0. Introducció

Els exercicis d'aquesta llista consisteixen a **implementar shaders** en llenguatge **GLSL 3.30 core profile.** A diferència del Compatibility Profile, el Core profile no defineix moltes variables uniforms (per exemple, matrius, materials, llums...) que són útils per experimentar amb shaders. Per aquest motiu, els exercicis estan pensats per ser executats dins l'entorn que proporciona el **viewer** (que defineix un petit conjunt d'aquestes variables) que fem servir a l'assignatura. Aquí teniu un resum d'aquest entorn:

Atributs (enviats pel viewer; per usar-los cal declarar-los als shaders):

```
layout (location = 0) in vec3 vertex;    // similar a gl_Vertex (però 3D)
layout (location = 1) in vec3 normal;    // idèntic a gl_Normal
layout (location = 2) in vec3 color;     // similar a gl_Color (però RGB)
layout (location = 3) in vec2 texCoord;  // similar a gl_MultiTexCoord0
```

Uniforms (enviats automàticament pel viewer; per usar-los cal declarar-los als shaders):

```
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 modelViewProjectionMatrix;

uniform mat4 modelMatrixInverse;
uniform mat4 viewMatrixInverse;
uniform mat4 projectionMatrixInverse;
uniform mat4 modelViewMatrixInverse;
uniform mat4 modelViewProjectionMatrixInverse;

uniform mat3 normalMatrix;

uniform vec4 lightAmbient;   // similar a gl_LightSource[0].ambient
uniform vec4 lightDiffuse;   // similar a gl_LightSource[0].diffuse
uniform vec4 lightSpecular;  // similar a gl_LightSource[0].specular
uniform vec4 lightPosition;  // similar a gl_LightSource[0].position
                             // (sempre estarà en eye space)

uniform vec4 matAmbient;     // similar a gl_FrontMaterial.ambient
uniform vec4 matDiffuse;     // similar a gl_FrontMaterial.diffuse
uniform vec4 matSpecular;    // similar a gl_FrontMaterial.specular
uniform float matShininess;  // similar a gl_FrontMaterial.shininess

uniform vec3 boundingBoxMin; // cantonada mínima de la capsa englobant
uniform vec3 boundingBoxMax; // cantonada màxima de la capsa englobant

uniform vec2 mousePosition;  // coordenades del cursor (window space)
                             // origen a la cantonada inferior esquerra
```

Si definiu variables uniform pròpies al vostre shader, el viewer us permetrà (en alguns casos) canviar-ne el valor. **En el cas de uniform definits per vosaltres, és molt recomanable donar-los un valor inicial adient** (això facilita provar els shaders immediatament, sense haver de tornar a entrar els valors dels uniforms cada cop que activem el shader). Exemples:

```
uniform bool hasTexture = true;
uniform float scale = 1.0;
```

El Core Profile també implica que **sempre cal proporcionar el Vertex Shader (VS) i el Fragment Shader (FS).** Per simplificar el desenvolupament, el viewer us proporciona un VS i un FS "per defecte" que podeu usar com a punt de partida.

Si en un exercici només es demana un VS i un FS, aquest haurà de funcionar correctament amb la versió "per defecte" de l'altre. En el moment d'escriure aquesta llista, les versions per defecte són aquestes (la versió actualitzada les podeu obtenir amb el viewer, amb la opció "New VS+FS...")

// VS

```
#version 330 core
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec3 color;
layout (location = 3) in vec2 texCoord;

out vec4 frontColor;
out vec2 vtexCoord;

uniform mat4 modelViewProjectionMatrix;
uniform mat3 normalMatrix;

void main() {
    vec3 N = normalize(normalMatrix * normal);
    gl_Position = modelViewProjectionMatrix * vec4(vertex.xyz, 1.0);
    frontColor = vec4(color,1.0) * N.z;
    vtexCoord = texCoord;
}
```

// FS

```
#version 330 core

in vec4 frontColor;
out vec4 fragColor;

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

Alguns exercicis que han sortit en controls d'altres anys fan referència a codi a completar. Podeu trobar els arxius a **/assig/grau-g/Arxius...** Nota: el codi antic està escrit en compatibility profile i us caldrà editar-lo.

1. Basic Lighting 1 (basicLighting1.*)

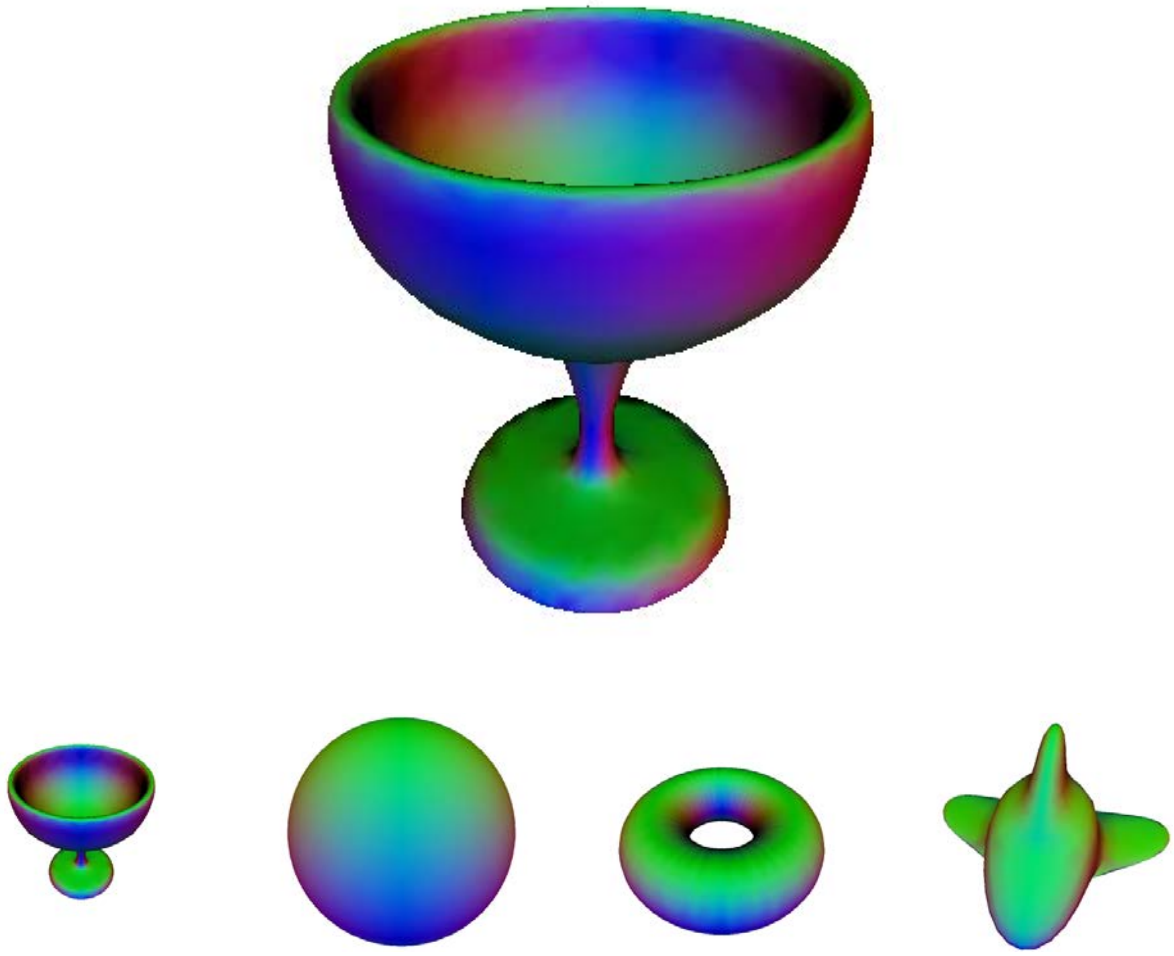
Durant el desenvolupament de shaders sovint ens pot interessar aconseguir un cert efecte d'il·luminació de la forma més senzilla possible.

Una possibilitat molt senzilla és multiplicar el color original per la component Z de la normal en coordenades de la càmera. L'efecte resultant és similar al que produiria el model de Lambert amb una font de llum blanca direccional alineada amb l'eix Z de la càmera.

Escriu un **VS** que calculi la il·luminació per vèrtex fent servir aquesta tècnica.

Recorda que per passar vectors normals de object space a eye space cal multiplicar per la normalMatrix.

Aquí tens un exemple del resultat esperat amb diversos models:

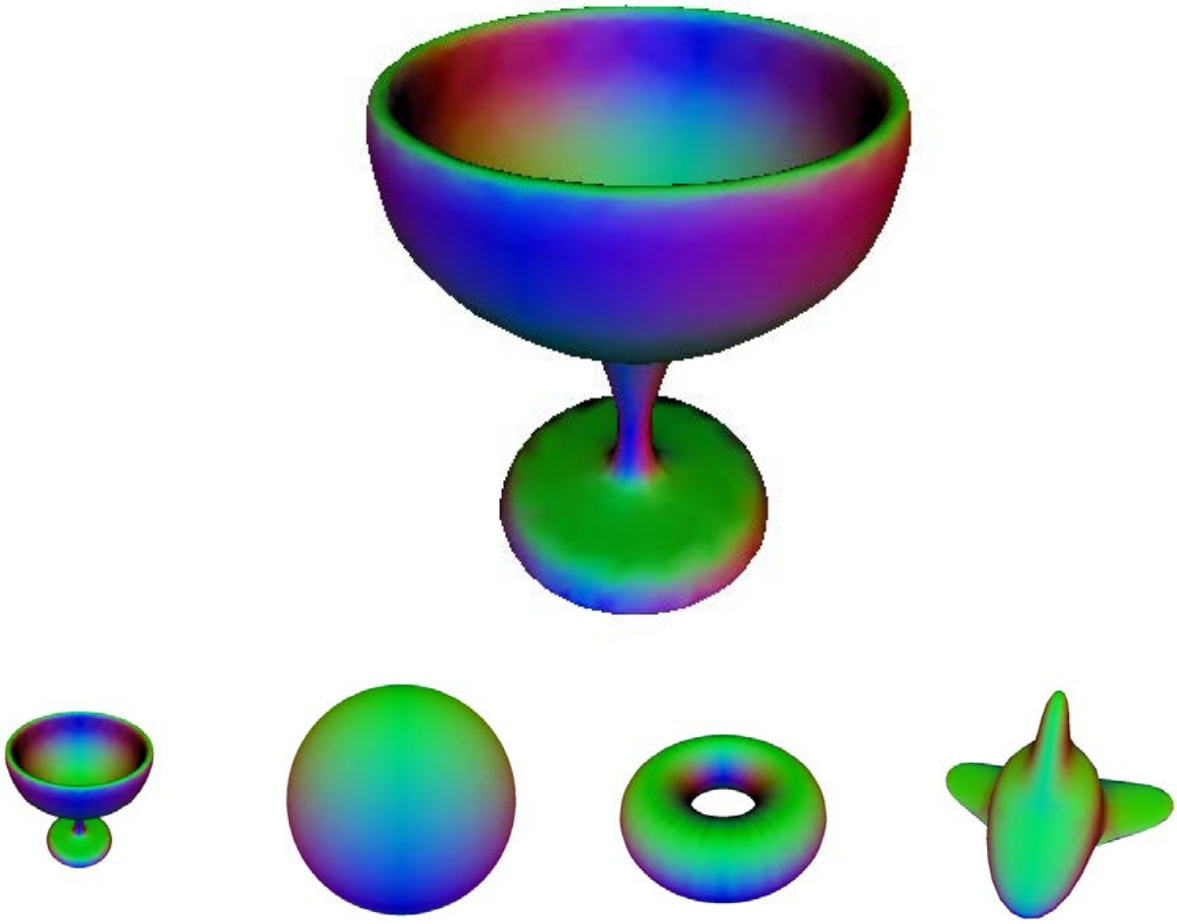


2. Basic Lighting 2 (basicLighting2.*)

Com a l'exercici anterior, volem aconseguir un efecte d'il·luminació de la forma més senzilla possible, multiplicant el color original per la component Z de la normal en coordenades de la càmera.

Escriu un **VS** i un **FS** que calculi la il·luminació **per fragment** fent servir aquesta tècnica. Per tant, caldrà que el VS li passi la normal al FS, per tal que aquest calculi la il·luminació amb la component Z de la normal (com abans, en eye space).

Aquí tens un exemple del resultat esperat amb diversos models:



3. Lighting 1 (lighting1.*)

Escriu un **vertex shader** que calculi la il·luminació per vèrtex de forma anàloga a com ho fa el pipeline fix d'OpenGL, fent servir les propietats del material i de la font de llum.

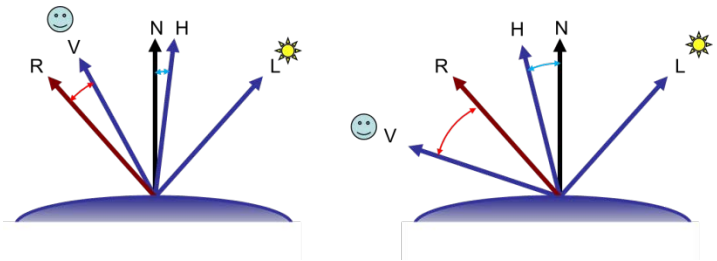
Només cal considerar una font de llum, que no és SPOT. La fórmula a aplicar serà la de **Blinn-Phong:**

$$K_aI_a + K_dI_d(N \cdot L) + K_sI_s(N \cdot H)^s$$

on

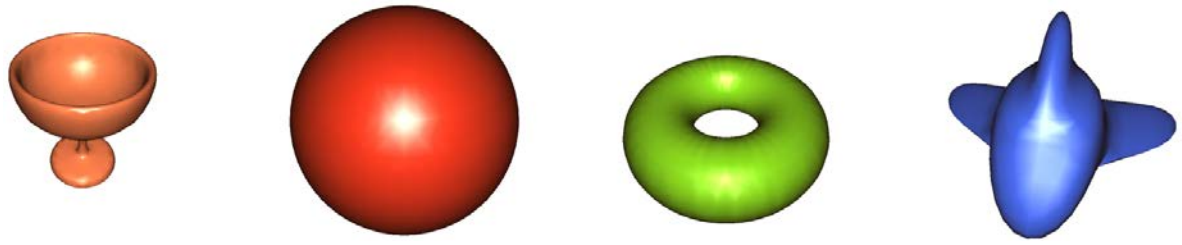
K_a, K_d, K_s = reflectivitat ambient, difosa i especular del material
s = shininess del material
I_a, I_d, I_s = propietats ambient, difosa i especular de la llum.
N = vector normal unitari (eye space)
L = vector unitari cap a la font de llum (eye space)
H = half vector = vector a mig camí entre V i L, on V és un vector unitari del vèrtex cap a la càmera. Es calcula com H = V + L, i normalitzant. Per defecte, OpenGL calculava H com si V fos (0, 0, 1), és a dir, com si l'observador estigués a l'infinit en direcció de les Z (així és com ho heu de fer per passar els tests).

A diferència del model de Phong, el model de Blinn calcula el terme especular a partir del cosinus de l'angle format pels vectors N i H (en blau a la figura), en comptes del format pels vectors R i V (en vermell):



Observació: a la fórmula anterior, cal evitar "restar" il·luminació quan els productes escalars N·L o N·H són negatius. Per tant haureu de fer servir max(0.0, dot(N, L)) i max(0.0, dot(N, H)).

Aquí teniu un exemple:



4. Lighting 2 (lighting2.*)

Escriu un **vertex shader** que calculi la il·luminació per vèrtex amb el **model de Phong.**

La fórmula a aplicar serà la de Phong:

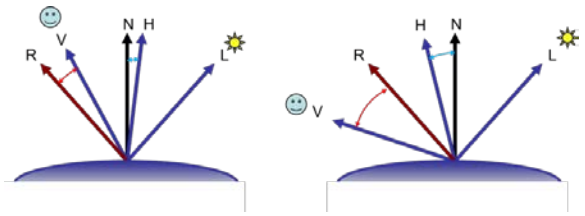
$$K_aI_a + K_dI_d(N\cdot L) + K_sI_s(R\cdot V)^s$$

on

K_a, K_d, K_s = reflectivitat ambient, difosa i especular del material
s = shininess del material
I_a, I_d, I_s = propietats ambient, difosa i especular de la llum
N = vector normal unitari
L = vector unitari cap a la font de llum
V = vector unitari del vèrtex cap a la càmera
R = reflexió del vector L respecte N. Es pot calcular com R=2(N·L)N-L

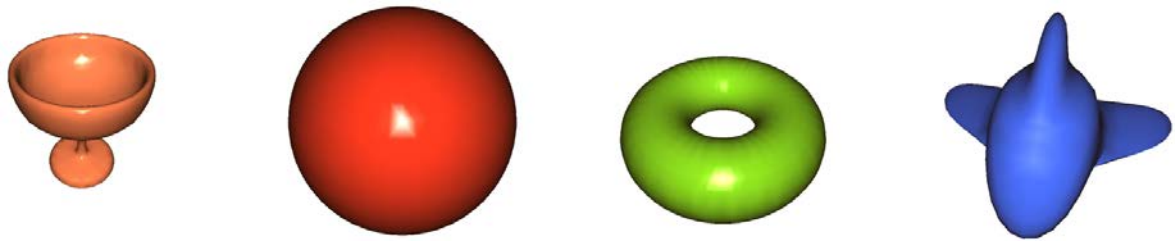
Totes les components haurien d'estar en coordenades de la càmera.

A diferència del model de Blinn, el model de Phong calcula el terme especular a partir del cosinus de l'angle format pels vectors R i V (en vermell a la figura), en comptes del format pels vectors N i H (en blau):



Observació: a la fórmula anterior, cal evitar "restar" il·luminació quan els productes escalars N·L o R·V són negatius. Per tant haureu de fer servir max(0.0, dot(N, L)) i max(0.0, dot(R, V)).

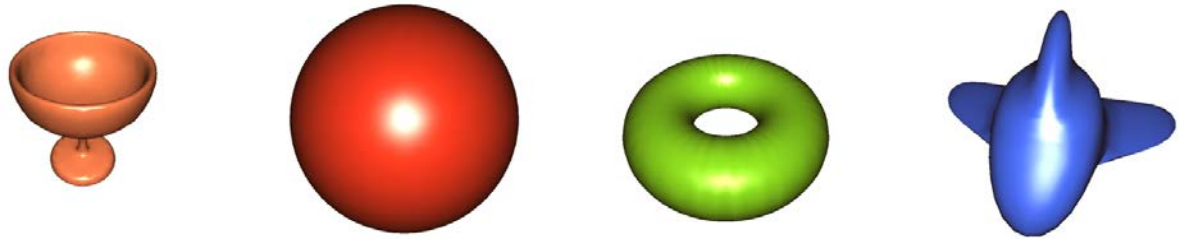
Aquí teniu un exemple:



5. Lighting 3 (lighting3.*)

Escriu un **vertex shader** i un **fragment shader** que calculi la il·luminació **per fragment** fent servir el model de **Blinn.**

Necessitareu que el fragment shader tingui accés a la posició del fragment i al vector normal (tots dos en eye space), per tant el vertex shader haurà d'enviar-li aquestes variables.

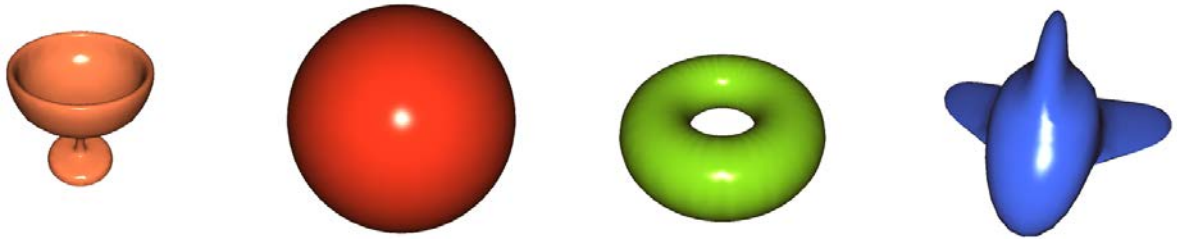


Aquesta imatge compara Blinn per vèrtex amb Blinn per fragment:

6. Lighting 4 (lighting4.*)

Escriu un **vertex shader** i un **fragment shader** que calculi la il·luminació per fragment fent servir el model de **Phong**.

Necessitareu que el fragment shader tingui accés a la posició del fragment i al vector normal (tots dos en eye space).



Aquí pots comparar Phong per vèrtex i Phong per fragment;

7. Lighting 5 (lighting5.*) (1er control lab, curs 2011-12, Q2)

Aquest problema és similar a Lighting (4), però es demana que els càlculs d'il·luminació es facin en eye space o world space depenent d'una variable uniform.

Escriu un **vertex shader** i un **fragment shader** que calculi la il·luminació **per fragment** fent servir el model de Phong. El color C del fragment es calcula en funció dels vectors N, V i L,

$$C(N, V, L) = K_a I_a + K_d I_d (N \cdot L) + K_s I_s (R \cdot V)^s$$

on

N = vector normal unitari en el punt
V = vector unitari del punt cap a la càmera
L = vector unitari del punt cap a la font de llum

Per tal de simplificar el problema, us proporcionem aquesta implementació de C(N, V, L), que podeu copiar i pegar al vostre shader i podeu cridar sense cap modificació:

