# Parcial 1

# Tipos de esqueleto

## Class A-Esqueleto (PER-VERTEX) (Deformación)

A-Vertex Shader

```
#version 330 core

// --- INPUTS (from your 3D model) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (to the Fragment Shader) ---
out vec4 frontColor; // El color FINAL calculado por vértice
out vec2 vtexCoord;  // La coordenada de textura

// --- UNIFORMS (from the viewer) ---
uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

void main()
{
    vec4 vertex_objectspace = vec4(vertex, 1.0);
    vec3 normal_objectspace = normal;

    // Calcula la normal en Eye Space
    vec3 N = normalize(normalMatrix * normal_objectspace);

    // Calcula el color (iluminación simple por Z)
    frontColor = vec4(color,1.0) * N.z;

    // Pasa la coordenada de textura
    vtexCoord = texCoord;

    // Calcula la posición final
    gl_Position = modelViewProjectionMatrix * vertex_objectspace;
}
```

PROF

**NOTA** Si te pide modificar la Projection y la Normal Space, se tiene que calcular por si solo:

```
#version 330 core
```

```
// --- INPUTS (from your 3D model) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (to the Fragment Shader) ---
out vec4 frontColor; // El color FINAL calculado por vértice
out vec2 vtexCoord;  // La coordenada de textura

// --- UNIFORMS (from the viewer) ---
uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

void main()
{
    //P_object = texture ...
    //P_eye_4 = modelViewMatrix * vec4(P_object, 1.0);

    // Calcula la posición final
    gl_Position = projectionMatrix * P_eye_4;
}
```

A-Fragment Shader

```
#version 330 core

// --- INPUT (from the Vertex Shader) ---
in vec4 frontColor; // Recibe el color interpolado

// --- OUTPUT ---
out vec4 fragColor;

void main()
{
    // Simplemente asigna el color calculado en el VS
    fragColor = frontColor;
}
```

PROF

## Class B-Esqueleto (PER-FRAGMENT) (Texturas+Iluminación)

B-Vertex Shader

```
#version 330 core

// --- INPUTS (from your 3D model) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
```

```
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (to the Fragment Shader) ---
out vec2 vtexCoord; // Pasa la coordenada de textura

// --- UNIFORMS (from the viewer) ---
uniform mat4 modelViewProjectionMatrix;

void main()
{
    // Pasa la coordenada de textura
    vtexCoord = texCoord;

    // Calcula la posición final
    gl_Position = modelViewProjectionMatrix * vec4(vertex, 1.0);
}
```

B-Fragment Shader

```
#version 330 core

// --- INPUT (from the Vertex Shader) ---
in vec2 vtexCoord;

// --- OUTPUT ---
out vec4 fragColor;

// --- UNIFORMS (Añadir los que necesites) ---
// (p.ej., uniform sampler2D colorMap;)

void main()
{
    // ================================================================
    // DEBES declarar una variable 'finalColor'.
    //
    // (p.ej., vec4 texColor = texture(colorMap, vtexCoord);)
    // (p.ej., vec4 finalColor = texColor;)
    // ================================================================

    // (Esta línea usará el 'finalColor' que has definido arriba)
    fragColor = finalColor;
}
```

PROF

# Ejercicio Tipo 1-Deformación de Geometría

- **Objetivo:** Cambiar la **forma**, **posición** o **animación** del modelo 3D.

- **Palabras Clave:** "rotar", "deformar", "proyectar", "estirar", "girar cabeza", "animar".
- **Ejemplos:** `Look`, `Dolphin`, `Dalify`, `Cubify`.
- **Dónde trabajas:** Casi todo el código va en el **Vertex Shader (`.vert`)**.
- **Esqueleto a Usar: Esqueleto 1 (Per-Vertex)**.
- **Por qué:** Este esqueleto está diseñado para hacer el trabajo principal en el Vertex Shader (en el bloque `STEP 3 (VS)`). La iluminación simple que calcula por defecto (`frontColor = ... * N.z`) suele ser la que piden estos ejercicios.

# Ejemplo 1-Dalify

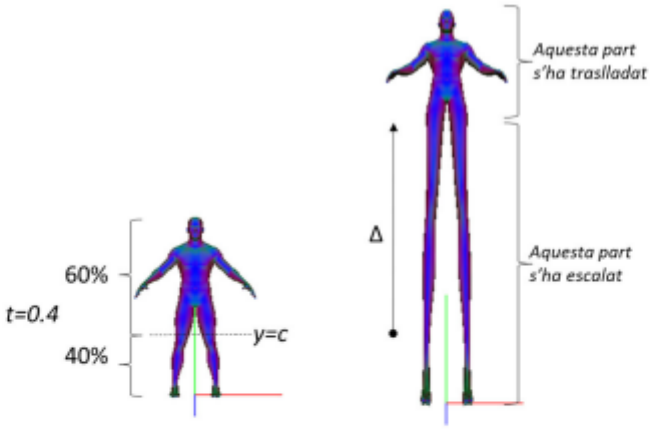
## 1.1-Enunciado

# Dalify (dalify.*) 2.5 punts

Escriu **VS+FS** per deformar el model en direcció vertical (eix Y en *model space*), per obtenir una aparença similar a la d'alguns animals en quadres de Salvador Dalí:

Les temptacions de Sant Antoni (Salvador Dalí, 1946)

El VS deformarà el model modificant únicament la coordenada Y en *model space*:

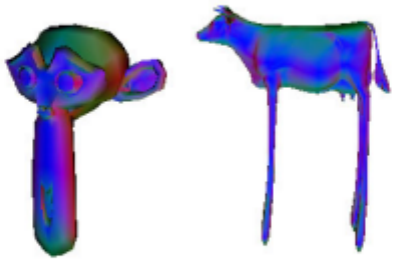



Sigui c el resultat d'interpolar linealment boundingBoxMin.y i boundingBoxMax.y, segons un paràmetre d'interpolació t, **uniform float t = 0.4**.

Si la coordenada Y és inferior a c, el VS li aplicarà l'escalat donat per **uniform float scale = 4.0** per tal d'allargar les potes del model. Altrament, no li aplicarà cap escalat, però sí una translació Δ en Y. Per calcular Δ, observeu que per tenir continuïtat a y=c, llavors **c*scale = c + Δ** (aïlleu Δ).

Degut a que no estem recalculant els plans de *clipping*, és possible que el model surti retallat.

PROF El FS farà les tasques habituals.



