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

A primitive is formed by one or more vertices. Vertices are not aligned to the pixel-grid.

A fragment is aligned to the pixel-grid with a depth.

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
output: 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

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

- **indirect (global) illumination:**
  light source → surface... → surface → eye

\[
L_0(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) \, d\omega_i
\]
The Rendering Equation

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

- **indirect (global) illumination:**
  light source → surface → ... → surface → eye

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

- **radiance towards viewer**
- **emitted radiance**
- **BRDF**
- **incident radiance from some direction**
The Rendering Equation

- **direct (local) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ eye

- **indirect (global) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ … $\rightarrow$ surface $\rightarrow$ eye

\[
L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t) (\omega_i \cdot n) \, d\omega_i
\]
The Rendering Equation

- **direct (local) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ eye

- **indirect (global) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ ... $\rightarrow$ surface $\rightarrow$ eye

\[
L_{o}(x, \omega_{o}, \lambda, t) = L_{e}(x, \omega_{o}, \lambda, t) + \int_{\Omega} f_{r}(x, \omega_{i}, \omega_{o}, \lambda, t) L_{i}(x, \omega_{i}, \lambda, t) (\omega_{i} \cdot n) \, d\omega_{i}
\]

Radiance towards viewer $\rightarrow$ emitted radiance $\rightarrow$ BRDF $\rightarrow$ incident radiance from some direction

Kajija “The Rendering Equation”, SIGGRAPH 1986
The Rendering Equation

- **direct (local) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ eye

- **indirect (global) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ ... $\rightarrow$ surface $\rightarrow$ eye

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

wavelength

radiance towards viewer

emitted radiance

BRDF

incident radiance from some direction

Kajija “The Rendering Equation”, SIGGRAPH 1986
The Rendering Equation

- **direct (local) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ eye

- **indirect (global) illumination:**
  light source $\rightarrow$ surface $\rightarrow$ ... $\rightarrow$ surface $\rightarrow$ eye

\[
\begin{align*}
L_o(x, \omega_o, \lambda, t) &= L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t) (\omega_i \cdot n) \, d\omega_i
\end{align*}
\]

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

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

Kajija “The Rendering Equation”, SIGGRAPH 1986
The Rendering Equation

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

\[
L_0(x, \omega_0) = \sum_{k=1}^{\text{num\_lights}} f_r(x, \omega_k, \omega_o) L_t(x, \omega_k)(\omega_k \cdot n)
\]

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

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

\[
L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t) L_t(x, \omega_i, \lambda, t)(\omega_i \cdot n) \, d\omega_i
\]

Kajija “The Rendering Equation”, SIGGRAPH 1986
The Rendering Equation

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

\[ L_0(x, \omega_0) = \sum_{k=1}^{num\_lights} f_r(x, \omega_k, \omega_o) L_i(x, \omega_k)(\omega_k \cdot n) \]
Bidirectional Reflectance Distribution Function (BRDF)

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

\[ \rho(\theta_r, \phi_r, \theta_i, \phi_i) \]

Ngan et al. 2004

Acquired data

Ashikhmin

Blinn-Phong

Cook-Torrance

Lafortune

Ward

Bidirectional Reflectance Distribution Function (BRDF)

- can approximate BRDF with a few simple components

\[ \rho(\theta_r, \phi_r, \theta_i, \phi_i) \]

- incident light direction
- normal
- diffuse component
- specular component
- surface
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

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

\[ m_{ambient} \cdot l_{ambient} \]
Phong Lighting: Diffuse

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

\[ m_{\text{diffuse}}^{\{R,G,B\}} \cdot l_{\text{diffuse}}^{\{R,G,B\}} \cdot \max(L \cdot N, 0) \]

dot product
Phong Lighting: Specular

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

\[ m_{\text{specular}} \cdot l_{\text{specular}} \cdot \max(R \cdot V, 0)^{\text{shininess}} \]
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

constant

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)

\[
\text{color}_{\{R,G,B\}} = m_{\{R,G,B\}}^{\text{ambient}} \cdot l_{\{R,G,B\}}^{\text{ambient}} + \sum_{i=1}^{\text{num lights}} \frac{1}{k_c + k_id_i + k_qd_i^2} \left( m_{\{R,G,B\}}^{\text{diffuse}} \cdot l_{\{R,G,B\}}^{\text{diffuse}} \cdot \max(L_i \cdot N, 0) + m_{\{R,G,B\}}^{\text{specular}} \cdot l_{\{R,G,B\}}^{\text{specular}} \cdot \max(R_i \cdot V, 0)^{\text{shininess}} \right)
\]

ambient \hspace{2cm} attenuation \hspace{2cm} diffuse \hspace{2cm} specular
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
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 **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

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

per-vertex lighting

interpolate colors

shaded surface
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

Vertex Processor (Programmable) → Rasterizer → Fragment Processor (Programmable) → Output Merging → Pixels

Raw Vertices & Primitives → Transformed Vertices & Primitives → Fragments → Processed Fragments

3D → 2D array of color-values

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Per-vertex Lighting v Per-fragment Lighting

- **vertex shader**
  - lighting calculations done for each vertex

- **fragment shader**
  - 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 won’t 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
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 (texture coordinates $u,v$)

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 (texture coordinates u,v)
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
Texture Mapping

• texture filtering: fragments don’t align with texture pixels (texels) $\rightarrow$ interpolate

https://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
Next Lecture: Vertex & Fragment Shaders, GLSL

vertex shader
- transforms & (per-vertex) lighting

fragment shader
- texturing
- (per-fragment) lighting

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


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