[CSE4170 기초 컴퓨터 그래픽스]

2019년도 1학기

강의자료 III

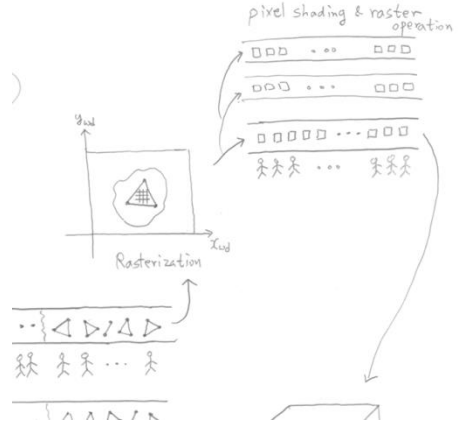
Rasterization:

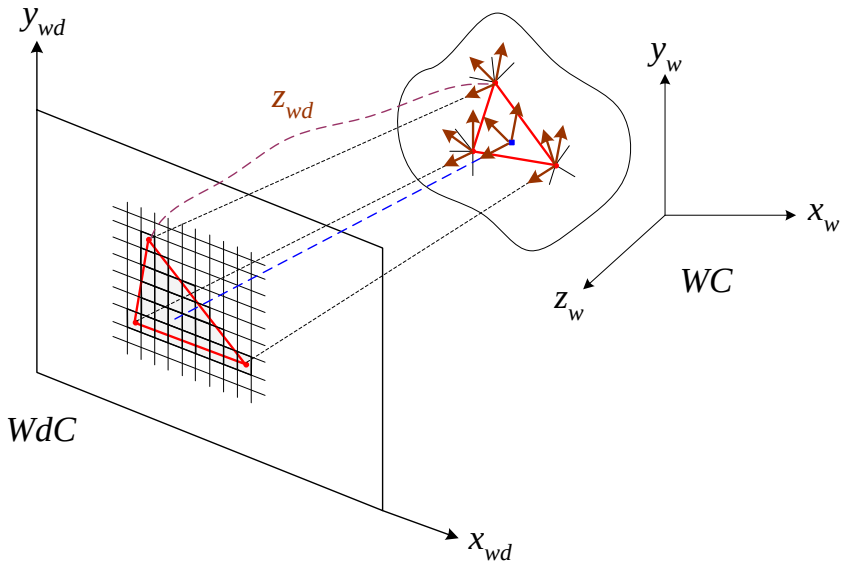
From Geometric Primitives to Fragments

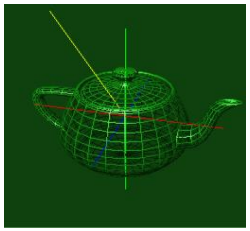
Rasterization and Pixel Shading

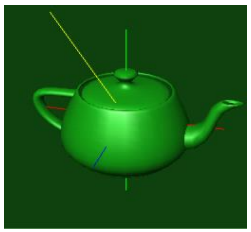
From Continuous to Discrete Spaces

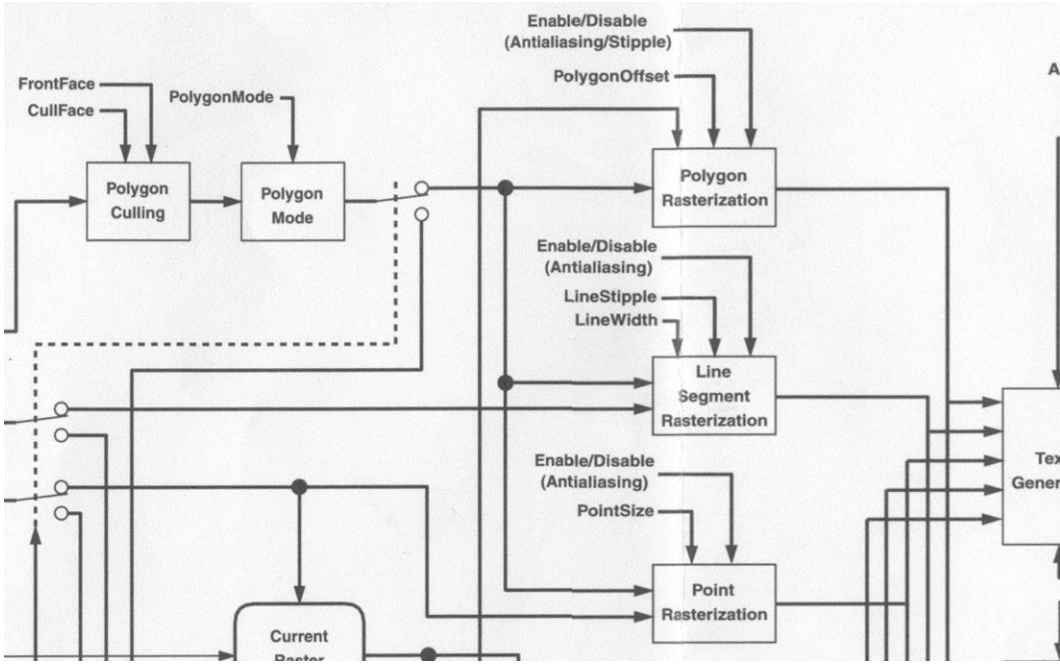
- WdC까지 흘러온 기하 프리미티브가 자신이 차지하는 pixel (fragment)로 쪼개짐 → **Rasterization**
  - 이후 각 프래그먼트에 칠할 최종 색깔을 결정 → **Pixel shading**
  - Pixel shading 이후 프레임 버퍼로 들어가기 전에 프래그먼트에 대하여 "어떤" 계산을 수행 → **Raster operation (per-fragment operation)**





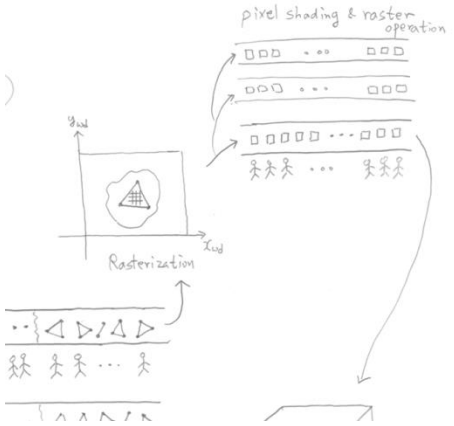


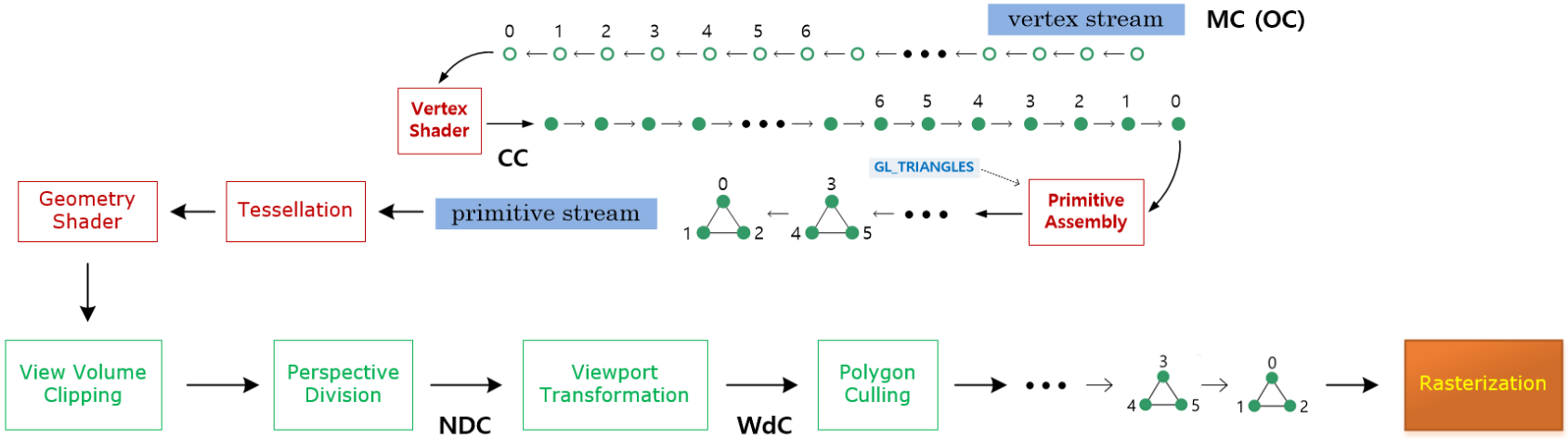


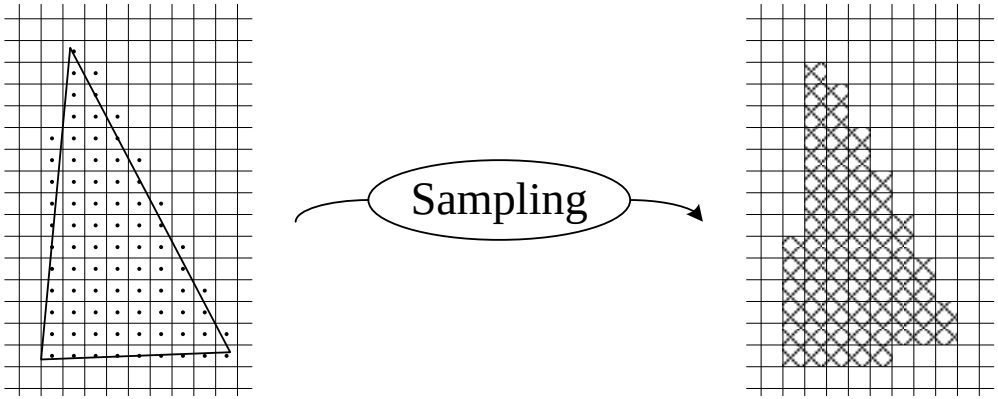


Rasterization and Per-Fragment Attributes

I. Rasterization 직전 각 primitive의 각 vertex에는 다음과 같은 정보가 붙어 있음 → Per-vertex attributes.
   ① Position: (x, y)
   ② Depth: z
   ③ w: $1/w$
   ④ Normal: (nx, ny, nz)
   ⑤ Color: (r, g, b, a)
   ⑥ Texture coordinate: (s, t, r, q)
   ⑦ **기타**







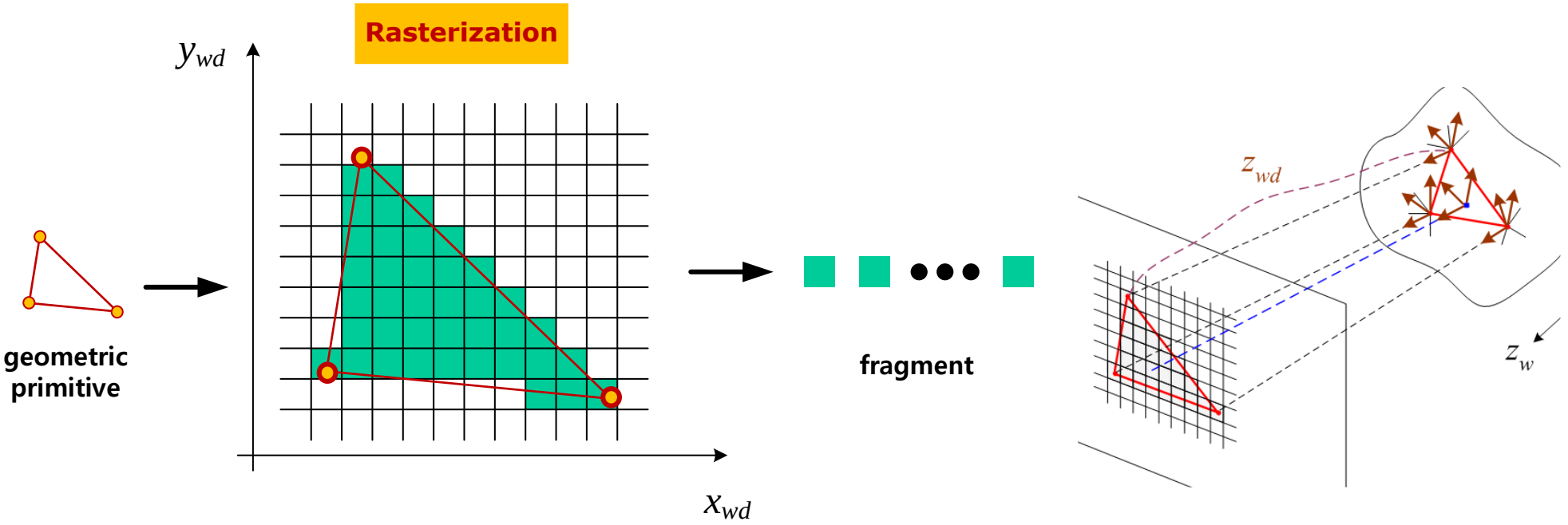
II. Rasterization 과정 시 primitive가 투영되는 pixel들의 위치뿐만 아니라, 나머지 속성 정보에 대해서도 **선형보간(linear interpolation, lerp)**을 통하여 각 fragment에 대한 속성 정보를 추정함.

III. 그 결과 per-fragment attribute를 가지는 fragment들이 생성이 됨.

✓ 일반적으로 각 fragment의 속성은 해당 픽셀을 통하여 보이는 물체 지점에서의 여러 성질을 저장하고 있음.

From Geometric Primitive to Fragments

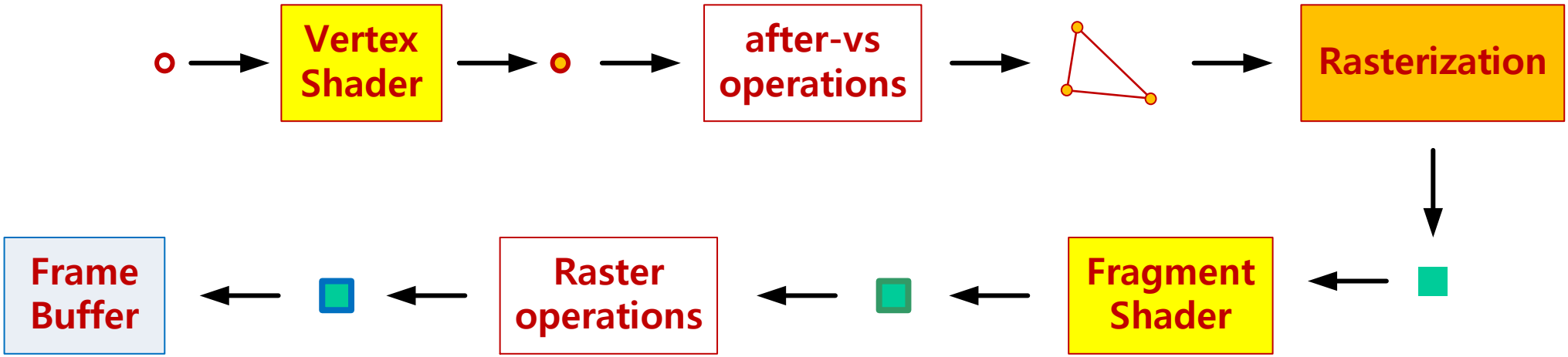
- Vertex attribute들과 fragment attribute들간의 정보 전달 과정



Rasterization 과정에서 꼭지점이 가지고 있는 성질들, 즉 vertex attribute들이

어떠한 방식으로 fragment attribute 형태로 프래그먼트들에게 전달이 될까?

Rendering Pipeline Overview



Rendering pipeline에서 vertex shading/fragment shading과 Rasterization 과정과의 관계에서 대하여 다시 한 번 상기하자.

Vertex Shader and Vertex Attributes

```
#version 330

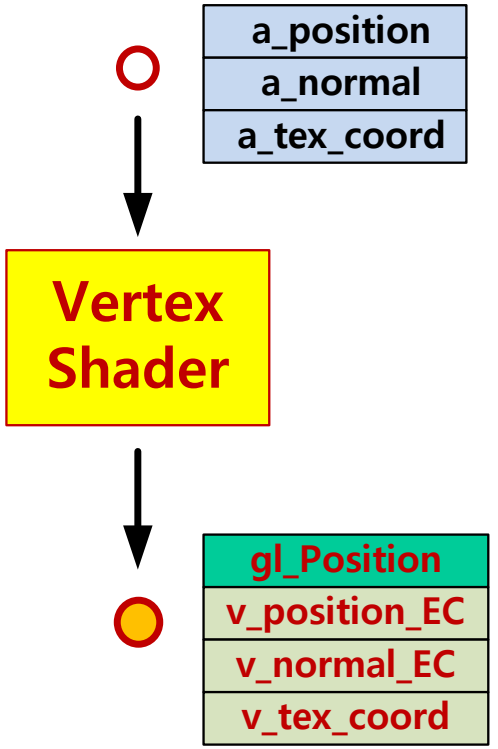
uniform mat4 u_ModelViewProjectionMatrix;
   ...

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_tex_coord;

out vec3 v_position_EC;
out vec3 v_normal_EC;
out vec2 v_tex_coord;

void main(void) {
        v_position_EC = vec3(u_ModelViewMatrix*vec4(a_position, 1.0f));
        v_normal_EC = normalize(u_ModelViewMatrixInvTrans*a_normal);
        v_tex_coord = a_tex_coord;

        gl_Position = u_ModelViewProjectionMatrix*vec4(a_position, 1.0f);
}
```



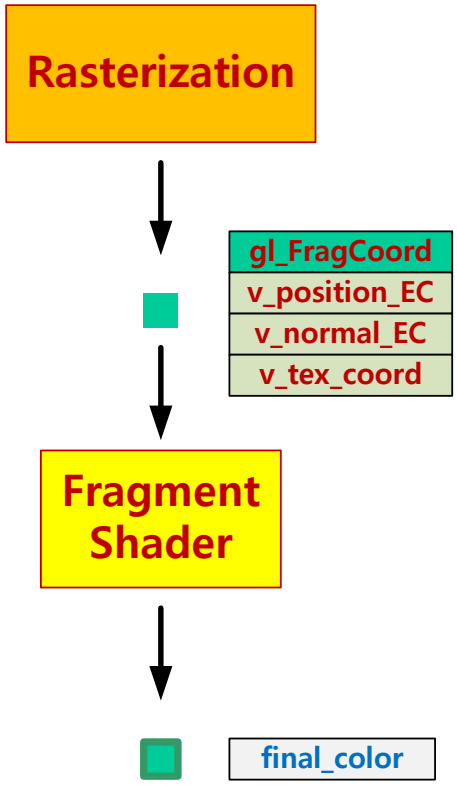
Fragment Shader and Fragment Attributes

```
#version 330
 ...
in vec3 v_position_EC;
in vec3 v_normal_EC;
in vec2 v_tex_coord;

layout (location = 0) out vec4 final_color;
 ...
void main(void) {
        vec4 base_color, shaded_color;
        float fog_factor;

        if (u_flag_texture_mapping)
                base_color = texture(u_base_texture, v_tex_coord);
        else
                base_color = u_material.diffuse_color;
        shaded_color = lighting_equation_textured(v_position_EC,
                                                  normalize(v_normal_EC), base_color);

        if (u_flag_fog) {
                fog_factor = (FOG_FAR_DISTANCE - length(v_position_EC.xyz))
                                /(FOG_FAR_DISTANCE - FOG_NEAR_DISTANCE);
                fog_factor = clamp(fog_factor, 0.0f, 1.0f);
                final_color = mix(FOG_COLOR, shaded_color, fog_factor);
        }
        else
                final_color = shaded_color;
}
```

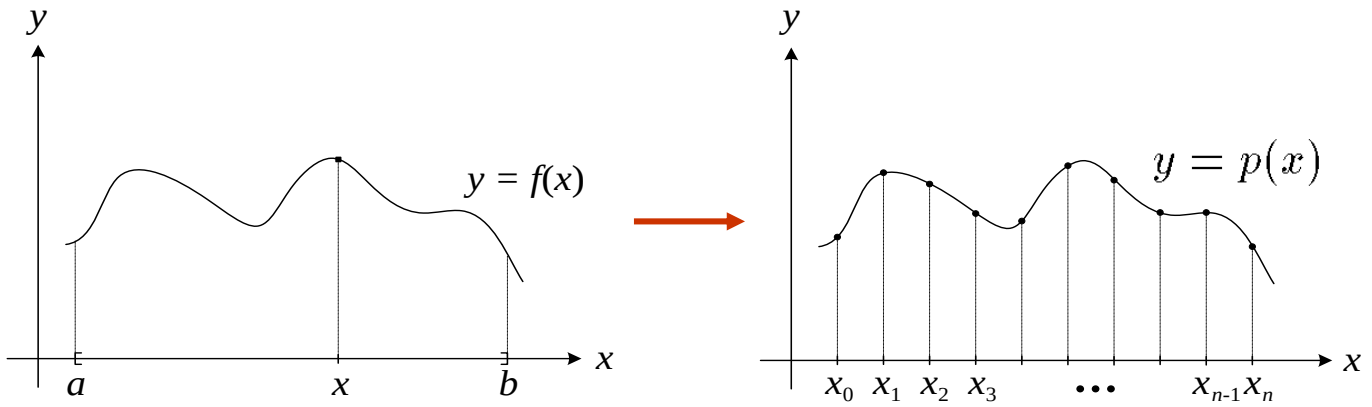


Interpolation by Polynomial (다항식 보간)

- Problem

  Given the values y_i $(i = 0, 1, 2, \cdots, n)$ of a function $y = f(x)$ at $n+1$ (distinct) points x_i $(i = 0, 1, 2, \cdots, n)$, find a polynomial $p(x)$ of degree $\leq n$ such that $p(x_i) = y_i$ for all $0 \leq i \leq n$.

x	x_0	x_1	x_2	$\cdots$	x_{n-1}	x_n
y	y_0	y_1	y_2	$\cdots$	y_{n-1}	y_n



- ✓ 어떤 원하는 정보를 $y = f(x)$, $z = f(x, y)$, ... 등과 같이 수학적으로 모델링 하였을 때, 많은 경우 정확한 수식을 모르거나 매우 복잡한 형태로 표현됨.
- ✓ 이러한 함수(정보)를 효과적으로 다루기 위하여 '샘플링 후 보간' 기법을 사용함.
- ✓ Why polynomial?

Linear Interpolation (선형 보간)

- Use $y = p_1(x) \equiv a_1 x + a_0 \qquad (n = 1)$

x	x_0	x_1
y	y_0	y_1

$$f(x) \approx p_1^0(x) \quad = \quad y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) = \frac{x_1 - x}{x_1 - x_0} y_0 + \frac{x - x_0}{x_1 - x_0} y_1$$

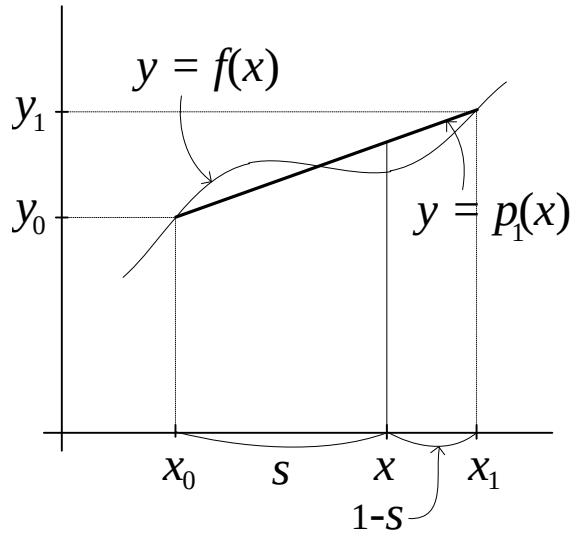
- Another representation

$$p_1^0(x) = y(s) \quad \equiv \quad (1 - s) \cdot y_0 + s \cdot y_1, \;\; \delta_y \equiv y_1 - y_0, \;\; s = \frac{x - x_0}{x_1 - x_0}$$

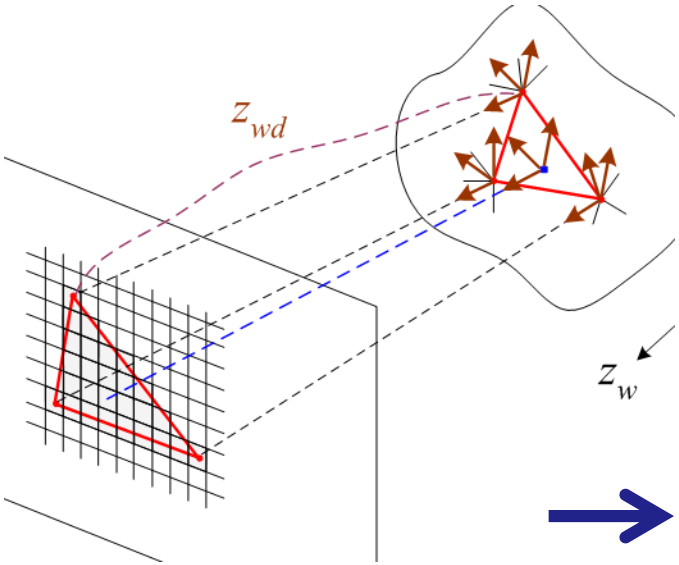
$$= \quad y_0 + s \cdot \delta_y, \;\; 0 \le s \le 1$$

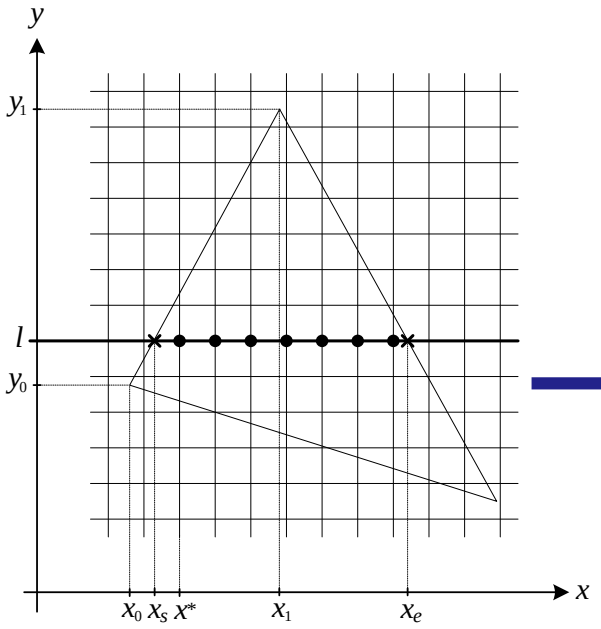
- Interpolation error

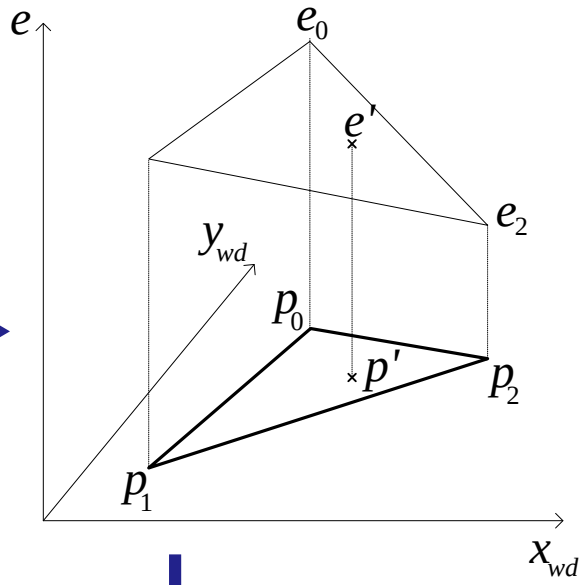
$$e(x) = \frac{f^{(2)}(\xi)}{2}(x - x_0)(x - x_1), \;\; \xi \in [x_0, x_1]$$



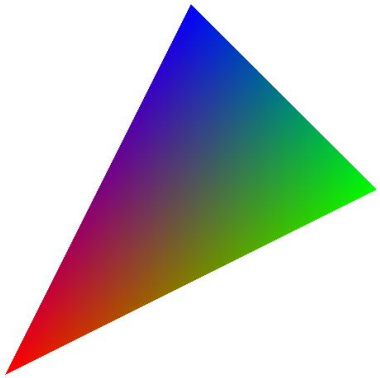
Pictorial Description of Linear Interpolation in Rasterization

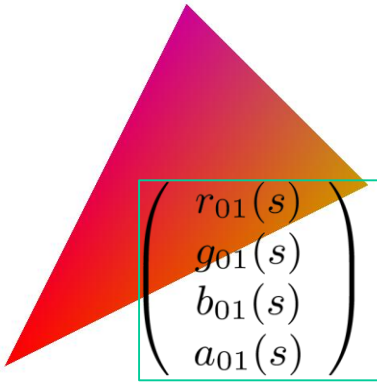






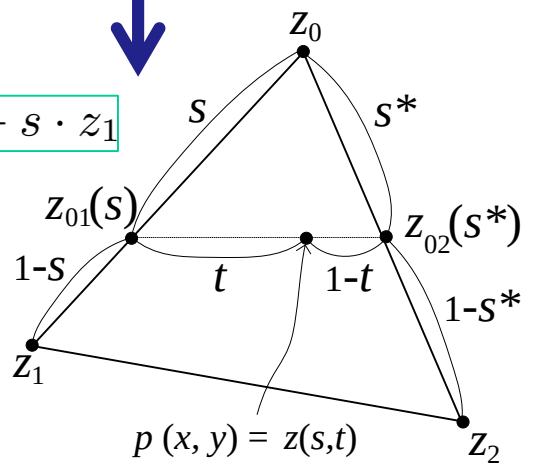
색깔 속성에 대한 선형 보간





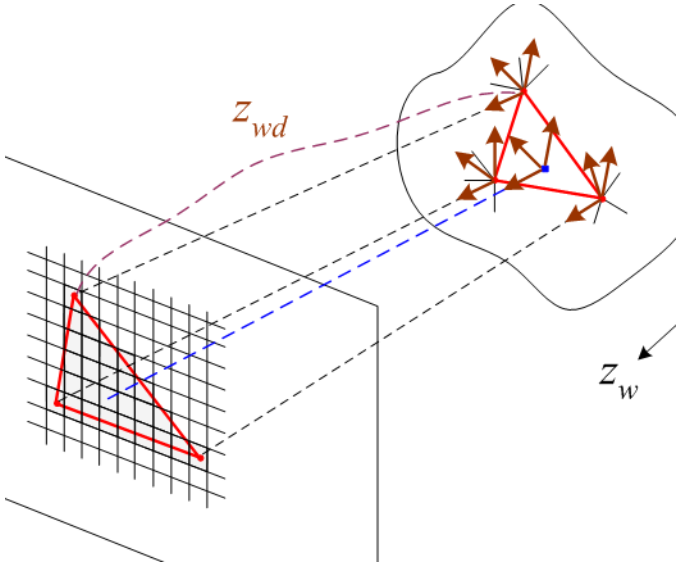
$$z_{01}(s) = (1-s)\cdot z_0 + s\cdot z_1$$

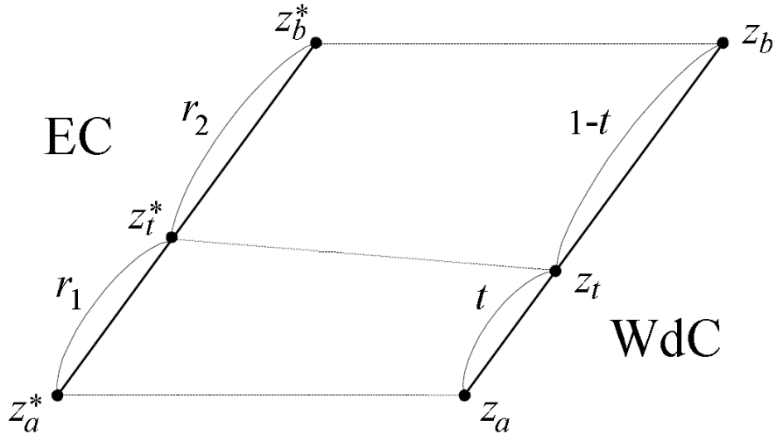
$$\begin{pmatrix} r_{01}(s) \\ g_{01}(s) \\ b_{01}(s) \\ a_{01}(s) \end{pmatrix} = (1-s)\begin{pmatrix} r_0(s) \\ g_0(s) \\ b_0(s) \\ a_0(s) \end{pmatrix} + s\begin{pmatrix} r_1(s) \\ g_1(s) \\ b_1(s) \\ a_1(s) \end{pmatrix}$$



Perspective-Corrected Interpolation in OpenGL

- Interpolation은 WdC 공간에서 수행이 되나, interpolation을 하려는 꼭지점 속성은 주로 EC, WC, MC 등의 좌표계에서 정의가 됨.
- Perspective projection을 하는 과정에서 원근감이 생기면서 공간이 왜곡 됨.
- 따라서 WdC 상에서의 거리의 비율과 실제 EC, WC, MC 등의 물체가 존재하는 좌표계 상에서의 거리의 비율이 다를 수 있음.
- 따라서 WdC가 아니라 물체가 존재하는 공간에서의 거리 비율을 사용하여 interpolation을 해야 함.





원근 교정을 통한 텍스춰 매핑

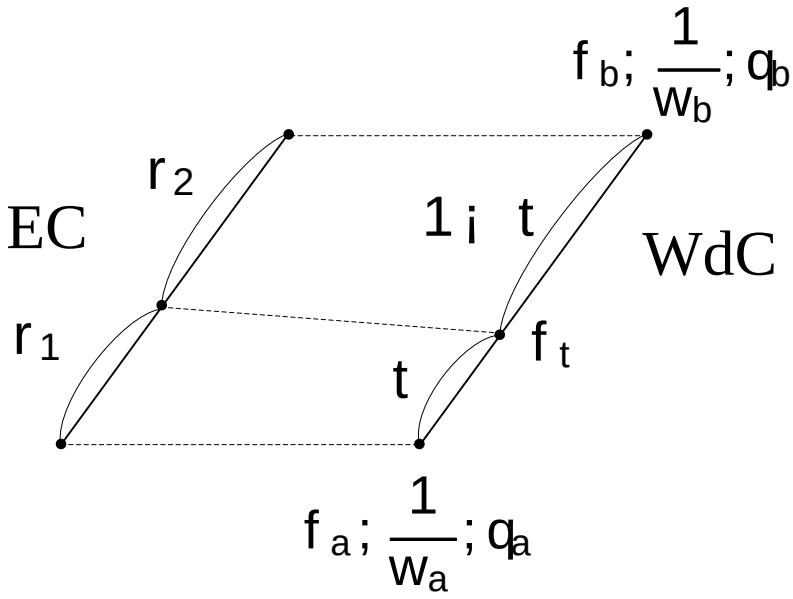
$\frac{1}{w_c}$와 $\frac{f}{w_c}$는 WdC에서 선형적으로 변함.

- **Linear interpolation with perspective correction**

$$f_t = \frac{(1-t)\cdot\frac{f_a}{w_a} + t\cdot\frac{f_b}{w_b}}{(1-t)\cdot\frac{1}{w_a} + t\cdot\frac{1}{w_b}}$$

- **For the depth value**

$$f_t = (1-t)\cdot f_a + t\cdot f_b$$



$$t : 1-t \neq r_1 : r_2 \text{ if } w_a \neq w_b$$

➢ For texture coordinates (s, t, r, q)

$$f_t = \frac{(1-t)\cdot\frac{f_a}{w_a} + t\cdot\frac{f_b}{w_b}}{(1-t)\cdot\frac{q_a}{w_a} + t\cdot\frac{q_b}{w_b}}$$

✓ What information does w carry?

$$w_c = -z_e$$

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{bmatrix} \frac{cot(\frac{fovy}{2})}{asp} & 0 & 0 & 0 \\ 0 & cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix}$$

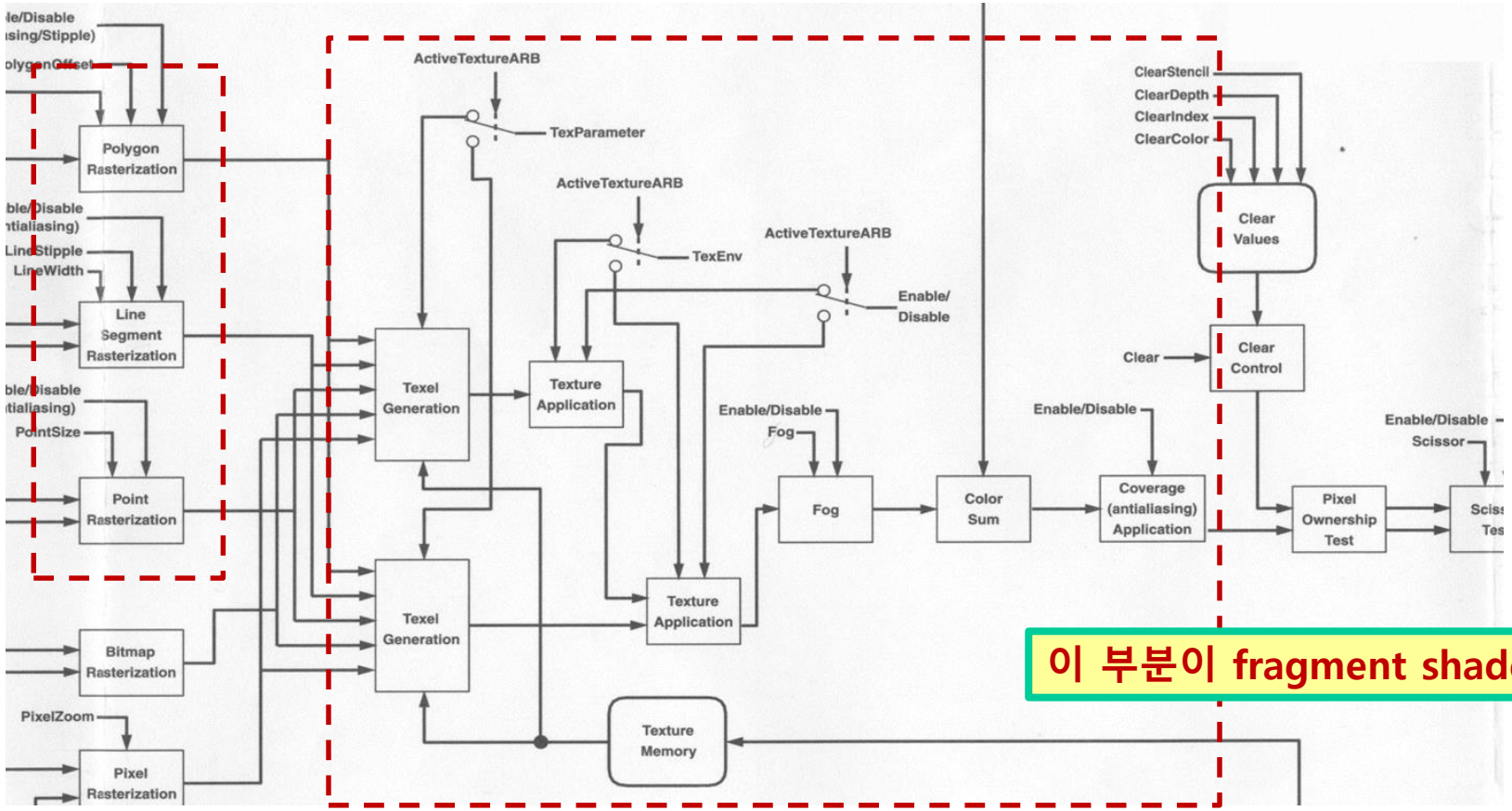
After the Rasterization Process

- 래스터화 과정을 통하여 생성되는 각 **프래그먼트(fragment)**는 해당 pixel에 대하여 다음과 같은 **per-fragment attributes**가 부여됨.

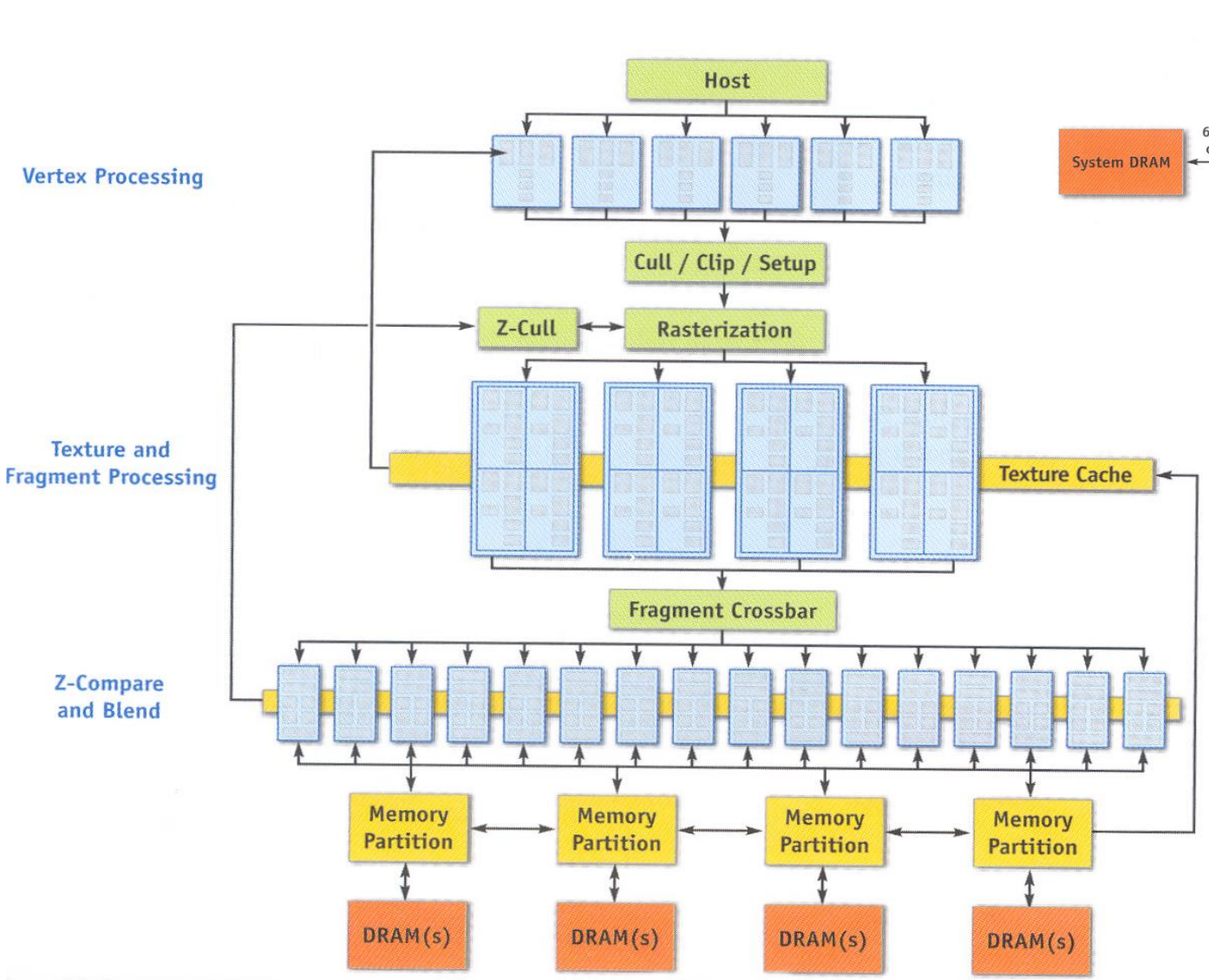
$$[(x_{ij}, y_{ij}), z_{ij}, 1/w_{ij,} (r_{ij}, g_{ij}, b_{ij}, a_{ij}), (s_{ij}, t_{ij}, r_{ij}, q_{ij}), f_{ij} \ldots]$$

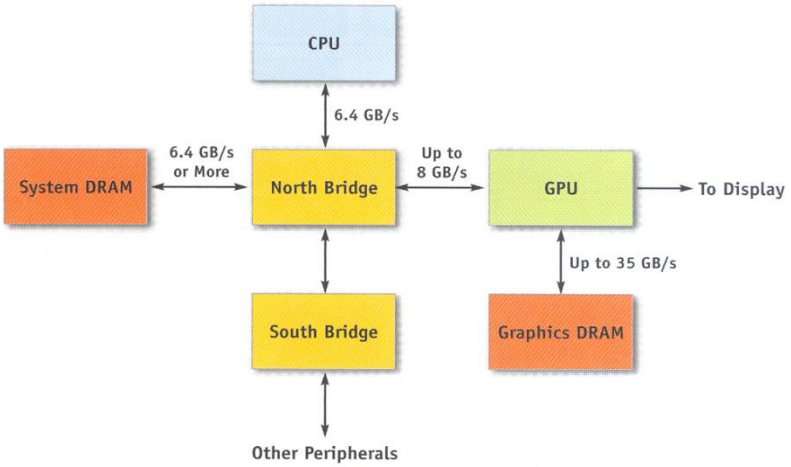
- Rendering pipeline을 따라 흘러가는 데이터의 형태가 rasterization 과정을 통하여 vertex에서 pixel 형태의 fragment로 바뀌게 됨!
  - Per-vertex & per-primitive operations → Per-pixel (per-fragment) operations

- Rasterization 과정 직후에 수행되는 계산은 바로 주어진 각 fragment에 대해 독립적으로 그 fragment에 주어진 정보를 사용하여 좌표 (x_{ij}, y_{ij}) 가 가리키는 pixel에 칠할 최종 색깔을 계산하는 것임 ← **pixel shading**



2004년: GeForce 6 Series (NV40) - 6800 Ultra





Shader Model 3.0
- standard FP32 precision
- dynamic branching
- longer shader lengths

© NVIDIA

Vertex and Fragment Attributes: NVIDIA G80

- **Vertex Program Attribute Variables (before vertex shader)**

Vertex Attribute Binding	Components	Underlying State
vertex.position	(x,y,z,w)	object coordinates
vertex.normal	(x,y,z,1)	normal
vertex.color	(r,g,b,a)	primary color
vertex.color.primary	(r,g,b,a)	primary color
vertex.color.secondary	(r,g,b,a)	secondary color
vertex.fogcoord	(f,0,0,1)	fog coordinate
vertex.texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex.attrib[n]	(x,y,z,w)	generic vertex attribute n
vertex.id	(id,-,-,-)	vertex identifier (integer)
vertex.instance	(i,-,-,-)	primitive instance number (integer)
vertex.texcoord[n..o]	(x,y,z,w)	array of texture coordinates
vertex.attrib[n..o]	(x,y,z,w)	array of generic vertex attributes

- **Vertex Program Result Variables (after vertex shader)**

Binding	Components	Description
result.position	(x,y,z,w)	position in clip coordinates
result.color	(r,g,b,a)	front-facing primary color
result.color.primary	(r,g,b,a)	front-facing primary color
result.color.secondary	(r,g,b,a)	front-facing secondary color
result.color.front	(r,g,b,a)	front-facing primary color
result.color.front.primary	(r,g,b,a)	front-facing primary color
result.color.front.secondary	(r,g,b,a)	front-facing secondary color
result.color.back	(r,g,b,a)	back-facing primary color
result.color.back.primary	(r,g,b,a)	back-facing primary color
result.color.back.secondary	(r,g,b,a)	back-facing secondary color
result.fogcoord	(f,*,*,*)	fog coordinate
result.pointsize	(s,*,*,*)	point size
result.texcoord	(s,t,r,q)	texture coordinate, unit 0
result.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
result.attrib[n]	(x,y,z,w)	generic interpolant n
result.clip[n]	(d,*,*,*)	clip plane distance
result.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
result.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
result.clip[n..o]	(d,*,*,*)	clip distances n thru o
result.id	(id,*,*,*)	vertex id

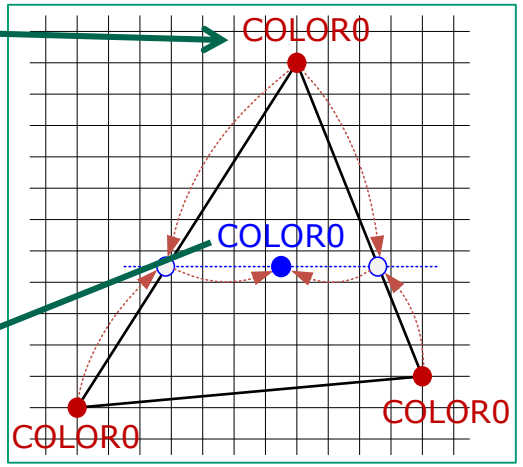
- **Fragment Program Attribute Variables (before fragment shader)**

	Fragment Attribute Binding	Components	Underlying State
	fragment.color	(r,g,b,a)	primary color
	fragment.color.primary	(r,g,b,a)	primary color
	fragment.color.secondary	(r,g,b,a)	secondary color
	fragment.texcoord	(s,t,r,q)	texture coordinate, unit 0
	fragment.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
	fragment.fogcoord	(f,-,-,-)	fog distance/coordinate
*	fragment.clip[n]	(c,-,-,-)	interpolated clip distance n
	fragment.attrib[n]	(x,y,z,w)	generic interpolant n
	fragment.texcoord[n..o]	(s,t,r,q)	texture coordinates n thru o
*	fragment.clip[n..o]	(c,-,-,-)	clip distances n thru o
	fragment.attrib[n..o]	(x,y,z,w)	generic interpolants n thru o
*	fragment.position	(x,y,z,1/w)	window position
*	fragment.facing	(f,-,-,-)	fragment facing
*	primitive.id	(id,-,-,-)	primitive number

- **Fragment Program Result Variables (after fragment shader)**

Binding	Components	Description
result.color	(r,g,b,a)	color
result.color[n]	(r,g,b,a)	color output n
result.depth	(*,*,d,*)	depth coordinate

Binding	Components	Des
result.position	(x,y,z,w)	pos
result.color	(r,g,b,a)	fro
result.color.primary	(r,g,b,a)	fro
result.color.secondary	(r,g,b,a)	fro
result.color.front	(r,g,b,a)	fro



Rasterization

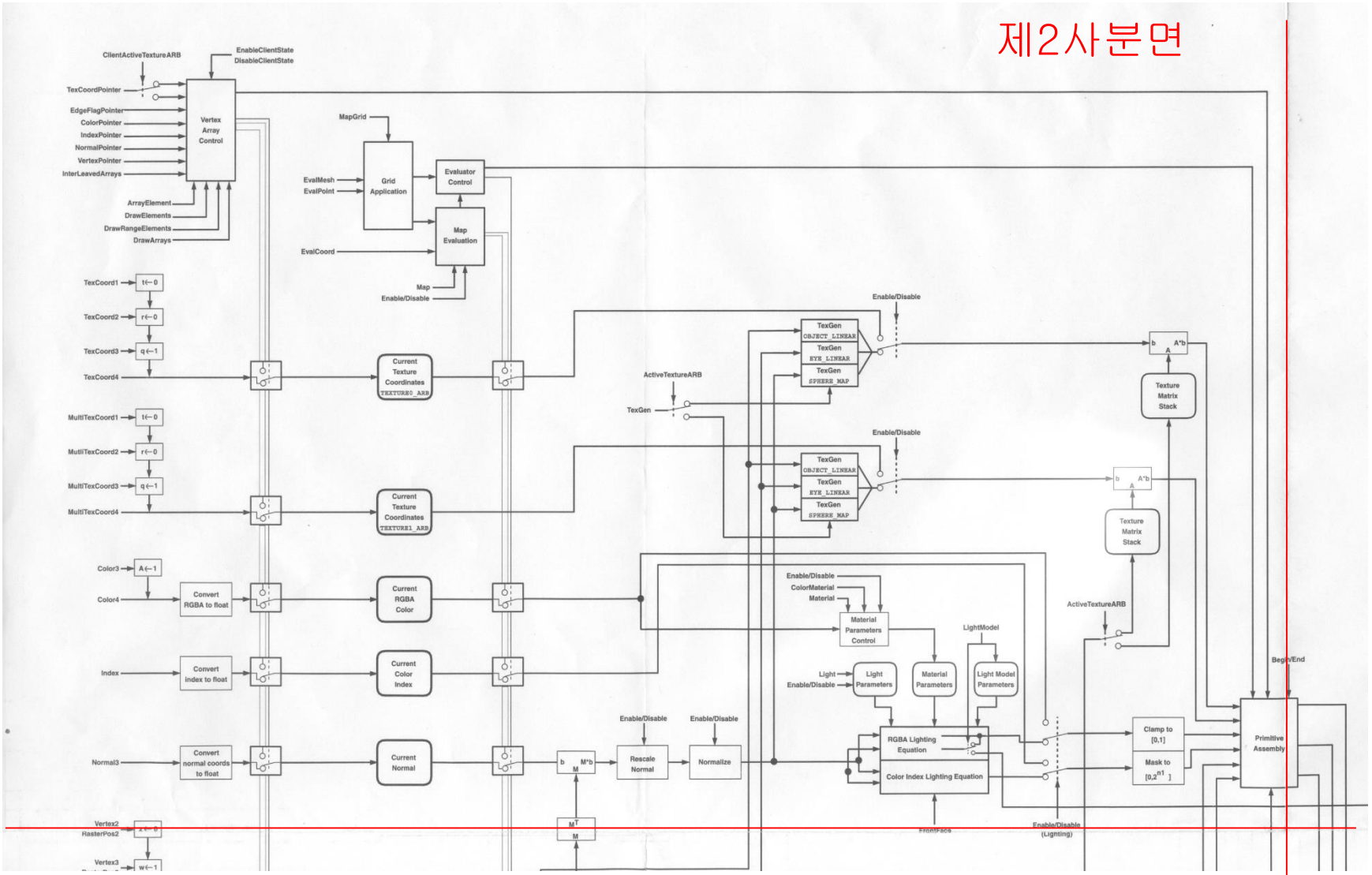
Fragment Attribute Binding	Components	Underl
fragment.color	(r,g,b,a)	primar
fragment.color.primary	(r,g,b,a)	primar
fragment.color.secondary	(r,g,b,a)	second
fragment.texcoord	(s,t,r,q)	textur
fragment.texcoord[n]	(s,t,r,q)	textur
fragment.fogcoord	(f,-,-,-)	fog di

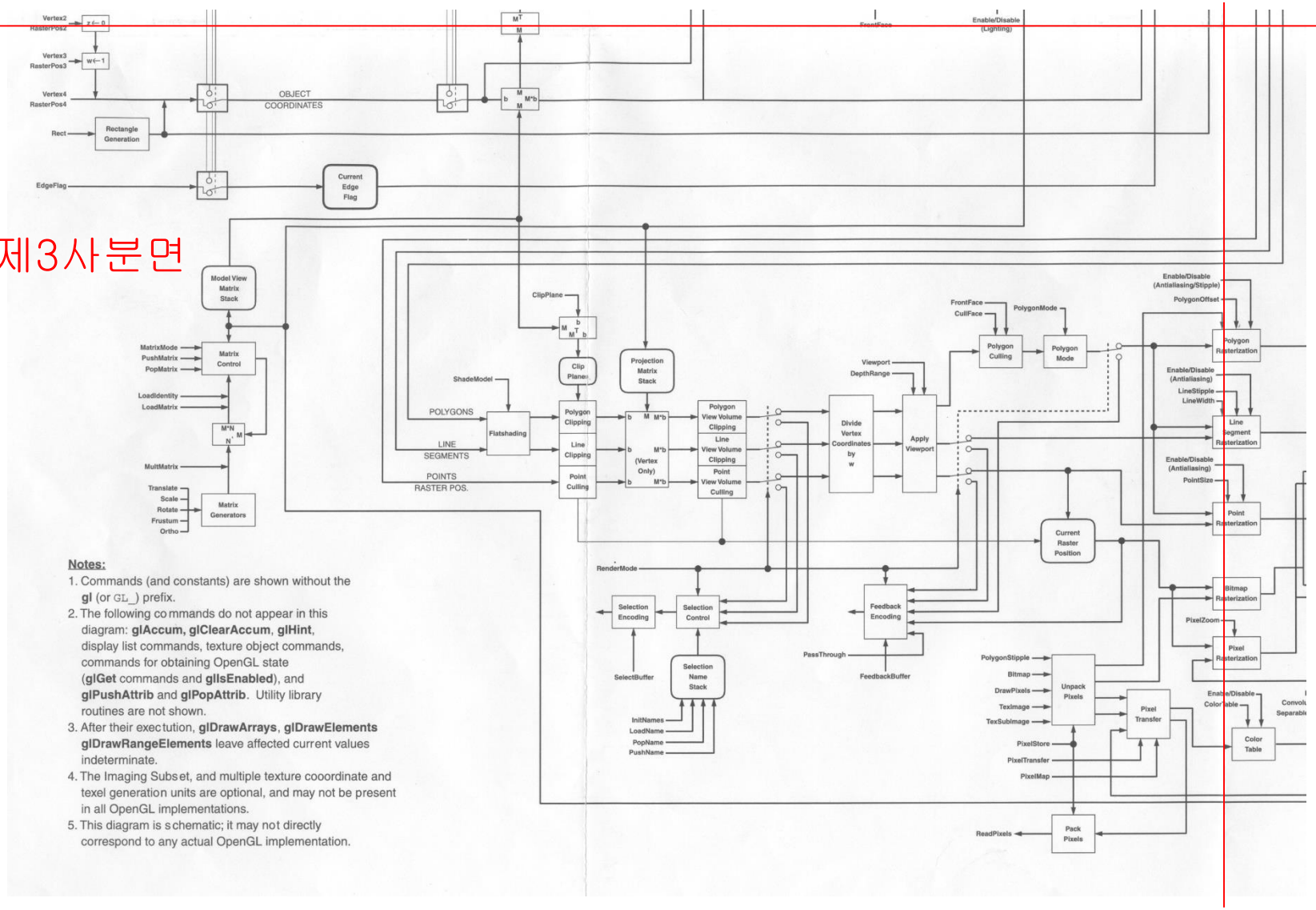
Streaming Processing in the Current GPU

- The current programmable GPU pipeline

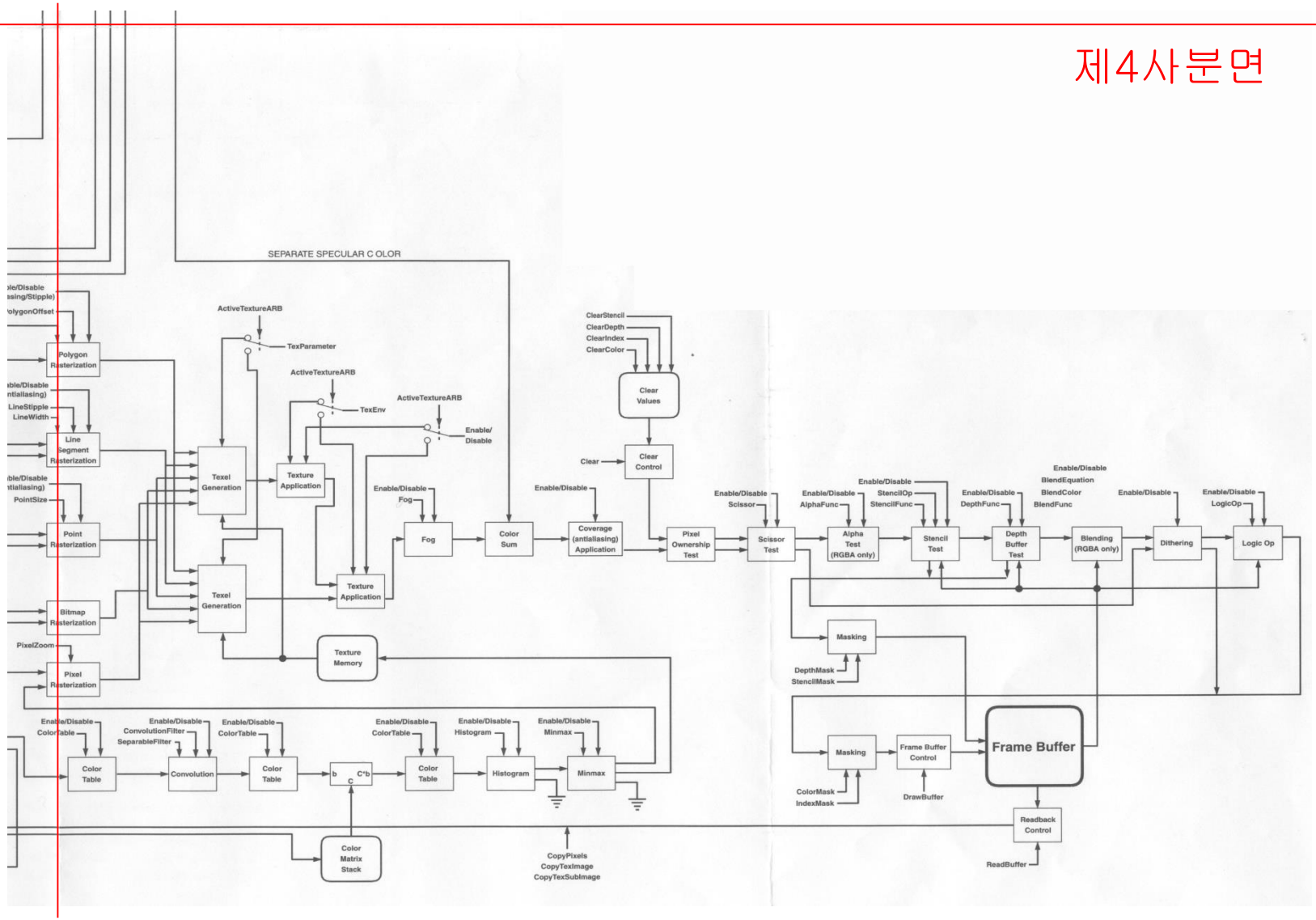
vertex stream → **Vertex Shader** → *vertex stream* → **Primitive Assembly** → *primitive stream* (→ **Geometry Shader** → *primitive stream*) → **Clipping & Setup** → **Rasterization** → *pixel stream* → **Pixel Shader** → *pixel stream* → **Raster Operation** → **Framebuffer**

- Comparison with the fixed-function OpenGL pipeline


제2사분면
ClientActiveTextureARB
EnableClientState
DisableClientState
TexCoordPointer
EdgeFlagPointer
ColorPointer
IndexPointer
NormalPointer
VertexPointer
InterLeavedArrays
Vertex Array Control
ArrayElement
DrawElements
DrawRangeElements
DrawArrays
MapGrid
EvalMesh
EvalPoint
Grid Application
Evaluator Control
Map Evaluation
EvalCoord
Map
Enable/Disable
TexCoord1
TexCoord2
TexCoord3
TexCoord4
Current Texture Coordinates
TEXTURE0_ARB
MultiTexCoord1
MutiTexCoord2
MultiTexCoord3
MultiTexCoord4
TEXTURE1_ARB
Color3
Color4
Convert RGBA to float
Current RGBA Color
Index
Convert index to float
Current Color Index
Normal3
Convert normal coords to float
Current Normal
Vertex2
RasterPos2
Vertex3
ActiveTextureARB
TexGen
TexGen OBJECT_LINEAR
TexGen EYE_LINEAR
TexGen SPHERE_MAP
Texture Matrix Stack
ColorMaterial
Material
Material Parameters Control
LightModel
Light Enable/Disable
Light Parameters
Material Parameters
Light Model Parameters
RGBA Lighting Equation
Color Index Lighting Equation
Rescale Normal
Normalize
Clamp to [0,1]
Mask to
Primitive Assembly
Begin/End
Enable/Disable (Lighting)

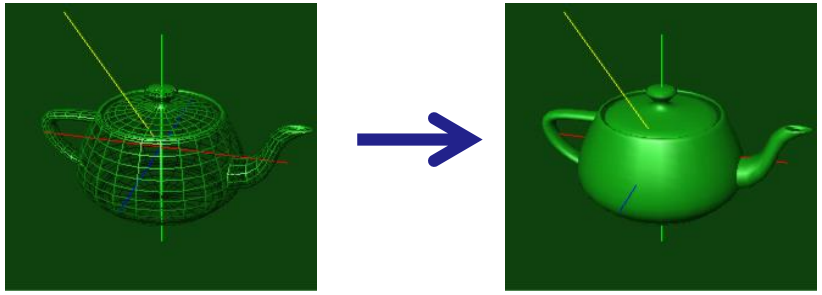

제3사분면
Vertex2
Vertex3
RasterPos3
Vertex4
RasterPos4
Rect
Rectangle Generation
EdgeFlag
OBJECT COORDINATES
Current Edge Flag
Model View Matrix Stack
MatrixMode
PushMatrix
PopMatrix
Matrix Control
LoadIdentity
LoadMatrix
MultMatrix
Translate
Scale
Rotate
Frustum
Ortho
Matrix Generators
ClipPlane
Clip Planes
ShadeModel
POLYGONS
LINE SEGMENTS
POINTS
RASTER POS.
Flatshading
Polygon Clipping
Line Clipping
Point Culling
Projection Matrix Stack
(Vertex Only)
Polygon View Volume Clipping
Line View Volume Clipping
Point View Volume Clipping
Viewport
DepthRange
Divide Vertex Coordinates by w
Apply Viewport
FrontFace
CullFace
PolygonMode
Polygon Culling
Polygon Mode
Enable/Disable (Antialiasing/Stipple)
PolygonOffset
Polygon Rasterization
Enable/Disable (Antialiasing)
LineStipple
LineWidth
Line Segment Rasterization
PointSize
Point Rasterization
Current Raster Position
RenderMode
Selection Encoding
Selection Control
SelectBuffer
Selection Name Stack
InitNames
LoadName
PopName
PushName
Feedback Encoding
PassThrough
FeedbackBuffer
Bitmap Rasterization
PixelZoom
Pixel Rasterization
PolygonStipple
Bitmap
DrawPixels
TexImage
TexSubImage
PixelStore
PixelTransfer
PixelMap
Unpack Pixels
Pixel Transfer
Color Table
ReadPixels
Pack Pixels
Notes:
1. Commands (and constants) are shown without the gl (or GL_) prefix.
2. The following commands do not appear in this diagram: glAccum, glClearAccum, glHint, display list commands, texture object commands, commands for obtaining OpenGL state (glGet commands and glIsEnabled), and glPushAttrib and glPopAttrib. Utility library routines are not shown.
3. After their execution, glDrawArrays, glDrawElements glDrawRangeElements leave affected current values indeterminate.
4. The Imaging Subset, and multiple texture cooordinate and texel generation units are optional, and may not be present in all OpenGL implementations.
5. This diagram is schematic; it may not directly correspond to any actual OpenGL implementation.

