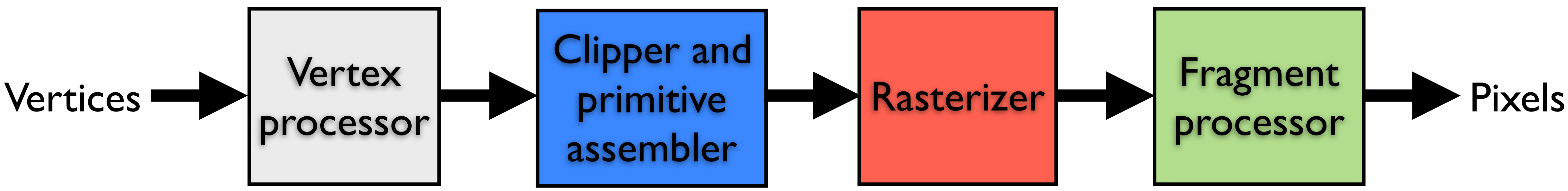
CS230 : Computer Graphics

Lecture 8

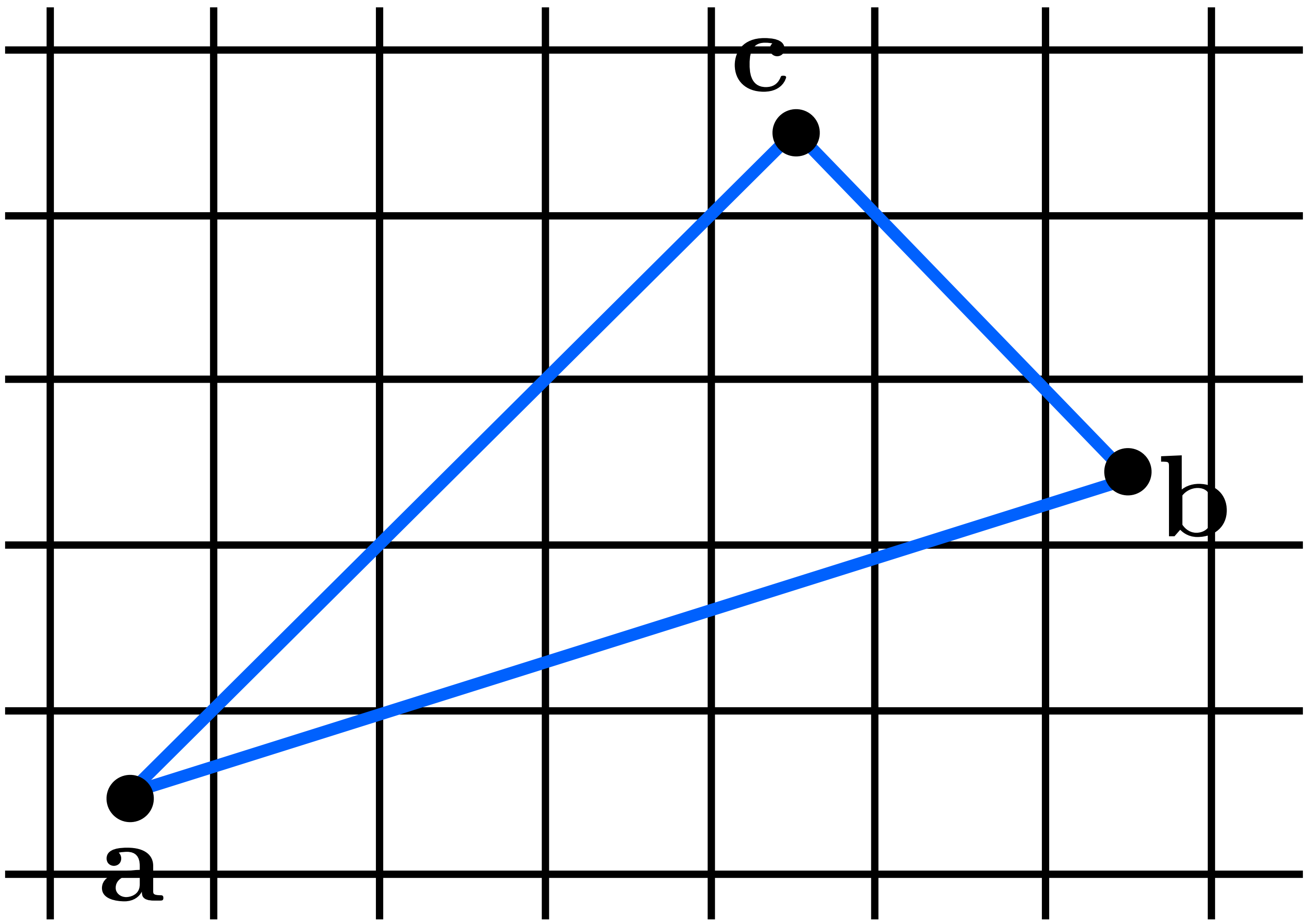
Tamar Shinar
Computer Science & Engineering
UC Riverside

3D graphics pipeline

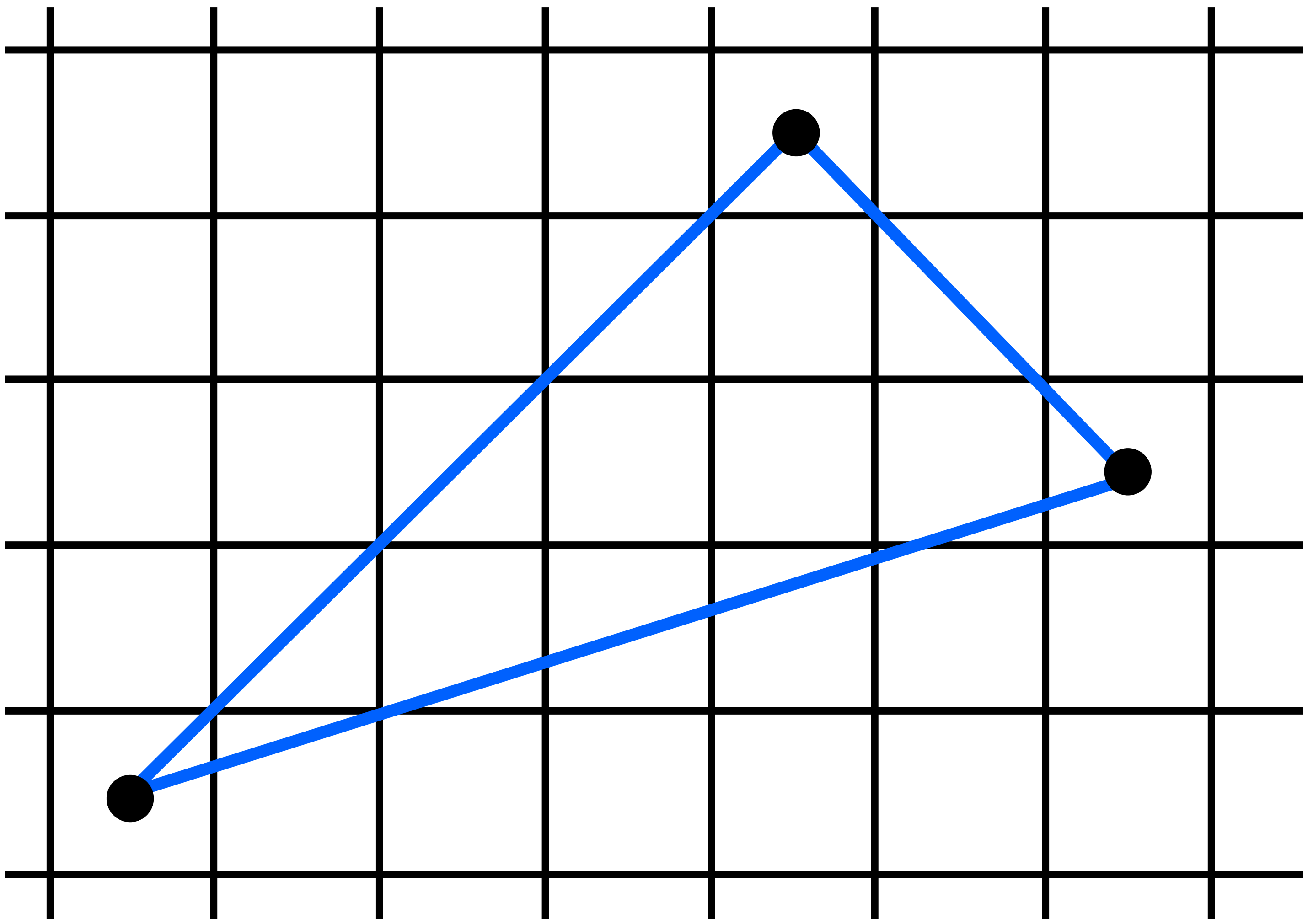


Triangle rasterization

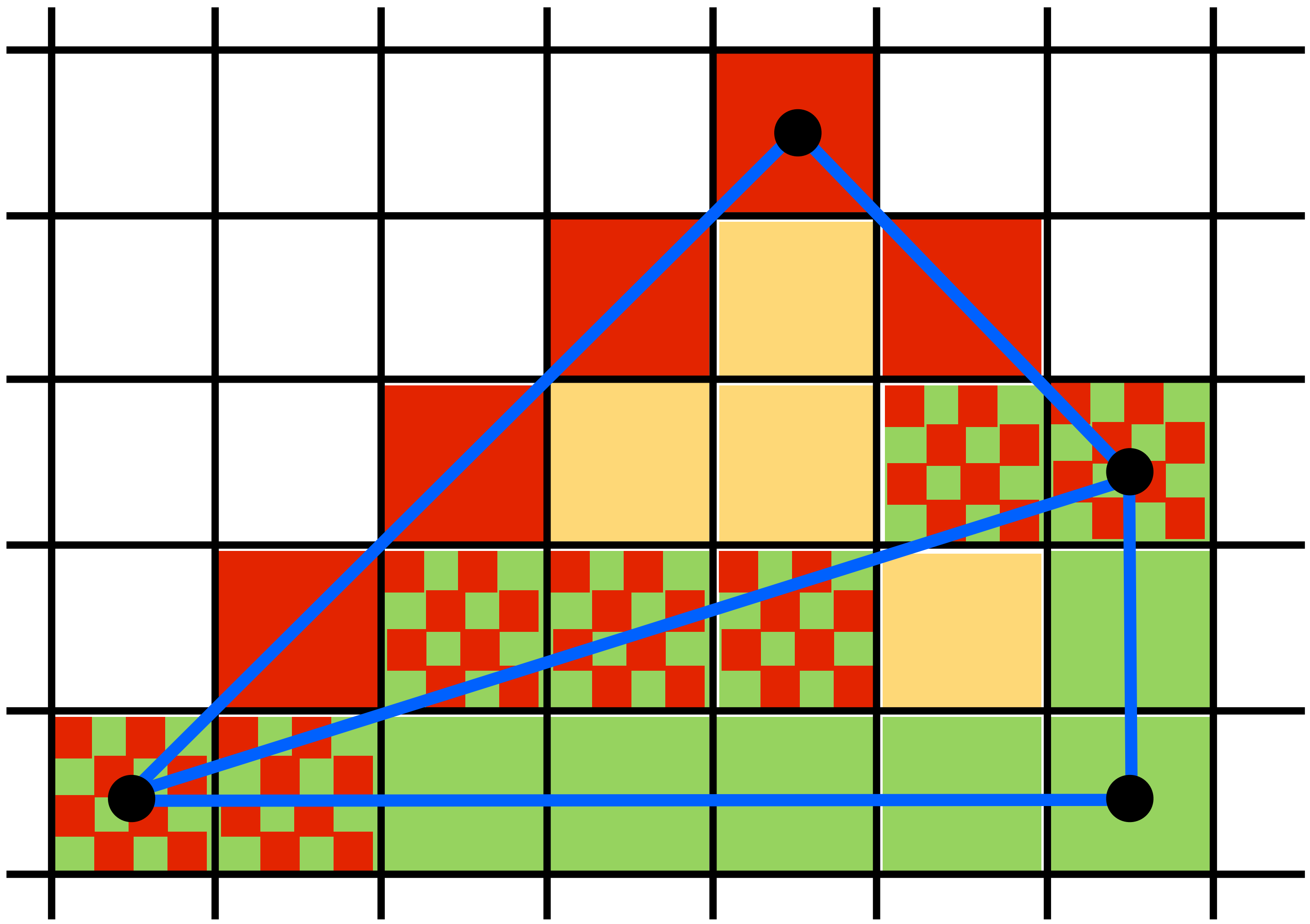
Which pixels should be used to approximate a triangle?



Triangle rasterization issues



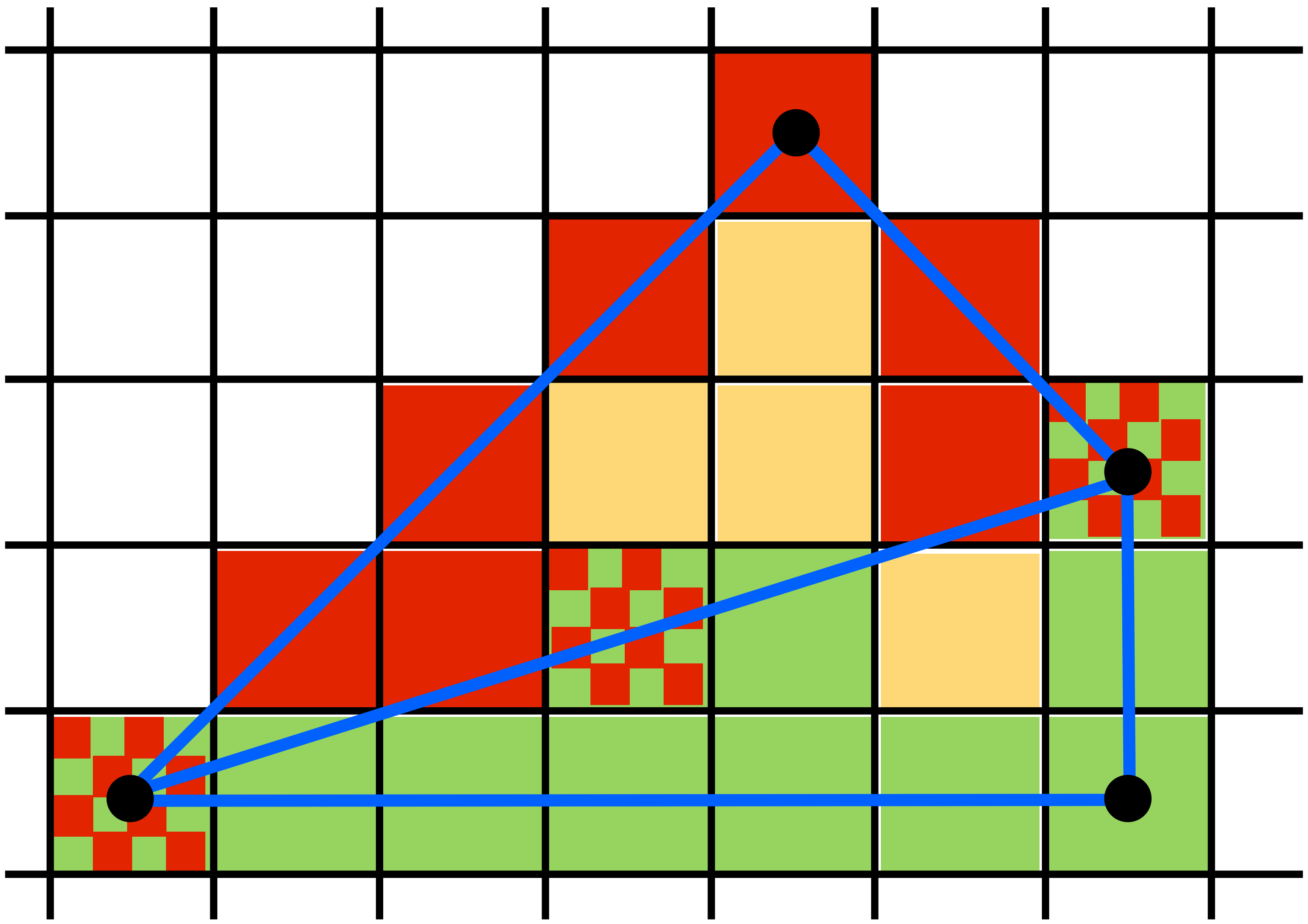
Which pixels should be used to approximate a triangle?



Who should fill in shared edge?

but who should fill in pixels for a shared edge?

