

GLSL– OpenGL Shading Language

Carlos Andújar

Febrero 2012

Continguts

- Introducció a GLSL
- Pipeline fix
- Pipeline programable
- Vertex shaders
- Fragment shaders
- Aspectes de GLSL
- API per definir shaders

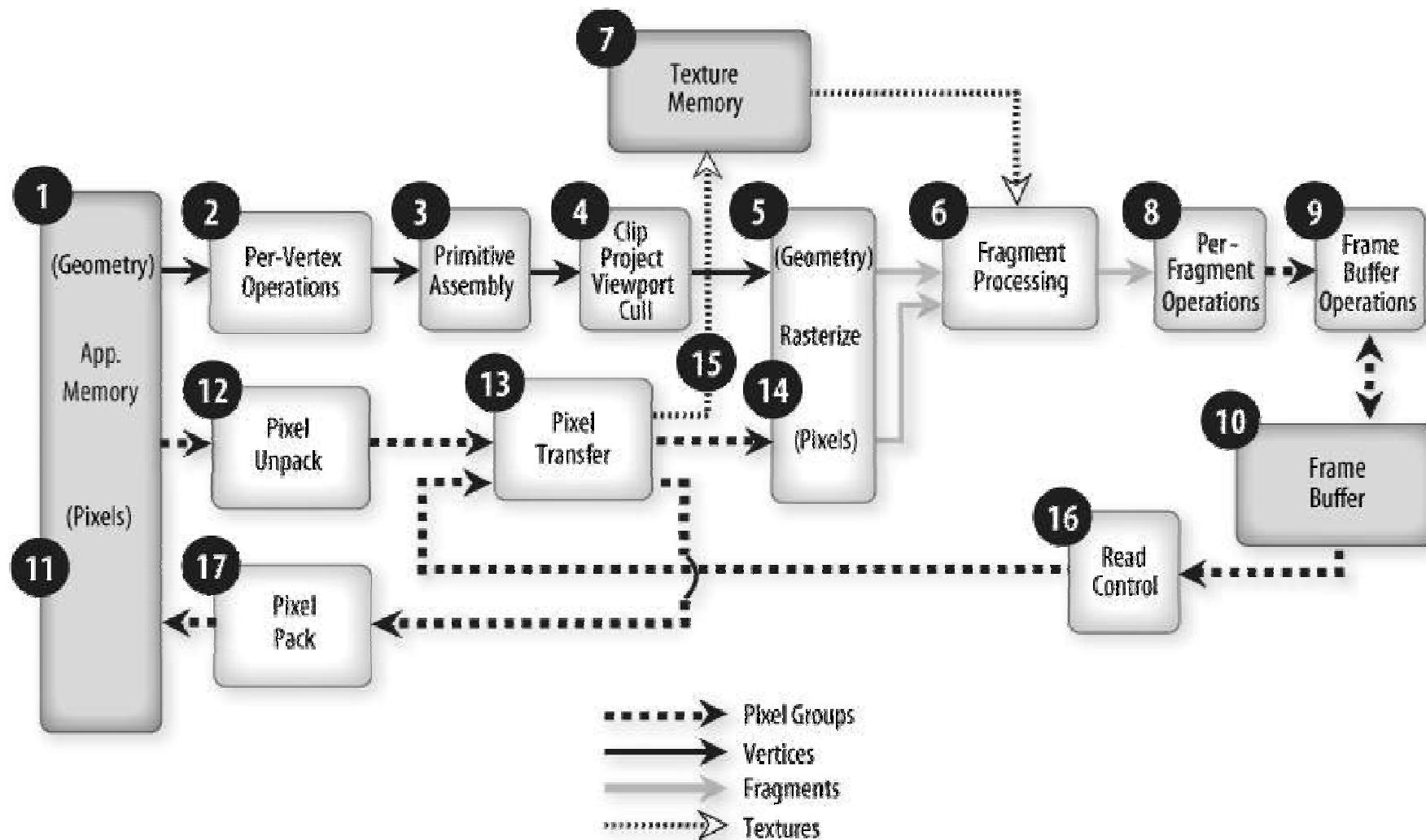
(*) Moltes de les figures d'aquests apunts són del llibre R. J. Rost,
OpenGL Shading Language, Addison-Wesley

Introducció a GLSL

- GLSL = OpenGL Shading Language
- Llenguatge d'alt nivell (similar a C) per escriure *vertex shaders*, *geometry shaders* i *fragment shaders* (programes que s'executen a la GPU).
- GLSL és estàndard a partir de la versió 2.0 d'OpenGL.

PIPELINE FIX OPENGL

Pipeline OpenGL



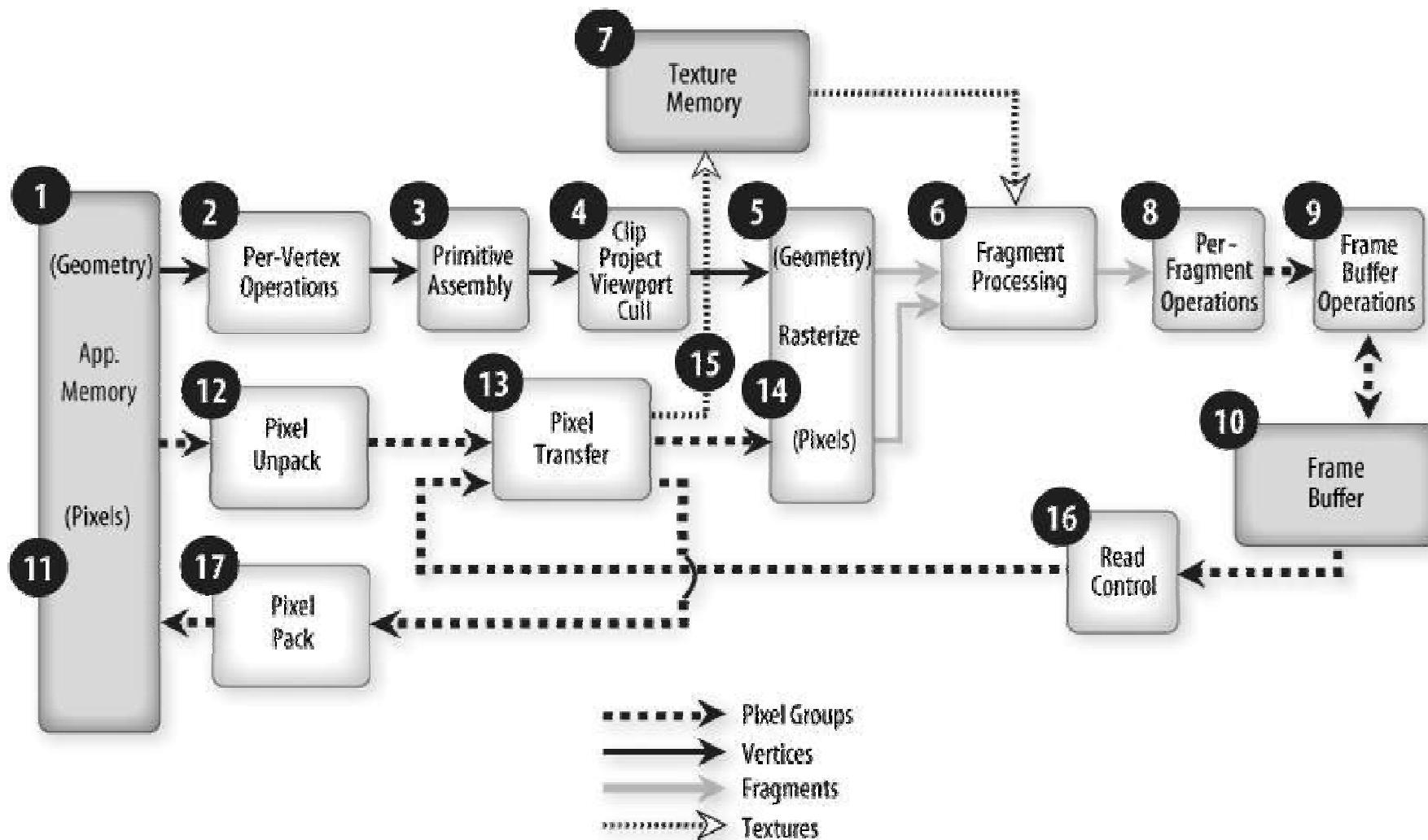
Pipeline OpenGL

1. Dibuix de primitives

Les primitives gràfiques (punts, línies, polígons...) es poden enviar a OpenGL de tres formes:

- Vèrtex a vèrtex: *glBegin*, *glVertex*, *glEnd*
- Vertex arrays: *glDrawArrays*, *glDrawElements*...
 - Vertex buffer object: vertex array emmagatzemat a la GPU
- Display lists: *glNewList*, *glCallList*, *glEndList*

Pipeline OpenGL

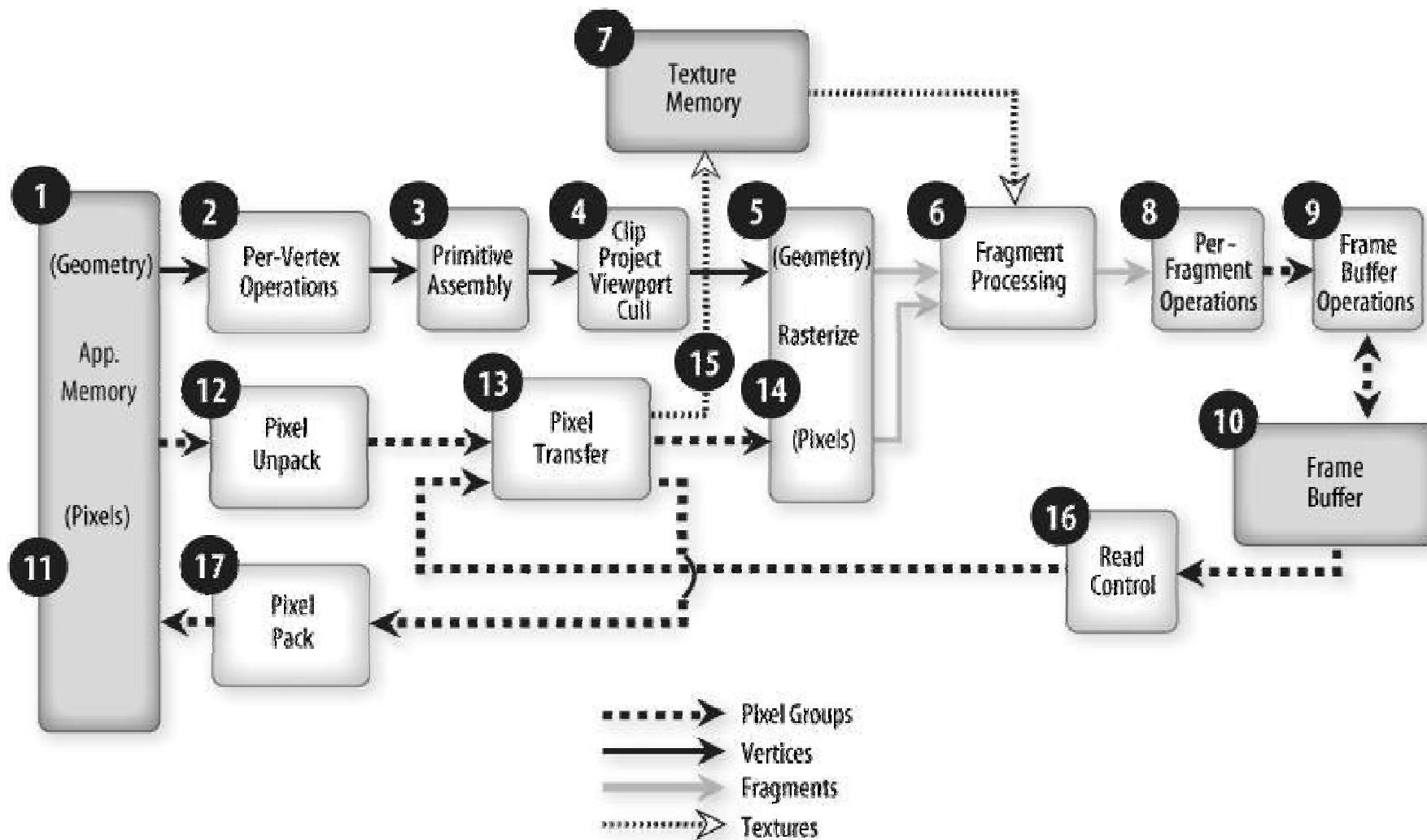


Pipeline OpenGL

2. Per-vertex operations

- Es transformen els vèrtexs (modelview i projection).
- Es transformen les normals (trasposta de l'inversa de la submatriu 3x3 de la modelview) i es calcula la il·luminació del vèrtex.
- Es generen coords de textura de forma automàtica.
- Es transformen les coord de textura (texture matrix).