```
vec4 light(vec3 N, vec3 V, vec3 L)
{
  N=normalize(N); V=normalize(V); L=normalize(L);
  vec3 R = normalize( 2.0*dot(N,L)*N-L );
  float NdotL = max( 0.0, dot( N,L ) );
  float RdotV = max( 0.0, dot( R,V ) );
  float Idiff = NdotL;
  float Ispec = 0;
  if (NdotL>0) Ispec=pow( RdotV, matShininess );
  return
    matAmbient  * lightAmbient +
    matDiffuse  * lightDiffuse * Idiff+
    matSpecular * lightSpecular * Ispec;
}
```

Els shaders rebran una variable **uniform bool world** indicant si els càlculs d'il·luminació s'han de fer en world space o eye space. Si world és fals, el vostre fragment shader haurà de cridar a light(N,V,L) amb N,V i L en eye space. Si és cert, s'haurà de cridar amb N,V i L en world space. **Important**: heu d'assumir que en aquest exercici la transformació de modelat és la identitat; per tant en aquest exercici object space i world space coincideixen.

La imatge resultant en tots dos casos haurà de ser la mateixa.

8. Basic Testure (basicTexture.*)

Escriu un **vertex shader** i un **fragment shader** per tenir il·luminació bàsica per vèrtex juntament amb textura.

El vertex shader haurà de calcular una il·luminació bàsica per cada vèrtex, per exemple com

```
frontColor = vec4(normalize(normalMatrix * normal).z);
```

Donat que el fragment shader utilitzarà coordenades de textura, el vertex shader haurà de passar-li el varying corresponent:

```
vtexCoord = texCoord;
```

El fragment shader calcularà el color del fragment simplement multiplicant el color que li arriba (amb il·luminació) pel color de la textura:

```
fragColor = frontColor * texture(colorMap, vtexCoord);
```

On la variable colorMap és una textura 2D:

```
uniform sampler2D colorMap;
```

Aquí teniu exemples del resultat esperat.

9. Animate Texture (animate-texture.*)

Modifica el vertex shader de Basic Texture per tal d'animar una mica la textura.

El que heu de fer és modificar la coordenada s (ó la coordenada t, o totes dues) en funció del temps (per exemple, incrementant la coordenada de textura a velocitat constant, segons un paràmetre

```
uniform float speed=0.1
```

proporcionat per l'usuari.

Feu servir el uniform time que us proporciona el viewer.

Aquí teniu un exemple del resultat esperat:

10. Tiling (tiling.*) (1er control laboratori, curs 2011-12, Q2)

Aquest exercici és semblant a Basic Texture, però sense il·luminació i modificant les coordenades de textura al vertex shader per tal de modificar el nombre de cops que es repeteix la textura.

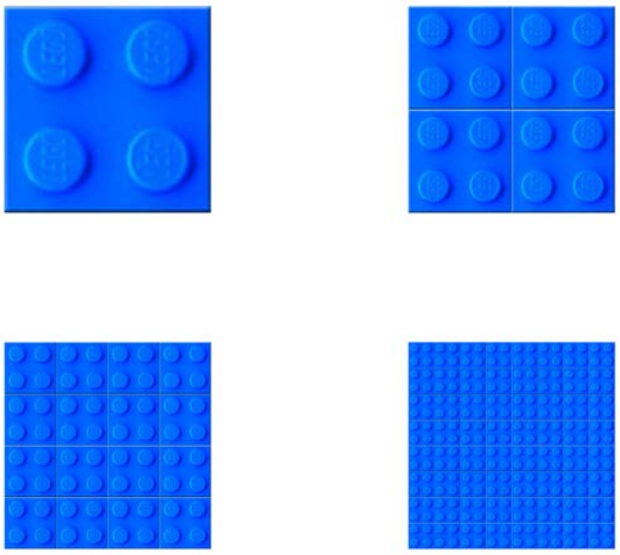
El **vertex shader** farà servir una variable