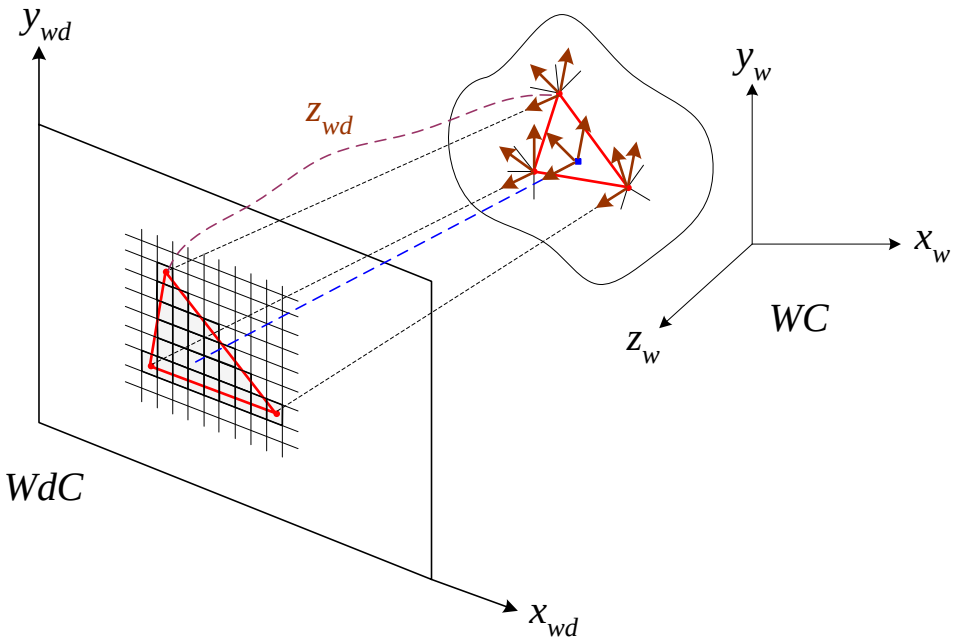
제4사분면


SEPARATE SPECULAR C OLOR
ActiveTextureARB
TexParameter
ActiveTextureARB
TexEnv
ActiveTextureARB
Enable/
Disable
ClearStencil
ClearDepth
ClearIndex
ClearColor
Clear Values
Clear
Clear Control
Polygon Rasterization
LineStipple
LineWidth
Line Segment Rasterization
PointSize
Point Rasterization
Bitmap Rasterization
PixelZoom
Pixel Rasterization
Texel Generation
Texture Application
Texel Generation
Texture Application
Texture Memory
Enable/Disable
Fog
Fog
Color Sum
Enable/Disable
Coverage (antialiasing) Application
Pixel Ownership Test
Enable/Disable
Scissor
Scissor Test
Enable/Disable
AlphaFunc
Alpha Test (RGBA only)
Enable/Disable
StencilOp
StencilFunc
Stencil Test
Enable/Disable
DepthFunc
Depth Buffer Test
Enable/Disable
BlendEquation
BlendColor
BlendFunc
Blending (RGBA only)
Enable/Disable
Dithering
Enable/Disable
LogicOp
Logic Op
Masking
DepthMask
StencilMask
Enable/Disable
ColorTable
Color Table
Enable/Disable
ConvolutionFilter
SeparableFilter
Convolution
Enable/Disable
ColorTable
Color Table
b
C
C*b
Enable/Disable
ColorTable
Color Table
Enable/Disable
Histogram
Histogram
Enable/Disable
Minmax
Minmax
Color Matrix Stack
CopyPixels
CopyTexImage
CopyTexSubImage
Masking
ColorMask
IndexMask
Frame Buffer Control
DrawBuffer
Frame Buffer
Readback Control
ReadBuffer

Pixel Shading in OpenGL

- 해당 pixel을 통해서 보이는 물체 지점의 색깔을 어떻게 계산할 것인가?
  - **물체 지점에서 반사되는 색깔을 어떻게 계산할 것인가?**
    - Phong's illumination model이 제공하는 공식에 기반을 둠.
    - Polygonal model의 어느 지점에 대해 조명 공식을 적용할 것인가?
  - **Real-time rendering의 요구 조건을 만족하면서 어떻게 고화질의 영상을 생성할 것인가?**
    - 미리 전처리 과정에서 생성한 여러 부류의 데이터를 Texture Map에 저장을 한 후, Texture Mapping 기법을 사용함.





[CSE4170 기초 컴퓨터 그래픽스]

2017년도 1학기

강의자료 III

Lighting Computation

Shading and Illumination Model

Rendering Equations

- The mathematical basis for all global illumination algorithms
- **Rendering equation for surfaces**

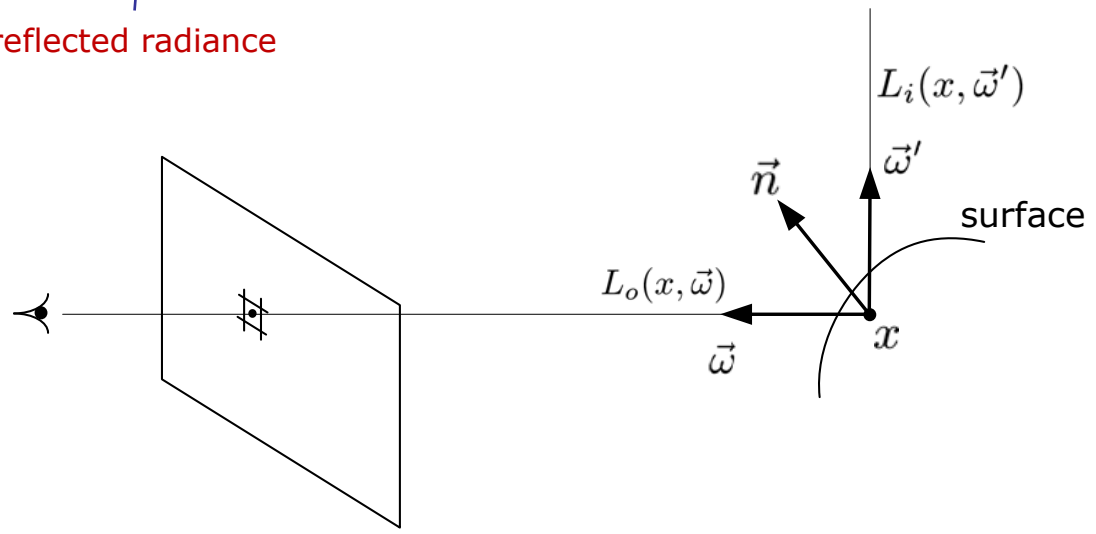
$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + L_r(x,\vec{\omega})$$

$$= L_e(x,\vec{\omega}) + \int_\Omega f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{\omega}' \cdot \vec{n})\, d\vec{\omega}'$$

outgoing radiance

emitted radiance

reflected radiance



Volume Rendering Equations

- **Volume rendering equation for participating media**

$$L(x,\vec{\omega}) = L(x+s\vec{\omega},\vec{\omega})\, e^{-\int_0^s \sigma_t(x+s'\vec{\omega})ds'} + \int_0^s L_e(x+s'\vec{\omega})\, e^{-\int_0^{s'} \sigma_t(x+t\vec{\omega})dt} ds'$$
$$+ \int_0^s \{e^{-\int_0^{s'} \sigma_t(x+t\vec{\omega})dt} \sigma_s(x+s'\vec{\omega}) \int_{\Omega_{4\pi}} p(x+s'\vec{\omega},\vec{\omega}',\vec{\omega}) L(x+s'\vec{\omega},\vec{\omega}')\, d\vec{\omega}'\}\, ds'$$

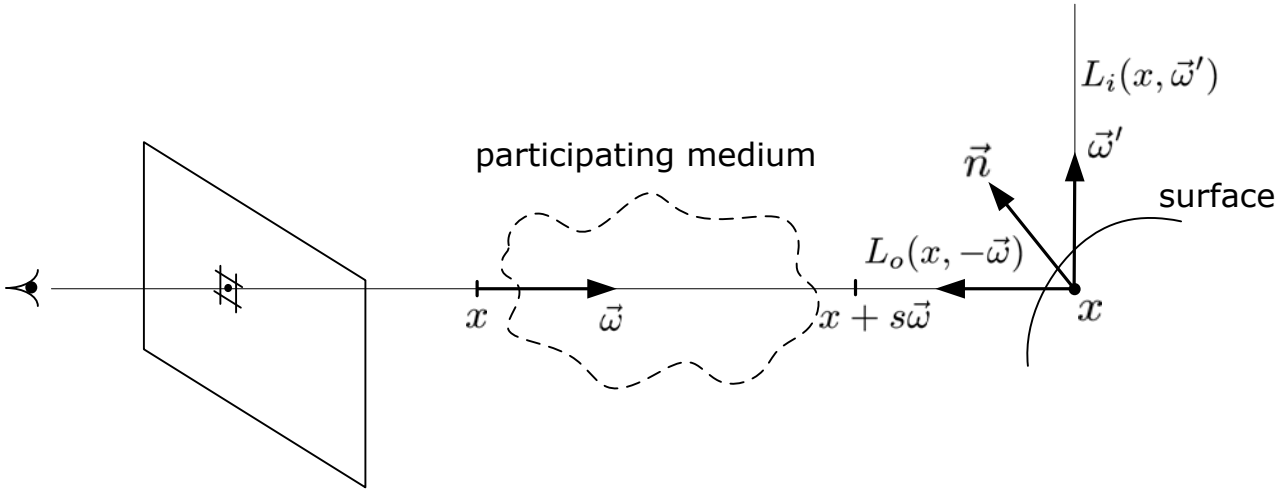
Outgoing radiance

Incoming radiance

Extinction due to absorption and out-scattering

In-scattering

Emission



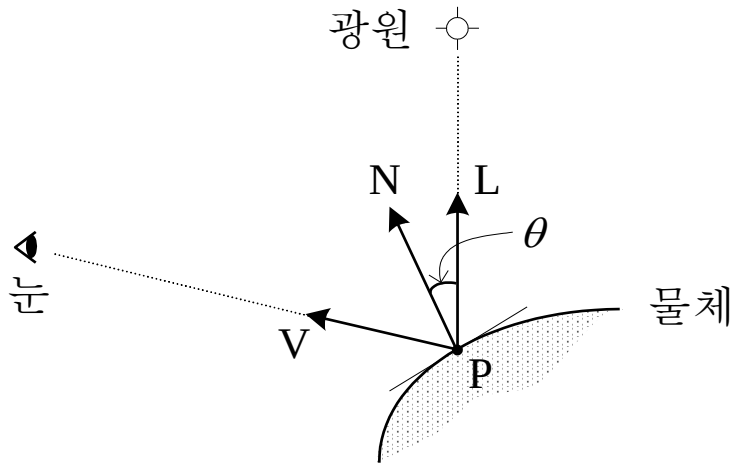
라이팅 계산 시 네 가지 고려 사항

① 광원이라는 물체를 어떻게 정의할 것인가?

② 쉐이딩을 하려는 물체상의 지점에 들어오는 빛의 경로, 방향, 밝기, 그리고 색깔 등을 어떠한 모델로 정의할 것인가?

③ 물체 표면에서 빛이 반사되는 방식을 어떻게 정의할 것인가?

④ 쉐이딩하려는 지점에 들어오는 빛과 물체 표면의 반사 성질이 결정되었을 때, 어떠한 계산 모델을 사용하여 시선 방향으로 반사되는 빛의 색깔을 결정할 것인가?



광원(Light Source)의 종류

- 광원: 스스로 빛을 발하는 물체
  - 전구, 형광등, 태양, 헤드라이트, ...
  - 밀폐된 방에 빛이 들어오는 뿌연 유리창, ...

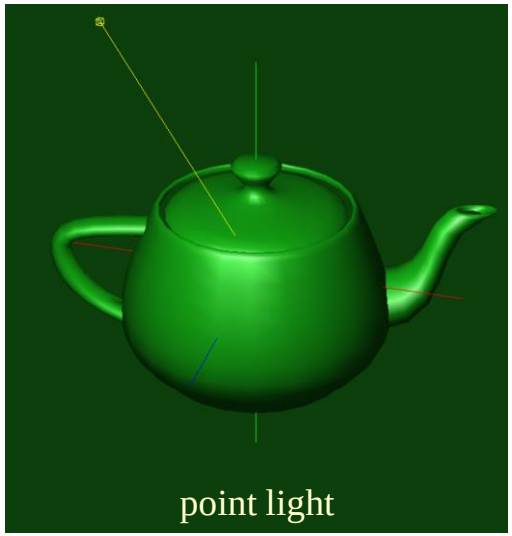
- 널리 쓰이는 광원

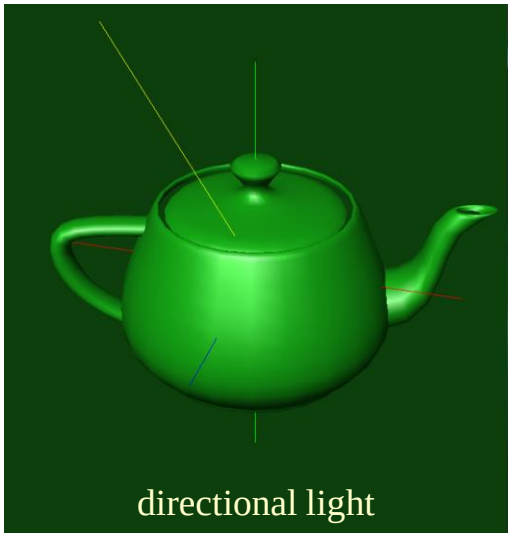
① **점 광원(point light source)**
  - 광원의 크기가 렌더링 하려는 전체 장면에 비하여 상대적으로 작을 경우에 사용.
  - 계산을 단순화시키기 위하여 한 점 (x, y, z)에 빛을 발하는 광원이 있다고 가정.
  - 예: 크기가 작은 전구, ...

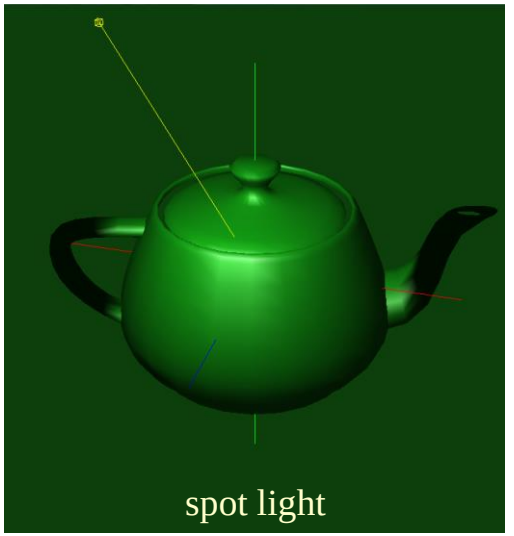
② **평행 광원(parallel light source 또는 directional light source)**
  - 광원이 물체로 부터 상당히 멀리 떨어져 있을 경우에 사용.
  - 계산을 단순화시키기 위하여 모든 빛이 주어진 방향 (x, y, z)에 평행하게 들어온다고 가정
  - 예: 태양 광선, ...
  - ☺ 점 광원과 평행 광원의 설정과 동차 좌표 (x, y, z, w)간의 관계

③ 스폿 광원(spot light source)

- 점 광원의 특수한 형태로 원뿔과 같이 일정한 범위로 빛을 발하는 광원
- 광원의 위치, 빛를 발하는 중심 방향과 범위의 설정이 필요함.
- 예: 손 전등, 헤드라이트, 무대 조명, ...





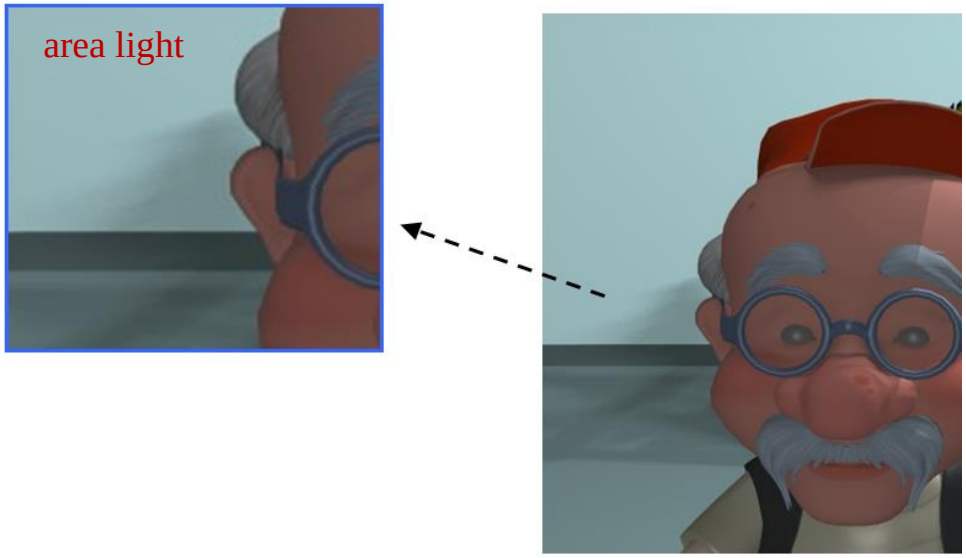


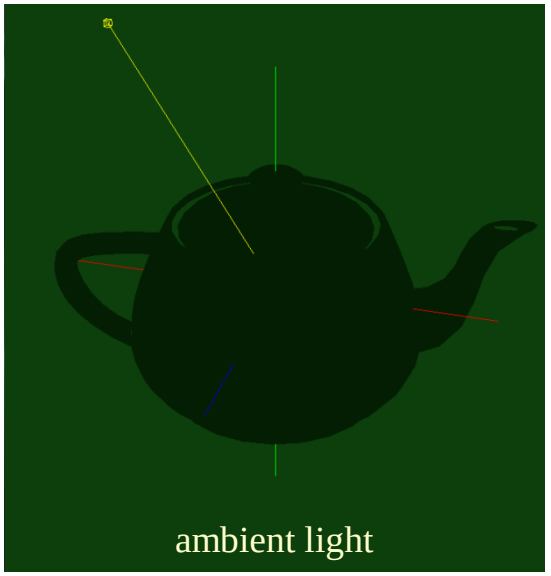
④ 면적 광원(area light source or distributed light source)

- 광원이 전체 장면이나 물체에 비하여 무시할 수 없는 면적을 가질 경우에 사용.
- 경우에 따라 길이, 면적, 체적을 가지는 물체로 가정.
- 상대적으로 계산량이 많으나 자연스러운 조명 효과를 얻을 수 있음.
- 예: 상대적으로 크기가 큰 형광등, ...

⑤ 앰비언트 광원(ambient light source)

- 광원이 분명히 존재하나 어디에 있는지 알기 힘들거나, 광원에서 들어오는 빛을 정확하게 또는 단순하게 모델링 하기 힘든 경우에 사용.
- 지역 조명 모델에서 처리하기 힘든 간접 조명의 흉내를 내주기도 함.
- 예: 흐린 날의 태양 광선, 실내에 깔려 있는 빛, ...



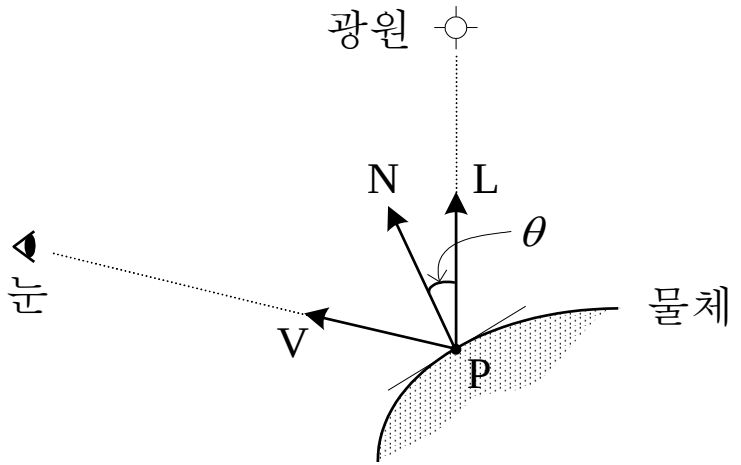


ⓘ ‘보통’ 실시간 렌더링 파이프라인에서는

① 점 광원, 평행 광원, 스폿 광원, 앰비언트 광원 등을 기본적으로 지원함.

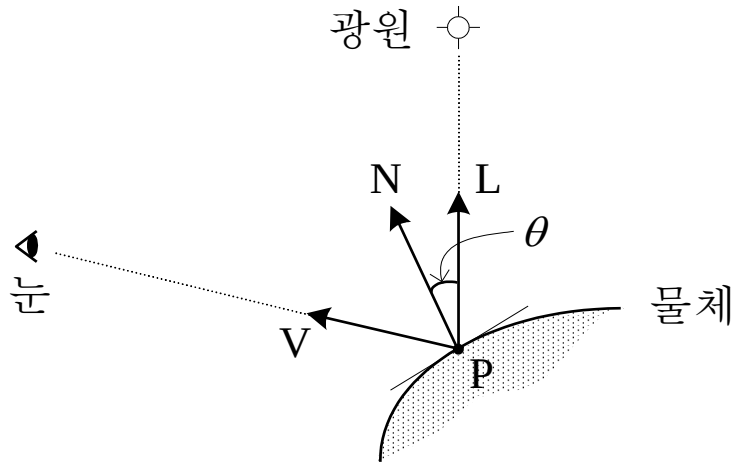
② 기본적으로 지역 조명 모델을 사용하기 때문에 광원에서 직접 들어오는 빛만 고려함.

③ 프로그래밍이 가능한 GPU가 등장함에 따라 다른 형태의 광원이나 전역 조명 모델 기능들이 자연스럽게 구현되기 시작함.



퐁의 조명 모델(Phong's Illumination Model)

- 문제
  - 물체 표면의 주어진 지점으로 빛이 들어왔을 때, 우리가 바라보고 있는 시선의 방향으로 반사되는 빛의 색깔을 어떠한 방식으로 계산을 할 것인가?



  - 한 가지 방법은 퐁의 조명 모델이라는 실험적인 모델을 사용하는 것임.
    - 이 모델은 입사 광선의 반사 형태를 결정하므로 퐁의 반사 모델(Phong's reflection model)이라고도 함.
    - 물리학적으로 정확한 모델은 아니나, 비교적 계산량이 적고 실험적으로 우수한 성능을 나타내기 때문에 렌더링 시 기본 모델로서 사용됨.

세 가지 종류의 반사

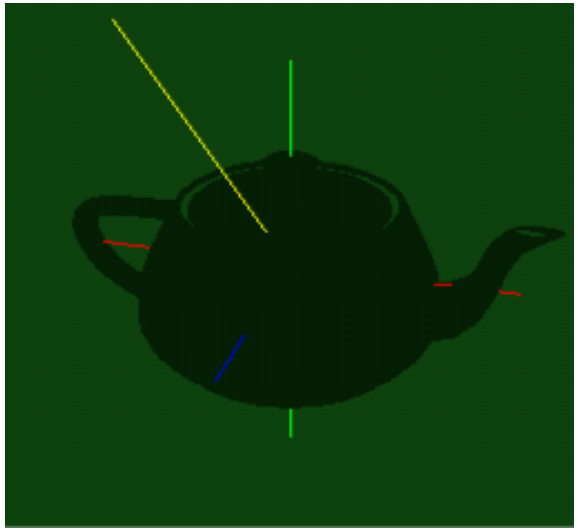
- 기본적으로 퐁의 조명 모델에서는 다음 세 가지 형태의 반사를 고려함.
  - 앰비언트 반사(ambient reflection)
  - 난반사(diffuse reflection)
  - 정반사(specular reflection)

$$I_\lambda = \boxed{I_{a\lambda} \cdot k_{a\lambda}} + \boxed{I_{l\lambda} \cdot k_{d\lambda} \cdot (N \circ L)} + \boxed{I_{l\lambda} \cdot k_{s\lambda} \cdot (R \circ V)^n}$$

ambient reflection　　diffuse reflection　　specular reflection

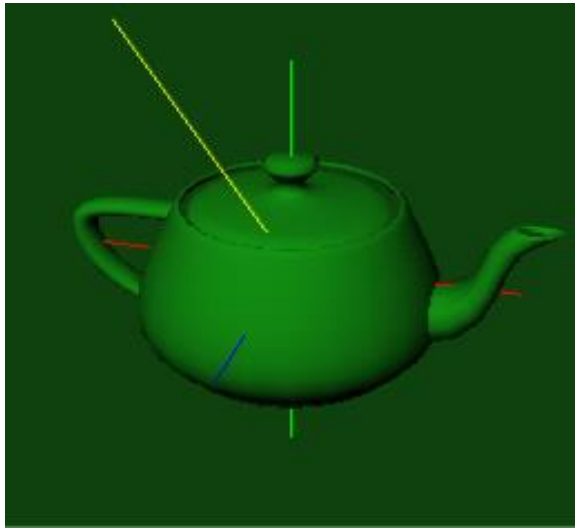
- 반사되는 빛의 색깔 I_λ
  - 정확학 계산을 하려면 가시 광선 영역의 모든 파장 λ 에 대하여 계산을 해야함.
  - 일반적으로 컴퓨터 그래픽스 분야에서는 빨강, 초록, 파랑에 해당하는 세 가지 파장에 대해서 계산을 함.

$$I_\lambda = \begin{pmatrix} I_R \\ I_G \\ I_B \end{pmatrix}$$



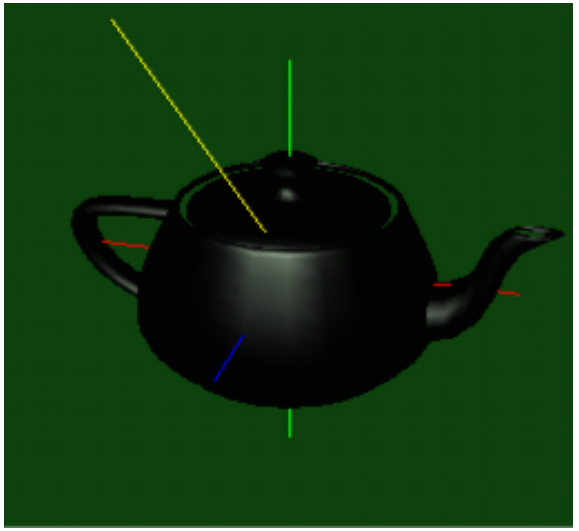
ambient reflection

+



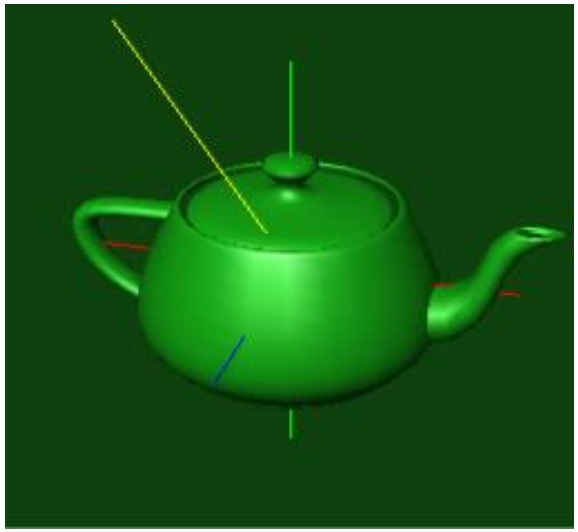
diffuse reflection

+

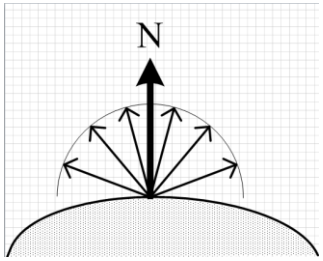


specular reflection

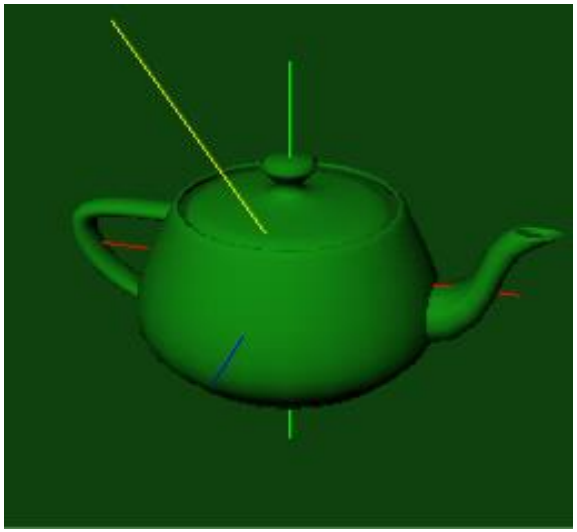
=



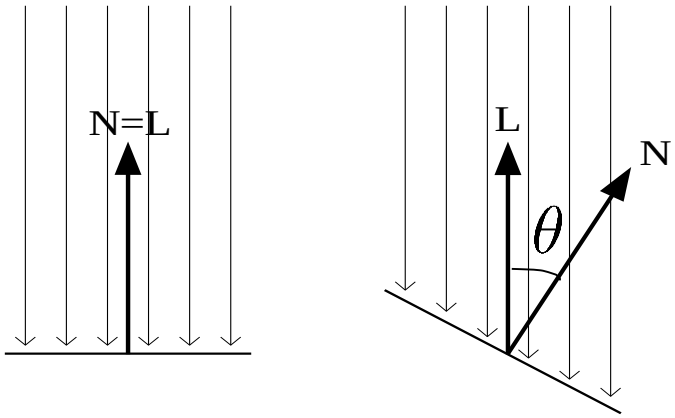
난반사(Diffuse Reflection)



- 특징
  - 입사 광선을 사방으로 고르게 동일한 밝기(색깔)로 반사시키는 형태의 반사를 시뮬레이션 하는데 사용.
  - 물체 표면이 종이, 분필, 석고상 등과 같이 반짝거리지 않고 좀 둔탁해 보이는 물체의 표면을 표현하는데 사용.
  - 바라보는 지점을 고정한 상태에서 시점을 옮겨도 동일한 밝기(색깔)로 보임.



- **램버트의 코사인 법칙(Lambert's cosine law)**
  - 물체 표면이 순수하게 난반사를 할 경우 그 물체를 이상적인 난반사체(ideal diffuse reflector), 또는 램버트 반사체(Lambertian reflector)라 함.
  - 코사인 법칙에 따라 빛을 반사함.
    - 반사되는 빛 에너지의 양은 $\cos\theta$에 비례.



- 난반사 공식

$$I_\lambda = I_{l\lambda} \cdot k_{d\lambda} \cdot \cos\theta = I_{l\lambda} \cdot k_{d\lambda} \cdot (N \circ L)$$

법선 벡터
(normal vector)

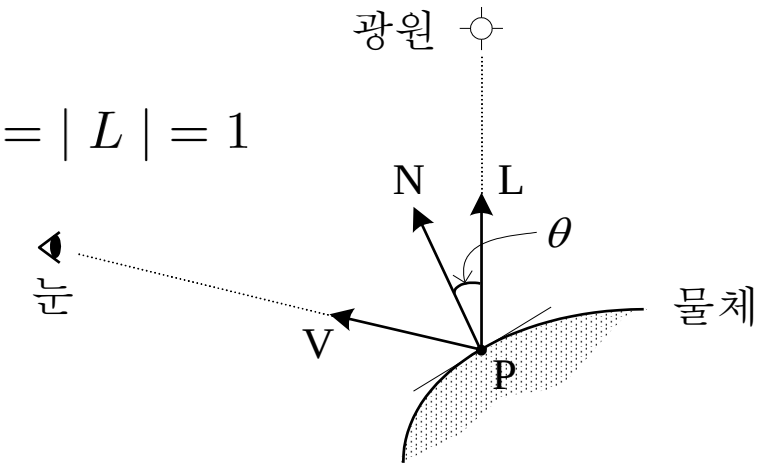
광원 벡터
(light vector)

광원의 색깔

난반사 계수
(diffuse reflection coefficient)

$$0 \le k_{dR}, k_{dG}, k_{dB} \le 1$$

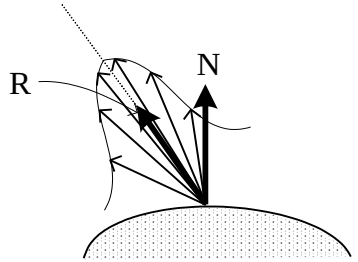
$$|V| = |N| = |L| = 1$$



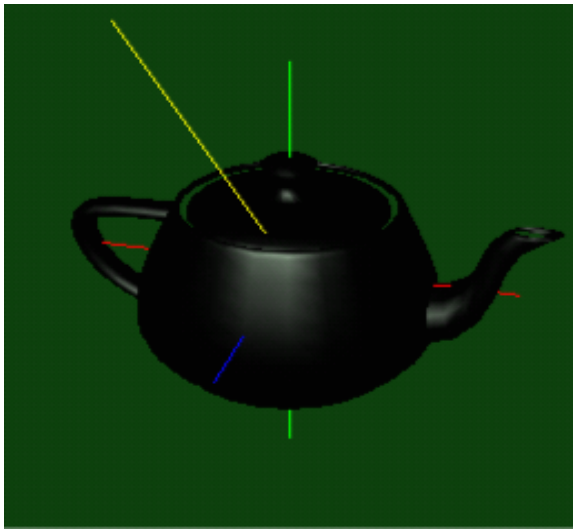
ⓘ 물체에 대한 반사 성질을 지정할 때 물체의 기본 색깔은 난반사 계수를 통하여 지정함!

$$k_{d\lambda} = (k_{dR},\ k_{dG},\ k_{dB})^t$$

정반사(Specular Reflection)



- 특징
  - 입사 광선을 특정 방향을 중심으로 집중적으로 반사시키는 형태의 반사를 시뮬레이션하는데 사용.
  - 거울, 자동차 표면, 금속, 광택을 낸 사과와 같이 반짝거리는 물체의 표면을 표현하는데 사용. → 하이라이트(highlight) 생성.
  - 바라보는 지점을 고정한 상태에서 시점을 옮기면 반사되는 빛의 밝기(색깔)가 급격히 변함.
  - 거울은 정반사를 하는 극단적인 예라 할 수 있음.



- 정반사 공식

$$I_\lambda = I_{l\lambda} \cdot w(\theta, \lambda) \cdot \cos^n \phi$$
$$\approx I_{l\lambda} \cdot k_{s\lambda} \cdot \cos^n \phi = I_{l\lambda} \cdot k_{s\lambda} \cdot (R \circ V)^n$$

정반사 방향
(specular reflection direction)

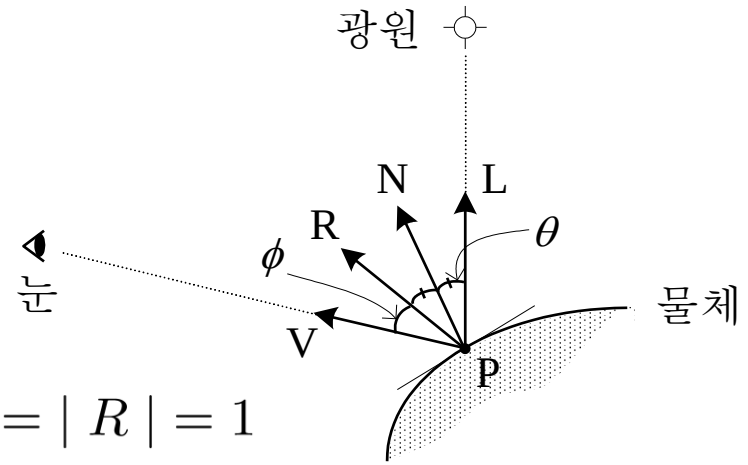
뷰 벡터
(view vector)

정반사 지수
(specular reflection exponent)

$1 \leq n \leq$ a few hundreds

정반사 계수
(specular reflection coefficient)

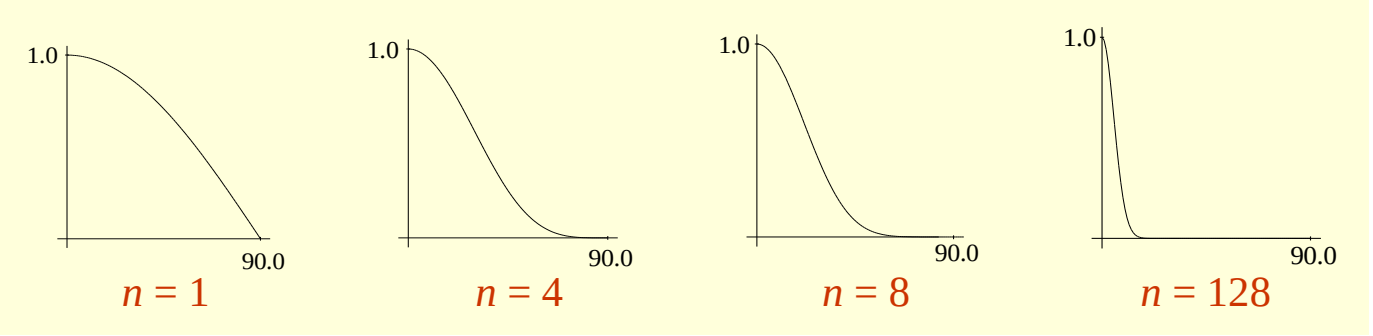
$0 \leq k_{sR}, k_{sG}, k_{sB} \leq 1$

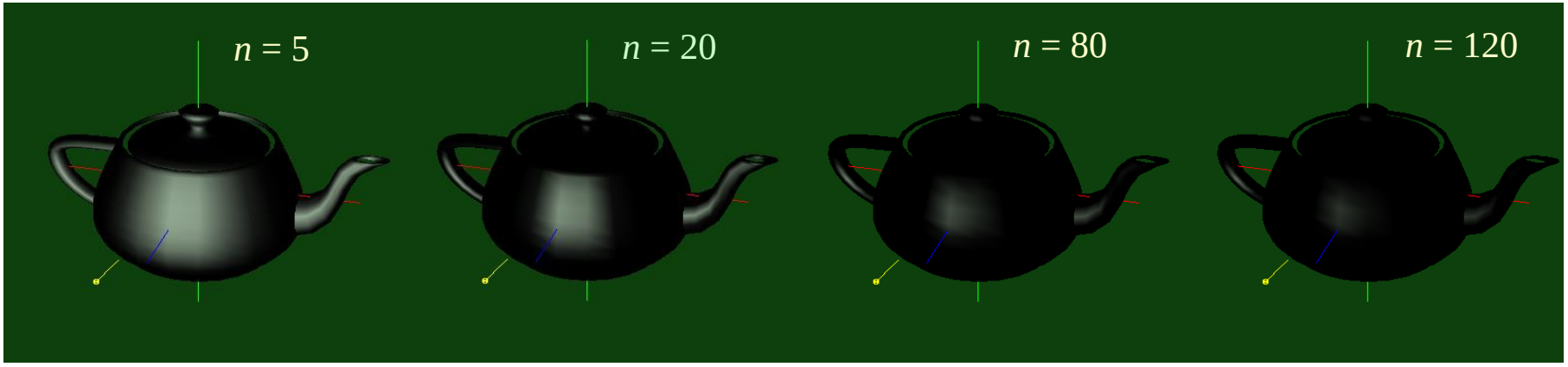


$$|V| = |N| = |L| = |R| = 1$$

- 정반사 지수 *n*의 역할
  - *n* 값을 통하여 시점 방향이 정반사 방향에서 벗어남에 따라 반사되는 빛의 세기가 약해지는 속도를 조절함.
  - 결과적으로 생성되는 하이라이트의 크기 결정.

$\cos^n \phi$



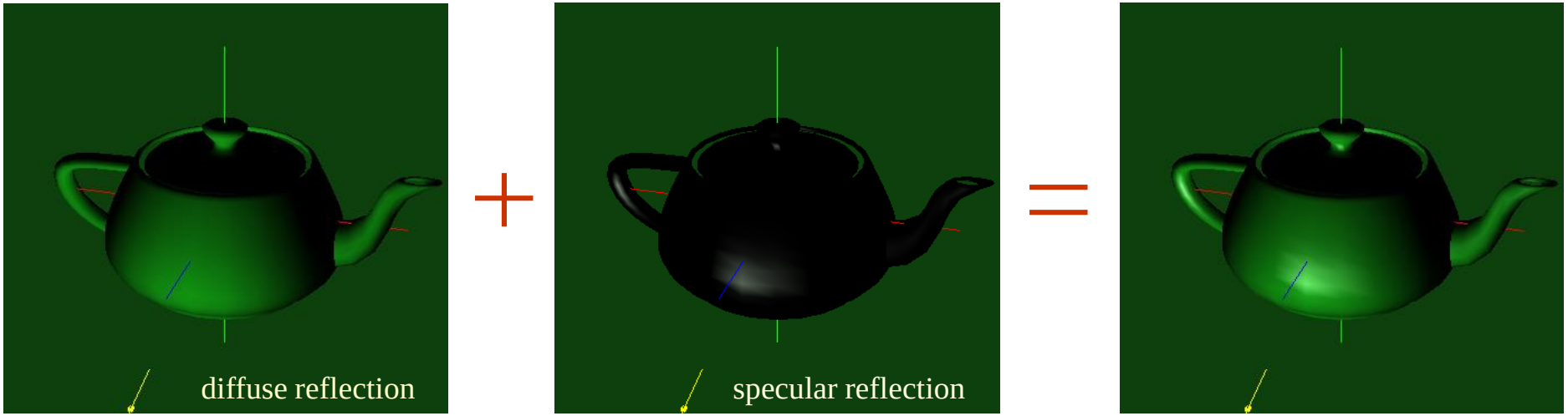


ⓘ 정반사에 의해 반사되는 빛의 색깔은 주로 광원의 색깔에 좌우됨.

→ 정반사 계수 $k_{s\lambda} = (k_{sR},\ k_{sG},\ k_{sB})^t$의 역할?

앰비언트 반사(Ambient Reflection)

- 특징
  - 지역 조명 모델은 기본적으로 광원에서 직접 들어오는 빛을 고려하기 때문에 간접적으로 들어오는 빛을 무시하게 됨.
  - 이러한 문제를 조금이나마 덜기 위하여 사방에 일정한 밝기의 빛이 고르게 퍼져 있다고 가정.
  - 간접적으로 들어오는 빛을 단순한(엉성한) 방법으로 모델링.



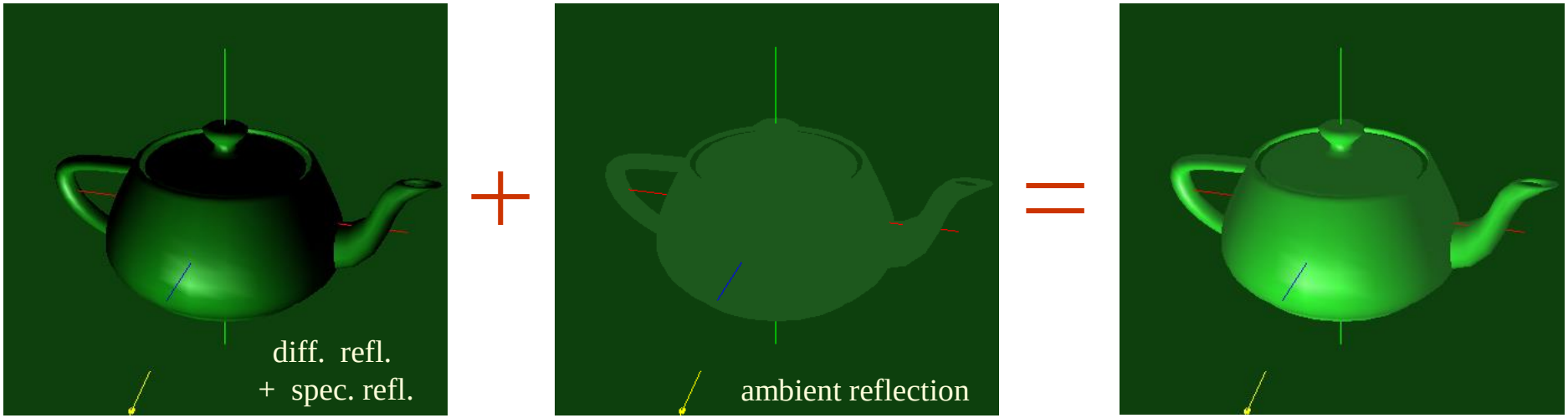
- 앰비언트 반사 공식

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda}$$

앰비언트 광원의 색깔

앰비언트 반사 계수
(ambient reflection coefficient)

$$0 \le k_{aR}, k_{aG}, k_{aB} \le 1$$



퐁의 조명 모델에 대한 변형

- 기본적인 퐁의 조명 모델
  - 물체의 고유 반사 성질
  - 광원의 색깔

$$\begin{aligned} I_\lambda &= I_{a\lambda} \cdot k_{a\lambda} + I_{l\lambda} \cdot k_{d\lambda} \cdot (N \circ L) + I_{l\lambda} \cdot k_{s\lambda} \cdot (R \circ V)^n \\ &= I_{a\lambda} \cdot k_{a\lambda} + I_{l\lambda} \cdot \{k_{d\lambda} \cdot (N \circ L) + k_{s\lambda} \cdot (R \circ V)^n\} \end{aligned}$$

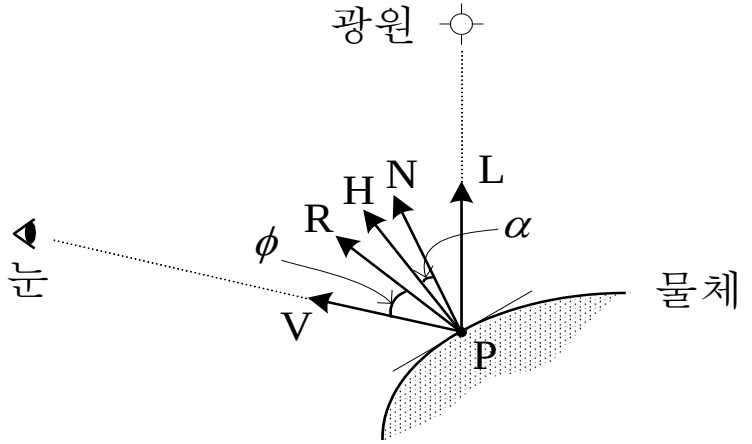
or

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + I_{l\lambda}^D \cdot k_{d\lambda} \cdot (N \circ L) + I_{l\lambda}^S \cdot k_{s\lambda} \cdot (R \circ V)^n$$

- 기본 모델에 대한 변형
  - 해프웨이 벡터 (Halfway vector)
  - 빛의 감쇠 효과 (Light attenuation)
  - 다중 광원 (multiple light sources)
  - …

해프웨이 벡터(Halfway Vector)

- 정반사 방향의 계산
  - 렌더링 파이프라인에 들어오는 각 꼭지점에 대하여 정반사 방향 R을 계산.
  - 다면체 모델이 어느 정도 클 경우 적지 않은 양의 부동 소수점 연산 필요.

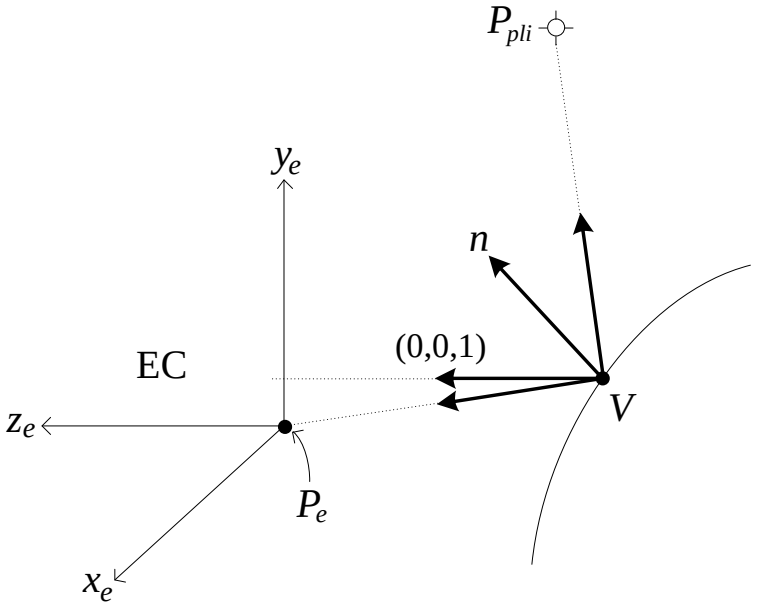


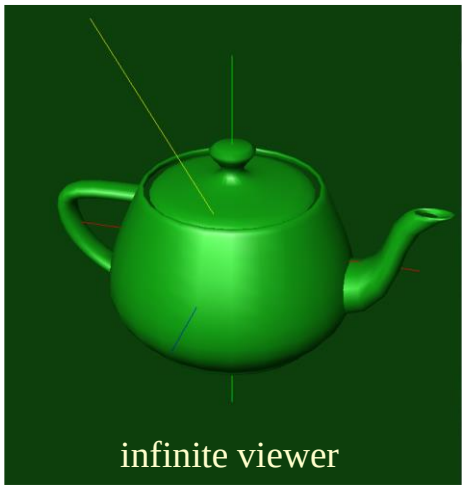
$$R = 2(N \circ L)N - L$$

- 정반사 공식 계산 시 해프웨이 벡터 H = $\frac{L+V}{|L+V|}$를 사용.

$$I_\lambda = I_{l\lambda} \cdot k_{s\lambda} \cdot (R \circ V)^n \longrightarrow I_\lambda = I_{l\lambda} \cdot k_{s\lambda} \cdot (N \circ H)^n$$

- **언제 halfway vector가 빛을 발할까?**
  - 평행 광원과 평행 투영을 사용하는 경우, 물체의 모든 지점에 대하여 L 벡터와 V 벡터가 상수 벡터이므로, H 벡터 또한 상수 벡터가 됨. → H 벡터는 한 번만 계산하면 됨.
- **과연 계산량을 줄이기 위하여 매번 평행 광원과 평행 투영을 사용해야 할까?**
  - **평행 광원**: 점 광원 대신에 평행 광원을 사용해도 일반적으로 큰 문제가 없음.
  - **평행 투영**: 원근 투영 대신 사용할 경우 원근감이 사라지므로 심각한 문제가 발생.
    - OpenGL에서는 뷰잉 계산에서는 원근 투영을 사용해도, 라이팅 계산 시에만 마치 평행 투영을 하는 것과 같은 기능을 제공 → infinite viewer versus local viewer





- 해프웨이 벡터를 사용할 경우

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + I_{l\lambda} \cdot \{k_{d\lambda} \cdot (N \circ L) + k_{s\lambda} \cdot (N \circ H)^n\}$$

ⓘ 해프웨이 벡터를 사용할 경우 원래의 정반사 값과는 약간 다른 값을 사용하게 되나, 이는 정반사 계수나 정반사 지수로 적절히 조정할 수 있음.

