Rasterization and Shading

Slides by Sarah Jobalia and Kevin Li, with modifications by Kate Baker

Housekeeping

- **HW1** is due on Thursday in live grading format
  - See the course website for more details + Zoom/queue links
  - Please submit on gradescope, too!

- Finalized grading times (Thursday)
  - Nicole: 3:00pm-5:15pm
  - Lvmin: 5:30pm-8:30pm

- Check website for office hours times / links

Revisiting Questions

- Metal vs OpenGL
  - Apple deprecated OpenGL in factor of Metal
  - Deprecated APIs remain present/usable for a while, active development will cease
  - Metal = similar pipeline to OpenGL, just has more control over GPU (memory, resource management, etc.)

Revisiting Questions

- Metal vs OpenGL
  - Apple deprecated OpenGL in factor of Metal
  - Deprecated APIs remain present/usable for a while, active development will cease
  - Metal = similar pipeline to OpenGL, just has more control over GPU (memory, resource management, etc.)

- Loop subdivision processing
  - When you compute old/new vertices, use the original (pre-subdivision) positions of the neighboring vertices

Last time…

- Representation of 3D geometry (meshes, splines, etc.)
- Now we have ways to store, make, and describe objects!

- Next step: how do we make these objects show up on the screen?

Lecture Outline

- How to draw a triangle
  - Image Representation
  - Rasterization
- How to color a triangle
  - Surface normals, shading, barycentric interpolation
- The OpenGL pipeline
  - Vertex vs. fragment shaders

Lecture Outline

- How to draw a triangle
  - Image representation
  - Rasterization
- How to color a triangle
  - Surface normals, shading, barycentric interpolation
- The OpenGL pipeline
  - Vertex vs. fragment shaders

Image Representation

- Images: represented as a 2D grid of colors → pixels

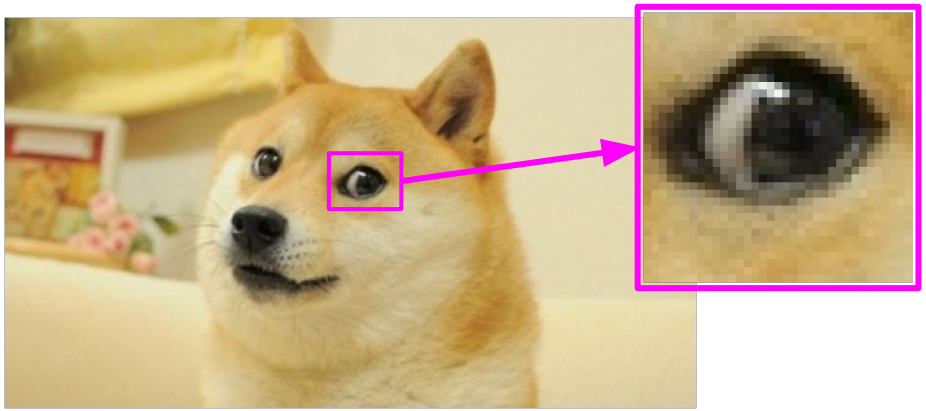
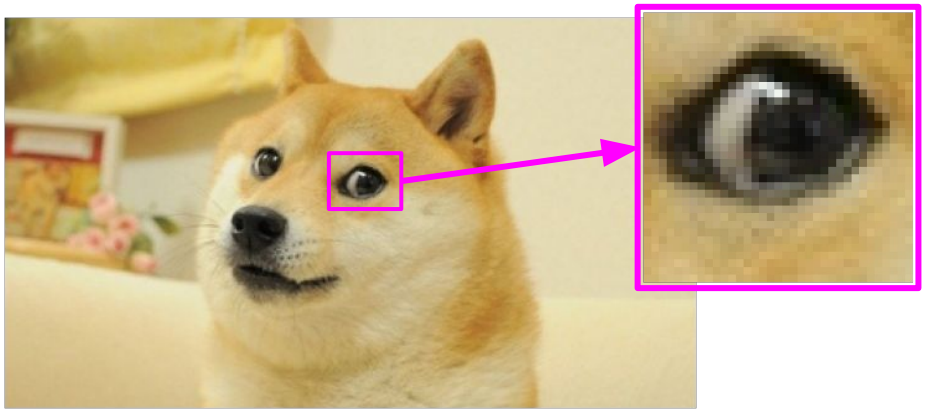
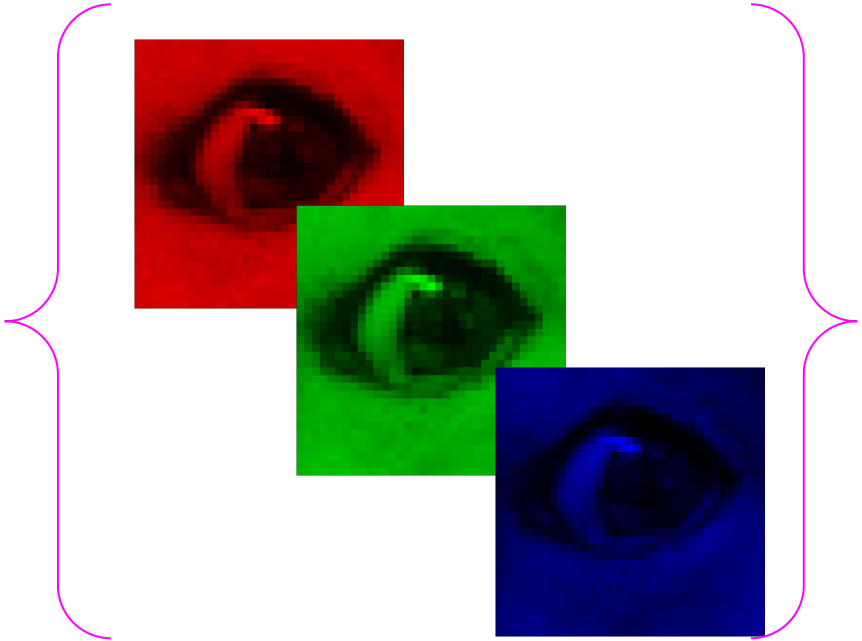


Image Representation

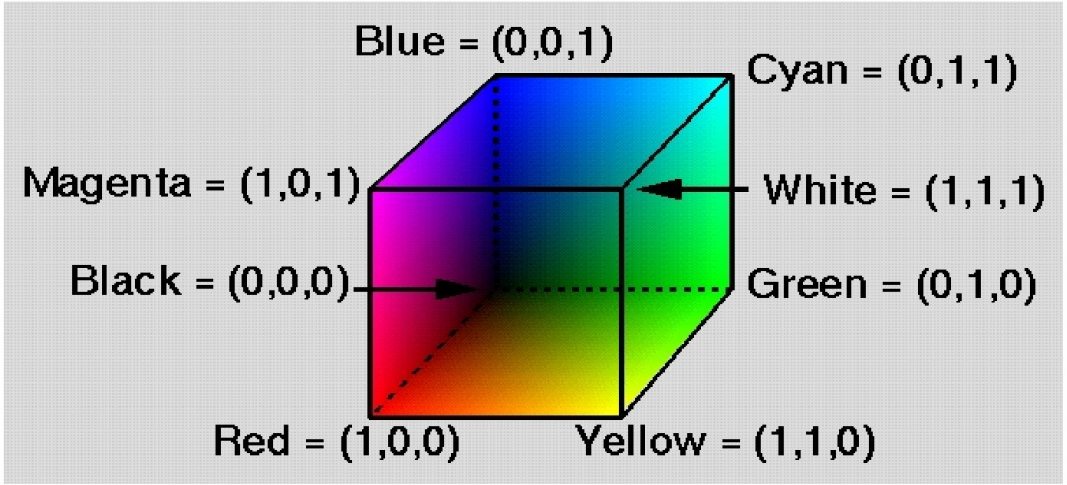
- Images: represented as a 2D grid of colors → pixels
- Often in RGB values
  - 1 grid for each color channel

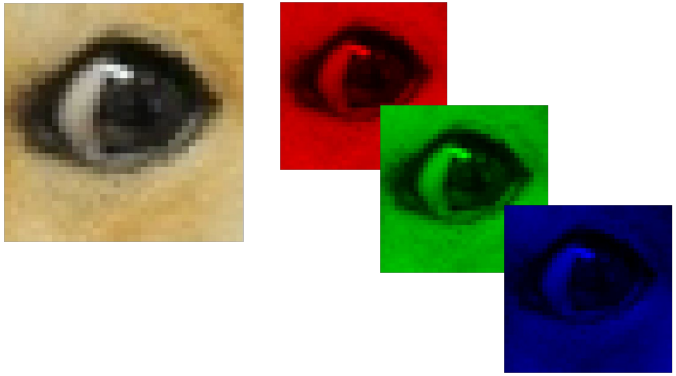




RGB Color Space

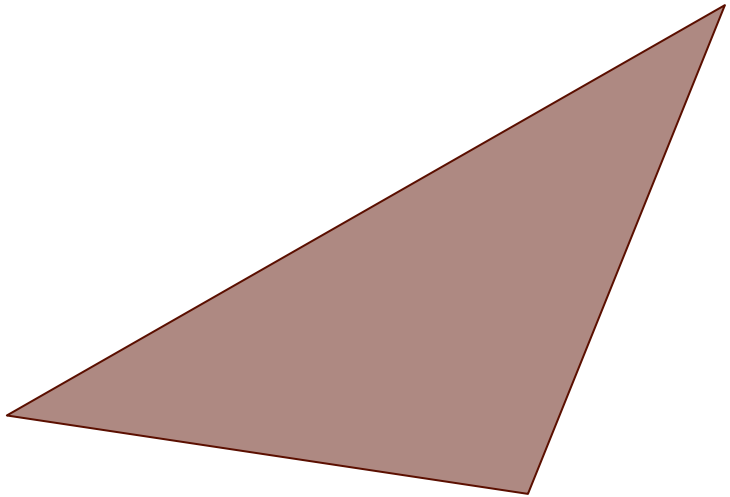
- The coordinate system used to describe colors
  - (0,0,0) = black, (1,1,1) = white
- Other color spaces exist (next lecture)





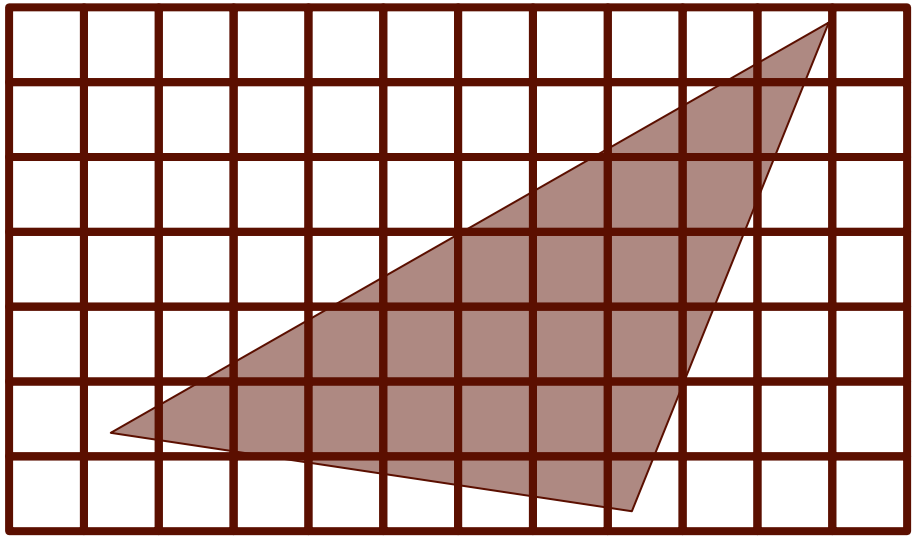
How to draw a shape on the display?

- Triangles (recall last lecture: benefits of the triangle)



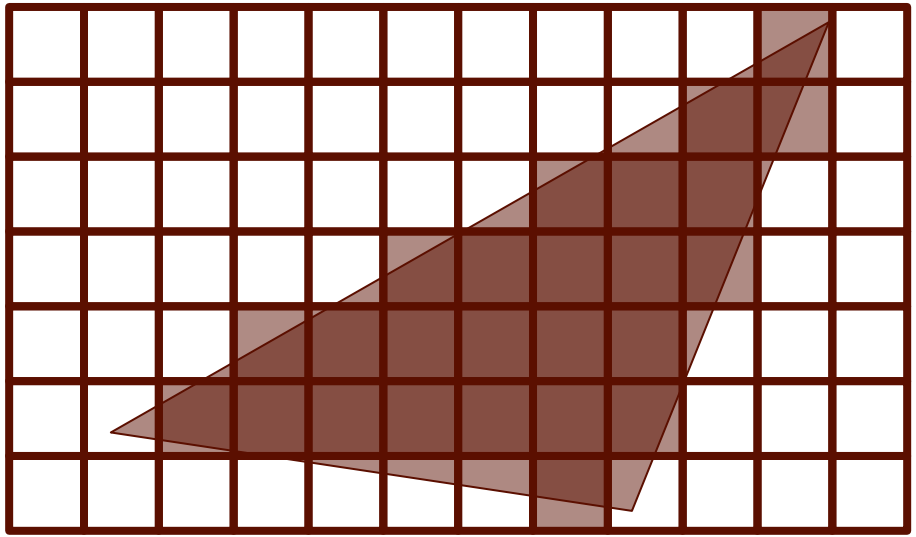
How to draw a shape on the display?