```
uniform int tiles=1
```

que indicarà el nombre de cops que volem que es repeteixi la textura (en horitzontal i vertical) respecte la parametrització original del model.

El **fragment shader** simplement calcularà el color del fragment assignant-li el color de la textura (sense il·luminació).

Aquí teniu un exemple del resultat esperat amb plane.obj i tiles=1,2,4 i 8.



11. Magnet (magnet.*) (variació del 1er control de laboratori, 2012-13 Q2)

Volem deformar la malla com si la font de llum fos una mena d'imant que exerceix una força sobre els vèrtexs propers de la malla.

Aquí teniu el resultat esperat amb l'objecte glass:

Escriu un **vertex shader** que calculi la nova posició del vèrtex V' (en *object space*) com a interpolació lineal entre la posició original del vèrtex V i la posició de la llum F,

$$V' = (1.0-w)V + wF$$

on el paràmetre d'interpolació lineal w el calcularem com

$$w = \text{clamp}(1/d^n, 0, 1)$$

on d és la distància entre V i F, i el paràmetre n és un **uniform float n=4** que controla la velocitat amb la que decreix la influència de la llum sobre els vèrtexs.

Nota: recordeu que el shader rep la posició del font de llum en *eye space*, però heu de fer els càlculs anteriors en *object space*.

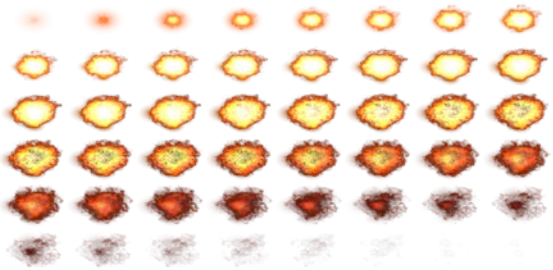
Un cop calculat el nou vèrtex, el VS escriurà la posició en clip space com és usual, i calcularà el color del vèrtex utilitzant directament la component z del vector normal en *eye space* (similar a l'exercici de basic lighting per vèrtex).

Identificadors

```
magnet.vert magnet.frag
uniform float n=4;
```

12. Explosion (explosion.*) (1er control de laboratori, 2013-14 Q1)

Una forma molt senzilla de simular una explosió consisteix en mostrar en seqüència diverses imatges capturades d'una explosió real (o simulada), animant-les en el temps. Per comoditat, podem ajuntar totes les imatges (*frames* de l'animació) en una única textura. Per aquest exercici usarem la textura RGBA **explosion.png** (cortesia d'April Young) que conté 8x6=48 frames, organitzats per files, i **frames.png**:



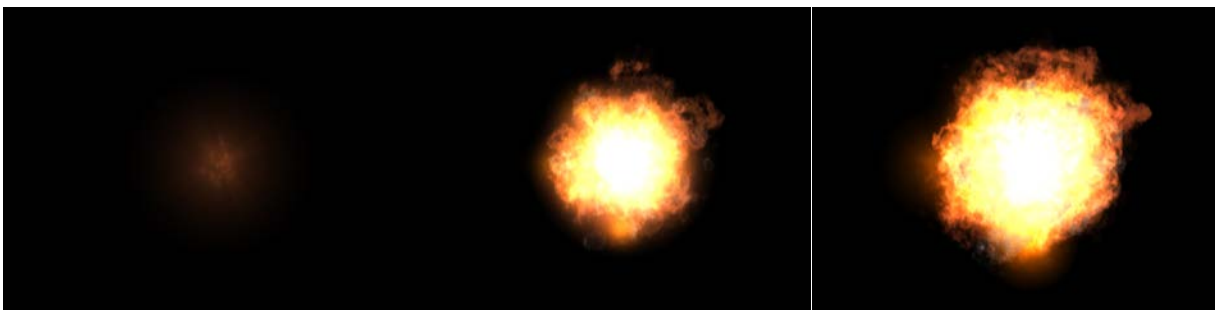
00	01	02	03	04	05	06	07
08	09	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47

Programa un **fragment shader** que mostri seqüencialment cadascun d'aquests frames, modificant convenientment les coordenades de textura. Per simplificar l'exercici, la velocitat de l'animació serà de 30 frames per segon, i per tant cada frame es mostrarà durant un slice de 1/30 segons. D'acord amb el valor de time, el frame que s'haurà de mostrar és:

- $0 \leq time < slice$: mostra frame 0
- $slice \leq time < 2 * slice$: mostra frame 1
- $2 * slice \leq time < 3 * slice$: mostra frame 2
- ...
- $47 * slice \leq time < 48 * slice$: mostra frame 47
- i així successivament, de forma cíclica.

Observeu que els frames estàn organitzats per files; el frame 0 ocupa en espai de textura el rectangle (0,0)→(1/8, 1/6), el frame 1 ocupa (1/8, 0)→(2/8,1/6), el frame 8 ocupa (0,1/6)→(1/8, 2/6), i així successivament. Les coordenades (s,t) que haureu d'utilitzar per accedir a la textura seran la suma de dos termes: un offset que depèn del frame (per exemple, (1/8, 1/6) pel frame 1), i un altre terme que depèn de texCoord escalat per (1/8, 1/6).

El color final del fragment l'heu de calcular com *color*.a * *color*, és a dir, el color de la mostra de la textura multiplicat per la seva component alpha. Resultats amb time=0, 0.25 i 0.5:



Identificadors (ús obligatori)

```
explosion.vert, explosion.frag
uniform sampler2D explosion;
```

13. Fire (fire.*) (1er control de laboratori, 2012-13 Q2)

Quan volem mostrar quelcom cremant, podem recórrer a diverses tècniques per a simular foc. Una de molt senzilla consisteix en mostrar en seqüència diverses textures capturades d'un foc real, animant-les en el temps.

Programa un **fragment shader** que mostri seqüencialment quatre textures foc-1.png, foc-2.png, foc-3.png i foc-4.png. Per a controlar la velocitat de l'animació, hauràs de disposar un **uniform float slice=0.1**, de forma que segons el valor de time es mostri una textura diferent:

- $0 \leq time < slice$: mostra foc-1
- $slice \leq time < 2 * slice$: mostra foc-2
- $2 * slice \leq time < 3 * slice$: mostra foc-3
- $3 * slice \leq time < 4 * slice$: mostra foc-4
- i així successivament, de forma cíclica.

Aquí veieu diversos frames del plane.obj cremant (time=0, 0.25, 0.55, 0.75).

Identificadors

```
fire.vert, fire.frag
uniform float slice=0.1;
uniform sampler2D sampler0;
uniform sampler2D sampler1;
uniform sampler2D sampler2;
uniform sampler2D sampler3;
```

14. Animate Vertices 1 (animate-vertices1.*)

Escriu un **vertex shader** que, abans de transformar el vèrtex a clip space, el mogui una certa distància *d(t)* en la direcció de la seva normal. Calculeu el valor de *d(t)* com una sinusoïdal amb una certa amplitud i freqüència:

```
uniform float amplitude=0.1;
uniform float freq = 1;      // expressada en Hz
```

Feu servir el uniform time que us proporciona el viewer.

Aquí teniu exemples de resultats:

15. Animate Vertices 2 (animate-vertices2.*)

Escriu un **vertex shader** que, abans de transformar el vèrtex a clip space, el mogui una certa distància *d(t)* en la direcció de la seva normal. Calculeu el valor de *d(t)* com una sinusoïdal amb una certa amplitud i freqüència (tots dos uniform float). Feu que la fase de la sinusoïdal depengui de 2*PI*s, on *s* és la **coordenada de textura del vèrtex**.

