Introduction à l'Informatique Graphique

Lecture 5. Rendering

Caroline Larboulette

Illumination

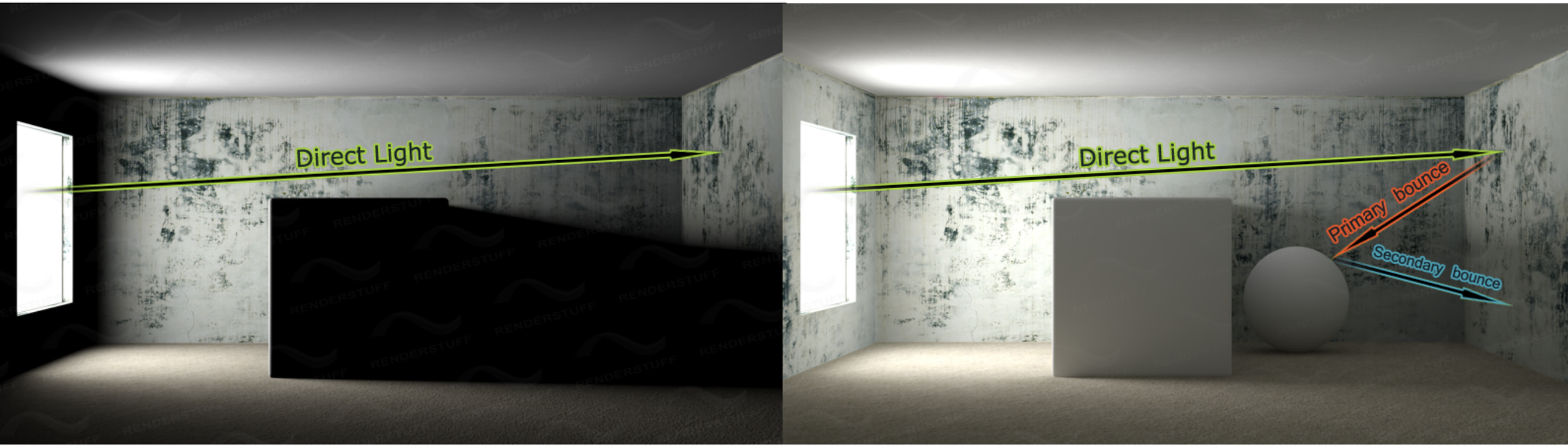
Global versus Local Models

- Local Illumination:
  - Only considers interactions between light sources and object surfaces (direct illumination + ambient)

- Global Illumination:
  - Also considers energy exchanges between the surfaces: light bounces from object to object before reaching the eye

Illumination

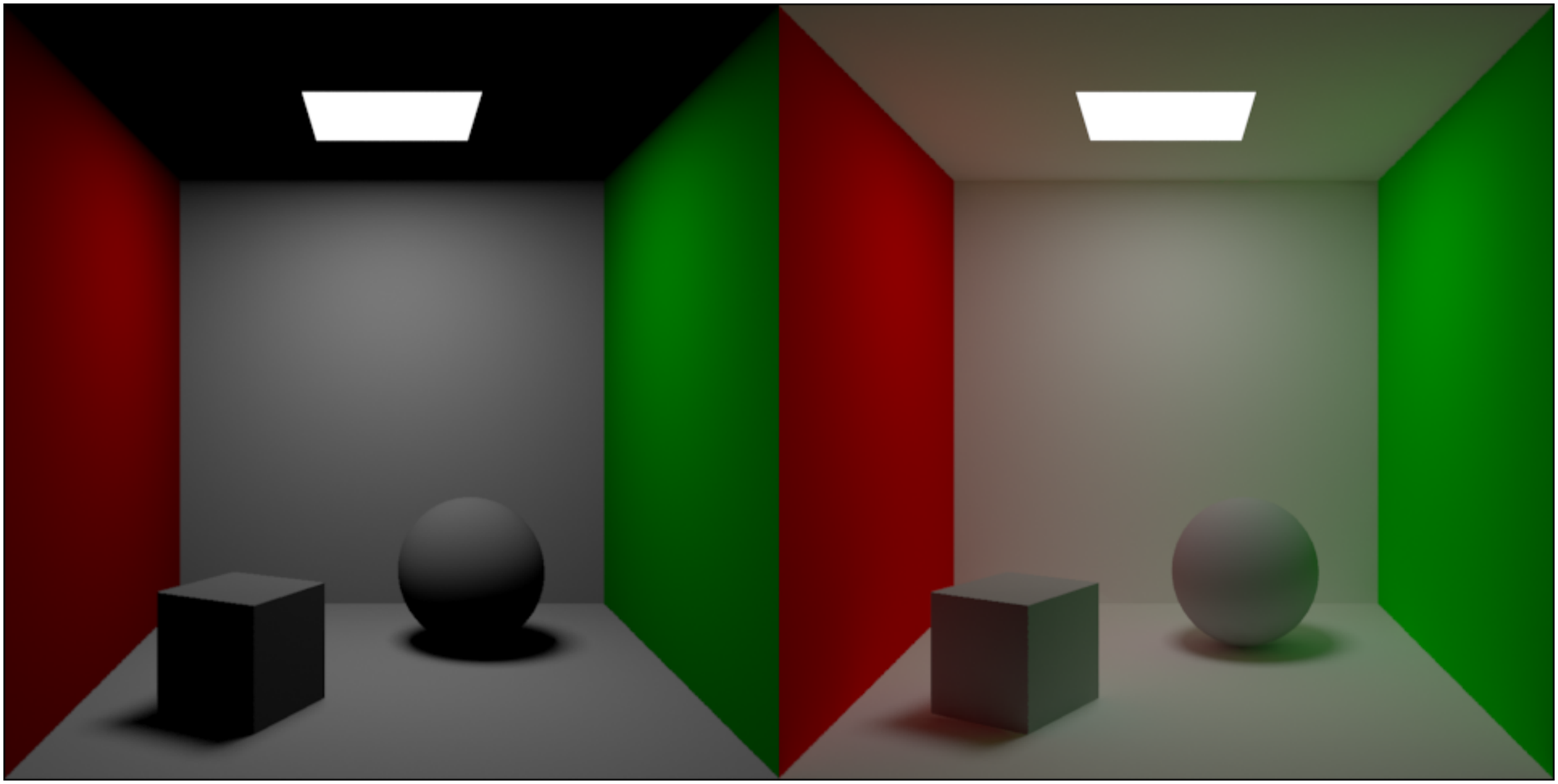
Global versus Local Models



Local **Global**

Illumination

Global versus Local Models



Local **Global**

Illumination

Global versus Local Models

- Local Illumination:
  - Only considers interactions between light sources and object surfaces (direct illumination + ambient)

- Global Illumination:
  - Also considers energy exchanges between the surfaces: light bounces from object to object before reaching the eye

Local Illumination

Real-Time Rendering

- Bibliography: Real-Time Rendering, T. Akenine-Moeller & E. Haines, A. K. Peters (most images from this lecture are taken from this book)

Overview

1. Rendering Pipeline
2. Light Sources
3. Material
4. Lighting
5. Shading
6. Textures
7. Shaders (GLSL)

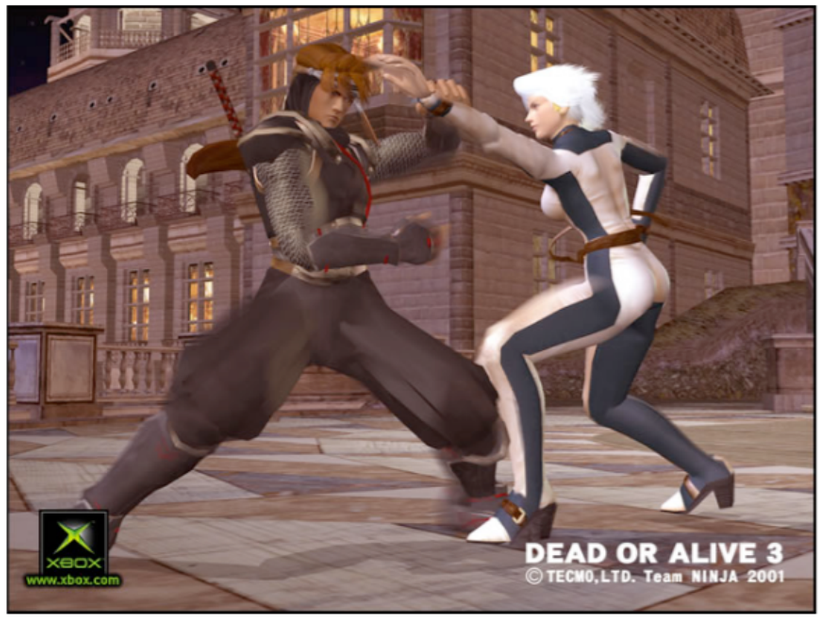
1. Rendering Pipeline

Evolution of Rendering

Virtua Fighter
(SEGA Corporation)

NV1
50K triangles/sec
1M pixel ops/sec

1995



Dead or Alive 3
(Tecmo Corporation)

Xbox (NV2A)
100M triangles/sec
1G pixel ops/sec

2001

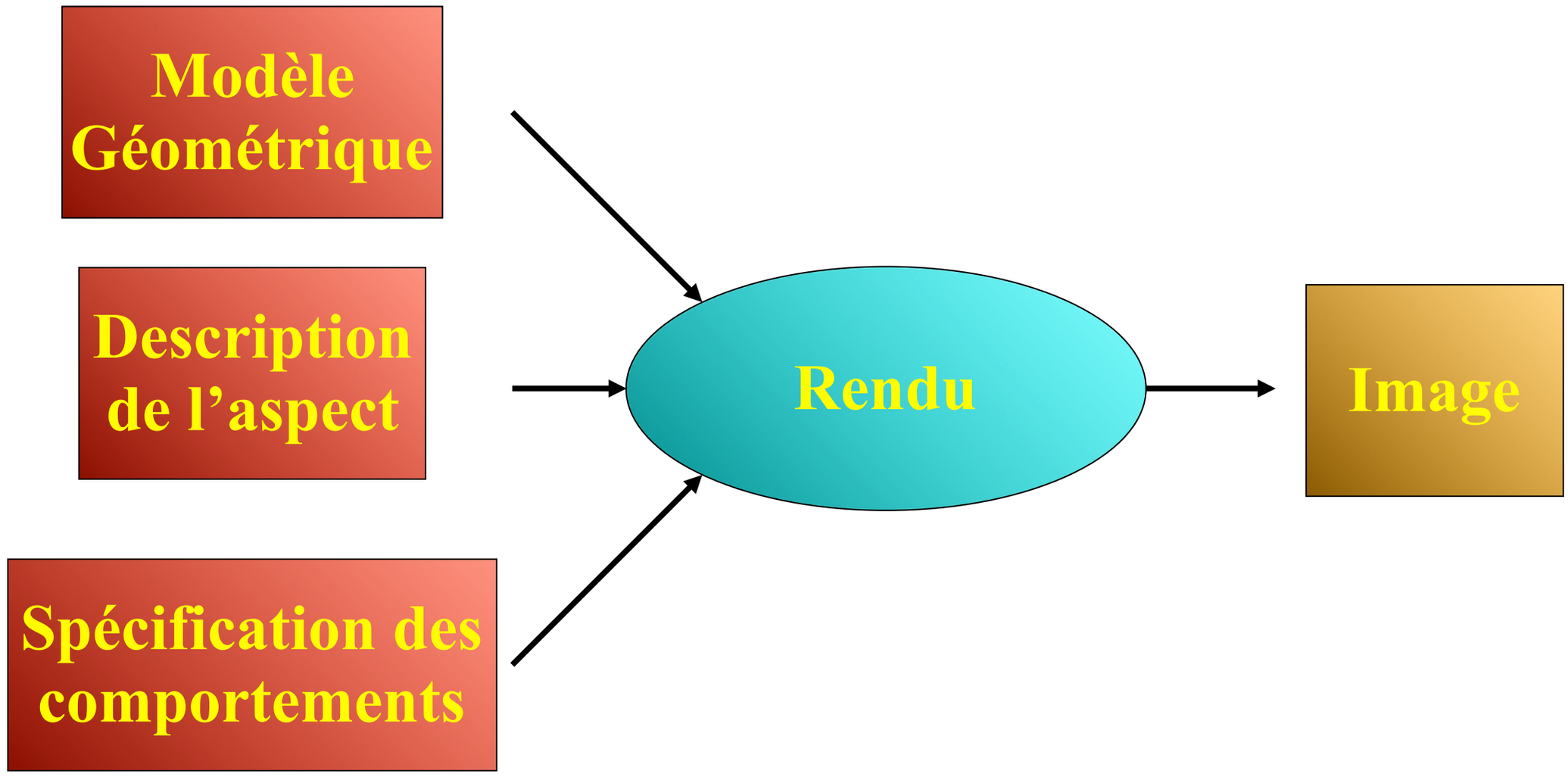
Dawn
(NVIDIA Corporation)

GeForce FX (NV30)
200M triangles/sec
2G pixel ops/sec

2003

Synthèse d'Images 3D

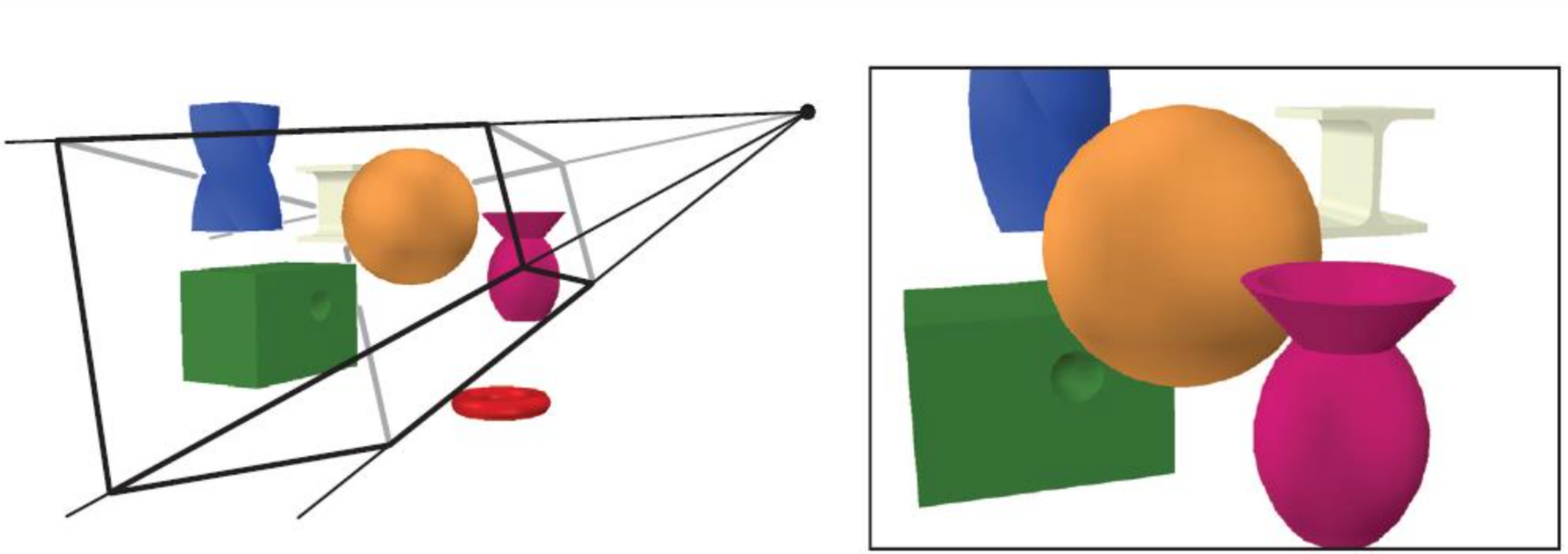
- Passage d'une modélisation tridimensionnelle à une représentation à l'écran



Rendering

The process of generating an image given

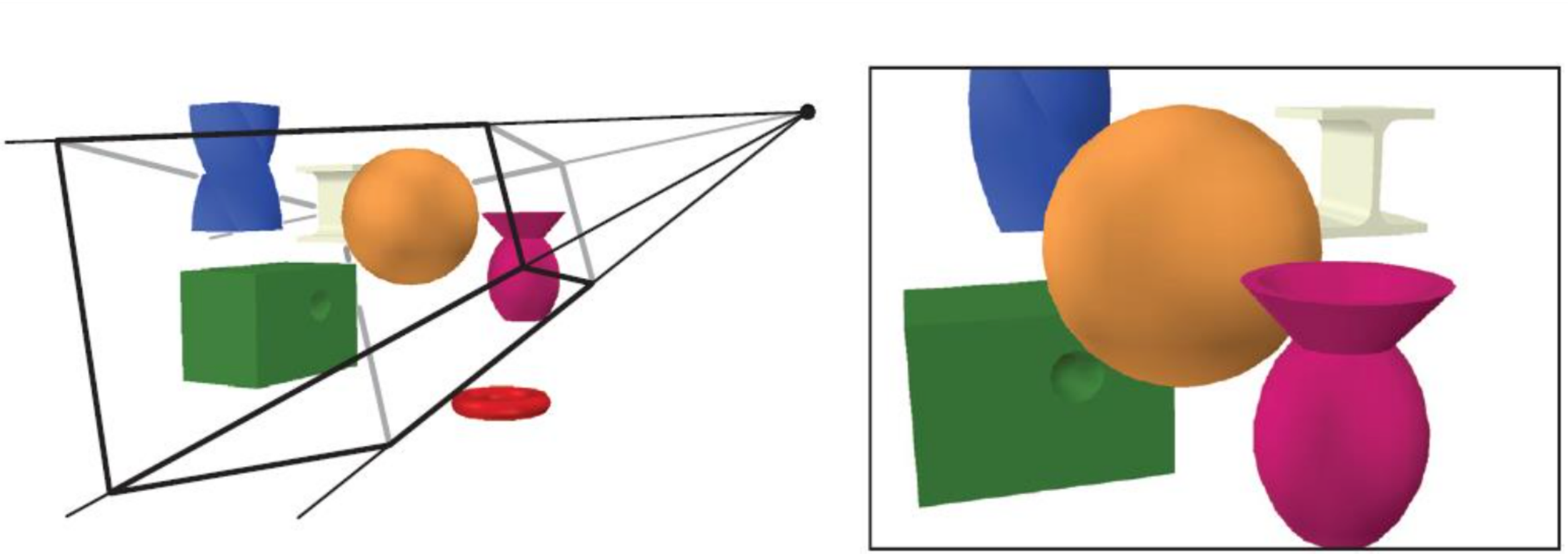
- Objects
- Light sources
- A virtual camera



[Akenine-Moeller et al.: Real-time Rendering]

