Computer Graphics

COMP3421/9415
2021 Term 3 Lecture 11

What did we learn before the break?

Games and Art

- Game Design and its relationship with Graphics
- The Art Pipeline
- Details of Modelling and Animation

What are we starting today?

Lighting

- What is light?
- Real World vs Simulation
- Starting on Lighting in Polygon Rendering

Light

What is light?

And why is it important for vision?

- Physics of real world light
- A particle or a wave (or both)
- A spectrum of electromagnetic radiation
- Travels in (mostly) straight lines
- Is effectively instantaneous
- Reflects off many surfaces in different ways

Image credit: Bungie Studios

Light and Colour

Colour is us seeing different wavelengths of light

- A mix of multiple wavelengths
- Human vision is detecting reflected light
- Different surfaces absorb and reflect different wavelengths

- . . . and in different directions

- Our eyes perceive colour based on which wavelengths reach them

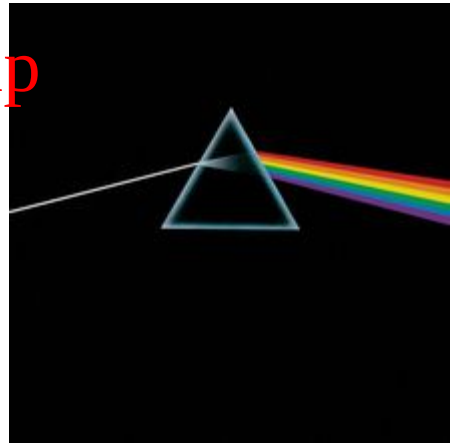


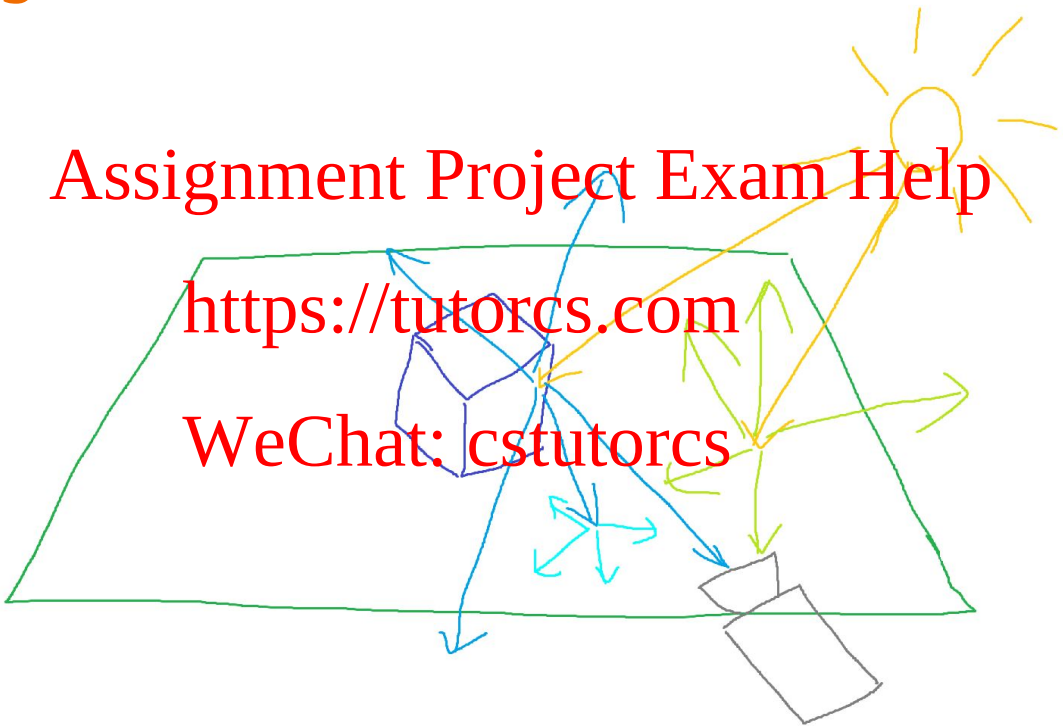
Image credit: Pink Floyd

Simulate Light

A physics based model for virtual lighting

- Send rays of light out from our light sources
  - Could need billions of these to match the number of photons in real light
- Bounce them off objects
  - Every bounce reflects the colour of the object
- If a reflected ray passes through the near plane of the frustum
- . . . and hits the camera
- Then colour the pixel with whatever colour the ray was

