Rasterisation & Blinn-Phong Lighting

Dr. Hamish Carr & Dr. Rafael Kuffner dos Anjos

Agenda

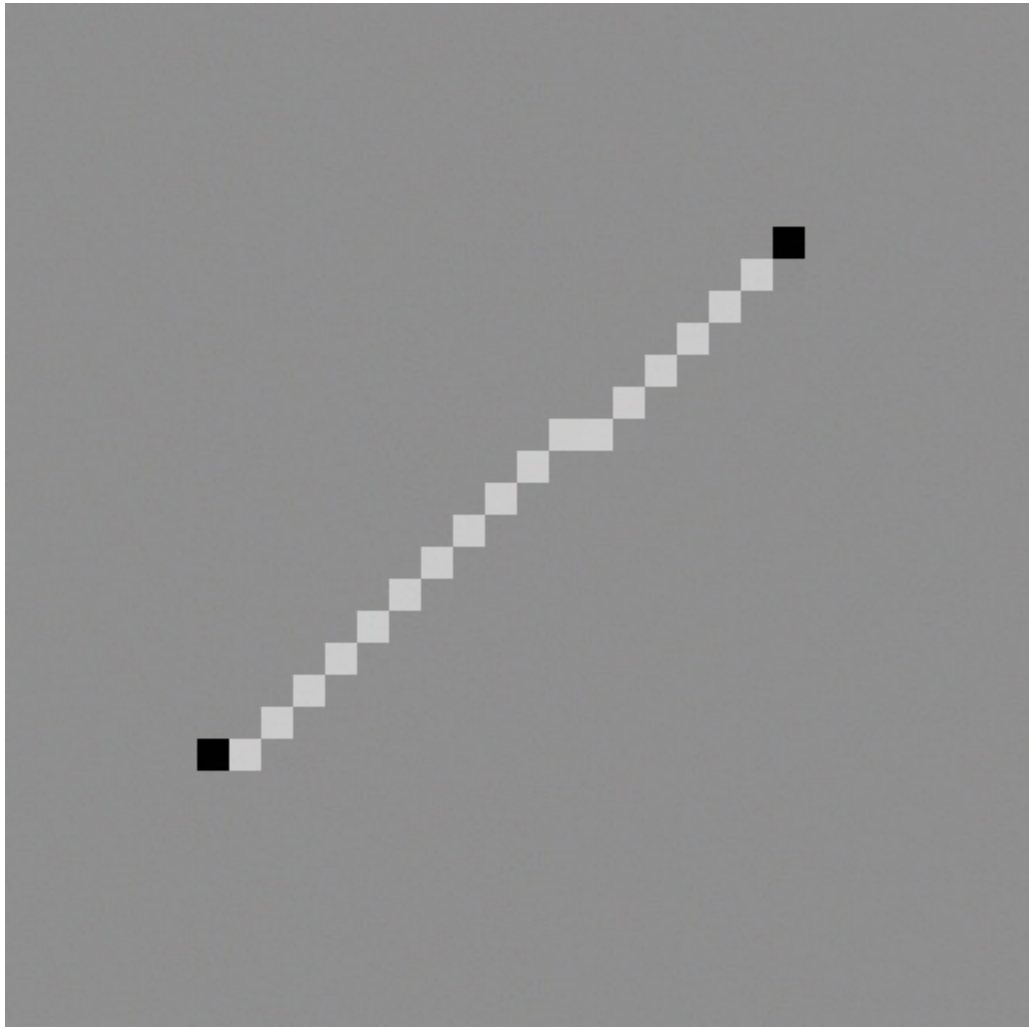
- Line Interpolation
- Triangle Interpolation
- Shading

How to draw a line?

- Bresenham's Algorithm
- Loop through explicit form with integer values:

```
for (x = x0; x < x1; x++)
        {
        y = mx + c;
        setPixel(x, y);
        }
```

- Rarely used anymore



UNIVERSITY OF LEEDS

Interpolation

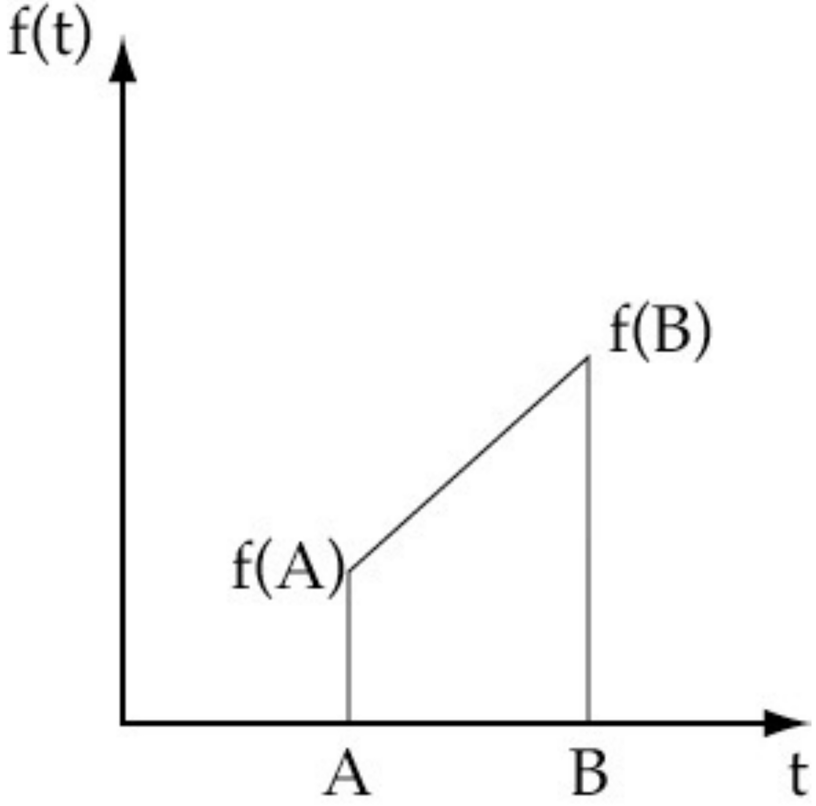
- Let's say we want a coloured gradient
  - At p, the line is 100% red, 0% blue
  - At q, the line is 0% red, 100% blue
  - In between, it varies smoothly
- This process is called *interpolation*

Linear Interpolation

- Let $f(x)$ be the colour
    - we use a straight line
    - f changes linearly:

$$f(t) = f(A) + \left(\frac{t - A}{B - A}\right)\left(f(B) - f(A)\right)$$

- So rasterise f at the same time as y



UNIVERSITY OF LEEDS

Implicit Form

- We want to draw a line 1 pixel *wide*
  - all pixels within 0.5 pixels of line
  - we know how to measure distance
- But this draws a *line,* not a *segment*

```
for (x = xMin; x < xMax; x++)
    for (y = yMin; y < yMax; y++)
        if (abs(distance((x, y), (x0, y0), (x1, y1))) < 0.5)
            setPixel(x, y);
```

Parametric Interpolation

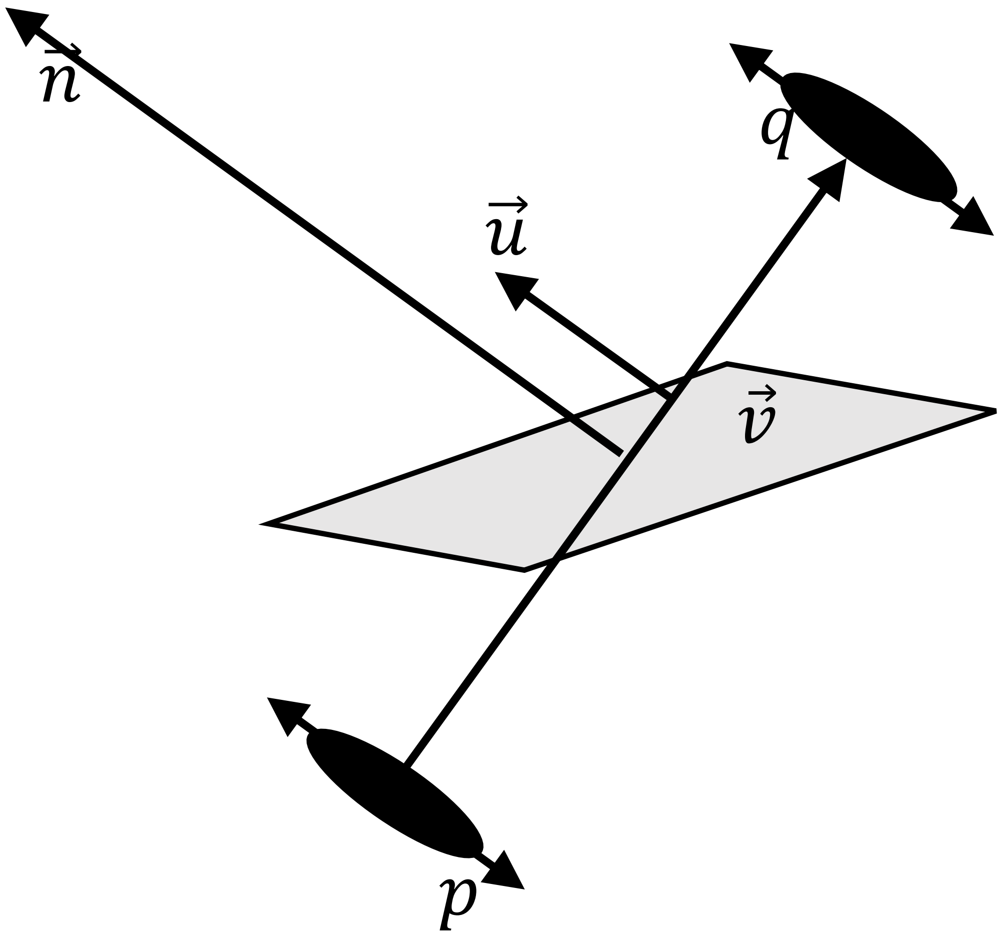
- Much easier
- Walk along the line one step at a time:

```
for (t = 0.0; t <= 1.0; t += 0.001)
    {
    point_r = point_p + (point_q - point_p) * t;
    colour = colour_p + (colour_q - colour_p) * t;
    setColour(colour);
    setPixel(round(r_x), round(r_y));
    }
```