**Identificadors obligatoris:**

dalify.vert, dalify.frag *(minúscules!)*
uniform float t =
uniform float scale = 4.0;

1.2-Vertex Shader

```glsl
#version 330 core

// --- INPUTS (from your 3D model) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (to the Fragment Shader) ---
out vec4 frontColor; // El color final calculado para este vértice
out vec2 vtexCoord;  // La coordenada de textura

// --- UNIFORMS (from the viewer) ---
uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

// --- UNIFORMS (Añadidos para "Dalify") ---
uniform float t = 0.4;
uniform float scale = 4.0;
uniform vec3 boundingBoxMin; // Proporcionado por el viewer
uniform vec3 boundingBoxMax; // Proporcionado por el viewer

void main()
{
    // Estas son tus variables iniciales
    vec4 vertex_objectspace = vec4(vertex, 1.0);
    vec3 normal_objectspace = normal; // No modificamos la normal

    // ================================================================
    // == STEP 3 (VS) - CÓDIGO DE DEFORMACIÓN "Dalify"
    // ================================================================

    // 1. Calcular 'c' (el punto de corte en Y)
    // Interpolar linealmente entre el min y max de Y usando 't'
    float c = mix(boundingBoxMin.y, boundingBoxMax.y, t);

    // 2. Calcular 'Δ' (Delta, la traslación)
    // El enunciado da la fórmula: c*scale = c + Δ
    // Aislamos Δ: Δ = c*scale - c
    float delta = c * (scale - 1.0);

    // 3. Aplicar la deformación (escalar o trasladar)
    if (vertex_objectspace.y < c)
    {
        // Parte de abajo: ESCALAR
        vertex_objectspace.y = vertex_objectspace.y * scale;
    }
    else
    {
        // Parte de arriba: TRASLADAR
        vertex_objectspace.y = vertex_objectspace.y + delta;
```

PROF

```
    }

    // ============================================================

    // Calcular la normal en Eye Space (sin modificar)
    vec3 N = normalize(normalMatrix * normal_objectspace);

    // Calcular el color ("tarea habitual" del esqueleto)
    frontColor = vec4(color, 1.0) * N.z;

    // Pasar la coordenada de textura
    vtexCoord = texCoord;

    // Calcular la posición final usando el VÉRTICE DEFORMADO
    gl_Position = modelViewProjectionMatrix * vertex_objectspace;
}
```

### 1.3-Fragment Shader

```
#version 330 core

in vec4 frontColor;
out vec4 fragColor;

void main()
{
    fragColor = frontColor;
}
```

---

## Ejemplo 2-Dolphin

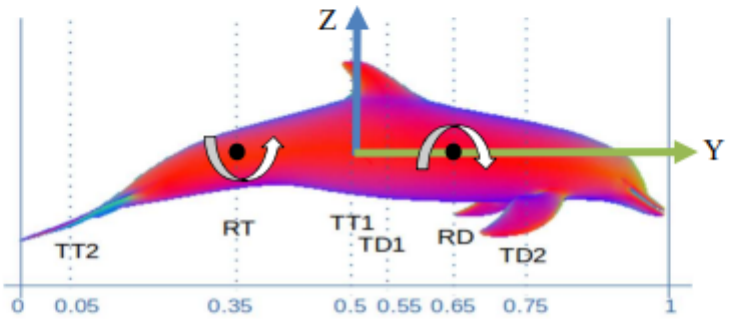
PROF

### 2.1-Enunciado

# Dolphin (dolphin.*)

Volem simular l'animació d'un dofí nadant. Per aconseguir-ho cal que el VS deformi el model tenint en compte les següents indicacions. L'animació durarà **un segon** i s'anirà repetint en el temps (caldrà fer servir una funció sinusoïdal amb un període apropiat). Per tal d'aplicar les transformacions corresponents dividirem el dofí en dues meitats i aplicarem una rotació a cada meitat en funció del temps.

**Model**: Per al model **dolphin.obj** amb els següents punts (tingueu en compte que els valors de cada punt de la figura són en relació a la llargària de la caixa contenidora en direcció Y):




**Punts de rotació**: La rotació de cada meitat serà al voltant d'un eix paral·lel a l'eix X i que passa per un punt de la forma (0,y,0), on la y varia segons la meitat. Per la meitat davantera, el punt de rotació serà RD, i per la meitat posterior el punt de rotació serà RT.

**Transició de la deformació:** La transformació s'aplicarà de manera suau (combinant amb smoothstep els vèrtexs originals i els transformats) en l'eix Y des del punt TD1 (on no hi haurà cap transformació) fins al punt TD2 (on la transformació serà màxima) per la part davantera, i des de TT1 fins a TT2 per la part posterior.

**Angles de la rotació:** L'angle de rotació per la part davantera variarà en [-PI/32, PI/32], i per la part posterior en [-PI/4, 0]. Tingueu en compte que les rotacions seran en sentits oposats, fent que el cap i la cua pugin i baixin a l'hora amb un petit offset de temps.

**Offset entre parts:** L'animació de la part davantera començarà 0.25 segons abans que la de la part posterior. És a dir, si usem time = 0 en l'animació de la part posterior, la part davantera es comportarà com si time = 0.25.

PROF

El color del dofí serà el gris clar (0.8, 0.8, 0.8), al qual el VS aplicarà il·luminació bàsica tenint en compte la component Z de la normal en *eye space*.

El FS farà els càlculs imprescindibles per a la visualització. Aquí tens els resultats esperats amb el model dolphin.obj per a diferents instants de temps (t=0s, 1.25s, 2.75s):

t = 0s  t = 1.25s  t = 2.75s

**Identificadors (ús obligatori):**
dolphin.vert, dolphin.frag
uniform float time;
const float PI = 3.1416;

2.2-Vertex Shader

```glsl
#version 330 core

// --- INPUTS (del modelo 3D) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (al Fragment Shader) ---
out vec4 frontColor; // El color final calculado para este vértice
out vec2 vtexCoord;  // La coordenada de textura

// --- UNIFORMS (del viewer) ---
uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

// --- UNIFORMS (Añadidos para "Dolphin") ---
uniform float time;
const float PI = 3.1416;

void main()
{
    // Variables iniciales
    vec4 vertex_objectspace = vec4(vertex, 1.0);
    vec3 normal_objectspace = normal;

    // ================================================================
    // == STEP 3 (VS) - LÓGICA DE DEFORMACIÓN "Dolphin"
    // ================================================================

    // --- 1. Calcular ángulos (con offset de tiempo) ---
    // La animación dura 1 segundo
    float time_base = 2.0 * PI * time;

    // Parte delantera: time + 0.25, rango [-PI/32, PI/32]
    // Usamos -cos para que en t=0 empiece "abajo" (como pide la imagen)
    float angle_davantera = -(PI/32.0) * cos(time_base);

    // Parte posterior: time + 0.0, rango [-PI/4, 0]
    // Usamos sin() para mapear al rango [-PI/4, 0]
    float angle_posterior = -PI/8.0 + (PI/8.0) * sin(time_base);


    // --- 2. Calcular factores de mezcla (smoothstep) ---
    // t_davantera: 0.0 si y < 0.55, 1.0 si y > 0.75
    float t_davantera = smoothstep(0.55, 0.75, vertex.y);

    // t_posterior: 0.0 si y > 0.5, 1.0 si y < 0.05
    float t_posterior = smoothstep(0.5, 0.05, vertex.y);
```