Simulate Light



Pros and Cons

This technique seems to make sense

- Accurate simulation of real light
- Nothing looks too complicated mathematically
- Realistic shadows and reflections

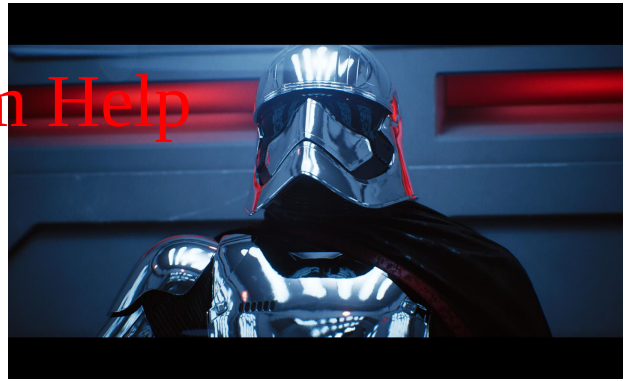
Wait, did you say Billions?

- Computationally very expensive
- Collision Detection is very expensive
- Most calculation is wasted

Ray Tracing

This technique has a name!

- A very big buzz word in present day Graphics
- Commercially available from 2018
- Before that, thought infeasible for realtime (still incredibly resource hungry)
- Used in film from the late 1990s



Real Time Ray Tracing
Image credit: Unreal Engine

Phong Lighting

Polygon Rendering and Light

We don't have a physics model

- We can't trace rays of light easily
- What are our tools?
- Vertices, and Triangles, Coordinates, Camera
- Vector and Matrix math
- Can we build an approximate lighting model?
- That computes a lot faster?

Phong Lighting

A Simple Idea for Polygon Rendering Lighting

- Compute Lighting per fragment
- Simplify the idea of bouncing light
- Three main parts:
- Ambient Light
- Diffuse Light
- Specular Light

Ambient Light

Indirect Light

- Even outside of direct light, there's some light
- Reflections off other surfaces
- Correct calculation takes a lot of light bounces
- . . . and doesn't usually have much effect!

In Phong Lighting

- Ambient light is constant throughout the scene

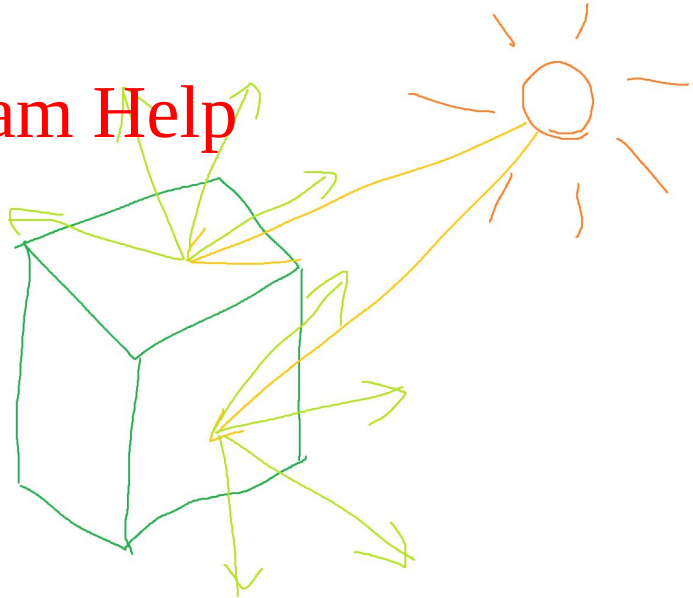
Diffuse Light

Directional Light on non-shiny surfaces

- Light directly from a source
- Hits a surface and scatters
- Matte or rough surface objects

