CS148 Summer 2025 - Lecture 3

07/1/2025

Rasterization and Shading

cs148.stanford.edu

Lecture Outline

- Recall last lecture:
  - Representation of 3D geometry (meshes, splines, etc.)

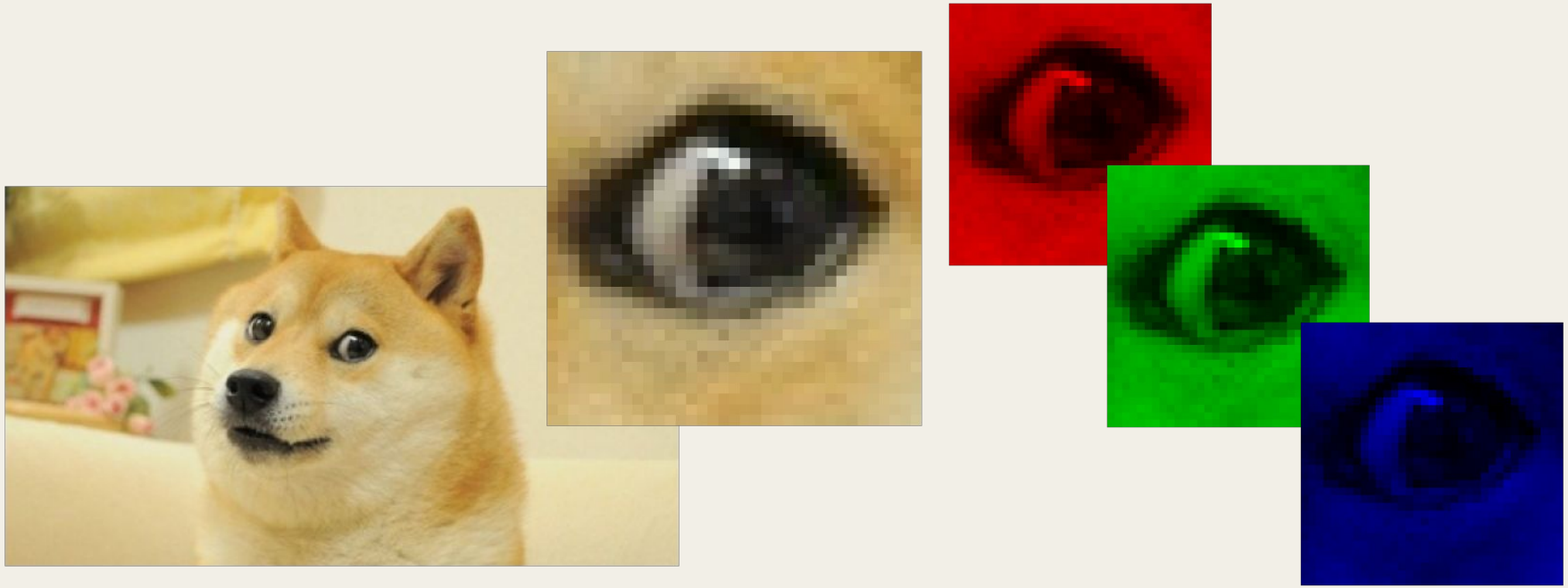
- Now we have ways to store and describe objects…
  how do we make them show up on screen?

Lecture Outline

- Image generation - the “simple/fast” version (1970s-)
  - How to draw a triangle - rasterization
  - How to color a triangle - surface normals, shading, barycentric interpolation
  - The OpenGL pipeline - vertex vs fragment shaders
  - Cameras
  - Comparison/segway to raytracing

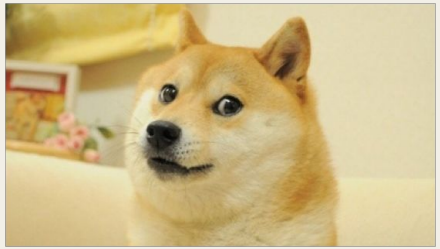
Image Representation

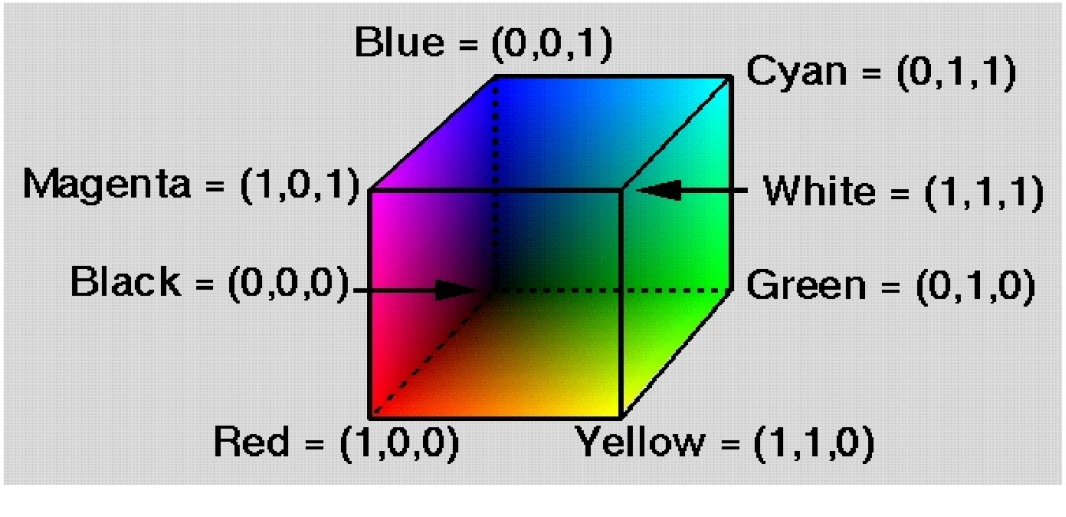
- Images: represented as a 2D grid of colors (pixels)
- Often in RGB values (1 grid for each color channel)



RGB Color Space

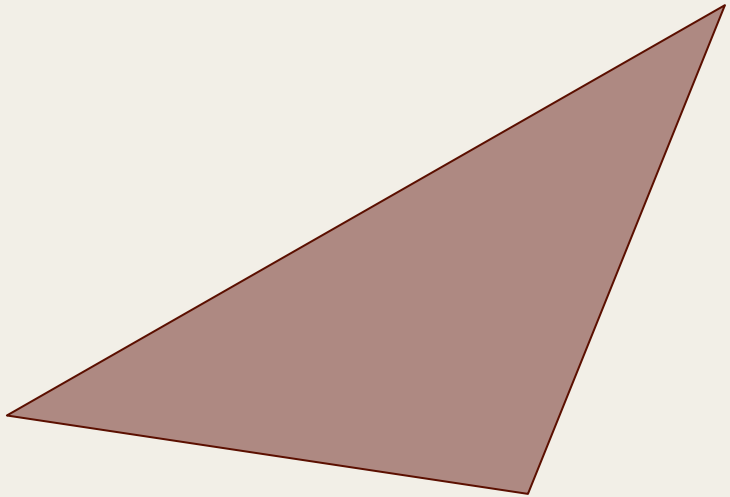
- Each image **I** is a 2D spatial grid of colors where (0,0,0)=black, (1,1,1)=white
- Other color spaces exist – next lecture





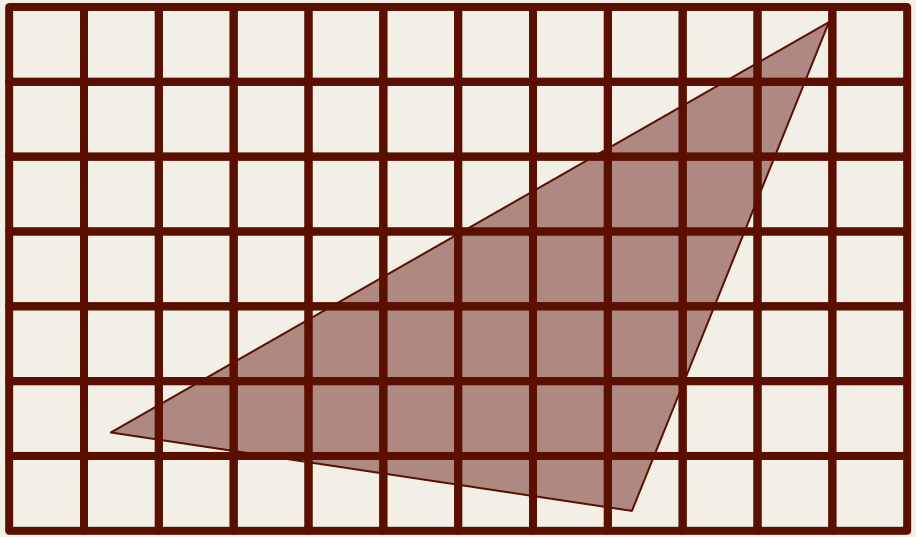
How to draw a shape on the display?

