The Graphics Pipeline and OpenGL II:
Lighting and Shading, Fragment Processing

Gordon Wetzstein

Stanford University

EE 267 Virtual Reality

Lecture 3

stanford.edu/class/ee267/

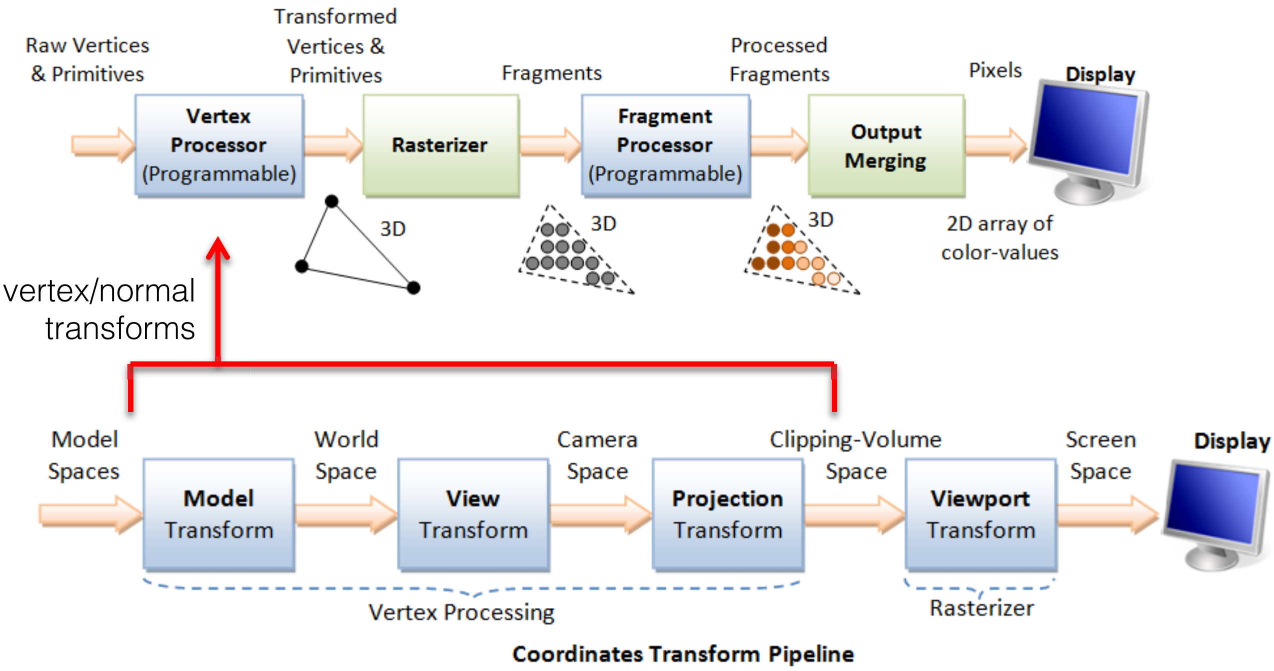
Announcements

- Waitlist is getting smaller, so stay on it if you're planning on taking the class; some students also offered to share kits
- questions for HW1? post on Ed Discussion and zoom office hours!
- WIM workshop 1: this Friday 2-3 pm, Packard 204 → if you are a WIM student, you ~~must~~ should attend!
- WIM HW1 going out this Friday

Lecture Overview

- rasterization
- the rendering equation, BRDFs
- lighting: computer interaction between vertex/fragment and lights
    - Phong lighting
- shading: how to assign color (i.e. based on lighting) to each fragment
    - Flat, Gouraud, Phong shading
- vertex and fragment shaders
- texture mapping

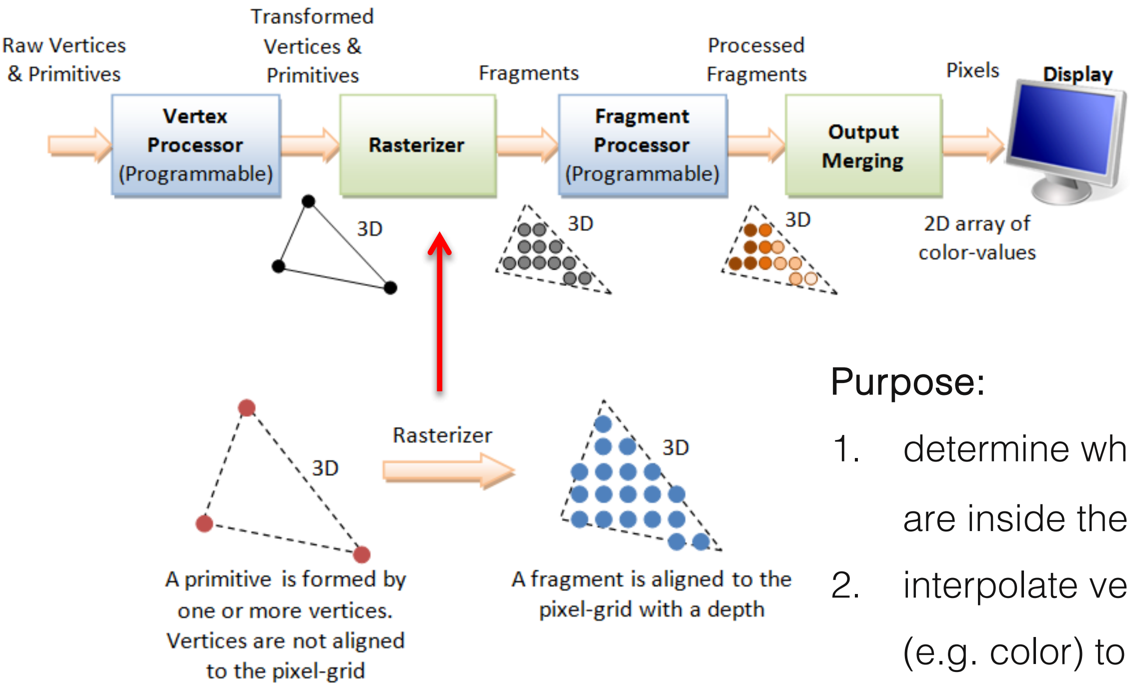
Review of Vertex/Normal Transforms



https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Rasterization

Rasterization

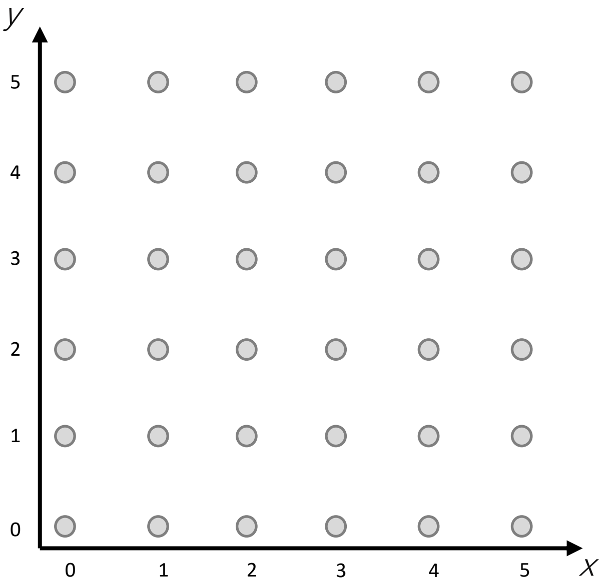


Purpose:

1. determine which fragments are inside the triangles
2. interpolate vertex attributes (e.g. color) to all fragments

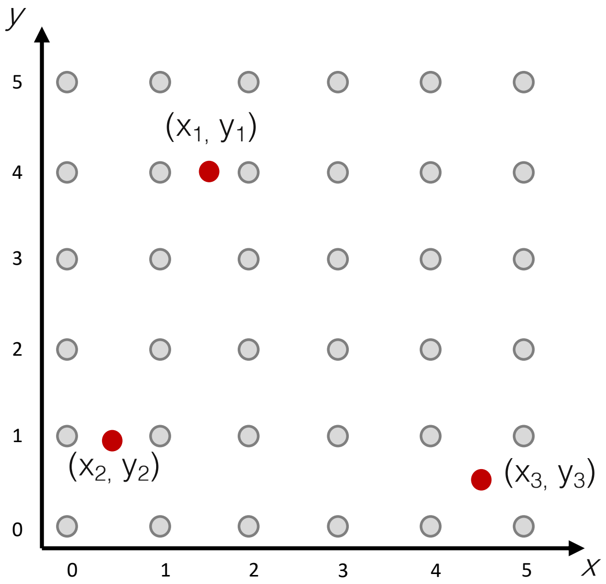
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Rasterization / Scanline Interpolation