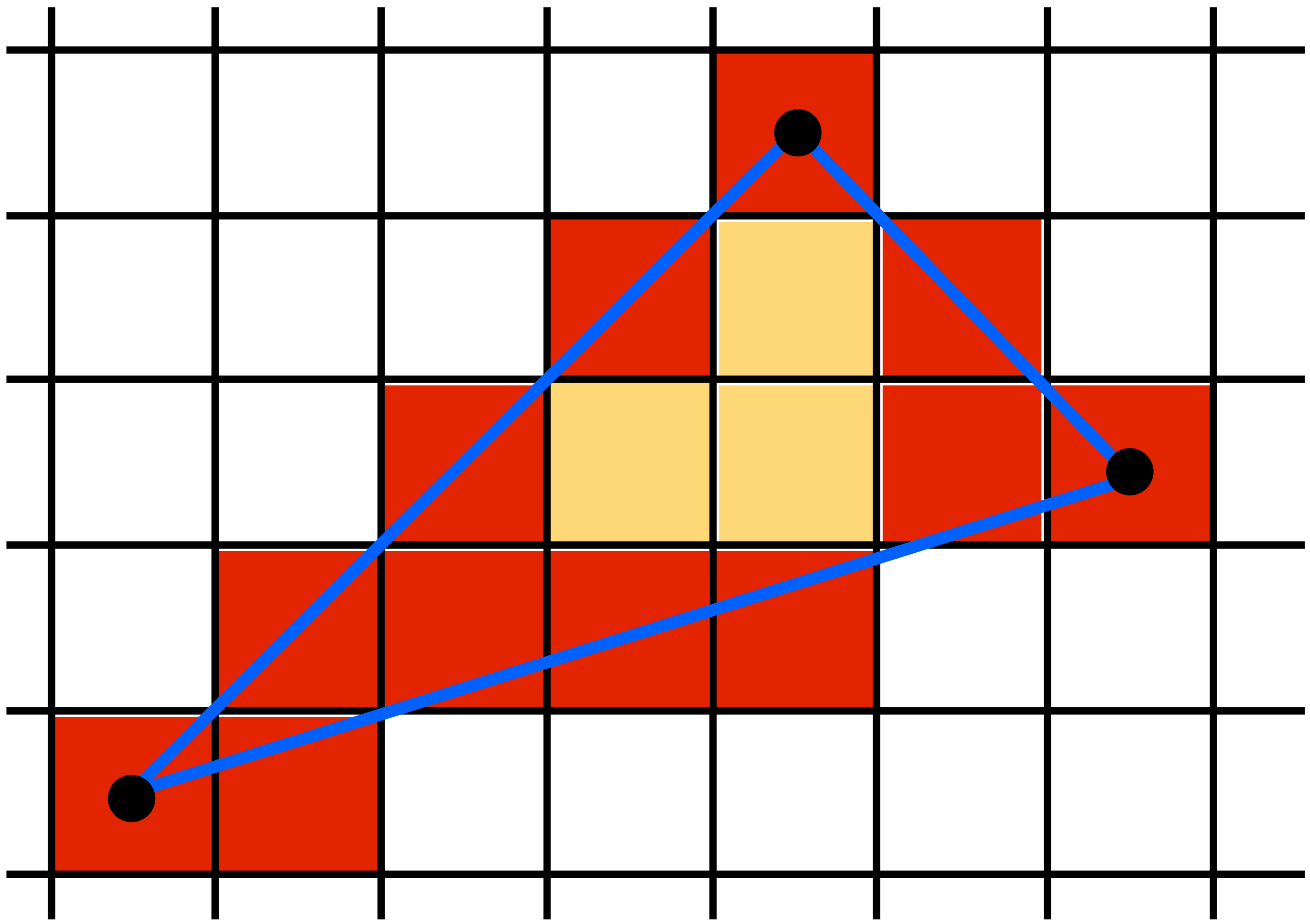
Which pixels should be used to approximate a triangle?



Who should fill in shared edge?

give to triangle that contains pixel center
– but we have some **ties**
why can't neither/both triangles draw the pixel?
    neither: gaps
    both: indeterminacy (due to indeterminate drawing order), incorrect, e.g., if both triangles are partially transparent
we want a **unique** assignment

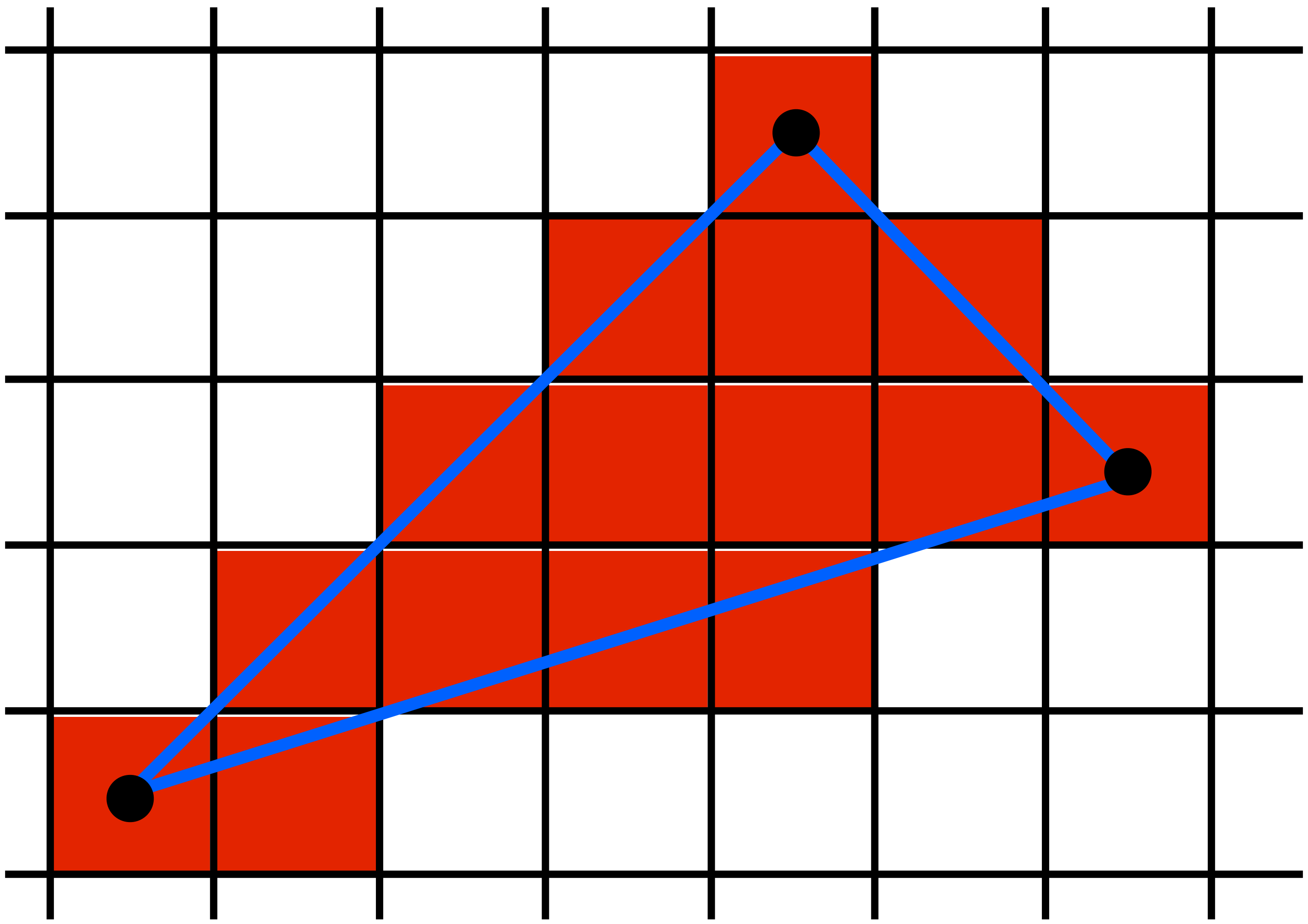
Which pixels should be used to approximate a triangle?



Use Midpoint Algorithm for edges and fill in?

That could be one possibility but we use a different approach based on barycentric coordinates

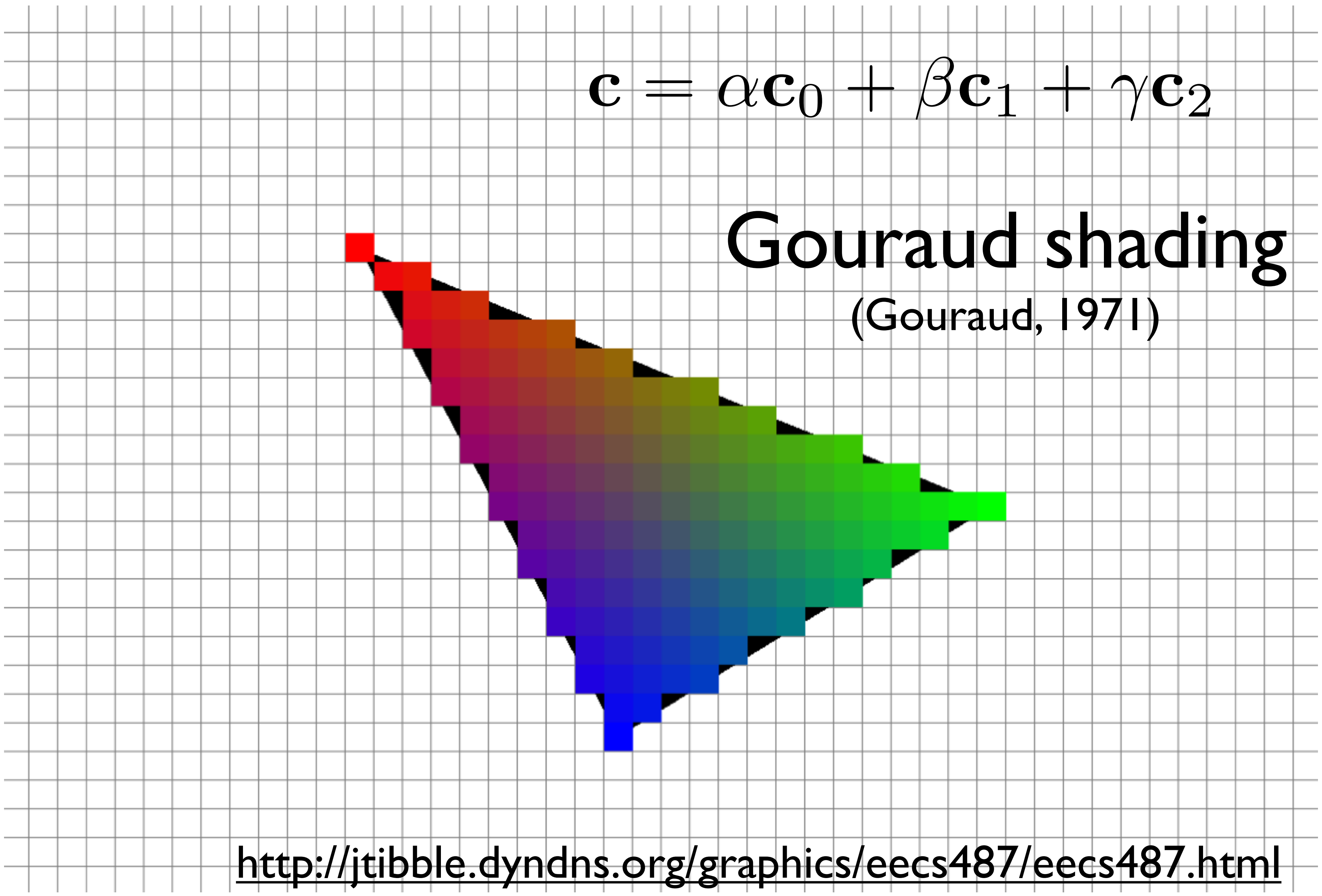
Which pixels should be used to approximate a triangle?



Use an approach based on barycentric coordinates

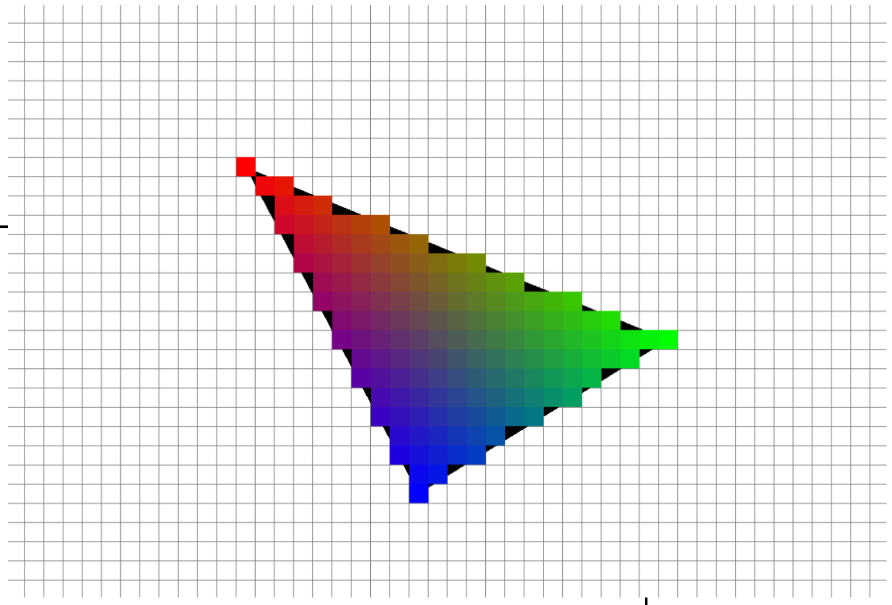
For each pixel, we compute its barycentric coordinates
If the coordinates are all $>= 0$, then the pixel is covered by the triangle

We can interpolate attributes using barycentric coordinates



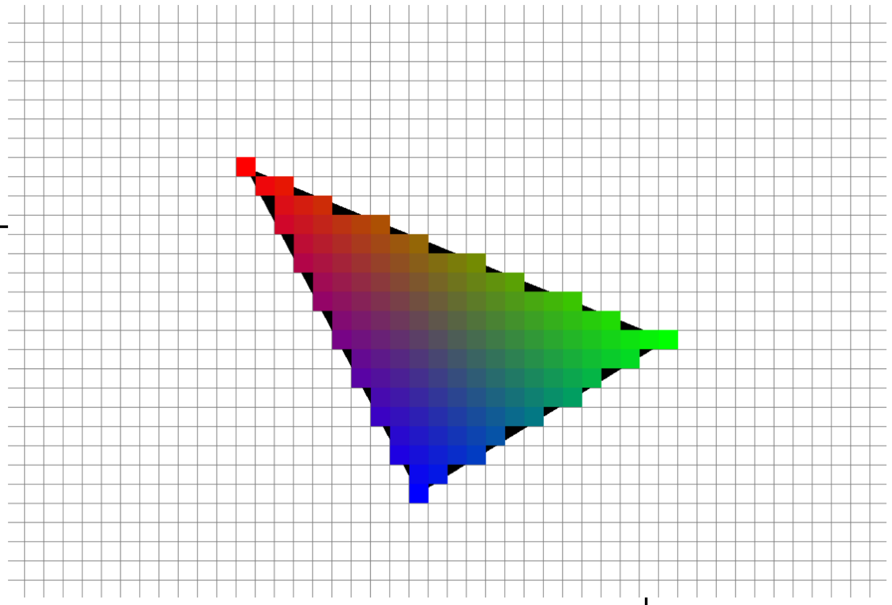
Using barycentric coordinates also has the advantage that we can easily interpolate colors or other attributes from triangle vertices

Triangle rasterization algorithm



for all x do
    for all y do
        compute (α, β, γ) for (x,y)
        if $(\alpha \in [0,1] \text{ and } \beta \in [0,1] \text{ and } \gamma \in [0,1])$ then
            $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
            drawpixel(x,y) with color c

Triangle rasterization algorithm

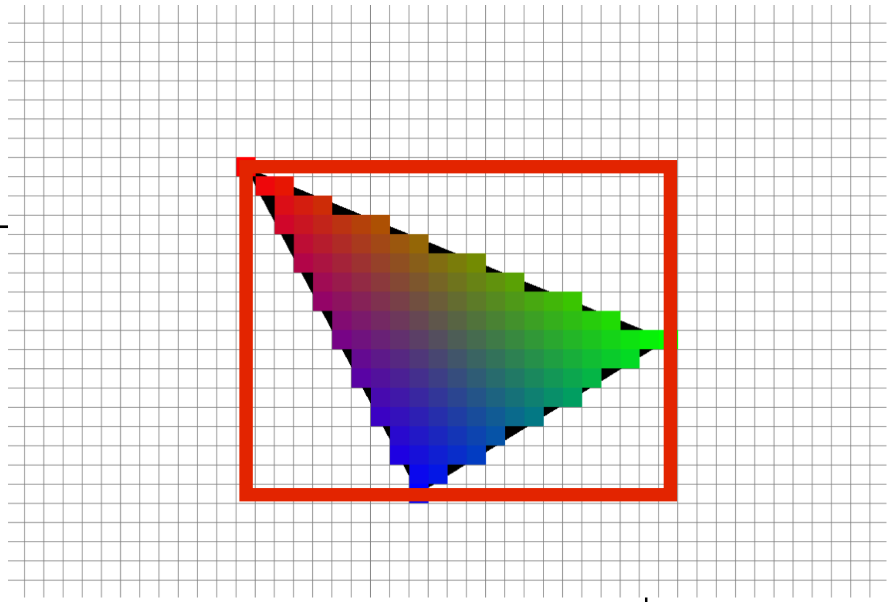


for all x do
    for all y do
        compute (α, β, γ) for (x,y)
        if $(\alpha \in [0,1] \text{ and } \beta \in [0,1] \text{ and } \gamma \in [0,1])$ then
            $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
            drawpixel(x,y) with color c

the rest of the algorithm is to make the steps in red more **efficient**

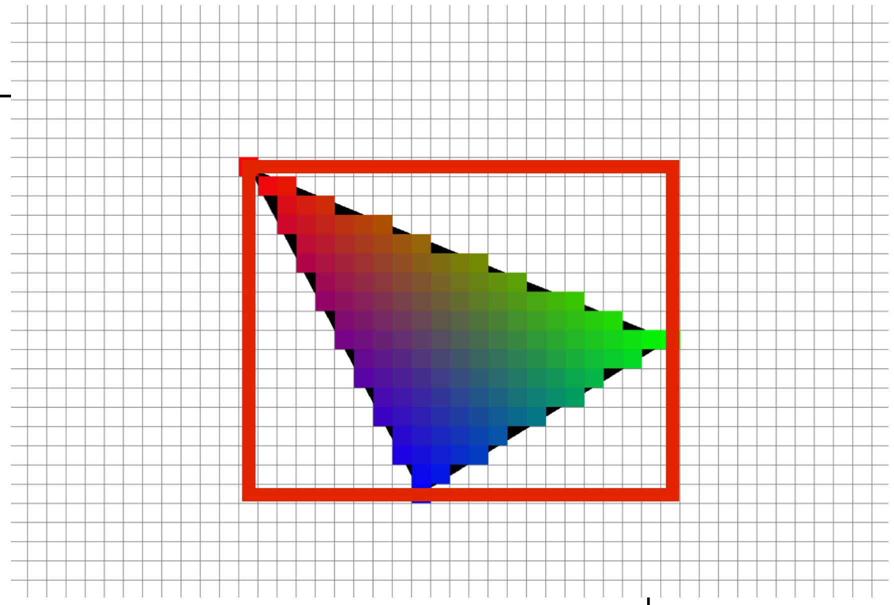
Triangle rasterization algorithm

use a bounding rectangle



for x in [x_min, x_max]
    for y in [y_min, y_max]
        compute (α, β, γ) for (x,y)
        if $(\alpha \in [0,1] \text{ and } \beta \in [0,1] \text{ and } \gamma \in [0,1])$ then
            $\mathbf{c} = \alpha \mathbf{c}_0 + \beta \mathbf{c}_1 + \gamma \mathbf{c}_2$
            drawpixel(x,y) with color c

