Universidad de Huelva

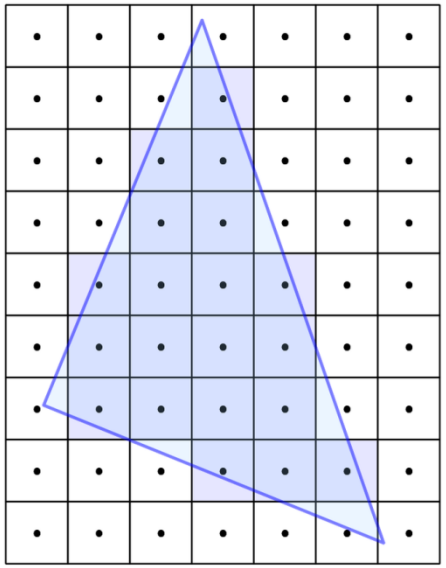
Tema 6

Color e iluminación

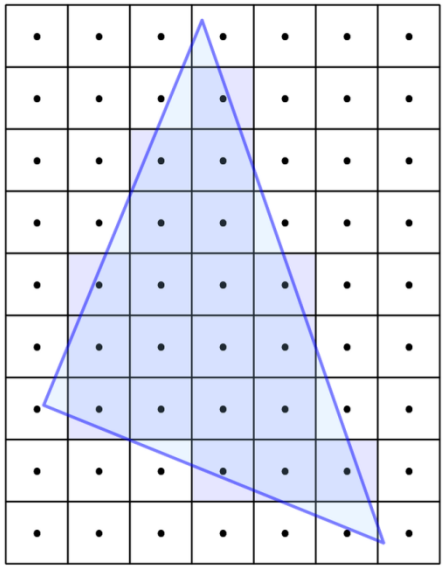
Índice

Índice

- Una vez calculadas las posiciones de los vértices en coordenadas homogéneas, el proceso de culling y clipping permite eliminar los vértices y primitivas que se encuentren fuera del volumen Clip.
- A continuación se transforman las posiciones a coordenadas *viewport*, considerando el número de píxels de altura y anchura de la imagen.
- El proceso de rasterización continua calculando cuales son los píxeles a dibujar para cada primitiva. Para ello básicamente se estudia si el centro del pixel se encuentra dentro o fuera del triángulo calculado en coordenadas *viewport*.



- El FragmentShader se ejecuta para cada uno de los píxeles seleccionados en la etapa de rasterización.
- El objetivo principal del FragmentShader es calcular el color del píxel. Sin embargo no existe una variable de salida predefinida para este valor, por lo que se toma como color la salida definida en la posición 0.



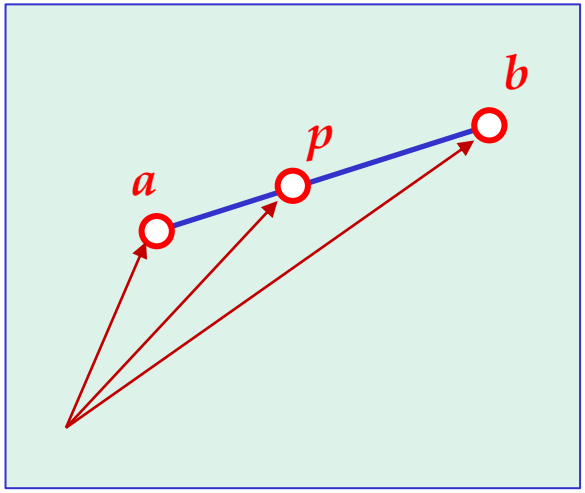
- Variables de entrada predefinidas en el Fragment Shader:
  - **gl_FragCoord**: coordenada del píxel a procesar en el shader. El origen de coordenadas es la esquina inferior izquierda de la ventana.
  - **gl_FrontFacing**: variable de tipo boolean que tiene el valor true si el fragmento proviene de una primitiva generada en su cara frontal y false en otro caso.
  - **gl_PrimitiveID**: índice de la primitiva asociada al fragmento.
  - **gl_SampleID**, **gl_SamplePosition**, **gl_SampleMaskIn**: variables utilizadas cuando la opción Multisample está activa.