빛의 감쇠 효과(Light Attenuation Effect)

- 광원과 물체간의 거리에 따른 밝기 조절을 원할 경우

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + f_{att}(d) \cdot I_{l\lambda} \cdot \{k_{d\lambda} \cdot (N \circ L) + k_{s\lambda} \cdot (N \circ H)^n\}$$

- 어떤 감쇠 함수를 사용할 것인가?

$$f_{att}(d) = \frac{1}{d^2}$$

$$f_{att}(d) = \frac{1}{k_0 + k_1 \cdot d + k_2 \cdot d^2}$$

$$f_{att}(d) = min(\frac{1}{k_0 + k_1 d + k_2 d^2}, 1)$$

다중 광원(Multiple Light Sources)

- 광원이 여러 개가 있을 경우

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + \sum_{i=0}^{m-1} f_{att}(d_i) \cdot I_{l_i\lambda} \cdot \{k_{d\lambda} \cdot (N \circ L_i) + k_{s\lambda} \cdot (N \circ H_i)^n\}$$

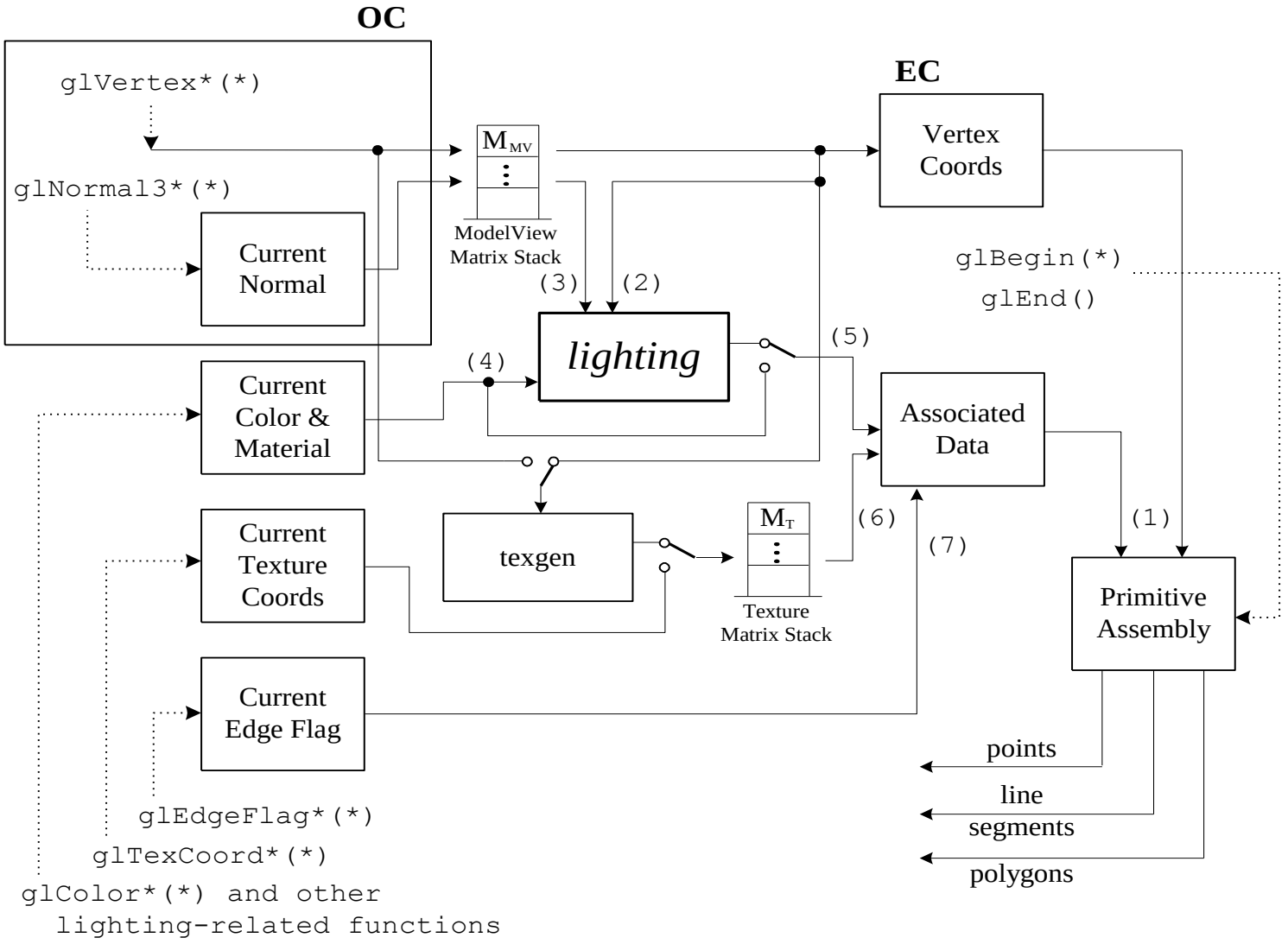
Material Parameter 예

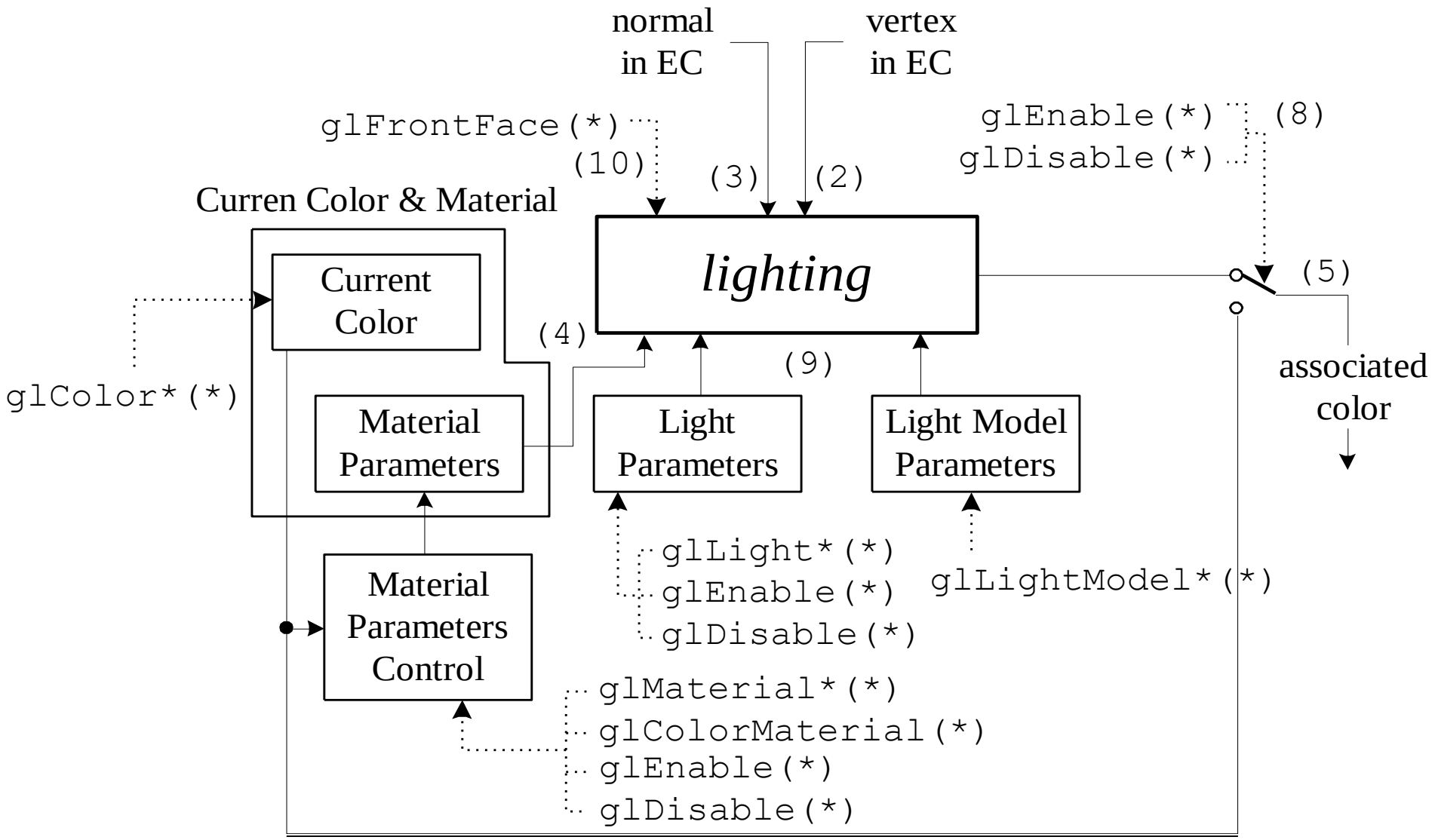
	$k_{a\lambda}$	$k_{d\lambda}$	$k_{s\lambda}$	n
Black Plastic	0.0, 0.0, 0.0	0.01, 0.01, 0.01	0.5, 0.5, 0.5	32
Brass	0.329412, 0.223529, 0.027451	0.780392, 0.568627, 0.113725	0.992157, 0.941176, 0.807843	27.9
Bronze	0.2125, 0.1275, 0.054	0.714, 0.4284, 0.18144	0.393548, 0.271906, 0.166721	25.6
Chrome	0.25, 0.25, 0.25	0.4, 0.4, 0.4	0.774597, 0.774597, 0.774597	76.8
Copper	0.19125, 0.0735, 0.0225	0.7038, 0.27048, 0.0828	0.256777, 0.137622, 0.086014	12.8
Emerald	0.0215, 0.1745, 0.0215	0.07568, 0.61424, 0.07568	0.633, 0.727811, 0.633	76.8
Gold	0.24725, 0.1995, 0.0745	0.75164, 0.60648, 0.22648	0.628281, 0.555802, 0.366065	51.2
Pearl	0.25, 0.20725, 0.20725	1, 0.829, 0.829	0.296648, 0.296648, 0.296648	11.3
Pewter	0.10588, 0.058824, 0.113725	0.427451, 0.470588, 0.541176	0.3333, 0.3333, 0.521569	9.8
Ruby	0.1745, 0.01175, 0.01175	0.61424, 0.04136, 0.04136	0.727811, 0.626959, 0.626959	76.8
Silver	0.19225, 0.19225, 0.19225	0.50754, 0.50754, 0.50754	0.508273, 0.508273, 0.508273	51.2
Yellow rubber	0.05, 0.05, 0.0	0.5, 0.5, 0.4	0.7, 0.7, 0.04	10.0

Phong의 조명 모델 적용 결과

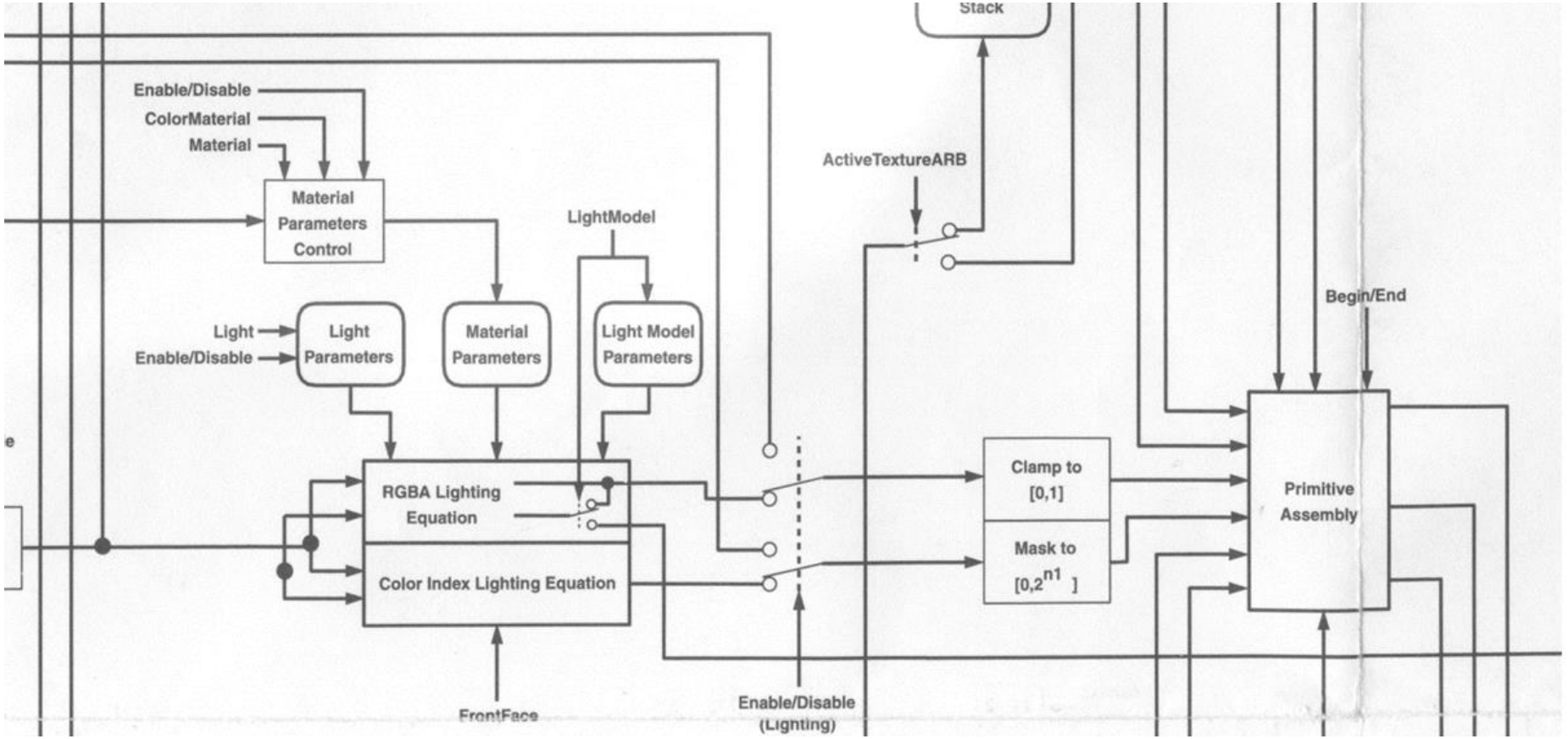
Fixed-Function Vertex Lighting in OpenGL

OpenGL에서의 라이팅 계산 구조

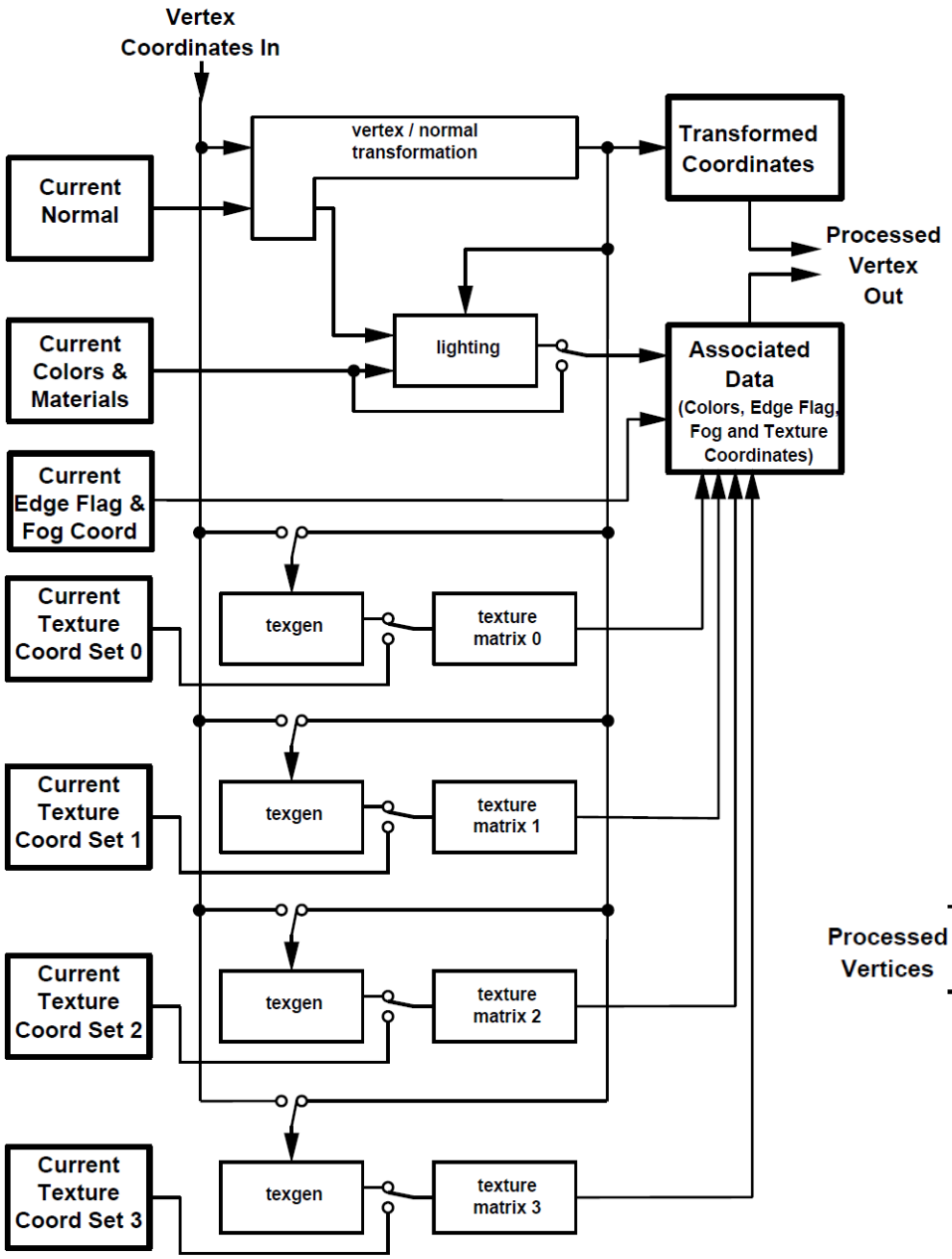



normal
in EC
vertex
in EC
glFrontFace(*)
(10)
(3)
(2)
glEnable(*)
glDisable(*)
(8)
Curren Color & Material
lighting
Current
Color
(4)
(9)
(5)
associated
color
glColor*(*)
Material
Parameters
Light
Parameters
Light Model
Parameters
glLight*(*)
glEnable(*)
glDisable(*)
glLightModel*(*)
Material
Parameters
Control
glMaterial*(*)
glColorMaterial(*)
glEnable(*)
glDisable(*)

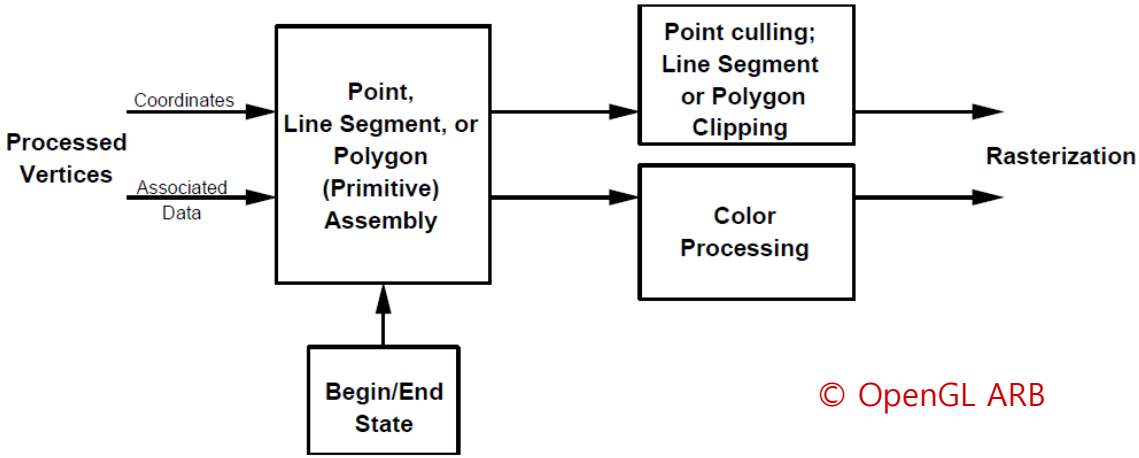
Lighting Computation in EC



Vertex Processing and Primitive Assembly in EC



기존의 fixed-function OpenGL pipeline에서는 EC에서 lighting 계산을 통하여 (또는 current color 지정을 통하여) 꼭지점에서의 color attribute가 결정되며, 이후 꼭지점과 함께 rasterization 과정까지 흘러감!



© OpenGL ARB

OpenGL Lighting Formula

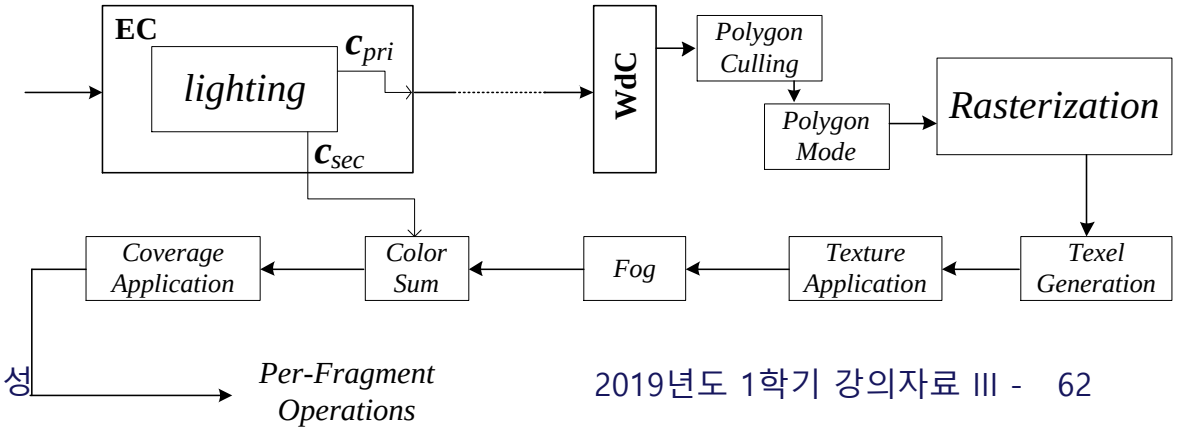
- **비 정반사 색깔 분리 모드**

$$\begin{aligned}\mathbf{c}_{pri} &= \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \\ &\sum_{i=0}^{n-1}(att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}] \\ \mathbf{c}_{sec} &= (0,0,0,1)\end{aligned}$$

- **정반사 색깔 분리 모드**

$$\mathbf{c}_{pri} = \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \sum_{i=0}^{n-1}(att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli}]$$

$$\mathbf{c}_{sec} = \sum_{i=0}^{n-1}(att_i)(spot_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}}\mathbf{s}_{cm} * \mathbf{s}_{cli}$$



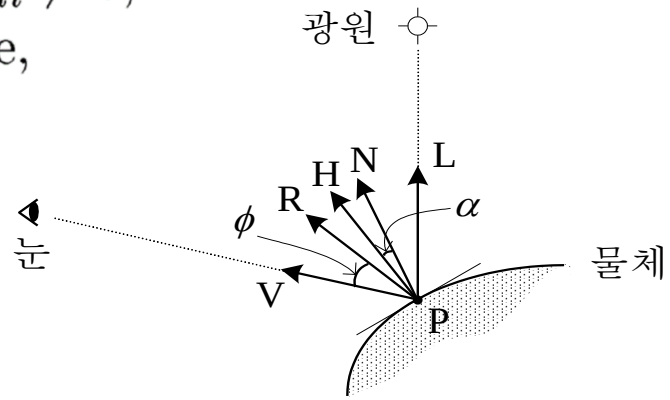
교과서 수준의 조명 공식과의 비교

$$I_\lambda = I_{a\lambda} \cdot k_{a\lambda} + \sum_{i=0}^{m-1} f_{att}(d_i) \cdot I_{l_i\lambda} \cdot \{k_{d\lambda} \cdot (N \circ L_i) + k_{s\lambda} \cdot (N \circ H_i)^n\}$$

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \\
&\sum_{i=0}^{n-1} (att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}] \\
\mathbf{c}_{sec} &= (0, 0, 0, 1)
\end{aligned}$$

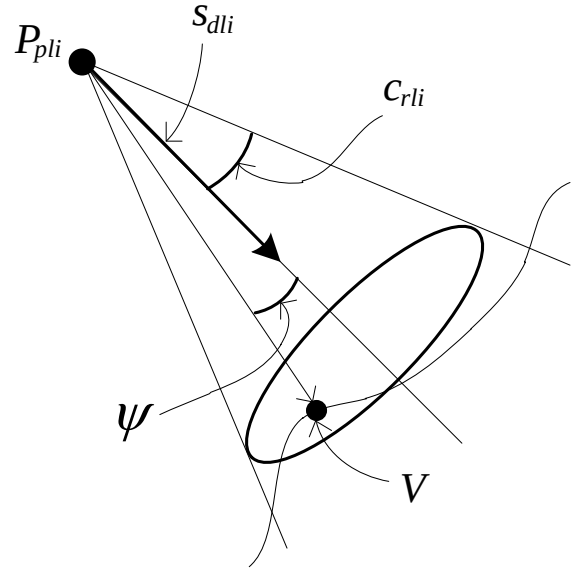
$$\mathbf{d}_1 \odot \mathbf{d}_2 = \max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0\} \qquad f_i = \begin{cases} 1, & \mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli} \neq 0, \\ 0, & \text{otherwise}, \end{cases}$$

$$\mathbf{h}_i = \begin{cases} \overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_e, & v_{bs} = \texttt{TRUE}, \\ \overrightarrow{\mathbf{VP}}_{pli} + (0 \quad 0 \quad 1)^T, & v_{bs} = \texttt{FALSE}, \end{cases}$$



- 스폿 광원 효과

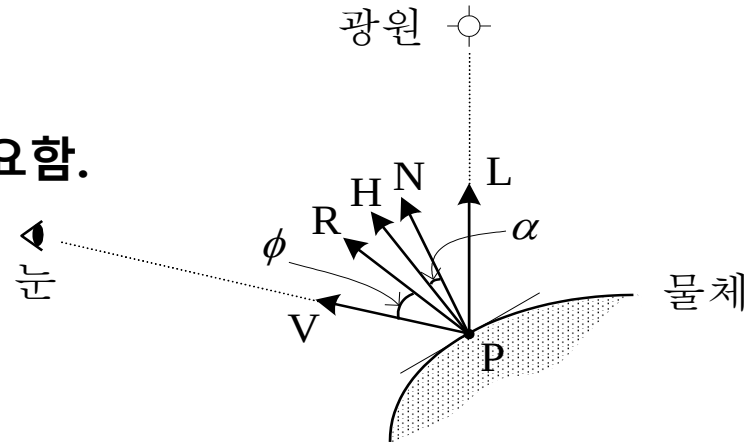
$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases}$$



- 빛의 광원 효과

$$att_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i}\|\mathbf{VP}_{pli}\| + k_{2i}\|\mathbf{VP}_{pli}\|^2}, & \text{if } \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases}$$

Lighting Parameters in Fixed-Function OpenGL



- **Geometry at vertices**
  - **기본적으로 동일한 좌표계에서의 P, N, V, L이 필요함.**
  - 기존 OpenGL에서는 EC에서 각 꼭지점에 대해 lighting 계산을 수행함.
    - **Position in EC ← Transform vertex into EC**
    - **Normal direction in EC ← Transform normal into EC**
    - View direction in EC ← -Position in EC or (0, 0, 1, 0)
- **Material parameters**
  - 물체를 구성하는 물질의 빛 반사 성질을 정의함.
- **Light Parameters**
  - 광원의 위치/방향 및 광원의 색깔을 정의함.
    - **Light direction in EC ← Transform light position/direction into EC**
- **Light model parameters**
  - Lighting 계산 전반에 걸친 성질을 정의함.

✓ 교재 3장 6절을 읽을 것!

물질 인자(Material Parameter)의 설정

$$\mathbf{c}_{pri} = \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \sum_{i=0}^{n-1} (att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}]$$

인자	타입	디폴트 값	설명
$\mathbf{a}_{cm}$	색깔	(0.2, 0.2, 0.2, 1.0)	물질의 앰비언트 색깔
$\mathbf{d}_{cm}$	색깔	(0.8, 0.8, 0.8, 1.0)	물질의 난반사 색깔
$\mathbf{s}_{cm}$	색깔	(0.0, 0.0, 0.0, 1.0)	물질의 정반사 색깔
$\mathbf{e}_{cm}$	색깔	(0.0, 0.0, 0.0, 1.0)	물질의 방사 색깔
s_{rm}	실수	0.0	물질의 정반사 지수 (범위: [0.0, 128.0])

표 3.2: 물질 인자

```
GLfloat mat_ambient[4] = {0.03784, 0.30712, 0.03784, 1.0};
GLfloat mat_diffuse[4] = {0.07568, 0.61424, 0.07568, 1.0};
GLfloat mat_specular[4] = {0.633, 0.727811, 0.633, 1.0};
GLfloat mat_shininess = 22.0;
      :
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);
```

광원 인자(Light Source Parameter)의 설정

$$\mathbf{c}_{pri} = \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \sum_{i=0}^{n-1} (att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}]$$

인자	타입	디폴트 값	설명
$\mathbf{a}_{cli}$	색깔	(0.0, 0.0, 0.0, 1.0)	i번 광원의 앰비언트 색깔
$\mathbf{d}_{cli}(i=0)$	색깔	(1.0, 1.0, 1.0, 1.0)	0번 광원의 난반사 색깔
$\mathbf{d}_{cli}(i>0)$	색깔	(0.0, 0.0, 0.0, 1.0)	i번 광원의 난반사 색깔
$\mathbf{s}_{cli}(i=0)$	색깔	(1.0, 1.0, 1.0, 1.0)	0번 광원의 정반사 색깔
$\mathbf{s}_{cli}(i>0)$	색깔	(0.0, 0.0, 0.0, 1.0)	i번 광원의 정반사 색깔
$\mathbf{P}_{pli}$	위치	(0.0, 0.0, 1.0, 0.0)	i번 광원의 위치
$\mathbf{s}_{dli}$	방향	(0.0, 0.0, -1.0)	i번 광원의 스폿 광원 방향
s_{rli}	실수	0.0	i번 광원의 스폿 광원 지수 (범위: [0.0, 128.0])
c_{rli}	실수	180.0	i번 광원의 스폿 광원 절단 각도 (범위: [0.0, 90.0] 또는 180.0)
k_{0i}	실수	1.0	i번 광원의 상수 감쇠 인자 (범위: [0.0, ∞])
k_{1i}	실수	0.0	i번 광원의 1차 감쇠 인자 (범위: [0.0, ∞])
k_{2i}	실수	0.0	i번 광원의 2차 감쇠 인자 (범위: [0.0, ∞])

표 3.3: 광원 인자

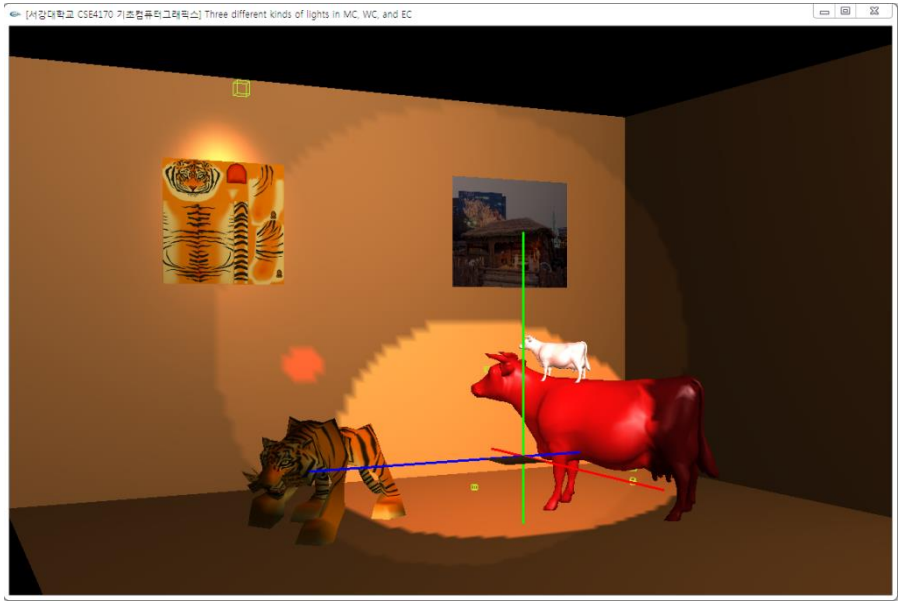
```
GLfloat light_position[4] = {0.0, 5.0, 5.0, 1.0};
GLfloat light_ambient_color[4] = {0.23, 0.23, 0.23, 1.0};
GLfloat light_diff_spec_color[4] = {0.95, 0.95, 0.95, 1.0};
      :
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient_color);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diff_spec_color);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_diff_spec_color);
```

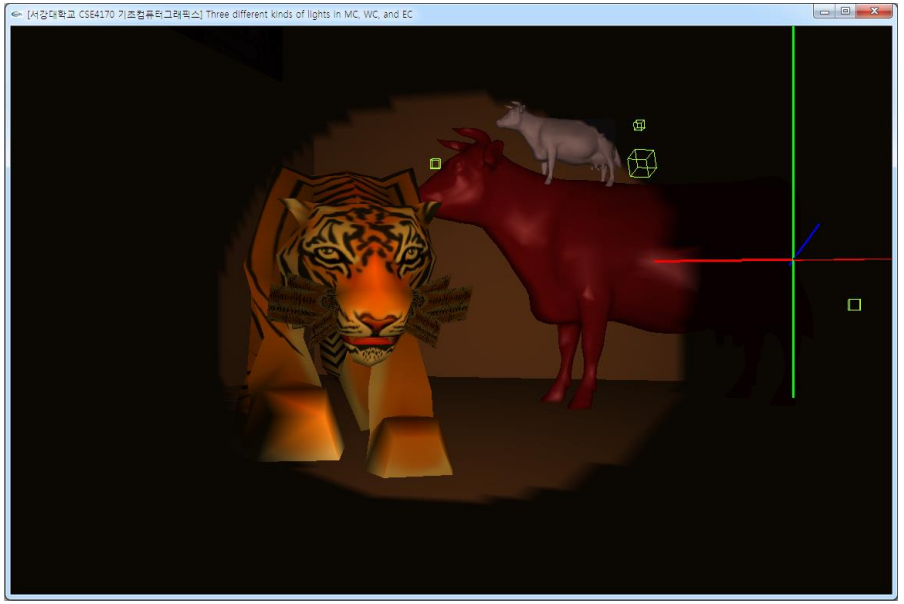
기하 속성을 가지는 광원 인자의 설정

- 광원의 위치 또는 방향 값이 어떤 좌표계에서의 의미를 가질까: **MC, WC, EC**?
  - glLight*(*) 함수를 사용하여 기하 정보를 설정하면, 바로 그 순간의 모델뷰 행렬을 사용하여 이 값들을 눈 좌표계로 변환
  - 기존의 OpenGL fixed-function pipeline에서는 눈좌표계에서 라이팅 계산.
  - 쉐이더를 사용할 경우 MC, WC, EC (또는 LC) 중 적절한 좌표계를 사용할 것.

```
GLfloat light_position[4] = {0.0, 5.0, 0.0, 1.0};
      ⋮
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(20.0, 1.0, 1.0, 100.0);


glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// Line (a)                EC
gluLookAt(4.0, 8.0, 14.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
// Line (b)                WC
glTranslatef(0.0, -1.5, 0.0);
glRotatef(-90.0, 1.0, 0.0, 0.0);
// Line (c)                MC
glBegin(GL_TRIANGLES);
      ⋮ // draw objects
glEnd();
```





조명 모델 인자(Lighting Model Parameter)의 설정

$$\mathbf{c}_{pri} = \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \sum_{i=0}^{n-1} (att_i)(spot_i)[\mathbf{a}_{cm} * \mathbf{a}_{cli} + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{srm}\mathbf{s}_{cm} * \mathbf{s}_{cli}]$$

인자	타입	디폴트 값	설명
$\mathbf{a}_{cs}$	색깔	(0.2, 0.2, 0.2, 1.0)	장면의 전역 앰비언트 광원 색깔
v_{bs}	부울형	FALSE	눈 좌표계에서의 관찰자 시점 TRUE : (0, 0, 0) , FALSE : (0, 0, ∞)
c_{es}	열거형	SINGLE_COLOR	정반사 계산 분리 여부
t_{bs}	부울형	FALSE	양면 조명 모드 사용 여부

표 3.4: 조명 모델 인자

```
void glLightModel{if}(GLenum pname, TYPE param);
void glLightModle{if}v(GLenum pname, TYPE *param);
```

✓ 교재 3.6절을 읽을 것!

라이팅 관련 OpenGL 프로그래밍

1) 기하 물체의 각 꼭지점을 기술할 때 그 점에서의 법선 벡터도 함께 정의.
2) 기하 물체의 표면에서의 빛의 반사 형태를 결정하는 물질 성질을 정의.
3) 사용하려는 조명 모델의 일반적인 성질을 설정.
4) 사용하려는 광원의 성질을 정의.
5) 조명 계산을 하도록 명시적으로 선언하고, 적절하게 광원을 점등.
6) 깊이 버퍼 기능을 사용.
7) 조명 계산에 영향을 미치는 다각형에 대한 성질을 설정.
    a. 다각형의 방향 설정.
    b. 뒷면 제거 여부 결정.
    c. 보이는 다각형을 어떻게 그릴 지 결정.
8) 사용하려는 쉐이딩 모델을 설정.

✓ 교재 3.6절을 읽을 것!

라이팅 관련 OpenGL 프로그래밍

1) 기하 물체의 각 꼭지점을 기술할 때 그 점에서의 법선 벡터도 함께 정의.
2) 기하 물체의 표면에서의 빛의 반사 형태를 결정하는 물질 성질을 정의.
3) 사용하려는 조명 모델의 일반적인 성질을 설정.
4) 사용하려는 광원의 성질을 정의.
5) 조명 계산을 하도록 명시적으로 선언하고, 적절하게 광원을 점등.
6) 깊이 버퍼 기능을 사용.
7) 조명 계산에 영향을 미치는 다각형에 대한 성질을 설정.
    a. 다각형의 방향 설정.
    b. 뒷면 제거 여부 결정.
    c. 보이는 다각형을 어떻게 그릴 지 결정.
8) 사용하려는 쉐이딩 모델을 설정.

```
typedef struct {
    int nvertex;
    GLfloat vertex[4][3];
    GLfloat normal[4][3];
} Polygon;
Polygon *teapot; int npoly_teapot;

void draw_teapot(void) {
  int i, j;
  Polygon *ptr;

  glPushMatrix();
    glRotatef(rotation_angle, 0.0, 1.0, 0.0);
    glTranslatef(0.0, -1.5, 0.0);
    glRotatef(-90.0, 1.0, 0.0, 0.0);

    i = 0; ptr = teapot;
    while(i++ < npoly_teapot) {
      glBegin(GL_POLYGON);
      for (j = 0; j < ptr->nvertex; j++) {
        glNormal3fv(ptr->normal[j]); ← 1
        glVertex3fv(ptr->vertex[j]);
      }
      glEnd();
      ptr++;
    }
  glPopMatrix();
}
```

```
void draw_axes_and_light_info (void) {
  glDisable(GL_LIGHTING);

  glBegin(GL_LINES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(-3.0, 0.0, 0.0); // x-axis
    glVertex3f(3.0, 0.0, 0.0);
    // draw the other axes and light line
  glEnd();

  if (cur_light_source_type != DIRECTIONAL) {
    glPushMatrix();
      glTranslatef(0.0, 5.0, 5.0);
      glutWireCube(0.05);
    glPopMatrix();
  }

  glColor3f(0.0, 1.0, 1.0);
  if (cur_lighting_mode == YES)
    glEnable(GL_LIGHTING);
}
```

```
void init_OpenGL(void) {
  glClearColor(0.05, 0.25, 0.05, 1.0);
  glEnable(GL_DEPTH_TEST); ← 6

  glMaterialfv(GL_FRONT, GL_AMBIENT, ← 2
                              mat_ambient); ...


  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ← 3
                            global_ambient);
  glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER,
                             1.0);

  glLightfv(GL_LIGHT0, GL_AMBIENT,
                   light_ambient_color); ← 4

    ...


  glEnable(GL_LIGHTING); ← 5
  glEnable(GL_LIGHT0);

  glFrontFace(GL_CCW); ← 7
  glCullFace(GL_BACK);
  glEnable(GL_CULL_FACE);

  glPolygonMode(GL_FRONT_AND_BACK, ← 7
                                   GL_FILL);
  glShadeModel(GL_SMOOTH); ← 8

    ...
```

```
  // Projection transformation here

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(4.0, 8.0, 14.0, 0.0, 0.0, 0.0,
                                0.0, 1.0, 0.0);
  glLightfv(GL_LIGHT0, GL_POSITION,
    light_position); ← 4

    ...
}


void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | ← 6
                       GL_DEPTH_BUFFER_BIT);

  draw_axes_and_light_info();
  draw_teapot();
  glutSwapBuffers();
}
```

퐁의 조명 모델과 OpenGL

```
/*
 *  teapots.c from OpenGL Redbook Examples
 *  This program demonstrates lots of material properties
 */
#include <stdlib.h>
#include <GL/glut.h>

void init(void) {
   GLfloat ambient[] = {0.0, 0.0, 0.0, 1.0};
   GLfloat diffuse[] = {1.0, 1.0, 1.0, 1.0};
   GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
   GLfloat position[] = {0.0, 3.0, 3.0, 0.0};

   GLfloat lmodel_ambient[] = {0.2, 0.2, 0.2, 1.0};
   GLfloat local_view[] = {0.0};

   glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
   glLightfv(GL_LIGHT0, GL_POSITION, position);
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
   glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, local_view);

   glFrontFace(GL_CW);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);

   glEnable(GL_NORMALIZE);
   glEnable(GL_DEPTH_TEST);
   glClearColor(0.0, 0.0, 0.0, 0.0);
}
```

```
/*
 * Move object into position.  Use 3rd through 12th
 * parameters to specify the material property.  Draw a teapot.
 */
void renderTeapot(GLfloat x, GLfloat y,
   GLfloat ambr, GLfloat ambg, GLfloat ambb,
   GLfloat difr, GLfloat difg, GLfloat difb,
   GLfloat specr, GLfloat specg, GLfloat specb, GLfloat shine) {
   GLfloat mat[4];

   glPushMatrix();
   glTranslatef(x, y, 0.0);
   mat[0] = ambr; mat[1] = ambg; mat[2] = ambb; mat[3] = 1.0;
   glMaterialfv(GL_FRONT, GL_AMBIENT, mat);
   mat[0] = difr; mat[1] = difg; mat[2] = difb;
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat);
   mat[0] = specr; mat[1] = specg; mat[2] = specb;
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat);
   glMaterialf(GL_FRONT, GL_SHININESS, shine * 128.0);
   glutSolidTeapot(1.0);
   glPopMatrix();
}
```

```
/*
 *  First column:  emerald, jade, obsidian, pearl, ruby, turquoise
 *  2nd column:  brass, bronze, chrome, copper, gold, silver
 *  3rd column:  black, cyan, green, red, white, yellow plastic
 *  4th column:  black, cyan, green, red, white, yellow rubber
 */
void display(void) {
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   renderTeapot(2.0, 17.0, 0.0215, 0.1745, 0.0215, 0.07568, 0.61424, 0.07568, 0.633, 0.727811, 0.633, 0.6);
   renderTeapot(2.0, 14.0, 0.135, 0.2225, 0.1575, 0.54, 0.89, 0.63, 0.316228, 0.316228, 0.316228, 0.1);
   renderTeapot(2.0, 11.0, 0.05375, 0.05, 0.06625, 0.18275, 0.17, 0.22525, 0.332741, 0.328634, 0.346435, 0.3);
   renderTeapot(2.0, 8.0, 0.25, 0.20725, 0.20725, 1, 0.829, 0.829, 0.296648, 0.296648, 0.296648, 0.088);
   renderTeapot(2.0, 5.0, 0.1745, 0.01175, 0.01175, 0.61424, 0.04136, 0.04136, 0.727811, 0.626959, 0.626959, 0.6);
   renderTeapot(2.0, 2.0, 0.1, 0.18725, 0.1745, 0.396, 0.74151, 0.69102, 0.297254, 0.30829, 0.306678, 0.1);
   renderTeapot(6.0, 17.0, 0.329412, 0.223529, 0.027451, 0.780392, 0.568627, 0.113725, 0.992157, 0.941176, 0.807843, 0.21794872);
   renderTeapot(6.0, 14.0, 0.2125, 0.1275, 0.054, 0.714, 0.4284, 0.18144, 0.393548, 0.271906, 0.166721, 0.2);
   renderTeapot(6.0, 11.0, 0.25, 0.25, 0.25, 0.4, 0.4, 0.4, 0.774597, 0.774597, 0.774597, 0.6);
   renderTeapot(6.0, 8.0, 0.19125, 0.0735, 0.0225, 0.7038, 0.27048, 0.0828, 0.256777, 0.137622, 0.086014, 0.1);
   renderTeapot(6.0, 5.0, 0.24725, 0.1995, 0.0745, 0.75164, 0.60648, 0.22648, 0.628281, 0.555802, 0.366065, 0.4);
   renderTeapot(6.0, 2.0, 0.19225, 0.19225, 0.19225, 0.50754, 0.50754, 0.50754, 0.508273, 0.508273, 0.508273, 0.4);
   renderTeapot(10.0, 17.0, 0.0, 0.0, 0.0, 0.01, 0.01, 0.01, 0.50, 0.50, 0.50, .25);
   renderTeapot(10.0, 14.0, 0.0, 0.1, 0.06, 0.0, 0.50980392, 0.50980392, 0.50196078, 0.50196078, 0.50196078, .25);
   renderTeapot(10.0, 11.0, 0.0, 0.0, 0.0, 0.1, 0.35, 0.1, 0.45, 0.55, 0.45, .25);
```

```
    renderTeapot(10.0, 8.0, 0.0, 0.0, 0.0, 0.5, 0.0, 0.0, 0.7, 0.6, 0.6, .25);
    renderTeapot(10.0, 5.0, 0.0, 0.0, 0.0, 0.55, 0.55, 0.55, 0.70, 0.70, 0.70, .25);
    renderTeapot(10.0, 2.0, 0.0, 0.0, 0.0, 0.5, 0.5, 0.0, 0.60, 0.60, 0.50, .25);
    renderTeapot(14.0, 17.0, 0.02, 0.02, 0.02, 0.01, 0.01, 0.01, 0.4, 0.4, 0.4, .078125);
    renderTeapot(14.0, 14.0, 0.0, 0.05, 0.05, 0.4, 0.5, 0.5, 0.04, 0.7, 0.7, .078125);
    renderTeapot(14.0, 11.0, 0.0, 0.05, 0.0, 0.4, 0.5, 0.4, 0.04, 0.7, 0.04, .078125);
    renderTeapot(14.0, 8.0, 0.05, 0.0, 0.0, 0.5, 0.4, 0.4, 0.7, 0.04, 0.04, .078125);
    renderTeapot(14.0, 5.0, 0.05, 0.05, 0.05, 0.5, 0.5, 0.5, 0.7, 0.7, 0.7, .078125);
    renderTeapot(14.0, 2.0, 0.05, 0.05, 0.0, 0.5, 0.5, 0.4, 0.7, 0.7, 0.04, .078125);
    glFlush();
}
```

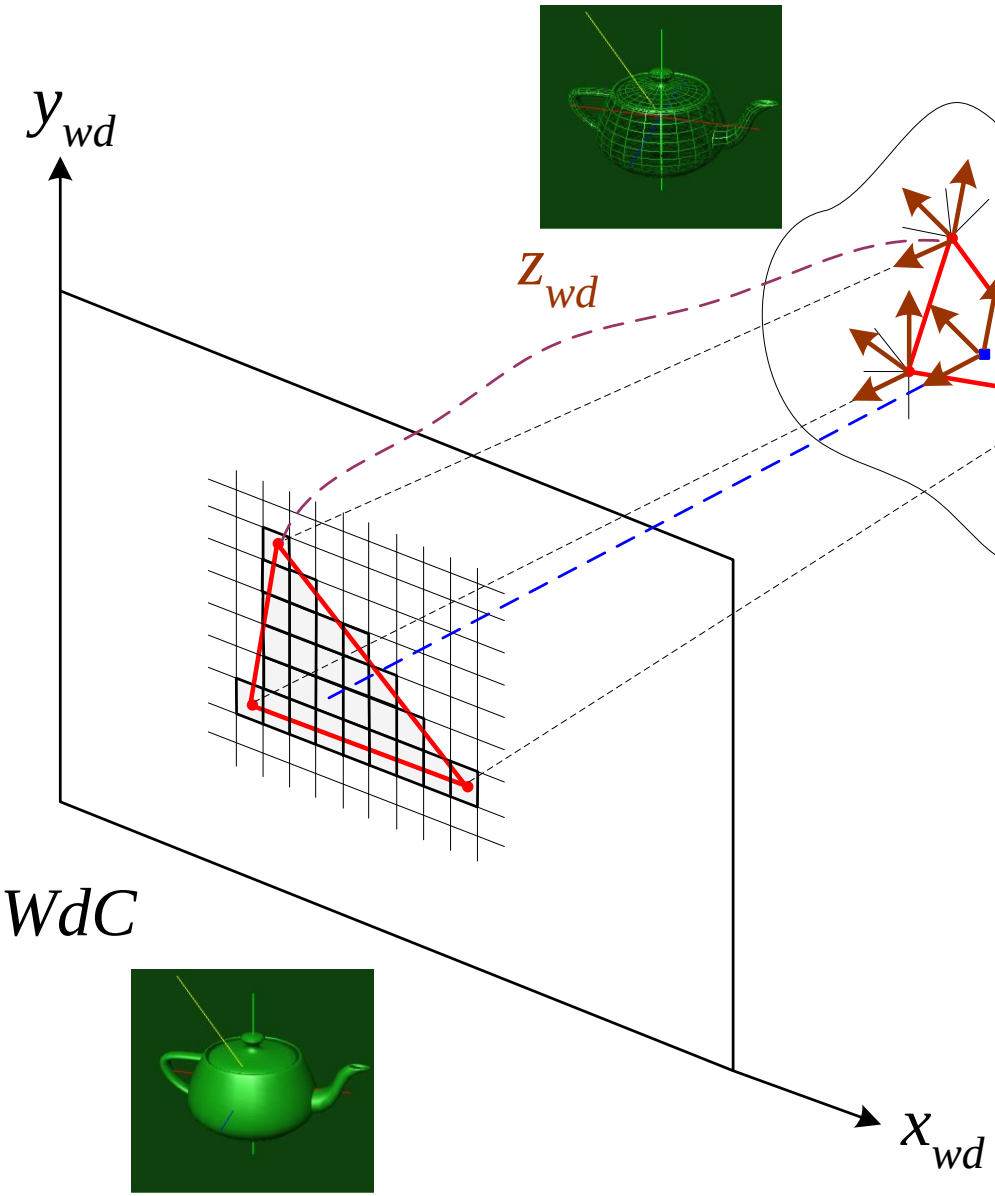
```
void reshape(int w, int h) {
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
      glOrtho(0.0, 16.0, 0.0, 16.0*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
   else
      glOrtho(0.0, 16.0*(GLfloat)w/(GLfloat)h, 0.0, 16.0, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
}

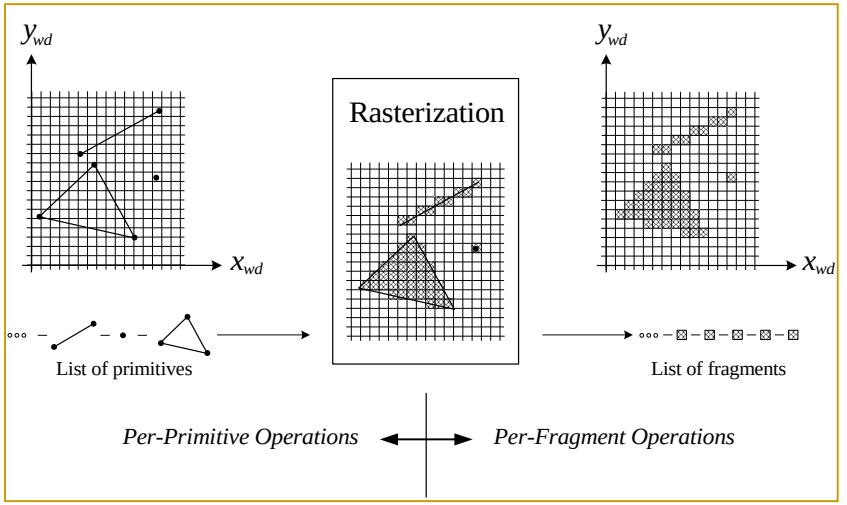
void keyboard(unsigned char key, int x, int y) {
   switch (key) { case 27: exit(0); break;  }
}

int main(int argc, char **argv) {
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(800, 960);
   glutInitWindowPosition(50,50);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutKeyboardFunc (keyboard);
   glutMainLoop();
   return 0;
}
```

Polygonal Shading Models:
Gouraud Shading versus Phong Shading

래스터화 과정과 다면체 모델에 대한 쉐이딩 모델의 관계





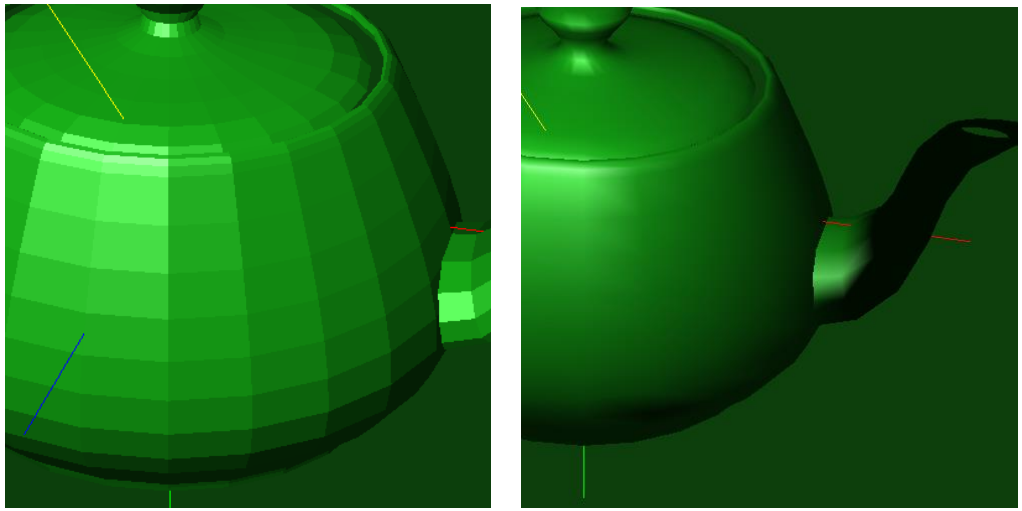
다면체 모델에 대한 꼭지점별 쉐이딩과 화소별 쉐이딩

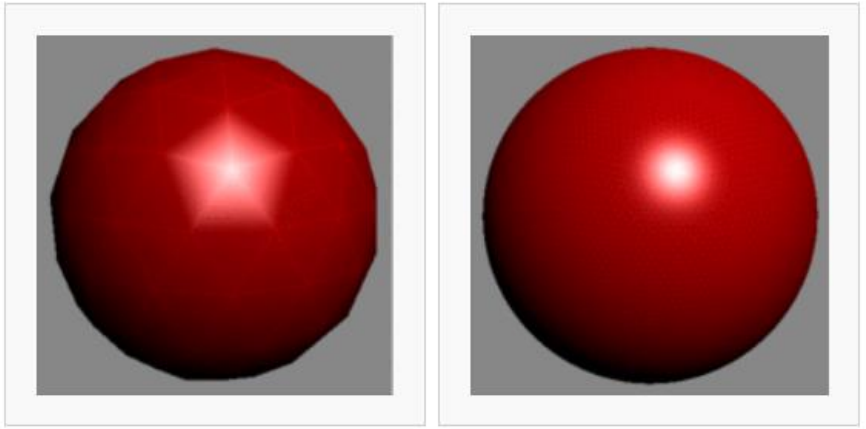
- 다면체 모델의 표면의 어느 지점에 대해 조명 모델 공식을 적용할 것인가?

- **꼭지점별 쉐이딩 (per-vertex shading)**
  ① 다면체 모델의 각 꼭지점에 대하여 퐁의 조명 모델을 적용하여 쉐이딩된 색깔을 계산한 후 꼭지점 속성으로 설정.
  ② 래스터화 과정에서 다각형 꼭지점의 색깔 속성을 각 화소에 대해 선형보간을 하여 쉐이딩 색깔을 계산.
  – '(과거의) 전통적인 fixed-function pipeline' 에서 주로 사용되던 방식.

예) Flat shading과 Gouraud shading

© Wikipedia

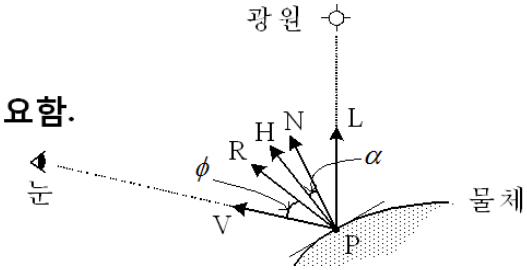




Gouraud shading의 문제

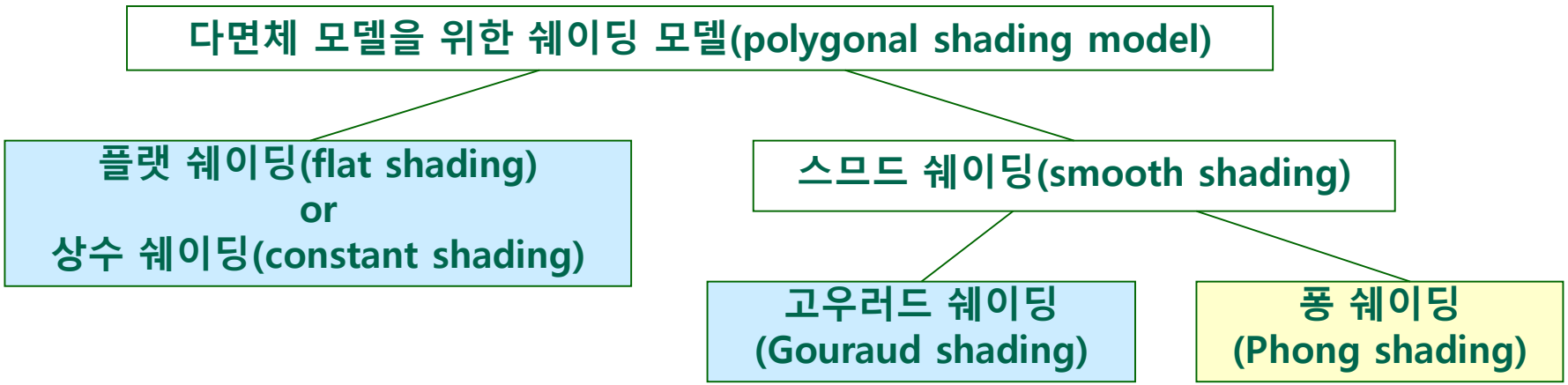
Lighting Parameters in Fixed-Function OpenGL

- **Geometry at vertices**
  - **기본적으로 동일한 좌표계에서의 P, N, V, L이 필요함.**
  - 기존 OpenGL에서는 EC에서 각 꼭지점에 대해 lighting 계산을 수행함.



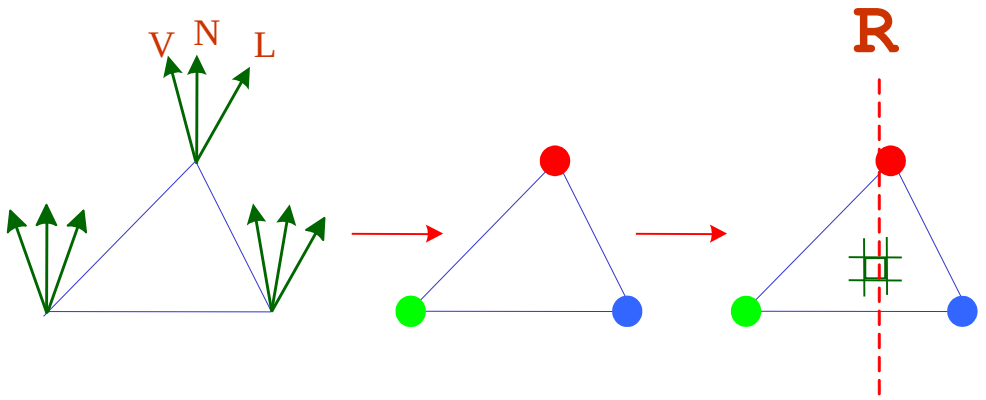
- **화소별 쉐이딩 (per-pixel shading)**
  - ① 각 꼭지점에 대하여 법선 벡터 등 라이팅 계산에 필요한 기하 정보를 속성으로 설정.
  - ② 래스터화 과정에서 각 꼭지점 속성인 기하 정보들을 선형 보간 하여 각 화소를 통하여 보이는 물체 지점에서의 (선형 보간된) 기하 정보를 계산.
  - ③ 이 기하 정보와 퐁의 조명 모델을 사용하여 화소에 칠할 색깔을 계산.