Pipeline OpenGL

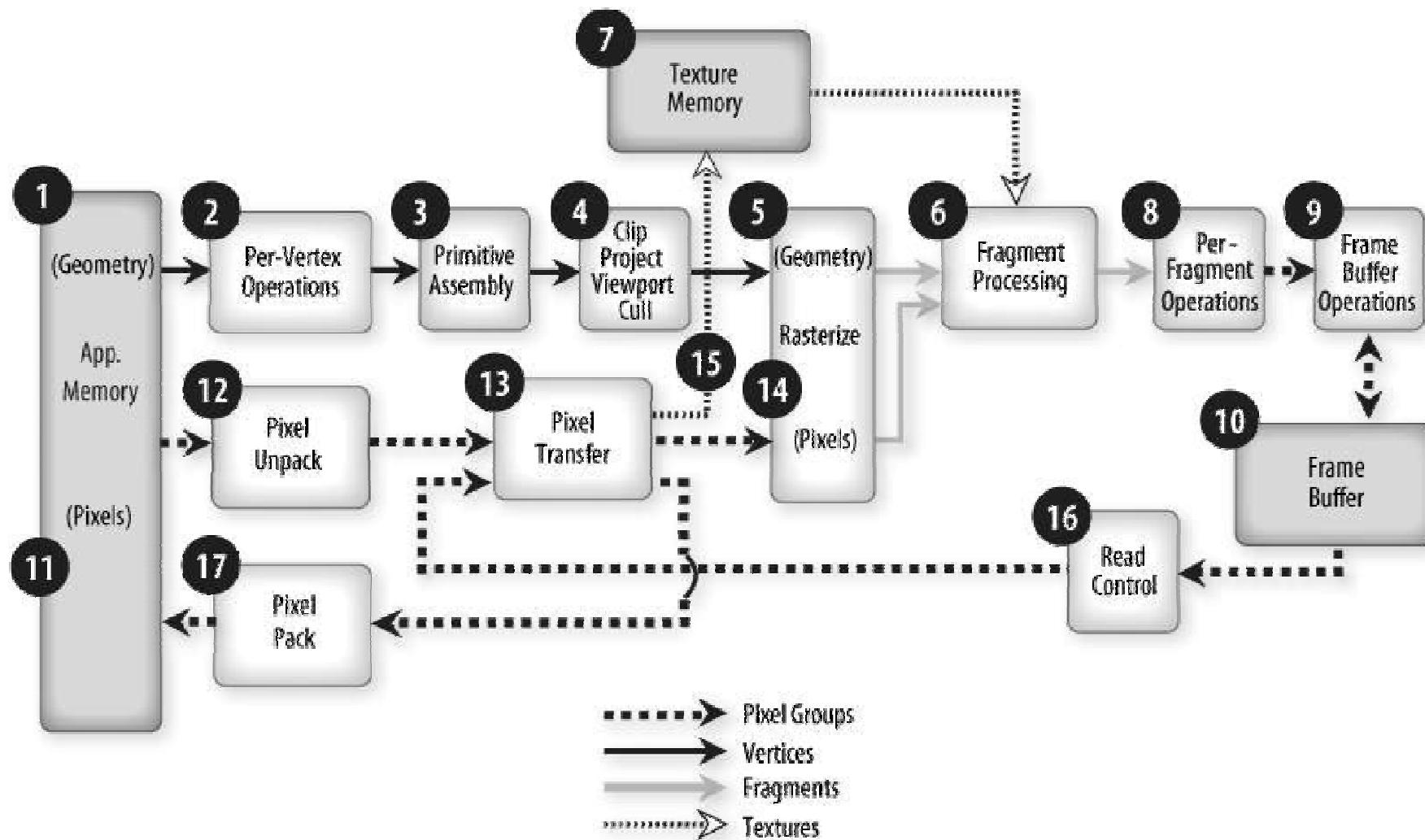


Pipeline OpenGL

3. Primitive assembly

- Els vèrtexs s'agrupen formant primitives.
- Cada primitiva GL_POINT, GL_LINES, GL_POLYGON requereix un clipping diferent.

Pipeline OpenGL

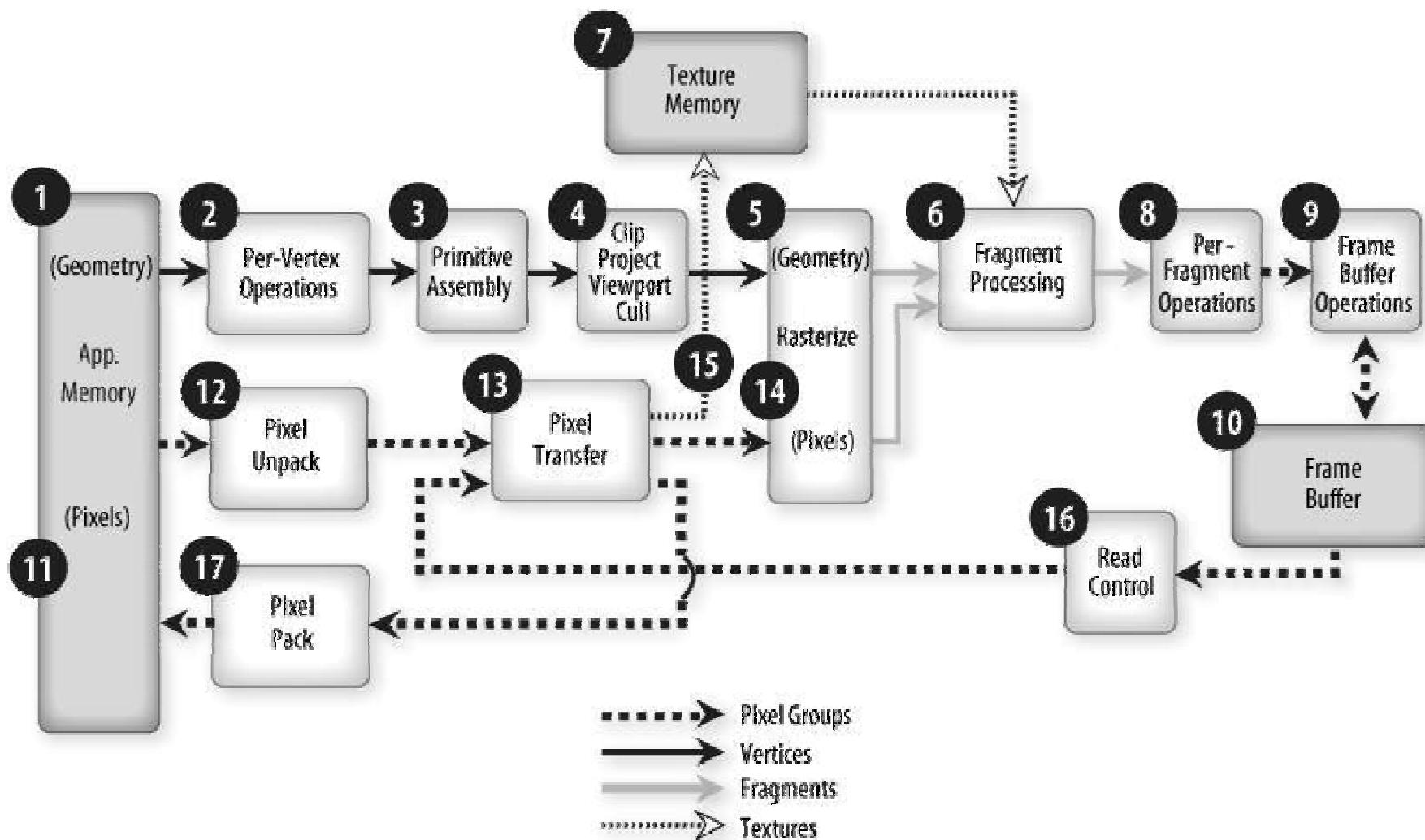


Pipeline OpenGL

4. Primitive processing

- Clipping (retallat) a la piràmide de visió.
- Divisió de perspectiva: es divideix (x,y,z) per w
- Viewport & depth transform → window coordinates
- Backface culling

Pipeline OpenGL

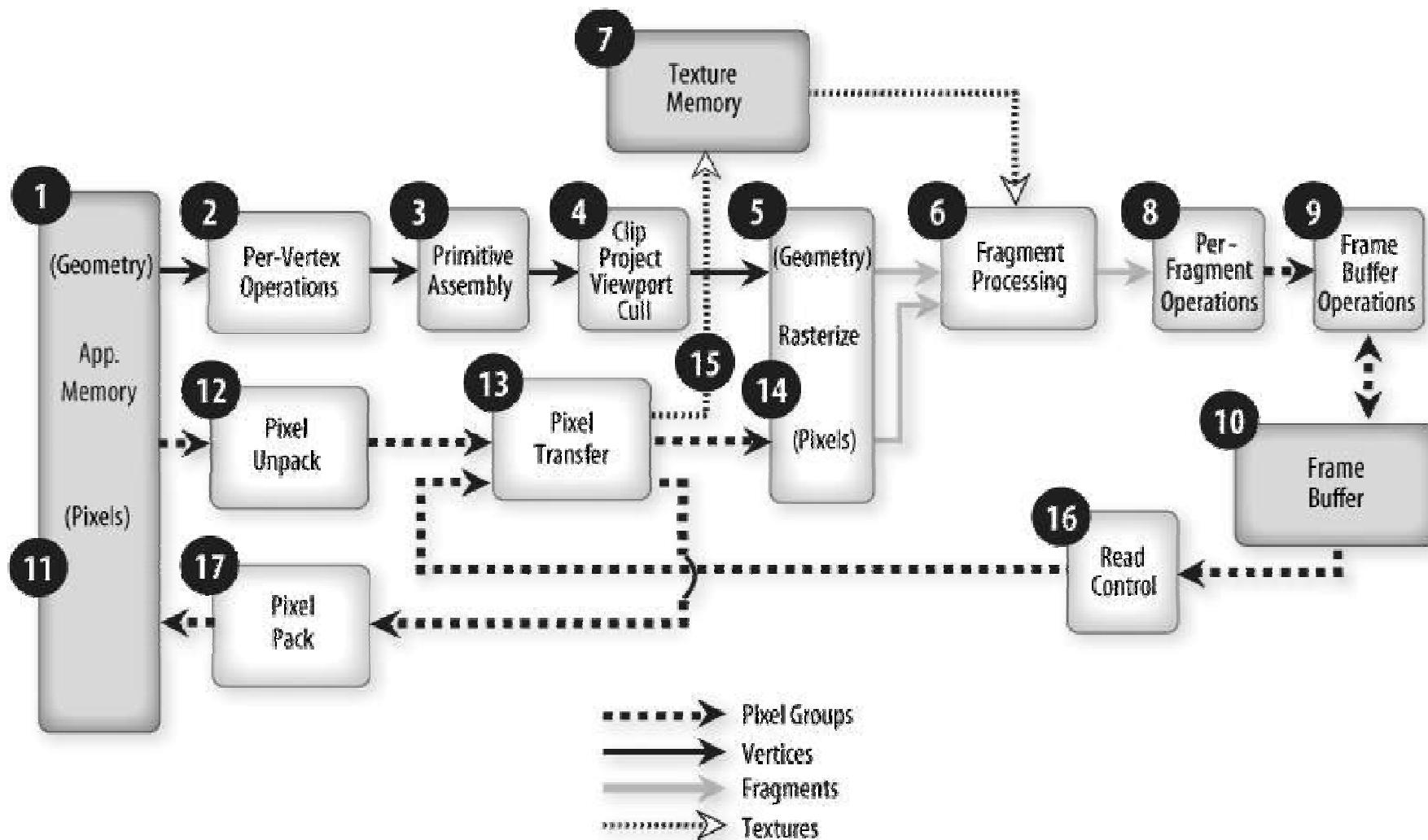


Pipeline OpenGL

5. Rasterització

- Generació dels fragments corresponents a la primitiva retallada.
- Cada fragment té diversos atributs:
 - Window coordinates (x,y,z)
 - Color primari (interpolat si Gouraud)
 - Color secundari (interpolat si Gouraud)
 - Coordenades de textura (interpolades)

Pipeline OpenGL

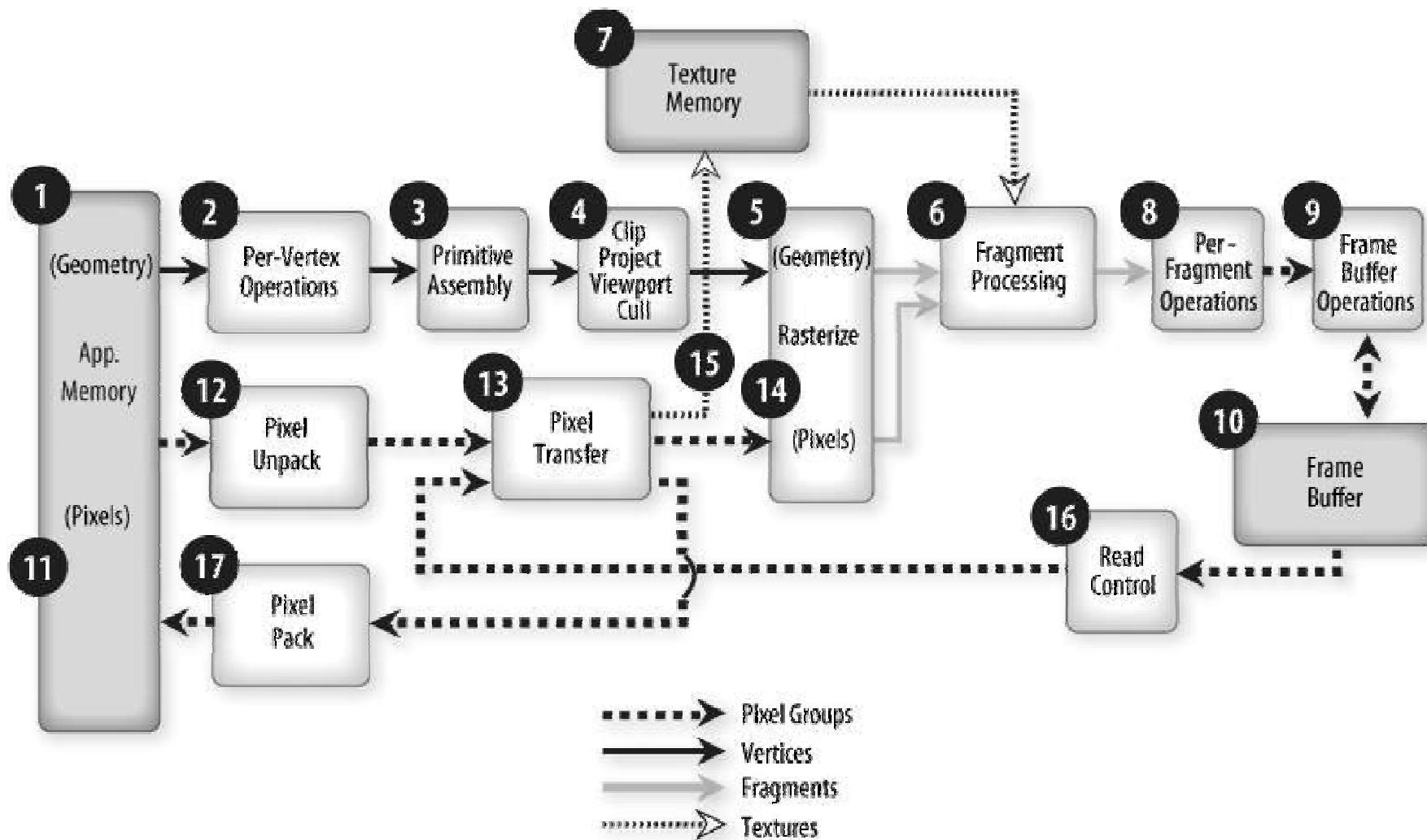


Pipeline OpenGL

6. Fragment processing

- Texture mapping
- Boira (fog)
- Color sum (combinació del color primari i secundari)

