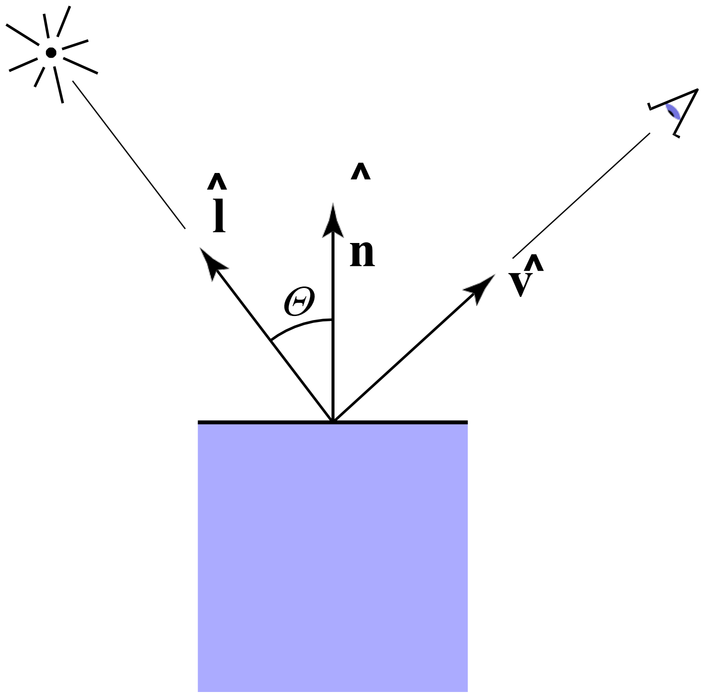
Shading II
(Shading, Pipeline and Texture Mapping)

Dr. Zhigang Deng

UNIVERSITY of HOUSTON DEPARTMENT OF COMPUTER SCIENCE

Last Lecture

- Shading 1
  - Blinn-Phong reflectance model
    - Diffuse
    - Specular
    - Ambient
  - At a **specific shading point**

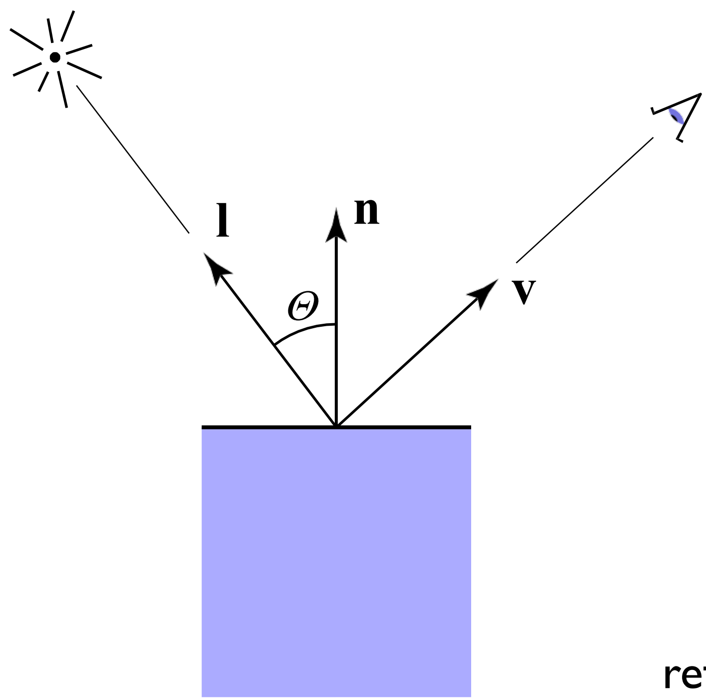


Today

- Shading 2
  - Blinn-Phong reflectance model
    - Specular and ambient terms
  - Shading frequencies
  - Graphics pipeline Texture
  - mapping Barycentric
  - coordinates

Recap: Lambertian (Diffuse) Term

Shading independent of view direction



energy arrived
at the shading point

$$L_d = k_d\,(I/r^2)\,\max(0, \mathbf{n}\cdot\mathbf{l})$$

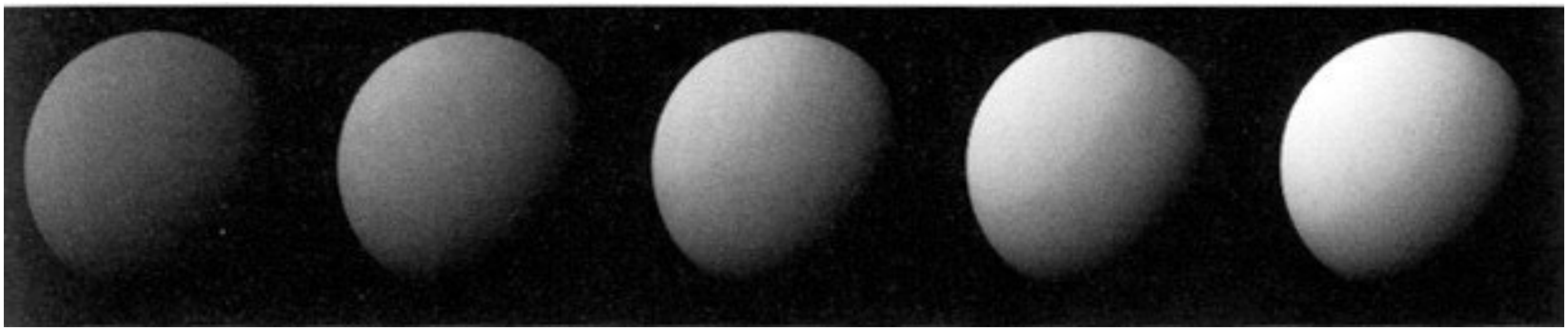
diffusely
reflected light

diffuse
coefficient
(color)

energy received
by the shading point

Recap: Lambertian (Diffuse) Term

Produces diffuse appearance



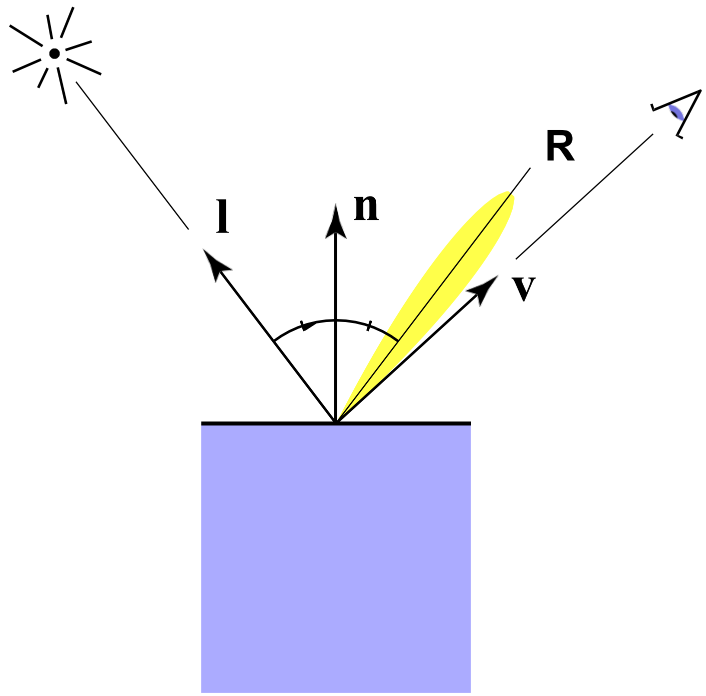
$k_d \longrightarrow$

[Foley et al.]

