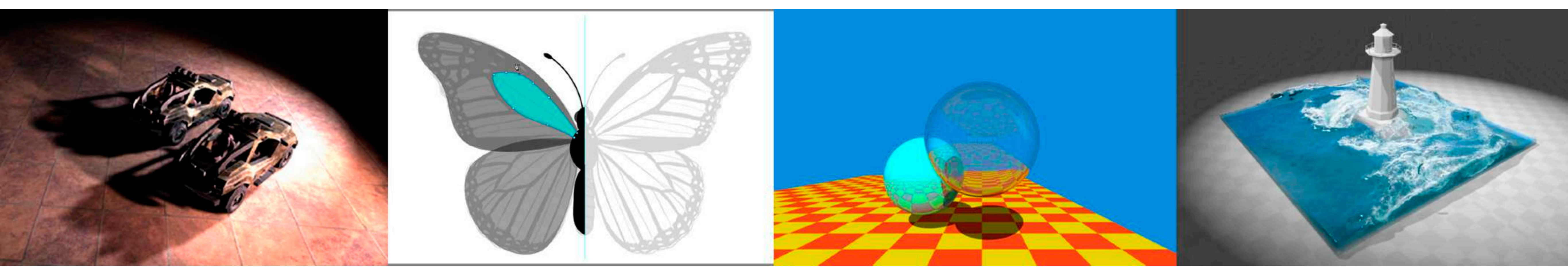
# Introduction to Computer Graphics

GAMES101, Lingqi Yan, UC Santa Barbara

# Lecture 8:

## Shading 2
## (Shading, Pipeline and Texture Mapping)



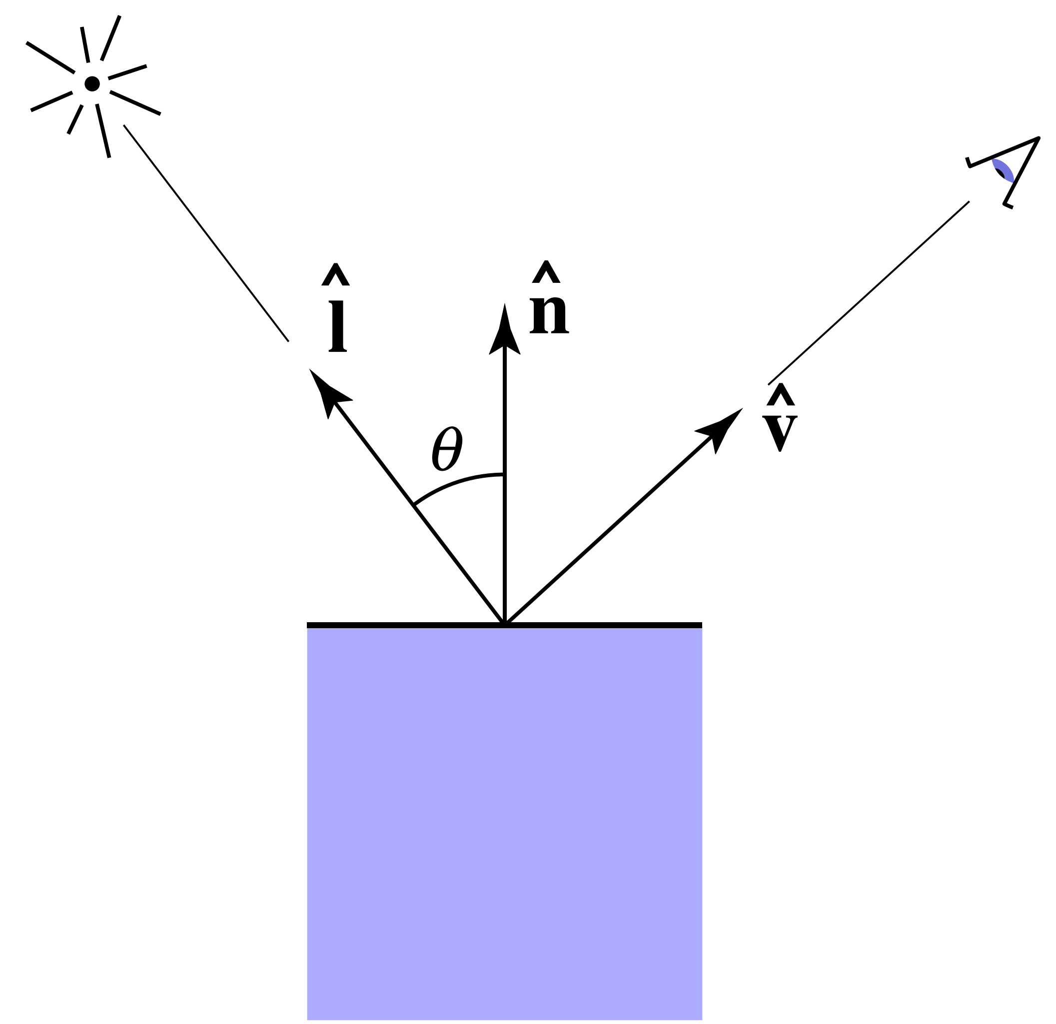
http://www.cs.ucsb.edu/~lingqi/teaching/games101.html

# Announcements

- Homework 2
  - 45 submissions so far
  - Upside down? No problem
  - Active discussions in the BBS, pretty good
- Next homework is for shading
- Today's topics
  - Easy, but a lot

# Last Lecture

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

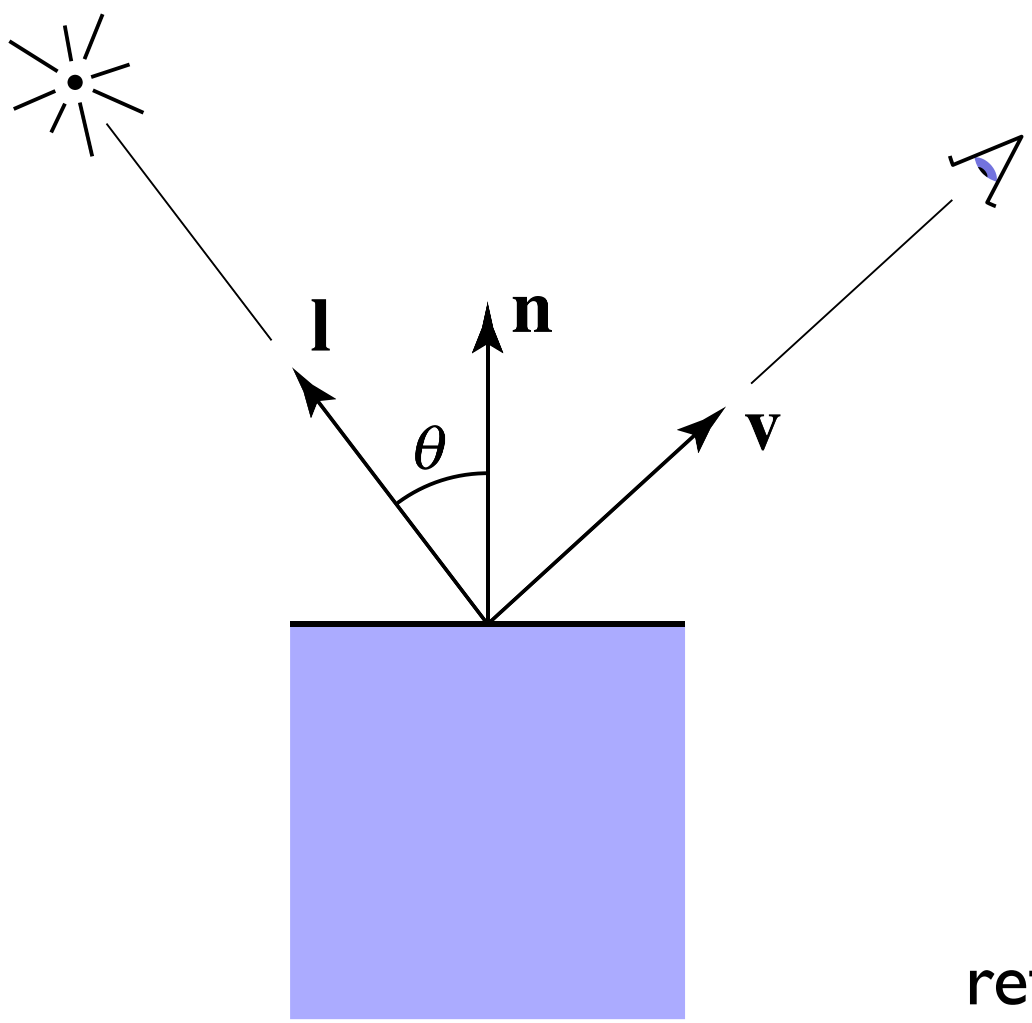


# Today

- 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

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

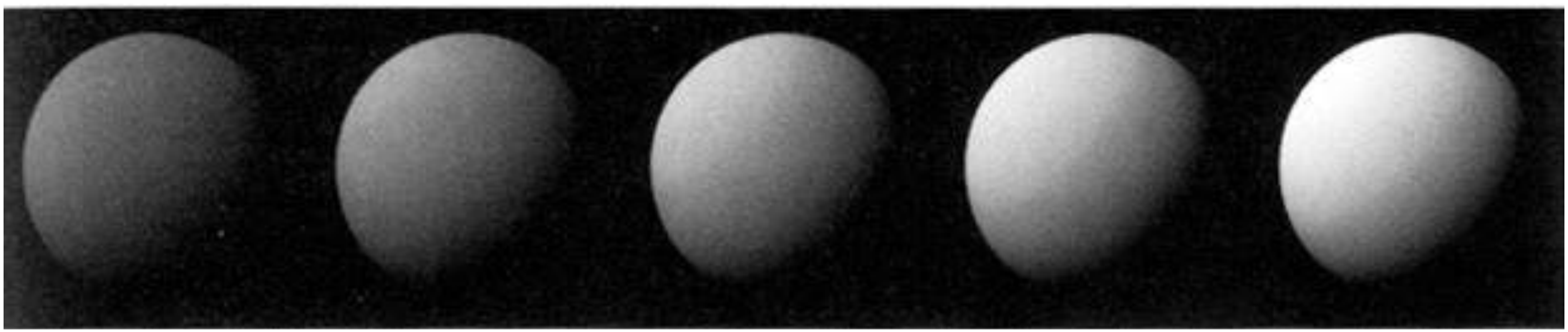
diffusely
reflected light

diffuse
coefficient
(color)

energy received
by the shading point

# Recap: Lambertian (Diffuse) Term

Produces diffuse appearance



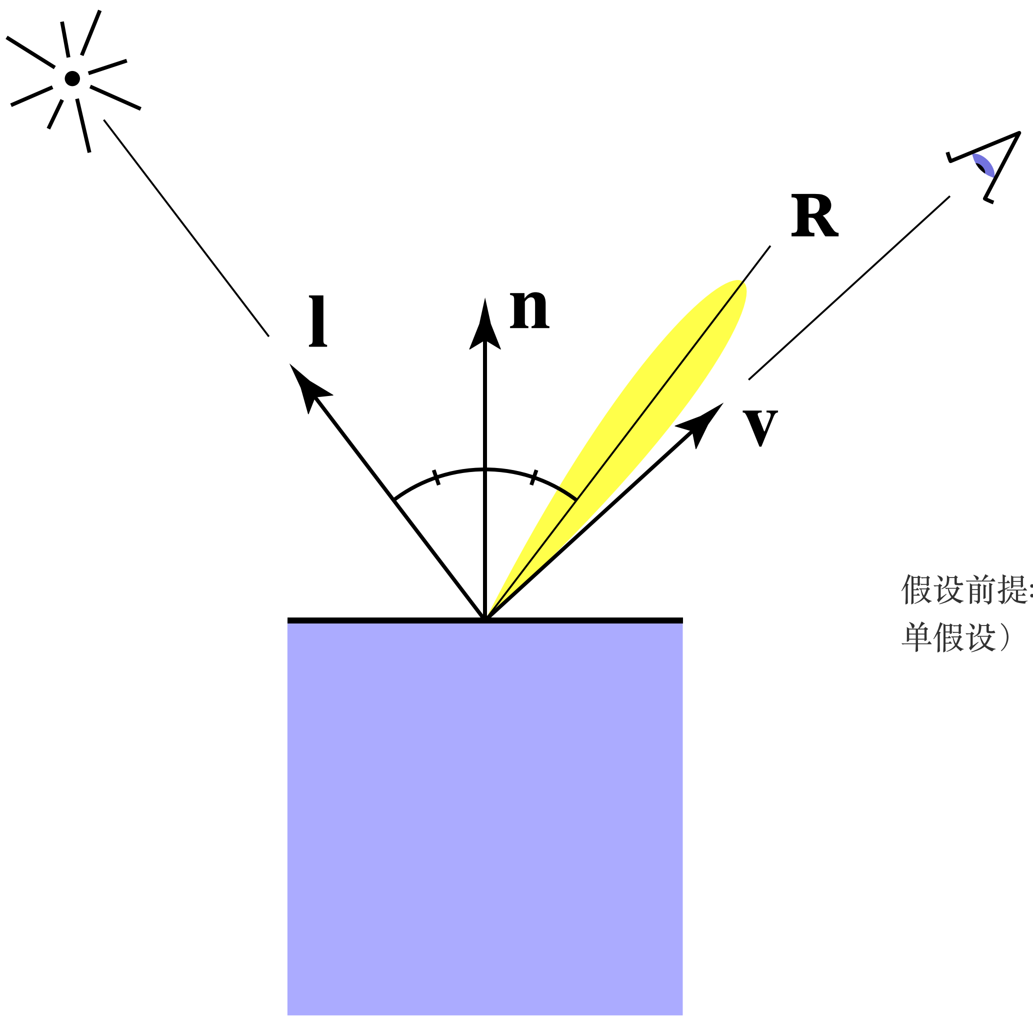
$k_d \longrightarrow$

[Foley et al.]

