

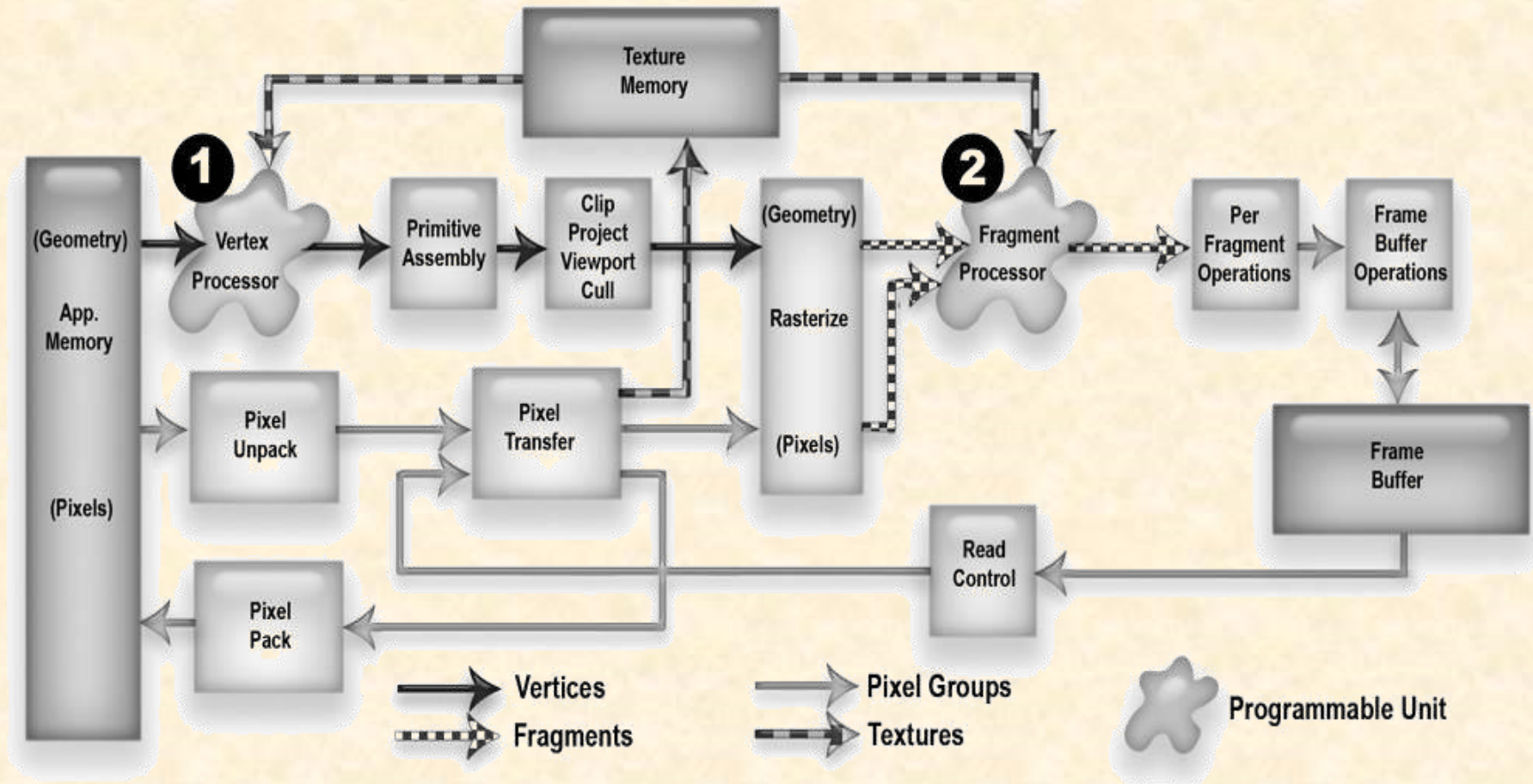
Gràfics 3D

Pintat de geometria, Shaders i Il·luminació

SGI – Màster MEI – curs 18/18 Q1

Pipeline programmable

Pipeline programmable



Pipeline programable

Vertex processor: part de la GPU capaç d'executar un programa per cada vèrtex.

Fragment processor: part de la GPU capaç d'executar un programa per cada fragment.

Shader: codi font d'un programa (o part) per la GPU

- Vertex shader, fragment shader, geometry shader

Program: executable d'un programa per la GPU

- Vertex program, fragment program, geometry program

Pipeline programable

Quan s'activa un **vertex program**, en lloc d'executar-se les operacions **per-vèrtex** prefixades d'OpenGL, s'executa el vertex program.

Quan s'activa un **fragment program**, en lloc d'executar-se les operacions **per-fragment** prefixades d'OpenGL, s'executa el fragment program.

Normalment el vertex/fragment program més senzill haurà de reproduir part de la funcionalitat fixa d'OpenGL.

Pintat de geometria en WebGL

Vertex Buffer Objects

- Permet emmagatzemar vertex arrays en la memòria del servidor (usualment tarja gràfica - memòria ràpida-) amb una eficient transmissió d'informació.
- Poden ser modificats.
- VBO crea un “buffer object” amb tota la informació associada als vèrtexs.

Vertex Buffer Objects

Utilització

- *glGenBuffers (n, *ids)*: Creació dels identificadors
- *glBindBuffer (target, buffid)*: Especifica el *buffer* actiu
target = GL_ARRAY_BUFFER o GL_ELEMENT_ARRAY_BUFFER
- *glBufferData (target, size, *data, usage)*:
Copia les dades al servidor i s'informa
de l'ús que se'n donarà (estàtic, dinàmic, molt dinàmic)

Per a pintar:

- *glBindBuffer (target, buffid)*: Activa *buffer*
- *glDrawArrays (mode, first, count)*: Pinta les dades
- *glDrawElements (mode, count, type, *indices)*:
Pinta amb indexs

Vertex Buffer Objects

Per a indicar a la GPU l'atribut dels vèrtexs a tenir en compte:

```
void glVertexAttribPointer (GLuint index, GLint size, GLenum type,  
                             GLboolean normalized, GLsizei stride, const GLvoid *pointer);
```

Indica les característiques de l'atribut del vèrtex identificat per *index*

index : nom de l'atribut

size : nombre de components que componen l'atribut

type : tipus de cada component (GL_FLOAT, GL_INT, ...)

normalized : indica si els valors de cada component s'han de normalitzar

stride : offset en bytes entre dos atributs consecutius (normalment 0)

pointer : offset del primer component del primer atribut respecte al buffer (normalment 0)

```
void glEnableVertexAttribArray (GLuint index);
```