- Choosing step size is important

Line Quads

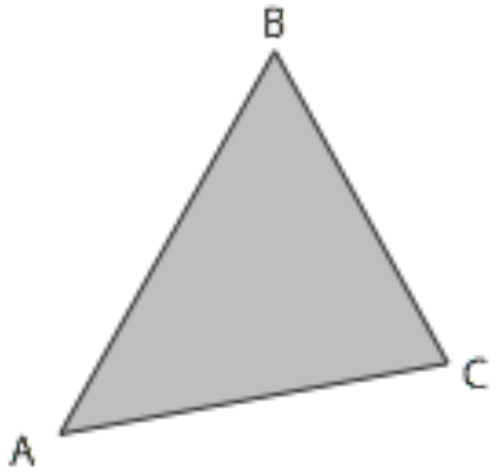
- Given points p, q
- Let $\vec{v} = q - p$
- Take normal $\vec{n} = (-v_y, v_x)$ (L1:S7)
- Normalise to a unit $\vec{u} = \frac{\vec{n}}{\|\vec{n}\|}$
- To draw a line of width w,
- Rasterise the quad:
  - $q - \frac{w}{2}\vec{u}, q + \frac{w}{2}\vec{u}, p + \frac{w}{2}\vec{u}, p - \frac{w}{2}\vec{u}$

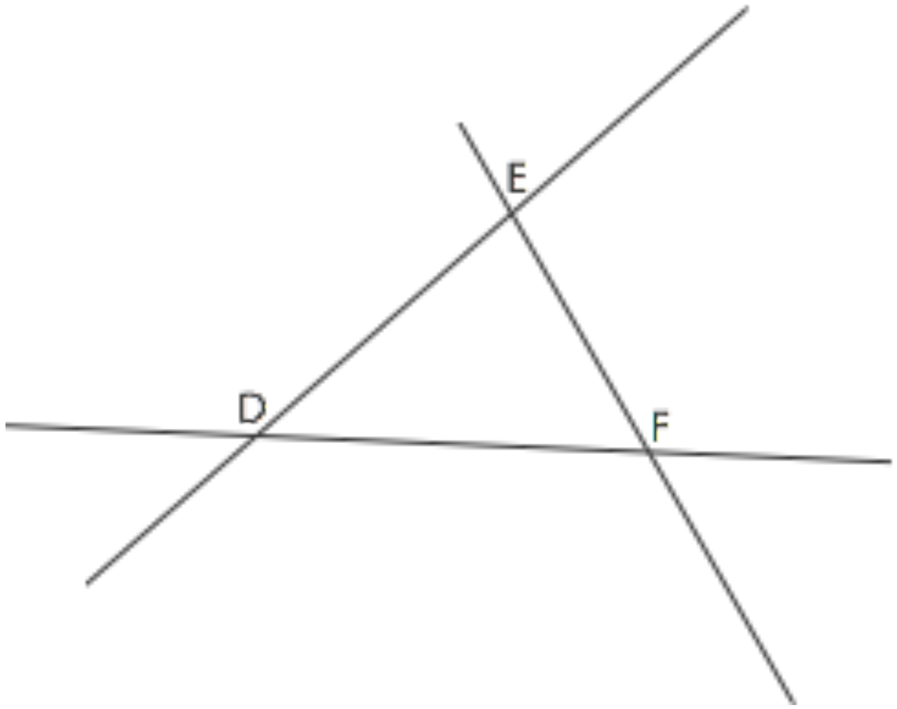


UNIVERSITY OF LEEDS

Triangles

- Defined by 3 points:
  - Or by 3 lines
- Drawing three lines is easy
- But what about *filled* triangles?
- Start with equations of triangles





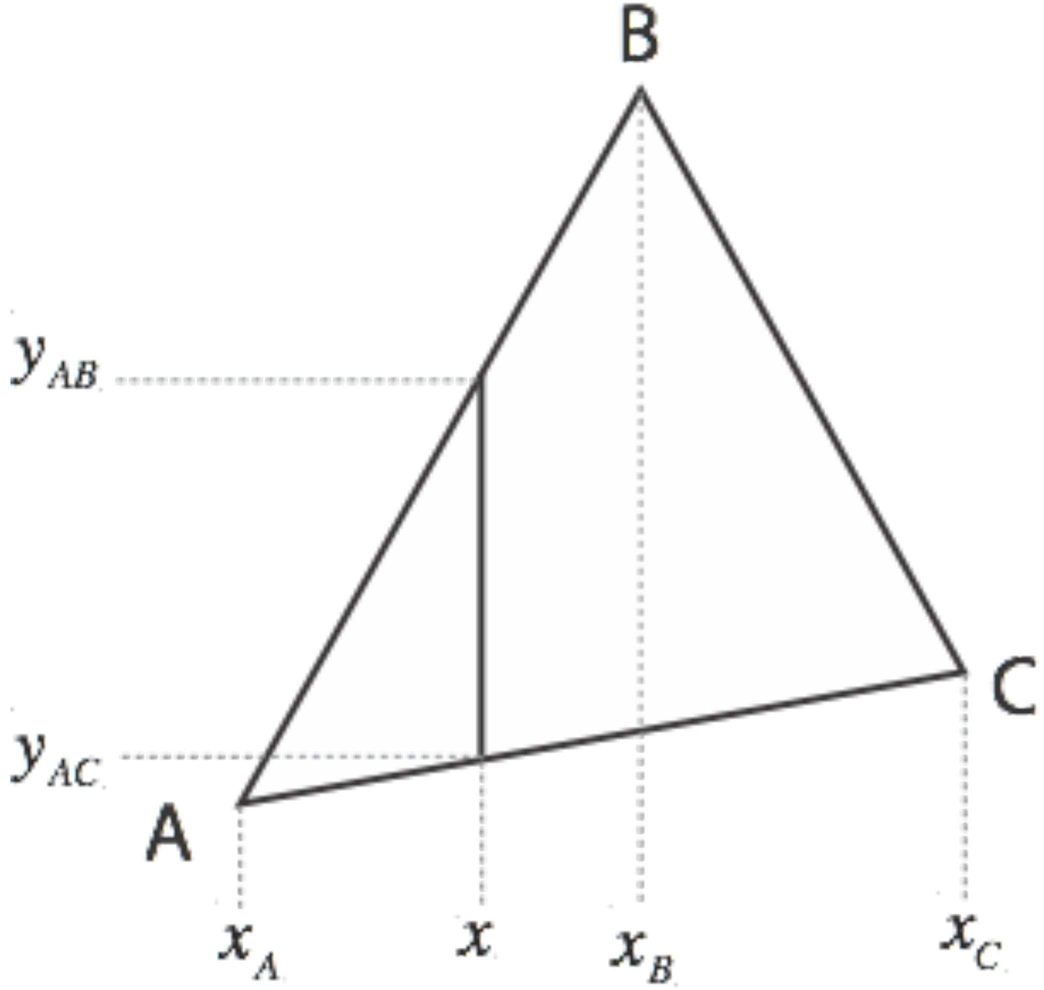
UNIVERSITY OF LEEDS

Explicit Form

- For any x, specify valid y

$$y_{AC} \le y \le y_{AB} \quad if \;\; x_A \le x \le x_B$$
$$y_{AC} \le y \le y_{BC} \quad if \;\; x_B \le x \le x_C$$