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



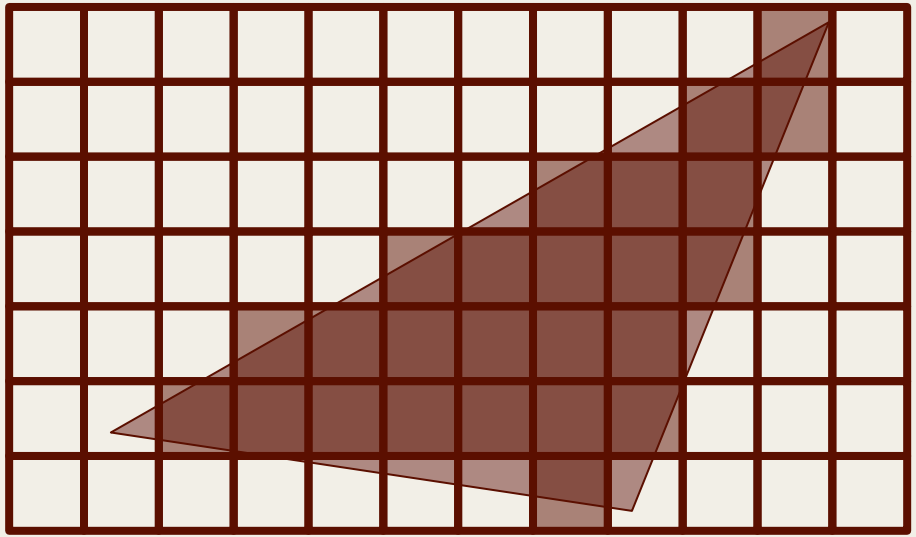
How to draw a shape on the display?

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



Rasterization

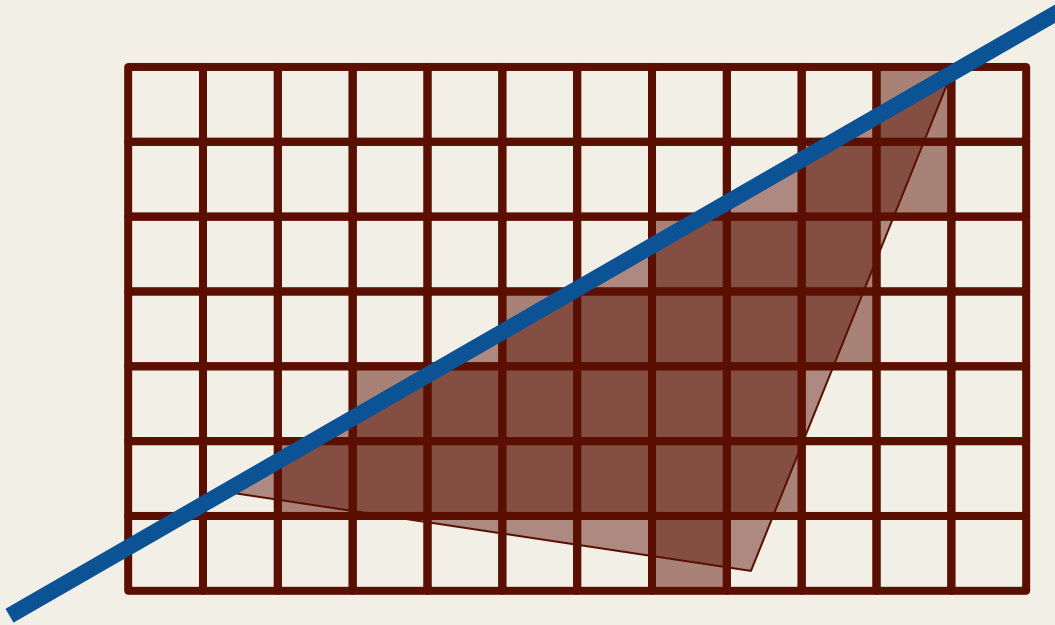
- For each pixel, if the center of the pixel is inside the triangle, consider it part of the triangle (and color it with the triangle's color.)
- How to determine whether a pixel center is inside?



Rasterization

- For each pixel, if the center of the pixel is inside the triangle, color it with the triangle's color.
- How to determine this?

High school algebra!
y=ax+b is also
ax+b-y=0



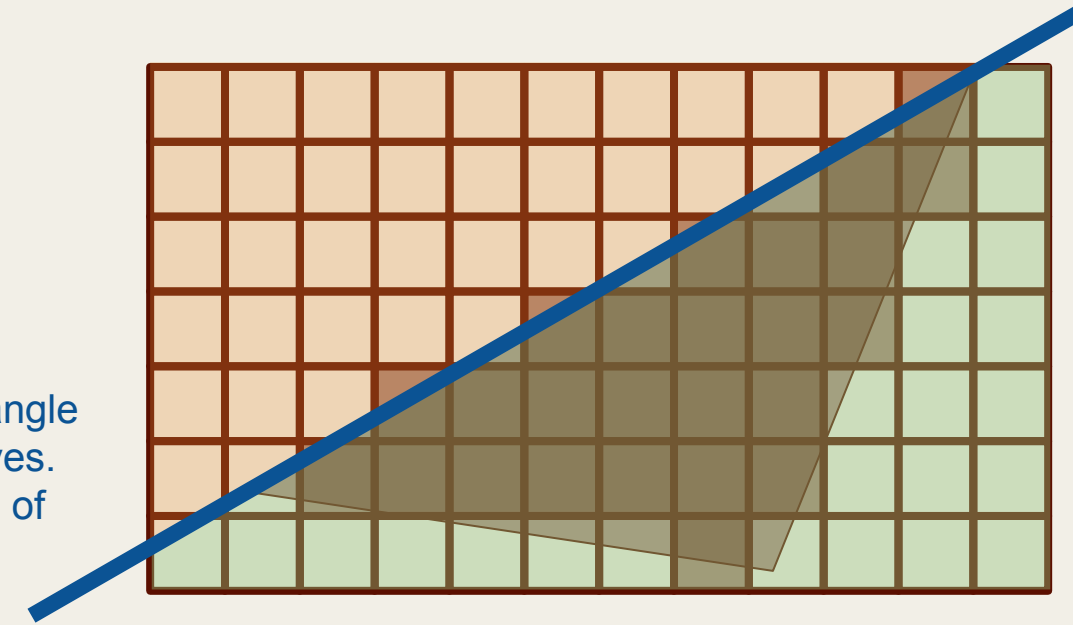
Rasterization

- For each pixel, if the center of the pixel is inside the triangle, color it with the triangle's color.
- How to determine this?

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

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

Each edge of the triangle splits space into halves. Take the intersection of 3 half-planes!

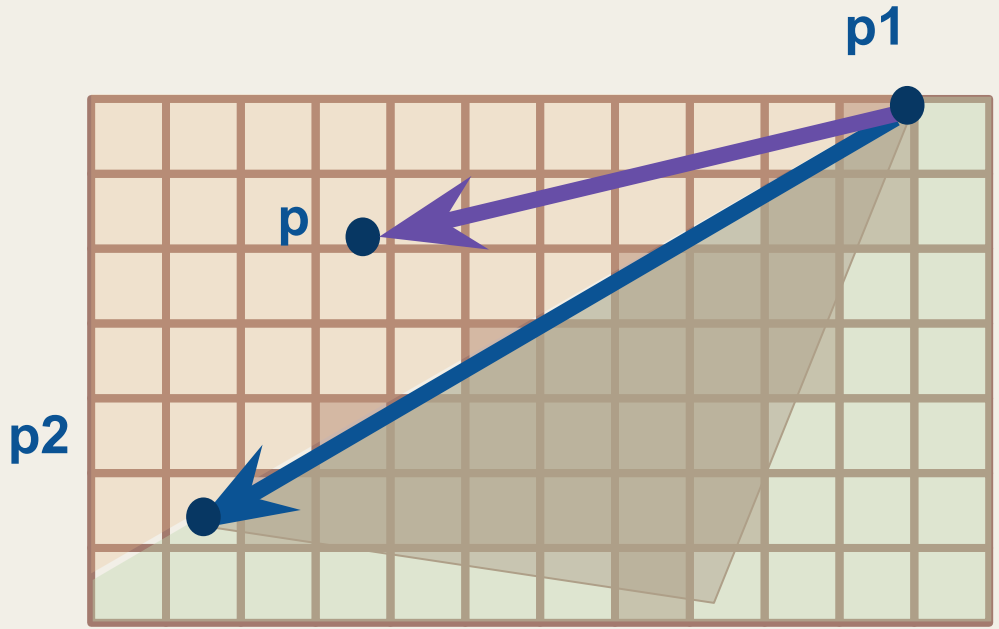


Rasterization

- For each pixel, if the center of the pixel is inside the triangle, color it with the triangle's color.
- How to determine this?