# Specular Term (Blinn-Phong)

Intensity depends on view direction

- Bright near mirror reflection direction




假设前提:观测向量 v 与镜面反射向量 R 的夹角 与 法向量 和 半程向量 的夹角相等（简单假设）

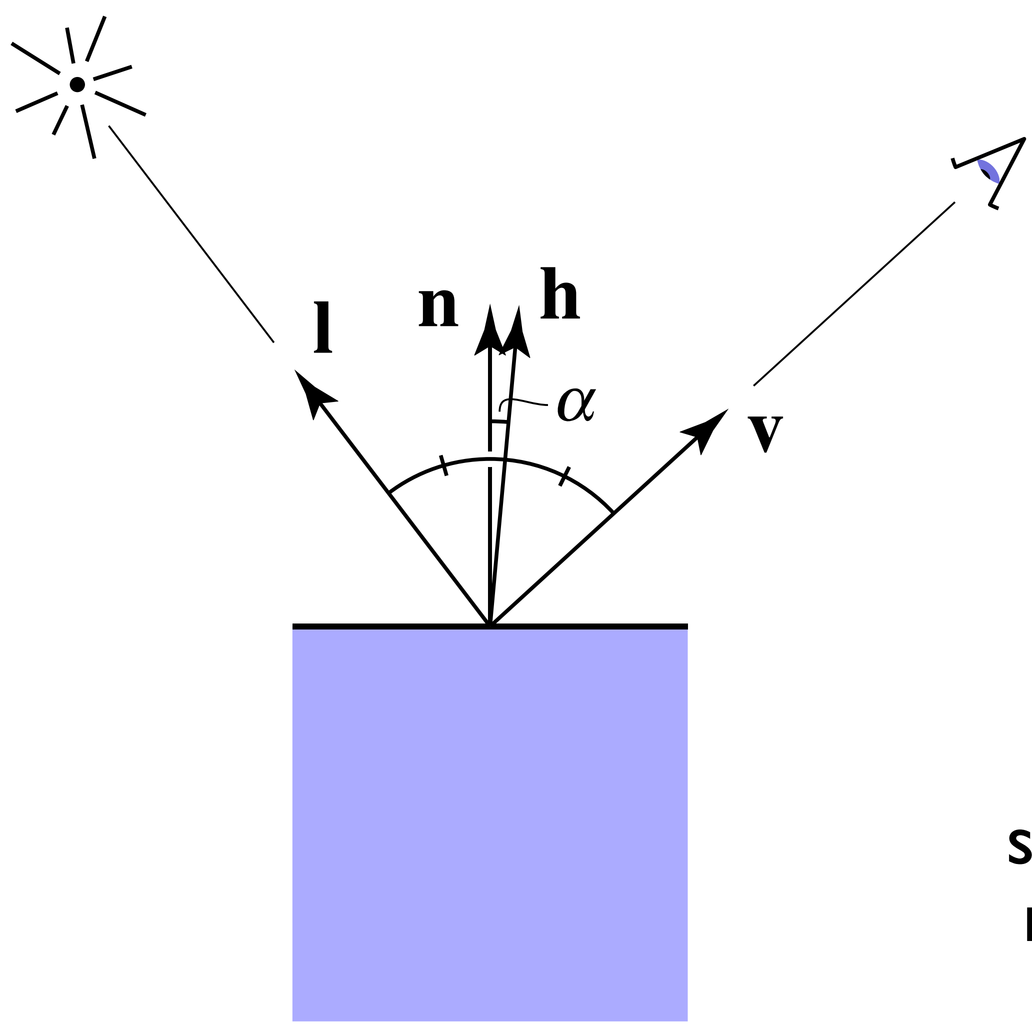
# Specular Term (Blinn-Phong)

高光相

V close to mirror direction ⇔ **half vector** near normal

视向角度越接近于反射角度，所接收的光强越强，用半程向量和法向量的接近程度来衡量

- Measure “near” by dot product of unit vectors




$$\mathbf{h} = \operatorname{bisector}(\mathbf{v}, \mathbf{l})$$

(半程向量)

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

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

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

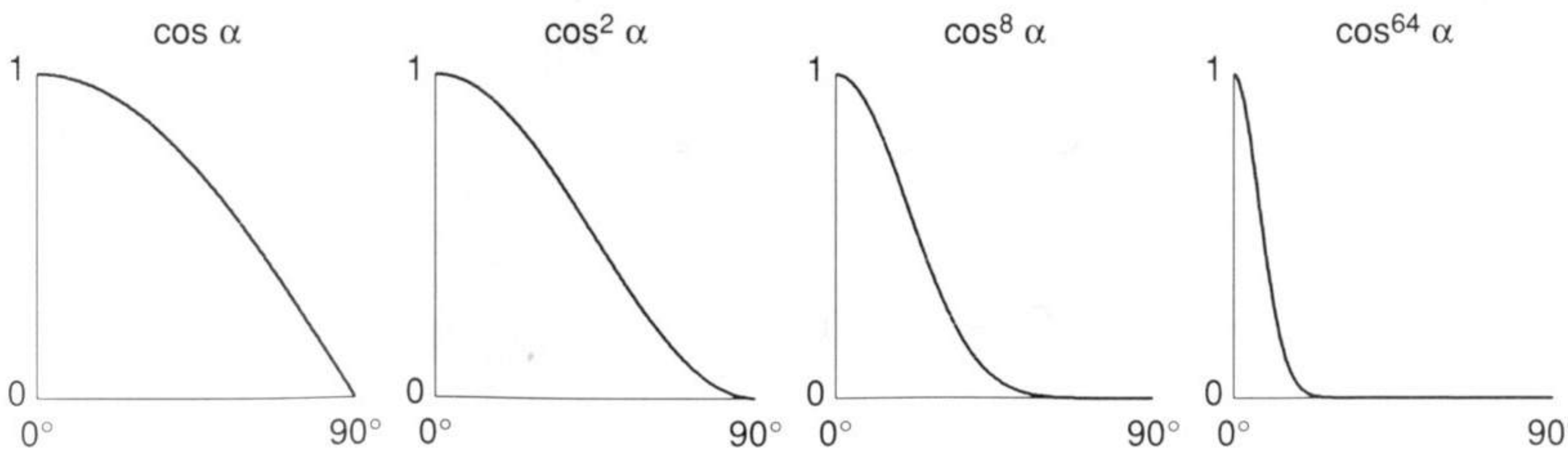
其中指数p来控制高光可见，一般选取100-200

specularly reflected light

specular coefficient

# Cosine Power Plots

Increasing p narrows the reflection lobe




[Foley et al.]

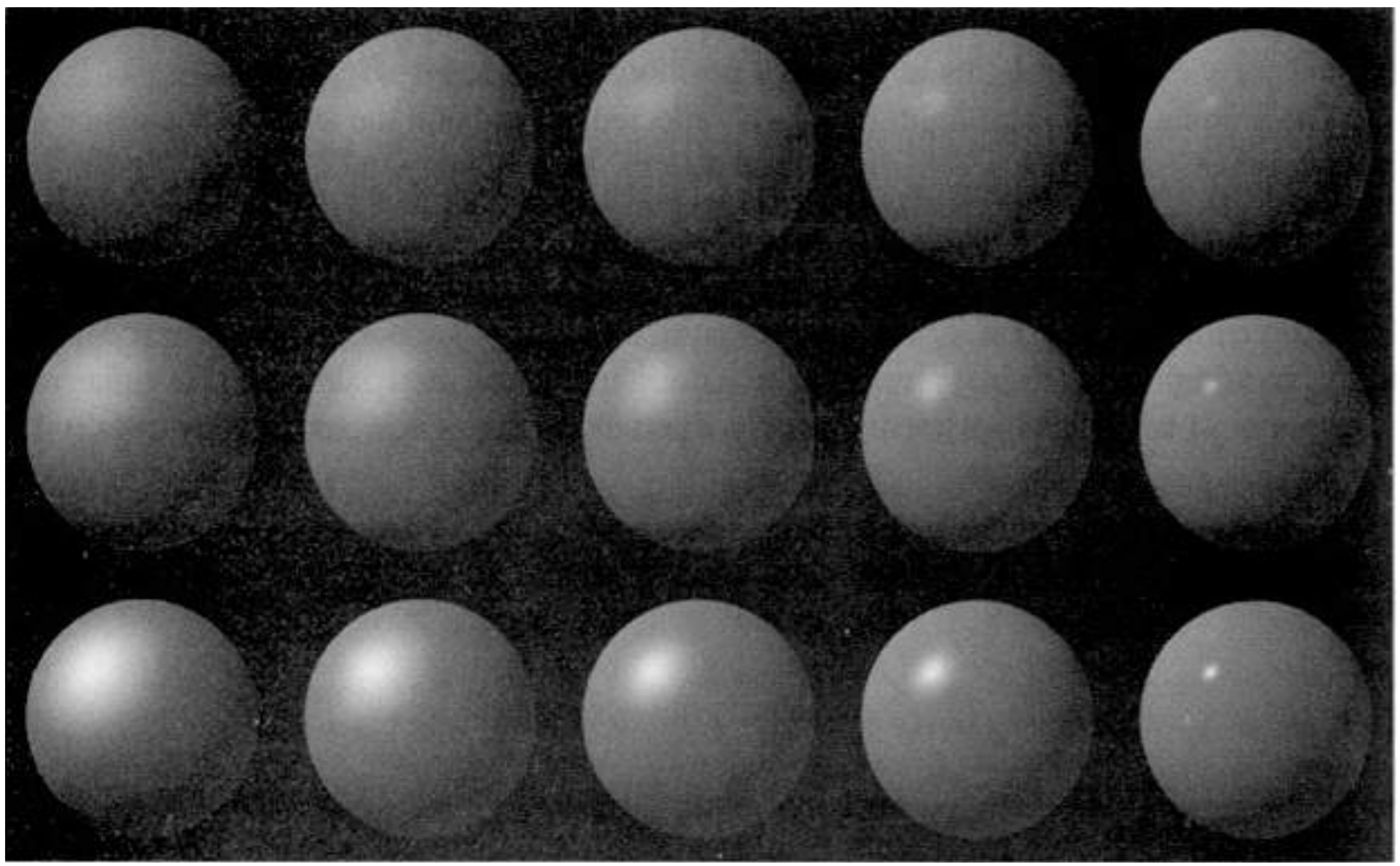
p 的值增加，更加反应高光对角度的敏感度

# Specular Term (Blinn-Phong)

Blinn-Phong

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

$k_s$



[Foley et al.]

Note: showing Ld + Ls together

$p$

不同光照强度和p的取值对高光效果的影响

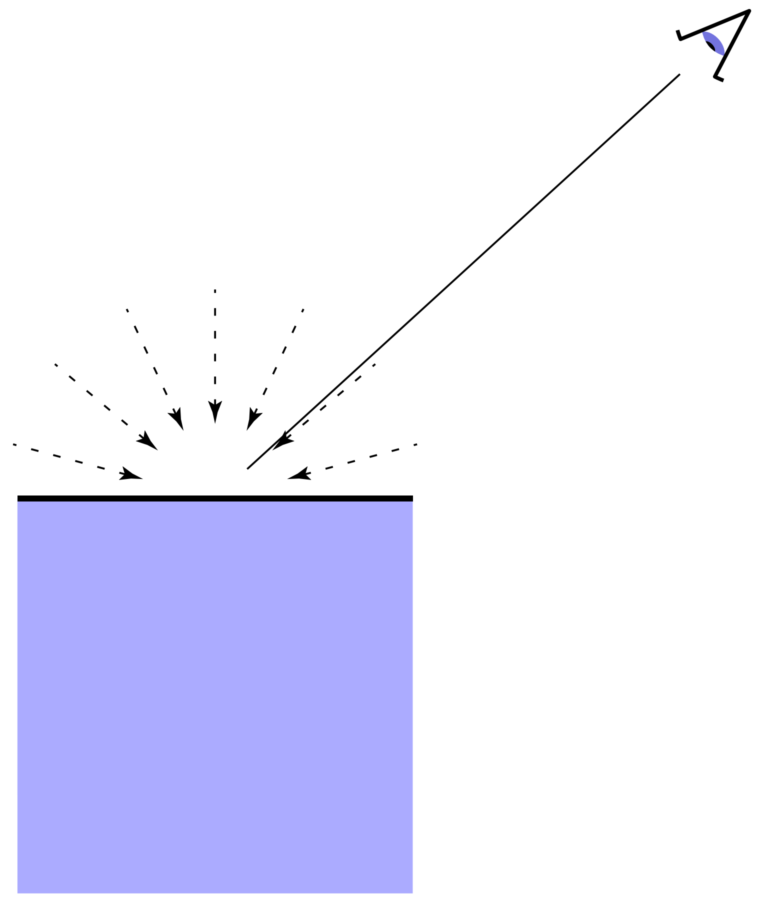
# Ambient Term

环境光照项

Shading that does not depend on anything

Blinn-Phone光照模型中，假设从四面八方反射而来的光的光强都是相等的，也就是说可以认为环境光强为一个常数 (实际上全局光照的计算要复杂的多)

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



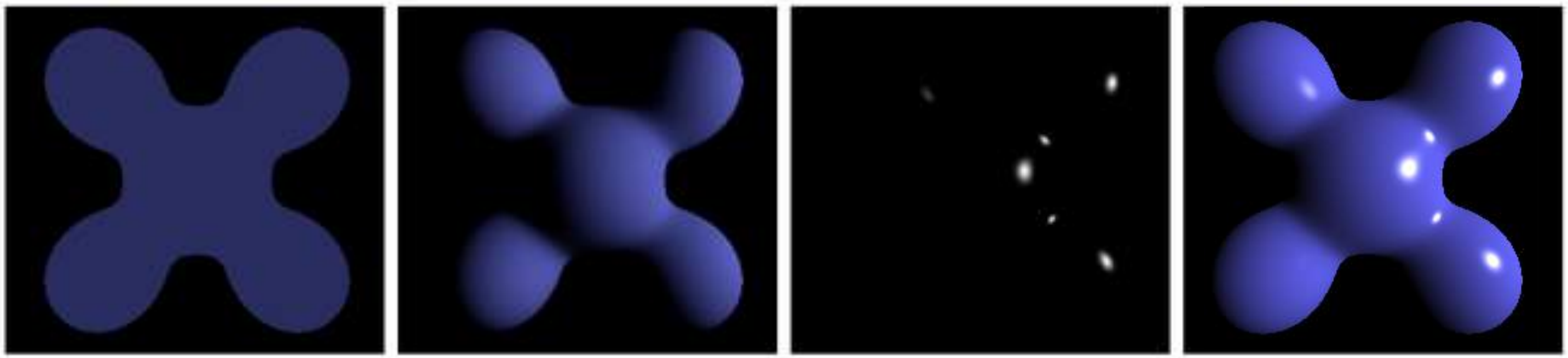
$$L_a = k_a \, I_a$$

ambient coefficient

reflected ambient light

# Blinn-Phong Reflection Model

完整的反射模型



**Ambient + Diffuse + Specular = Blinn-Phong Reflection**

$$
\begin{aligned}
L &= L_a + L_d + L_s \\
&= k_a\, I_a + k_d\, (I/r^2)\, \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, (I/r^2)\, \max(0, \mathbf{n} \cdot \mathbf{h})^p
\end{aligned}
$$

# Questions?

# 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

以三角面为单位进行着色，对于光滑的几何体效果很差

# Shade each vertex (Gouraud shading)

**Gouraud** shading

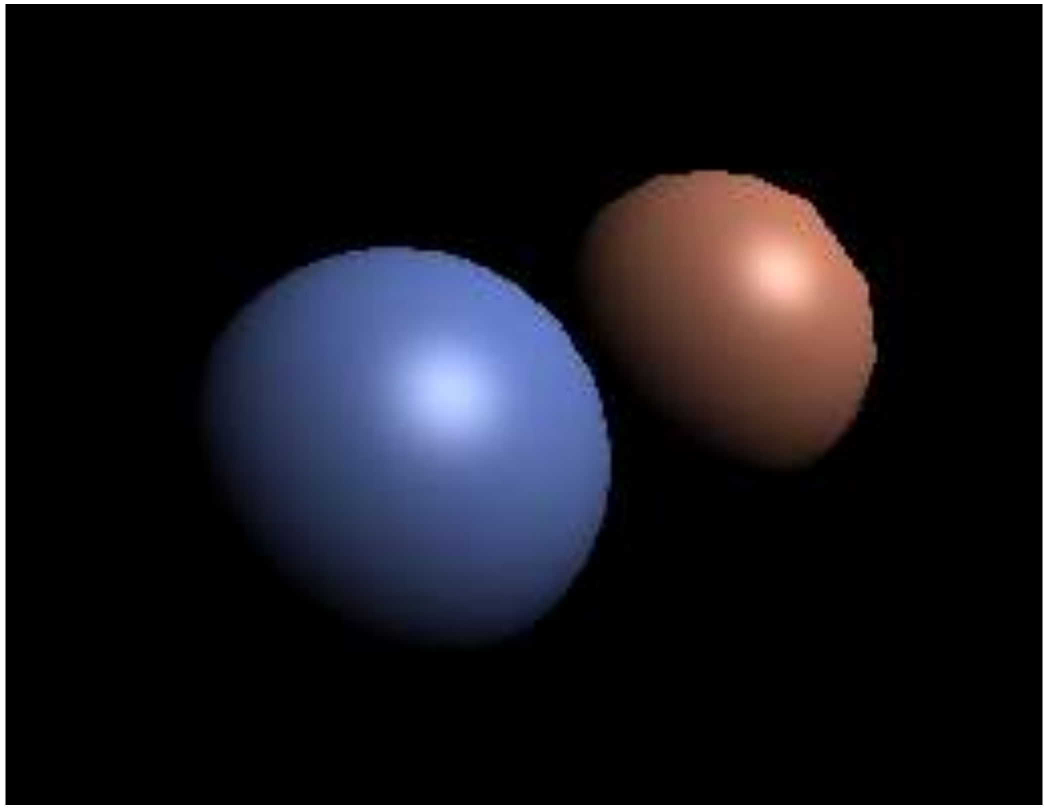
- Interpolate colors from vertices across triangle
- Each vertex has a normal vector (how?)