PROF

```
    // --- 3. Calcular deformación (Rotación en X con pivote) ---

    // a) Deformación delantera (Pivote RD en y=0.65)
    float c_d = 0.65;
    float cos_d = cos(angle_davantera);
    float sin_d = sin(angle_davantera);

    vec4 P_davantera = vec4(
        vertex.x,
        (vertex.y - c_d) * cos_d - vertex.z * sin_d + c_d,
        (vertex.y - c_d) * sin_d + vertex.z * cos_d,
        1.0
    );
    vec3 N_davantera = vec3(
        normal.x,
        normal.y * cos_d - normal.z * sin_d,
        normal.y * sin_d + normal.z * cos_d
    );

    // b) Deformación posterior (Pivote RT en y=0.35)
    float c_p = 0.35;
    float cos_p = cos(angle_posterior);
    float sin_p = sin(angle_posterior);

    vec4 P_posterior = vec4(
        vertex.x,
        (vertex.y - c_p) * cos_p - vertex.z * sin_p + c_p,
        (vertex.y - c_p) * sin_p + vertex.z * cos_p,
        1.0
    );
    vec3 N_posterior = vec3(
        normal.x,
        normal.y * cos_p - normal.z * sin_p,
        normal.y * sin_p + normal.z * cos_p
    );

    // --- 4. Mezclar (blend) las deformaciones ---
    // Mezclamos la parte delantera
    vertex_objectspace = mix(vertex_objectspace, P_davantera,
t_davantera);
    normal_objectspace = mix(normal_objectspace, N_davantera,
t_davantera);

    // Mezclamos la parte posterior (sobre el resultado anterior)
    vertex_objectspace = mix(vertex_objectspace, P_posterior,
t_posterior);
    normal_objectspace = mix(normal_objectspace, N_posterior,
t_posterior);

    // ============================================================

    // Calcular la normal en Eye Space (con la normal deformada)
    vec3 N = normalize(normalMatrix * normal_objectspace);
```

PROF

```
    // Calcular el color (Gris 0.8 * N.z)
    frontColor = vec4(0.8, 0.8, 0.8, 1.0) * N.z;

    // Pasar la coordenada de textura
    vtexCoord = texCoord;

    // Calcular la posición final (con el vértice deformado)
    gl_Position = modelViewProjectionMatrix * vertex_objectspace;
}
```

### 2.3-Fragment Shader

```
#version 330 core

in vec4 frontColor;
out vec4 fragColor;

void main()
{
    fragColor = frontColor;
}
```

---

## Ejemplo 3-Spring

### 3.1-Enunciado

# Spring (spring.*) 2.5 punts

Escriu **VS+FS** que simulin l'expansió i compressió cícliques del model 3D com si fos una molla (vegeu el vídeo **spring.mp4** al zip de l'enunciat.

El VS s'encarregarà de l'animació, que tindrà dues fases **que es repetiran cada 3.5 segons.**

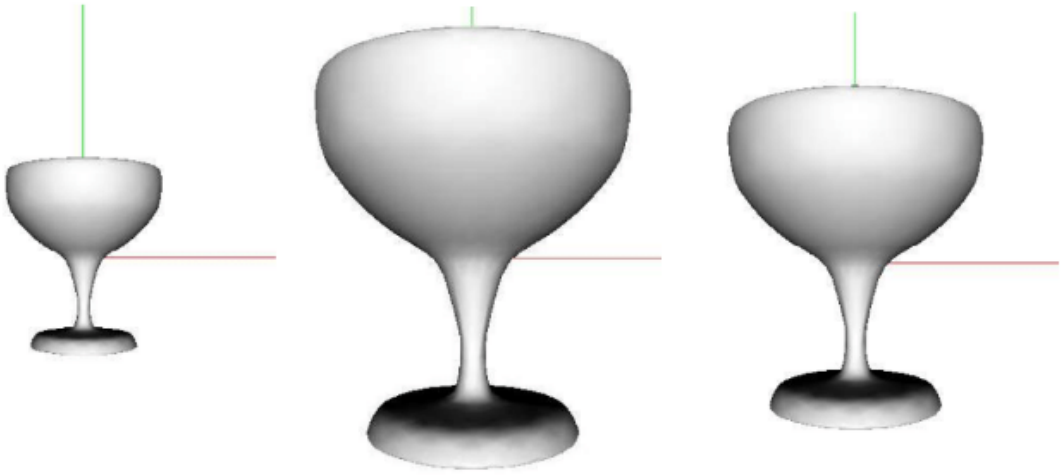
La primera fase (expansió) tindrà una durada de 0.5 segons i mourà els vèrtexs des de l'origen de coordenades fins a la seva posició original en *model space*. Per a calcular la interpolació linial entre aquestes dues posicions, feu que el paràmetre d'interpolació linial sigui $(t/0.5)^3$, on t és el temps en segons *des de l'inici del període* (per exemple, quan time = 4, t = 0.5).

La segona fase (compressió) tindrà una durada de **3 segons**, i mourà els vèrtexs des de la seva posició inicial en *model space* cap a l'origen. Ara però volem que els vèrtexs es moguin a velocitat uniforme. Penseu com heu de calcular el paràmetre d'interpolació lineal, a partir d'un valor t que dins de cada període estarà dins l'interval [0.5, 3.5).

Un cop calculada la posició del vèrtex en *model space*, caldrà transformar-lo a *clip space* com feu usualment. El color del vèrtex serà el gris que té per components la Z de la normal en *eye space*.

El FS farà les tasques per defecte.

Aquí teniu el resultat als instants de temps 0.4, 0.5, 1:



PROF

**Fitxers i identificadors (ús obligatori):**
`spring.vert, spring.frag`

## 3.2-Vertex Shader

```
#version 330 core

// --- INPUTS (del modelo 3D) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;
```

```glsl
// --- OUTPUTS (al Fragment Shader) ---
out vec4 frontColor; // El color final calculado
out vec2 vtexCoord;  // La coordenada de textura

// --- UNIFORMS (del viewer) ---
uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

// --- UNIFORMS (Añadido para "Spring") ---
uniform float time; // Necesario para la animación

void main()
{
    // Variables de deformación
    float interp_factor = 0.0; // El factor de mezcla 't'
    vec3 vertex_objectspace = vertex;
    vec3 normal_objectspace = normal;
    vec3 origen = vec3(0.0, 0.0, 0.0);

    // ================================================================
    // == STEP 3 (VS) - LÓGICA DE ANIMACIÓN "Spring" con MIX
    // ================================================================

    // 1. Encontrar el tiempo actual dentro del ciclo de 3.5s
    float t_period = mod(time, 3.5);

    // 2. Comprobar en qué fase estamos
    if (t_period < 0.5)
    {
        // FASE 1: EXPANSIÓN (0.0s a 0.5s)
        // El enunciado pide (t/0.5)^2
        float t_exp = t_period / 0.5;
        interp_factor = pow(t_exp, 2.0);

        // Aplicamos la lógica 'mix' (Origen -> Vértice)
        vertex_objectspace = mix(origen, vertex, interp_factor);
    }
    else
    {
        // FASE 2: COMPRESIÓN (0.5s a 3.5s)

        // Mapeamos [0.5, 3.5] a [0.0, 1.0] (Lineal)
        float t_comp = (t_period - 0.5) / 3.0;

        // Aplicamos la lógica 'mix' (Vértice -> Origen)
        // (Esto es lo mismo que 1.0 - t_comp)
        vertex_objectspace = mix(vertex, origen, t_comp);

        // Guardamos el factor de escala para la normal
        interp_factor = 1.0 - t_comp;
    }
```

PROF

```
    // 3. Deformar la normal (¡LA PARTE QUE FALTABA!)
    // (Igual que en 'Dalify', debemos deformar la normal)
    // Añadimos 0.0001 para evitar dividir por cero
    normal_objectspace = normal / (interp_factor + 0.0001);

    // ============================================================

    // Calcular la normal en Eye Space (con la normal deformada)
    vec3 N = normalize(normalMatrix * normal_objectspace);

    // Calcular el color:
    frontColor = vec4(N.z, N.z, N.z, 1.0);

    // Pasar la coordenada de textura
    vtexCoord = texCoord;

    // Calcular la posición final (con el vértice deformado)
    gl_Position = modelViewProjectionMatrix * vec4(vertex_objectspace,
1.0);
}
```