Rendering

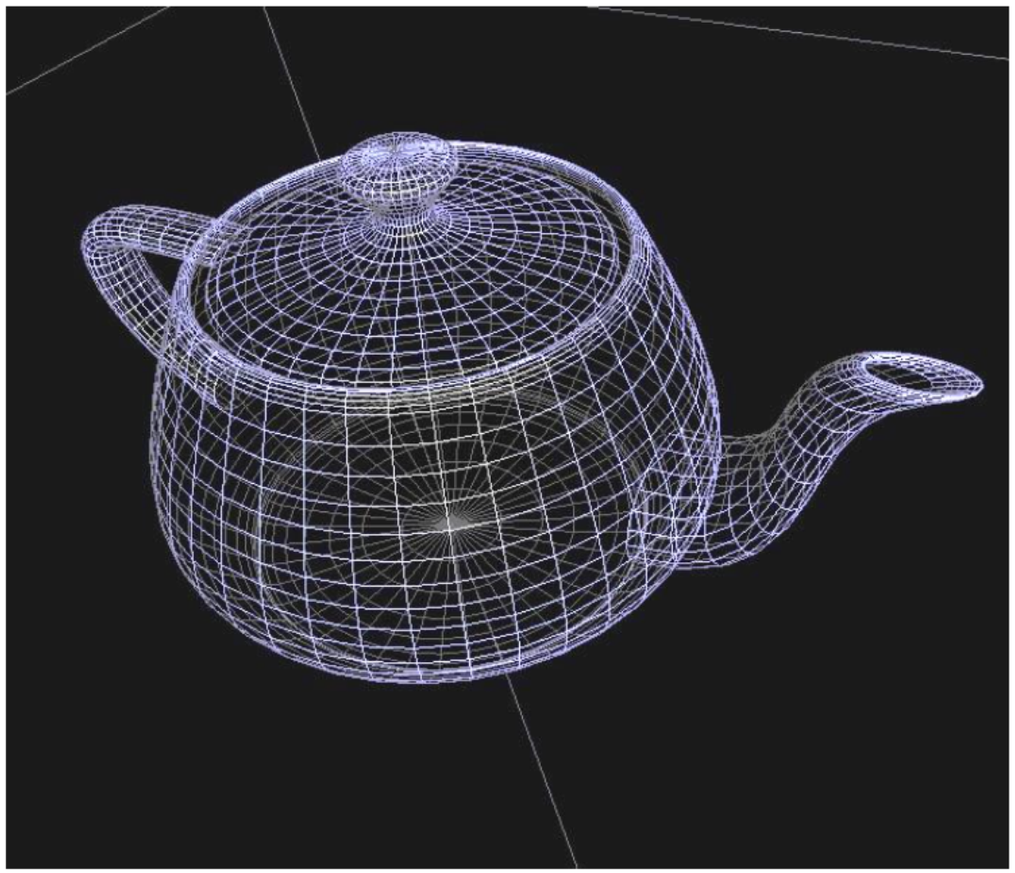
- Rendering algorithm for generating 2D images from 3D scenes
- Done by transforming geometric primitives (lines, polygons) into raster image representations (pixels)



[Akenine-Moeller et al.: Real-time Rendering]

Rasterization

- 3D objects are represented by vertices, lines and polygons
- These primitives are processed to obtain a 2D image



[Akenine-Moeller]

Rendering

Two approaches:

- Local Illumination Models
  - OpenGL / DirectX
  - GLSL (shaders)
- Global Illumination Models
  - Raytracing
  - Pathtracing
  - Photon Mapping
  - Radiosity
  - …

Rendering Pipeline

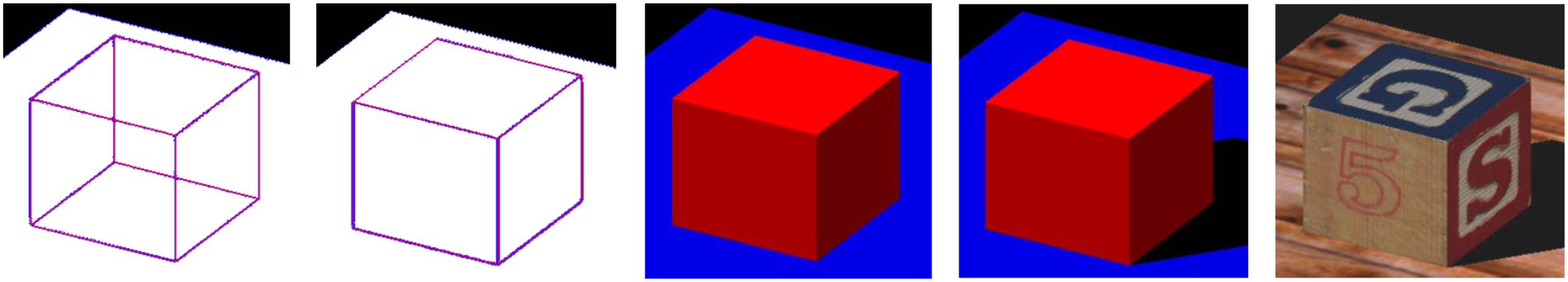
- For real-time graphics (local illumination models)
- Rendering or Graphics Pipeline = processing stages
- Supported by commodity graphics hardware
    - GPU: Graphics Processing Units
    - Computes stages of the rasterization-based rendering pipeline
- OpenGL and directX are software interfaces (APIs)
    - This course gives the details assuming the use of OpenGL

Rendering Pipeline

- 3D input
  - A virtual camera
    - position, orientation, focal length
  - Objects
    - vertices, lines, polygons
    - geometric and material properties (position, normal, color, texture coordinates)
  - Light sources
    - position, direction, color, intensity
  - Textures
    - images to describe the color of an object
- 2D output
  - Per-pixel color values in the framebuffer

Rendering Pipeline - Tasks

- Resolve visibility
- Evaluate a lighting model
- Compute shadows (not a core functionality)
- Apply textures



visibility lighting model shadow texture

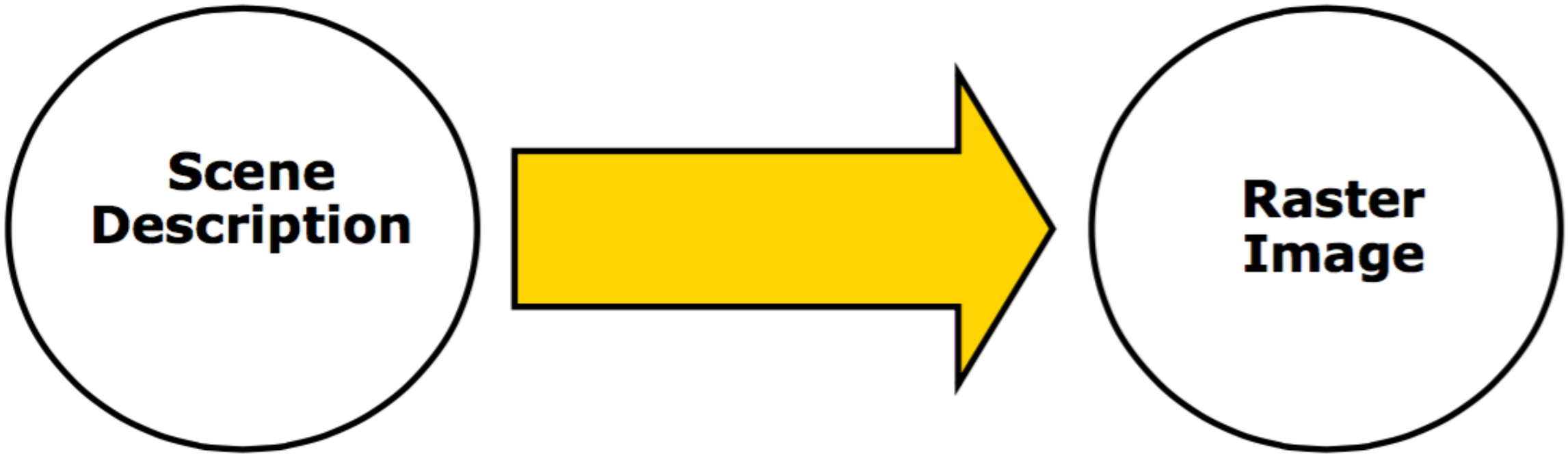
[Wright et al.: OpenGL SuperBible]

Rendering Pipeline - Stages

1. Processing of primitives
2. Vertex shader
3. Primitive assembly
4. Processing of fragments
5. Fragment shader
6. Pixel storage in framebuffer

(0). Installing/compiling programs (i.e. vertex and fragment shaders)

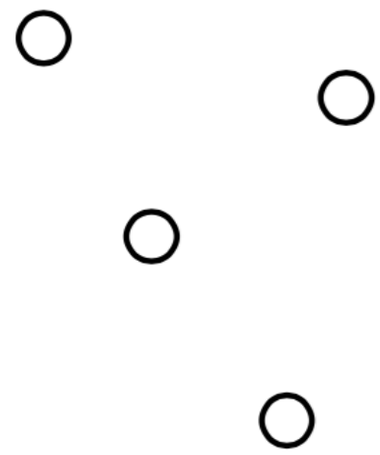
Rendering Pipeline



Rendering Pipeline

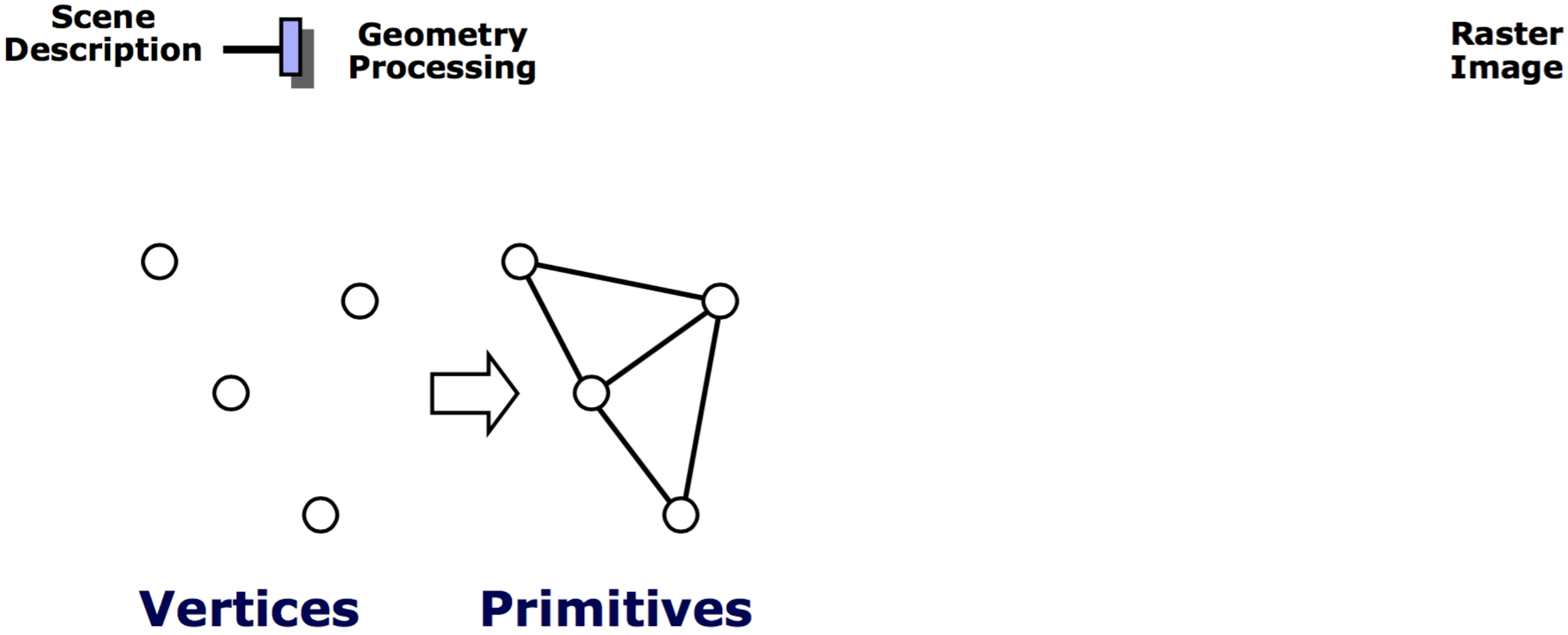
Scene Description

Raster Image

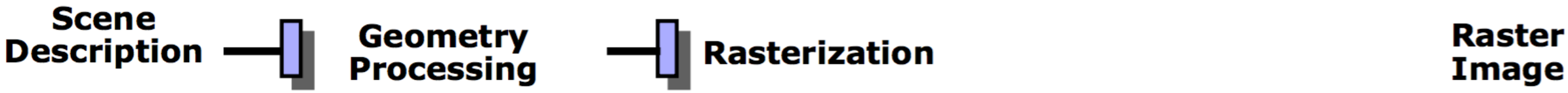


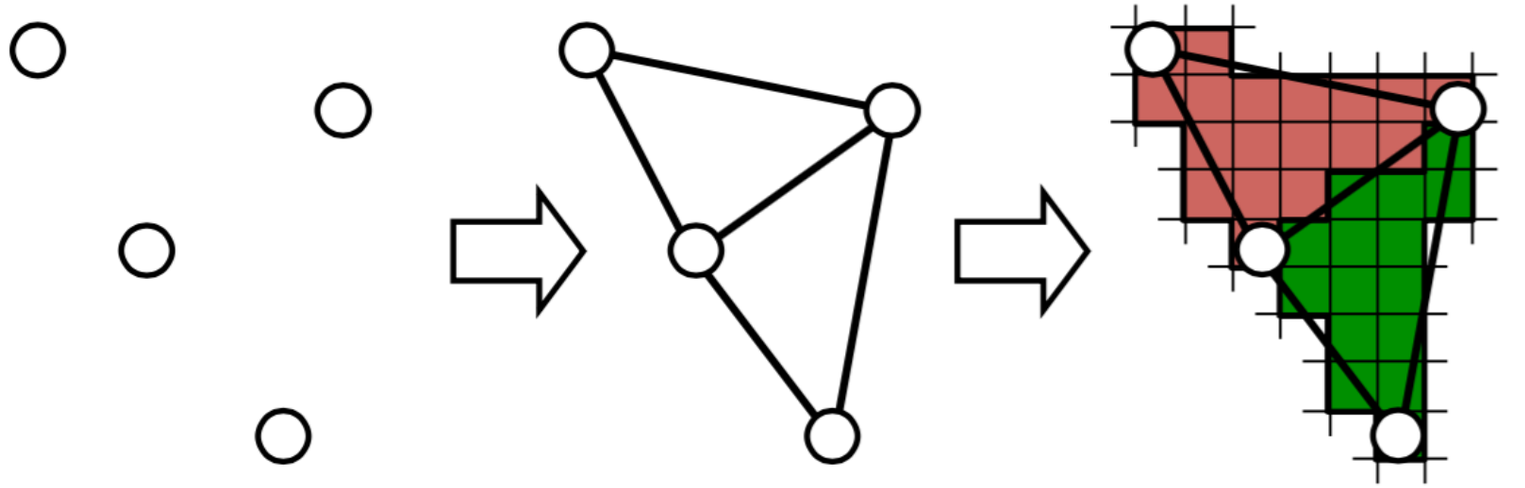
Vertices

Rendering Pipeline



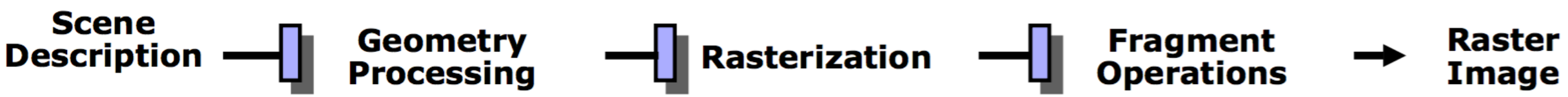
Rendering Pipeline

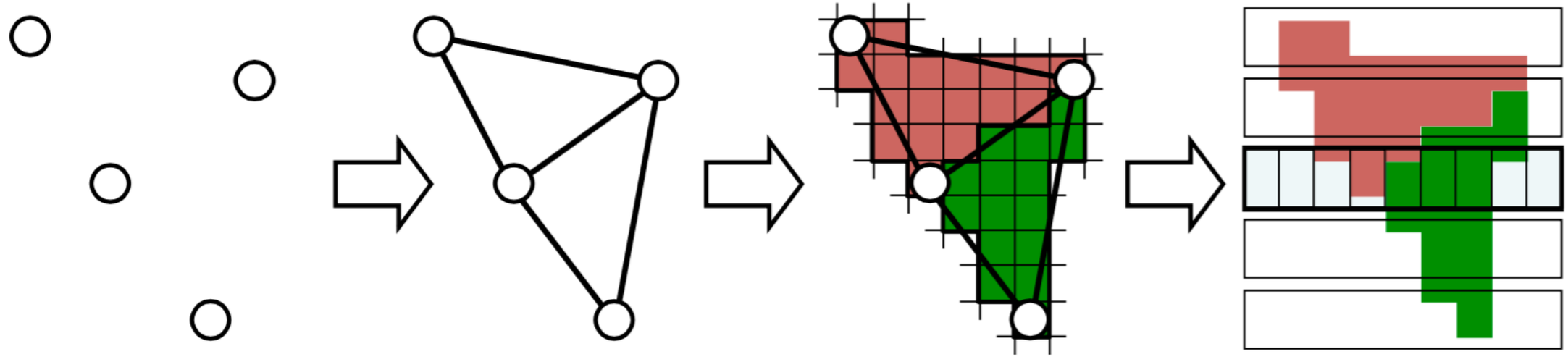




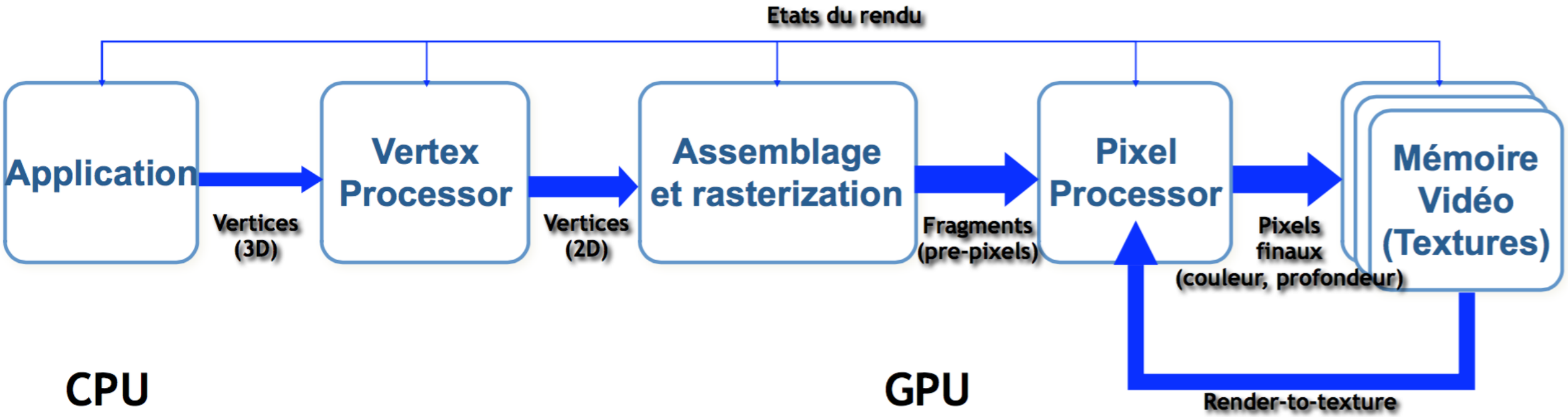
Vertices **Primitives** **Fragments**

Rendering Pipeline

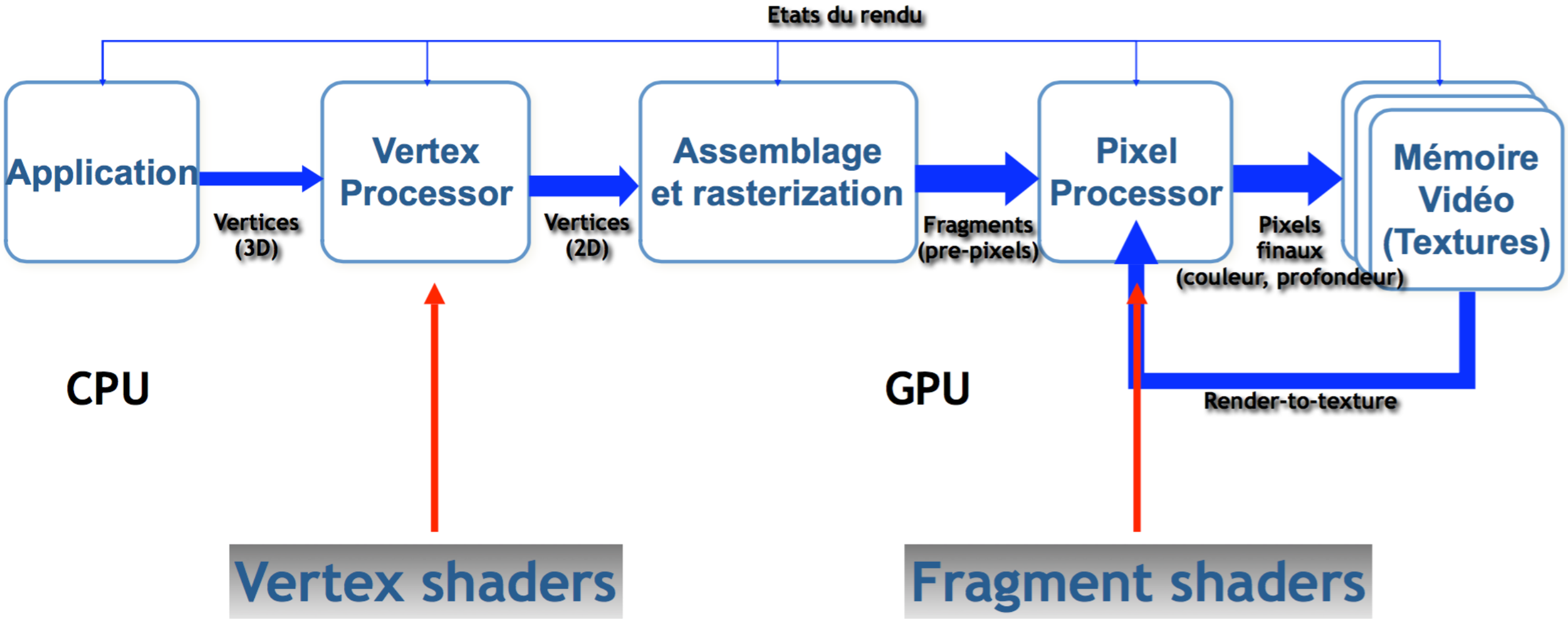




Rendering Pipeline



Rendering Pipeline



1. Processing of primitives
2. Vertex shader
3. Primitive assembly
4. Processing of fragments
5. Fragment shader
6. Pixel storage in frame buffer