以顶点为单位进行着色，通过插值计算，实现点与点之间颜色的平滑过渡

# 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**

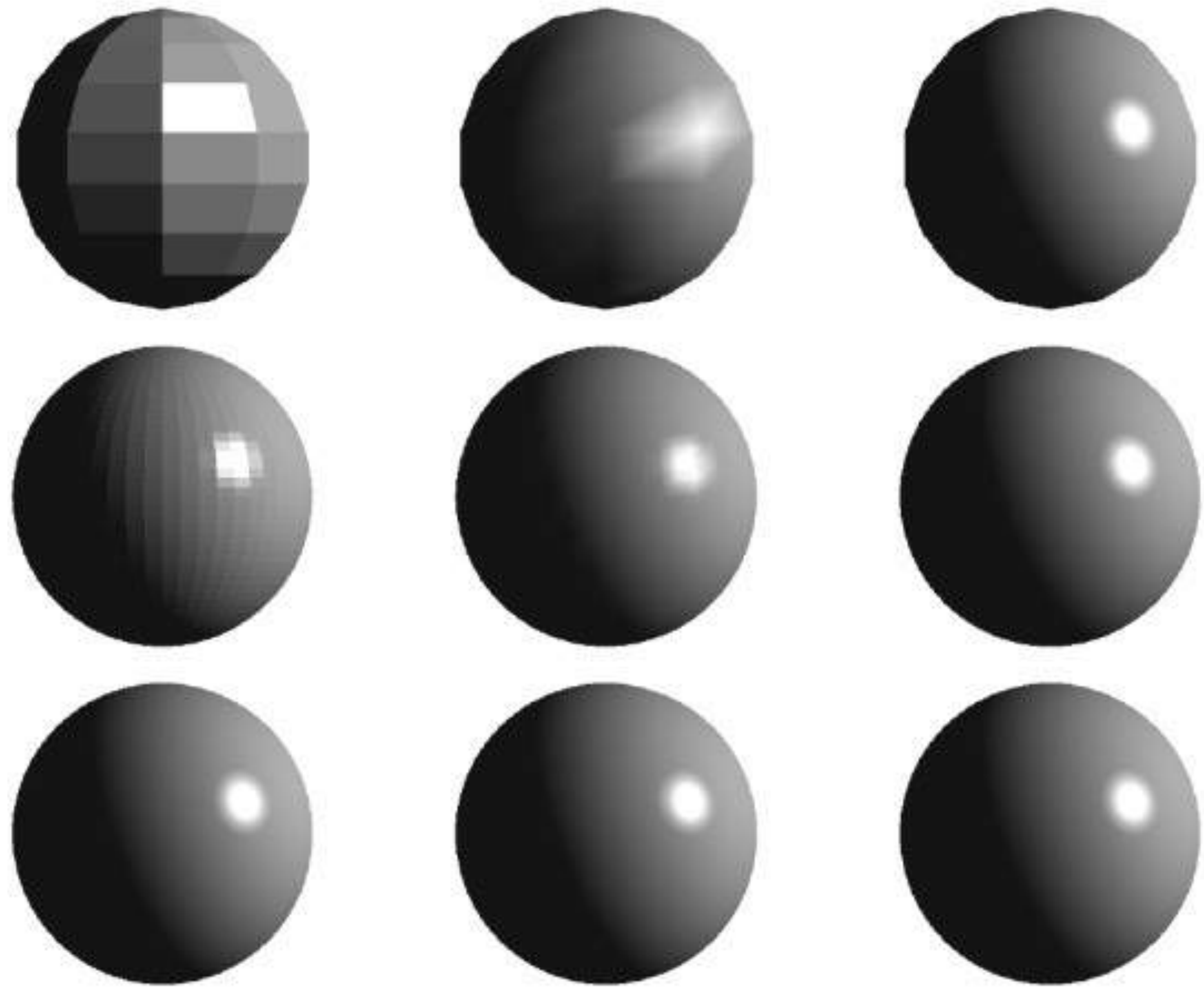


以片元（像素）为单位进行着色，对每个点计算一次光照，点的法向量是通过顶点法向量插值得到的，冯氏着色最接近现实， 可以在减少三角面数的情况下达到相同的效果(插值后法向量会光滑变化)，当然，性能开销也非常大

# Shading Frequency: Face, Vertex or Pixel

**Num Vertices**

当基元数量很多时，各个方法的着色结果基本一致



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

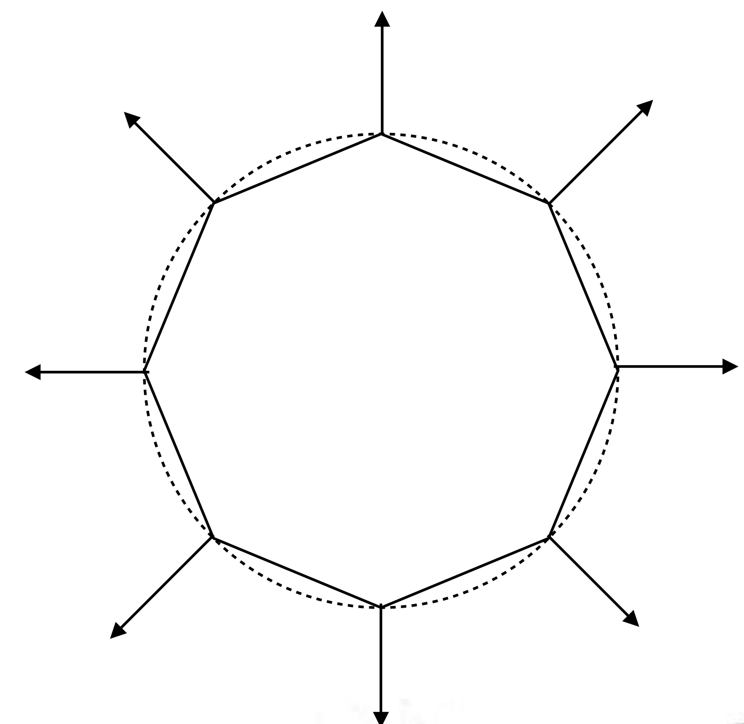
Image credit: Happyman, http://cg2010studio.com/

# Defining Per-Vertex Normal Vectors

基于集合形状获取顶点的法向量

Best to get vertex normals from the underlying geometry

- e.g. consider a sphere

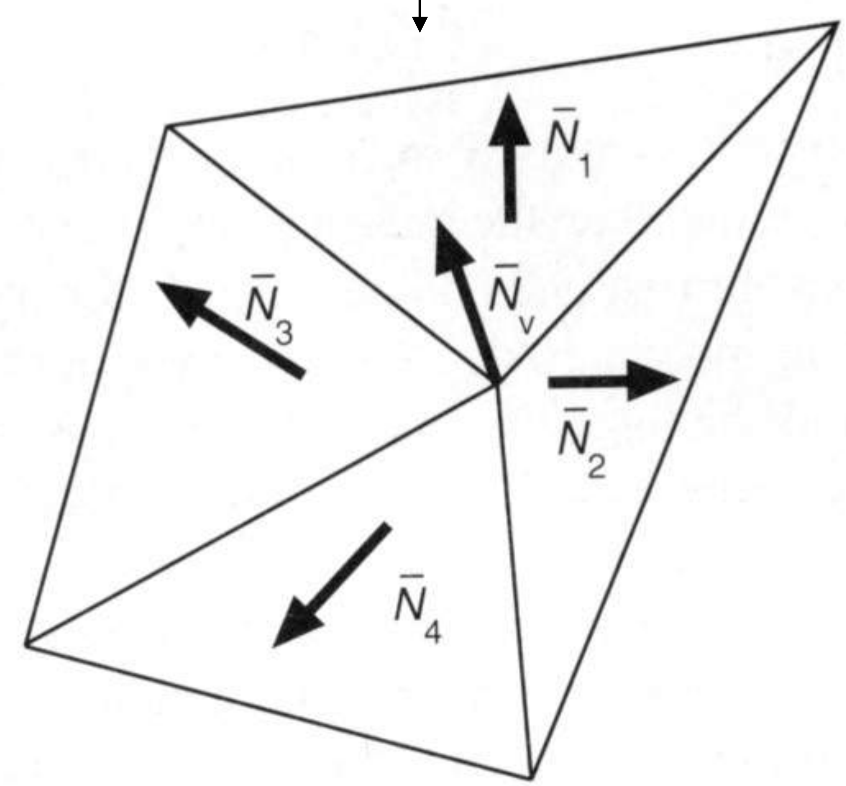


Otherwise have to infer vertex normals from triangle faces

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

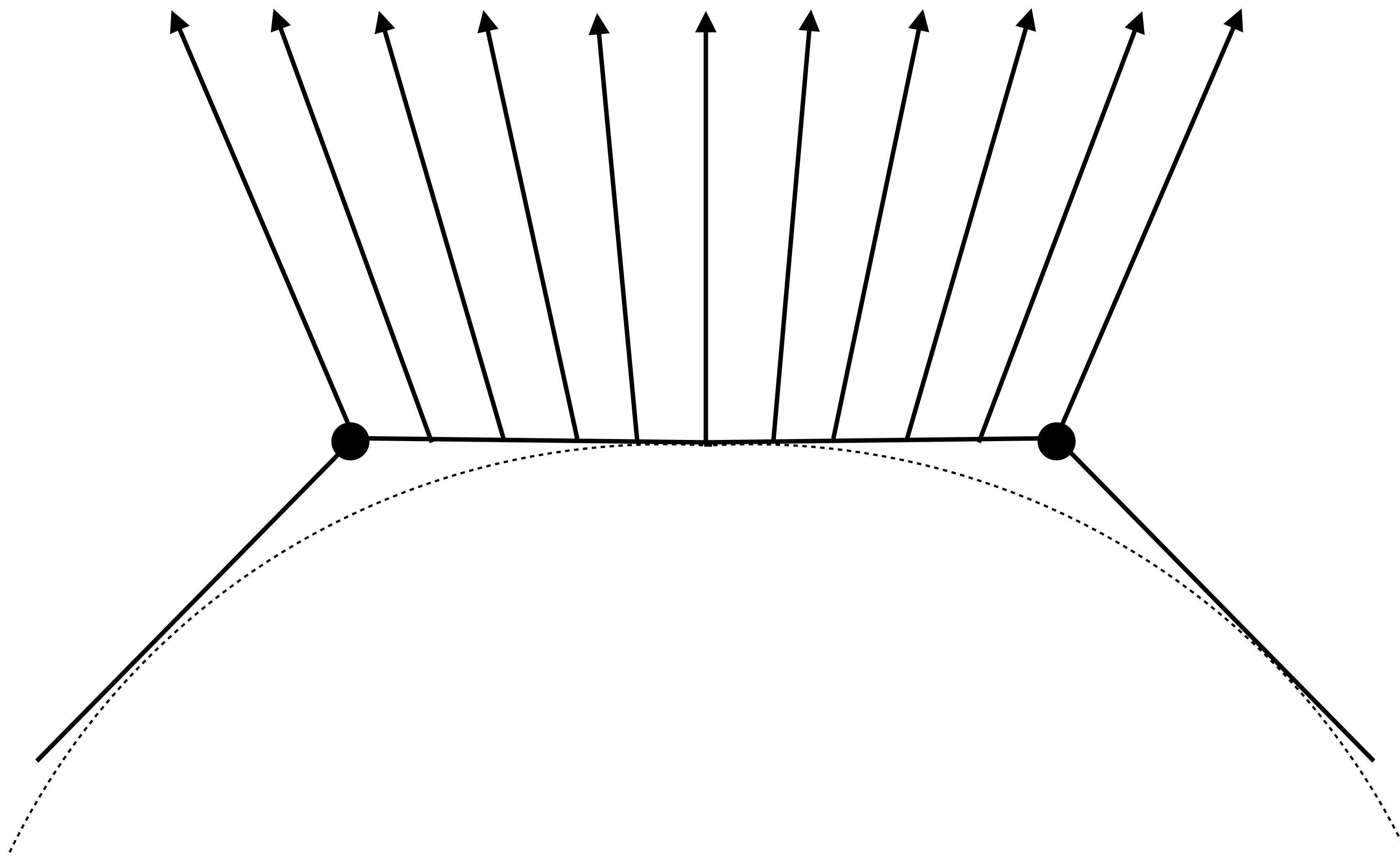
$$N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|}$$