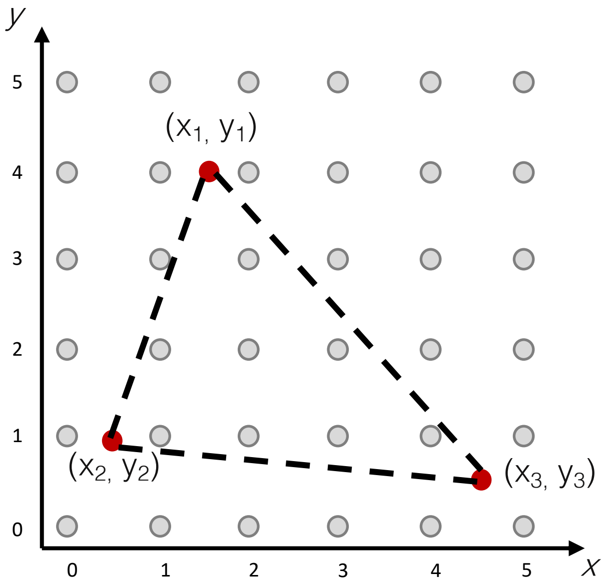
- grid of 6x6 fragments

Rasterization / Scanline Interpolation



- grid of 6x6 fragments
- 2D vertex positions after transformations

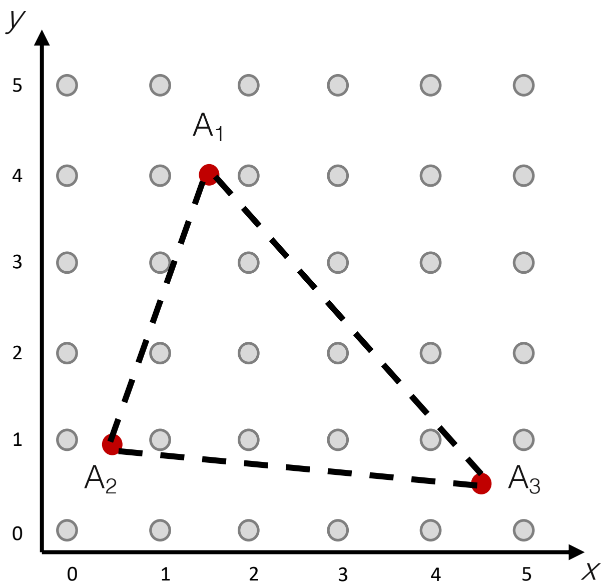
Rasterization / Scanline Interpolation



- grid of 6x6 fragments
- 2D vertex positions after transformations

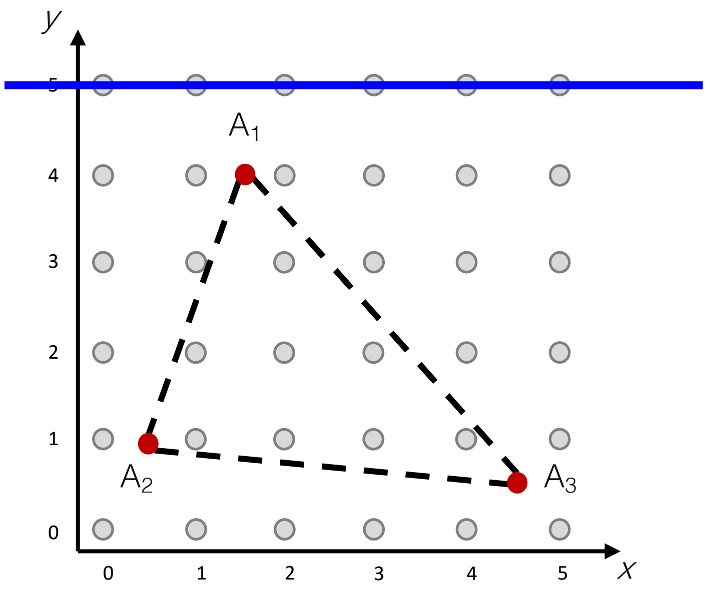
  \+ edges = triangle

Rasterization / Scanline Interpolation



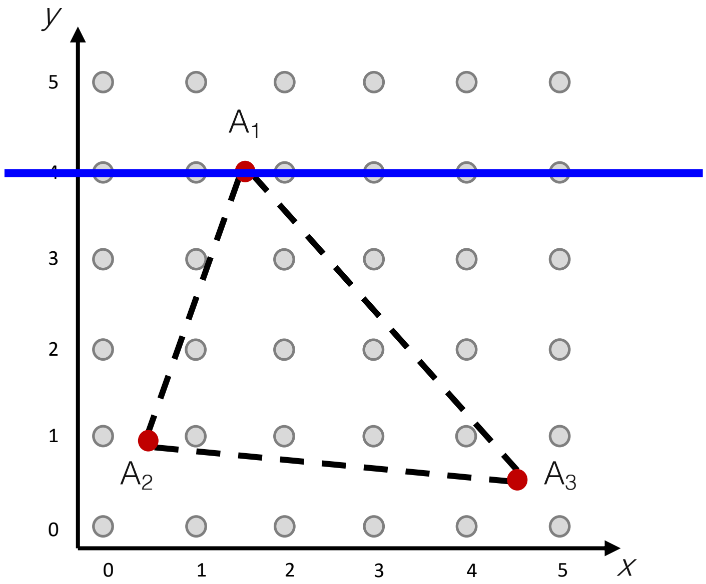
- grid of 6x6 fragments
- 2D vertex positions after transformations + edges = triangle
- each vertex has 1 or more attributes A, such as R/G/B color, depth, …
- user can assign arbitrary attributes, e.g. surface normals

Rasterization / Scanline Interpolation



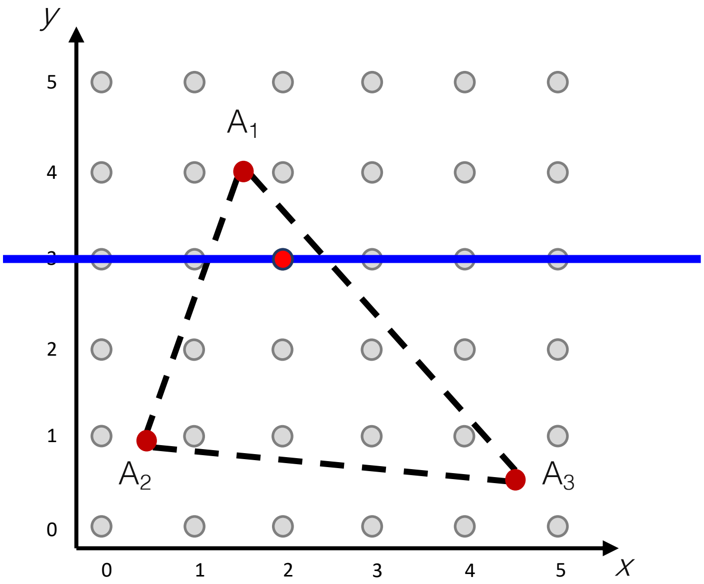
- scanline moving top to bottom

Rasterization / Scanline Interpolation



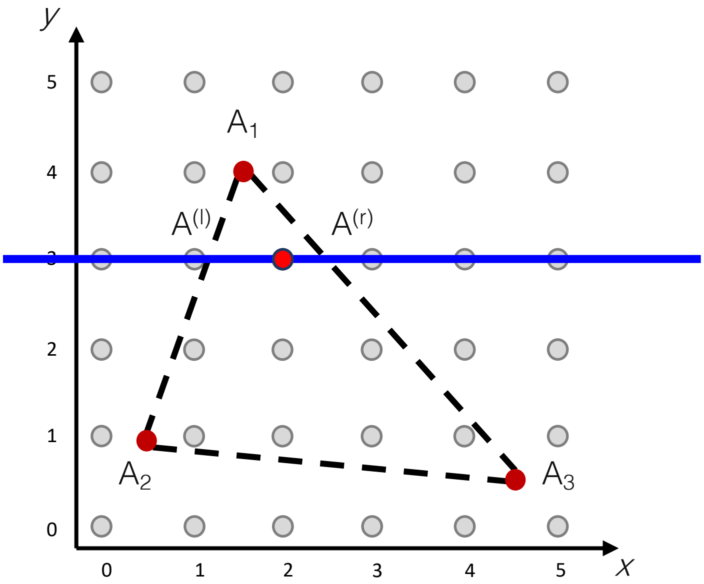
- scanline moving top to bottom

Rasterization / Scanline Interpolation



- scanline moving top to bottom
- determine which fragments are inside the triangle