Stage 1: Processing of Primitives

Also called geometry stage

- Processes all vertices independently in the same way
- Performs transformations per vertex, lighting per vertex
- If no attributes are given, OpenGL uses default attributes
  - Minimum attribute: position
  - Other attributes: color, normals, texture coordinates

```
glBegin(GL_TRIANGLES);
  glVertex3fv(a);
  glVertex3fv(b);
  glVertex3fv(c);
glEnd();
```

1. Processing of primitives
2. Vertex shader
3. Primitive assembly
4. Processing of fragments
5. Fragment shader
6. Pixel storage in frame buffer

Stage 2: Vertex Shader

- Projects the vertices into the camera frame onto image plane

X' = PM . X

M: modelview matrix (local -> global -> camera)

P: projection matrix (onto image plane)

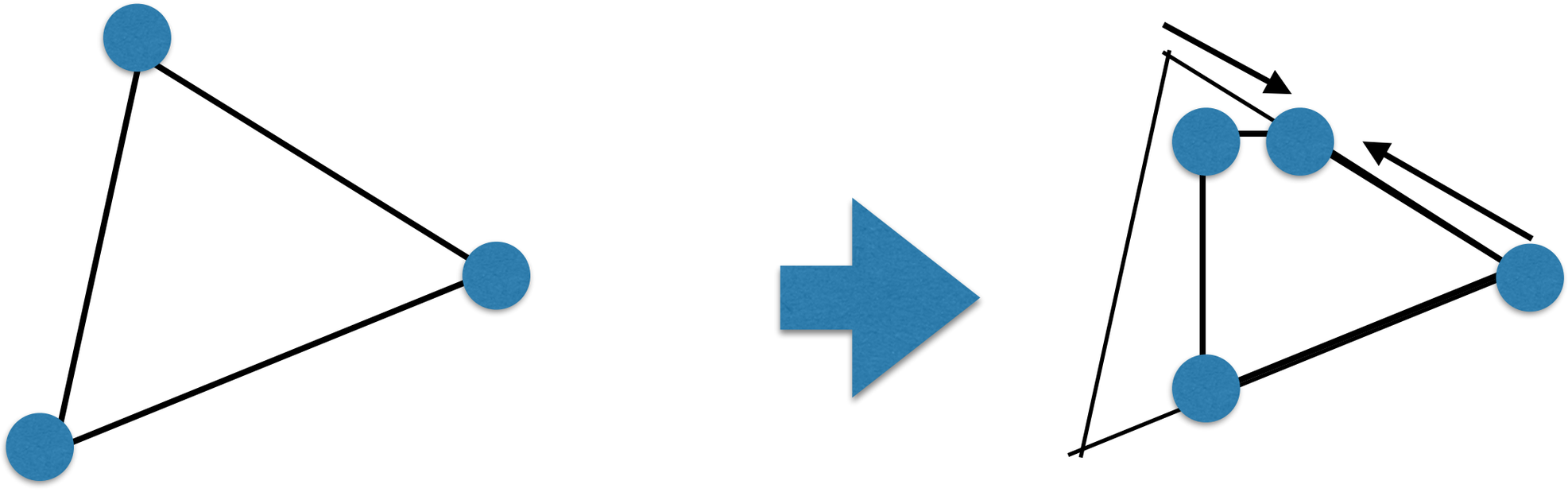
- May add additional operations: generates, modifies, discard primitives

```
gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
```

1. Processing of primitives
2. Vertex shader
3. Primitive assembly
4. Processing of fragments
5. Fragment shader
6. Pixel storage in frame buffer

Stage 3: Primitive Assembly

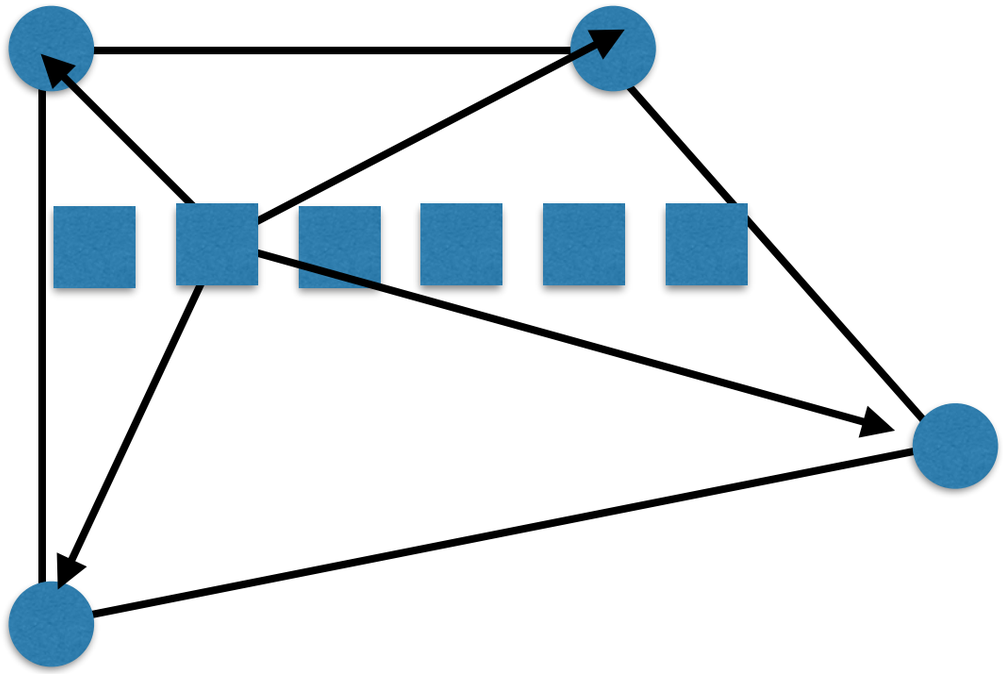
- Assembles primitives (vertices, lines, triangles)
- Converts primitives into a raster image (projection, is it visible ?)
- If vertices are created, their properties are interpolated (segment)



1. Processing of primitives
2. Vertex shader
3. Primitive assembly
4. Processing of fragments
5. Fragment shader
6. Pixel storage in frame buffer

Stage 4: Processing of Fragments

- Generates fragments candidates
  - Draws the image line by line, pixel by pixel
- Fragment attributes are interpolated from vertices of a primitive
  - Position, color, transparency, depth



1. Processing of primitives
2. Vertex shader
3. Primitive assembly
4. Processing of fragments
5. Fragment shader
6. Pixel storage in frame buffer

Stage 5: Fragment Shader

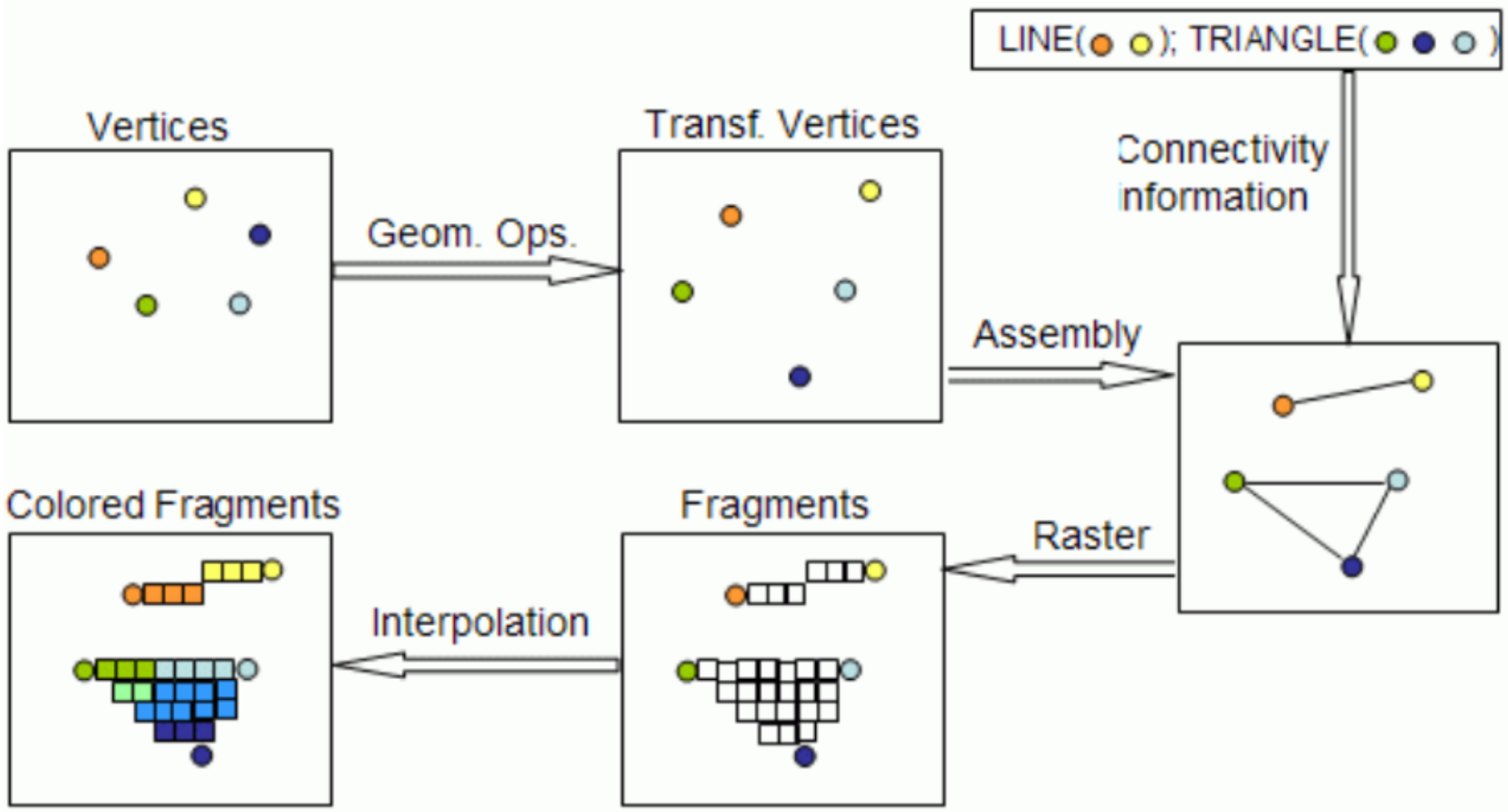
- Determines the color, transparency and depth of the fragment
- Values computed automatically by OpenGL may be changed to… whatever !
- Remember all values are obtained by interpolation (need to renormalize normals for example)

```
gl_FragColor = gl_FrontColor;
```

1. Processing of primitives
2. Vertex shader
3. Primitive assembly
4. Processing of fragments
5. Fragment shader
6. Pixel storage in frame buffer

Stage 6: Pixel Storage in Framebuffer

- Creation of the final image
  - Keeps only the fragments that are closer to the camera



[Lighthouse 3D]

Stage 6: Depth Test

- Compares depth value of the fragment and depth value of the framebuffer at the position of the fragment
- Used for resolving the visibility
- If the depth value of the fragment is larger than the depth value of the framebuffer, the fragment is discarded
- Otherwise the fragment passes the test and is (potentially) written in the frame buffer (overwriting the current color and depth)
- There are other possible tests

Fragment Processing

- Texture lookup (looking up a texel in a texture image)
- Texturing (combination of color and texel)
- Fog (color based on fog color and depth value)
- Antialiasing (adaptation of alpha value and color)
- Scissor test (masked rendering)
- Alpha test (transparency, billboarding)
- Stencil test (masking, shadows)
- Depth test (visibility)
- Blending (when alpha values are used)
- Dithering (finite number of colors)
- Logical operations / masking

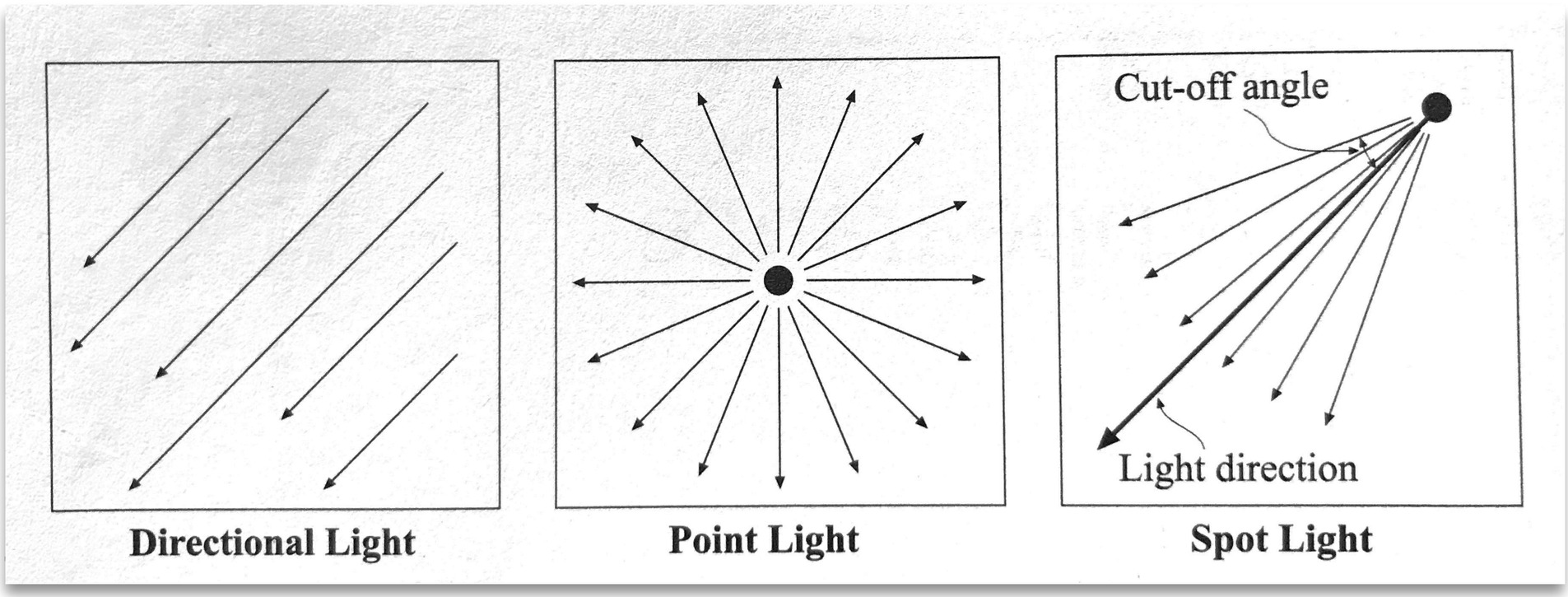
Overview

1. Rendering Pipeline
2. Light Sources
3. Material
4. Lighting
5. Shading
6. Textures
7. Shaders (GLSL)

2. Light Sources

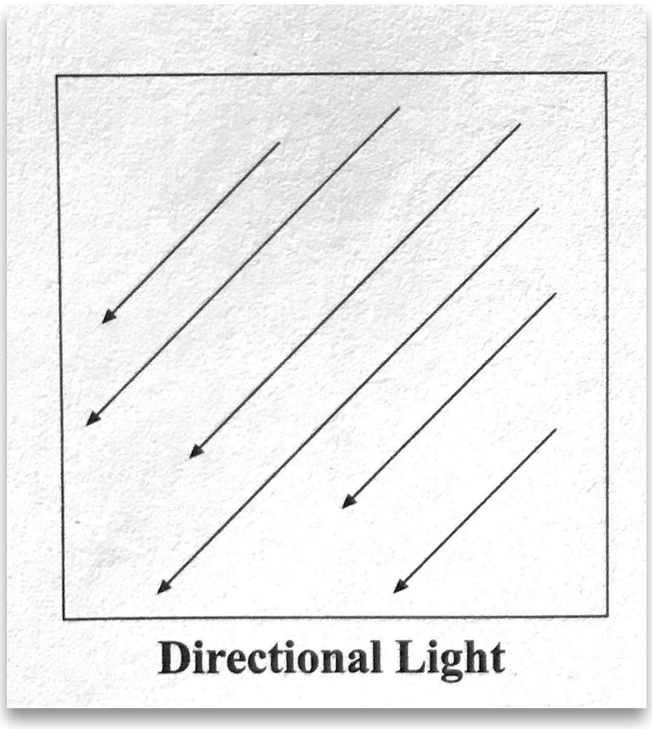
Light Sources

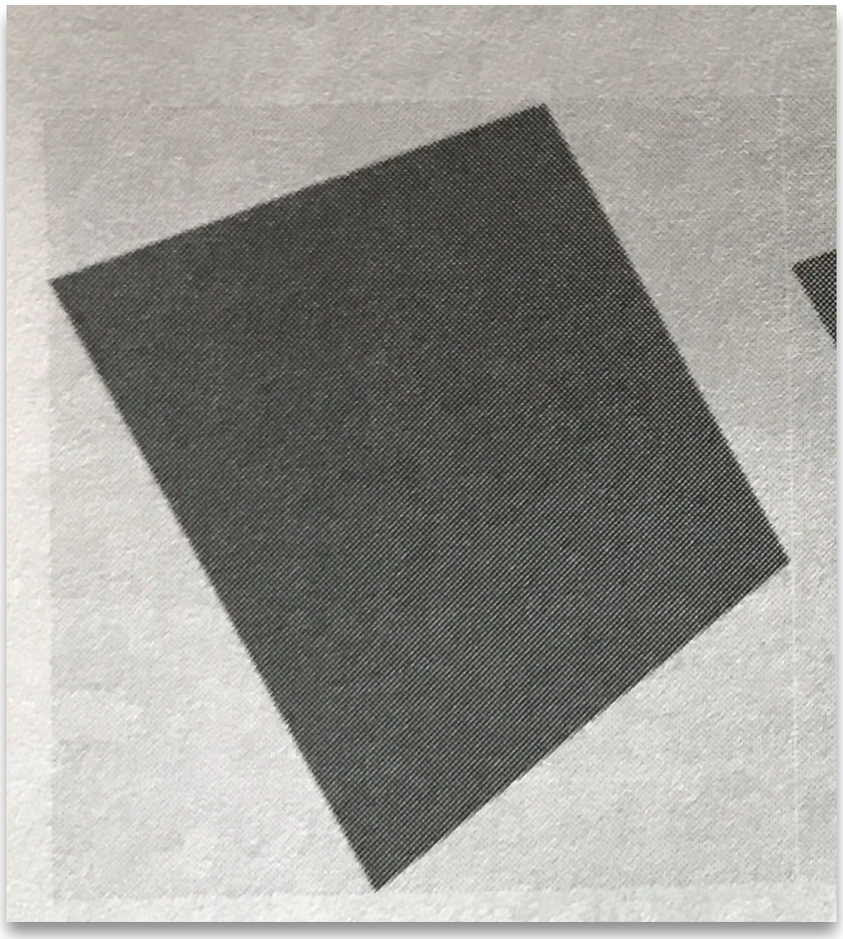
- 3 different types of light sources
  - Directional lights
  - Point lights
  - Spotlights