- Variables de salida predefinidas en el FragmentShader:
  - **gl_FragDepth**: valor de profundidad asociado al píxel que se almacenará en el buffer de profundidad. Debe ser un float entre 0 y 1. Por defecto se toma el valor de la componente z en coordenadas homogéneas.
  - **gl_SampleMask**: variable de salida utilizada cuando la opción Multisample está activa

- Las entradas definidas por el usuario en el Fragment Shader corresponden a las salidas asociadas a los vértices en el Vertex Shader.
- Para cada ejecución del Fragment Shader los valores de las entradas correspondientes a cada pixel se calculan como una interpolación de los valores asociados a los vértices.

- En el caso de las líneas, las posiciones de los puntos $\boldsymbol{p}$ situados entre los vértices ($\boldsymbol{a}$ y $\boldsymbol{b}$) pueden calcularse como

$$\boldsymbol{p} = \alpha \cdot \boldsymbol{a} + \beta \cdot \boldsymbol{b}$$



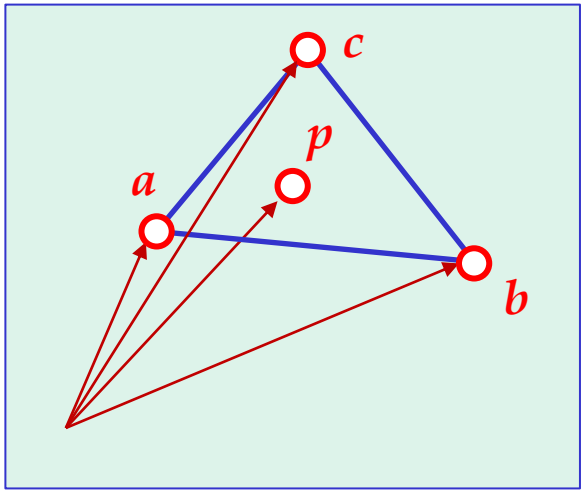
donde $\alpha,\beta \in [0,1]$ y $\alpha + \beta = 1$.

- Las propiedades de cualquier punto $\boldsymbol{p}$ del fragmento se obtienen por interpolación utilizando los valores α y β.

$$Attr(\boldsymbol{p}) = \alpha \cdot Attr(\boldsymbol{a}) + \beta \cdot Attr(\boldsymbol{b})$$

- En el caso de los triángulos, las posiciones de los puntos $\boldsymbol{p}$ situados entre los vértices ($\boldsymbol{a}$, $\boldsymbol{b}$ y $\boldsymbol{c}$) pueden calcularse como

$$\boldsymbol{p} = \alpha \cdot \boldsymbol{a} + \beta \cdot \boldsymbol{b} + \gamma \cdot \boldsymbol{c}$$



donde $\alpha,\beta,\gamma \in [0,1]$ y $\alpha + \beta + \gamma = 1$.

- Las propiedades de cualquier punto $\boldsymbol{p}$ del fragmento se calculan por interpolación utilizando los valores α, β y γ.

$$Attr(\boldsymbol{p}) = \alpha \cdot Attr(\boldsymbol{a}) + \beta \cdot Attr(\boldsymbol{b}) + \gamma \cdot Attr(\boldsymbol{c})$$

- La interpolación de los valores es el comportamiento por defecto.
- El modificador **flat** se utiliza para indicarle al proceso de rasterización que el valor de un atributo no debe interpolarse, sino que debe tomarse como el valor del último vértice de la primitiva.
- En realidad, el valor que se toma en los atributos de tipo *flat* se puede configurar con el comando **glProvokingVertex()**.
- Este comando admite dos valores:
  - GL_FIRST_VERTEX_CONVENTION
  - GL_LAST_VERTEX_CONVENTION (valor por defecto)

- OpenGL utiliza el formato RGBA (red, green, blue, alpha) para definir los colores.
- En la mayoría de los ordenadores actuales, el color se representa en 32 bits (8 bits para cada componente, es decir, de 0 a 255).
- En OpenGL, los componentes se suelen expresar en números reales entre 0.0 y 1.0. De esta forma, para expresar un color se suele utilizar un vector vec4.
- El lenguaje GLSL permite acceder a las componentes de los vectores utilizando los campos r, g, b, y a. Cuando un vector almacena información sobre un color se suele utilizar estos nombres para acceder a sus componentes.