```
uniform float amplitude=0.1;
uniform float freq=1;   // expressada en Hz
```

Feu servir el uniform time.



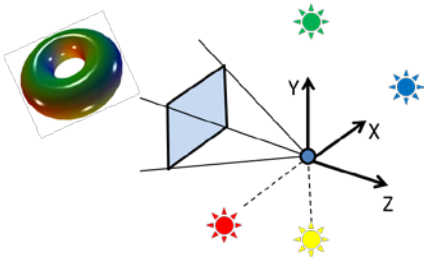

16. 4 Lights (4lights.*) (basat en 1er control de laboratori, 2014-15 Q1)

Al ZIP de l'enunciat us proporcionem un **vertex shader** i un **fragment shader** que calculen la il·luminació per fragment basant-se en el model de Phong.

En aquest exercici cal ignorar els materials i les llums que envia el viewer, ja que posarem els valors directament al codi del FS.

La versió que us passem del FS fa els càlculs d'il·luminació en **object space,** tenint en compte una única llum blanca situada a l'origen. Cal que modifiqueu la funció **main** del FS per tal que calculi la il·luminació del fragment tenint en compte **únicament 4 llums**, amb les posicions (que aquí us donem **en eye space**) i els colors següents:

Posició llum	**Color**
(0, 10, 0)	verd
(0, -10, 0)	groc
(10, 0, 0)	blau
(-10, 0, 0)	vermell



La funció **light** del FS no l'heu de modificar.

Addicionalment, heu d'afegir una variable **uniform bool rotate=true** que indiqui si cal aplicar una **rotació a les llums** respecte **l'eix Z de la càmera**, on l'angle de rotació serà directament θ_z***=time*** radians. Recordeu que la rotació d'un punt respecte l'eix Z es pot calcular multiplicant aquesta matriu pel punt:

$$\begin{matrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{matrix}$$

Identificadors (ús obligatori):

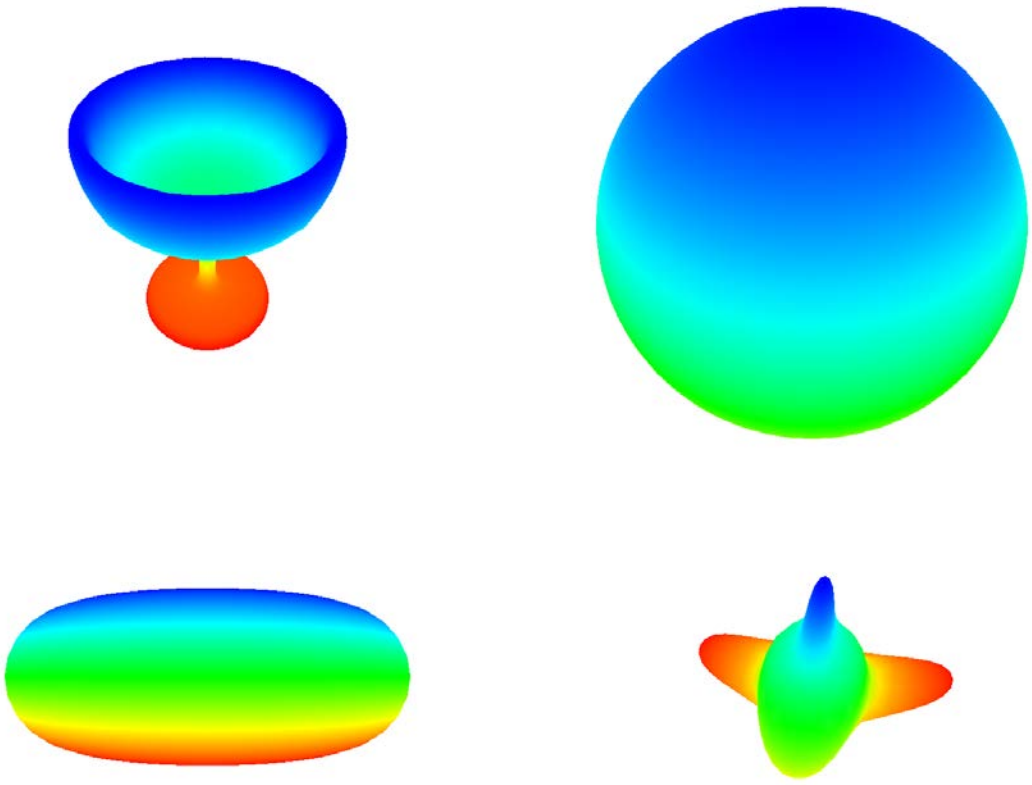
```
4lights.vert  4lights.frag
uniform float time;
uniform bool rotate;
```

17. Color gradient 1 (gradient-1.*)

Escriu un **vertex shader** que apliqui un gradient de color al model segons la seva coordenada Y en object space. Feu servir els uniforms amb la capsa englobant de l'escena per obtenir els valors extrems de la coordenada Y del model, minY, maxY.

El gradient de color estarà format per la interpolació d'aquests cinc colors: red, yellow, green, cian, blue. L'assignació s'haurà de fer de forma que els vèrtexs amb y=minY es pintin de vermell, i els vèrtexs amb y=maxY es pintin de blau.

Per la interpolació lineal entre colors consecutius del gradient, feu servir la funció **mix**. Una altra funció que us pot ser útil és **fract**, la qual retorna la part fraccionaria de l'argument.



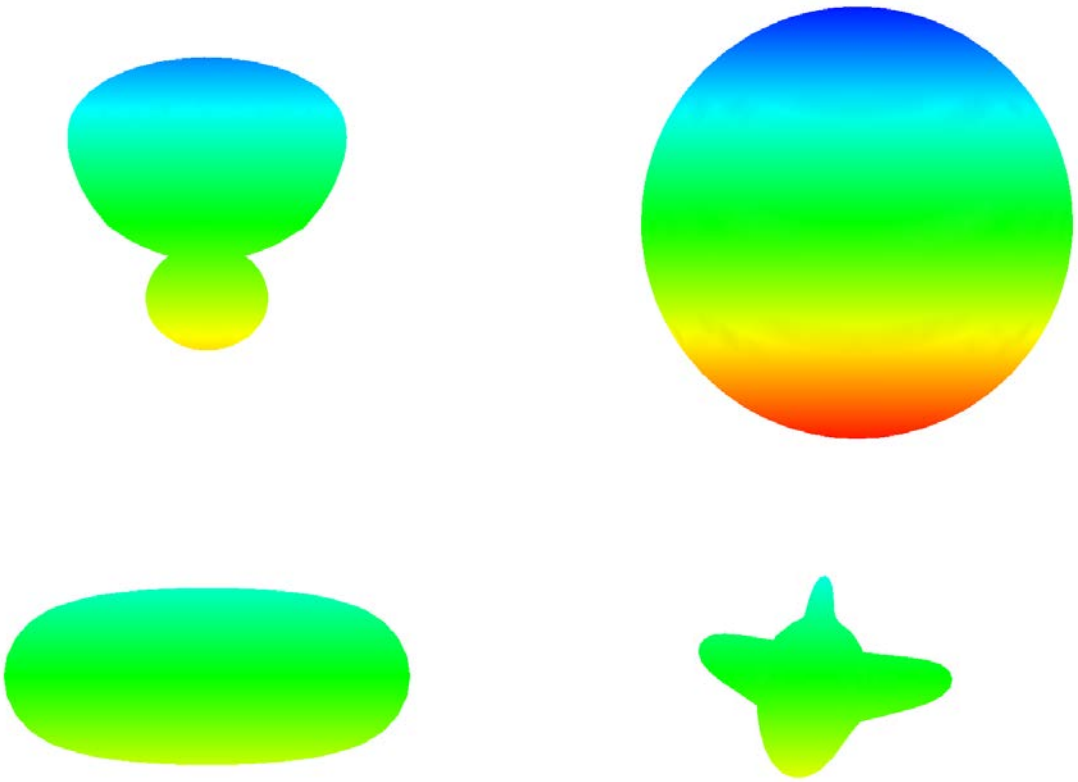
18. Color gradient 2 (gradient-2.*)

Escriu un **vertex shader** que apliqui un gradient de color al model segons la seva coordenada Y en coordenades normalitzades de dispositiu (és a dir, després de la divisió de perspectiva).

El gradient de color estarà format per la interpolació d'aquests cinc colors: red, yellow, green, cian, blue. L'assignació s'haurà de fer de forma que els vèrtexs amb y=-1.0 es pintin de vermell, i els vèrtexs amb y=1.0 es pintin de blau.

Per la interpolació lineal entre colors consecutius del gradient, feu servir la funció **mix**. Una altra funció que us pot ser útil és **fract**, la qual retorna la part fraccionaria de l'argument.

El resultat dependrà òbviament de la càmera.



19. Foldme (foldme.*) (1er control laboratori, curs 2012-13, Q1)

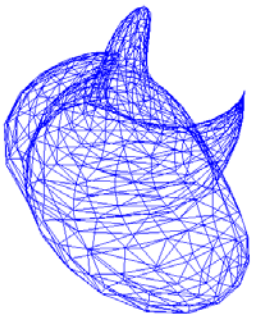
Escriu un **vertex shader** que, abans de passar el vèrtex a clip space, li apliqui una rotació $R_Y(\varphi)$ respecte l'eix Y. L'angle de rotació φ, en radians, l'heu de calcular com

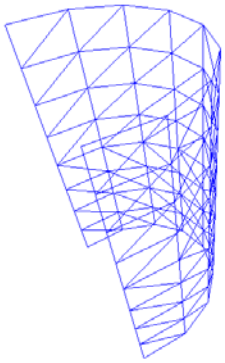
$$\varphi = -time \cdot s,$$

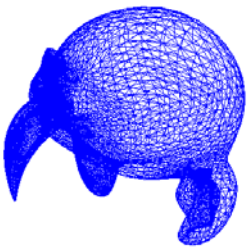
on *s* és la coordenada de textura *s* del vèrtex, i *time* és el uniform que ens proporciona el viewer amb el temps en segons. Observeu que, tal i com està definit l'angle φ, a mesura que avança el temps el model ha de girar en sentit horari respecte l'eix Y. Recordeu que la rotació d'un punt respecte l'eix Y es pot calcular multiplicant aquesta matriu pel punt:

$$\begin{pmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{pmatrix}$$

El VS ha d'assignar al vèrtex el **color blau**, sense cap càlcul d'il·luminació.







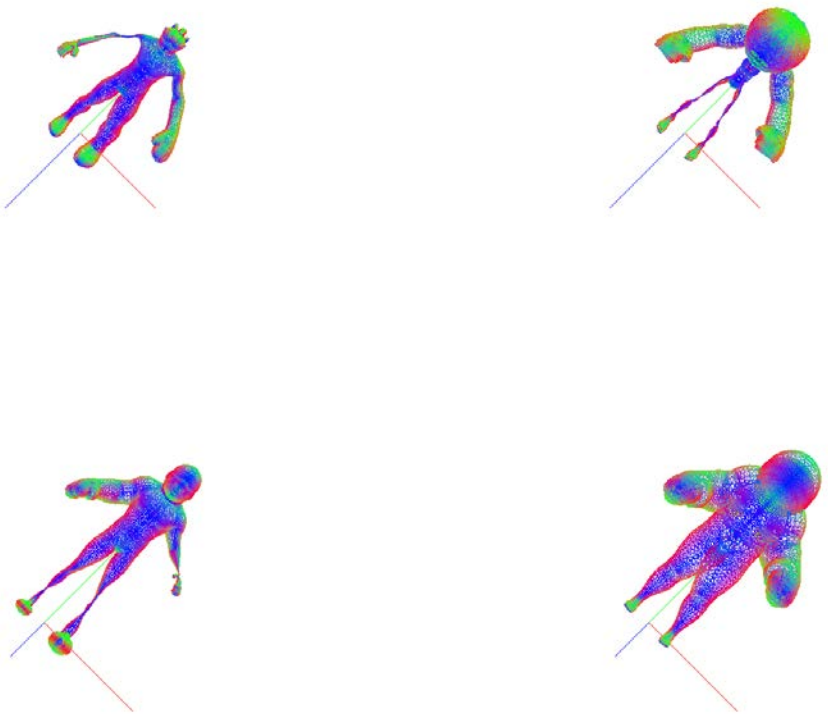
20. Oscillate (oscillate.*) (basat en 1er control laboratori, curs 2012-13, Q1; editat 13 oct 2015)

Escriu un vertex shader que pertorbi cada vèrtex del model (en object space) en la direcció de la seva normal, desplaçant-lo una distància *d* que variï sinusoidalment amb una amplitud *d* i període 2π segons. Calculeu la amplitud d com

$$d=(r/10)y$$

on r és la meitat de la mida de la diagonal de la capsa contenidora de l'escena (feu servir boundingBoxMin i boundingBoxMax), y és la coordenada *y* del vèrtex, agafada en *eye space* si el **uniform bool eyespace** és cert; altrament s'agafarà la component *y* en *object space*.

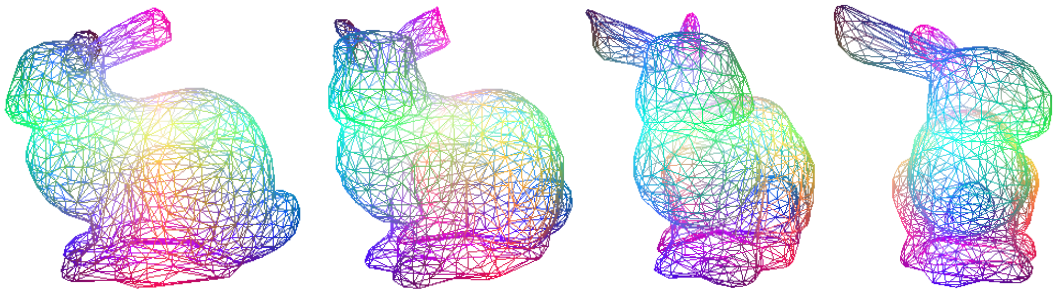
El VS haurà d'assignar com a color del vèrtex directament el **color.**



21. Auto-rotate (auto-rotate.*) (1er control laboratori, curs 2011-12, Q2)

Escriu un **vertex shader** que, abans de transformar cada vèrtex, li apliqui una rotació al voltant de l'eix Y. El shader rebrà un **uniform float speed** amb la velocitat de rotació angular (en rad/s). Feu servir la variable **uniform float time** per l'animació.

Aquí teniu els resultats (en wireframe) amb el bunny, amb speed = 0.5 rad/s:



time=0 time=1 time=2 time=3

Recordeu que la rotació d'un punt respecte l'eix Y es pot calcular multiplicant aquesta matriu pel punt:

$$\begin{pmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{pmatrix}$$

22. Bouncing (bouncing.*) (1er control de laboratori, 2014-15 Q1)

Escriu un **vèrtex shader** que faci que l'objecte reboti per la pantalla dintre dels límits definits per un

```
uniform float scale=8;
```

Caldrà que apliquis a l'objecte la translació **T** resultant de calcular:

$$\mathbf{T} = \text{scale} * (\mathbf{T0} + \mathbf{V} * \mathbf{t})$$

on la translació inicial és **T0 = (-1, -1, 0)** i la velocitat és **V = (2, 2,0)**.

Si a la fórmula anterior féssim servir t=time, després d'un temps l'objecte sortiria de la pantalla i no tornaria. Per a fer que reboti podem fer que el temps que apliquem a les equacions de moviment no surti d'un cert rang. Hauràs de fer servir la funció **triangleWave** que per un valor de x=0 torna 1, disminueix uniformement fins a 0 a x=1, torna a augmentar uniformement fins a 1 a x=2, i així successivament:

```
float triangleWave(float x) {
    return abs(mod(x, 2) - 1.0);
}
```

A més a més, volem usar diferents valors de temps per cada coordenada. Per tant, en el càlcul de la translació T, en compte de multiplicar directament la velocitat per time, fareu servir un producte (component a component) pel vector 3D

$$\mathbf{t} = (\text{triangleWave}(\text{time} / 1.618),\ \text{triangleWave}(\text{time}),\ 0)$$

Per a que tota la trajectòria de l'objecte sigui visible, un cop aplicada la translació caldrà que apliquis un escalat uniforme als vèrtexs, equivalent a dividir les tres coordenades de cada vèrtex pel **uniform float scale**.

També hauràs d'il·luminar cada vèrtex fent servir el color **vec4(0.3, 0.3, 0.9, 1.0)** i la component Z de la normal en **eye space**.

Identificadors (ús obligatori):

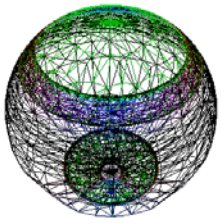
```
bouncing.vert
uniform float time;
uniform float scale;
```

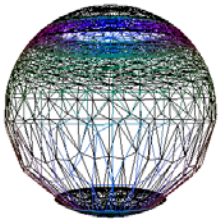
23. Spherize (spherize.*) (1er control laboratori, curs 2011-12, Q2)

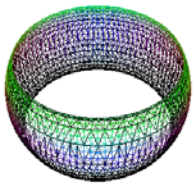
Escriu un **vertex shader** que, abans de transformar cada vèrtex, el projecti sobre una esfera de radi unitat centrada a l'origen del sistema de coordenades del model.

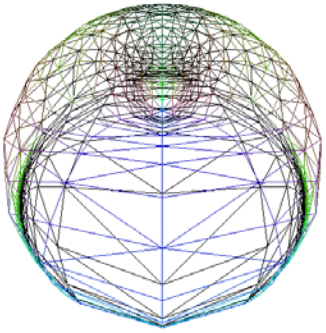
La projecció es farà en la direcció del vector que uneix l'origen del sistema de coordenades del model amb el vèrtex (òbviament en model space).

Aquí teniu els resultats esperats (en wireframe) amb diferents models (torus, boid):









24. Twist (twist.*) (1er control de laboratori, 2013-14 Q1)

Escriu un **vertex shader** que apliqui a cada vèrtex una **transformació de modelat** consistent en una rotació de θ_y radians respecte l'eix Y del model.

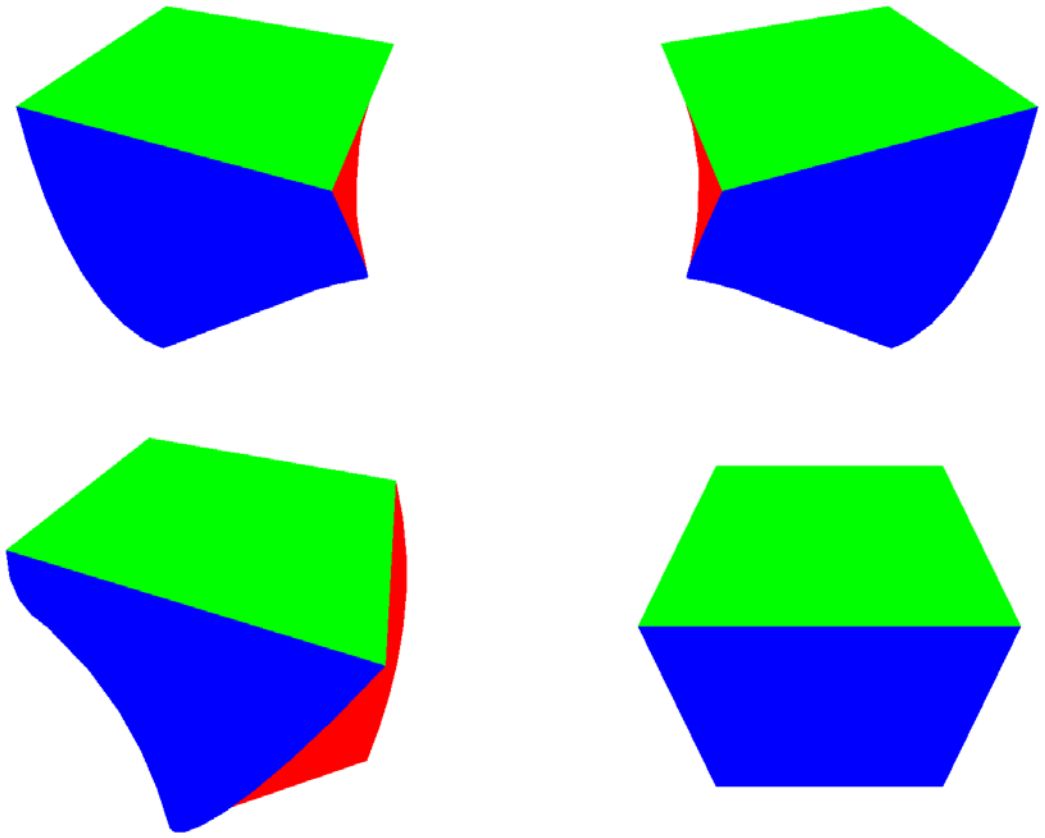
L'angle de rotació θ_y l'heu de calcular com

$$\theta_y = 0.4\, y \sin(t),$$

on y és la coordenada y del vèrtex en object space, i t és el temps en segons. Recordeu que la rotació d'un punt respecte l'eix Y es pot calcular multiplicant aquesta matriu pel punt:

$$\begin{pmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{pmatrix}$$

El VS també haurà de fer les tasques habituals (pas a clip space i propagació del color que li arriba, **sense il·luminació**).



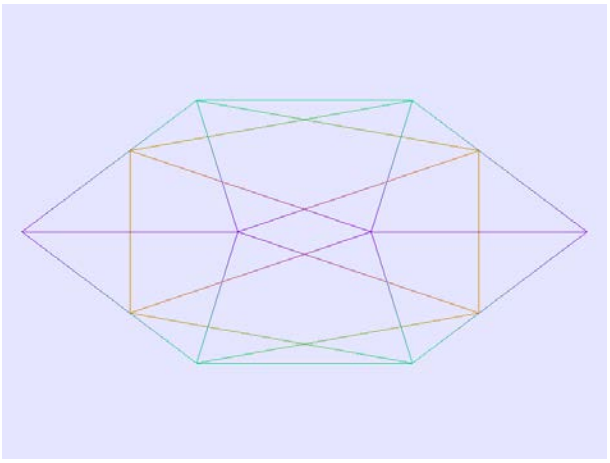
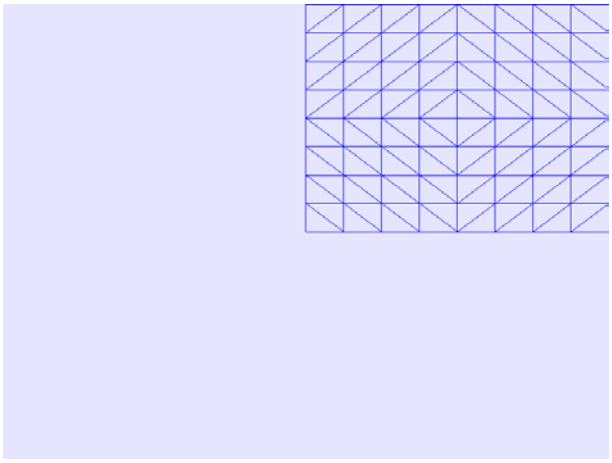
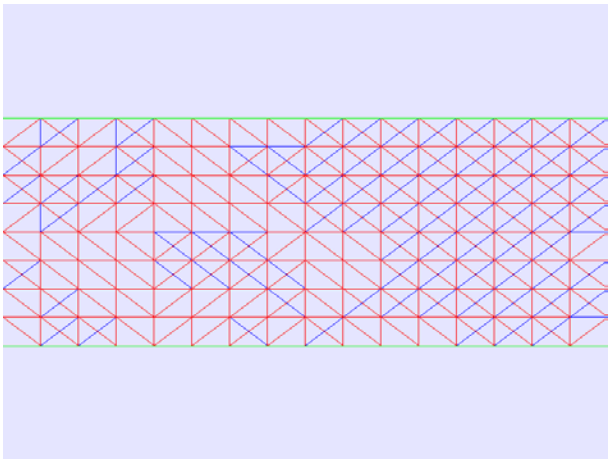
Aquests són alguns fotogrames del vídeo **twist.mp4:**

25. UV unfold (uv-unfold.*)

Escriu un **vertex shader** que mostri el model projectat sobre l'espai de textura (assumint que té coordenades de textura 2D definides). Atès que l'espai de textura no està acotat, l'usuari proporcionarà el rectangle de l'espai de textura que vol visualitzar, mitjançant els seus extrems Min, Max:

```
uniform vec2 Min=vec2(-1,-1);
uniform vec2 Max=vec2(1,1);
```

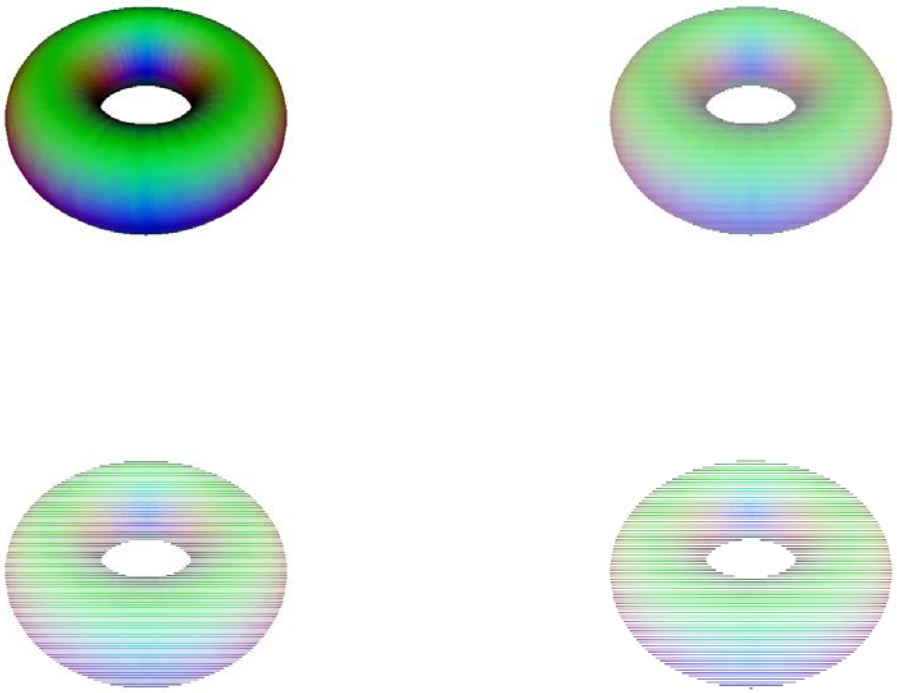
Amb independència de la càmera, el model projectat sobre el rectangle delimitat per Min, Max ha d'ocupar tot el viewport.


26. CRT Display (crt-display.*)

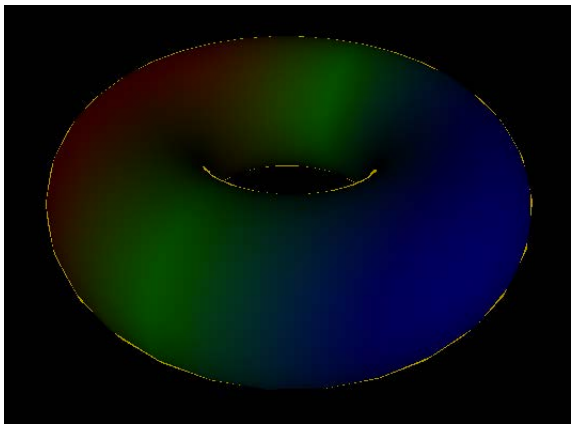
Escriu un **fragment shader** que simuli l'aparença de les imatges dels antics tubs CRT. Per aconseguir aquest efecte, caldrà eliminar (*discard*) tots els fragments d'algunes línies del viewport. En concret, caldrà que només sobrevisquin els fragments d'una de cada *n* línies, on *n* és un **uniform int** proporcionat per l'usuari.

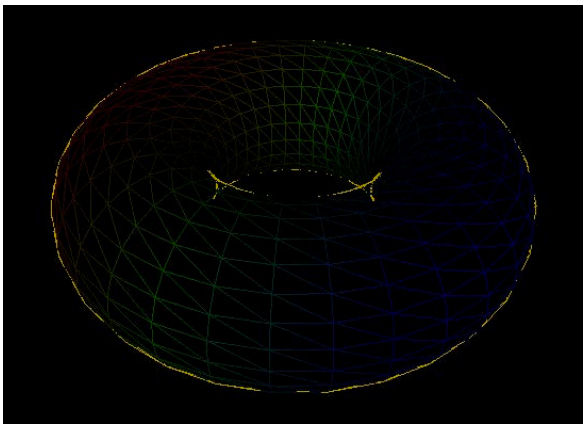
Observació: quan feu servir gl_FragCoord, tingueu en compte que per defecte les coordenades (x,y) en window space fan referència al centre del píxel. Per exemple, un fragment a la cantonada inferior esquerra de la finestra té coordenades (0.5, 0.5).



27. Profile (2on Control de laboratori, 2012-13 Q2; per aquest exercici no hem creat cap joc de proves)

Escriu un **fragment shader** que resalti el perfil del model amb un color groc fosc (rgb:0.7,0.6,0.0) com mostra la figura. Per a fer-ho, s'ajudarà d'un **vèrtex shader** que li proporcionarà les coordenades d'ull del fragment i la normal també en coordenades d'ull. A més, farà servir dos **uniform float**, **epsilon i light**. Per a cada fragment calcularem el producte escalar del vector (normalitzat) que l'uneix a la càmera i de la seva normal (també unitària), i si aquest producte és de magnitud inferior al llindar establert per l'**epsilon** (en un o altre sentit), donarem al fragment el color de silueta que apareix més amunt. Altrament, donarem el color normal del fragment, multiplicat pel valor de **light** i modulat per la z de la normal, com en "Basic lighting(2)".





Ambdues figures estan fetes amb $epsilon = 0.1$ i $light = 0.5$, amb càmeres arbitràriament rotades, però en la segona està activat "Wireframe rendering" i en la primera no.

Identificadors

```
profile.vert
profile.frag

uniform float epsilon;
uniform float light;
```

28. Calculant la normal al FS (calculant-la-normal.*)

Escriu un **vertex shader** i un **fragment shader** per calcular la il·luminació per fragment fent (n'hi ha prou amb el terme de Lambert), però sense fer servir **normal** (imagineu que l'aplicació no està enviant explícitament cap normal). Per tant, la normal l'haureu de calcular al fragment shader.

Pista: Per tal de calcular la normal al fragment shader, podeu fer servir les funcions de dFdx i dFdy, que aproximen les derivades parcials de l'argument proporcionat (l'especificació la teniu a sota). Penseu quin argument us cal per poder obtenir dos vectors tangents a la superfície. Amb el producte vectorial d'aquests dos vectors podeu obtenir un vector normal a la superfície.

Aquí teniu un exemple del resultat. Observeu que no ni ha suavitzat d'aresta, ja que tots els fragments d'un mateix polígon generen (aproximadament) la mateixa normal:

Especificació

dFdx, dFdy — return the partial derivative of an argument with respect to x or y

Declaration

genType dFdx(genType p);

genType dFdy(genType p);

Parameters

p - Specifies the expression of which to take the partial derivative.

Description

Available only in the fragment shader, dFdx and dFdy return the partial derivative of expression p in x and y, respectively. Deviatives are calculated using local differencing. It is assumed that the expression p is continuous and therefore, expressions evaluated via non-uniform control flow may be undefined.

29. Checkerboard 1 (checkerboard-1.*)

Escriu un **fragment shader** que apliqui al model una textura procedural que imiti un tauler d'escacs. Donat que la textura ha de ser procedural, no es permet l'ús de cap sampler.

Podeu assumir que el model té coordenades de textura.

Aquí teniu la imatge que s'espera:

La part de l'espai de textura entre el (0,0) i el (1,1) està ocupada pel tauler convencional de 8x8 cel·les. Podeu assumir que aquest tauler es repeteix indefinidament en totes dues direccions.

Penseu en una forma fàcil d'esbrinar si el color del fragment ha de ser negre o gris clar, dependent de les coordenades (s,t) de la textura.

30. Checkerboard 2 (checkerboard-2.*)

Escriu un **fragment shader** que apliqui al model una textura procedural que imiti un tauler d'escacs. Donat que la textura ha de ser procedural, no es permet l'ús de cap sampler.

Podeu assumir que el model té coordenades de textura.

La part de l'espai de textura entre el (0,0) i el (1,1) estarà ocupada pel tauler de *n*x*n* cel·les, on *n* és un uniform proporcionat per l'aplicació.