Specular Term (Blinn-Phong)

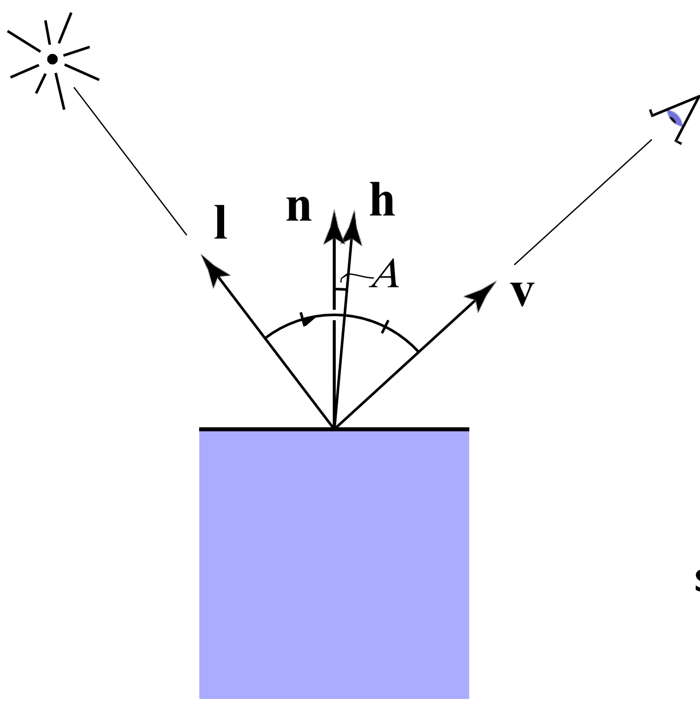
Intensity depends on view direction

- Bright near mirror reflection direction



Specular Term (Blinn-Phong)

- V close to mirror direction ⇔ **half vector** near normal
  - Measure "near" by dot product of unit vectors



- $\mathrm{h} = \mathrm{bisector}(\mathrm{v}, \mathrm{l})$

$$= \frac{\mathrm{v} + \mathrm{l}}{|\mathrm{v} + \mathrm{l}|}$$

$$L_s = k_s (I/r^2) \max(0, \cos \boldsymbol{\alpha})^p$$

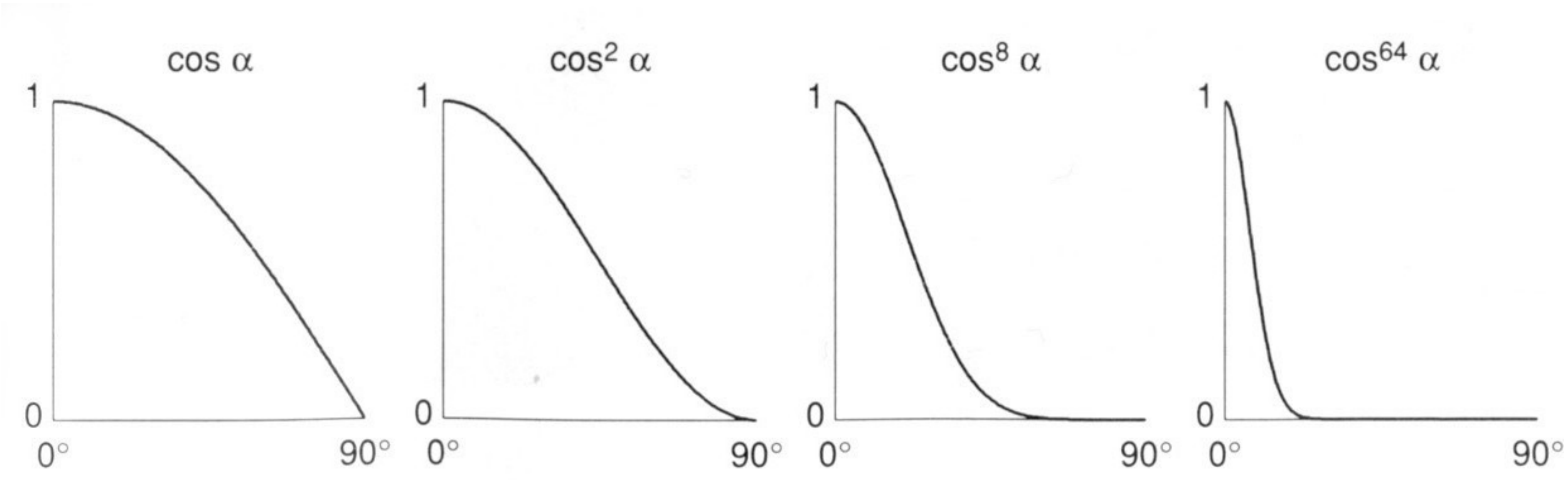
$$= k_s (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p$$

specularly reflected light

specular coefficient

Cosine Power Plots

Increasing p narrows the reflection lobe



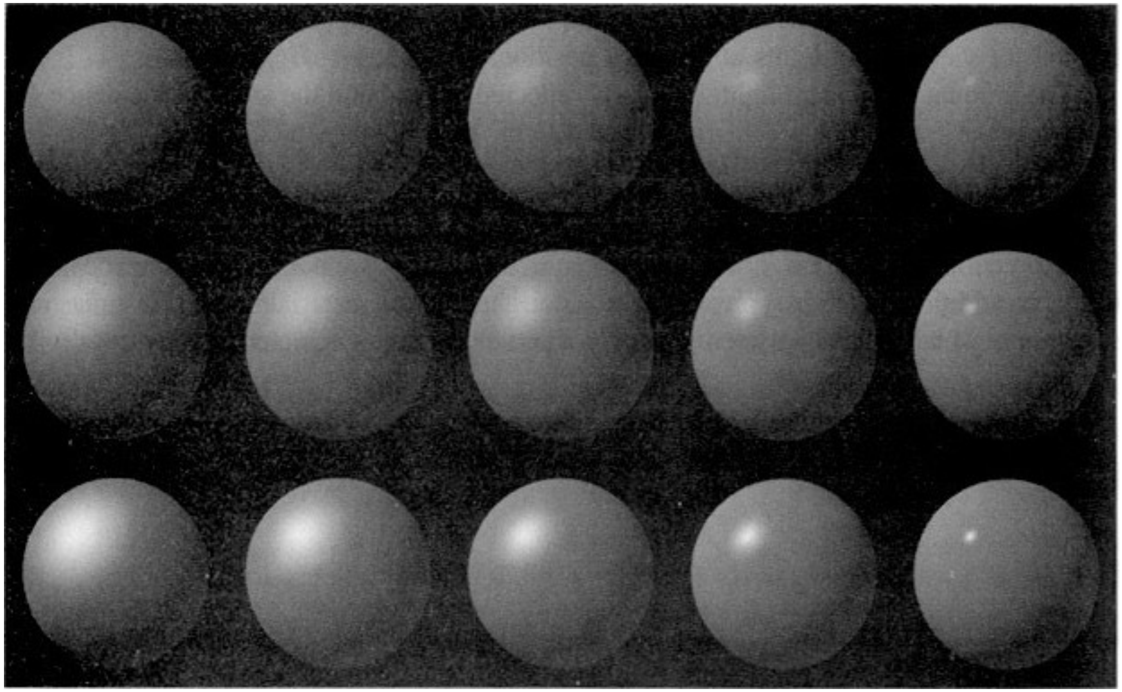
[Foley et al.]

Specular Term (Blinn-Phong)

Blinn-Phong

$$L_s = k_s\,(I/r^2)\,\max(0, \mathbf{n}\cdot\mathbf{h})^p$$

k_s



p

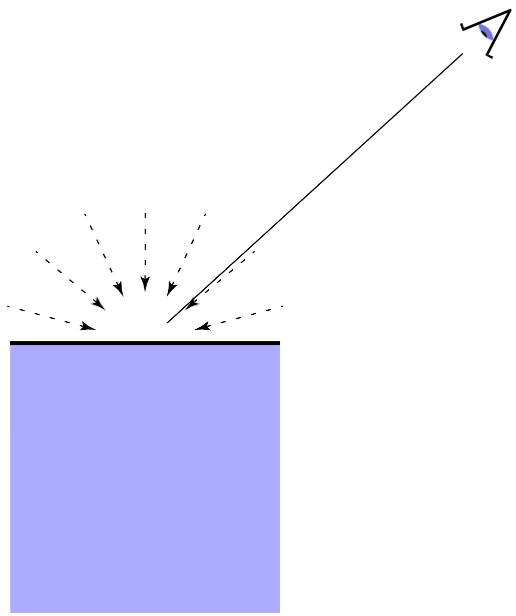
Note: showing
Ld + Ls together

[Foley et al.]

Ambient Term

Shading that does not depend on anything

- Add constant color to account for disregarded illumination and fill in black shadows
- This is approximate / fake!