In Phong Lighting

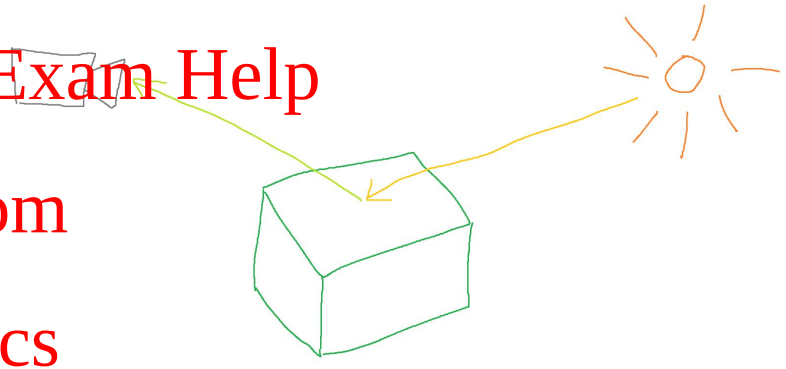
- Calculated based on light source
- and angle to surface



Specular Light

Direct Reflections from shiny surfaces

- Light from a source
- Hits a surface and bounces directly
- Shiny or reflective objects



In Phong Lighting

- Calculated based on direction to camera as well as direction to light source

Light Calculation per Fragment

$$I_\mathrm{p} = k_\mathrm{a} i_\mathrm{a} + \sum_{m \in \text{ lights}} (k_\mathrm{d} (\hat{L}_m \cdot \hat{N}) i_{m,\mathrm{d}} + k_\mathrm{s} (\hat{R}_m \cdot \hat{V})^\alpha i_{m,\mathrm{s}}).$$

Image credit: Wikipedia

Equation for the colour of a fragment

- Ambient: ambient light colour * surface colour
- +
- Diffuse: light colour * angle of light to surface * surface colour
- +
- Specular: light colour * angle of reflected light compared to viewing angle * surface colour
- =
- Total colour in the fragment

Break Time

Cutting Corners

- Graphics is a field full of dirty hacks for optimisation
- We know that Ray Tracing is accurate but expensive
- So we invent tricks like Phong Lighting

- Polygon Rendering is itself a trick to reduce computation

Graphics is all a compromise between visual quality and speed

Lights and Materials

Where does light come from?

In the real world

- The Sun
- Lamps and light bulbs

In our virtual scenes

- Directional Lights (simulates the sun)
- Point Lights (have a position in the scene)
- And others . . .

Representation of Light sources

Directional Light

- Represented by a vector (a direction!)
- Considered to be so far away that its direction is the same everywhere in the scene

Point Light(s)

- Represented by a point (a position in the scene!)
- Direction to fragments will need to be calculated

Materials

We've been calling these textures

- But there's so much more!
- Materials are surface information
- Colour is one part of the surface information
- But there are other things:
  - How shiny is it?
  - Does it have ripples or bumps in it?
- An object can have multiple materials!

Ambient Lighting

Ambient Lighting Equation

Calculated per scene per fragment

- $I_a = i_a * k_a$
- I_a Final intensity of ambient light
- i_a Intensity of ambient light in the scene
- k_a Ambient reflectivity of fragment

Eg: Ambient Light is a bit reddish (0.1, 0.05, 0.05), the object's material reflects bluish in ambient light (0.2, 0.9. 0.2)

Final ambient Light: (0.1, 0.05, 0.05) * (0.2, 0.9. 0.2) = (0.02, 0.045, 0.01)