more direct definition:
use cross products!
Remember: right-hand rule

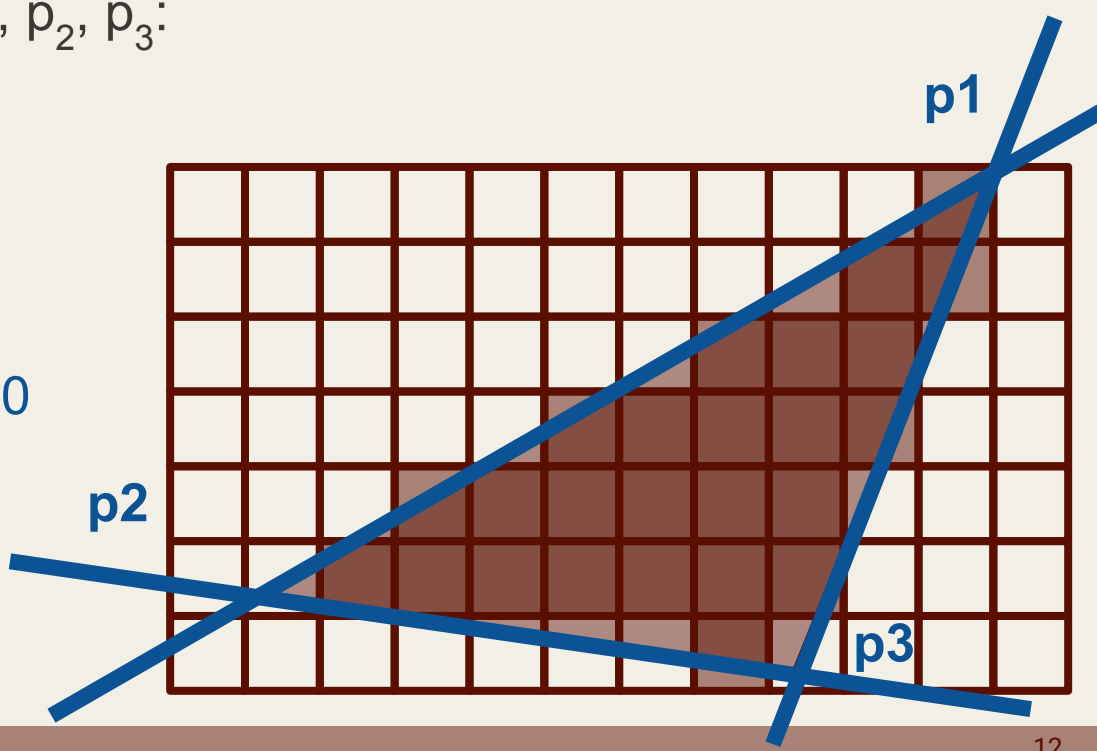
$(p-p_1) \times (p_2-p_1) < 0$
$(p-p_1) \times (p_2-p_1) = 0$
$(p-p_1) \times (p_2-p_1) > 0$



Rasterization

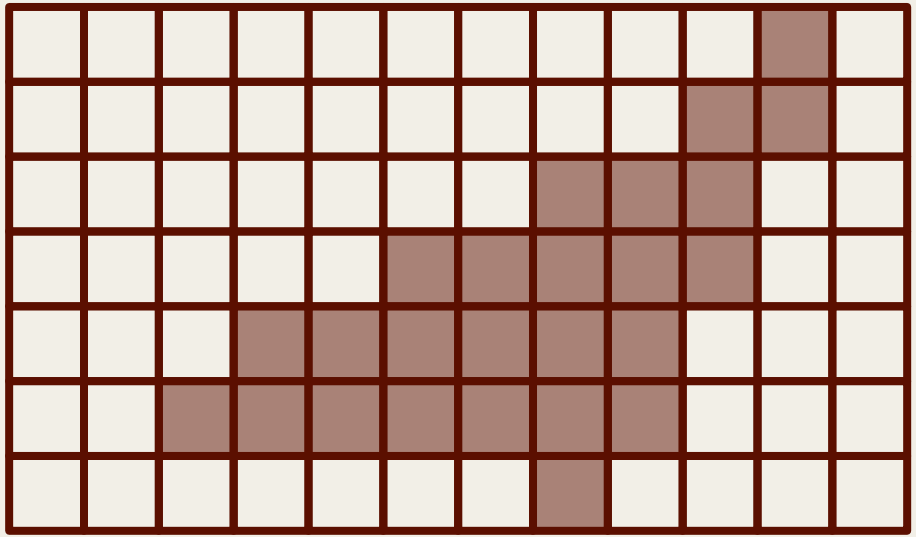
- Intersection of 3 half-planes
  Given triangle defined by p_1, p_2, p_3:
  $S1(p) = (p-p_1) \times (p_2-p_1)$
  $S2(p) = (p-p_2) \times (p_3-p_2)$
  $S3(p) = (p-p_3) \times (p_1-p_3)$

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



Rasterization

- Not perfect – lots of engineering details to consider
  - How to compute this most efficiently?
  - What happens when a bunch of triangles intersect at pixel center?
  - Make things less jagged, by computing how much of a pixel is within a triangle?



- We now know how to draw a 2D triangle.
  How did we get the 3D triangle to 2D in the first place?

  – topic of next lecture!

Lecture Outline

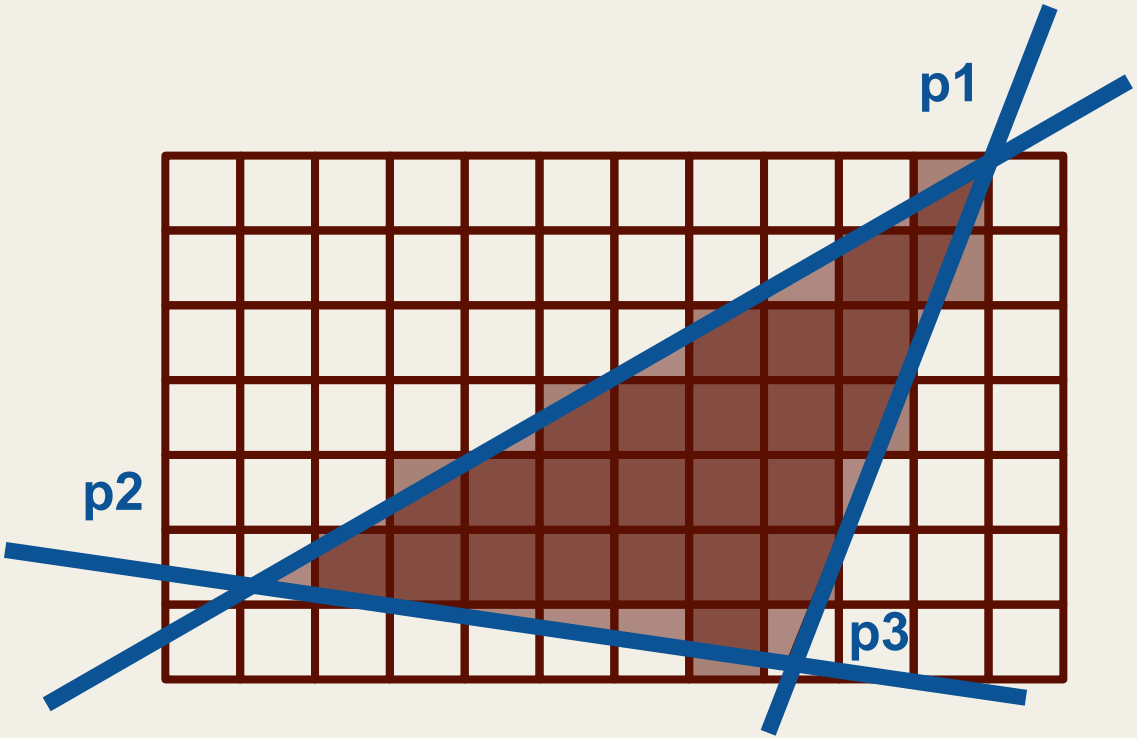
- Image generation - the “simple/fast” version (1970s-)
  - ~~How to draw a triangle - rasterization~~
  - How to color a triangle - surface normals, shading, barycentric interpolation
  - The OpenGL pipeline - vertex vs fragment shaders
  - Cameras
  - Comparison/segway to raytracing

How to shade a triangulated object?