Triangle rasterization algorithm

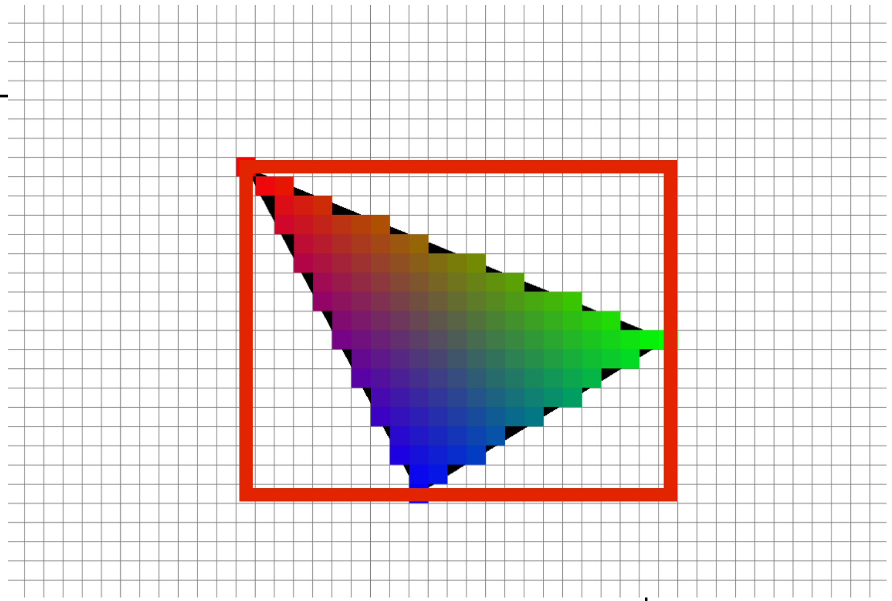


for x in [x_min, x_max]
    for y in [y_min, y_max]
        $\alpha = f_{bc}(x, y)/f_{bc}(x_a, y_a)$
        $\beta = f_{ca}(x, y)/f_{ca}(x_b, y_b)$
        $\gamma = f_{ab}(x, y)/f_{ab}(x_c, y_c)$
        if $(\alpha \in [0,1] \text{ and } \beta \in [0,1] \text{ and } \gamma \in [0,1])$ then
            $\mathbf{c} = \alpha \mathbf{c}_0 + \beta \mathbf{c}_1 + \gamma \mathbf{c}_2$
            drawpixel(x,y) with color c

<whiteboard> : computing alpha, beta, and gamma

Triangle rasterization algorithm

Optimizations?

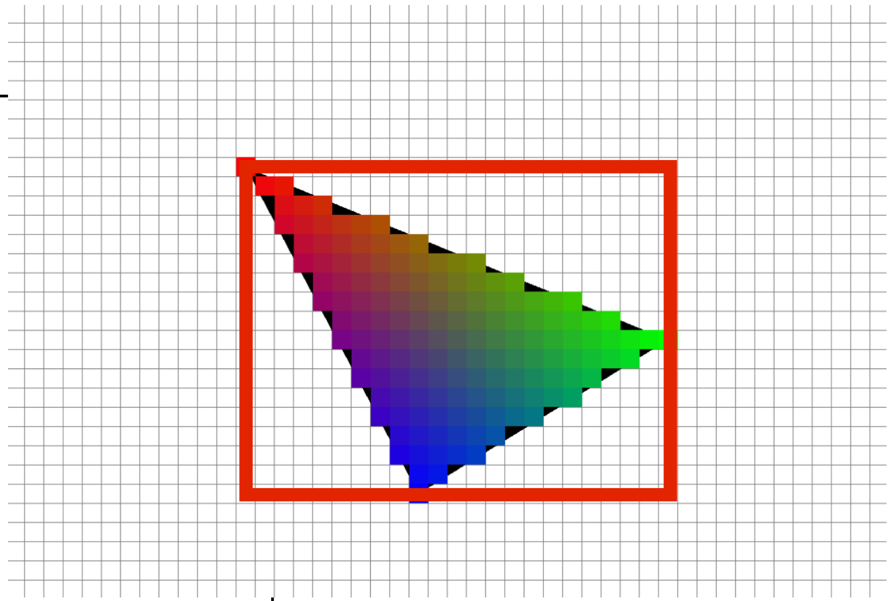


for x in [x_min, x_max]
    for y in [y_min, y_max]
        $\alpha = f_{bc}(x, y)/f_{bc}(x_a, y_a)$
        $\beta = f_{ca}(x, y)/f_{ca}(x_b, y_b)$
        $\gamma = f_{ab}(x, y)/f_{ab}(x_c, y_c)$
        if $(\alpha \in [0,1] \text{ and } \beta \in [0,1] \text{ and } \gamma \in [0,1])$ then
            $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
            drawpixel(x,y) with color c

1. can make computation of bary. coords. **incremental**
- f(x,y) = Ax+By+C
- f(x+1,y) = f(x,y) + A
2. **color** computation can also be made **incremental**
3. **alpha > 0 and beta > 0 and gamma > 0** (if true => they are also less than one)

Triangle rasterization algorithm

Optimizations?



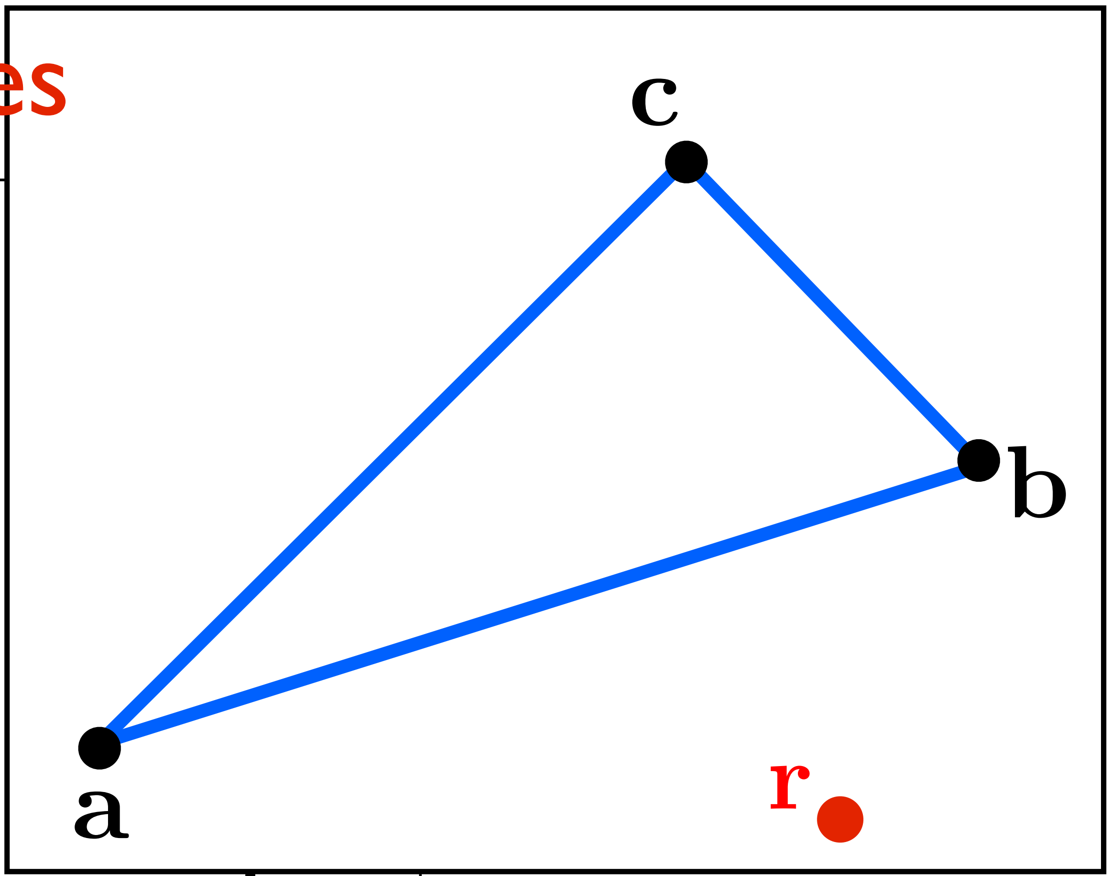
for x in [x_min, x_max]
    for y in [y_min, y_max]
        $\alpha = f_{bc}(x,y)/f_{bc}(x_a,y_a)$
        $\beta = f_{ca}(x,y)/f_{ca}(x_b,y_b)$
        $\gamma = f_{ab}(x,y)/f_{ab}(x_c,y_c)$
        if $(\alpha \geq 0 \text{ and } \beta \geq 0 \text{ and } \gamma \geq 0)$ **then**
            $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
            drawpixel(x,y) with color c

make computation of bary. coords. incremental
color can also be computed incrementally
don't need to check upper bound

1. can make computation of bary. coords. **incremental**
- f(x,y) = Ax+By+C
- f(x+1,y) = f(x,y) + A
2. **color** computation can also be made **incremental**
3. **alpha > 0 and beta > 0 and gamma > 0** (if true => they are also less than one)

Triangle rasterization algorithm

dealing with shared triangle edges

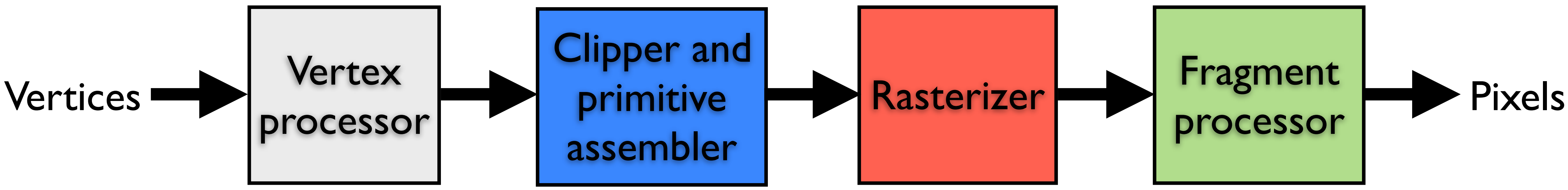


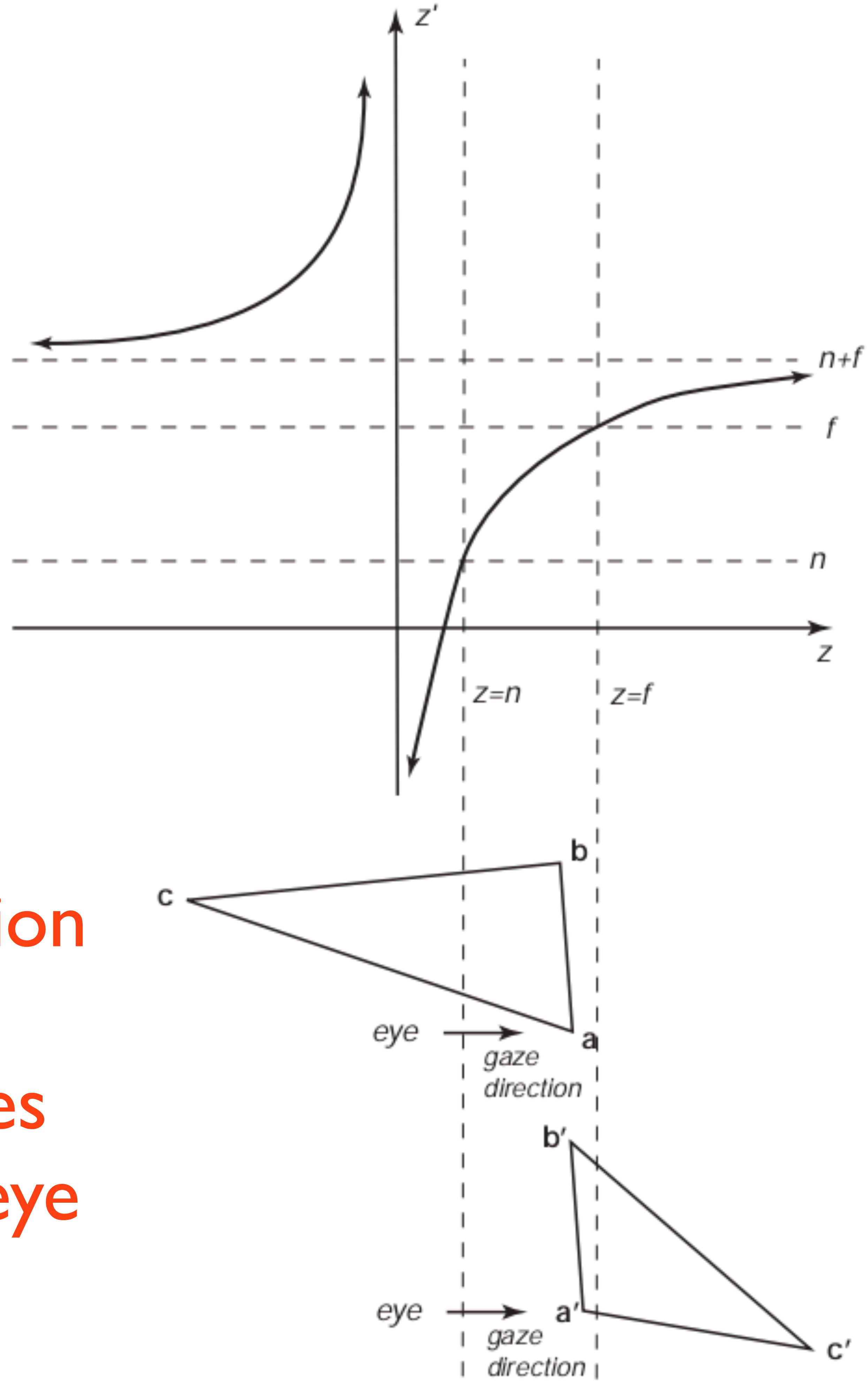
for x in [x_min, x_max]
    for y in [y_min, y_max]
        $\alpha = f_{bc}(x,y)/f_{bc}(x_a,y_a)$
        $\beta = f_{ac}(x,y)/f_{ac}(x_b,y_b)$
        $\gamma = f_{ab}(x,y)/f_{ab}(x_c,y_c)$
        if $(\alpha \geq 0 \text{ and } \beta \geq 0 \text{ and } \gamma \geq 0)$ then
            if $(\alpha > 0 \text{ or } f_{bc}(\mathbf{a})f_{bc}(\mathbf{r}) > 0) \text{ and}$ $(\beta > 0 \text{ or } f_{ca}(\mathbf{b})f_{ca}(\mathbf{r}) > 0) \text{ and}$ $(\gamma > 0 \text{ or } f_{ab}(\mathbf{c})f_{ab}(\mathbf{r}) > 0)$ then
                $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
                drawpixel(x,y) with color c