Ambient Lighting Result

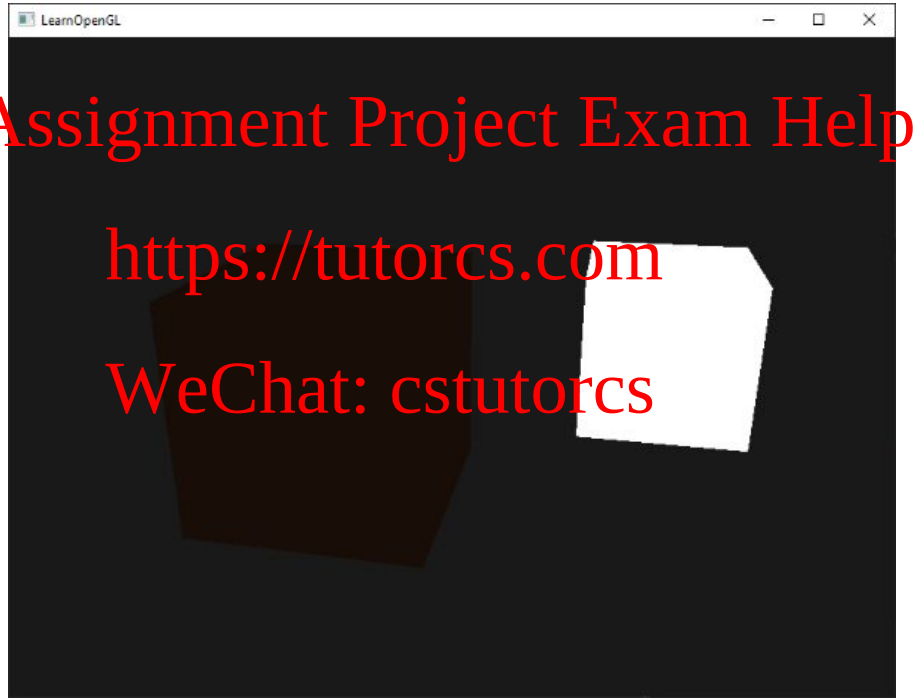


Image credit: learnopengl.com

Normals

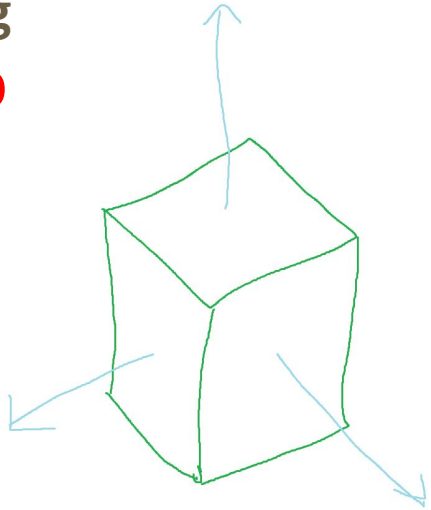
Normals - Directions of surfaces

Diffuse Lighting needs to know where a surface is facing

- Normal: a vector perpendicular to the surface
- Shows the "facing" direction

Attaching Normals to polygons

- Where can we store information in a polygon?
- Triangles have no data storage!
- It has to be in the vertices!



Normals attached to Vertices

Normals, Vertices and Triangles

- Normal data in vertices makes sense
- Frag shaders can already use vertex data
- Normals can be generated using triangle data
- Specific normals can be stored in the vertex attributes

Making Normals

Generating Normals from Triangles

- Cross Product of triangle edges
- Triangle winding order is now important!
- If we create vectors between vertices

- 1->3 x 1->2 is different from 1->2 x 1->3

- Counter-clockwise is the convention for the "front" of a polygon

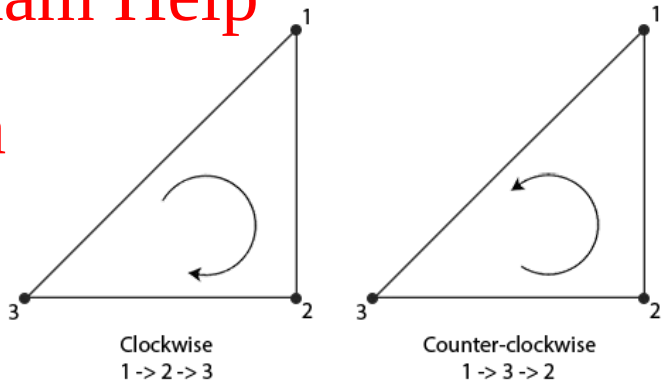


Image credit: learnopengl.com

Normalise Normals

Directions shouldn't have magnitude

- Vectors can have a direction and magnitude
- If we're using them purely as a direction . . .
- They should be length 1

- Lighting calculations will rely on all directions being length 1 vectors
- When in doubt, normalise!

