

---

## Basic lighting (1)

---

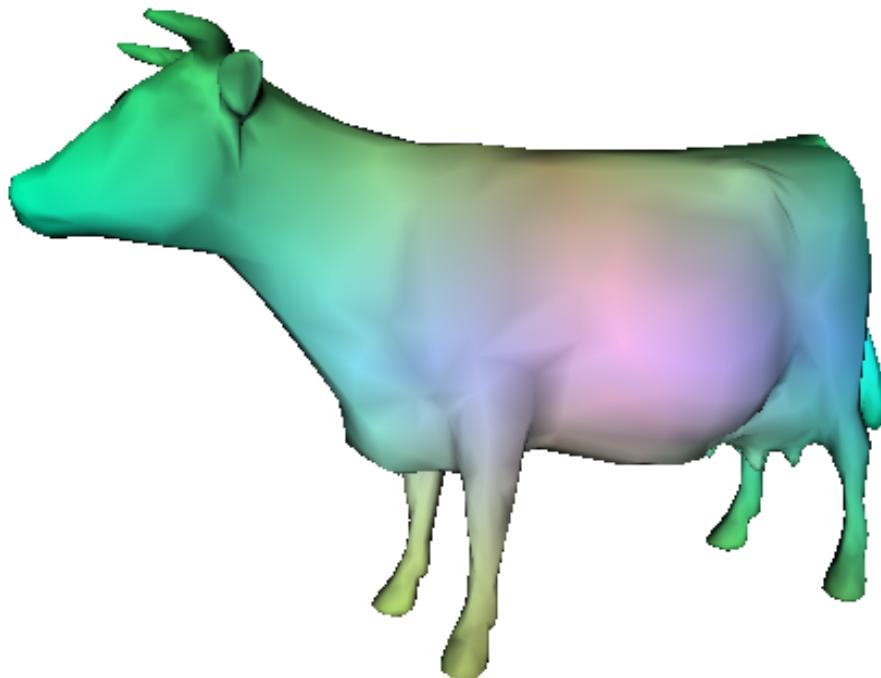
Durant el desenvolupament de shaders sovint ens pot interessar aconseguir un cert efecte d'il·luminació de la forma més senzilla possible.

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 **vertex shader** 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 gl\_NormalMatrix.

Aquí tens un exemple del resultat esperat amb el model de la vaca:



---

## Basic lighting (2)

---

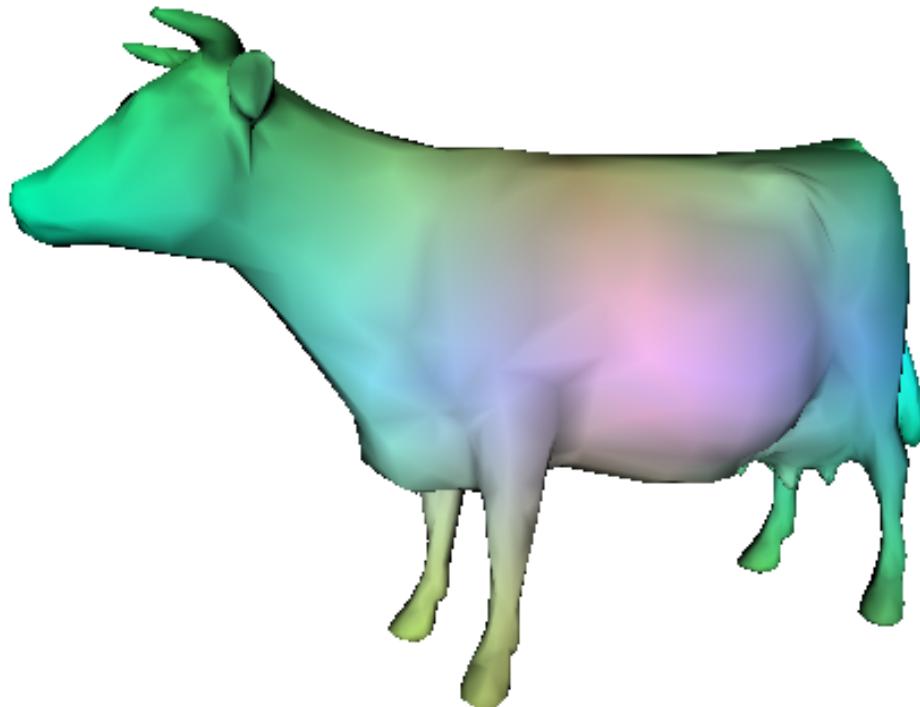
Durant el desenvolupament de shaders sovint ens pot interessar aconseguir un cert efecte d'il·luminació de la forma més senzilla possible.