Directional Light

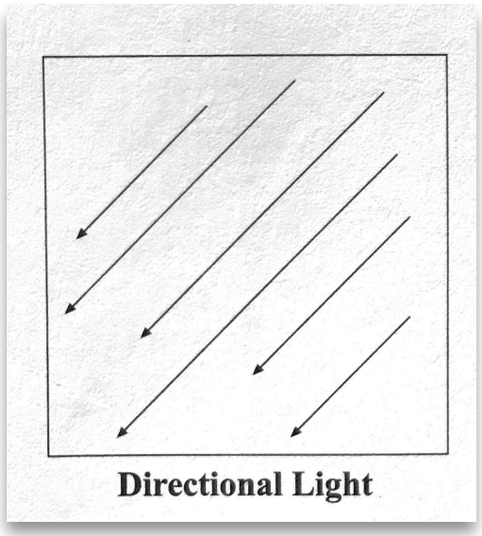
- Positioned infinitely far away from objects
- Light rays are parallel
- Example: the sun

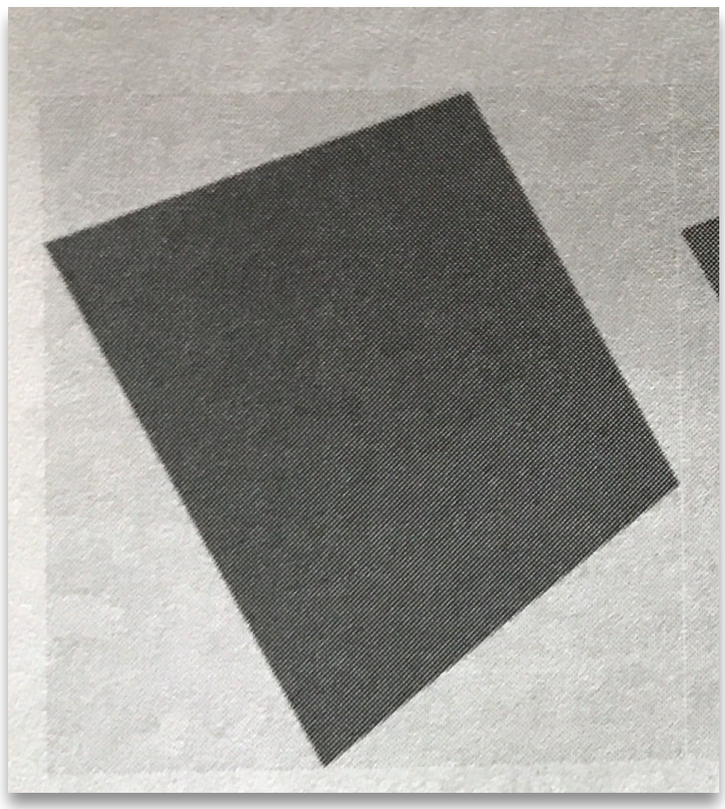




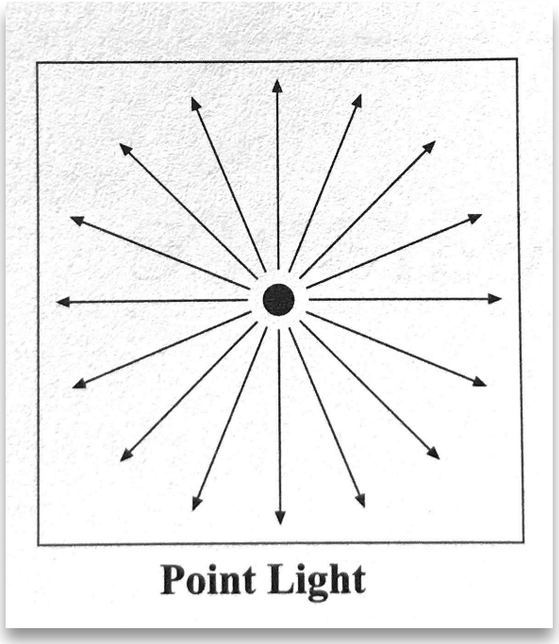
Directional Light

- Parameters : direction + intensity + color
  - Lpos: light source position (w=0)
  - Lamb : ambient intensity color
  - Ld : diffuse intensity color
  - Ls : specular intensity color



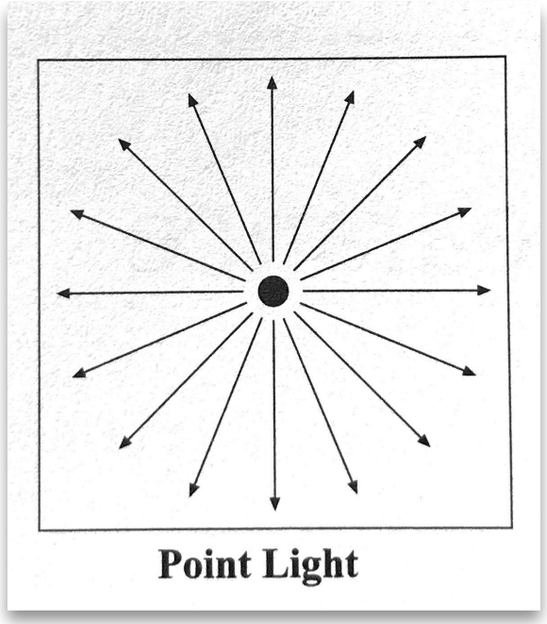


Point Light

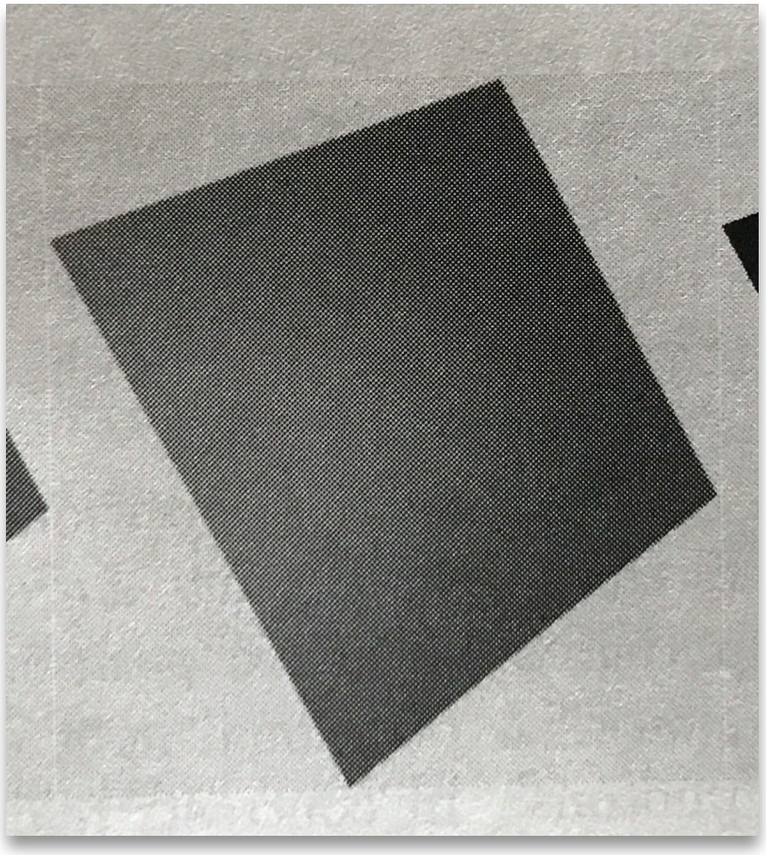


- Positional light (has a location in space)
- Light rays are send uniformly in every direction
- Attenuation (optional): intensity decreases as distance from the light increases
  - 1/(kc + kl*d + kq*d*d)
  - d = distance
  - kc (constant), kl (linear), kq (quadratic)

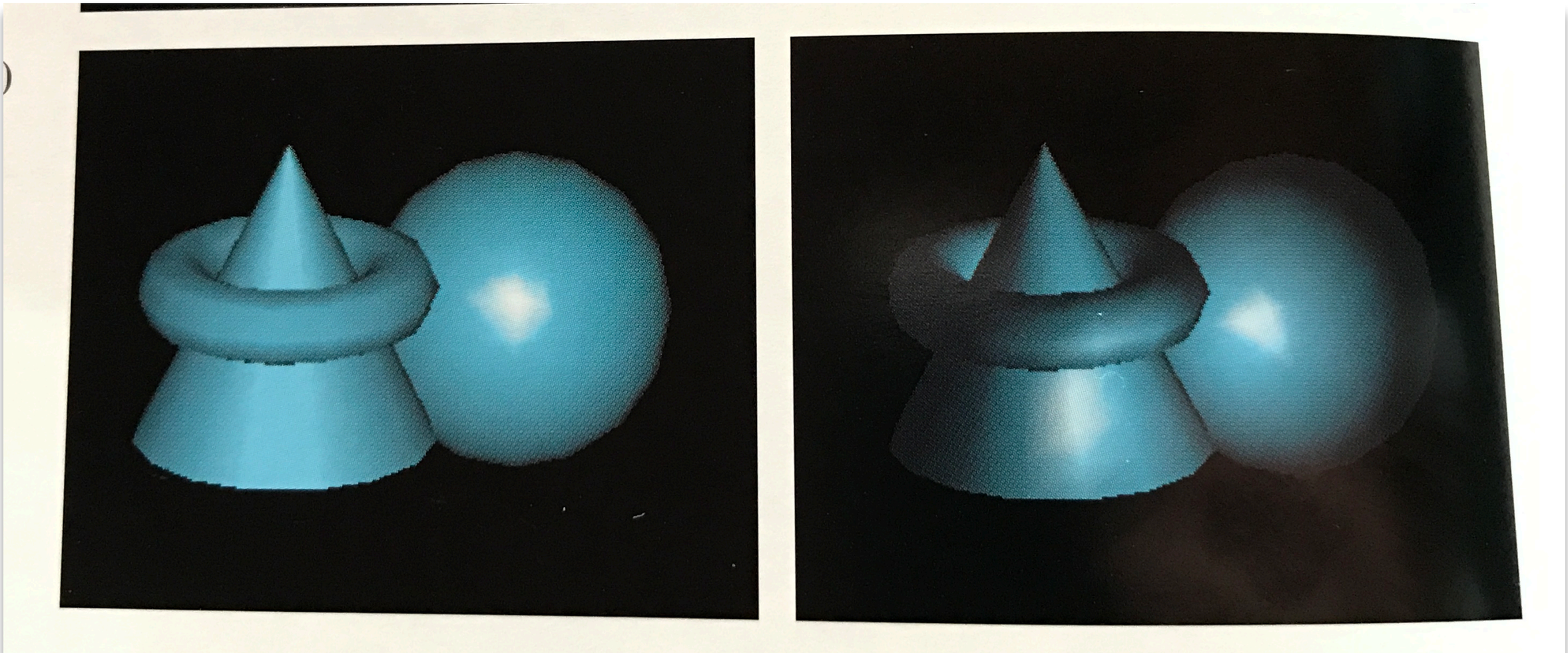
Point Light



- Parameters : position + intensity + color
  - Lpos: light source position (w=1)
  - Lamb : ambient intensity color
  - Ld : diffuse intensity color
  - Ls : specular intensity color
- Attenuation
  - kc, kl, kq



Directional vs Point Light

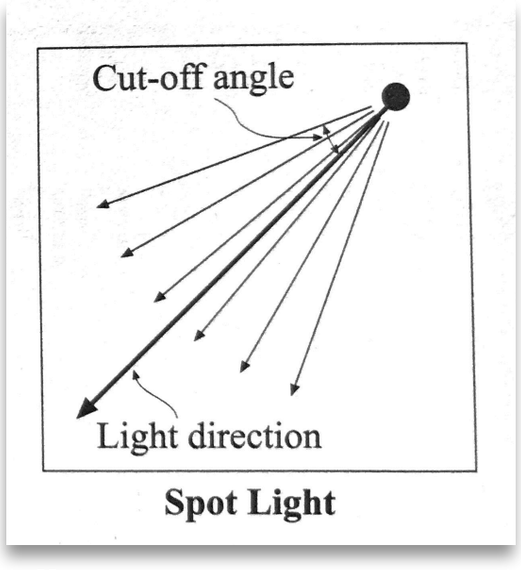


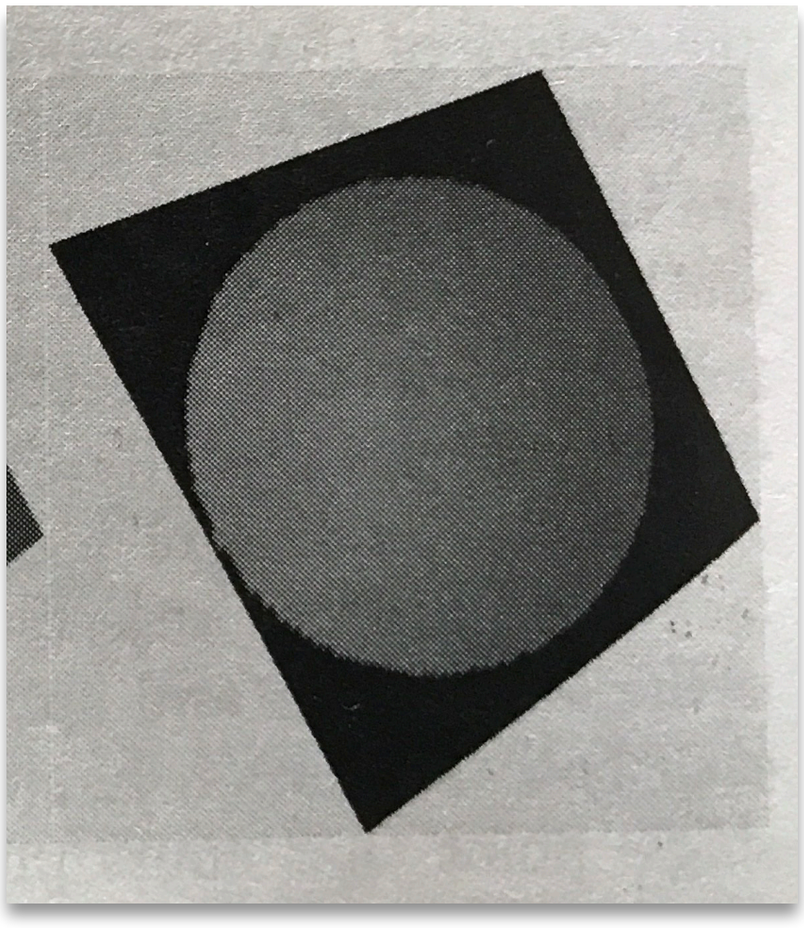
Directional Light

Point Light

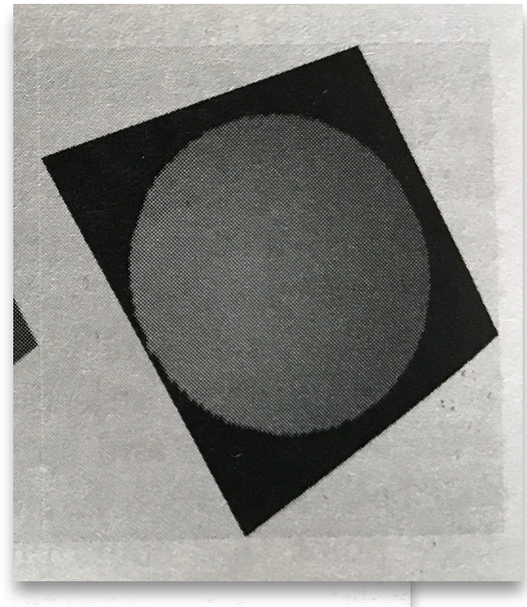
Spotlight

- Positional light (has a location in space)
- Emits rays only within the volume of a cone
  - Light direction
  - Cut-off angle
- Example: flashlight

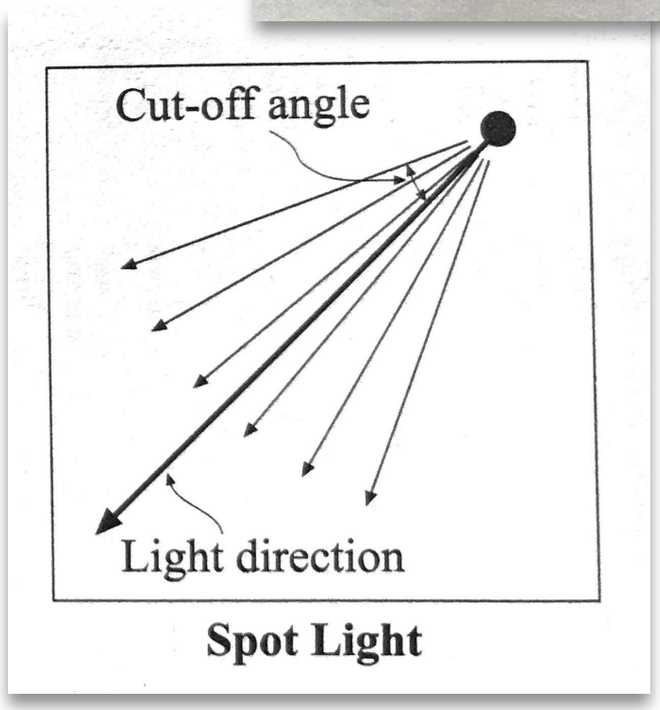




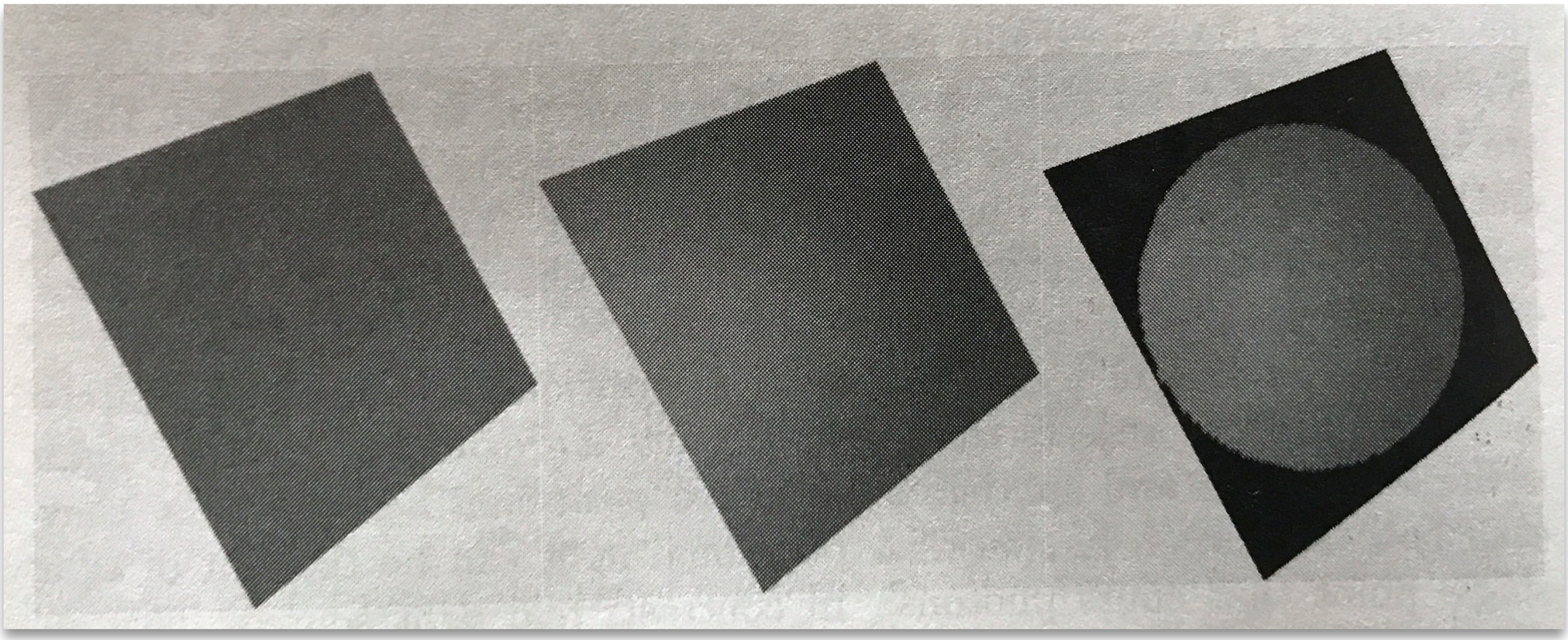
Spotlight



- Parameters : position + intensity + color
  - Lpos: light source position (w=1)
  - Lamb : ambient intensity color
  - Ld : diffuse intensity color
  - Ls : specular intensity color
- Ldir: spot direction
- Lcut: cut-off angle (half the angle of the cone)
- Lexp: spot exponent (fall off from the center of the cone, optional)
- Attenuation (optional): kc, kl, kq