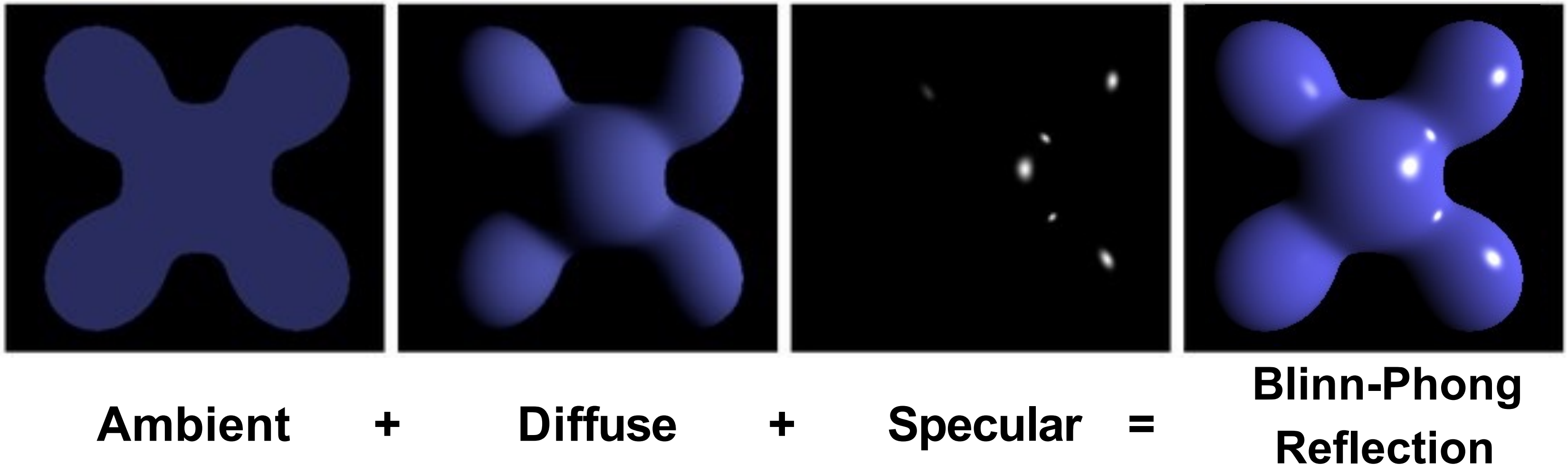


$$L_a = k_a I_a$$

Ambient coefficient

reflected ambient light

Blinn-Phong Reflection Model



$L = L_a + L_d + L_s$

$= k_a I_a + k_d (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p$

Shading Frequencies

Shading Frequencies

What caused the shading difference?

Shade each triangle (flat shading)

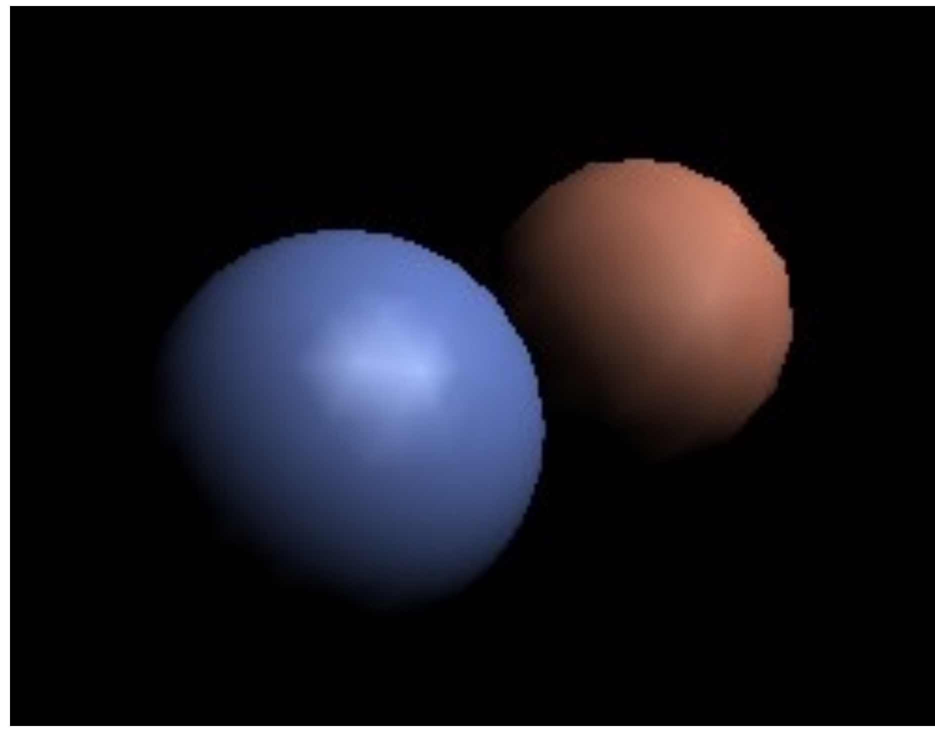
Flat shading

- Triangle face is flat
  - — one normal vector
- Not good for smooth surfaces

Shade each vertex (Gouraud shading)

Gouraud shading

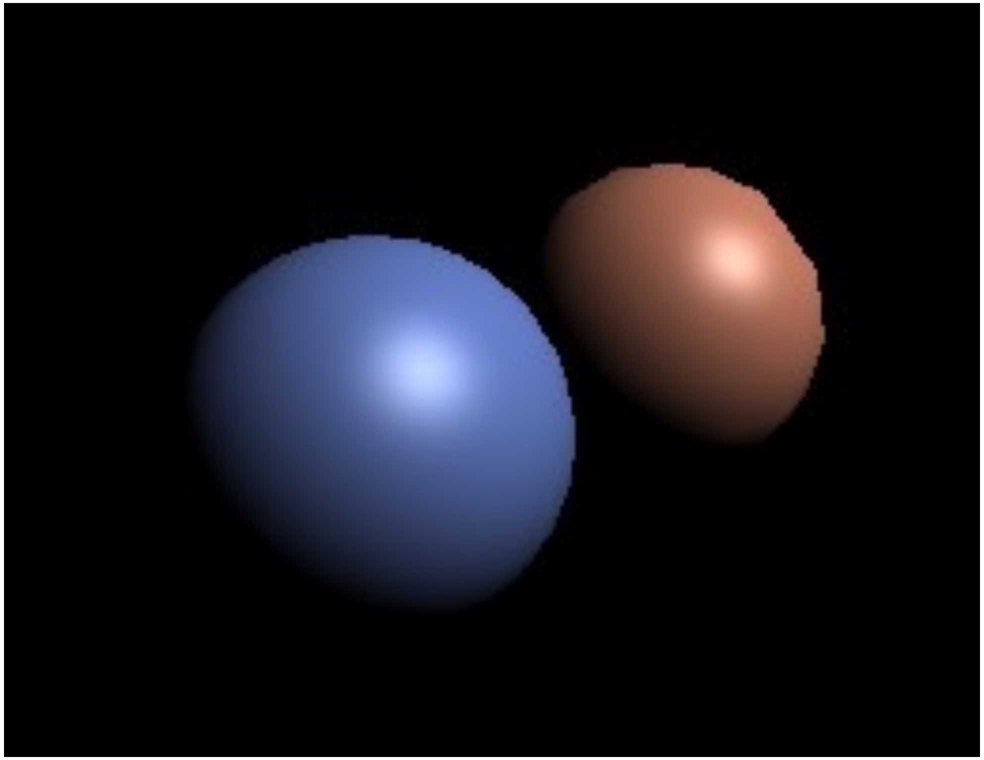
- Interpolate colors from vertices across triangle
- Each vertex has a normal vector (how?)