Pipeline OpenGL

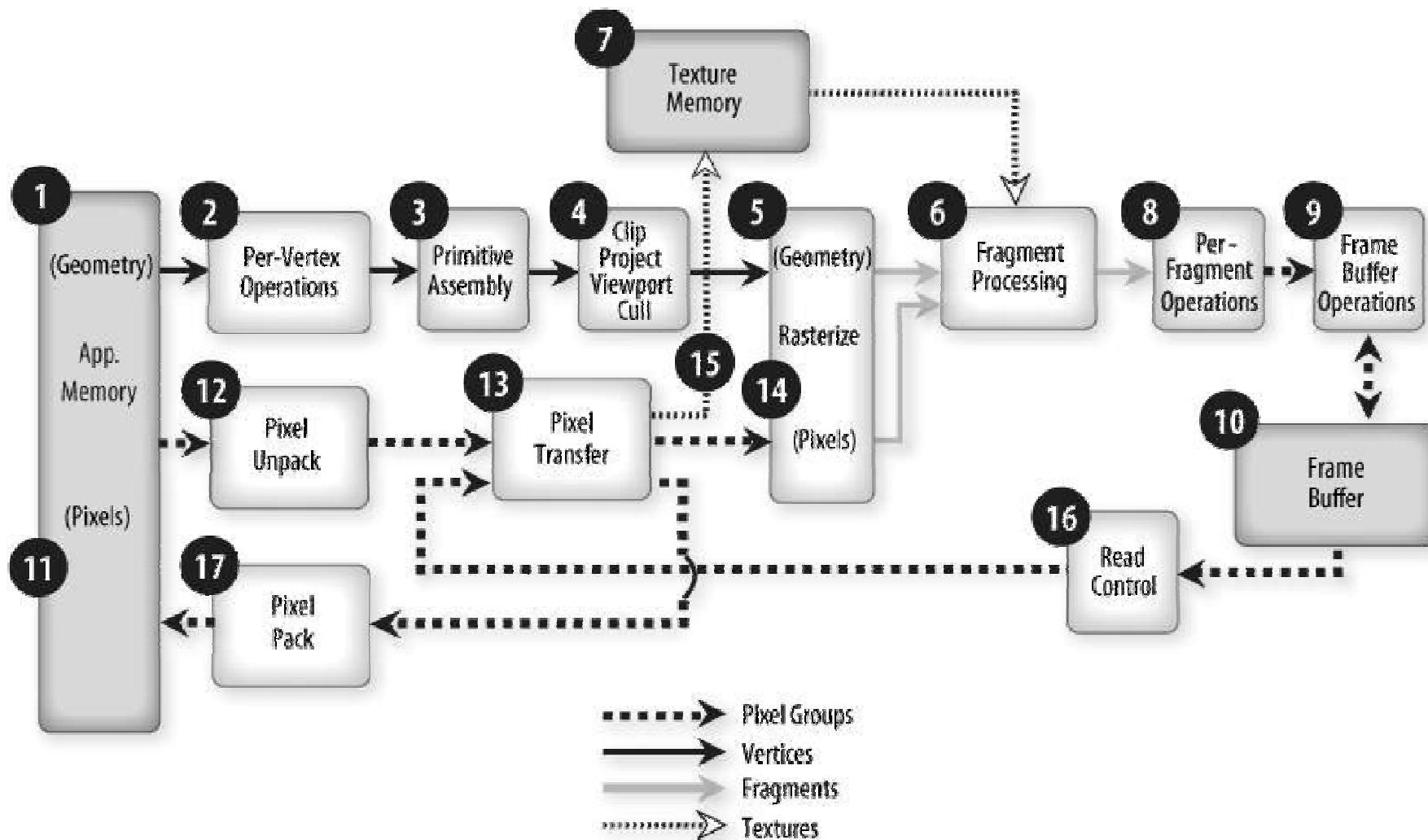


Pipeline OpenGL

8. Per-fragment operations

- Pixel ownership
- Scissor test
- Alpha test
- Stencil test
- Depth test (test Z-buffer)
- Blending
- Dithering
- Logical Ops (glLogicOp)

Pipeline OpenGL



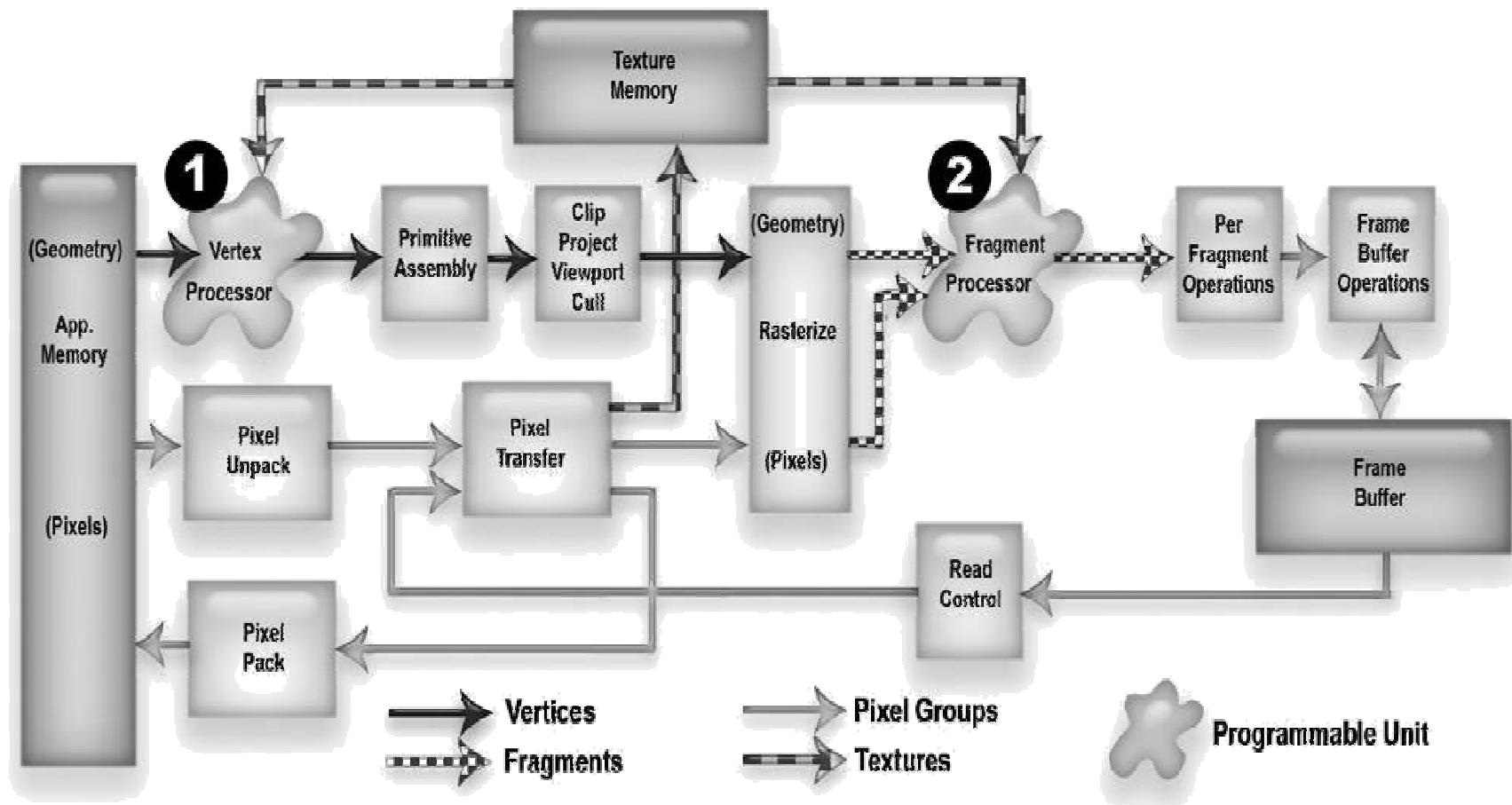
Pipeline OpenGL

9. Frame buffer operations

- Es modifiquen els buffers que s'hagin escollit amb glDrawBuffers
- Es veu afectada per glColorMask, glDepthMask...

PIPELINE PROGRAMABLE

Pipeline programmable



Pipeline programable

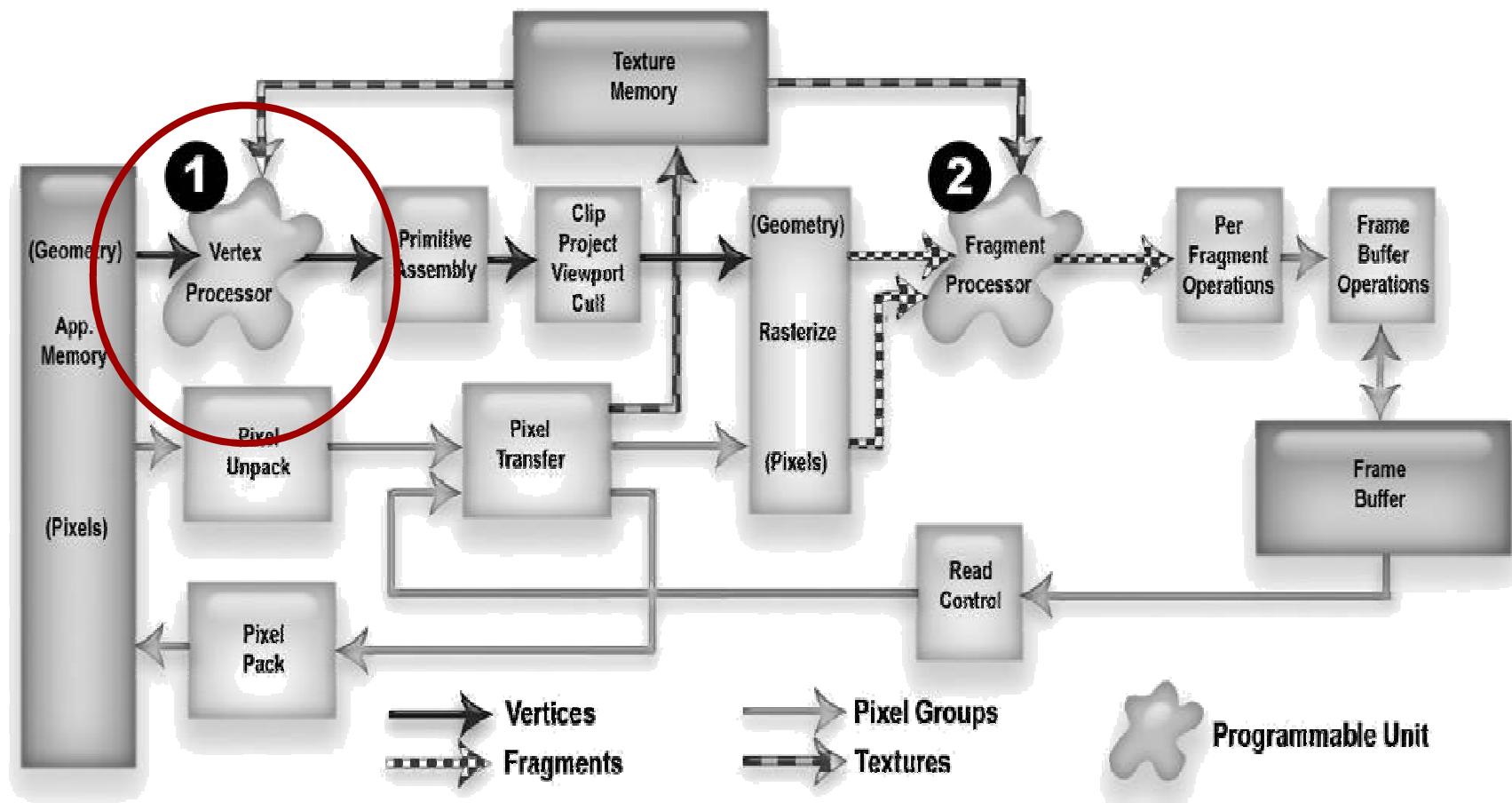
- **Vertex processor:** part de la GPU capaç d'executar un programa per cada vèrtex.
- **Fragment processor:** part de la GPU capaç d'executar un programa per cada fragment.
- **Shader:** codi font d'un programa (o part) per la GPU
 - Vertex shader, fragment shader, geometry shader
- **Program:** executable d'un programa per la GPU
 - Vertex program, fragment program, geometry program