Comparison



Overview

1. Rendering Pipeline
2. Light Sources
3. Material
4. Lighting
5. Shading
6. Textures
7. Shaders (GLSL)

3. Material

Material

- Parameters that describe the material
- Kamb: ambient material color
- Kd: diffuse material color
- Ks: specular material color
- n: shininess (if specular object)
- I_e: emissive material color (if object emits light)

Overview

1. Rendering Pipeline
2. Light Sources
3. Material
4. Lighting
5. Shading
6. Textures
7. Shaders (GLSL)

4. Lighting

Lighting

Definition

- Lighting: interaction between material and light sources as well as the interaction with the geometry of the objects to be rendered
  - Computed thanks to an object BRDF
  - BRDF = Bi-Directional Reflectance Distribution Function
- Computes the exact colour of a given 3D point in the scene

Lighting Models

BRDF

- BRDF defines the behavior of the material
  - Absorption
  - Reflexion
  - Refraction
  - Transmission
  - Fluorescence
- Takes into account the microscopic aspect of the surface
- Depends on the wavelength of the light

Modèle d'éclairement local (Lighting)

- Exprime l'intensité lumineuse d'un point due à :
  - la lumière ambiante que produit dans toutes les directions un éclairage uniforme
  - les sources lumineuses qui sont à l'origine :
    - des réflexions diffuses et spéculaires
    - des ombres portées
  - la réflectance et la transmittance d'un objet
    - $I_r = I_a + I_d + I_s$
  - I_a est l'intensité ambiante
  - I_d est l'intensité diffuse
  - I_s est l'intensité spéculaire

Lighting Equation

- $I_r = I_a + I_d + I_s$
- $I_{tot} = I_r + I_e$ (if material is emissive)

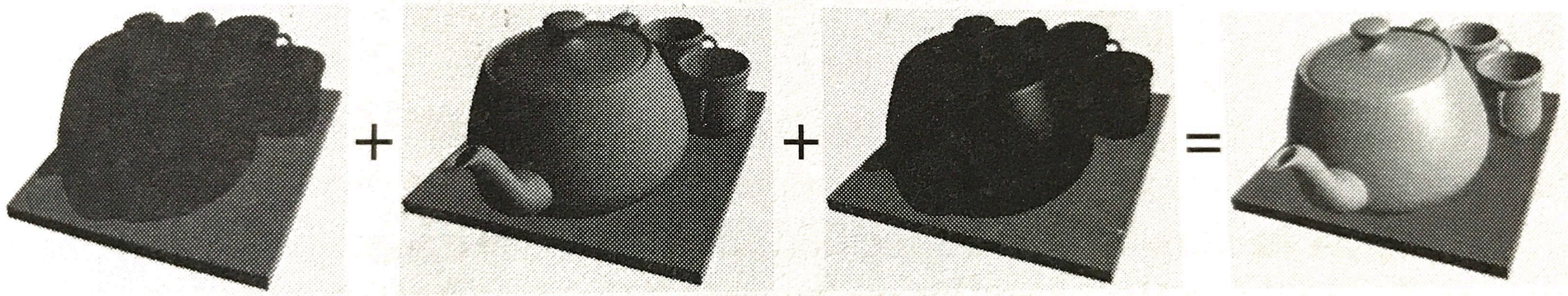


Figure 4.13. The basic lighting equation is illustrated for a teapot by adding (from the left) the ambient, the diffuse, and the specular components. The resulting lighting is shown at the right. *(Tea cup model is reused courtesy of Joachim Helenklaken.)*

Éclairage ambiant

- Chaque objet est montré avec une intensité intrinsèque :
  - Remplace les échanges lumineux indirects
  - Univers irréel d'objets lumineux non réfléchissants
  - $I_a = K_{amb} \times L_{amb}$
    - K_{amb} : couleur ambiante de l'objet
    - L_{amb} : couleur ambiante de la source de lumière
    - Componentwise multiplication

Diffuse Reflexion

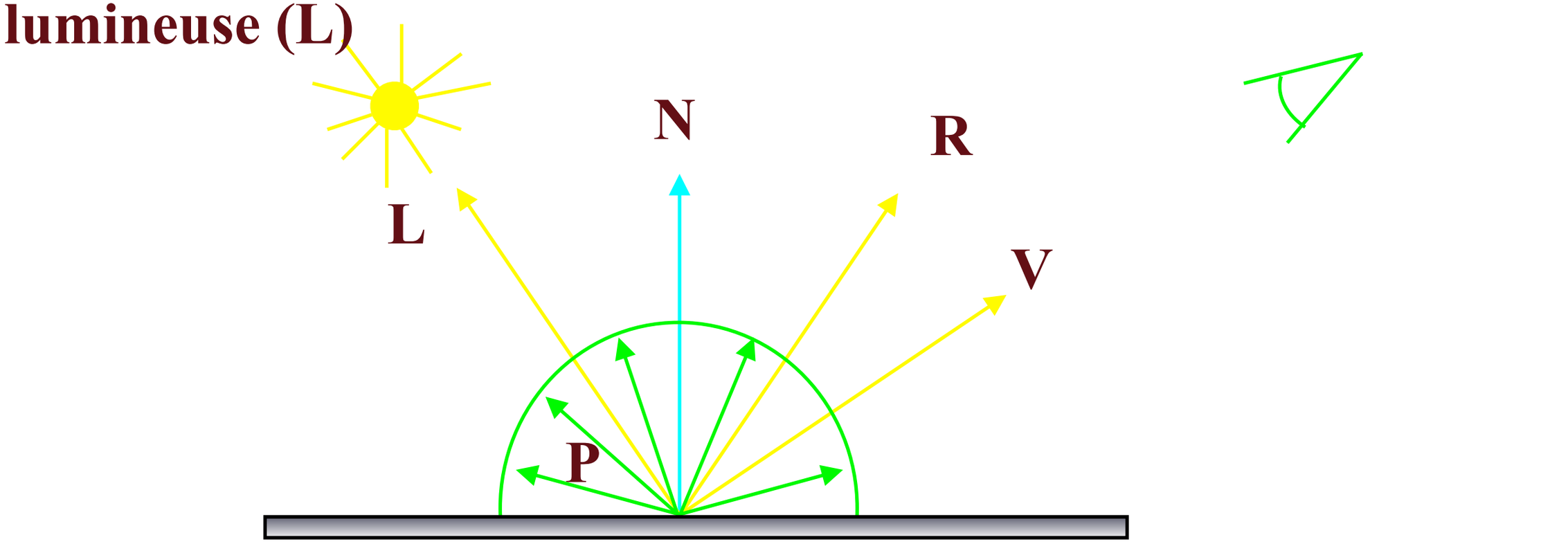
Lambert

- une partie de la lumière incidente pénètre dans l'objet et ressort avec la même intensité dans toutes les directions
  - $I_d = K_d \times L_d \cdot \cos(N, L) / d^2$
    - K_d : couleur diffuse de l'objet
    - L_d : couleur diffuse de la source lumineuse
    - N : normale à la surface au point P
    - L : direction dans laquelle se trouve la source lumineuse
    - d : distance entre la source et l'objet éclairé
  - liée à la rugosité microscopique :
    - plus K_d est grand, plus la diffusion est grande

Diffuse Reflexion

Lambert

- $I_d = K_d \times L_d \cdot N.L / d^2$ (N et L normalisés)



Diffuse Reflexion

Lambert Example

Sphère purement Lambertienne avec une lumière qui se déplace

Specular Reflexion

Phong

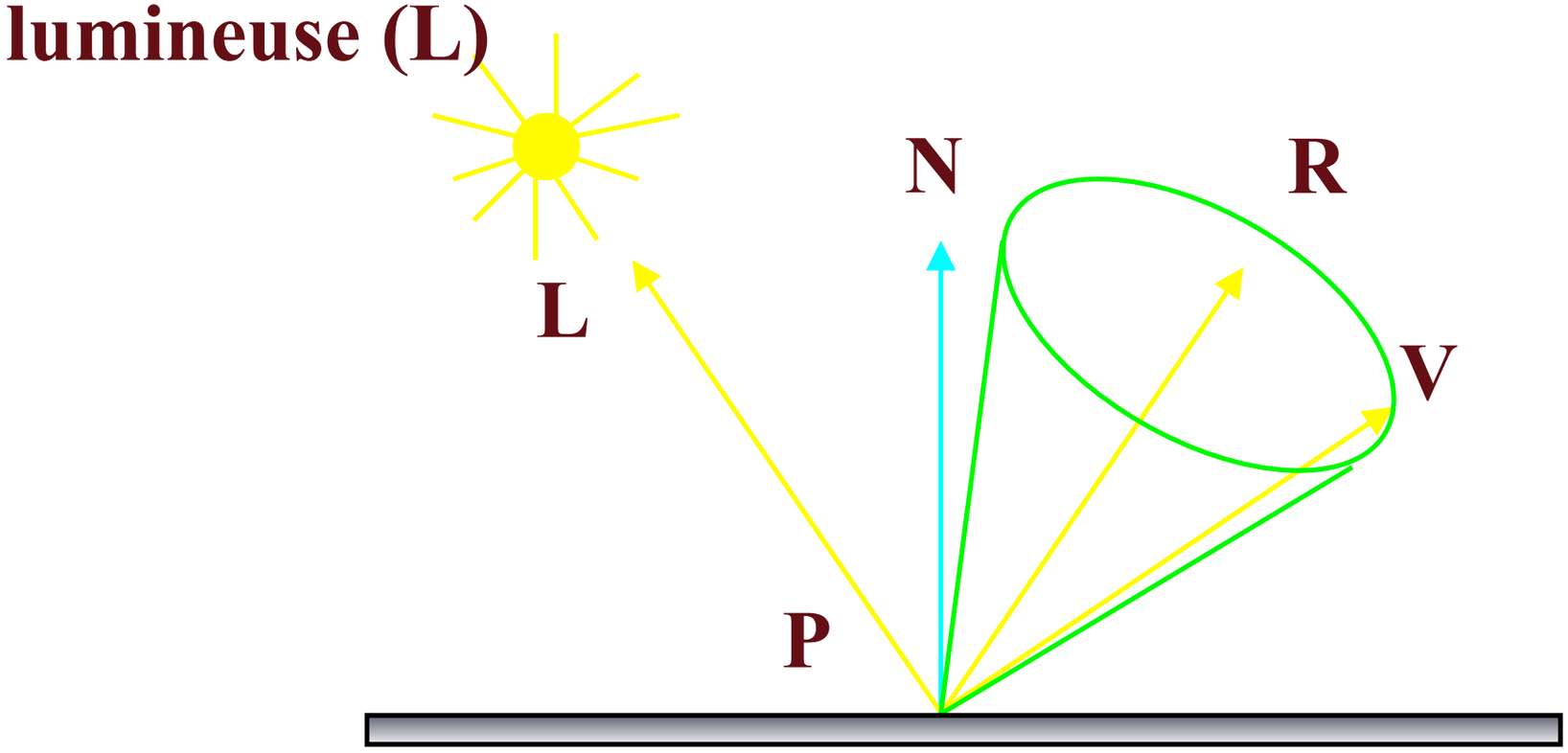
- **Modèle de Phong :**
  - Réflexion par la surface de l'objet de la lumière incidente qui n'a pas pénétré dans l'objet, dépend de la direction d'observation
  - $I_s = K_s \times L_s \cdot \cos^n(R, V) / d^2$

- **K_s : couleur spéculaire de l'objet**
- **L_s : couleur spéculaire de la source lumineuse**
- **R : direction de réflexion spéculaire maximale**
- **V : direction dans laquelle se trouve l'observateur**
- **n est l'exposant de réflexion spéculaire de l'objet (shininess):**
  - **n grand = objet lisse et brillant = cône spéculaire étroit**
  - **n petit = objet terne = cône spéculaire large**
  - **pour le verre, n = 200**

Specular Reflexion

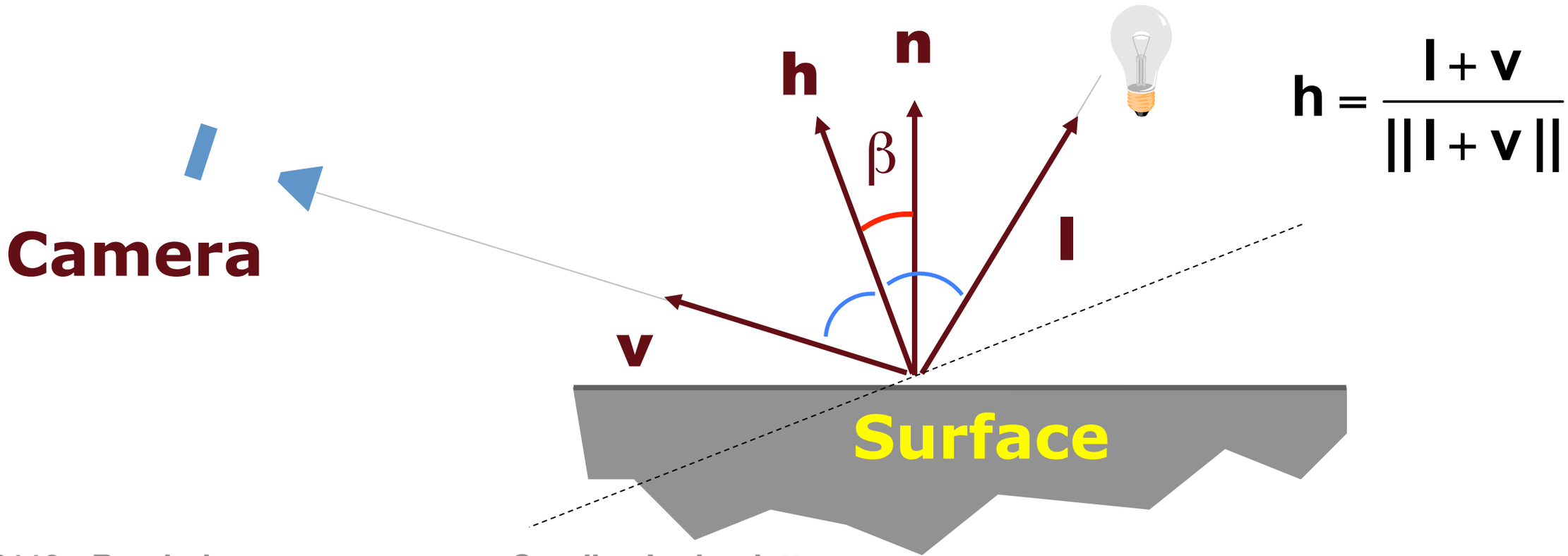
Phong

- $I_s = K_s \times L_s \,.\, (R.V)^n / d^2$ (R et V normalisés)



Specular Reflexion
Blinn-Torrance

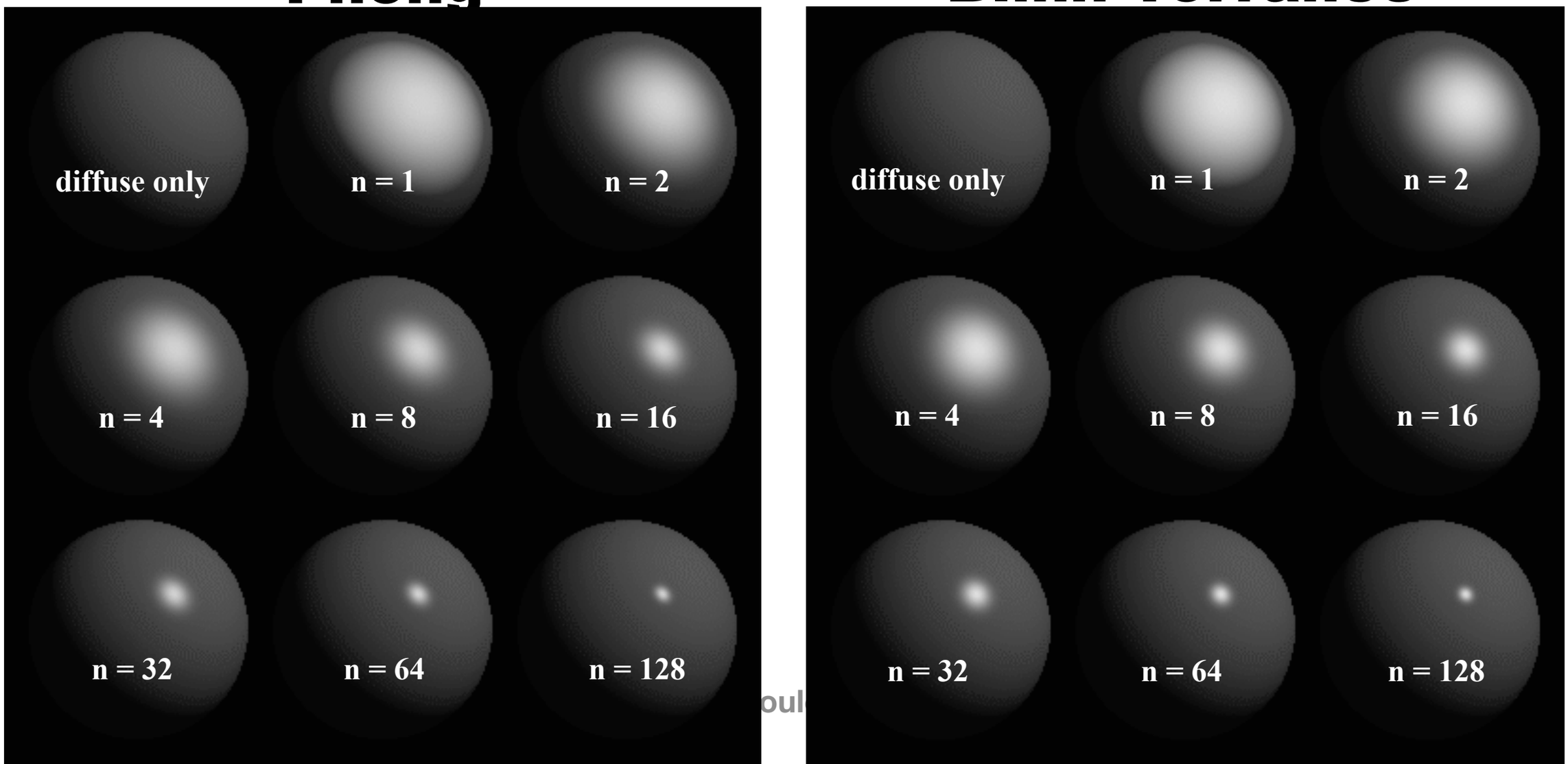
- **Paramètres**
  - **k_s: coefficient de réflection spéculaire**
  - **n : exposant de spécularité (shininess)**
- $K_S \times L_S \cdot (R.V)^n / d^2 = K_S \times L_S \cdot (N.H)^n / d^2$