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 **vertex shader** i un **fragment shader** que calculi la il·luminació per fragment fent servir aquesta tècnica.

Caldrà que declareu una variable varying a tots dos shaders per tal que el fragment shader pugui consultar la normal.

Aquí tens un exemple del resultat esperat amb el model de la vaca:



## Lighting (1)

Escriu un **vertex shader** que calculi la il·luminació per vèrtex de forma anàloga a com ho fa OpenGL, fent servir les propietats del material i de la font de llum (només cal que considereu la primera font de llum GL\_LIGHT0).

Suposeu que les llums no són SPOT. Per tant la fórmula a aplicar serà la de Blinn-Phong:

$$K_e + K_a(M_a + I_a) + K_d I_d(N \cdot L) + K_s I_s (N \cdot H)^s$$

on

$K_e$  = emissió del material

$K_a$ ,  $K_d$ ,  $K_s$  = reflectivitat ambient, difosa i especular del material

$s$  = shininess del material

$M_a$  = llum ambient del model (gl\_LightModel.ambient)

$I_a$ ,  $I_d$ ,  $I_s$  = propietats ambient, difosa i especular de la llum GL\_LIGHT0.

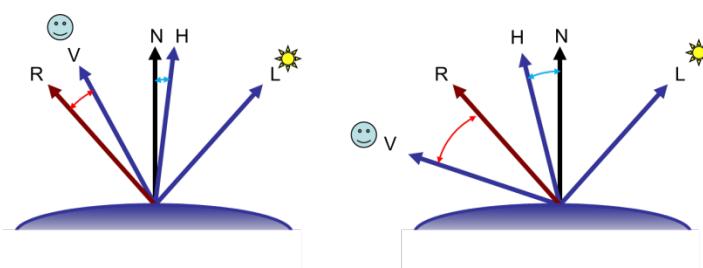
$N$  = vector normal unitari (eye space)

$L$  = 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 calcula  $H$  com si  $V$  fos  $(0, 0, 1)$ , és a dir, com si l'observador estigués a l'infinít en direcció de les Z.

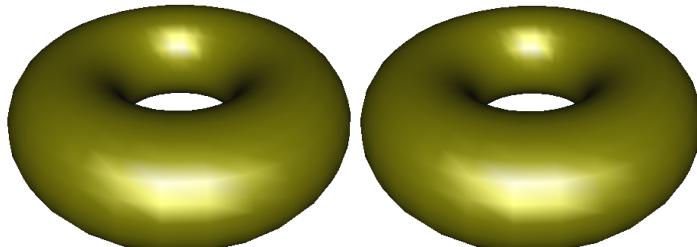
Aquest comportament per defecte es pot canviar modificant amb glLightModelfv el paràmetre GL\_LIGHT\_MODEL\_LOCAL\_VIEWER.

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 escalaris  $N \cdot L$  o  $N \cdot H$  són negatius. Per tant haureu de fer servir  $\max(0.0, \text{dot}(N, L))$  i  $\max(0.0, \text{dot}(N, H))$ .

Si ho implementeu correctament, el resultat ha de ser indistingible de la il·luminació que calcula OpenGL. Aquí teniu un exemple, amb i sense shader:



## Lighting (2)

Escriu un **vertex shader** que calculi la il·luminació per vèrtex amb el model de Phong (només cal que considereu la primera font de llum GL\_LIGHT0).

Suposeu que les llums no són SPOT. Per tant la fórmula a aplicar serà la de Phong:

$$K_e + K_a(M_a + I_a) + K_d I_d(N \cdot L) + K_s I_s(R \cdot V)^s$$

on

$K_e$  = emissió del material

$K_a$ ,  $K_d$ ,  $K_s$  = reflectivitat ambient, difosa i especular del material

$s$  = shininess del material

$M_a$  = llum ambient del model (gl\_LightModel.ambient)

$I_a$ ,  $I_d$ ,  $I_s$  = propietats ambient, difosa i especular de la llum GL\_LIGHT0.