3.3-Fragment Shader

```
#version 330 core

in vec4 frontColor;
out vec4 fragColor;

void main()
{
    fragColor = frontColor;
}
```

PROF

# Ejercicio Tipo 2-Texturas

- **Objetivo:** Decidir el color de un **píxel** basándose en texturas, coordenadas, o lógica (`if`, `distance`, `fract`, etc.).
- **Palabras Clave:** `texture()`, `sampler2D`, `colorMap`, `vtexCoord`, `if`, `discard`, `procedural`.
- **Ejemplos:** `Digits`, `Smile`, `Flag`, `Beach`, `Hunter`.
- **Dónde trabajas:** Todo el código va en el **Fragment Shader (`.frag`)**.
- **Esqueleto a Usar: Esqueleto 2 (Per-Fragment)**.
- **Por qué:** Necesitas control total por píxel. Tendrás que usar tu "cheatsheet" para añadir `uniforms` (como `sampler2D`) y "snippets" (como "Pasar Normal" si el efecto depende de `v_normal_eye`, como en "Smile") al esqueleto 2.

# Ejemplo 1-Hunter

## 1.1-Enunciado

# hunter (hunter.*)

useu ~/assig/grau-g/Viewer/GlarenaSL per resoldre aquest exercici. **Podria fallar,** si feu servir la vostra versió

Volem simular uns binocles que ens acosten els detalls d'una textura que ocupa tot el viewport, com en aquestes imatges:

Aquest exercici sols funciona amb l'objecte `plane.obj`, texturat amb l'escena triada. Per resoldre'l, has d'implementar:

1. un VS que sols emet les coordenades de textura de cada vèrtex, i les coordenades de cada vèrtex del model, com si ja estiguessin en coordenades de clipping (val a dir que has de fer servir la identitat com a `modelViewProjectionMatrix`.

2. Us proporcionem un arxiu `blur.glsl` que conté una funció que heu de fer servir al vostre FS. Aquesta funció mostreja una textura (s'hi accedeix amb el `sampler2D jungla`), en les coordenades que rep com paràmetre, però la desenfoca. Colorejeu els fragments amb el resultat retornat per aquesta funció, obtenint un viewport omplert per la textura triada, però desenfocada.

3. Afegirem ara els binocles. Per posicionar-los, farem servir el `uniform vec2 mousePositon`, que ens dona les coordenades del ratolí en aquell moment, en píxels, amb l'origen de coordenades a la cantonada inferior esquerra del viewport. També disposem del `uniform vec2 viewport` que ens retorna l'amplada i alçada del viewport en píxels. Per una posició donada del ratolí, la part transparent dels binocles consisteix de dos cercles superposats de radi 100 píxels, centrats en dos punts 80 píxels a esquerra i dreta del ratolí. La vorera negra dels binocles son la part que cau fora d'aquesta porció transparent, de dos circumferències de gruix 5 píxels. Afegeix codi al teu shader que dibuixi la vorera, i que estigui preparat per colorejar els píxels de la porció transparent de forma diferent a la resta. (pista: fixeu-vos que podeu fer servir les coordenades de textura com a coordenades, i podeu convertir a píxels usant `viewport`).

4. Ja sols queda simular l'òptica dels binocles. El uniform float magnific indicarà el factor d'augment que volem que tinguin. Així, donarem a cada fragment F dins dels binocles el color de la textura en un punt P que es troba entre F i el ratolí, i tal que la dist(F,ratolí) és igual a magnific*dist(P, ratolí).

PROF

**Identificadors obligatoris:**
`hunter.vert, hunter.frag`

```
uniform vec2 mousePosition;
uniform vec2 viewport;
uniform sampler2D jungla;
uniform float magnific = 3;
```

1.2-Vertex Shader

```
#version 330 core

// --- INPUTS (del modelo 3D) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;   // Lo mantenemos
layout (location = 2) in vec3 color;    // Lo mantenemos
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (al Fragment Shader) ---
out vec2 vtexCoord;

// --- UNIFORMS (del viewer) ---
// No usamos 'modelViewProjectionMatrix'
// El problema pide usar la identidad.

void main()
{
    // 1. Pasar la coordenada de textura
    vtexCoord = texCoord;

    // 2. Usar la posición del vértice como posición de clipping
    // (como pide el enunciado)
    gl_Position = vec4(vertex, 1.0);
}
```

1.3-Fragment Shader

PROF

```
#version 330 core

// --- INPUT (del Vertex Shader) ---
in vec2 vtexCoord; // Rango [0, 1]

// --- OUTPUT ---
out vec4 fragColor;

// --- UNIFORMS (Añadidos para "hunter") ---
uniform vec2 mousePosition;
uniform vec2 viewport;    // <-- 'blur.glsl' también necesita esto
uniform sampler2D jungla;   // <-- 'blur.glsl' también necesita esto
uniform float magnific = 3.0;

// ================================================================
// == CÓDIGO PEGADO DE blur.glsl [INICIO] ==
// ================================================================
// adaptat de https://www.shadertoy.com/view/Xltfzj.
// no és realment Gaussià
// **requereix** que hi hagi declarat un sampler2D jungla!
```

```
// retorna el color corresponent a les coordenades de textura coords.
vec4 blurImage( in vec2 coords )
{
    float Pi = 6.28318530718; // Pi*2
    float Directions = 16.0; // BLUR DIRECTIONS (Default 16.0 - More is
better but slower)
    float Quality = 8.0; // BLUR QUALITY (Default 4.0 - More is better
but slower)
    float Size = 10.0; // BLUR SIZE (Radius)

    vec2 Radius = Size/viewport;

    vec4 Color = texture(jungla, coords);
    for( float d=0.0; d<Pi; d+=Pi/Directions)
    {
        float cd = cos(d);
        float sd = sin(d);
        for(float i=1.0/Quality; i<=1.0; i+=1.0/Quality)
        {
            Color += texture(jungla, coords+vec2(cd,sd)*Radius*i);
        }
    }

    // Output to screen
    Color /= Quality * Directions - 15.0;
    return  Color;
}
// ============================================================
// == CÓDIGO PEGADO DE blur.glsl [FIN] ==
// ============================================================


void main()
{
    // ============================================================
    // == LÓGICA DE "hunter" VA AQUÍ ==
    // ============================================================

    // 1. Color por defecto: la jungla desenfocada
    // Ahora el linker PUEDE encontrar esta función
    vec4 finalColor = blurImage(vtexCoord);

    // 2. Convertir coordenadas a píxeles
    vec2 pixelCoord = vtexCoord * viewport;

    // 3. Definir los centros de los binoculares
    vec2 centerL = mousePosition + vec2(-80.0, 0.0);
    vec2 centerR = mousePosition + vec2( 80.0, 0.0);

    // 4. Calcular distancias (en píxeles)
    float distL = distance(pixelCoord, centerL);
    float distR = distance(pixelCoord, centerR);
```