- Assumes B is above AC



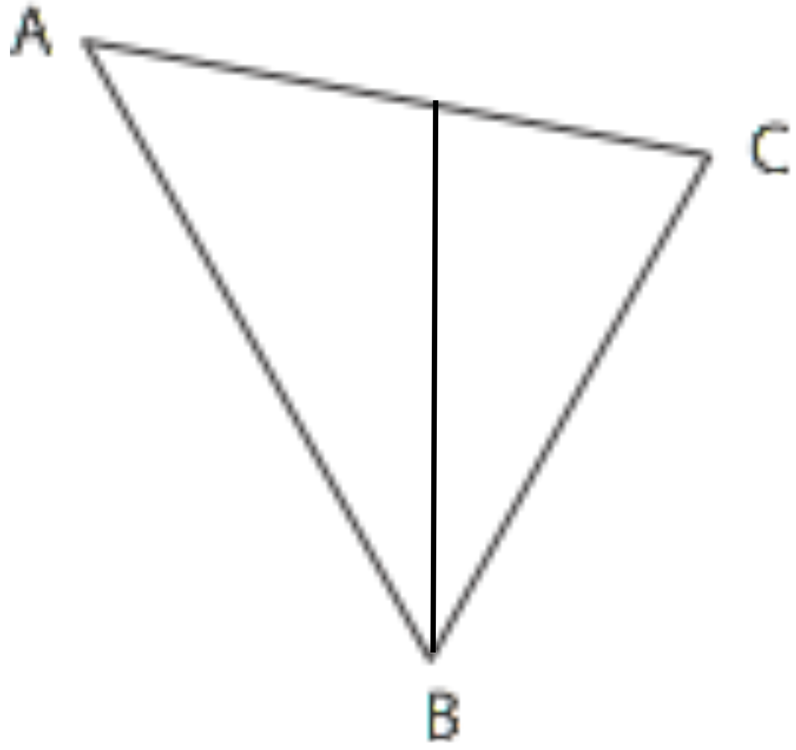
UNIVERSITY OF LEEDS

Explicit Form, II

- If B is below AC:

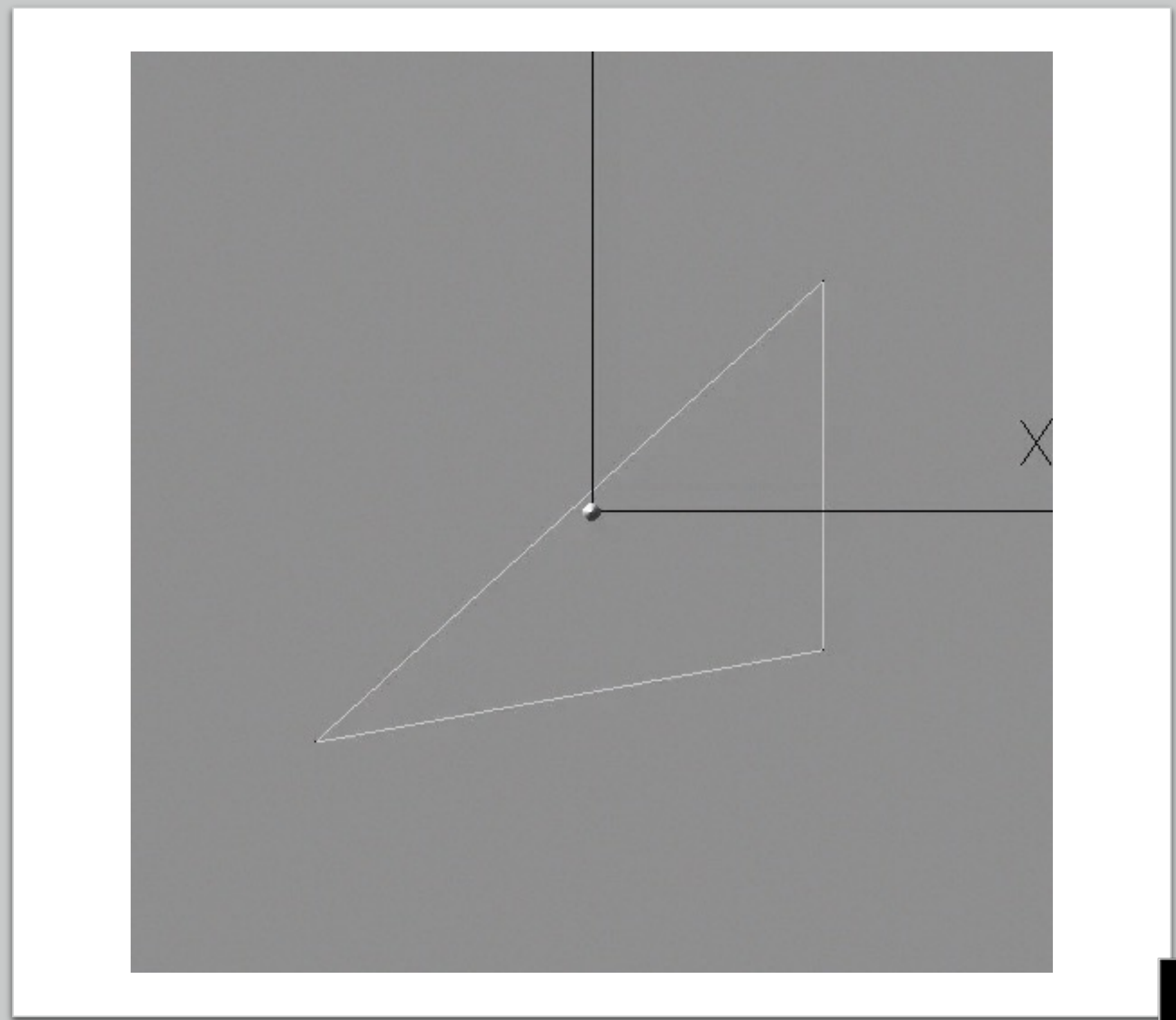
$$y_{AB} \le y \le y_{AC} \quad if \;\; x_A \le x \le x_B$$

$$y_{BC} \le y \le y_{AC} \quad if \;\; x_B \le x \le x_C$$



Raster Scan Algorithm

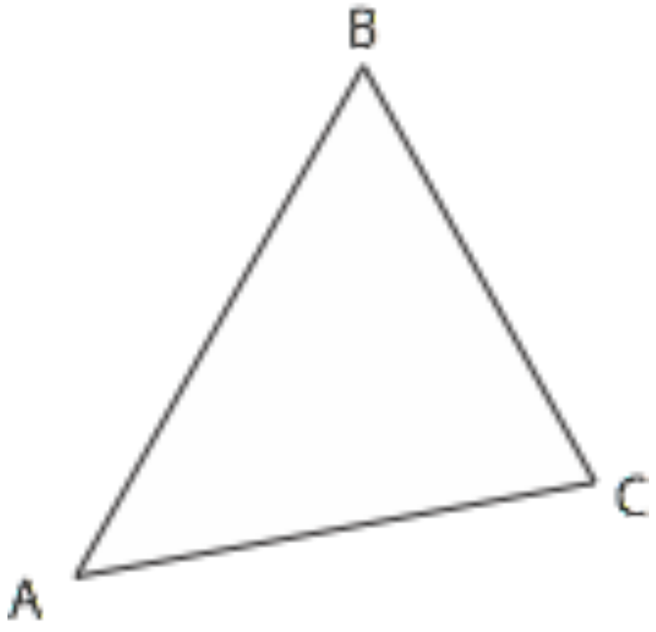
- Algorithm scans one line at a time
  - raster scan (raster is Latin for a rake)
  - scan conversion of triangles to pixels