- compute f_12(r), f_20(r) and f_01(r) and make sure r doesn't hit a line

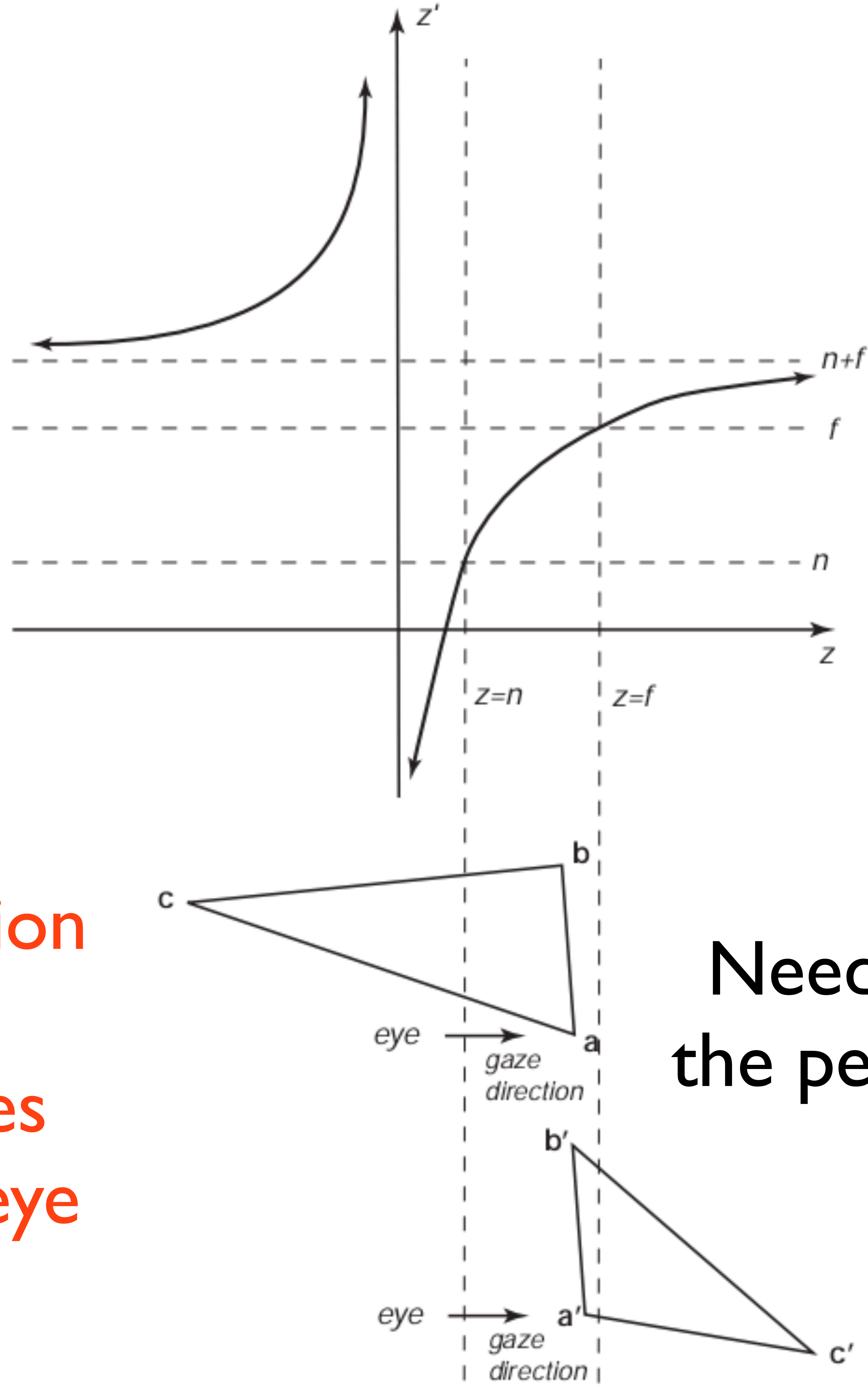
Clipping

3D graphics pipeline





Perspective transformation incorrectly maps vertices behind the eye

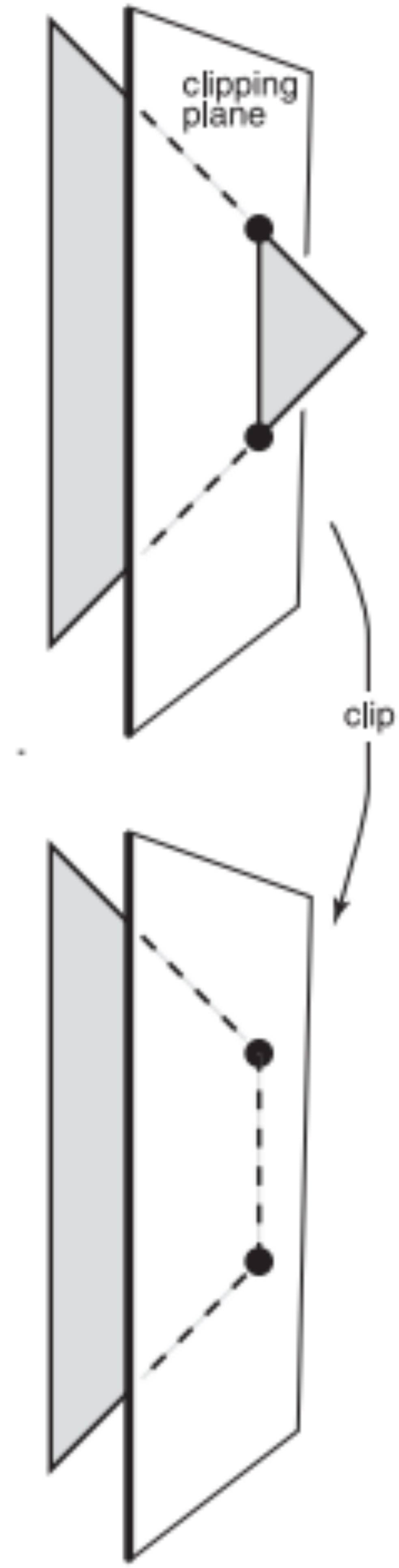


Perspective transformation incorrectly maps vertices behind the eye

Need to clip <u>before</u> the perspective divide

Clipping usually takes place in one of two places:
- in world coordinates against the six sides of the view volume, or
- after the 4D transformation but <u>before</u> perspective division (i.e., in homogenous coordinates)

Clip triangle against a plane



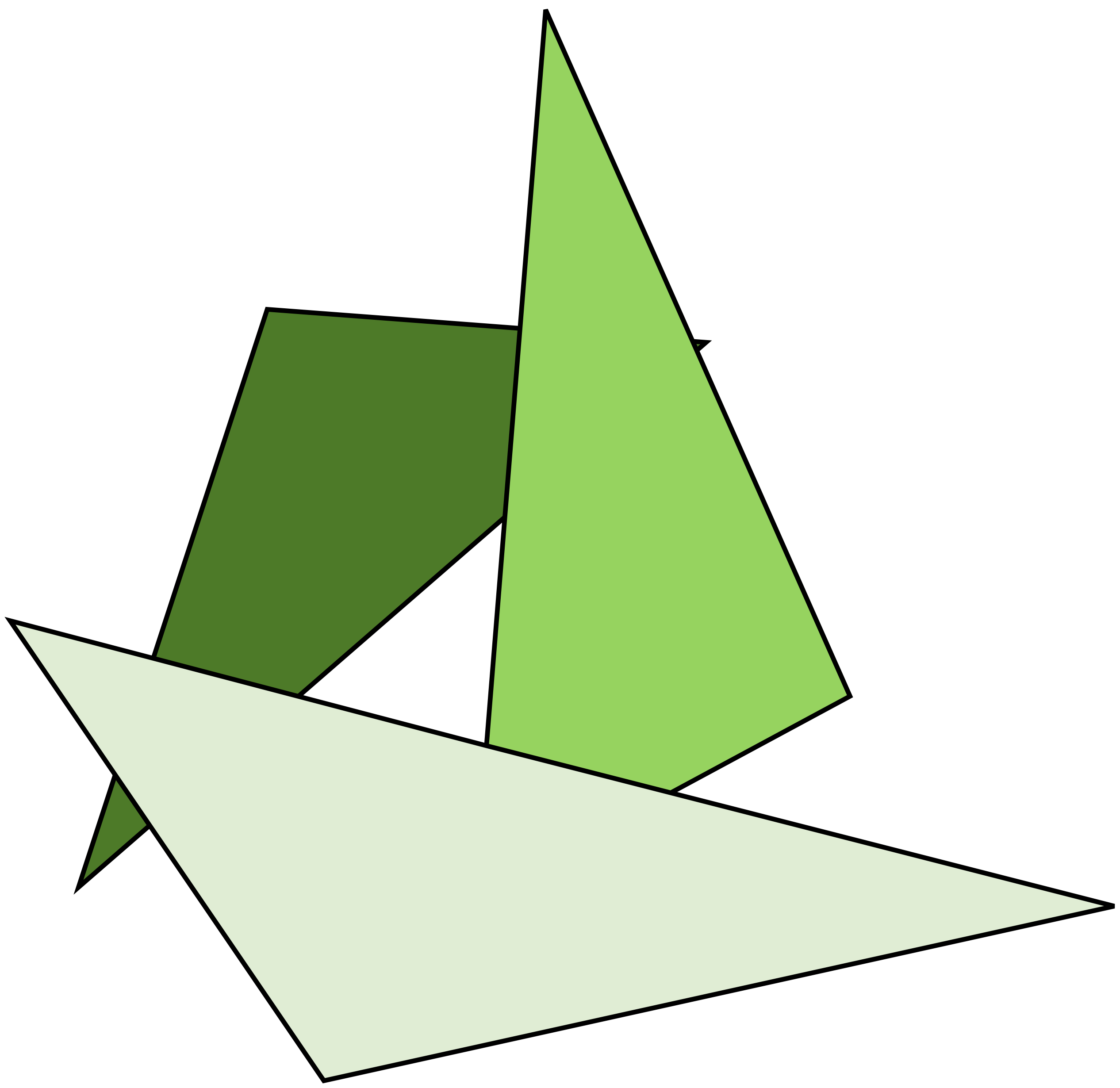
Simple pipeline examples

- Simple **2D** pipeline
  - application inputs pixel coordinates, pipeline only does the rasterization phase and overwrites framebuffer contents
- Simple **3D** pipeline
  - viewing transformation (camera, projection, and viewport), followed by rasterization

but how to deal with hidden surfaces?

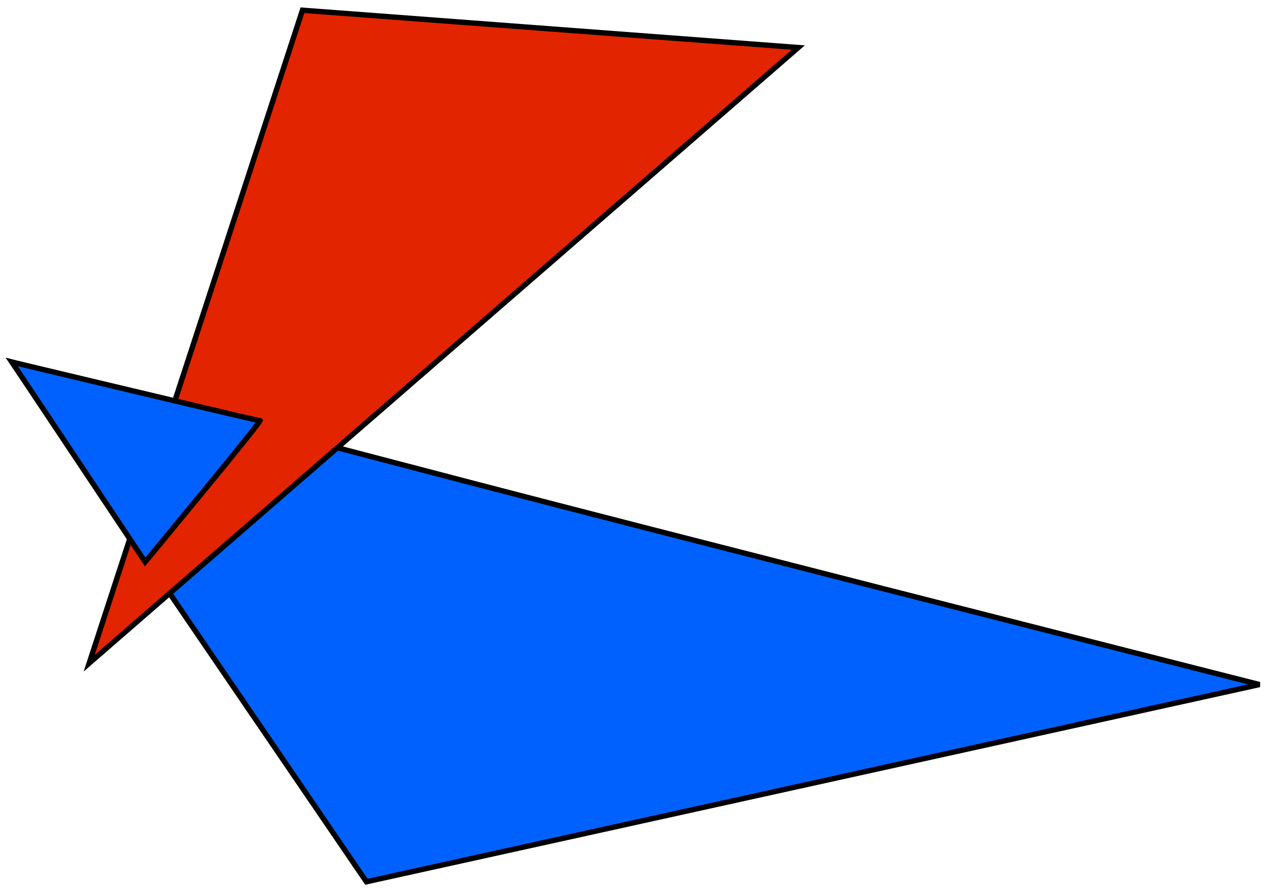
Hidden Surface Removal

Painter's algorithm



draw primitives in back-to-front order

Painter's algorithm

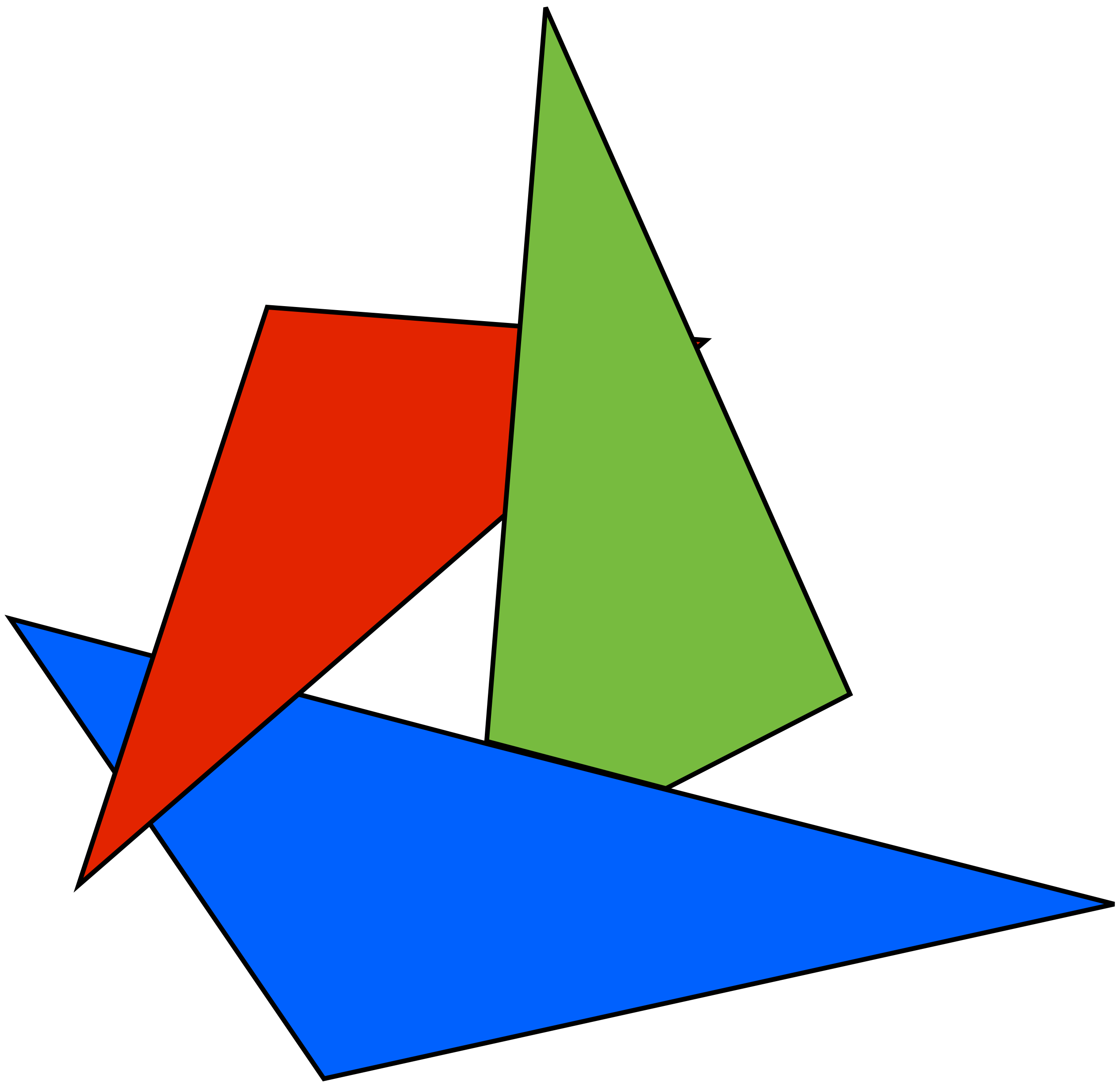


draw primitives in back-to-front order

problem: triangle intersection

who's in front of whom?

Painter's algorithm



draw primitives in back-to-front order

problem: occlusion cycle

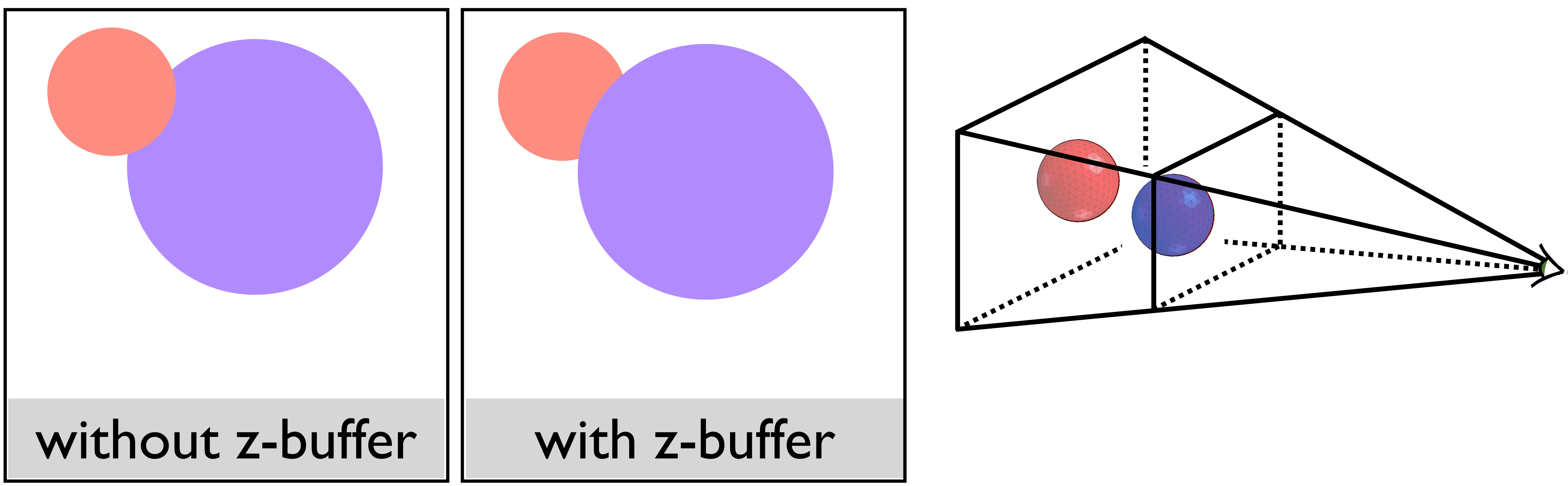
also, sorting primitives by depth is **slow**

Use a *z-buffer* for hidden surface removal

at each pixel, record distance to the closest object that has been drawn in a *depth* buffer

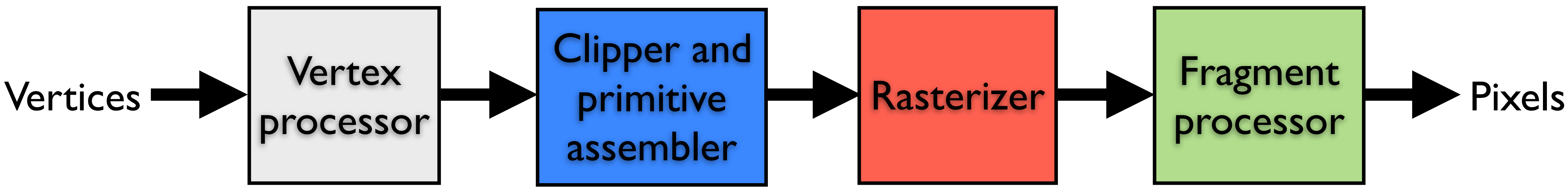
Use a *z-buffer* for hidden surface removal