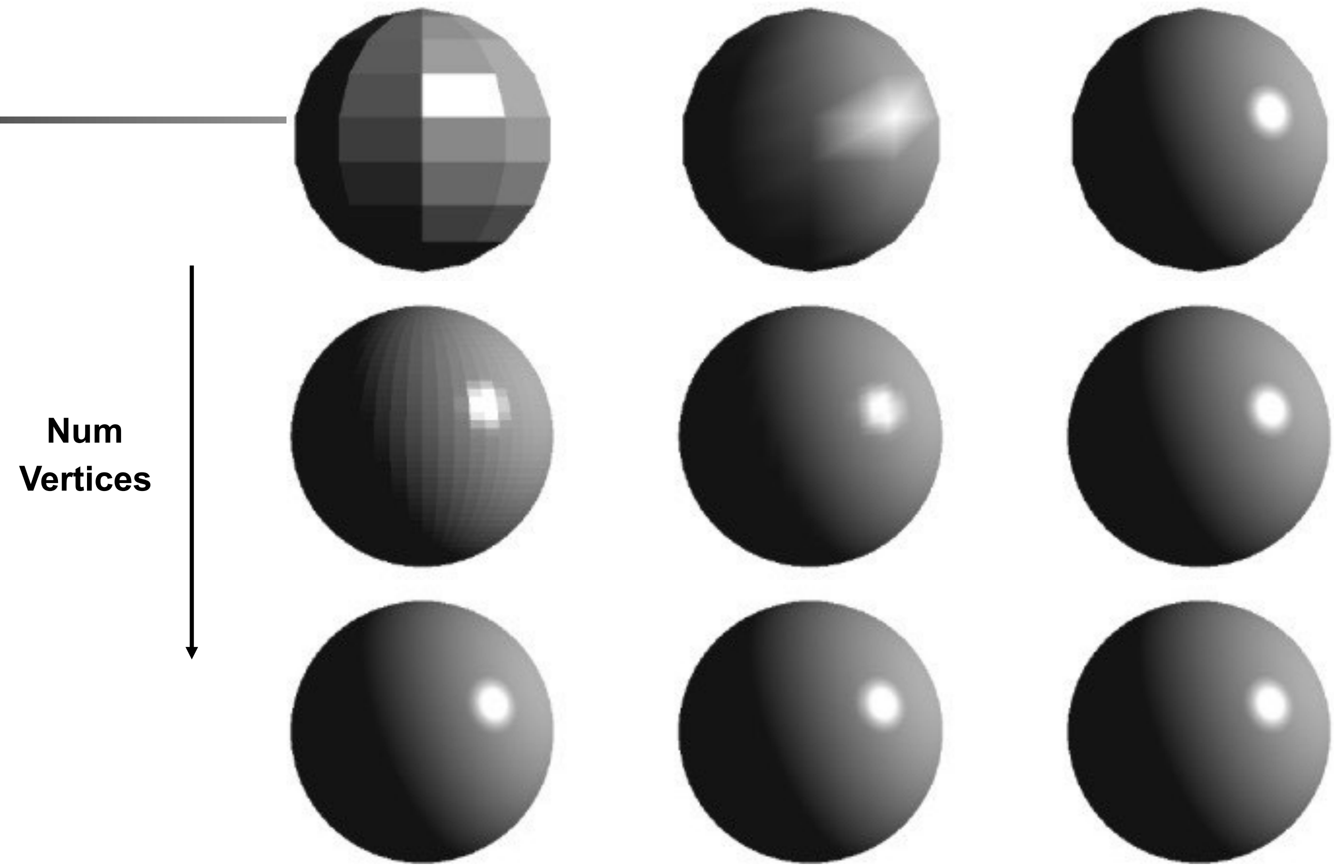
Shade each pixel (Phong shading)

Phong shading

- Interpolate normal vectors across each triangle
- Compute full shading model at each pixel
- Not the **Blinn-Phong Reflectance Model**



Shading Frequency: Face, Vertex or Pixel



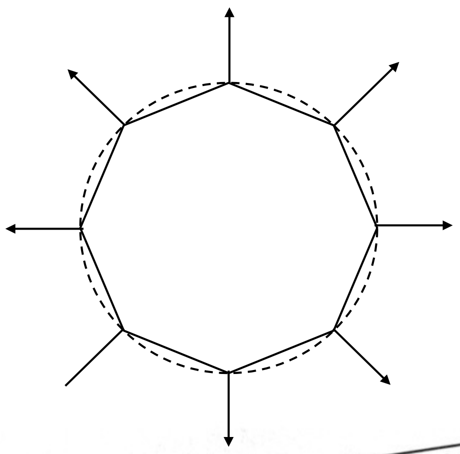
Shading freq. :	Face	Vertex	Pixel
Shading type :	Flat	Gouraud	Phong

Image credit: Happyman, http://cg2010studio.com/

Defining Per-Vertex Normal Vectors

Best to get vertex normals from the underlying geometry

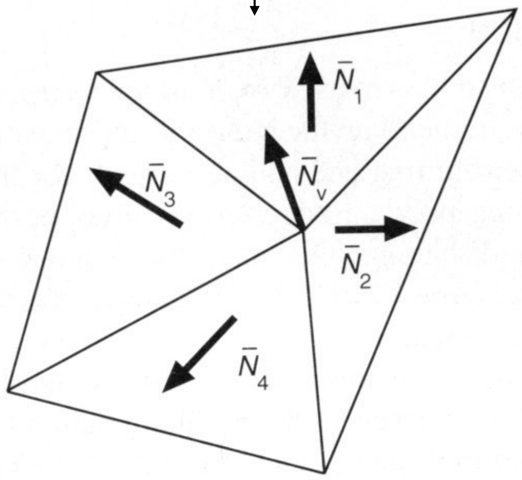
- e.g. consider a sphere



Otherwise have to infer vertex normals from triangle faces

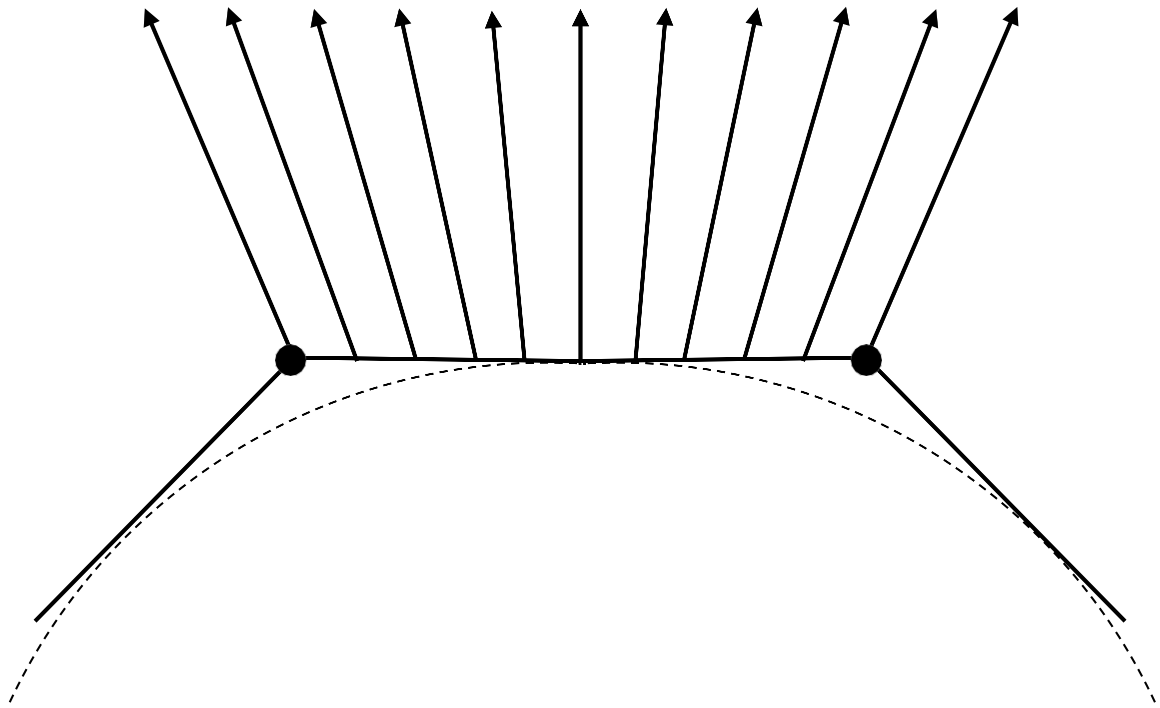
- Simple scheme: **average surrounding face normals**