Activa l'atribut del vèrtex identificat per *index*

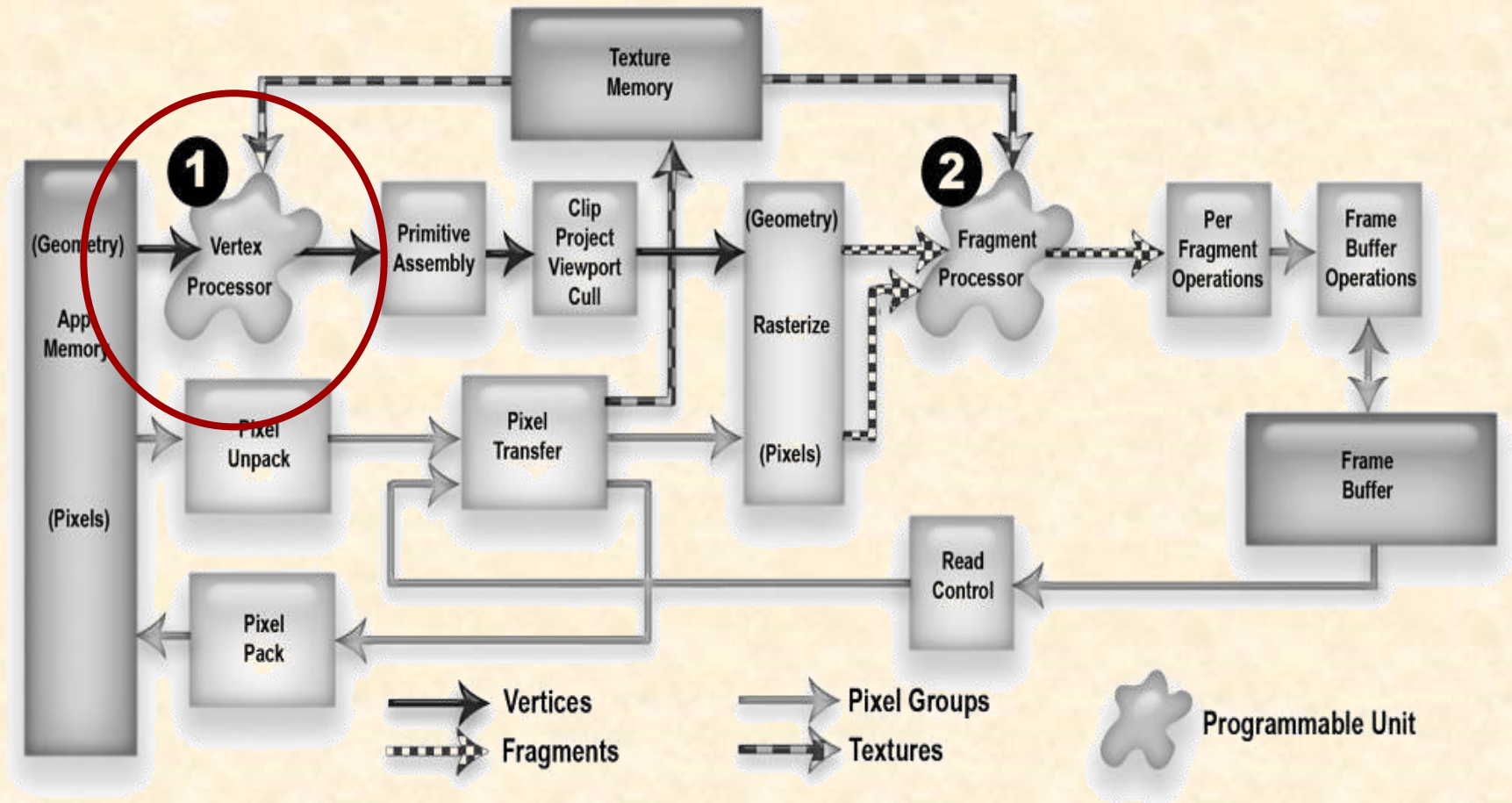
index : nom de l'atribut a activar

Vertex Buffer Objects

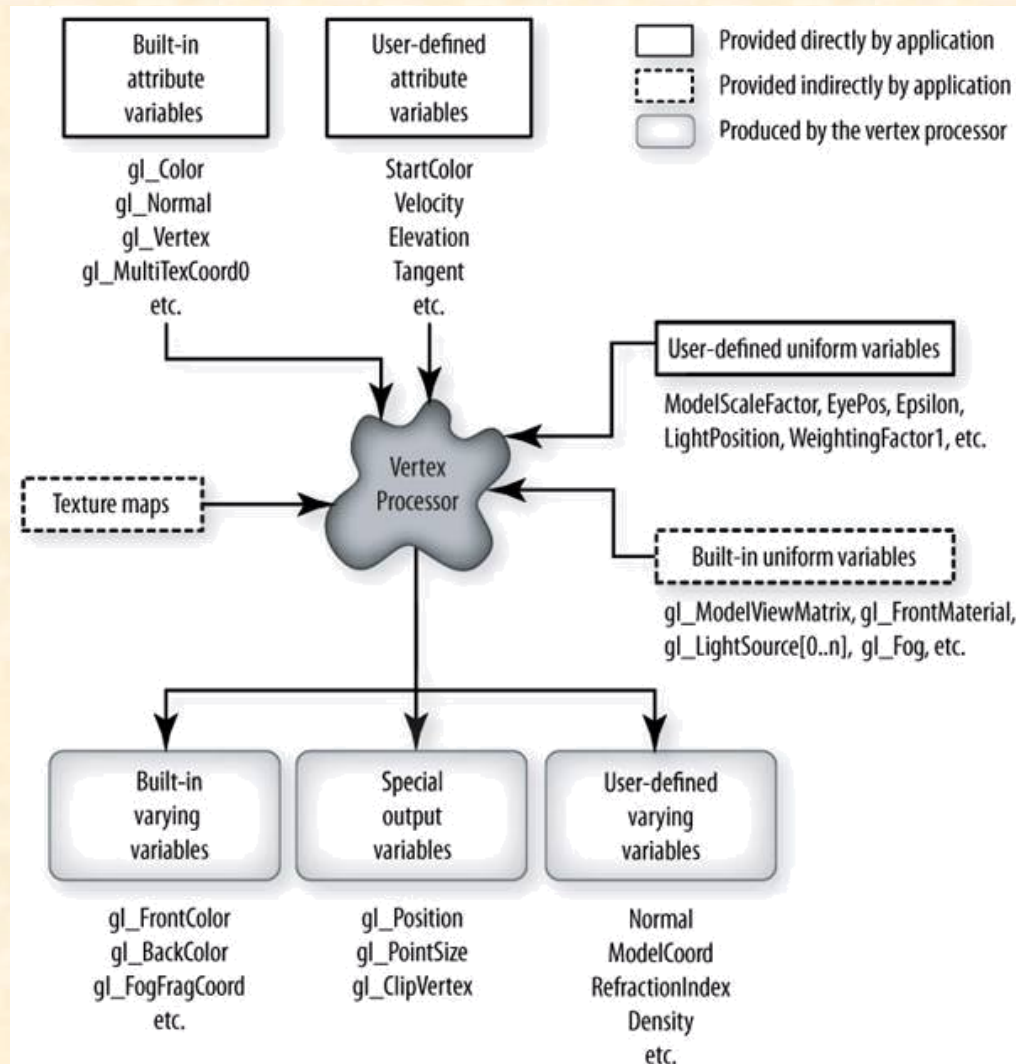
- Amb *vertex buffer objects*:
 - OpenGL copia les dades en el moment d'inicialitzar-les
 - En temps de pintat només es passa una espècie de punter

Shaders

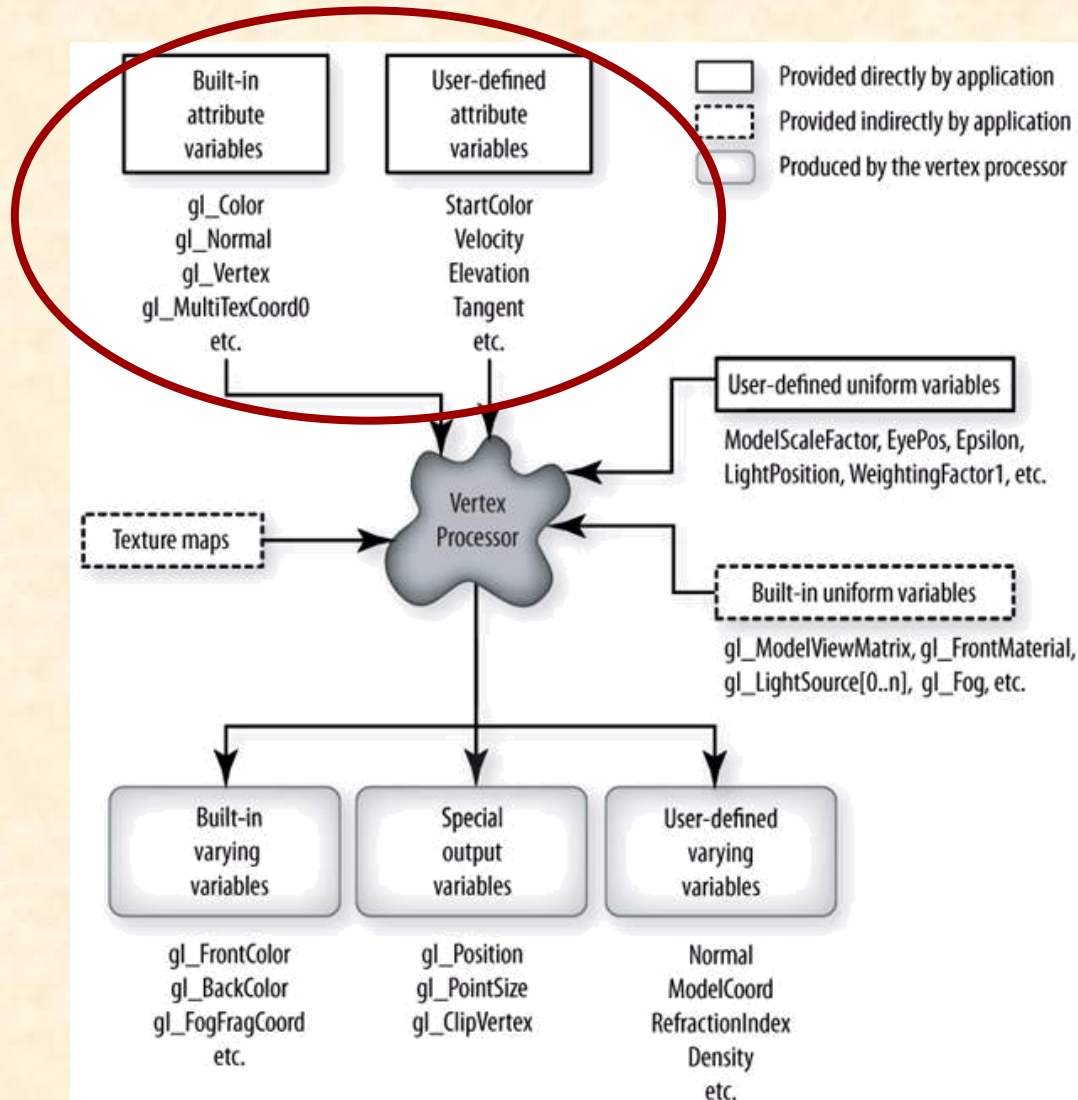
Vertex shaders



Vertex shaders



Vertex shaders



Vertex shaders

Attribute variables: són variables que representen els *atributs* d'un *vèrtex*. Poden canviar de valor per cada *vèrtex* d'una mateixa primitiva.

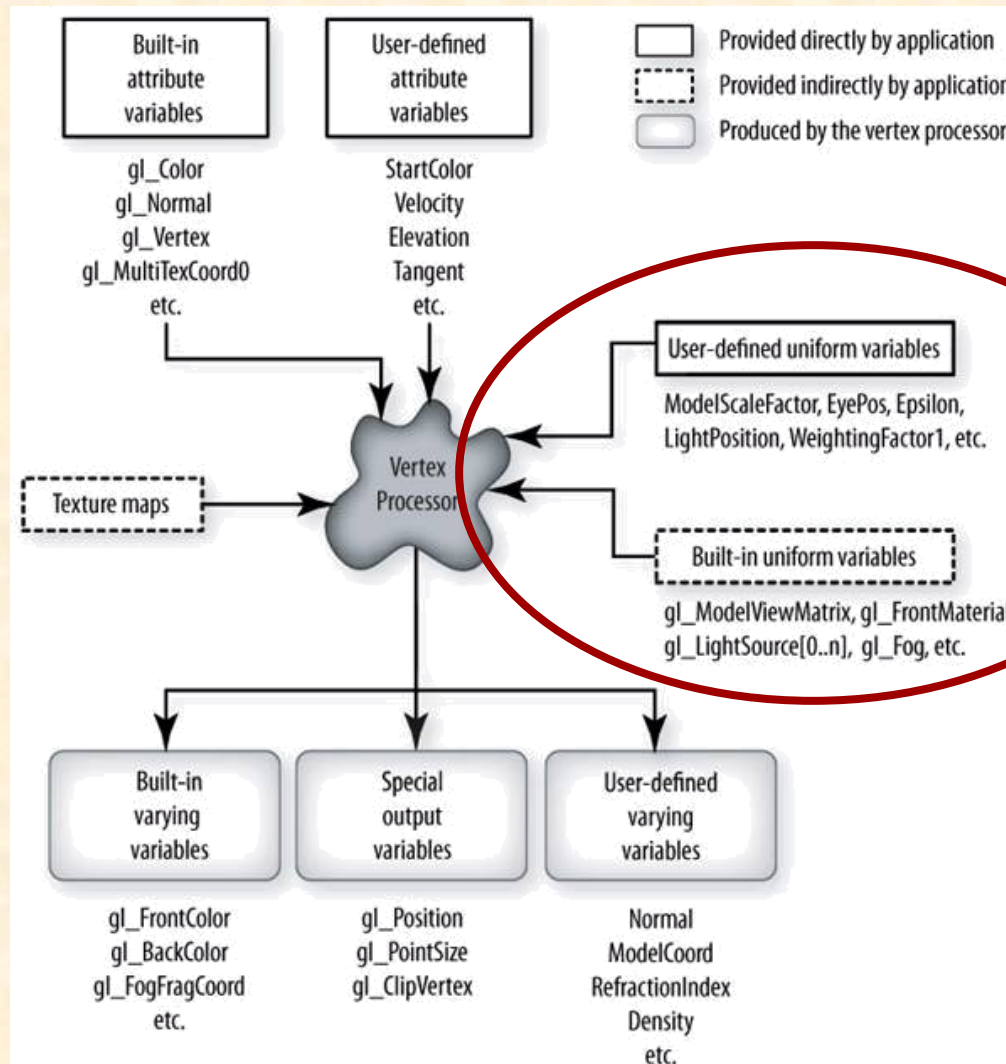
Built-in attributes: atributs predefinitos (*en WebGL tots són User-defined*)

- Des de l'aplicació s'envien amb glColor, glNormal...
- Des del shader s'accedeixen amb gl_Color, gl_Normal, gl_Vertex...