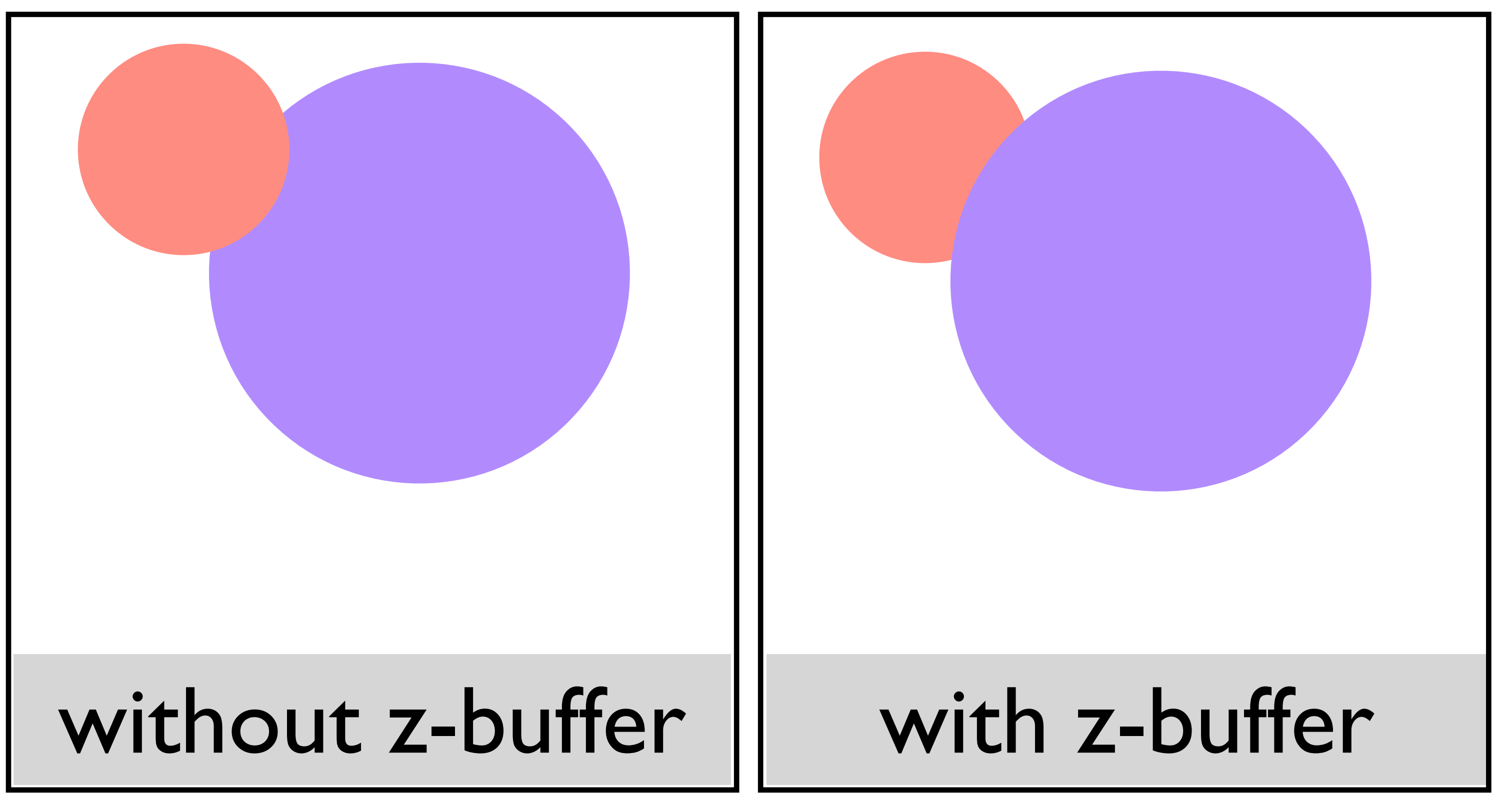
at each pixel, record distance to the closest object that has been drawn in a *depth* buffer

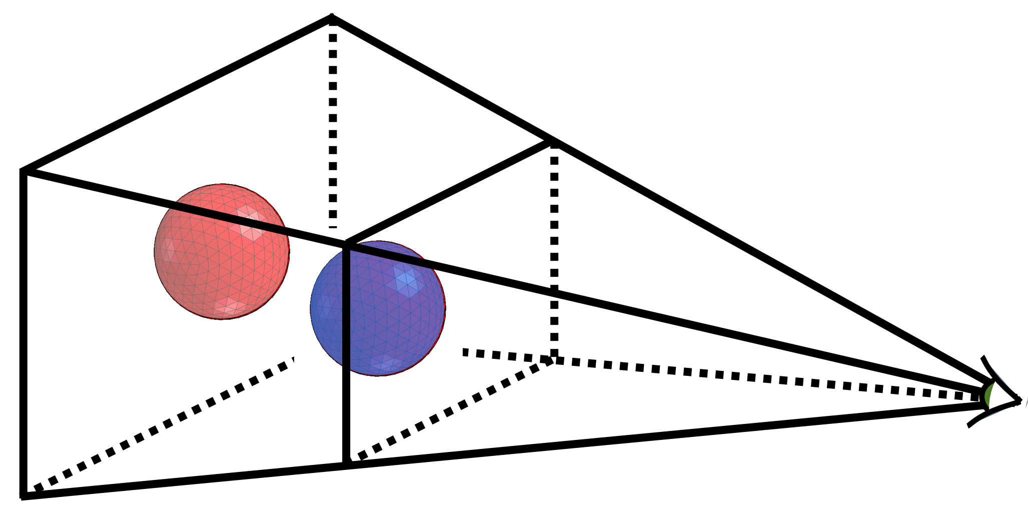


- assume both spheres of the same size, red drawn last

Use a *z-buffer* for hidden surface removal

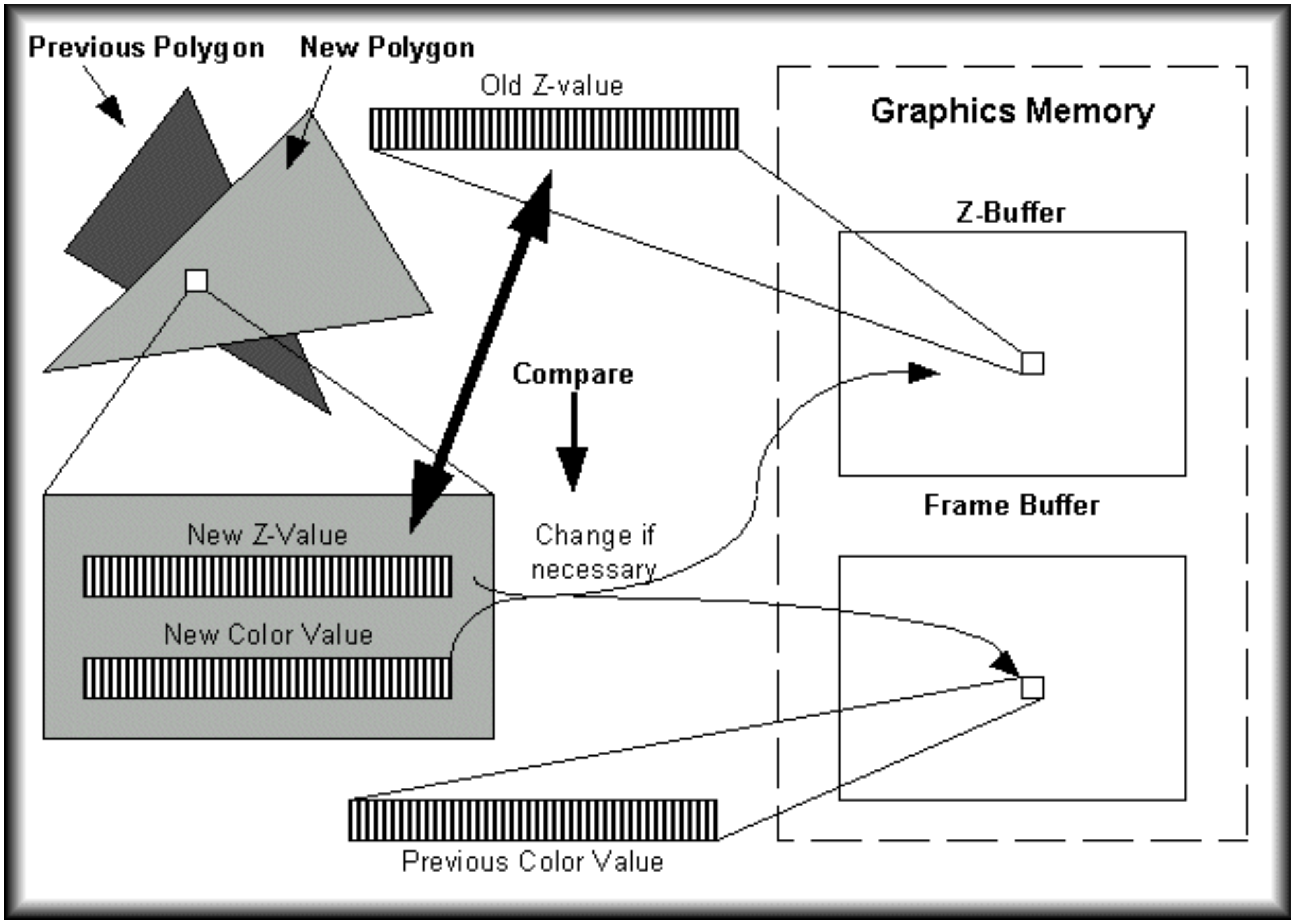






done in the **fragment blending** phase
- each fragment must carry a depth

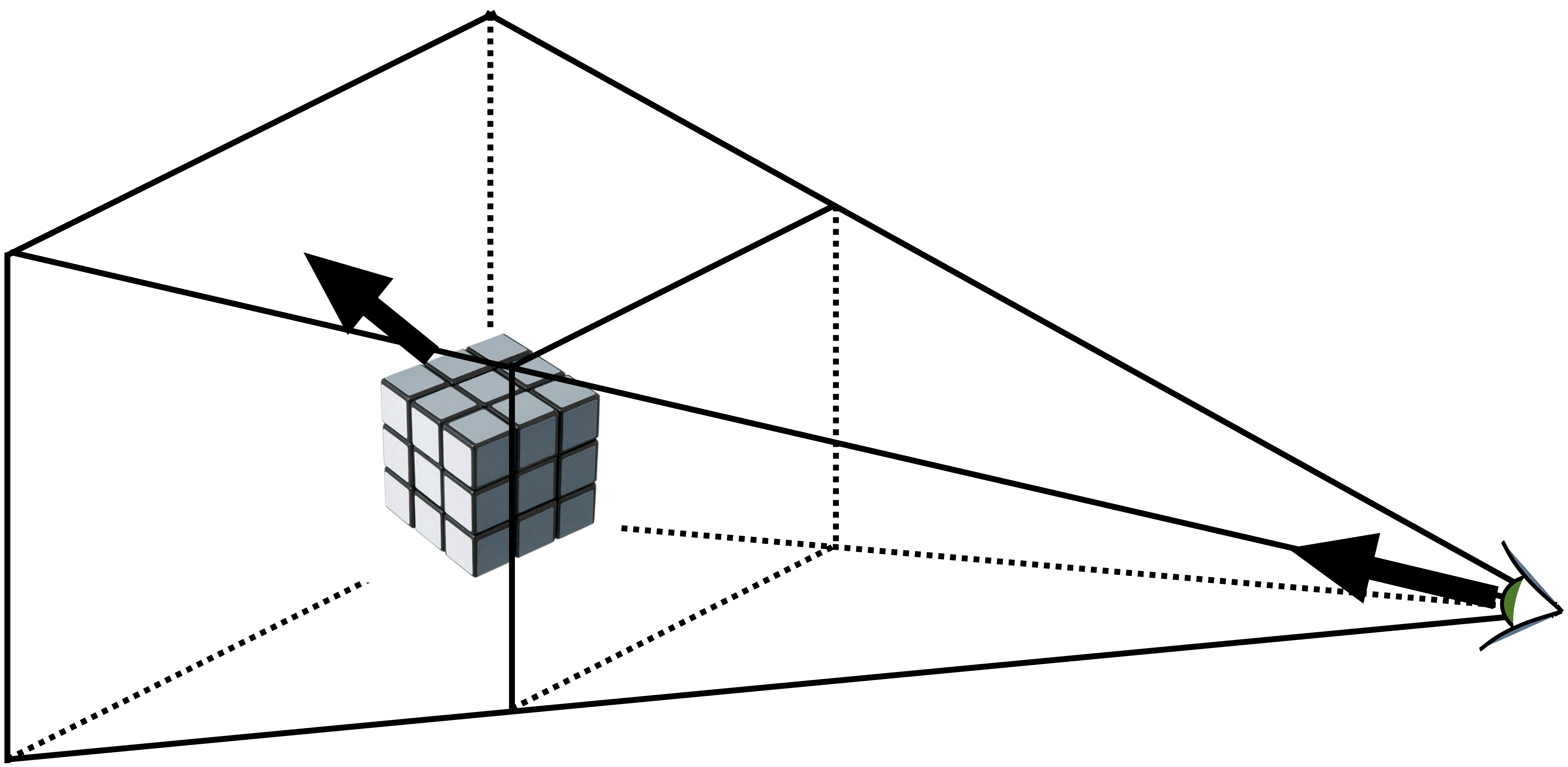
Use a *z-buffer* for hidden surface removal



http://www.beyond3d.com/content/articles/41/

graphics memory contains the previous color value and associated z value.
computing a new color and z value. Compare z-values and if new z value is bigger than old z value, overwrite

Backface culling: another way to eliminate hidden geometry



this is only okay for closed surfaces

Hidden Surface Removal in OpenGL

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

glEnable(GL_DEPTH_TEST);