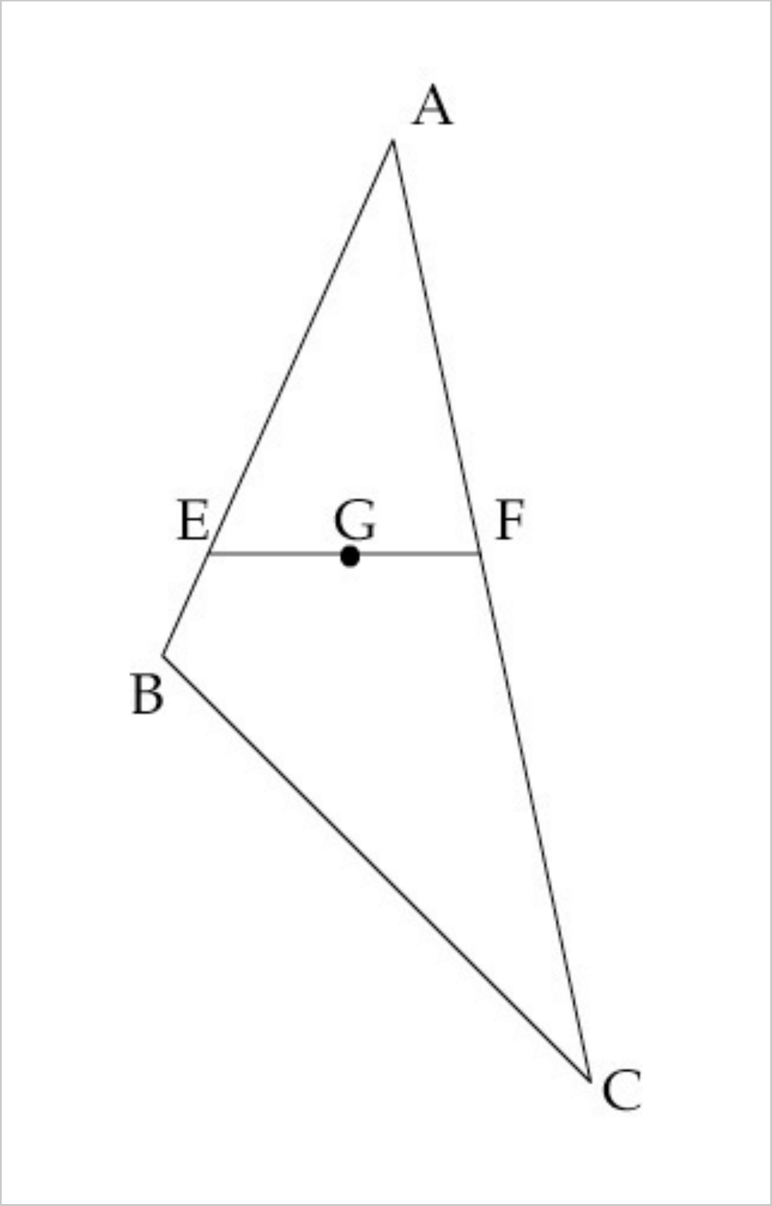
Explicit Algorithm

- Also called *linewise* scan or *raster scan*
  - usually loops horizontally, not vertically

```
Sort A, B, C so Ax < Bx < Cx
Find slopes mAB, mAC, mBC,
Find y-intercepts cAB, cAC, cBC
for (x = Ax; x <= Bx; x++)
  { // for each column
  yMin = mAC * x + cAC; yMax = mAB * x + cAB;
  if (yMin < yMax)
    swap(yMin, yMax);
  for (y = yMin; y <= yMax; y++)
      setPixel(x,y);
  } // for each column
```



UNIVERSITY OF LEEDS



Linewise Interpolation

- To compute *f(G)*:
  - Interpolate *f(E)* from *f(A)*, *f(B)*
  - Interpolate *f(F)* from *f(A)*, *f(C)*
  - Interpolate *f(G)* from *f(E)*, *f(F)*
- Perform for each of *R*,*G*,*B*

UNIVERSITY OF LEEDS

Implicit / Normal Form

- Based on *normal* form of lines:

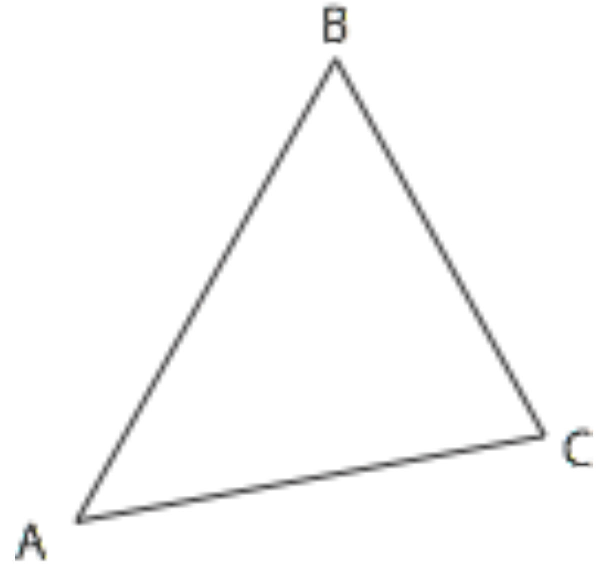
$$\vec{n} \cdot p - c = \begin{cases} - & \text{to } \textit{left} \text{ of line} \\ 0 & \textit{on} \text{ line} \\ + & \text{to } \textit{right} \text{ of line} \end{cases}$$

- Also known as the *half-plane test*

Look up L1:S10!

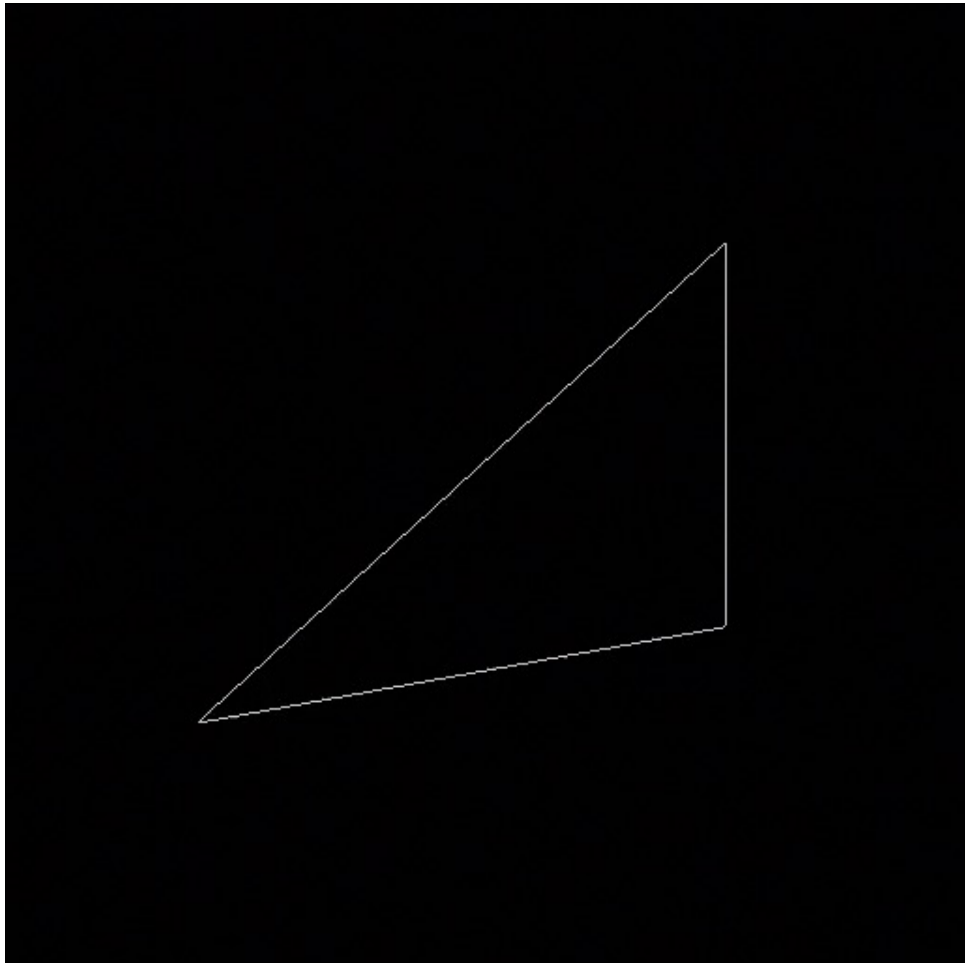
Winding Order

- *Inside* depends on the *winding order*
  - which direction we *wind*
  - ABC is clockwise (CW)
    - inside on right
  - ACB is counterclockwise (CCW)
    - inside on left



Half-Plane Test: What is inside the triangle?

- Each test divides plane in half:
  - Red vs. Not-Red
  - Green vs. Not-Green
  - Blue vs. Not-Blue
- Triangle is *inside* each



Implicit Algorithm

- Assume CCW winding order (left is inside)

```
for (x = xMin; x < xMax; x++)
    for (y = yMin; y < yMax; y++)
        if (  (x,y) leftOf (A,B) &&
              (x,y) leftOf (B,C) &&
              (x,y) leftOf (C,A))
            setPixel(x,y);
```

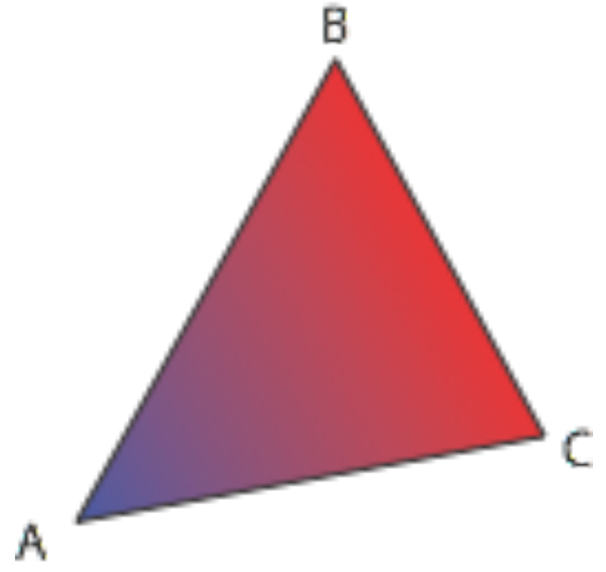
- But what about colour interpolation?
    - As with lines, we need *parametric* form

Parametric Form

- For a line pq, $t = 0.0$ at p, $t = 1.0$ at q
- How can we parameterize a triangle for interpolation?