- Fragment Shader de un color fijo
  - Si queremos que todos los píxeles generados sean de color rojo podemos utilizar un Fragment Shader muy sencillo.

```
#version 400

out vec4 FragColor;

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

- Fragment Shader de un color fijo
  - Podemos utilizar un color fijo que sea configurable. Para ello tendremos que usar una variable uniforme.

```
#version 400

uniform vec4 ObjectColor;

out vec4 FragColor;

void main()
{
    FragColor = ObjectColor;
}
```

- Color vinculado a los vértices
  - También podemos considerar el color como un atributo del vértice. En ese caso, la etapa de rasterizado e interpolación realizará una interpolación entre los colores de los vértices para calcular el color de cada pixel del fragmento.

```
#version 400

layout(location = 0) in vec3 VertexPosition;
layout(location = 1) in vec3 VertexColor;

out vec4 ObjectColor;

void main()
{
    ObjectColor= vec4( VertexColor, 1.0);
    gl_Position = MVP * vec4( VertexPosition, 1.0);
}
```

- Color plano
  - Se puede indicar que el color de todos los píxeles del fragmento no se interpole, sino que se tome el color del último vértice de la primitiva. Para eso se utiliza el modificador *flat*, que indica que una salida del VertexShader no debe ser interpolada.

```
#version 400

layout(location = 0) in vec3 VertexPosition;
layout(location = 1) in vec3 VertexColor;

flat out vec4 ObjectColor;

void main()
{
    ObjectColor= vec4( VertexColor, 1.0);
    gl_Position = MVP * vec4( VertexPosition, 1.0);
}
```

- Color plano
  - El modificador *flat* debe incluirse también en el FragmentShader.

```
#version 400

flat in vec4 ObjectColor;

out vec4 FragColor;