glEnable(GL_CULL_FACE);
```

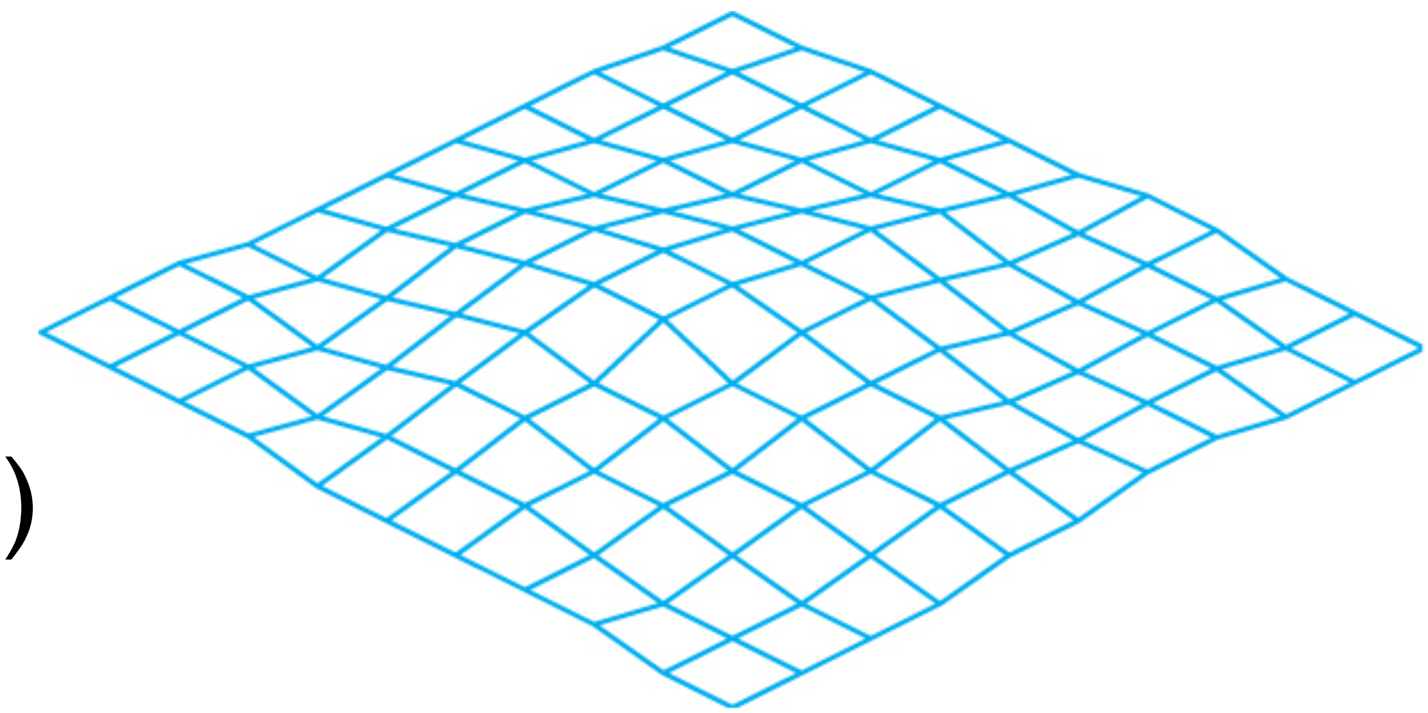
For a perspective transformation, there is more precision in the depth buffer for z-values closer to the near plane

Shading Polygonal Geometry

Smooth surfaces are often approximated by polygons

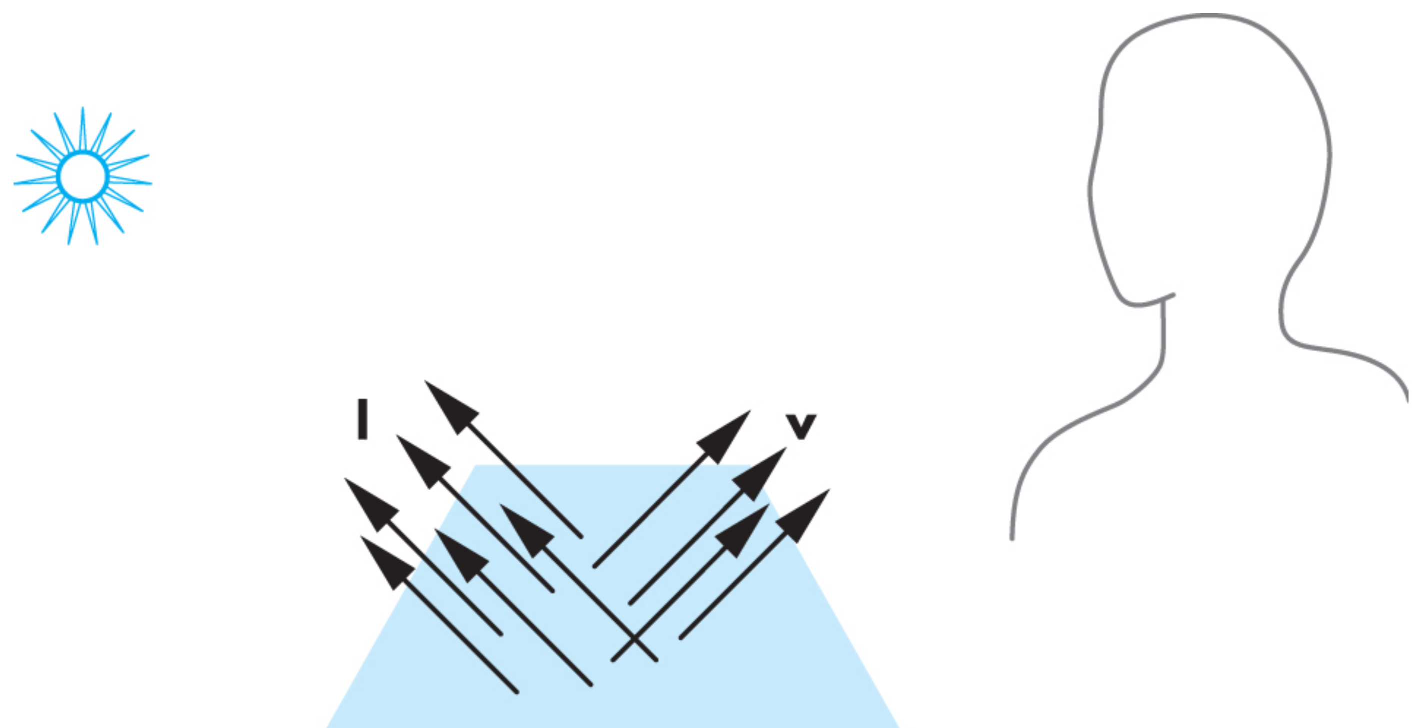
Shading approaches:

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



each polygon is flat and has a well-defined normal

Flat Shading



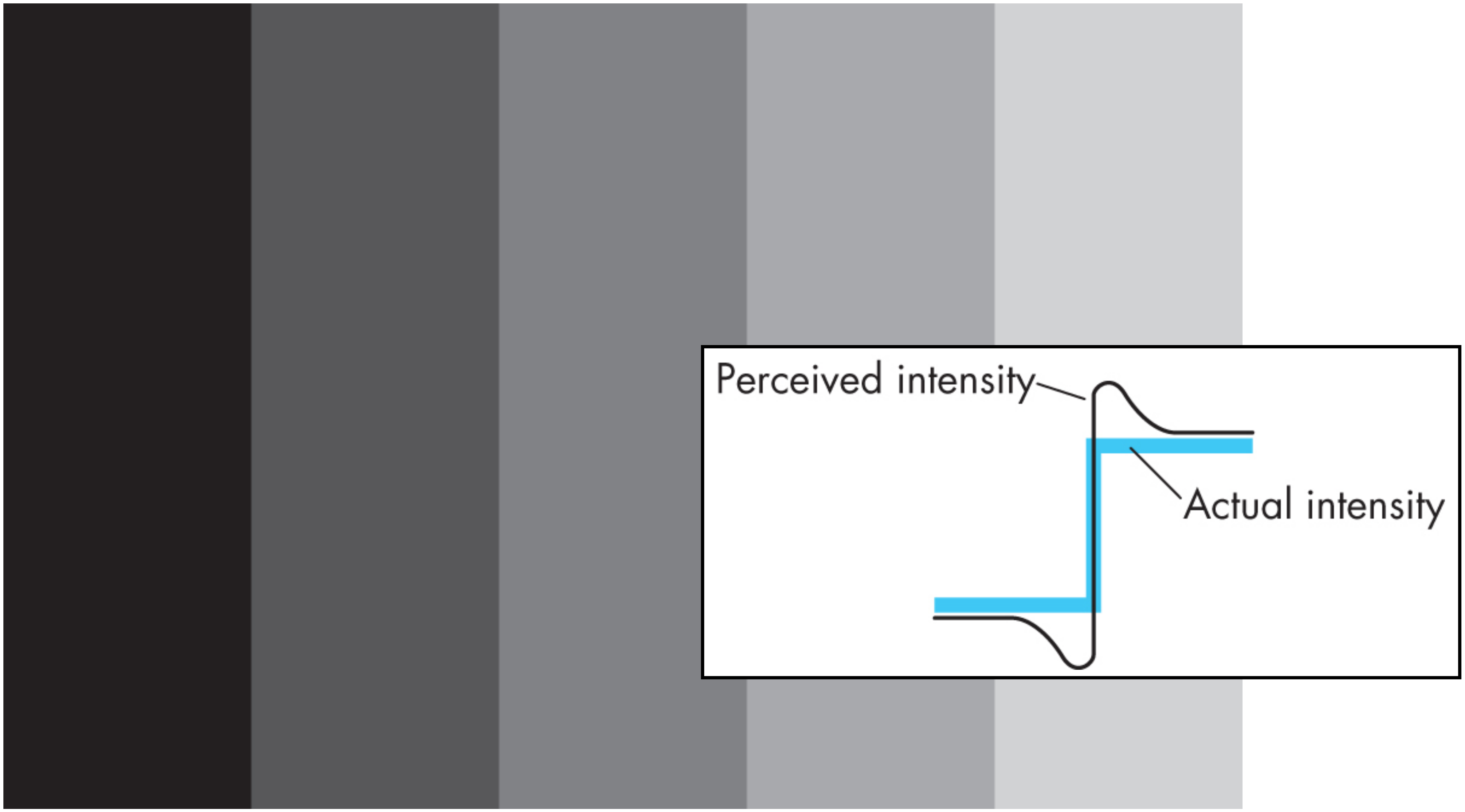
do the shading calculation once per **polygon**

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

In general, l, n, and v vary from point to point on a surface. If we assume a distant viewer, v can be thought of as constant. If we assume a distant light source, l can be thought of as constant. For a flat polygon, n is constant.

If the light source or viewer is not at inf, we need heuristic for picking color – e.g., first vertex, or polygon center

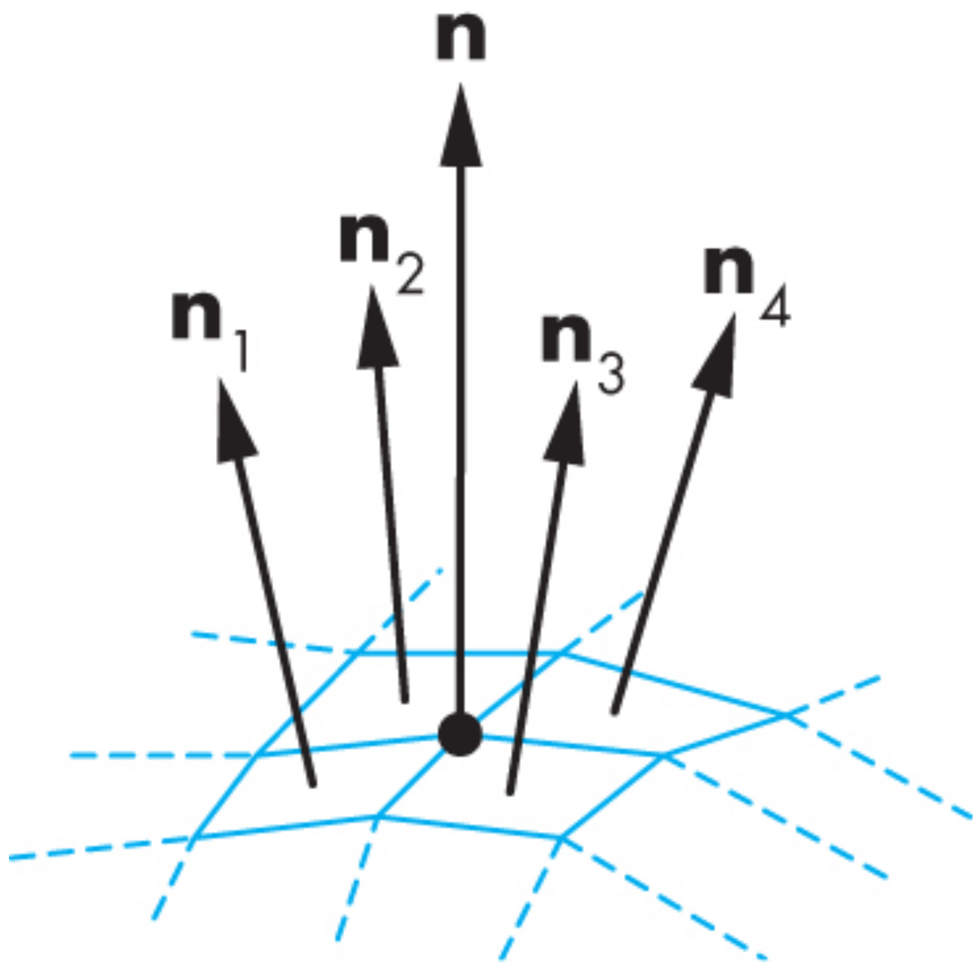
Mach Band Effect



Flat shading doesn't usually look too good.
The **lateral inhibition** effect makes flat shading seem even worse.

Smooth Shading

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

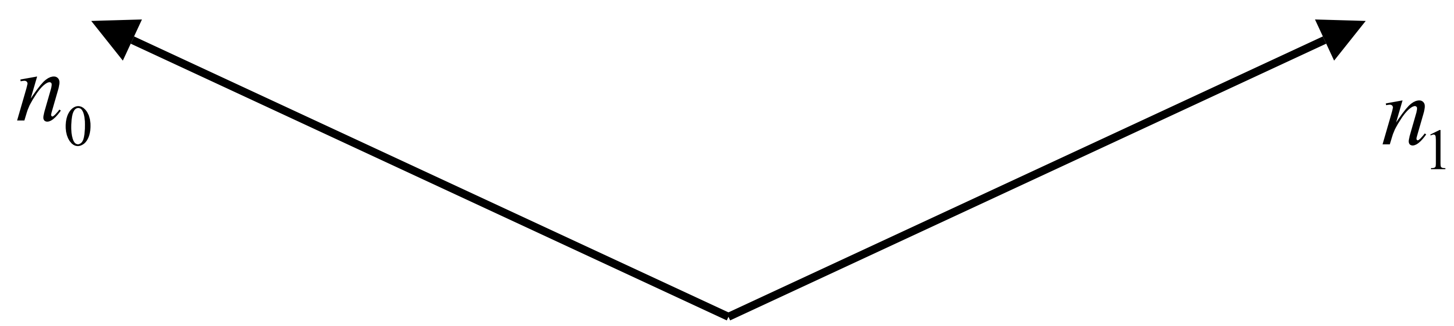


do the shading calculation once per **vertex**

We assign the vertex normals based on the surrounding polygon normals

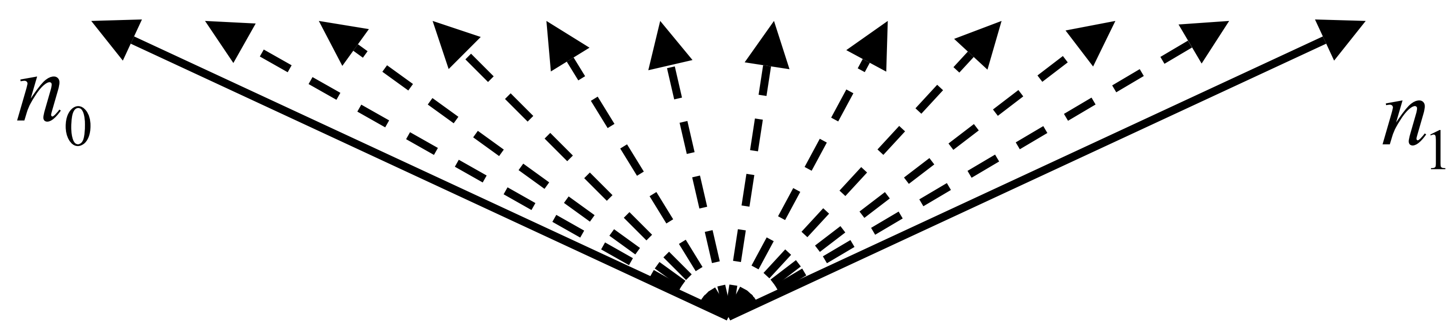
Interpolating Normals

- Must renormalize



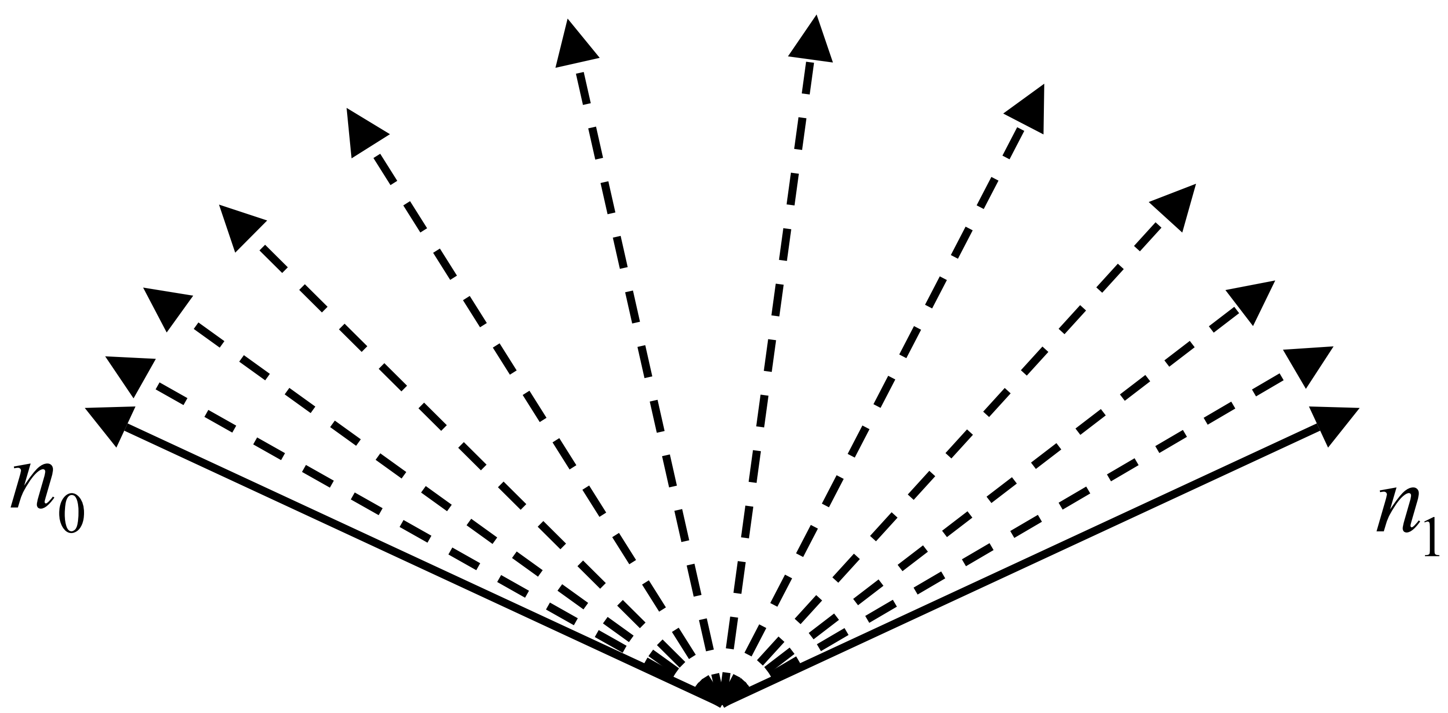
Interpolating Normals

- Must renormalize

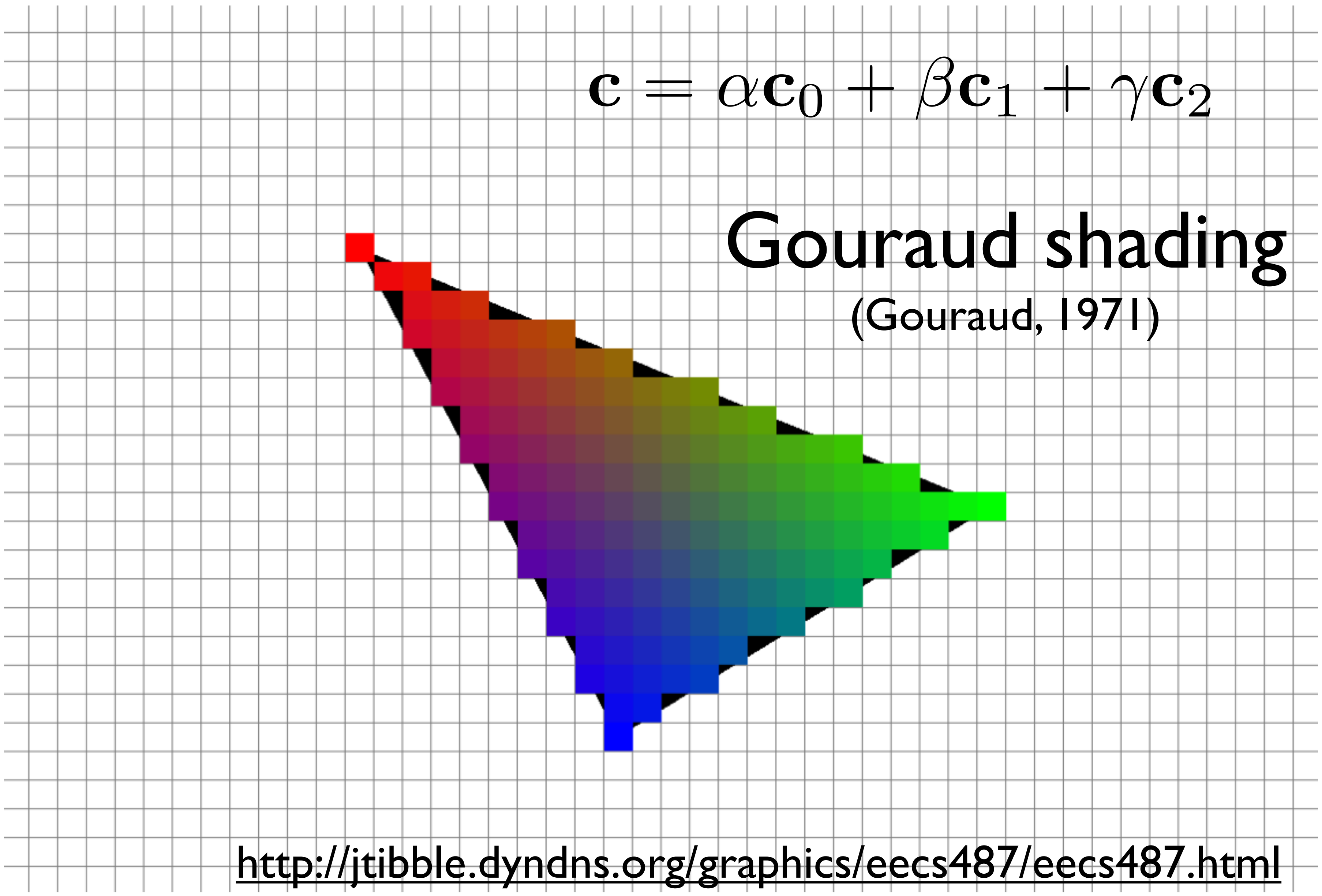


Interpolating Normals

- Must renormalize

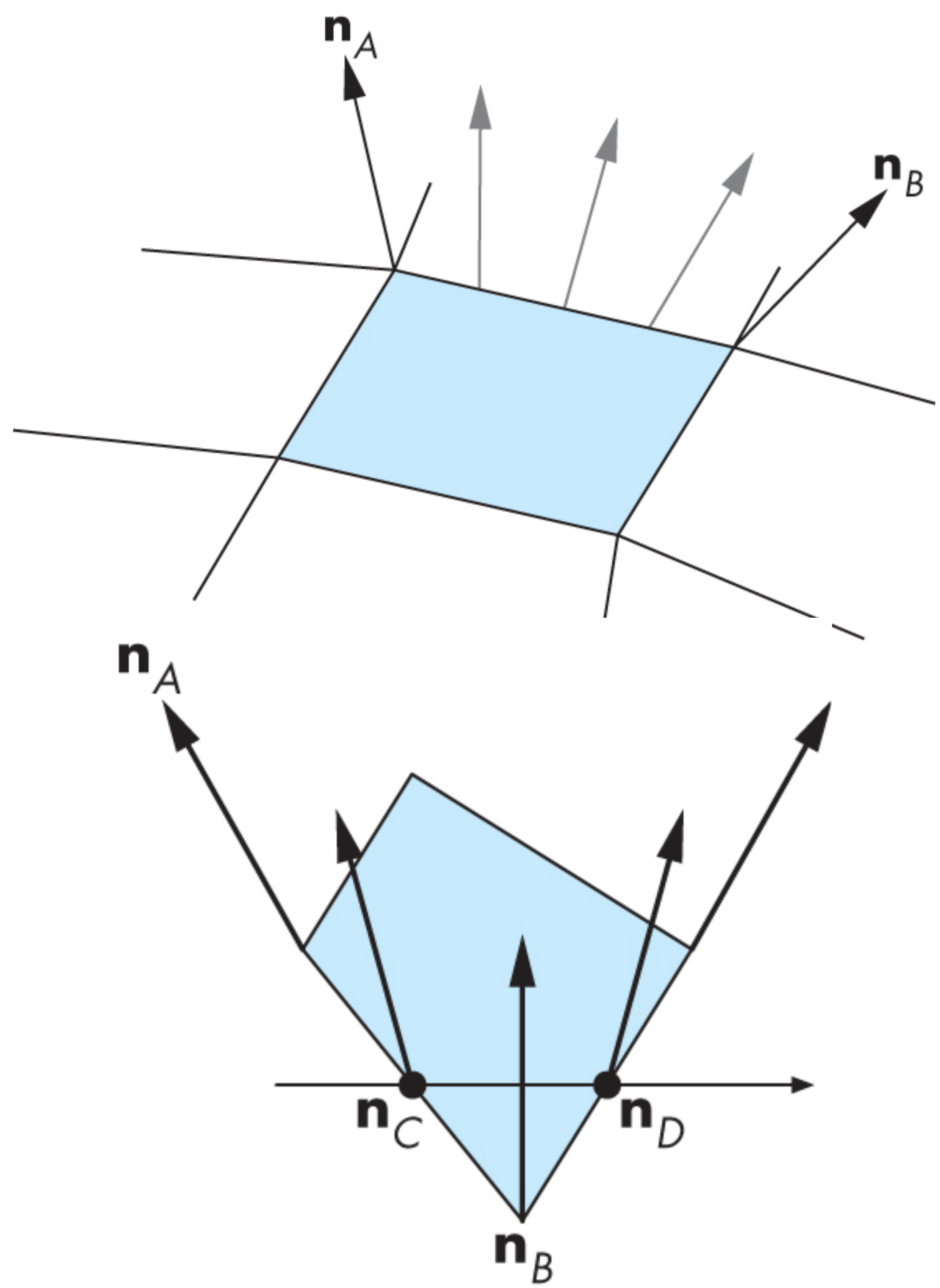


We can interpolate attributes using barycentric coordinates



Using barycentric coordinates also has the advantage that we can easily interpolate colors or other attributes from triangle vertices

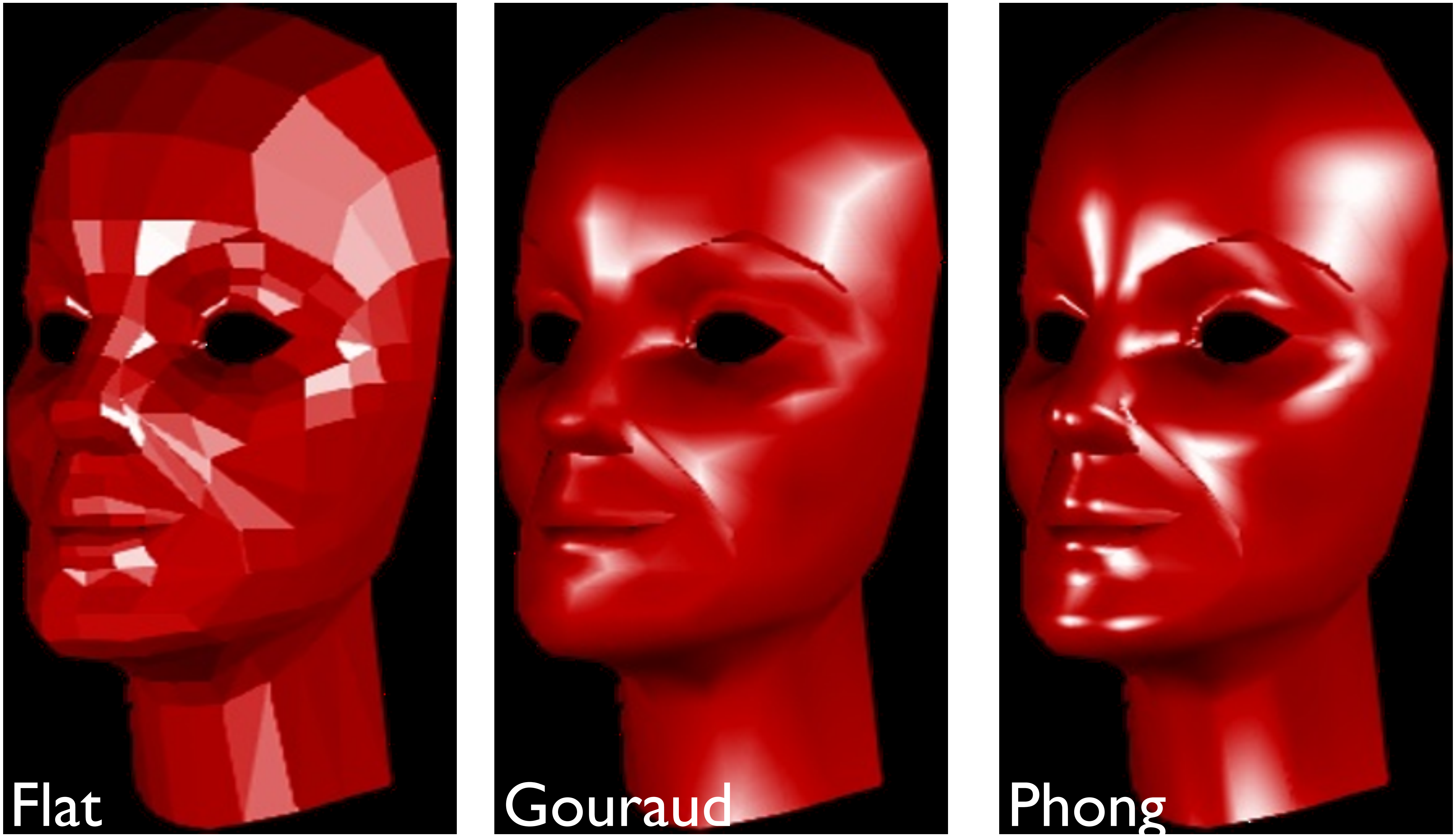
Phong Shading



do the shading calculation once per **fragment**

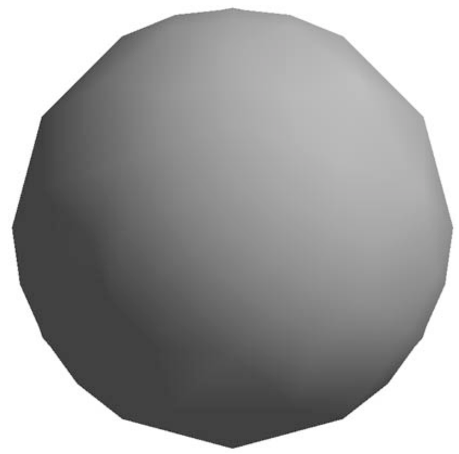
Phong shading requires normals to be interpolated across each polygon -- this wasn't part of the fixed function pipeline.
This can now be done in the pipeline in the fragment shader.

Comparison

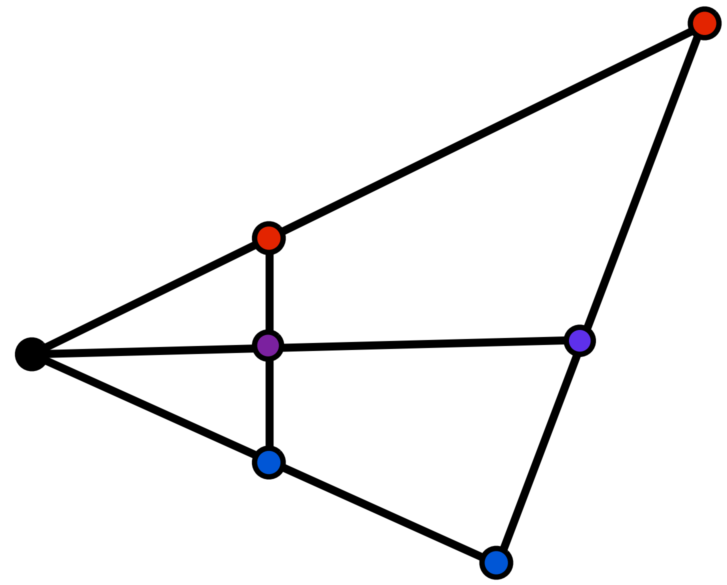


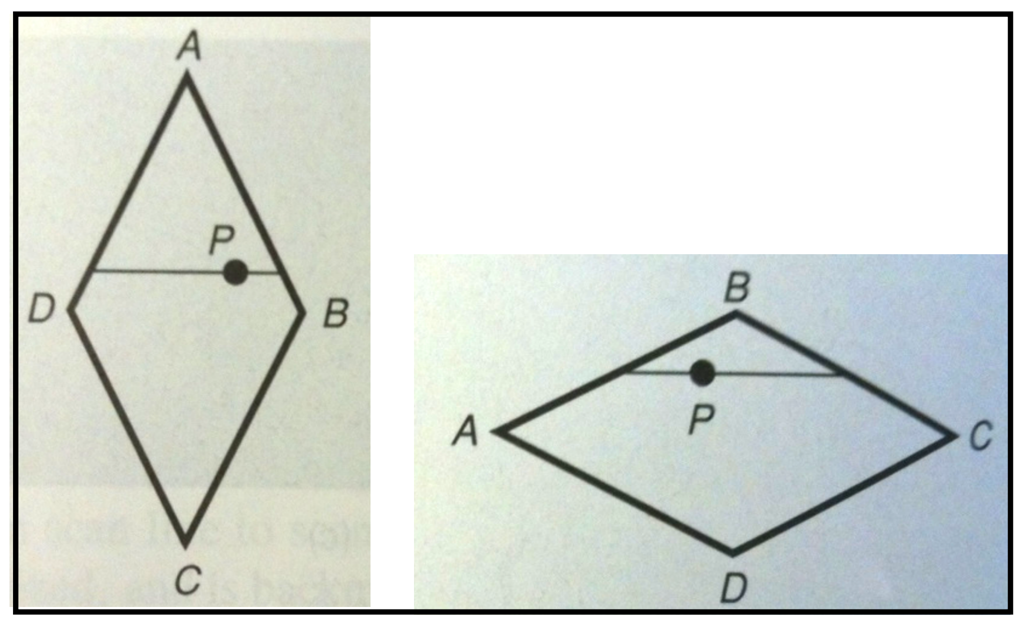
- Phong interpolation looks smoother -- can see edges on the Gouraud model
- but Phong is a lot more work
- both Phong and Gouraud require vertex normals
- both Phong and Gouraud leave silhouettes

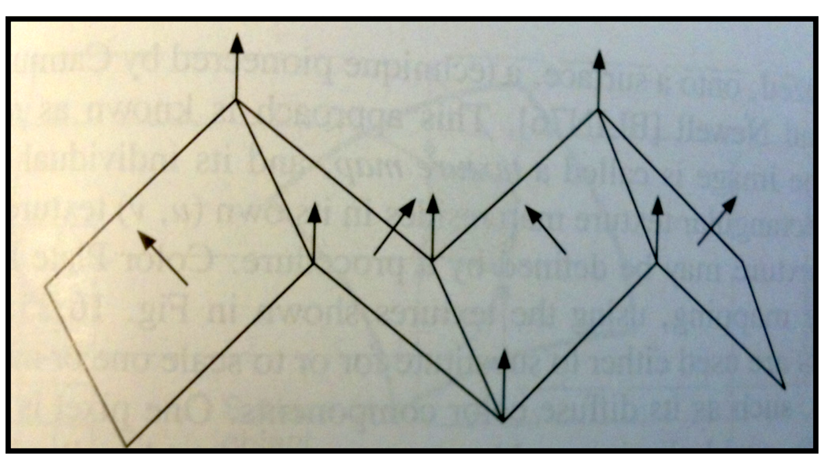
Problems with Interpolated Shading



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



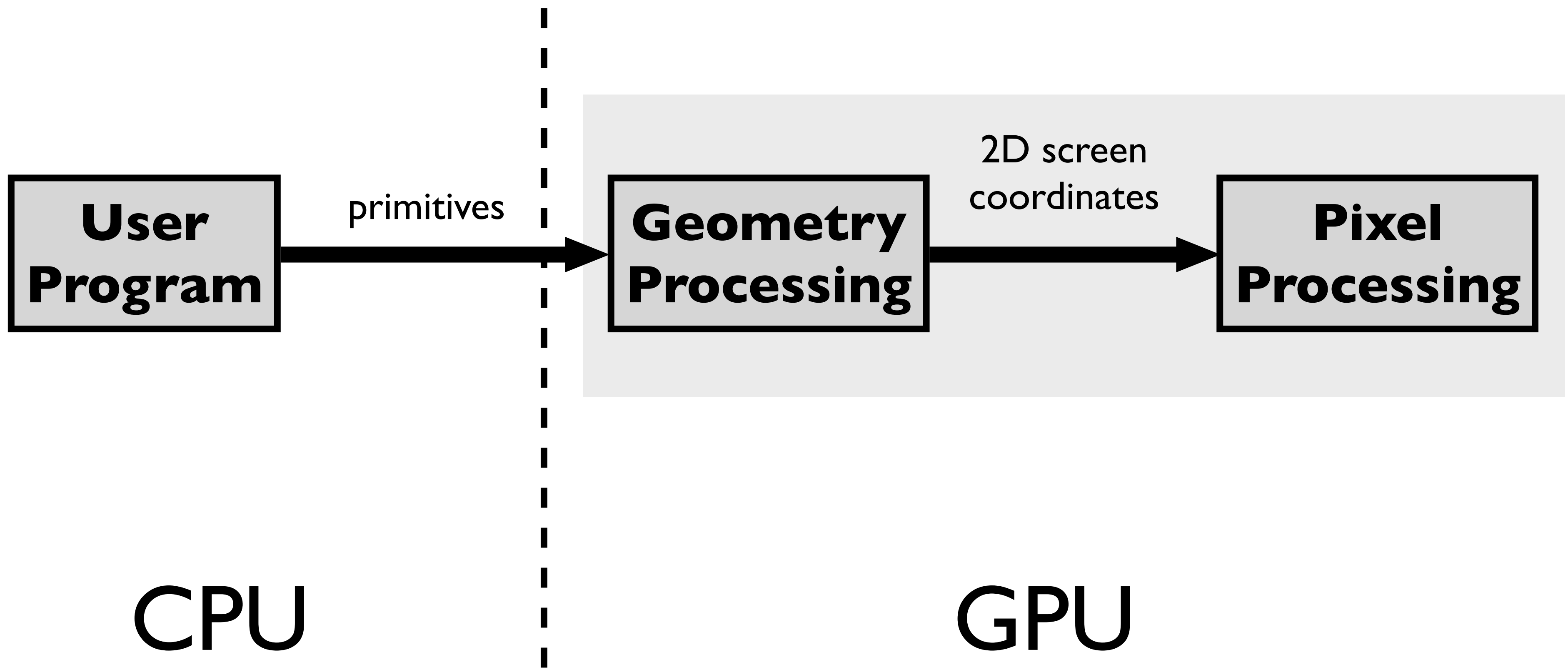




[Foley, van Dam, Feiner, Hughes]

Programmable Shading

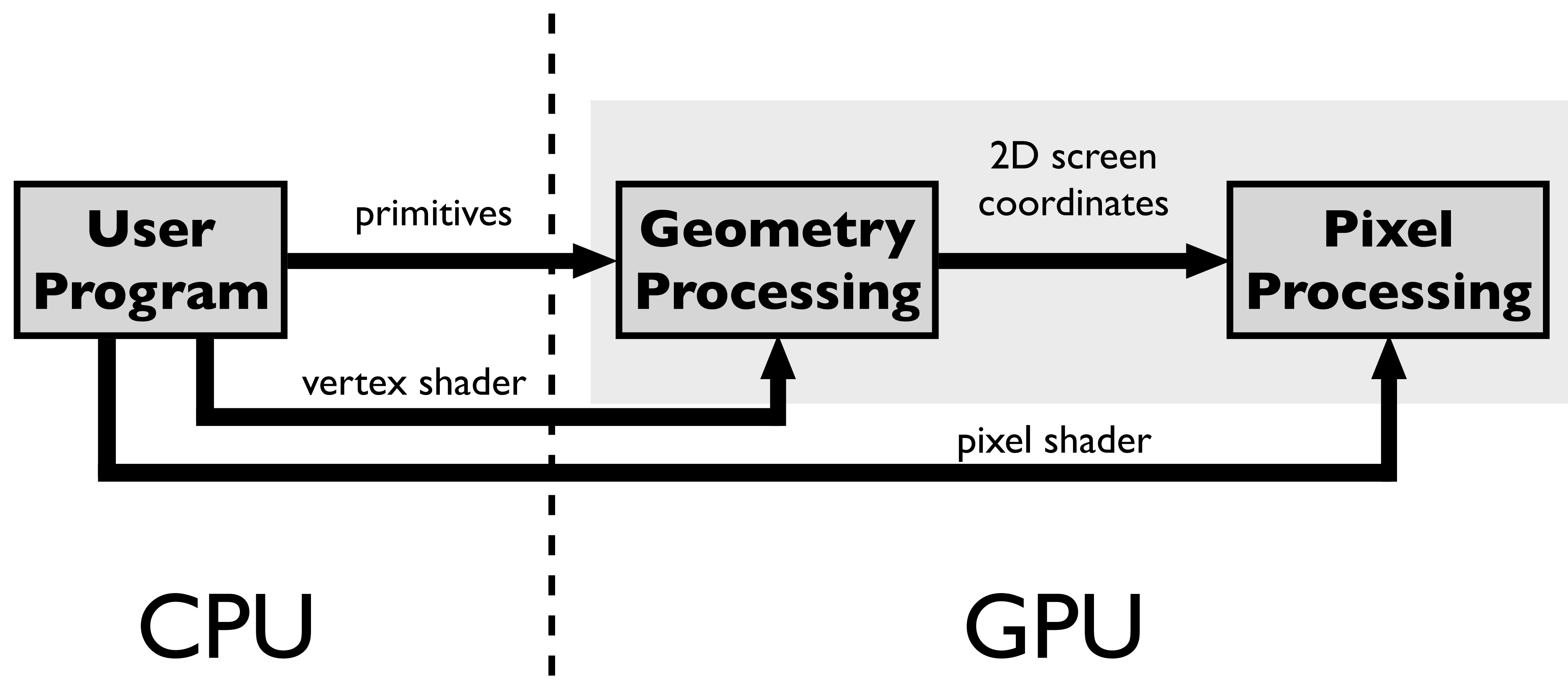
Fixed-Function Pipeline



Control pipeline through GL state variables

- The application supplies geometric primitives through a graphics API such as **OpenGL** or **DirectX**
- control of pipeline operation through state variables only

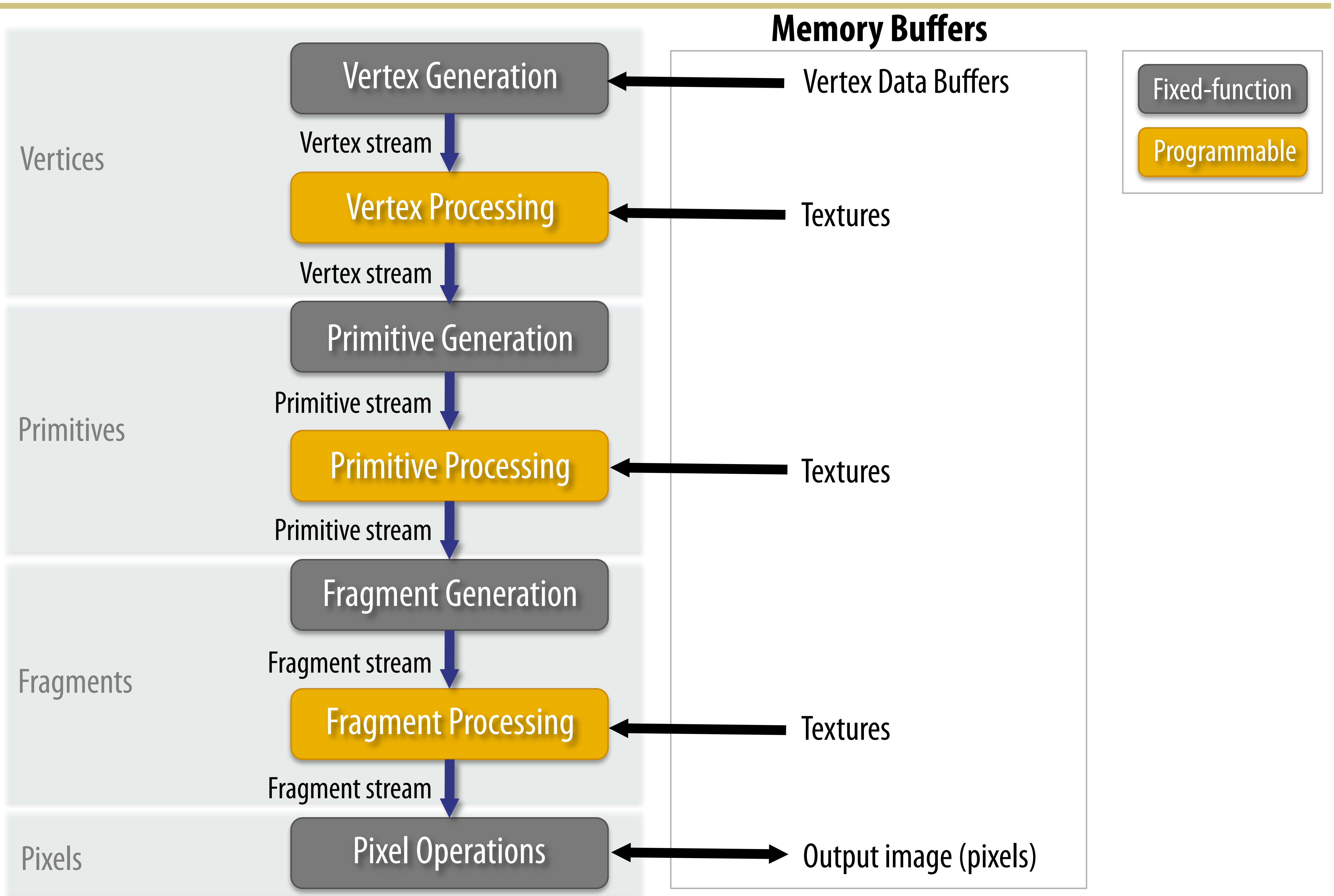
Programmable Pipeline



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

- can supply shader programs to carry out vertex processing, geometry processing, and pixel processing

Graphics pipeline



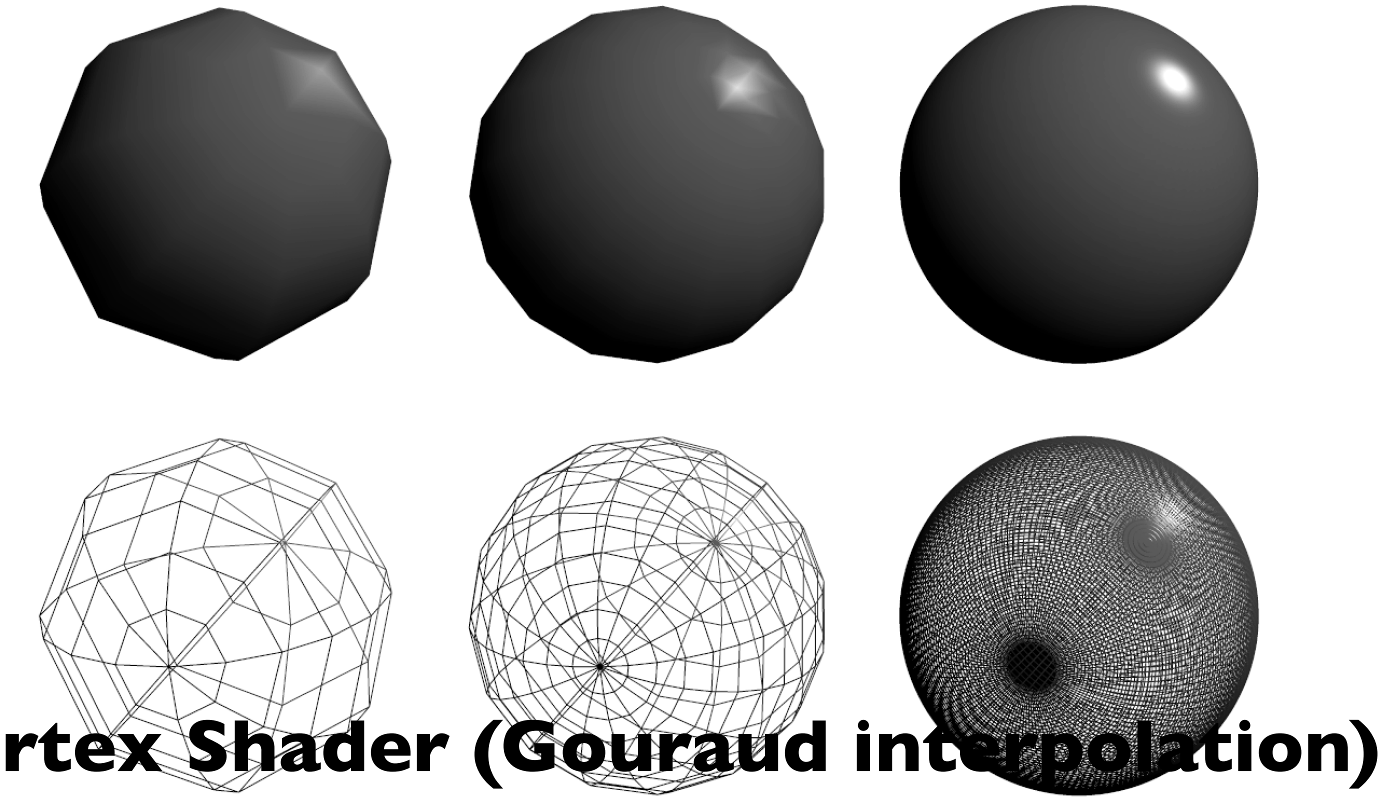
Phong reflectance in vertex and pixel shaders using GLSL