Triangle Interpolation

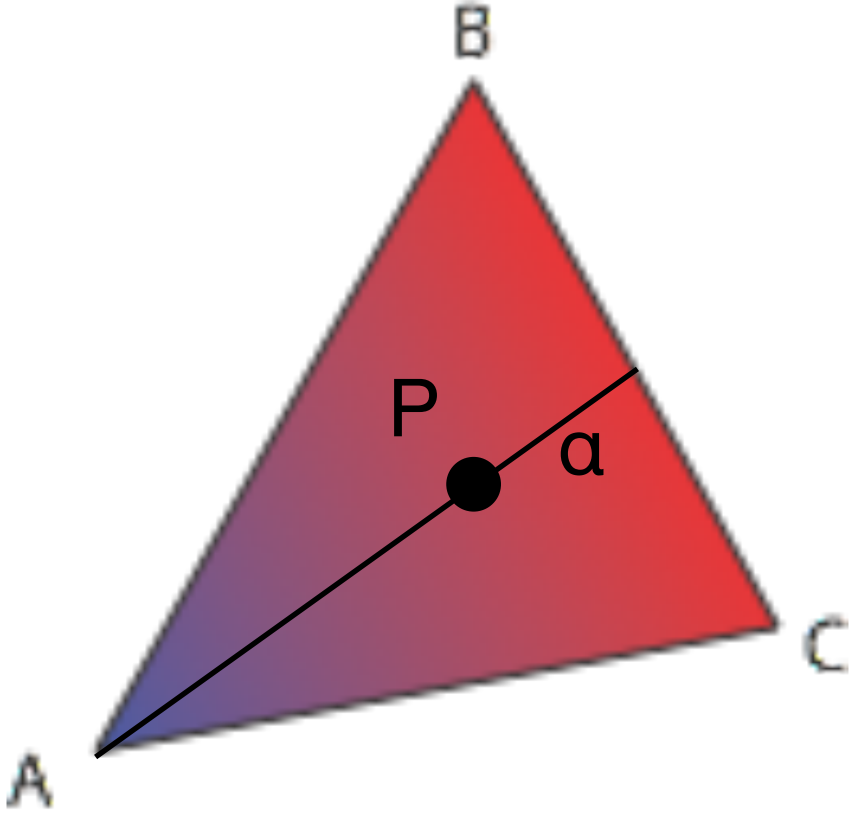
- Pick a vertex *A*
  - Set 100% blue at A
  - Set 0% blue at CB
- In between, varies *linearly*
  - perpendicular to CB



The Parameter α

- Colour depends on *distance* from CB
- Call this distance α
  - Parametrize so that:
- $\alpha = 1.0$ at A
- $\alpha = 0.0$ at BC

$$\alpha = \frac{dist(P, CB)}{dist(A, CB)}$$



And, obviously...

For any point P

$$\alpha = \frac{dist(P,CB)}{dist(A,CB)}$$

$$\beta = \frac{dist(P,AC)}{dist(B,AC)}$$

$$\gamma = \frac{dist(P,BA)}{dist(C,BA)}$$

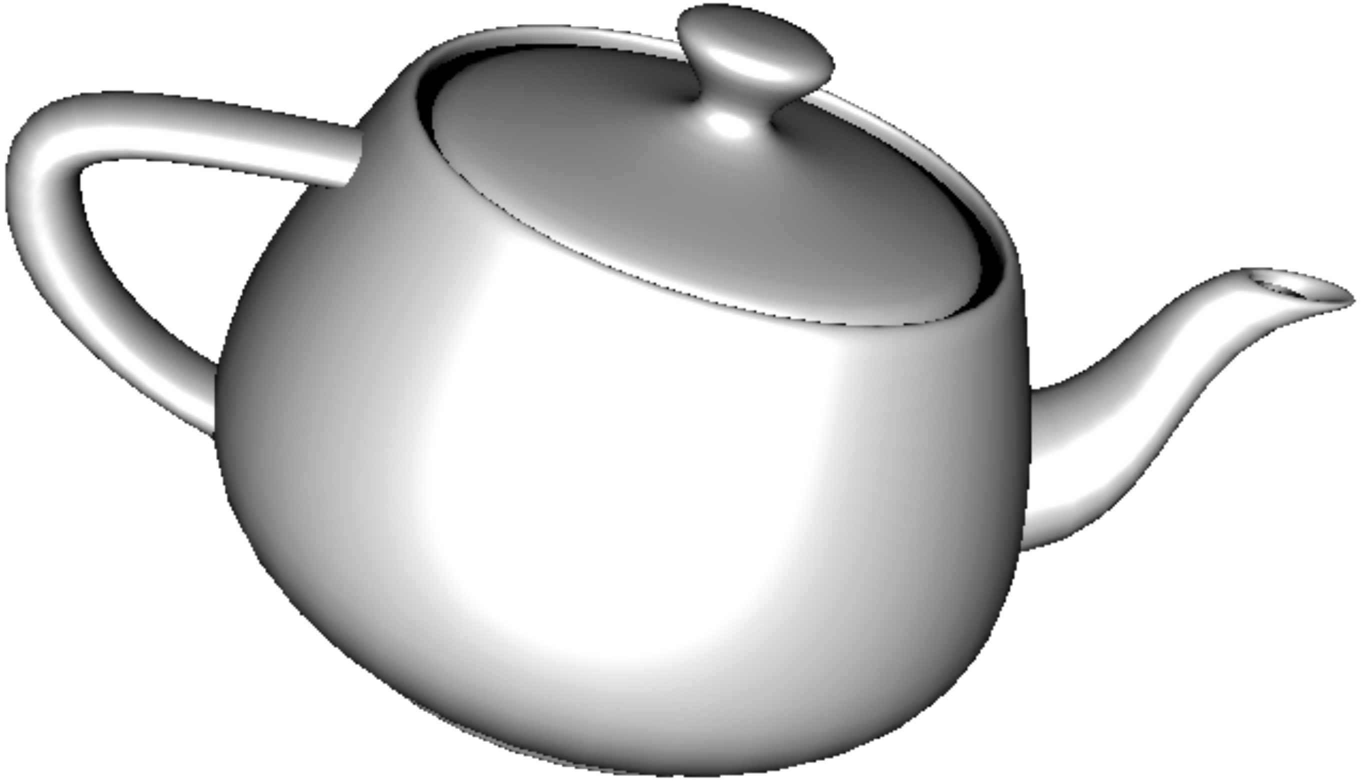
Barycentric Coordinates

- α, β, γ are called *barycentric coordinates*
- Conveniently, $\alpha + \beta + \gamma = 1.0$
- We only have two parameters!
- But we have three weights: interpolate between 3 vertices (color, normal, textures etc)

Parametric Algorithm

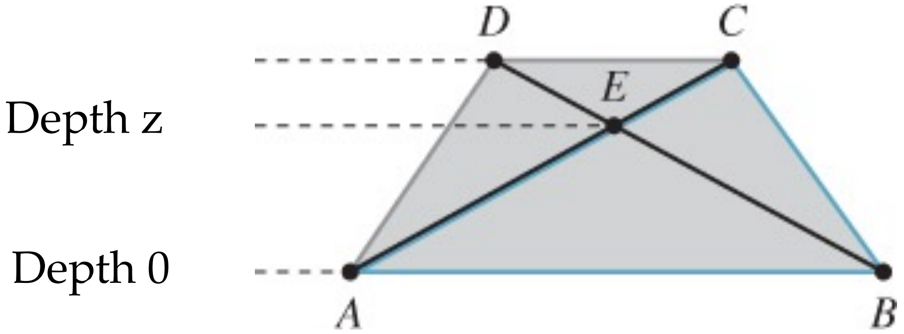
```
for (x = xMin; x < xMax; x++)
    for (y = yMin; y < yMax; y++)
        alpha = distance((x,y), BC) / distance(A, BC);
        beta = distance((x,y), AC) / distance(B, AC);
        gamma = distance((x,y), AB) / distance(C, AB);
        if ((alpha < 0.0) || (beta < 0.0) || (gamma < 0.0))
            continue;
        colour = alpha * colour(A) + beta * colour(B)
                  + gamma * colour(C);
        setColour(colour);
        setPixel(x,y);
```

Shading



Perspective Interpolation

- Assume ABCD is a square
- E is *not* half-way between B & D visually
- But it needs to be mathematically
- So we have to correct it for the depth z