User-defined attributes: (cal declarar-los)

- Des de l'aplicació s'envien amb glVertexAttrib i es lliguen a un nom amb glGetAttribLocation.
- Des del shader s'accedeixen amb un nom arbitrari definit per l'usuari: velocitat, etc.

Vertex shaders



Vertex shaders

Uniform variables: són variables que canvien amb poca freqüència. Com a molt poden canviar un cop *per a cada primitiva* (però no per a cada vèrtex de la primitiva).

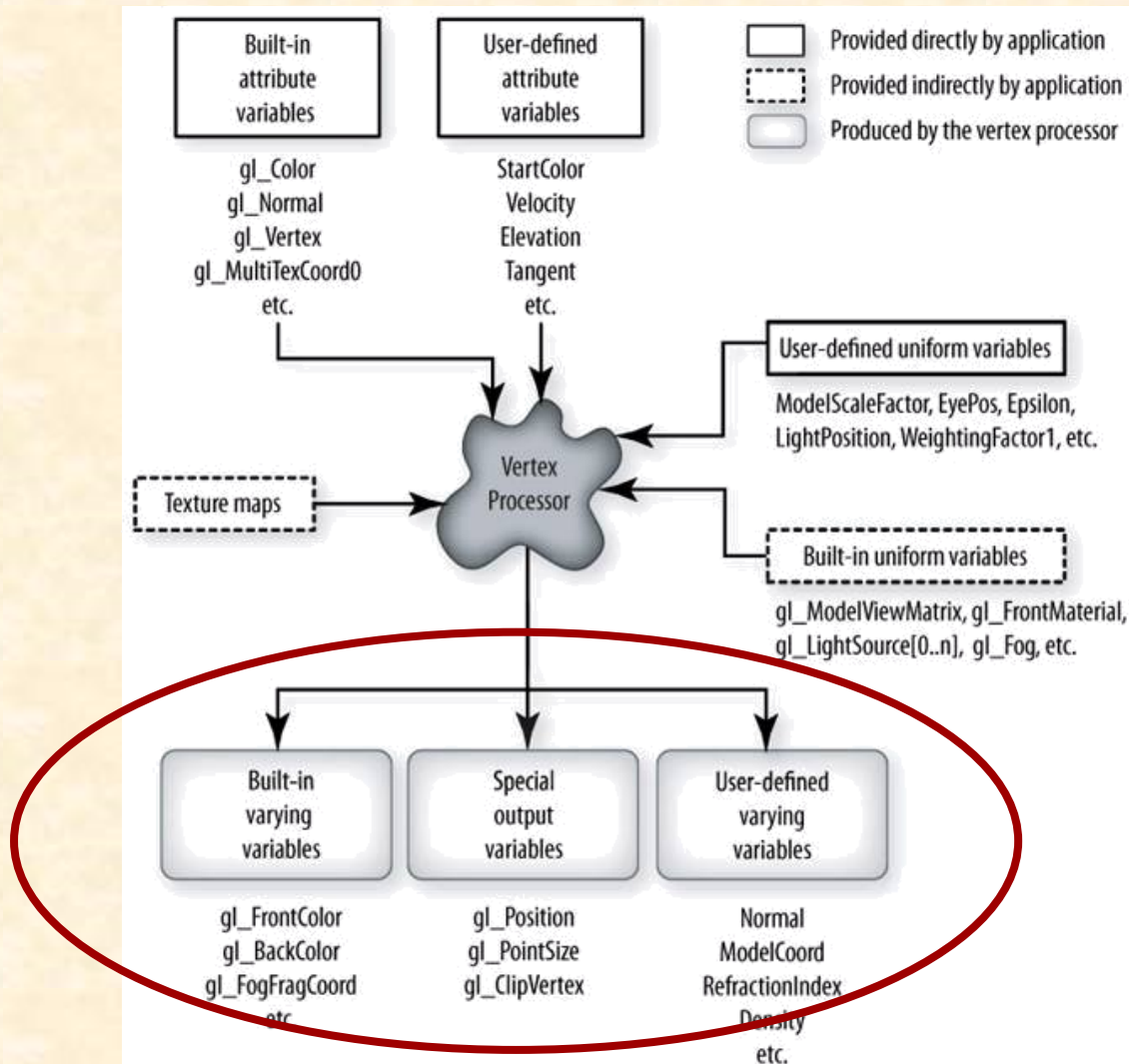
Built-in variables: són variables d'estat OpenGL (*en WebGL tots són User-defined*)

- Des del shader s'accedeixen amb `gl_ModelViewMatrix`, `gl_LightSource[0..n]`, etc

User-defined variables: cal declarar-les

- Des de l'aplicació s'envien amb `glUniform` i es lliguen a un nom amb `glGetUniformLocation`.
- Des del shader s'accedeixen amb un nom arbitrari definit per l'usuari: `EyePos`, etc.

Vertex shaders



Vertex shaders

Varying variables: són variables que es passen del vertex program al fragment program.

Pel vertex program són de sortida.

Pel fragment program són d'entrada, i es calculen per interpolació.

Built-in: predefinides (gl_FrontColor...) *En WebGL tots són User-defined*

User-defined: definides per l'usuari: (Normal, Refraction...); cal declarar-les.

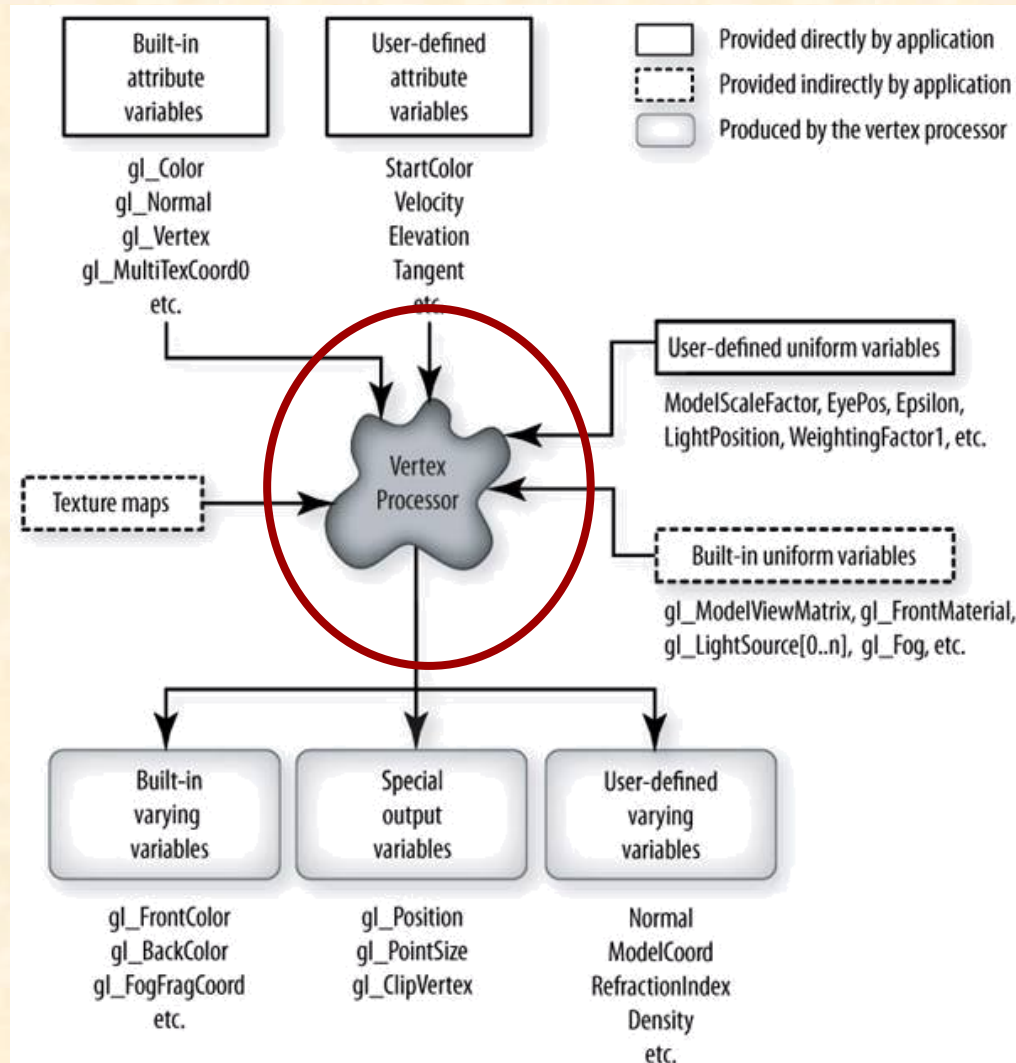
Vertex shaders

Special output variables: són variables que representen valors que ha de calcular el vertex program.

Com totes les predefinides, comencen amb `gl_` i no cal declarar-les.

Com a mínim ha de calcular **`gl_Position`**: coordenades del vèrtex en coordenades de clipping. Normalment ho farà multiplicant el vèrtex per la Modelview i la Projection.

Vertex shaders



Vertex shaders

Un vertex program s'executa per cada vèrtex que s'envia a OpenGL.