- Triangles (recall last lecture: benefits of the triangle)
- One approach: go from “3 vertices” to RGB grid via rasterization



Rasterization

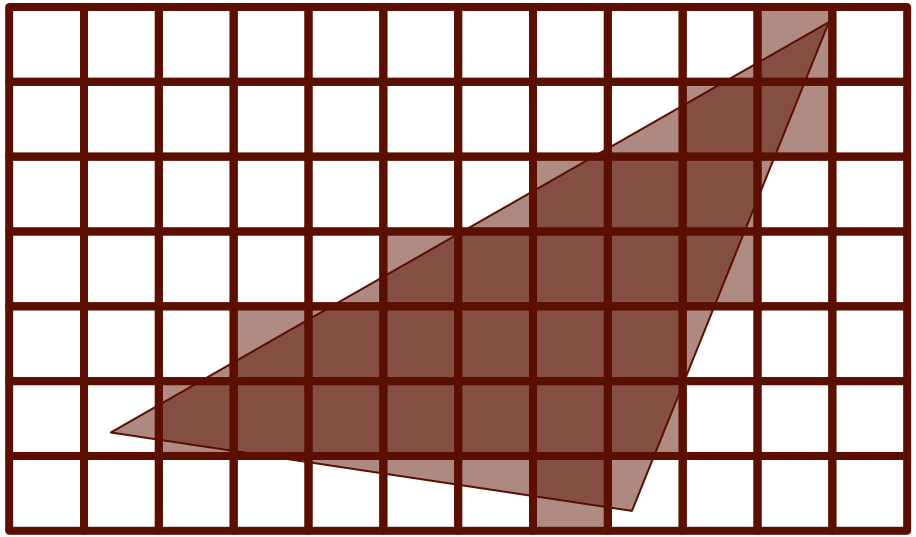
- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color



Rasterization

- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color

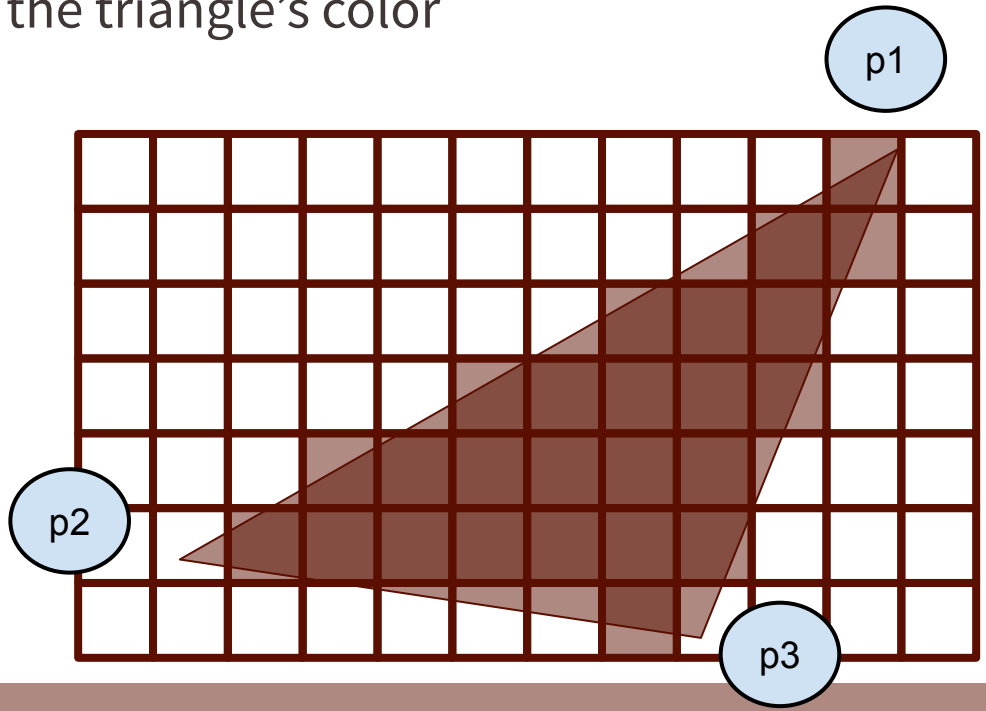
- To determine if inside the triangle:



Rasterization

- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color

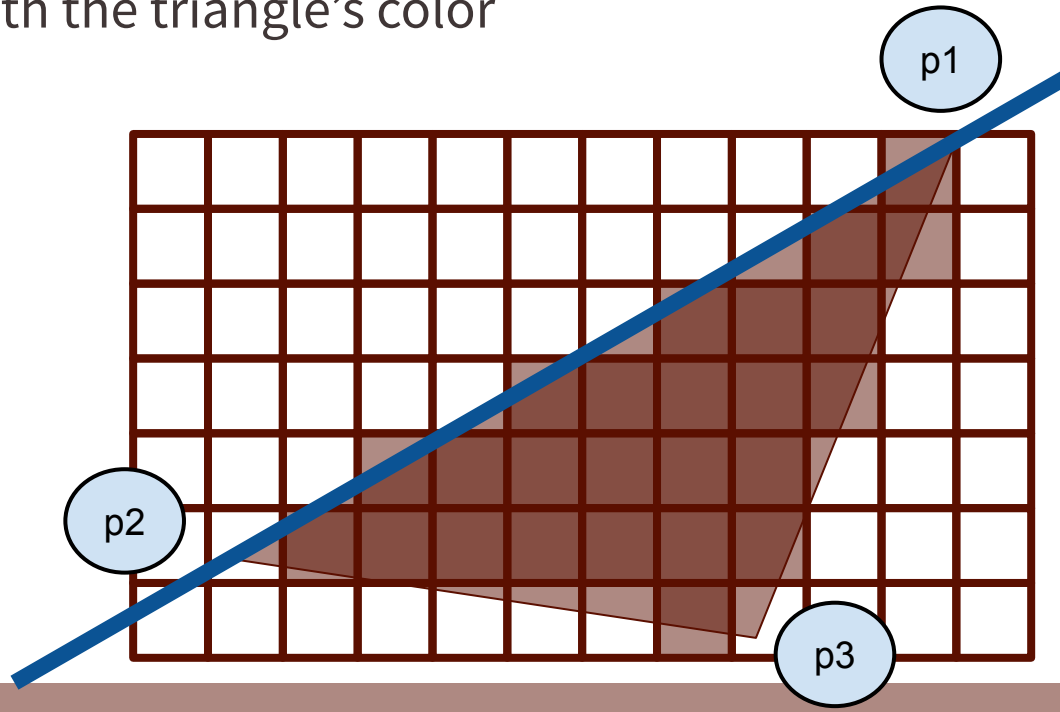
- To determine if inside the triangle:
  - A triangle with vertices p1, p2, p3 defines **3 edges**



Rasterization

- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color

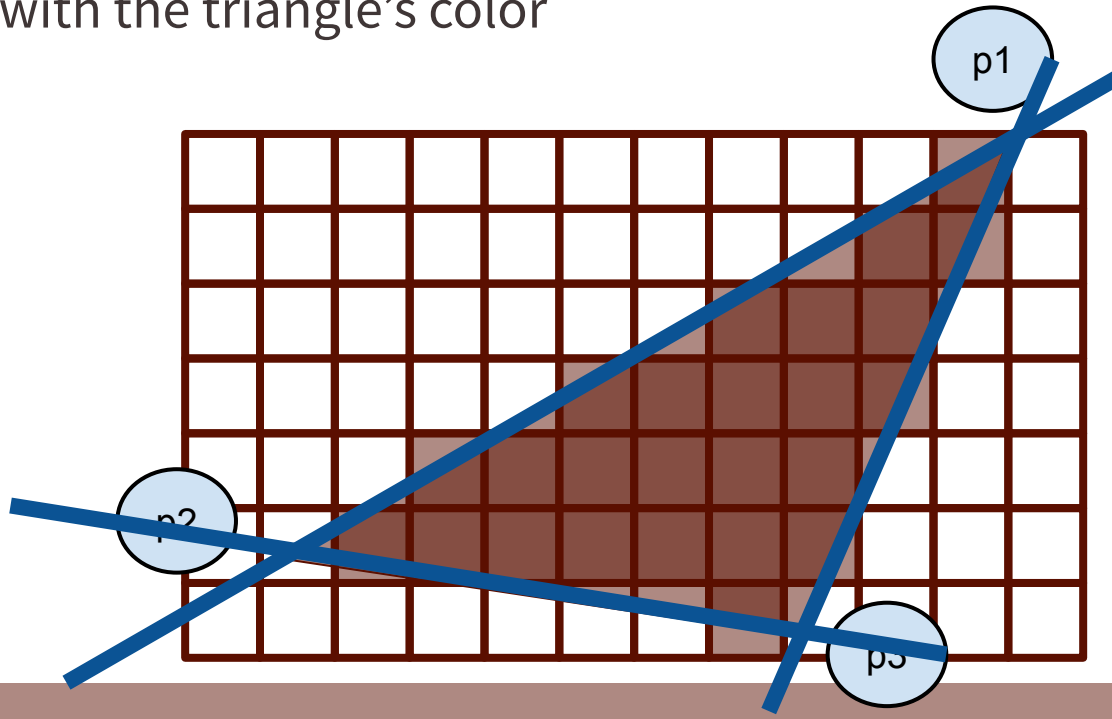
- To determine if inside the triangle:
  - A triangle with vertices p1, p2, p3 defines **3 edges**
  - Each edge splits the plane into 2 halves



Rasterization

- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color

- To determine if inside the triangle:
  - A triangle with vertices p1, p2, p3 defines **3 edges**
  - Each edge splits the plane into 2 halves

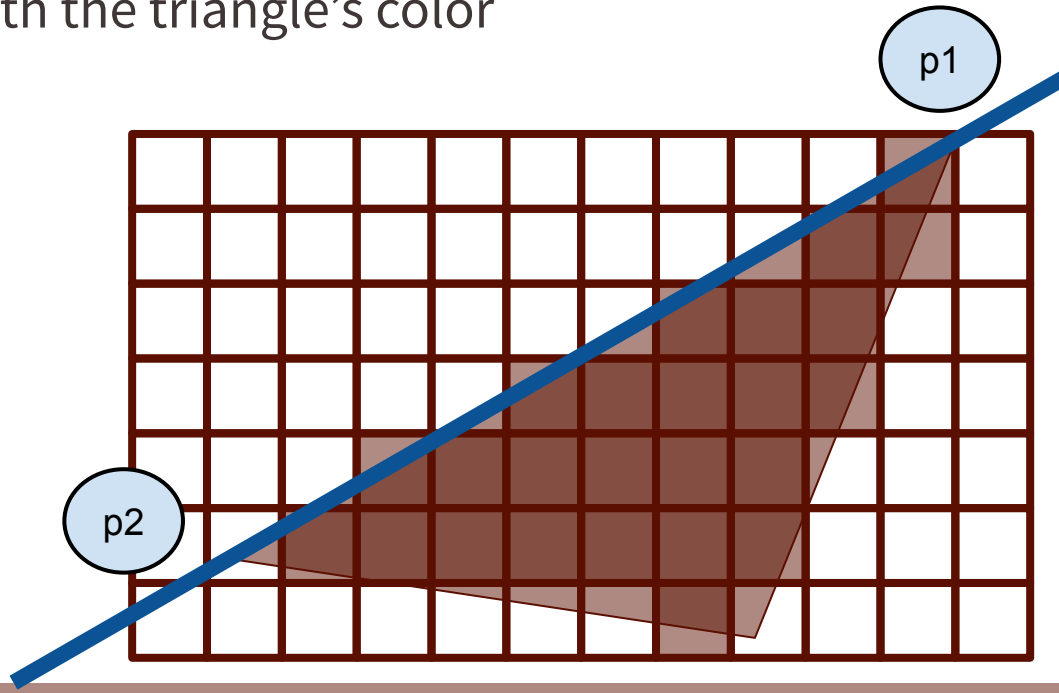


Rasterization

- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color

- To determine if inside the triangle:

y = ax + b is also **ax + b - y = 0**



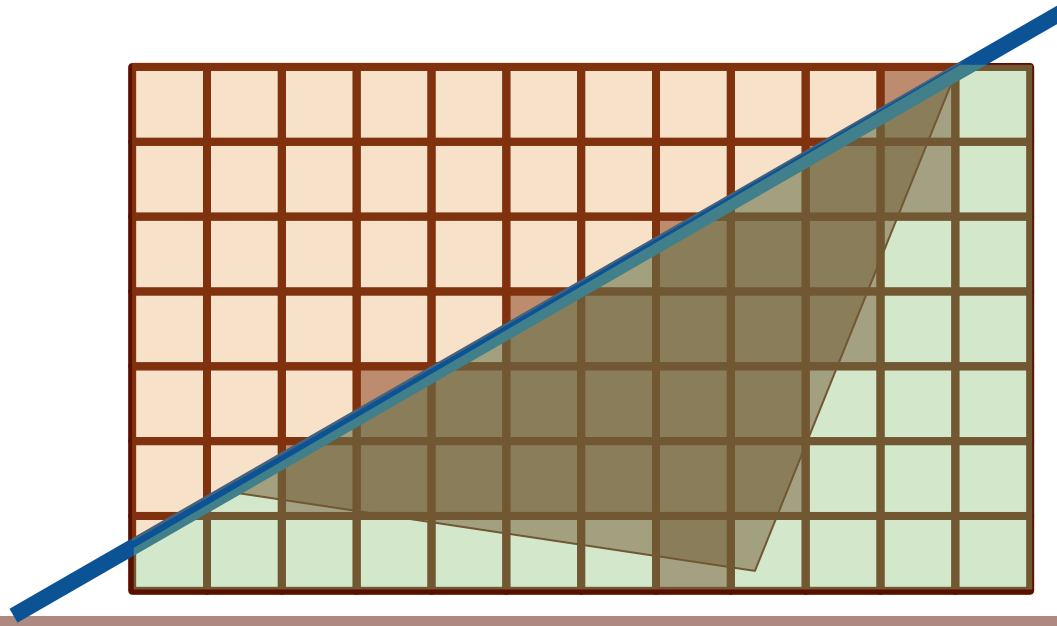
Rasterization

- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color

- To determine if inside the triangle:

y = ax + b is also **ax + b - y = 0**

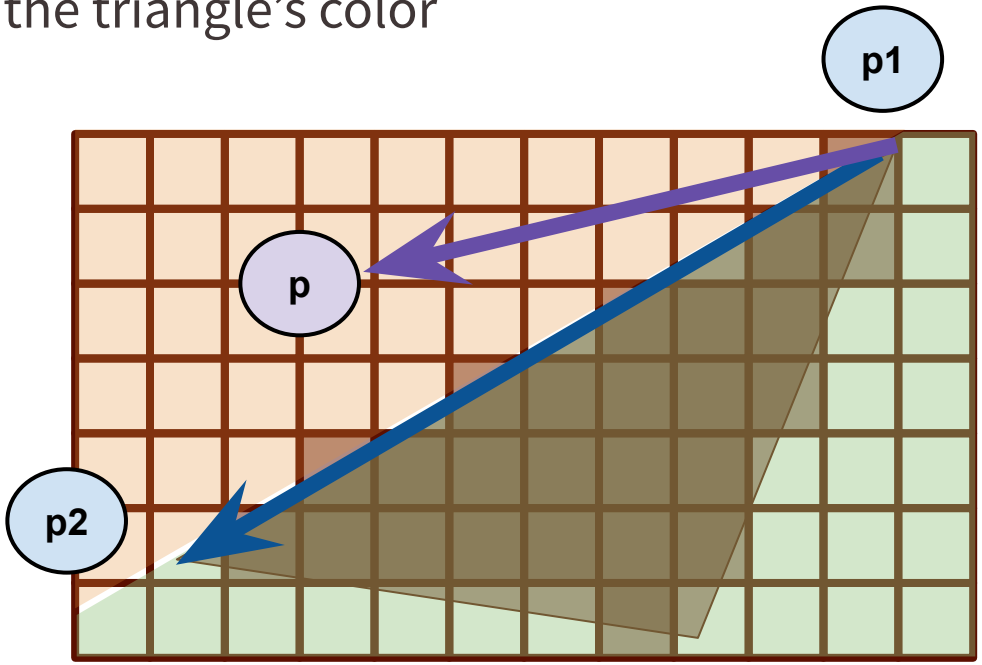
ax + b - y > 0
ax + b - y < 0



Rasterization

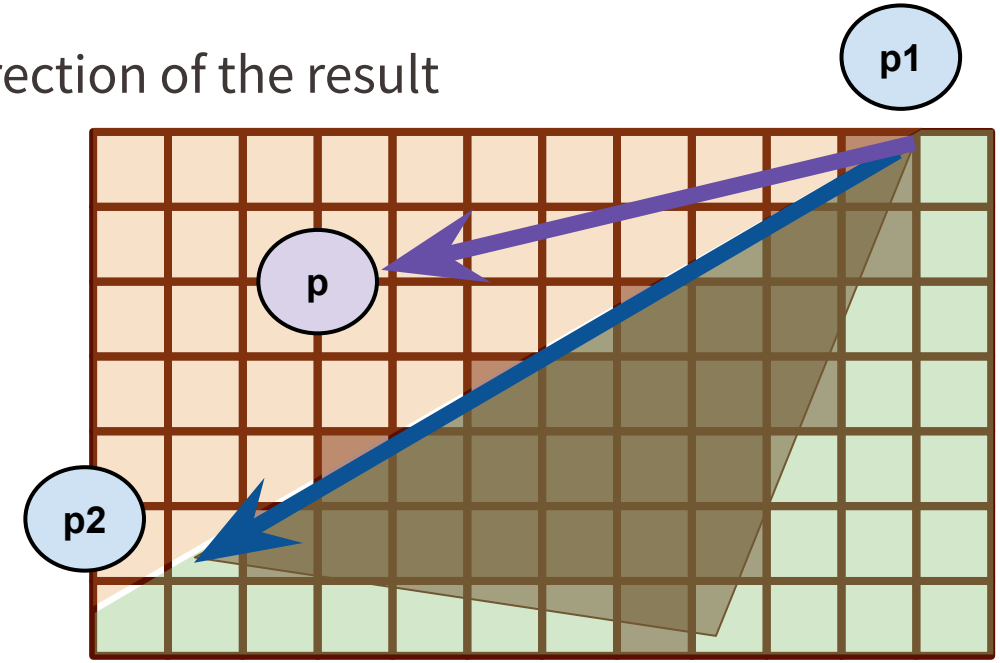
- For each pixel:
  - If the center of the pixel is inside the triangle, consider it part of the triangle and color it with the triangle's color

More direct definition: use **cross products.**



Review: the right hand rule

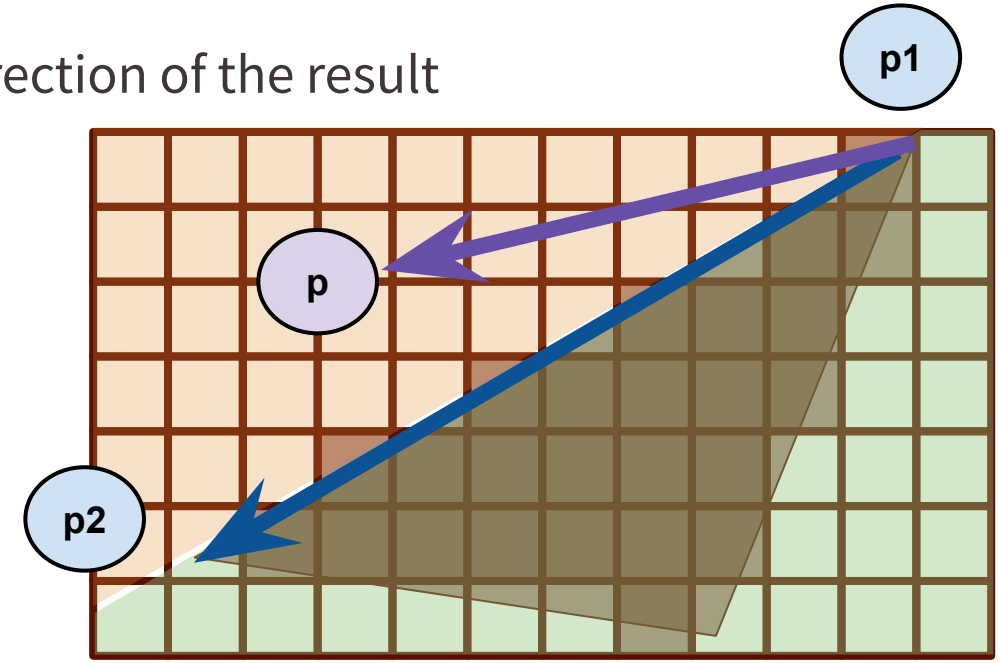
- For two vectors **A x B**
  - Point your fingers in the direction of **A**
  - Curl them toward **B**
  - Your thumb points in the direction of the result



Review: the right hand rule

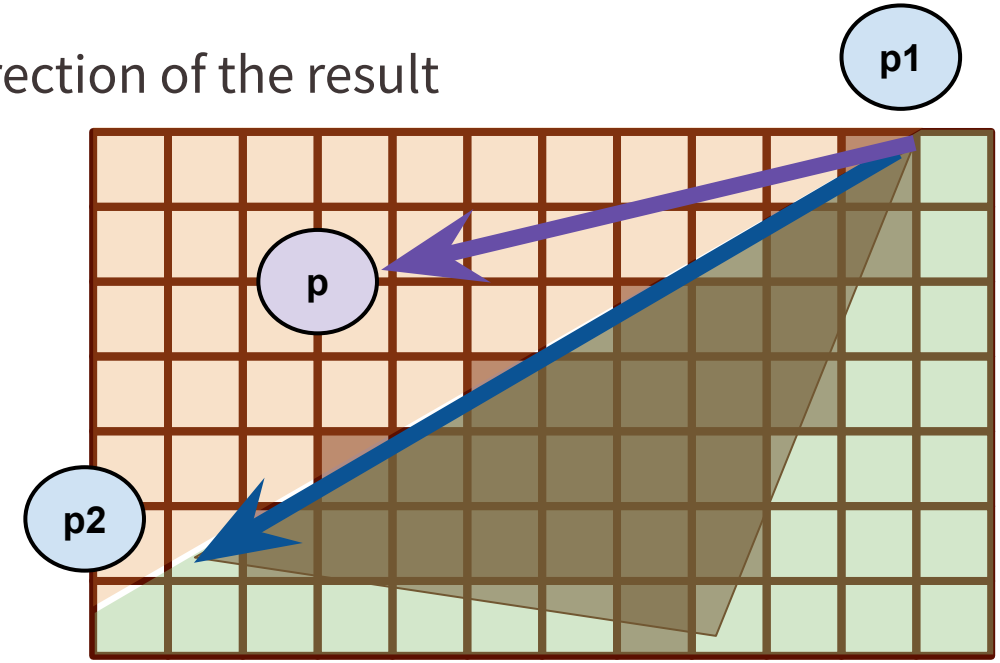
- For two vectors **A x B**
  - Point your fingers in the direction of **A**
  - Curl them toward **B**
  - Your thumb points in the direction of the result

Sign is **positive**!



Review: the right hand rule

- For two vectors **A x B**
  - Point your fingers in the direction of **A**
  - Curl them toward **B**
  - Your thumb points in the direction of the result

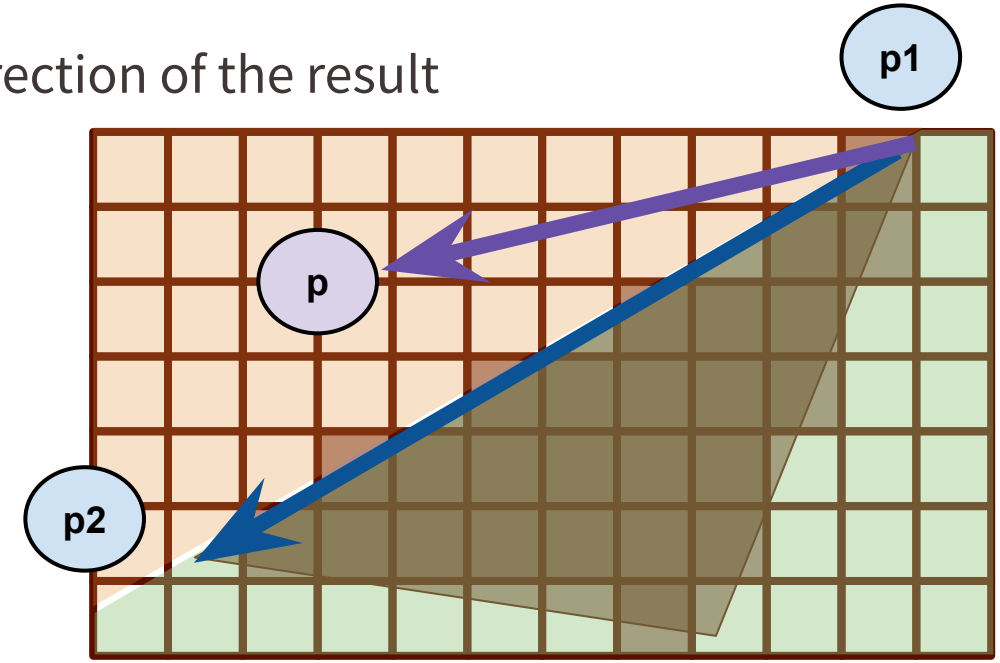


Sign is **positive**!