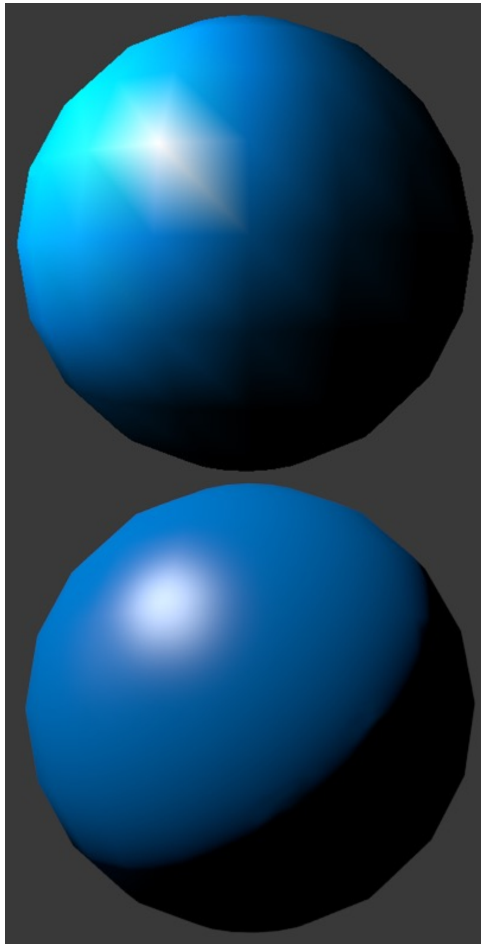


Hyperbolic Interpolation

- Do perspective division on the value u
  - $colour' = colour/z$ (for each vertex)
- Interpolate both *colour* and *1/z*
- Then reconstruct u:
  - $colour = \frac{colour\prime}{\frac{1}{z}}$ (using interpolated values)
- Usually done with the linewise raster scan
  - Otherwise it gets really messy

Gouraud vs. Phong Shading

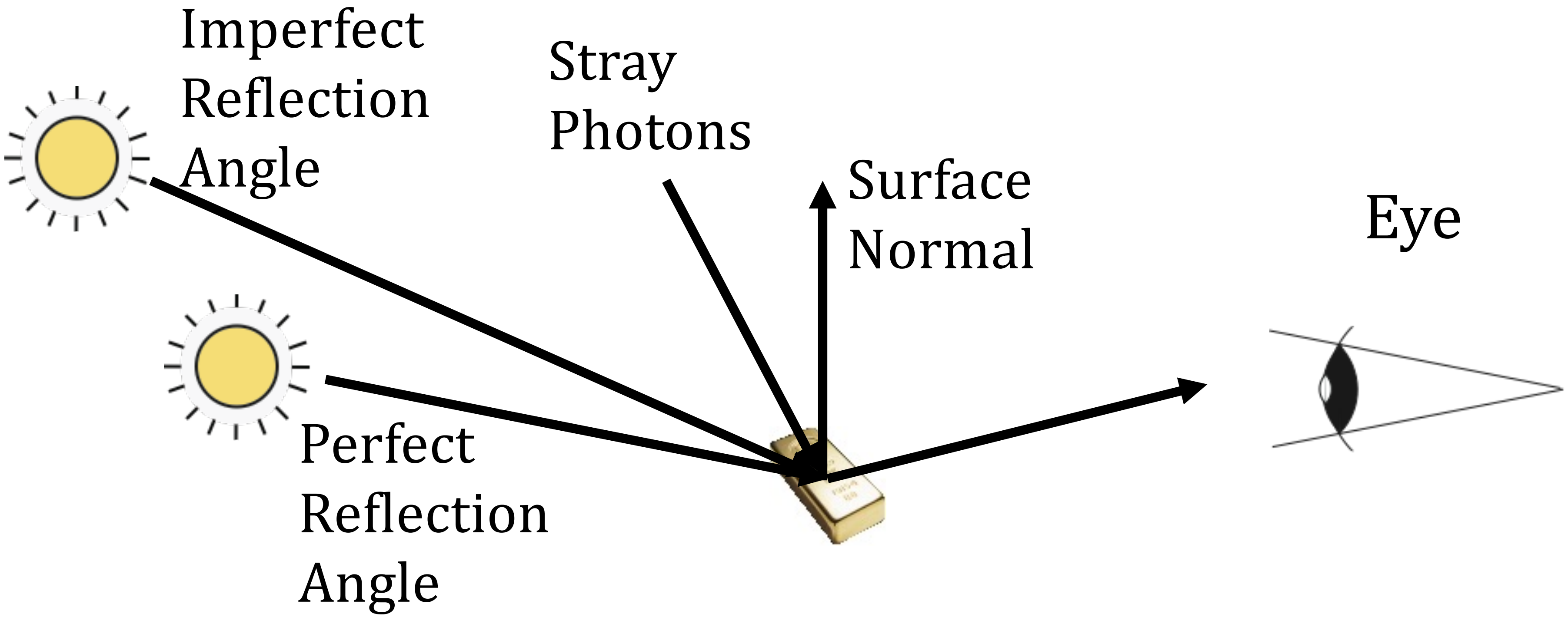
- Gouraud shading computes light per vertex
  - Then interpolates *colour* across triangle
  - Standard (old) OpenGL solution
- Phong shading computes light per point
  - Interpolates *normal* across triangle
  - And is more expensive computationally



Local & Global Illumination

- Global illumination simplifies lighting
  - All surfaces get the same lighting
  - I.e. the light interacts with *every* surface
    - Other objects don’t block the light source
    - And indirect lighting is ignored
- Local illumination is better, but more costly
  - So we will use global illumination

Origin of Photons



- For any point, light comes from all over

UNIVERSITY OF LEEDS

Blinn-Phong Lighting Model

- Total lighting at a point is:
  - specular (shiny) reflection, plus
  - diffuse (matt) reflection, plus
  - ambient (background) reflection, plus
  - emitted light

$$I_{total}(p) = I_{specular}(p) + I_{diffuse}(p) + I_{ambient}(p) + I_{emitted}(p)$$

Emitted Light

- Light from a glowing object
- For simplicity, uniform in all directions
- Not affected by incoming light

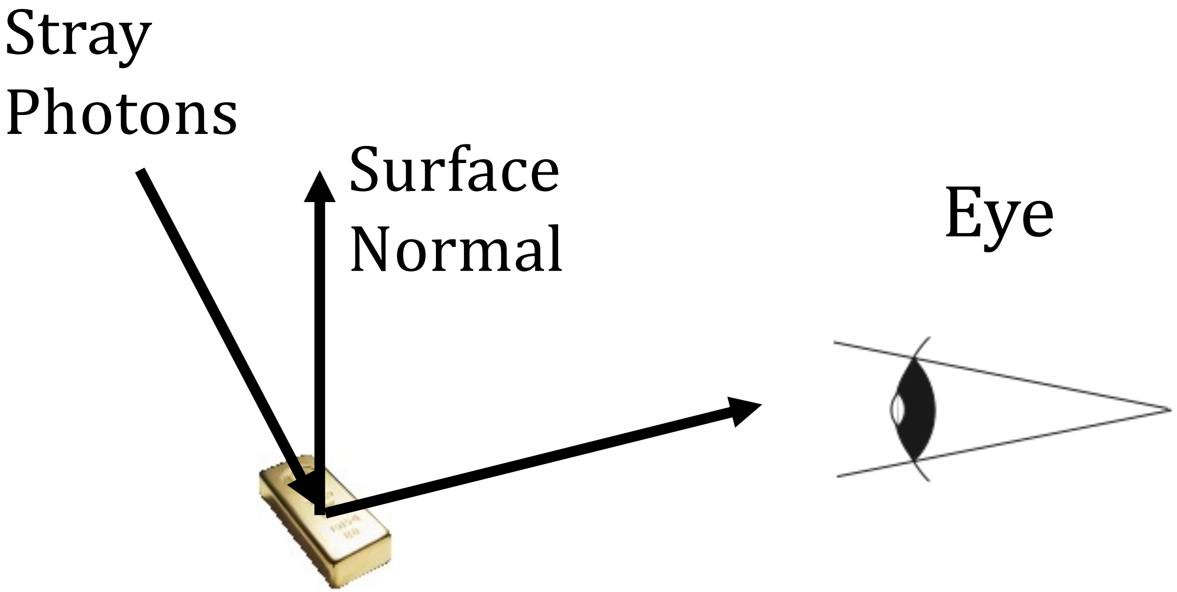
$$I_{emitted}(p) = l_{emitted}$$

$$l_{emitted} = \text{emitted intensity of light}$$

UNIVERSITY OF LEEDS

Ambient Lighting

- Some photons have bounced around
  - Hard to identify their source
  - Roughly same number everywhere