Rasterization / Scanline Interpolation

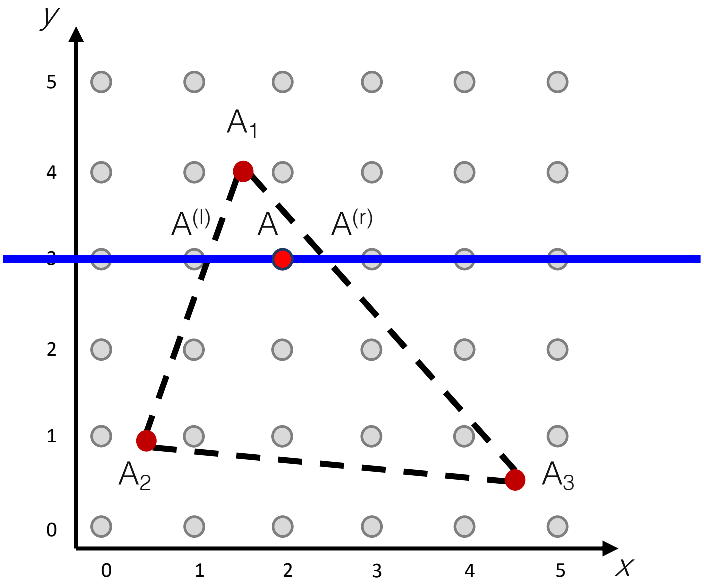


- scanline moving top to bottom
- determine which fragments are inside the triangle
- **interpolate attribute along edges in y**
- $y^{(l/r)}$ are the y coordinates of $A^{(l/r)}$, i.e. the y coordinate of the scanline

$$A^{(l)} = \left(\frac{y^{(l)} - y_2}{y_1 - y_2}\right) A_1 + \left(\frac{y_1 - y^{(l)}}{y_1 - y_2}\right) A_2$$

$$A^{(r)} = \left(\frac{y^{(r)} - y_3}{y_1 - y_3}\right) A_1 + \left(\frac{y_1 - y^{(r)}}{y_1 - y_3}\right) A_3$$

Rasterization / Scanline Interpolation

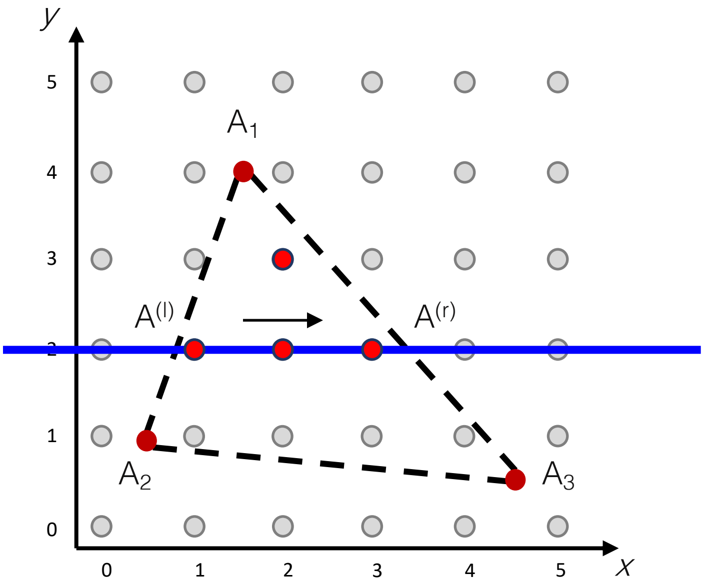


- scanline moving top to bottom
- determine which fragments are inside the triangle
- interpolate attribute along edges in y
- **then interpolate along x**
- $x^{(l/r)}$ are the x coordinates of $A^{(l/r)}$, which can be computed via similar triangles

$$A = \left(\frac{x - x^{(l)}}{x^{(r)} - x^{(l)}}\right) A^{(r)} + \left(\frac{x^{(r)} - x}{x^{(r)} - x^{(l)}}\right) A^{(l)}$$

final, interpolated attribute A at fragment

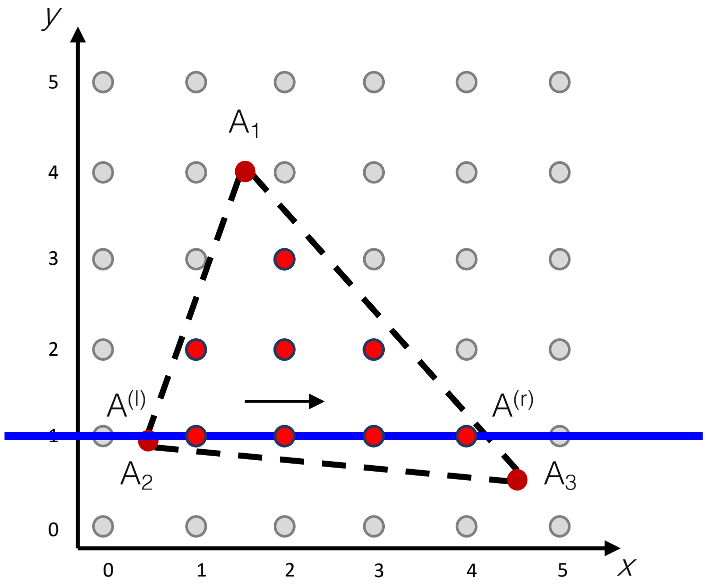
Rasterization / Scanline Interpolation



repeat:

- interpolate attribute along edges in y
- then interpolate along x

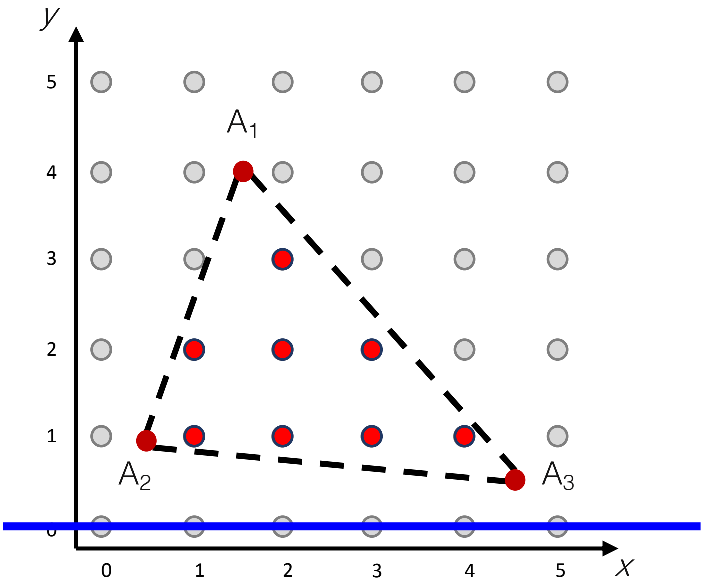
Rasterization / Scanline Interpolation



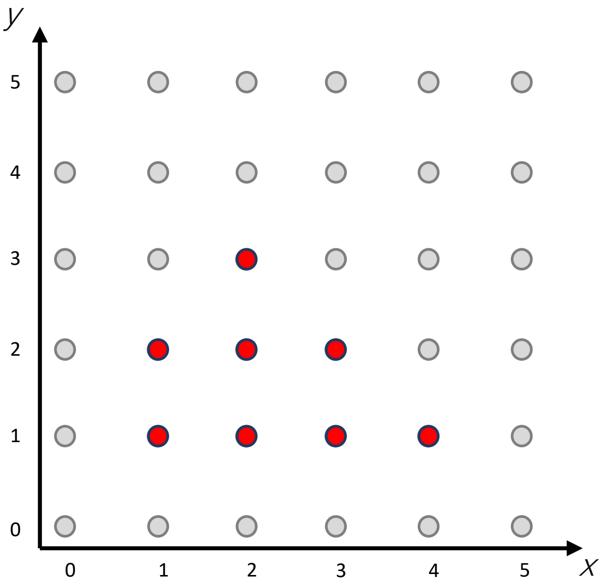
repeat:

- interpolate attribute along edges in y
- then interpolate along x

Rasterization / Scanline Interpolation



Rasterization / Scanline Interpolation

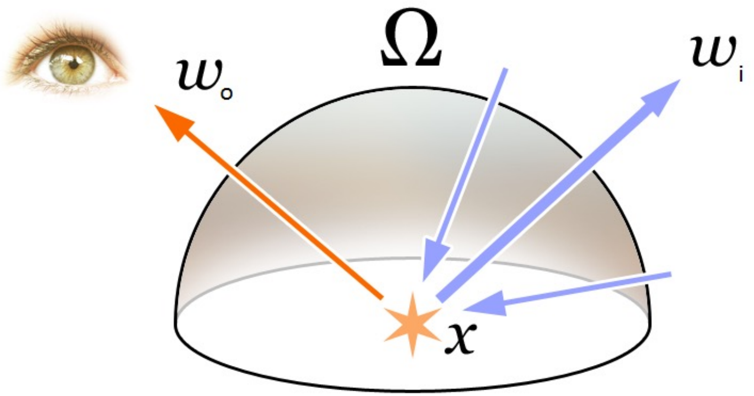


<u>output</u>: set of fragments inside triangle(s) with interpolated attributes for each of these fragments

Lighting & Shading

(how to determine color and what attributes to interpolate)

The Rendering Equation



- <u>direct (local) illumination</u>:

  light source → surface → eye

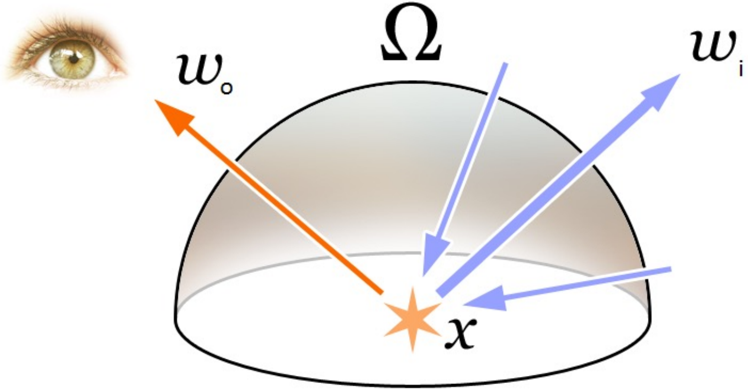
- <u>indirect (global) illumination</u>:

  light source → surface→ … → surface → eye

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

Kajija “The Rendering Equation”, SIGGRAPH 1986

The Rendering Equation



- <u>direct (local) illumination</u>:

  light source → surface → eye

- <u>indirect (global) illumination</u>:

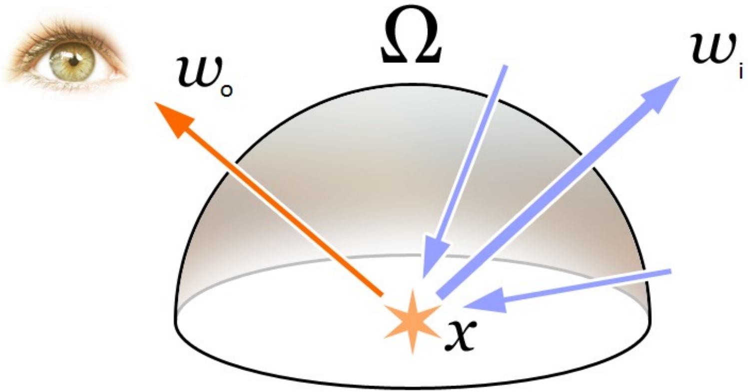
  light source → surface→ … → surface → eye

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

radiance towards viewer — emitted radiance — BRDF — incident radiance from some direction

Kajija "The Rendering Equation", SIGGRAPH 1986

The Rendering Equation



- <u>direct (local) illumination</u>:

light source → surface → eye

- <u>indirect (global) illumination</u>:

light source → surface→ … → surface → eye

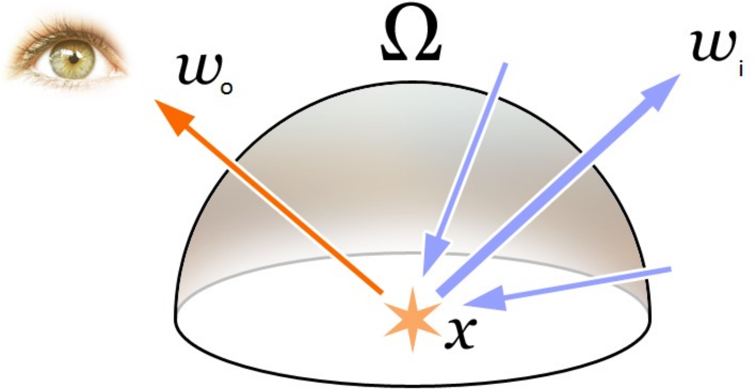
3D location

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

radiance towards viewer — emitted radiance — BRDF — incident radiance from some direction

Kajija "The Rendering Equation", SIGGRAPH 1986

The Rendering Equation



- <u>direct (local) illumination</u>:

  light source → surface → eye

- <u>indirect (global) illumination</u>:

  light source → surface→ … → surface → eye

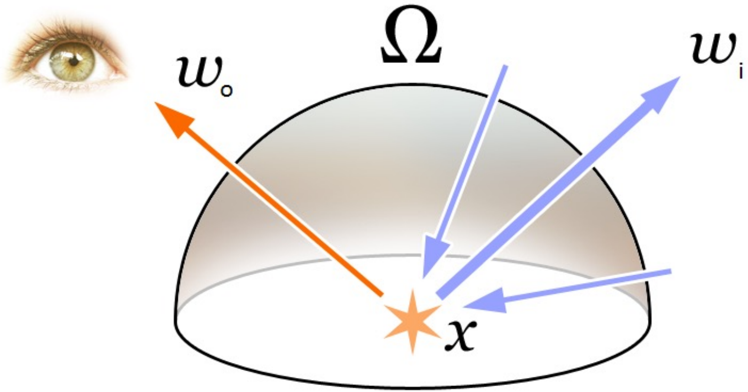
Direction towards viewer

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

radiance towards viewer · emitted radiance · BRDF · incident radiance from some direction

Kajija “The Rendering Equation”, SIGGRAPH 1986

The Rendering Equation



- <u>direct (local) illumination</u>:

  light source → surface → eye

- <u>indirect (global) illumination</u>:

  light source → surface→ … → surface → eye

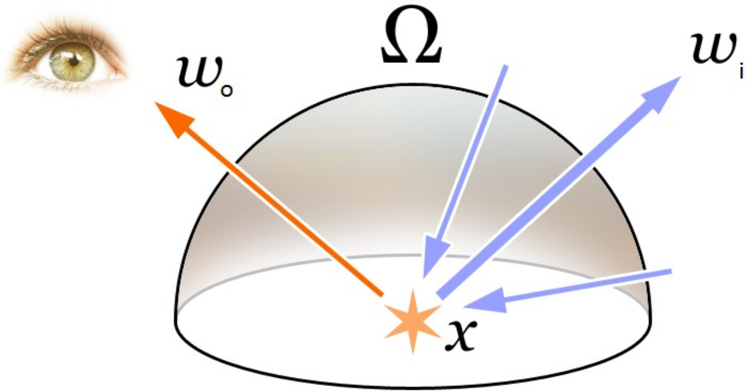
wavelength

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

radiance towards viewer | emitted radiance | BRDF | incident radiance from some direction

Kajija “The Rendering Equation”, SIGGRAPH 1986

The Rendering Equation



- <u>direct (local) illumination</u>:

  light source → surface → eye

- <u>indirect (global) illumination</u>:

  light source → surface→ … → surface → eye

time

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

radiance towards viewer | emitted radiance | BRDF | incident radiance from some direction

Kajija “The Rendering Equation”, SIGGRAPH 1986

The Rendering Equation

- drop time, wavelength (RGB) & global illumination to make it simple

- direct (local) illumination:
  light source → surface → eye

- ~~indirect (global) illumination:~~
  ~~light source → surface→ … → surface → eye~~

$$L_o(\mathbf{x},\,\omega_o,\,\lambda,\,t) \;=\; L_e(\mathbf{x},\,\omega_o,\,\lambda,\,t) + \int_\Omega f_r(\mathbf{x},\,\omega_i,\,\omega_o,\,\lambda,\,t)\,L_i(\mathbf{x},\,\omega_i,\,\lambda,\,t)\,(\omega_i\cdot\mathbf{n})\,\mathrm{d}\,\omega_i$$

Kajija "The Rendering Equation", SIGGRAPH 1986

The Rendering Equation

- drop time, wavelength (RGB), emission & global illumination to make it simple

$$L_0(x,\omega_0) = \sum_{k=1}^{num\_lights} f_r(x,\omega_k,\omega_o) L_i(x,\omega_k)(\omega_k \cdot n)$$

- direct (local) illumination:
  light source → surface → eye

- ~~indirect (global) illumination:
  light source → surface→ … → surface → eye~~

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)\, L_i(\mathbf{x}, \omega_i, \lambda, t)\, (\omega_i \cdot \mathbf{n})\, \mathrm{d}\,\omega_i$$

Kajija "The Rendering Equation", SIGGRAPH 1986

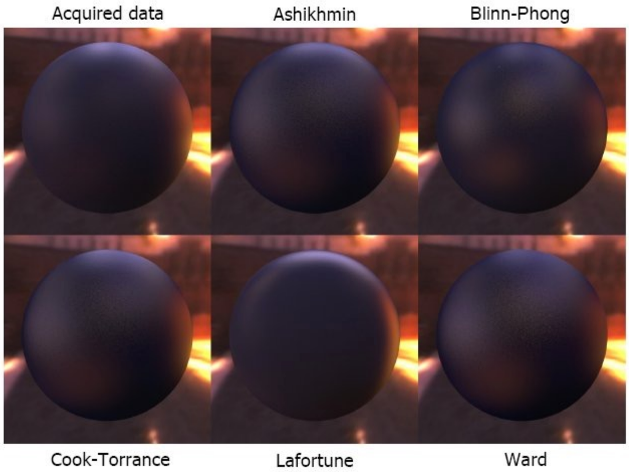
The Rendering Equation

- drop time, wavelength (RGB), emission & global illumination to make it simple

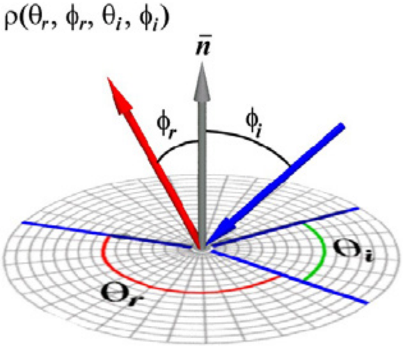
$$L_0(x,\omega_0) = \sum_{k=1}^{num\_lights} f_r\left(x,\omega_k,\omega_o\right) L_i(x,\omega_k)\left(\omega_k \cdot n\right)$$