Les tasques *habituals* d'un vertex program són:

- Transformar el vèrtex (object space → clip space)
- Transformar i normalitzar la normal del vèrtex (eye space)
- Calcular la il·luminació del vèrtex
- Generar coordenades de textura del vèrtex
- Transformar les coords de textura

Exemple vertex shader

// uniform qualified variables are changed at most once per primitive

uniform float CoolestTemp;

uniform float TempRange;

uniform mat4 uMVMatrix;

uniform mat4 uPMatrix;

// attribute qualified variables are typically changed per vertex

attribute float VertexTemp;

attribute vec3 aVertexPosition;

// varying variables communicate from the vertex to fragment

varying float Temperat;

void main() {

 Temperat = (VertexTemp - CoolestTemp) / TempRange;

 gl_Position = uPMatrix * uMVMatrix * vec4 (aVertexPosition, 1.0);

}

Exemples de VS

Il·luminació bàsica dependent de la normal

Usar la normal com a color

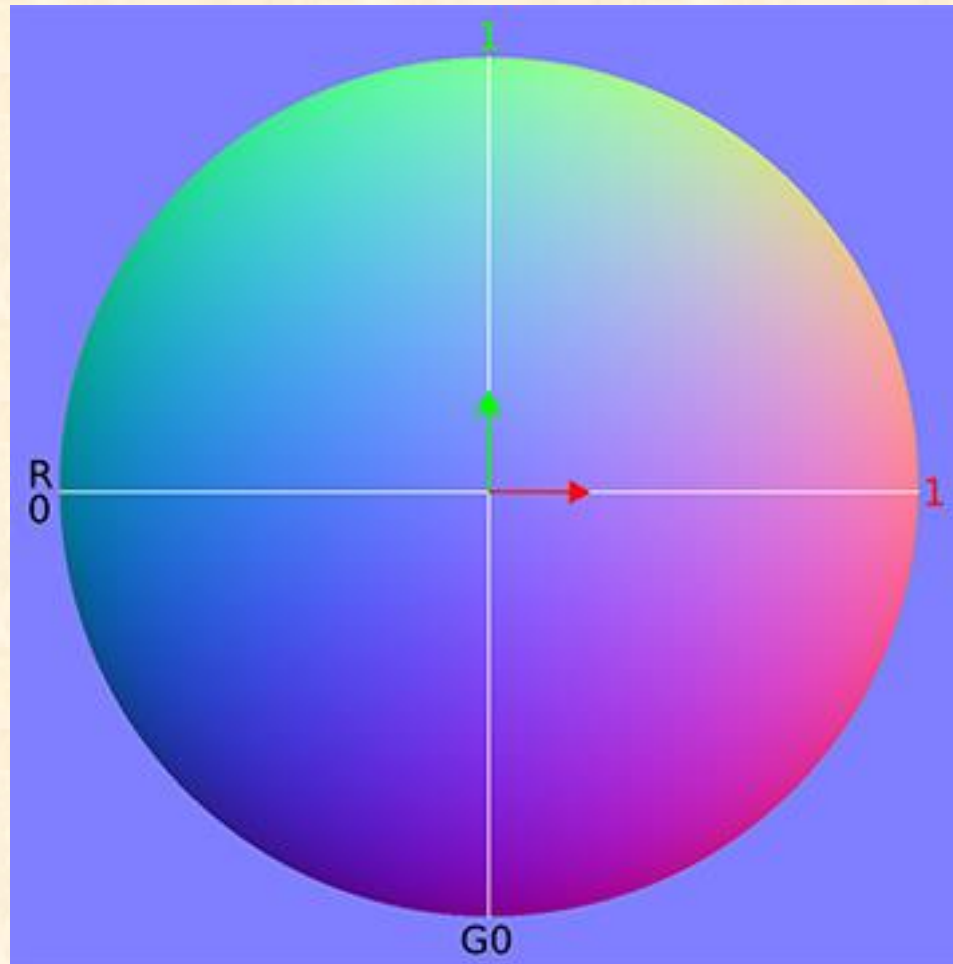
Deformació en el sentit de la normal

Altres:

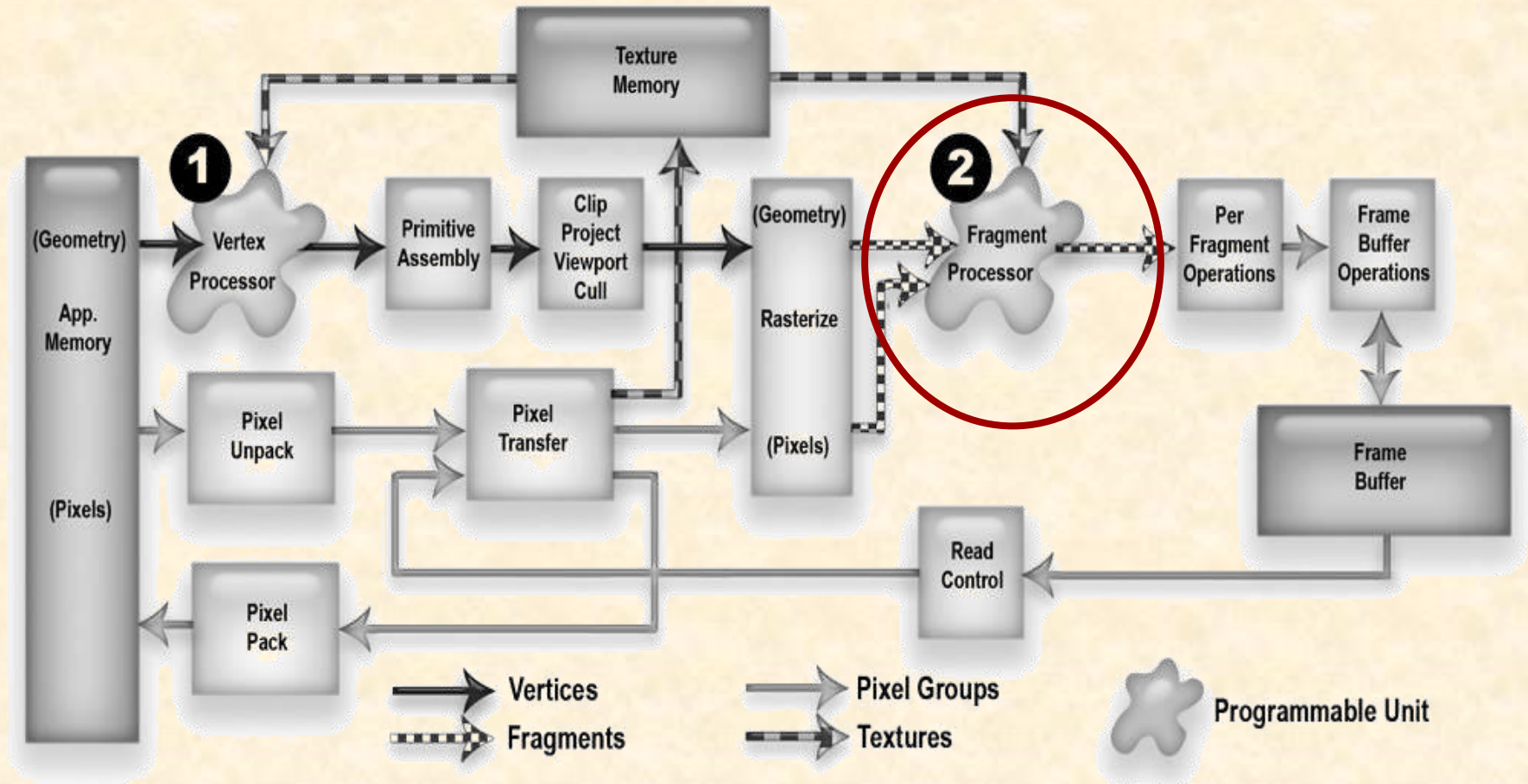
Projectar suaument sobre $Y=0$

Gradient de color

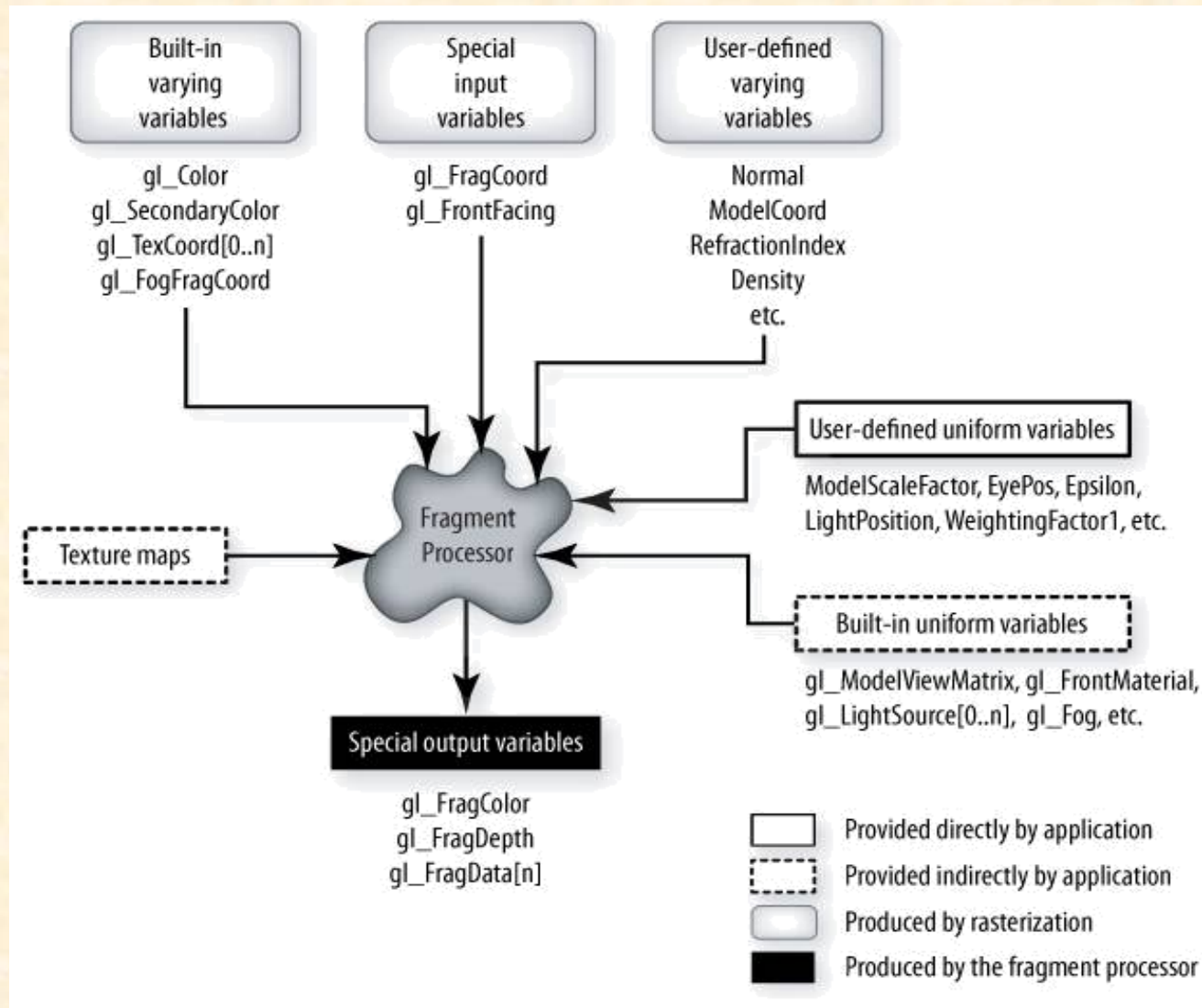
Normal com a color



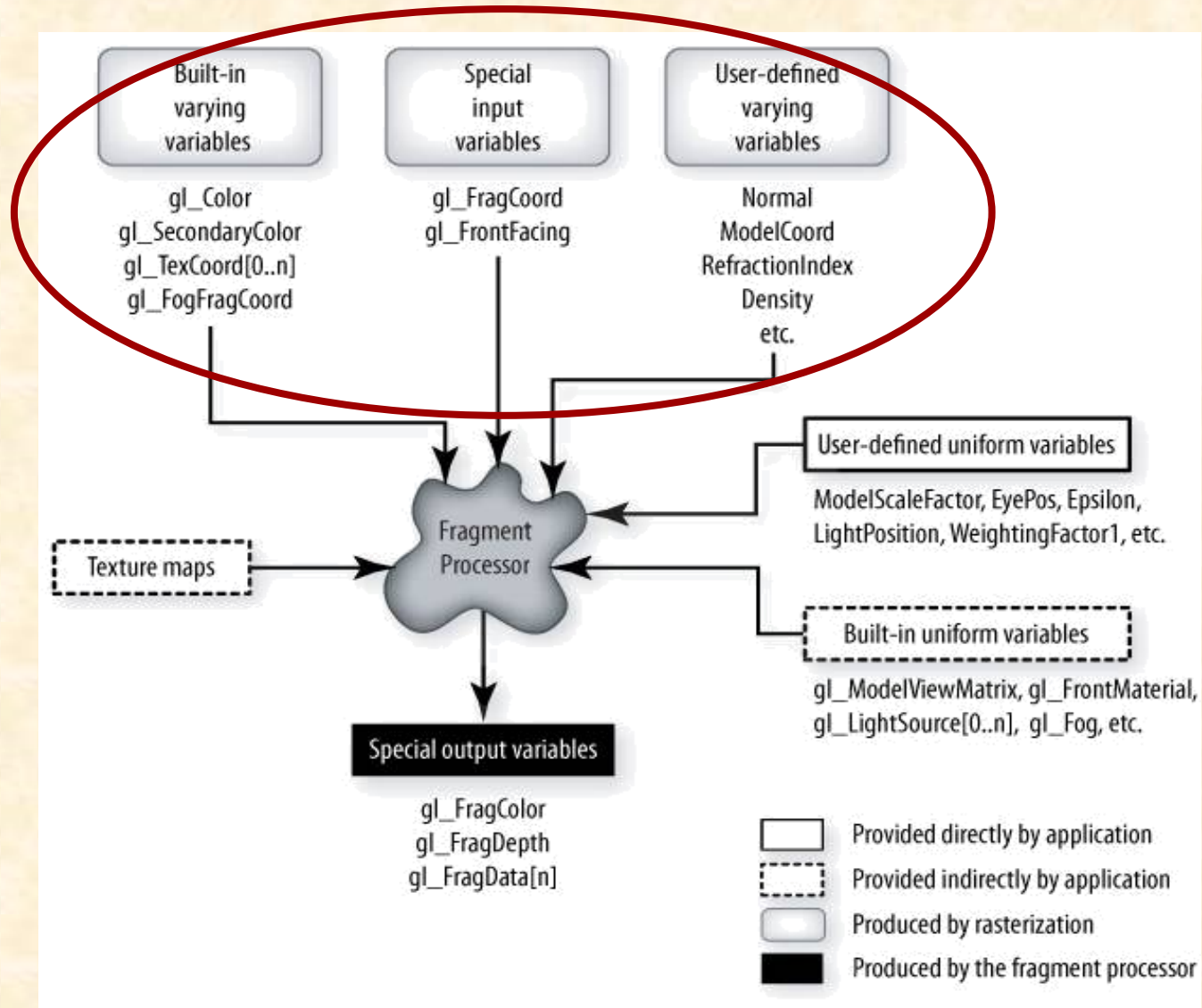
Fragment shaders



Fragment shaders



Fragment shaders



Fragment shaders

Varying variables: són variables que es calculen al vertex shader i s'envien al fragment shader. Els valors que arriben per a cada fragment són el resultat d'interpol·lar els valors calculats a cada vèrtex.

Varying predefinitos:

- `varying vec4 gl_Color;`
- `varying vec4 gl_TexCoord[];`
- ...

En WebGL no hi ha cap predefinit, tots s'han de declarar!

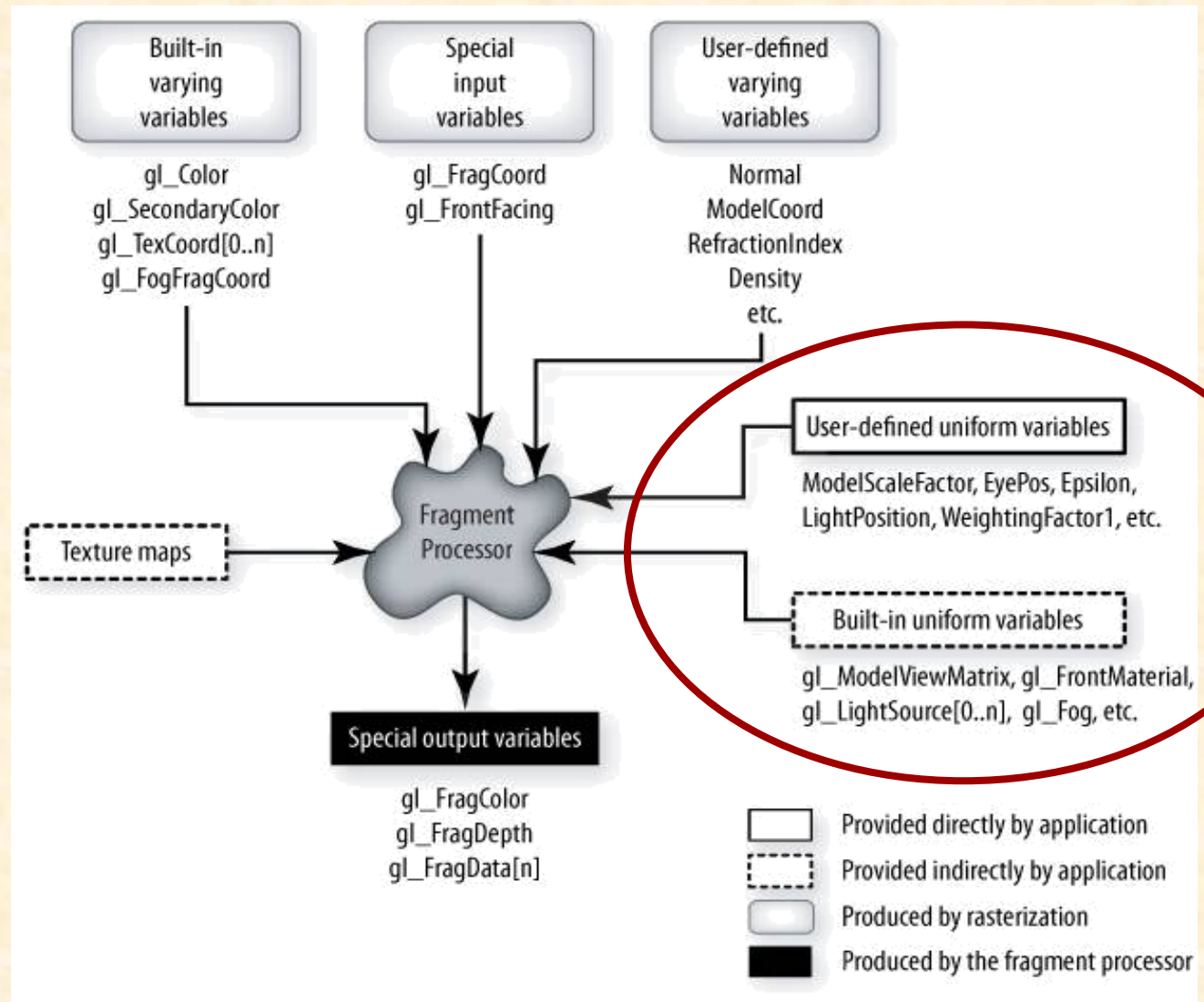
Fragment shaders

Special input variables: calculats per OpenGL de forma automàtica; es poden llegir al fragment shader:

```
vec4 gl_FragCoord;    // coordenades del fragment (window space)
```

```
bool gl_FrontFacing;  // true si el fragment és d'un polígon frontface
```