UNIVERSITY OF LEEDS

$$I_{ambient}(p) = l_{ambient} r_{ambient}$$

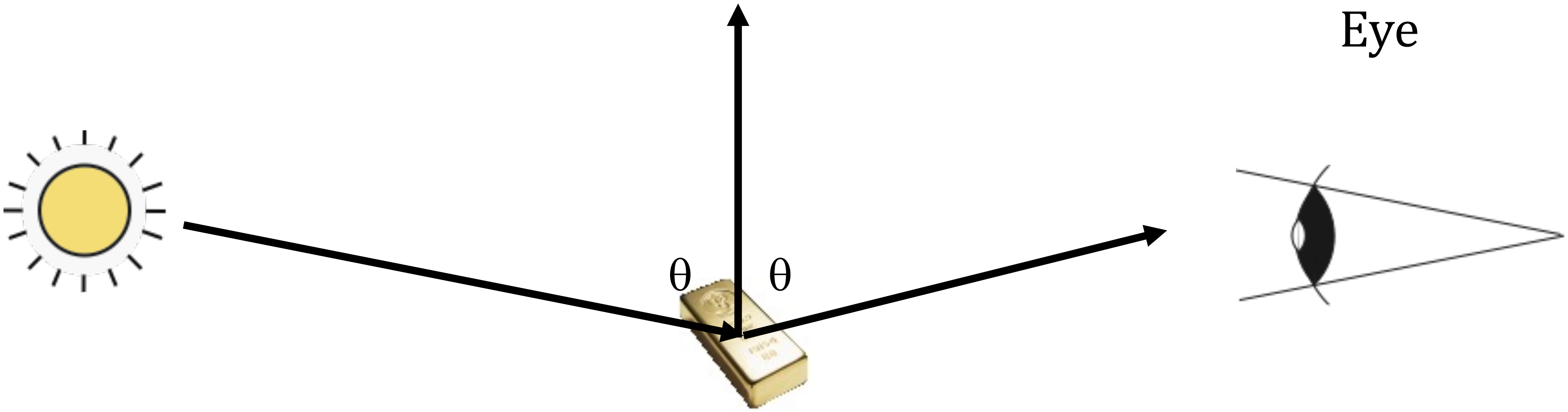
$$l_{ambient} = \text{ambient intensity of light}$$

$$\mathrm{r}_{ambient} = \text{ambient reflectivity (albedo) of surface}$$

Ambient Light: *Uniform on all surfaces,* but some reflect more

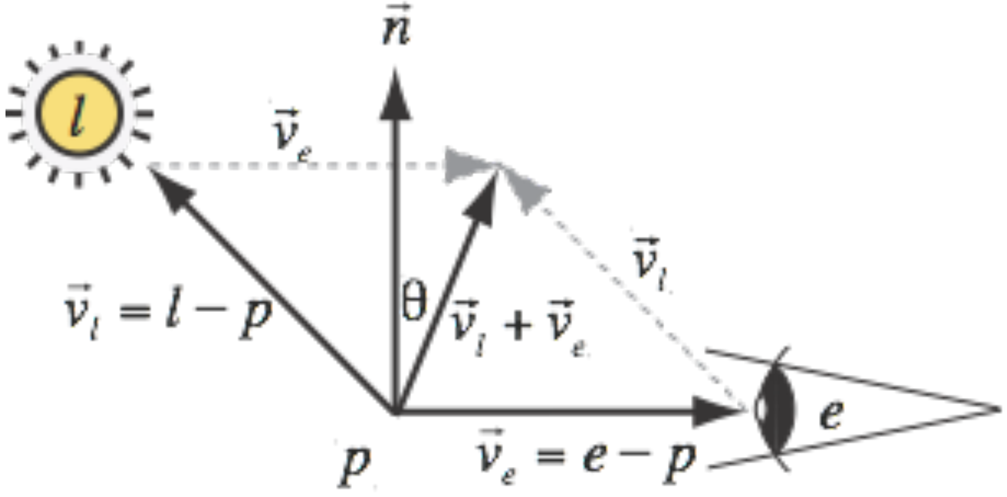
Perfect Reflection

- Angle of incidence = angle of reflection
- Normal vector is bisector



Specular Reflection

- Specular light spreads out a little bit
- Reflects strongly for angles close to perfect
  - i.e. if the bisector is close to *n*



UNIVERSITY OF LEEDS

Specular Reflection

- Based on angle between normal and bisector

$$\cos\theta = \frac{\vec{n}\bullet(\vec{v}_b)}{\|\vec{n}\|\|\vec{v}_b\|}, \vec{v}_b = \frac{\vec{v}_l + \vec{v}_e}{2}$$

- Use an exponent to adjust size of highlight

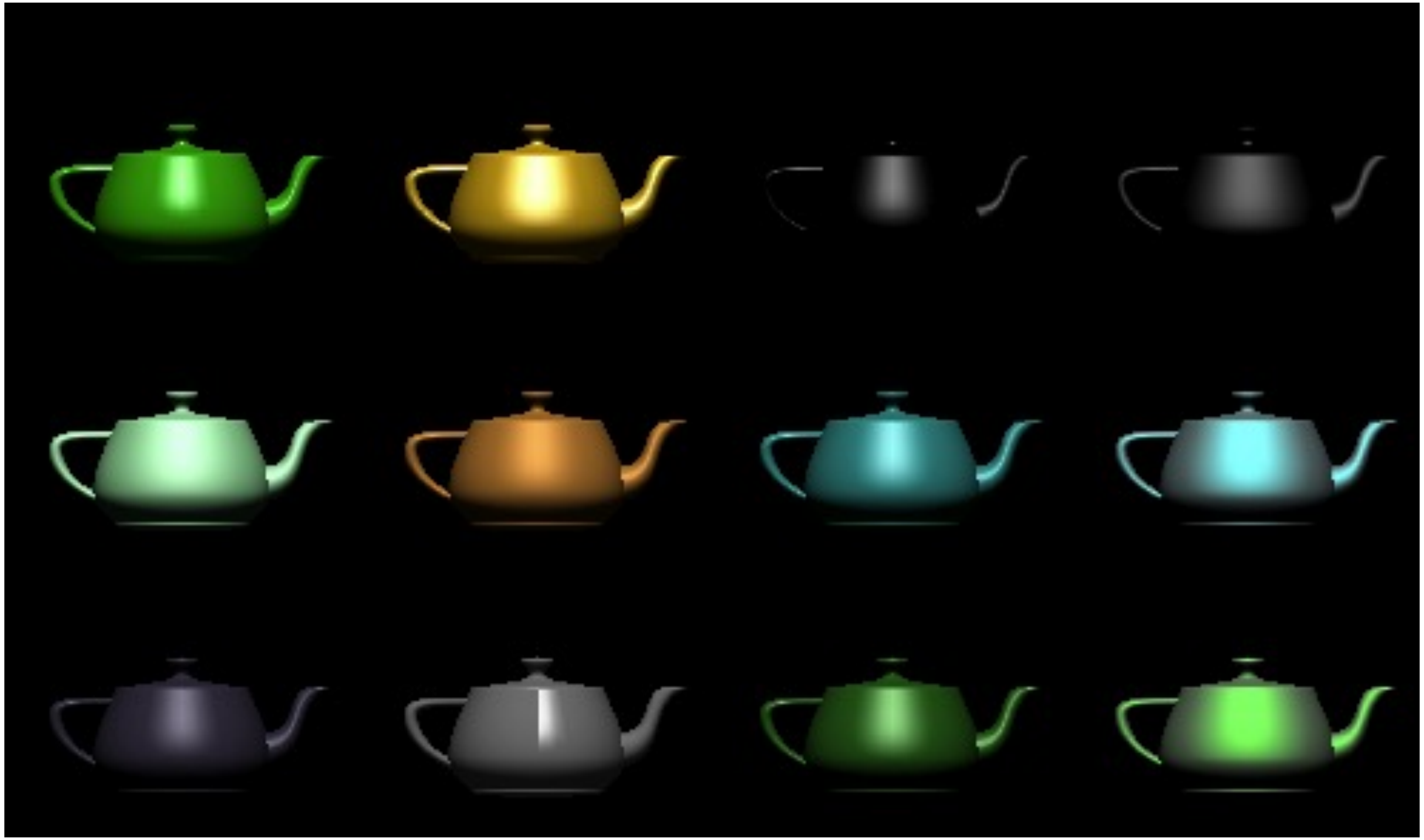
$$I_{specular}(p) = l_{specular} r_{specular} \left(\frac{\vec{n}\bullet(\vec{v}_b)}{\|\vec{n}\|\|\vec{v}_b\|} \right)^{h_{specular}}$$