$(p-p_1) \times (p_2-p_1) < 0$
$(p-p_1) \times (p_2-p_1) = 0$
$(p-p_1) \times (p_2-p_1) > 0$

Review: the right hand rule

- For two vectors **A x B**
  - Point your fingers in the direction of **A**
  - Curl them toward **B**
  - Your thumb points in the direction of the result

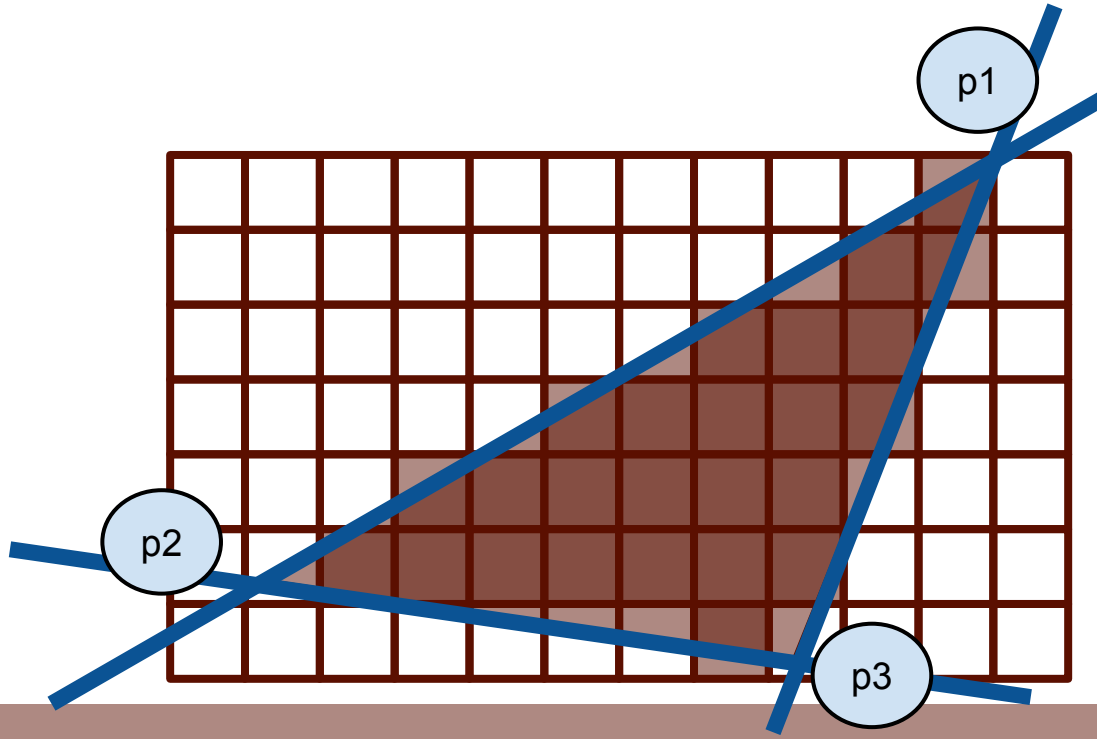


Sign is **positive**!

$(p-p_1) \times (p_2-p_1) < 0$
$(p-p_1) \times (p_2-p_1) = 0$
$(p-p_1) \times (p_2-p_1) > 0$

Rasterization

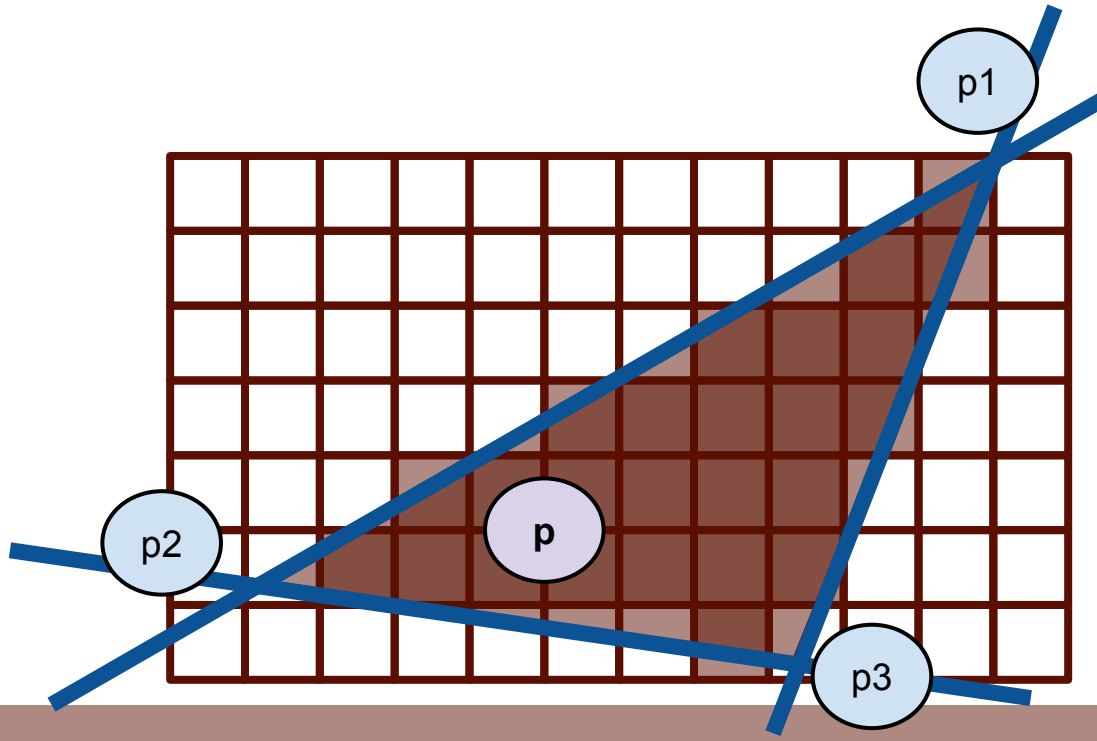
- The inside of the triangle is the **intersection of 3 half-planes**
- Given triangle defined by p1,p2,p3:



Rasterization

- The inside of the triangle is the **intersection of 3 half-planes**
- Given triangle defined by p1,p2,p3:

$$S1(p) = (p - p_1) \times (p_2 - p_1)$$
$$S2(p) = (p - p_2) \times (p_3 - p_2)$$
$$S3(p) = (p - p_3) \times (p_1 - p_3)$$

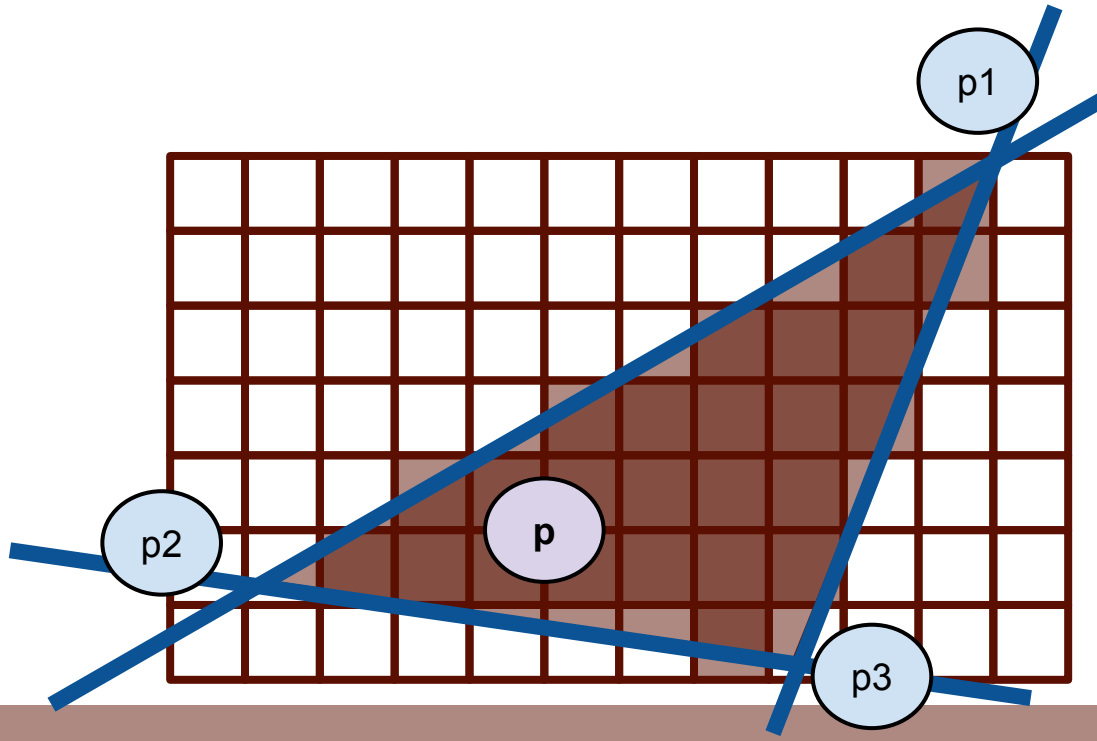


Rasterization

- The inside of the triangle is the **intersection of 3 half-planes**
- Given triangle defined by p1,p2,p3:

$$S1(p) = (p - p_1) \times (p_2 - p_1)$$
$$S2(p) = (p - p_2) \times (p_3 - p_2)$$
$$S3(p) = (p - p_3) \times (p_1 - p_3)$$

★ for all pixel centers **p**, if **max(S1(p), S2(p), S3(p))<0** then **p** is inside triangle



Rasterization

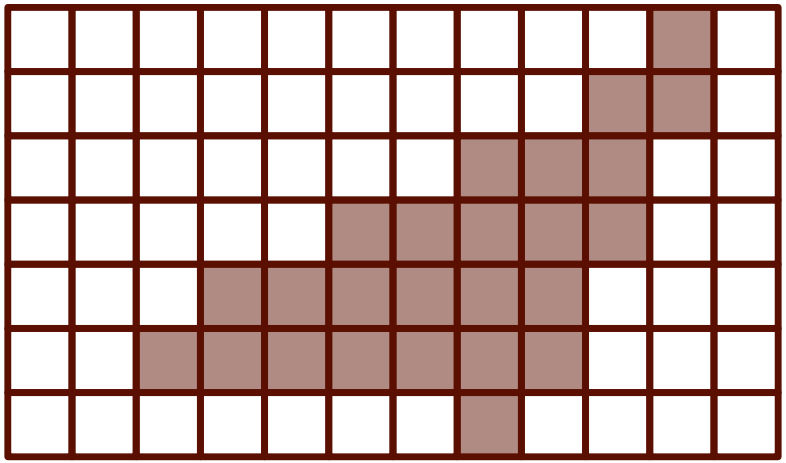
- This process if not perfect – there are many more engineering details to consider:
  - How to compute this most efficiently?

Rasterization

- This process if not perfect – there are many more engineering details to consider:
  - How to compute this most efficiently?
  - What happens when a bunch of triangles intersect at pixel center?

Rasterization

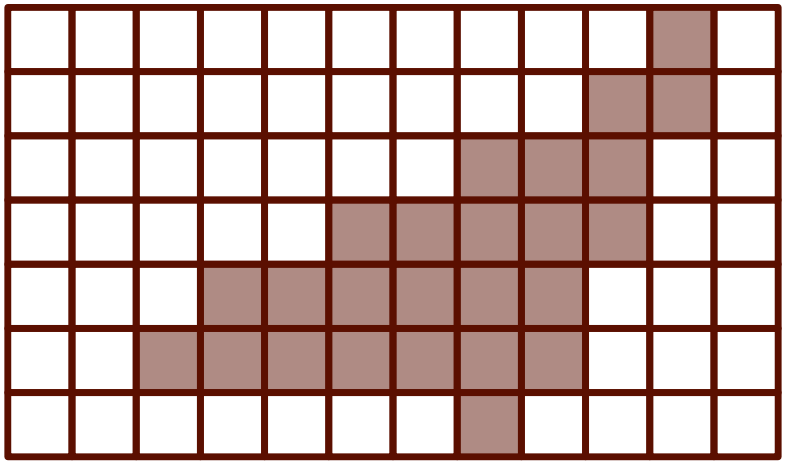
- This process if not perfect – there are many more engineering details to consider:
  - How to compute this most efficiently?
  - What happens when a bunch of triangles intersect at pixel center?
  - How to make things less jagged by computing how much of a pixel is within a triangle?



Rasterization

- This process if not perfect – there are many more engineering details to consider:
  - How to compute this most efficiently?
  - What happens when a bunch of triangles intersect at pixel center?
  - How to make things less jagged by computing how much of a pixel is within a triangle?

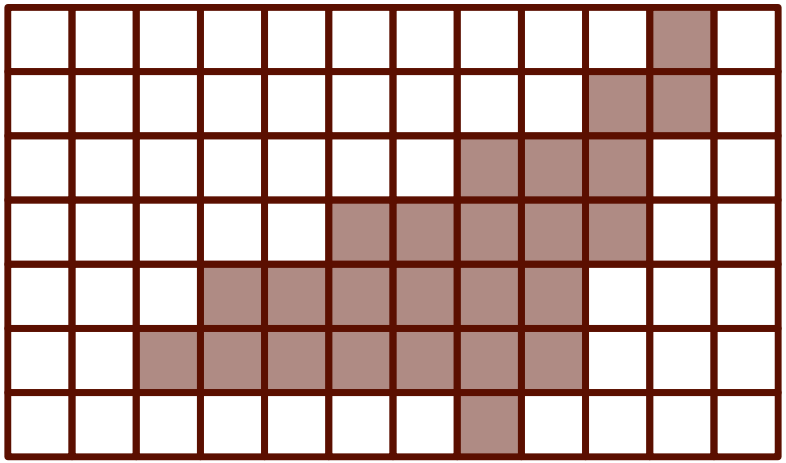
- We now know how to draw a 2D triangle!



Rasterization

- This process if not perfect – there are many more engineering details to consider:
  - How to compute this most efficiently?
  - What happens when a bunch of triangles intersect at pixel center?
  - How to make things less jagged by computing how much of a pixel is within a triangle?

- We now know how to draw a 2D triangle!
  - Next lecture: how we got the original 3D triangle to be on a 2D screen in the first place



Lecture Outline

- How to draw a triangle
  - Image representation
  - Rasterization
- How to color a triangle
  - Surface normals, shading, barycentric interpolation
- The OpenGL pipeline
  - Vertex vs. fragment shaders

How do we shade a triangulated object?

How do we shade a triangulated object?