How to shade a triangulated object?

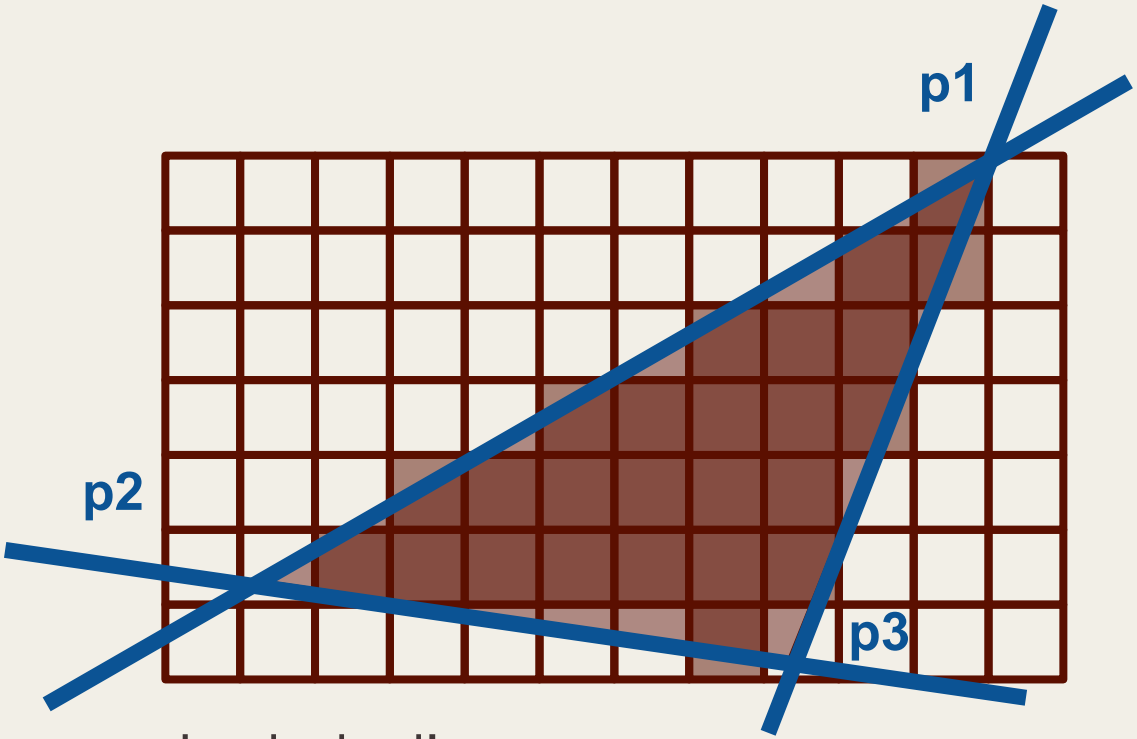
- Fun Fact: The Utah teapot (1975) is the first implicit surface model
- University of Utah: Where a lot of graphics research first started.
  Ivan Sutherland
  1962 - Sketchpad, precursor to CAD/drawing tablets
  1968 - First VR headset

Flat Shading



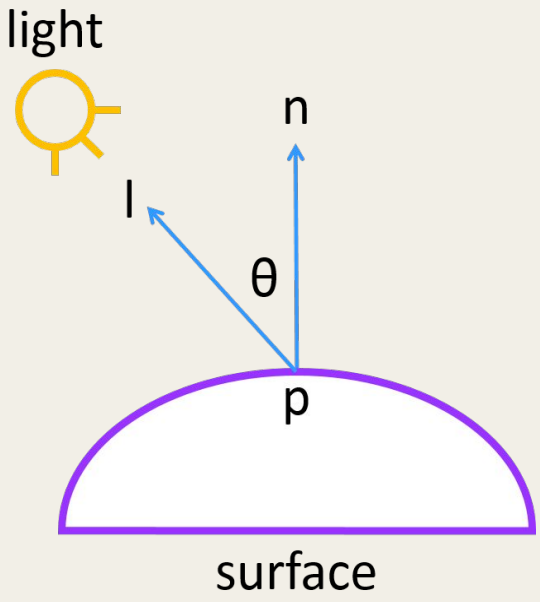
- Color every pixel inside a triangle with average of colors at vertices.

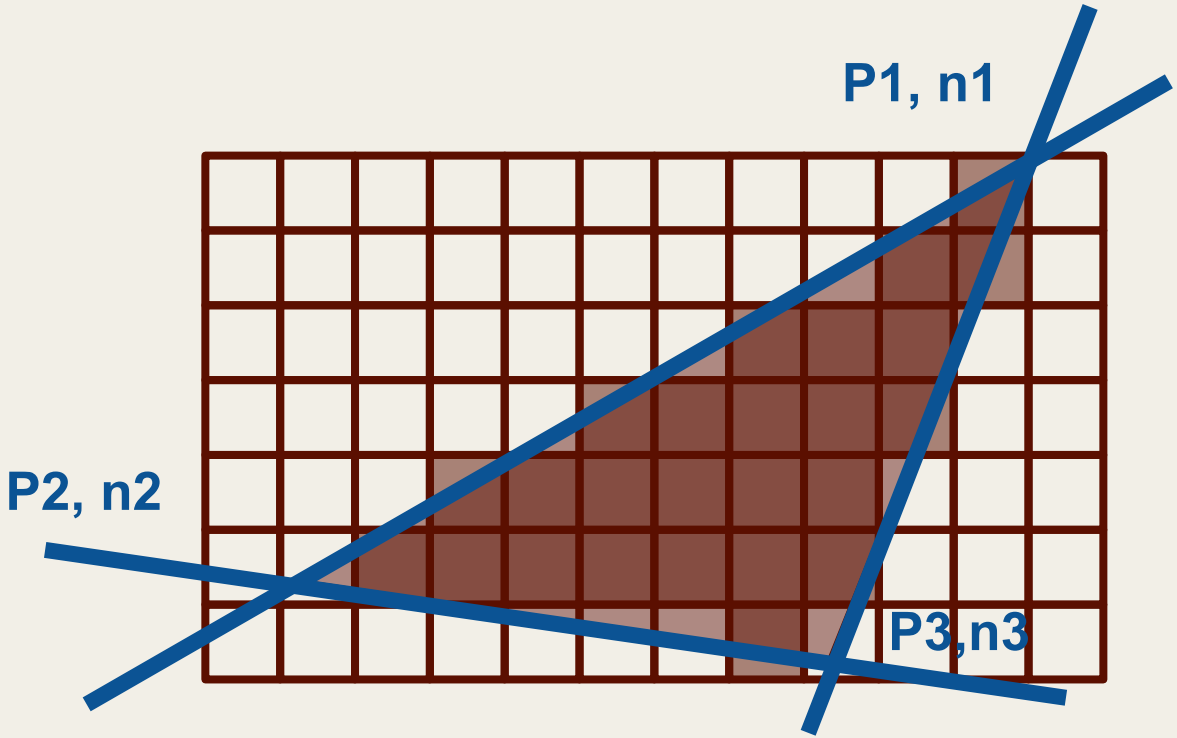
Flat Shading



- Looks bad!
- Also storing colors at vertices is not robust – think change of light.