void main()
{
    FragColor = ObjectColor;
}
```

- Cuando se utiliza un color plano para dibujar los objetos el resultado es muy poco realista porque se pierde la sensación de volumen.
- Para conseguir un color realista en vez de aplicar un color a cada superficie hay que pensar en la forma en la que la luz va a iluminar el modelo.
- El color con el que vemos los objetos se debe a que la luz choca contra ellos y es en parte absorbida y en parte dispersada. Por ejemplo, un objeto es azul si absorbe todos los colores salvo el azul.
- El modelo de iluminación más utilizado consiste en considerar la iluminación como la suma de tres efectos diferentes: luz ambiental, luz difusa y luz especular.

- La luz ambiental es una luz que procede de todas direcciones y que al chocar con los cuerpos se dispersa en todas las direcciones. La luz ambiental provoca un color que no cambia al modificar la posición de los cuerpos o su orientación.
- La luz difusa es una luz que proviene de una determinada dirección y que al chocar con los objetos se dispersa en todas direcciones. La intensidad con la que la luz difusa se dispersa depende del ángulo con el que incide sobre la superficie de manera si la luz incide de forma vertical su brillo es mayor que si incide de forma oblicua.
- La luz especular es una luz que proviene de una determinada dirección y al chocar con un objeto se refleja con el mismo ángulo de incidencia.

- Para calcular el color de cada punto de un objeto es necesario conocer tanto el tipo de luz que incide sobre el objeto como las propiedades de absorción de la luz de dicho objeto.
- Las propiedades de un material frente a la luz ambiental se describen mediante las componentes RGBA. Por ejemplo, un objeto azul se definiría como (0.0, 0.0, 1.0, 1.0).
- Si la luz ambiental fuera un blanco puro (1.0, 1.0, 1.0, 1.0) el color del objeto sería azul puro (0.0, 0.0, 1.0, 1.0), pero si la luz ambiental fuera rojo puro (1.0, 0.0, 0.0, 1.0) el objeto absorbería toda esa luz y se vería negro (0.0, 0.0, 0.0, 1.0).
- De la misma manera se puede definir el efecto de la luz difusa y de la luz especular sobre el objeto.

- El color final de cada punto del objeto se calcula como la suma de los efectos de color con cada tipo de luz:

$$C_F = C_A + C_D + C_S$$

- El modelo de iluminación de Gouraud, propuesto en 1971, consiste en calcular el color de cada vértice en base a estos efectos. El color de cada píxel se genera en el FragmentShader por interpolación de los colores de los vértices.
- El modelo de Phong, propuesto en 1973, consiste en interpolar las propiedades de los vértices (posición y normal) para cada píxel y calcular el color directamente en el FragmentShader. Esto resulta computacionalmente más costoso, pero genera un coloreado de mayor calidad.

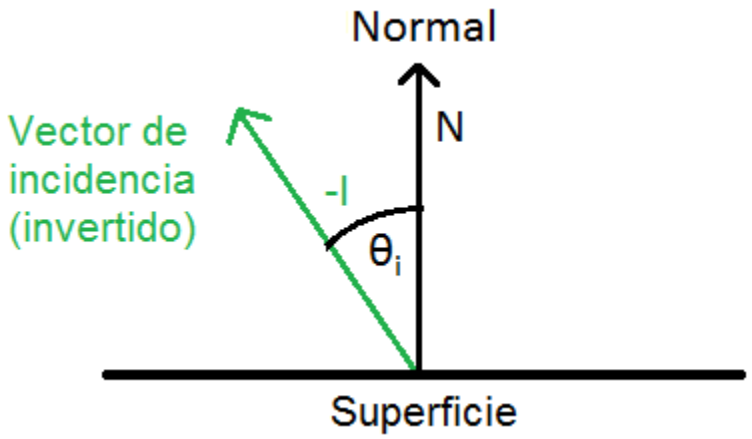
- C_A es la componente de color provocada por la luz ambiental. Se calcula como el producto de la luz ambiental, L_A, y color del material ante la luz ambiental, M_A.

$$C_A = L_A * M_A$$

- C_D es la componente de color provocada por la luz difusa. Se calcula como el producto de la luz difusa, L_D, y color del material ante la luz difusa, M_D, multiplicados por un factor de incidencia de la luz. Este factor de incidencia se calcula como el producto entre el vector de dirección de la luz, I, y el vector normal a la superficie, N.

$$C_D = L_D * M_D * FI$$

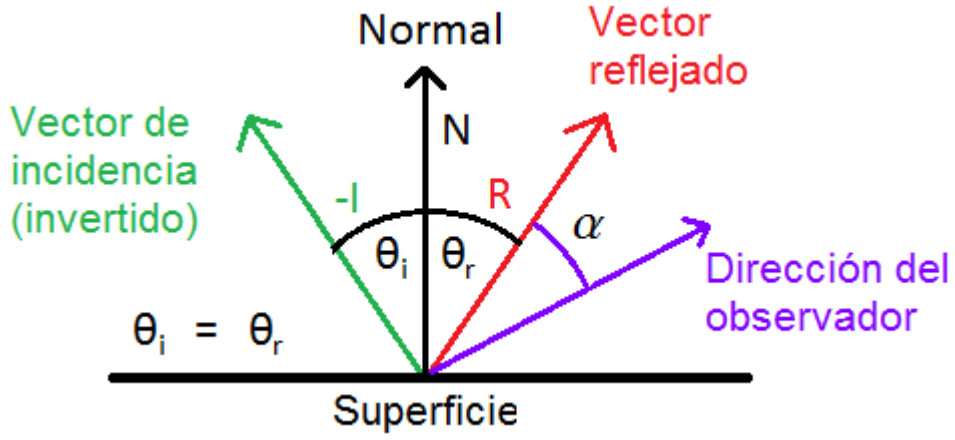
$$FI = (-I)\cdot N = -\left(I_x*N_x + I_y*N_y + I_z*N_z + I_w*N_w \right) = \cos(\theta)$$



- C_S es la componente de color provocada por la luz especular. Se calcula como el producto de la luz especular, L_S, y color del material ante la luz especular, M_S, multiplicados por un factor especular. Este factor especular se calcula como el producto entre el vector de posición del observador y el vector de reflexión de la luz, elevado a una cierta potencia.

$$C_S = L_S * M_S * FS$$

$$FS = (R \cdot V)^S = (\cos(\alpha))^S \; ; \quad R = I - 2 * (I \cdot N) * N$$



- Es importante tener en cuenta el sistema de coordenadas en el que se describe cada vector.
- Los vectores normales son atributos de los vértices y, por tanto, se describen en coordenadas del objeto.
- La dirección de la luz se describe generalmente en coordenadas de modelo.
- La dirección del observador se calcula a partir de la posición del vértice en coordenadas del observador.
- Para realizar todos los cálculos es importante transformar todos los vectores a un sistema de coordenadas común (el del observador).

- VertexShader que desarrolla el modelo de Phong