对一般几何形状对顶点所关联的面的法线求平均



# Defining Per-Pixel Normal Vectors

Barycentric interpolation (introducing soon) of vertex normals

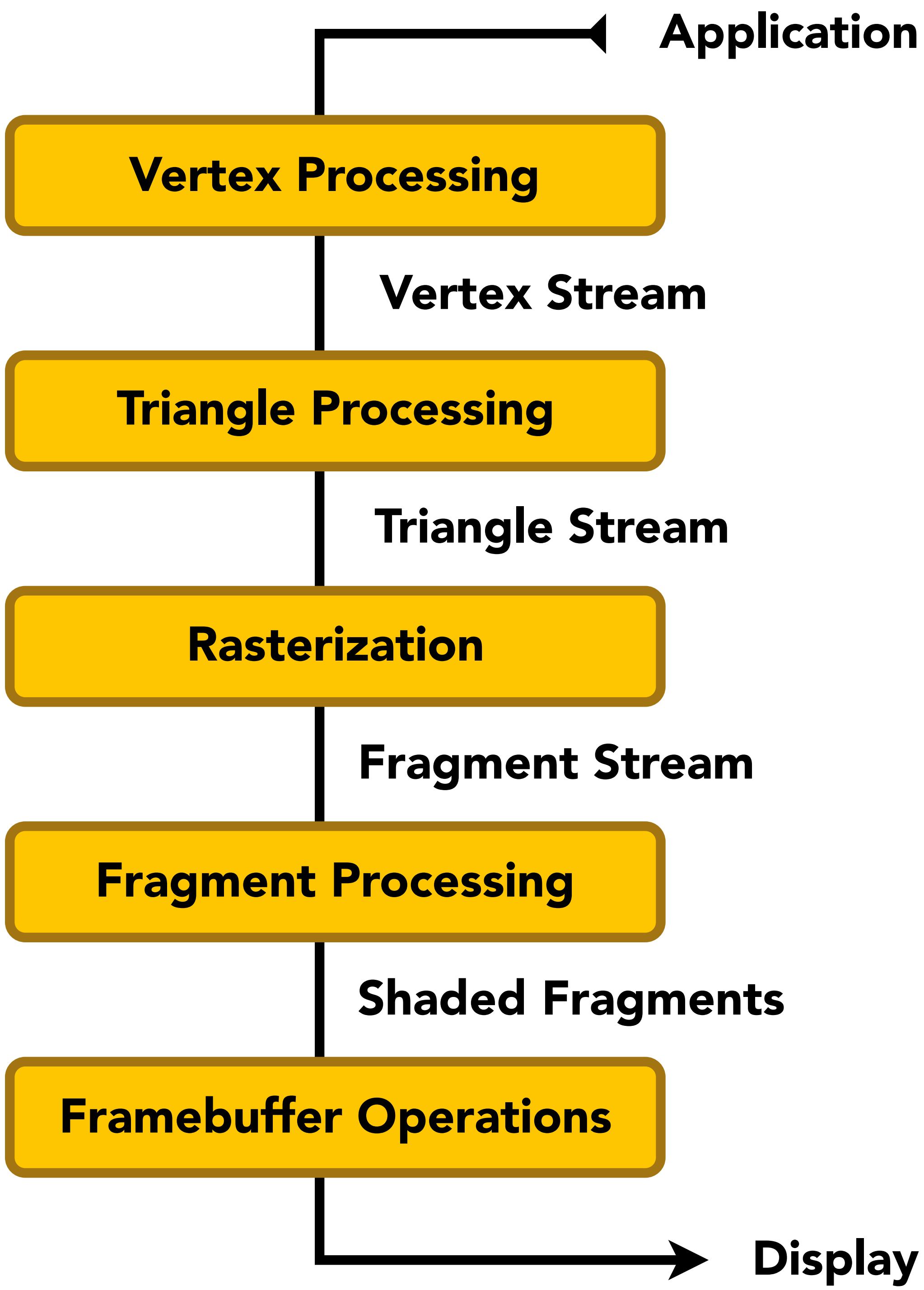


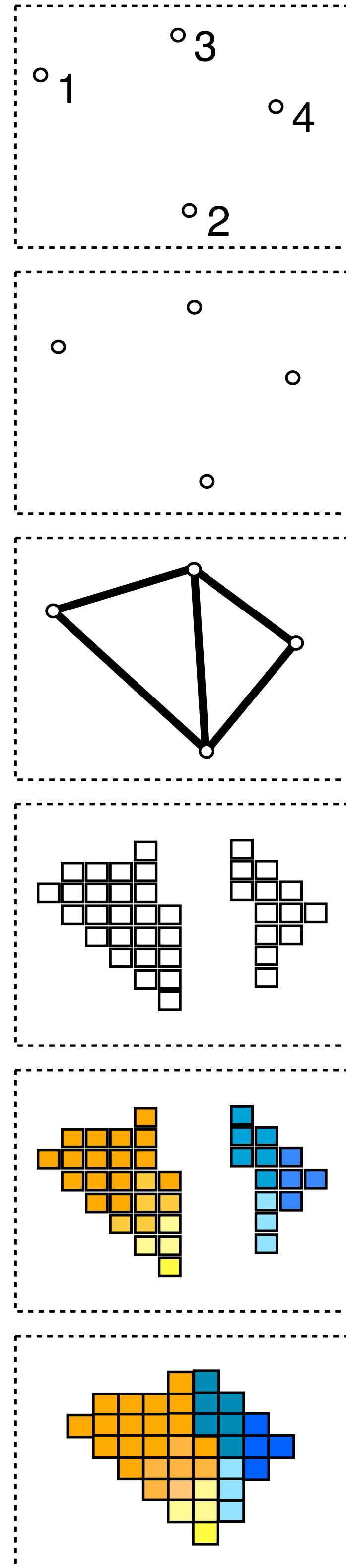
Don't forget to normalize the interpolated directions

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

# Graphics Pipeline

图形管线




**Input: vertices in 3D space**

MVP 发生在 Vertex Processing，将 3d 点映射到 2d

**Vertices positioned in screen space**

**Triangles positioned in screen space**

Sampling triangle 发生在 Rasterization

**Fragments (one per covered sample)**

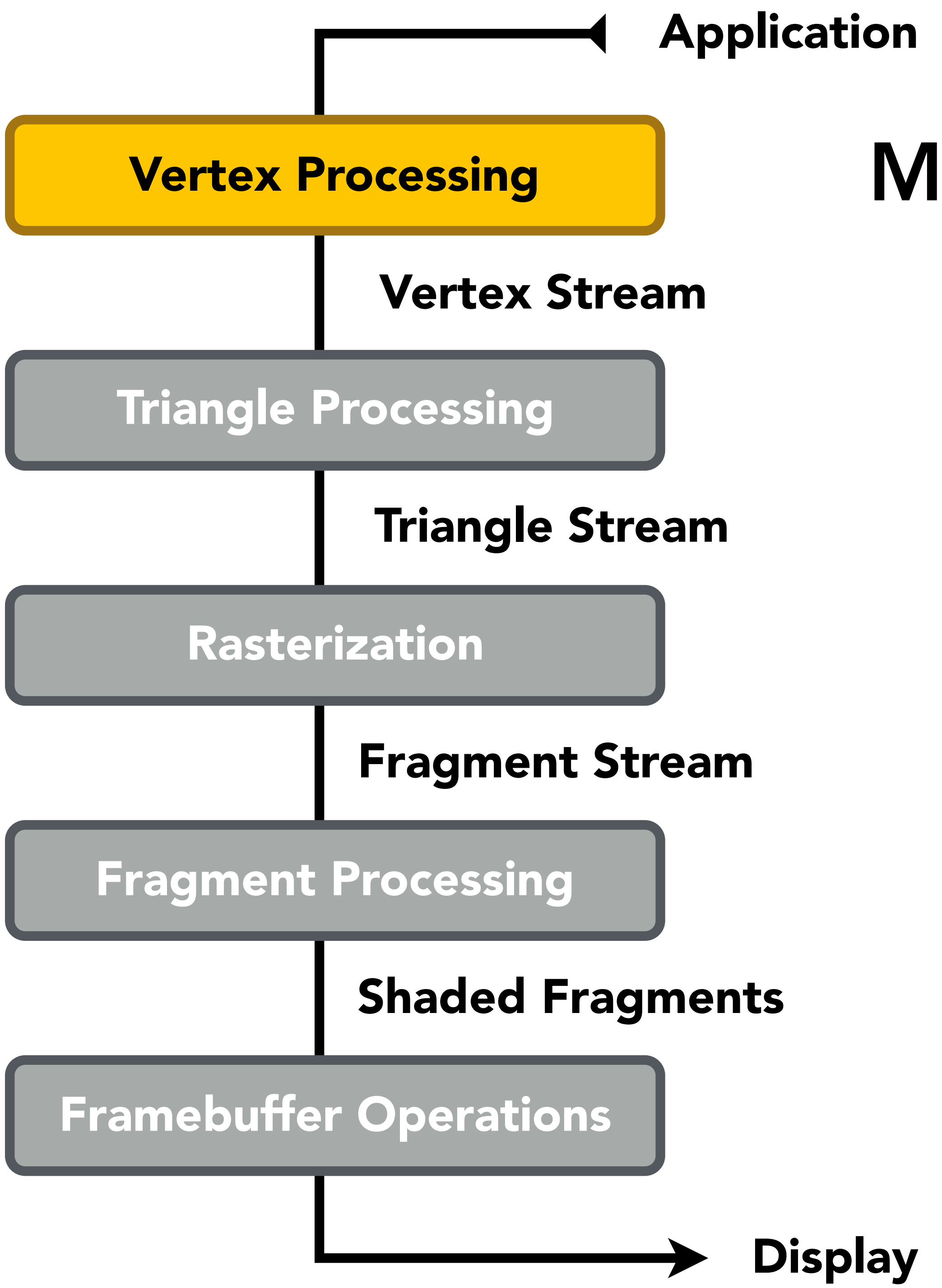
Z-Buffer Visibility Tests 发生在 Fragment Processing

**Shaded fragments**

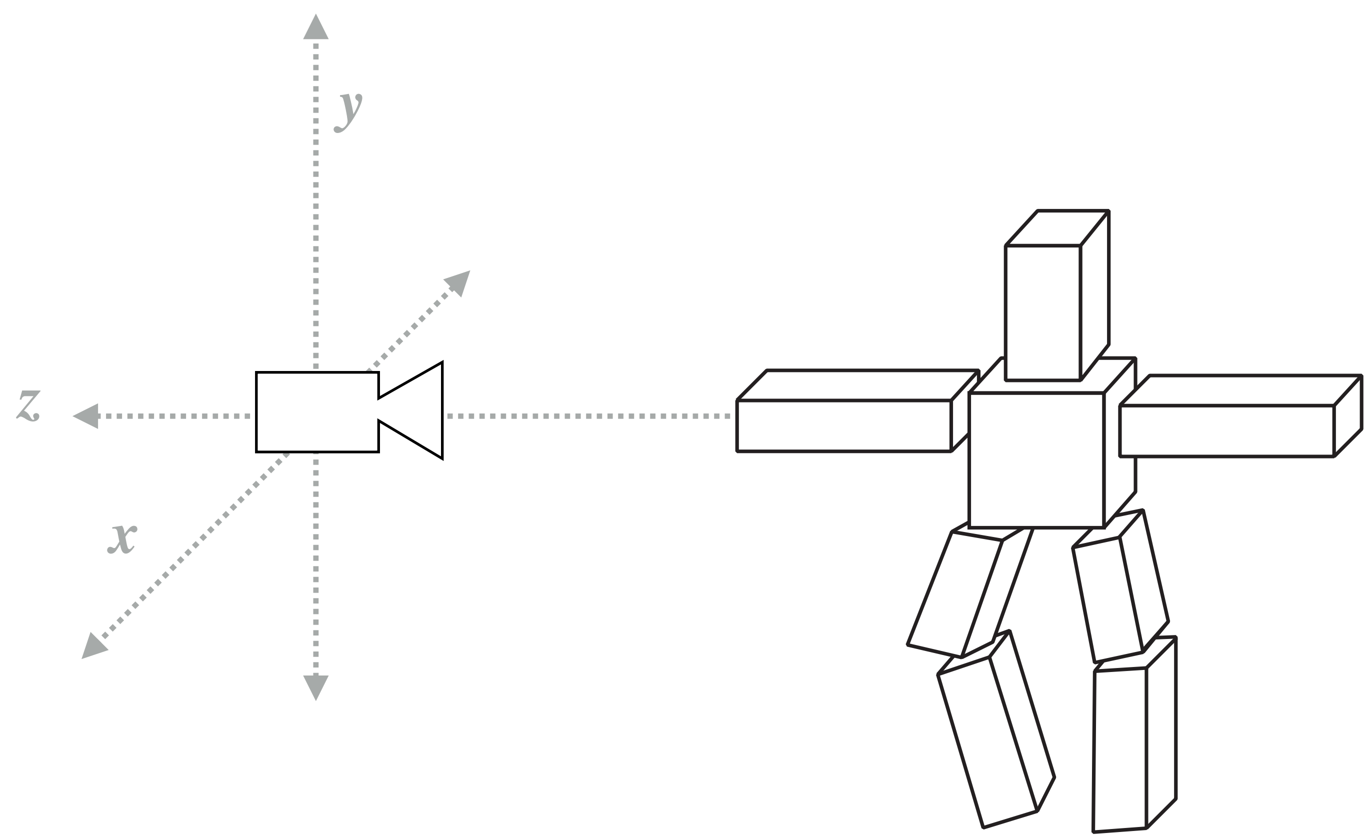
Shading 发生在 Vertex Processing, Fragment Processing
- 顶点着色可以发生在 Vertex Processing
- Phong shading 发生在 Fragment Processing