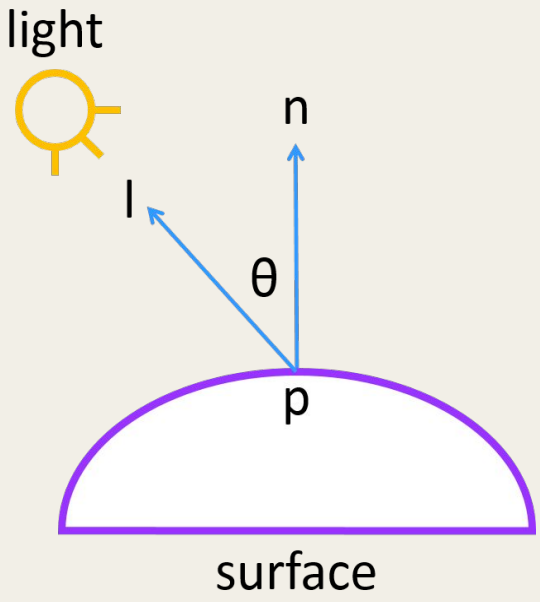
Surface Normals

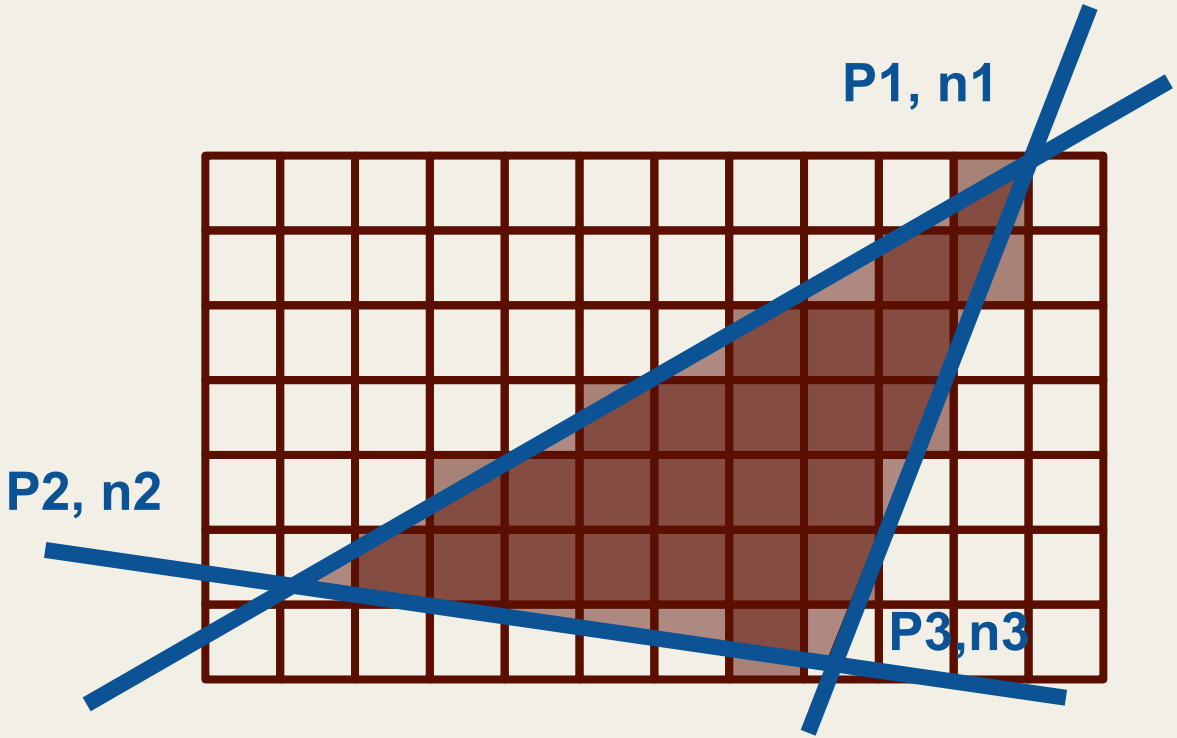




- At each vertex, store the **normal vector** that points outwards from the object surface in 3D space at that vertex.

Surface Normals





- Have n1,n2,n3 now in addition to p1,p2,p3!
- OBJ files typically store both list of vertices and list of normals.

Phong Reflection Model

Developed circa 1973 (again, at U. of Utah)
We'll cover light/optics more rigorously later,
but for now, let's talk heuristics!

Phong Reflection Model

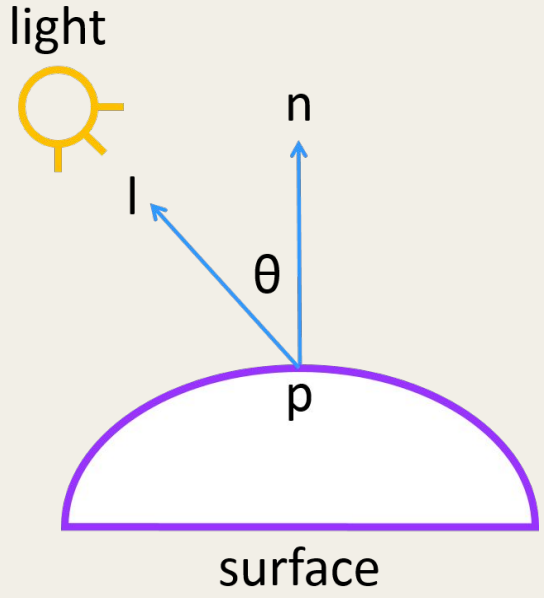
- **Ambient:** “Base color”. Light bounces around the environment.
  Even if you have little light, objects in shadow aren’t usually pure black.
- **Diffuse:** “Rough material”. Given the same light source, objects look brighter when hit “perpendicularly”. (Will talk about why later in Lights and Optics!)
- **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!)

- Let c be the color we’re computing at p
- Let the vector from p to the light be l
- Let the normal vector at p be n
- Lambert’s Cosine Law: $c \propto \cos\theta$
  (from physicist Johann Heinrich Lambert (1760))



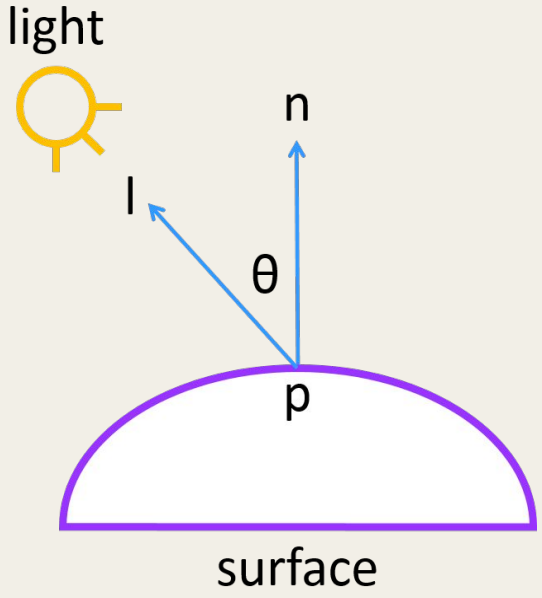
Phong Reflection Model - Diffuse

- **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 is also the dot product of the 2 vectors!

  So… $c \propto \cos\theta$

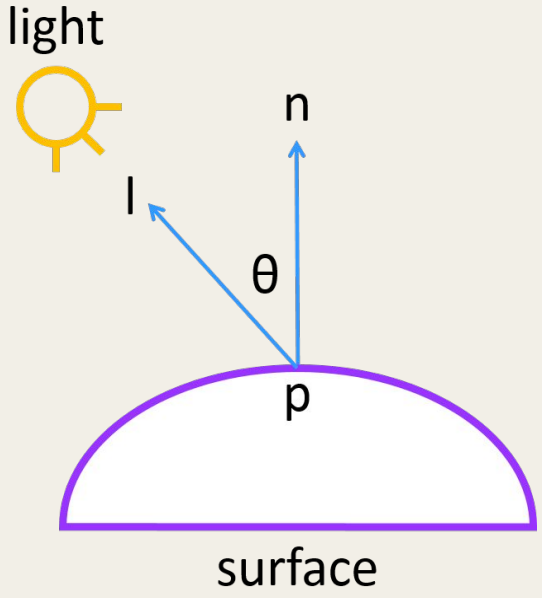
  $= c \propto n \cdot l$



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!)

- So… $c \propto \cos\theta$
  $= c \propto n \cdot l$
- Define Cd and Cl to represent:
  - Cd, the diffuse material of the surface (represent the “roughness” of the surface)
  - Cl, the color of the light
- $c_{diffuse} = c_d c_l n \cdot l$

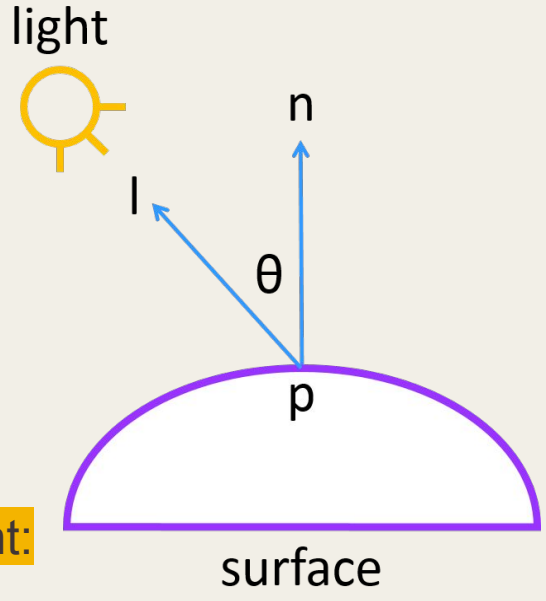


Phong Reflection Model - Diffuse

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

- So… $c \propto \cos\theta$
  $= c \propto n \cdot l$
- Define Cd and Cl to represent:
  - Cd, the diffuse material of the surface (represent the “roughness” of the surface)
  - Cl, the color of the light
- $c_{diffuse} = c_d c_l n \cdot l$

  Need a MAX for objects facing away from the light:
  $c_{diffuse} = c_d c_l max(0, n \cdot l)$.