Bidirectional Reflectance Distribution Function (BRDF)

- many different BRDF models exist: analytic, data driven (i.e. captured)



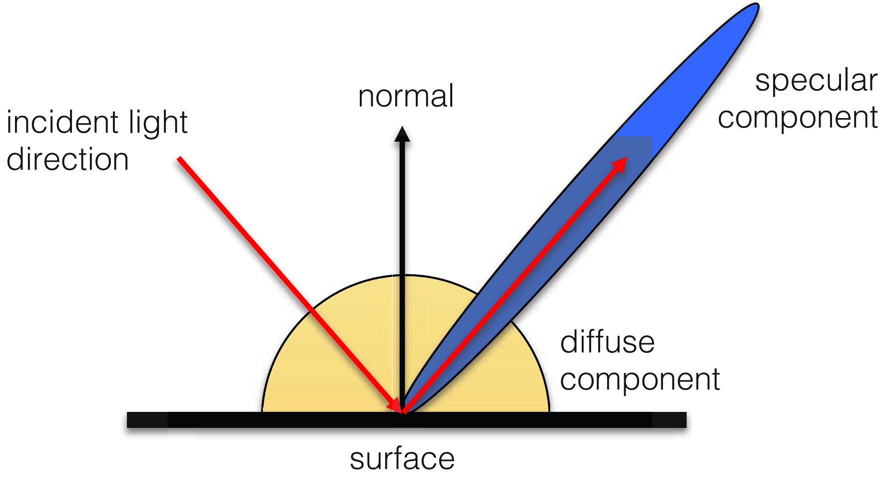
Ngan et al. 2004

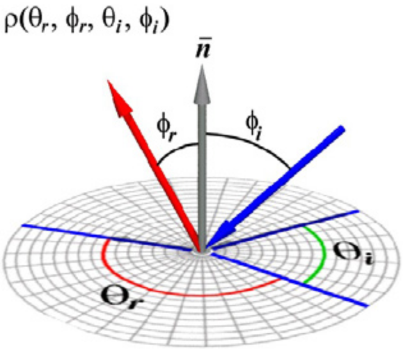


http://escience.anu.edu.au/lecture/cg/GlobalIllumination/BRDF.en.html

Bidirectional Reflectance Distribution Function (BRDF)

- can approximate BRDF with a few simple components

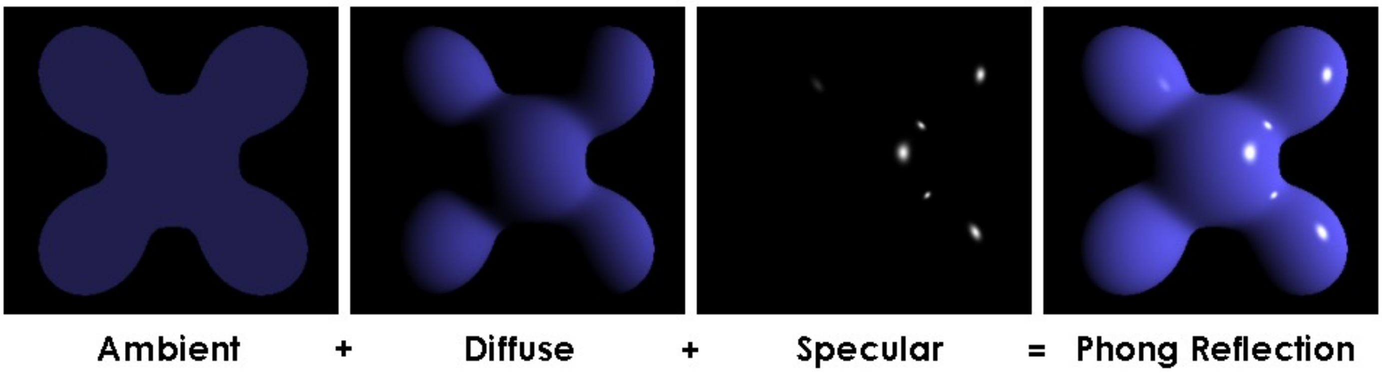




http://escience.anu.edu.au/lecture/cg/GlobalIllumination/BRDF.en.html

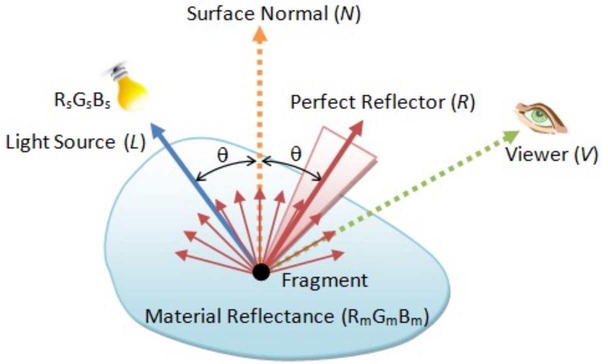
Phong Lighting

- emissive part can be added if desired
- calculate separately for each color channel: RGB



Phong Lighting

- simple model for direct lighting
- ambient, diffuse, and specular parts
- requires:
  - material color m_{RGB} (for each of ambient, diffuse, specular)
  - light color l_{RGB} (for each of ambient, diffuse, specular)



L normalized vector pointing towards light source

N normalized surface normal

V normalized vector pointing towards viewer

$$R = 2(N \bullet L)N - L$$

normalized reflection on surface normal

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Phong Lighting: Ambient

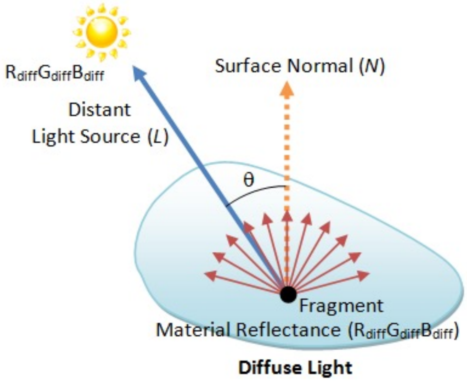
- independent of light/surface position, viewer, normal
- basically adds some background color

$$m_{\{R,G,B\}}^{ambient} \cdot l_{\{R,G,B\}}^{ambient}$$

Ambient

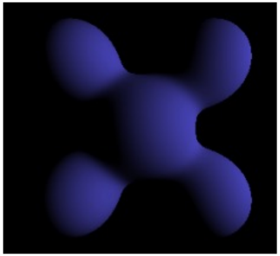
Phong Lighting: Diffuse

- needs normal and light source direction
- adds intensity cos-falloff with incident angle



$$m^{diffuse}_{\{R,G,B\}} \cdot l^{diffuse}_{\{R,G,B\}} \cdot \max(L \bullet N, 0)$$

dot product

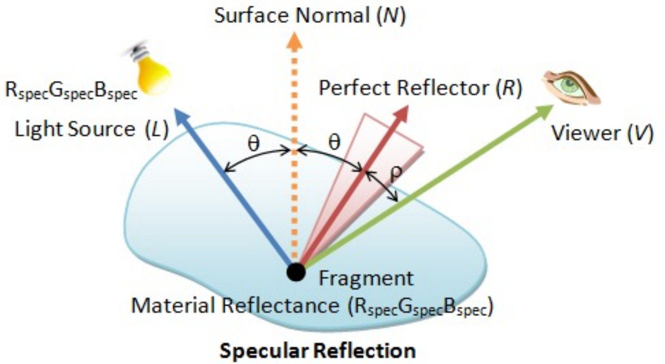


Diffuse

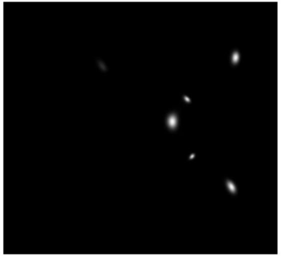
https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Phong Lighting: Specular

- needs normal, light & viewer direction
- models reflections = specular highlights
- shininess – exponent, larger for smaller highlights (more mirror-like surfaces)



$$m_{\{R,G,B\}}^{specular} \cdot l_{\{R,G,B\}}^{specular} \cdot \max(R \bullet V, 0)^{shininess}$$

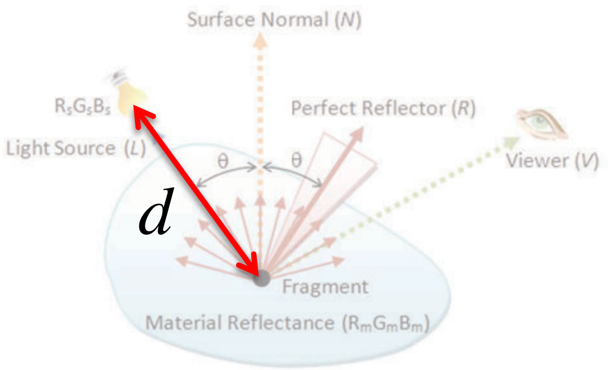


Specular

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Phong Lighting: Attenuation

- models the intensity falloff of light w.r.t. distance
- The greater the distance, the lower the intensity



$$\frac{1}{k_c + k_l d + k_q d^2}$$

constant linear quadratic attenuation

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Phong Lighting: Putting it all Together