Pipeline programable

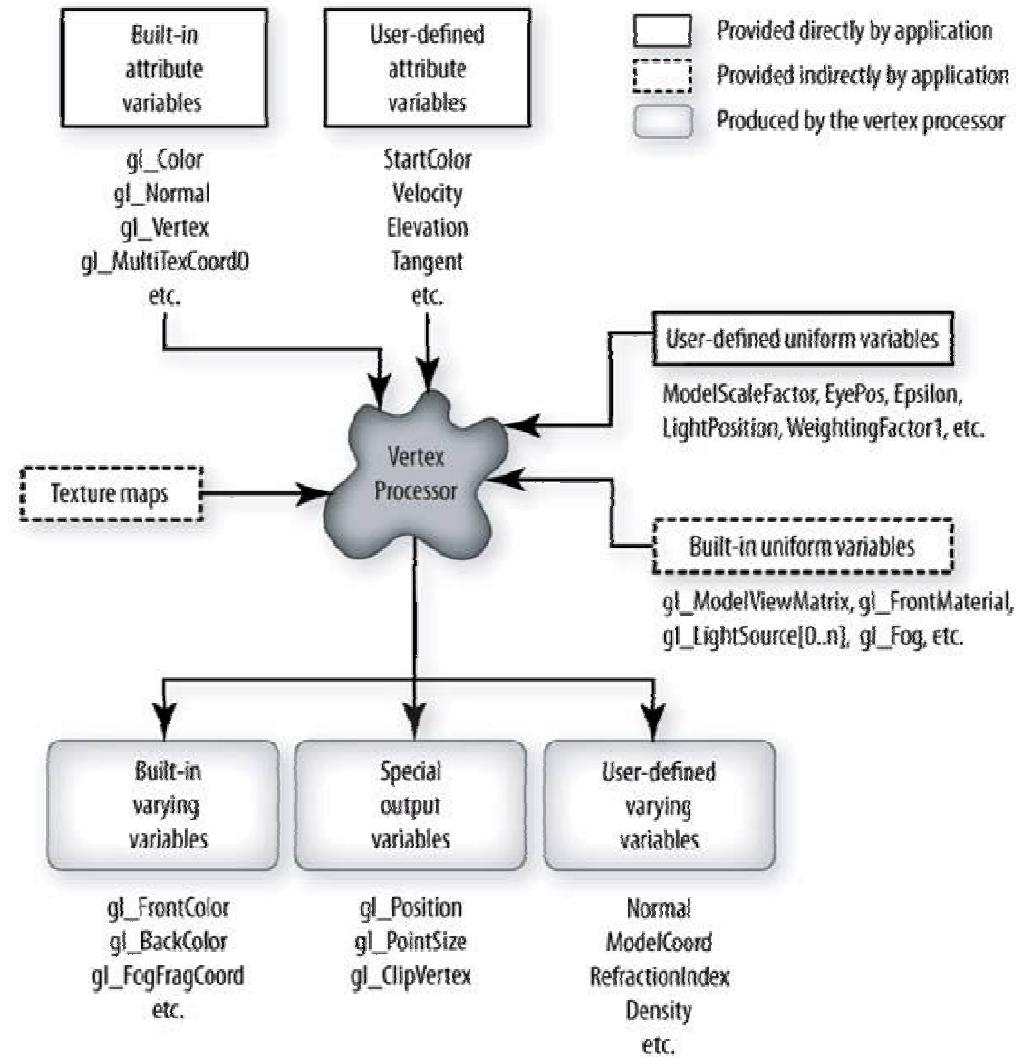
- Quan s'activa un **vertex program**, en lloc d'executar-se les operacions **per-vèrtex** prefixades d'OpenGL, s'executa el vertex program.
- Quan s'activa un **fragment program**, en lloc d'executar-se les operacions **per-fragment** prefixades d'OpenGL, s'executa el fragment program.
- Normalment el vertex/fragment program més senzill haurà de reproduir part de la funcionalitat fixa d'OpenGL.

VERTEX SHADERS

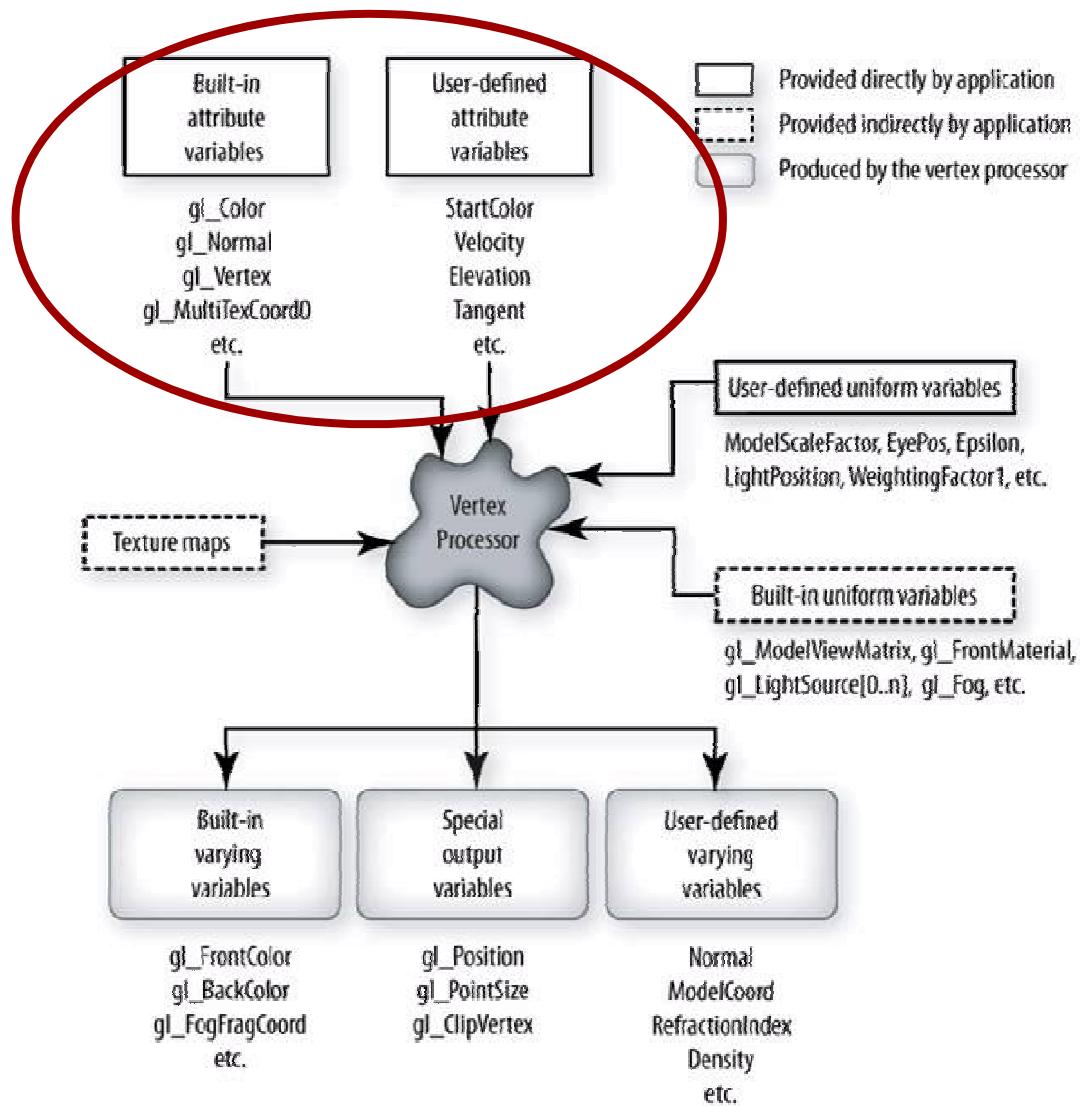
Vertex shaders



Vertex shaders



GLSL Vertex program



Vertex shaders

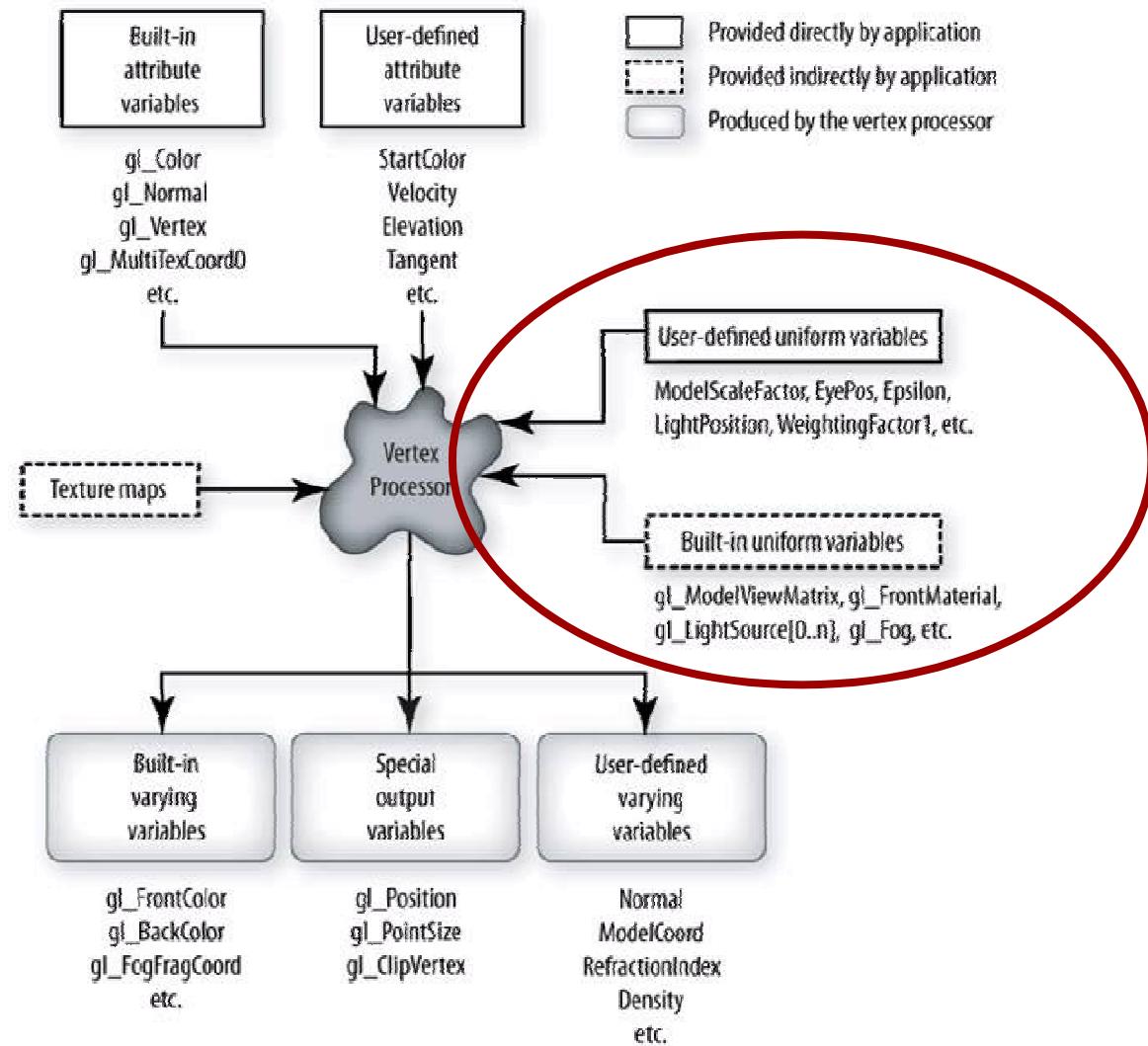
- **Attribute** variables: són variables que representen els *atributs d'un vèrtex*. Poden canviar de valor per cada vèrtex d'una mateixa primitiva.
 - **Built-in** attributes: atributs predefinits (no cal declarar-los)
 - Des de l'aplicació s'envien amb glColor, glNormal...
 - Des del shader s'accedeixen 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 glBindAttribLocation.
 - Des del shader s'accedeixen amb un nom arbitrari definit per l'usuari: velocitat, etc.

Vertex shaders

Llista **atributs predefinits** pels vèrtexs:

vec4 gl_Color;	glColor()
vec4 gl_SecondaryColor;	glSecondaryColor()
vec3 gl_Normal;	glNormal();
vec4 gl_Vertex;	glVertex();
vec4 gl_MultiTexCoord0;	glTexCoord();
vec4 gl_MultiTexCoord1; // . . . up to N	
float gl_FogCoord;	glFogCoord();

Vertex shaders



Vertex 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 variables:** són variables d'estat OpenGL
 - Des del shader s'accedeixen amb `gl_ModelViewMatrix`, `glLightSource[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

Llista variables **uniform predefinides** més importants:

```
// Matrix state
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;

// Derived matrix state that provides inverse and transposed versions
uniform mat3 gl_NormalMatrix; // transpose of the inverse of 3x3 MV
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
```

