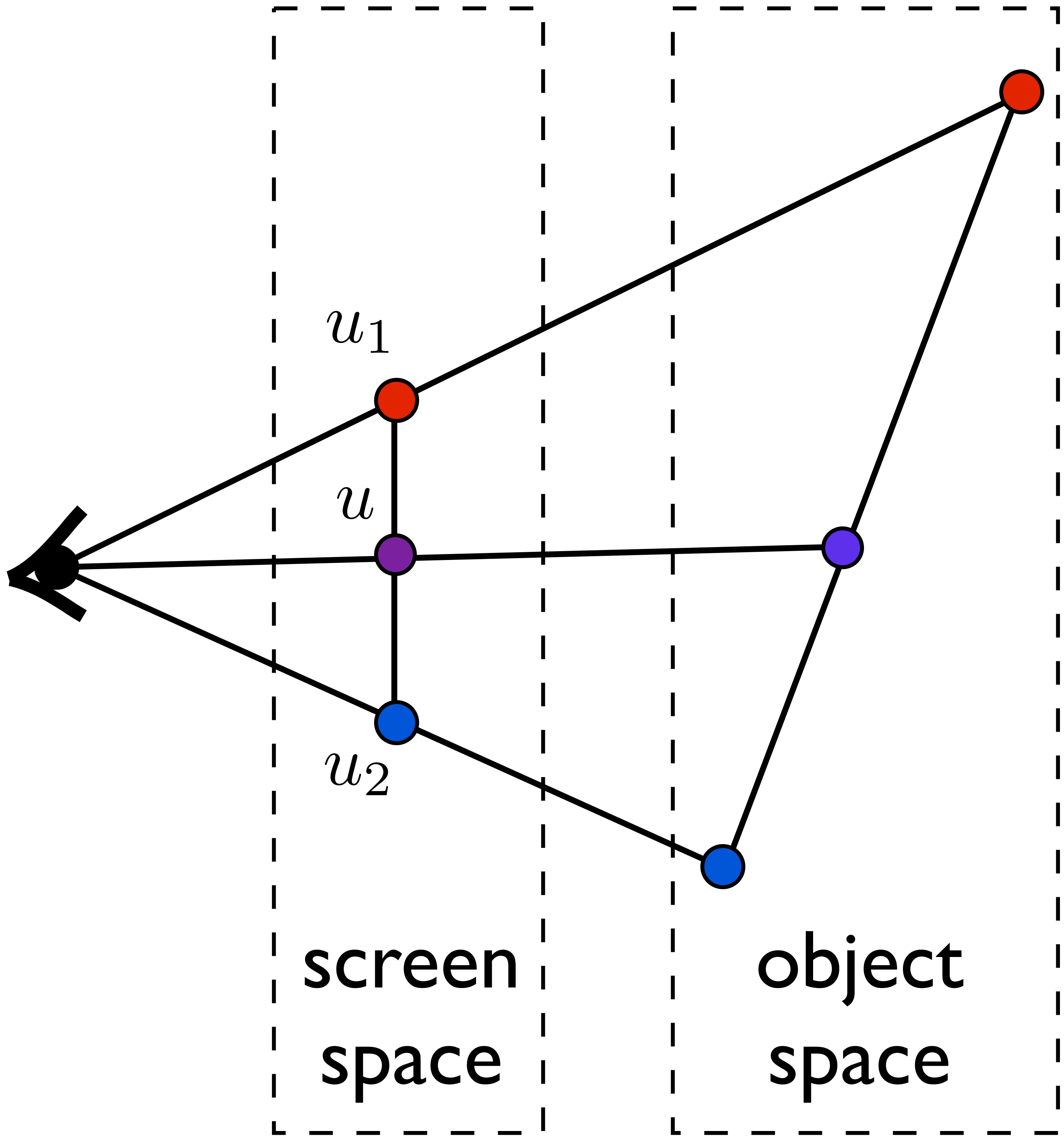


$$u = \frac{1}{2}u_1 + \frac{1}{2}u_2$$

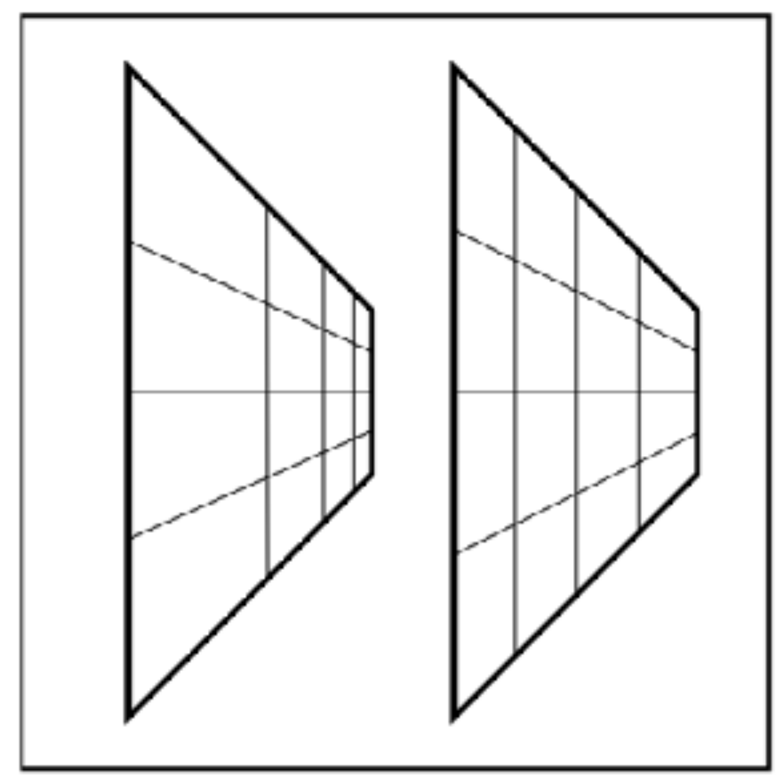


$$u = \frac{1}{2}u_1 + \frac{1}{2}u_2$$