PROF

```
    // 5. Comprobar si estamos en la VORERA NEGRA (radio 100 a 105)
    if ( (distL > 100.0 && distL < 105.0) ||
         (distR > 100.0 && distR < 105.0) )
    {
        finalColor = vec4(0.0, 0.0, 0.0, 1.0); // Negro
    }
    // 6. Comprobar si estamos en la LENTE (radio < 100)
    else if (distL < 100.0 || distR < 100.0)
    {
        // --- Lógica de Magnificación ---
        vec2 M_tex = mousePosition / viewport;
        vec2 P_tex = M_tex + (vtexCoord - M_tex) / magnific;

        // Muestreamos la textura *original* (nítida) en el punto P
        finalColor = texture(jungla, P_tex);
    }

    // ================================================================

    // --- Asignación Final ---
    fragColor = finalColor;
}
```

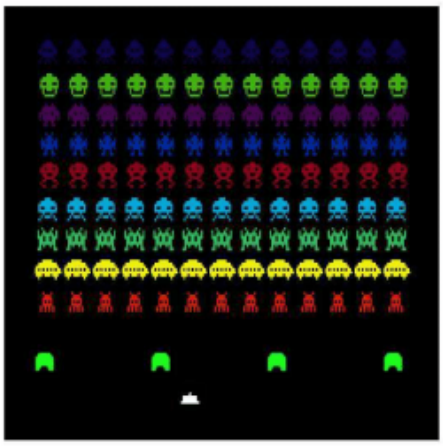
---

## Ejemplo 2-Invaders

### 2.1-Enunciado

# Invaders (invaders.*) Useu: ~/assig/grau-g/Viewer/GLarenaSL

Escriu **VS+FS** per tal de dibuixar quelcom similar a una pantalla del conegut *Space Invaders*. Aquí teniu la textura invaders.jpg, i un exemple del resultat esperat (amb l'objecte plane):



Observeu que la darrera columna de la textura inclou un canó blanc i un escut verd.

El **VS** farà les tasques imprescindibles.

El **FS** serà l'encarregat de triar el color del fragment, d'acord amb les coordenades de textura que rebrà del VS, i que per l'objecte plane estan dins [0,1].

Per tal d'aconseguir la màxima puntuació, caldrà que:

- Hi hagi un canó blanc a la part inferior (4 punts)
- Hi hagi alguns escuts en una fila per sobre del canó (3 punts)
- Hi hagi com a mínim 6 fileres amb els extraterrestres invasors; cada filera mostrarà un únic tipus d'extraterrestre (3 punts).

**El test és només orientatiu.**

**Identificadors obligatoris:**

PROF

`invaders.vert, invaders.frag` (has escrit **invaders** correctament? En minúscules?)
`uniform sampler2D colormap;`

## 2.2-Vertex Shader

```
#version 330 core

// --- INPUTS (del modelo 3D) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (al Fragment Shader) ---
out vec2 vtexCoord;
```

```
// --- UNIFORMS (del viewer) ---
uniform mat4 modelViewProjectionMatrix;

void main()
{
    // Pasar la coordenada de textura
    vtexCoord = texCoord;

    // Calcular la posición final
    gl_Position = modelViewProjectionMatrix * vec4(vertex, 1.0);
}
```

2.3-Fragment Shader

```
#version 330 core

// --- INPUT (from the Vertex Shader) ---
in vec2 vtexCoord; // Coordinate (s, t) in the range [0, 1]

// --- OUTPUT ---
out vec4 fragColor;

// --- UNIFORMS (Added for "invaders") ---
uniform sampler2D colormap; // The "invaders.png" texture

void main()
{
    // ================================================================
    // == "invaders" LOGIC GOES HERE ==
    // ================================================================

    // 1. Default color: BLACK (this is our space background)
    vec4 finalColor = vec4(0.0, 0.0, 0.0, 1.0);

    // The texture is a 4x4 grid
    vec2 tileSize = vec2(0.25, 0.25);
    vec2 tile = vec2(0.0);
    vec2 localCoord = vec2(0.0);
    bool draw = true;

    // Divide the screen into rows using the T (Y) coordinate
    if (vtexCoord.t > 0.8) {
        // Row 6 (e.g., blue alien)
        // Texture (Col 0, Row 3) -> Tile Y-coord = 3
        tile = vec2(0.0, 3.0);
        localCoord.s = fract(vtexCoord.s * 12.0); // 12 aliens
        localCoord.t = (vtexCoord.t - 0.8) / 0.1; // Map [0.8, 0.9] to
[0, 1]
    } else if (vtexCoord.t > 0.7) {
        // Row 5 (e.g., green alien)
```

PROF

```
        // Texture (Col 1, Row 3) -> Tile Y-coord = 3
        tile = vec2(1.0, 3.0);
        localCoord.s = fract(vtexCoord.s * 12.0);
        localCoord.t = (vtexCoord.t - 0.7) / 0.1;
    } else if (vtexCoord.t > 0.6) {
        // Row 4 (e.g., other green alien)
        // Texture (Col 0, Row 2) -> Tile Y-coord = 2
        tile = vec2(0.0, 2.0);
        localCoord.s = fract(vtexCoord.s * 12.0);
        localCoord.t = (vtexCoord.t - 0.6) / 0.1;
    } else if (vtexCoord.t > 0.5) {
        // Row 3 (e.g., purple alien)
        // Texture (Col 1, Row 2) -> Tile Y-coord = 2
        tile = vec2(1.0, 2.0);
        localCoord.s = fract(vtexCoord.s * 12.0);
        localCoord.t = (vtexCoord.t - 0.5) / 0.1;
    } else if (vtexCoord.t > 0.4) {
        // Row 2 (e.g., yellow alien)
        // Texture (Col 0, Row 1) -> Tile Y-coord = 1
        tile = vec2(0.0, 1.0);
        localCoord.s = fract(vtexCoord.s * 12.0);
        localCoord.t = (vtexCoord.t - 0.4) / 0.1;
    } else if (vtexCoord.t > 0.3) {
        // Row 1 (e.g., small red alien)
        // Texture (Col 0, Row 0) -> Tile Y-coord = 0
        tile = vec2(0.0, 0.0);
        localCoord.s = fract(vtexCoord.s * 12.0);
        localCoord.t = (vtexCoord.t - 0.3) / 0.1;
    } else if (vtexCoord.t > 0.2) {
        // Row of SHIELDS (green)
        tile = vec2(3.0, 0.0);

        // --- Centering Logic ---

        // 1. Define row/sprite dimensions
        float rowHeight = 0.1;
        float spriteWidth = 0.1; // To keep a 1:1 aspect ratio
        float numShields = 4.0;

        // 2. Width of one "zone" (1.0 / 4.0 = 0.25)
        float zoneWidth = 1.0 / numShields;

        // 3. Get the local t-coordinate (0.0 to 1.0)
        localCoord.t = (vtexCoord.t - 0.2) / rowHeight;

        // 4. Get the local s-coordinate (0.0 to 1.0)
        localCoord.s = fract(vtexCoord.s * numShields);

        // 5. Calculate ratios for centering
        // How much of the zone does the sprite fill? (0.1 / 0.25 = 0.4)
        float fill_ratio = spriteWidth / zoneWidth;
        // How much padding on *one side*? ( (1.0 - 0.4) / 2.0 = 0.3 )
        float padding_ratio = (1.0 - fill_ratio) / 2.0;
```

```
        // 6. Check if the pixel is in the drawable "window"
        // (i.e., not in the left 30% or right 30% padding)
        if (localCoord.s > padding_ratio && localCoord.s < (1.0 -
padding_ratio))
        {
            // It is in the window [0.3, 0.7].
            // We must re-map this range back to [0.0, 1.0] for the
sprite.

            // 1. Shift it: [0.3, 0.7] -> [0.0, 0.4]
            float shifted_s = localCoord.s - padding_ratio;

            // 2. Scale it: [0.0, 0.4] -> [0.0, 1.0]
            localCoord.s = shifted_s / fill_ratio;
        } else {
            // We are in the padding (the empty 30% on left or right)
            draw = false;
        }
    } else if (vtexCoord.t > 0.1) {
        // Row of the CANNON (white)
        // Texture (Col 3, Row 1) -> Tile Y-coord = 1
        tile = vec2(3.0, 1.0);
        if (vtexCoord.s > 0.45 && vtexCoord.s < 0.55) {
            localCoord.s = (vtexCoord.s - 0.45) / 0.10;
            localCoord.t = (vtexCoord.t - 0.1) / 0.10;
        } else {
            draw = false;
        }
    } else {
        // Empty space at the bottom
        draw = false;
    }

    // 2. If 'draw' is true, we attempt to draw a sprite.
    if (draw) {

        // ================================================
        // == CHANGE IS HERE ==
        // The line 'localCoord.t = 1.0 - localCoord.t;'
        // was REMOVED to fix the upside-down sprites.
        // ================================================

        // Calculate the final coordinate to sample from the texture
        vec2 sampleCoord = (tile + localCoord) * tileSize;

        // Read the texture color
        vec4 texColor = texture(colormap, sampleCoord);

        // Check the brightness
        // (The texture atlas has a black background)
        float brightness = texColor.r + texColor.g + texColor.b;
```

```
        if (brightness > 0.1) {
            // If it's NOT the black background,
            // update finalColor to the sprite color.
            finalColor = texColor;
        }
        // If it IS the black background (brightness < 0.1),
        // we do nothing. finalColor stays black.
    }
    // If 'draw' is false, we do nothing.
    // finalColor stays black from the beginning.

    // ================================================================

    // --- Final Assignment ---
    fragColor = finalColor;
}
```