  예) Phong shading

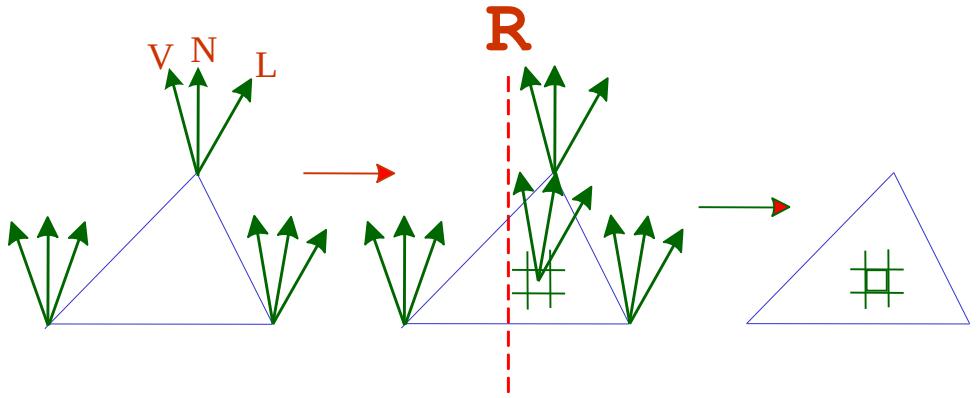


꼭지점별 쉐이딩과 화소별 쉐이딩의 비교

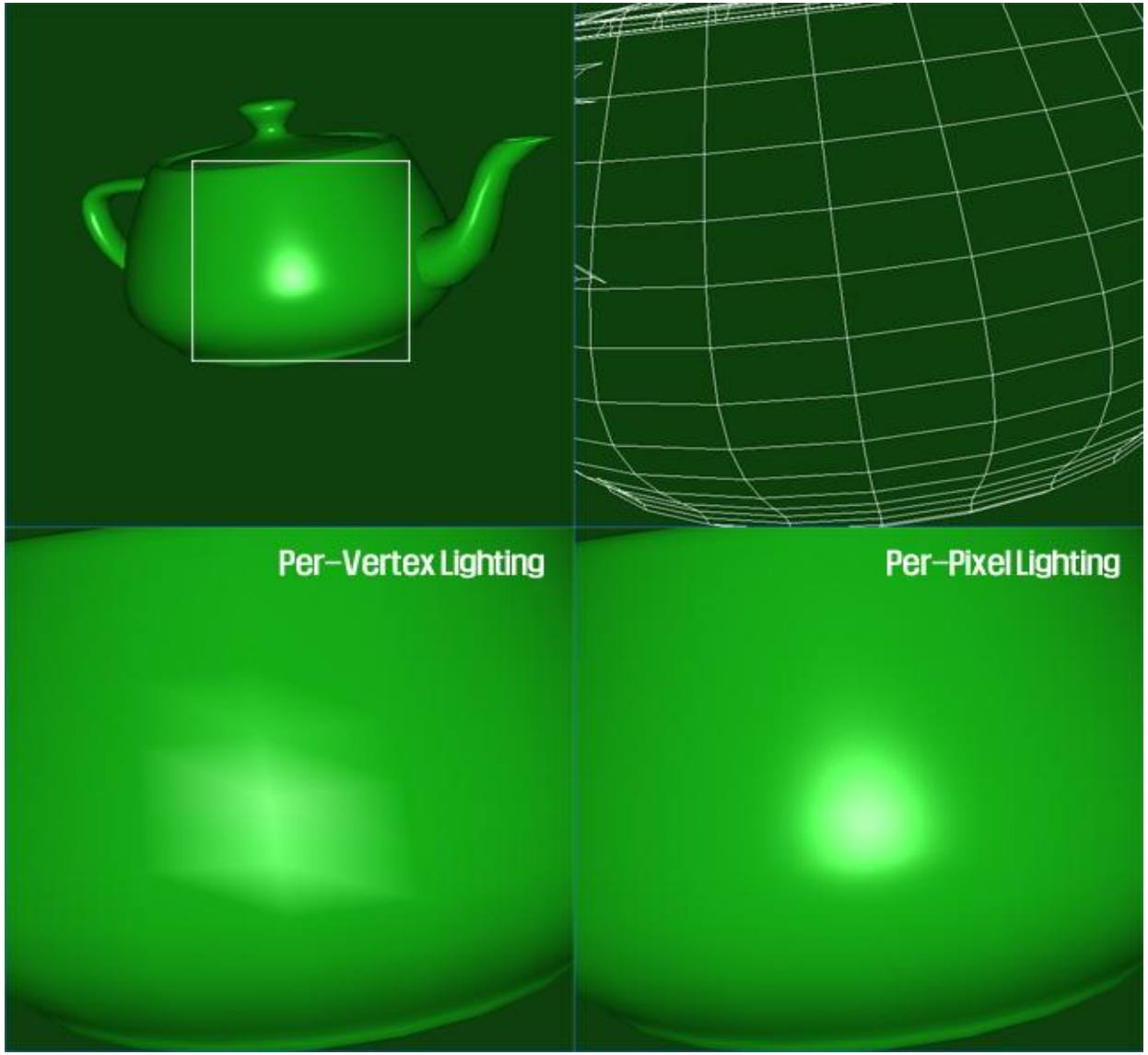
- **Gouraud shading**

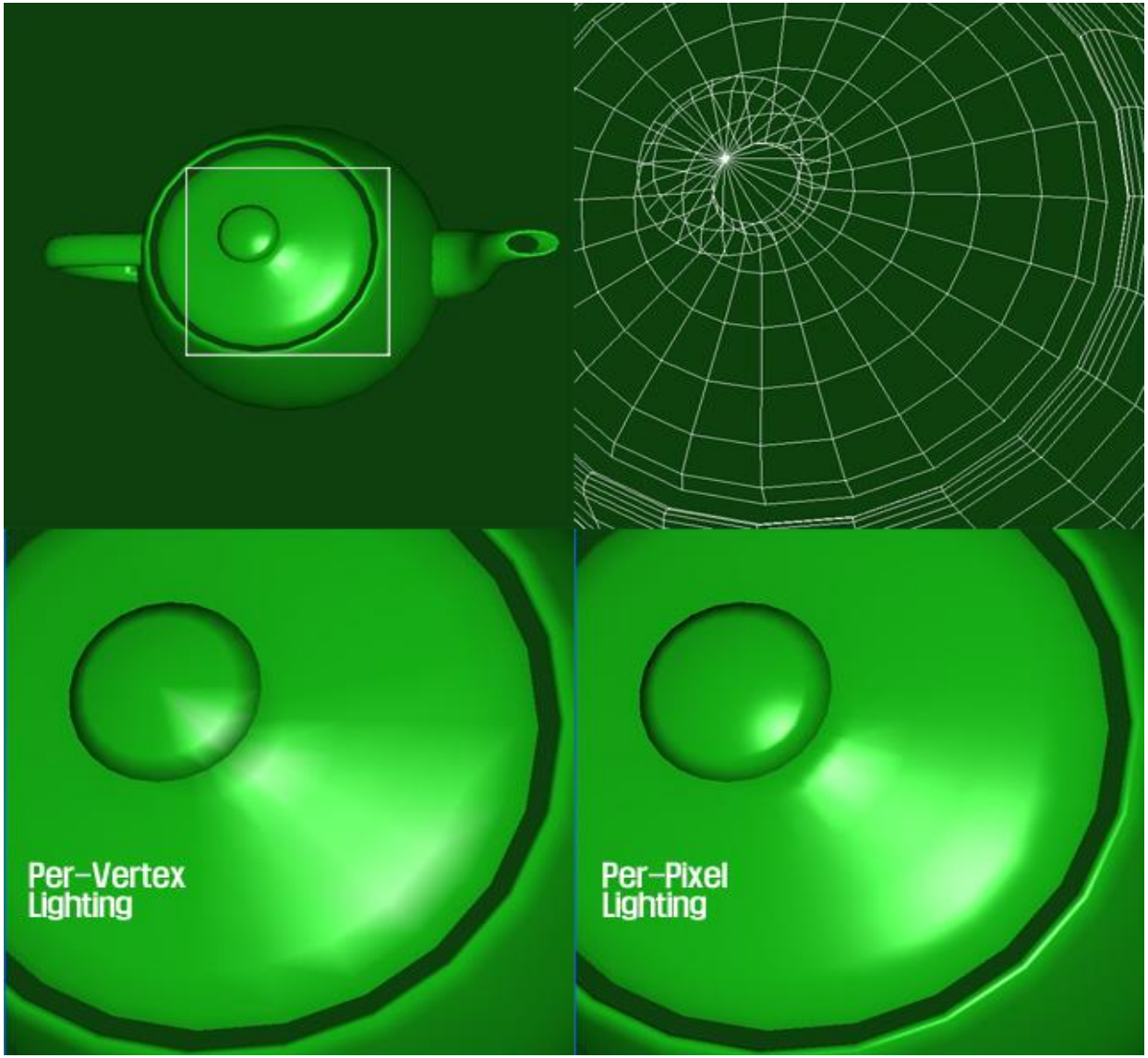


- **Phong shading**



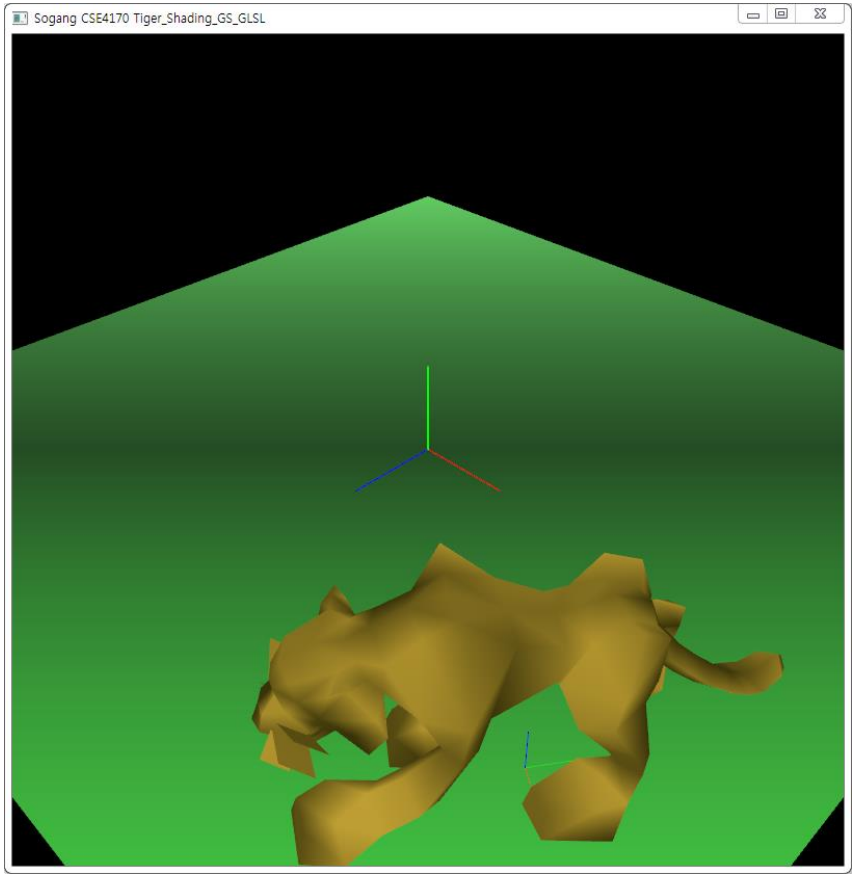
✓ **기존의 fixed-function pipeline에서는 기본적으로 Gouraud shading을 제공하였으나, 최근의 programmable GPU를 장착한 그래픽스 하드웨어에서는 Phong shading 효과를 쉽게 생성할 수 있음.**


Per-Vertex Lighting
Per-Pixel Lighting


Per-Vertex
Lighting
Per-Pixel
Lighting

GLSL 4.3 프로그램 작성 예 4

-- OpenGL 3D Gouraud Shading



A Gouraud Shading Example in GLSL 4.3

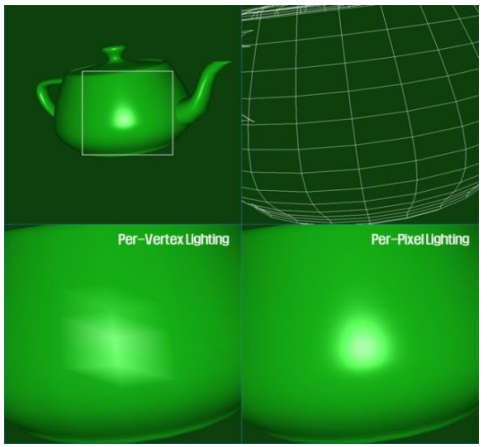
- 이 예제 프로그램에서는
  - Vertex shader
    - 꼭지점의 좌표를 OC에서 CC로 기하 변환을 하고 (**gl_Position**),
    - EC로 꼭지점 좌표와 법선 벡터로 변환을 한 후, 해당 꼭지점에 대하여 퐁의 조명 모델 계산을 수행 후, 그 결과를 out variable에 담아 넘겨줌 (**v_shaded_color**).
  - Fragment shader
    - 입력으로 래스터화 과정에서 보간을 통하여 얻어진 해당 픽셀을 통하여 보이는 물체 지점에 대한 색깔을 픽셀 색깔로 정함 (**final_color**).

<Fragment Shader>

```
in vec4 v_shaded_color;

layout (location = 0) out vec4 final_color;

void main(void) {
    final_color = v_shaded_color;
}
```



Shaders

<Vertex Shader>

```
...
uniform vec4 u_global_ambient_color;
#define NUMBER_OF_LIGHTS_SUPPORTED 4
uniform LIGHT u_light[NUMBER_OF_LIGHTS_SUPPORTED];
uniform MATERIAL u_material;
uniform mat4 u_ModelViewProjectionMatrix;
uniform mat4 u_ModelViewMatrix;
uniform mat3 u_ModelViewMatrixInvTrans;

const float zero_f = 0.0f;
const float one_f = 1.0f;

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
out vec4 v_shaded_color;

vec4 lighting_equation(in vec3 P_EC, in vec3 N_EC) {
    vec4 color_sum;
    float local_scale_factor, tmp_float;
    vec3 L_EC;

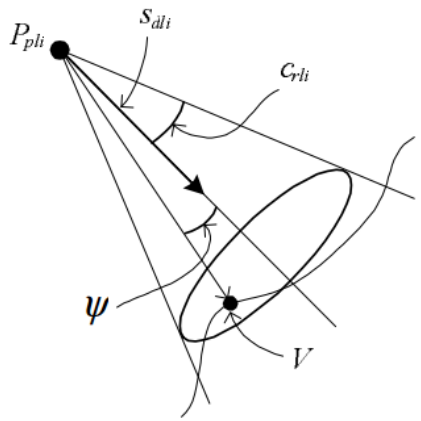
    color_sum = u_material.emissive_color + u_global_ambient_color * u_material.ambient_color;
    for (int i = 0; i < NUMBER_OF_LIGHTS_SUPPORTED; i++) {
       if (!u_light[i].light_on) continue;
```

$\begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$:directional light

```
local_scale_factor = one_f;
if (u_light[i].position.w != zero_f) { // point light source
    L_EC = u_light[i].position.xyz - P_EC.xyz;
    if (u_light[i].light_attenuation_factors.w  != zero_f) {
        vec4 tmp_vec4;
        tmp_vec4.x = one_f;
        tmp_vec4.z = dot(L_EC, L_EC);
        tmp_vec4.y = sqrt(tmp_vec4.z);
        tmp_vec4.w = zero_f;
        local_scale_factor = one_f/dot(tmp_vec4, u_light[i].light_attenuation_factors);
    }
    L_EC = normalize(L_EC);
    if (u_light[i].spot_cutoff_angle < 180.0f) { // [0.0f, 90.0f] or 180.0f
        float spot_cutoff_angle = clamp(u_light[i].spot_cutoff_angle, zero_f, 90.0f);
        vec3 spot_dir = normalize(u_light[i].spot_direction);
        tmp_float = dot(-L_EC, spot_dir);
        if (tmp_float >= cos(radians(u_light[i].spot_cutoff_angle))) {
            tmp_float = pow(tmp_float, u_light[i].spot_exponent);
        }
        else
            tmp_float = zero_f;
        local_scale_factor *= tmp_float;
        }
    }
```

$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$: positional light (w≠0)

$$att_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i}\|\mathbf{VP}_{pli}\| + k_{2i}\|\mathbf{VP}_{pli}\|^2}, & \text{if } \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases}$$



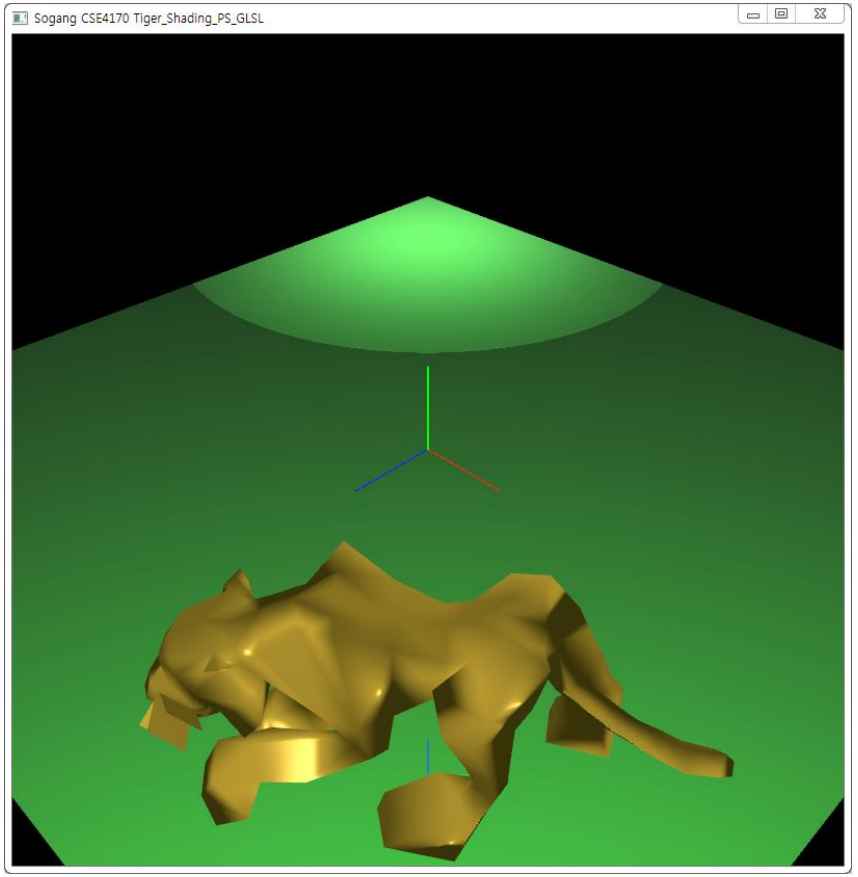
$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases}$$

```
        else // directional light source
            L_EC = normalize(u_light[i].position.xyz);
        if (local_scale_factor > zero_f) {
            // local ambient reflection
            vec4 local_color_sum = u_light[i].ambient_color * u_material.ambient_color;
            // diffuse reflection
            tmp_float = max(zero_f, dot(N_EC, L_EC)); // see if it is a backlight
            local_color_sum += u_light[i].diffuse_color*u_material.diffuse_color*tmp_float;
            // specular reflection
            vec3 H_EC = normalize(L_EC - normalize(P_EC));
            tmp_float = max(zero_f, dot(N_EC, H_EC));
            if (tmp_float > zero_f)
                local_color_sum += u_light[i].specular_color*u_material.specular_color*pow(tmp_float, u_material.specular_exponent);
            color_sum += local_scale_factor*local_color_sum;
        }
    }
    return color_sum;
}
void main(void) {
                vec3 position_EC = vec3(u_ModelViewMatrix*vec4(a_position, one_f));
                vec3 normal_EC = normalize(u_ModelViewMatrixInvTrans*a_normal);

                v_shaded_color = lighting_equation(position_EC, normal_EC);
                gl_Position = u_ModelViewProjectionMatrix*vec4(a_position, one_f);
}
```

GLSL 4.3 프로그램 작성 예 5

-- OpenGL 3D Phong Shading



A Phong Shading Example in GLSL 4.3

- 이 예제 프로그램에서는
  - Vertex shader
    - 꼭지점의 좌표를 OC에서 CC로 기하 변환을 하고 (**gl_Position**),
    - EC로 꼭지점 좌표와 법선 벡터로 변환을 한 후, 그 결과를 out variable에 담아 넘겨줌 (**v_position_EC, v_normal_EC**).
  - Fragment shader
    - 입력으로 래스터화 과정에서 보간을 통하여 얻어진 해당 픽셀에 대하여 퐁의 조명 모델 계산을 수행 후, 그 결과를 픽셀 색깔로 정함 (**final_color**).

Shaders

<Vertex Shader>

```
uniform mat4 u_ModelViewProjectionMatrix;
uniform mat4 u_ModelViewMatrix;
uniform mat3 u_ModelViewMatrixInvTrans;

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
out vec3 v_position_EC;
out vec3 v_normal_EC;

void main(void) {
		v_position_EC = vec3(u_ModelViewMatrix*vec4(a_position, 1.0f));
		v_normal_EC = normalize(u_ModelViewMatrixInvTrans*a_normal);

		gl_Position = u_ModelViewProjectionMatrix*vec4(a_position, 1.0f);
}
```

<Fragment Shader>

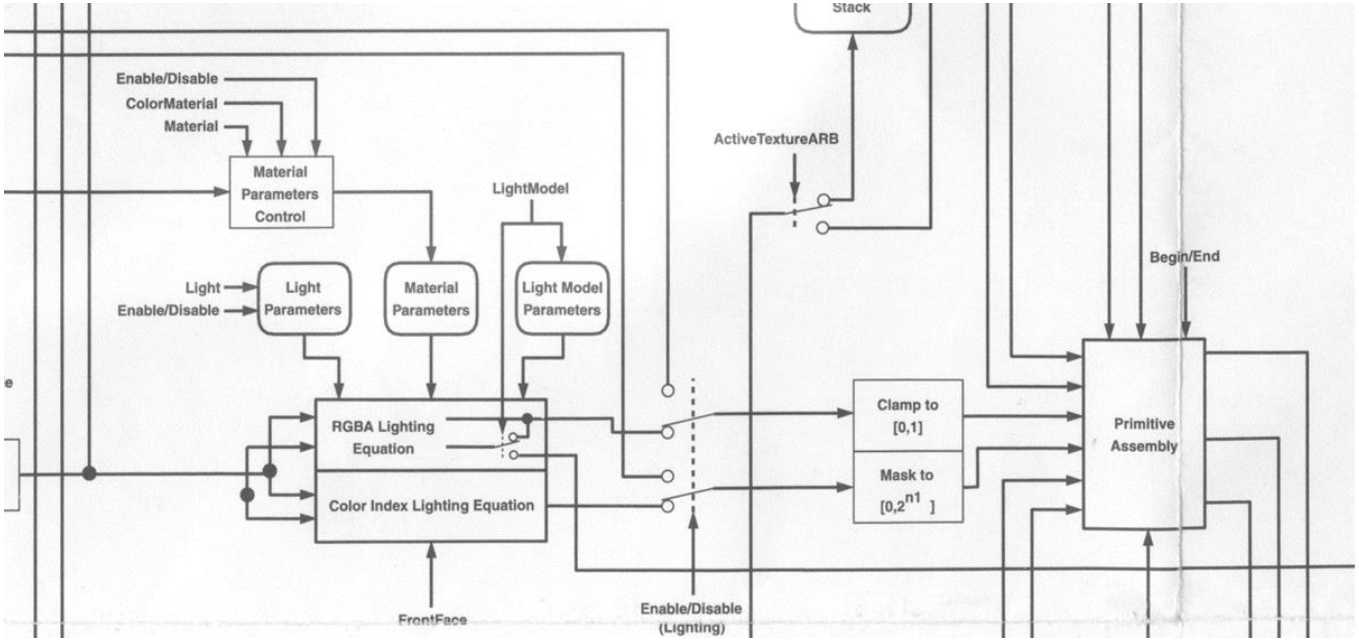
```
...
uniform vec4 u_global_ambient_color;
#define NUMBER_OF_LIGHTS_SUPPORTED 4
uniform LIGHT u_light[NUMBER_OF_LIGHTS_SUPPORTED];
uniform MATERIAL u_material;

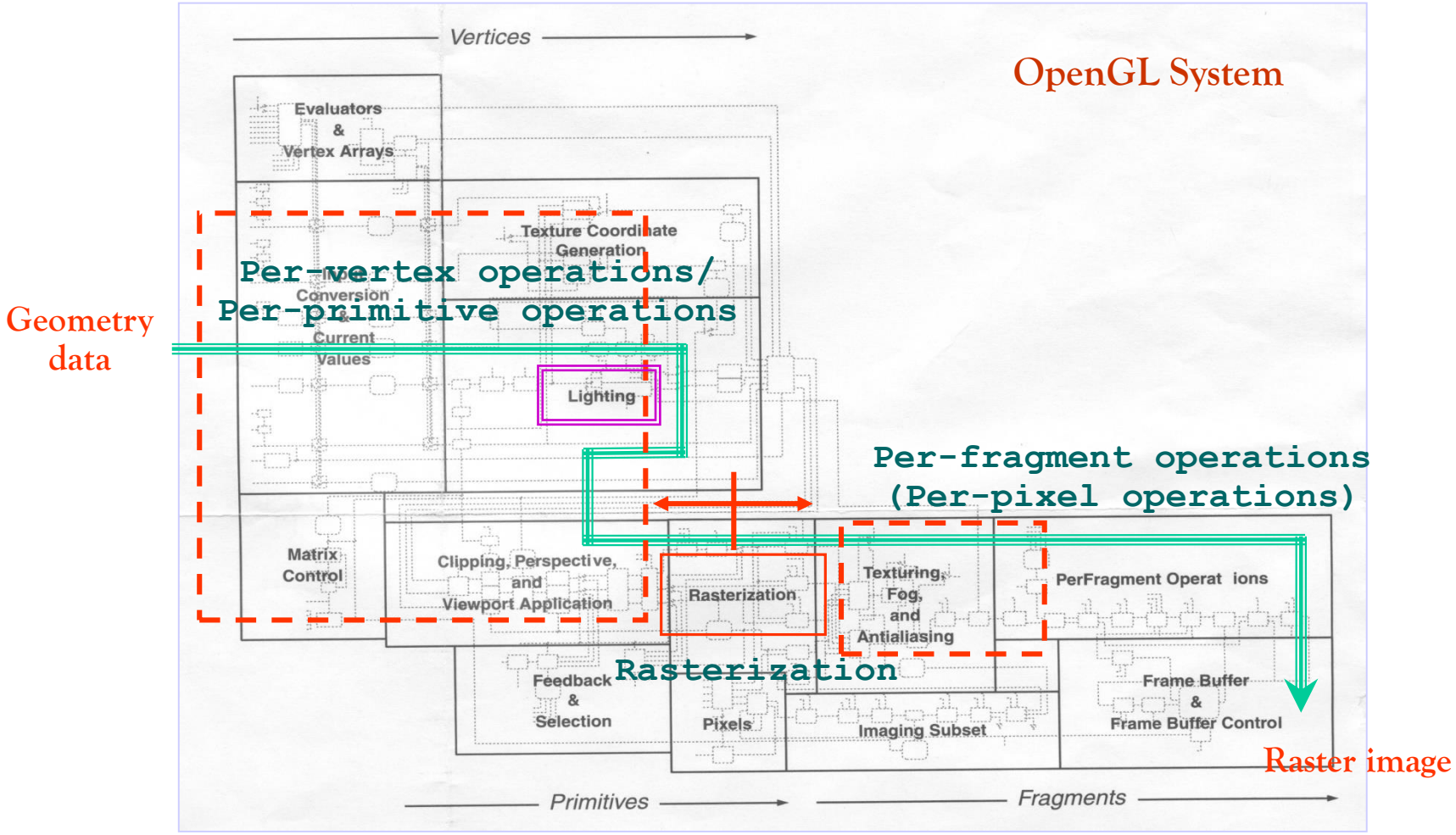
const float zero_f = 0.0f;
const float one_f = 1.0f;

in vec3 v_position_EC;
in vec3 v_normal_EC;
layout ( location = 0 ) out vec4 final_color;
...
void main(void) {
    // lighting_equation( ) is similar to  the Gouraud shading example
    final_color = lighting_equation(v_position_EC, normalize(v_normal_EC));
}
```

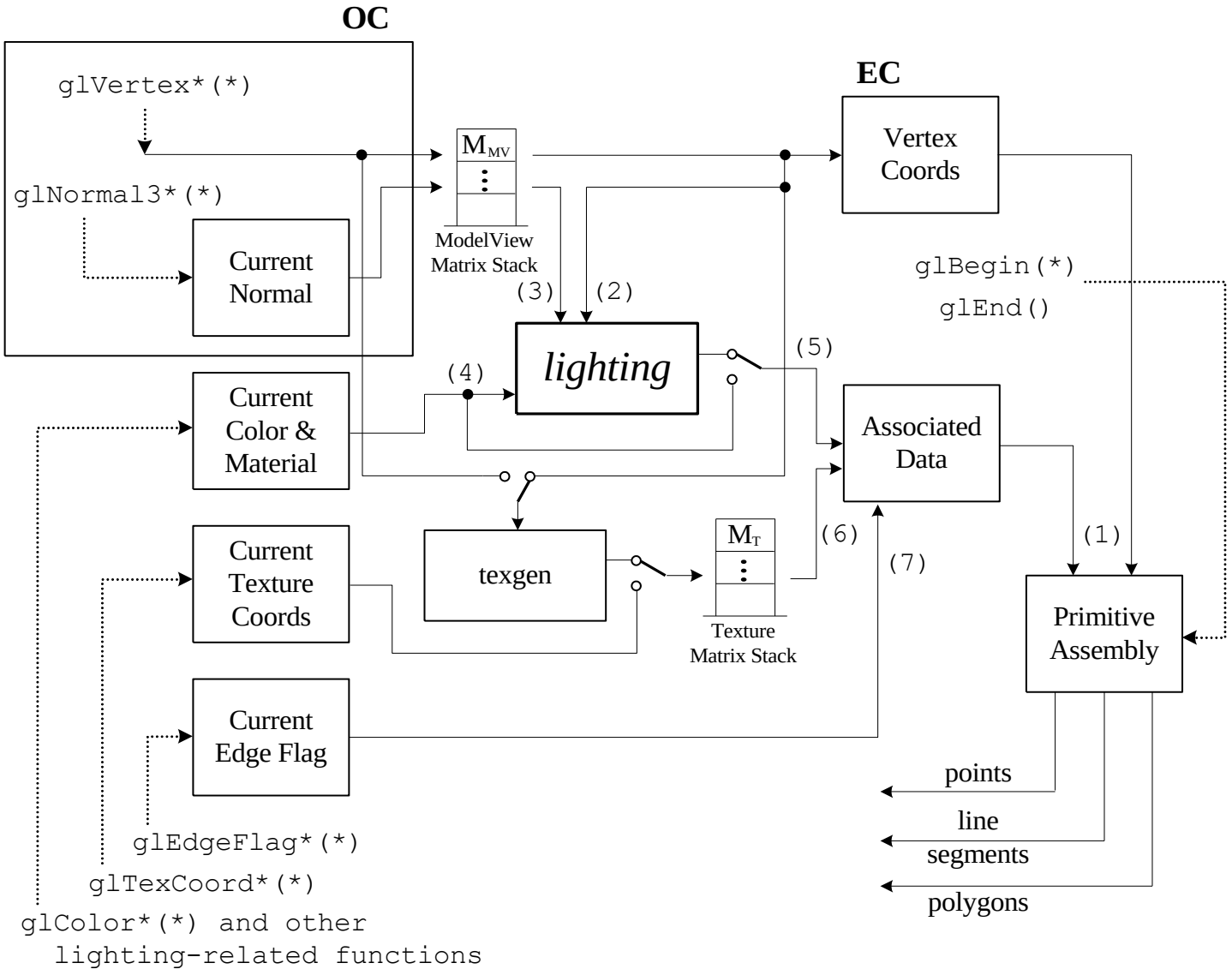
(Fixed-function) OpenGL 렌더링 파이프라인에서는

- 다면체 모델의 각 꼭지점에 대하여,
  - 물체 좌표계에서 설정된 꼭지점과 법선 벡터를 모델뷰 행렬 스택의 행렬을 사용하여 눈 좌표계(EC)로 변환하여 각 꼭지점에 대한 색깔 속성을 결정함.
    - glColor*(*) 함수로 설정한 현재 색깔(current color)로 지정, 또는
    - 라이팅 모듈에서 지역 조명 모델에 기반을 둔 퐁의 조명 모델 사용하여 쉐이딩이 된 색깔(shaded color)을 계산.



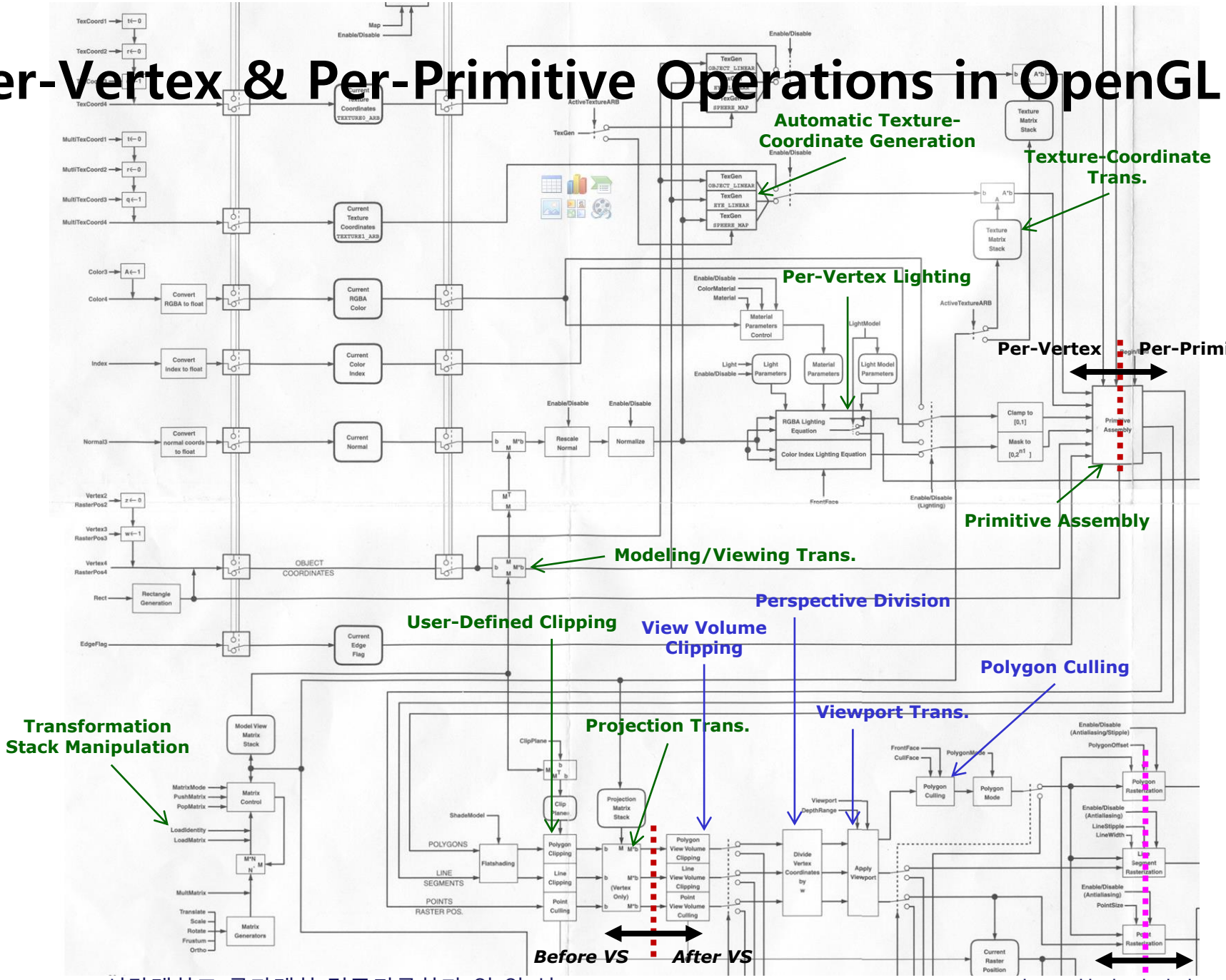

Vertices
OpenGL System
Evaluators & Vertex Arrays
Texture Coordinate Generation
Conversion
Current Values
Lighting
Per-vertex operations/
Per-primitive operations
Geometry data
Per-fragment operations
(Per-pixel operations)
Matrix Control
Clipping, Perspective, and Viewport Application
Rasterization
Texturing, Fog, and Antialiasing
PerFragment Operat ions
Rasterization
Feedback & Selection
Pixels
Imaging Subset
Frame Buffer & Frame Buffer Control
Raster image
Primitives
Fragments

프리미티브들의 조합


OC
EC
glVertex*(*)
glNormal3*(*)
Current Normal
M_{MV}
ModelView Matrix Stack
(3)
(2)
Vertex Coords
glBegin(*)
glEnd()
lighting
(5)
(4)
Current Color & Material
Associated Data
Current Texture Coords
texgen
M_T
Texture Matrix Stack
(6)
(7)
(1)
Primitive Assembly
Current Edge Flag
points
line segments
polygons
glEdgeFlag*(*)
glTexCoord*(*)
glColor*(*) and other lighting-related functions

Summary on Per-Vertex and Per-Primitive Operations in OpenGL

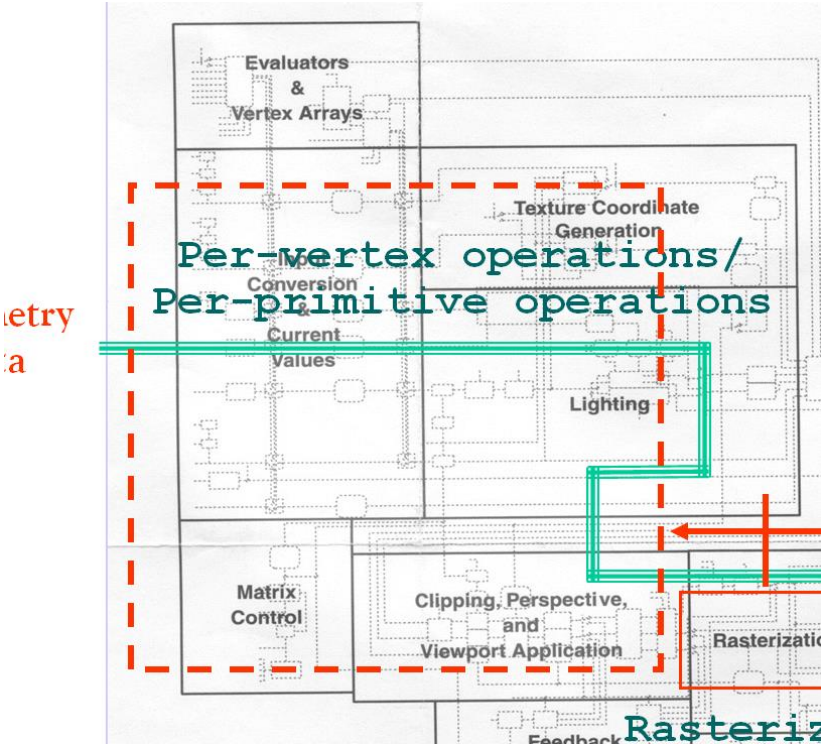
Per-Vertex & Per-Primitive Operations in OpenGL



Per-Vertex/Per-Primitive Operations과 Vertex Shader와의 관계

- 현재 vertex shader는 OC에서 CC까지의 연산 (view volume clipping 직전)을 대치함.

- 과연 어떤 주요 계산들이 수행되었는가?
  - **Modeing/Viewing/Projection transformation**
  - Per-vertex lighting in EC
  - Primitive assembly
  - Automatic texture-coordinate generation
  - User-defined clipping
  - Etc.



- 과연 이러한 개념의 계산들을 vertex shader에서 어떻게 처리할 것인가?
  - 무엇보다도 **OC에서 CC까지의 기하 변환**을 반드시 수행 해야 함.

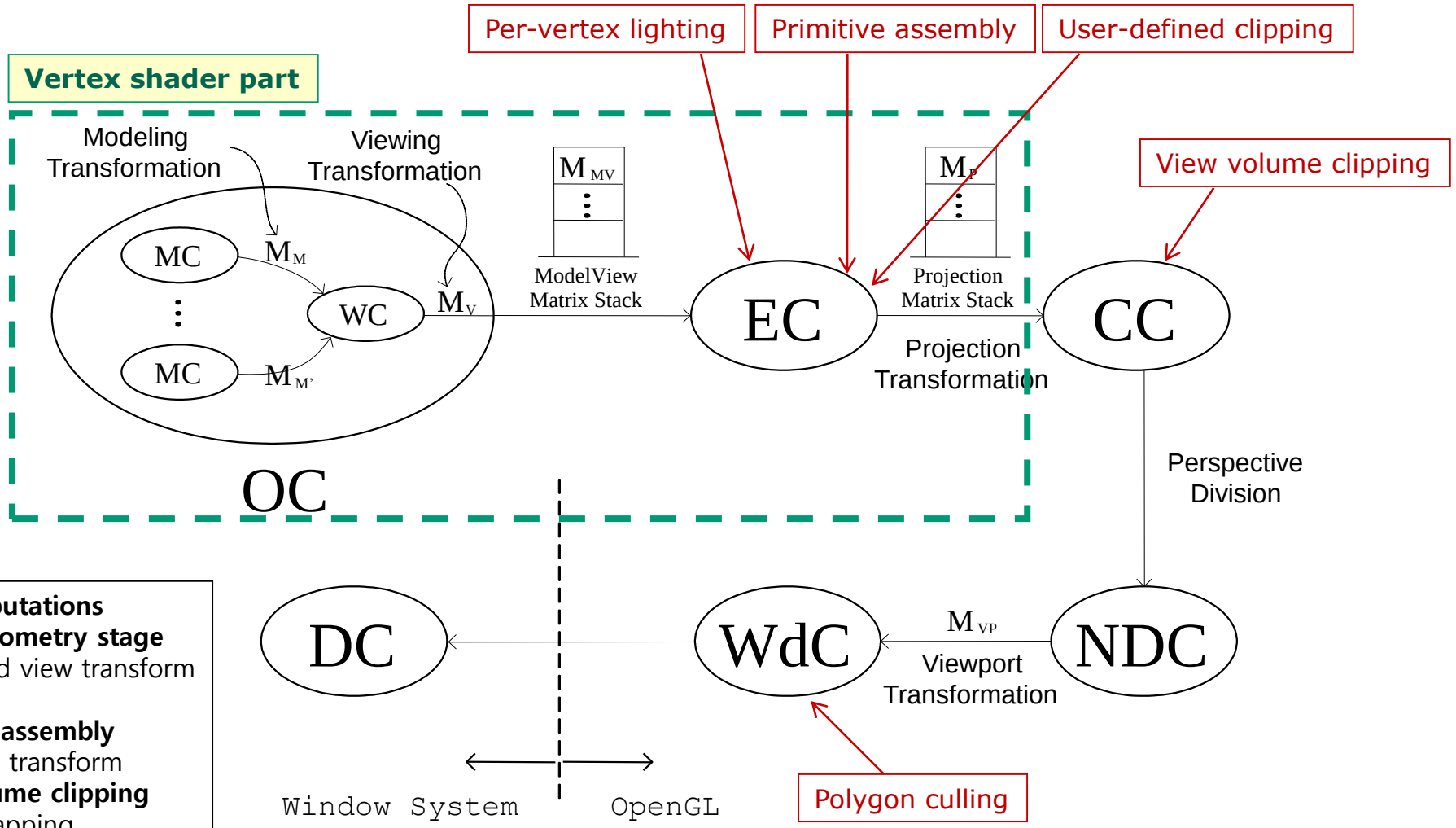
```
void main() {

   ...

  gl_Position = mvp_matrix*a_position;

 }
```

Vertex Shader Part in Fixed-Function OpenGL Pipeline



Basic computations in geometry stage
- Model and view transform
- **Lighting**
- **Primitive assembly**
- Projection transform
- **View volume clipping**
- Screen mapping
- **Polygon culling**
- **Etc.**

최신 GPU에서는 geometry stage의 어느 부분이 vertex shader 및 geometry shader 형태로 프로그램이 가능할까?

After Rasterization

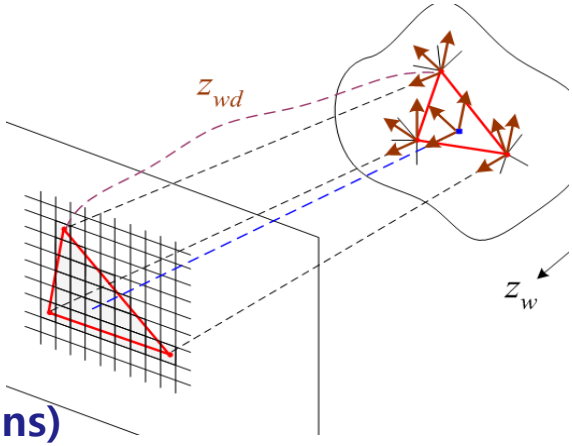
-- Pixel Shading and Raster Operations

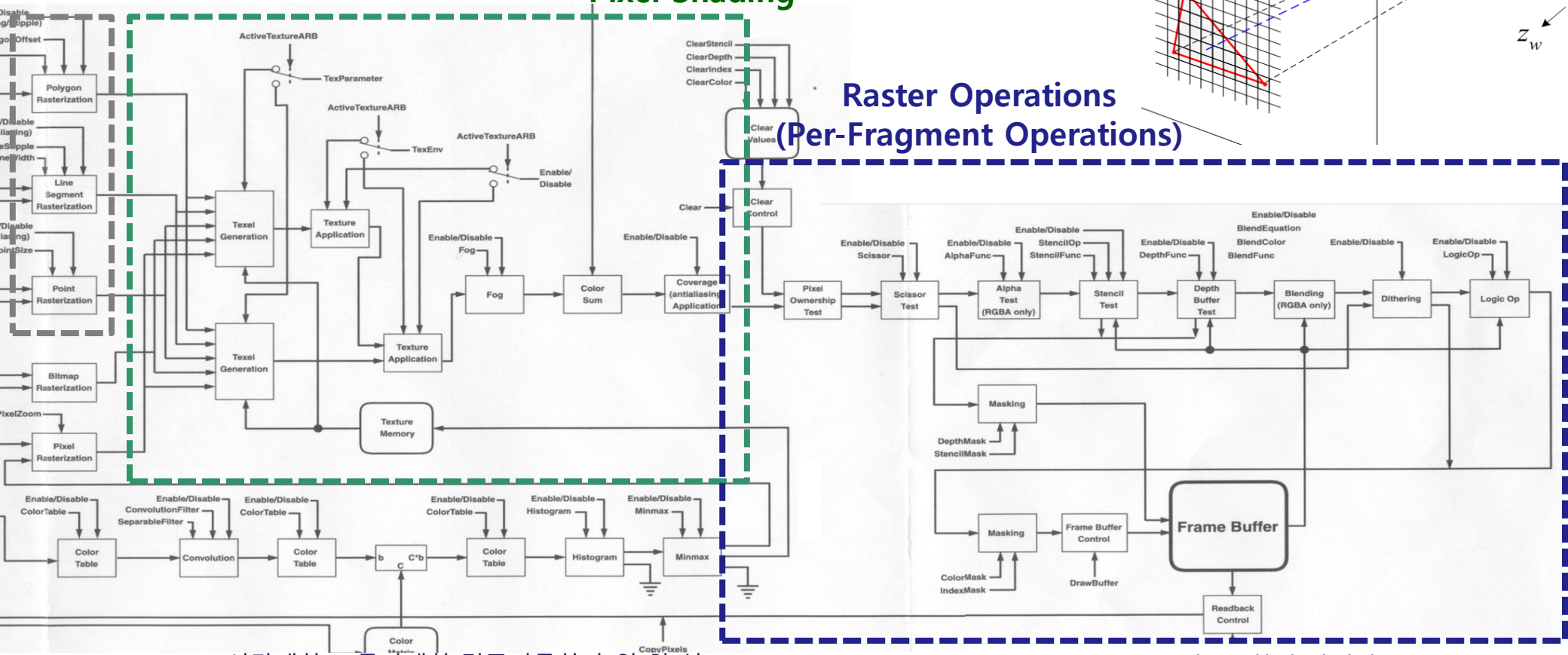
Where Are We Now?

- 래스터화 과정을 통하여 생성되는 각 **프래그먼트(fragment)**는 해당 pixel에 대하여 다음과 같은 **per-fragment attributes**가 부여됨.

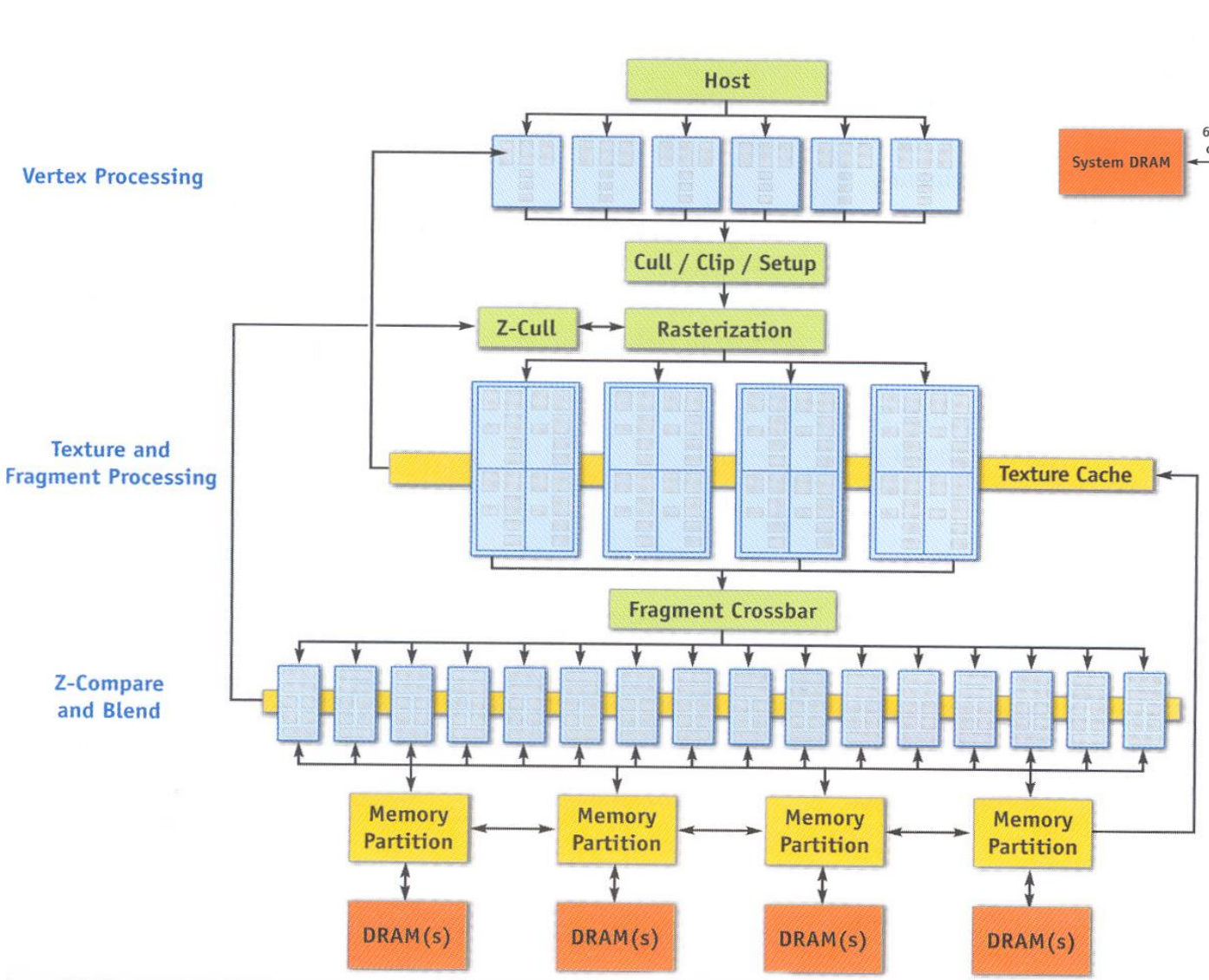
$$[(x_{ij}, y_{ij}), z_{ij}, 1/w_{ij}, (r_{ij}, g_{ij}, b_{ij}, a_{ij}), (s_{ij}, t_{ij}, r_{ij}, q_{ij}), f_{ij} \ldots]$$

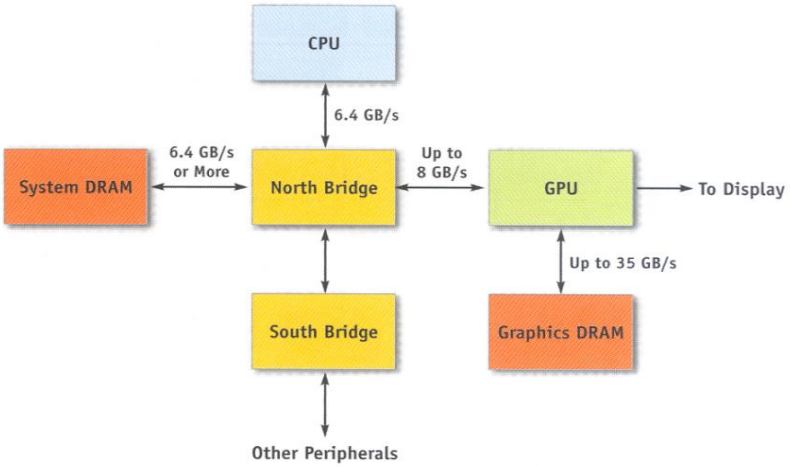
  - 기본적으로 해당 픽셀의 최종 색깔을 계산하는데 필요한 정보가 주어져 있음.





2004년: GeForce 6 Series (NV40) - 6800 Ultra

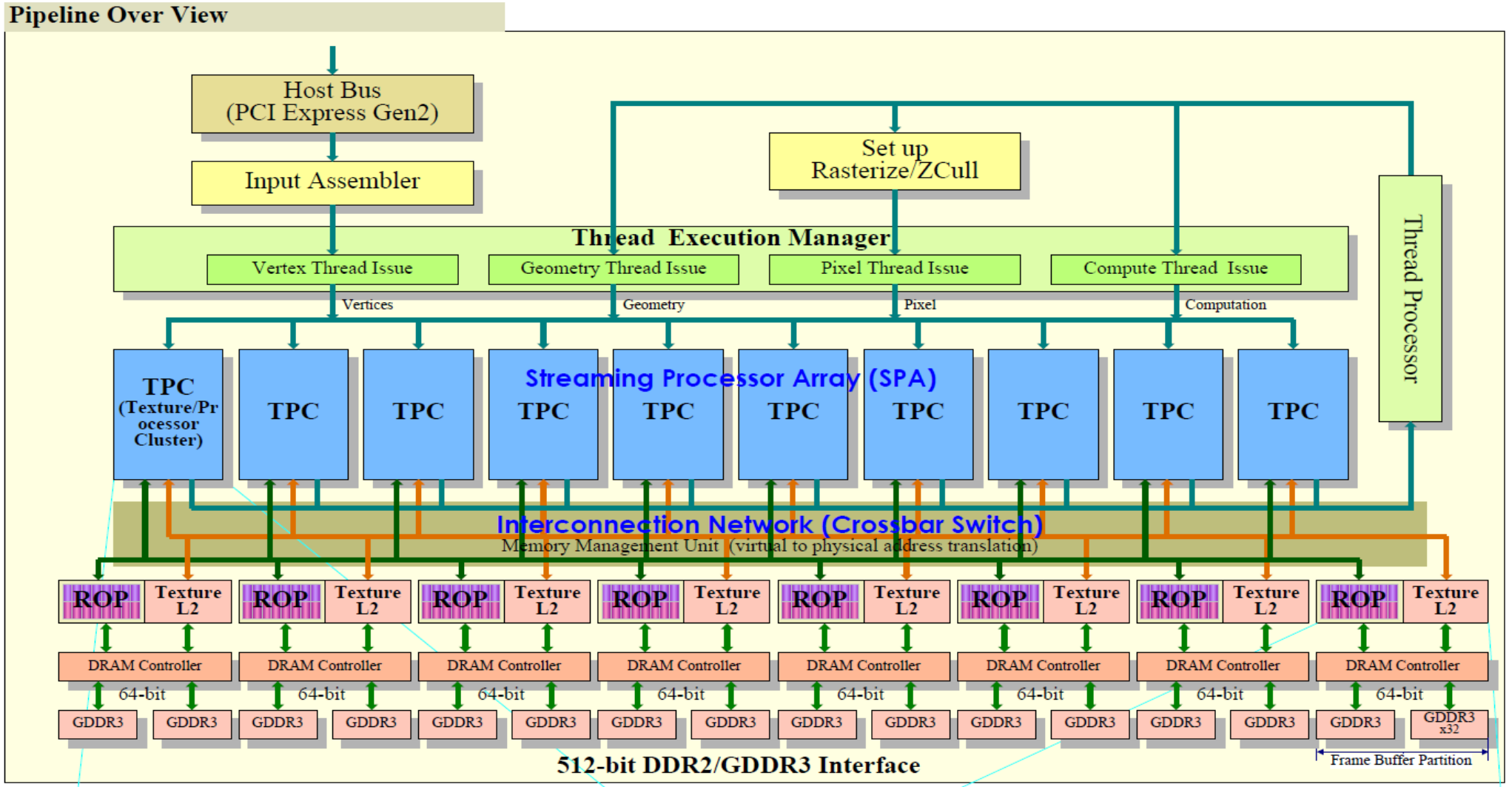


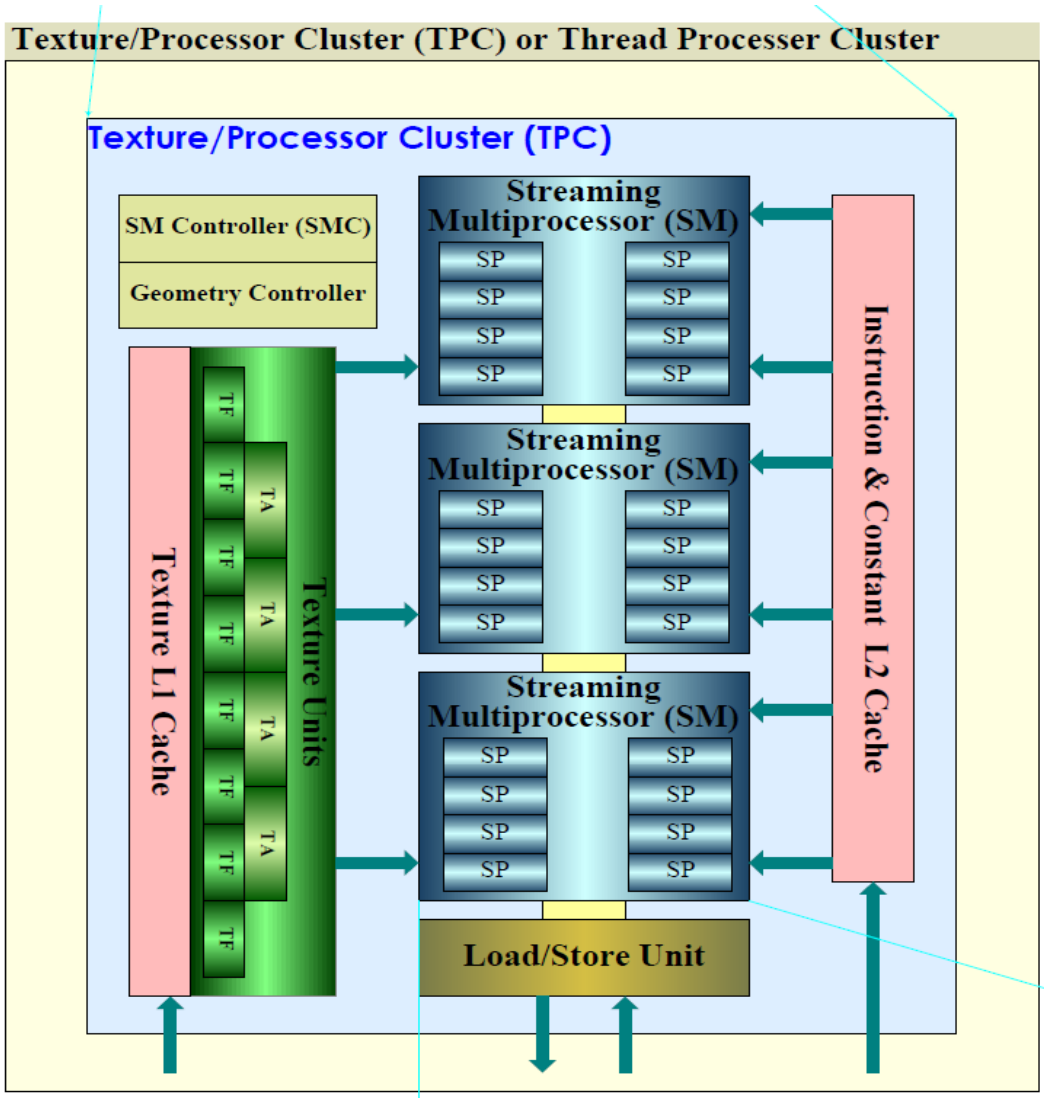


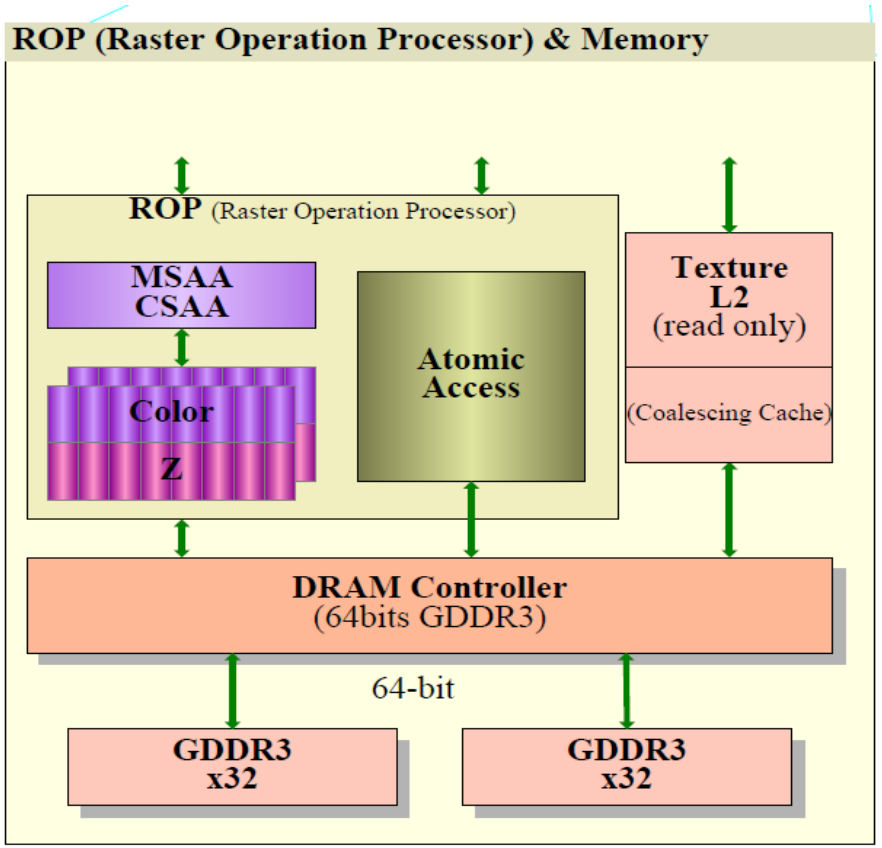
Shader Model 3.0
- standard FP32 precision
- dynamic branching
- longer shader lengths

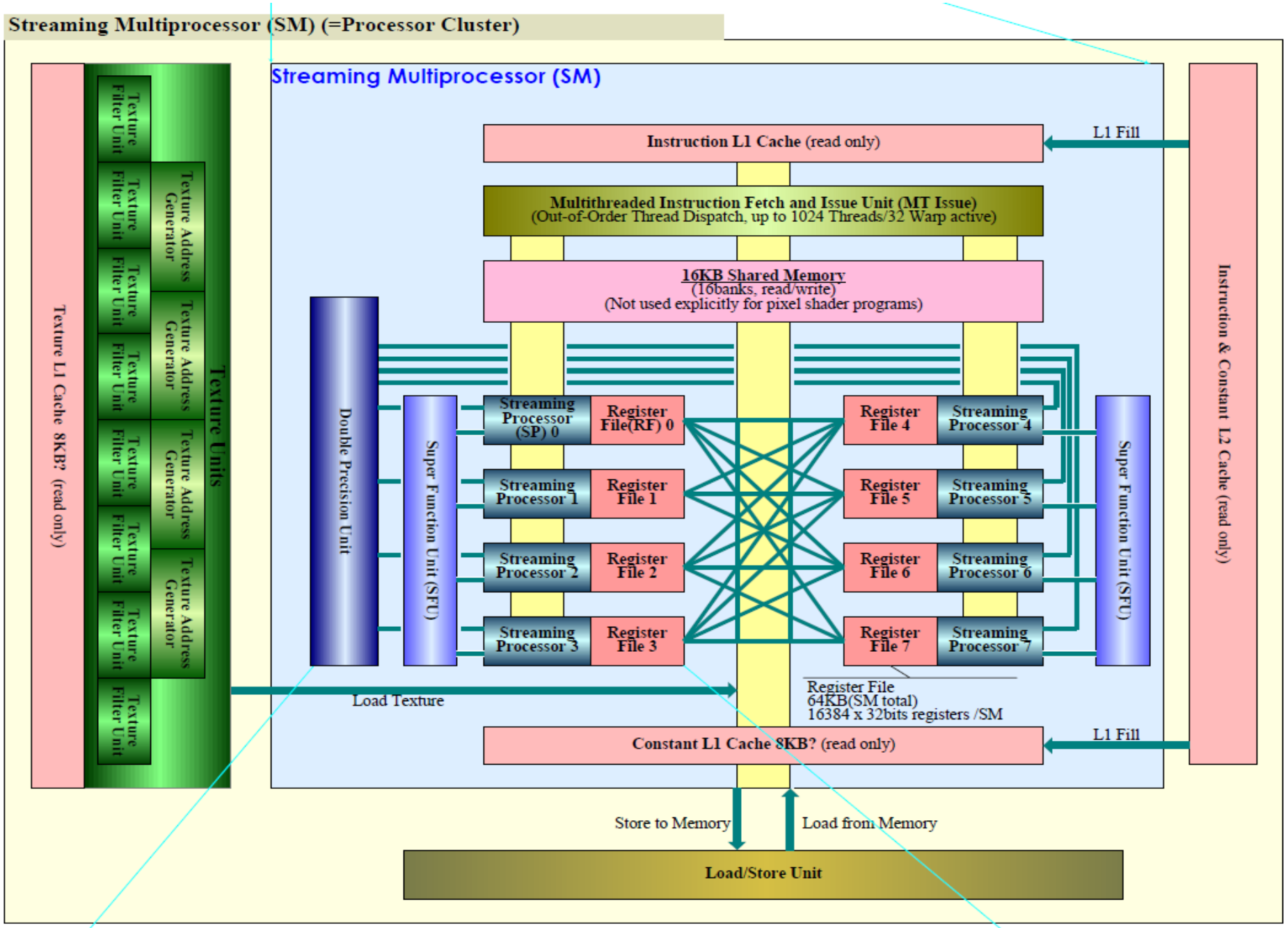
© NVIDIA

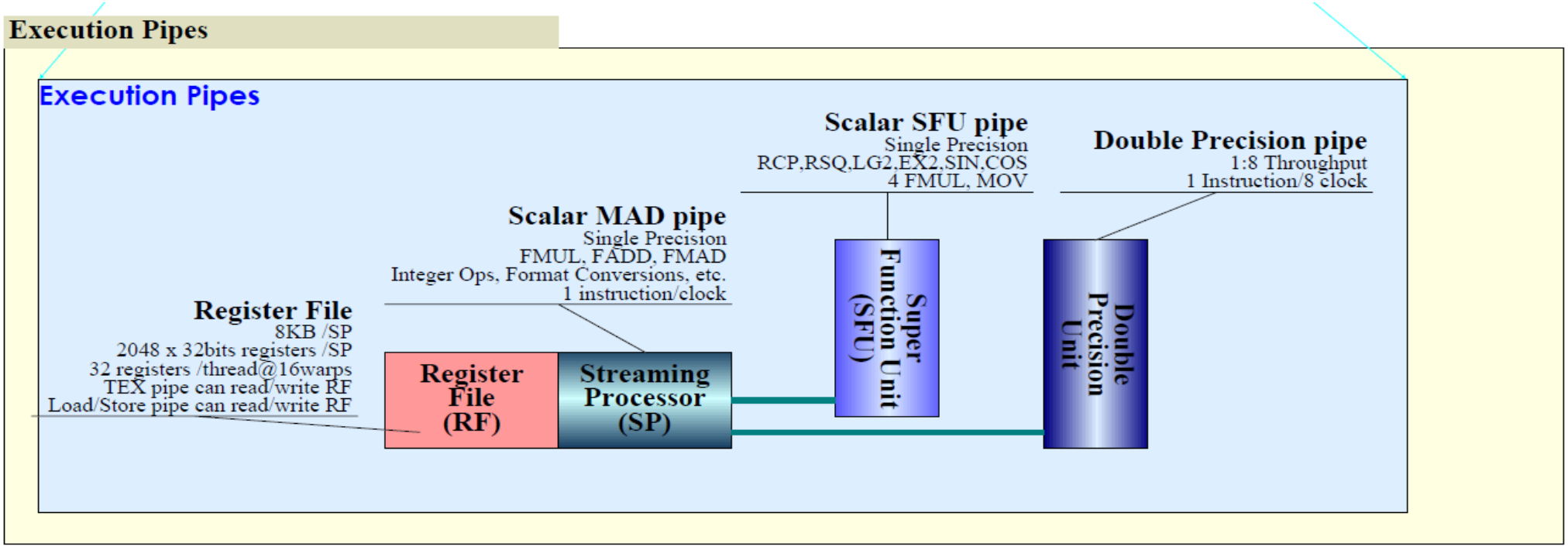
2008년: GeForce 200 Series (GT200) - GTX 280




Texture/Processor Cluster (TPC) or Thread Processer Cluster
Texture/Processor Cluster (TPC)
SM Controller (SMC)
Geometry Controller
Streaming Multiprocessor (SM)
SP
Texture L1 Cache
TF
TA
Texture Units
Instruction & Constant L2 Cache
Load/Store Unit


ROP (Raster Operation Processor) & Memory
ROP (Raster Operation Processor)
MSAA
CSAA
Color
Z
Atomic Access
Texture L2
(read only)
(Coalescing Cache)
DRAM Controller
(64bits GDDR3)
64-bit
GDDR3
x32
GDDR3
x32


Streaming Multiprocessor (SM) (=Processor Cluster)
Streaming Multiprocessor (SM)
Texture L1 Cache 8KB? (read only)
Texture Units
Texture Filter Unit
Texture Address Generator
Instruction L1 Cache (read only)
L1 Fill
Multithreaded Instruction Fetch and Issue Unit (MT Issue)
(Out-of-Order Thread Dispatch, up to 1024 Threads/32 Warp active)
16KB Shared Memory
(16banks, read/write)
(Not used explicitly for pixel shader programs)
Instruction & Constant L2 Cache (read only)
Double Precision Unit
Super Function Unit (SFU)
Streaming Processor (SP) 0
Register File(RF) 0
Streaming Processor 1
Register File 1
Streaming Processor 2
Register File 2
Streaming Processor 3
Register File 3
Register File 4
Streaming Processor 4
Register File 5
Streaming Processor 5
Register File 6
Streaming Processor 6
Register File 7
Streaming Processor 7
Load Texture
Register File
64KB(SM total)
16384 x 32bits registers /SM
Constant L1 Cache 8KB? (read only)
Store to Memory
Load from Memory
Load/Store Unit


Execution Pipes
Execution Pipes
Scalar SFU pipe
Single Precision
RCP,RSQ,LG2,EX2,SIN,COS
4 FMUL, MOV
Double Precision pipe
1:8 Throughput
1 Instruction/8 clock
Scalar MAD pipe
Single Precision
FMUL, FADD, FMAD
Integer Ops, Format Conversions, etc.
1 instruction/clock
Register File
8KB /SP
2048 x 32bits registers /SP
32 registers /thread@16warps
TEX pipe can read/write RF
Load/Store pipe can read/write RF
Register File (RF)
Streaming Processor (SP)
Super Function Unit (SFU)
Double Precision Unit

[CSE4170 기초 컴퓨터 그래픽스]

2017년도 1학기

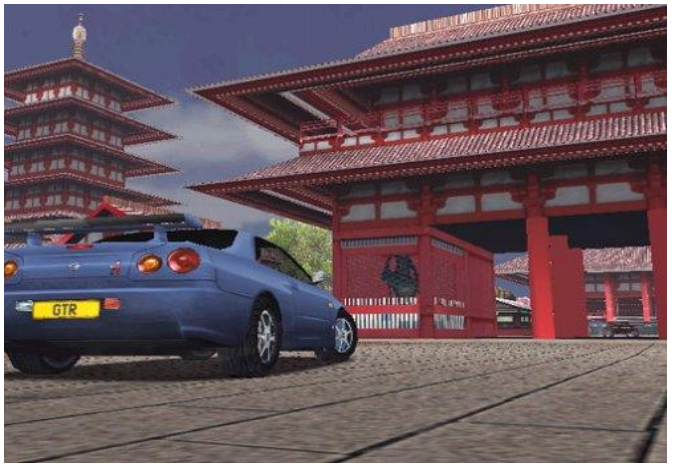
강의자료 III

Texture Mapping

Texture Mapping
and its Applications

Texture Mapping in Real-Time Rendering

- A method for adding detail, surface texture, or color to computer-generated graphic or 3D model (*from wikipedia*)

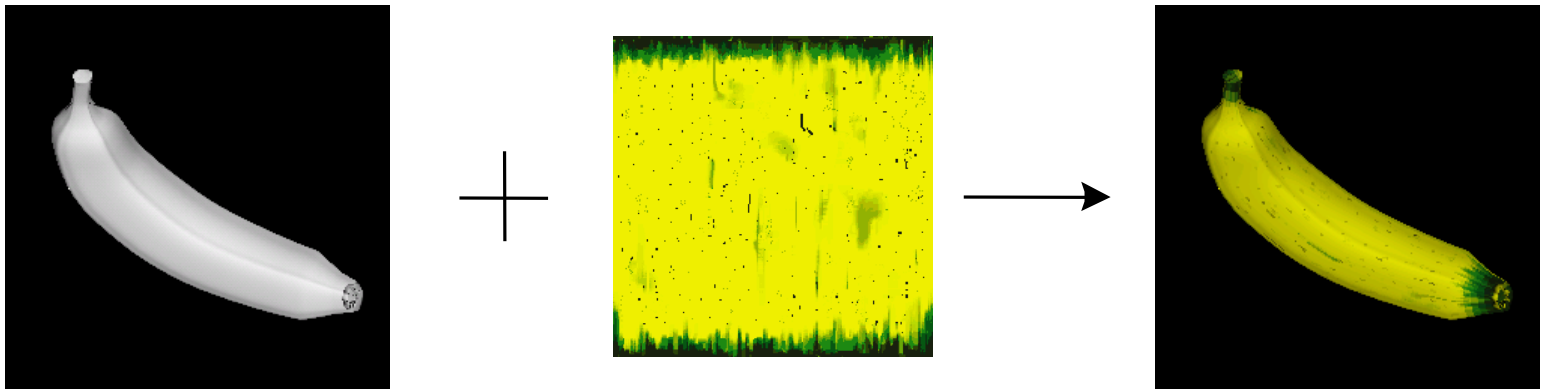


Texture mapping
is
everywhere!