Specular Reflexion

Examples

- The shininess coefficient n controls the size of the specular spot

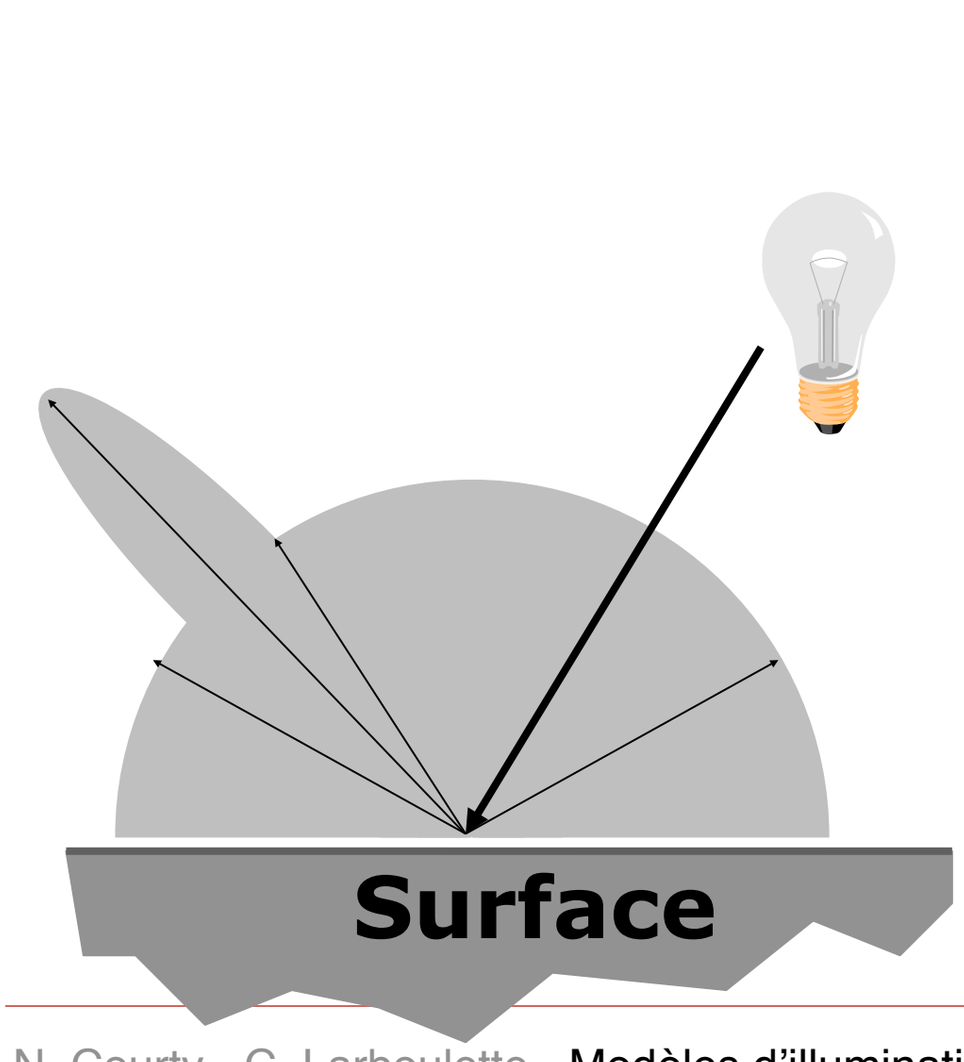


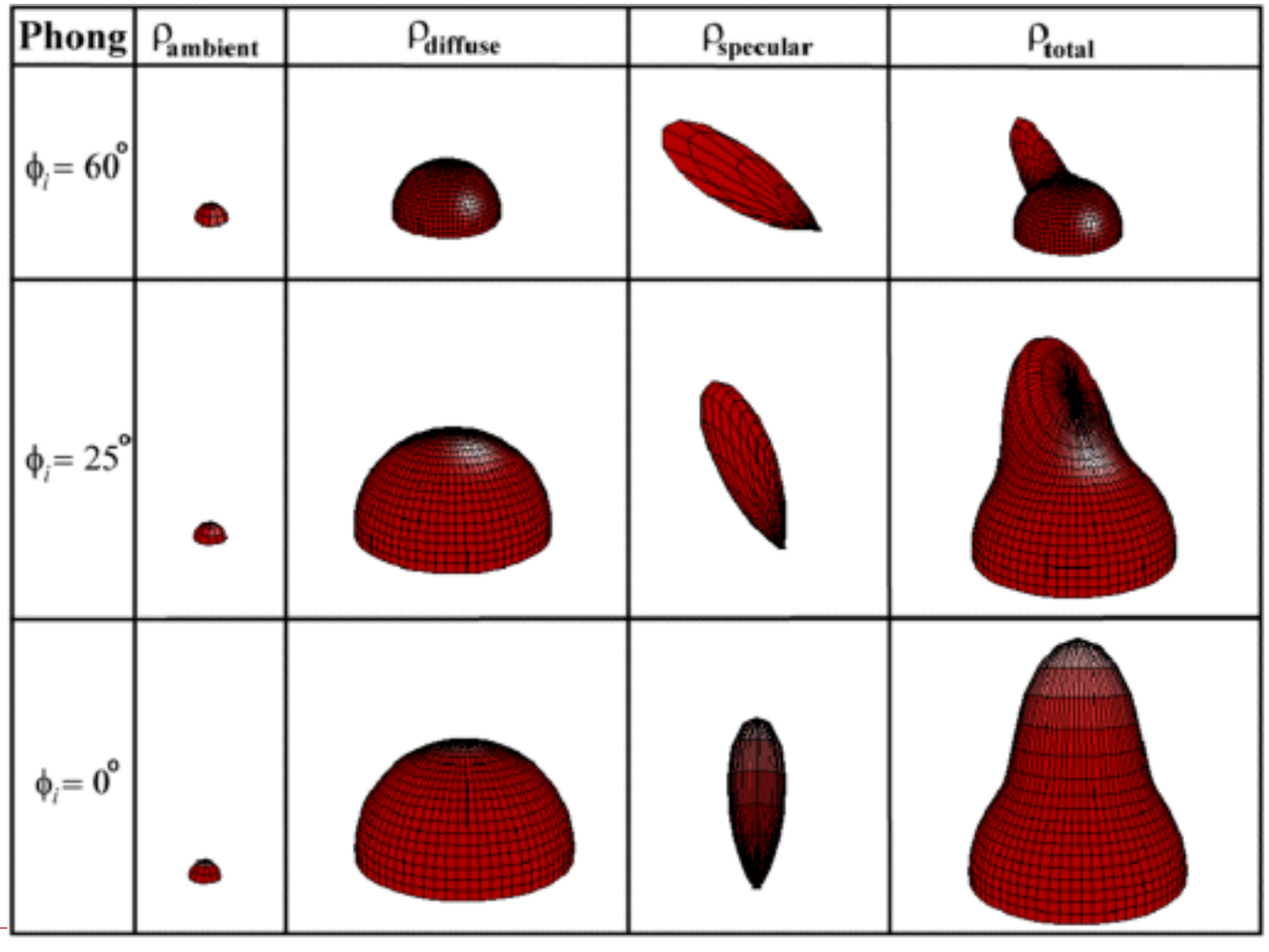
Quelques remarques …

- $K_s + K_d = 1$
- Si plusieurs sources, il faut sommer …
- On a besoin de connaître plusieurs directions :
  - les normales aux facettes et les directions des sources de lumière et de l'observateur :
    - les directions L, N et R sont coplanaires
    - l'angle <L, N> est égal à l'angle <N, R>
- Les coefficients K_s et K_d dépendent normalement de la longueur d'onde
  - Approximation : on les exprime selon des vecteurs à 3 composantes : rouge, vert, bleu
- $I_d + I_s = (K_d \,.\, \cos(N, L) + K_s \,.\, \cos^n(R, V)) \,.\, I_{source} / d^2$
  - on parle de réflectance bidirectionnelle
  - Ou **BRDF (Bidirectional Reflectance Distribution Function)**

BRDF

- Cas du modèle de Phong
  - Somme des 3 composantes





Modèle/mesures

- Le modèle de Phong n'est qu'une simplification
  - Une fonction qu'il est possible d'évaluer

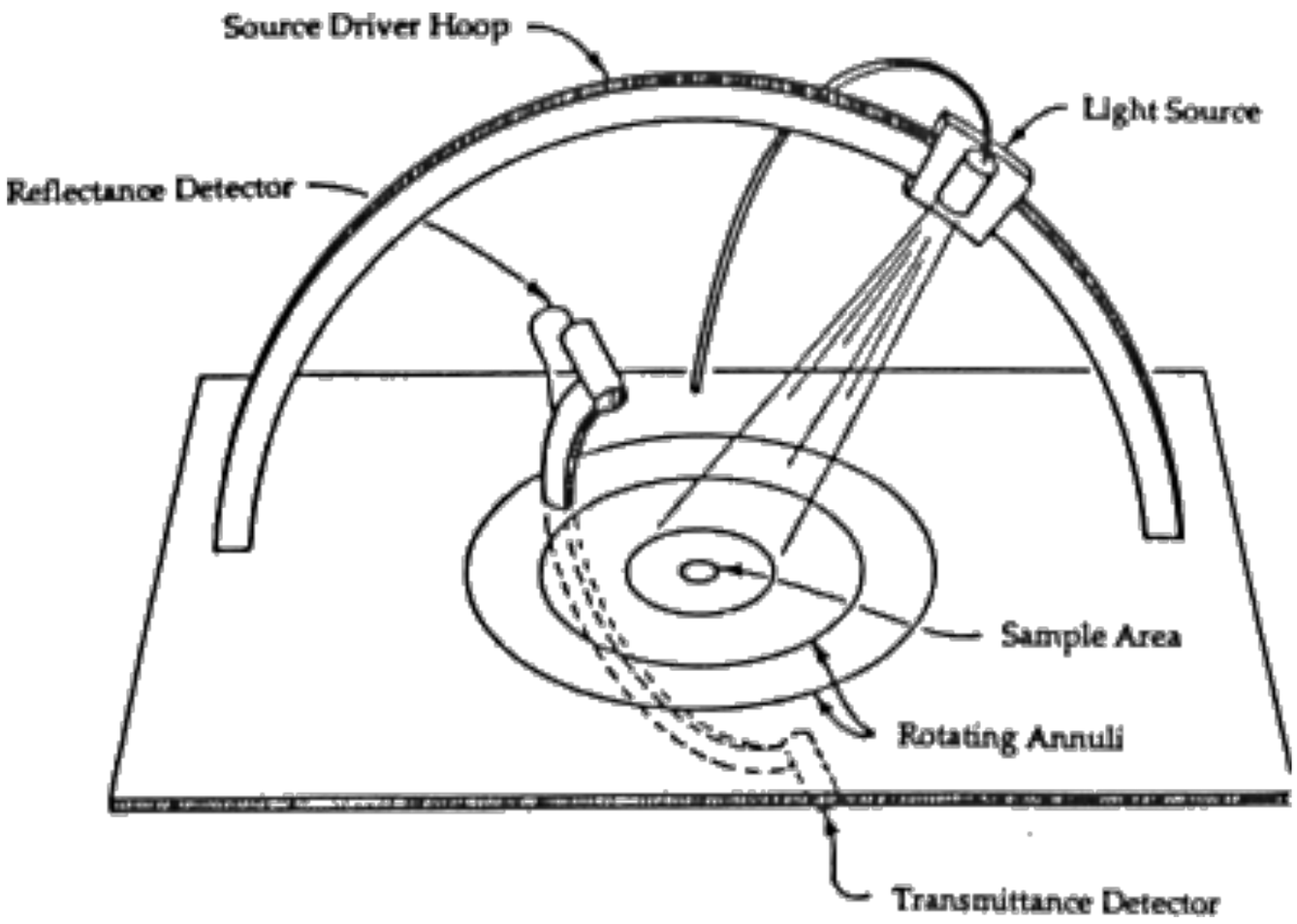
- Pour des réflections réalistes :
  - Il est plus simple de mesurer ce terme
  - On éclaire

BRDFs en pratique

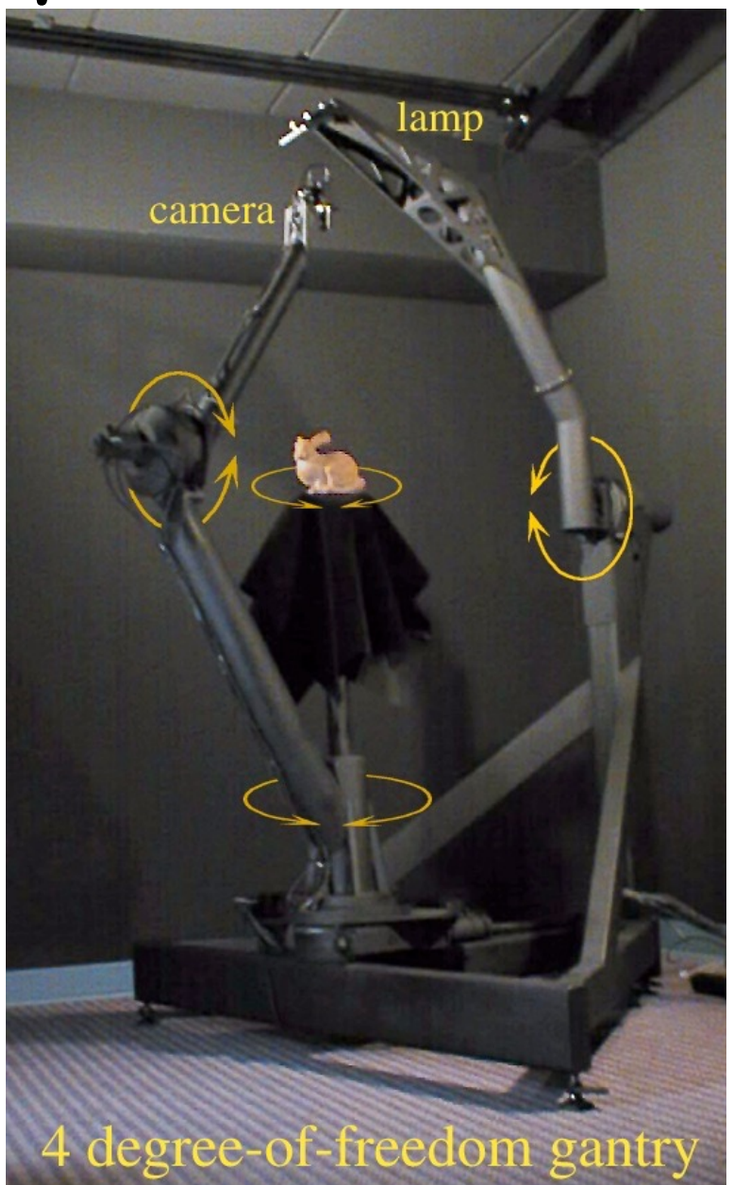
- http://www.virtualcinematography.org/publications/acrobat/BRDF-s2003.pdf
- Les habits de l’Agent Smith sont obtenus synthétiquement avec des BRDF mesurées [Matrix]

Measuring BRDFs

- Gonioréflectomètre
  - 4 degrés de liberté : goniospectrophotomètre



Source: Greg Ward



En application

- http://www.virtualcinematography.org/publications/acrobat/BRDF-s2003.pdf

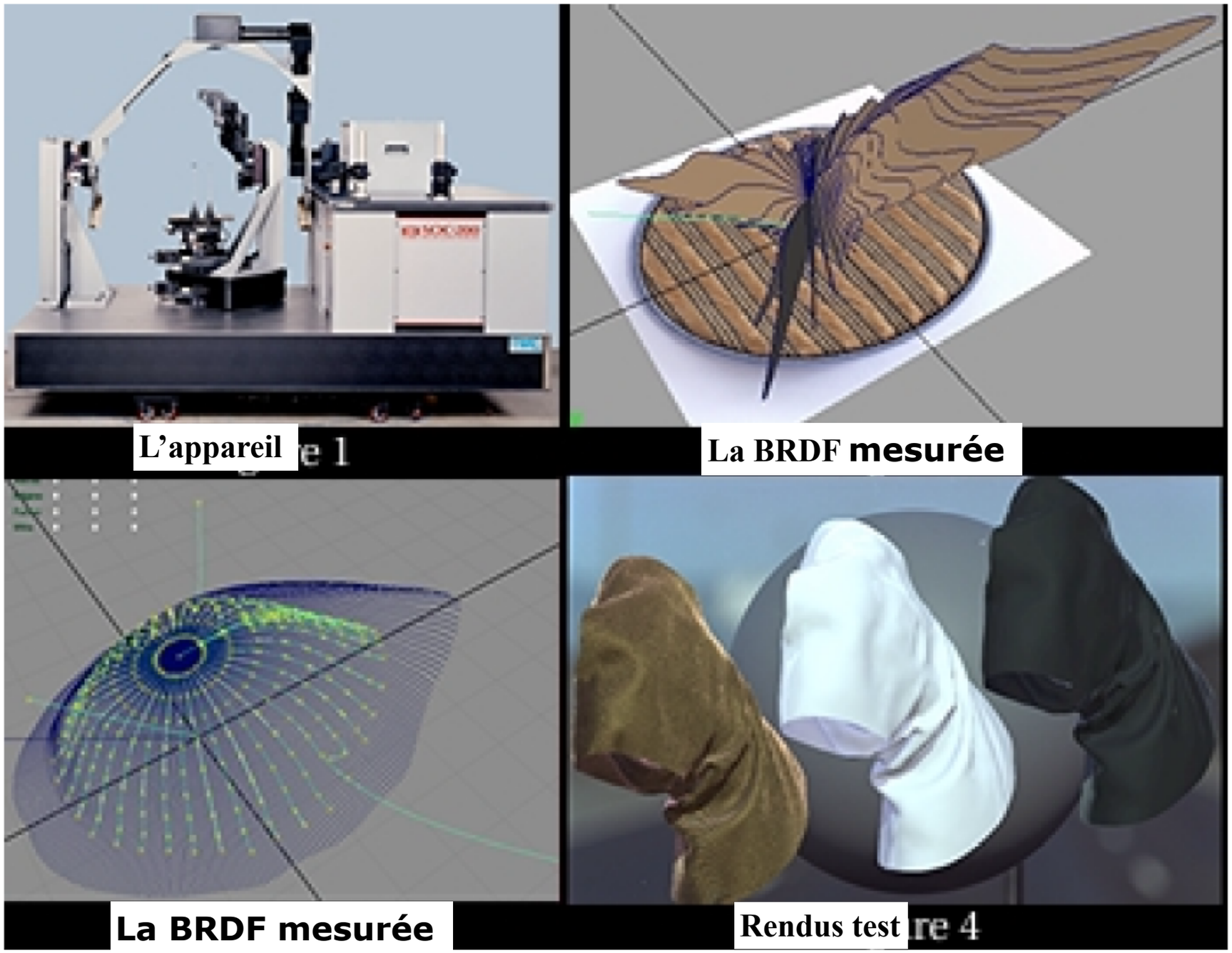
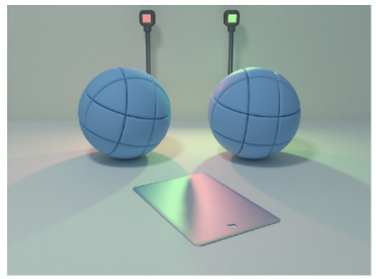