---

## Ejemplo 3-Flag

### 3.1-Enunciado

---

# Flag (no hi ha test)

Escriu **VS+FS** que, amb l'objecte **plane.obj,** dibuixi de forma procedural una bandera similar a aquesta:

El VS farà les tasques imprescindibles, escalant la coordenada Y per tal que la relació d'aspecte sigui 2:1.

El FS calcularà el color per tal reproduir quelcom *semblant* a la figura.

**Identificadors obligatoris:**
`flag.vert, flag.frag`

PROF

### 3.2-Vertex Shader

```
#version 330 core

// --- INPUTS (from your 3D model) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (to the Fragment Shader) ---
out vec2 vtexCoord;

// --- UNIFORMS (from the viewer) ---
uniform mat4 modelViewProjectionMatrix;

void main()
{
    vtexCoord = texCoord;

    // Create a new vertex variable
    vec4 new_vertex = vec4(vertex, 1.0);

    // Scale the Y coordinate by 0.5 to make the
    // 1x1 plane into a 1x0.5 (2:1 aspect ratio) rectangle
    new_vertex.y = new_vertex.y * 0.5;

    // Calculate the final position using the MODIFIED vertex
    gl_Position = modelViewProjectionMatrix * new_vertex;
}
```

### 3.3-Fragment Shader

PROF

```
#version 330 core

// --- INPUT (del Vertex Shader) ---
in vec2 vtexCoord;

// --- OUTPUT ---
out vec4 fragColor;

// --- CONSTANTS (Definimos los colores y formas) ---
const vec4 COLOR_GREEN  = vec4(0.0, 1.0, 0.0, 1.0);
const vec4 COLOR_YELLOW = vec4(1.0, 1.0, 0.0, 1.0);
const vec4 COLOR_BLUE   = vec4(0.0, 0.0, 1.0, 1.0);

// --- Constantes del Círculo Azul ---
const float BLUE_CIRCLE_RADIUS = 0.12;
const vec2  BLUE_CENTER_TEX = vec2(0.5, 0.5); // Centro de la pantalla
```

```
// --- Constantes de la Banda Amarilla ---
const float BAND_OUTER_RADIUS = 0.20; // Más grande que el azul (0.12)
const float BAND_THICKNESS = 0.02;
const vec2  BAND_CENTER_TEX = vec2(0.5, 0.70); // Centro movido a T=0.7

void main()
{
    // a. Empezar con el fondo verde
    vec4 finalColor = COLOR_GREEN;

    // b. Dibujar el rectángulo amarillo
    if (vtexCoord.s > 0.15 && vtexCoord.s < 0.85 && vtexCoord.t > 0.15
&& vtexCoord.t < 0.85) {
        finalColor = COLOR_YELLOW;
    }

    // --- 2. Círculo Azul (Cálculo separado) ---
    // Centramos y corregimos el aspecto para el círculo azul
    float centered_s_blue = vtexCoord.s - BLUE_CENTER_TEX.s;
    float centered_t_blue = (vtexCoord.t - BLUE_CENTER_TEX.t) * 0.5;
    float dist_blue = length(vec2(centered_s_blue, centered_t_blue));

    // Dibujamos el círculo azul
    if (dist_blue < BLUE_CIRCLE_RADIUS) finalColor = COLOR_BLUE;

    // --- 3. Banda Amarilla (Cálculo separado) ---
    // Centramos y corregimos el aspecto para la banda amarilla
    float centered_s_band = vtexCoord.s - BAND_CENTER_TEX.s;
    float centered_t_band = (vtexCoord.t - BAND_CENTER_TEX.t) * 0.5;
    float dist_band = length(vec2(centered_s_band, centered_t_band));

    // Calculamos el radio interior
    float band_inner_radius = BAND_OUTER_RADIUS - BAND_THICKNESS;

    // Comprobamos si el píxel está en el "anillo"
    if (dist_band < BAND_OUTER_RADIUS && dist_band > band_inner_radius)
    {
        // "Dibujar solo si el píxel está en la mitad inferior de la
pantalla
        if (vtexCoord.t < 0.5) finalColor = COLOR_YELLOW;
    }

    // --- Asignación Final ---
    fragColor = finalColor;
}
```

PROF

# Ejercicio Tipo 3-Iluminación (Phong)

- **Objetivo:** Calcular el modelo de luz de Phong (Ambiental, Difuso, Especular) para cada **píxel**.
- **Palabras Clave:** `Phong`, `Iluminación`, `N`, `L`, `V`, `R`, `matDiffuse`, `lightSpecular`, etc.
- **Ejemplos:** `Nlights`, `LightChange`, `8lights`.
- **Dónde trabajas:** El cálculo principal va en el **Fragment Shader (`.frag`)**.
- **Esqueleto a Usar: Esqueleto 2 (Per-Fragment)**.
- **Por qué:** Este es el caso más claro para el Esqueleto 2. Debes usar tu "cheatsheet" para añadir los "snippets" **obligatorios**:
    1. **"Pasar Normal"** (para obtener `v_normal_eye` para **N**).
    2. **"Pasar Posición del Ojo"** (para obtener `v_position_eye` para **V** y **L**).
    3. **Todos los `uniforms`** de materiales (`mat...`) y luces (`light...`) que necesites.