컴퓨터 그래픽스 분야에서의 텍스춰 매핑 기법 예

- 제한된 수의 다각형을 사용해야하는 실시간 렌더링에 있어 비교적 적은 추가 비용으로 이미지의 사실성을 상당히 높일 수 있는 기법임.
- 다양한 형태의 텍스춰 맵 데이터를 사용하여 렌더링 작업의 사실성을 높이고자 하는 기법 → 텍스춰 맵은 어떤 정보를 저장하고 있을까?

① 1D/2D/3D texture mapping



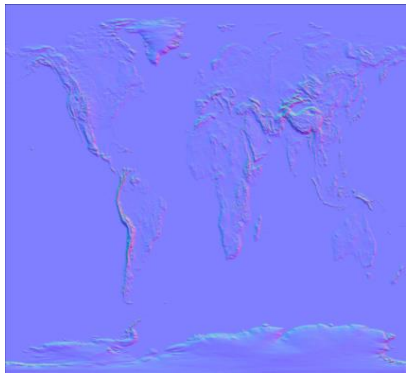
Polygonal model

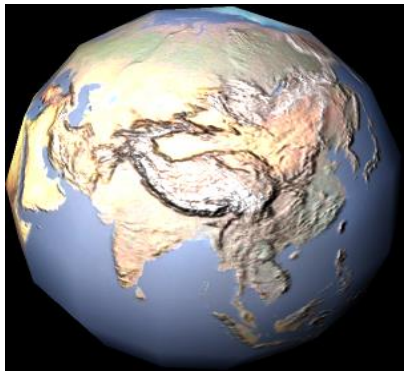
Texture map

Texture mapped image

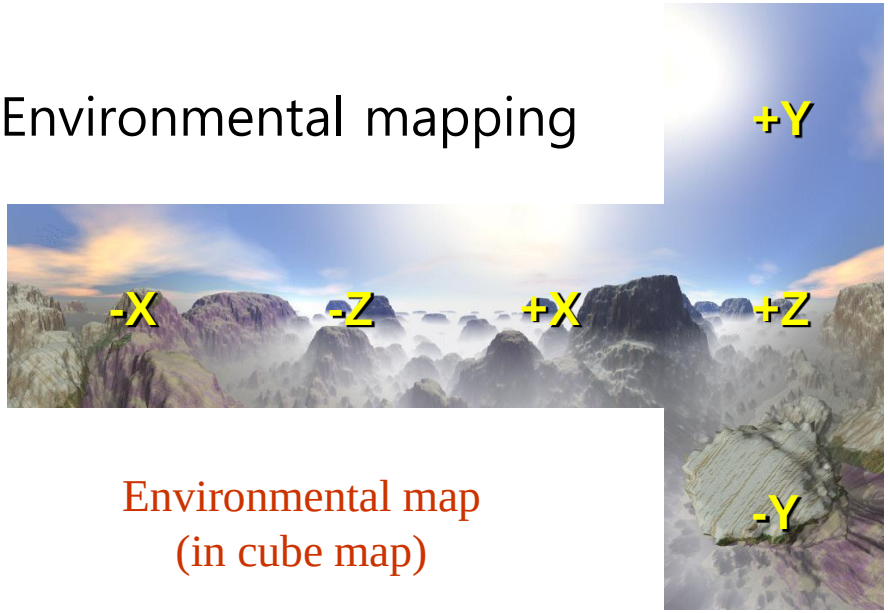
② Bump mapping

Bump map

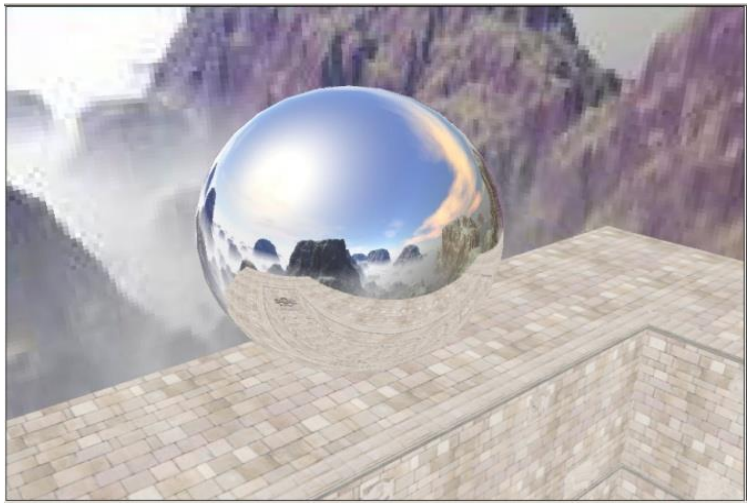




③ Environmental mapping

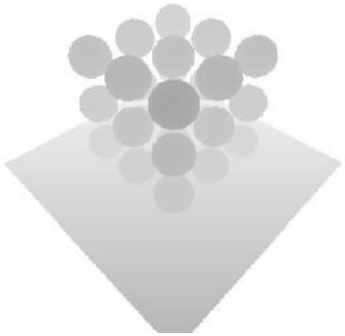


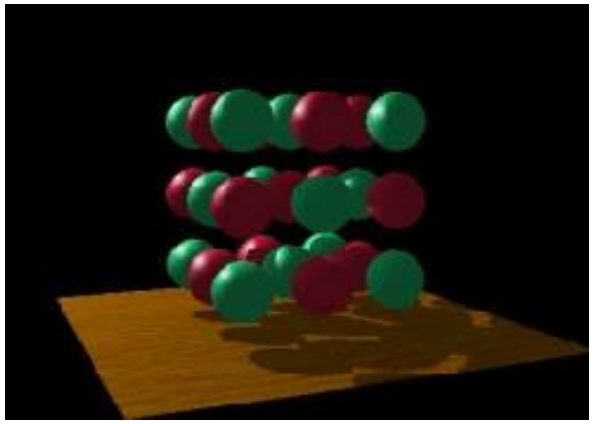
Environmental map
(in cube map)



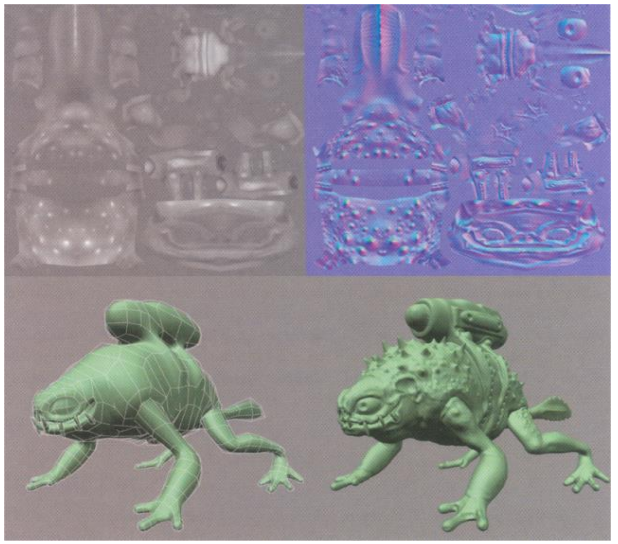
④ Shadow mapping

Shadow map



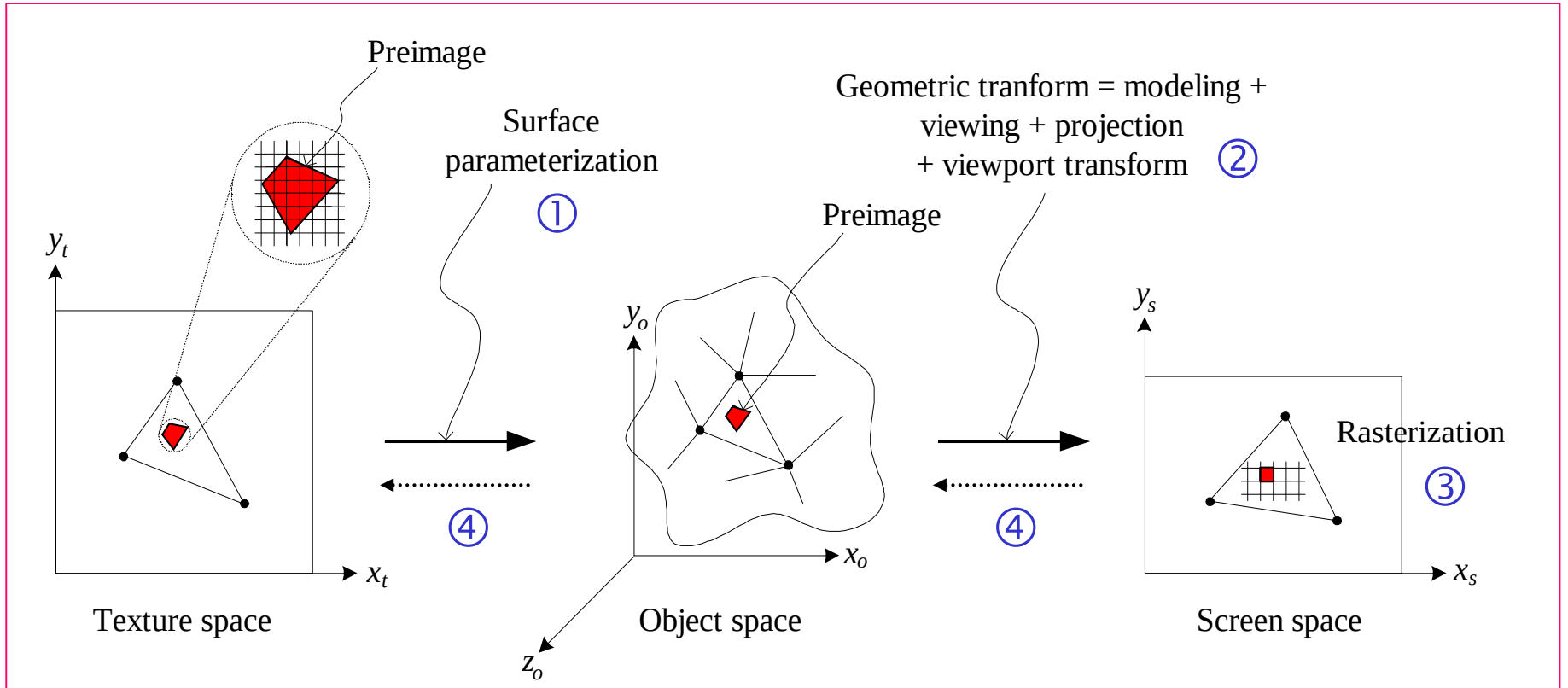


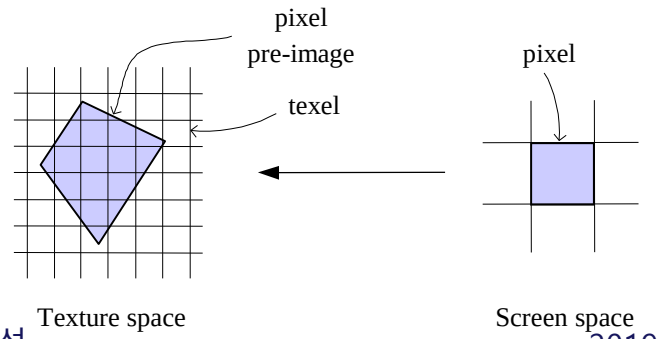
⑤ Displacement mapping, …



Texture Mapping for Polygonal Models

What Computations Must Be Performed?





개념적인 2차원 텍스춰 매핑 과정

① 표면 매개화(surface parameterization) 과정
- 텍스춰 이미지를 물체에 어떻게 입힐 것인가?
- 물체의 각 점에 매핑되는 텍스춰 이미지의 좌표가 필요함:

$$(x_o, y_o, z_o) \leftrightarrow (x_t, y_t)$$

② 기하 변환 과정
- 기하 변환은 물체의 각 점과 투영 화면에서의 위치간의 매핑 관계를 결정:

$$(x_o, y_o, z_o) \leftrightarrow (x_s, y_s)$$

③ 래스터화 과정
- 각 프리미티브가 투영되는 화소들을 찾아 주는 과정.

④ 텍스춰 색깔 계산 과정
- 각 화소에 '적절히' 텍스춰 색깔로 칠해주는 과정.
  - 각 화소를 통하여 보이는 텍스춰 이미지의 색깔을 어떻게 계산할 것인가?
  - 계산된 텍스춰 색깔을 물체의 원래의 색깔과 어떻게 혼합을 할 것인가?

실시간 렌더링을 위한 텍스춰 매핑에 대한 고려 인자

1. 텍스춰 이미지의 내용과 그에 관련된 성질을 어떻게 정의할 것인가? 또한 동시에 여러 개의 텍스춰를 사용할 경우 어떻게 하면 텍스춰 이미지들을 효과적으로 다룰 수 있을까?

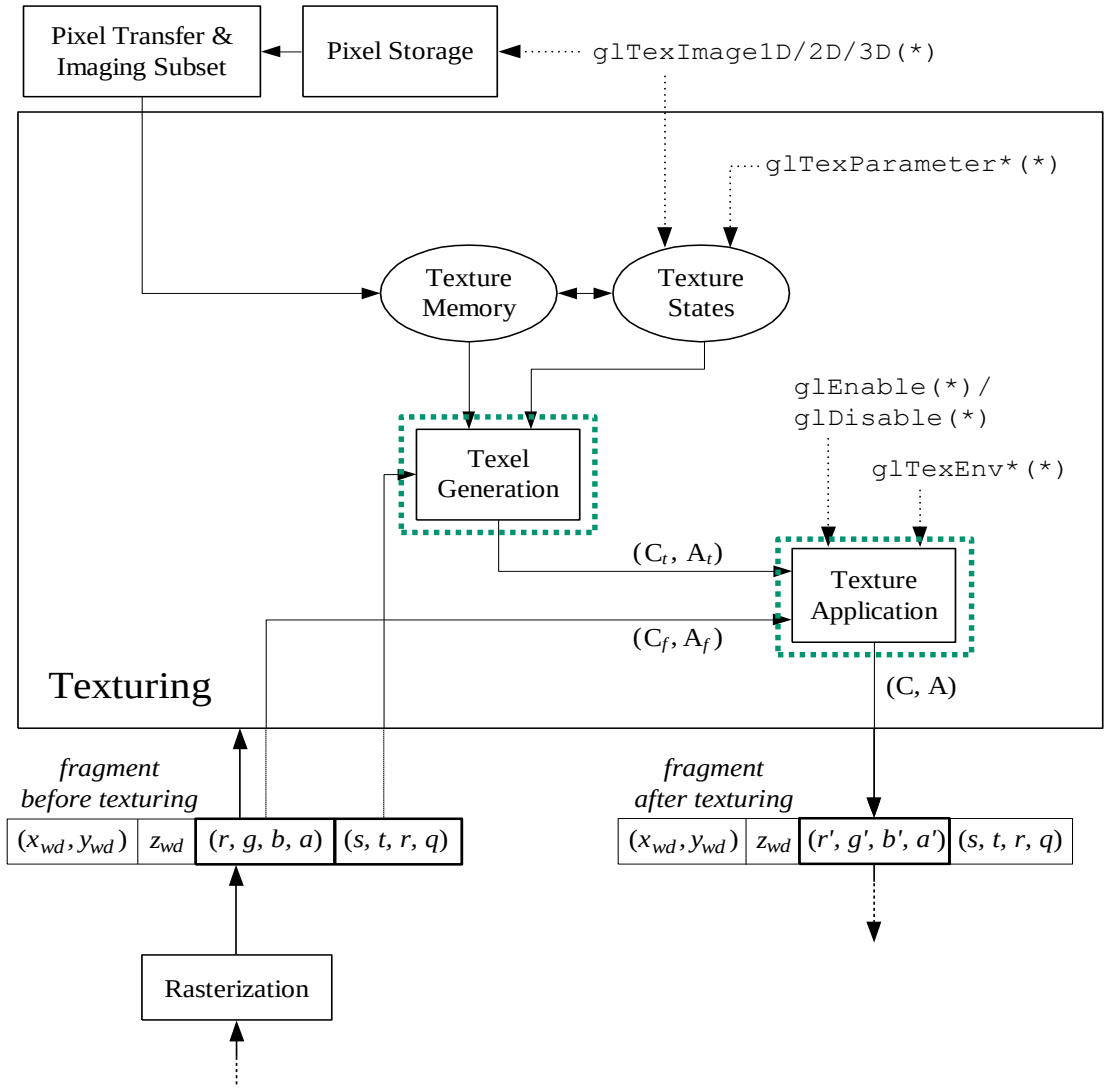
2. 다면체 모델에 대하여 어떠한 방식으로 2차원 텍스춰를 붙여줄 것인가? ← **Surface Parameterization**

3. 다면체 모델이 투영되는 각 화소에 해당하는 텍스춰의 색깔을 어떻게 구할 것인가? ← **Texel Generation**

4. 텍스춰 매핑 과정에 영향을 미치는 여러 성질들을 어떻게 정의할 것인가?

5. 각 화소에 대하여 텍스춰 색깔과 물체의 기반 색깔을 어떻게 혼합할 것인가? ← **Texel Application**

Overview of Texture Mapping in Open GL



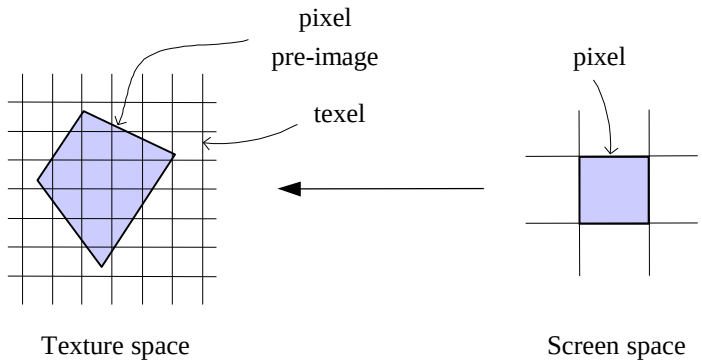
Texel Generation and Texture Application

텍스춰 매핑 구현에 있어 중요한 두 가지 문제

① Texel generation 문제

- 텍스춰 메모리에 올려져 있는 텍스춰 이미지로부터 텍스춰 모듈로 흘러들어오는 픽셀(프래그먼트)에 대한 원상 영역에 해당하는 텍스춰 색깔 (C_t, A_t) 을 어떻게 가져올 것인가?
- 가장 중요한 문제 중의 하나는 **필터링 문제**를 어떻게 실시간적으로 해결을 할 것인가임.

현재 fragment shader에서 컨트롤이 ↑ 불가능. ↓ 가능.

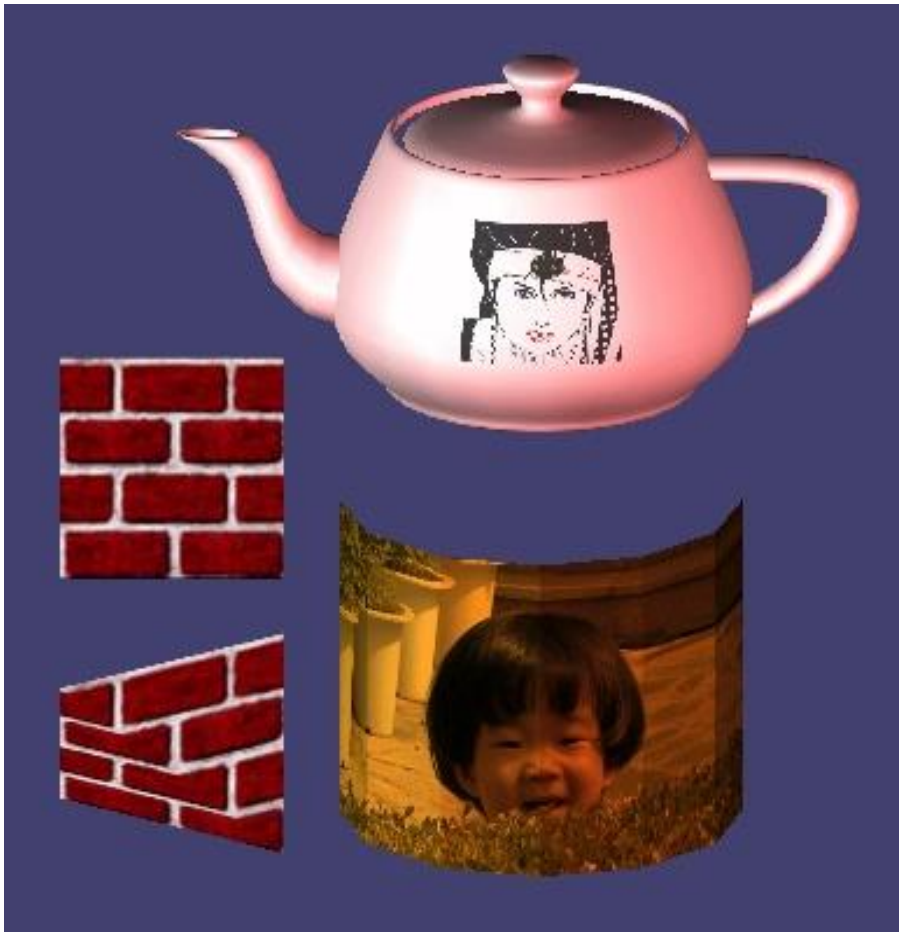


② Texel application 문제

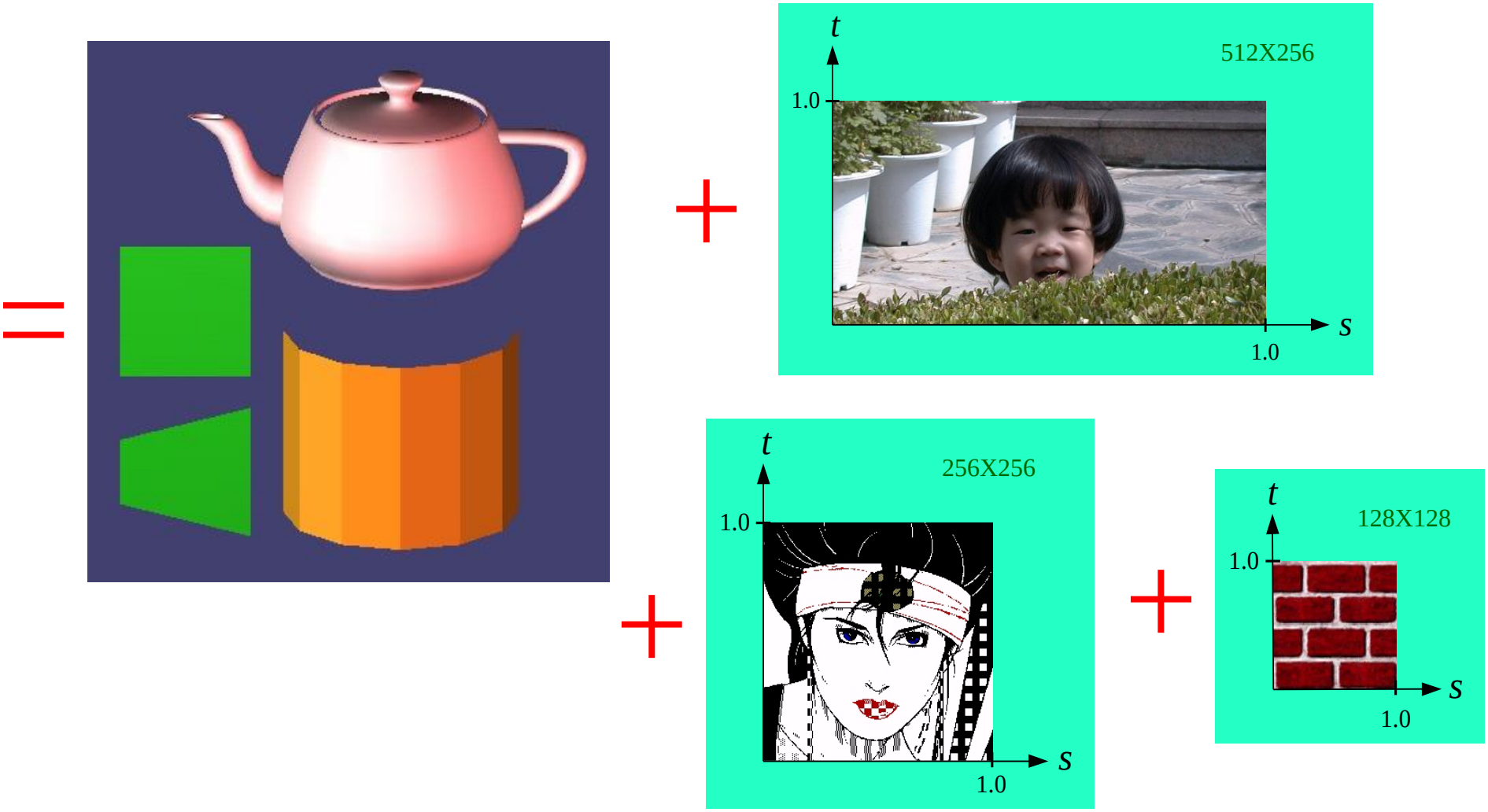
- 텍스춰 이미지로 부터 가져온 색깔 (C_t, A_t) 과 (라이팅 모듈에서 그래픽스 프리미티브들의 각 꼭지점 마다 설정하여 래스터화 과정에서 픽셀에 대하여 보간한) 물체의 기반 색깔 (C_f, A_f) 을 어떻게 혼합을 할 것인가?

OpenGL Programming for Texture Mapping

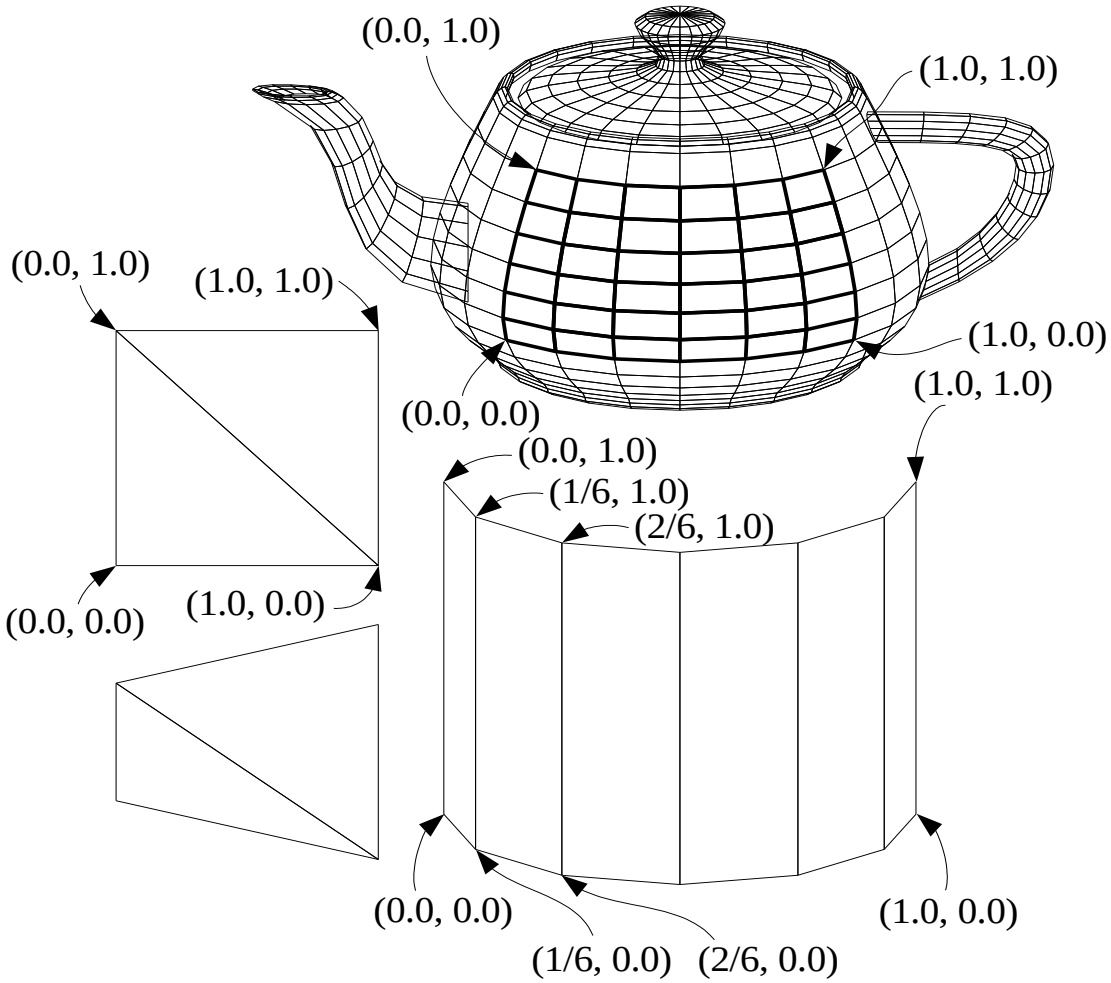
예제를 통한 텍스춰 매핑 프로그래밍



• 해상도에 상관 없이 텍스춰 공간에서 s와 t 좌표는 각각 0과 1사이의 값으로 정규화가 됨.



텍스춰 좌표의 설정



- OpenGL에서는 3차원 공간의 점을 내부적으로 (x, y, z, w)로 표현하듯이, 2차원 외에도 3차원 텍스춰 매핑을 지원을 하기 위하여 텍스춰 좌표를 (s, t, r, q)로 표현함.

- 여기서 q의 역할은 w와 유사함.

- 2차원 텍스춰 매핑을 하기 위하여 s와 t 좌표를 설정하면, 디폴트로 r은 0, 그리고 q는 1로 설정이 됨.

OpenGL을 통한 텍스춰 매핑 구현

① 텍스춰 이미지의 설정과 텍스춰 객체
② 텍스춰 상태와 텍셀 생성 인자의 설정
③ 텍스춰 적용 함수의 설정
④ 텍스춰 좌표의 설정과 텍스춰 좌표의 변환

① 텍스춰 이미지의 설정과 텍스춰 객체

- 2차원 텍스춰 이미지의 설정

```
void glTexImage2D(GLenum target, GLint level, Glint internalFormat,
      GLsizei width, GLsizei height, GLint border, GLenum format,
      GLenum type, const GLvoid *texels); 함수
```

- **target**: 일반적으로 **GL_TEXTURE_2D** 사용
- **level**: level-of detail 숫자, 기본 이미지의 레벨은 0
- **internalFormat**: 텍스춰 데이터가 텍스춰 메모리에 저장되는 내부 형식
  - GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, GL_RGBA, GL_ALPHA, GL_INTENSITY, GL_R3_G3_B2, GL_RGBA16, …
- **width**, **height**: 텍스춰 데이터의 가로-세로 해상도
- **border**: 텍스춰 데이터의 경계의 폭, 0 또는 1
- **format, type, texels:** 주기억장치에 저장되어 있는 텍스춰 데이터에 대한 인자
  - format: 각 텍셀에 저장된 데이터의 내용
    - GL_RGB, GL_RGBA, GL_LUMINANCE, GL_LUMINANCE, …
  - type: 각 텍셀에 대한 데이터의 저장 형식
    - GL_UNSIGNED_BYTE, GL_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, …
  - texels: 데이터가 저장된 메모리에 대한 포인터

- 개념적으로 glTexImage2D(*) 함수는 주기억장치에 있는 텍스춰 데이터를 텍스춰링 모듈에서 사용할 수 있는 형태로 변환하여 텍스춰 메모리에 올려주는 역할을 함.

- 주기억장치에 format 형식으로 저장되어 있는 각 텍셀의 색깔 정보는 일단 RGBA 데이터로 변환이 된 후, internalFormat에 따라 아래와 같은 방식으로 변환이 됨.

Internal format	Used RGBA channels	Resulting internal elements
GL_ALPHA	A	A
GL_LUMINANCE	R	L
GL_LUMINANCE_ALPHA	R, A	L, A
GL_INTENSITY	R	I
GL_RGB	R, G, B	R, G, B
GL_RGBA	R, G, B, A	R, G, B, A

```
typedef struct {
  int ns, nt;
  GLubyte *tmap;
} Texture;
Texture tex_sy, tex_ng, tex_br;

// texture 1
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_sy.ns, tex_sy.nt, 0, GL_RGB,
   GL_UNSIGNED_BYTE, tex_sy.tmap);

// texture 2
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tex_ng.ns, tex_ng.nt, 0, GL_RGBA,
   GL_UNSIGNED_BYTE, tex_ng.tmap);

// texture 3
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_br.ns, tex_br.nt, 0, GL_RGB,
   GL_UNSIGNED_BYTE, tex_br.tmap);
```

- **텍스춰 객체(texture objects)**
  - 텍스춰 데이터 자체에 대한 정보와 그 텍스춰 이미지로부터 텍셀의 값을 구하는 방식에 대한 모든 정보를 저장해주는 자료 구조
  - 여러 장의 텍스춰 이미지를 효율적으로 사용할 수 있도록 해줌.
  - 관련 함수
    - `void glGenTextures(GLsizei n, GLuint *textureNames);`
    - `void glBindTexture(GLenum target, GLuint texture);`
    - `void glDeleteTextures(GLsizei n, const GLuint *textures);`
    - …

```
GLuint tex_name[3];

void set_textures(void){
  glGenTextures(3, tex_name);

  glBindTexture(GL_TEXTURE_2D, tex_name[0]);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_sy.ns, tex_sy.nt, 0, GL_RGB,
               GL_UNSIGNED_BYTE, tex_sy.tmap);

  glBindTexture(GL_TEXTURE_2D, tex_name[1]);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tex_ng.ns, tex_ng.nt, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, tex_ng.tmap);

  glBindTexture(GL_TEXTURE_2D, tex_name[2]);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_br.ns, tex_br.nt, 0, GL_RGB,
               GL_UNSIGNED_BYTE, tex_br.tmap);
}
```

② 텍스춰 상태와 텍셀 생성 인자의 설정

- 텍셀 생성(texel generation) 과정 시 어떤 텍스춰로부터 텍스춰 색깔을 가져올 지, 또한 어떠한 방식으로 그러한 값을 구할 지 등의 계산 과정에 영향을 미치는 인자들의 설정 과정
  - (C_t, A_t)를 어떻게 구할 것인가?

- 프로그래머가 설정한 정보를 텍스춰 객체의 일부인 텍스춰 상태(texture states)라는 자료 구조에 저장

1) `glTexImage2D(*)` 함수로 설정이 되는 정보
   - 텍스춰 이미지 자체에 대한 정보
   - 텍스춰, 해상도, 경계 두께, 내부 형식, 각 요소 필드에 할당된 비트 수 등

2) `glTexParameteri(*)` 함수로 설정이 되는 정보
   - 현재 프래그먼트의 텍스춰 좌표가 주어졌을 때, 텍스춰 이미지를 액세스하여 텍스춰 색깔을 계산하는 방식에 영향을 미치는 인자들에 정보
   - 다음 슬라이드 참조

void glTexParameteri(GLenum target, GLenum pname, GLint param); 함수

	pname	param
mag. filter	GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
min. filter	GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR GL_LINEAR_MIPMAP_LINEAR
wrap mode (s)	GL_TEXTURE_WRAP_S	GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_EDGE
wrap mode (t)	GL_TEXTURE_WRAP_T	same as above
wrap mode (r)	GL_TEXTURE_WRAP_R	same as above
texture border color	GL_TEXTURE_BORDER_COLOR	RGBA color: (0, 0, 0, 0)
min LOD	GL_TEXTURE_MIN_LOD	floating-point number: -1000.0
max LOD	GL_TEXTURE_MAX_LOD	floating-point number: 1000.0
base mipmap level	GL_TEXTURE_BASE_LEVEL	interger number: 0
maximum mipmap level	GL_TEXTURE_MAX_LEVEL	integer number: 1000
texture priority	GL_TEXTURE_PRIORITY	floating-point number between 0 and 1

```
GLuint tex_name[3];

void set_textures(void){
  glGenTextures(3, tex_name);

  glBindTexture(GL_TEXTURE_2D, tex_name[0]);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_sy.ns, tex_sy.nt, 0, GL_RGB,
               GL_UNSIGNED_BYTE, tex_sy.tmap);

  glBindTexture(GL_TEXTURE_2D, tex_name[1]);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tex_ng.ns, tex_ng.nt, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, tex_ng.tmap);

  glBindTexture(GL_TEXTURE_2D, tex_name[2]);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, tex_br.ns, tex_br.nt, 0, GL_RGB,
               GL_UNSIGNED_BYTE, tex_br.tmap);
}
```

③ 텍스춰 적용 함수의 설정

- 텍셀 적용(texel application) 과정 시 텍스춰 메모리에서 가져온 텍셀의 색깔과 현재 처리하려는 프래그먼트의 색깔과 어떻게 혼합을 할 지에 대한 설정 과정
  - 텍셀 생성과정에서 구한 (C_t, A_t) 와 원래의 프래그먼트의 색깔을 (C_f, A_f) 를 어떻게 섞어 (C, A) 를 구할 것인가?
  - OpenGL의 경우 네 가지의 적용 방식 제공
    - `GL_DECAL`
    - `GL_REPLACE`
    - `GL_MODULATE`
    - `GL_BLEND`

int. format	GL_REPLACE	GL_MODULATE
GL_ALPHA	$C = C_f,\ A = A_t$	$C = C_f,\ A = A_f A_t$
GL_LUMINANCE	$C = L_t,\ A = A_f$	$C = C_f L_t,\ A = A_f$
GL_LUMINANCE_ALPHA	$C = L_t,\ A = A_t$	$C = C_f L_t,\ A = A_f A_t$
GL_INTENSITY	$C = I_t,\ A = I_t$	$C = C_f I_t,\ A = A_f I_t$
GL_RGB	$C = C_t,\ A = A_f$	$C = C_f C_t,\ A = A_f$
GL_RGBA	$C = C_t,\ A = A_t$	$C = C_f C_t,\ A = A_f A_t$

int. format	GL_DECAL	GL_BLEND
GL_ALPHA	undefined	$C = C_f,\ A = A_f A_t$
GL_LUMINANCE	undefined	$C = C_f(1 - L_t) + C_c L_t,\ A = A_f$
GL_LUMINANCE_ALPHA	undefined	$C = C_f(1 - L_t) + C_c L_t,\ A = A_f A_t$
GL_INTENSITY	undefined	$C = C_f(1 - I_t) + C_c I_t,\ A = A_f(1 - I_t) + A_c I_t$
GL_RGB	$C = C_t,\ A = A_f$	$C = C_f(1 - C_t) + C_c C_t,\ A = A_f$
GL_RGBA	$C = C_f(1 - A_t) + C_t A_t,\ A = A_f$	$C = C_f(1 - C_t) + C_c C_t,\ A = A_f A_t$

void glTexEnvf(GLenum target, GLenum pname, GLfloat param); 함수

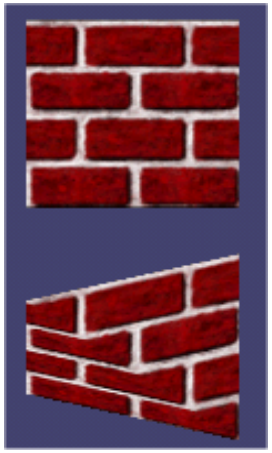
```
// teapot
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

// hcylinder
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

// quadrilaterals 3
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

GL_DECAL

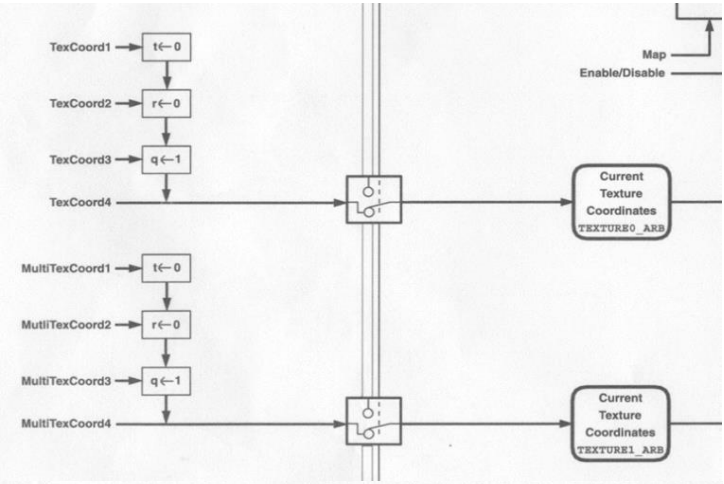
GL_MODULATE

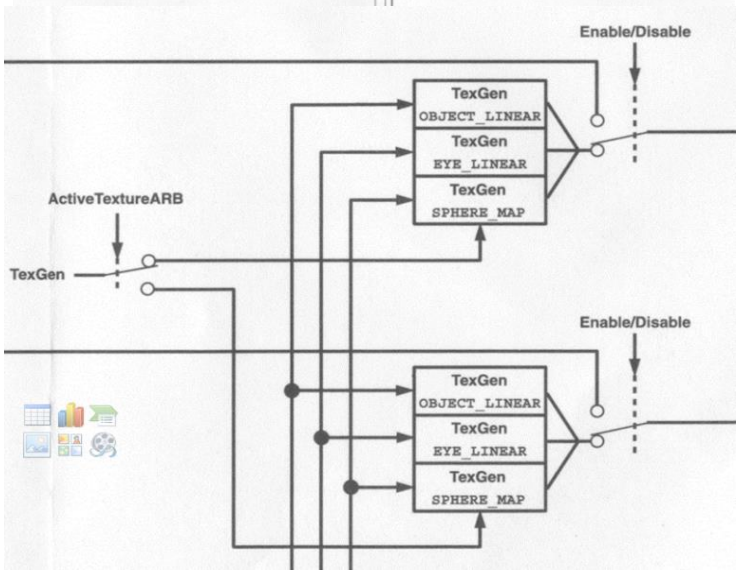


GL_REPLACE

④ 텍스춰 좌표의 설정과 텍스춰 좌표의 변환

- ①, ②,③ 과정을 통하여 텍스춰 매핑과 관련한 설정을 마친 뒤, 다면체 모델을 그려주는 과정에서 각 꼭지점에 대한 텍스춰 좌표를 설정해주어야 함.
- OpenGL의 경우 크게 두 가지 방법으로 좌표를 설정할 수 있음.





  - 모델링 과정에서 미리 꼭지점마다 텍스춰 좌표를 구해 놓은 후, 렌더링 과정에서 각 꼭지점에 대한 정보를 기술할 때 다음 등의 함수를 사용하여 설정하는 방식 → **정적인 방법**

```
void glTexCoord2f(GLfloat s, GLfloat t); 함수
void glTexCoord2fv(const GLfloat *v); 함수
```

  - 렌더링 계산 시 프로그래머가 설정한 방식으로 각 꼭지점마다 동적으로 텍스춰 좌표를 자동 생성하는 방식 → **동적인 방법**

이 방식은 이제 vertex shader에서 처리.

```
void draw_object(Object *obj) {
  int i, j;
  Polygon *ptr;

  i = 0; ptr = obj->polygon;
  while(i++ < obj->npoly)  {
     glBegin(GL_POLYGON);
       for (j = 0; j < ptr->nvert; j++) {
          glTexCoord2fv(ptr->tcoord[j]);
          glNormal3fv(ptr->norm[j]);
          glVertex3fv(ptr->vert[j]);
       }
    glEnd();
    ptr++;
  }
}

void draw_teapot(void) {
   glShadeModel(GL_SMOOTH);
   glBindTexture(GL_TEXTURE_2D, tex_name[1]);
   glTexEnvf(GL_TEXTURE_ENV,
    GL_TEXTURE_ENV_MODE, GL_DECAL);
```

```
 glMaterialfv(GL_FRONT, GL_AMBIENT,
    mat_ambient1);
          …
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
    glTranslatef(0.8, 0.7, 0.0);
    glRotatef(180.0, 0.0, 1.0, 0.0);
    glRotatef(-90.0, 1.0, 0.0, 0.0);
    glScalef(0.95, 0.95, 0.95);

    glMatrixMode(GL_TEXTURE);
    glPushMatrix();
      glTranslatef(-0.5, 0.0, 0.0);
      glScalef(2.0, 1.0, 1.0);
      draw_object(&teapot);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
  glPopMatrix();
}
```

```
void draw_hcylinder(void) {
  glShadeModel(GL_FLAT);
  glMaterialfv(GL_FRONT, GL_AMBIENT,
    mat_ambient0);
      …
  glBindTexture(GL_TEXTURE_2D, tex_name[0]);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
    GL_MODULATE);

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
    glTranslatef(0.8, -1.7, 0.0);
    glScalef(1.8, 1.8, 1.8);
    draw_object(&hcyl);
  glPopMatrix();
}

void draw_quadrilaterals(void) {
   glBindTexture(GL_TEXTURE_2D, tex_name[2]);
   glTexEnvf(GL_TEXTURE_ENV,
    GL_TEXTURE_ENV_MODE, GL_REPLACE);
   glMaterialfv(GL_FRONT, GL_AMBIENT,
    mat_ambient2);
    …
   glMatrixMode(GL_MODELVIEW);
   glPushMatrix();
     glTranslatef(-2.5, 0.0, 0.0);
```

```
   glBegin(GL_TRIANGLES);
       glNormal3f(0.0, 0.0, 1.0);
       glTexCoord2f(1.0, 0.0);
       glVertex3f(1.0, -1.0, 0.0);
       glTexCoord2f(1.0, 1.0);
       glVertex3f(1.0, 1.0, 0.0);
       glTexCoord2f(0.0, 1.0);
       glVertex3f(-1.0, 1.0, 0.0);

       glTexCoord2f(1.0, 0.0);
       glVertex3f(1.0, -1.0, 0.0);
       glTexCoord2f(0.0, 1.0);
       glVertex3f(-1.0, 1.0, 0.0);
       glTexCoord2f(0.0, 0.0);
       glVertex3f(-1.0, -1.0, 0.0);
     glEnd();
  glPopMatrix();

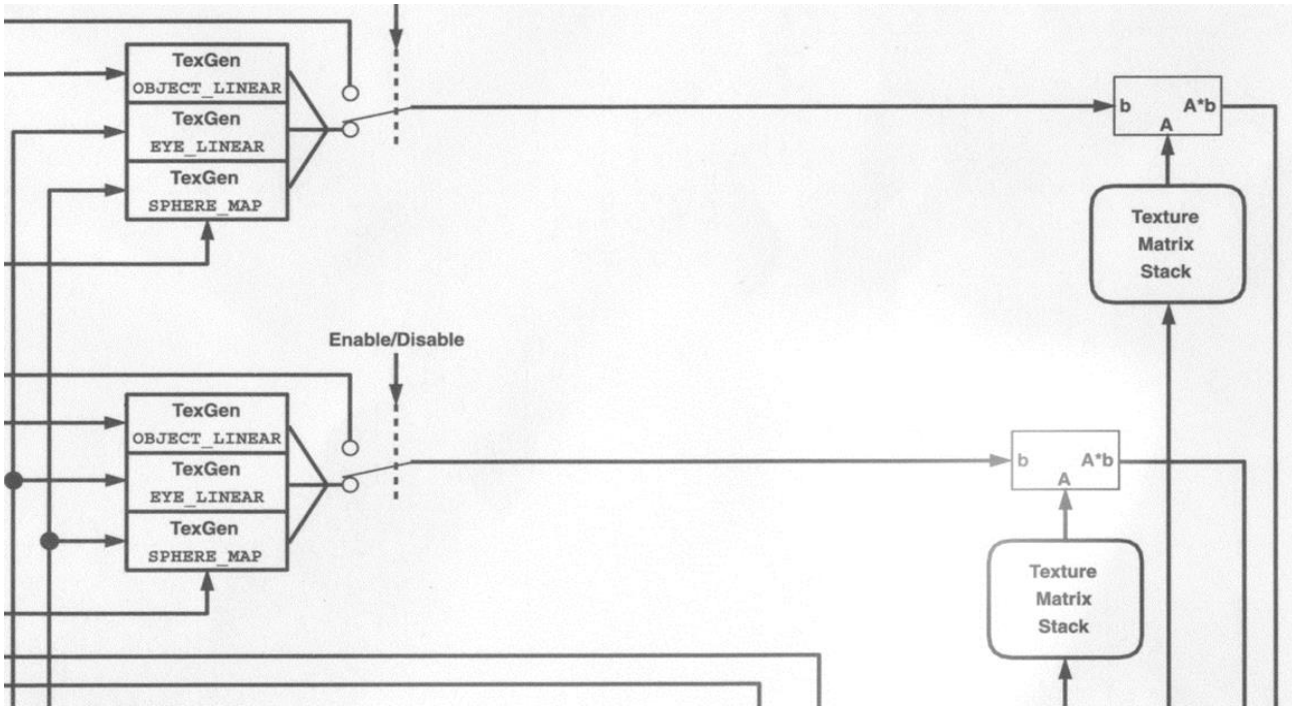
  glPushMatrix();
    glTranslatef(-2.5, -2.5, 0.0);
    glBegin(GL_TRIANGLES);
    glNormal3f(0.0, 0.0, 1.0);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(1.0, -1.0, 0.0);
       …
    glEnd();
  glPopMatrix();
}
```

```
void display(void) {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  draw_hcylinder();
  draw_teapot();
  draw_quadrilaterals();
  glFlush();
}
```

Texture Matrix Stack

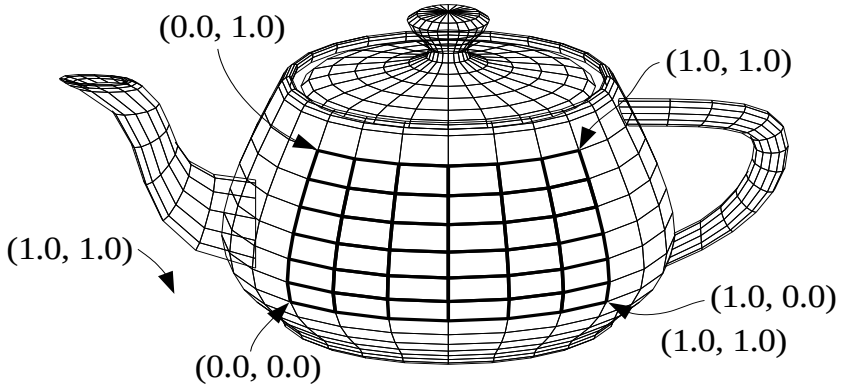
이 방식은 이제 vertex shader에서 처리.

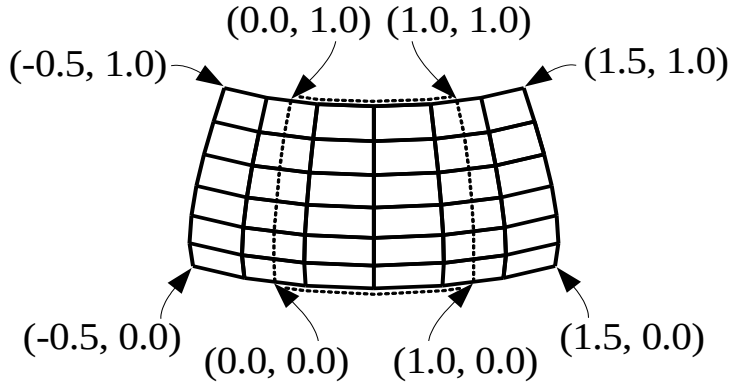
- 꼭지점에 대한 텍스춰 좌표가 결정되면 (위의 두 가지 중 어떤 방법을 사용하건), 그 좌표가 꼭지점에 붙여지기 전에 텍스춰 행렬 스택(Texture Matrix Stack)이라 부르는 스택의 탑에 있는 4행 4열 변환 행렬에 곱해져 변환이 된 후 꼭지점에 붙여짐.
  - 초기 상태로 이 행렬 스택에는 단위 행렬이 올려져 있음.
  - 기하 변환 과정에서 사용한 OpenGL 함수를 그대로 사용할 수 있음.
  - 이 기능은 다양한 방식으로 텍스춰 이미지에 대한 조작을 가능케 해줌.