Photo **Synthèse** **Photo** **Synthèse**

Plusieurs modèles de BRDF

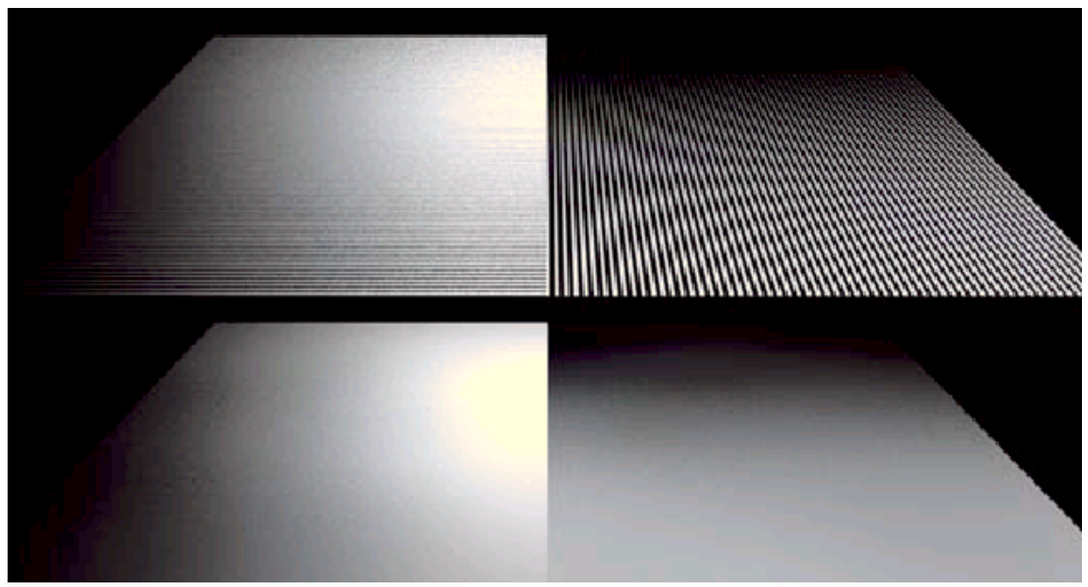
- Phénoménologiques
  - Phong [75]
    - Blinn [77]
  - Ward [92]
  - Lafortune et al. [97]
  - Ashikhmin et al. [00]
- Physiques
  - Cook-Torrance [81]
  - He et al. [91]



Temps de calcul croissant

BRDFs anisotropiques

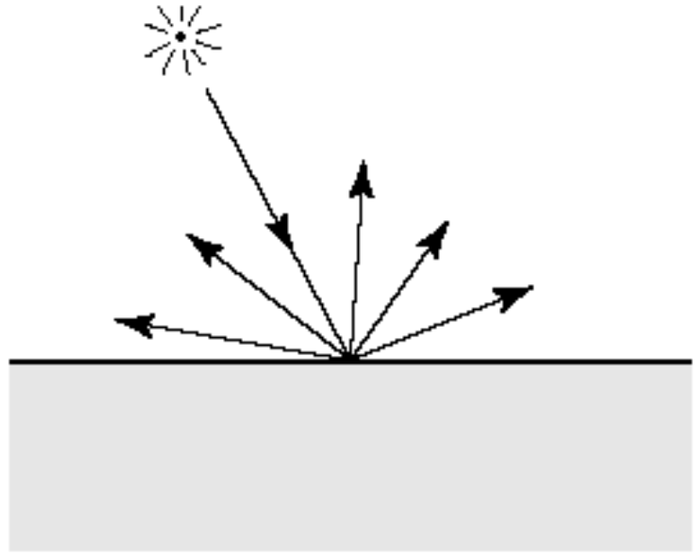
- Surfaces avec une microgéométrie fortement orientée
- Exemples:
  - Métals brossés,
  - Cheveux, fourrures, vêtements



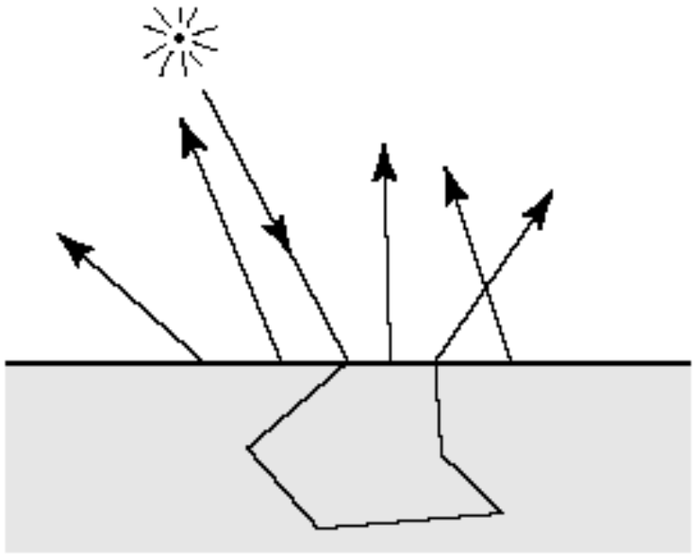
Source: Westin et.al 92

Autres phénomènes

- Réfraction de Fresnel
  - Tendance d'une surface à être plus réflective pour les angles d'observation proche de 0
  - Peut se traduire par la BRDF
- Choses que la BRDF néglige
  - Surface non uniforme
  - Subsurface scattering
    - Transport de la lumière à l'intérieur de la surface, et résurgence à un autre endroit,
    - ex : marbre, peau
  - Transmission
    - La lumière traverse l'objet (notion de **réfraction**, voir lancer de rayons)



BRDF



courtesy of
H. Jensen

BSSRDF (Bidirectional Surface Scattering Distribution Function)

Exemple: Subsurface scattering

BRDF

BSSRDF

figures extraites de Jensen et al. (2001)

Subsurface Scattering

BSSRDF

Lait pasteurisé et demi-écrémé avec BSSRDF; lait avec BRDF

Shading

Lighting and Shading

Lighting: interaction between material and light sources as well as the interaction with the geometry of the objects to be rendered (computed thanks to an object BRDF = Bi-Directional Reflectance Distribution Function)

Shading: process of performing lighting computations and determining pixels’ colors from them

Lissage de facettes : shading

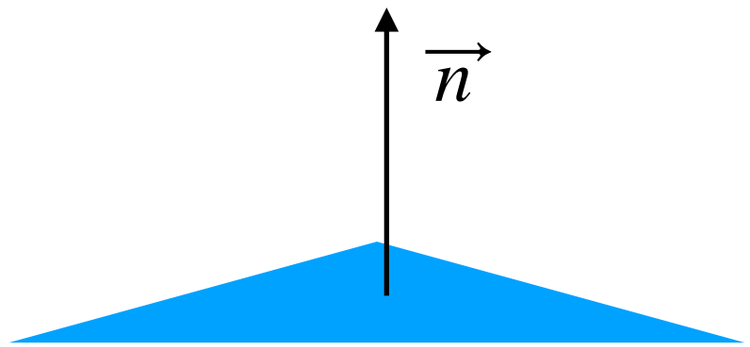
- Comment interpoler en chaque fragment la valeur de chaque couleur calculée sur un sommet du maillage ?
  - généralement par une interpolation bilinéaire
- Cette opération de remplissage est appelée lissage (ou ombrage)

- 3 types de lissage :
  - Flat Shading (light per polygon)
  - Lissage de Gouraud (light per vertex)
    - interpolation bilinéaire des couleurs des sommets de la primitive en chaque fragment
  - Lissage de Phong (light per fragment)
    - interpolation **des normales** en chaque sommet, puis calcul d'un modèle d'éclairage en chaque fragment

Flat Shading

Light per polygon

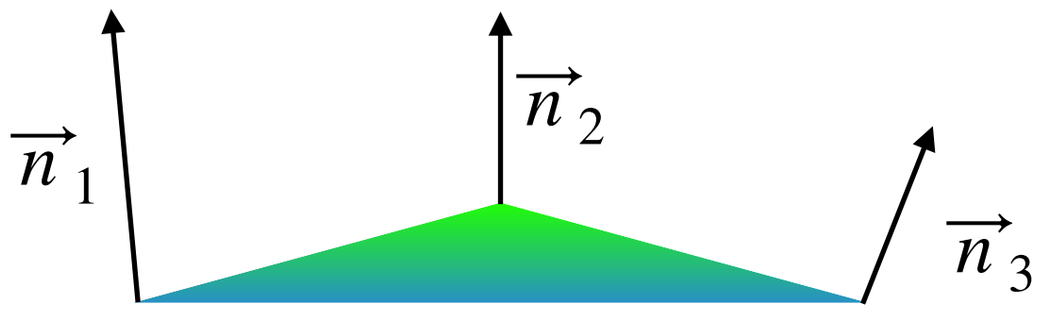
- One color per polygon
  - One normal per polygon



Smooth Shading

Light per vertex

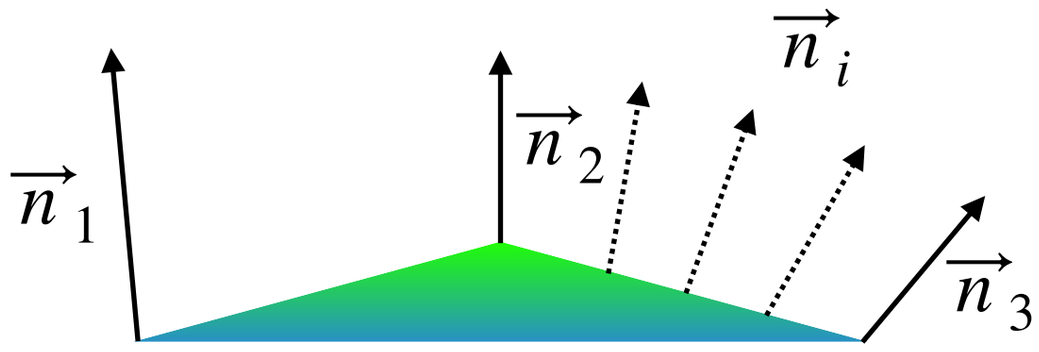
- Also called Gouraud Shading
- Basic smooth shading in graphics libraries like OpenGL
- One color per vertex
  - One normal per vertex
  - Color inside the polygon is interpolated



Phong Shading

Light per pixel

- Also called light per fragment
- One color per fragment
  - One normal per vertex
  - Normal inside the polygon is interpolated
  - Color is computed from the interpolated normal



Comparison

Per vertex / per pixel lighting

- Une implémentation possible de ce lissage est un calcul de l'éclairage se fait au niveau du pixel !
  - Et non plus au niveau du vertex
  - Un calcul d'éclairage en chaque point de l'image !

per vertex lighting

per fragment lighting

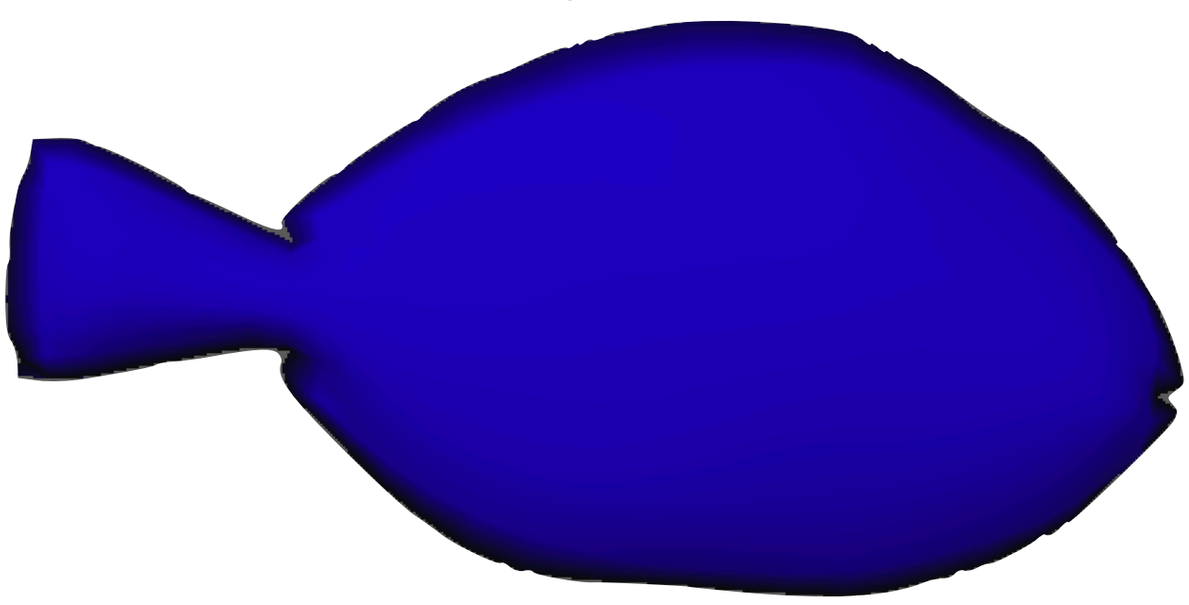
Overview

1. Rendering Pipeline
2. Light Sources
3. Material
4. Lighting
5. Shading
6. Textures
7. Shaders (GLSL)

6. Textures

Texture

- Principle: paint an apparence on an object (like adding a wallpaper)
- Textures are applied in 3D and undergo viewing transformations
- Rendering more natural than materials only
- Results depends on the image quality



What is it ?

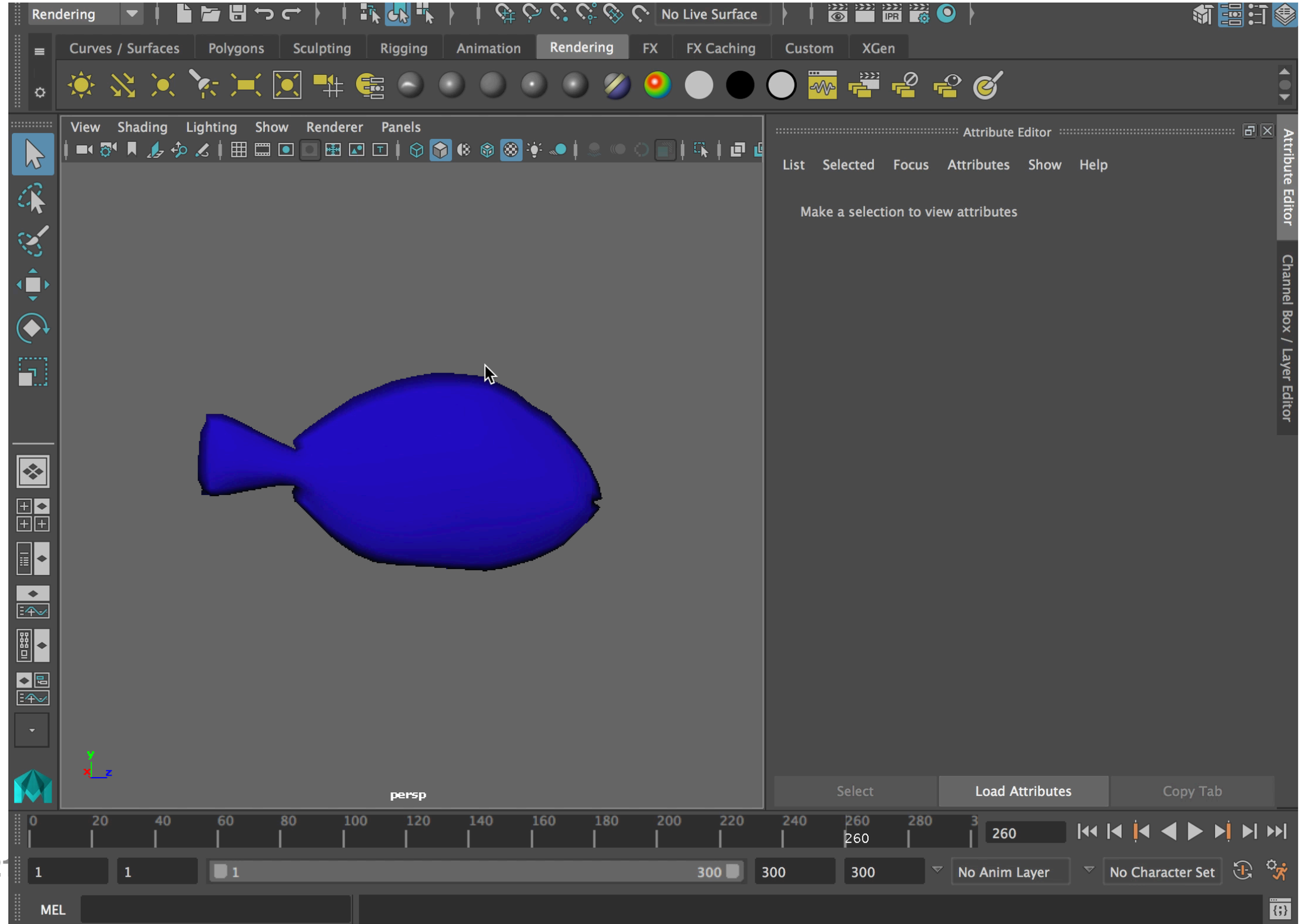
- Texture is a rectangular array of data
  - color data
  - luminance data
  - color and alpha data
- Individual values in the texture array are texels

Texture Mapping

1. Create a rectangular (square) image of data (texture)
2. Project/map this image to the 3D surface: computation of UV coordinates

- Problem: rectangular texture can be mapped to non-rectangular regions

Texture Mapping



UV Editor

Polygons View Select Tool Image Textures UV Sets Help

0.00 1.00 sRGB gamma 0.000 0.000

V-Axis

1

UV coordinates
U = 0.7 ; V = 0.61

0.9 0.8 0.7 0.6 0.5 0.4 0.3 0.2 0.1

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

-0.4 -0.3 -0.2 -0.1

0

1

U-Axis

Steps in Texture Mapping

- In openGL, you need to perform several steps to enable texture mapping:

1. Create a texture object and specify a texture for that object
2. Indicate how the texture is to be applied to each pixel
3. Enable texture mapping
4. Draw the scene, supplying both texture and geometric coordinates

1. Create Texture Object

```
static GLuint textureName;

glGenTextures(1, &textureName);

glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textureName);

//Parameters to say how the texture is used
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// Nice texture coordinate interpolation
glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );

// The texture clamps at the edges (no repeat)
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );

QImage glImg = QGLWidget::convertToGLFormat(img);  // flipped 32bit RGBA

 // Binds the img texture...
glTexImage2D(GL_TEXTURE_2D, 0, 3, glImg.width(), glImg.height(), 0,
              GL_RGBA, GL_UNSIGNED_BYTE, glImg.bits());
```

2. Indicate How the Texture is Applied

- 4 possible modes
  - decal: the texture color is the final color of the fragment
  - replace
  - modulate: combines the effect of lighting and texturing
  - constant: a constant color blended with the fragment color in function of the texture value

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

3. Enable Texture Mapping

- glEnable(GL_TEXTURE_1D);
- glEnable(GL_TEXTURE_2D);
- glDisable(GL_TEXTURE_1D);
- glDisable(GL_TEXTURE_2D);

4. Draw the Scene

```
glBegin(GL_QUADS);
   glTexCoord2f(0.0,    1.0-v_max);
   glVertex2i(0,0);
   glTexCoord2f(0.0,    1.0);
   glVertex2i(0,height());
   glTexCoord2f(u_max, 1.0);
   glVertex2i(width(),height());
   glTexCoord2f(u_max, 1.0-v_max);
   glVertex2i(width(),0);
glEnd();
```

Overview

1. Rendering Pipeline
2. Light Sources
3. Material
4. Lighting
5. Shading
6. Textures
7. Shaders (GLSL)

7. Shaders (GLSL)

Rappel : le pipeline de rendu

- Transformations successives des objets et de leurs attributs en image
- Réalisées en majeur partie sur la carte graphique

- Idée de la programmation du pipeline :
  - Configurer certains éléments du pipe pour optimiser les calculs
  - Obtenir un rendu spécifique

Collection de shaders

- Il est ainsi possible d'associer à différents objets de la scène plusieurs méthodes de rendu
  - Intégration des shaders dans le graphe de scène

- Bibliothèque de matériaux → bibliothèque de shaders
  - Intégration dans la plupart des outils de modélisation/rendu (Maya, Max,…)

Historique

- Premier outil permettant de programmer des « shaders » : Renderman Shading Language (RSL)
  - Contrôle des attributs du rendu
  - Pas de programmation au niveau hardware
  - Gérer en interne dans le moteur de rendu

Historique

- Programmation du hardware
  - Assembleur : depuis 2000
  - Cg (NVidia) : 2002
  - HLSL : DirectX **H**igh **L**evel **S**hading **L**anguage 2003
  - OpenGL 1.5 et GlSlang (OpenGL 2.0) 2004

Pourquoi des langages de programmation ?