Fragment shaders

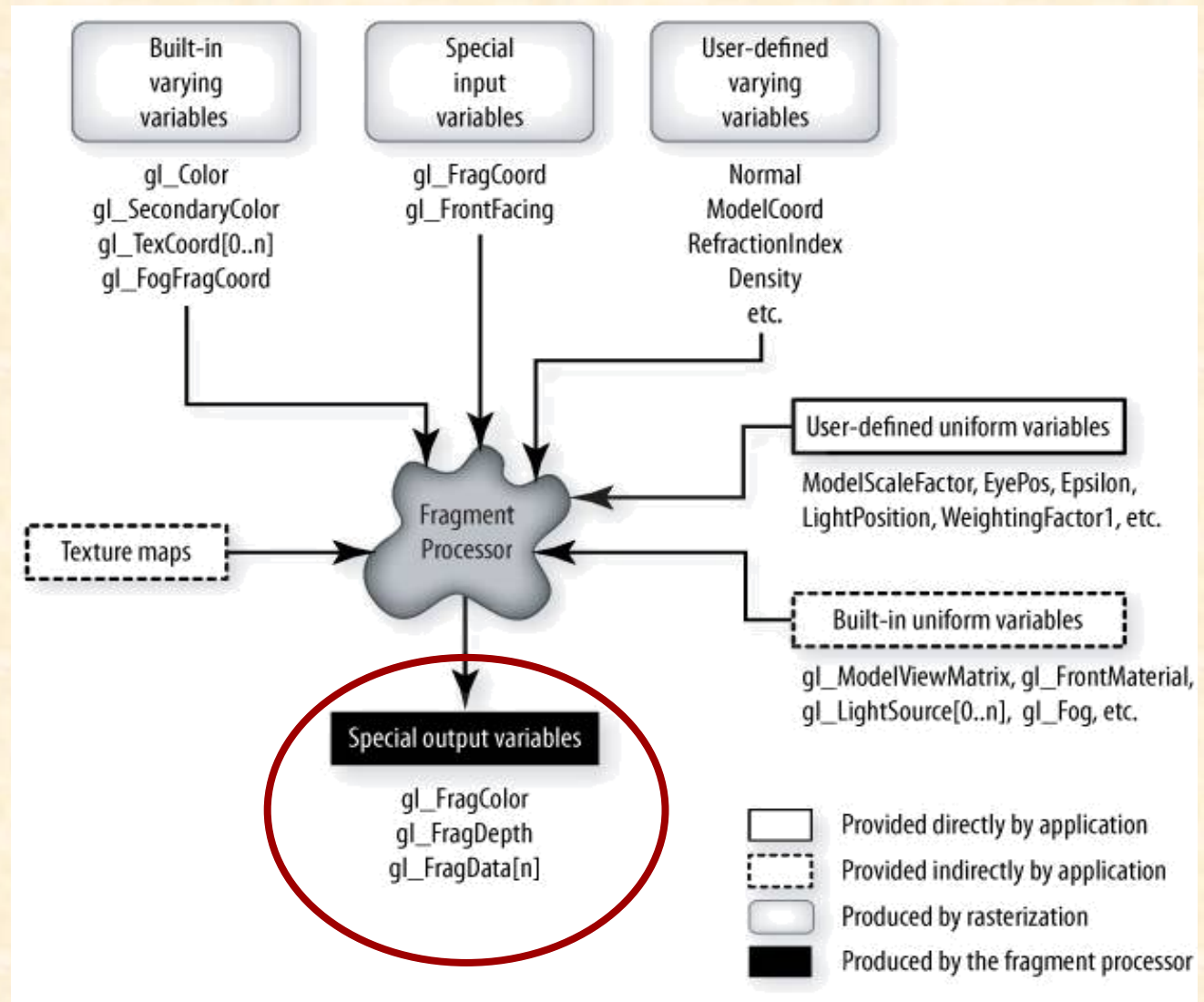


Fragment shaders

Uniform variables: són variables que canvien amb poca freqüència. Com a molt poden canviar un cop per cada primitiva (però NO per cada vèrtex de la primitiva).

- **Built-in uniforms:** corresponen a variables d'estat d'OpenGL (***en WebGL tots són User-defined***)
 - Des del shader s'accedeixen amb `gl_ModelViewMatrix`, `gl_LightSource[0..n]`, etc
 - Són les mateixes pels vertex shaders i pels fragment shaders.
- **User-defined uniforms:**
 - Des de l'aplicació s'envien amb `glUniform` i es lliguen a un nom amb `glGetUniformLocation`.
 - Des del shader s'accedeixen amb un nom arbitrari definit per l'usuari: `EyePos`, etc.

Fragment shaders

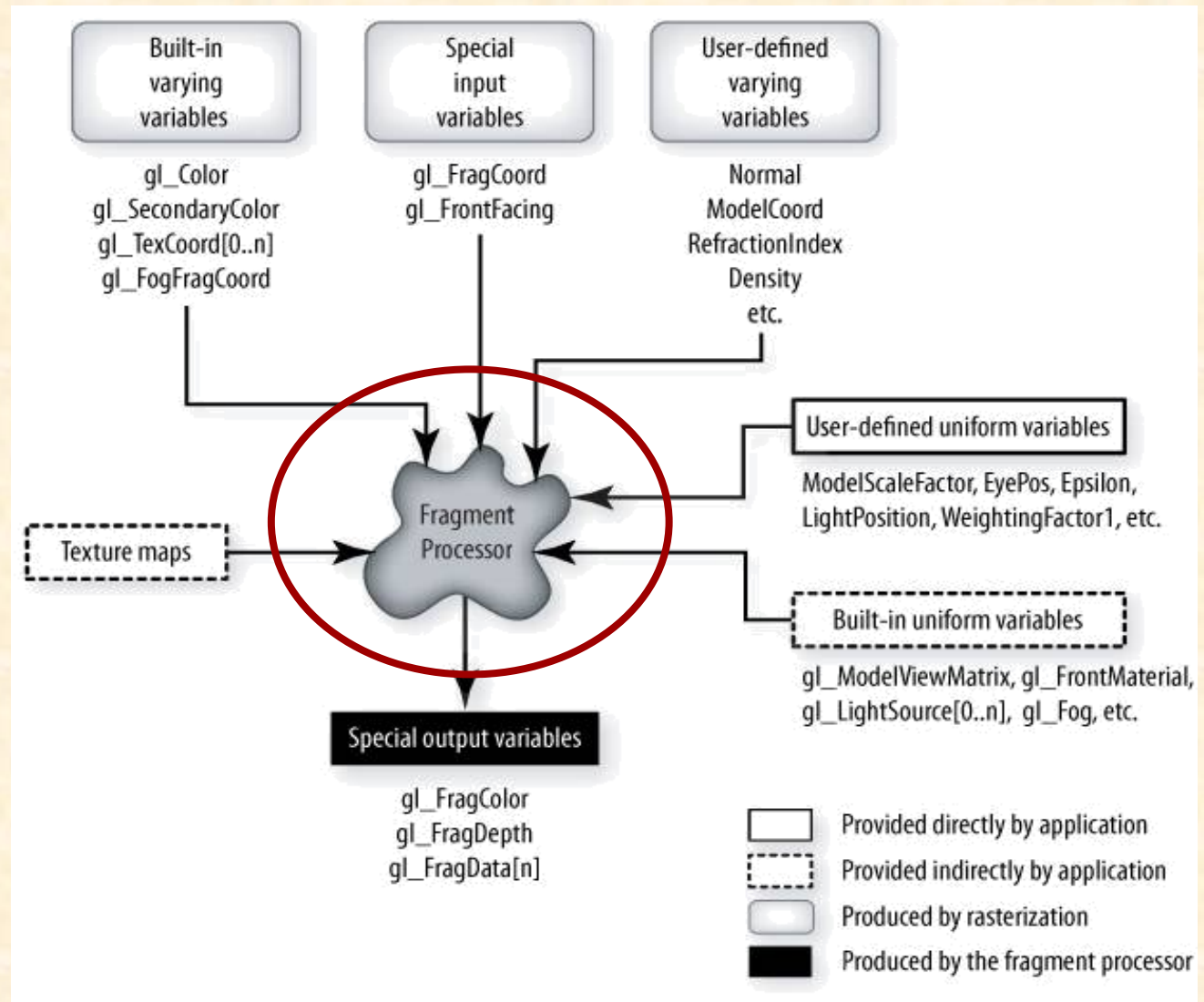


Fragment shaders

Special output variables: són els valors que ha de calcular el fragment shader:

- `vec4 gl_FragColor` // color del fragment (abans de blending)
- `float gl_FragDepth` // depth final del fragment (pel z-buffer)
- `vec4 gl_FragData[]` // usat per MRT (`glDrawBuffers`)

Fragment shaders



Fragment shaders

Un fragment shader s'executa per cada fragment que produeix cada primitiva.

Les tasques habituals d'un fragment shader són:

- Accedir a textura
- Incorporar el color de la textura
- Incorporar efectes a nivell de fragment (ex. boira).

I el que no pot fer un fragment shader:

- Canviar les coordenades del fragment (sí pot canviar **gl_FragDepth**)
- Accedir a informació d'altres fragments

Exemple fragment shader

```
uniform vec3 CoolestColor;
uniform vec3 HottestColor;
// Temperat contains the interpolated per-fragment value of
// temperature set by the vertex shader
varying float Temperat;
void main()
{
    // get a color between coolest and hottest colors, using
    // the mix() built-in function
    vec3 color = mix (CoolestColor, HottestColor, Temperat);
    // make a vector of 4 floats numbers by appending alpha of 1.0
    gl_FragColor = vec4 (color, 1.0);
}
```

Aspectes del llenguatge GLSL

Introducció a GLSL

GLSL = OpenGL Shading Language

Llenguatge d'alt nivell (similar a C) per escriure *shaders* (programes que s'executen a la GPU).

GLSL és estàndard a partir de la versió 2.0 d'OpenGL.

Aspectes de GLSL

Sintaxi similiar a C/C++

El punt d'inici és la funció **void main()**

Extensions al llenguatge C:

- Tipus float, int, bool
- Tipus **vec2**, **vec3**, **vec4** (vectors)
- Tipus **mat2**, **mat3**, **mat4** (matrius 2x2, 3x3, 4x4)
- Tipus **sampler1D**, **sampler2D**, **sampler3D** (textures 1D, 2D i 3D)
- Qualificadors **attribute**, **uniform**, **varying**

Aspectes de GLSL

Funcions pre-definides:

- Matemàtiques, geomètriques (producte escalar, vectorial...)
- Accés a un valor d'una textura.

Diferències bàsiques amb C/C++:

- No hi ha conversió automàtica de tipus.
- No suporta: apuntadors, char, double, short, long.
- Les funcions tenen paràmetres d'entrada **in**, sortida **out** i entrada/sortida **inout**. En tots els casos es passa per valor.

Tipus en GLSL

Tipus existents en C/C++:

- float `float a = 2.4e5;`
- int `int numTextures = 4;`
- bool `bool found = false;`

Tipus en GLSL: vectors

Vectors de 2,3,4 components de float, int, bool

- `vec2, vec3, vec4` → vectors de 2,3,4 floats
- `ivec2, ivec3, ivec4` → vectors de 2,3,4 int's
- `bvec2, bvec3, bvec4` → vectors de 2,3,4 bool's

Un vector pot representar:

- Punts/vectors: es pot accedir amb `.x, .y, .z, .w`
- Colors: es pot accedir amb `.r, .g, .b, .a`
- Coord textura: es pot accedir amb `.s, .t, .p, .q`

Tipus en GLSL: matrius

Matrius 2x2 ,3x3, 4x4 de float

- mat2, mat3, mat4

Exemple:

```
mat4 transform;
```

```
vec4 col = transform[2];    // vector 3ª columna
```

```
float v = transform[col][fila];
```

Tipus en GLSL: samplers

Representen textures 1D, 2D, 3D

- sampler1D, sampler2D, sampler3D
- samplerCube → textura cube-mapping
- sampler2DShadow → textura per shadow mapping

Exemple:

```
uniform sampler2D mySampler;
```

```
vec4 color = texture2D(mySampler, gl_TexCoord[0].st);
```

Tipus en GLSL: struct

Tenen un comportament molt similar a C/C++:

```
struct MyLight
{
    vec3 position;
    vec3 color;
};
```

```
MyLight light0 (vec3(1.0, 0.0, 0.0), vec3(1.0, 1.0, 1.0));
```

Tipus en GLSL: arrays

Es poden definir arrays de qualsevol tipus.

Quan es passen com a paràmetre es comporta com si es copies tot l'array.

```
struct MyLight  
{  
    vec3 position;  
    vec3 color;  
};
```

```
MyLight lights[2];  
mat4 matrius[3];
```

Variables en GLSL

Es poden declarar immediatament abans del seu ús.