$$N_v = \frac{\sum_i N_i}{\| \sum_i N_i \|}$$



Defining Per-Pixel Normal Vectors

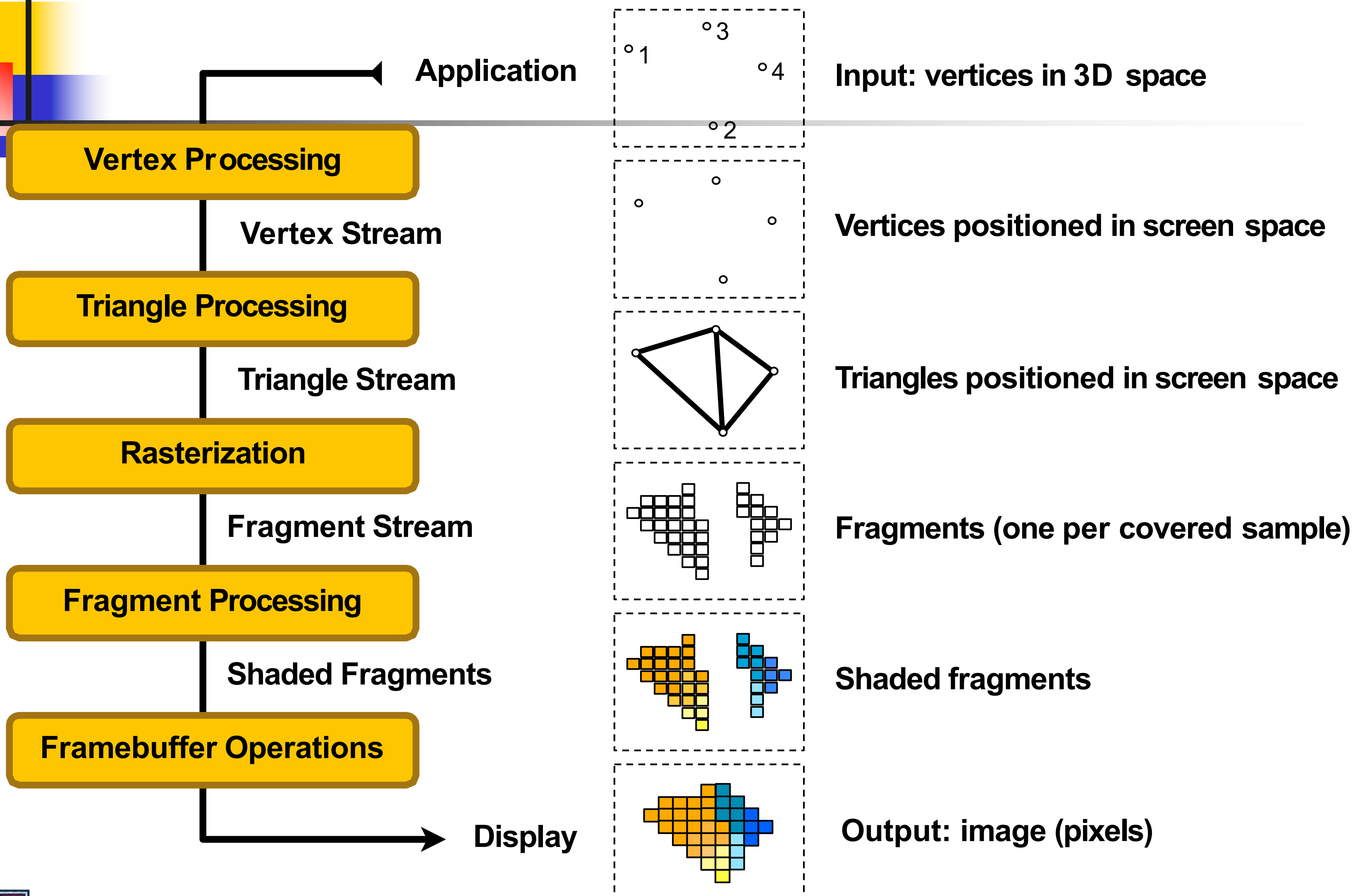
Barycentric interpolation of vertex normals



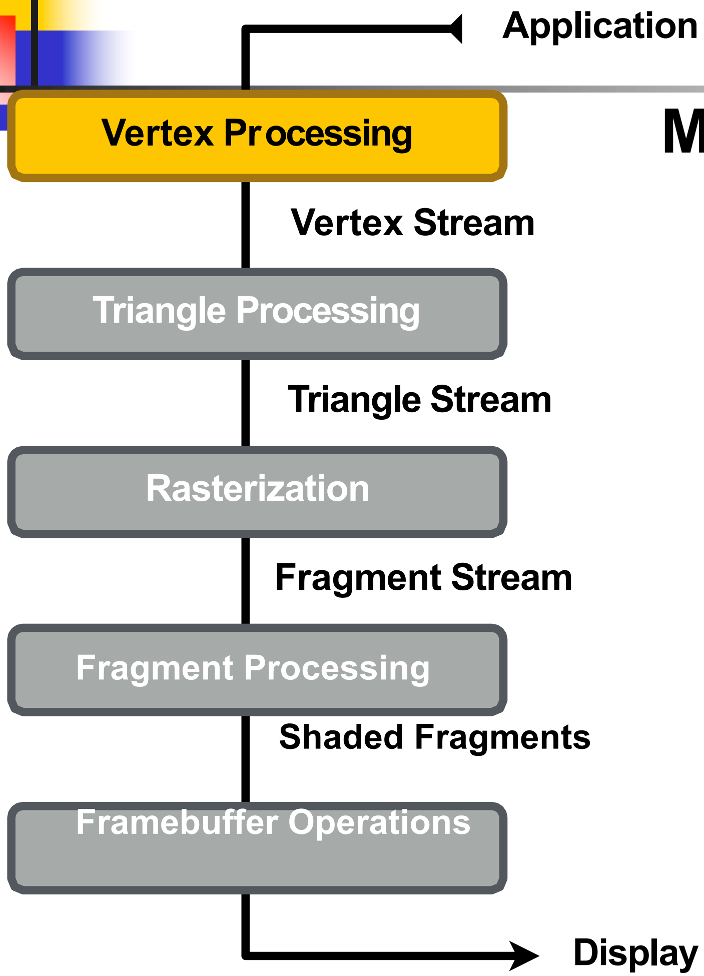
Don’t forget to normalize the interpolated directions