**Output: image (pixels)**

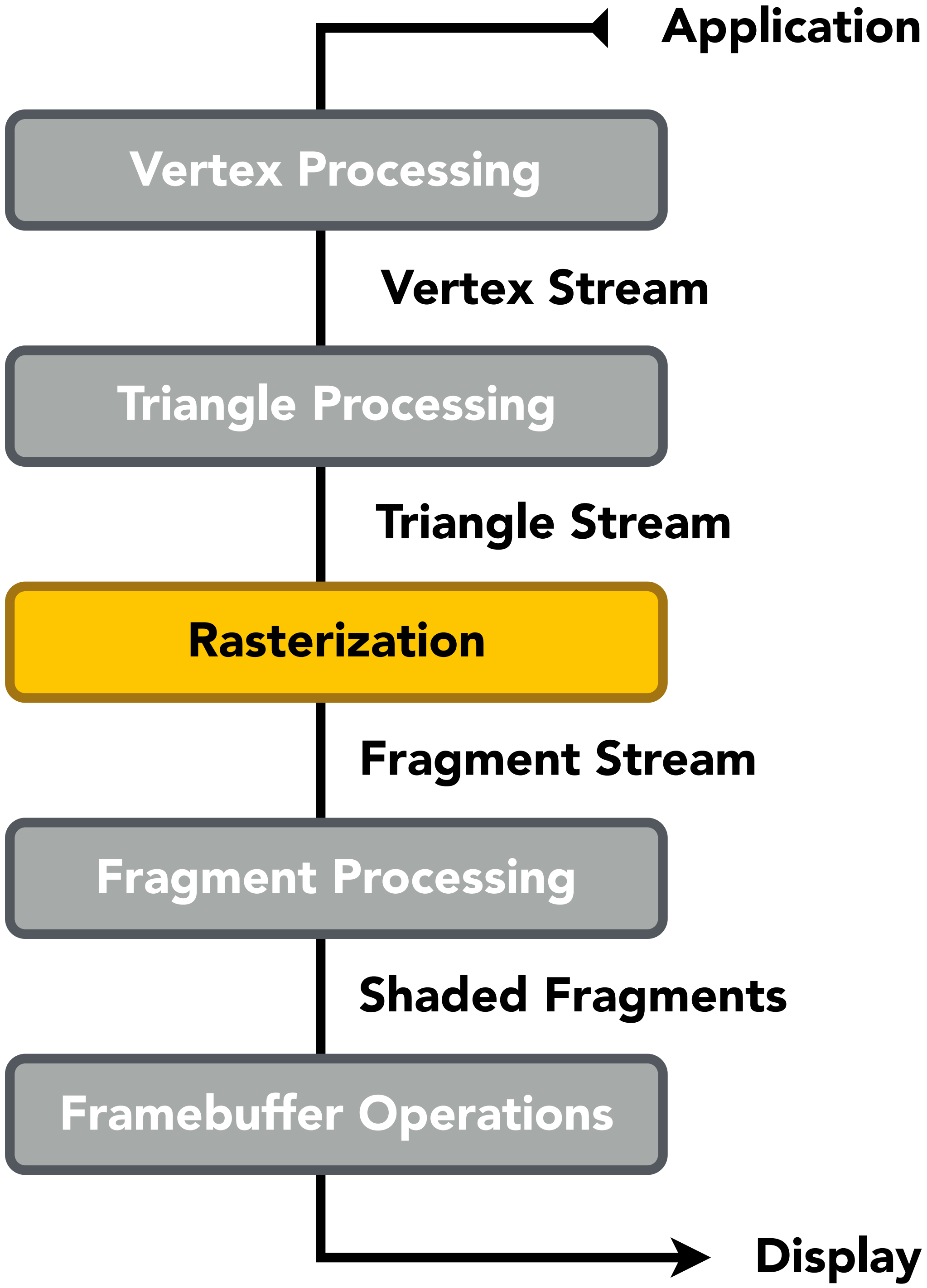
# Graphics Pipeline



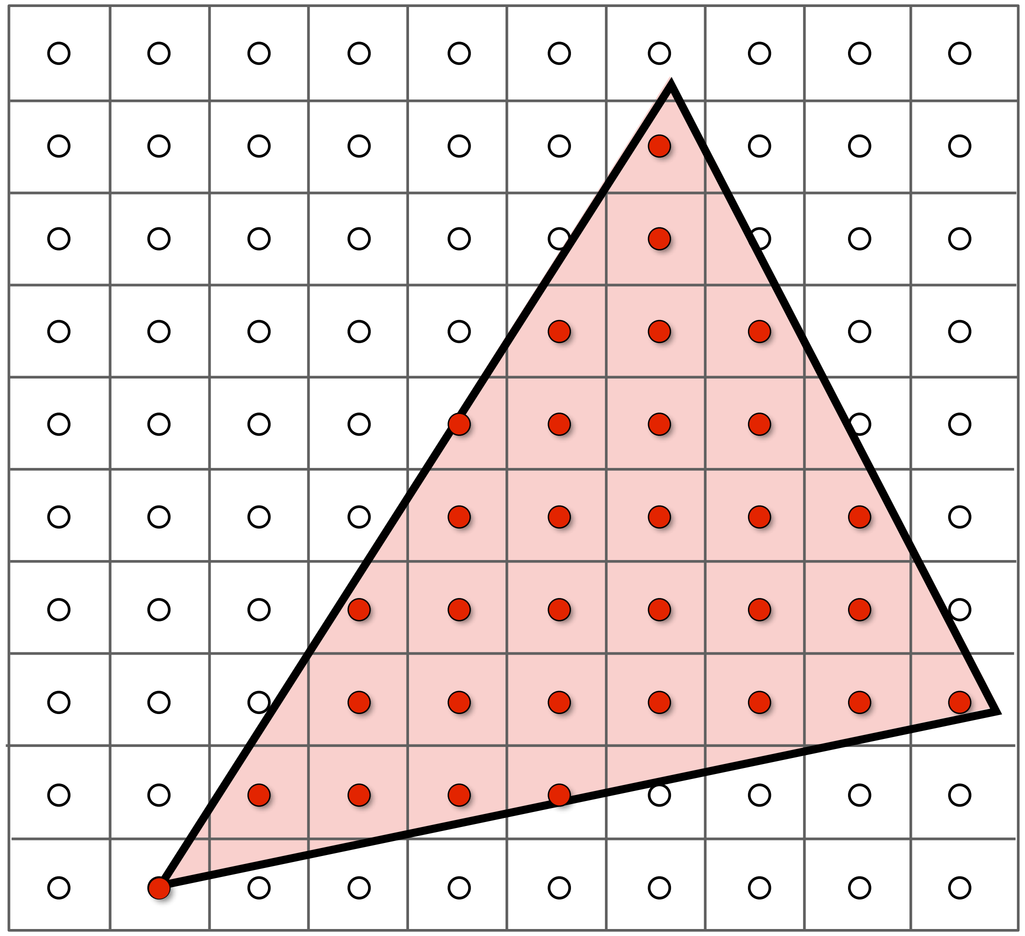

## Model, View, Projection transforms



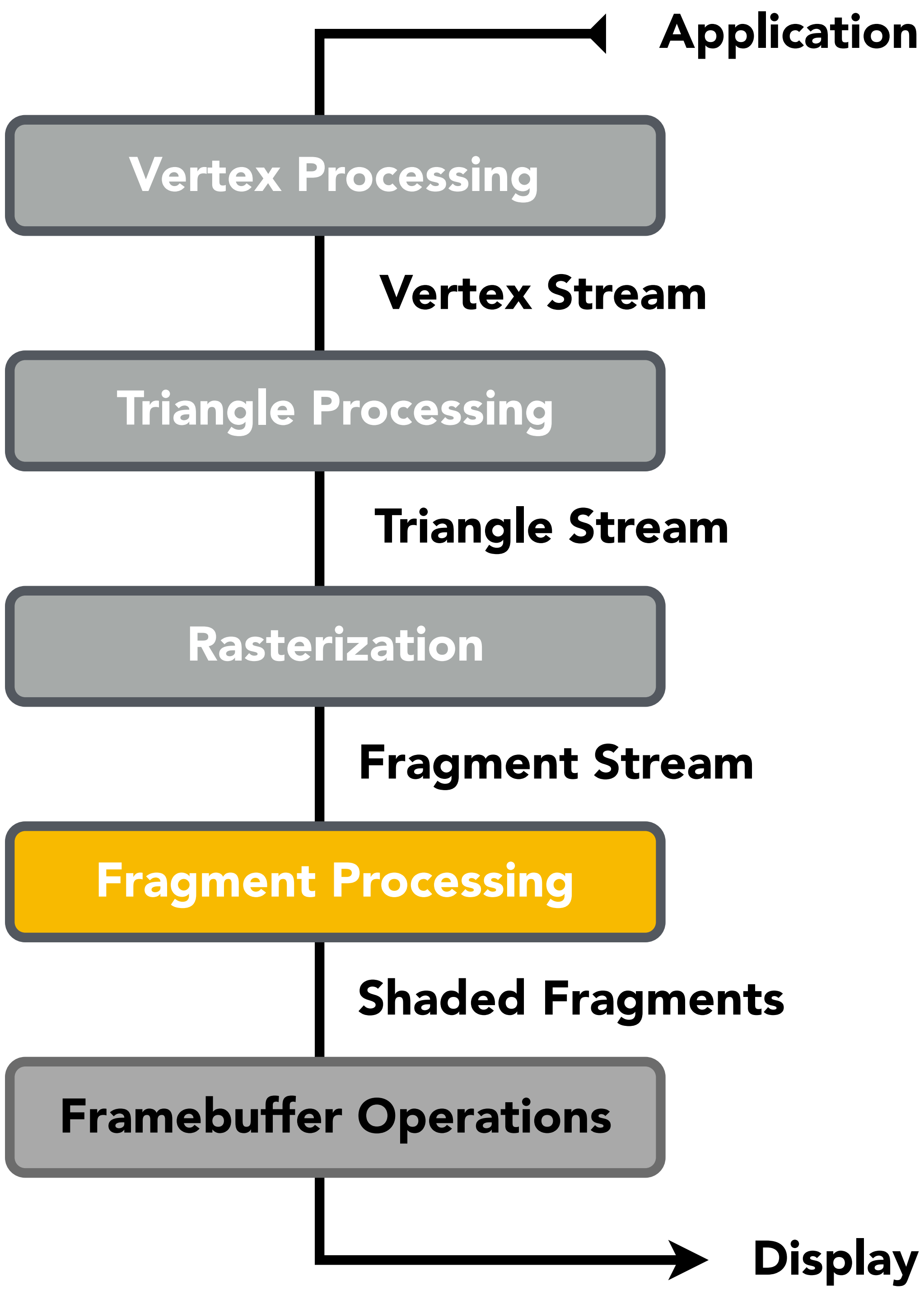
# Graphics Pipeline




## Sampling triangle coverage

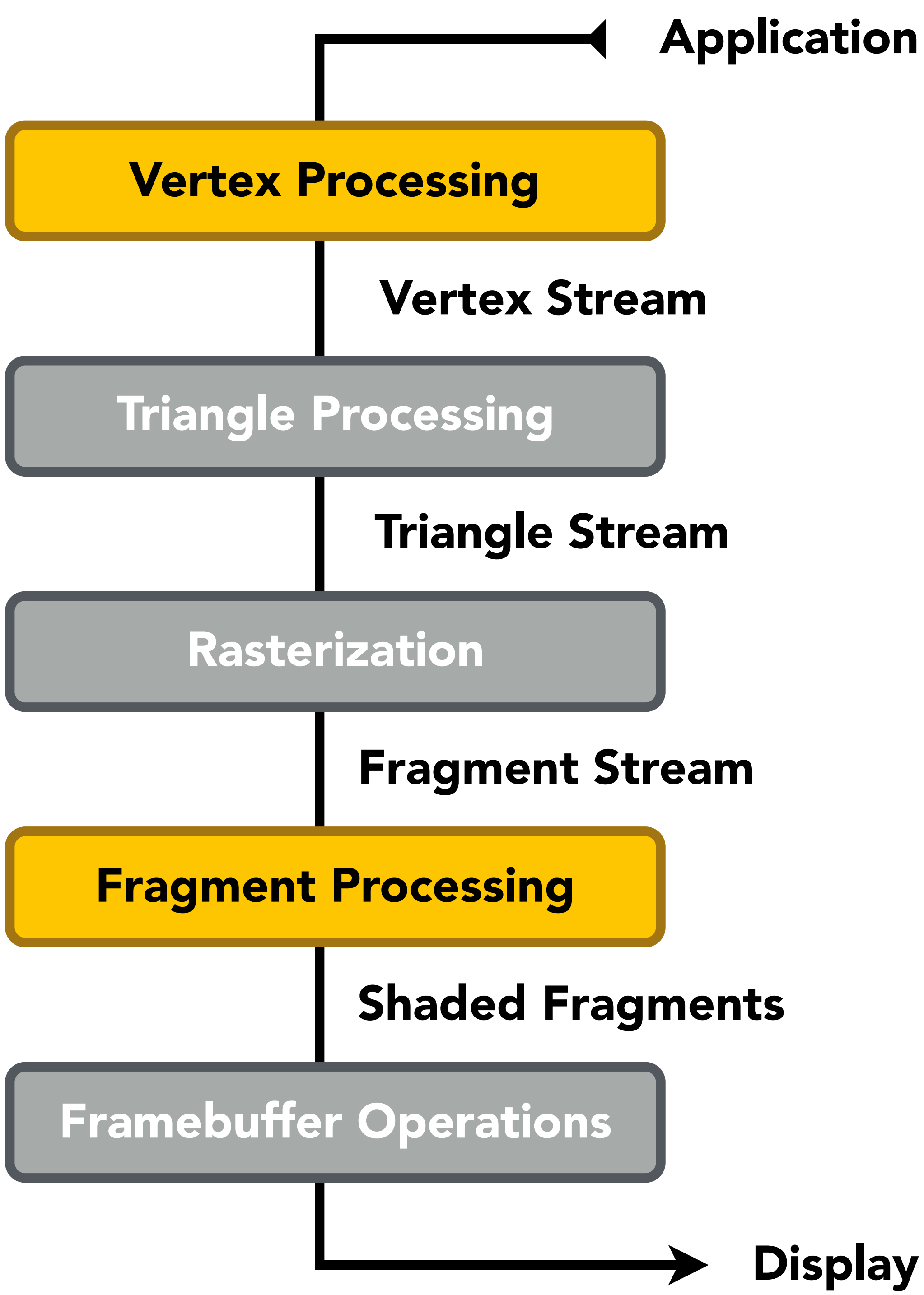


# Rasterization Pipeline

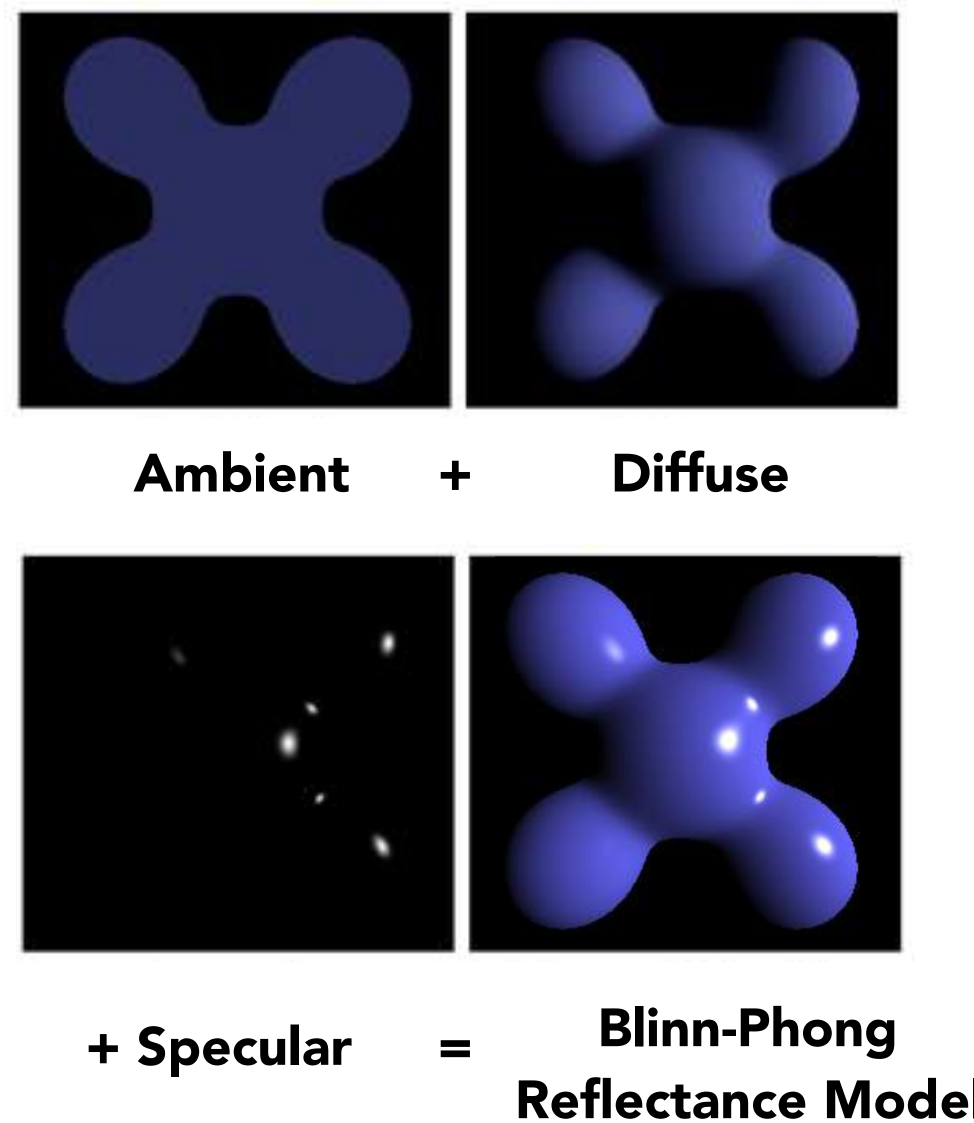



## Z-Buffer Visibility Tests

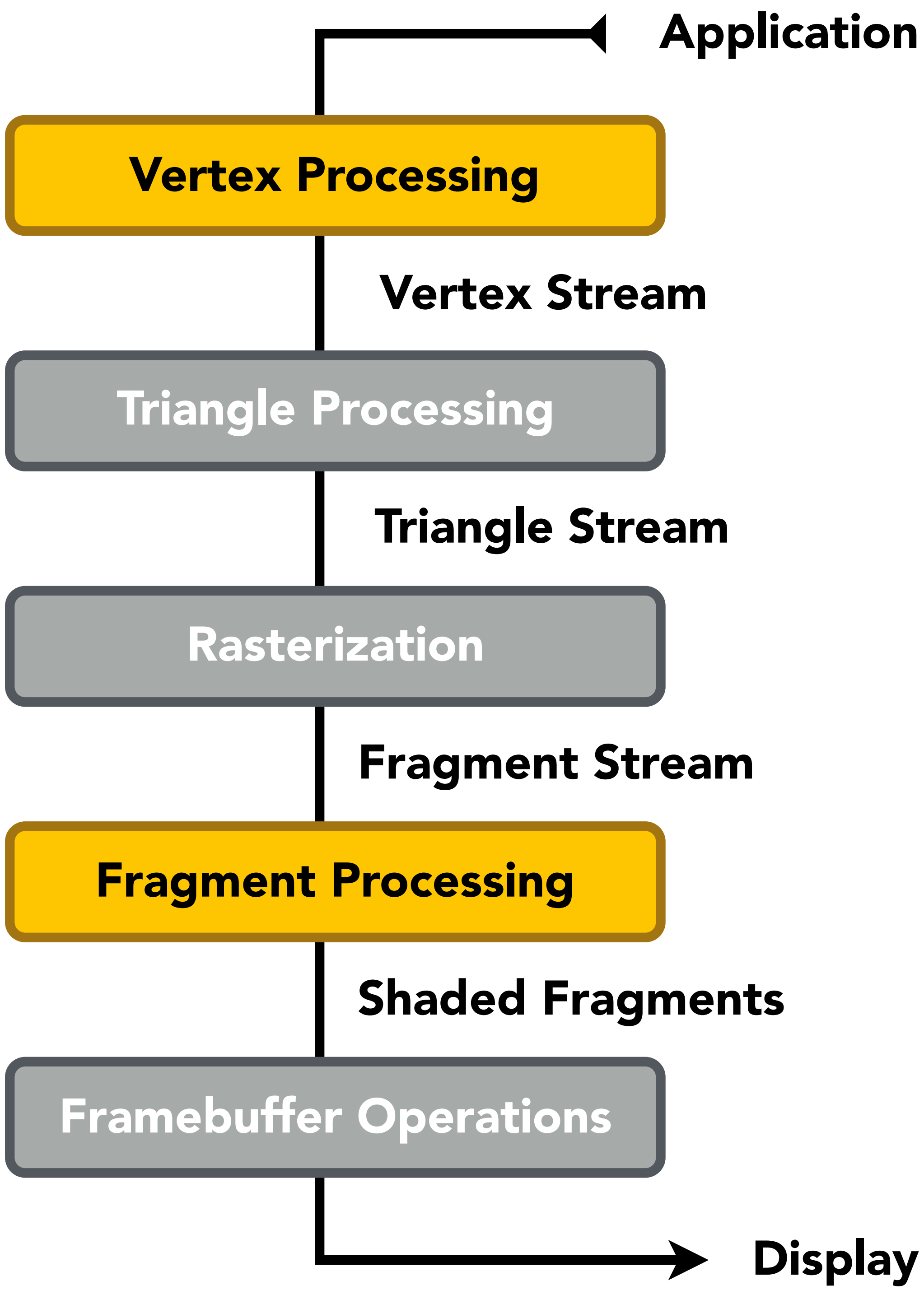
# Graphics Pipeline




## Shading

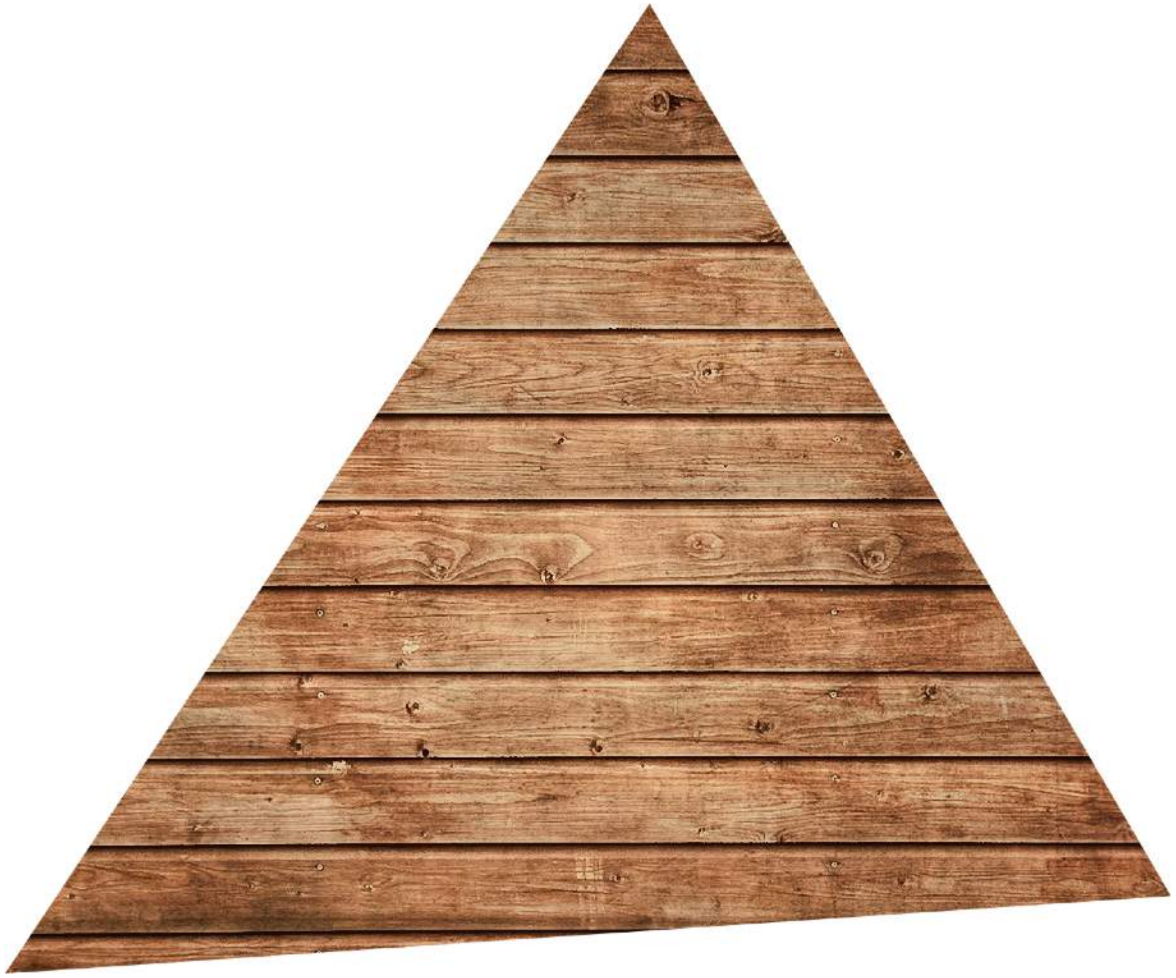


# Graphics Pipeline




**Texture mapping**

**(introducing soon)**



# Shader Programs

shader 编程

GPU 允许编程 vertex and fragment processing 阶段

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

Example GLSL fragment shader program