```
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();
}
```



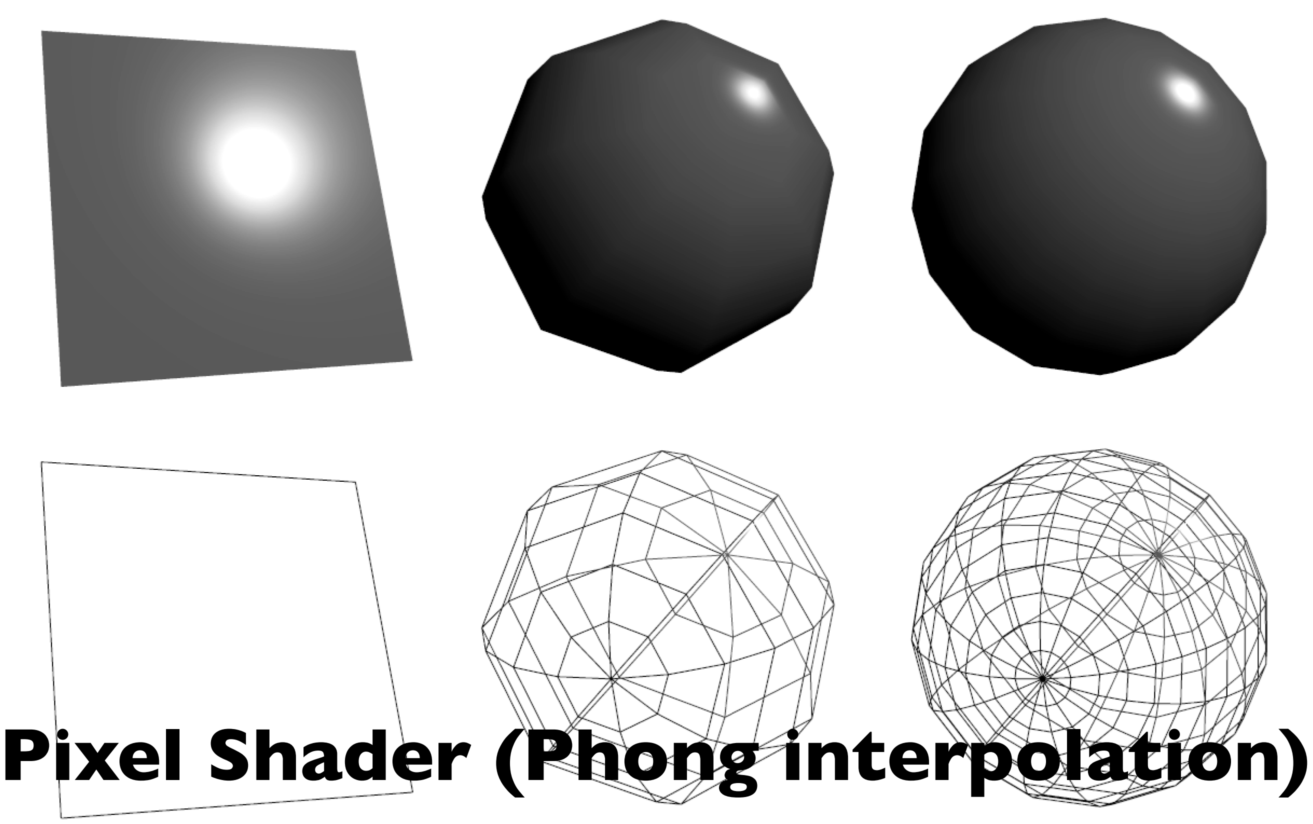