```
glMatrixMode(GL_TEXTURE);
glPushMatrix();
  glTranslatef(-0.5, 0.0, 0.0);
  glScalef(2.0, 1.0, 1.0);
  draw_object(&teapot);
glPopMatrix();
```

$$M_T = T(-0.5, 0, 0) \cdot S(2, 1, 1)$$





GL_CLAMP

GL_REPEAT

프로그래머 입장에서 본 텍스춰 매핑 과정

① 사용하려는 텍스춰 이미지를 텍스춰 메모리에 올려 놓고(예를 들어 `glTexImage2D(*)` 함수 사용),

② 텍스춰 좌표 값이 주어졌을 때 텍스춰 이미지로부터 그에 대응되는 텍스춰 색깔을 계산하는 방식을 결정하는 텍스춰 상태 값을 적절히 설정을 하고 (`glTexParameter*(*)` 함수 사용),

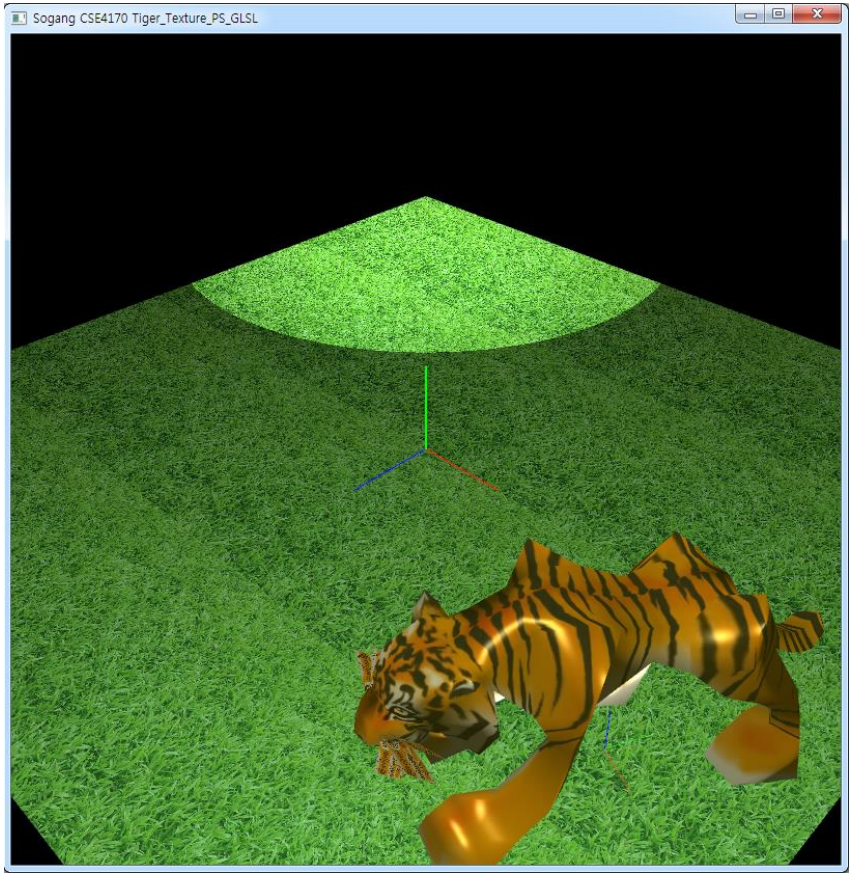
③ 텍스춰 색깔과 물체의 기반 색깔을 어떻게 합성을 할 것인가를 결정한 후 (`glTexEnv*(*)` 함수 사용),

④ 기하 물체를 그릴 때 각 꼭지점에 대하여 텍스춰 좌표를 적절히 설정을 해주면 됨 (`glTexCoord2*(*)` 함수를 사용하거나 자동 생성되도록 설정).

☺ `glEnable(GL_TEXTURE_2D);` 문장을 통하여 명시적으로 텍스춰 매핑 기능을 활성화할 것.

GLSL 4.3 프로그램 작성 예 6

-- OpenGL 3D Phong Shading with Texture



Application

```
...
// floor object
GLuint rectangle_VBO, rectangle_VAO;
GLfloat rectangle_vertices[6][8] = {  // vertices enumerated counterclockwise
    { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f },
    { 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 4.0f, 0.0f },
    { 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 4.0f, 4.0f },
    { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f },
    { 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 4.0f, 4.0f },
    { 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 4.0f }
};
Material_Parameters material_floor;

void prepare_floor(void) { // Draw coordinate axes.
    // Initialize vertex buffer object.
    glGenBuffers(1, &rectangle_VBO);

    glBindBuffer(GL_ARRAY_BUFFER, rectangle_VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(rectangle_vertices), &rectangle_vertices[0][0], GL_STATIC_DRAW);

    // Initialize vertex array object.
    glGenVertexArrays(1, &rectangle_VAO);
    glBindVertexArray(rectangle_VAO);
```

Application

```
GLfloat rectangle_vertices[6][8] = {
    { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f },
    { 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 4.0f, 0.0f },
    { 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 4.0f, 4.0f },
    { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f },
    { 1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 4.0f, 4.0f },
    { 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 4.0f }
};
```

```
    glBindBuffer(GL_ARRAY_BUFFER, rectangle_VBO);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), BUFFER_OFFSET(0));
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), BUFFER_OFFSET(3 * sizeof(float)));
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), BUFFER_OFFSET(6 * sizeof(float)));
    glEnableVertexAttribArray(2);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
    ...
    glActiveTexture(GL_TEXTURE0 + TEXTURE_ID_FLOOR); // select the active texture unit
    glBindTexture(GL_TEXTURE_2D, texture_names[TEXTURE_ID_FLOOR]); // bind a texture object to the active texture unit

    My_glTexImage2D_from_file("Data/grass_tex.jpg");

    // set the default filtering modes
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // set up wrapping modes
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    ...
}
```

layout (location = 0) in vec3 a_position

layout (location = 1) in vec3 a_normal

layout (location = 2) in vec3 a_tex_coord

```
void My_glTexImage2D_from_file(char *filename) {
    ...
    // allocate storage for the texture data
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_BGRA, GL_UNSIGNED_BYTE, data);
    ...
}
```

Shaders

<Vertex Shader>

```
uniform mat4 u_ModelViewProjectionMatrix;
Uniform mat4 u_ModelViewMatrix;
Uniform mat3 u_ModelViewMatrixInvTrans;

Layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_tex_coord;
out vec3 v_position_EC;
out vec3 v_normal_EC;
out vec2 v_tex_coord;

void main(void) {
        v_position_EC = vec3(u_ModelViewMatrix*vec4(a_position, 1.0f));
        v_normal_EC = normalize(u_ModelViewMatrixInvTrans*a_normal);
        v_tex_coord = a_tex_coord;

        gl_Position = u_ModelViewProjectionMatrix*vec4(a_position, 1.0f);
}
```

$$f = \frac{d_{\max} - |z|}{d_{\max} - d_{\min}} = \begin{cases} 0 & \text{, 100\% fog} \\ 1 & \text{, no fog} \end{cases}$$

<Fragment Shader>

```
...
uniform sampler2D u_base_texture;

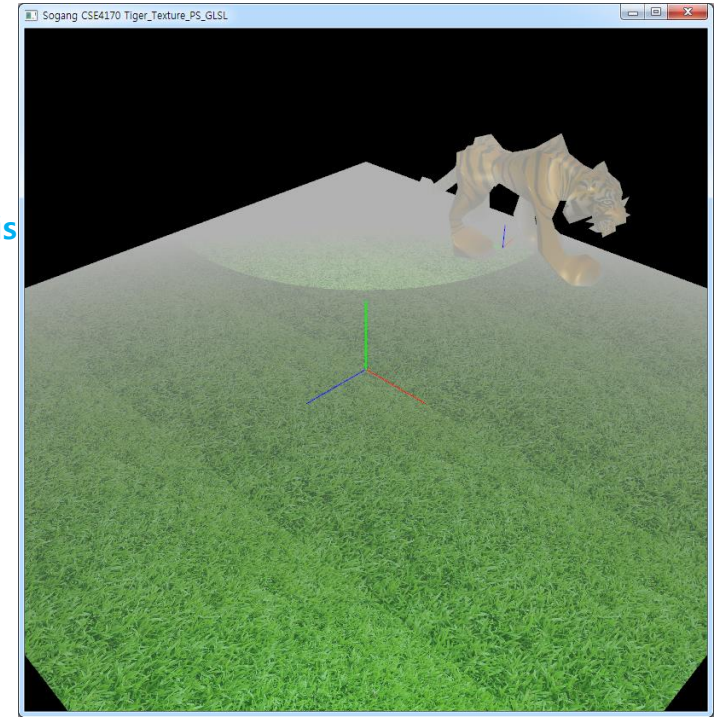
uniform bool u_flag_texture_mapping = true;
uniform bool u_flag_fog = false;
...
in vec2 v_tex_coord;
layout (location = 0) out vec4 final_color;
...
#define FOG_COLOR vec4(0.7f, 0.7f, 0.7f, 1.0f)
#define FOG_NEAR_DISTANCE 350.0f
#define FOG_FAR_DISTANCE 700.0f

void main(void) {
    vec4 base_color, shaded_color;
    float fog_factor;

    if (u_flag_texture_mapping)
        base_color = texture(u_base_texture, v_tex_coord);
    else
        base_color = u_material.diffuse_color;
        shaded_color = lighting_equation_textured(v_position_EC,
                        normalize(v_normal_EC), base_color);
    if (u_flag_fog) {
        fog_factor=(FOG_FAR_DISTANCE-length(v_position_EC.xyz))
                    /(FOG_FAR_DISTANCE- FOG_NEAR_DISTANCE);
        fog_factor = clamp(fog_factor, 0.0f, 1.0f);
        final_color = mix(FOG_COLOR, shaded_color, fog_factor);
    }
    else
        final_color = shaded_color;
}
```

Distance from the eye where the fog is minimal

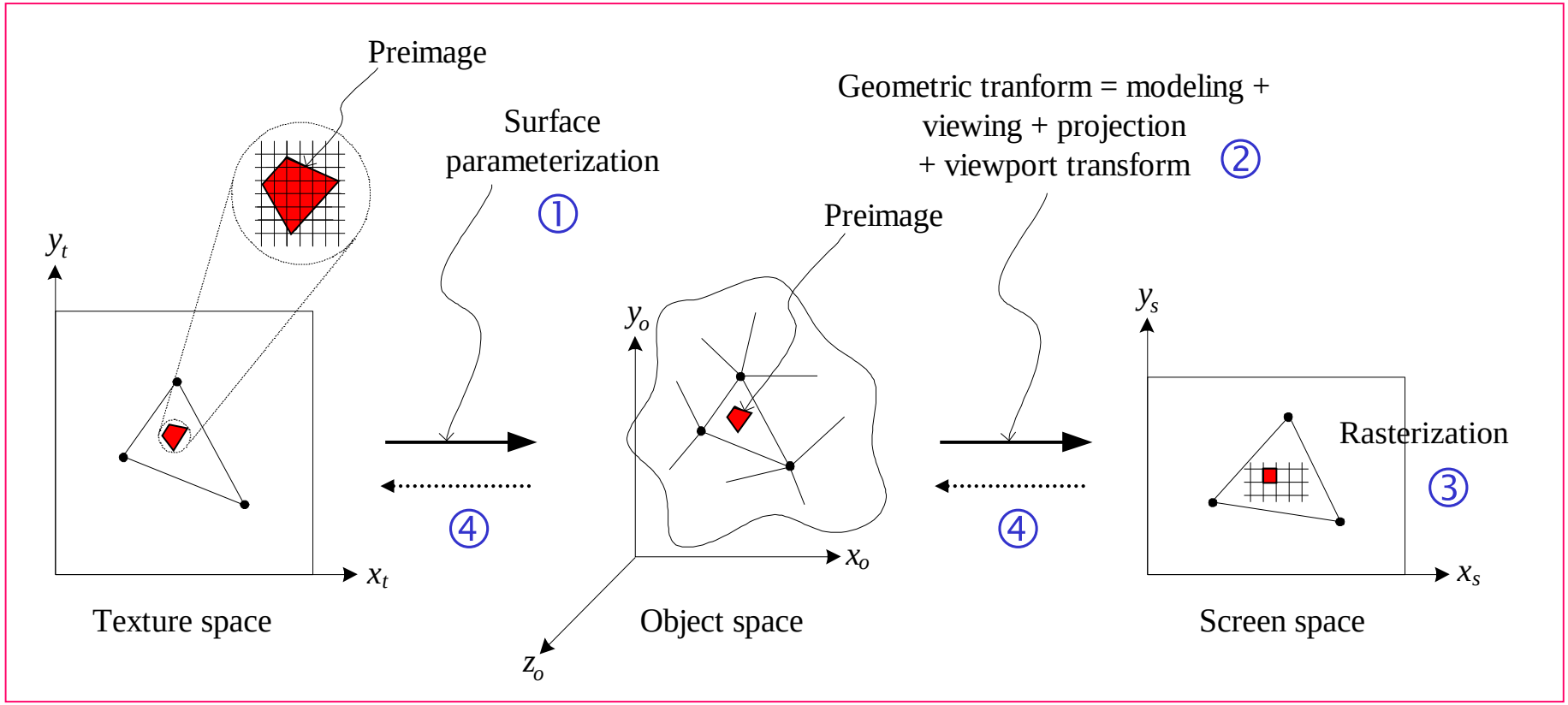
Distance where the fog color obscures all other colors in the scene



u_flag_fog == TRUE

Texture Filtering

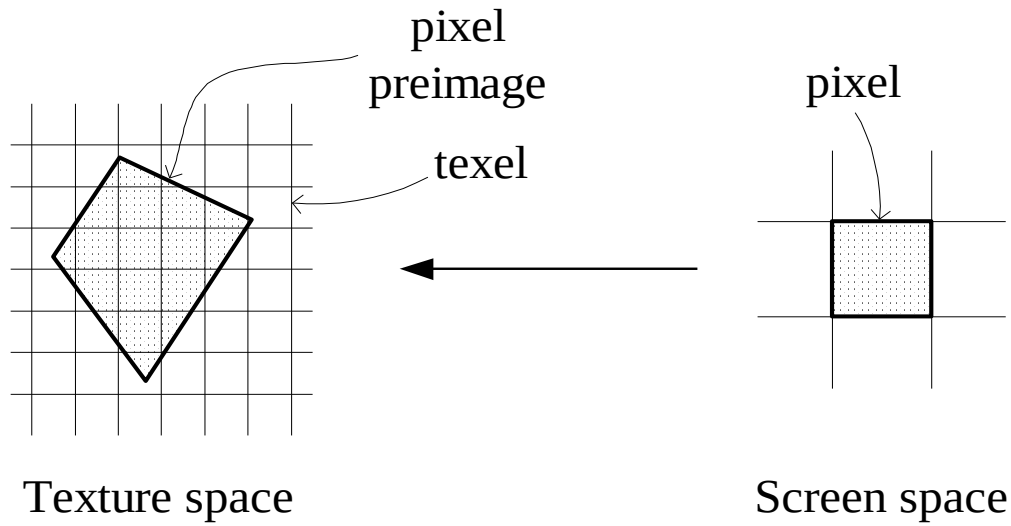
Texture Mapping Process Again

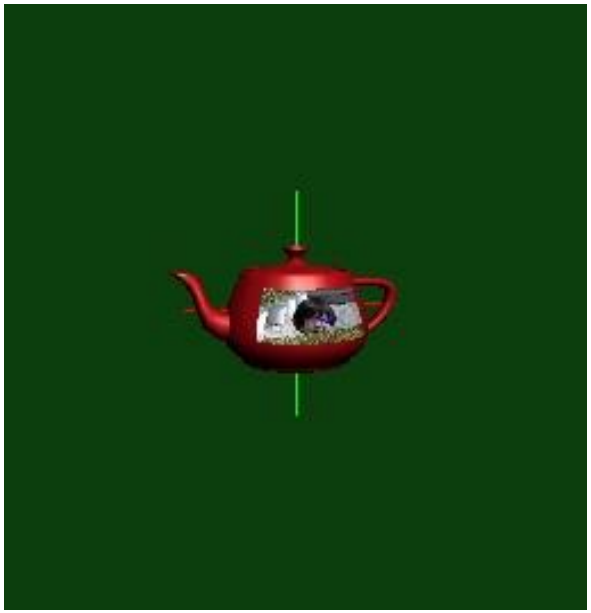


- **일반적으로 세 공간간의 매핑은 매우 복잡한 형태로 나타남.**
- **어떻게 하면 픽셀에 대응되는 preimage 영역의 색깔을 가급적 정확하고 빠르게 추정할 수 있을까?**

텍스춰 필터링 문제

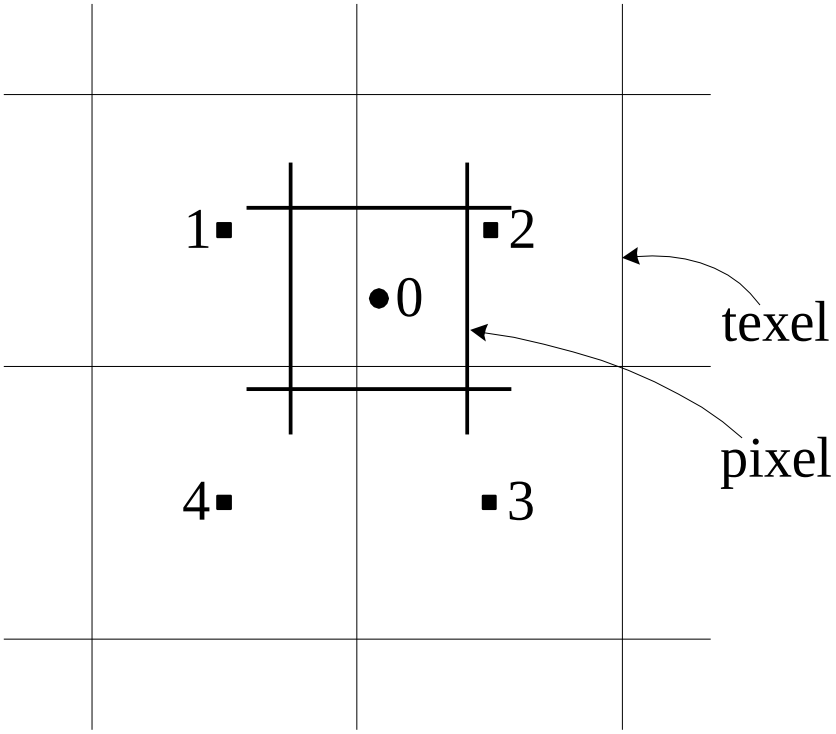
- ***Texture filtering*** *or* ***texture smoothing*** *is the method used to determine the texture color for a texture mapped pixel, using the colors of nearby texels (pixels of the texture).* – Wikipedia.



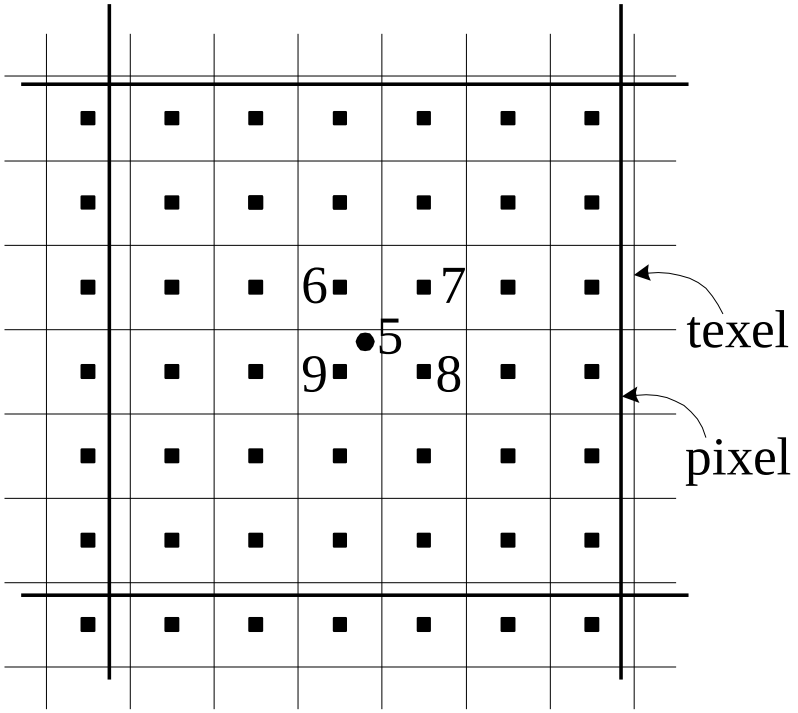


주어진 픽셀 영역과 텍스춰 이미지 공간에서의 매핑 영역간의 상대적인 크기는?

Texture Magnification and Minification



Magnification (확대)



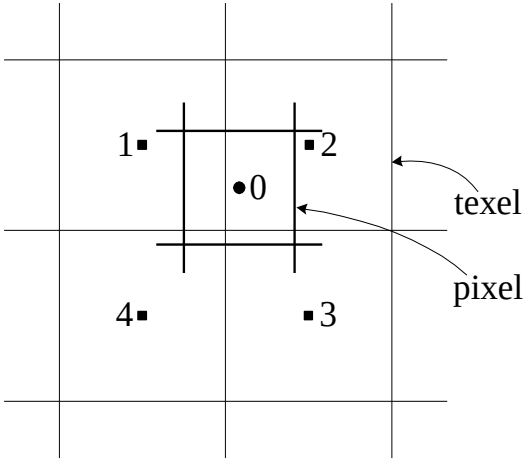
Minification (축소)

텍스춰 매핑을 위한 필터링 방법

- **최근 필터(nearest neighbor filter)**
  - GL_NEAREST
- **이선형 필터(bilinear interpolation filter)**
  - GL_LINEAR
- **삼선형 필터(trilinear interpolation filter):** Mipmap filter
  - GL_LINEAR_MIPMAP_LINEAR
- **혼합 필터(hybrid filter)**
  - GL_NEAREST_MIPMAP_LINEAR
  - GL_LINEAR_MIPMAP_NEAREST
  - GL_NEAREST_MIPMAP_NEAREST

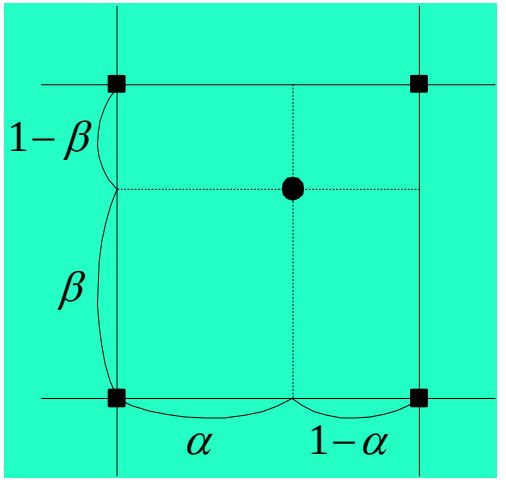
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, ??? );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, ??? );
```

확대 상황을 위한 필터



최근 필터

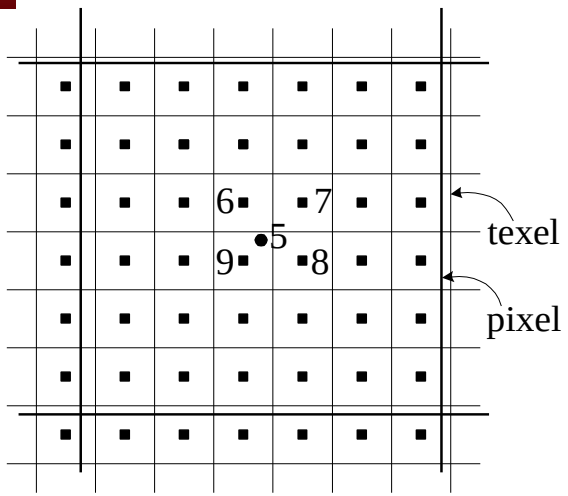
이선형 필터



축소 상황을 위한 필터

최근 필터

이선형 필터



축소 상황이 심해져도 이선형 필터가 잘 작동할까?

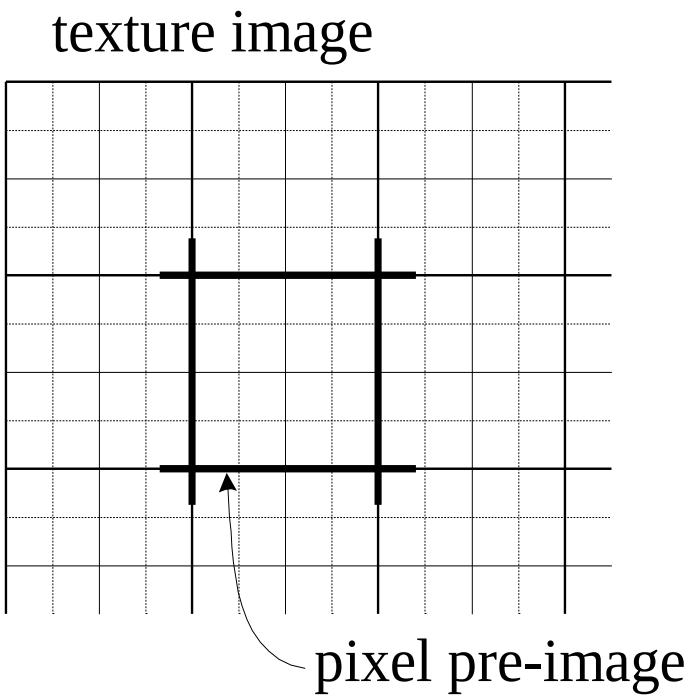
이선형 필터

삼선형 필터

축소 상황이 심해지면 구조적으로 이선형 필터도 불충분.

밉매핑(Mip-mapping)

- MIP(*multum in parvo*) = many things in small place

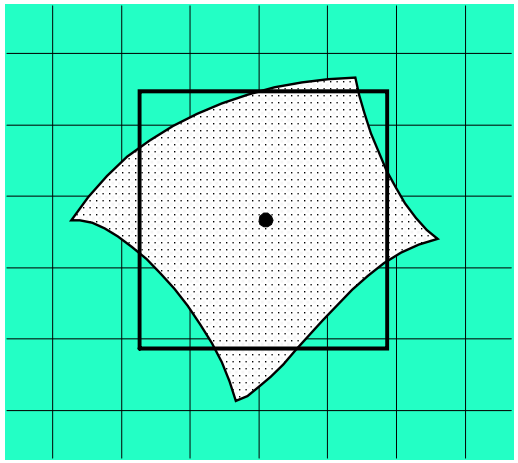


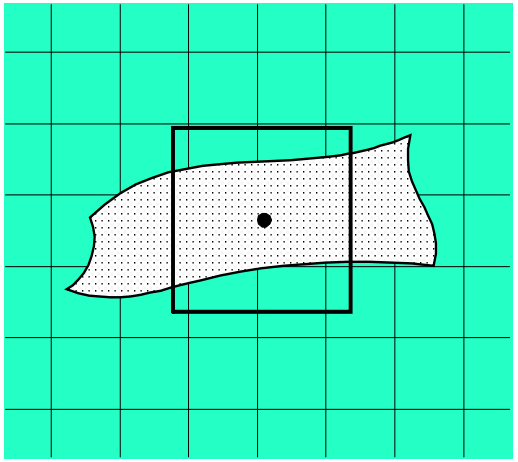
밉맵의 원리

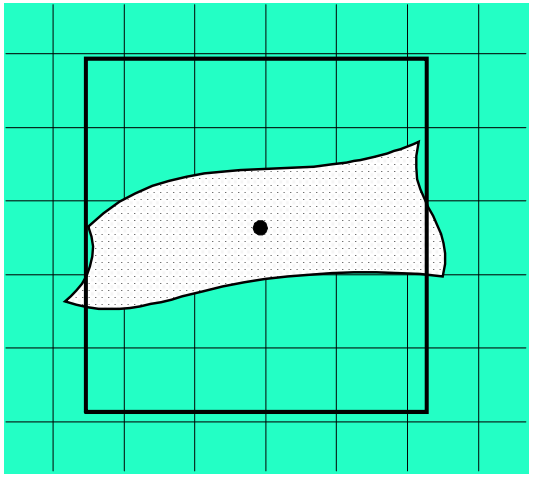
밉맵의 예

Box 필터 크기의 추정

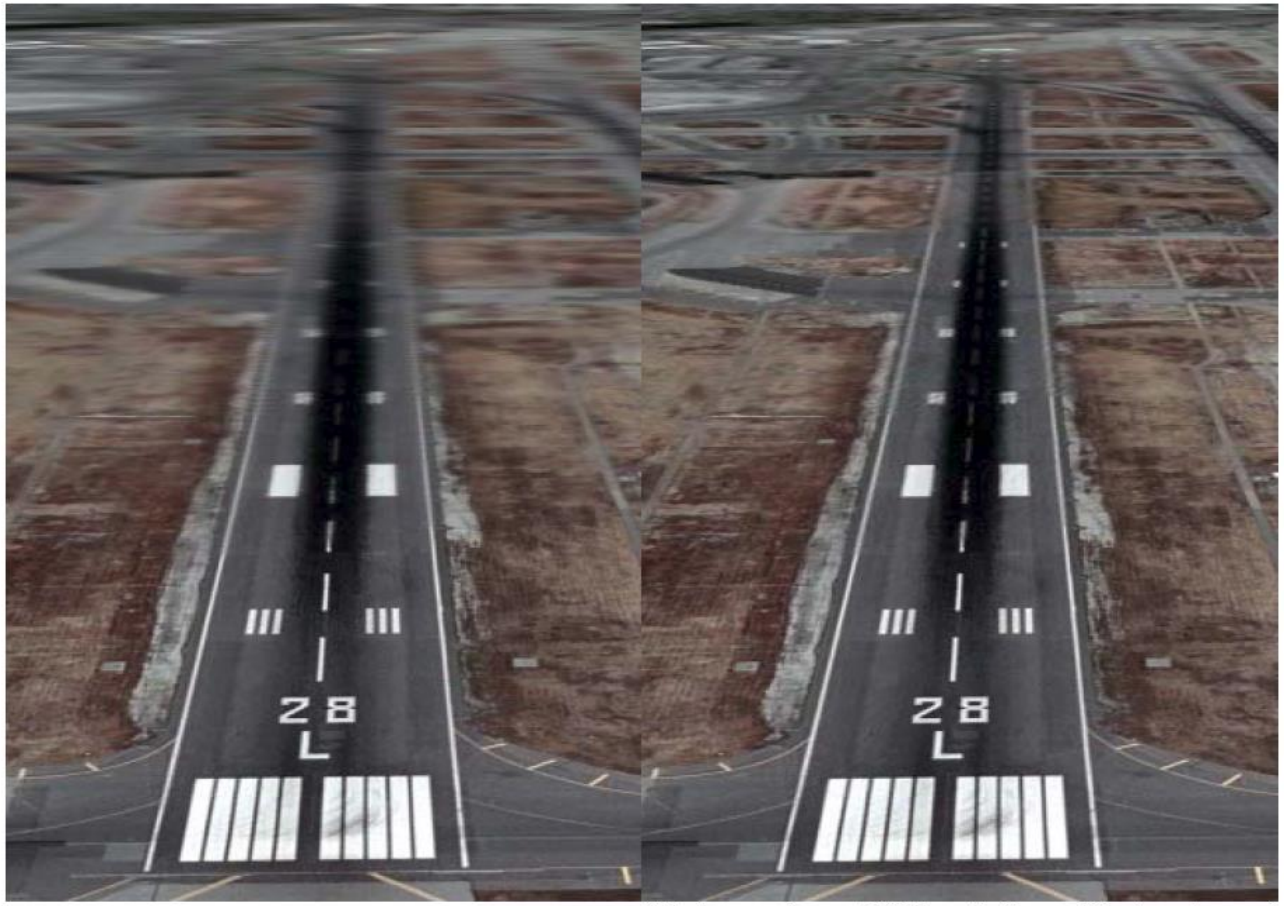
Square < Rectangle < Trapezoid







Isotropic versus Anisotropic Filtering



(Photo courtesy of Wikipedia for public domain use)

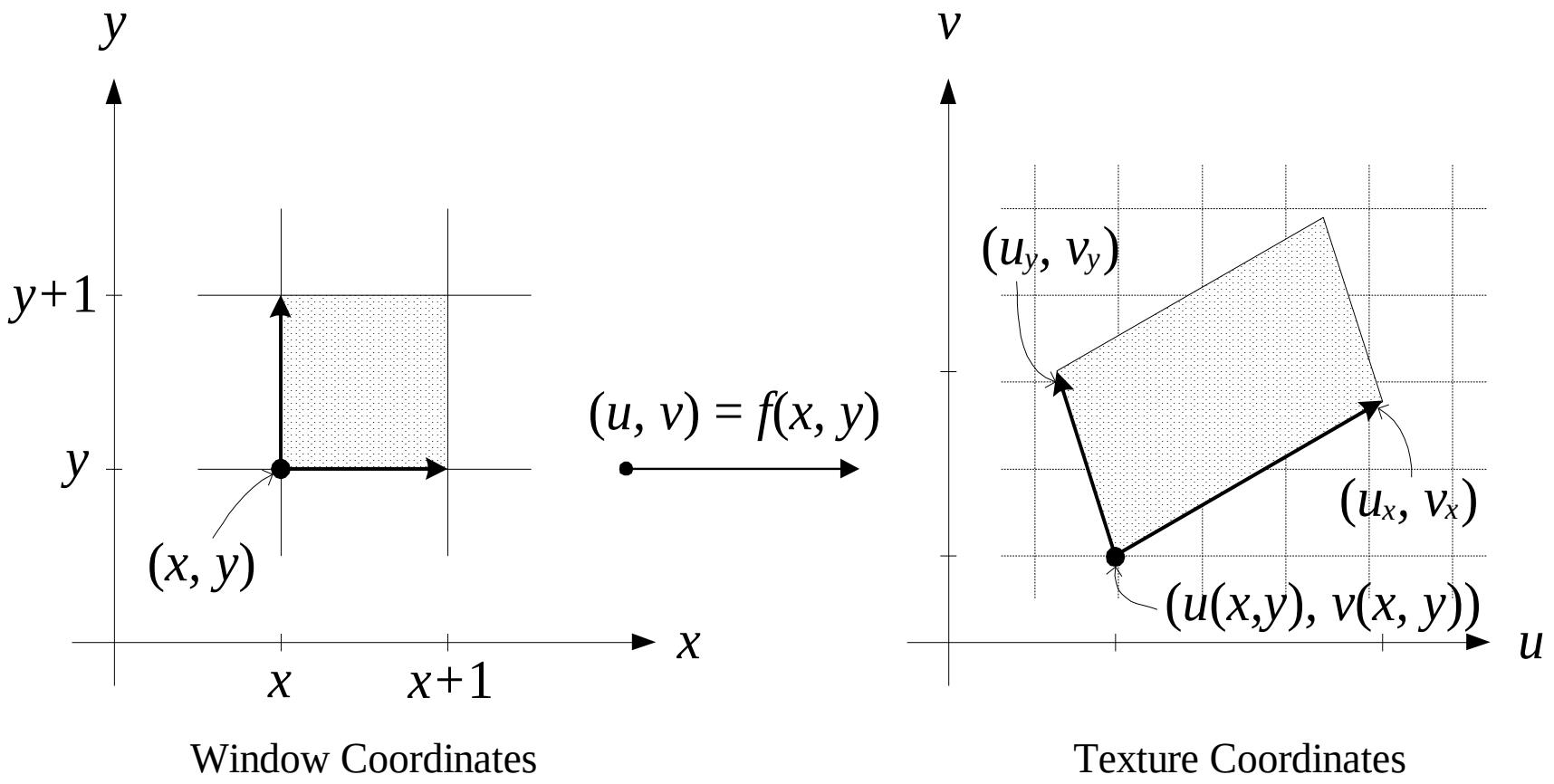
Figure 20. Isotropic trilinear mipmapping (left) vs. anisotropic trilinear mipmapping (right)

From **NVIDIA GeForce 8800 GPU Architecture Overview**

Anisotropic Filtering (AF) improves the clarity and sharpness of various scene objects that are viewed at sharp angles and/or recede into the distance (Figure 20). One example is a roadway billboard with text that looks skewed and blurred when viewed at a sharp angle (with respect to the camera) when standard bilinear and trilinear isotropic texture filtering methods are applied. Anisotropic filtering (combined with trilinear mipmapping) allows the skewed text to look much sharper. Similarly, a cobblestone roadway that fades into the distance can be sharpened with anisotropic filtering.

Anisotropic filtering is memory bandwidth intensive, particularly at high AF levels. For example, 16×AF means that up to 16 bilinear reads per each of two adjacent mipmap levels (128 memory reads total) are performed, and a weighted average is used to derive final texture color to apply to a pixel.

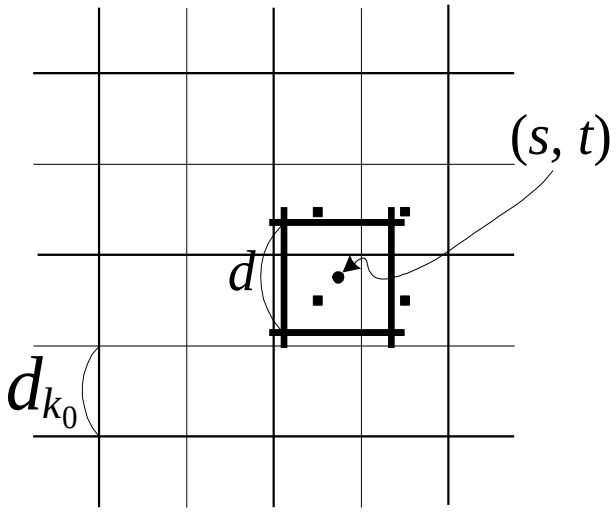
Mipmap Level의 결정



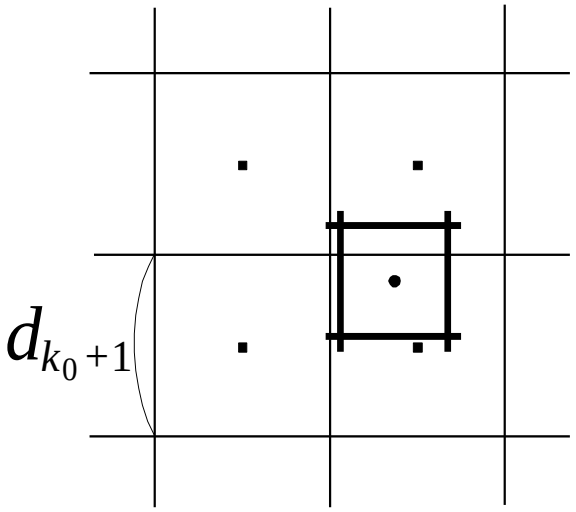
Window Coordinates

Texture Coordinates

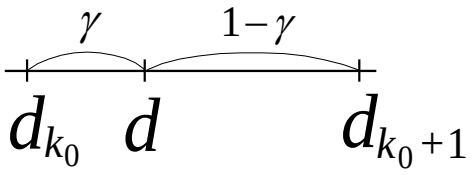
Trilinear Interpolation (삼선형 보간)

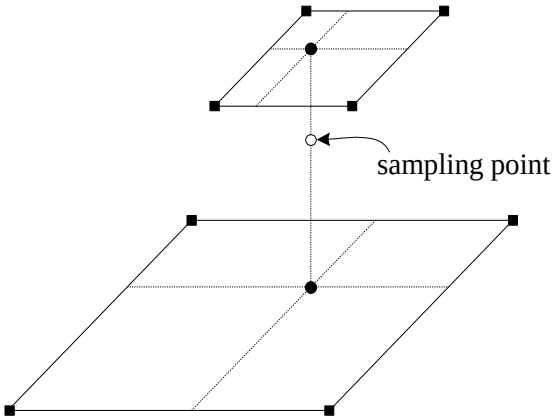


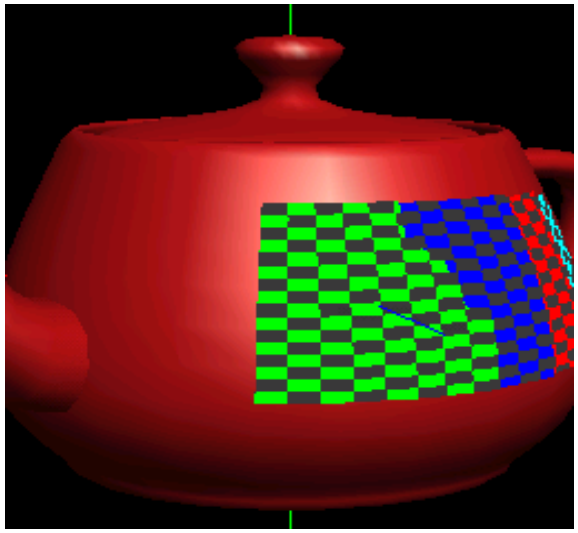
level k_0 texture



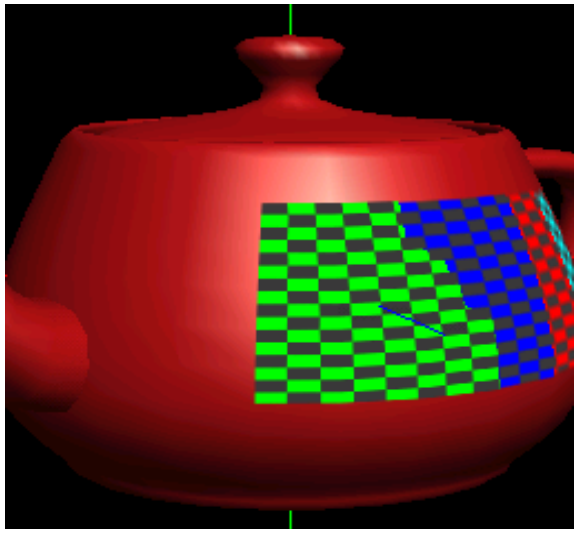
level k_0 + 1 texture



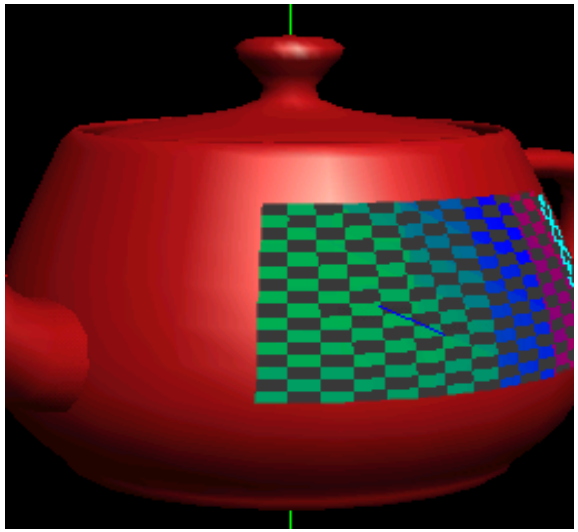




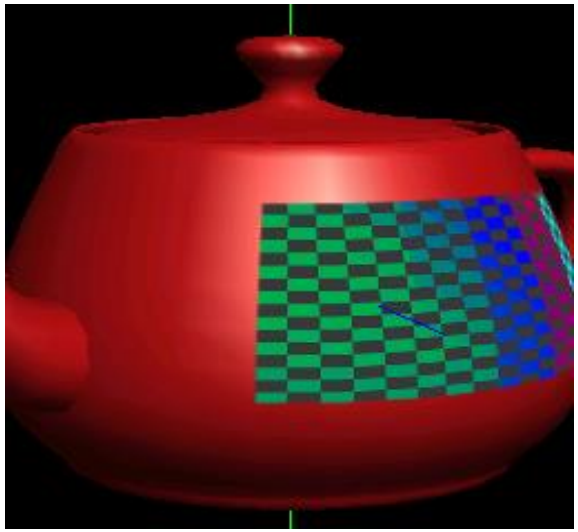
NEAREST_NEAREST



LINEAR_NEAREST



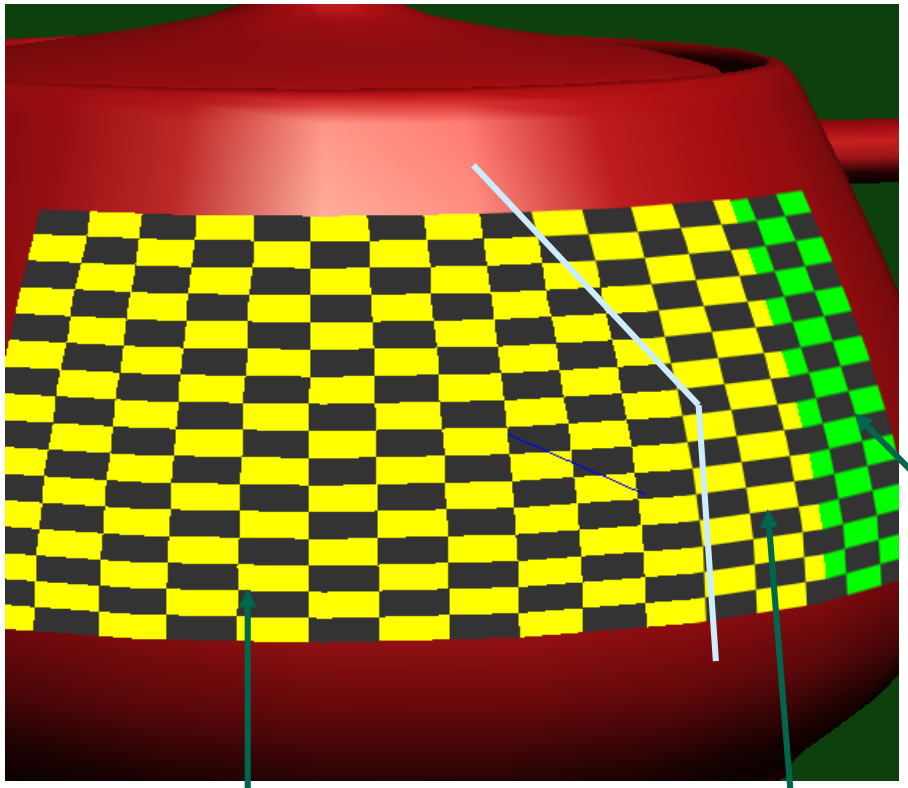
NEAREST_LINEAR



LINEAR_LINEAR

확대 상황과 축소 상황 예

- 사용 필터
  - ✓ 확대 필터: GL_NEAREST
  - ✓ 축소 필터: GL_LINEAR_MIPMAP_LINEAR



magnification

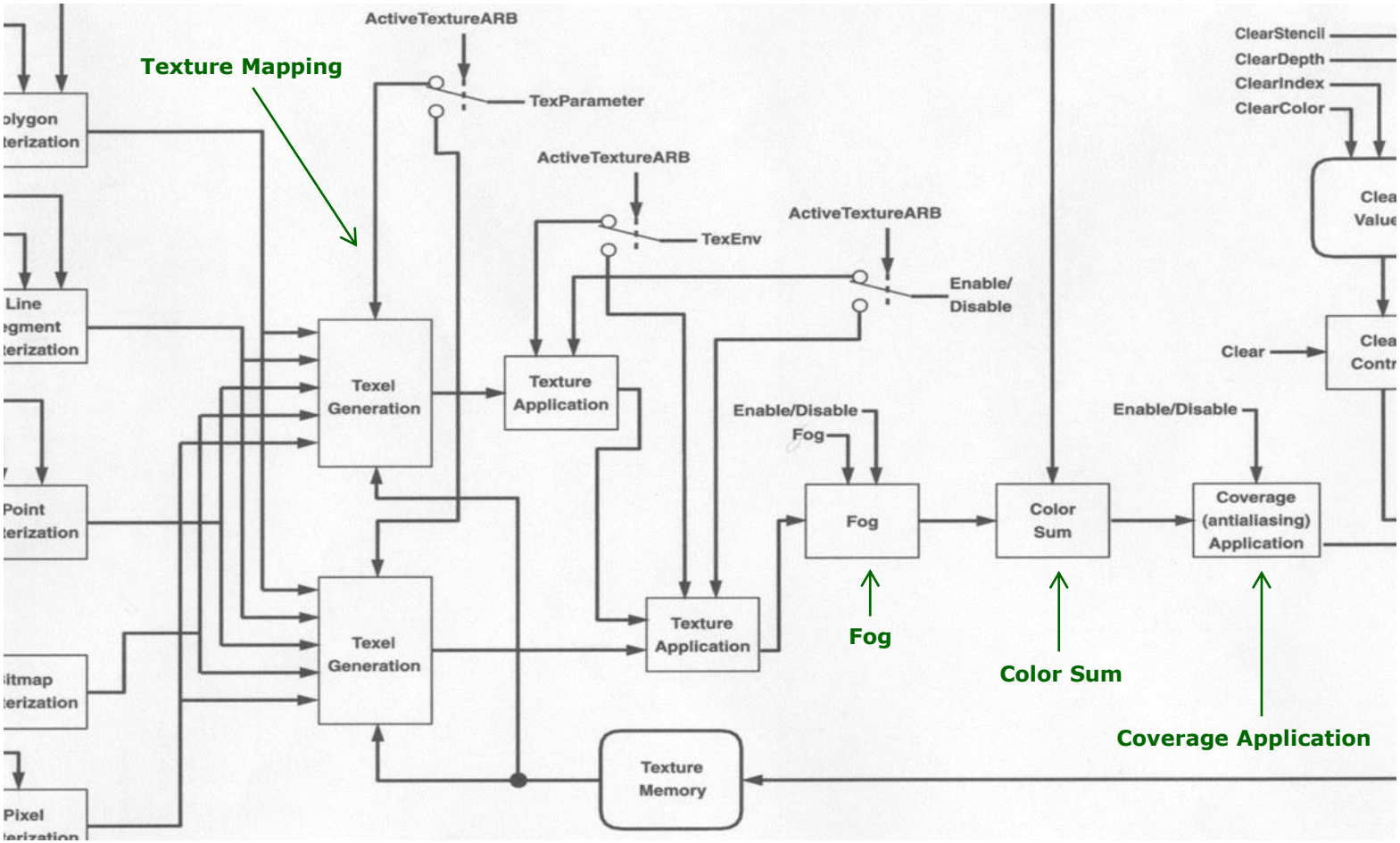
minification: level 0 texture

minification: level 1 texture

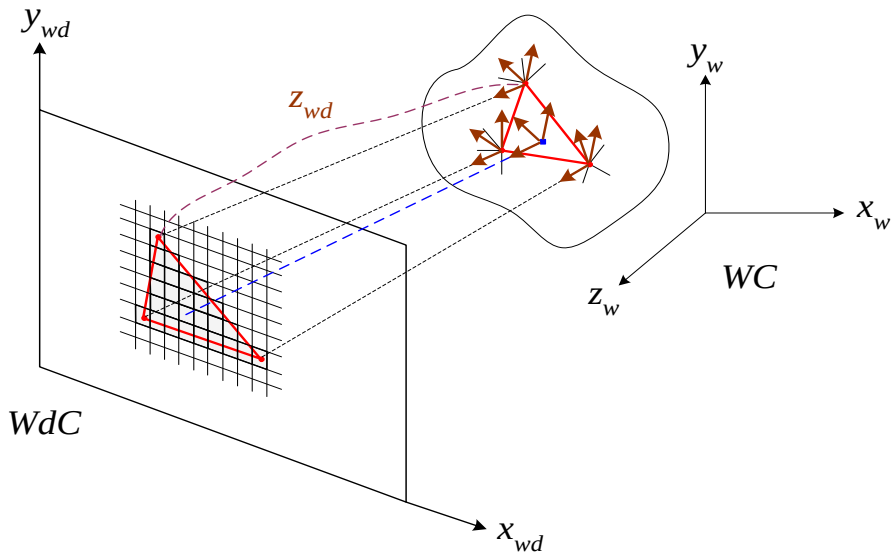
Texture Mapping, Fog, Color Sum, and Coverage(Antialiasing) Application

Pixel Shading in Fixed-Function OpenGL Pipeline

- **이 pixel shading 부분의 목적은 픽셀에 칠할 최종 색깔을 결정하는 것임.**
  - 바로 이 부분이 fragment (pixel) shader로 대치 됨.
  - 그 이후 Raster Operation (Per-Fragment Operation)은 아직도 fixed-function 형태로 남아 있음.



Fog Computation



- 현재 fragment에 대한 depth 값 (z 값)에 기반을 두어 fog color와 현재 fragment color를 blending factor *f*를 사용하여 혼합할 수 있도록 함.
- **Blending factor의 계산**
  - **GL_EXP**
    $$f = \exp(-d \cdot z)$$
  - **GL_EXP2**
    $$f = \exp(-(d \cdot z)^2)$$
  - **GL_LINEAR**
    $$f = \frac{e - z}{e - s}$$

z is the eye-coordinate distance from the eye to the fragment center.

- **Fogged color의 계산**
  $$C = fC_r + (1 - f)C_f$$

```
glClearColor(0.6f,0.6f,0.6f,1.0f);
glFogi(GL_FOG_MODE, GL_EXP);
glFogfv(GL_FOG_COLOR, FogColor);
glFogf(GL_FOG_DENSITY, 0.25f);
// glFogf(GL_FOG_START, 0.5f);
// glFogf(GL_FOG_END, 100.0f);
glHint(GL_FOG_HINT, GL_DONT_CARE);
glEnable(GL_FOG);
```

이 기능은 이제 vertex shader/fragment shader에서 처리하면 됨.
(f 값은 per-vertex 또는 per-fragment로 계산 가능)

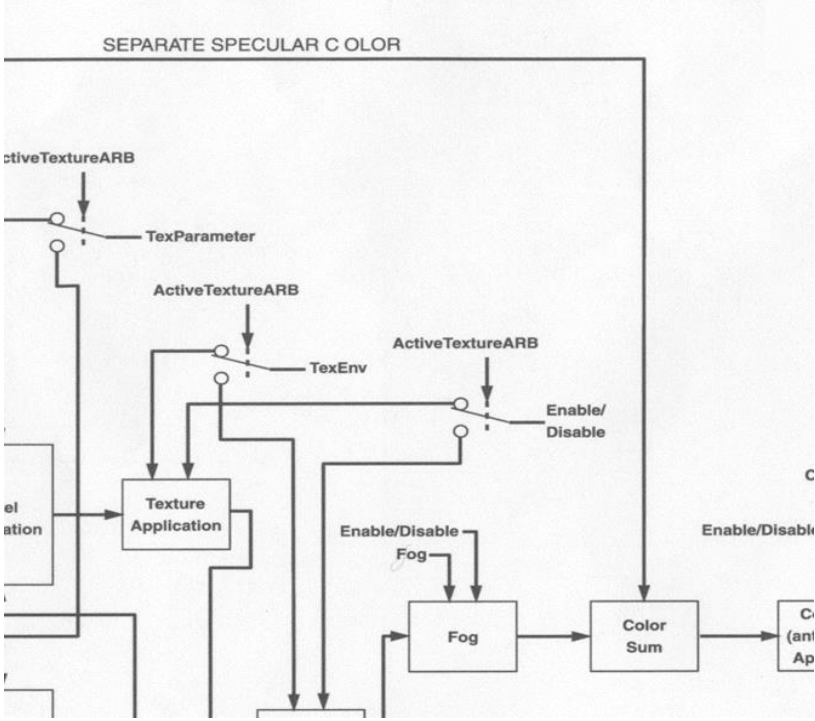
Color Sum

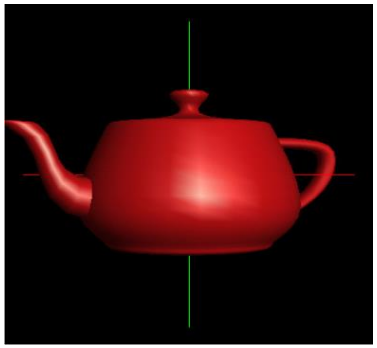
- Lighting 계산 과정에서 primary color와 secondary color로 색깔을 분리.

$$\mathbf{c}_{pri} = \mathbf{e}_{cm} + \mathbf{a}_{cm} * \mathbf{a}_{cs} + \sum$$

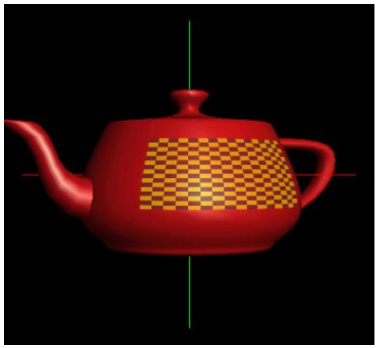
$$\mathbf{c}_{sec} = \sum_{i=0}^{n-1} (att_i)(spot_i)(f_i)(\mathbf{I}$$

- Pixel shading 과정에서 텍스춰 매핑 이후 그때까지 계산한 primary color와 이전에 계산을 해둔 secondary color와 더할 수 있도록 함.

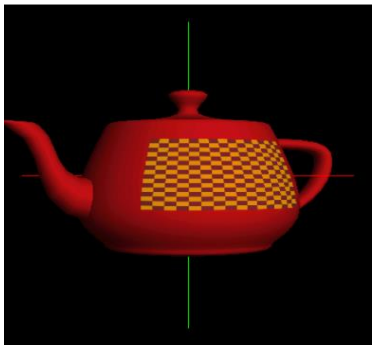




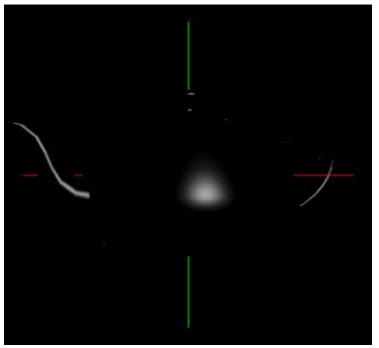
Gouraud Shading



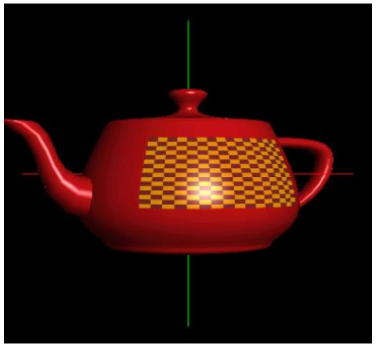
Gouraud Shading
+
Texture Mapping



Gouraud Shading
(Primary Color)
+
Texture mapping



Gouraud Shading
(Secondary Color)



Color Sum

이 기능은 이제 fragment shader에서 처리하면 됨.

[CSE4170 기초 컴퓨터 그래픽스]

2017년도 1학기

강의자료 III

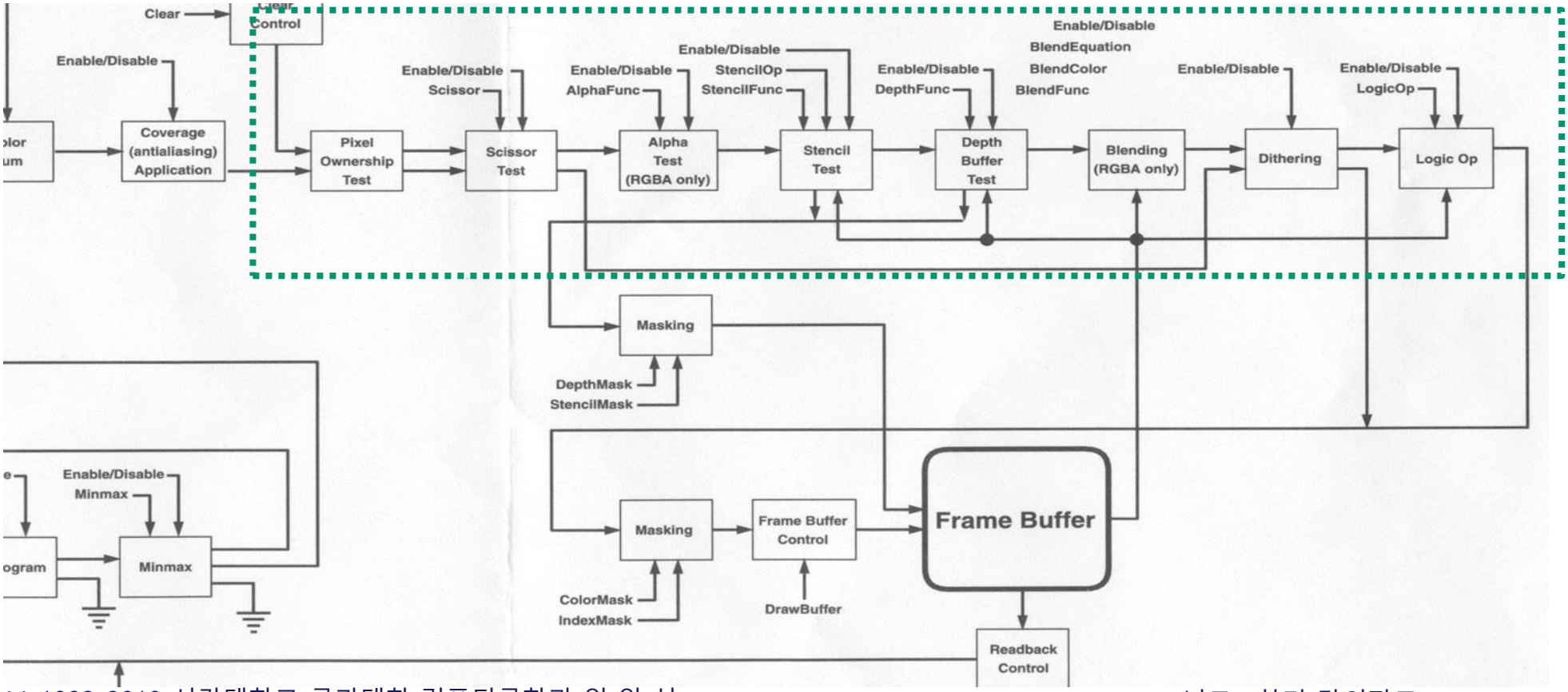
Rater Operation

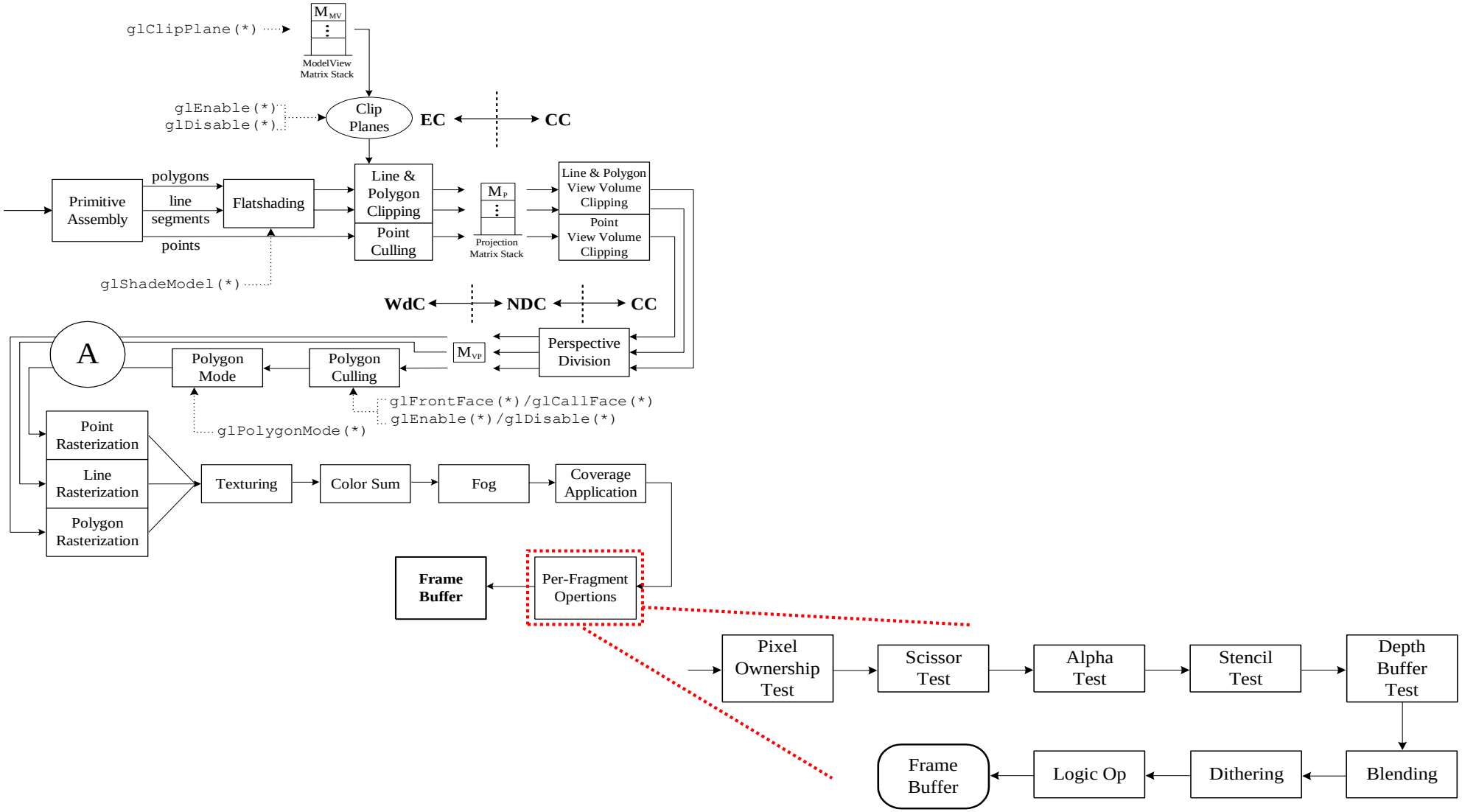
(Per-Fragment Operation)

Per-Fragment Operations

- WdC 좌표 (x_{wd}, y_{wd})를 가지는 현재 fragment에 대하여, pixel shading 한 결과를 frame buffer의 해당 메모리 영역에 write하기 전에 그 "적절성" 여부를 확인하거나 fragment color를 일부 수정할 수 있는 기능을 제공.