```
uniform float n = 8;
```

Podeu assumir que aquest tauler es repeteix indefinidament en totes dues direccions.

Aquí teniu la imatge que s'espera.

31. Checkerboard 3 (checkerboard-3.*)

Escriu un **fragment shader** que apliqui al model una textura procedural que imiti una graella regular (vegeu la figura). Donat que la textura ha de ser procedural, no es permet l'ús de cap sampler.

Podeu assumir que el model té coordenades de textura.

La part de l'espai de textura entre el (0,0) i el (1,1) estarà ocupada pel tauler de *n*x*n* cel·les, on *n* és un uniform proporcionat per l'aplicació.

```
uniform float n = 8;
```

Podeu assumir que aquest tauler es repeteix indefinidament en totes dues direccions.

Aquí teniu la imatge que s'espera.

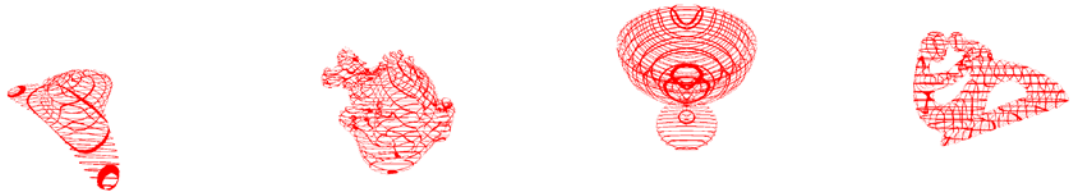
32. Senyera (senyera.*) (1er control laboratori, curs 2012-13, Q1)

Escriu un **fragment shader** que "texturi" el model amb el color d'una textura procedural 1D, tal com mostra la figura. El color del fragment dependrà de la coordenada de textura *s* del fragment. Sigui *f* la part fraccionària de la coordenada *s*, i sigui $a = 1/9$.

El FS haurà d'assignar al fragment el **color groc** pur si *f* està a qualsevol dels següents intervals: [0, a), [2a,3a), [4a,5a), [6a,7a], [8a,9a); altrament li assignarà el color **vermell** pur.

33. Checkerboard 4 (checkerboard-4.*)

Escriu un **fragment shader** similar al demanat als exercicis anteriors, però ara haureu de descartar (discard) els fragments que abans es pintaven de color gris clar. El resultat serà una mena de visualització en filferros del model, però que dependrà de la seva parametrització en comptes de com està dividit el model en cares.



34. CircularStripes (cstripes.*) (1er control de laboratori, 2012-13 Q2)

Escriu un **fragment shader** que "texturi" el model amb el color d'una textura procedural, tal com mostra la figura. El color del fragment dependrà de les coordenades (s,t) del fragment, i del valor d'un **uniform int nstripes** i d'un altre **uniform vec2 origin**. Més concretament, dependrà de la longitud del vector $(s,t) - origin$, de forma que si aquesta està a l'interval $[0, \frac{1}{nstripes})$, el fragment serà vermell; si està en $[\frac{1}{nstripes}, \frac{2}{nstripes})$ serà groc, i així successivament.

Identificadors

uniform int nstripes = 16;
uniform vec2 origin=vec2(0,0);

35. Hinomaru (hinomaru.*) (1er control de laboratori, 2013-14 Q2)

Escriviu un *fragment shader* que, de manera procedural, generi la bandera del Japó. Podeu suposar que aquest shader el provarem només amb l'objecte Plane, que té coordenades de textura del (0,0) al (1,1). Els punts i vectors que mencionem en aquest exercici fan referència a coordenades **en espai de textura**.

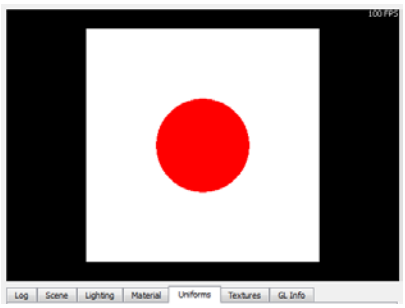


Figura 1: Resultat esperat.

Observeu que els fons és de color blanc, i el cercle central és de color vermell. Les coordenades de textura al centre són **C**=(0.5, 0.5). El color del fragment serà vermell en un **radi de 0.2** al voltant de C. És a dir, serà vermell quan el punt representat per les coordenades de textura (s,t) del fragment estigui a distància inferior o igual a 0.2 del punt (0.5, 0.5). Altrament serà blanc.

36. Hinomaru 2 (2on control de laboratori, 2013-14 Q2; per aquest exercici no hi ha test)

Escriu un *fragment shader* que, de manera procedural, generi una de les dues variants de la bandera del Japó. A les figures el veiem aplicat a l'objecte Plane, que té coordenades de textura del (0,0) al (1,1). Els punts i vectors que mencionem en aquest exercici fan referència a coordenades **en espai de textura**.

El FS usarà un **uniform bool classic** per indicar quina de les dues versions s'ha de generar.

Si **classic és cert**, caldrà generar la versió clàssica de la bandera del Japó (Figura 1).

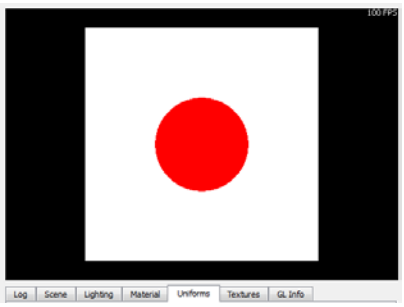


Figura 1: Resultat esperat quan **classic** és cert.

Observeu que els fons és de color blanc, i el cercle central és de color vermell. Les coordenades de textura al centre són **C**=(0.5, 0.5). El color del fragment serà vermell en un **radi de 0.2** al voltant de C. És a dir, serà vermell quan el punt representat per les coordenades de textura (s,t) del fragment estigui a distància inferior o igual a 0.2 del punt (0.5, 0.5). Altrament serà blanc.

Si **classic és fals,** caldrà generar la versió amb els 16 rajos que surten del sol (Figura 2).

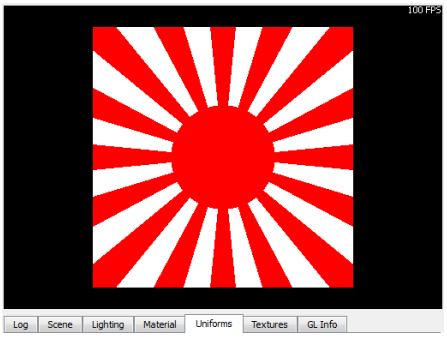


Figura 2: Resultat esperat quan **classic** és fals.

Pista: Hi ha 16 rajos vermells (més 16 "blancs") que cobreixen els 2π radians de la circumferència. Per tant, cada raig cobreix un angle $\varphi = \pi/16$ radians. Calculeu el vector **u** que va del centre (0.5, 0.5) al punt (s,t) del fragment. Amb **atan**(u.t, u.s) podeu calcular l'angle $\theta \in [-\pi, \pi]$ que defineix aquest vector amb l'eix s. El fragment serà vermell si **mod**$(\theta/\varphi + 0.5, 2) < 1$ o bé si és a dins del cercle de radi 0.2.

37. Disco-sphere (1er control de laboratori, 2012-13 Q2)

Volem simular de forma senzilla una esfera platejada similar a les que es poden veure en algunes discoteques:

Podeu veure el resultat esperat amb l'objecte Sphere i la textura **gold.png** al vídeo **disco-sphere.mp4.**

Observeu que l'esfera gira al voltant de l'eix Y a una velocitat de 0.1 rad/s. Per tant, el **vertex shader** aplicarà al vèrtex una rotació de 0.1**time* radians al voltant de l'eix Y (tot això en object space), i després escriurà gl_Position amb el nou vèrtex (en clip space). També caldrà que el **vertex shader** escrigui la posició del vèrtex en eye space en una variable varying que farà servir el fragment shader.

El **fragment shader** calcularà el color del fragment utilitzant una textura

uniform sampler2D sampler;

de la següent manera. Primer calcularà una aproximació de la normal **n** (en eye space) fent servir les funcions dFdx, dFdy, de forma anàloga a l'exercici *calculant la normal al fragment shader*. Això és necessari perquè el viewer proporciona normals per vèrtex, però en aquest shader ens interessa una normal que sigui aproximadament igual per tots els fragments d'una mateixa cara.

Després farà servir les components ($\mathbf{n}_x$, $\mathbf{n}_y$) de la normal per calcular el color del fragment com el producte

$$T * \mathbf{n}_z$$

on T és el color del texel al punt (s,t) = ($\mathbf{n}_x$, $\mathbf{n}_y$) de la textura.

Identificadors