```
varying vec4 v;
varying vec3 n;

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;
}
```



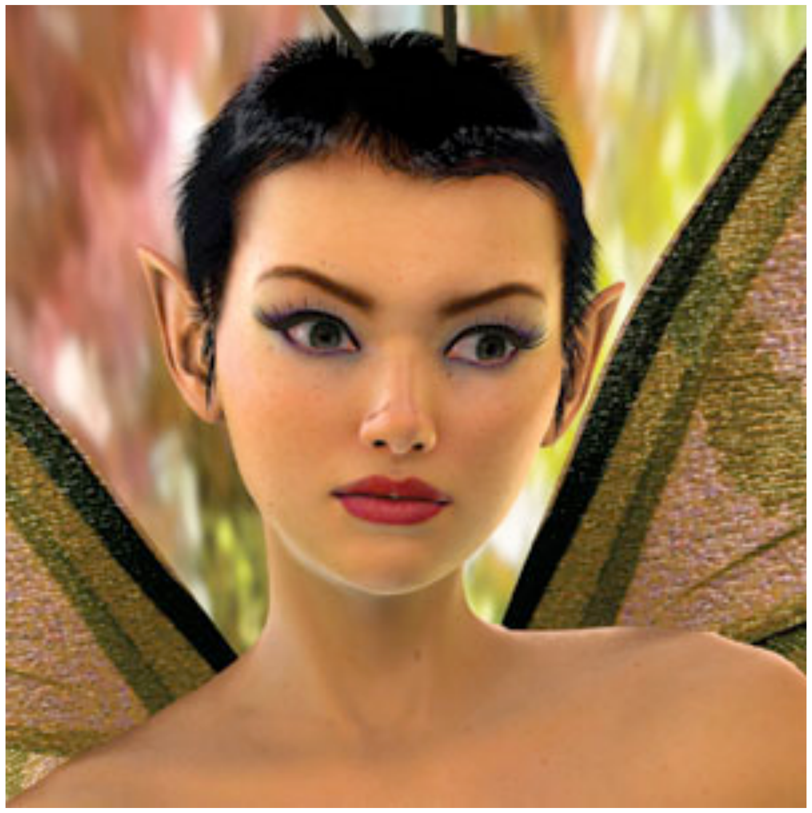
[Shirley and Marschner]

Phong reflectance as a vertex shader
- vertex shaders can be used to move/animate verts
- linear interpolation of vertex lighting

as a fragment shader
- each fragment is calculated individually - don’t know about neighboring pixles

Call of Juarez DX10 Benchmark, ATI

Rusty car shader, NVIDIA



Dawn, NVIDIA

Programmable shader examples from NVIDIA and ATI