Vertex shaders

```
// Material State
struct gl_MaterialParameters
{
    vec4 emission;
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    float shininess;
};

uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Vertex shaders

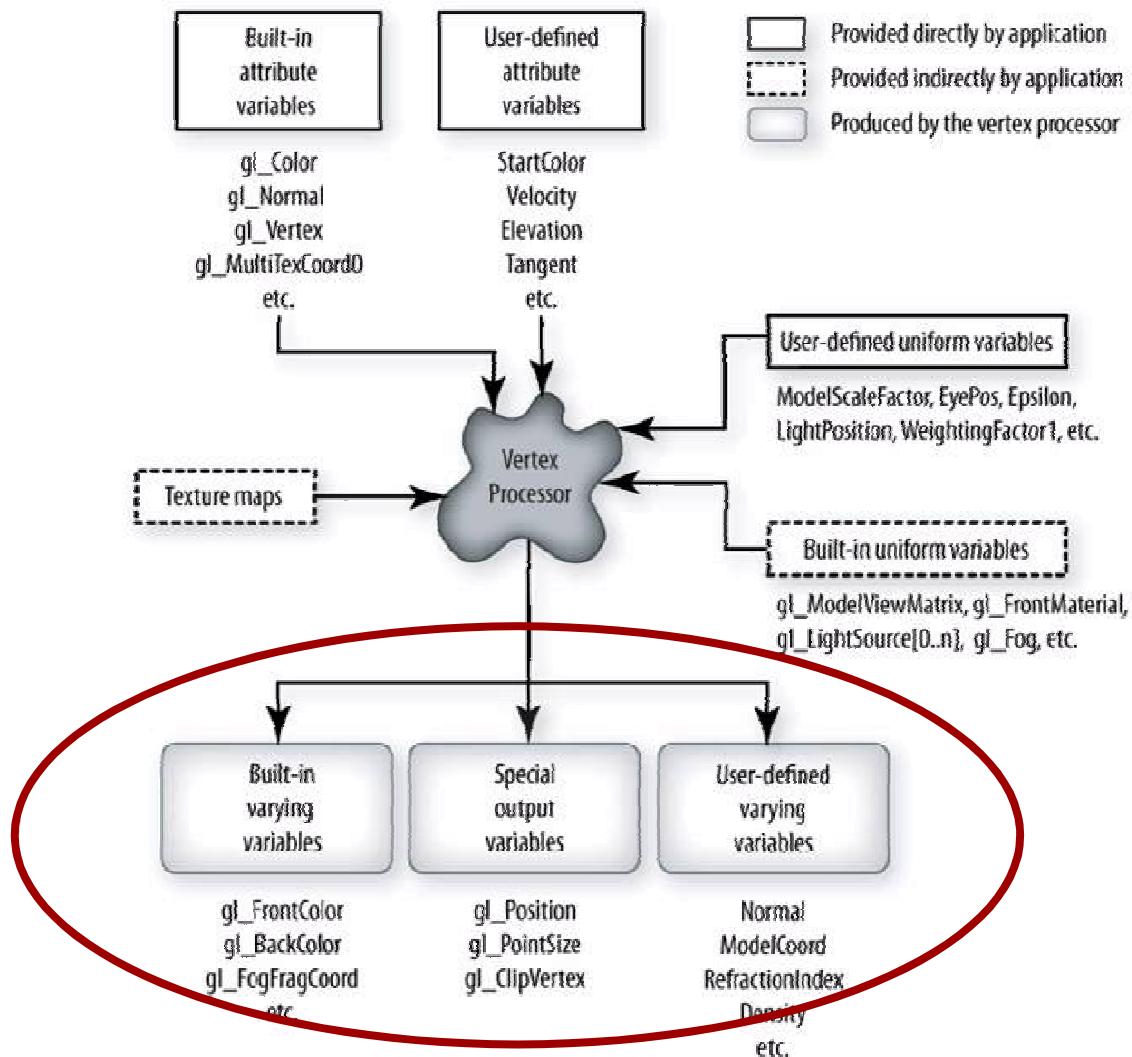
```
// Light State
struct gl_LightSourceParameters
{
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
    vec4 position;
    vec3 spotDirection;
    ...
};

uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

Vertex shaders

```
// Texture Environment and Generation  
uniform vec4 gl_TextureEnvColor[gl_MaxTextureUnits];  
uniform vec4 gl_EyePlaneS[gl_MaxTextureCoords];  
uniform vec4 gl_EyePlaneT[gl_MaxTextureCoords];  
...  
uniform vec4 gl_ObjectPlaneS[gl_MaxTextureCoords];  
uniform vec4 gl_ObjectPlaneT[gl_MaxTextureCoords];  
...
```

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

Vertex shaders

Llista variables **varying** predefinides:

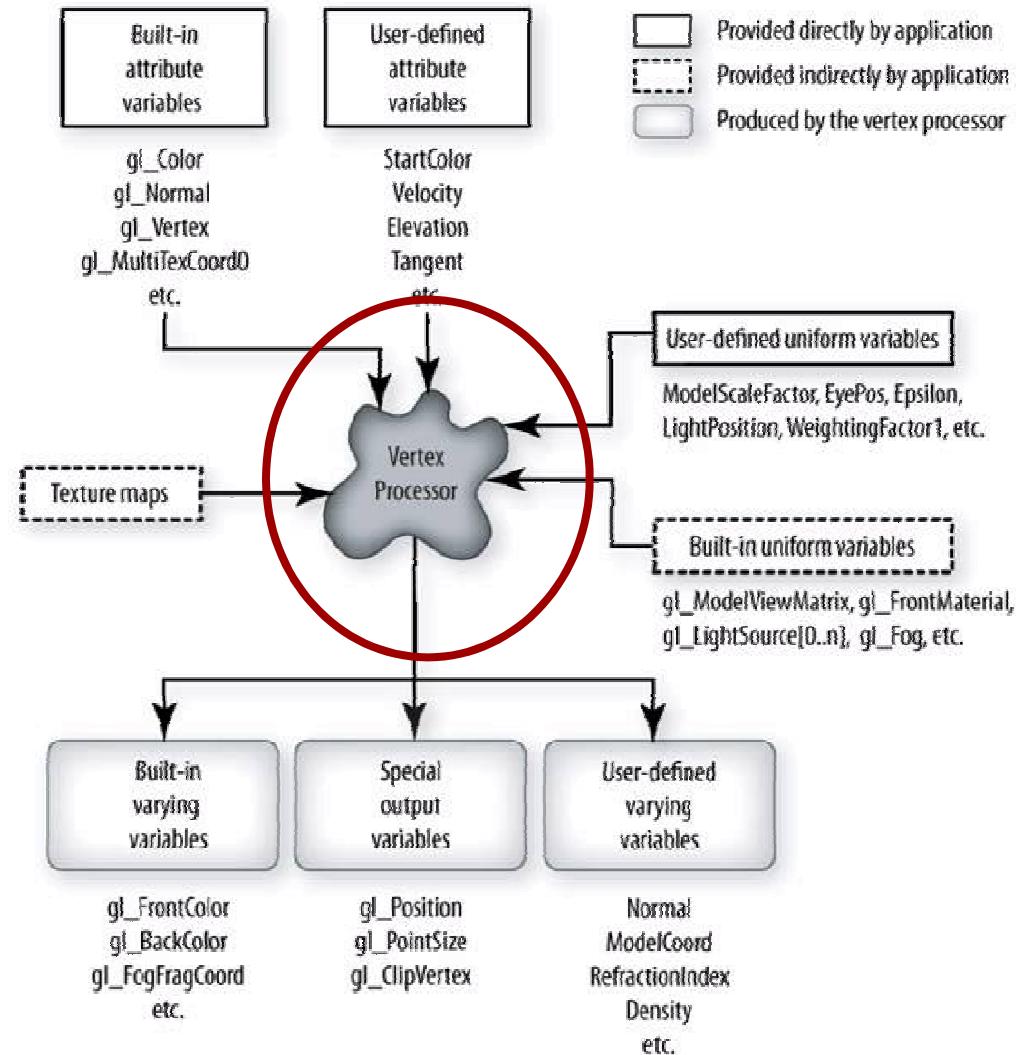
```
varying vec4 gl_FrontColor;  
varying vec4 gl_BackColor;  
varying vec4 gl_FrontSecondaryColor;  
varying vec4 gl_BackSecondaryColor;  
varying vec4 gl_TexCoord[];  
varying float gl_FogFragCoord;
```

- Els valors escrits a `gl_FrontColor`, `gl_BackColor`, ... s'usen, un cop es determina si la primitiva es backface o no, per calcular `gl_Color` and `gl_SecondaryColor` (entrada pel fragment shader).

Vertex shaders

- **Special output** variables: són variables que representen valors que ha de calcular el vertex program.
- 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;
// attribute qualified variables are typically changed per vertex
attribute float VertexTemp;
// varying variables communicate from the vertex to fragment
varying float Temperature;
void main()
{
    Temperature = (VertexTemp - CoolestTemp) / TempRange;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

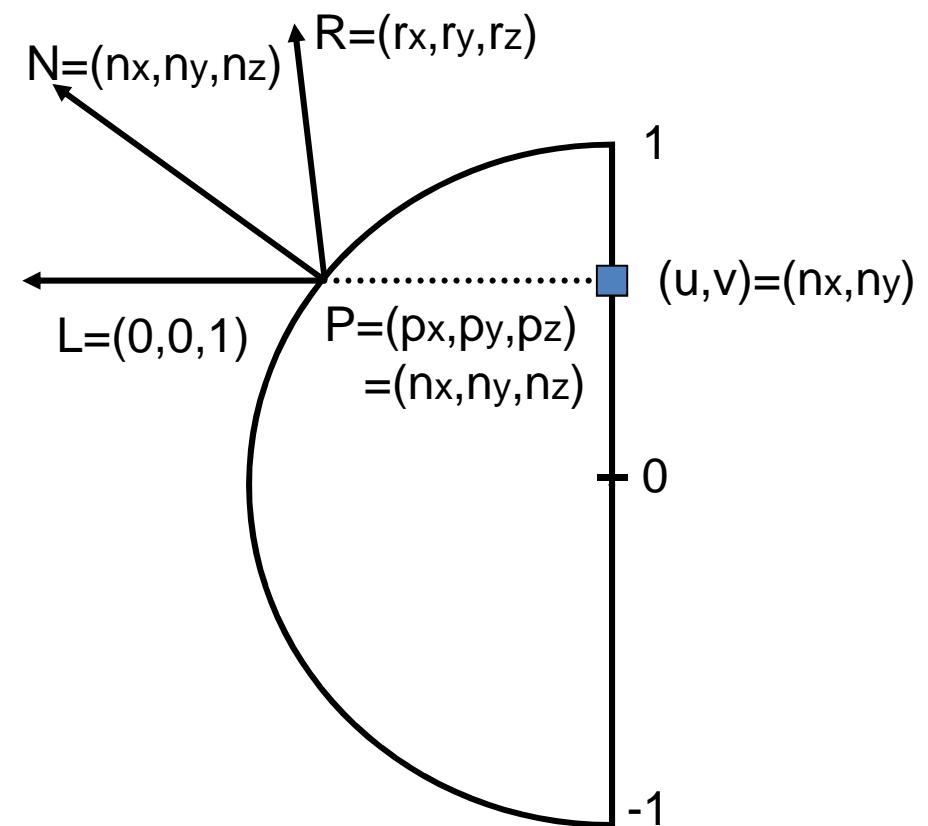
Exemples de VS

- Il·luminació bàsica dependent de la normal
- Usar la normal com a color
- Deformació en el sentit de la normal
- Projectar suavament sobre $Y=0$
- Gradient de color

Sphere mapping (repàs)

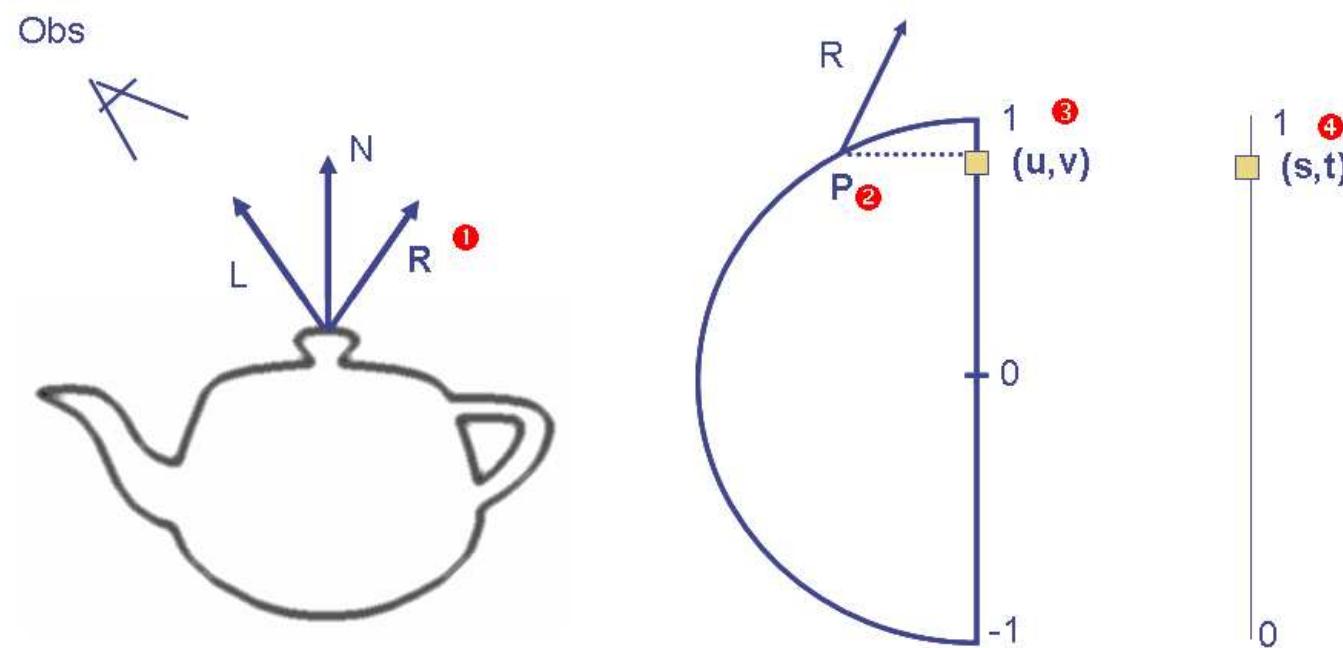
Relació vector reflectit i entre coords (u,v):

$$R = (2n_z n_x, 2n_z n_y, 2n_z^2 - 1)$$



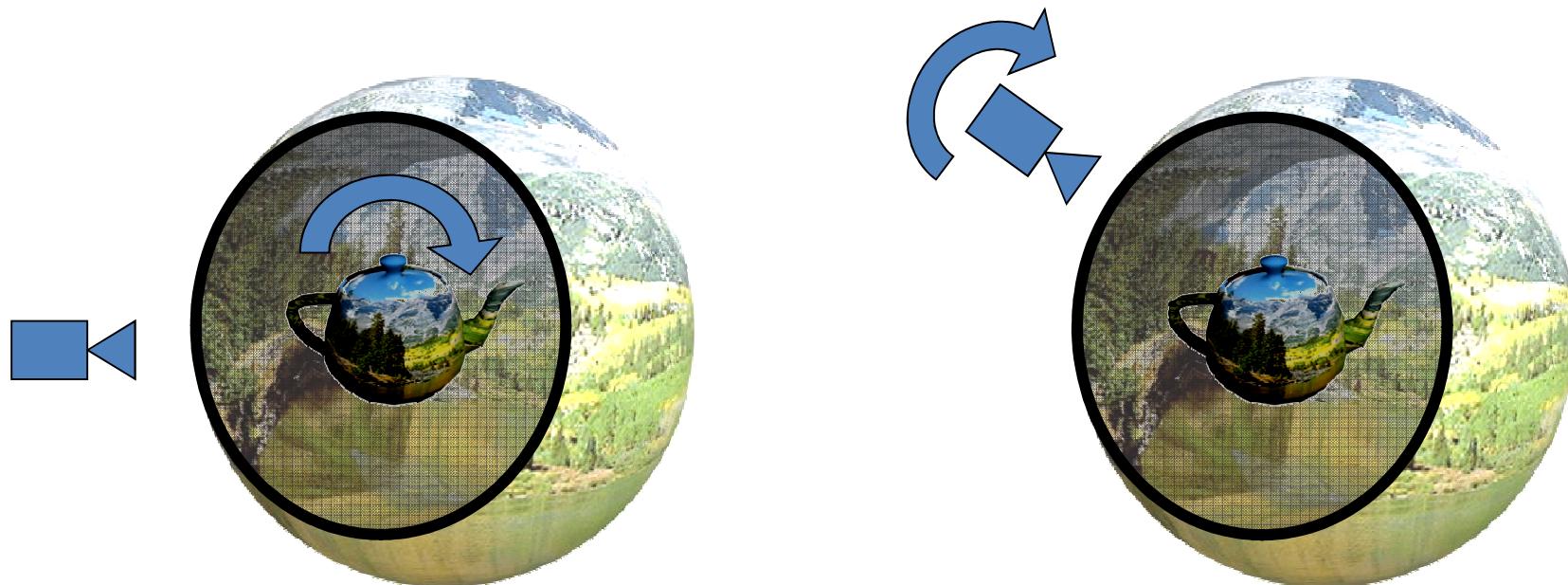
Sphere mapping (repàs)