Graphics (**Real-time Rendering**) Pipeline

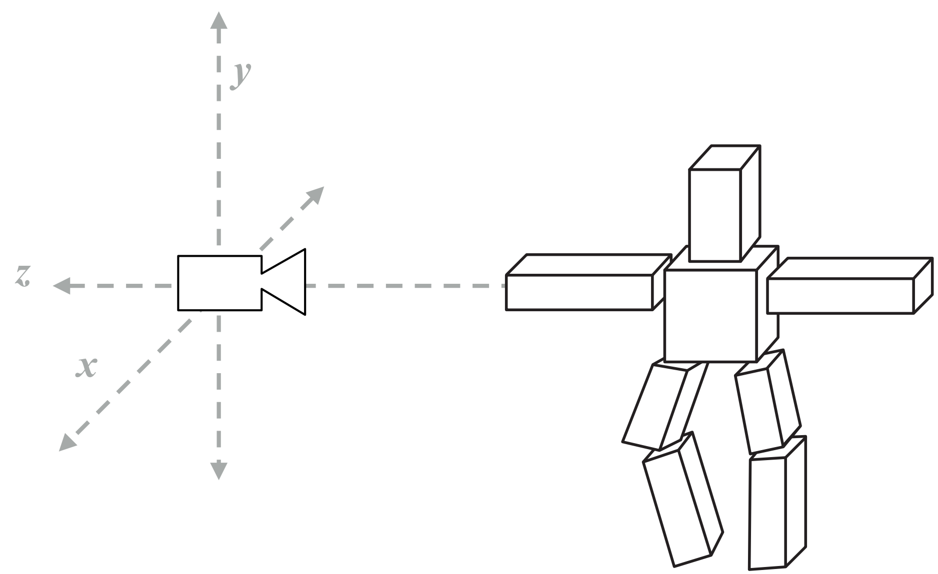
Graphics Pipeline



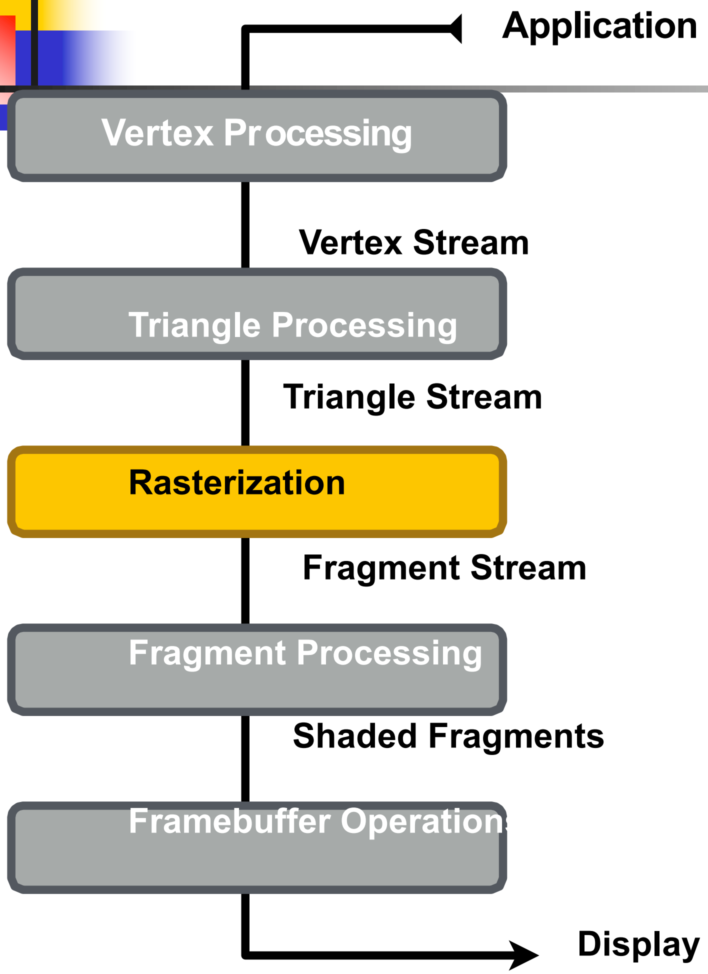
Graphics Pipeline



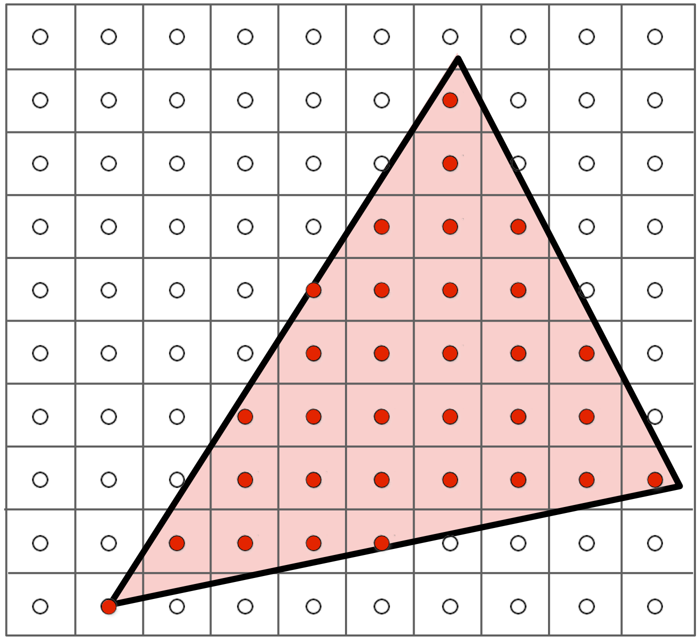
Model, View, Projection transforms



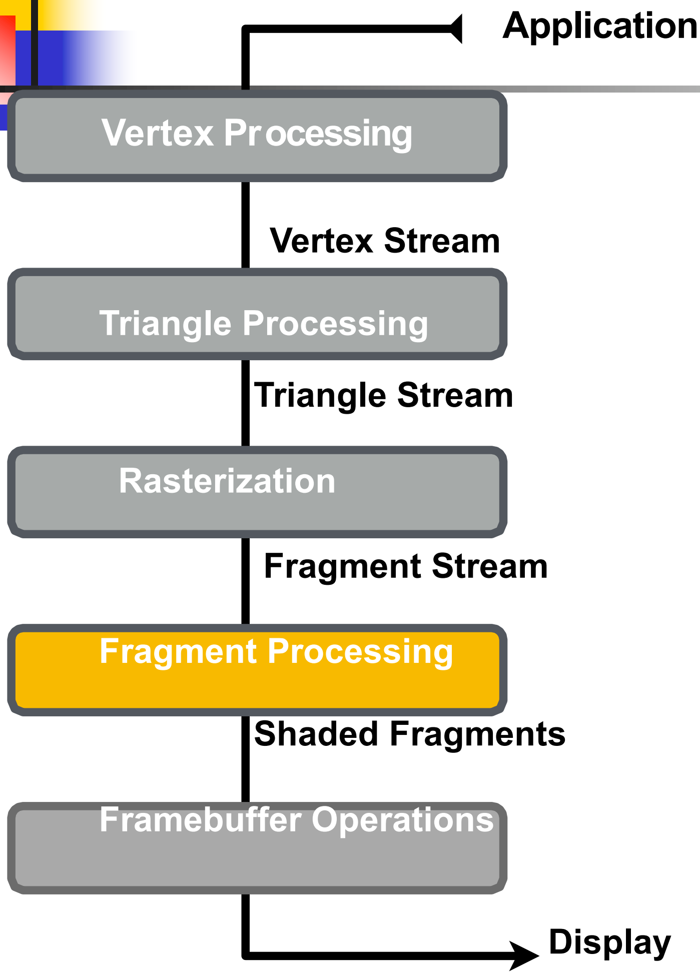
Graphics Pipeline



Sampling triangle coverage

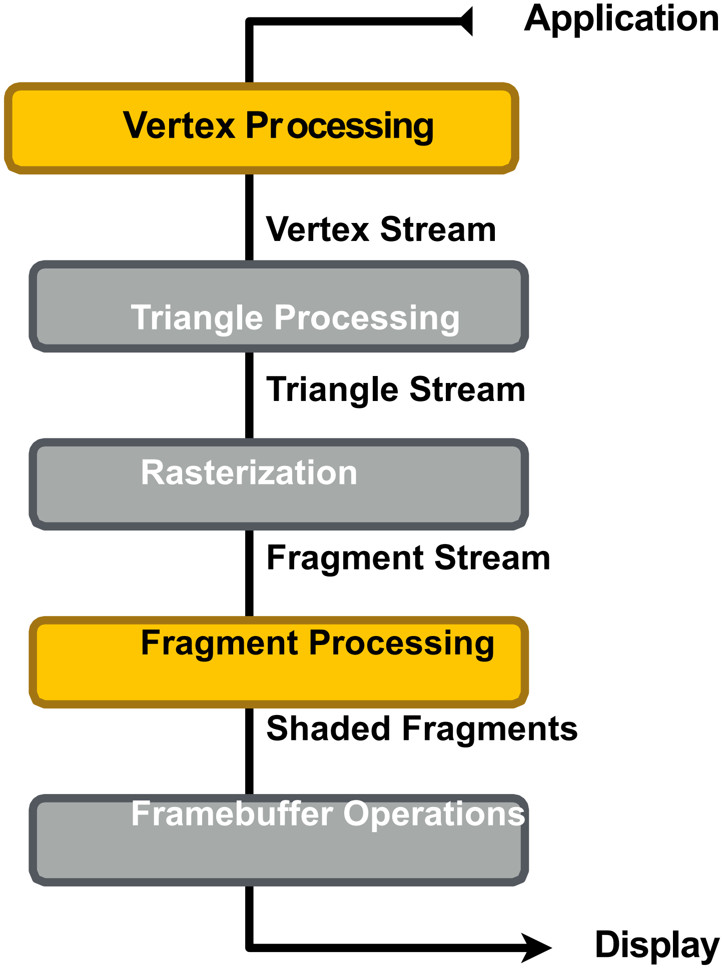


Rasterization Pipeline

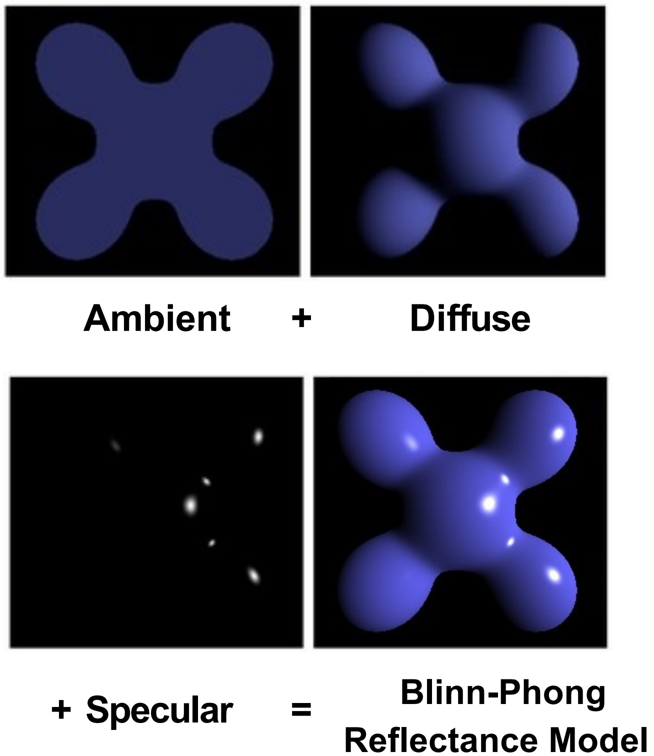


Z-Buffer Visibility Tests

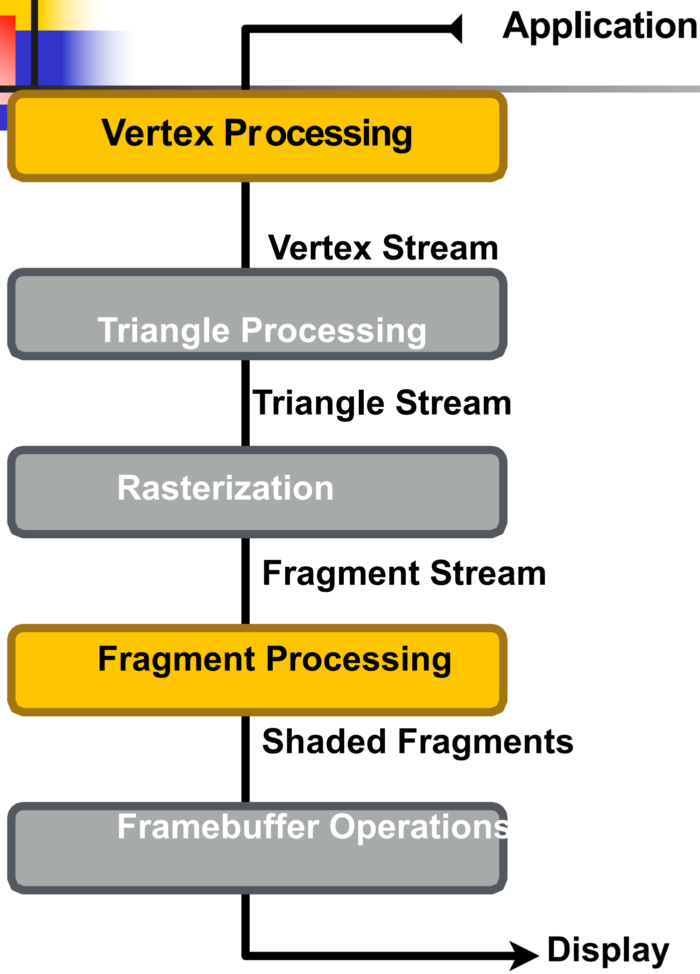
Graphics Pipeline



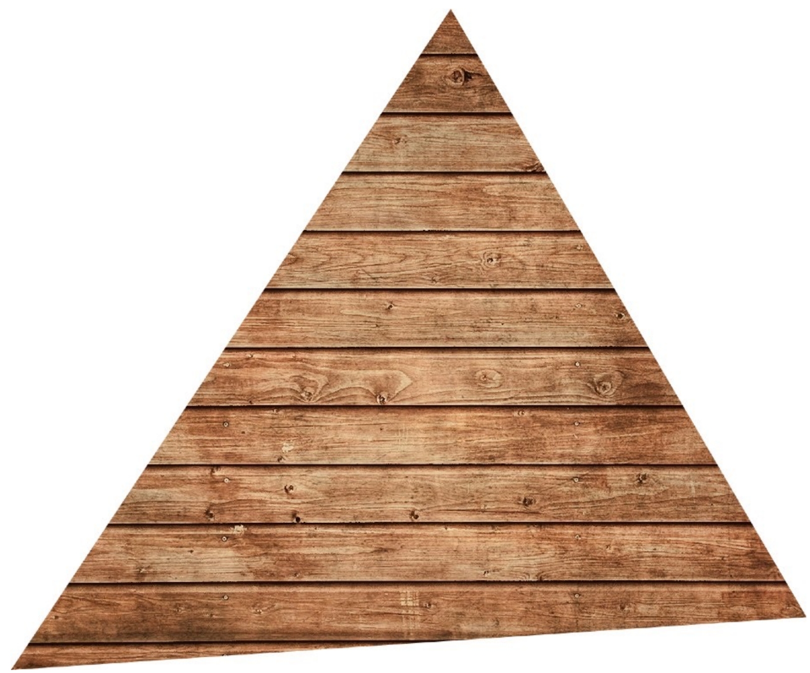
Shading



Graphics Pipeline



Texture mapping



Shader Programs

- Program vertex and fragment processing stages
- Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

```
uniform sampler2D myTexture;  uniform vec3 lightDir;
varying vec2 uv;  varying vec3 norm;

void diffuseShader()
{
vec3 kd;
kd = texture2d(myTexture, uv);
kd *= clamp(dot(–lightDir, norm), 0.0, 1.0);  gl_FragColor =
vec4(kd, 1.0);
}
```