Shader 中只用考虑一个顶点或一个像素是如何运作的。

```
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

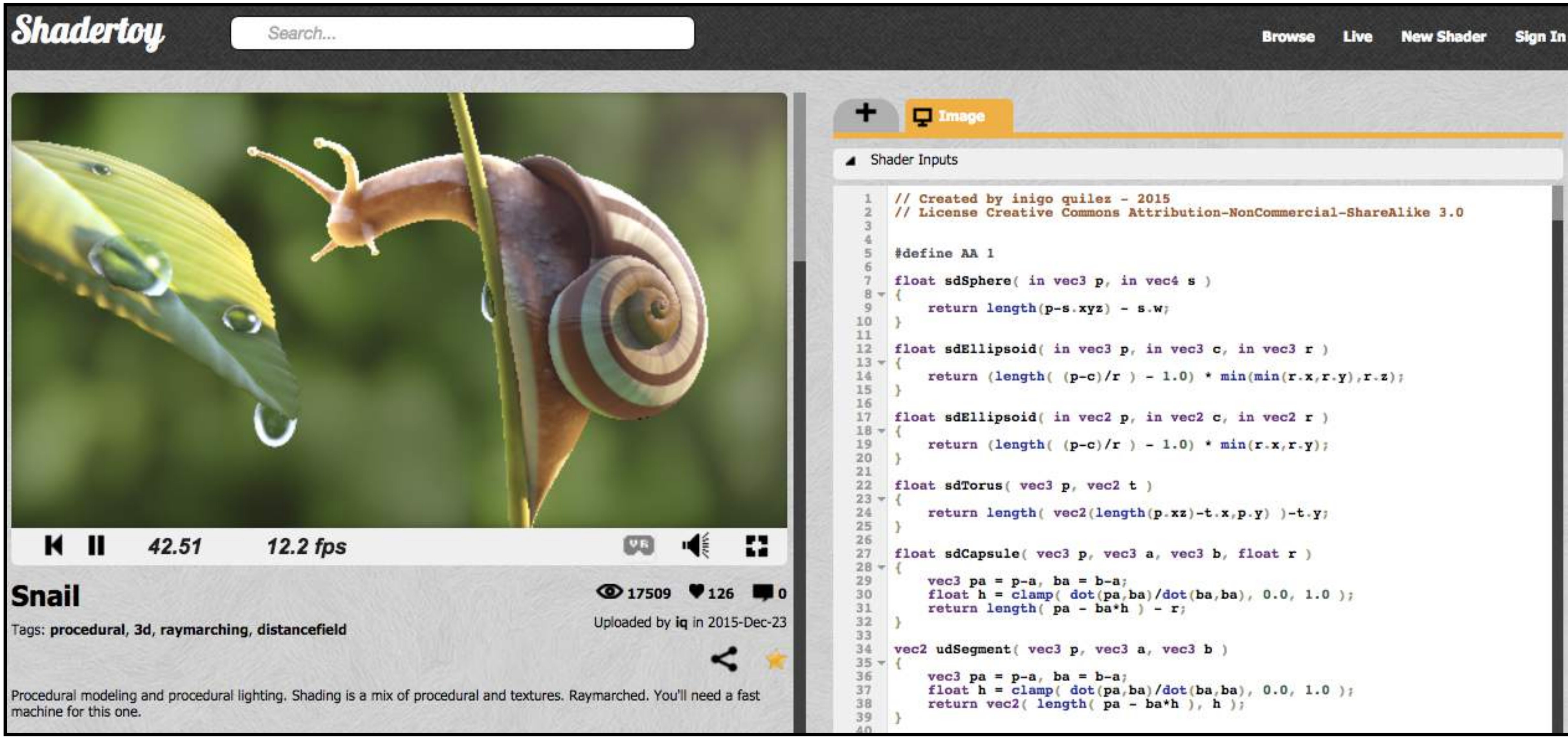
- Program vertex and fragment processing stages
- Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

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

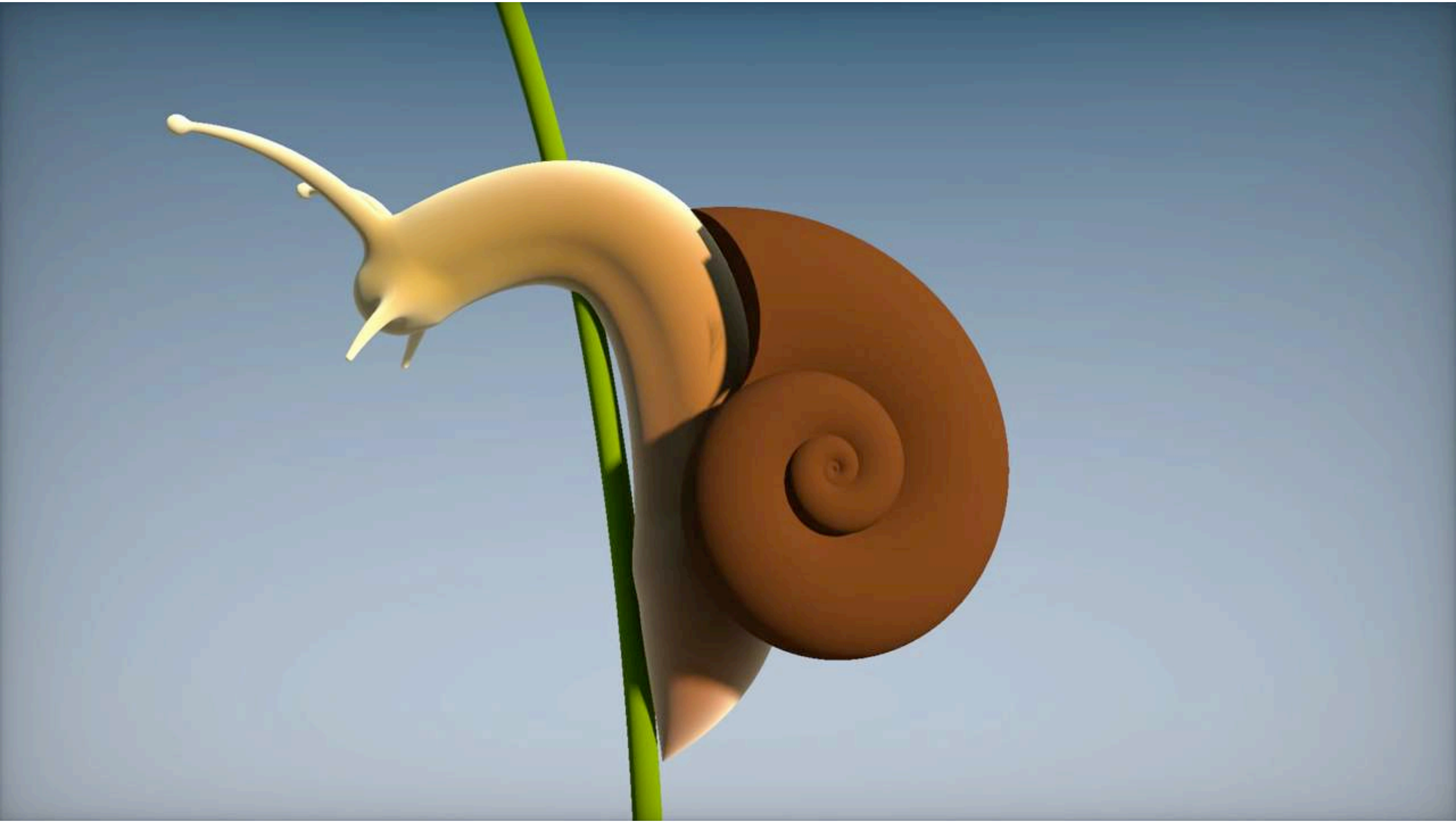
# Snail Shader Program




## 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

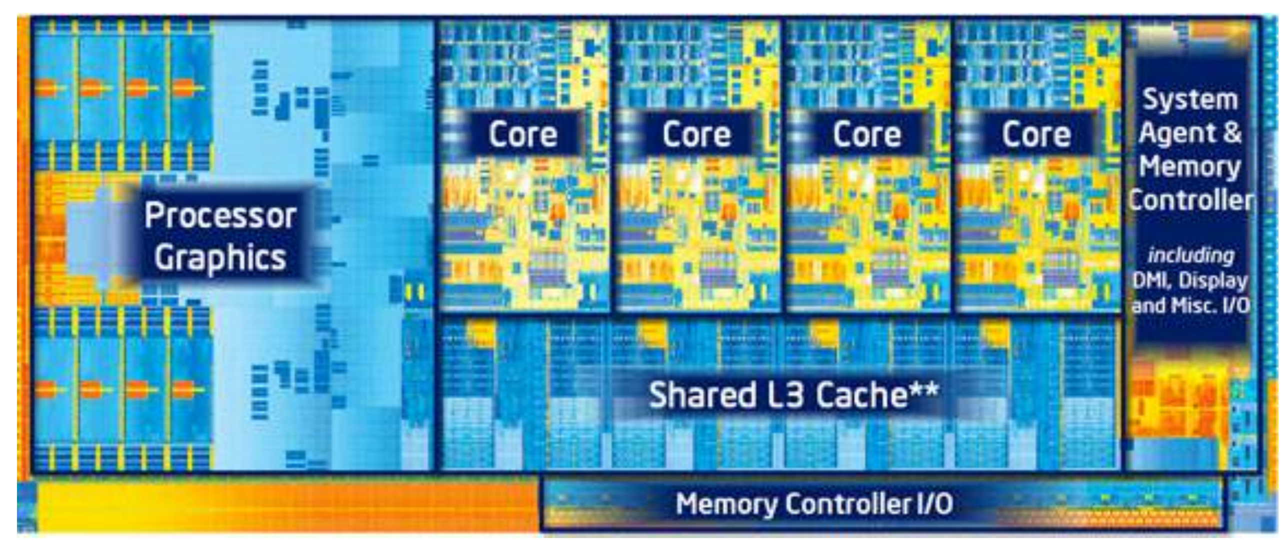
- **100's of thousands to millions of triangles in a scene**
- **Complex vertex and fragment shader computations**
- **High resolution (2-4 megapixel + supersampling)**
- **30-60 frames per second (even higher for VR)**

Unreal Engine Kite Demo (Epic Games 2015)

# Graphics Pipeline Implementation: GPUs

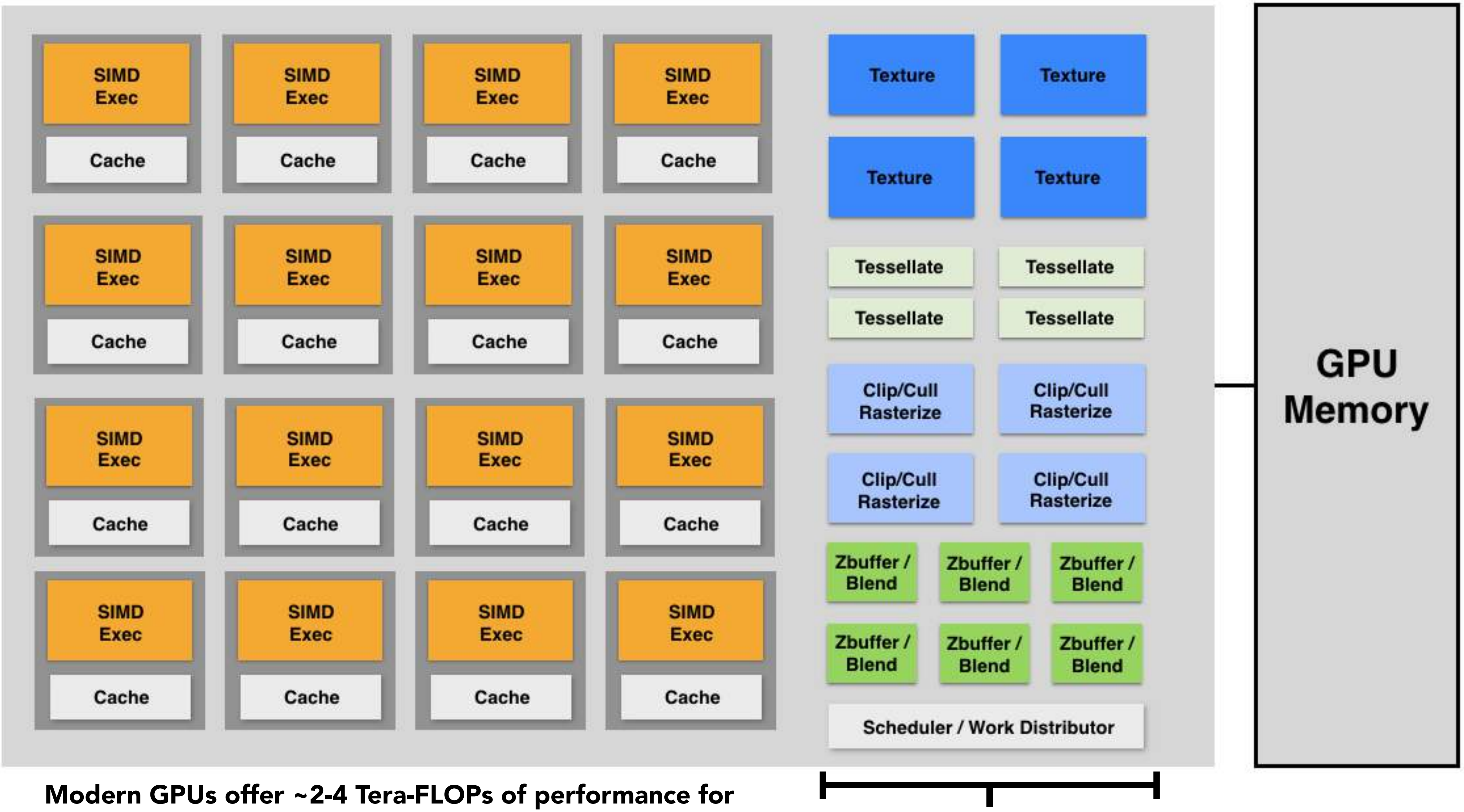