Programmable pipeline에서도 이 부분은 fixed-function 형태로 남아 있음.




glClipPlane(*)
M_MV
ModelView Matrix Stack
glEnable(*)
glDisable(*)
Clip Planes
EC
CC
Primitive Assembly
polygons
line segments
points
Flatshading
Line & Polygon Clipping
Point Culling
M_P
Projection Matrix Stack
Line & Polygon View Volume Clipping
Point View Volume Clipping
glShadeModel(*)
WdC
NDC
CC
A
Polygon Mode
Polygon Culling
M_VP
Perspective Division
glFrontFace(*)/glCallFace(*)
glEnable(*)/glDisable(*)
glPolygonMode(*)
Point Rasterization
Line Rasterization
Polygon Rasterization
Texturing
Color Sum
Fog
Coverage Application
Frame Buffer
Per-Fragment Opertions
Pixel Ownership Test
Scissor Test
Alpha Test
Stencil Test
Depth Buffer Test
Blending
Dithering
Logic Op
Frame Buffer

- **Pixel Ownership Test**
  - Determine if the pixel at location (x_{wd}, y_{wd}) in the framebuffer is owned by the current GL context.
    - If failed, the fragment is discarded or some subset of the subsequent per-fragment operations are applied to the fragment.
- **Scissor Test** `glEnable(GL_SCISSOR_TEST);`
  - Determine if (x_{wd}, y_{wd}) lies within the scissor rectangle defined by

    ```
    glScissor(GLint x, GLint y, GLsizei width, GLsizei height);
    ```

    - If failed, the fragment is discarded.
- **Alpha Test** `glEnable(GL_ALPHA_TEST);`
  - Discard a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value.
  - The condition is set by

    ```
    glAlphaFunc(GLenum func, GLclampf ref);
    ```

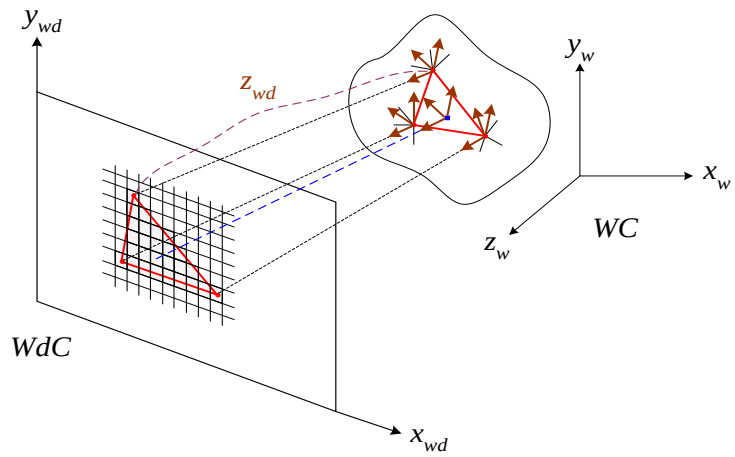
    - `For instance, glAlphaFunc(GL_GREATER, 0.1);`

이 기능은 이제 fragment shader에서 처리하면 됨. OpenGL ES 2.0에서는 이 기능이 제거됨.

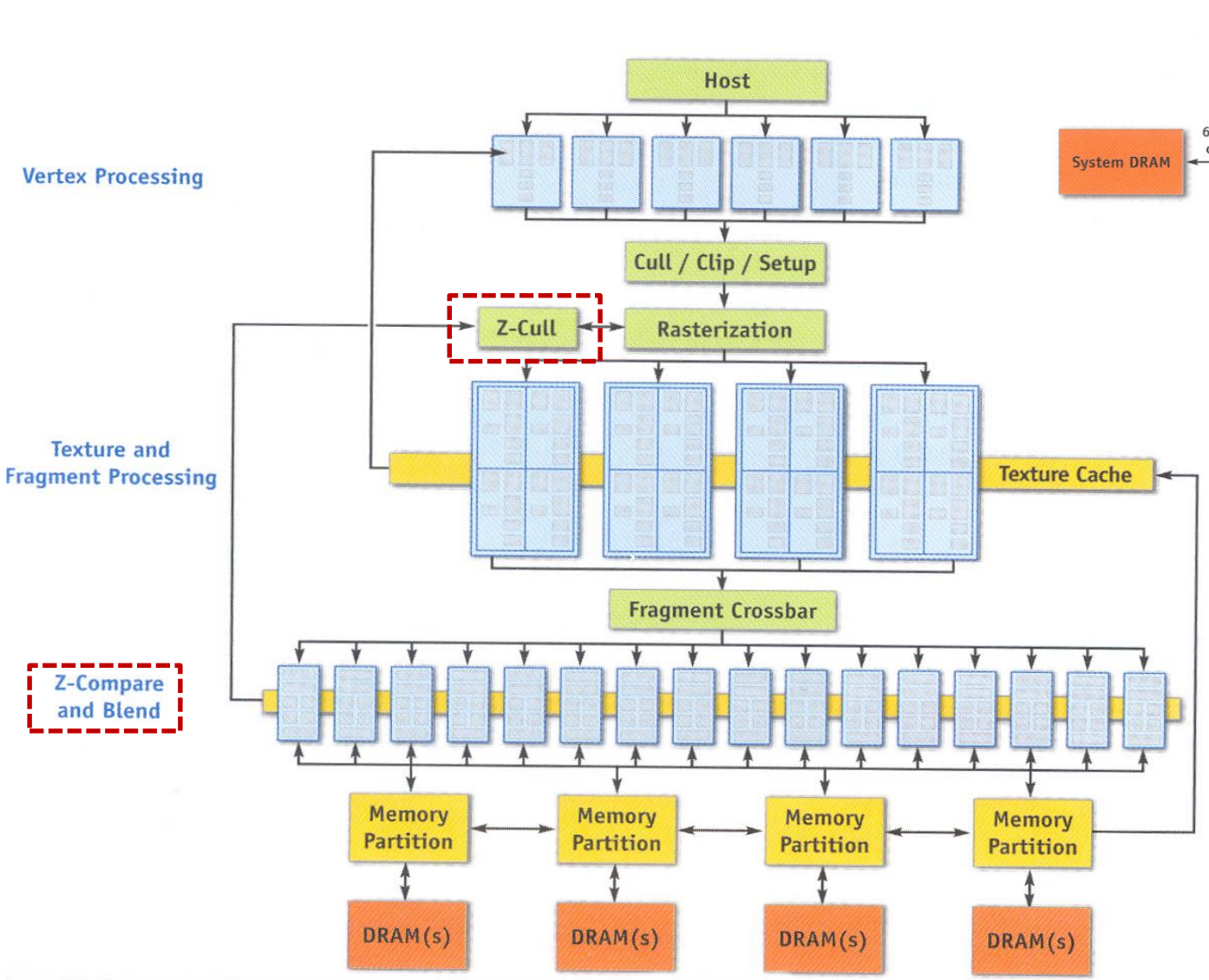
- **Stencil Test** `glEnable(GL_STENCIL_TEST);`
  - Conditionally discard a fragment based on the outcome of a comparison between the value in the **stencil buffer** at location (x_{wd}, y_{wd}) and a reference value.

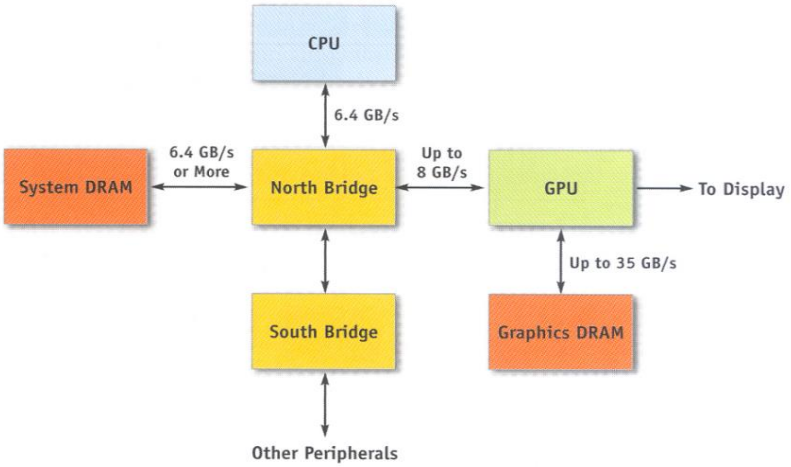
- **Depth Test** `glEnable(GL_DEPTH_TEST);`
  - Conditionally discard a fragment based on the outcome of a comparison between the depth value in the **depth buffer** at (x_{wd}, y_{wd}) location and the depth value of the incoming fragment.
    - If the test passes, the value of the depth buffer at the fragment's (x_{wd}, y_{wd}) location is set to the fragment's z_w value.
  - The comparison is specified with the function

    `glDepthFunc(GLenum func);`
    - **`GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_EQUAL, GL_GREATER, GL_GEQUAL, GL_NOTEQUAL`**
  - The default is **`GL_LESS`**, and is for **hidden-surface removal**.
    - For each pixel, the depth buffer always stores the distance to the closest object location drawn into the pixel so far.



2004년: GeForce 6 Series (NV40) - 6800 Ultra

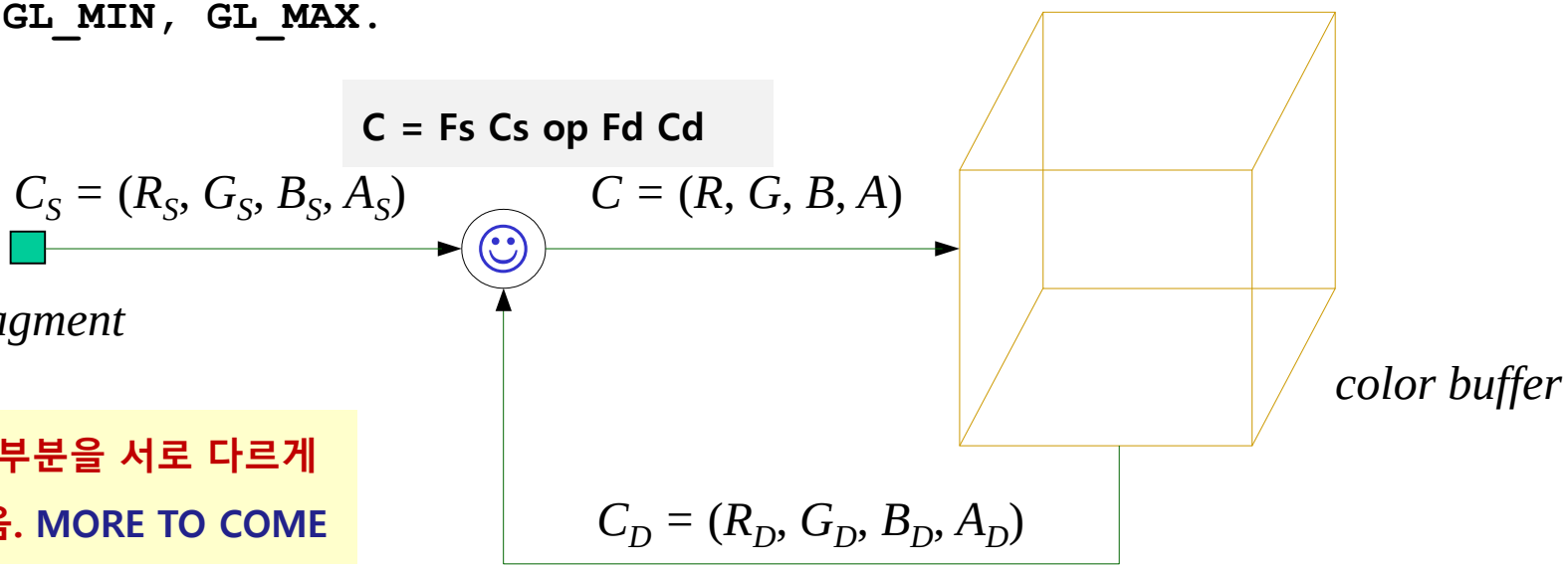




Shader Model 3.0
- standard FP32 precision
- dynamic branching
- longer shader lengths

© NVIDIA

- **Blending** `glEnable(GL_BLEND);`
  - Combine the incoming fragment's RGBA color (source color) with the RGBA color (destination color) stored in the framebuffer at the incoming fragment's (x_{wd}, y_{wd}) location.
  - The blending operation is defined by the functions
    - `glBlendFunc(GLenum sfactor, GLenum dfactor);`
    - `glBlendColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);`
    - `glBlendEquation(GLenum mode);`
      - `GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT, GL_MIN, GL_MAX.`



최근에는 RGB와 A 부분을 서로 다르게 blending 할 수 있음. MORE TO COME

- **Dithering** `glEnable(GL_DITHER);`
  - When the number of colors available in the framebuffer is limited, simulate greater color depth using dithering.

- **Logical Operation** `glEnable(GL_COLOR_LOGIC_OP);`
  - A logical operation is applied between the incoming fragment's color and the color values stored at the corresponding location in the framebuffer.

OpenGL ES 2.0에서는 이 기능이 제거됨.

Texture Mapping Application

- Shadow Mapping -

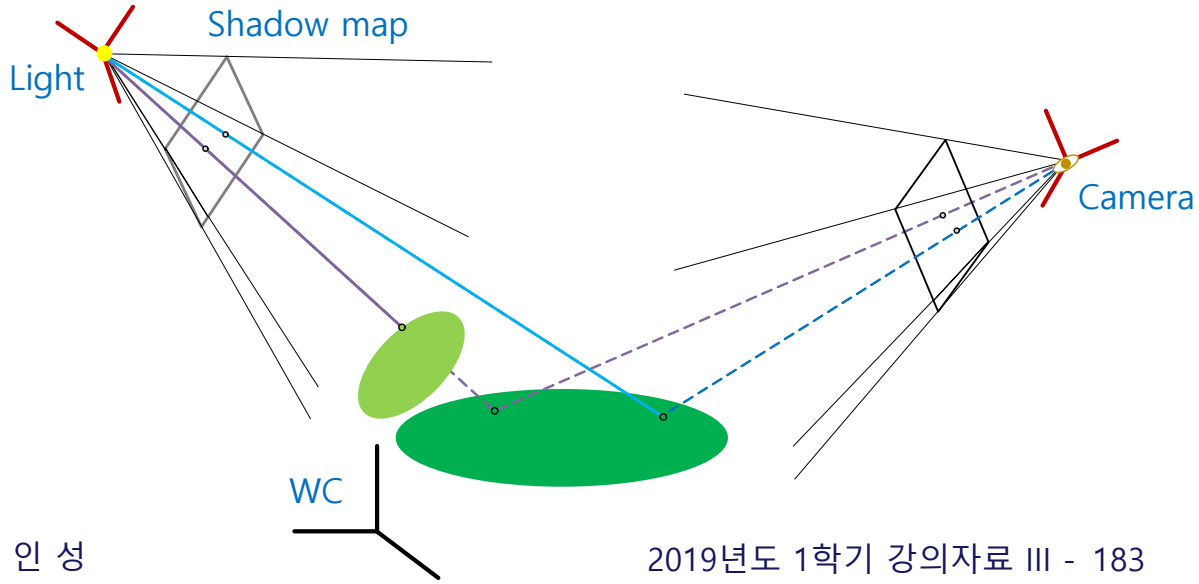
Shadow Generation Using Shadow Map

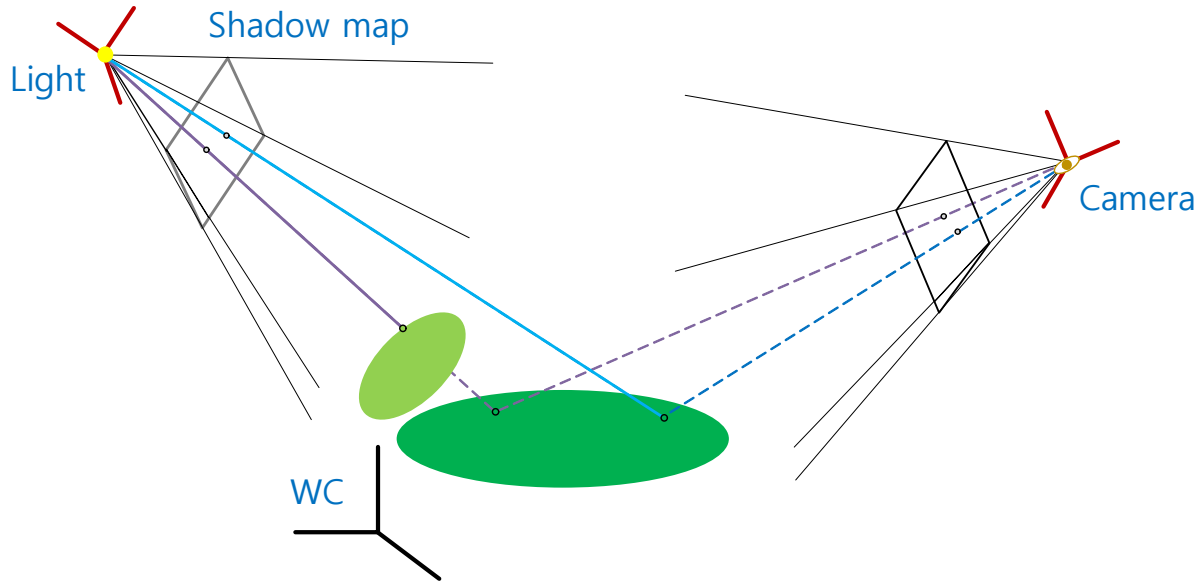
- **Basic idea**

  [First Step] Render the scene from the light's point of view, and store the depth buffer into a texture map, called the shadow map.

  [Second Step] Draw the scene from the camera's point of view, applying the shadow map.

  - For a current fragment, find the distance from the **corresponding surface point** to the light.
  - Compare the distance against the **corresponding depth map value**.
  - Depending on the result, the surface point is in shadow or in light.





Rendering without shadow

Shadow map

Rendering with shadow

Pass 1: Shadow Map Generation

- **How**
  - Draw the scene using the camera corresponding to the light.
    - **Viewing transformation** for determining the position and orientation of the light
    - **Projection transformation** for determining the field of view of the light
  - Use the OpenGL's Framebuffer object to store the depth map directly into a texture map.
  - After the generation, the resulting shadow map can be accessed through the texture coordinates (s, t), each of which ranges from 0 to 1.

```
// Initialize the shadow map
ShadowMapping.texture_unit = TEXTURE_INDEX_SHADOW;
ShadowMapping.shadow_map_width = SHADOW_MAP_WIDTH;
ShadowMapping.shadow_map_height = SHADOW_MAP_HEIGHT;
ShadowMapping.near_dist = SHADOW_MAP_NEAR_DIST;
ShadowMapping.far_dist = SHADOW_MAP_FAR_DIST;
ShadowMapping.shadow_map_border_color[0] = 1.0f;
ShadowMapping.shadow_map_border_color[1] = 0.0f;
ShadowMapping.shadow_map_border_color[2] = 0.0f;
ShadowMapping.shadow_map_border_color[3] = 0.0f;

// Initialize the shadow map
glGenTextures(1, &ShadowMapping.shadow_map_ID);
glActiveTexture(GL_TEXTURE0 + ShadowMapping.texture_unit);
glBindTexture(GL_TEXTURE_2D, ShadowMapping.shadow_map_ID);
glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT32,
               ShadowMapping.shadow_map_width,
               ShadowMapping.shadow_map_height);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
               GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
               GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
               GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
               GL_CLAMP_TO_BORDER);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
                 ShadowMapping.shadow_map_border_color);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC,
                GL_LESS);
glUseProgram(h_ShaderProgram_TXPS);
glUniform1i(loc_shadow_texture, ShadowMapping.texture_unit);
glUseProgram(0);

// Initialize the Frame Buffer Object for rendering shadows
glGenFramebuffers(1, &ShadowMapping.FBO_ID);
glBindFramebuffer(GL_FRAMEBUFFER, ShadowMapping.FBO_ID);
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
                       ShadowMapping.shadow_map_ID, 0);
glDrawBuffer(GL_NONE);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
                                    GL_FRAMEBUFFER_COMPLETE) {
    fprintf(stderr, "Error: the framebuffer object for shadow
mapping is not complete...₩n");
    exit(-1);
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);

ViewMatrix_SHADOW = glm::lookAt(glm::vec3(light[0].position[0],
                    light[0].position[1], light[0].position[2]),
                    glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
ProjectionMatrix_SHADOW = glm::perspective(TO_RADIAN*60.0f,
          1.0f, ShadowMapping.near_dist, ShadowMapping.far_dist);
BiasMatrix = glm::mat4(0.5f, 0.0f, 0.0f, 0.0f, 0.0f, 0.5f, 0.0f, 0.0f,
                       0.0f, 0.0f, 0.5f, 0.0f, 0.5f, 0.5f, 0.5f, 1.0f);
```

```
void build_shadow_map(void) {
  glm::mat4 ViewProjectionMatrix_SHADOW;
  ViewProjectionMatrix_SHADOW = ProjectionMatrix_SHADOW
                                          * ViewMatrix_SHADOW;
  glBindFramebuffer(GL_FRAMEBUFFER, ShadowMapping.FBO_ID);
  glViewport(0, 0, ShadowMapping.shadow_map_width,
                   ShadowMapping.shadow_map_height);

  glClear(GL_DEPTH_BUFFER_BIT);
  glEnable(GL_POLYGON_OFFSET_FILL);
  glPolygonOffset(10.0f, 10.0f);

  glUseProgram(h_ShaderProgram_shadow);

  ModelViewProjectionMatrix =  ... ;
  glUniformMatrix4fv(loc_ModelViewProjectionMatrix_shadow, 1,
                           GL_FALSE, &ModelViewProjectionMatrix[0][0]);
  draw_floor();
  // Draw other objects too.

  glUseProgram(0);

  glFinish();

  glDisable(GL_POLYGON_OFFSET_FILL);
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Vertex shader

```
#version 330

uniform mat4 u_ModelViewProjectionMatrix;

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_tex_coord;

void main(void) {
   gl_Position = u_ModelViewProjectionMatrix*vec4(a_position, 1.0f);
}
```

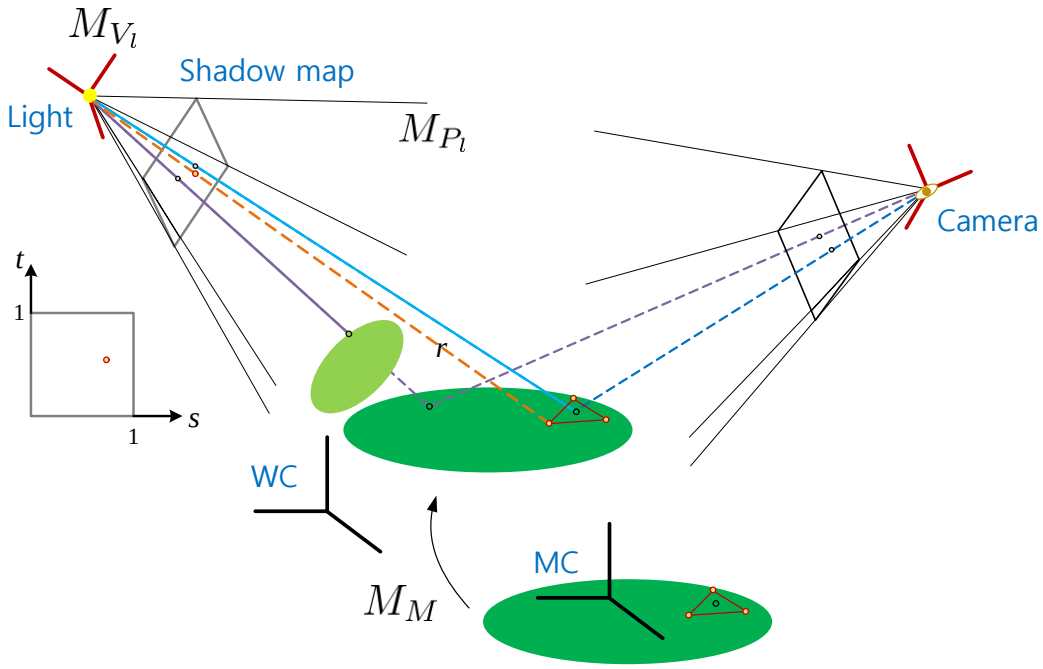
Fragment shader

```
#version 330

void main(void) {
   // do nothing!
}
```

Pass 2: Shadow Map Application

- **Hoq**
  - During the regular rendering, how do we know the **shadow map coordinates (s, t)** and the **distance r** for the **current fragment's surface point**?
    - During the **vertex shading** stage, use an attribute of the vertex to hold the (S, T, R, Q) information that can extract (s, t, r).



- During the **rasterization** stage, the vertex attributes are linearly interpolated for the current fragment.
- During the **fragment shading** stage, the **textureProj()** function automatically performs the perspective division to produce (s, t, r) for the current fragment.

$$\begin{pmatrix} S \\ T \\ R \\ Q \end{pmatrix} = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot M_{P_l} \cdot M_{V_l} \cdot \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} s \\ t \\ r \end{pmatrix} = \begin{pmatrix} S/Q \\ T/Q \\ R/Q \end{pmatrix}$$

```
void draw_scene_with_shadow(void) {
  glm::mat4 ModelMatrix;
  glViewport(0, 0, WINDOW_param.width, WINDOW_param.height);

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glUseProgram(h_ShaderProgram_simple);
  // Draw axes.
  glUseProgram(h_ShaderProgram_TXPS);
  glUniform1i(loc_shadow_texture, ShadowMapping.texture_unit);
  set_material_floor();
  glUniform1i(loc_base_texture, TEXTURE_INDEX_FLOOR);
  ModelMatrix = ... ;
  ModelViewMatrix = ... ;
  ModelViewProjectionMatrix = ... ;
  ModelViewMatrixInvTrans = ...  ;
  ShadowMatrix = BiasMatrix * ProjectionMatrix_SHADOW *
                 ViewMatrix_SHADOW * ModelMatrix;

  glUniformMatrix4fv(loc_ModelViewProjectionMatrix_TXPS, 1,
                     GL_FALSE, &ModelViewProjectionMatrix[0][0]);
  glUniformMatrix4fv(loc_ModelViewMatrix_TXPS, 1, GL_FALSE,
                     &ModelViewMatrix[0][0]);
  glUniformMatrix3fv(loc_ModelViewMatrixInvTrans_TXPS, 1,
                     GL_FALSE, &ModelViewMatrixInvTrans[0][0]);
  glUniformMatrix4fv(loc_ShadowMatrix_TXPS, 1, GL_FALSE,
                     &ShadowMatrix[0][0]);
  draw_floor();
  // Draw other objects too.
  glUseProgram(0);
}
```

Vertex shader

```
#version 330

uniform mat4 u_ModelViewProjectionMatrix;
uniform mat4 u_ModelViewMatrix;
uniform mat3 u_ModelViewMatrixInvTrans;
uniform mat4 u_ShadowMatrix;

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_tex_coord;

out vec3 v_position_EC;
out vec3 v_normal_EC;
out vec2 v_tex_coord;
out vec4 v_shadow_coord;

void main(void) {
  v_position_EC = vec3(u_ModelViewMatrix*vec4(a_position, 1.0f));
  v_normal_EC = normalize(u_ModelViewMatrixInvTrans*a_normal);
  v_tex_coord = a_tex_coord;

  v_shadow_coord = u_ShadowMatrix * vec4(a_position, 1.0f);

  gl_Position = u_ModelViewProjectionMatrix * vec4(a_position, 1.0f);
}
```

Fragment shader

```
#version 330
...
uniform sampler2D u_base_texture;
uniform sampler2DShadow  u_shadow_texture;
...
uniform bool u_flag_texture_mapping = true;
uniform bool u_flag_fog = false;
...
in vec3 v_position_EC;
in vec3 v_normal_EC;
in vec2 v_tex_coord;
in vec4 v_shadow_coord;

layout (location = 0) out vec4 final_color;

vec4 lighting_equation_textured(in vec3 P_EC,
                                in vec3 N_EC, in vec4 base_color) {
  vec4 color_sum;
  float local_scale_factor, tmp_float, shadow_factor;
  vec3 L_EC;
  color_sum = u_material.emissive_color
                          + u_global_ambient_color * base_color;

  for (int i = 0; i < NUMBER_OF_LIGHTS_SUPPORTED; i++) {
    if (!u_light[i].light_on) continue;

    if (u_light[i].shadow_on) {
      shadow_factor = textureProj(u_shadow_texture, v_shadow_coord);
    }
    else shadow_factor = 1.0f;
    local_scale_factor = one_f;
    // Compute normalized L_EC and local_scale_factor.
    ...
    if (local_scale_factor > zero_f) {
      vec4 local_color_sum = u_light[i].ambient_color
                                        * u_material.ambient_color;
      if (shadow_factor > zero_f) {
        // Add up direct illumination to local_color_sum.
        ...
      }
      color_sum += local_scale_factor*local_color_sum;
    } // if (local_scale_factor > zero_f) {
  } // if (local_scale_factor > zero_f) {
  return color_sum;
}

#define FOG_COLOR vec4(0.7f, 0.7f, 0.7f, 1.0f)
#define FOG_NEAR_DISTANCE 350.0f
#define FOG_FAR_DISTANCE 700.0f

void main(void) {
  vec4 base_color, shaded_color;
  float fog_factor;

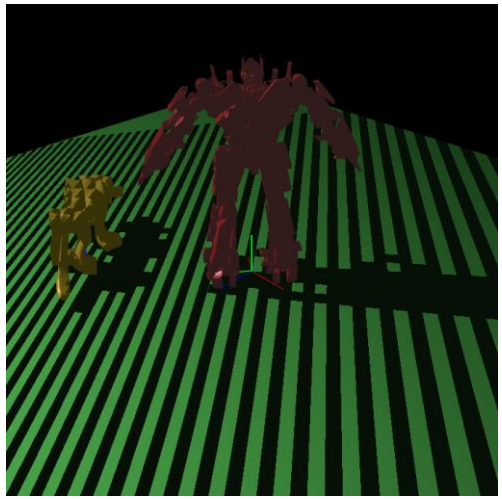
  if (u_flag_texture_mapping)
    base_color = texture(u_base_texture, v_tex_coord);
  else
    base_color = u_material.diffuse_color;
```

```
    shaded_color = lighting_equation_textured(v_position_EC,
                                    normalize(v_normal_EC), base_color);

    if (u_flag_fog) {
        fog_factor = (FOG_FAR_DISTANCE – length(v_position_EC.xyz))
                        / (FOG_FAR_DISTANCE - FOG_NEAR_DISTANCE);
        fog_factor = clamp(fog_factor, 0.0f, 1.0f);
        final_color = mix(FOG_COLOR, shaded_color, fog_factor);
    }
    else
    final_color = shaded_color;
}
```

Shadow Mapping without Depth Biasing

- **Shadow map resolution: 64x64**

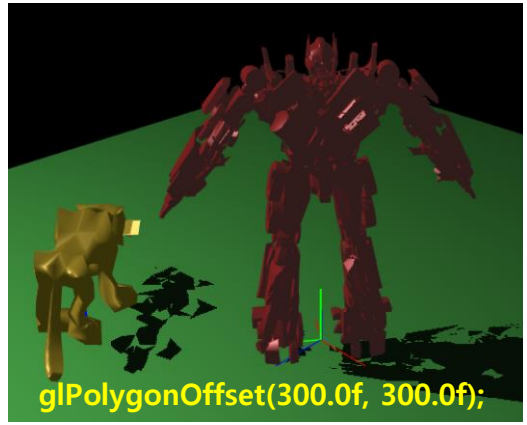



- **Shadow map resolution: 4096x4096**


- **Why does this happen?**
- **Does increasing the shadow map resolution help?**

Depth Biasing

- What would happen for excessive biasing?



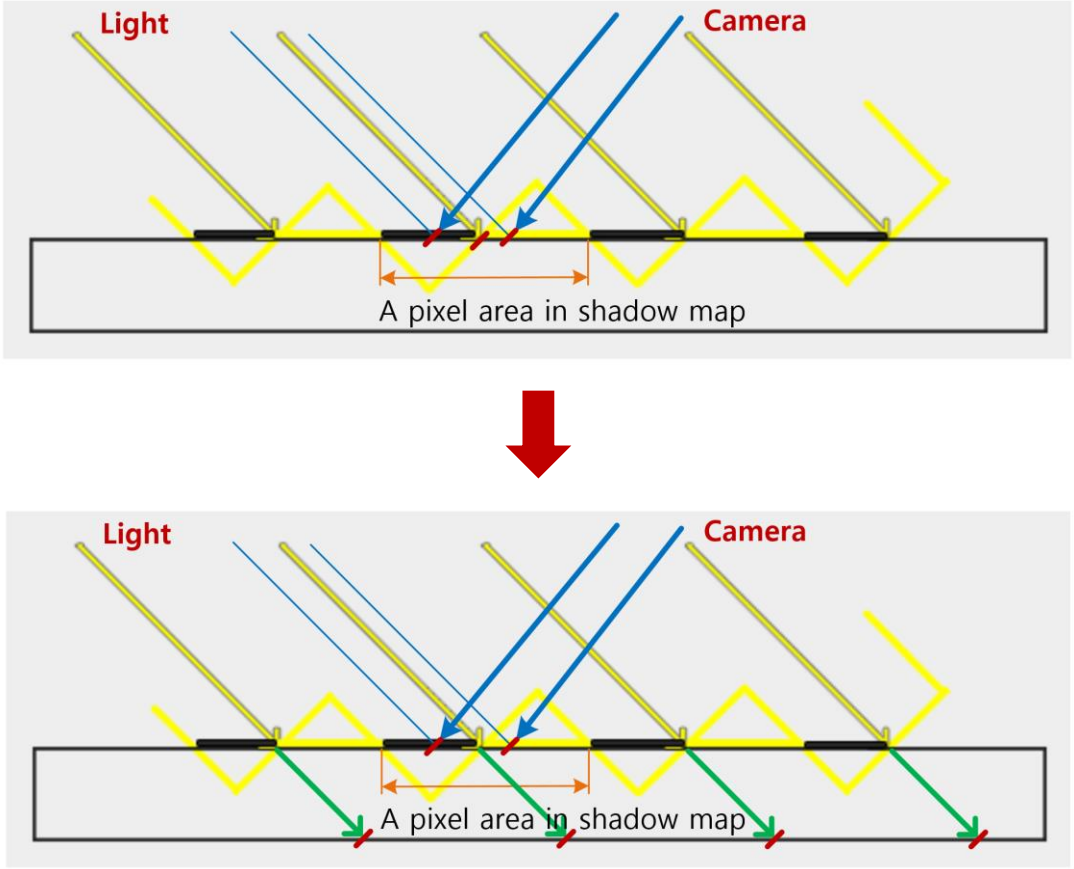
- **How**
  - During depth map generation, increase depth values slightly using `void glPolygonOffset(GLfloat factor, GLfloat units);`

```
void build_shadow_map(void) {
   ...
  glBindFramebuffer(GL_FRAMEBUFFER, ShadowMapping.FBO_ID);
  glViewport(0, 0, ShadowMapping.shadow_map_width,
                   ShadowMapping.shadow_map_height);

  glClear(GL_DEPTH_BUFFER_BIT);
  glEnable(GL_POLYGON_OFFSET_FILL);
  glPolygonOffset(10.0f, 10.0f);

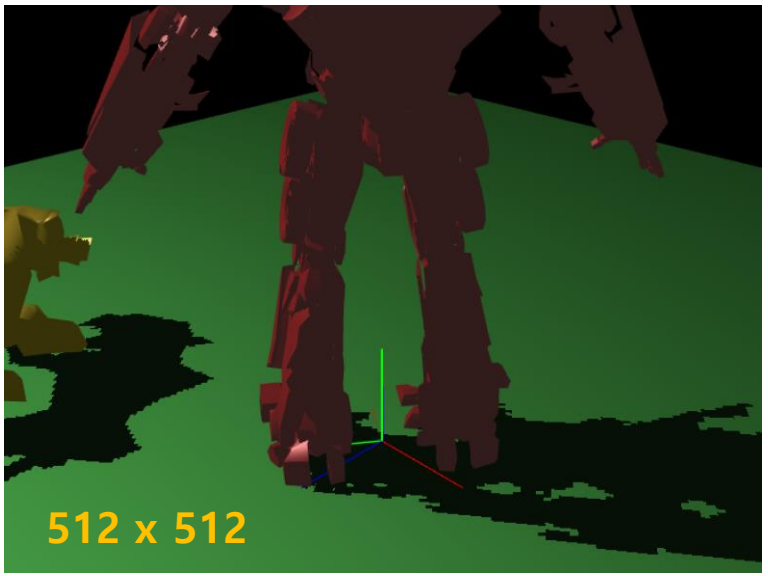
  glUseProgram(h_ShaderProgram_shadow);
    ... // Draw other objects.
  glUseProgram(0);
  glFinish();

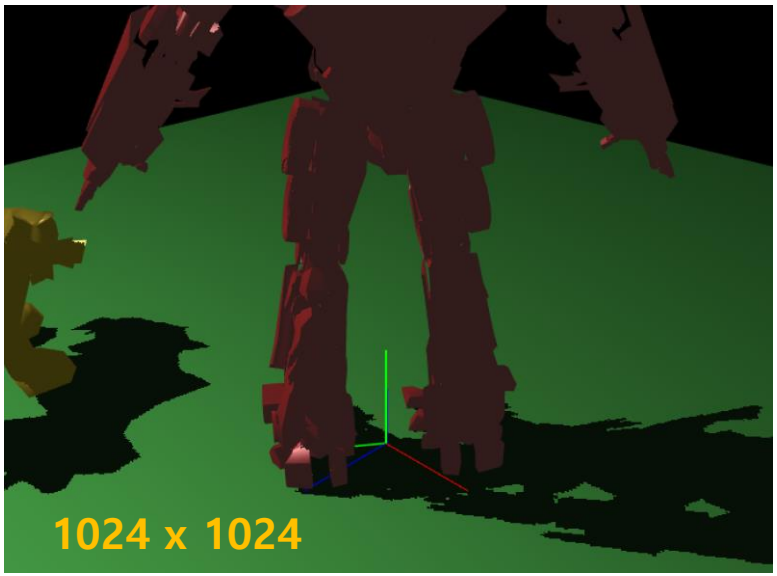
  glDisable(GL_POLYGON_OFFSET_FILL);
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

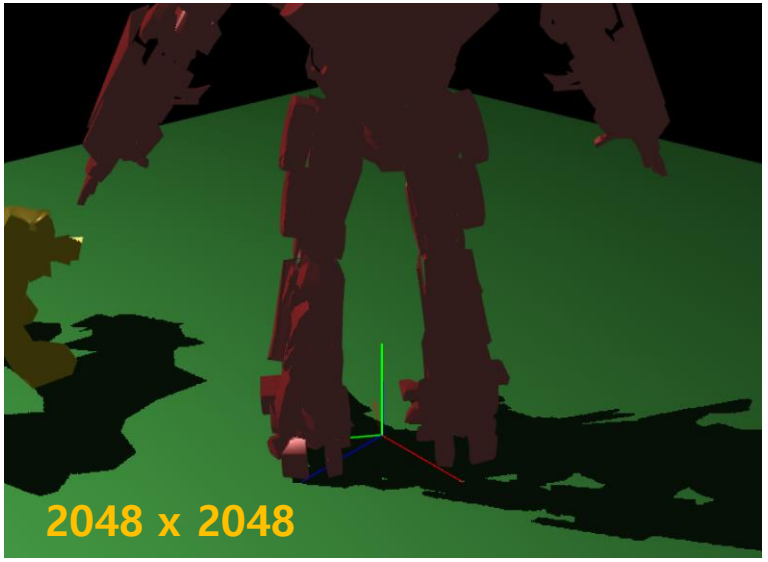


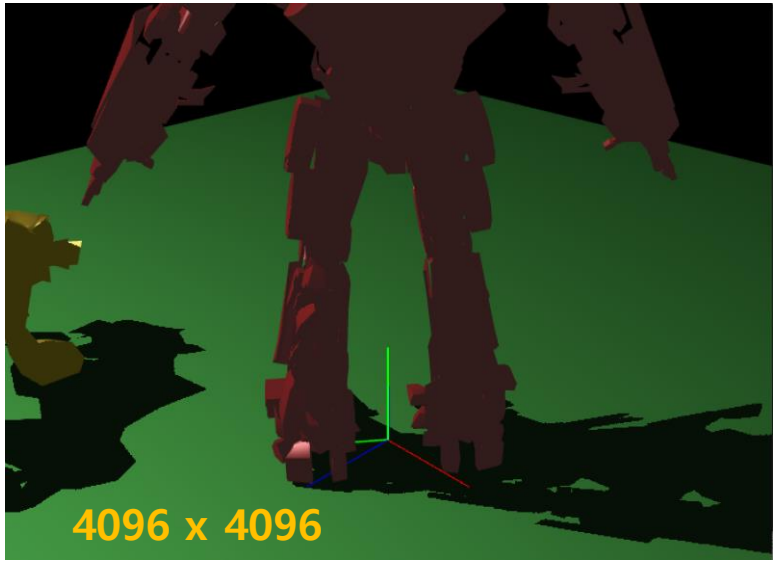
glPolygonOffset(10.0f, 10.0f);

Shadow Mapping with Depth Biasing







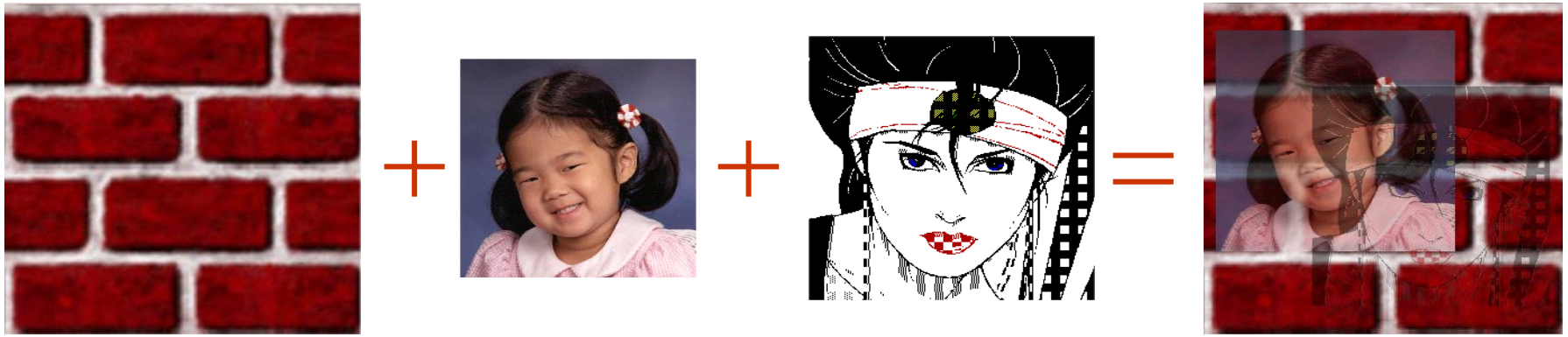


- Does simply increasing the shadow map resolution help?
- What are better ways to alleviate aliases in shadow mapping?

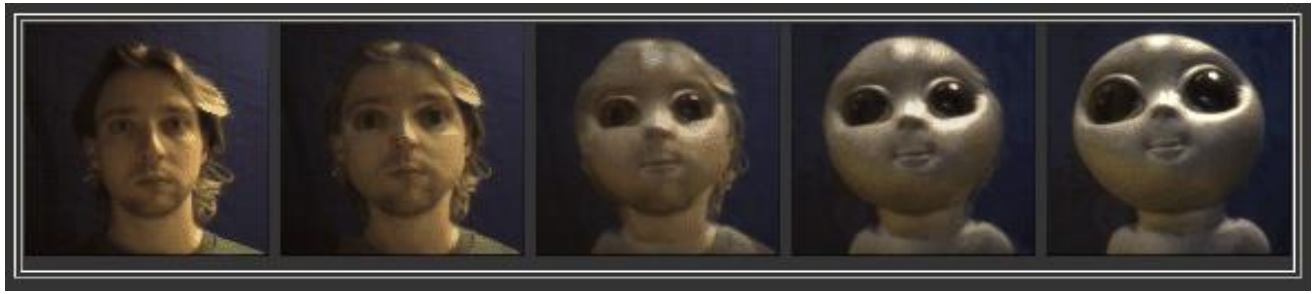
RGBA Color Model and Blending

이미지 합성과 RGBA 색깔 모델

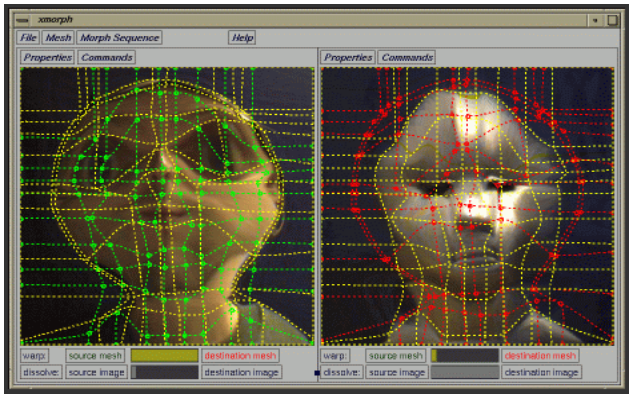
- 이미지 합성(image composition) 기법의 활용 예
  - Blue-Screen Matting
    ① Background 이미지 A 생성.
    ② 파란색의 Background를 가진 foreground 이미지 B 생성.
    ③ foreground 이미지 B 중 파란색이 아닌 부분을 background 이미지 A와 합성.
  - 다양한 이미지의 합성



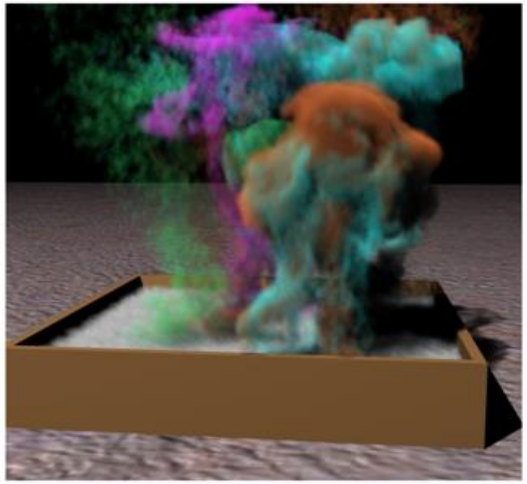
– 이미지 모핑(morphing)

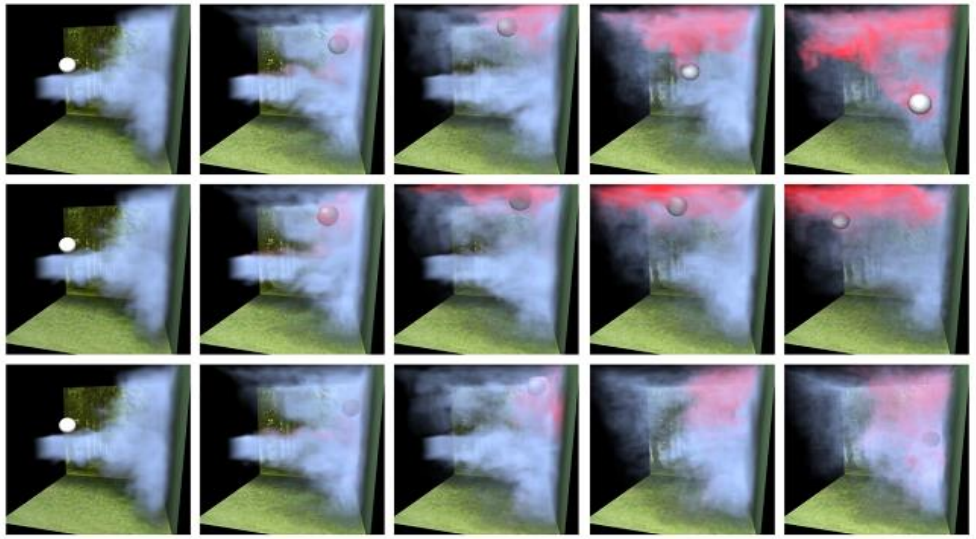


<Courtesy of XMORPH>



– 기타 등등

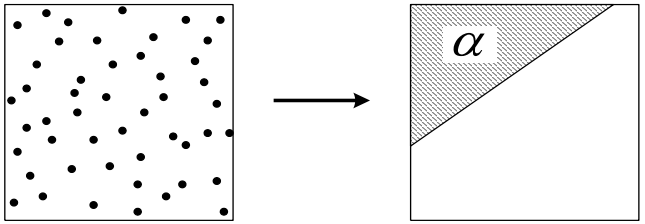




<Courtesy of Sogang Graphics Lab.>

알파 채널(Alpha Channel)

- 알파 채널
  - 이미지에 대한 다양한 조작을 위하여 화소당 R, G, B 채널뿐만 아니라 그 화소의 '속성'을 나타내주는 'A' 채널을 사용.
  - 각 화소는 (R, G, B, α)로 표현 → RGBA 색깔 모델
  - α 값은 응용하려는 문제에 따라 그 값의 의미가 조금씩 다르게 사용됨.
- 불투명도(opacity)로서의 α
  - (R, G, B)의 색깔을 가지는 미립자들이 랜덤하게 분포.
  - 이 미립자들이 화소의 뒤로부터 들어오는 빛을 α (0 <= α <= 1) 의 확률로 가로 막음.
    - α 비율만큼 이 미립자의 색깔로 보이고, 백그라운드로부터의 색깔이 1- α 의 비율만큼 보임.
    - α 비율만큼 이 미립자의 색깔로 보이고, 1- α 의 비율만큼은 아직 색깔이 정의되지 않은 상태.
  - 만약 α가 0이면, ...



$0 \le \alpha \le 1$

- 알파 채널 값은 다음과 같은 다양한 의미(용도)로서 사용이 됨.
  - 불투명도(opacity)
  - 매트 정보(matte information)
  - 포함 정보(coverage information)
  - 혼합 인자(mixing factor)
  - 기타 등등

알파 채널을 사용한 두 이미지의 합성 예

- 두 이미지 S와 D의 합성
  - 문제를 간단히 하기 위하여 두 이미지의 화소들이 서로 정렬되어 있다고 가정 → 두 이미지의 대응되는 화소끼리 어떻게 합성할 것인가?

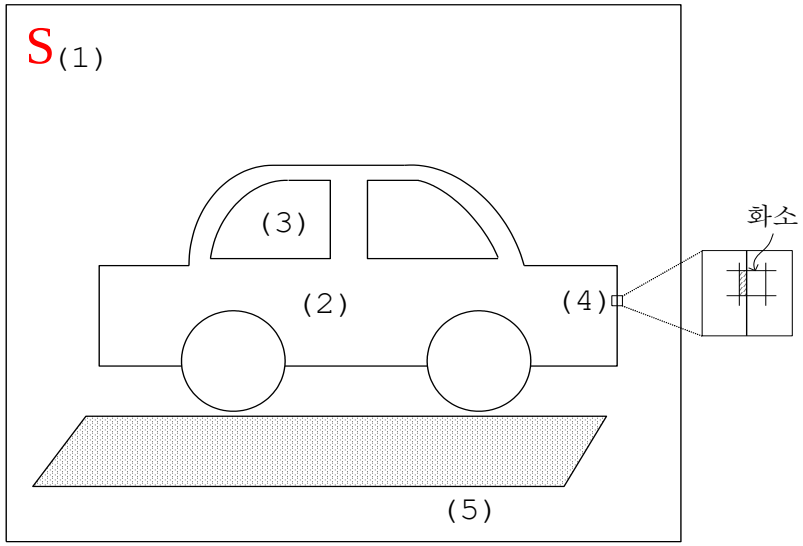
(1) (0, 0, 0, 0)

(2) (R_b, G_b, B_b, 1)

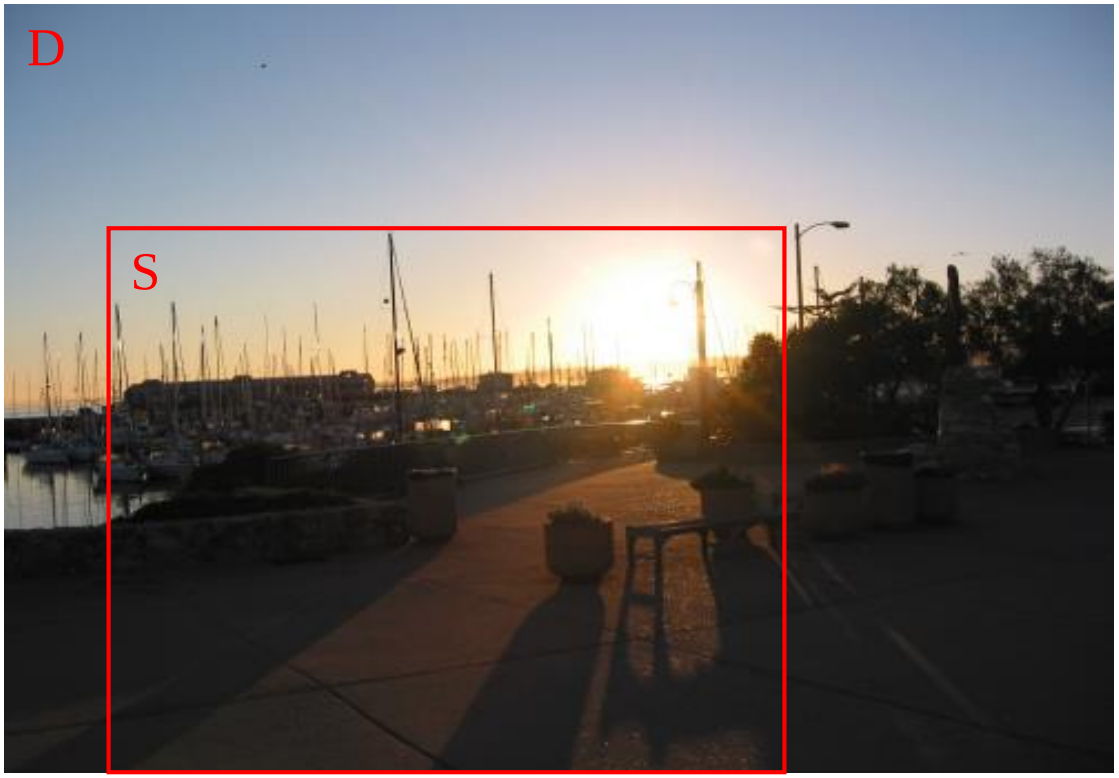
(3) (R_g, G_g, B_g, 0.15)

(4) (R_b, G_b, B_b, 0.25)

(5) (0.2, 0.2, 0.2, 0.3)

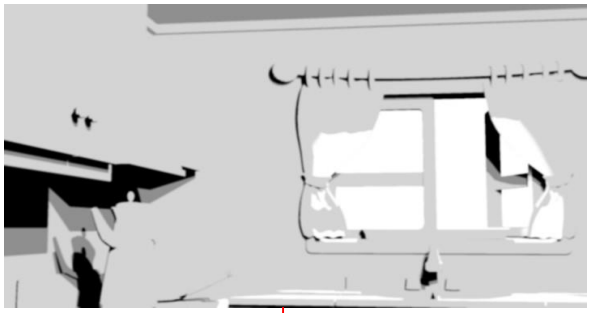


(R, G, B, 1)



알파 채널을 사용한 세 이미지의 합성 예

– 배경, 그림자, 캐릭터를 분리하여 렌더링 한 후 하나의 영상으로 합성



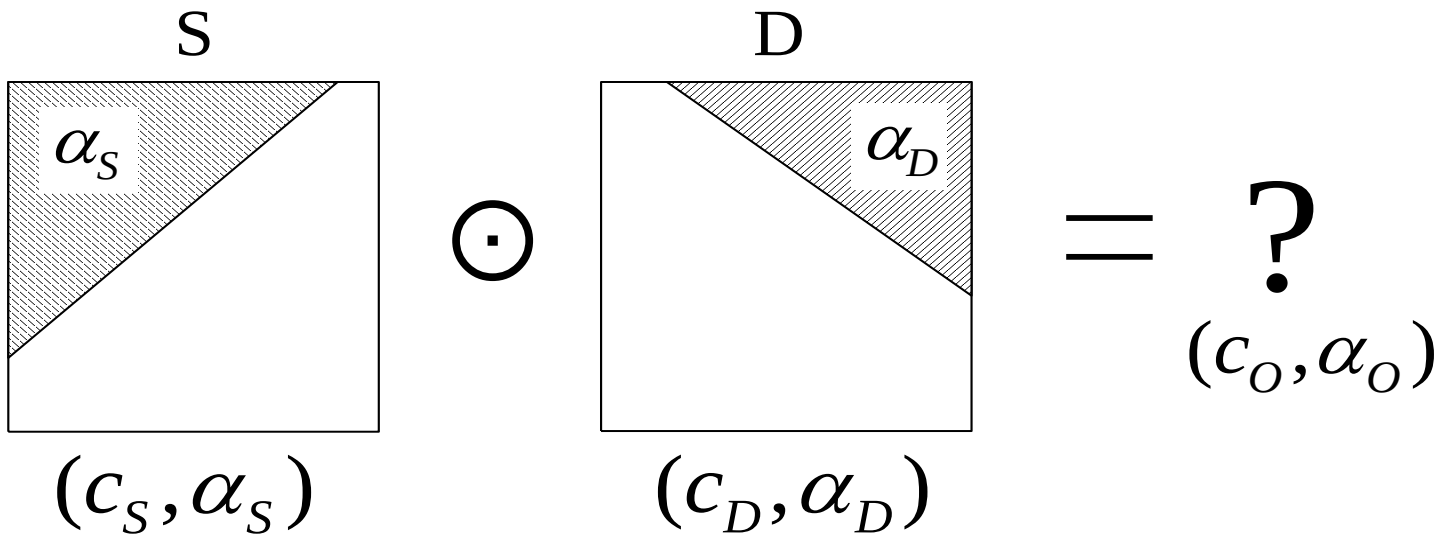
미리 곱한 색깔(Pre-multiplied Color)

- 미리 곱한 색깔 = 연합 색깔(associated color)
  - $(R, G, B, \alpha) \rightarrow (r, g, b, \alpha) = (\alpha R, \alpha G, \alpha B, \alpha)$
  - 미리 곱한 색깔 예
    - (0.8, 0.0, 0.0, 1.0)
    - (0.32, 0.0, 0.0, 0.4) = (0.8*0.4, 0.0*0.4, 0.0*0.4, 0.4)
    - (0.32, 0.32, 0.32, 0.32)
    - (0.0, 0.0, 0.0, 1.0)
    - (0.0, 0.0, 0.0, 0.5)
    - (0.0, 0.0, 0.0, 0.0)

☺ 미리 곱한 색깔은 합성 계산을 효율적으로 수행할 수 있게 해줌.