```
disco-sphere.vert
disco-sphere.frag
uniform float time;
uniform sampler2D sampler;
```

38. Uncover (uncover.*) (1er control laboratori, curs 2012-13, Q1)

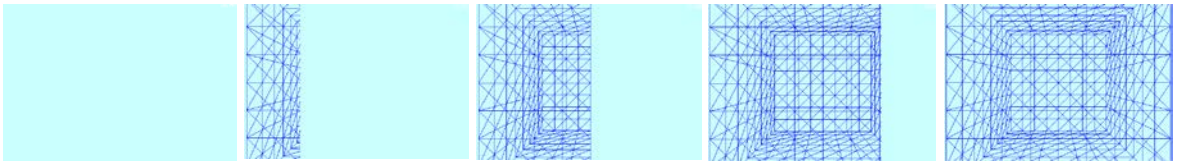
Una transició típica als programes d'edició de video/presentacions és descobrir el contingut de forma progressiva, per exemple d'esquerra a dreta. Escriu un **vertex shader** i un **fragment shader** que conjuntament simulin aquesta transició. Per aconseguir aquest efecte, cal que el FS comenci descartant tots els fragments, i els vagi mostrant a mesura que passi el temps, progressivament d'esquerra a dreta. El temps de l'animació serà de dos segons. Per tant:

- Al començament (time=0), el FS descartarà tots els fragments
- Al cap d'un segon (time=1), serà visible només la meitat esquerra del viewport,
- A partir dels dos segons (time>=2), no es descartarà cap fragment.

El FS assignarà als fragments no descartats el **color blau** (no cal cap càlcul d'il·luminació).

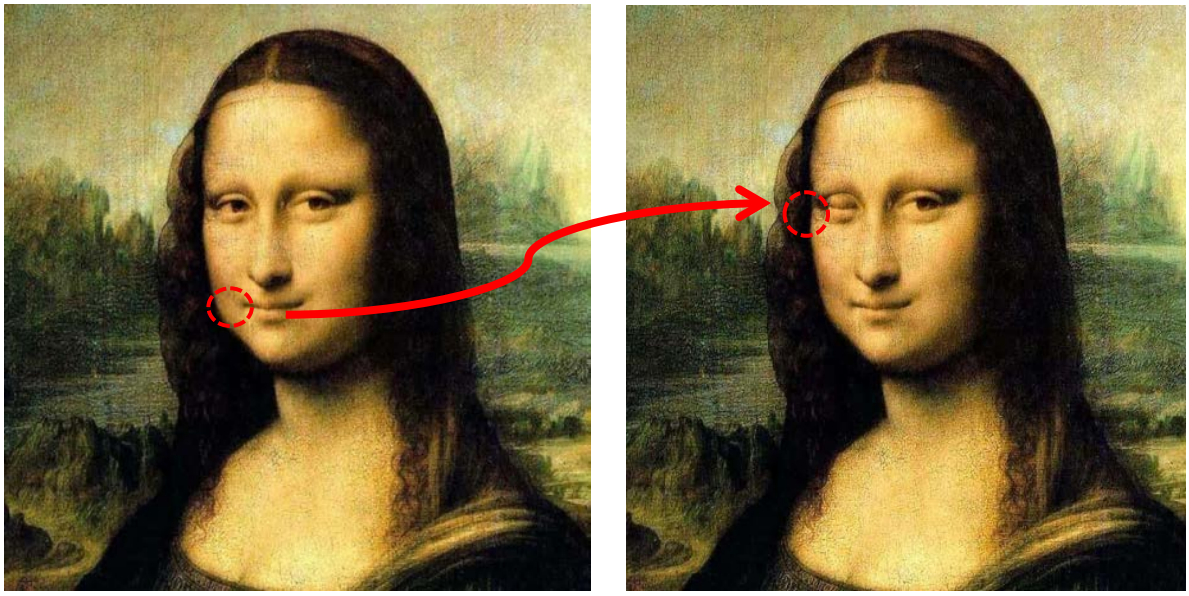
Donat que no podeu accedir a la mida del viewport, us recomanem que la decisió de descartar o no un fragment es basi en la coordenada x del fragment en NDC.

Aquí teniu els resultats esperats amb el cub i time variant entre 0 i 2.



39. gioconda.frag (2on control de laboratori, 2014-15 Q1; per aquest exercici no hi ha test)

Al ZIP de l'enunciat trobareu una imatge de La Gioconda (**gioconda.png**). Escriu un FS que, quan s'utilitza amb **l'objecte Plane,** simuli que La Gioconda ens pica el seu ull esquerre cíclicament, com teniu al vídeo **gioconda.mp4**:



Una forma fàcil d'aconseguir aquest efecte és que els texels al **voltant de l'ull** agafin el color de la textura **al voltant dels llavis**. Sabem que, en **espai normalitzat de textura**:

- L'ull està centrat aproximadament al punt (s,t) = **(0.393, 0.652)**
- La boca està centrada al punt (s,t) = **(0.45, 0.48)**, per tant a un **offset (0.057, -0.172)** respecte l'ull.
- La zona circular que volem copiar té un **radi de 0.025**

Si **fract(time) <= 0.5**, el FS aplicarà la textura amb les coordenades de textura originals (ulls oberts).

Si **fract(time)>0.5**, el FS aplicarà la textura amb les coordenades de textura originals, tret dels fragments amb coordenades (s,t) dins del cercle de l'ull, pels quals caldrà accedir a la textura afegint l'offset que hem mencionat abans.

No cal cap mena d'il·luminació.

Identificadors (ús obligatori):

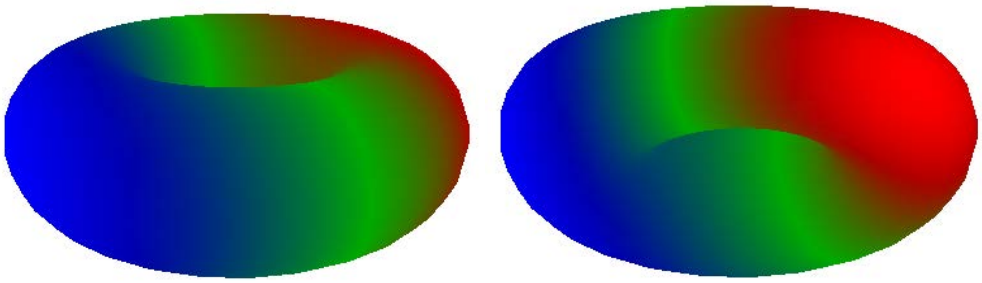
```
gioconda.frag
uniform float time;
uniform sampler2D sampler;
```

40. Reverse Z 1 (reverse-z1.*)

La funció d'OpenGL glDepthFunc permet triar el tipus de depth test a aplicar per OpenGL. Per defecte, el test és GL_LESS, és a dir, un fragment passa el test si la seva Z (en window space) és més petita que la Z emmagatzemada al depth buffer.

Escriu un **vertex shader** que modifiqui la Z dels vèrtexs per tal d'aconseguir el mateix efecte que tindria cridar glDepthFunc amb GL_GREATER des de l'aplicació. El resultat és que s'invertirà la visibilitat de les cares, sent visibles les cares més llunyanes a l'observador.

Aquí teniu un exemple amb el model del torus, sense invertir i invertint la Z:

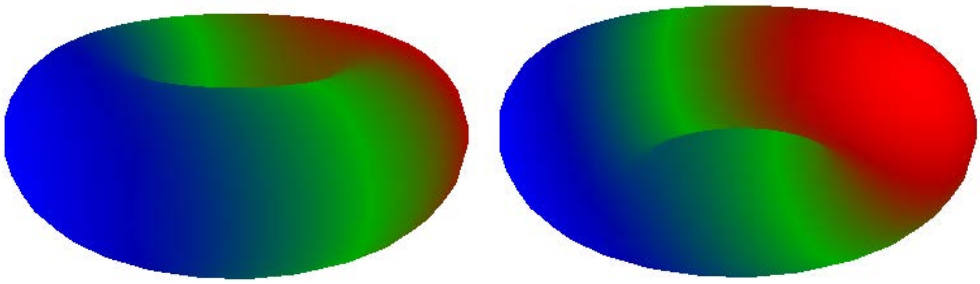


41. Reverse Z 2 (reverse-z2.*)

La funció d'OpenGL glDepthFunc permet triar el tipus de depth test a aplicar per OpenGL. Per defecte, el test és GL_LESS, és a dir, un fragment passa el test si la seva Z (en window space) és més petita que la Z emmagatzemada al depth buffer.

Escriu un **fragment shader** que modifiqui la Z dels fragments per tal d'aconseguir el mateix efecte que tindria cridar glDepthFunc amb GL_GREATER des de l'aplicació. El vertex shader farà les tasques per defecte. El resultat és que s'invertirà la visibilitat de les cares, sent visibles les cares més llunyanes a l'observador.

Aquí teniu un exemple amb el model del torus (amb shader per defecte, i amb el shader que es demana):



42. Projective texture mapping (2on control laboratori, curs 2011-12, Q2)

Escriu un **vertex shader** i un **fragment shader** que texturin el model usant projective texture mapping.

El **vertex shader** haurà d'escriure en `vtexCoord` les coordenades de textura homogènies del vèrtex calculades usant les matrius de projective texture mapping, fent que les matrius MODELVIEW i PROJECTION del projector siguin les mateixes que les de la càmera, sense oblidar l'escalat i translació habituals a projective texture mapping.

El **fragment shader** calcularà el color del fragment multiplicant la component Z de la normal en eye space (com a basic lighting) pel color de la textura `uniform sampler2D sampler`.

Aquí teniu el resultat esperat amb el torus i l'esfera (textura rock.png):

43. Parallax hallucination (1er control de laboratori, 2013-14 Q2)

A l'exercici *Animate Texture* vàreu provar l'efecte de modificar les coordenades de textura (s,t). En aquell exercici, les coordenades pertorbades es calculaven com **(s,t)+offset**, on **offset** era un **vec2** amb components que depenien del temps. Ara us demanem quelcom similar, però canviant la manera amb la que calculeu l'offset.

Siguin (s,t) les coordenades de textura que rep el fragment shader. El fragment shader començarà consultant el color **c** de la textura en aquest punt (s,t). Sigui **m** el **màxim** de les components rgb del color **c**. Aquest **m** entre [0,1] ens aproxima la lluminositat del color del texel. Això us permetrà definir un **vec2 u** amb components (m,m). Observeu que el mòdul de **u** serà gran pels colors clars i petit pels colors més foscos. Al vector **u** li aplicareu una rotació de $\theta=2\pi t$ radians, on t és el temps en segons. Recordeu que la matriu de rotació 2D és

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

El vector offset serà simplement **(a/100.0)u**, on **a** és un **uniform float**, i **u** és el vector que heu rotat abans. El color final del fragment serà el color de la textura (**uniform sampler2D map**) al punt **(s,t)+offset**.

El resultat esperat (una mica subtil) amb l'objecte Plane i **a=0.5** el teniu a **hallu-gray-stones.mp4** i **hallu-color-stones.mp4**, fent servir les textures amb el mateix nom (Figura 3). Observareu que els colors més clars descriuen una circumferència més àmplia que els colors foscos, creant un efecte de parallax que el sistema visual interpreta com relleu.

Figura 3: Textures hallu-gray-stones.png i hallu-color-stones.png

Identificadors

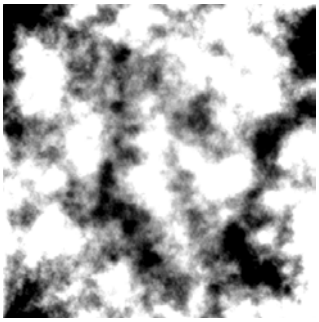
```
hallu.frag
uniform sampler2D map;
uniform float time;
uniform float a;
```

44. Texture splatting (splatting.*) (2on control laboratori, curs 2011-12, Q2)

Una tècnica que es fa servir sovint per a texturar terrenys consisteix a combinar diferents textures amb l'ajut de soroll fractal, de forma que el valor del soroll determina el pes de cada textura en el fragment.

La figura mostra tres textures (rock, grass, noise) i el resultat final desitjat (amb l'objecte *Plane*):






Observeu que als fragments on el soroll és proper a 0 es mostra el color de la roca, mentre que quan el soroll és proper a 1 es mostra el color de la vegetació.

Escriu un **fragment shader** que, donades tres textures com les anteriors, calculi el color del fragment com a interpolació lineal entre el color de la textura de roca i el color de la textura de vegetació, on el pes de la interpolació lineal depèn del valor de funció de soroll (agafeu la component r de la textura **noise.png** com a valor del soroll).

Totes tres textures s'accedeixen amb les coordenades de textura que envia el viewer. No feu servir cap tipus d'il·luminació.

```
uniform sampler2D noise0;
uniform sampler2D rock1;
uniform sampler2D grass2;
```

45. Marble (1er control de laboratori, 2013-14 Q1)

Escriu un **vèrtex shader** i un **fragment shader** que simulin l'aparença del marbre. El **VS** haurà de proporcionar al FS la informació que necessita (normal i posició en object space). El **FS** calcularà el color de cada fragment de la següent manera. Primer, caldrà generar unes coordenades de textura (s,t) utilitzant la tècnica basada en dos plans S i T:

$$s = a_s x + b_s y + c_s z + d_s w \qquad t = a_t x + b_t y + c_t z + d_t w$$

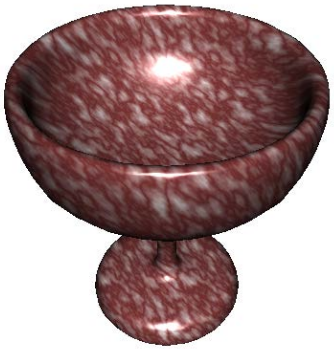
on [x y z w] són les coordenades del punt en **object space**, i els plans S i T els heu definir com:

$$(a_s\ b_s\ c_s\ d_s) = 0.3*(0, 1, -1, 0) \qquad (a_t\ b_t\ c_t\ d_t) = 0.3*(-2, -1, 1, 0)$$

Un cop calculades les coordenades de textura (s,t) al FS, calculareu un valor *v* en [0,1] prenent una mostra del canal vermell (r) de la textura **noise_smooth.png** al punt (s,t). El color difós de l'objecte el calculareu com un gradient de color que estarà format per la interpolació d'aquests tres colors: *white*, *redish* = (0.5, 0.2, 0.2, 1.0) i *white*, utilitzant el valor *v*. El càlcul s'ha de fer de forma que *v*=0 doni *white*, *v*=0.5 doni *redish*, i *v*=1.0 doni un altre cop *white*. Després de calcular aquest color difós, el color final del fragment serà el que retorni la funció **shading**, que haureu de cridar amb N essent un vector unitari en **eye space**, Pos essent la posició en **eye space**, i *diffuse* essent el color difós calculat prèviament.