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 - Ambient

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

- Very simplified concept – just define a term Ca to represent the color to give the object if no light!
- So… $c_{ambient} = c_a$
- Putting ambient + diffuse together:

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

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 - Specular

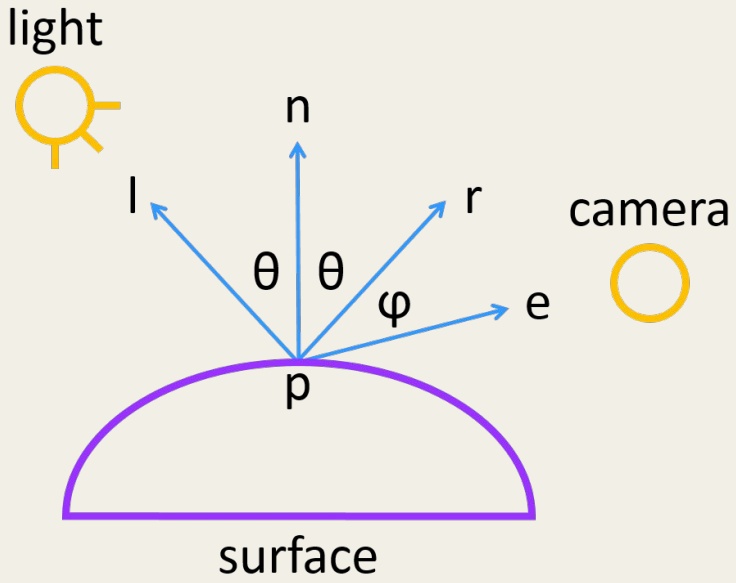
- **Specular:** “Shiny material”. Bright highlights when light reflects into our eyes.

- Similar to diffuse, define Cs and Cl:
  - Cs, the specular material of the surface (represent how well surface reflects light)
  - Cl, the color of the light
- Let r be the reflection of the l across n

  Let e be a vector from p to the camera/eye

$$c_{specular} = c_s c_l e \cdot r$$

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



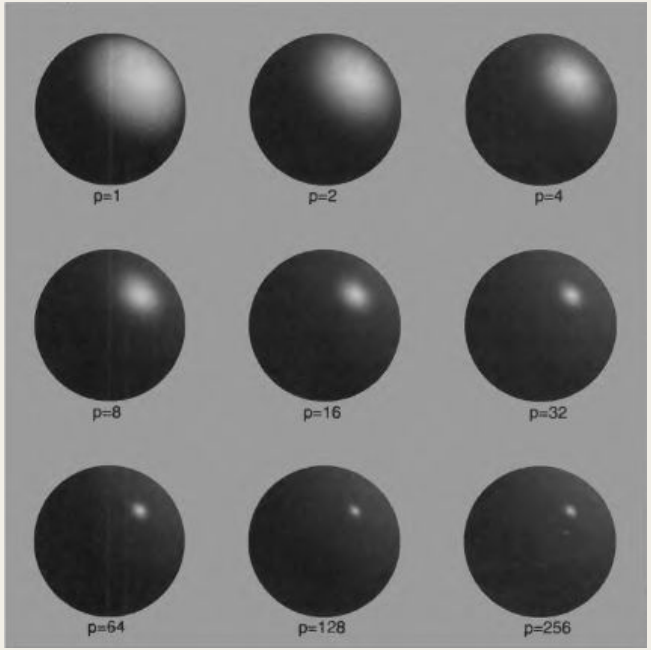
Phong Reflection Model - Specular

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

- $c_{specular} = c_s c_l max(0, e \cdot r)$
- Also introduce a “shininess value” called **the Phong exponent** that specifies how shiny we want to tune the material.

  Denote as p or alpha:

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

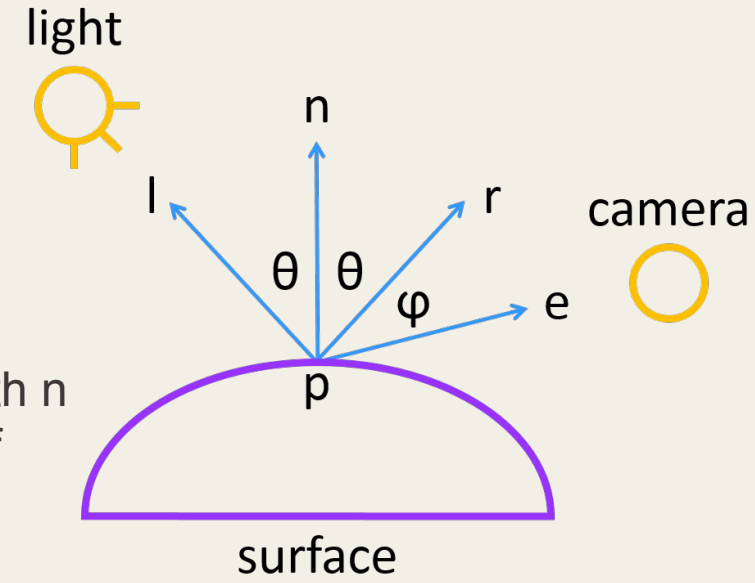


Phong Reflection Model - Specular

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

- $c_{specular} = c_s c_l max(0, e \cdot r)^{\alpha}$
- Problem: r is not trivial to compute

  Easier approach:
  - Compute halfway vector between l and e as new vector h
  - When e lines up with r, then h lines up with n
  - Can approximate using the dot product of n and h, instead of e and r



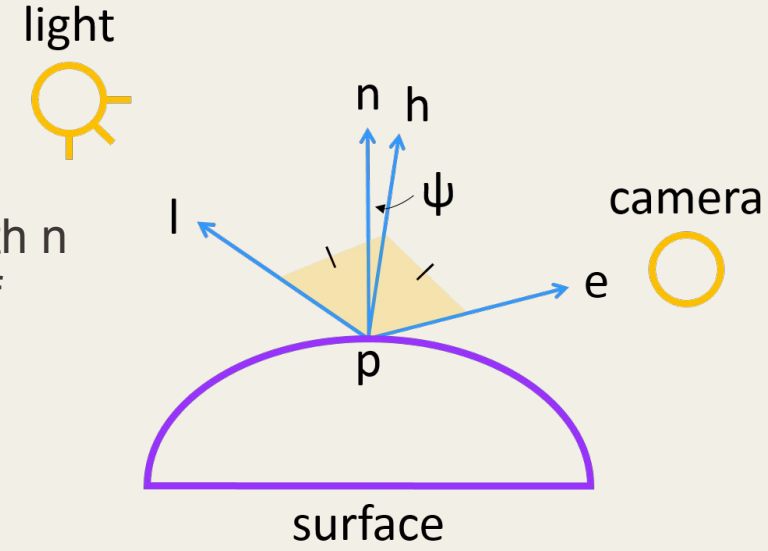
Phong Reflection Model - Specular

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

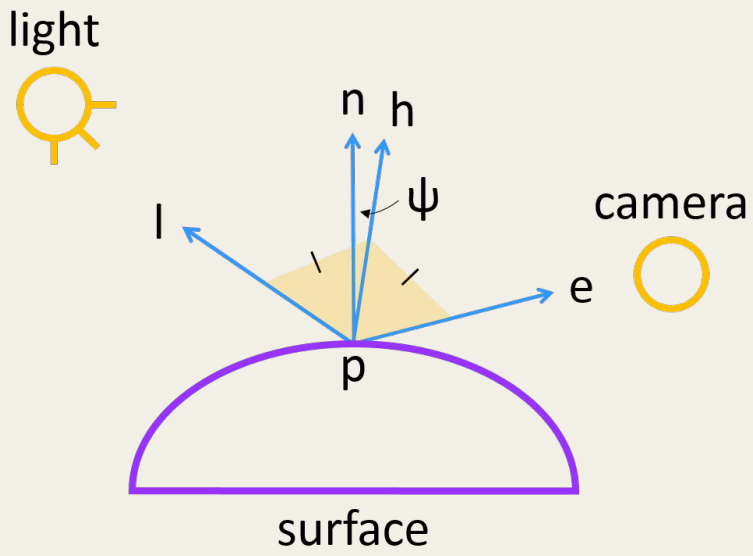
- Easier approach:
  - Compute halfway vector between l and e as new vector h
  - When e lines up with r, then h lines up with n
  - Can approximate using the dot product of n and h, instead of e and r