Interpolation with screen space weights is incorrect



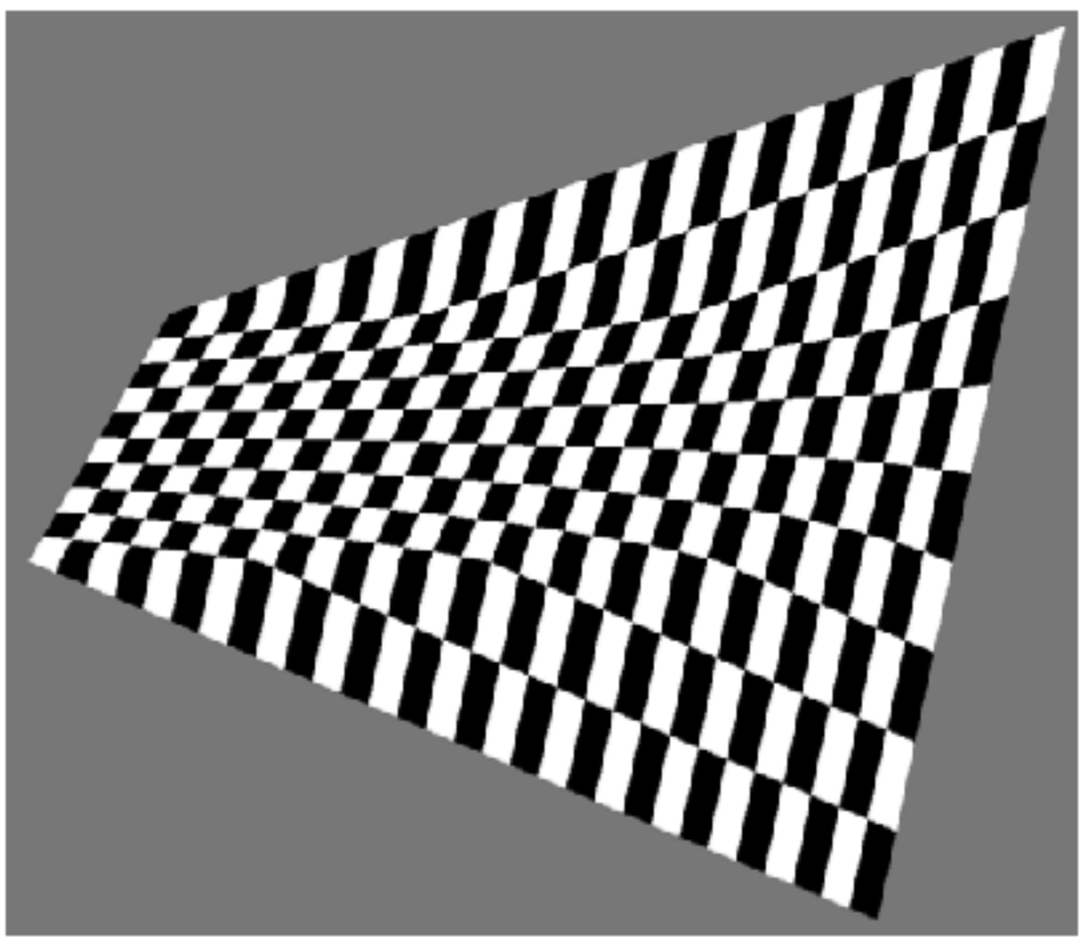
$$u = \frac{1}{2}u_1 + \frac{1}{2}u_2$$

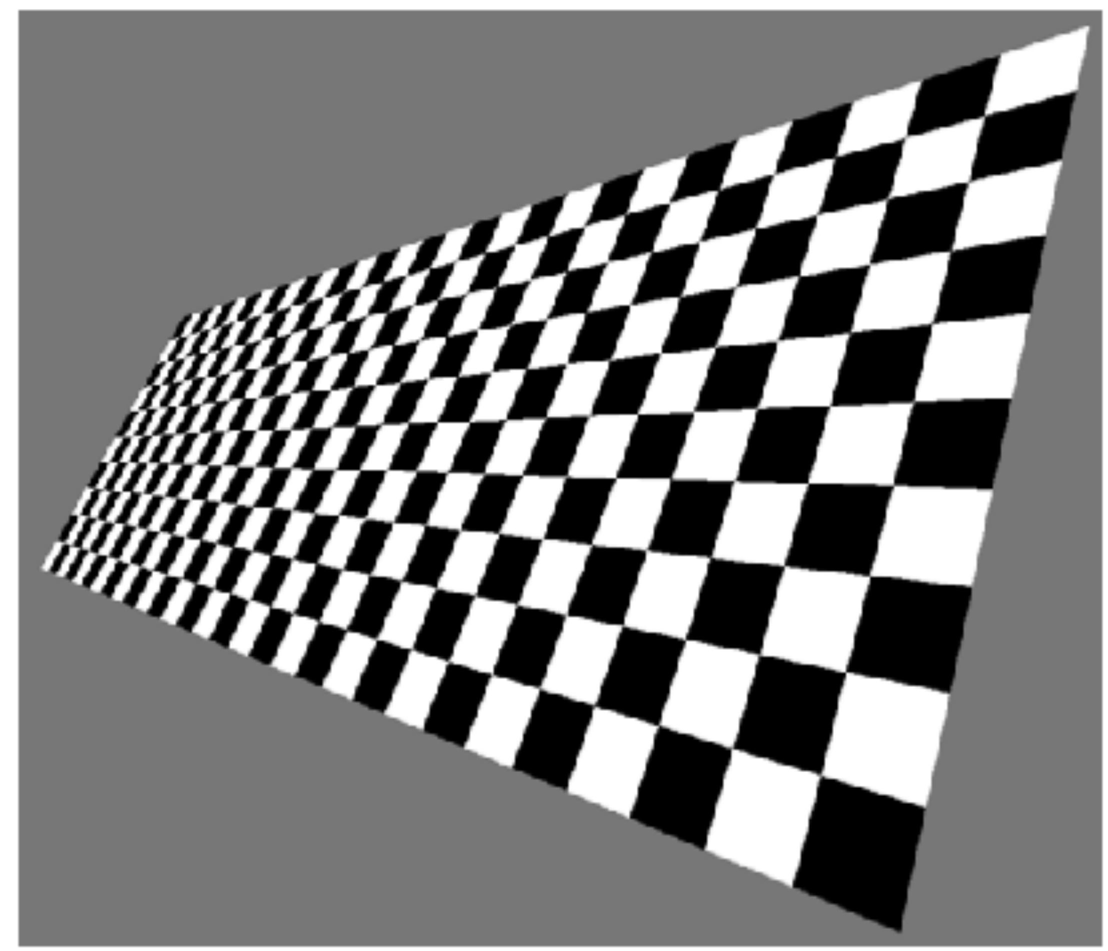