```
#version 400

layout(location = 0) in vec3 VertexPosition;
layout(location = 1) in vec3 VertexNormal;

uniform mat4 MVP;
uniform mat4 ModelViewMatrix;

out vec3 Position;
out vec3 Normal;

void main() {
    vec4 n4 = ModelViewMatrix*vec4(VertexNormal,0.0);
    vec4 v4 = ModelViewMatrix*vec4(VertexPosition,1.0);
    Normal = vec3(n4);
    Position = vec3(v4);
    gl_Position = MVP * vec4( VertexPosition, 1.0);
}
```

- FragmentShader que desarrolla el modelo de Phong

```
#version 400

in vec3 Position;
in vec3 Normal;
uniform mat4 ViewMatrix;
struct LightInfo {
    vec3 Ldir;
    vec3 La;
    vec3 Ld;
    vec3 Ls;
};
uniform LightInfo Light;
struct MaterialInfo{
    vec3 Ka;
    vec3 Kd;
    vec3 Ks;
    float Shininess;
};
uniform MaterialInfo Material;
```

- FragmentShader que desarrolla el modelo de Phong (sigue)

```
…
out vec4 FragColor;

vec3 ads() {
    vec4 s4 = ViewMatrix*vec4(Light.Ldir,0.0);
    vec3 n = normalize(Normal);
    vec3 v = normalize(-Position);
    vec3 s = normalize(-vec3(s4));
    vec3 r = reflect(-s,n);
    float dRate = max(dot(s,n), 0.0);
    float sRate = pow(max(dot(r,v),0.0),
                      Material.Shininess);

    vec3 ambient = Light.La * Material.Ka;
    vec3 difusse = Light.Ld * Material.Kd * dRate;
    vec3 specular = Light.Ls * Material.Ks * sRate;

    return ambient + difusse + specular;
}
```

- FragmentShader que desarrolla el modelo de Phong (fin)

```
…

void main()
{
    vec3 Color = ads();
    FragColor = vec4( Color, 1.0 );
}
```

- El modelo anterior puede modificarse de varias formas para mejorar su eficiencia o mejorar el resultado visual.
- Una opción se considerar que el vector v es siempre (0,0,1). Esto equivale a decir que el observador está muy alejado de los objetos y, por tanto, todos los vectores v son aproximadamente iguales (*non-local viewer*). Esta aproximación genera resultados similares y ahorra el cálculo de v. De hecho, las versiones no programables de OpenGL utilizan esta aproximación por defecto.
- También se puede asumir que el vector de dirección de la luz (*Ldir*) es un vector unitario y no es necesario normalizarlo.

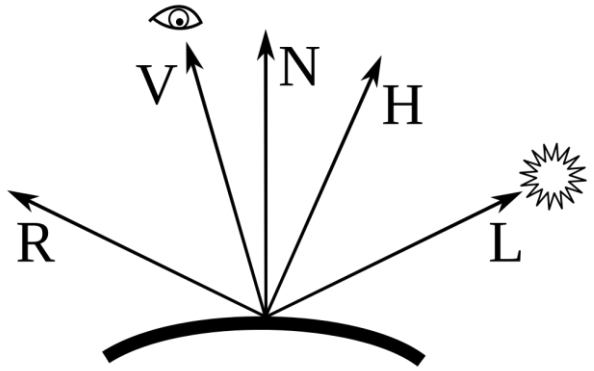
- Modelo *non-local viewer*:

```
vec3 ads() {
    vec4 s4 = ViewMatrix*vec4(Light.Ldir,0.0);
    vec3 n = normalize(Normal);
    vec3 s = -vec3(s4);
    vec3 r = reflect(-s,n);
    float dRate = max(dot(s,n), 0.0);
    float sRate = pow(max(r.z,0.0),Material.Shininess);

    vec3 ambient = Light.La * Material.Ka;
    vec3 difusse = Light.Ld * Material.Kd * dRate;
    vec3 specular = Light.Ls * Material.Ks * sRate;

    return ambient + difusse + specular;
}
```

- Otra aproximación, conocida como *halfway vector* o modelo de Blinn, consiste en sustituir la expresión (***r***·***v***) en el término especular por la expresión (***h***·***n***), donde el vector *h* se calcula como ***v***+***s***. De esta forma se evita el cálculo de *r* y su normalización.



- También se puede considerar que el foco de luz no está en el infinito sino en una posición determinada, *Lpos*.
- En ese caso, el vector de incidencia de la luz sobre el punto (***-s***) se puede calcular como la diferencia entre la posición de la luz y la posición del punto, *Vpos*.

$$-\boldsymbol{s} = \text{normalize}(\boldsymbol{Lpos} - \boldsymbol{Vpos})$$

- Se pueden añadir efectos a la luz, como una dirección principal del foco y un ángulo de corte a partir del cual se anula la iluminación.

- Se puede simular facilmente un efecto de niebla si se mezcla el color de cada pixel con un color de niebla (típicamente un cierto tono gris). La niebla provoca que los puntos más alejados se vean de forma más difusa.
- La proporción de la mezcla se calcula en función de la distancia del punto a la cámara.
- Una opción sencilla es calcular la proporción de niebla como una función lineal de la distancia:

$$f = (d_{max} - |z|) / (d_{max} - d_{min})$$

- Donde d_{min} es la distancia mínima a partir de la cual comienza a aplicarse el efecto de la niebla y d_{max} es la distancia máxima a la que se satura el efecto y el color del pixel queda como el color de la niebla.

- FragmentShader que desarrolla el efecto niebla

```
#version 400
…

struct FogInfo {
  float maxDist;
  float minDist;
  vec3 color;
};
uniform FogInfo Fog;