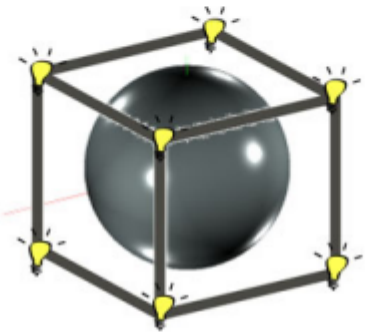
---

# Ejemplo 1-8lights

## 1.1-Enunciado

## 8lights (8lights.*) 2.5 punts

Escriu **VS+FS** per aplicar il·luminació de Phong **per fragment**, amb **8 llums fixos respecte l'escena**. Concretament, les posicions dels llums en *world space* coincidiran amb els 8 vèrtexs de la capsa contenidora de l'escena (useu boundingBoxMin i boundingBoxMax per obtenir la posició d'aquests llums).



El VS farà les tasques habituals i passarà al FS les dades necessàries (vèrtex i normal) pel càlcul d'il·luminació.

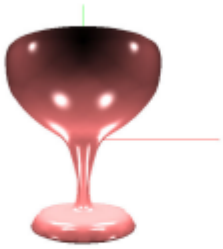
El FS calcularà el color del fragment acumulant la contribució dels *8* llums. Per evitar imatges massa saturades, useu per cada llum l'expressió

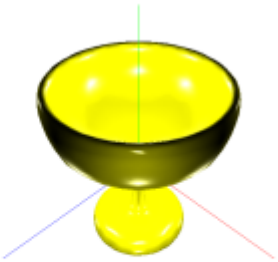
$$\sum K_d I_d (N \cdot L_i)/\mathbf{2} \; + \; K_s I_s (R_i \cdot V)^S$$

la qual **ignora la contribució ambient** i **divideix la contribució difosa per 2.**

Pels llums i material usa les propietats habituals (matDiffuse, matSpecular, lightDiffuse, lightSpecular...).

Vigila amb l'eficiència, per exemple, mira de no fer crides innecessàries a normalize().





PROF

**Identificadors obligatoris:**

8lights.vert, 8lights.frag *(minúscules!)*

### 1.2-Vertex Shader

```
#version 330 core

// --- INPUTS (from your 3D model) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;
```

```
// --- OUTPUTS (to the Fragment Shader) ---
out vec2 vtexCoord;
out vec3 v_normal_eye;   // Added from cheatsheet
out vec3 v_position_eye; // Added from cheatsheet

// --- UNIFORMS (from the viewer) ---
// We need these for the snippets
uniform mat3 normalMatrix;   // Added from cheatsheet
uniform mat4 modelViewMatrix;  // Added from cheatsheet
uniform mat4 projectionMatrix; // Added from cheatsheet

void main()
{
    // Pass the texture coordinate
    vtexCoord = texCoord;

    // --- "Pass Eye Position" Logic ---
    vec4 pos_eye_4 = modelViewMatrix * vec4(vertex, 1.0);
    v_position_eye = vec3(pos_eye_4);

    // --- "Pass Normal" Logic ---
    v_normal_eye = normalize(normalMatrix * normal);

    // --- "Pass Eye Position" Logic (continued) ---
    gl_Position = projectionMatrix * pos_eye_4;
}
```

### 1.3-Fragment Shader

```
#version 330 core

// --- INPUT (from the Vertex Shader) ---
in vec2 vtexCoord;
in vec3 v_normal_eye;   // Added from cheatsheet
in vec3 v_position_eye; // Added from cheatsheet

// --- OUTPUT ---
out vec4 fragColor;

// --- UNIFORMS (Added for "8lights") ---
// Standard lighting uniforms (from cheatsheet)
uniform vec4 lightDiffuse;
uniform vec4 lightSpecular;
uniform vec4 matDiffuse;
uniform vec4 matSpecular;
uniform float matShininess;

// Uniforms to get light positions
uniform vec3 boundingBoxMin;
uniform vec3 boundingBoxMax;
```

PROF

```
uniform mat4 viewMatrix; // To convert lights from World to Eye space

void main()
{
    // ================================================================
    // == STEP 3 (FS) - "8lights" Logic
    // ================================================================

    // --- a. Initialize ---
    // Start with black, as the formula ignores ambient
    vec4 finalColor = vec4(0.0, 0.0, 0.0, 1.0);

    // --- b. Define the 8 light positions in World Space ---
    // (We get these from the bounding box corners)
    vec3 lightPos_world[8];
    lightPos_world[0] = vec3(boundingBoxMin.x, boundingBoxMin.y,
boundingBoxMin.z); // 0
    lightPos_world[1] = vec3(boundingBoxMax.x, boundingBoxMin.y,
boundingBoxMin.z); // 1
    lightPos_world[2] = vec3(boundingBoxMin.x, boundingBoxMax.y,
boundingBoxMin.z); // 2
    lightPos_world[3] = vec3(boundingBoxMax.x, boundingBoxMax.y,
boundingBoxMin.z); // 3
    lightPos_world[4] = vec3(boundingBoxMin.x, boundingBoxMin.y,
boundingBoxMax.z); // 4
    lightPos_world[5] = vec3(boundingBoxMax.x, boundingBoxMin.y,
boundingBoxMax.z); // 5
    lightPos_world[6] = vec3(boundingBoxMin.x, boundingBoxMax.y,
boundingBoxMax.z); // 6
    lightPos_world[7] = vec3(boundingBoxMax.x, boundingBoxMax.y,
boundingBoxMax.z); // 7

    // --- c. Get N and V (constants for all lights) ---
    vec3 N = normalize(v_normal_eye);
    vec3 V = normalize(-v_position_eye); // Vector to viewer

    // --- d. Start the loop ---
    for (int i = 0; i < 8; i++)
    {
        // --- e. Transform this light's pos to Eye Space ---
        vec3 lightPos_eye = vec3(viewMatrix * vec4(lightPos_world[i],
1.0));

        // --- f. Calculate L and R for this light ---
        vec3 L = normalize(lightPos_eye - v_position_eye);
        vec3 R = reflect(-L, N);

        // --- g. Calculate Diffuse & Specular (per formula) ---
        float NdotL = max(0.0, dot(N, L));

        // Diffuse term (divided by 2)
        vec4 dif = (matDiffuse * lightDiffuse * NdotL) / 2.0;
```

PROF

```
        // Specular term
        vec4 spec = vec4(0.0);
        if (NdotL > 0.0) // only add specular if light hits
        {
            float RdotV = pow(max(0.0, dot(R, V)), matShininess);
            spec = matSpecular * lightSpecular * RdotV;
        }

        // --- h. Accumulate ---
        finalColor += dif + spec;
    }

    finalColor.a = 1.0; // Ensure alpha is 1

    // ================================================================


    // --- Final Assignment ---
    fragColor = finalColor;
}
```