Diffuse Lighting

Diffuse Lighting Equation

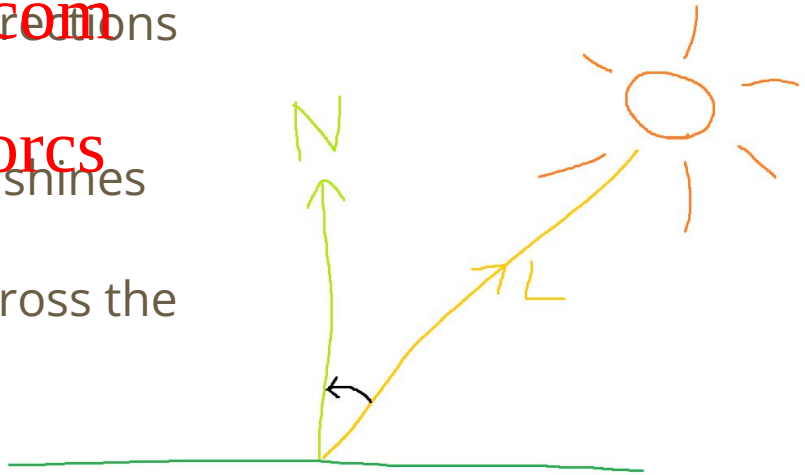
Calculated per light per fragment

- $I_d = k_d * (L.N) * i_d$
- I_d Final intensity of diffuse light
- k_d Diffuse reflectivity of fragment
- L Direction to light from fragment
- N Surface Normal
- i_d Diffuse intensity of light source

Breaking down the Diffuse Equation

What's the L.N bit?

- A dot product of two direction vectors
- dot product results in a single number
- Scale of the difference in the vectors directions
  - 90°, dot product = 0
  - 0°, dot product = 1
- The light is the brightest when the light shines directly at the surface
- There is no light if the light is shining across the surface



Diffuse Lighting

The Equation Explained

- Take the intensity of the light
- and the diffuse reflectivity of the surface (surface colour)
- Multiply them (just like ambient lighting)
- Then multiply that colour by 0.0-1.0
- This number is a representation of how directly the surface normal is aiming at the light source
- We find that out by using dot product of the two normalised vectors

Diffuse Lighting Result

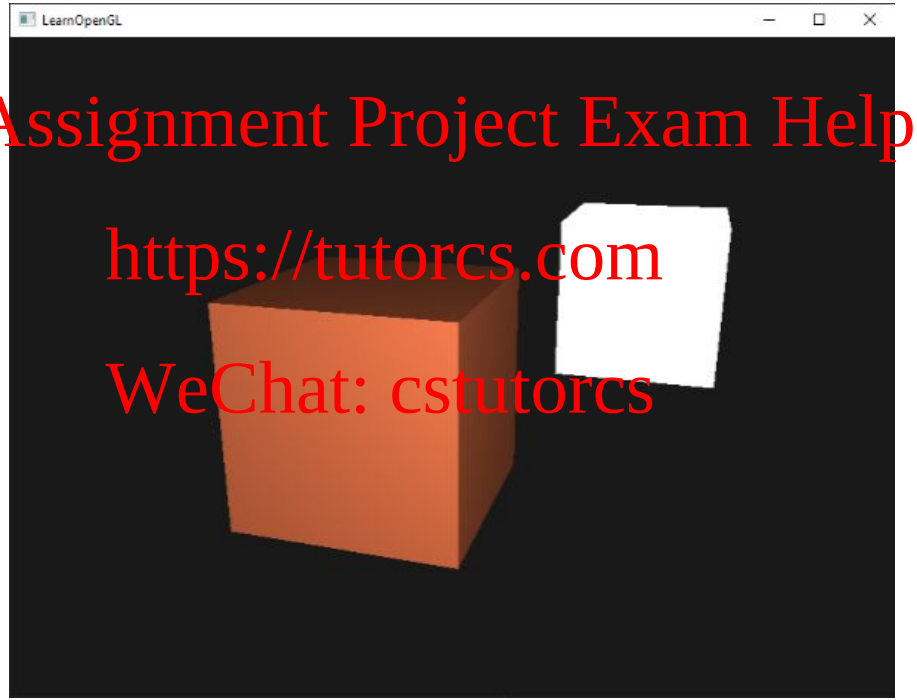


Image credit: learnopengl.com

What did we learn today?

Lighting

- The difference between real and virtual
- Possibilities for simulation of light
- Phong Lighting, an approximation of light
- Beginning to look in detail at the lighting algorithm