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

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



Phong Reflection Model aka Lighting Model



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

- Putting it all together:

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

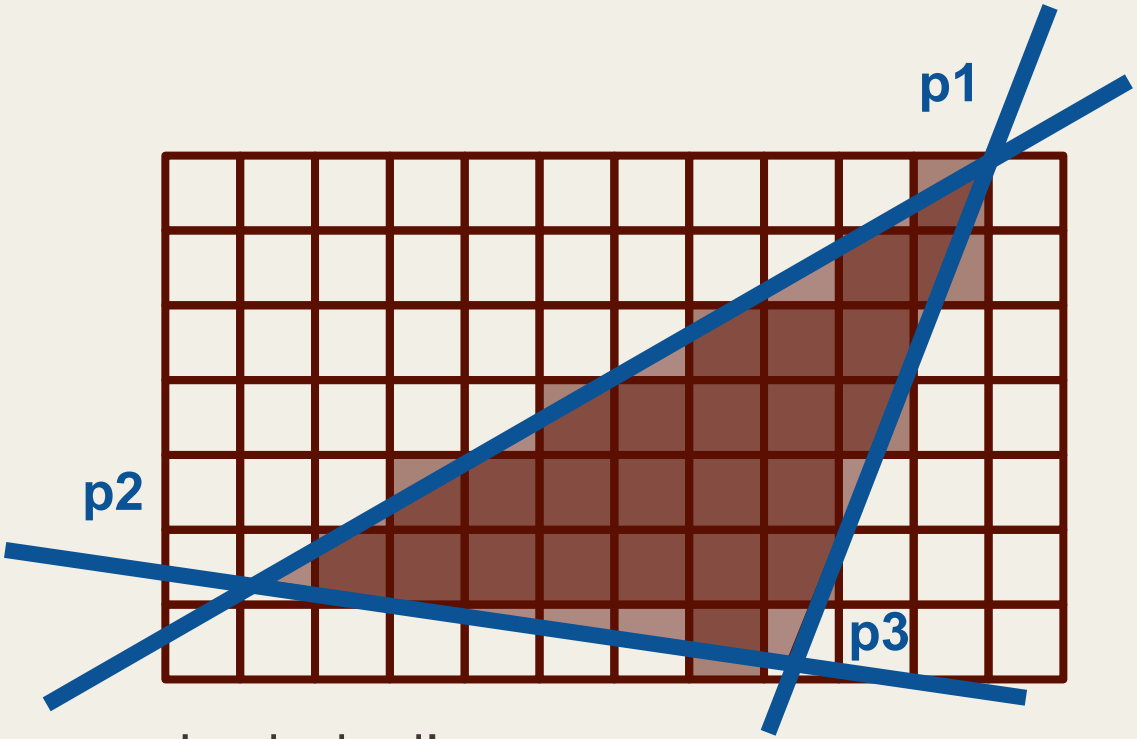
$$c = c_a + c_d c_l max(0, n \cdot l) + c_s c_l max(0, n \cdot h)^{\alpha}$$

Phong Reflection Model

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

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

Flat Shading



- Looks bad!
- ~~Also storing colors at vertices is not robust – think change of light.~~

Smooth vs Flat shading

How do we make it look smooth?
(Approximating smooth geometry with triangles)

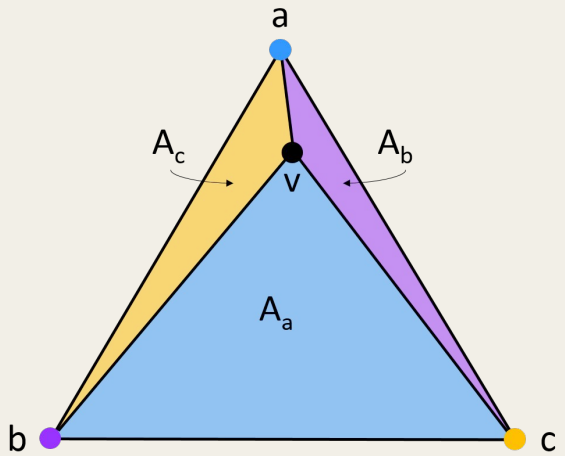
Barycentric Interpolation

- Suppose you have color values computed at each vertex a, b, and c of your triangle.
- For any point v inside your triangle, divide the up the triangle into 3 triangles with separate areas

  The interpolation for the color at v is then:

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

- This is called Gourad Shading!



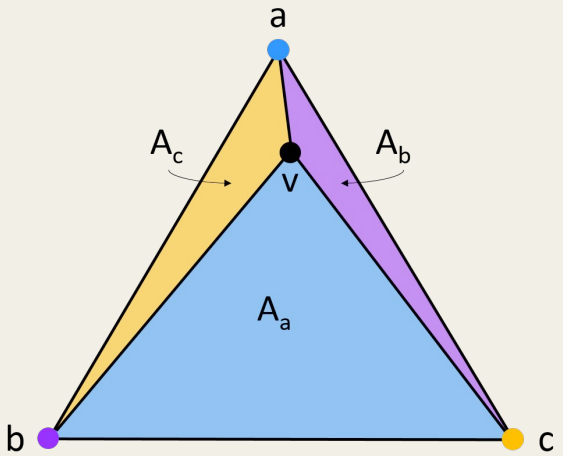
Barycentric Interpolation

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

  The interpolation for the **normal** at v is then:

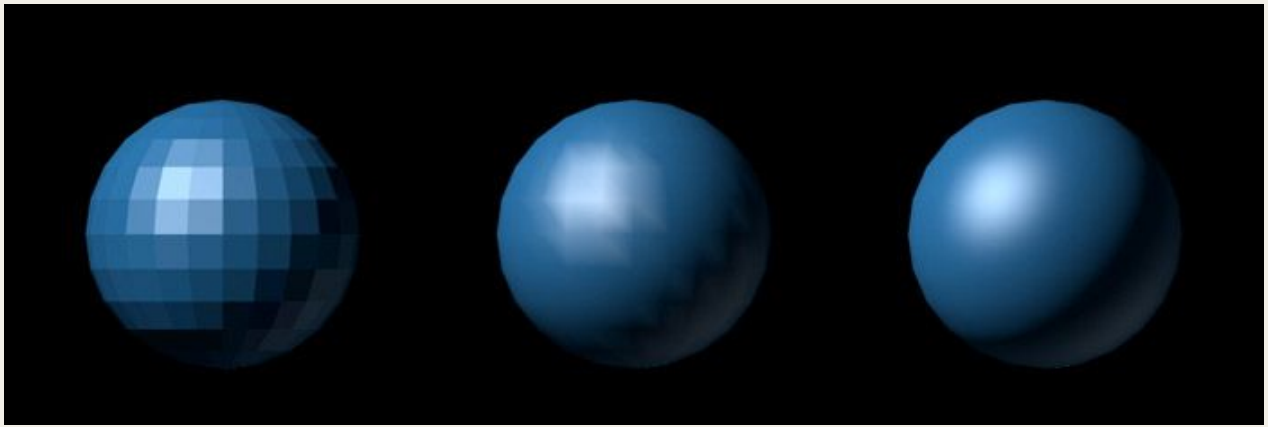
$$n_v = \frac{A_a}{A_{total}} n_a + \frac{A_b}{A_{total}} n_b + \frac{A_c}{A_{total}} n_c$$

  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.
- 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!