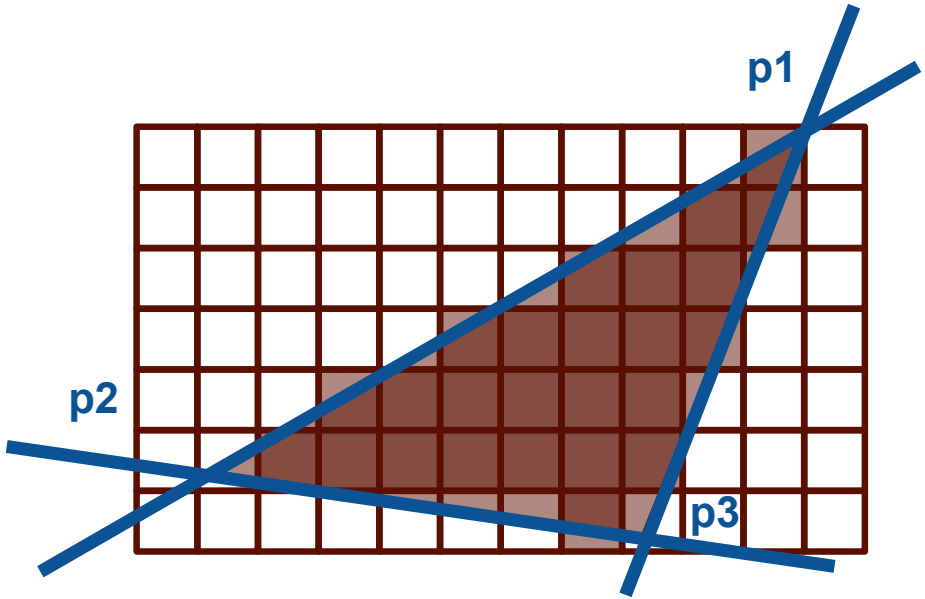
- Fun fact: the Utah teapot (1975) is the **first implicit surface model**
  - University of Utah: where a lot of graphics research originated

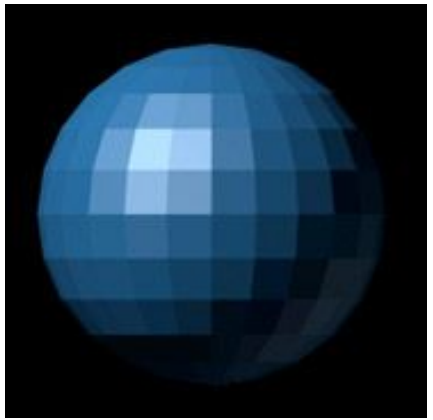
Flat Shading

- Color every pixel inside a triangle with an average of colors at vertices

Flat Shading

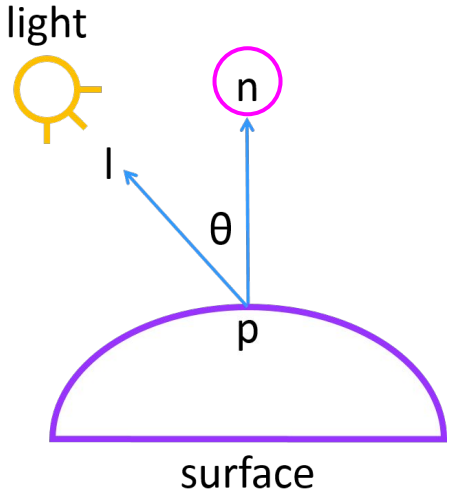
- Color every pixel inside a triangle with an average of colors at vertices
- Not great…
- Also storing colors at vertices is not robust (e.g. change of light)





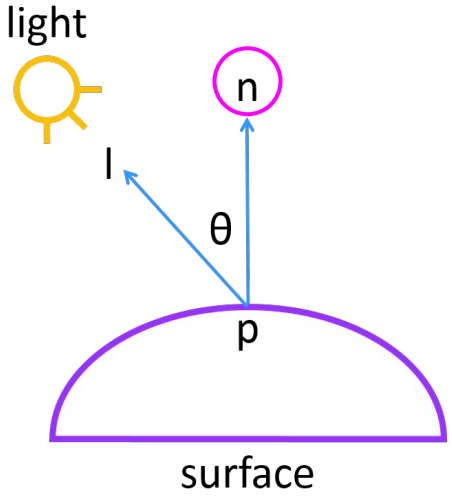
Surface Normals

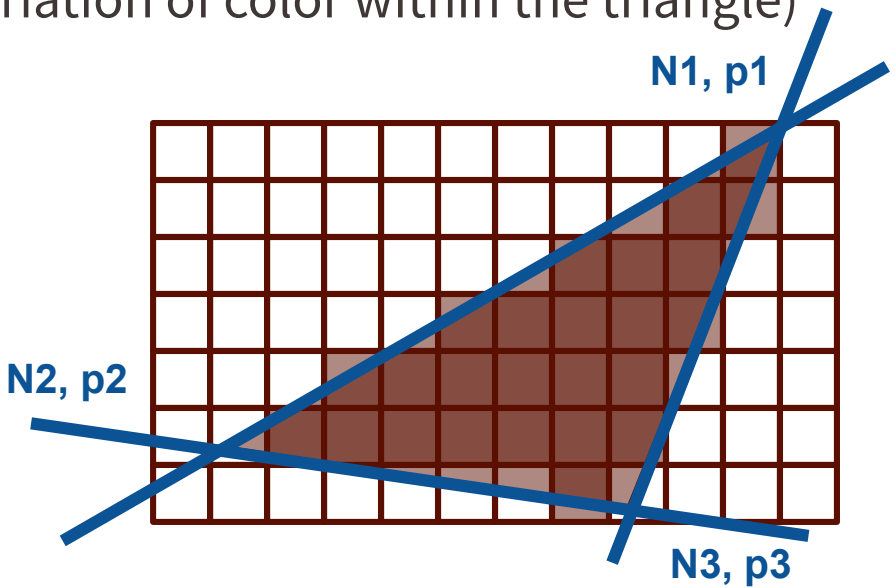
- At each vertex, store the normal vector that points outwards from the object surface in 3D space at that vertex



Surface Normals

- At each vertex, store the normal vector that points outwards from the object surface in 3D space at that vertex
- Have N1, N2, N3 in addition to p1, p2, p3
- Normals allow for interpolation (variation of color within the triangle)





Phong Reflection Model

- Developed in 1973 (University of Utah again!)
- A way to break the problem up into ambient, diffuse, and specular

Phong Reflection Model

- **Ambient:** “Base color”. Light bounces around the environment. Even if you have little light, objects in shadow aren’t usually pure black.

- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

C_i = ambient + diffuse + specular

Phong Reflection Model

- **Ambient:** “Base color”. Light bounces around the environment. Even if you have little light, objects in shadow aren’t usually pure black.

- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

C_i = ambient + diffuse + specular

Phong Reflection Model: Ambient

- **Ambient:** “Base color”. Light bounces around the environment. Even if you have little light, objects in shadow aren’t usually pure black.

- More simple!
- **Ca** ⇒ represents the color to give the object if there is no light

Phong Reflection Model: Ambient

- **Ambient:** “Base color”. Light bounces around the environment. Even if you have little light, objects in shadow aren’t usually pure black.

- More simple!
- **Ca** ⇒ represents the color to give the object if there is no light
- So… $c_{ambient} = c_a$

Phong Reflection Model: Ambient

- **Ambient:** “Base color”. Light bounces around the environment. Even if you have little light, objects in shadow aren’t usually pure black.

- More simple!
- **Ca** ⇒ represents the color to give the object if there is no light
- So… $c_{ambient} = c_a$

- Putting ambient in the equation for color:

  $c = c_a$

  Where c represents our final color

Phong Reflection Model

- ~~**Ambient:** "Base color". Light bounces around the environment. Even if you have little light, objects in shadow aren't usually pure black.~~

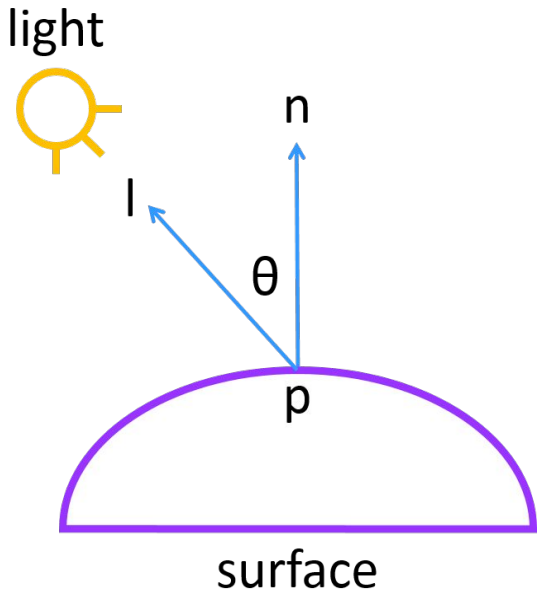
- **Diffuse:** "Rough material". Given the same light source, objects look brighter when hit "perpendicularly". (Will talk about why later in Lights and Optics!)

- **Specular:** "Shiny material". Bright highlights when light reflects into our eyes.

C_i = ambient + diffuse + specular

Phong Reflection Model: Diffuse

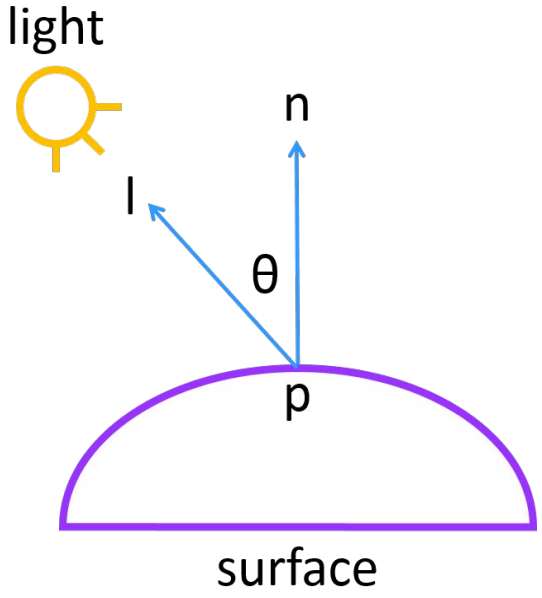
- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)



Phong Reflection Model: Diffuse

- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)

- **c** ⇒ the color we’re computing at p
- **l** ⇒ vector from p to the light
- **n** ⇒ normal vector at p
- **Lambert’s cosine law** ⇒ $c \propto \cos\theta$

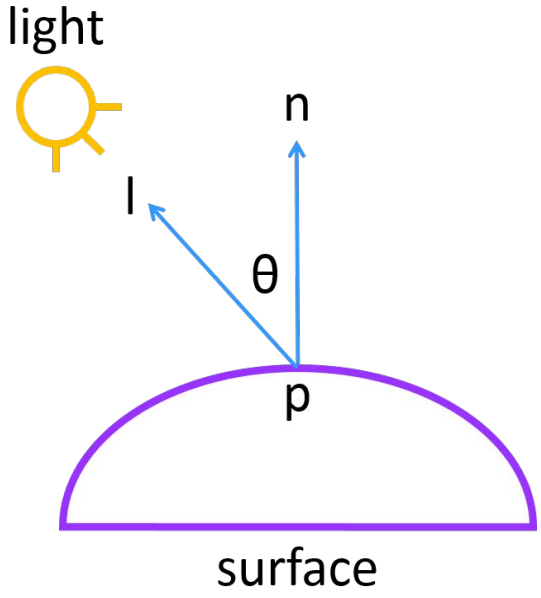


Phong Reflection Model: Diffuse

- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)

- Remember dot products!
- The cosine of the angle between 2 **outward** pointing **unit** vectors at a point ⇒ the dot product of the 2 vectors

$$c \propto \boxed{\cos\theta} = c \propto \boxed{n \cdot l}$$



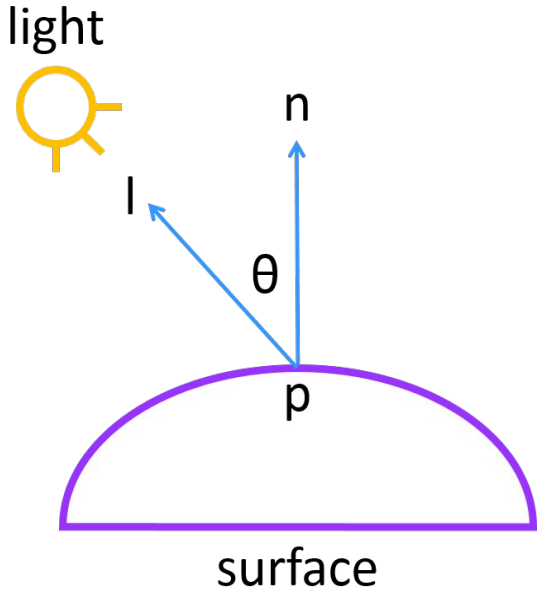
Phong Reflection Model: Diffuse

- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)

$$c \propto \cos\theta \ = c \propto n \cdot l$$

- **Cd** ⇒ the diffuse material of the surface (“roughness” of the surface)
- **Cl** ⇒ the color of the light

$$c_{diffuse} = c_d c_l n \cdot l$$



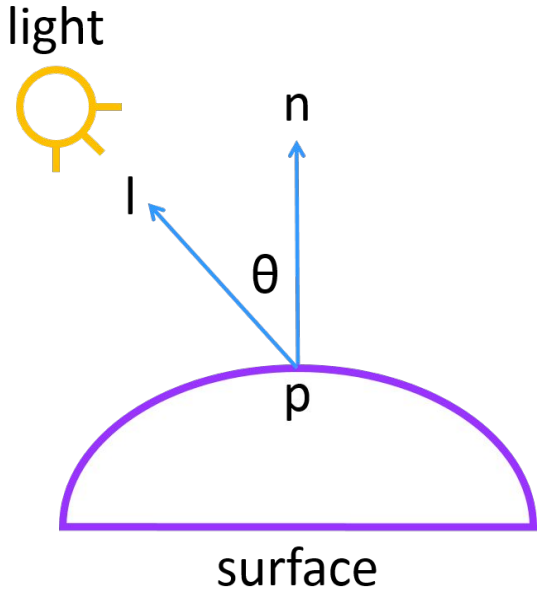
Phong Reflection Model: Diffuse

- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)

$$c \propto \cos\theta \; = c \propto n \cdot l$$

- **Cd** ⇒ the diffuse material of the surface (“roughness” of the surface)
- **Cl** ⇒ the color of the light
- Need a **max** for objects facing **away** from the light:

$$c_{diffuse} = c_d c_l max(0, n \cdot l)$$



Phong Reflection Model: Diffuse

- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)

$$c_{diffuse} = c_d c_l max(0, n \cdot l)$$

- Putting diffuse in the equation for color (with ambient):

$$c = c_a + c_d c_l max(0, n \cdot l) + ...$$

Phong Reflection Model

- ~~**Ambient:** "Base color". Light bounces around the environment. Even if you have little light, objects in shadow aren't usually pure black.~~

- ~~**Diffuse:** "Rough material". Given the same light source, objects look brighter when hit "perpendicularly". (Will talk about why later in Lights and Optics!)~~

- **Specular:** "Shiny material". Bright highlights when light reflects into our eyes.