correct distorted

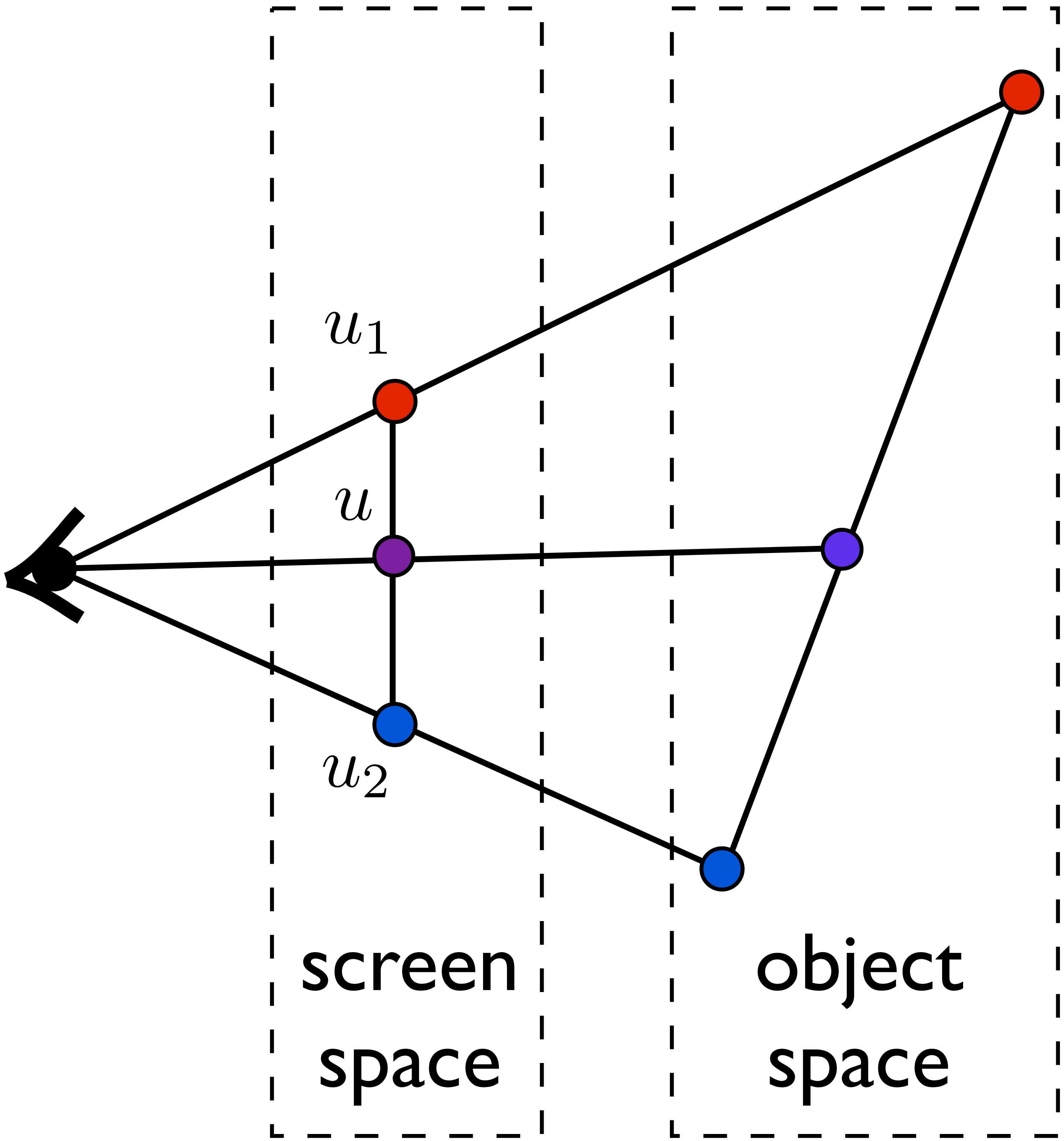
Perspective correct interpolation

Using screen space weights looks wrong for textures