void main() {
  float dist = abs( Position.z );
  float fogFactor = (Fog.maxDist - dist)/
                    (Fog.maxDist - Fog.minDist);
  fogFactor = clamp( fogFactor, 0.0, 1.0);
  vec3 shadeColor = ads();
  vec3 color = mix(Fog.color, shadeColor, fogFactor);
  FragColor = vec4(color,1.0);
}
```

- Otras formas utilizadas para calcular el factor de proporción de la niebla se basan en una relación exponencial en vez de lineal:

$$f = exp\ (-d \cdot |z|\ )$$

o

$$f = exp\ (-d \cdot z^2\ )$$

- Donde d es un factor conocido como densidad de niebla.

- Hasta ahora, todos los colores que hemos considerado han sido opacos (es decir, la componente alpha era 1.0). Para tener en cuenta efectos de transparencia es necesario utilizar valores diferentes para esta componente alpha.
- Por defecto, OpenGL no tiene en cuenta las transparencias sino que utiliza el buffer de profundidad para detectar cual es la superficie más cercana y utiliza el color de esa superficie para calcular el color final a mostrar en la imagen.
- La transparencia provoca una mezcla (*blending*) de colores entre el color del material traslúcido y el color de la superficie que se encuentre detrás. Para activar los efectos de transparencia hay que ejecutar

glEnable(GL_BLEND);

- La imagen generada en el proceso de rendering se almacena en el COLOR_BUFFER.
- Si la opción de profundidad (GL_DEPTH_TEST) no está activada, los polígonos se dibujan en el orden en el que se generan.
- Si la opción de profundidad está activada, al recibir la información de un polígono se calcula la profundidad de cada punto y se compara con la profundidad de lo pintado anteriormente. Si el nuevo polígono se encuentra más cerca se sobreescribe el COLOR_BUFFER. Si está más lejos no se modifica el COLOR_BUFFER.
- Si la opción de mezcla está activada, el color final se calcula combinando el color que se encuentra en el buffer (*destination color*) con el del nuevo polígono (*source color*).

- Por defecto, la función de mezcla para cada componente RGBA es la siguiente:

$$C_f = (C_s * S) + (C_d * D)$$

- Para asignar los valores de S y D se utiliza la función

glBlendFunc(GLenum S, GLenum D);

- Los valores de S y D pueden ser los indicados en la siguiente tabla, en la que las componentes (R_s, G_s, B_s, A_s) se refieren al *source color* y (R_d, G_d, B_d, A_d) se refieren al *destination color*:

Función	Componentes RGB	Componente A
GL_ZERO	(0,0,0)	0
GL_ONE	(1,1,1)	1
GL_SRC_COLOR	(Rs, Gs, Bs)	As
GL_ONE_MINUS_SRC_COLOR	(1-Rs, 1-Gs, 1-Bs)	1-As
GL_DST_COLOR	(Rd, Gd, Bd)	Ad
GL_ONE_MINUS_DST_COLOR	(1-Rd, 1-Gd, 1-Bd)	1-Ad
GL_SRC_ALPHA	(As, As, As)	As
GL_ONE_MINUS_SRC_ALPHA	(1-As, 1-As, 1-As)	1-As
GL_DST_ALPHA	(Ad, Ad, Ad)	Ad
GL_ONE_MINUS_DST_ALPHA	(1-Ad, 1-Ad, 1-Ad)	1-Ad
GL_CONSTANT_COLOR	(Rc, Gc, Bc)	Ac
GL_ONE_MINUS_CONSTANT_COLOR	(1-Rc, 1-Gc, 1-Bc)	1-Ac
GL_CONSTANT_ALPHA	(Ac, Ac, Ac)	Ac
GL_ONE_MINUS_CONSTANT_ALPHA	(1-Ac, 1-Ac, 1-Ac)	1-Ac
GL_SRC_ALPHA_SATURATE	Min(As, 1-As)	1

- Los valores de las opciones que usan constantes se fijan con la función **glBlendColor(r, g, b, a).**

- Por ejemplo:

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

- Para un destination color (1.0f, 0.0f, 0.0f, 1.0f) y un source color (0.0f,0.0f, 1.0f, 0.6f) la mezcla queda

$$RGBA_f = (0.0f,0.0f,1.0f,0.6f)*0.6 + (1.0f,0.0f,0.0f,1.0f)*0.4 =$$

$$(0.4f, 0.0f, 0.6f, 0.76f)$$

- También es posible modificar la función de mezcla:

glBlendEquation(GLenum mode);

- Donde el modo puede ser

Modo	Función
GL_FUNC_ADD	$C_f = (C_s * S) + (C_d * D)$
GL_FUNC_SUBTRACT	$C_f = (C_s * S) - (C_d * D)$
GL_FUNC_REVERSE_SUBTRACT	$C_f = (C_d * D) - (C_s * S)$
GL_MIN	$C_f = \min(C_s , C_d)$
GL_MAX	$C_f = \max(C_s , C_d)$