두 화소 S와 D의 합성

- T. Porter and T. Duff, "Compositing Digital Images," *SIGGRAPH '84*, pp.253-259, 1984.



$$c_S = (\alpha_S R_S, \alpha_S G_S, \alpha_S B_S) \qquad c_D = (\alpha_D R_D, \alpha_D G_D, \alpha_D B_D)$$

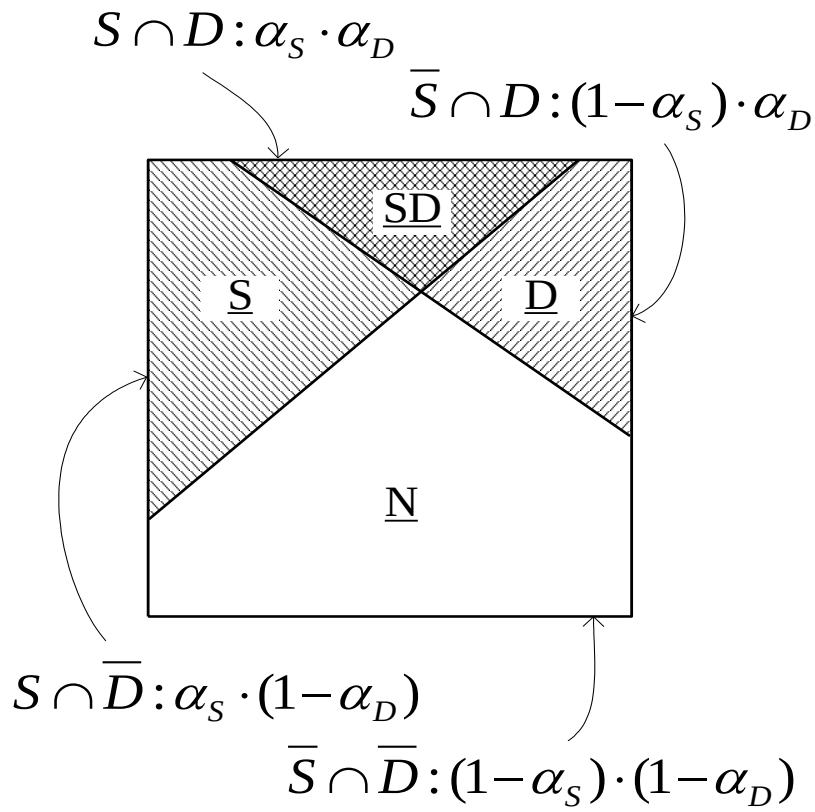
(C_S, α_S)
$C_S = (R_S, G_S, B_S)$

(C_D, α_D)
$C_D = (R_D, G_D, B_D)$

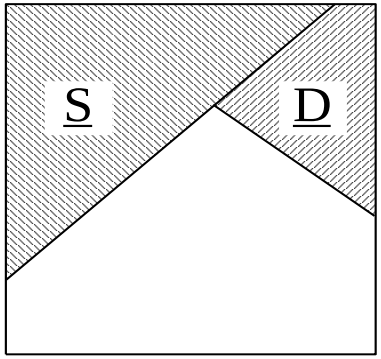
합성의 경우

- 두 화소를 겹칠 때 네 개의 영역으로 나눌 수 있음.
  - *S*와 *D* 입자 중 어떤 입자도 보일 수 있는 영역 SD
  - *S* 입자의 색깔만 보일 수 있는 영역 S
  - *D* 입자의 색깔만 보일 수 있는 영역 D
  - 어느 입자도 보이지 않는 영역 N

- 각 영역을 어떤 색깔로 보이게 할 것인가?
  - SD *S*, *D*, *N* (3가지)
  - S: *S*, *N* (2가지)
  - D: *D*, *N* (2가지)
  - N: *N* (1가지)
  - → 모두 12(=3*2*2*1)가지의 선택 가능

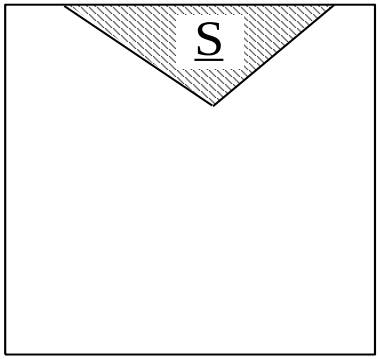


선택의 예



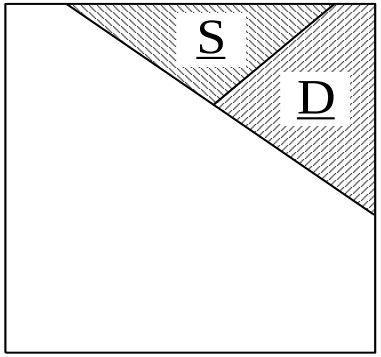
(S, S, D, N)

S **over** D



(S, N, N, N)

S **in** D



(S, N, D, N)

S **atop** D

12가지의 합성 연산

		F_S	F_D
clear	(N, N, N, N)	0	0
S	(S, S, N, N)	1	0
D	(D, N, D, N)	0	1
S **over** D	(S, S, D, N)	1	$1-\alpha_S$
D **over** S	(D, S, D, N)	$1-\alpha_D$	1
S **in** D	(S, N, N, N)	α_D	0
D **in** S	(D, N, N, N)	0	α_S
S **held-out-by** D	(N, S, N, N)	$1-\alpha_D$	0
D **held-out-by** S	(N, N, D, N)	0	$1-\alpha_S$
S **atop** D	(S, N, D, N)	α_D	$1-\alpha_S$
D **atop** S	(D, S, N, N)	$1-\alpha_D$	α_S
S **xor** D	(N, S, D, N)	$1-\alpha_D$	$1-\alpha_S$

합성 색깔의 계산 $(c_O, \alpha_O) \leftarrow (c_S, \alpha_S) \odot (c_D, \alpha_D)$

- F_S: S 화소에서 α_S의 비율로 존재하는 미립자들 중 결과 화소 O에 살아남는 미립자의 비율
- F_D: D 화소에서 α_D의 비율로 존재하는 미립자들 중 결과 화소 O에 살아남는 미립자의 비율

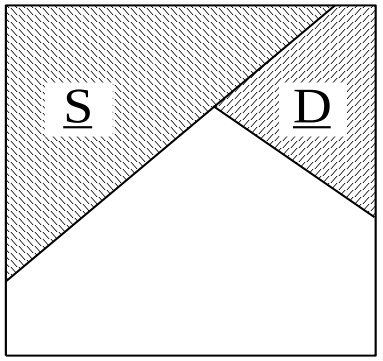
$$\alpha_O = \alpha_S F_S + \alpha_D F_D$$

$$\begin{aligned} c_O &= \alpha_O \cdot C_O = \alpha_O \cdot \frac{(\alpha_S F_S) \cdot C_S + (\alpha_D F_D) \cdot C_D}{\alpha_S F_S + \alpha_D F_D} \\ &= \alpha_O \cdot \frac{(\alpha_S F_S) \cdot C_S + (\alpha_D F_D) \cdot C_D}{\alpha_O} \\ &= \alpha_S F_S C_S + \alpha_D F_D C_D \\ &= \alpha_S F_S \frac{c_S}{\alpha_S} + \alpha_D F_D \frac{c_D}{\alpha_D} \\ &= c_S F_S + c_D F_D \end{aligned}$$

$$\begin{pmatrix} c_O \\ \\ \alpha_O \end{pmatrix} = F_S \begin{pmatrix} c_S \\ \\ \alpha_S \end{pmatrix} + F_D \begin{pmatrix} c_D \\ \\ \alpha_D \end{pmatrix}$$

- ☺ 미리 곱한 색깔을 사용하면 이미지 합성 연산을 간결하게 표현할수 있음.
- ☺ 원하는 합성 연산을 계산하기 위하여 그에 대응하는 F_S와 F_D의 값을 사용하면 됨.

합성 연산 예: S over D



(S, S, D, N)

S **over** D

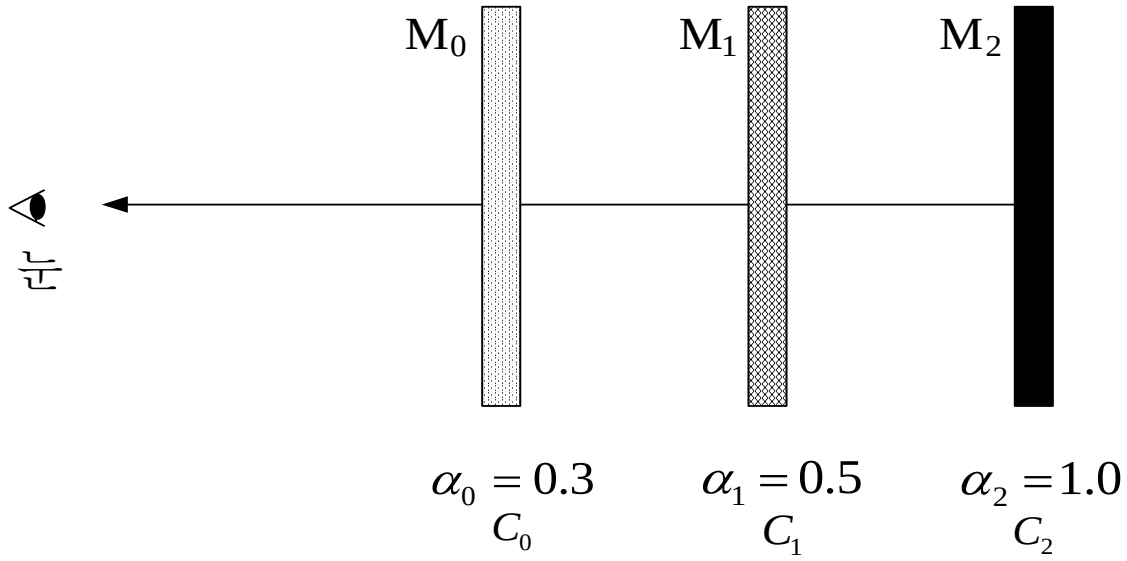
- $F_S = 1$, $F_D = 1 - \alpha_S$

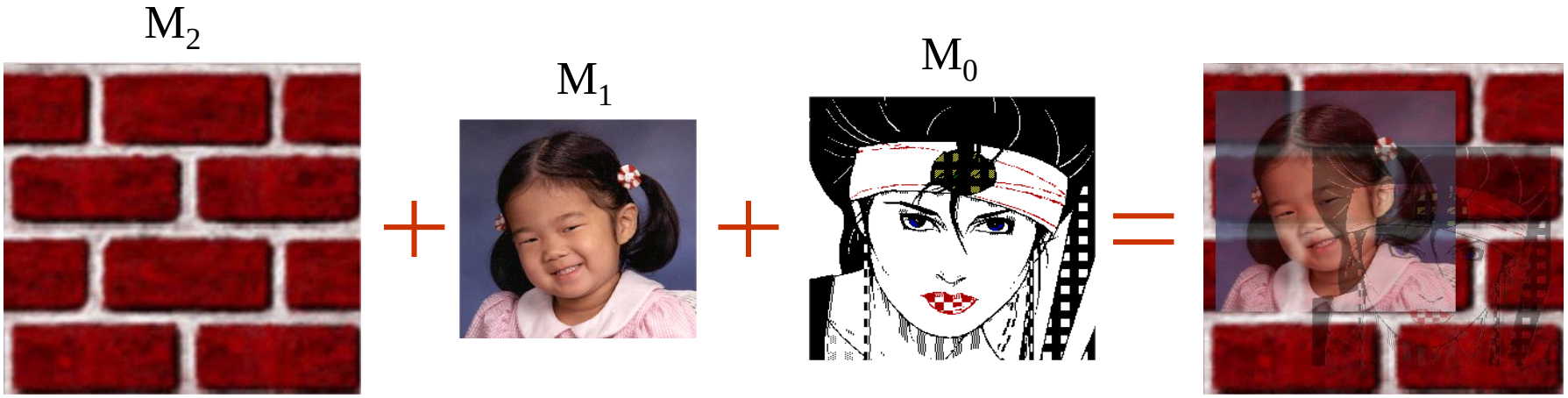
$$\begin{pmatrix} c_O \\ \alpha_O \end{pmatrix} = \begin{pmatrix} c_S \\ \alpha_S \end{pmatrix} + (1 - \alpha_S) \begin{pmatrix} c_D \\ \alpha_D \end{pmatrix}$$

$$= \begin{pmatrix} \alpha_S C_S \\ \alpha_S \end{pmatrix} + (1 - \alpha_S) \begin{pmatrix} \alpha_D C_D \\ \alpha_D \end{pmatrix}$$

☺ **over** 연산은 투명한 물체를 렌더링하거나, 안개 등의 기상 효과, 텍스춰의 혼합, 앤티 앨리어싱 기법들을 구현하는데 유용하게 사용이 됨.

over 연산의 예: (M_0 over M_1) over M_2





$$
\begin{aligned}
c_{01} &= \alpha_0 \cdot C_0 + (1-\alpha_0)\cdot\alpha_1\cdot C_1 \\
&= 0.3\cdot C_0 + (1-0.3)\cdot 0.5\cdot C_1 = 0.3C_0 + 0.35C_1 \\
\alpha_{01} &= \alpha_0 + (1-\alpha_0)\cdot\alpha_1 \\
&= 0.3 + (1-0.3)\cdot 0.5 = 0.65
\end{aligned}
$$

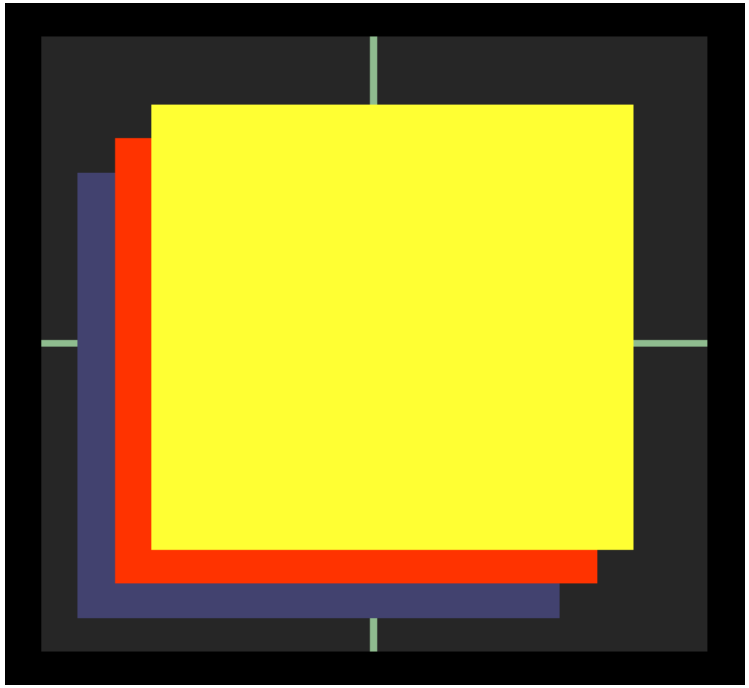
$$
c_{012} = 0.3C_0 + 0.35C_1 + 0.35C_2
$$

$$
\alpha_{012} = 1.0
$$

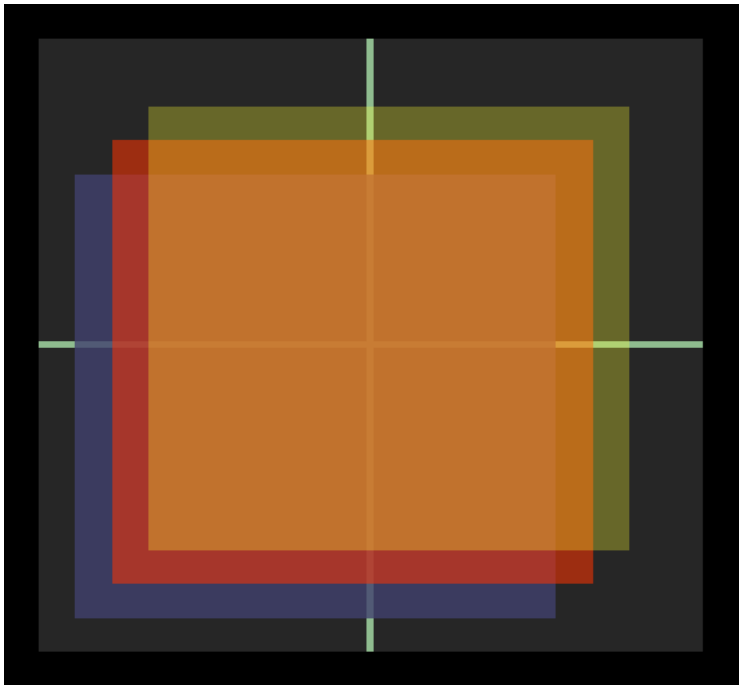
$$
\longrightarrow C_{012} = c_{012}
$$

☺ **over** 연산에는 결합 법칙이 성립함. 즉 전후 관계만 만족되면 어떤 연산자를 먼저 계산해도 결과는 동임함.

OpenGL을 이용한 over 연산



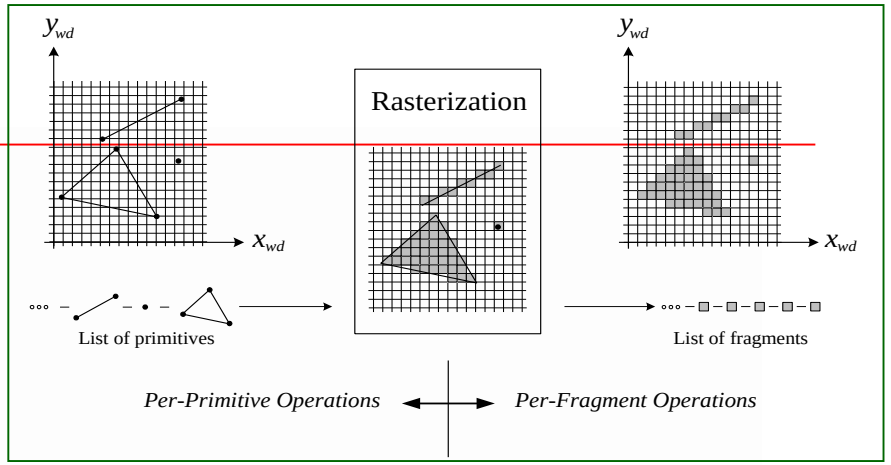
불투명한 물체



투명한 물체

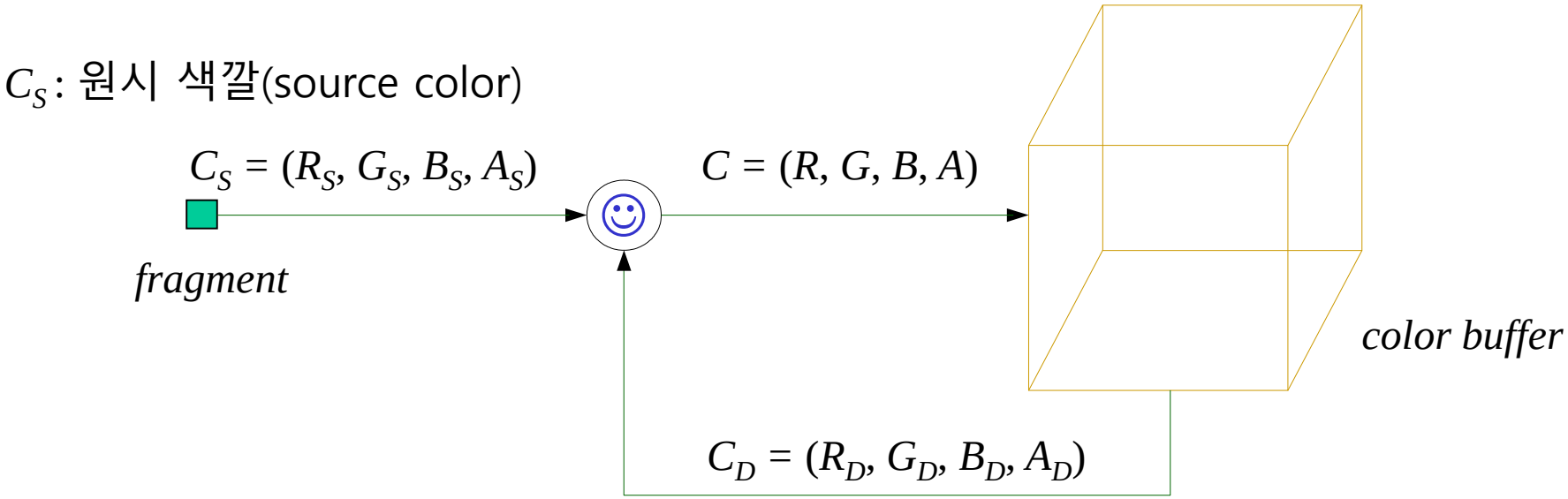
- 사각형과 선분을 앞에서 뒤로가면서(front-to-back), 또는 뒤에서 앞으로 오면서(back-to-front) 순서대로 **over** 연산을 사용하여 그림.
- OpenGL에서는 **over** 연산을 어떻게 구현할 것인가?

OpenGL에서의 알파 혼합



제4사분면

색깔의 혼합(Blending) 과정



Source: currently incoming fragment
Destination: existing fragment in buffers with the same address

(S_R, S_G, S_B, S_A) : 원시 인자(source factor)

(D_R, D_G, D_B, D_A) : 목적 인자(destination factor)

$$C = (R, G, B, A) = (R_S \bullet S_R + R_D \bullet D_R, G_S \bullet S_G + G_D \bullet D_G, B_S \bullet S_B + B_D \bullet D_B, A_S \bullet S_A + A_D \bullet D_A)$$

Void glBlendFunc(GLenum sfactor, GLenum dfactor); 함수

- sfactor와 dfactor를 통하여 혼합 방식을 결정.
- glEnable(GL_BLEND); 함수와 glDisable(GL_BLEND); 함수의 호출을 통하여 혼합 기능을 on/off 시킴.
- 디폴트 상태에는 off되어 있음.

GL Constant	Parameter	Value
GL_ZERO	sfactor & dfactor	(0, 0, 0, 0)
GL_ONE	sfactor & dfactor	(1, 1, 1, 1)
GL_DST_COLOR	sfactor	(DR, DG, DB, DA)
GL_ONE_MINUS_DST_COLOR	sfactor	(1-DR, 1-DG, 1-DB, 1-DA)
GL_SRC_COLOR	dfactor	(SR, SG, SB, SA)
GL_ONE_MINUS_SRC_COLOR	dfactor	(1-SR, 1-SG, 1-SB, 1-SA)
GL_SRC_ALPHA	sfactor & dfactor	(SA, SA, SA, SA)
GL_ONE_MINUS_SRC_ALPHA	sfactor & dfactor	(1-SA, 1-SA, 1-SA, 1-SA)
GL_DST_ALPHA	sfactor & dfactor	(DA, DA, DA, DA)
GL_ONE_MINUS_DST_ALPHA	sfactor & dfactor	(1-DA, 1-DA, 1-DA, 1-DA)

- 혼합의 예
  - **glBlendFunc(GL_ONE, GL_ZERO);**
    \/ Default *\/*
  - **glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);**
    \/ OVER: Back-to-Front *\/*
  - **glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);**
    \/ OVER: Front-to-Back *\/*
    - 😐 알파 채널이 필요함.

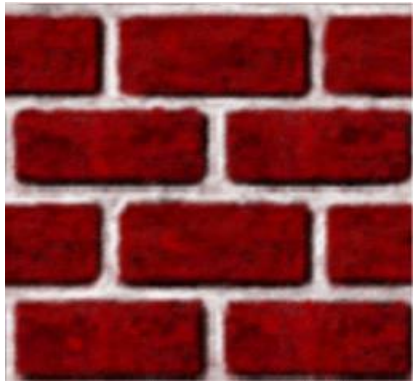


image 1

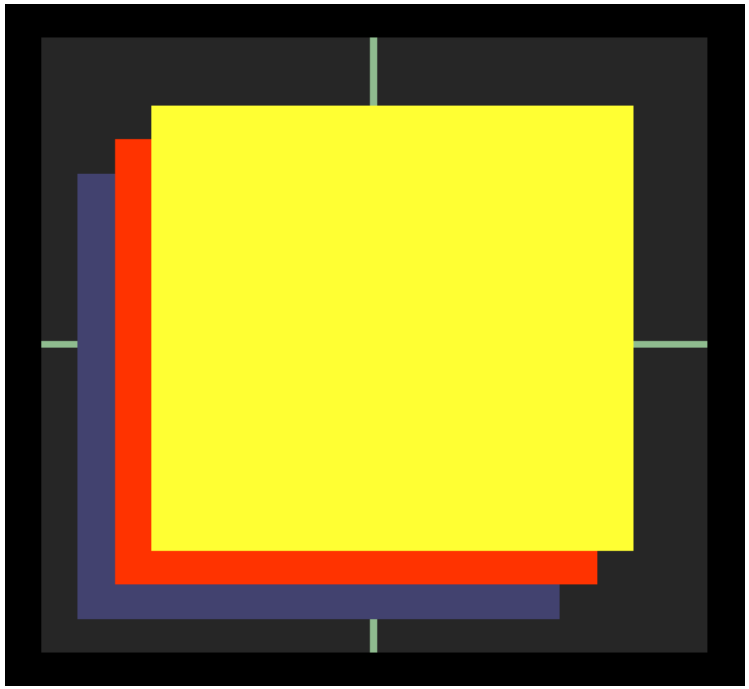
image 2

image 3

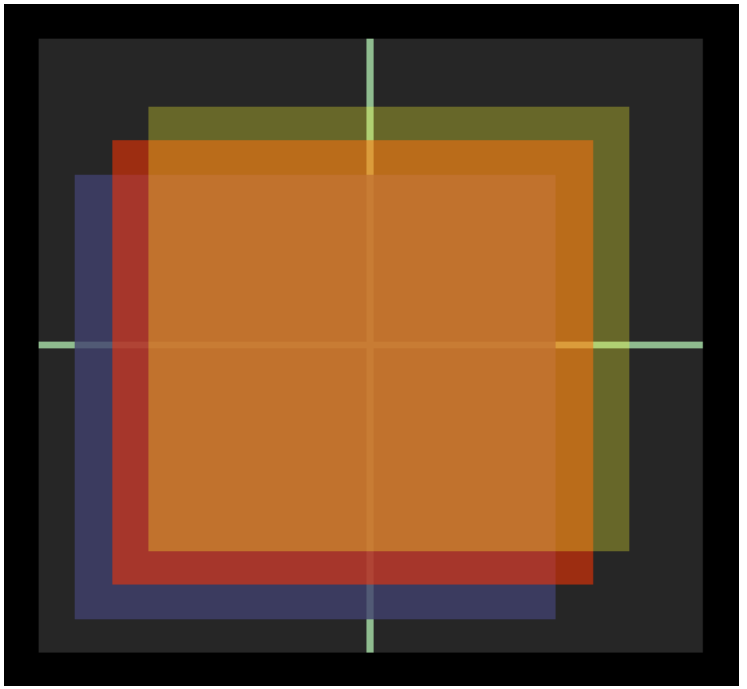
```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ZERO);
draw_image_1();
glBlendFunc(GL_SRC_ALPHA,
            GL_ONE_MINUS_SRC_ALPHA);
draw_image_2();
draw_image_3();
glDisable(GL_BLEND);
```

```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ZERO);
draw_image_1();
glBlendFunc(GL_ONE, GL_ONE);
draw_image_2();
glDisable(GL_BLEND);
```

OpenGL을 통한 합성 예 I (COMPAITIBILITY PROFILE)



불투명한 물체



투명한 물체

```
#include <stdio.h>
#include <stdlib.h>
#include <gl/glut.h>

int blendmode = 0, testmode = '0';

void draw_rectangle(GLfloat l, GLfloat r, GLfloat b, GLfloat t,
                    GLfloat R, GLfloat G, GLfloat B, GLfloat A) {
    glColor4f(R, G, B, A);
    glBegin(GL_QUADS);
      glVertex2f(l, b); glVertex2f(r, b); glVertex2f(r, t); glVertex2f(l, t);
    glEnd();
}

// 4: Grey Rectangle, 3: Green Lines, 2: Blue Rectangle,
// 1: Red Rectangle, 0: Yellow Rectangle

void test0(void) {
    /* Default mode when no blending is used. */
    glBlendFunc(GL_ONE, GL_ZERO);
    draw_rectangle(-4.5, 4.5, -4.5, 4.5, 0.15, 0.15, 0.15, 1.0); // 4

    glColor4f(0.561, 0.737, 0.561, 1.0); // 3
    glLineWidth(5.0);
    glBegin(GL_LINES);
      glVertex2f(-4.5, 0.0); glVertex2f(4.5, 0.0);
      glVertex2f(0.0, -4.5); glVertex2f(0.0, 4.5);
    glEnd();

    draw_rectangle(-4.0, 2.5, -4.0, 2.5, 0.259, 0.259, 0.435, 0.8); // 2
```

```
    draw_rectangle(-3.5, 3.0, -3.5, 3.0, 1.0, 0.2, 0.0, 0.55); // 1

    draw_rectangle(-3.0, 3.5, -3.0, 3.5, 1.0, 1.0, 0.2, 0.3); // 0
}

void test1(void) {
    /* OVER: Back-to-Front */
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    draw_rectangle(-4.5, 4.5, -4.5, 4.5, 0.15, 0.15, 0.15, 1.0); // 4

    glColor4f(0.561, 0.737, 0.561, 1.0); //3
    glLineWidth(5.0);
    glBegin(GL_LINES);
      glVertex2f(-4.5, 0.0); glVertex2f(4.5, 0.0);
      glVertex2f(0.0, -4.5); glVertex2f(0.0, 4.5);
    glEnd();

    draw_rectangle(-4.0, 2.5, -4.0, 2.5, 0.259, 0.259, 0.435, 0.8); // 2

    draw_rectangle(-3.5, 3.0, -3.5, 3.0, 1.0, 0.2, 0.0, 0.55); // 1

    draw_rectangle(-3.0, 3.5, -3.0, 3.5, 1.0, 1.0, 0.2, 0.3); // 0
}
```

```
void test2(void) {
    /* OVER: Front-to-Back */
    glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
    draw_rectangle(-3.0, 3.5, -3.0, 3.5, 0.3*1.0, 0.3*1.0, 0.3*0.2, 0.3); // 0

    draw_rectangle(-3.5, 3.0, -3.5, 3.0, 0.55*1.0, 0.55*0.2, 0.0, 0.55); // 1

    draw_rectangle(-4.0, 2.5, -4.0, 2.5, 0.8*0.259, 0.8*0.259, 0.8*0.435, 0.8); // 2

    glColor4f(1.0*0.561, 1.0*0.737, 1.0*0.561, 1.0); // 3
    glLineWidth(5.0);
    glBegin(GL_LINES);
      glVertex2f(-4.5, 0.0); glVertex2f(4.5, 0.0);
      glVertex2f(0.0, -4.5); glVertex2f(0.0, 4.5);
    glEnd();

    draw_rectangle(-4.5, 4.5, -4.5, 4.5, 1.0*0.15, 1.0*0.15, 1.0*0.15, 1.0); // 4

}
```

```
void display (void) {
    glClear(GL_COLOR_BUFFER_BIT);
    if (blendmode) glEnable(GL_BLEND);
    switch(testmode) {
    case '0':
          test0();  /* Default mode when no blending is used. */
          break;
    case '1':
          test1();  /* OVER: Back-to-Front */
          break;
    case '2':
          test2();  /* OVER: Front-to-Back */
          break;
    }
    if (blendmode) glDisable(GL_BLEND);
    glFlush();
}

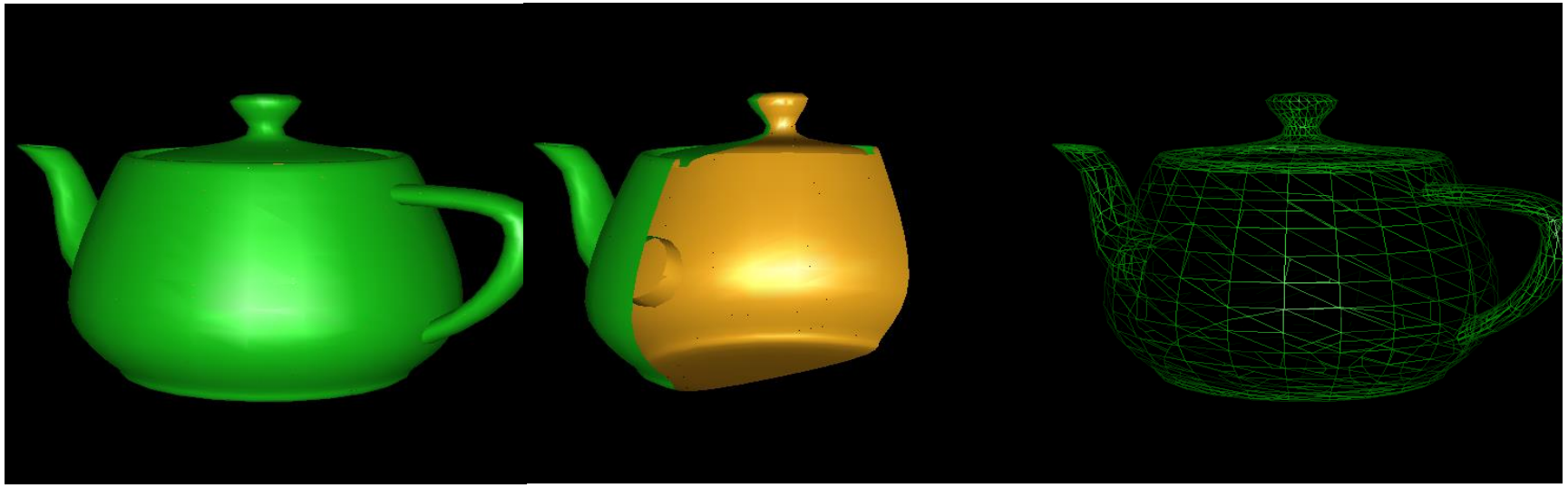
void keyboard (unsigned char key, int x, int y) {
    switch (key) {
    case '0': case '1': case '2': testmode = key;
          glutPostRedisplay();
          break;
    case 'b': blendmode = 1; glutPostRedisplay();
          break;
    case 'n': blendmode = 0; glutPostRedisplay();
          break;
    case 'q': exit(-1);
          break;
    }
}
```

```
void init_OpenGL() {
    int r, g, b, a;

    glGetIntegerv(GL_RED_BITS, &r);
    glGetIntegerv(GL_GREEN_BITS, &g);
    glGetIntegerv(GL_BLUE_BITS, &b);
    glGetIntegerv(GL_ALPHA_BITS, &a);
    printf("*** BIT PLANES: (R, G, B, A) = (%d, %d, %d, %d)\n",
          r, g, b, a);
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-5.0, 5.0, -5.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA);
    glutInitWindowSize(512, 512);
    glutCreateWindow("Blending Example");
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    init_OpenGL();
    glutMainLoop();
}
```

투명한 물체의 렌더링



- 😐 다면체 모델의 각 다각형을 '시점을 기준으로 하여 정렬'한 후, 앞에서 뒤로 가면서, 또는 뒤에서 앞으로 오면서 적절히 **over** 연산을 통하여 혼합.
- ☹ BSPT(Binary Space Partitioning Tree) 등의 자료 구조를 사용하여 다각형들을 정렬하여야 함.

기타 이미지 합성 연산

- 일진 합성 연산(unary composition operations)
  - $\text{darken}(S, \phi) = (\phi \cdot r_s, \phi \cdot g_s, \phi \cdot b_s, \alpha_s),\ (0 \le \phi \le 1)$
  - $\text{dissolve}(S, \sigma) = (\sigma \cdot r_s, \sigma \cdot g_s, \sigma \cdot b_s, \sigma \cdot \alpha_s),\ (0 \le \sigma \le 1)$
  - $\text{opaque}(S, \omega) = (r_s, g_s, b_s, \omega \cdot \alpha_s)$
- 이진 합성 연산(binary composition operations)
  - **clear, S, D, over, in, held-out-by, atop, xor**
  - S **plus** D = $(r_S + r_D,\ g_S + g_D,\ b_S + b_D,\ \alpha_S + \alpha_D)$

(주로 $\alpha_S + \alpha_D = 1$인 경우 유용함.)

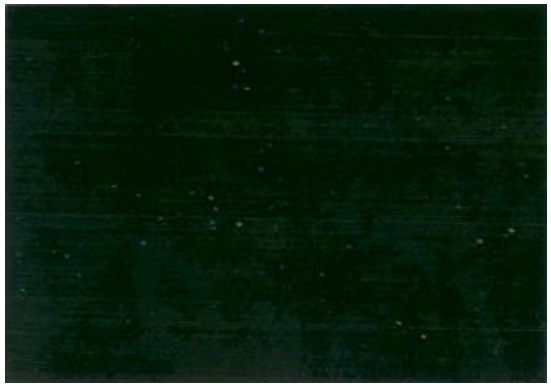
사용 예: O = (**dissolve**(S, t)) **plus** (**dissolve**(D, 1-t)) $(0 \le t \le 1)$전이 효과

이미지 합성 예

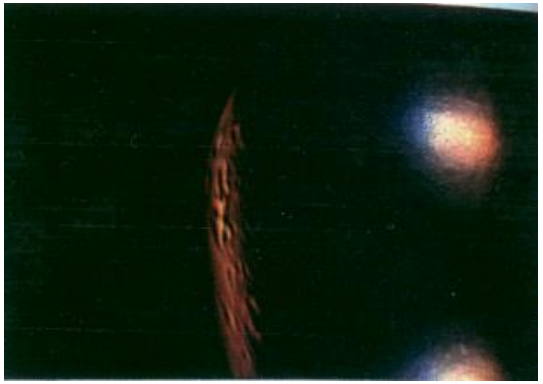
- `D = (A in C) over B;`

- ```
  Foreground = FrgdGrass over Rock over Fence over Shadow
               over BkgdGrass;
  GlossyRoad = Puddle over (PostReflection atop (PlantReflection
               atop Road));
  Hillside = Plant over GlossyRoad over Hill;
  Background = Rainbow plus Darkbow over Mountains over Sky;
  Image1 = Foreground over Hillside over Background;
  ```
  ```

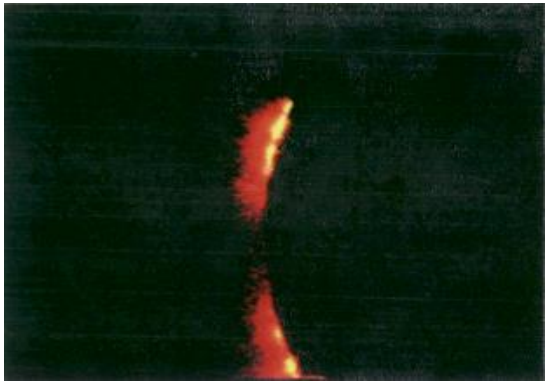
- `Image2 = (Ffire` **`plus`** `(Bfire` **`out`** `Planet))` **`over darken`**`(Planet, .8)`
  `over` `Stars;`



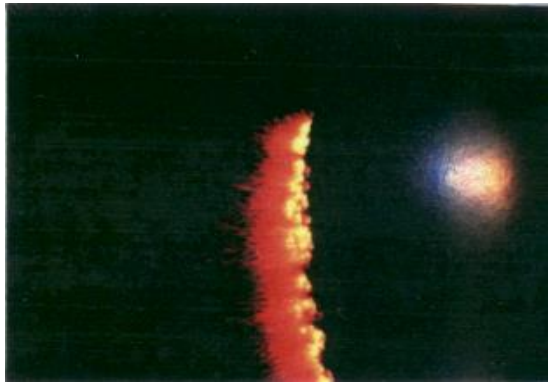
Stars



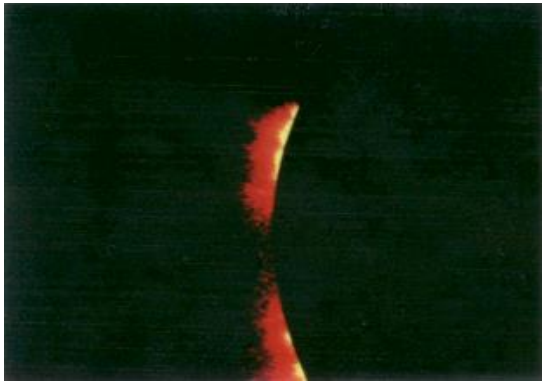
Planet



BFire



FFire



Bfire **out** Planet

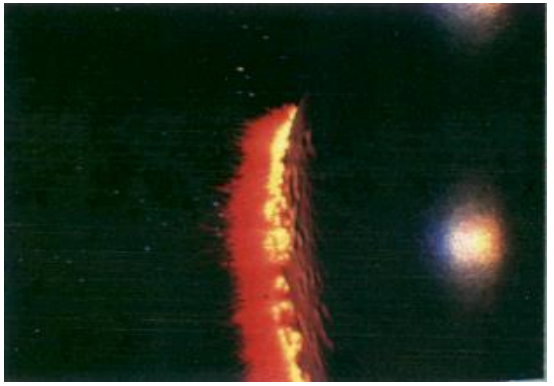


Image2

<Courtesy of Porter & Duff>

Composition of Sampled Data

$$\begin{pmatrix} c_O \\ \alpha_O \end{pmatrix} = \begin{pmatrix} c_S \\ \alpha_S \end{pmatrix} + (1-\alpha_S)\begin{pmatrix} c_D \\ \alpha_D \end{pmatrix}$$

$$= \begin{pmatrix} \alpha_S C_S \\ \alpha_S \end{pmatrix} + (1-\alpha_S)\begin{pmatrix} \alpha_D C_D \\ \alpha_D \end{pmatrix}$$

- Back-to-Front Composition

$$\begin{pmatrix} c_O \\ \alpha_O \end{pmatrix} = \begin{pmatrix} c_S \\ \alpha_S \end{pmatrix} + (1-\alpha_S)\begin{pmatrix} c_D \\ 1 \end{pmatrix}$$

$$= 1 \cdot \begin{pmatrix} \alpha_S C_S \\ \alpha_S \end{pmatrix} + (1-\alpha_S) \cdot \begin{pmatrix} C_D \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} C_O \\ 1 \end{pmatrix}$$

```
glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
```

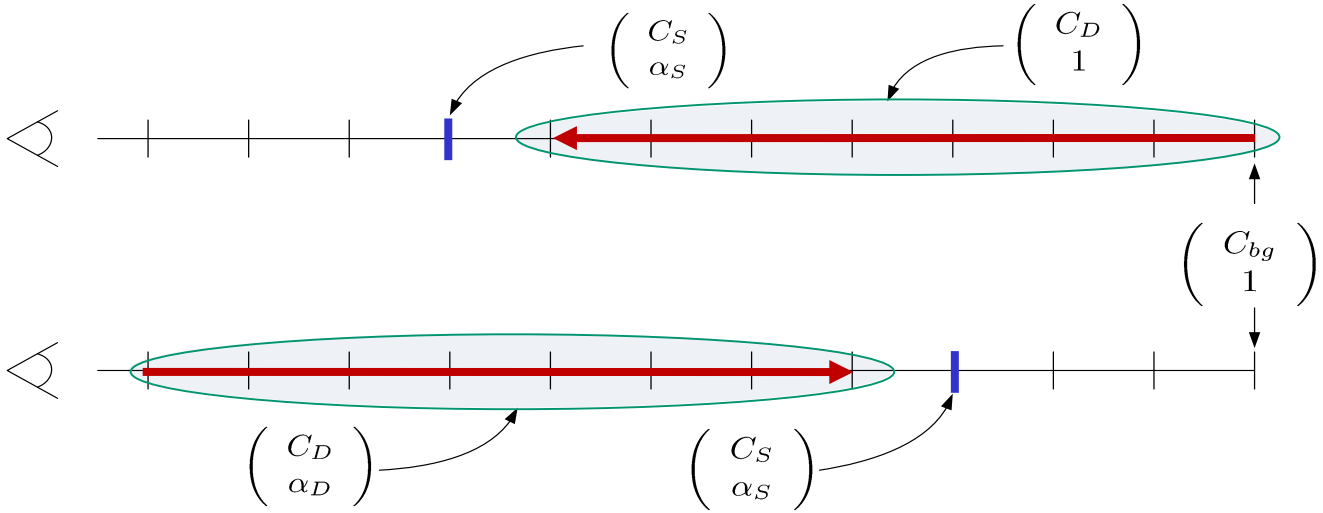
- Front-to-Back Composition

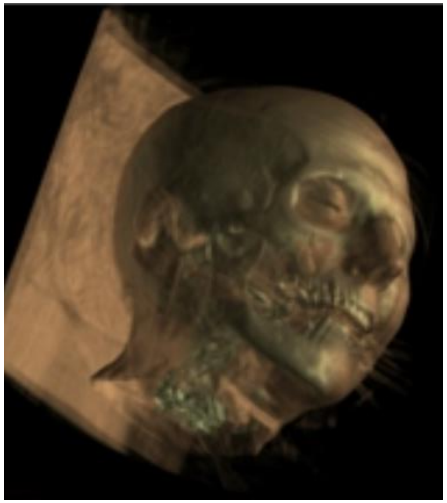
$$\begin{pmatrix} c_O \\ \alpha_O \end{pmatrix} = \begin{pmatrix} \alpha_D C_D \\ \alpha_D \end{pmatrix} + (1-\alpha_D)\begin{pmatrix} \alpha_S C_S \\ \alpha_S \end{pmatrix}$$

$$= 1 \cdot \begin{pmatrix} \alpha_D C_D \\ \alpha_D \end{pmatrix} + (1-\alpha_D) \cdot \begin{pmatrix} \alpha_S C_S \\ \alpha_S \end{pmatrix}$$

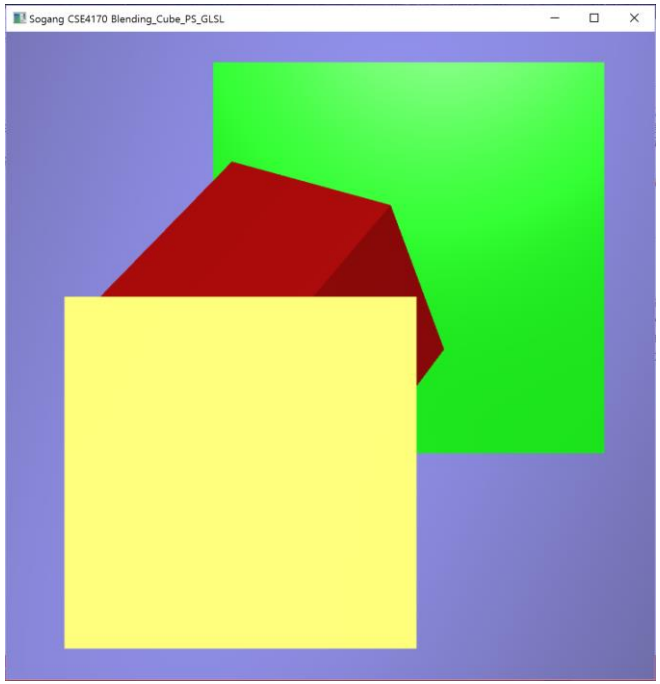
$$= \begin{pmatrix} \alpha_O C_O \\ \alpha_O \end{pmatrix}$$

```
glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
```

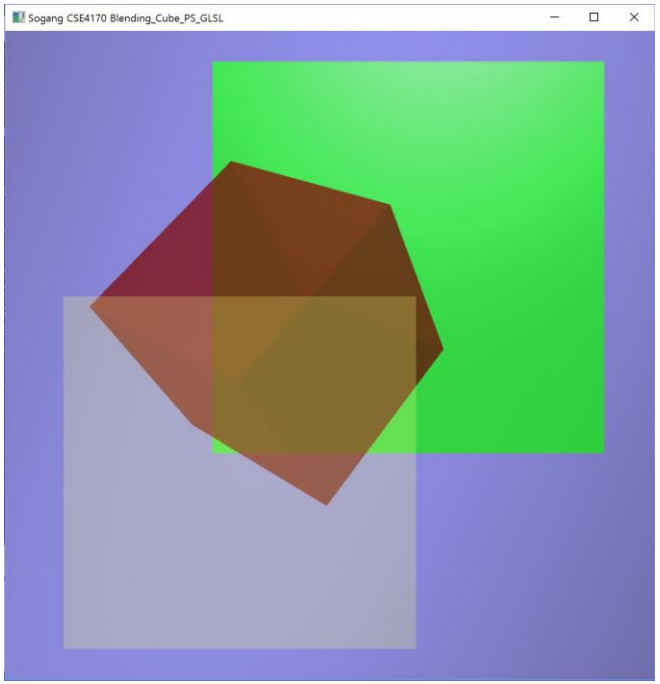




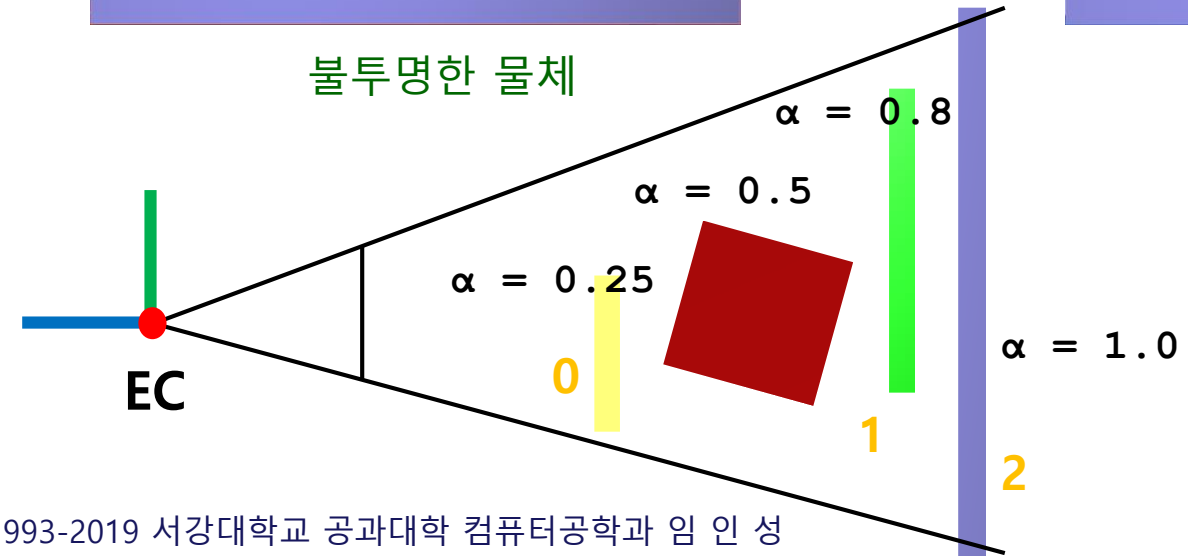
OpenGL을 통한 합성 예 II (CORE PROFILE)



불투명한 물체



투명한 물체



```
int flag_blend_mode = 0;     // 0: blending off, 1: blending on
int flag_back_to_front = 1; // 0: front-to-back drawing, 1: back-to-front drawing

void display(void) { // display callback
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  if (flag_blend_mode) {
    glEnable(GL_BLEND); glDisable(GL_DEPTH_TEST);
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // alpha must initially be zero!
  }
  else glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

  if (flag_back_to_front)
    draw_objects_Back_to_Front();
  else
    draw_objects_Front_to_Back();

  glutSwapBuffers();

  if (flag_blend_mode) {
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
  }
}
```

```
void draw_objects_Back_to_Front(void) {
  glUseProgram(h_ShaderProgram_PS);
  if (flag_blend_mode) {
    glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA); // 수정
    glUniform1i(loc_u_flag_blending, 1); // Back_to_Front
  }
  else
    glUniform1i(loc_u_flag_blending, 0);

  // draw rectangle 2
  set_material_rectangle(2);
  glUniform1f(loc_u_fragment_alpha, rectangle_alpha[2]); // alpha = 1.0
  ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-50.0f, -50.0f, -50.0f));
  ModelViewMatrix = glm::scale(ModelViewMatrix, glm::vec3(100.0f, 100.0f, 100.0f));
  ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
  ModelViewMatrixInvTrans = glm::inverseTranspose(glm::mat3(ModelViewMatrix));
  glUniformMatrix4fv(loc_ModelViewProjectionMatrix_PS, 1, GL_FALSE,
                                                &ModelViewProjectionMatrix[0][0]);
  glUniformMatrix4fv(loc_ModelViewMatrix_PS, 1, GL_FALSE, &ModelViewMatrix[0][0]);
  glUniformMatrix3fv(loc_ModelViewMatrixInvTrans_PS, 1, GL_FALSE, &ModelViewMatrixInvTrans[0][0]);
  draw_rectangle();

  // draw rectangle 1
  set_material_rectangle(1);
  glUniform1f(loc_u_fragment_alpha, rectangle_alpha[1]);  // alpha = 0.8
  ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-6.0f, -5.0f, -40.0f));
```

In case of blending off, the blending factors are ignored.
Or, you could set them with **glBlendFunc(GL_ONE, GL_ZERO);**

Draw object in EC!

```
ModelViewMatrix = glm::scale(ModelViewMatrix, glm::vec3(20.0f, 20.0f, 20.0f));
ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
ModelViewMatrixInvTrans = glm::inverseTranspose(glm::mat3(ModelViewMatrix));
glUniformMatrix4fv(loc_ModelViewProjectionMatrix_PS, 1, GL_FALSE,
                                                &ModelViewProjectionMatrix[0][0]);
glUniformMatrix4fv(loc_ModelViewMatrix_PS, 1, GL_FALSE, &ModelViewMatrix[0][0]);
glUniformMatrix3fv(loc_ModelViewMatrixInvTrans_PS, 1, GL_FALSE, &ModelViewMatrixInvTrans[0][0]);
draw_rectangle();

// draw cube
glEnable(GL_CULL_FACE);
set_material_cube();
glUniform1f(loc_u_fragment_alpha, cube_alpha);  // alpha = 0.5
ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-2.0f, 1.0f, -30.0f));
ModelViewMatrix = glm::scale(ModelViewMatrix, glm::vec3(4.0f, 4.0f, 4.0f));
ModelViewMatrix = glm::rotate(ModelViewMatrix, rotation_angle_cube, glm::vec3(1.0f, 1.0f, 1.0f));
ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
ModelViewMatrixInvTrans = glm::inverseTranspose(glm::mat3(ModelViewMatrix));
glUniformMatrix4fv(loc_ModelViewProjectionMatrix_PS, 1, GL_FALSE,
                                                    &ModelViewProjectionMatrix[0][0]);
glUniformMatrix4fv(loc_ModelViewMatrix_PS, 1, GL_FALSE, &ModelViewMatrix[0][0]);
glUniformMatrix3fv(loc_ModelViewMatrixInvTrans_PS, 1, GL_FALSE, &ModelViewMatrixInvTrans[0][0]);
glCullFace(GL_FRONT);
draw_cube();
glCullFace(GL_BACK);
draw_cube();
glDisable(GL_CULL_FACE);
```

Draw cube twice!!!

```
void draw_cube(void) {
  glFrontFace(GL_CCW);
    …
```

```
  // draw rectangle 0
  set_material_rectangle(0);
  glUniform1f(loc_u_fragment_alpha, rectangle_alpha[0]);  // alpha = 0.25
  ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-6.8f, -7.5f, -20.0f));
  ModelViewMatrix = glm::scale(ModelViewMatrix, glm::vec3(9.0f, 9.0f, 9.0f));
  ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
  ModelViewMatrixInvTrans = glm::inverseTranspose(glm::mat3(ModelViewMatrix));
  glUniformMatrix4fv(loc_ModelViewProjectionMatrix_PS, 1, GL_FALSE,
                                                  &ModelViewProjectionMatrix[0][0]);
  glUniformMatrix4fv(loc_ModelViewMatrix_PS, 1, GL_FALSE, &ModelViewMatrix[0][0]);
  glUniformMatrix3fv(loc_ModelViewMatrixInvTrans_PS, 1, GL_FALSE, &ModelViewMatrixInvTrans[0][0]);
  draw_rectangle();

  glUseProgram(0);
}
```

```
void main(void) {
  vec4 lighting_color = lighting_equation(v_position_EC,
                                          normalize(v_normal_EC));

  if (u_flag_blending == 0) {
    final_color = vec4(lighting_color.rgb, 1.0f);
  }
  else {
    final_color = vec4(u_fragment_alpha * lighting_color.rgb,
                                                u_fragment_alpha);
  }
}
```

Fragment Shader

```
void draw_objects_Front_to_Back(void) {
  glUseProgram(h_ShaderProgram_PS);
  if (flag_blend_mode) {
    glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);
    glUniform1i(loc_u_flag_blending, 2); // Front_to_Back
  }
  else
    glUniform1i(loc_u_flag_blending, 0);

  // draw rectangle 0
  set_material_rectangle(0);
  glUniform1f(loc_u_fragment_alpha, rectangle_alpha[0]); // alpha = 0.25
  ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-6.8f, -7.5f, -20.0f));
    …
  draw_rectangle();

  // draw cube
  glEnable(GL_CULL_FACE);
  set_material_cube();
  glUniform1f(loc_u_fragment_alpha, cube_alpha);  // alpha = 0.5
  ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-2.0f, 1.0f, -30.0f));
   …
  glCullFace(GL_BACK);
  draw_cube();
  glCullFace(GL_FRONT);
  draw_cube();
  glDisable(GL_CULL_FACE);
```

Be careful with the drawing order!!!

```
  // draw rectangle 1
  set_material_rectangle(1);
  glUniform1f(loc_u_fragment_alpha, rectangle_alpha[1]); // alpha = 0.8
  ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-6.0f, -5.0f, -40.0f));
    …
  draw_rectangle();

  // draw rectangle 2
  set_material_rectangle(2);
  glUniform1f(loc_u_fragment_alpha, rectangle_alpha[2]); // alpha = 1.0
  ModelViewMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-50.0f, -50.0f, -50.0f));
    …
  draw_rectangle();

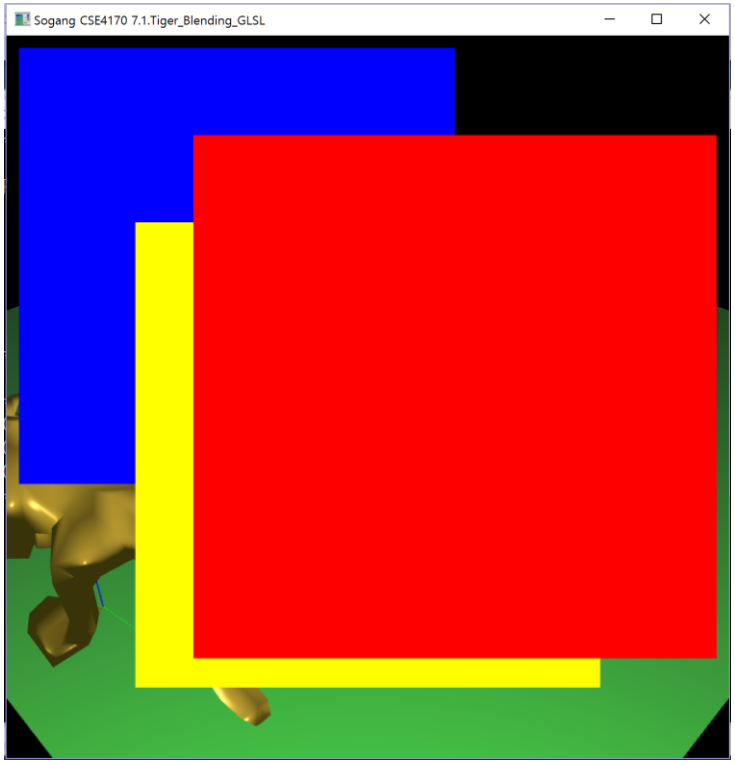
  glUseProgram(0);
}
```

```
void main(void) {
  vec4 lighting_color = lighting_equation(v_position_EC,
                                          normalize(v_normal_EC));

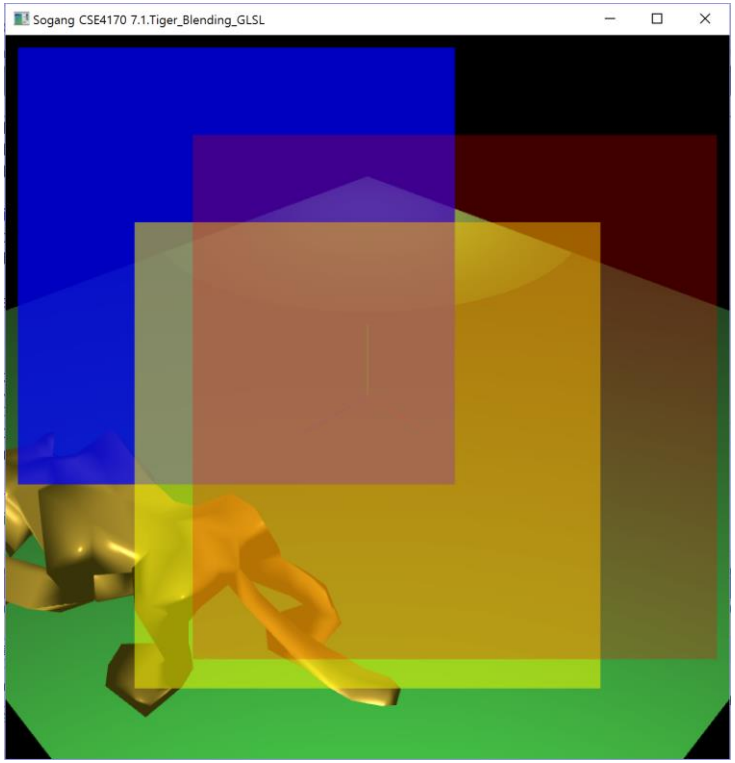
  if (u_flag_blending == 0) {
    final_color = vec4(lighting_color.rgb, 1.0f);
  }
  else {
    final_color = vec4(u_fragment_alpha * lighting_color.rgb,
                                          u_fragment_alpha);
  }
}
```

Fragment Shader

OpenGL을 통한 합성 예 III (CORE PROFILE)



불투명한 필터



투명한 필터