1. Calcular R a partir de N i L
2. $z = \sqrt{((R.z+1)/2)}$, $u = R.x/(2*z)$, $v = R.y/(2*z)$
3. $s = (u+1)/2$, $t = (v+1)/2$



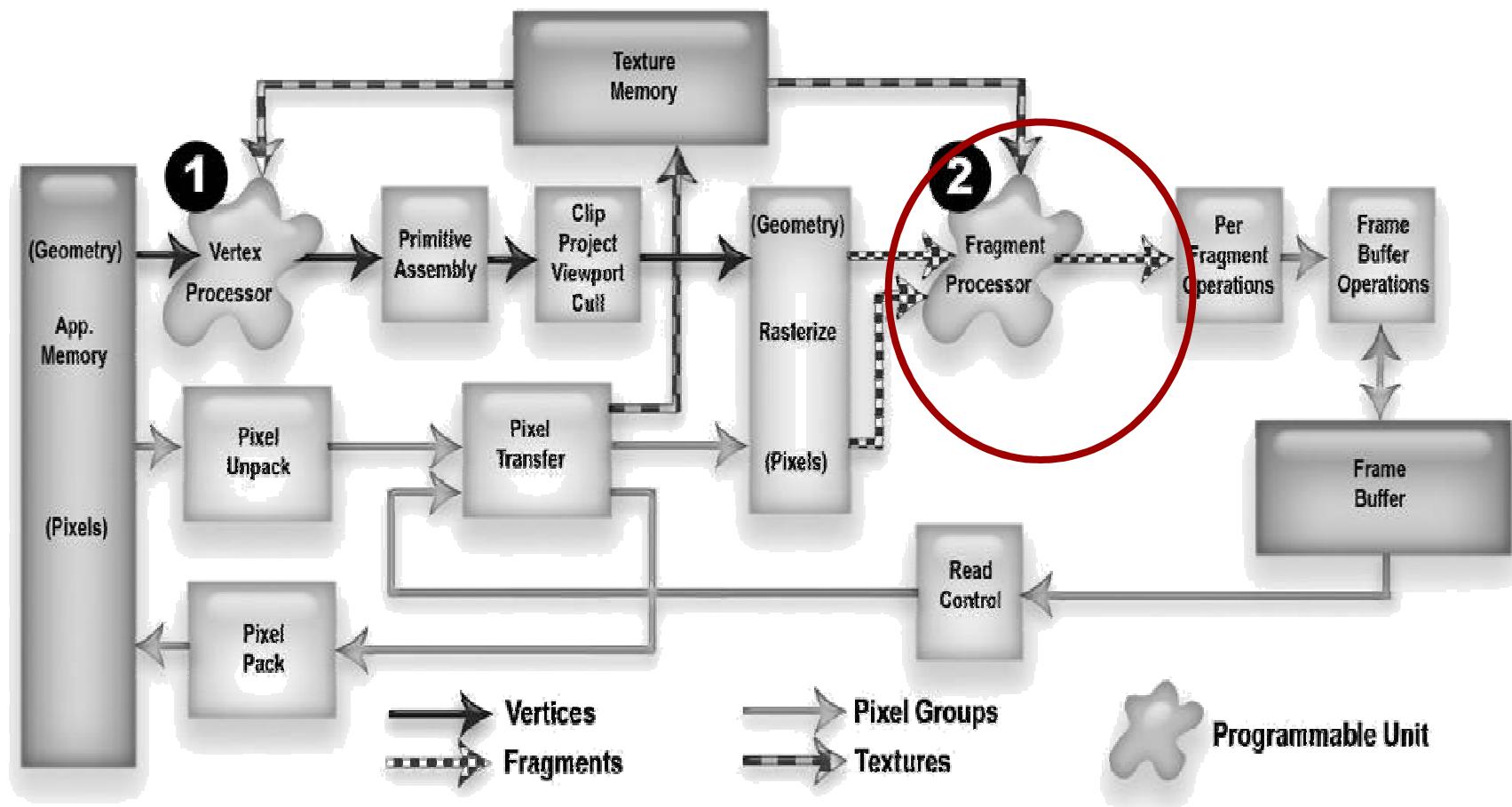
Eye/world coordinates

- Càlcul amb vèrtex, normal en eye coords
 - El entorn serà estàtic respecte la càmera
 - L'objecte sempre reflecteix “la mateixa part” de l'entorn
- Càlcul amb vèrtex, normal en world coords
 - El entorn serà dinàmic respecte la càmera
 - L'objecte reflecteix diferents parts de l'entorn

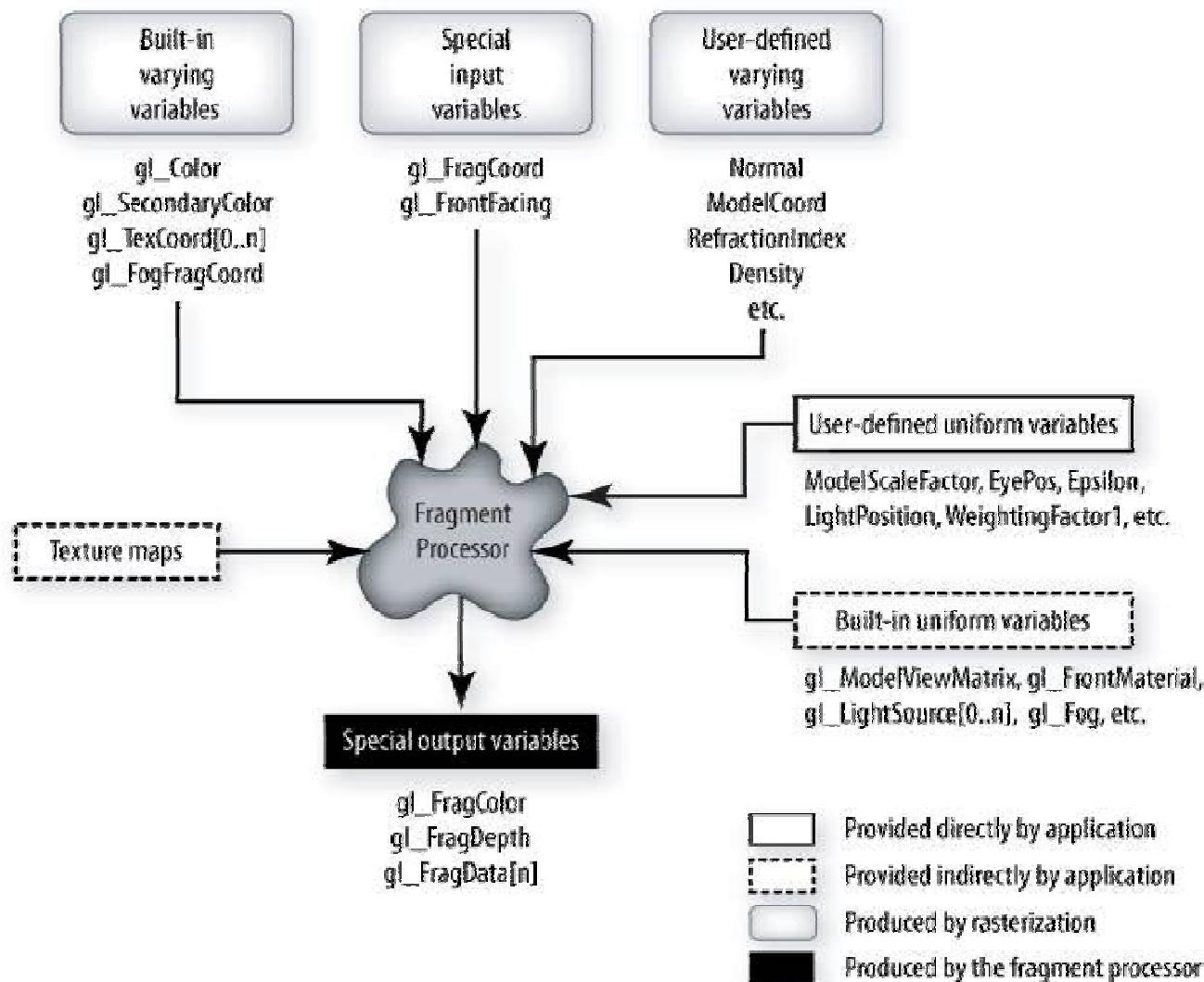


FRAGMENT SHADERS

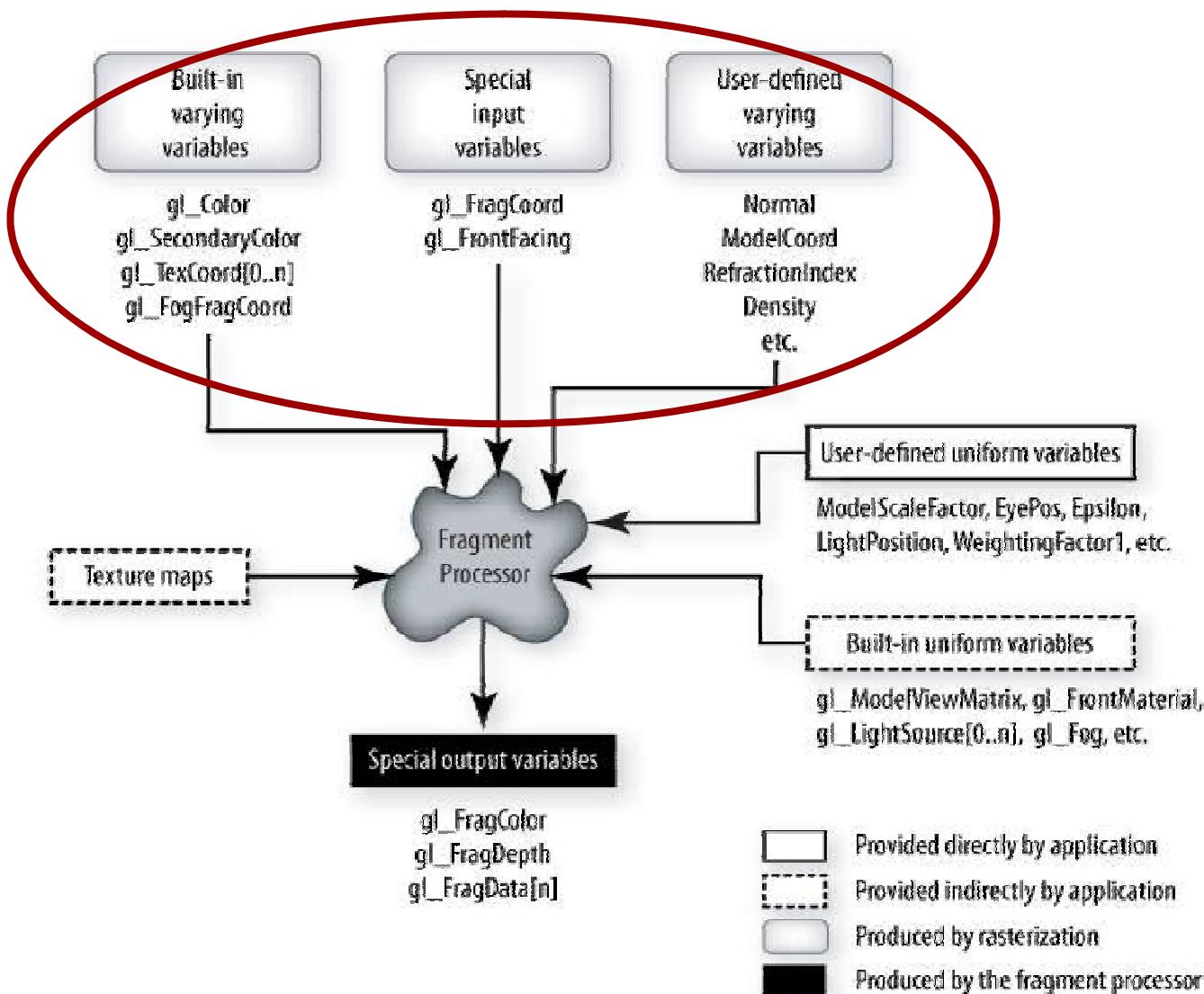
Fragment shaders



Fragment shaders



Fragment shaders



Fragment shaders

- **Varying variables:** com s'ha dit abans, són variables que es calculen al vertex processor i s'envien al fragment processor. Els valors que arriben per cada fragment són el resultat d'interpolar els valors calculats a cada vèrtex.
- Llista varying variables:

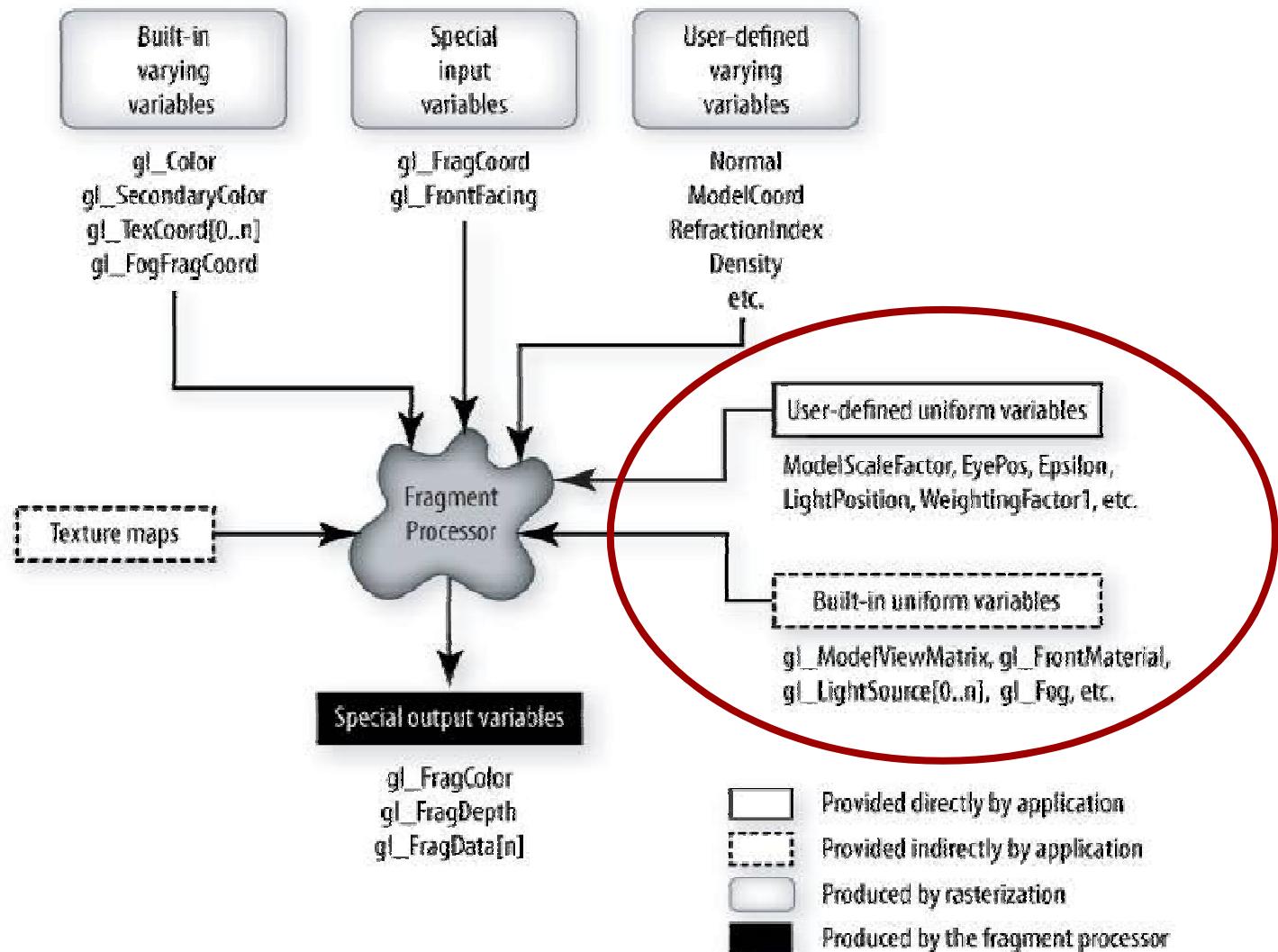
```
varying vec4 gl_Color;  
varying vec4 gl_SecondaryColor;  
varying vec4 gl_TexCoord[];  
varying float gl_FogFragCoord;
```

Fragment shaders

- **Special input** variables: Són valors del fragment que calcula OpenGL de forma automàtica, i que es poden llegir al fragment program:

```
vec4 gl_FragCoord; // coords fragment  
bool gl_FrontFacing; // true si frontface
```

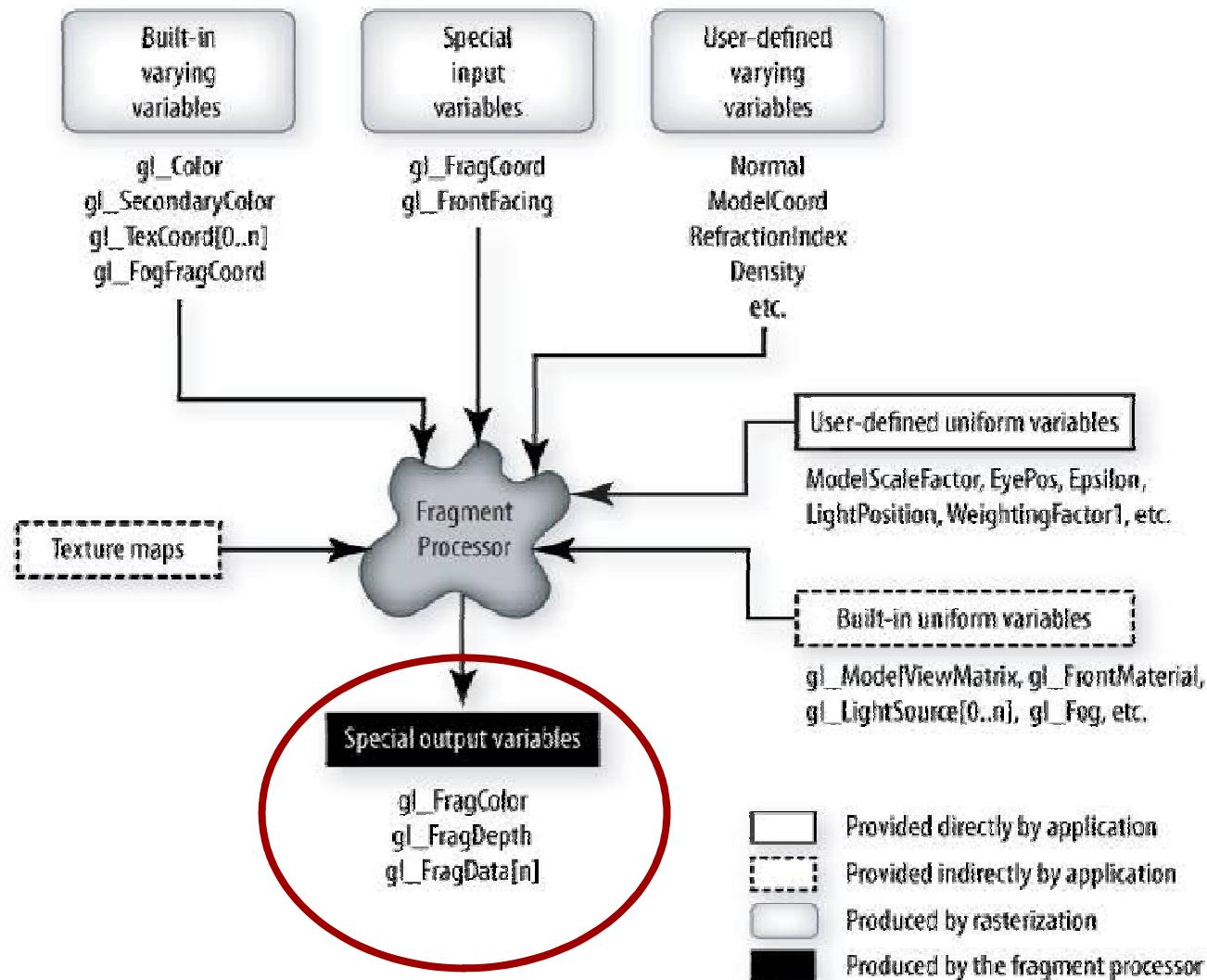
Fragment shaders



Fragment shaders

- **Uniform variables:** són variables que canvien amb poca freqüència. Com a molt poden canviar un cop per cada primitiva (però per cada vèrtex de la primitiva).
 - **Built-in variables:** corresponen a variables d'estat OpenGL
 - Des del shader s'accedeixen amb `gl_ModelViewMatrix`, `glLightSource[0..n]`, etc
 - Són les mateixes pels vertex programs i pels fragment programs.
 - **User-defined variables:** variables definides per l'usuari
 - Des de l'aplicació s'envien amb `glUniformi` es lliguen a un nom amb `glGetUniformLocation`.
 - Des del shader s'accedeixen amb un nom arbitrari definit per l'usuari: `EyePos`, etc.

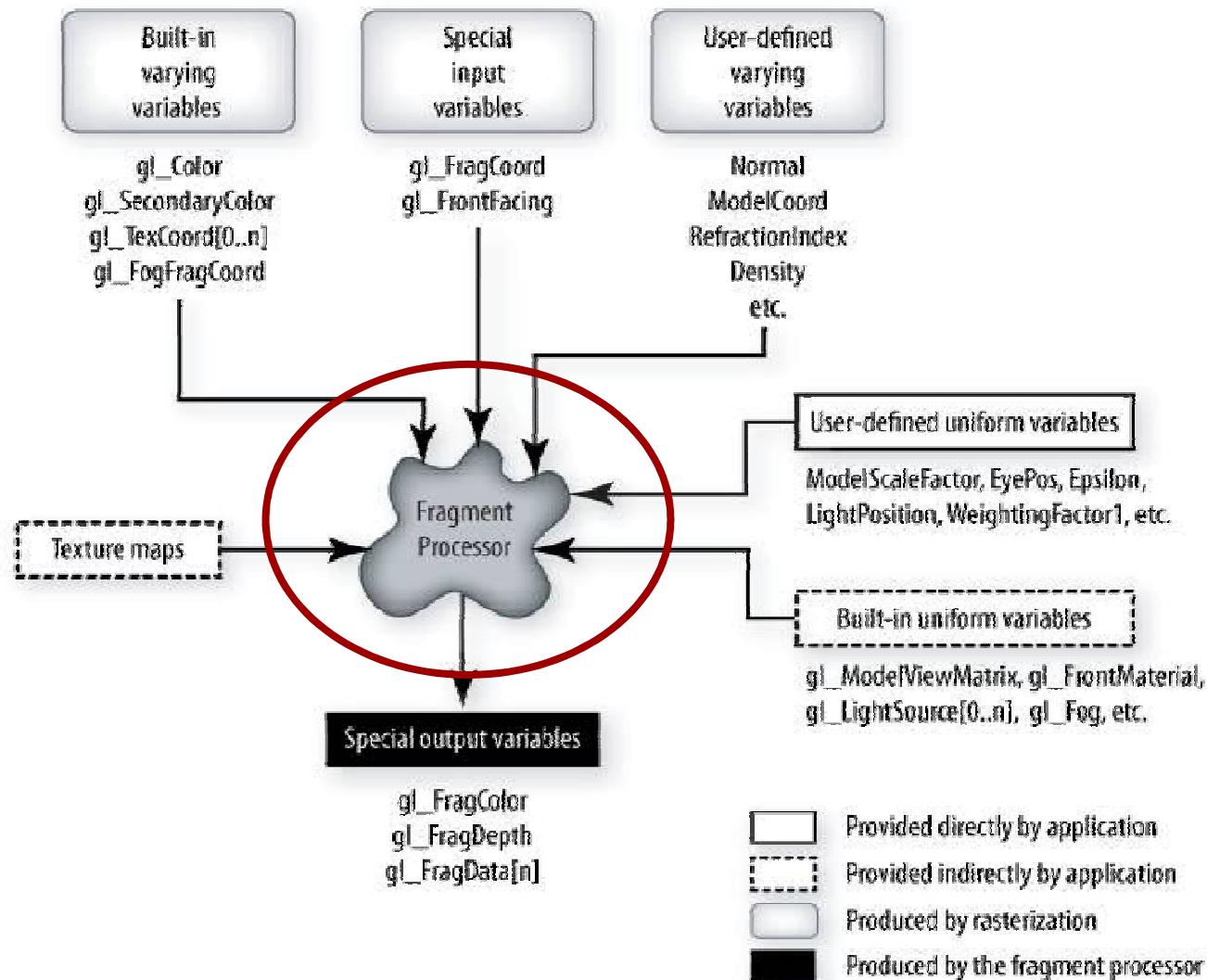
Fragment shaders



Fragment shaders

- **Special output** variables: són els valors que ha de calcular el fragment shader, i que continuaran la resta del pipeline d'OpenGL:
 - `vec4 gl_FragColor`: color final del fragment (abans de blending)
 - `float gl_FragDepth`: depth final del fragment (s'usarà en el test de z-buffer)
 - `vec4 gl_FragData[]`: usado para Multiple Render Targets (`glDrawBuffers`)

Fragment shaders



Fragment shaders

- Un fragment program s'executa per cada fragment que produeix cada primitiva.
- Les tasques habituals d'un fragment program 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 program:
 - 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;  
// Temperature 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);  
}
```

Exemples de FS

- Sense access a textura
 - Toon shading
 - “Textura” procedural (Stripes)
 - *Gradient (versió per-fragment)*
- Amb accés a textura
 - Sphere map
 - Normal Mapping

API OPENGL PER DEFINIR SHADERS

API per definir shaders

- Creació i compilació de shaders individuals:
 - `glCreateShader`, `glShaderSource`, `glCompileShader`
- Creació i muntatge de programes:
 - `glCreateProgram`, `glAttachShader`, `glLinkProgram`
- Activació de programes:
 - `glUseProgram` – instal·la el vertex program, el fragment program, o tots dos.

Exemple complet: vertex shader

```
attribute float vertexTemp;  
varying float temperature;  
void main()  
{  
    temperature = vertexTemp;  
    gl_Position=gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

Exemple complet: frag shader

```
uniform bool reverse;
varying float temperature;
void main()
{
    vec3 c;
    float t = temperature;
    if (reverse) t = 1.0 - temperature;
    if (t <= 0.25)    c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 1.0), (t      ) / 0.25);
    else if (t <= 0.5) c = mix(vec3(0.0, 1.0, 1.0), vec3(0.0, 1.0, 0.0), (t-0.25)/0.25);
    else if (t <= 0.75) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (t-0.50)/0.25);
    else if (t <= 1.0) c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (t-0.75)/ 0.25);
    gl_FragColor = vec4(c, 1.0);
```