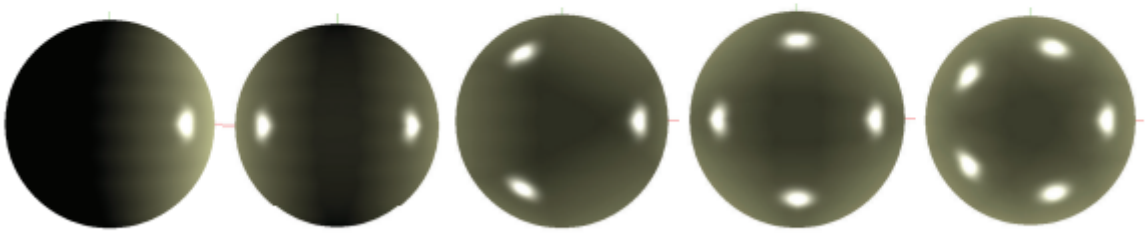
---

## Ejemplo 2-Nlights

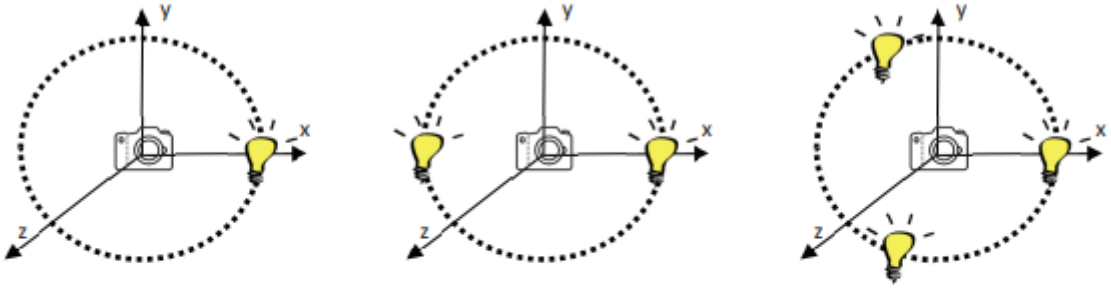
### 2.1-Enunciado

# Nlights (nlights.*)

Escriu **VS+FS** per aplicar il·luminació de Phong **per fragment**, amb n llums fixos respecte la càmera, on n és un **uniform int n=4**. Aquí tens l'esfera amb n=1-5 llums:



Els llums estaran situats al voltant d'un cercle de **radi 10** situat al **pla Z=0 de la càmera** i centrat a la càmera. El primer llum estarà situat al punt de coordenades eye space (10, 0, 0), i la resta estaran equidistribuïts seguint el cercle, com es mostra a la figura per n=1-3 llums:




El VS farà les tasques habituals i passarà al FS les dades necessàries (vèrtex i normal) pel càlcul d'il·luminació.

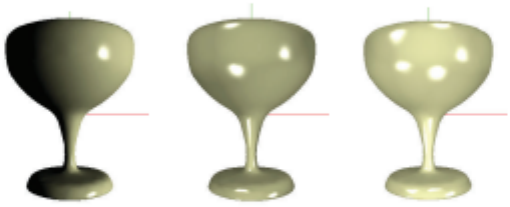
El FS calcularà el color del fragment acumulant la contribució dels *n* llums. Per evitar imatges massa saturades, useu l'expressió

$$\sum K_d\, I_d\, (N \cdot L_i)/\sqrt{n} \;+\; K_s\, I_s\, (R_i \cdot V)^S$$

la qual **ignora la contribució ambient** i **divideix la contribució difosa per $\sqrt{n}$.**

PROF

Aquí teniu la copa amb n=1,3,5 llums:



Vigila amb l'eficiència, per exemple, mira de no fer crides innecessàries a normalize().

**Identificadors obligatoris:**

```
uniform int n = 4;
const float pi = 3.141592;
```

## 2.2-Vertex Shader

```
#version 330 core

// --- INPUTS (from your 3D model) ---
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

// --- OUTPUTS (to the Fragment Shader) ---
out vec2 vtexCoord;
out vec3 v_normal_eye;   // <-- From "Pass Normal"
out vec3 v_position_eye; // <-- From "Pass Eye Position"

// --- UNIFORMS (from the viewer) ---
// We need these for the snippets
uniform mat3 normalMatrix;   // <-- From "Pass Normal"
uniform mat4 modelViewMatrix;  // <-- From "Pass Eye Position"
uniform mat4 projectionMatrix; // <-- From "Pass Eye Position"

void main()
{
    // Pass the texture coordinate
    vtexCoord = texCoord;

    // --- "Pass Eye Position" Logic ---
    vec4 pos_eye_4 = modelViewMatrix * vec4(vertex, 1.0);
    v_position_eye = vec3(pos_eye_4);

    // --- "Pass Normal" Logic ---
    v_normal_eye = normalMatrix * normal;

    // --- "Pass Eye Position" Logic (continued) ---
    gl_Position = projectionMatrix * pos_eye_4;
}
```

PROF

## 2.3-Fragment Shader

```
#version 330 core

// --- INPUT (from the Vertex Shader) ---
in vec2 vtexCoord;
in vec3 v_normal_eye;   // <-- From "Pass Normal"
in vec3 v_position_eye; // <-- From "Pass Eye Position"

// --- UNIFORMS (Add as needed) ---
// Problem-specific uniforms
uniform int n = 4;
const float pi = 3.141592;
```

```
// Standard lighting uniforms (from cheatsheet)
uniform vec4 lightDiffuse;
uniform vec4 lightSpecular;
uniform vec4 matDiffuse;
uniform vec4 matSpecular;
uniform float matShininess;

// --- OUTPUT ---
out vec4 fragColor;

void main()
{
    // ================================================================
    // == STEP 3 (FS) - "Nlights" Logic
    // ================================================================

    // --- a. Initialize ---
    // Start with black, since there is no ambient term
    vec4 finalColor = vec4(0.0, 0.0, 0.0, 1.0);

    // --- b. Get N and V (constants for all lights) ---
    vec3 N = normalize(v_normal_eye);
    vec3 V = normalize(-v_position_eye); // Vector to viewer

    // --- c. Get the formula's divisor ---
    float sqrt_n = sqrt(float(n));

    // --- d. Start the loop ---
    for (int i = 0; i < n; i++)
    {
        // --- e. Calculate this light's position ---
        float angle = 2.0 * pi * float(i) / float(n);
        vec3 lightPos_eye = vec3(10.0 * cos(angle), 10.0 * sin(angle),
0.0);

        // --- f. Calculate L and R for this light ---
        vec3 L = normalize(lightPos_eye - v_position_eye);
        vec3 R = reflect(-L, N);

        // --- g. Calculate Diffuse & Specular (per formula) ---
        float NdotL = max(0.0, dot(N, L));

        // Diffuse term (divided by sqrt(n))
        vec4 dif = (matDiffuse * lightDiffuse * NdotL) / sqrt_n;

        // Specular term
        vec4 spec = vec4(0.0);
        if (NdotL > 0.0) // only add specular if light hits
        {
            float RdotV = pow(max(0.0, dot(R, V)), matShininess);
            spec = matSpecular * lightSpecular * RdotV;
        }
```

PROF

```
        // --- h. Accumulate ---
        finalColor += dif + spec;
    }

    finalColor.a = 1.0; // Ensure alpha is 1

    // ================================================================


    // --- Final Assignment ---
    fragColor = finalColor;
}
```

---

## Ejemplo 3-

### 3.1-Enunciado

### 3.2-Vertex Shader

```

```

### 3.3-Fragment Shader

```

```

---

## Ejemplo 3-

PROF

### 3.1-Enunciado

### 3.2-Vertex Shader

```

```

### 3.3-Fragment Shader

```

```

---