- this is a simple, but efficient lighting model
- has been used by OpenGL for ~25 years
- absolutely NOT sufficient to generate photo-realistic renderings (take a computer graphics course for that)

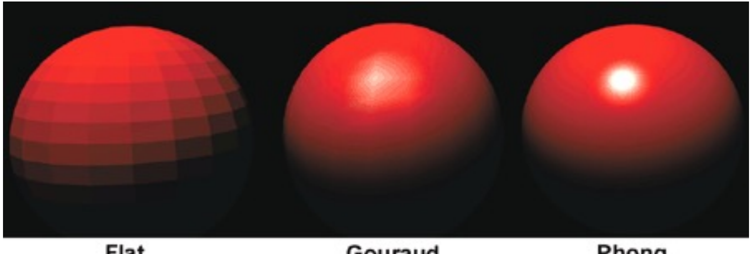
$$color_{\{R,G,B\}} = \underbrace{m^{ambient}_{\{R,G,B\}} \cdot l^{ambient}_{\{R,G,B\}}}_{\text{ambient}} + \sum_{i=1}^{num\_lights} \underbrace{\frac{1}{k_c + k_l d_i + k_q d_i^2}}_{\text{attenuation}} \left(\underbrace{m^{diffuse}_{\{R,G,B\}} \cdot l^{diffuse}_{i,\{R,G,B\}} \cdot \max(L_i \bullet N, 0)}_{\text{diffuse}} + \underbrace{m^{specular}_{\{R,G,B\}} \cdot l^{specular}_{i\{R,G,B\}} \cdot \max(R_i \bullet V, 0)^{shininess}}_{\text{specular}} \right)$$

Lighting Calculations

- *all lighting calculations happen in camera/view space!*
  - transform vertices and normals into camera/view space
  - calculate lighting, i.e. per color (i.e., given material properties, light source color & position, vertex position, normal direction, viewer position)

Lighting v Shading

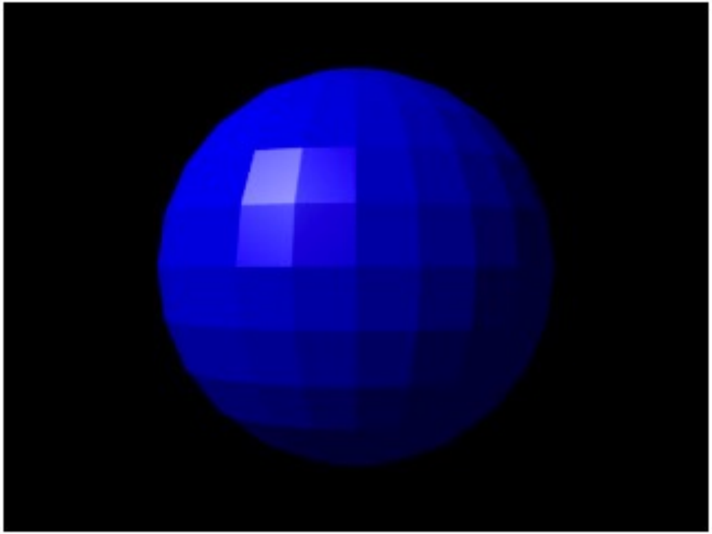
- lighting: interaction between light and surface (e.g. using Phong lighting model; think about this as "what formula is being used to calculate intensity/color")
- shading: how to compute color of each fragment (e.g. what attributes to interpolate and where to do the lighting calculation)
  1. Flat shading
  2. Gouraud shading (per-vertex lighting)
  3. Phong shading (per-fragment lighting) - different from Phong lighting



courtesy: Intergraph Computer Systems

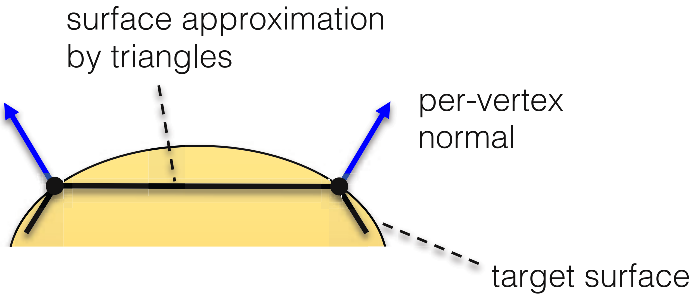
Flat Shading

- compute color only once per triangle (i.e. with Phong lighting)
- pro: usually fast to compute; con: creates a flat, unrealistic appearance
- we won't use it



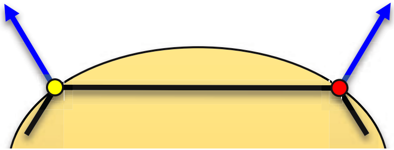
Gouraud or Per-vertex Shading

- compute color once per <u>vertex</u> (i.e. with Phong lighting)
- <u>interpolate per-vertex colors to all fragments within the triangles!</u>
- pro: usually fast-ish to compute; con: flat, unrealistic specular highlights



Gouraud Shading or Per-vertex Lighting

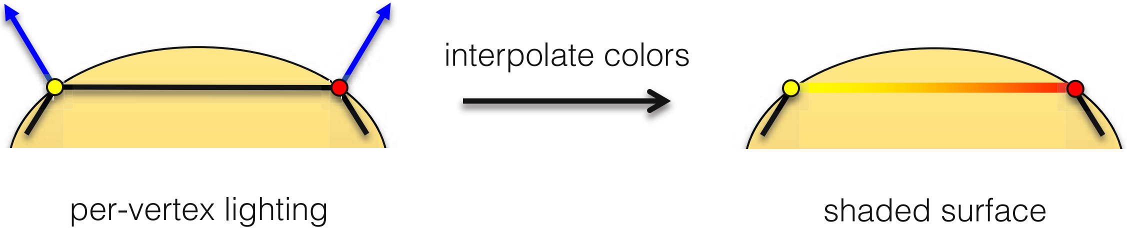
- compute color once per vertex (i.e. with Phong lighting)
- interpolate per-vertex colors to all fragments within the triangles!
- pro: usually fast-ish to compute; con: flat, unrealistic specular highlights



per-vertex lighting

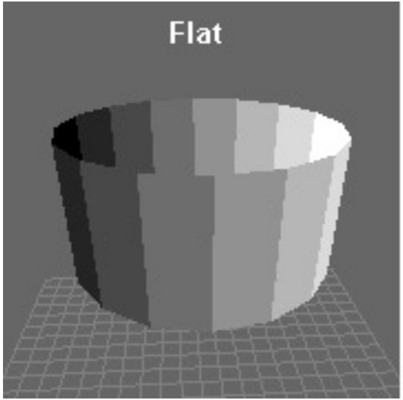
Gouraud Shading or Per-vertex Lighting

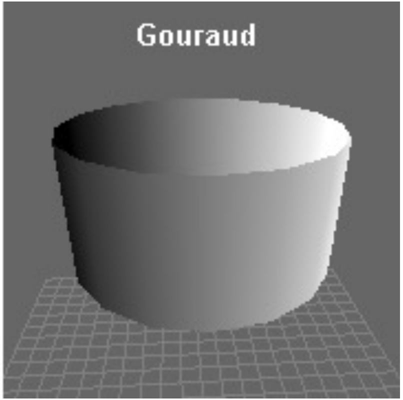
- compute color once per vertex (i.e. with Phong lighting)
- interpolate per-vertex colors to all fragments within the triangles!
- pro: usually fast-ish to compute; con: flat, unrealistic specular highlights

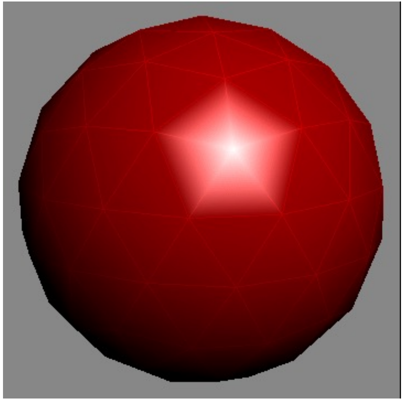


Gouraud Shading or Per-vertex Lighting

- compute color once per vertex (i.e. with Phong lighting)
- interpolate per-vertex colors to all fragments within the triangles!
- pro: usually fast-ish to compute; con: flat, unrealistic specular highlights