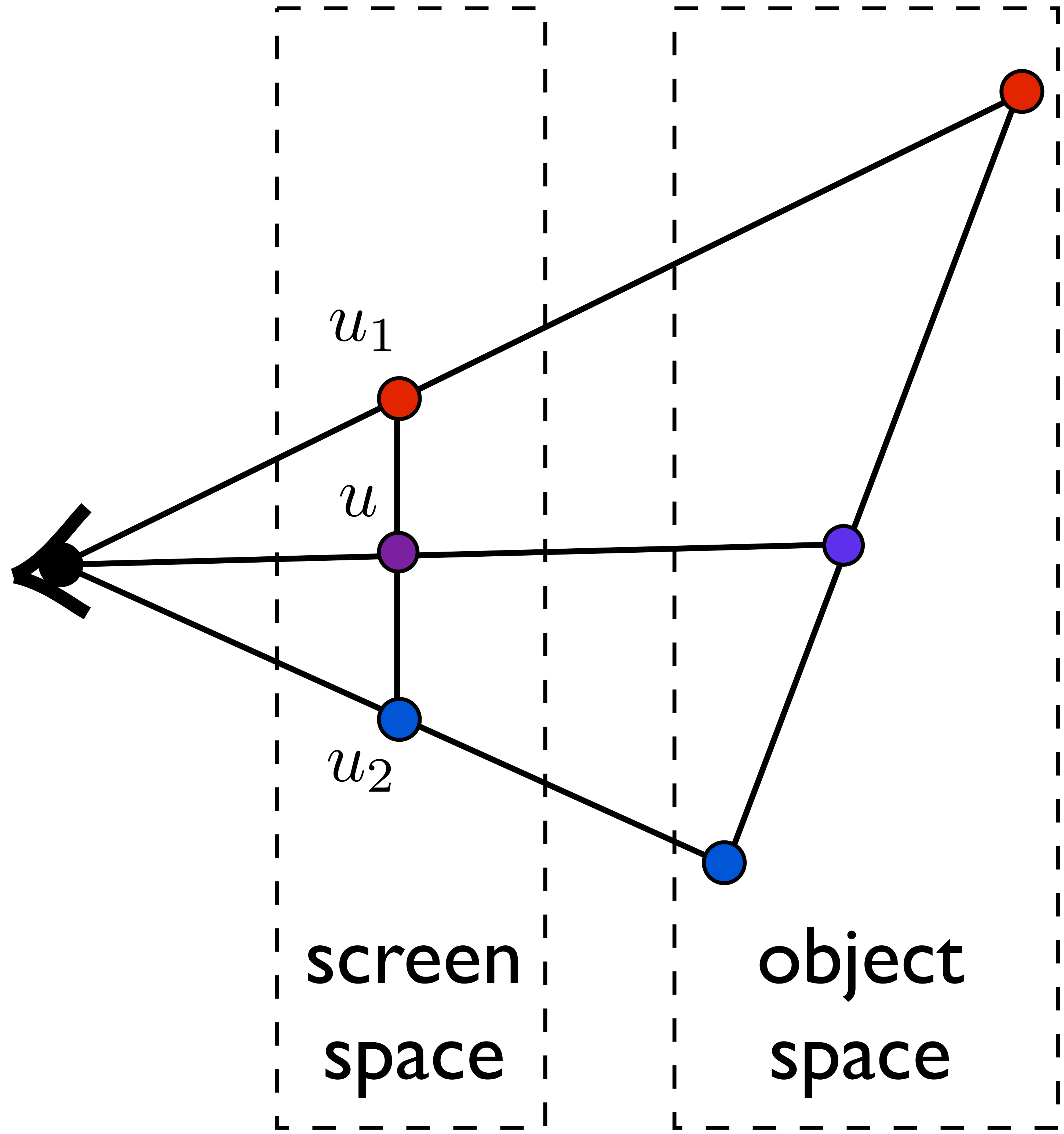
[Heckbert and Morton, 1990]



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

Do we need to transform back to object space?

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



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

Do we need to transform back to object space?

NO!

<whiteboard>