C_i = ambient + diffuse + specular

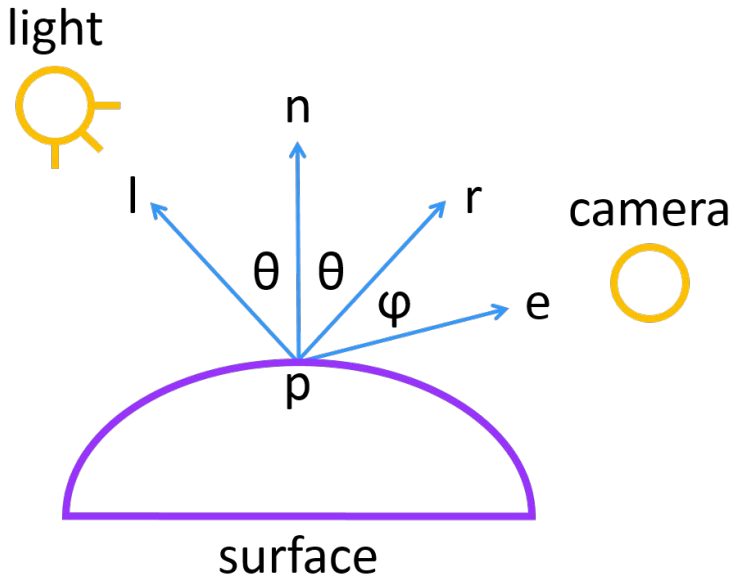
Phong Reflection Model: Specular

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

Phong Reflection Model: Specular

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

- Similar to diffuse:
- **Cs** ⇒ the specular material of the surface
- **Cl** ⇒ the color of the light
- **r** ⇒ the reflection of the l across n
- **e** ⇒ a vector from p to the camera/eye

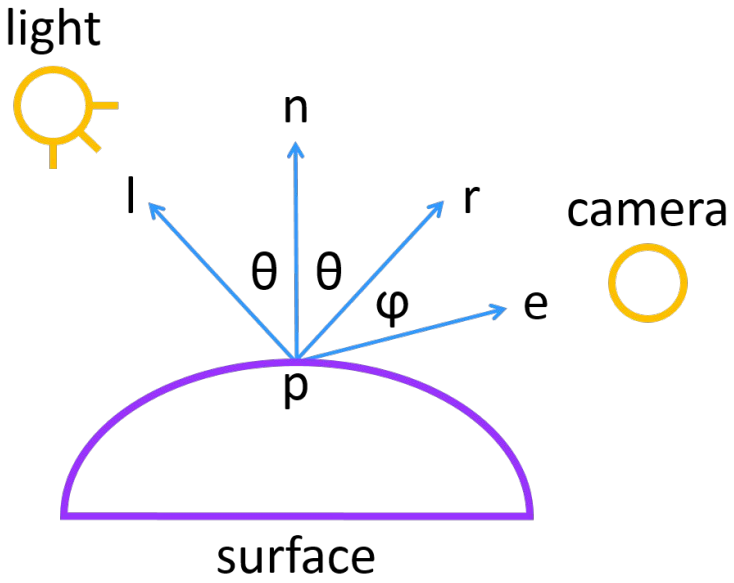


Phong Reflection Model: Specular

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

- Similar to diffuse:
- **Cs** ⇒ the specular material of the surface
- **Cl** ⇒ the color of the light
- **r** ⇒ the reflection of the l across n
- **e** ⇒ a vector from p to the camera/eye

$$c_{specular} = c_s c_l e \cdot r$$
$$c_{specular} = c_s c_l max(0, e \cdot r)$$



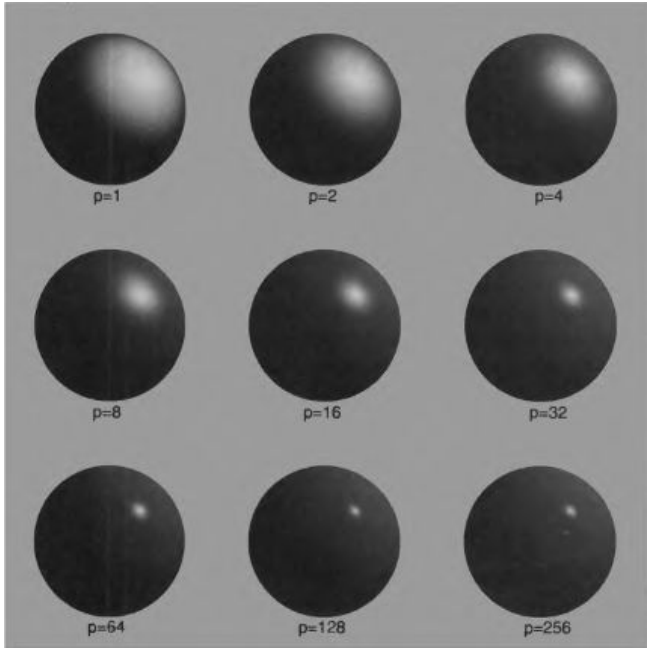
Phong Reflection Model: Specular

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

$$c_{specular} = c_s c_l max(0, e \cdot r)$$

- **alpha** or **p** ⇒ a “shininess value” (the **Phong exponent**) that specifies how shiny we want to tune the material

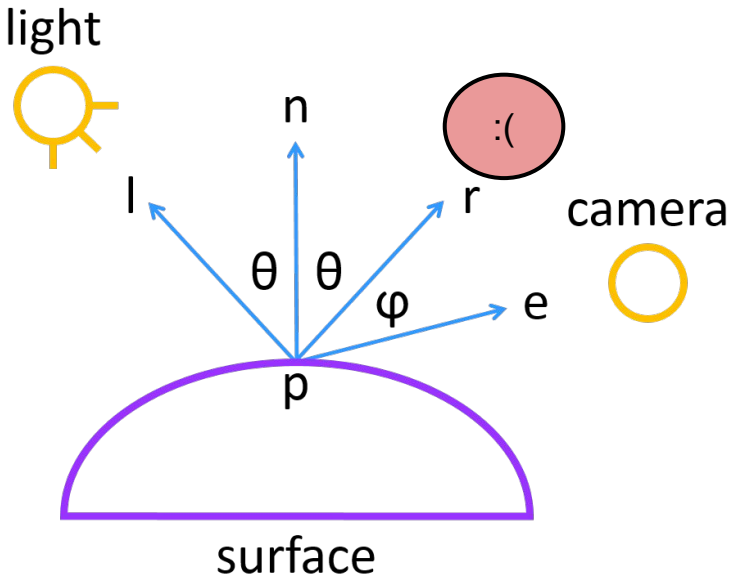
$$c_{specular} = c_s c_l max(0, e \cdot r)^{\alpha}$$



Phong Reflection Model: Specular

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

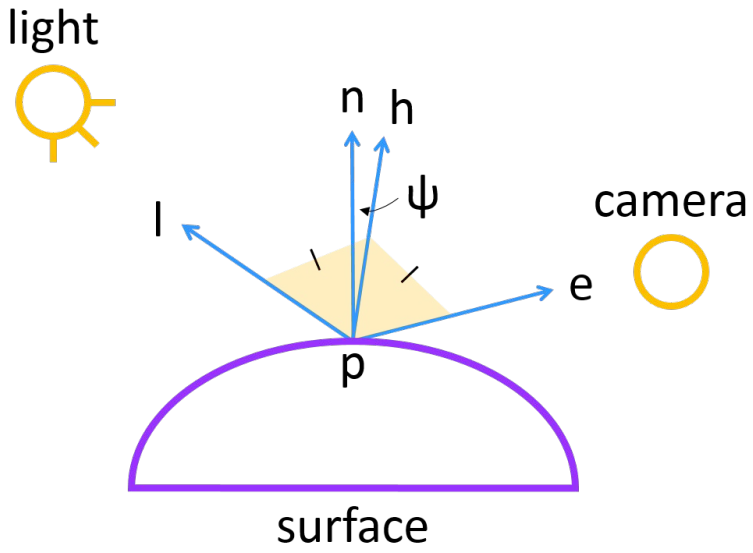
- Problem: r is not trivial to compute



Phong Reflection Model: Specular

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

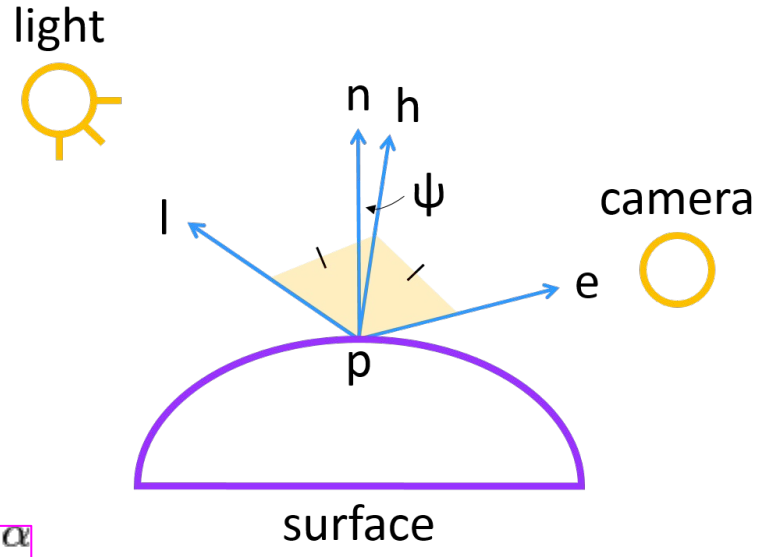
- Problem: r is not trivial to compute
- Easier approach:
  - Compute the halfway vector between l and e as new vector **h**
  - When h = n, we are looking directly at the mirror reflection
  - Can approximate using the dot product of n and h, instead of e and r



Phong Reflection Model: Specular

- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

- Problem: r is not trivial to compute
- Easier approach:
  - Compute the halfway vector between l and e as new vector **h**
  - When h = n, we are looking directly at the mirror reflection
  - Can approximate using the dot product of n and h, instead of e and r



$$h = \frac{e + l}{|e + l|}$$

$$c_{specular} = c_s c_l max(0, n \cdot h)^{\alpha}$$

Phong Reflection Model: Lighting Model

- Now we can put it all together:

$$c = c_{ambient} + c_{diffuse} + c_{specular}$$

$$c = c_a + c_d c_l max(0, n \cdot l) + c_s c_l max(0, n \cdot h)^{\alpha}$$

C_i = ambient + diffuse + specular

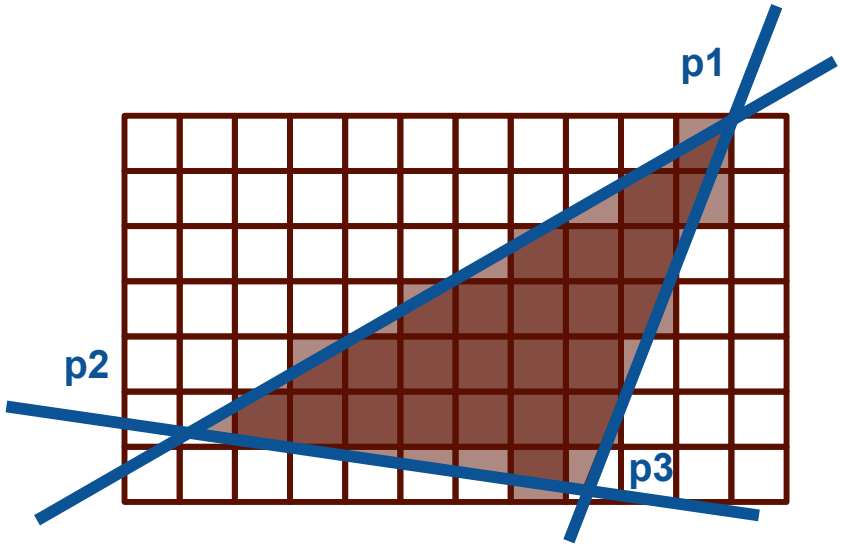
Phong Reflection Model: Lighting Model

- **One last step…**
- Our model computes color based on light well at each **vertex**
  - But what about the part of the triangle **in between** the vertices?