**Specialized processors for executing graphics pipeline computations**


**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**

# Texture Mapping

# Different Colors at Different Places?


Pattern on ball

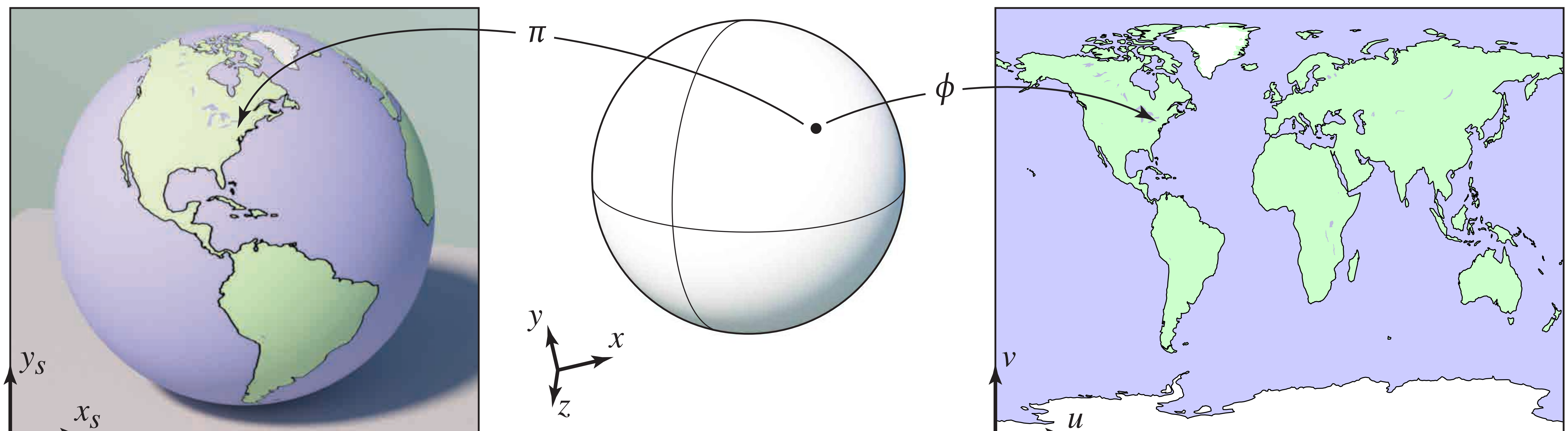
Wood grain on floor

# Surfaces are 2D

Surface lives in 3D world space

将三维世界坐标平铺后展开到2维坐标形式，展开后的每三个点形成的三角形就是一个纹理

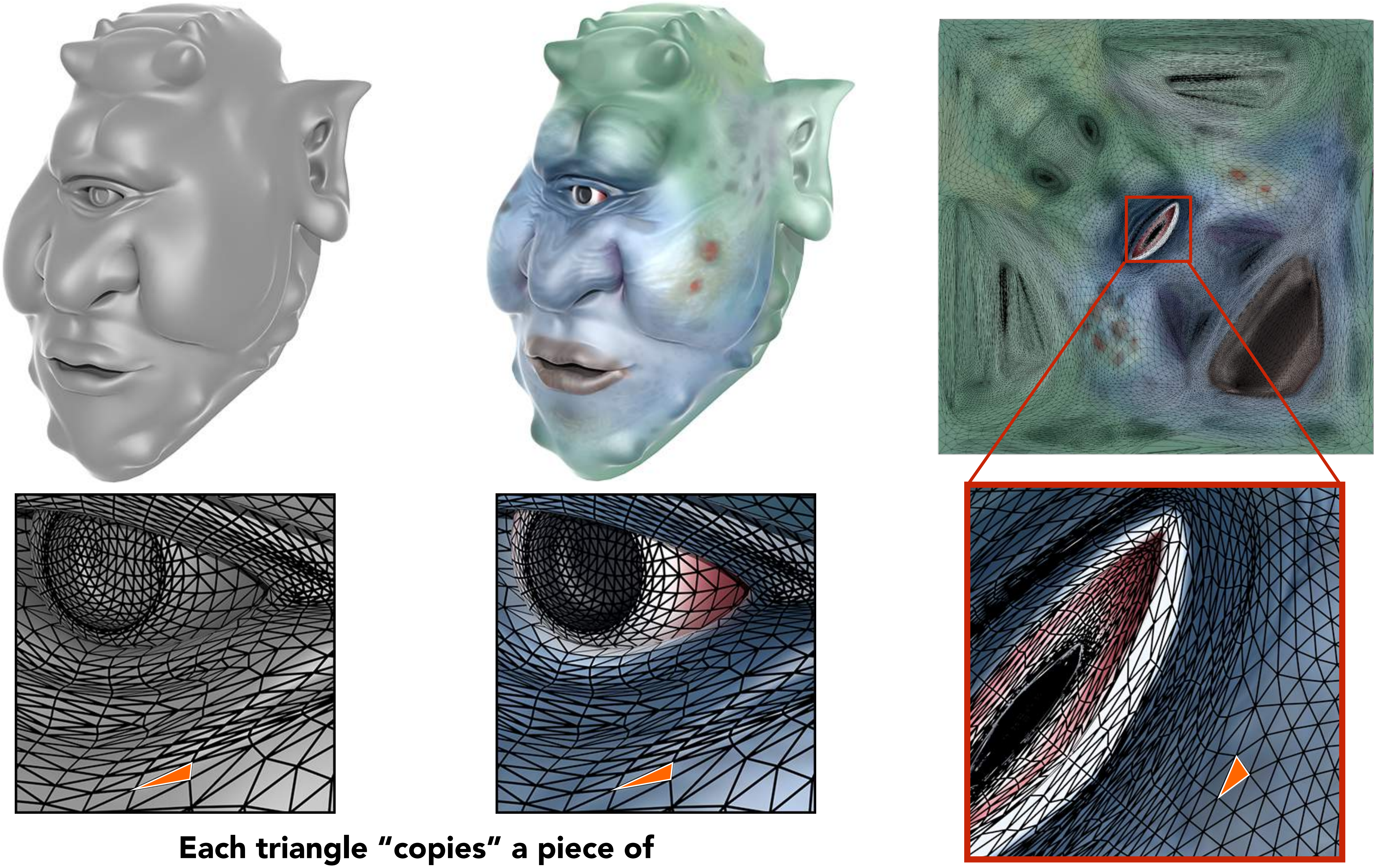
Every 3D surface point also has a place where it goes in the 2D image (**texture**).




Screen space　　World space　　Texture space

# Texture Applied to Surface

**Rendering without texture** **Rendering with texture** **Texture**



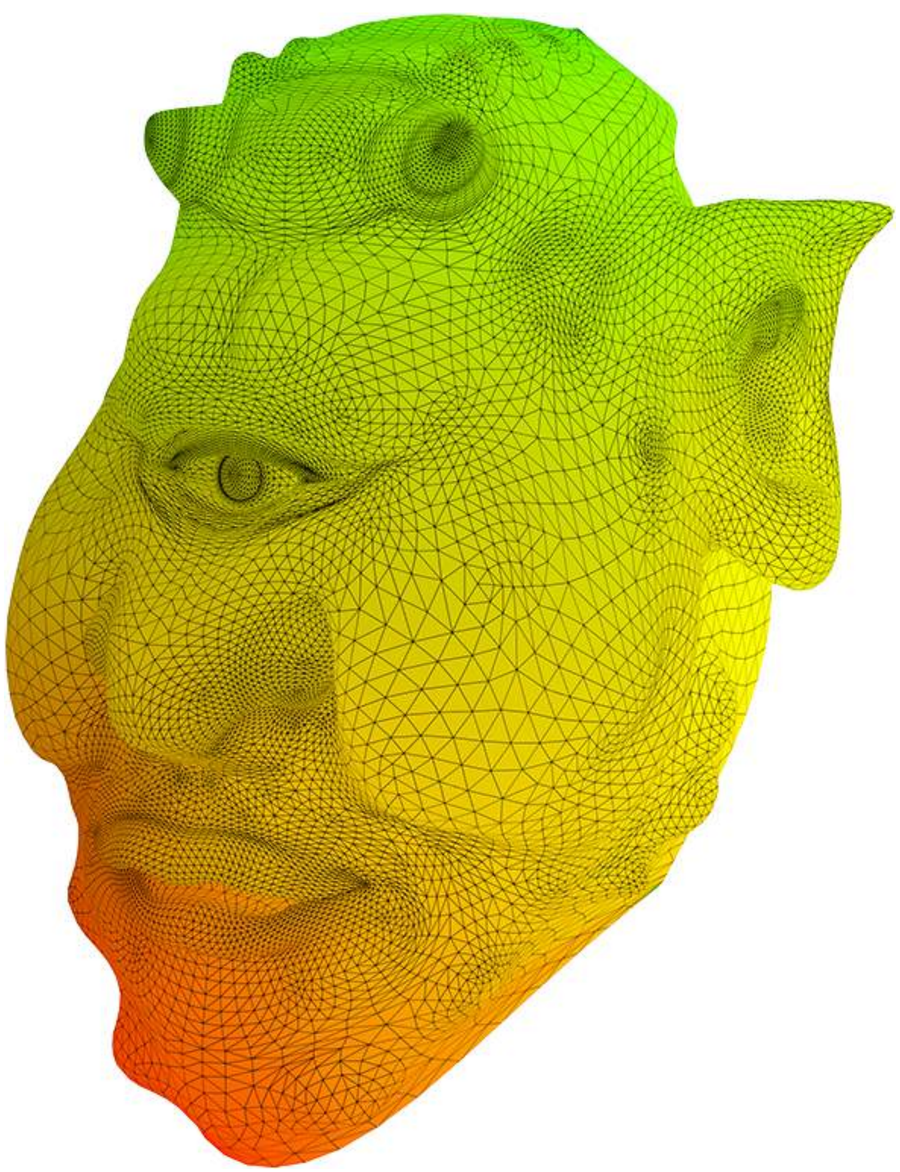
**Zoom**

**Each triangle "copies" a piece of the texture image to the surface.**

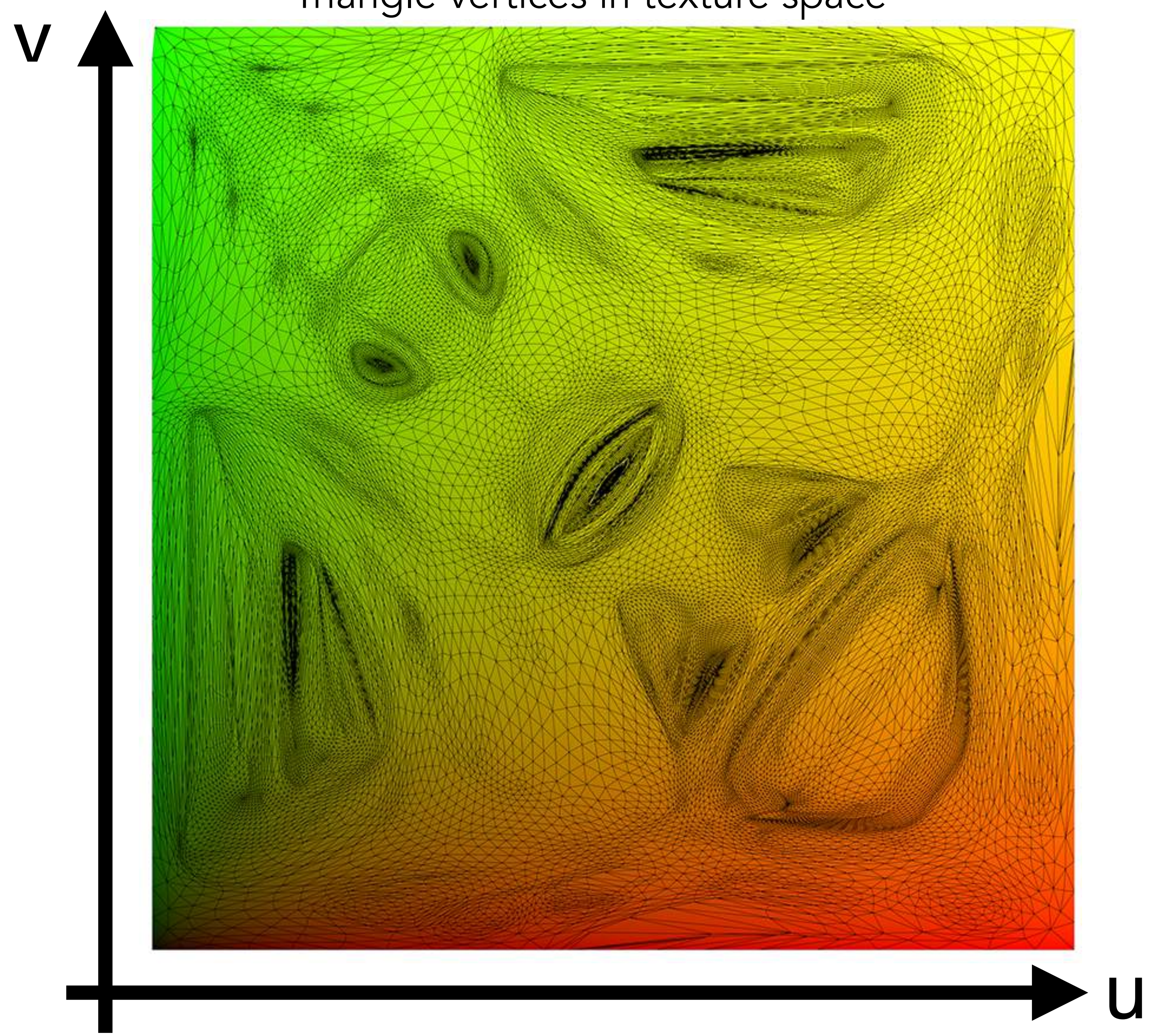