$$l_{specular} = \text{specular intensity of light}$$

$$\mathrm{r}_{specular} = \text{specular reflectivity (albedo) of surface}$$

$$\mathrm{h}_{specular} = \text{specular highlight coefficient}$$

Specular Highlights



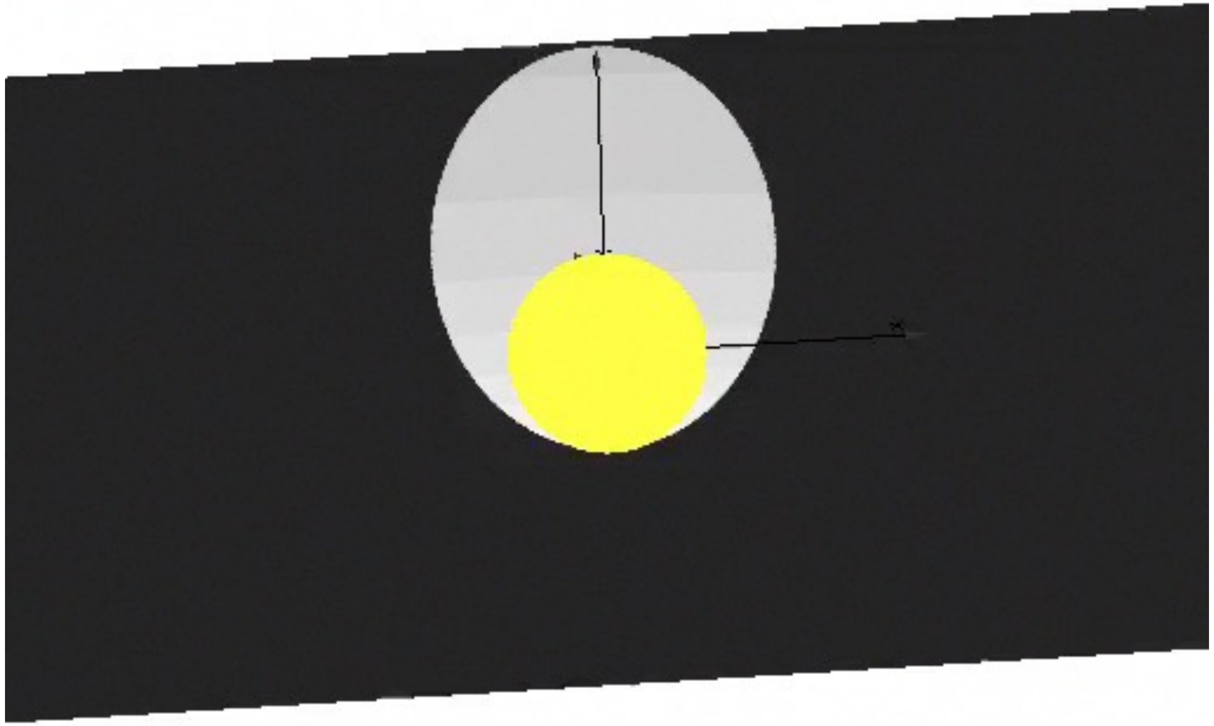
Diffuse Light

- Diffuse light is from rough surfaces
  - rough at the microscopic scale
  - normal is essentially random
    - although surface is oriented
- Diffuse light still uses normal vector

UNIVERSITY OF LEEDS

Diffuse Lighting

- At large angles, light is more spread out
- Light per unit area proportional to $\cos \theta_i$
- But not to $\cos \theta_r$



Diffuse Computation

- Light is spread over surface
  - depending on incident angle
  - but not on reflection angle

$$
\begin{aligned}
I_{diffuse}(p) &= l_{diffuse} r_{diffuse} \cos\theta_i \\
&= l_{diffuse} r_{diffuse} \frac{\vec{n} \bullet \vec{v}_l}{\|\vec{n}\| \|\vec{v}_l\|}
\end{aligned}
$$

$l_{diffuse}$ = diffuse intensity of light

$\mathrm{r}_{diffuse}$ = diffuse reflectivity (albedo) of light

Putting it Back Together

For colour, do this once each for R,G,B

$$I_{total}(p) = I_{specular}(p) + I_{diffuse}(p) + I_{ambient}(p) + I_{emitted}(p)$$

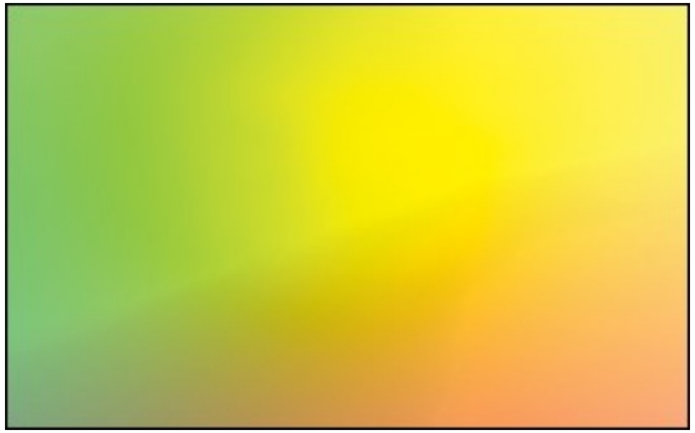
$$= l_{specular} r_{specular} \left(\frac{\vec{n} \bullet (\vec{v}_b)}{\left\| \vec{n} \right\| \left\| \vec{v}_b \right\|} \right)^{h_{specular}}$$

$$+ l_{diffuse} r_{difffuse} \frac{\vec{n} \bullet \vec{v}_l}{\left\| \vec{n} \right\| \left\| \vec{v}_l \right\|}$$

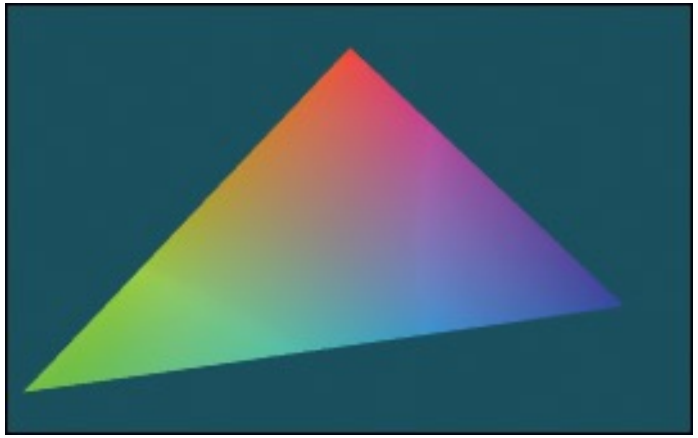
$$+ l_{ambient} r_{ambient}$$

$$+ l_{emitted}$$

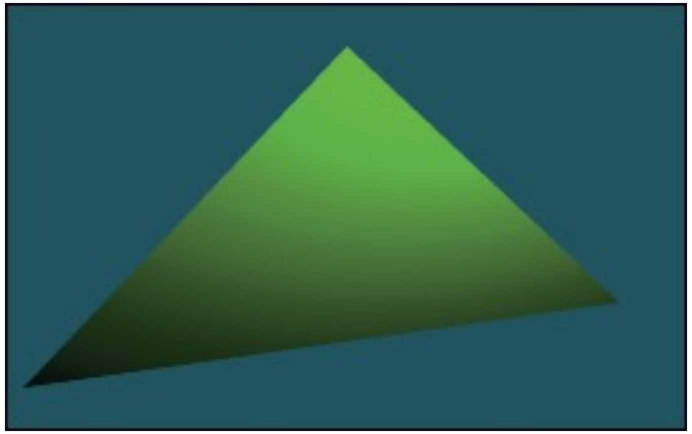
One Step At A Time



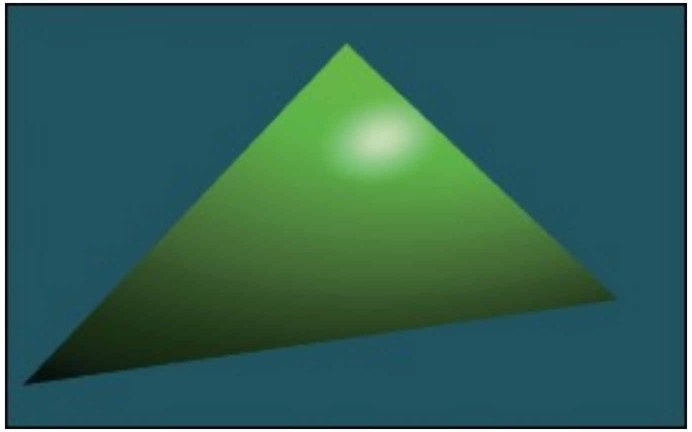
RGB ~ i,j (debug)



RGB ~ αβγ (debug)



Lambertian (diffuse)



Blinn-Phong

Saving to PPM File

- A common convention when testing
  - Dump the image to a file
  - PPM is a very simple text format
  - Inefficient, but simple
  - Simplifies the debug cycle
- But only allows integer values
  - Usually in the range 0-255

Debug hint for A1!!

Images by

- S19 - Apurv das
- S32- Photographycourse
- S33 - Jake Givens
- S39 - Greg Rosenke
- From Unsplash.com