Es poden inicialitzar les variables (excepte les variables **attribute**, **uniform** i **varying**)

La inicialització de tipus bàsics és igual que en C/C++:

```
float b=2.6;
```

La inicialització de tipus agregats adopta la forma de crida a un constructor:

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
```

```
MyLight L = MyLight(v, v);
```


Variables en GLSL

La inicialització de matrius es fa per columnes

```
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
```

```
m = [ 1.0  3.0 ]  
     [ 2.0  4.0 ]
```

Qualificadors en GLSL

attribute

- App → VP (poden variar per vèrtex)
- Es declaren en àmbit global; són read-only (*)
- Només poden ser float, vec2, vec3, vec4, mat2, mat3, mat4

uniform

- App → VP/FP (poden variar per primitiva)
- Es declaren en àmbit global; són read-only (*)
- Poden ser de qualsevol tipus.

varying

- VP → FP
- Es declaren en àmbit global; són de sortida pel VP i read-only pel FP

const

- Valor constant del VP/FP; són read-only.

<sense qualificador>

- Variable local o global que es pot llegir i escriure
- El temps de vida està limitat a cada execució del shader

Control en GLSL

Funcionament similar a C+:

- for, while, break, continue
- if/else
- **discard** → s'utilitza per descartar el fragment (impedir que s'actualitzi el frame buffer)

Operadors en GLSL

Swizzling:

El operador de selecció “.” es pot usar per seleccionar els elements d’un vector en un ordre determinat.

Exemples:

```
vec4 v;  
v.rgba;           // vec4  
v.rgb;            // vec3  
v.xy;             // vec2  
v.abgr;           // vec4, diferent ordre  
v.xy = vec2(2.0, 3.0); // només canvia x, y
```

Operadors en GLSL

La majoria d'operadors, quan s'apliquen a un vector, en realitat s'apliquen a cada component. Exemples:

```
vec4 u,v,w;
```

```
float a;
```

```
w=u+v;    // suma de dos vectors
```

```
w=a+u;    // es suma l'escalar a cada component
```

```
w++;      // incrementa totes les components
```

Funcions en GLSL

Qualificadors de paràmetres de funcions:

in: paràmetre d'entrada (es passa per valor).

Copy in but don't copy back out; still writable within the function

out: paràmetre de sortida (es retornarà per valor)

Only copy out; readable, but undefined at entry to function

inout: paràmetre d'entrada/sortida (es passa i es retorna per valor)

Copy in and copy out

Funcions en GLSL

Exemple:

```
void computeCoord(in vec3 normal, inout vec3 coord)
{
    coord = coord + normal;
}
```