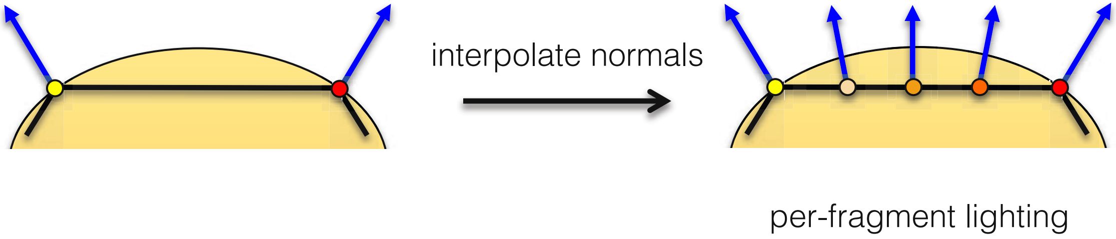




https://en.wikipedia.org/wiki/Gouraud_shading

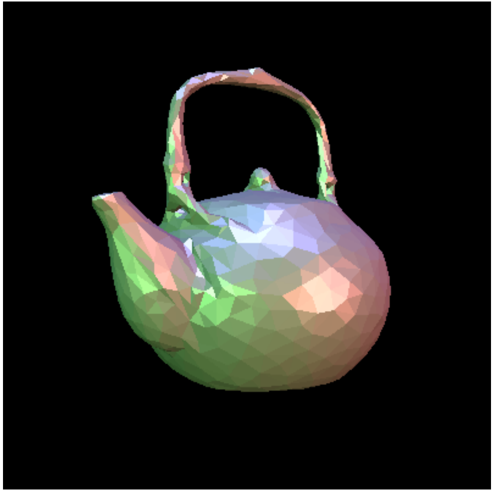
Phong Shading or Per-fragment Lighting

- compute color once per fragment (i.e. with Phong lighting)
- need to interpolate per-vertex normals to all fragments to do the lighting calculation!
- pro: better appearance of specular highlights; con: usually slower to compute

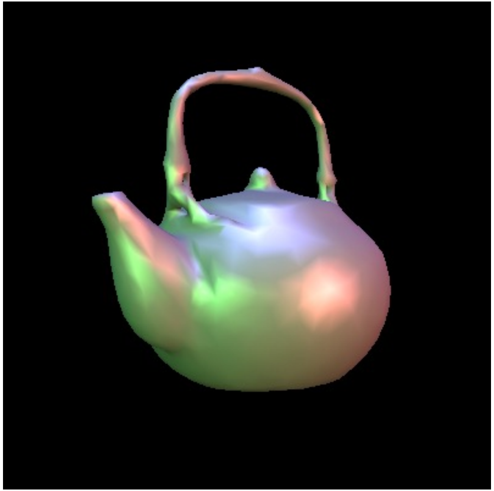


Shading

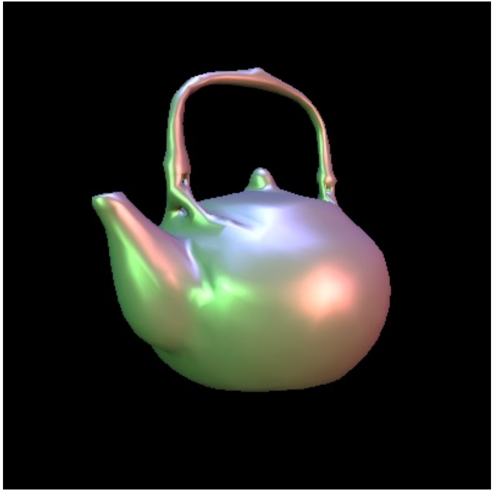
Flat Shading



Gouraud Shading

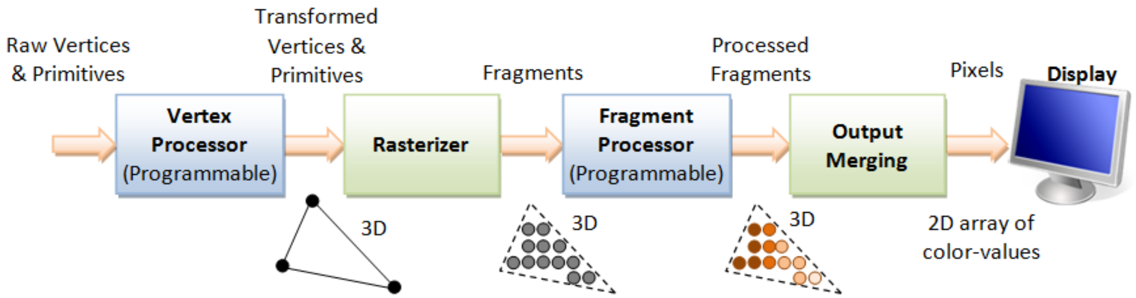


Phong Shading



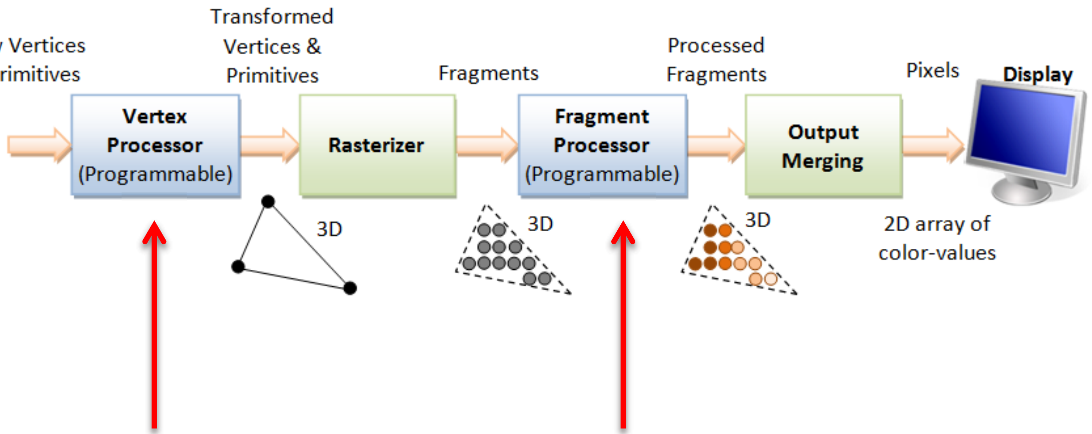
http://www.decew.net/OSS/timeline.php

Back to the Graphics Pipeline



https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Per-vertex Lighting v Per-fragment Lighting



<u>vertex shader</u>

- lighting calculations done for each vertex

<u>fragment shader</u>

- lighting calculations done for each fragment

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Vertex and Fragment Shaders

- shaders are small programs that are executed in parallel on the GPU for each vertex (vertex shader) or each fragment (fragment shader)

- vertex shader (*before rasterizer*):
  - modelview projection transform of vertex & normal (see last lecture)
  - if per-vertex lighting: do lighting calculations here (otherwise omit)

- fragment shader (*after rasterizer*):
  - assign final color to each fragment
  - if per-fragment lighting: do all lighting calculations here (otherwise omit)

Fragment Processing

- lighting and shading (per-fragment) – same calculations as per-vertex shading, but executed for each fragment
- texture mapping

these also happen, but don't worry about them (we wont touch these):

- fog calculations
- alpha blending
- hidden surface removal (using depth buffer)
- scissor test, stencil test, dithering, bitmasking, …

Depth Test

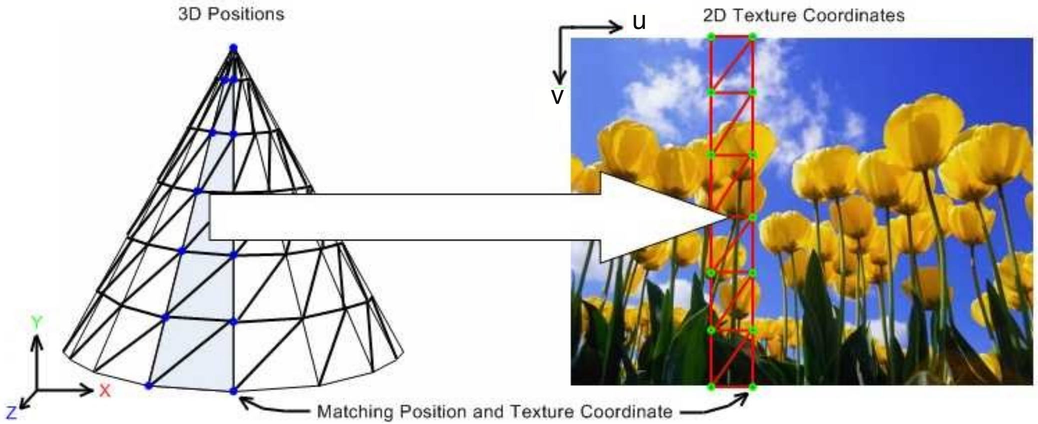
- oftentimes we have multiple triangles behind each other, the depth test determines which one to keep and which one to discard
- if depth of fragment is smaller than current value in depth buffer → overwrite color and depth value using current fragment; otherwise discard fragment

color buffer

depth buffer

Texture Mapping

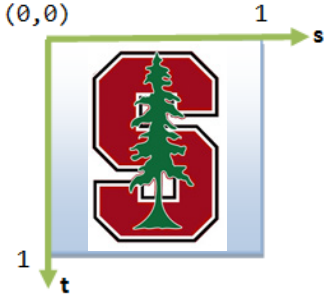
- texture = 2D image (e.g. RGBA)
- we want to use it as a “sticker” on our 3D surfaces
- mapping from vertex to position on texture (<u>texture coordinates</u> u,v)