Exemple complet: shader.h

```
#include "glew.h"

class ShaderTest {
public:
    ShaderTest();
    ~ShaderTest();

    void render();

private:
    GLuint vertexShaderID;
    GLuint fragmentShaderID;
    GLuint programID;
};
```

Exemple complet: shader.cpp

```
ShaderTest::ShaderTest()
{
    // Crear vertex shader
    GLchar *sourceVertexShader[] =
    {
        "attribute float vertexTemp; "
        "varying float temperature; "
        "void main() {"
        "    temperature = vertexTemp; "
        "    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex; "
        "}"
    };
    vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShaderID, 1, (const GLchar**)sourceVertexShader, 0);
    glCompileShader(vertexShaderID);

    <continua...>
}
```

Exemple complet: shader.cpp

```
...
// Crear fragment shader
GLchar *sourceFragmentShader[] = {
    "uniform bool reverse;
    "varying float temperature;
    "void main() {
        " vec3 c;
        " float t = temperature;
        " if (reverse) t = 1.0 - temperature;
        " if (t <= 0.25)    c = mix(vec3(0.0, 0.0, 1.0), vec3(0.0, 1.0, 1.0), (t      ) / 0.25);
        " else if (t <= 0.5)  c = mix(vec3(0.0, 1.0, 1.0), vec3(0.0, 1.0, 0.0), (t - 0.25) / 0.25);
        " else if (t <= 0.75) c = mix(vec3(0.0, 1.0, 0.0), vec3(1.0, 1.0, 0.0), (t - 0.50) / 0.25);
        " else if (t <= 1.0)  c = mix(vec3(1.0, 1.0, 0.0), vec3(1.0, 0.0, 0.0), (t - 0.75) / 0.25);
        " gl_FragColor = vec4(c, 1.0); "
    };
}

fragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderID, 1, (const GLchar**)sourceFragmentShader, 0);
glCompileShader(fragmentShaderID);
<continua...>
```

Exemple complet: shader.cpp

```
...
// Crear programa amb els dos shaders anteriors

programID = glCreateProgram();
glAttachShader(programID, vertexShaderID);
glAttachShader(programID, fragmentShaderID);
glLinkProgram(programID);
}
```

Exemple complet: shader.cpp

```
...
// Crear programa amb els dos shaders anteriors

programID = glCreateProgram();
glAttachShader(programID, vertexShaderID);
glAttachShader(programID, fragmentShaderID);
glLinkProgram(programID);

}

ShaderTest::~ShaderTest()
{
    glDeleteShader(vertexShaderID);
    glDeleteShader(fragmentShaderID);
    glDeleteProgram(programID);
}
```

Exemple complet: shader.cpp

```
void ShaderTest::render()
{
    glUseProgram(programID);

    GLint vertexTemp = glGetAttribLocation(programID, "vertexTemp");
    GLint reverse = glGetUniformLocation(programID, "reverse");

    glUniform1i(reverse, 1);

    glBegin(GL_POLYGON);
    glVertexAttrib1f(vertexTemp, 0.5f); glVertex3f(0,0,0);
    glVertexAttrib1f(vertexTemp, 1.0f); glVertex3f(0, 0, 8192);
    glVertexAttrib1f(vertexTemp, 0.0f); glVertex3f(8192,0,0);
    glEnd();

    glUseProgram(0);
}
```

ASPECTES DEL LLENGUATGE GLSL

Aspectes de GLSL

- Sintaxi similar 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)
 - Calificadors **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 soporta: 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 column
```

```
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;
```

Tipus en GLSL: arrays

- Es poden definir arrays de qualsevol tipus.
- Quan es passen com a paràmetre es comporta com si es copiés tot l'array.

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

```
MyLight lights[2] = MyLight[2] (vec3(1.0, 0.0, 0.0), vec3(1.0, 1.0, 1.0));
```

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)

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;
```

```
}
```

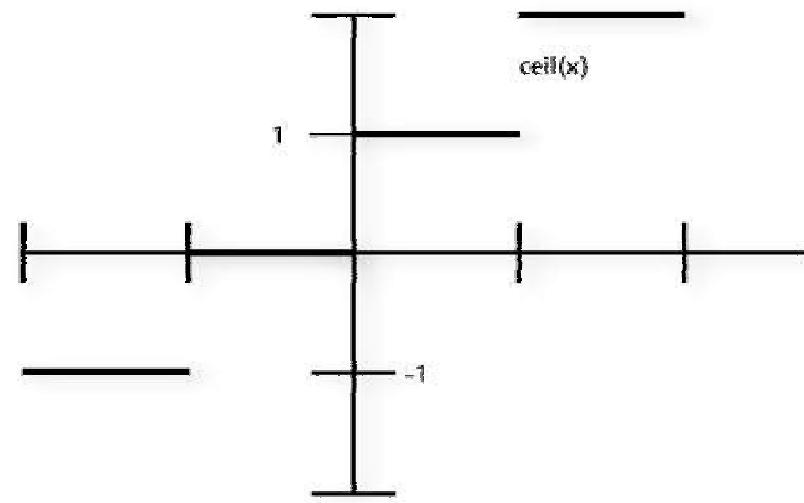
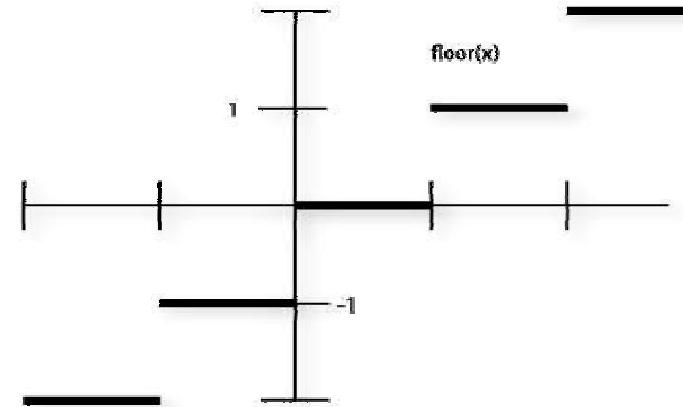
Funcions predefinides

Funcions matemàtiques:

- $\sin()$, $\cos()$, $\tan()$,
 $\text{asin}()$, $\text{acos}()$, $\text{atan}()$
- $\text{radians}()$, $\text{degrees}()$
- $\text{pow}()$, $\text{exp}()$, $\text{exp2}()$,
 $\text{log}()$, $\text{log2}()$, $\text{sqrt}()$
- $\text{abs}()$, $\text{floor}()$, $\text{ceil}()$

$\text{floor}(8.2) \rightarrow 8.0$

$\text{ceil}(8.2) \rightarrow 9.0$

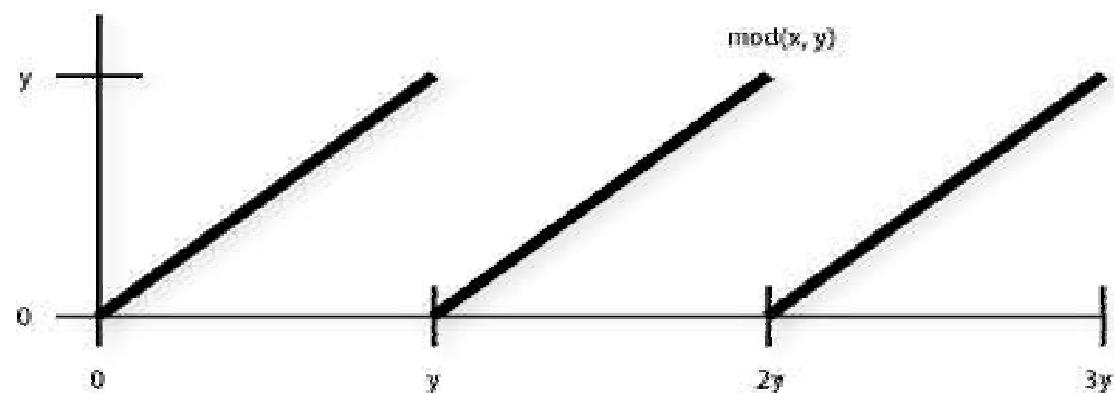
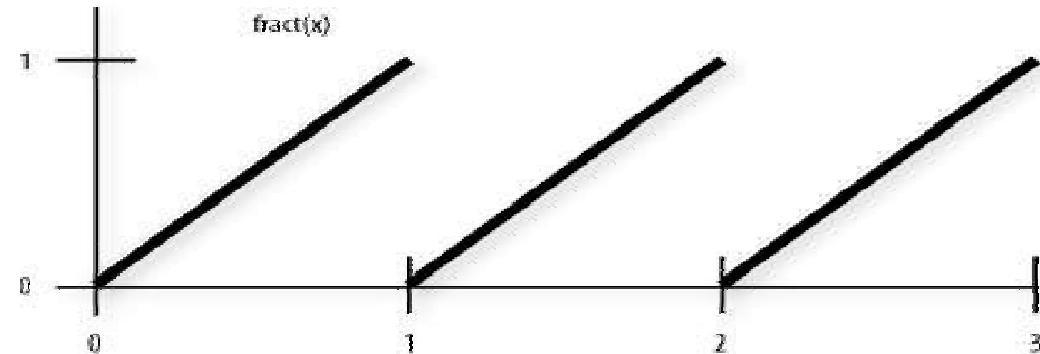


Funcions predefinides

Funcions matemàtiques:

- fract(), mod()

$$\text{fract}(8.4) \rightarrow 0.4$$

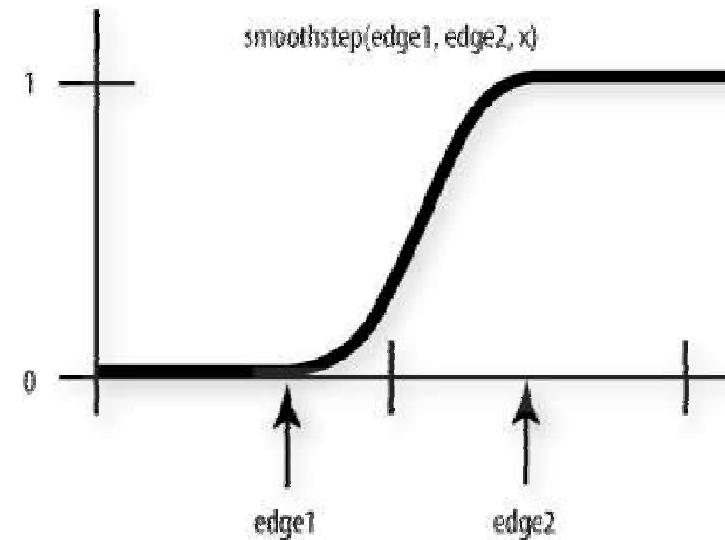
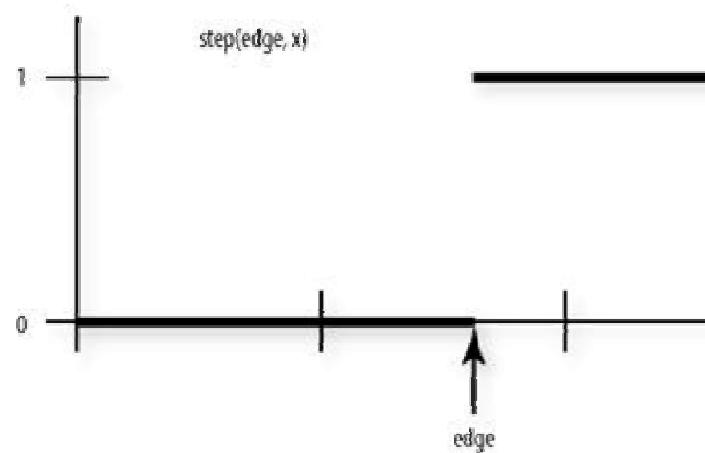
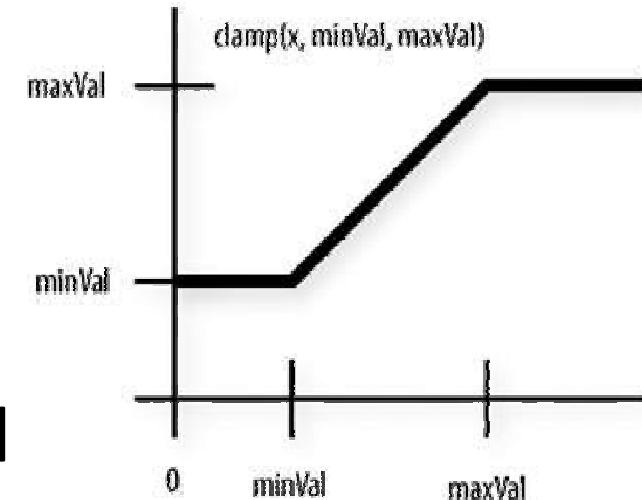


Funcions predefinides

`clamp()`, `mix()`, `smoothstep()`

`c=clamp(c, 0.0, 1.0)`

`mix(v, q, 0.3) \rightarrow 0.7*v + 0.3*q`



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 l, vec N); // sentit de l!
- vec refract(vec l, vec N, float mu);
- Producte de matrius i vectors amb matrius (ex. M*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);`

Operadors en GLSL

Operator	Description
[]	Index
.	Member selection and swizzle
++ --	Postfix increment/decrement
++ --	Prefix increment/decrement
- !	Unary negation and logical not
* /	Multiply and divide
+ -	Add and subtract
< > <= >=	Relational
== !=	Equality
&&	Logical and
^^	Logical exclusive or
	Logical inclusive or
?:	Selection
= += -= *= /=	Assignment

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.rgb;           // vec3  
v.rgba;          // vec4  
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
```

Més exemples

- Lattice (fragment shader)
- NormalMap + NormalMapping
- Il·luminació amb OpenGL (Phong, Blinn)
- Cube mapping
- Sky mapping
- Fresnel