```
vec3 computeCoord(in vec3 normal, in vec3 coord)
{
    return coord + normal;
}
```

Funcions predefinides

Funcions matemàtiques:

`sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`

`radians()`, `degrees()`

`pow()`, `exp()`, `exp2()`, `log()`, `log2()`, `sqrt()`

`abs()`, `floor()`, `ceil()`

- `floor(8.2) → 8.0`

- `ceil (8.2) → 9.0`

`fract()`, `mod()`

- `fract(8.4) → 0.4`

Funcions predefinides

Funcions geomètriques (vec és float, vec2, vec3 ó vec4)

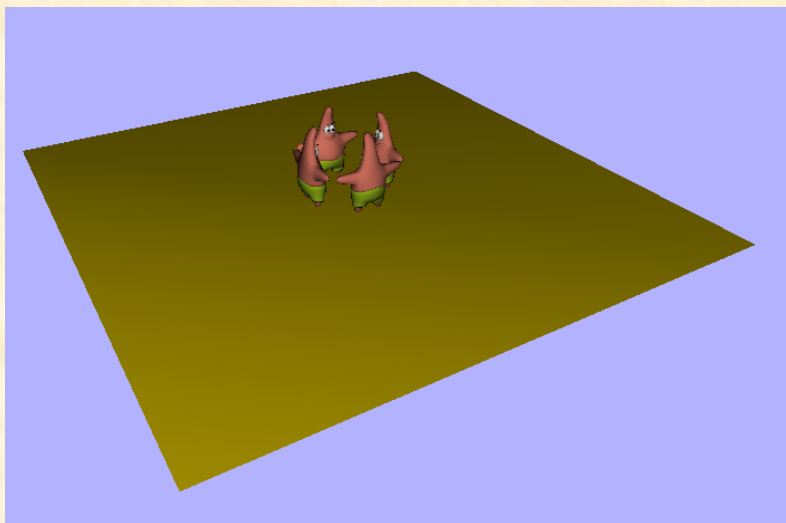
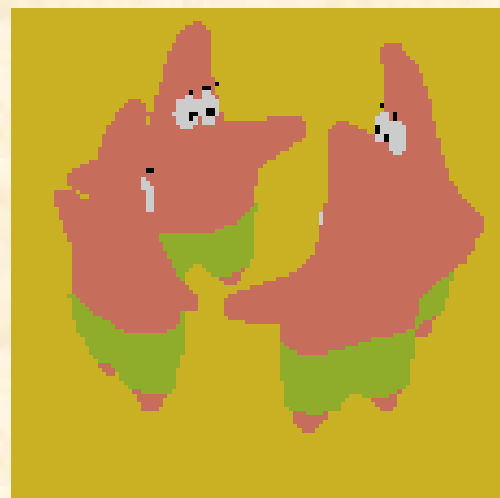
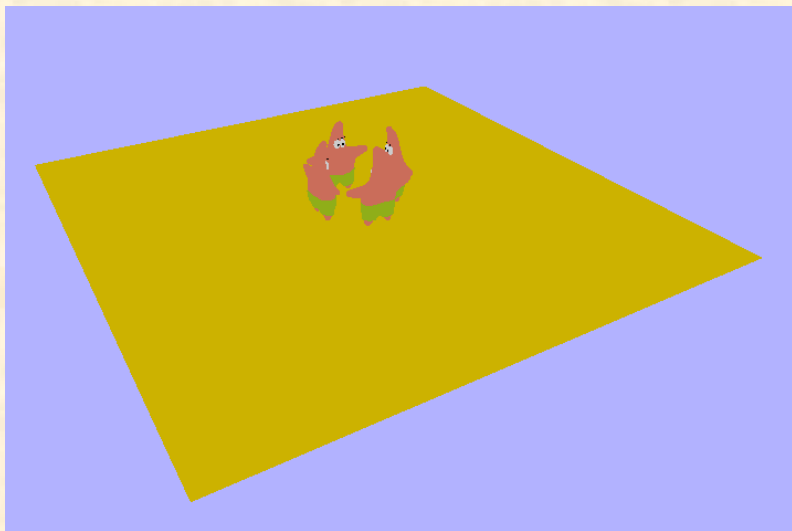
- float length(vec v);
- float distance(vec, vec); // distància entre dos punts
- float dot(vec, vec); // prod. escalar
- vec3 cross(vec3, vec3); // prod. vectorial
- vec normalize(vec); // retorna vector unitari
- vec reflect(vec I, vec N); // sentit de I!
- vec refract(vec I, vec N, float mu);
- Producte de matrius i vectors amb matrius (ex. $M \cdot v$)

Funcions predefinides



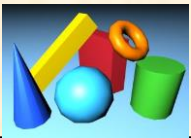
Funcions d'accés a textures:

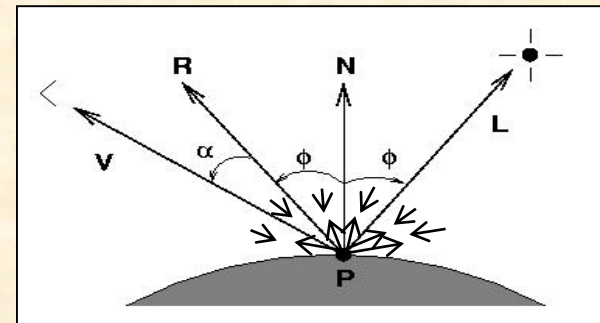
- `vec4 texture1D(sampler1D sampler, float coord);`
- `vec4 texture2D(sampler2D sampler, vec2 coord);`
- `vec4 texture3D(sampler3D sampler, vec3 coord);`
- `vec4 textureCube(samplerCube s, vec3 coord);`

Càlcul de la il·luminació



Resum Models Empírics

| Color d'un punt degut a... | Depèn de la normal? | Depèn de l'observador? | Exemple |
|----------------------------|---------------------|------------------------|---|
| Model ambient | No | No |  |
| Model difús | Sí | No |  |
| Model especular | Sí | Sí |  |



$$I_{\lambda}(P) = I_{a\lambda} k_{a\lambda} + \sum_i (I_{f_i\lambda} k_{d\lambda} \cos(\Phi_i)) + \sum_i (I_{f_i\lambda} k_{s\lambda} \cos^n(\alpha_i))$$

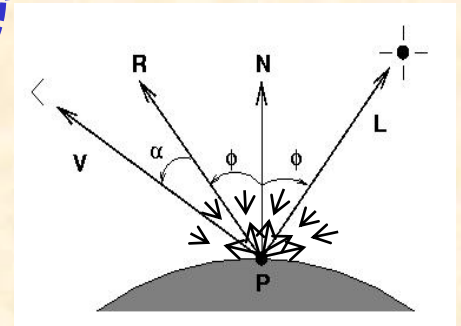
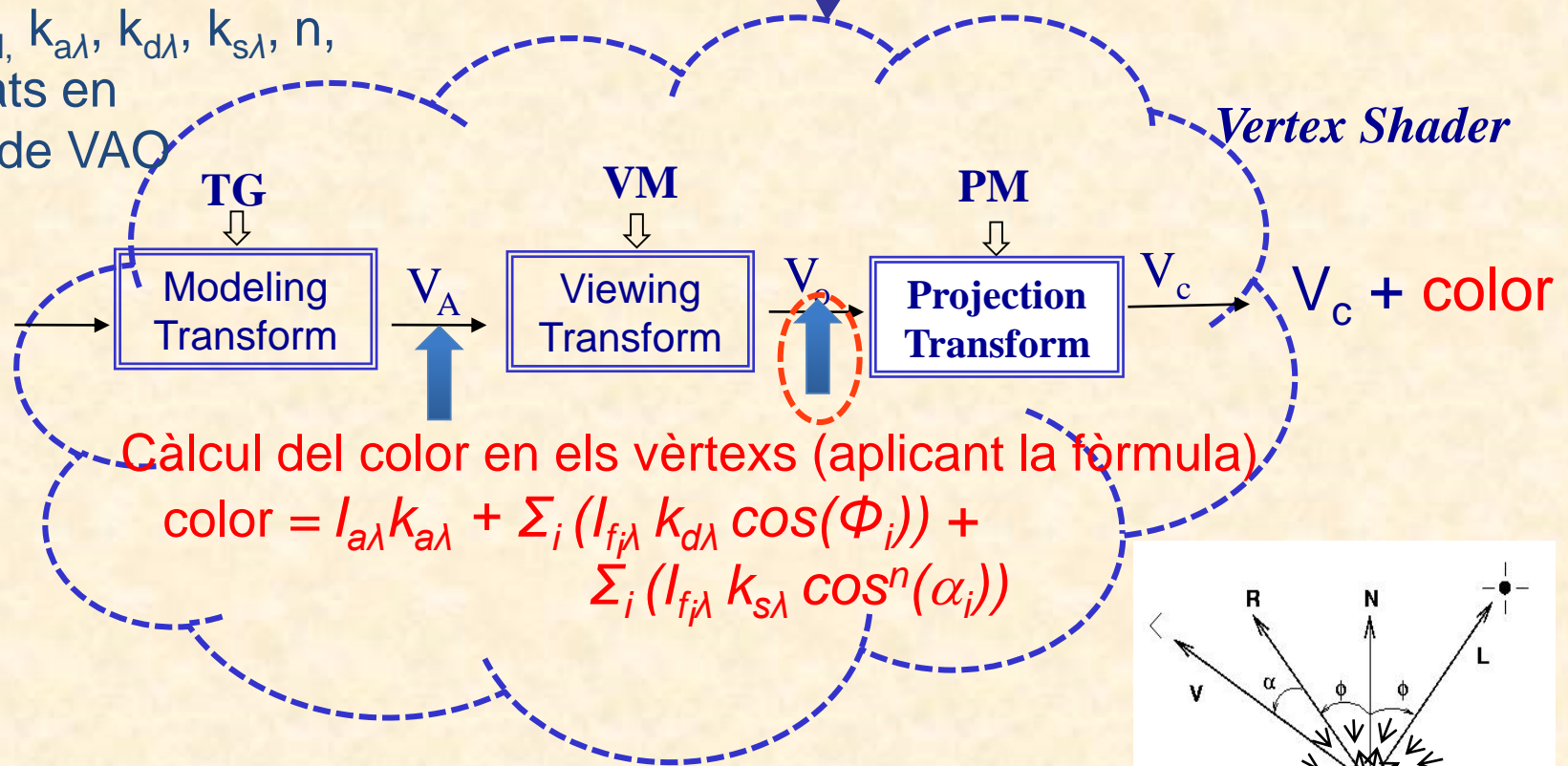
Càlcul del color del punt en el Vertex Shader

Uniforms:

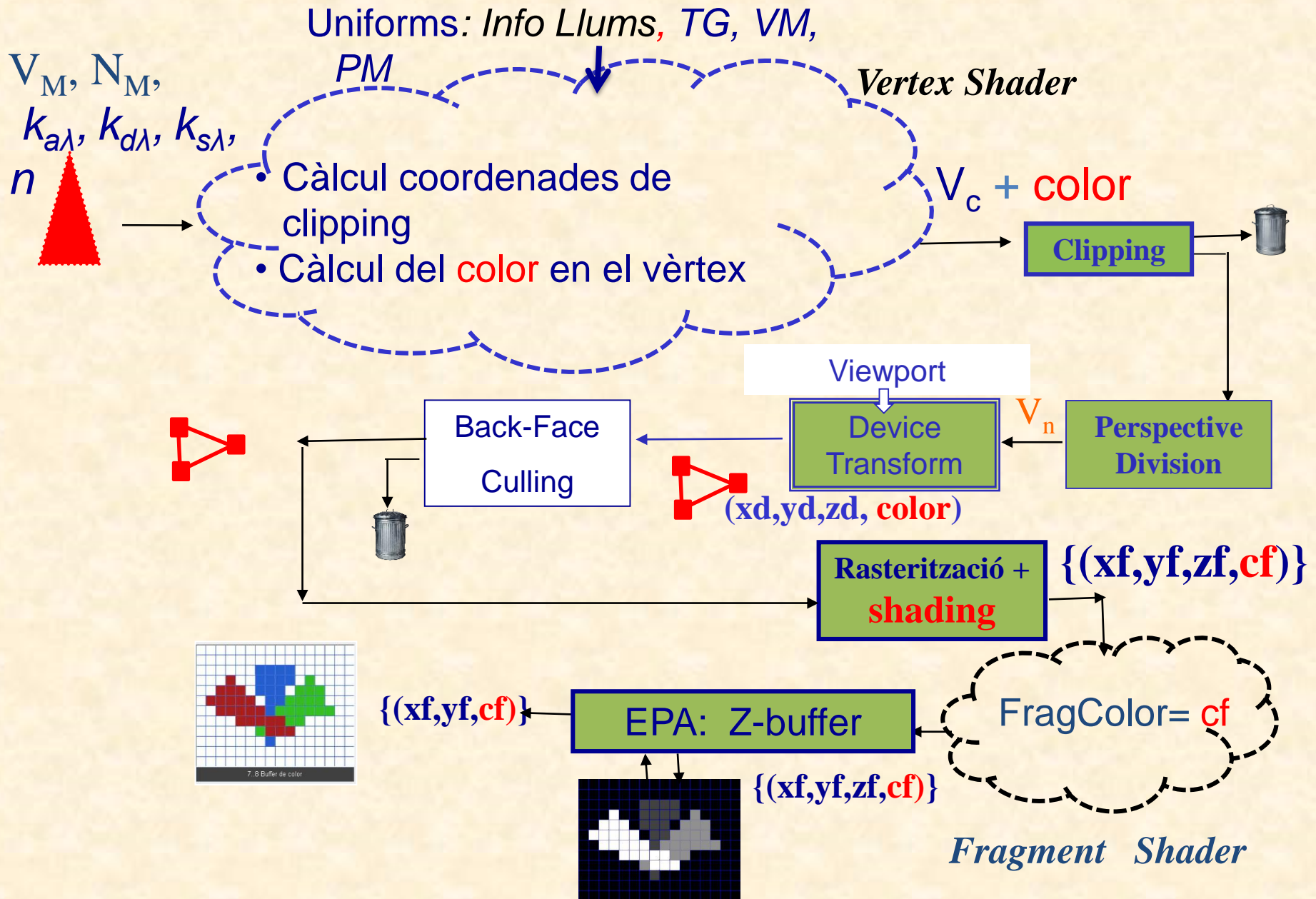
- Fonts de llum actives => color, posició
- Color llum ambient

Atributs del vèrtex:

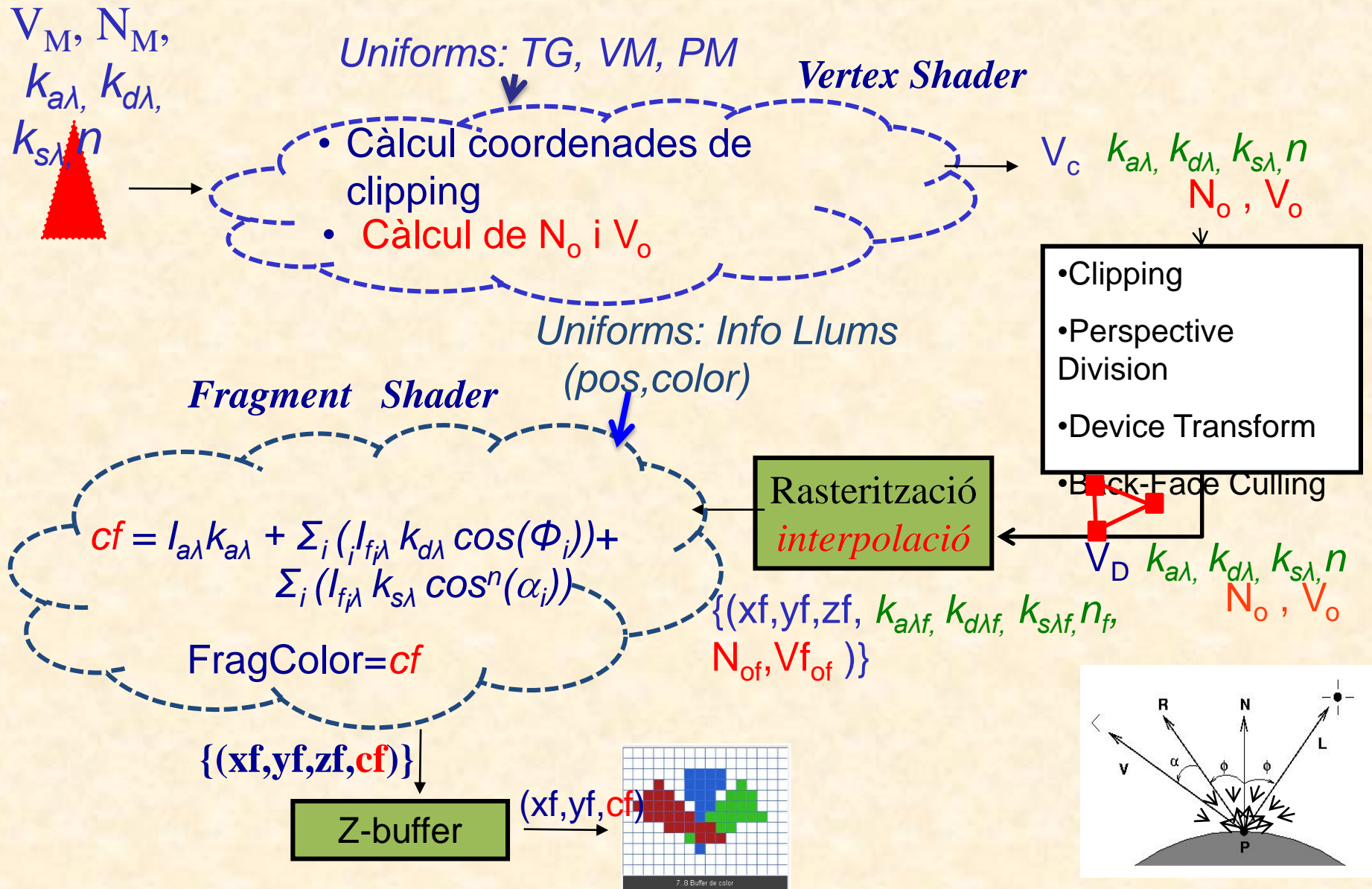
V_M , N_M , $k_{a\lambda}$, $k_{d\lambda}$, $k_{s\lambda}$, n ,
guardats en
VBOs de VAO



Shading (colorat) de polígons



Phong Shading: Càlcul color en el Fragment Shader



Gràfics 3D

Pintat de geometria, Shaders i Il·luminació

SGI – Màster MEI – curs 17/18 Q1