C_i = **ambient** + **diffuse** + **specular**

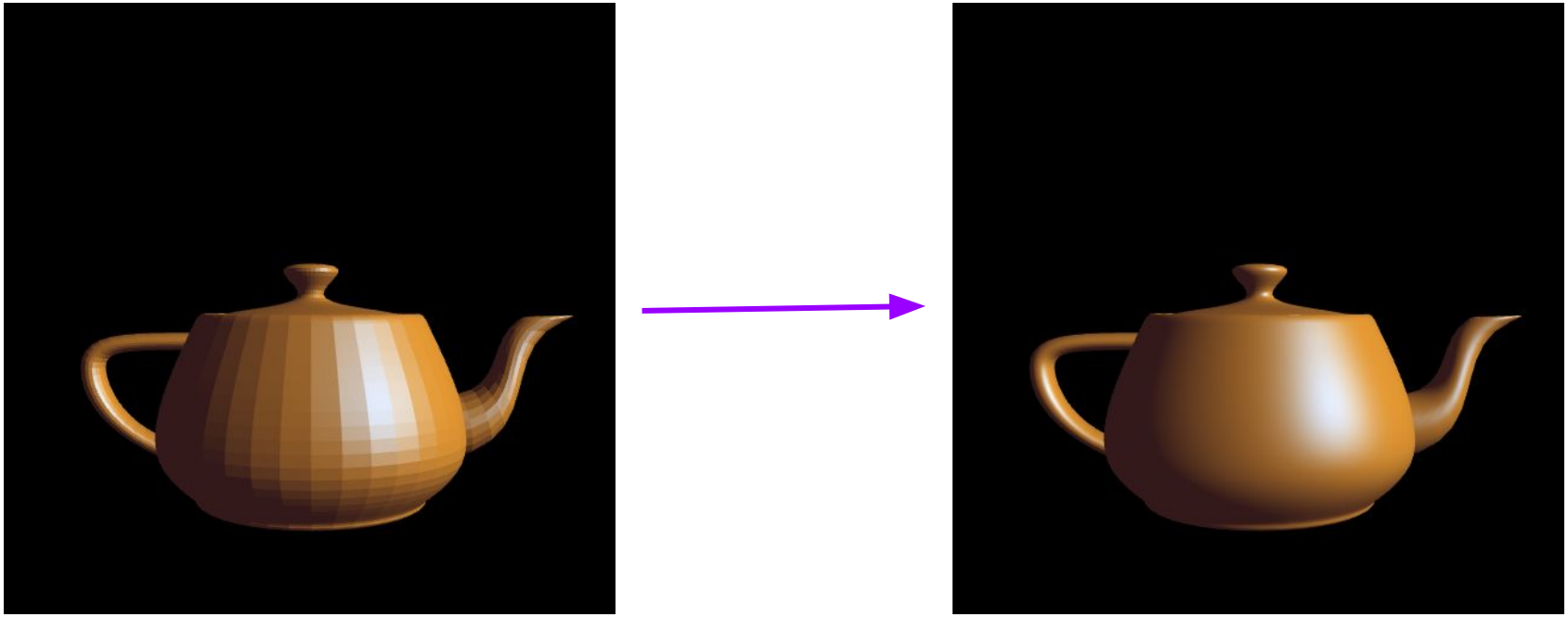
Reminder: Flat Shading

- [Ignore storing colors at vertices for now]



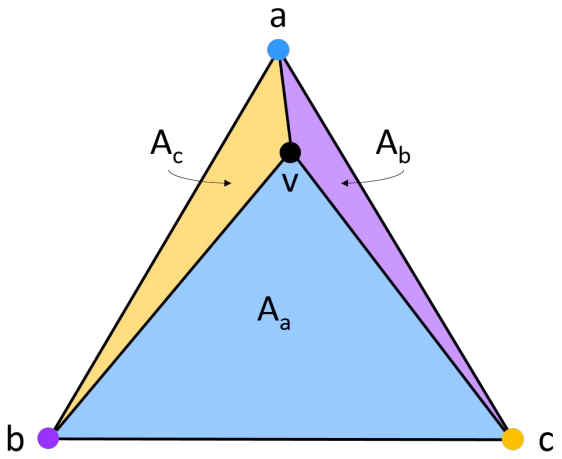
Smooth vs. Flat Shading

- Approximating smooth geometry with triangles



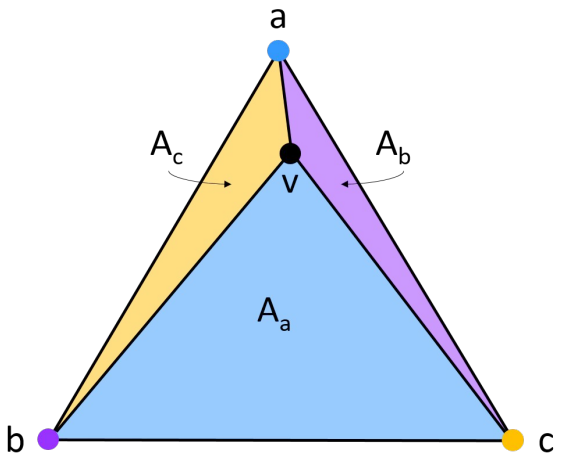
Barycentric Interpolation

- A way of determining the best color for a pixel



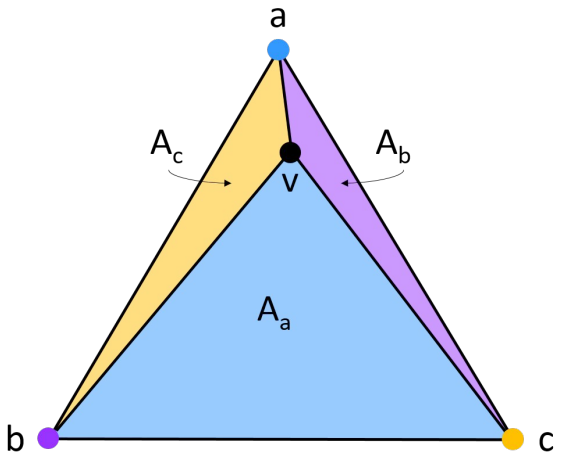
Barycentric Interpolation

- A way of determining the best color for a pixel
- Suppose you have **color values** computed at each vertex a,b,c of your triangle (ca,cb,cc)



Barycentric Interpolation

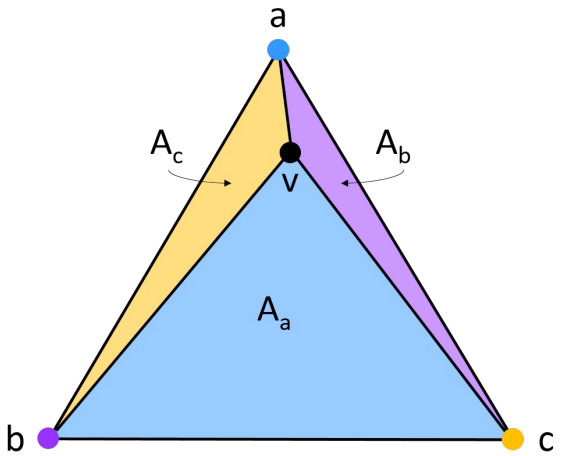
- A way of determining the best color for a pixel
- Suppose you have **color values** computed at each vertex a,b,c of your triangle (ca,cb,cc)
- For any point v inside your triangle, divide the triangle into 3 triangles with separate areas



Barycentric Interpolation

- A way of determining the best color for a pixel
- Suppose you have **color values** computed at each vertex a,b,c of your triangle (ca,cb,cc)
- For any point v inside your triangle, divide the triangle into 3 triangles with separate areas
- The interpolation for the **color** at v is:

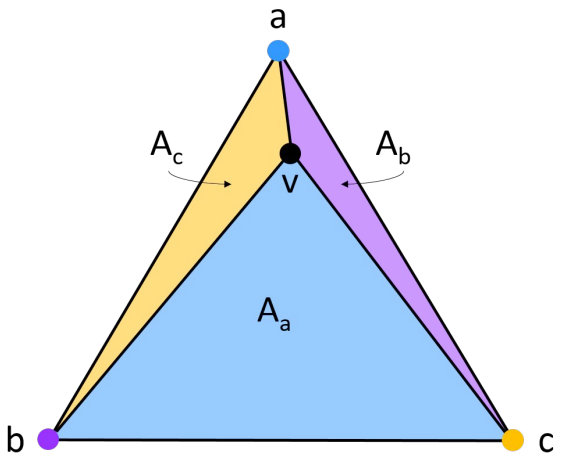
$$c_v = \frac{A_a}{A_{total}} c_a + \frac{A_b}{A_{total}} c_b + \frac{A_c}{A_{total}} c_c$$



- This is also called **Gourad Shading**

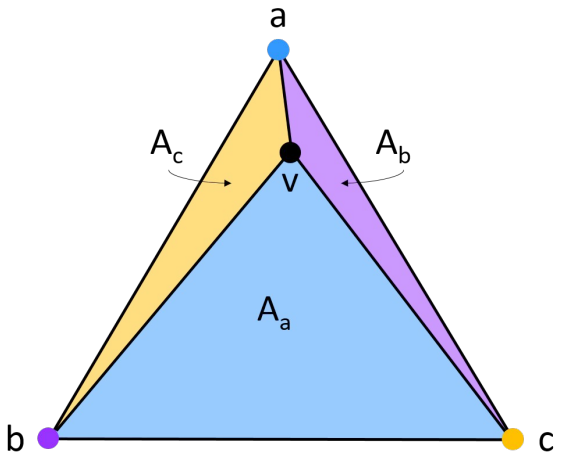
Barycentric Interpolation (alt.)

- Alternatively, suppose you have **normals** computed at each vertex a,b,c of your triangle (na,nb,nc).



Barycentric Interpolation (alt.)

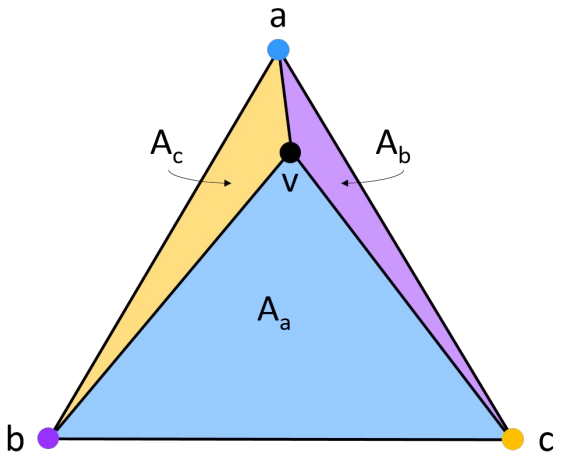
- Alternatively, suppose you have **normals** computed at each vertex a,b,c of your triangle (na,nb,nc).
- For any point v inside your triangle, divide the triangle into 3 triangles with separate areas



Barycentric Interpolation (alt.)

- Alternatively, suppose you have **normals** computed at each vertex a,b,c of your triangle (na,nb,nc).
- For any point v inside your triangle, divide the triangle into 3 triangles with separate areas
- The interpolation for the **normal** at v is:

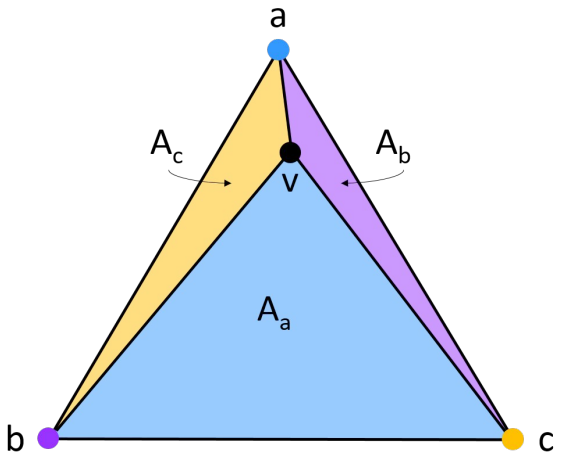
$$n_v = \frac{A_a}{A_{total}} n_a + \frac{A_b}{A_{total}} n_b + \frac{A_c}{A_{total}} n_c$$



Barycentric Interpolation (alt.)