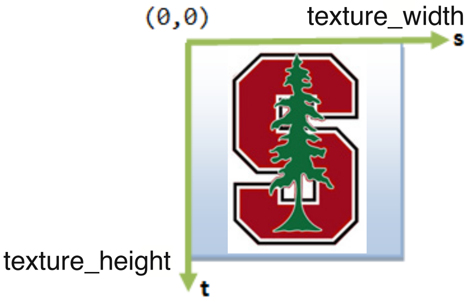
https://blogs.msdn.microsoft.com/danlehen/2005/11/06/3d-for-the-rest-of-us-texture-coordinates/ (sorry, this website seems to be discontinued)

Texture Mapping

- texture = 2D image (e.g. RGBA)
- we want to use it as a "sticker" on our 3D surfaces
- mapping from vertex to position on texture (<u>texture coordinates</u> u,v)



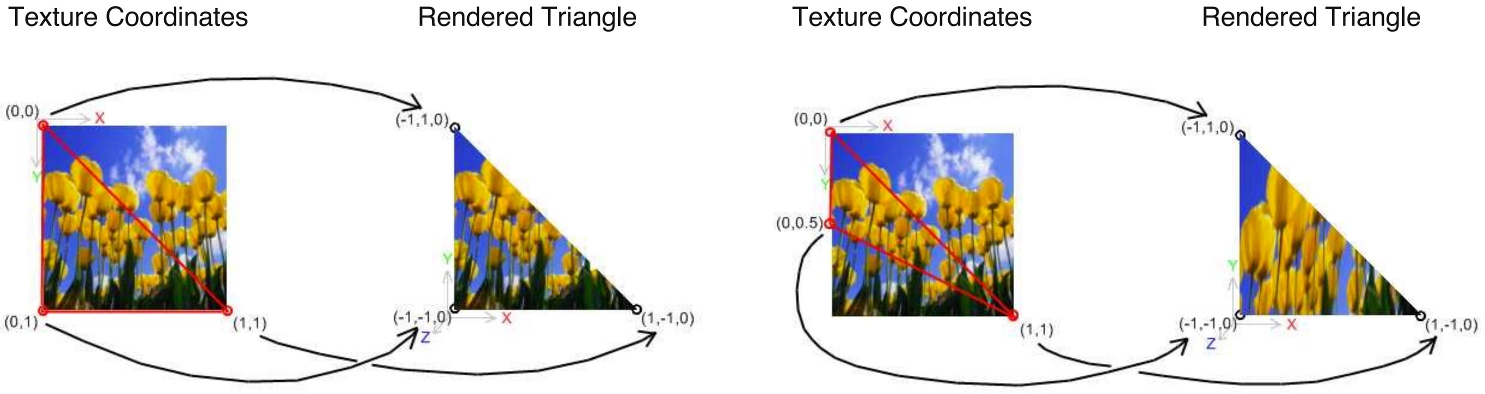
Normalized Texture Coordinates



Non-normalized Texture Coordinates

Texture Mapping

- same texture, different texture coordinates

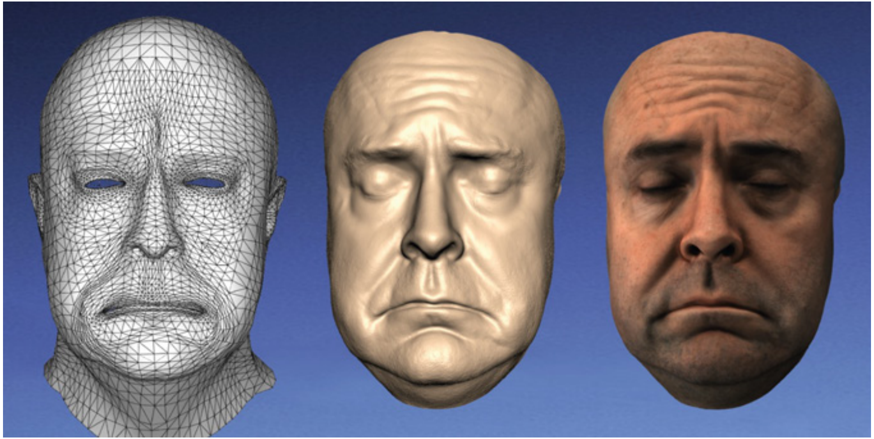


https://blogs.msdn.microsoft.com/danlehen/2005/11/06/3d-for-the-rest-of-us-texture-coordinates/ (sorry, this website seems to be discontinued)

Texture Mapping

- texture mapping faces

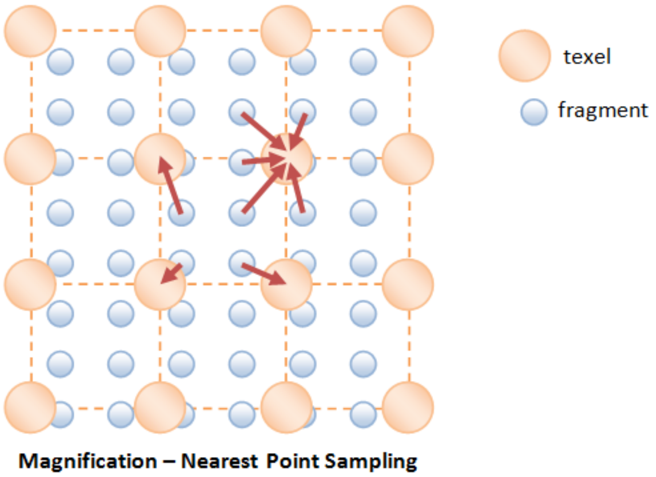
turbosquid.com

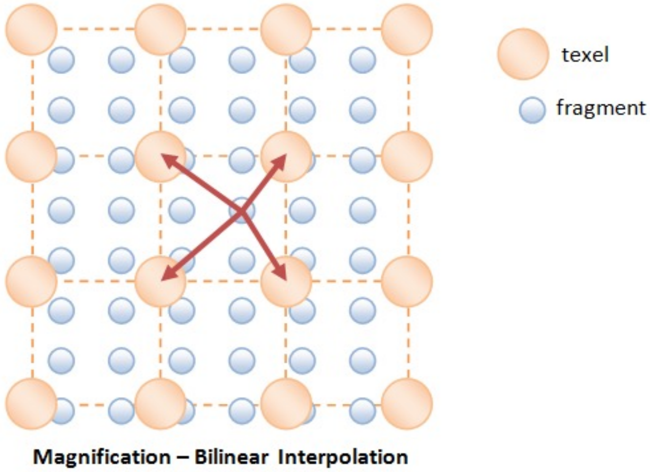


Bermano et al. 2013

Texture Mapping

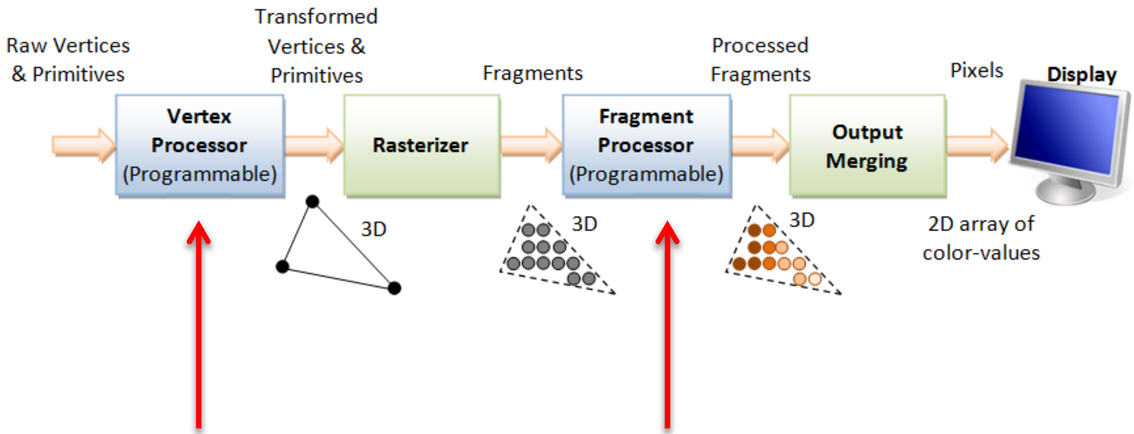
- texture filtering: fragments don't align with texture pixels (texels) → interpolate





https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Next Lecture: Vertex & Fragment Shaders, GLSL



<u>vertex shader</u>

- transforms & <u>(per-vertex) lighting</u>

<u>fragment shader</u>

- <u>texturing</u>
- <u>(per-fragment) lighting</u>

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

Summary

- rasterization
- the rendering equation, BRDFs
- lighting: computer interaction between vertex/fragment and lights
    - Phong lighting
- shading: how to assign color (i.e. based on lighting) to each fragment
    - Flat, Gouraud, Phong shading
- vertex and fragment shaders
- texture mapping

Further Reading

- good overview of OpenGL (deprecated version) and graphics pipeline (missing a few things) :

  https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

- textbook: Shirley and Marschner “Fundamentals of Computer Graphics”, AK Peters, 2009

- definite reference: “OpenGL Programming Guide” aka “OpenGL Red Book”

- WebGL / three.js tutorials: https://threejs.org/