# Visualization of Texture Coordinates

Each triangle vertex is assigned a texture coordinate (u,v)

Visualization of texture coordinates



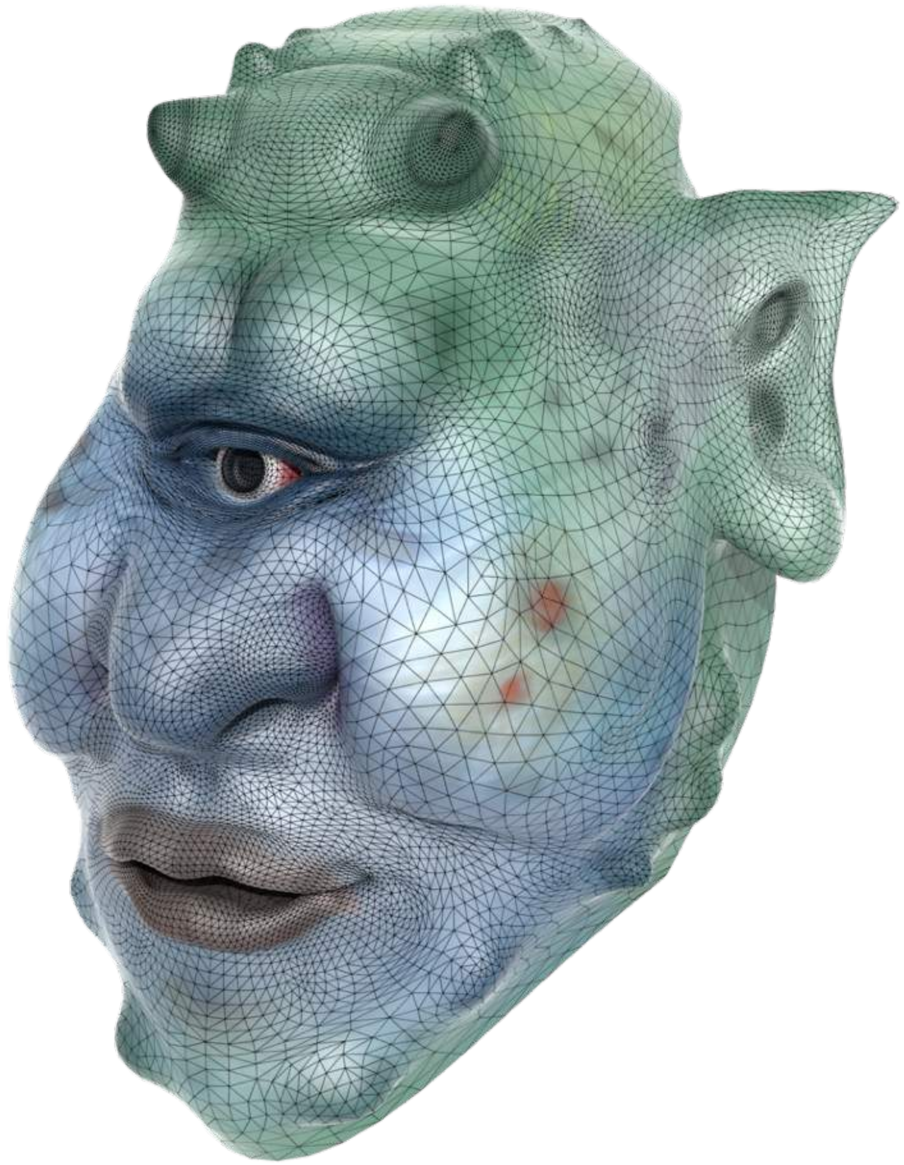
Triangle vertices in texture space



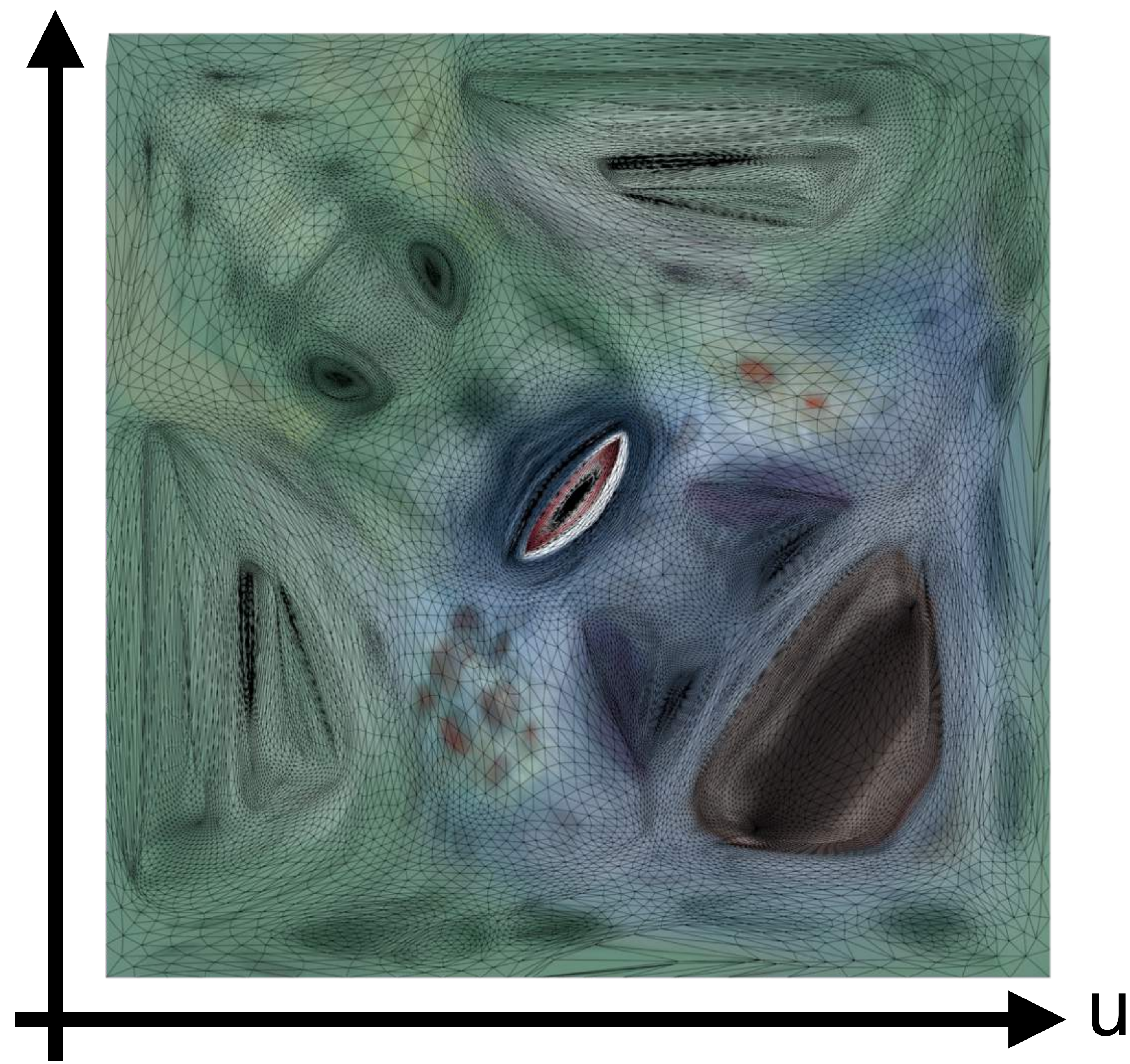

3d 表面的每个三角形在 2d 图上都对应一个纹理坐标

# Texture Applied to Surface

Rendered result



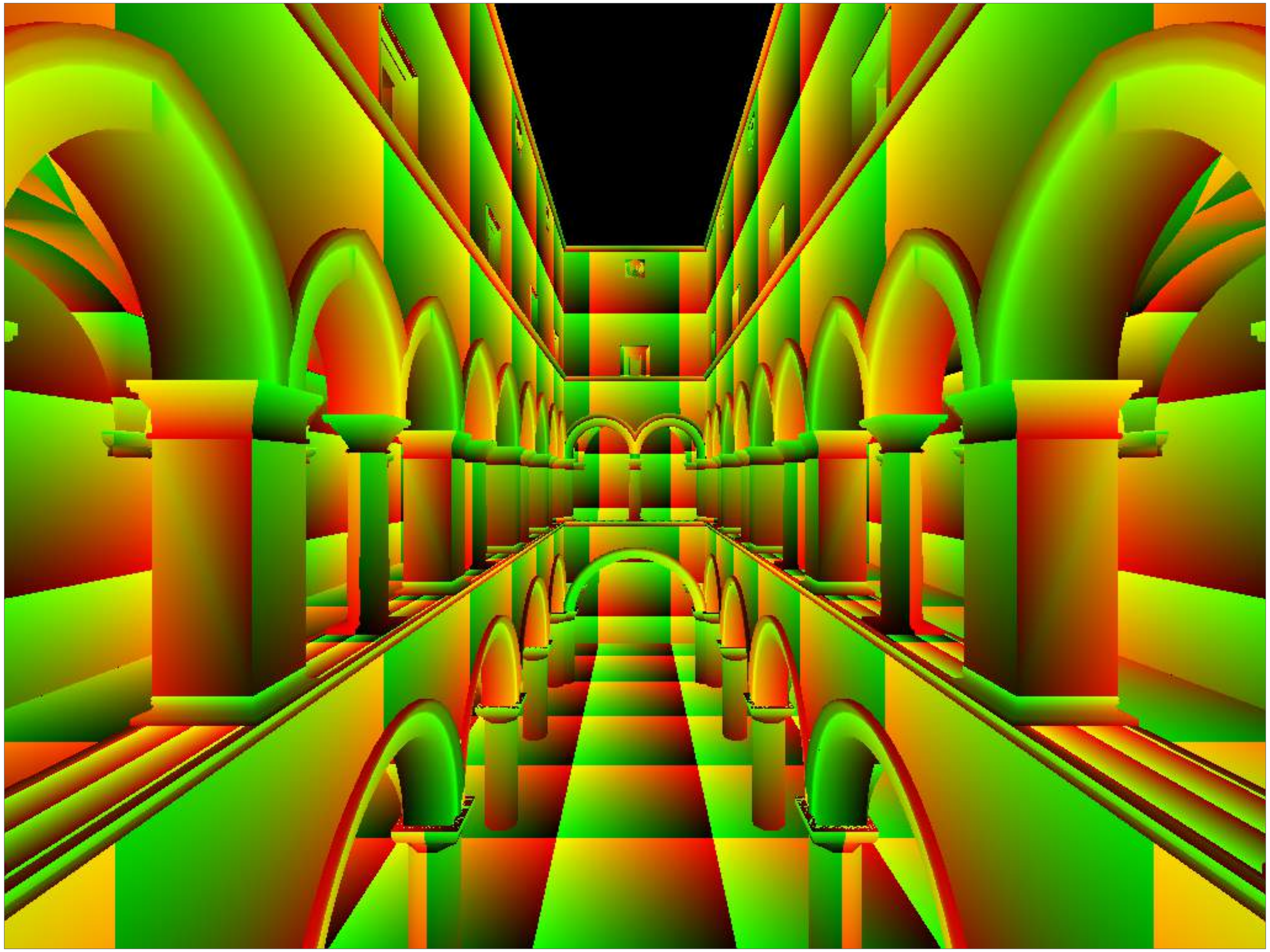
Triangle vertices in texture space



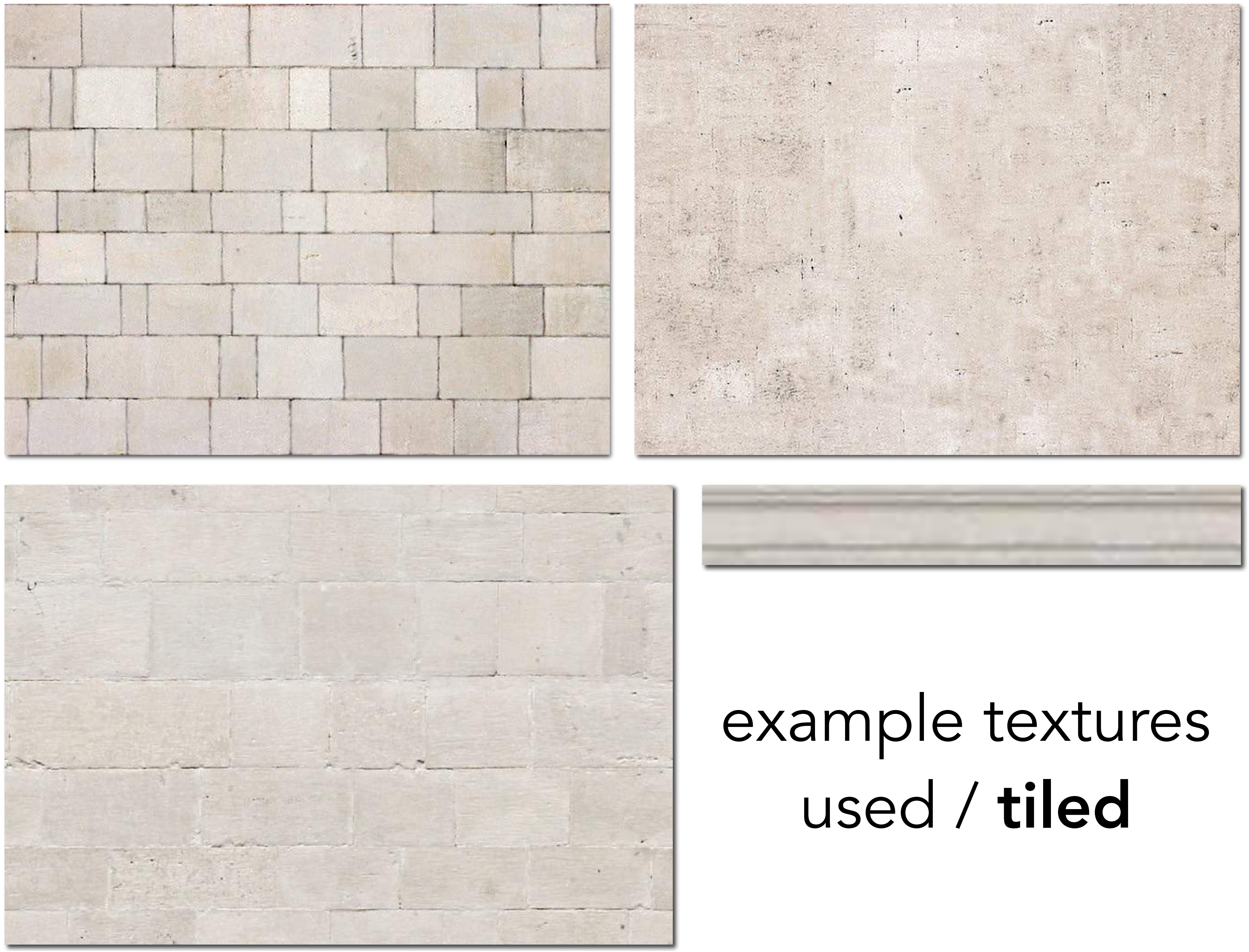

三维图形的每个三角面顶点都可以对应一个uv坐标系下的坐标，uv坐标范围约定在 [0,1]之间

# Textures applied to surfaces

# Visualization of texture coordinates



# Textures can be used multiple times!



example textures
used / **tiled**

# Thank you!

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