- **Shader function executes once per fragment.**
- **Outputs color of surface at the current fragment's screen sample position.**
- **This shader performs a texture lookup to obtain the surface's material color at this point, then performs a diffuse lighting calculation.**

Shader Programs

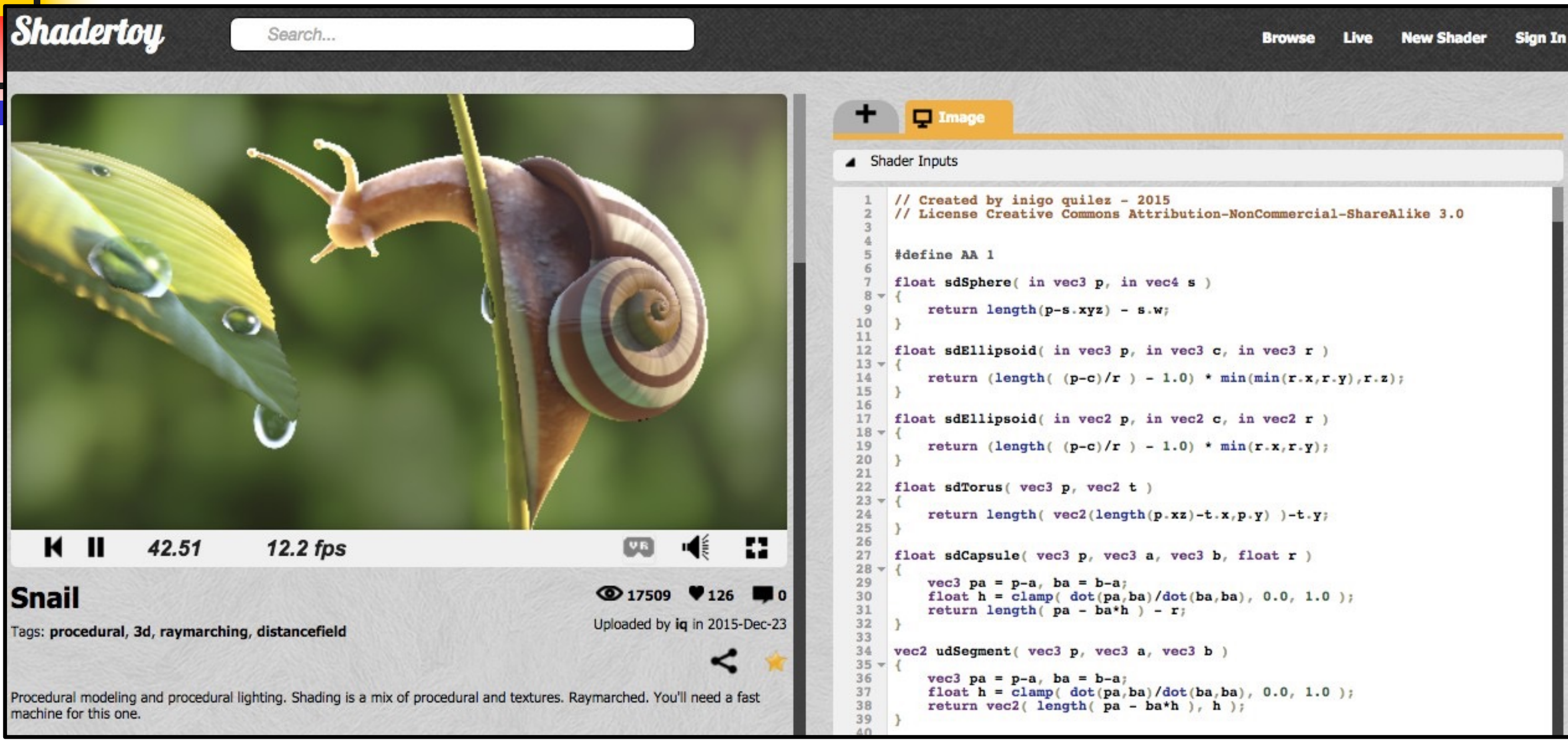
- Program vertex and fragment processing stages
- Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

```
uniform sampler2D myTexture;          // program parameter
uniform vec3 lightDir;                // program parameter
varying vec2 uv;  varying vec3        // per fragment value (interp. by rasterizer)
norm;                                 // per fragment value (interp. by rasterizer)

void diffuseShader()
{
vec3 kd;
kd = texture2d(myTexture, uv);                      // material color from texture
kd *= clamp(dot(–lightDir, norm), 0.0, 1.0);        // Lambertian shading model
gl_FragColor = vec4(kd, 1.0);                       // output fragment color
}
```