Programmer en assembleur est dur

- Les cartes actuelles supportent des tailles de programme > 1000 lignes de code
- Plus facile à écrire, à réutiliser
- Plus facile à débugger

Exemple : modèle de phong

Assembleur

```
...
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

Version CG

```
float3 cSpecular = pow(max(0, dot(Nf,
H)),
                       phongExp).xxx;
float3 cPlastic = Cd * (cAmbient +
cDiffuse) +
                  Cs * cSpecular;
```

Vertex shaders

Exécuté une fois pour chaque vertex

Transformation des coordonnées 3D en coordonnées 2D

- Passage dans le repère de la caméra (*modelview*)
- Calcul de normales, etc.
- Illumination des sommets
- Calcul des coordonnées de texture, transformation des textures

Que peut on faire avec des vertex shaders ?

- Déplacer les points (ex : skinning)
- Faire des calculs de normales particuliers (ex : bump mapping)

Fragment shaders

- Exécuté une fois pour chaque fragment (pré-pixel)
- Calcul de la couleur finale :
  - Application de texture
  - Calcul de la profondeur finale

- Limitations :
  - Aucun fragment ne peut être généré (issu de la rasterisation)
  - Pas d'information sur l'objet auquel il appartient
  - Positions des fragments fixes

- Que peut on faire avec des fragment shaders ?
  - Texture procédurale,
  - Per-pixel lighting
  - …

Etude de cas : GLSL

- OpenGL Shading Language
- Bibliothèques de fonctions type C
- Dispose de ses propres structures de données
  - Vecteurs
  - Matrices
  - Textures
- Accès intégré aux états OpenGL

Simple Shader Example

Color

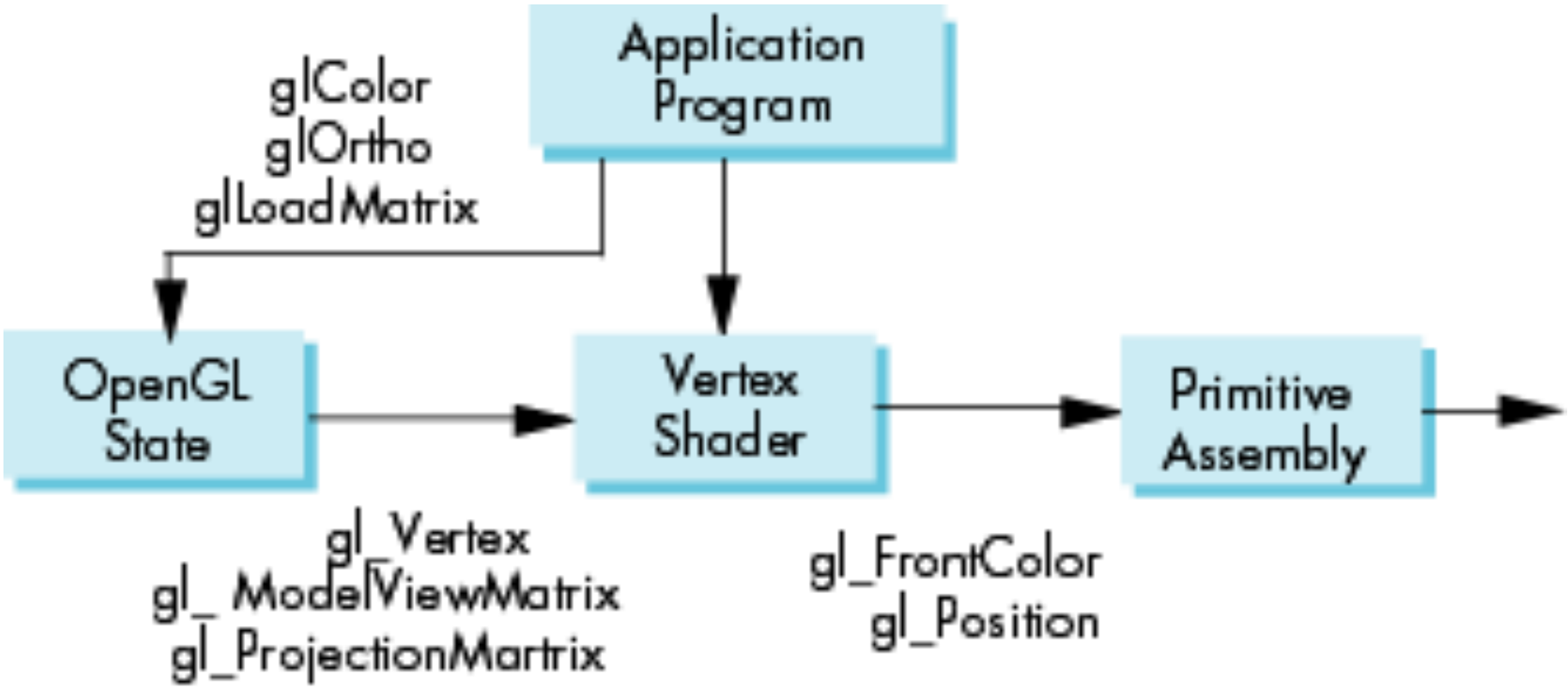
Vertex Shader

```
varying vec4 col;

void main(void)
{
   gl_Position = ftransform();
   //Equivalent to gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;

   col = vec4(0.0, 1.0, 1.0, 1.0);
}
```

Fragment Shader

```
varying vec4 col; //varying variables have the same name in vertex and fragment shaders

void main(void)
{
   gl_FragColor = col;
}
```

Modèle d’éxécution



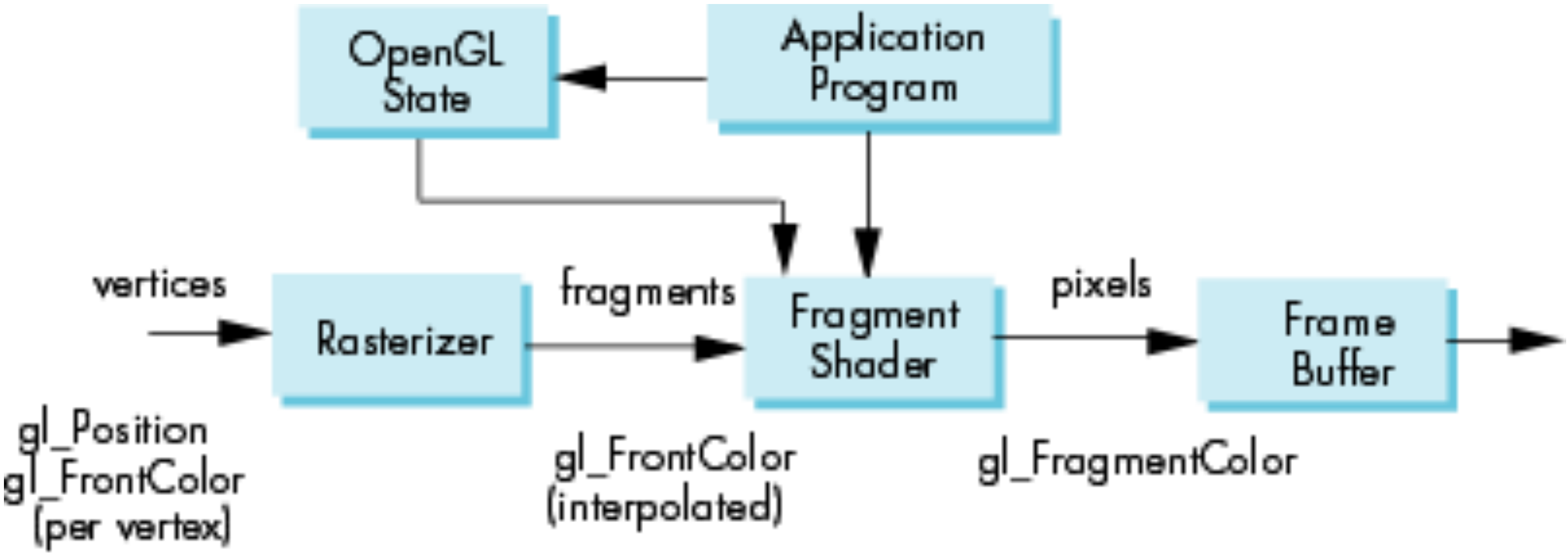
Simplest Vertex Shader

Transforms Vertices

Vertex Shader

```
void main(void)
{
   gl_Position = ftransform();
 }
```

Modèle d'éxécution



Simplest Fragment Shader

Propagates Front Color

Fragment Shader

```
void main(void)
{
   gl_FragColor = gl_Color;
}
```

Types de données

- Types C : int, float, bool
- Vecteurs :
  - float vec2, vec3, vec4
  - Aussi vecteurs d'entiers ou de booléens
- Matrices: mat2, mat3, mat4
  - Stockées par colonnes
  - Référencement standard m[row][column]
- Constructeurs de style C++
  - vec3 a =vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

Pointeurs

- Pas de pointeurs en GLSL
- Utilisation de structures type C
  - Passage en paramètre dans les fonctions
- Mais aussi appel de fonctions avec des types de base
  - Ex : passage d'une mat3 en paramètre d'entrée de ou de sortie de fonction

```
mat3 myFunc(mat3 myMatrix)
```

Qualificatifs

- Qualifie la nature d'une variable
- Les variables peuvent changer
  - Une fois par primitive
  - Une fois par vertex
  - Une fois par fragment
  - A n'importe quel moment dans l'application

- Les attributs des vertices sont interpolés après la rasterisation

Qualificatifs Attribute

- Les variables Attribute peuvent changer au plus une fois par vertex
  - pas accessible dans les fragment shaders

- Variables d'état OpenGL
  - `gl_Color`
  - `gl_ModelViewMatrix`

- Définis par l'utilisateur (dans le programme)
  - `attribute float temperature`
  - `attribute vec3 velocity`

Qualificatifs Uniform

- Variables constantes pour une primitive entière

- Peuvent être modifiées hors d’un **`glBegin`** et **`glEnd`**

- Pas modifiables dans le shader

Qualificatifs Varying

- Variables passées entre le vertex et le fragment shader
- Automatiquement interpolées au moment de la rasterisation

- Prédéfinis :
  - Couleurs des Vertex (gl_Color)
  - Coordonnées de texture (gl_TexCoord[num texture])

- Définis par l'utilisateur
  - Nécessite l'équivalent dans le fragment shader

Exemple : Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
varying vec3 color_out;

void main(void)
{
  gl_Position =
   gl_ModelViewProjectionMatrix*gl_Vertex;
  color_out = red;
}
```

Exemple : Fragment Shader

```
varying vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
```

Fonctions: transmission de valeurs

- Passées par un Return
- Transmission par copie
- 3 possibilités :
  - **in**
  - **out**
  - **inout**

Functions: example

```
void MyFunction(in float inputValue, out int outputValue, inout float inAndOutValue)
{
  inputValue = 0.0;
  outputValue = int(inAndOutValue + inputValue);
  inAndOutValue = 3.0;
}

void main()
{
  float in1 = 10.5;
  int out1 = 5;
  float out2 = 10.0;
  MyFunction(in1, out1, out2);
}
```

Opérateurs et fonctions

- Fonctions C standard
  - Trigonométrique
  - Arithmétique
  - Normalisation, longueur d'un vecteur, etc…
- Surcharge pour les types prédéfinis :

```
mat4 a;
vec4 b, c, d;
c = b*a; // Vecteur colonne
d = a*b; // vecteur ligne
```

Swizzling et sélection

- Accès à un élément de tableau par [] ou (.) avec
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - `a[2], a.b, a.z, a.p` sont identiques
- L'opérateur de Swizzling permet de manipuler les composants facilement

```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

Lier un Shader à OpenGL

- Extensions OpenGL
  - ARB_shader_objects
  - ARB_vertex_shader
  - ARB_fragment_shader
- OpenGL 2.0
  - identique en terme d'utilisation
  - pas besoin d'utiliser les suffixes

Objet programme

- Encapsulation du shader
  - Peut contenir plusieurs shaders

```
GLuint myProgObj;
myProgObj = glCreateProgram();
 /* define shader objects here */
glUseProgram(myProgObj);
glLinkProgram(myProgObj);
```

Lire un Shader

- Les shaders sont ajoutés à l’objet programme et compilés

```
glShaderSource( … )
```

Lecture d'un shader

- Construit un buffer contenant le code source du shader

```
char* readShaderSource(const char* shaderFile){
    struct stat statBuf;
    FILE* fp = fopen(shaderFile, "r");
    char* buf;
 stat(shaderFile, &statBuf);
buf = (char*) malloc(statBuf.st_size + 1 * sizeof(char));
fread(buf, 1, statBuf.st_size, fp);
buf[statBuf.st_size] = '\0';
fclose(fp);
return buf;}
```

Ajouter un vertex shader

```
GLint vShader;
GLunit myVertexObj;
GLchar vShaderfile[] = "my_vertex_shader";
GLchar* vSource = readShaderSource(vShaderFile);

myVertexObj = glCreateShader(GL_VERTEX_SHADER);

glShaderSource(myVertexObj,1, vSource, NULL);

glCompileShader(myVertexObj);

glAttachObject(myProgObj, myVertexObj);
```

Attributs de vertex

- Les attributs de vertex sont nommés dans les shaders
- L’éditeur de liens construit une table
- L’application peut lire cette table et lier la variable de l’application

  - Processus identique pour les variables uniformes

Exemple

```
GLint colorAttrib;
colorAttrib = glGetAttribLocation(myProgObj,
        "myColor");
/* myColor est le nom de la variable dans le
shader */


GLfloat color[4];
glVertexAttrib4fv(colorAttrib, color);
/* color est la variable dans l'application */
```

Exemple 2 : variable uniforme

Creation de la *variable location* et association avec la variable du shader (init du programme)

```
GLint angleParam;
angleParam = glGetUniformLocation(myProgObj,
      "angle");
```

Mise a jour de la variable dans le program

```
GLfloat my_angle;
my_angle = 5.0 ;
glUniform1f(angleParam, my_angle);
```

Samplers = textures

- Défini pour des textures 1, 2 ou 3 D

- Dans le shader:

```
uniform sampler2D myTexture;
Vec2 texcoord;
Vec4 texcolor = texture2D(myTexture,texcoord);
```

- Dans l'application :

```
texMapLocation =
 glGetUniformLocation(myProg,"myTexture");
glUniform1i(texMapLocation, 0);
/* associe à l'unité de texture 0 */
```

Exemples de shaders

- Scaling
- Génération de perturbations sinusoidales sur un objet
- Systèmes de particules
- Shading
    - Per-vertex (Gouraud Shading)
    - Per-pixel (Phong Shading)

Shaders

Vertex Shader

```
void main(void)
{
  vec4 v = gl_Vertex;
  v.x = v.x * 2.0;
  v.z = v.z * 0.5;
  gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Fragment Shader

```
void main(void)
{
  gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Shaders Scale

Vertex Shader

```
void main(void)
{
  vec4 v = gl_Vertex;
  v.x = v.x * 2.0;
  v.z = v.z * 0.5;
  gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Fragment Shader

```
void main(void)
{
  gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Vertex Shader

```
Vertex Shader

uniform float time;

void main()
{
   vec4 v = gl_Vertex;
   float s = 1.0 + 0.1*sin(v.s*time)*sin(v.z*time);
   v.y = s * v.y;
   gl_Position = gl_ModelViewProjectionMatrix * v;
}

Fragment Shader

void main()
{
   gl_FragColor = gl_Color;
}
```

Vertex Shader : Wave

Vertex Shader

```
uniform float time;

void main()
{
  vec4 v = gl_Vertex;
  float s = 1.0 + 0.1*sin(v.s*time)*sin(v.z*time);
  v.y = s * v.y;
  gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

Fragment Shader

```
void main()
{
   gl_FragColor = gl_Color;
}
```

Vertex Shader

Vertex Shader

```
attribute vec3 vel;
uniform float time;

void main()
{
  float g = 0.0981;
  float m = 1.0;
  vec3 object_pos;
  object_pos.x = gl_Vertex.x + vel.x*time;
  object_pos.y = gl_Vertex.y + vel.y*time - g/m*time*time;
  object_pos.z = gl_Vertex.z + vel.z*time;
  gl_Position = gl_ModelViewProjectionMatrix * vec4(object_pos,1);
}
```

Fragment Shader

```
void main()
{
  gl_FragColor = gl_Color;
}
```

Vertex Shader: Particle System

Vertex Shader

```
attribute vec3 vel;
uniform float time;

void main()
{
  float g = 0.0981;
  float m = 1.0;
  vec3 object_pos;
  object_pos.x = gl_Vertex.x + vel.x*time;
  object_pos.y = gl_Vertex.y + vel.y*time - g/m*time*time;
  object_pos.z = gl_Vertex.z + vel.z*time;
  gl_Position = gl_ModelViewProjectionMatrix * vec4(object_pos,1);
}
```

Fragment Shader

```
void main()
{
  gl_FragColor = gl_Color;
}
```

Gouraud (per vertex) - Vertex Shader I

```
/* modified Gouraud vertex shader (without distance term) */
 void main(void)
{
  float f;
/* compute normalized normal, light vector, view vector,
        half-angle vector in eye coordinates */

  vec3 norm = normalize(gl_NormalMatrix * gl_Normal);
 vec3 lightv = normalize(gl_LightSource[1].position.xyz
- (gl_ModelViewMatrix * gl_Vertex).xyz);
 vec3 viewv = - normalize( (gl_ModelViewMatrix * gl_Vertex).xyz);
 vec3 halfv = normalize( lightv + viewv);

 if (dot(lightv, norm) > 0.0)
        f = 1.0;
 else
        f = 0.0;
```

Gouraud (per vertex) - Vertex Shader II

```
/* compute diffuse, ambient, and specular contributions */

vec4 ambient = gl_FrontMaterial.ambient * gl_LightSource[1].ambient;

vec4 diffuse = gl_FrontMaterial.diffuse *
gl_LightSource[1].diffuse * dot(lightv,norm);

vec4 specular = gl_FrontMaterial.specular *
gl_LightSource[1].specular * pow(dot(norm,halfv),
gl_FrontMaterial.shininess);

  vec4 color = f*(ambient+diffuse+specular);
  color.a = 1.0;

  gl_FrontColor = color;
  gl_Position = ftransform();
}
```

Gouraud (per vertex) - Fragment Shader : identité

/* pass-through fragment shader */

```
void main(void)
{
  gl_FragColor = gl_Color;
}
```

Phong (per pixel) - Vertex Shader

```
/* vertex shader for per-fragment Phong shading */
varying vec3 normale;
varying vec4 positione;

void main(){
   normale = gl_NormalMatrix*gl_Normal;
   positione = gl_ModelViewMatrix*gl_Vertex;
   gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

Phong (per pixel) - Fragment Shader

```
varying vec3 normale;
varying vec4 positione;

void main(){
vec3 norm = normalize(normale);
vec3 lightv = normalize(gl_LightSource[0].position-positione.xyz);
vec3 viewv = normalize(positione);
vec3 halfv = normalize(lightv + viewv);
vec4 diffuse = max(0, dot(lightv, viewv))*gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
vec4 ambient = gl_FrontMaterial.ambient*gl_LightSource[0].ambient;

vec3 specular = pow(max(0, dot(norm, halfv), gl_FrontMaterial.shininess)
                    * gl_FrontMaterial.specular*gl_LightSource[0].specular);
vec3 color = vec3(ambient + diffuse + specular);

gl_FragColor = vec4(color, 1.0);
}
```