- Alternatively, suppose you have **normals** computed at each vertex a,b,c of your triangle (na,nb,nc).
- For any point v inside your triangle, divide the triangle into 3 triangles with separate areas
- The interpolation for the **normal** at v is:

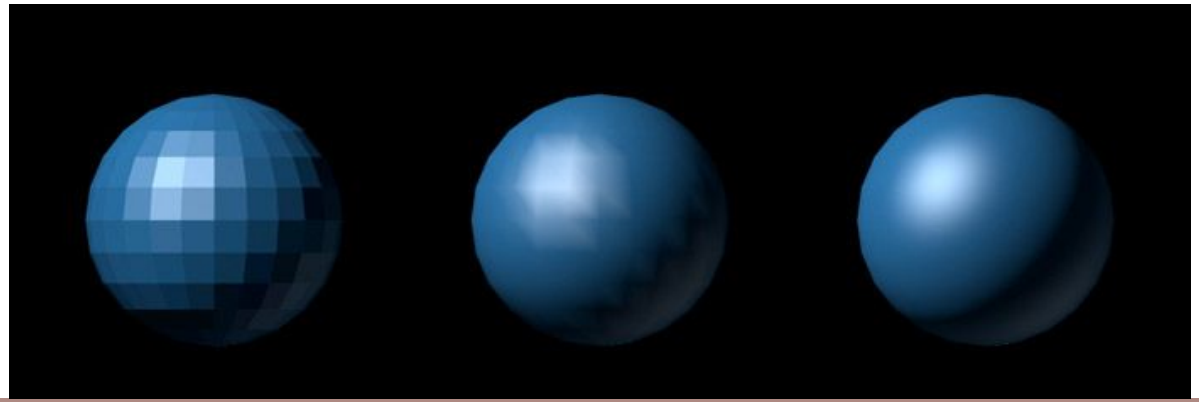
$$n_v = \frac{A_a}{A_{total}} n_a + \frac{A_b}{A_{total}} n_b + \frac{A_c}{A_{total}} n_c$$



- Then, use the Phong Reflection Model to compute the color at the interpolated normal!
- This is called **Phong Shading!**

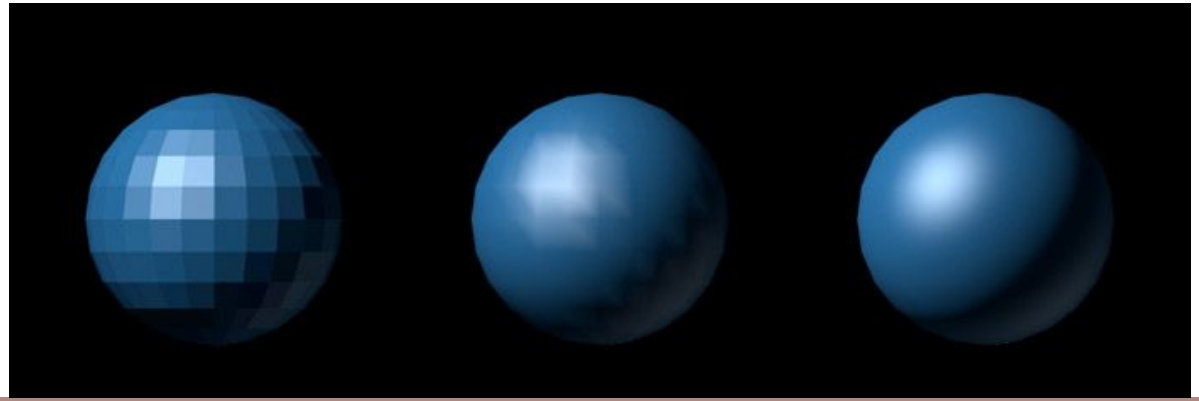
Flat vs. Gourad vs. Phong Shading

- **Flat**: shade the triangle using the average of the colors computed at each vertex – simple, fast, but looks bad.



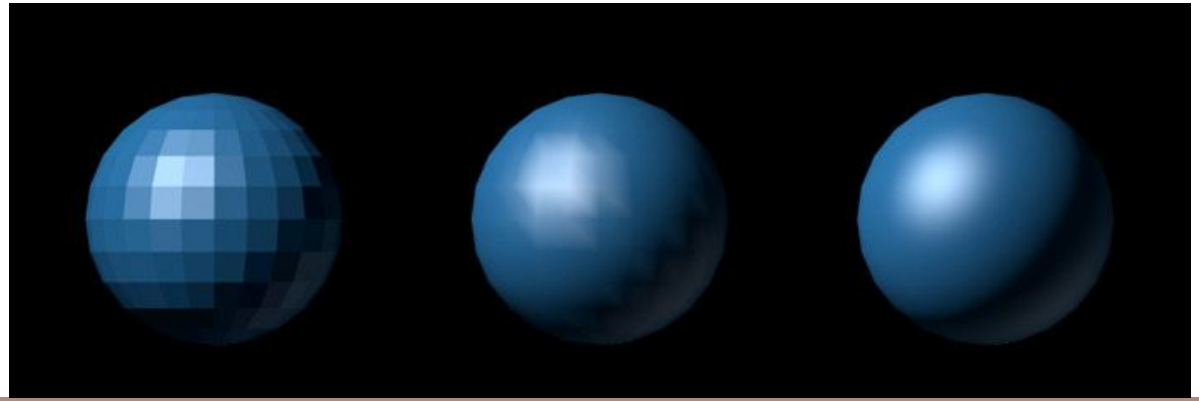
Flat vs. Gourad vs. Phong Shading

- **Flat**: shade the triangle using the average of the colors computed at each vertex – simple, fast, but looks bad.
- **Gourad**: shade the triangle by interpolating the colors across each vertex – good balance between speed and visual result.



Flat vs. Gourad vs. Phong Shading

- **Flat**: shade the triangle using the average of the colors computed at each vertex – simple, fast, but looks bad.
- **Gourad**: shade the triangle by interpolating the colors across each vertex – good balance between speed and visual result.
- **Phong**: interpolate the normal across each vertex, then compute the color for each point in the triangle – expensive but best look!



Limitations

- Rasterization = taping triangle shaped stickers onto a screen
  - How do you do transparency?
  - How do you do shadows/self-occlusions?

Limitations

- Rasterization = taping triangle shaped stickers onto a screen
  - How do you do transparency?
  - How do you do shadows/self-occlusions?

- Some of these we can “hack” by doing multiple-passes of rasterization
  - Shadow mapping

Limitations

- Rasterization = taping triangle shaped stickers onto a screen
  - How do you do transparency?
  - How do you do shadows/self-occlusions?

- Some of these we can “hack” by doing multiple-passes of rasterization
  - Shadow mapping

- More physically-based way: **raytracing**

Limitations

- Rasterization = taping triangle shaped stickers onto a screen
  - How do you do transparency?
  - How do you do shadows/self-occlusions?

- Some of these we can “hack” by doing multiple-passes of rasterization
  - Shadow mapping

- More physically-based way: **raytracing**

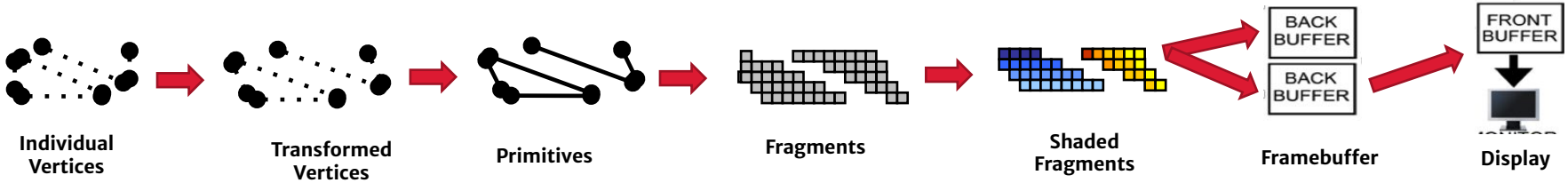
- Speed is still a big advantage
  - Real-time graphics, pre-viz for more expensive rendering methods

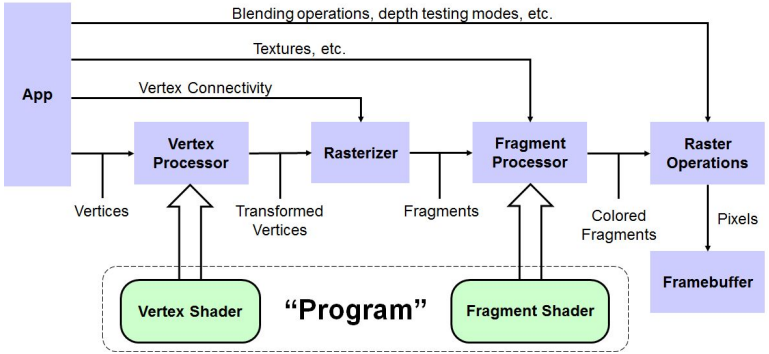
Lecture Outline

- How to draw a triangle
  - Image representation
  - Rasterization
- How to color a triangle
  - Surface normals, shading, barycentric interpolation
- The OpenGL pipeline
  - Vertex vs. fragment shaders

OpenGL pipeline

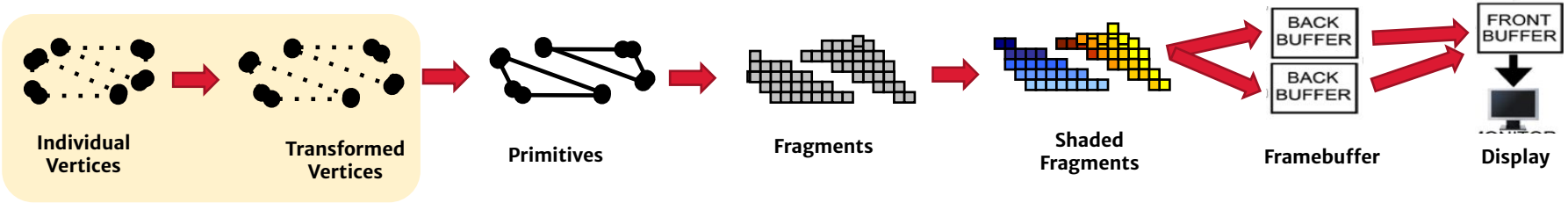
- Standard rasterization pipeline (OpenGL/DirectX)





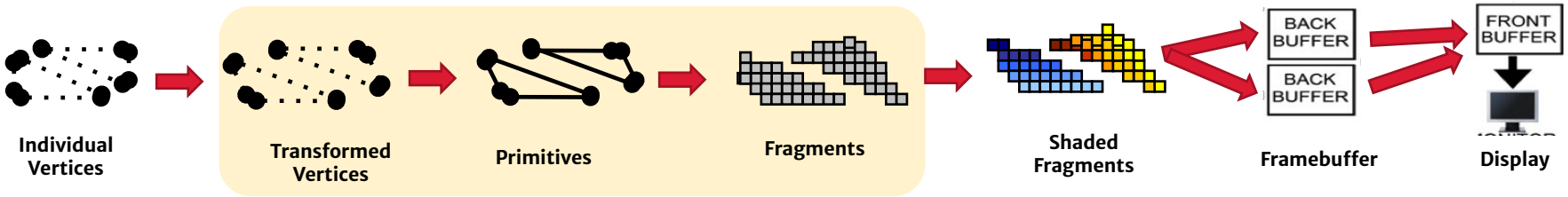
OpenGL pipeline

- Apply transformations to each vertex
- This includes transformations (from last lecture) and camera transformations (our next lecture)
- This happens in the **vertex shader** for per-vertex operations



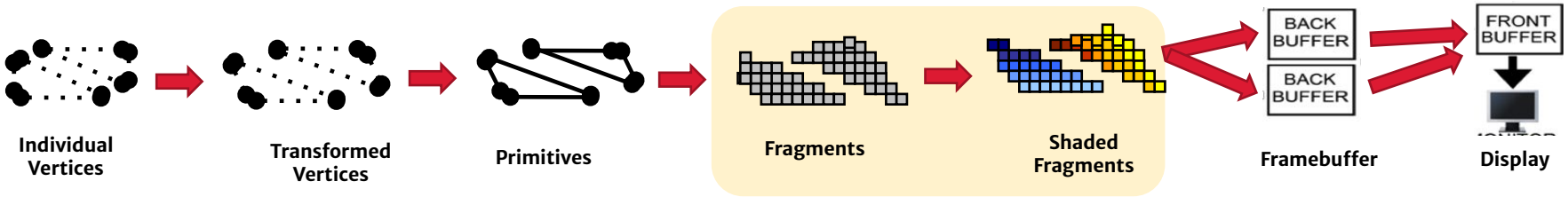
OpenGL pipeline

- Interpolate all per vertex quantities (like the normal) across every triangle for every pixel in the triangle
- Some vertices/shapes get discarded depending on certain factors (e.g. relation to camera)



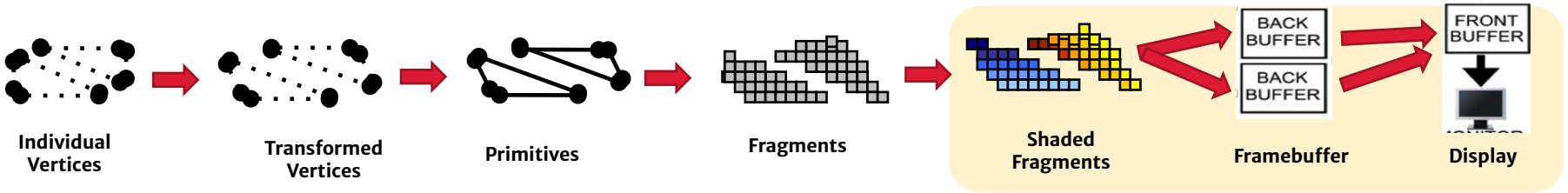
OpenGL pipeline

- Rasterization and shading happens in the **fragment shader** for per-pixel operations



OpenGL pipeline

- Display logistics
- Shaded fragments get written to a buffer in memory
- Once back buffer is processed, it gets swapped into the front buffer all at once
  - Avoids flickering / half drawn frames



Additional Resources + Exploring!

- The Utah Teapot
- Shadow Mapping
- Stanford CS348K
- OpenGL pipeline
- Metal API

- Fundamentals of Computer Graphics, Steve Marschner and Peter Shirley
  - Preview, Book