Snail Shader Program

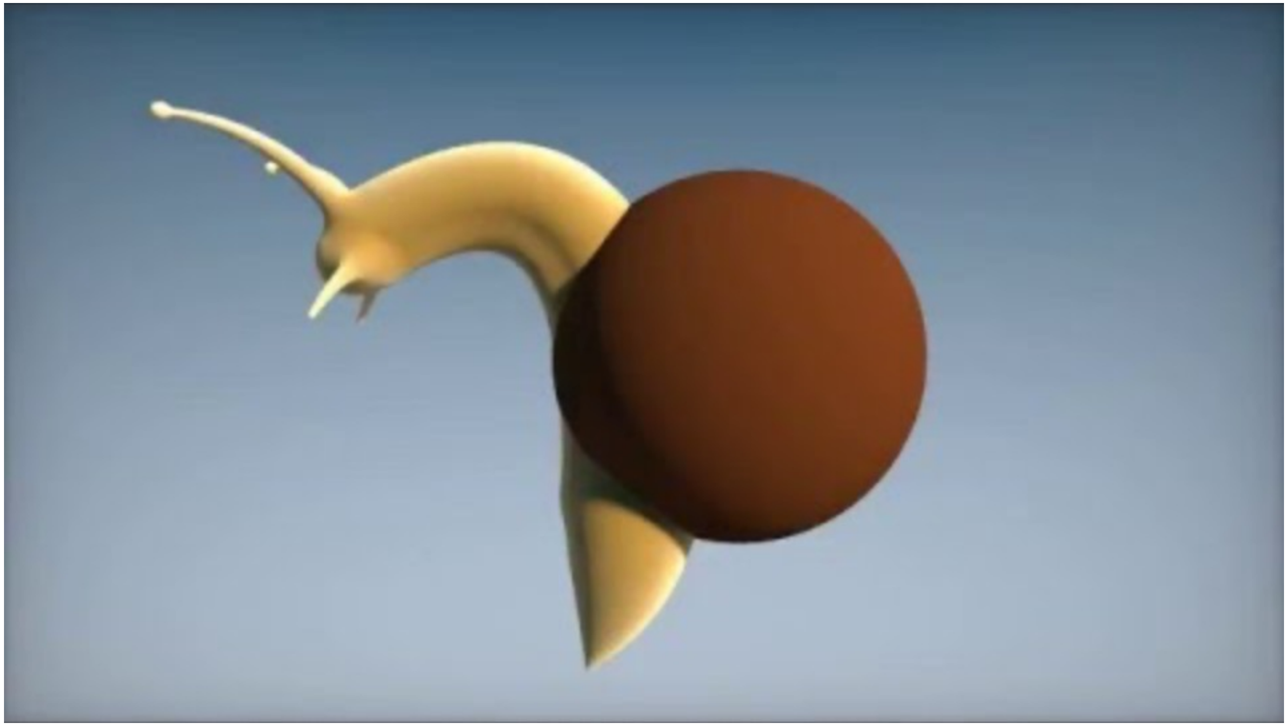


Inigo Quilez

Procedurally modeled, 800 line shader.

http://shadertoy.com/view/ld3Gz2

Snail Shader Program



Inigo Quilez, https://youtu.be/XuSnLbB1j6E

Goal: Highly Complex 3D Scenes in Realtime

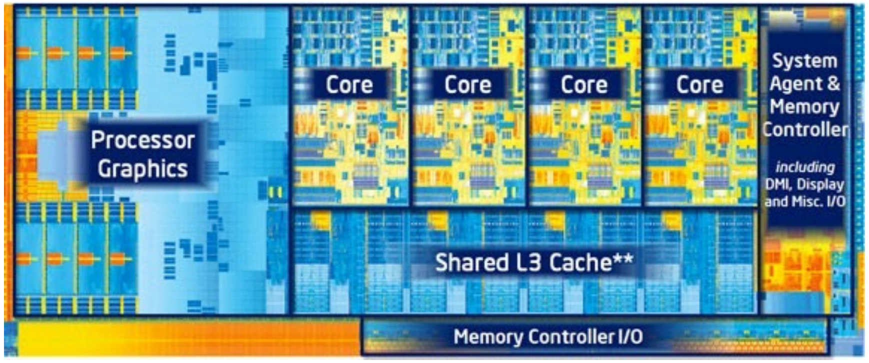
- **100's of thousands to millions of triangles in a scene**
- **Complex vertex and fragment shader computations**
- **High resolution (2-4 megapixel + supersampling)**
- **30-60 frames per second (even higher for VR)**

Unreal Engine Kite Demo (Epic Games 2015)

Graphics Pipeline Implementation: GPUs

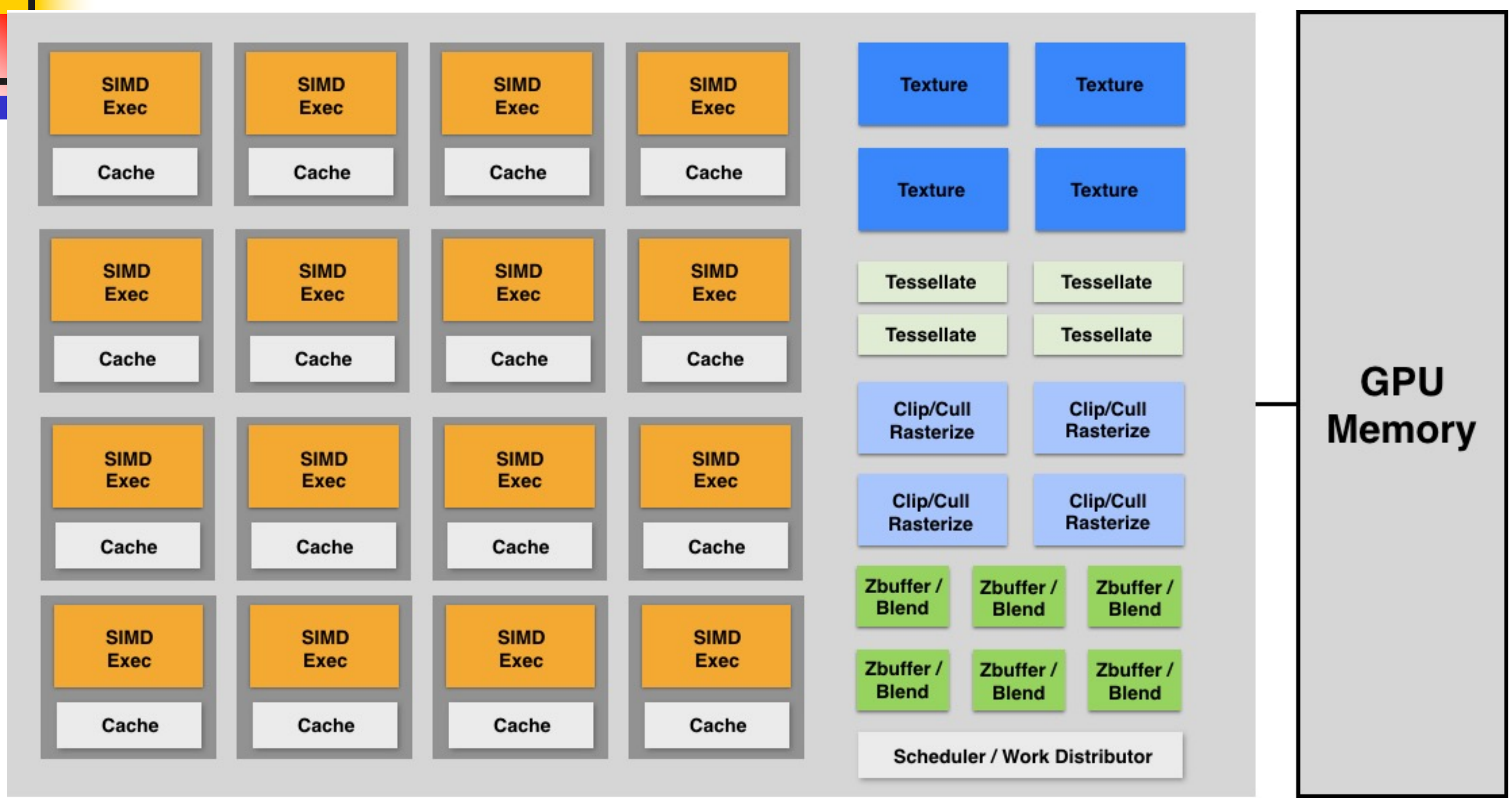
- **Specialized processors for executing graphics pipeline computations**

Discrete GPU Card (NVIDIA GeForce Titan X)



Integrated GPU: (Part of Intel CPU die)

GPU: Heterogeneous, Multi-Core Procesor



Modern GPUs offer ~2-4 Tera-FLOPs of performance for executing vertex and fragment shader programs

Tera-Op's of fixed-function compute capability over here

Thank you!

(And thank Prof. Lingqi Yan, Prof. Ravi Ramamoorthi and Prof. Ren Ng for many of the slides!)