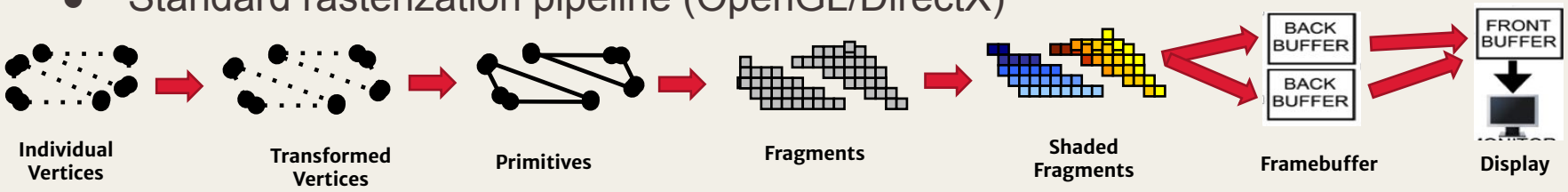


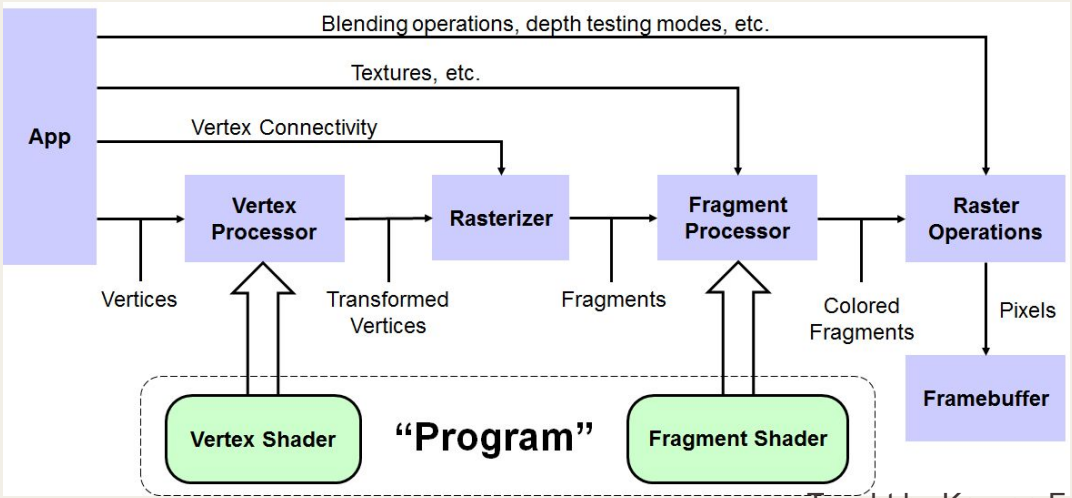
Lecture Outline

- Image generation - the “simple/fast” version (1970s-)
  - ~~How to draw a triangle - rasterization~~
  - ~~How to color a triangle - surface normals, shading, barycentric interpolation~~
  - The OpenGL pipeline - vertex vs fragment shaders
  - Cameras
  - Comparison/segway to raytracing

Tying everything together…

- Standard rasterization pipeline (OpenGL/DirectX)





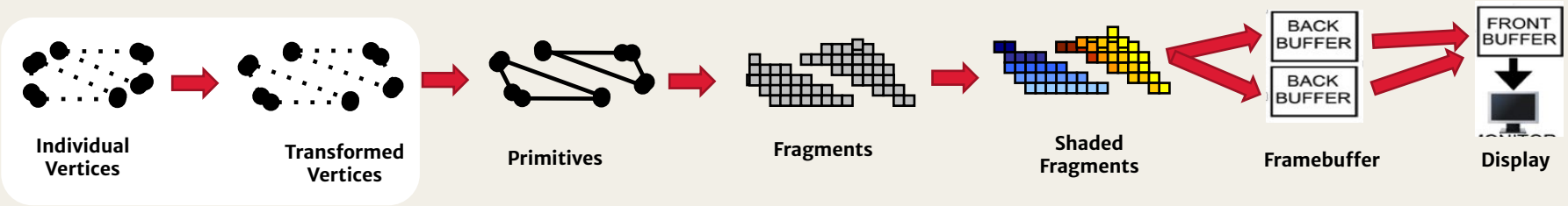
Cornell CS4620

Taught by Kayvon Fatahalian in Spring:
https://gfxcourses.stanford.edu/cs348k/spring25

Tying everything together…

- Step 1: Apply transformations to each vertex. This includes transformations as mentioned last lecture and also camera transformations from the next lecture.

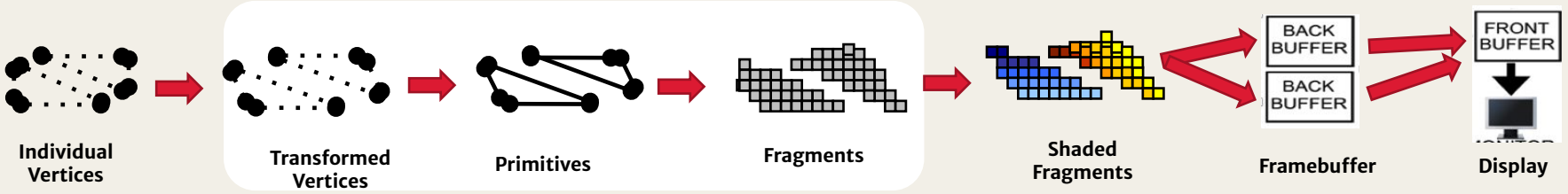
  This all happens in what we call the **vertex shader** for per-vertex operations.



Tying everything together…

- Step 1: Apply transformations to each vertex. This includes transformations as mentioned last lecture and also camera transformations from the next lecture.

  This all happens in what we call the **vertex shader** for per-vertex operations**.**
- Step 2: Interpolate all per-vertex quantities (like the normal!) across every triangle for every pixel in the triangle..

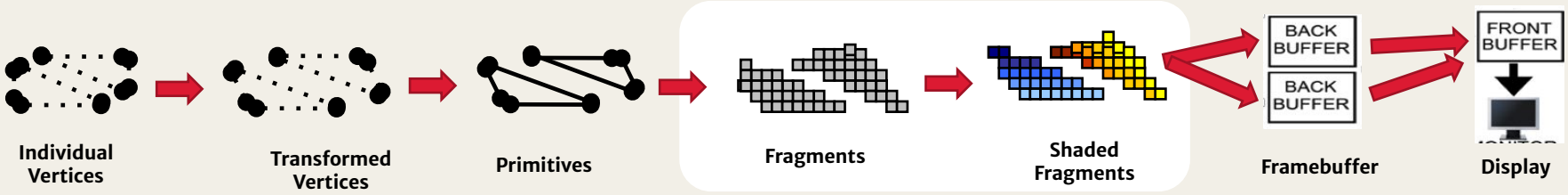


Tying everything together…

- Step 1: Apply transformations to each vertex. This includes transformations as mentioned last lecture and also camera transformations from the next lecture.

  This all happens in what we call the **vertex shader** for per-vertex operations.
- Step 2: Interpolate all per-vertex quantities (like the normal!) across every triangle, then compute the color for every pixel in a triangle.

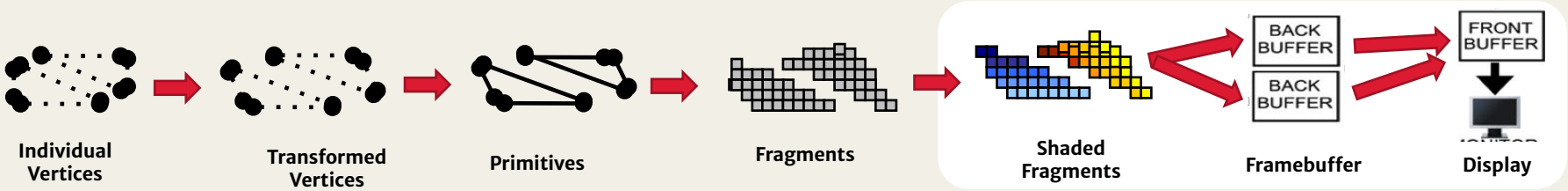
  This all happens in what we call the **fragment shader** for per-pixel operations.



Tying everything together…

- Step 1: Apply transformations to each vertex. This includes transformations as mentioned last lecture and also camera transformations from the next lecture. This all happens in what we call the **vertex shader** for per-vertex operations**.**
- Step 2: Interpolate all per-vertex quantities (like the normal!) across every triangle, then compute the color for every pixel in a triangle. This all happens in what we call the **fragment shader** for per-pixel operations.

- Last step for next class…



Limitations

- Rasterization = taping triangle shaped stickers onto a screen
  - How do you do transparency?
  - How do you do shadows/self-occlusions?
  - How do you handle complex phenomena/light transport?
- Some of these we can “hack” by doing multiple-passes of rasterization
  - Shadow mapping
- More physically-based way: **raytracing** and pathtracing
- Speed is still a big advantage:
  Real-time graphics, pre-viz for more expensive rendering methods