```
vec4 shading(vec3 N, vec3 Pos, vec4 diffuse) {
    vec3 lightPos = vec3(0.0,0.0,2.0);
    vec3 L = normalize( lightPos - Pos );
    vec3 V = normalize( -Pos);
    vec3 R = reflect(-L,N);
    float NdotL = max( 0.0, dot( N,L ) );
    float RdotV = max( 0.0, dot( R,V ) );
    float Ispec = pow( RdotV, 20.0 );
    return diffuse * NdotL + Ispec;
}
```

Aquí teniu exemples dels resultats esperats amb la textura **noise_smooth.png**:



Identificadors (ús obligatori)

```
marble.vert   marble.frag
uniform sampler2D noise;
```

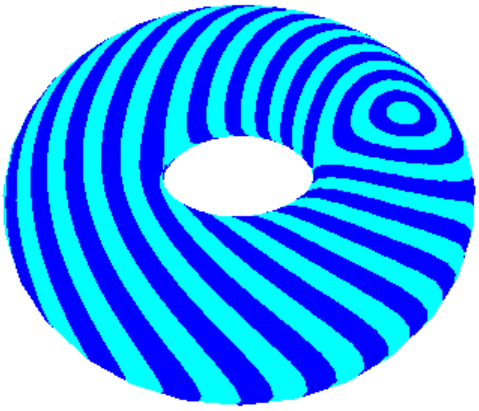
46. solidtex.vert, solidtex.frag (1er control de laboratori, 2014-15 Q1)

Volem texturar un objecte amb vetes però després de provar diferents textures, estem desanimats, perquè no hi ha forma de fer que la textura de les vetes sigui consistent en passar d'una cara a la cara veïna, i tot al voltant de l'objecte. Però un amic ens proposa una solució prometedora: definir una textura sintètica (com has fet als Checkerboard*, per exemple), però amb coordenades de textura tridimensionals, de manera que a cada punt li correspongui un color calculat a partir de les seves **coordenades en world space** (podeu assumir que no hi ha transformació de modelat).

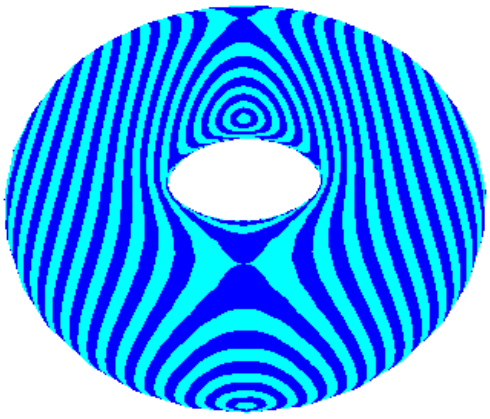
Anem a implementar aquesta idea, on la textura (sintètica) estarà formada per bandes alternades de colors **blau i cian**. Ho farem amb un VS i un FS, de manera que el FS decideixi el color de cada fragment a partir de les seves coordenades en world space, i dels valors dels següents **uniforms**: **origin** (que contindrà les coordenades d'un punt en world space), **axis** (que contindrà les components d'un vector, no necessàriament unitari, en world space), i **slice** que serà un float que determinarà el gruix de les vetes.

Per cada fragment, calcularem (en world space) la distància **d** del punt a la recta que passa per origin i té la direcció de l'eix **axis**. Si aquesta distància està en [0, slice), assignarem el color cian; si està en [slice, 2*slice) assignarem el color blau; si està en [2*slice, 3*slice), altre cop cian; i així successivament.

Aquí teniu el resultat amb el torus:



origin =(1,0,0), axis=(0,1,0), slice=0.1



origin=(0,0,0), axis=(0,0,1), slice=0.05

Identificadors (ús obligatori):

```
solidtex.vert, solidtex.frag
uniform vec3 origin;
uniform vec3 axis;
uniform float slice;
```

47. inking.frag (2on control de laboratori, 2014-15 Q1)

Escriu un **fragment shader** que detecti els canvis de color en la textura aplicada al model i que si aquests superen un cert llindar canviï el color del fragment a negre. L'efecte (amb els paràmetres correctes) hauria de ser semblant a l'entintat que es fa en els còmics.

Per a poder calcular si el color varia molt al voltant del fragment, heu de prendre quatre mostres de la textura d'entrada que es trobin a l'esquerra, a la dreta, a dalt i a baix de les coordenades de textura. En concret si el fragment té coordenades de textura (s, t) caldrà prendre mostres a les coordenades:

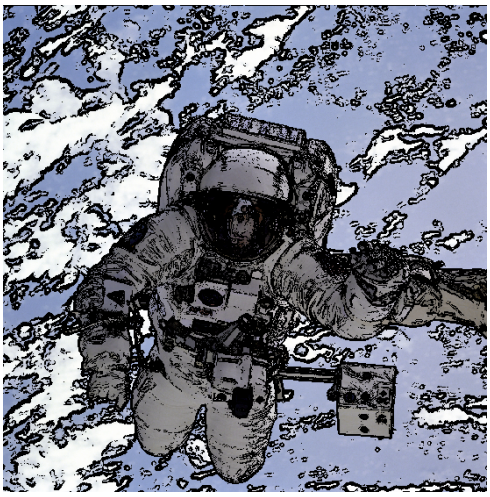
left = (s, t) + edgeSize * (-1, 0) / textureSize
right = (s, t) + edgeSize * (1, 0) / textureSize
bottom = (s, t) + edgeSize * (0, -1) / textureSize
top = (s, t) + edgeSize * (0, 1) / textureSize

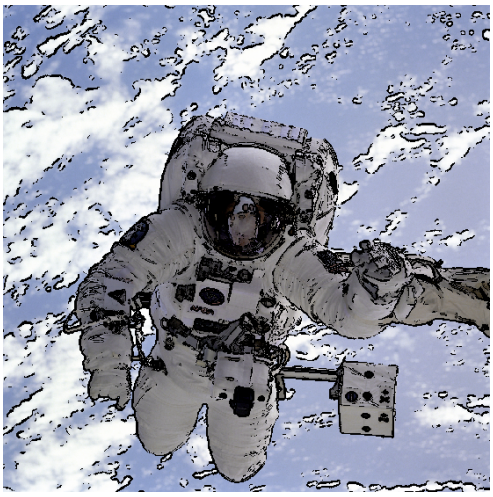
on **uniform int textureSize** indica la mida de la textura en texels i el **uniform int edgeSize** la mida en píxels del canvi de color que volem detectar. Amb aquestes coordenades (s,t) haureu de consultar la textura **tex** i mesurar les diferències de color en direcció X i Y:

GX = ||tex(right) - tex(left)||
GY = ||tex(top) - tex(bottom)||

Si feu servir la longitud del vector 2D que té per components GX i GY, el valor resultant és un bon estimador del canvi de color al punt (s, t). Si aquest resultat és major que un cert llindar (**uniform float threshold**) llavors canvieu el color del fragment a negre. Si no, assigneu-li al fragment el color de la textura a les coordenades (s, t).

Teniu disponible una textura **astronaut.png** per provar el shader. Aquí teniu els resultat amb **textureSize = 1024**, **edgeSize = 2**, i **thresholds 0.1 i 0.2:**





Identificadors (ús obligatori):

```
inking.frag
uniform int textureSize;
uniform int edgeSize;
uniform float threshold;
```

48. Sphere Mapping (2on control laboratori, curs 2011-12, Q2)

Escriviu un **vertex shader** i un **fragment shader** que simulin que l'objecte és un mirall especular usant la tècnica de sphere mapping.

Feu que el càlcul de les coordenades de textura es faci al **fragment shader**, a partir de la posició i la normal interpolades al fragment.

Els shaders rebran una variable **uniform bool worldSpace** que indicarà si els càlculs s'han de fer amb la posició i la normal en *world space* (suposem que no hi ha transformació de modelat) o en *eye space*.

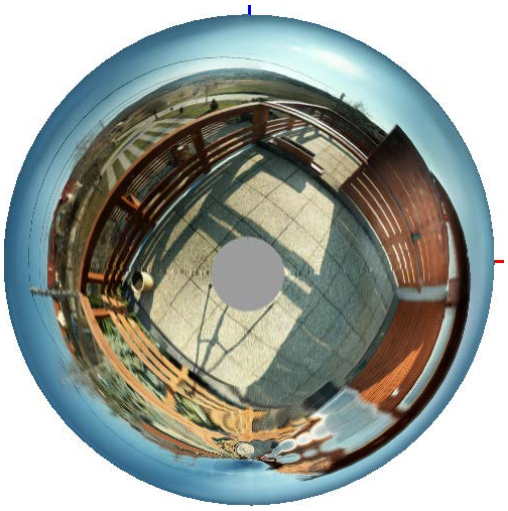
Aquí teniu un exemple dels resultats esperats amb l'esfera (vista frontalment i des de sota) amb la textura **spheremap.png** (càlculs en *eye space*):

Vista frontal, eye space

Vista des de sota, eye space

I els mateixos resultats però ara amb càlculs en *world space*:

Vista frontal, world space



Vista des de sota, world space

Podeu fer servir aquesta funció per obtenir una mostra del sphere map, donat el vector reflectit R:

```
vec4 sampleSphereMap(sampler2D sampler, vec3 R)
{
   float z = sqrt((R.z+1.0)/2.0);
   vec2 st = vec2((R.x/(2.0*z)+1.0)/2.0, (R.y/(2.0*z)+1.0)/2.0);
   st.y = -st.y;
   return texture(sampler, st);
}
```

49. Skymap (2on control laboratori, curs 2012-13, Q1)

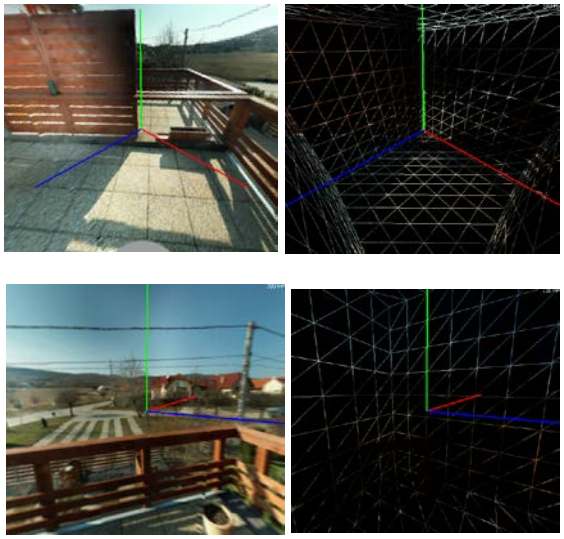
Escriu un **vertex shader** i un **fragment shader** que permetin simular un entorn (representat amb un spheremap) amb geometria senzilla (com ara un cub).

El VS simplement haurà d'escriure gl_Position de la forma habitual i fer arribar al FS la posició del vèrtex en *object space*.

El FS haurà de calcular el vector unitari V en la direcció que va de la posició del fragment a la posició de l'observador (tot en *object space*). El color final del fragment serà simplement el color del texel del sphere map (`uniform sampler2D spheremap`) corresponent al vector V. Per aquest darrer pas podeu utilitzar aquesta funció que, donat un spheremap i un vector unitari V, retorna el color en la direcció donada per V:

```
vec4 sampleSphereMap(sampler2D sampler, vec3 V)
{
  float z = sqrt((V.z+1.0)/2.0);
  vec2 st = vec2((V.x/(2.0*z)+1.0)/2.0, (V.y/(2.0*z)+1.0)/2.0);
  return texture(sampler, st);
}
```

Aquí teniu alguns exemples quan fiquem la càmera dins de l'objecte Cube (amb un fov de 90 graus):



50. Glossy (2on control de laboratori, 2013-14 Q1)

Al fitxer adjunt us proporcionem una implementació (VS+FS) de *sphere mapping*. Volem aconseguir l'efecte de la figura, fent servir un paràmetre enter *r* que determini la quantitat de mostres del sphere map que es tenen en compte per calcular el color de cada fragment (boid.obj, *r*=0, 5, 10, 20, 40):

El **FS** que us proporcionem té una funció `sampleTexture` que pren una mostra de la textura (feu servir **glossy.png**) al punt de coordenades de textura (s,t):

```
vec4 sampleTexture(sampler2D sampler, vec2 st, int r)
{
   return texture(sampler, st);
}
```

El que us demanem és que completeu aquesta funció per tal que retorni el promig dels colors d'una regió quadrada de la textura centrada en el punt *st*. En concret, si C(s,t) el que retorna la funció texture2D pel punt (s,t), i (W,H)=(512,512) és la mida de la imatge, la funció `sampleTexture` ha de retornar

$$\frac{1}{(2r+1)^2}\sum_{i=-r}^{r}\sum_{j=-r}^{r} C\left(s+\frac{i}{W}, t+\frac{j}{H}\right)$$

Observeu que el nombre de crides a texture2D serà $(2r+1)^2$, i per tant és d'esperar que el frame rate baixi considerablement per valors grans de *r*. **Hi ha un cert risc de penjar el PC amb valors grans de r.** Hem afegit un min(40,r) al codi per tal de limitar aquest risc, però mireu de fer les proves amb valors petits.

Aquí teniu un altre exemple del resultat esperat (glass.obj, r=40):

Identificadors:

```
glossy.vert   glossy.frag
uniform int r;
uniform sampler2D glossy;
```