$N$  = vector normal unitari

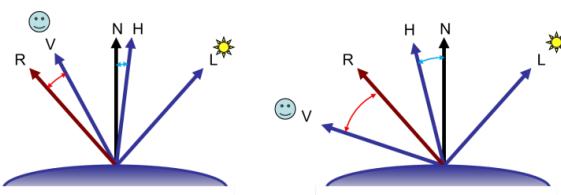
$L$  = vector unitari cap a la font de llum

$V$  = vector unitari del vèrtex cap a la càmera

$R$  = reflexió del vector  $L$  respecte  $N$ . Es pot calcular com  $R=2(N \cdot L)N-L$

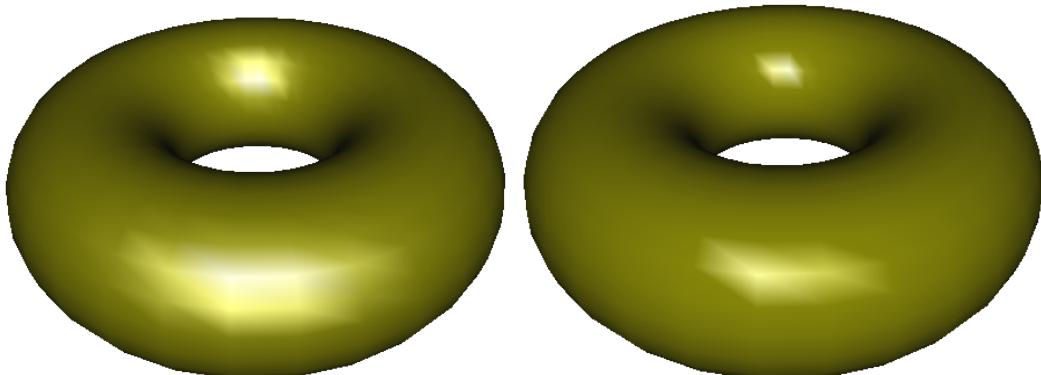
Totes les components estan en coordenades de la càmera.

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 escalaris  $N \cdot L$  o  $R \cdot V$  són negatius. Per tant haureu de fer servir  $\max(0.0, \text{dot}(N, L))$  i  $\max(0.0, \text{dot}(R, V))$ .

Aquí teniu una comparació del model Blinn-Phong amb el model de Phong:

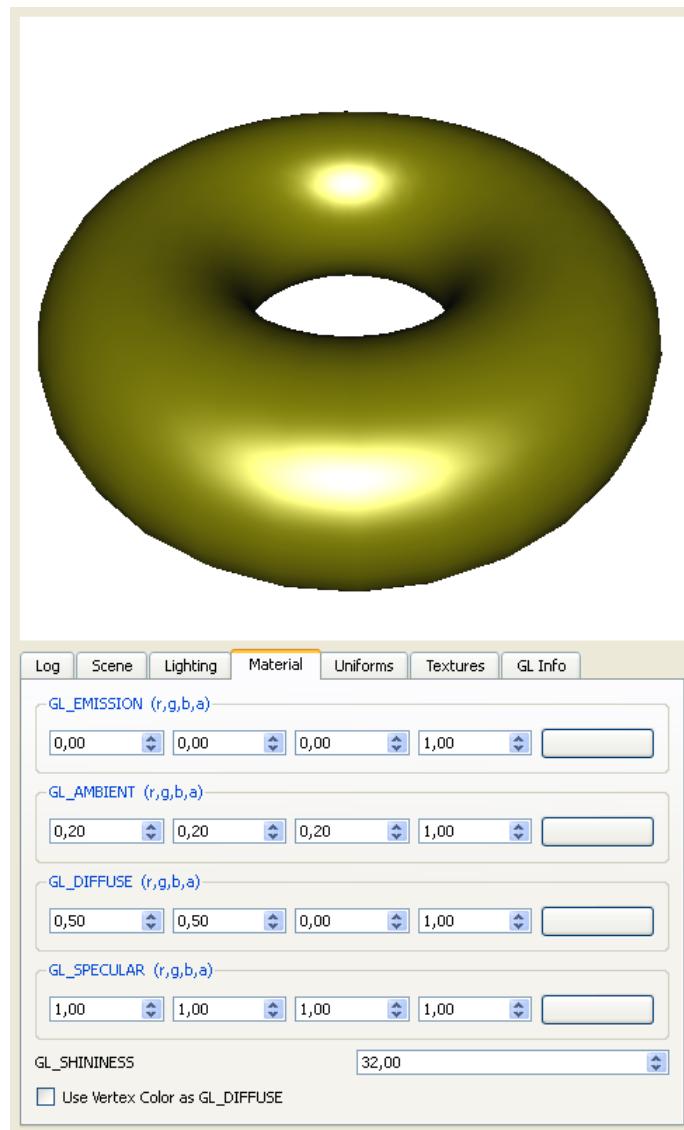


## Lighting (3)

Escriu un **vertex shader** i un **fragment shader** que calculi la il·luminació per fragment fent servir el model de Blinn. Només cal que considereu la primera font de llum GL\_LIGHT0.

Necessitarreu 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 de definir aquestes variables varying.

Aquí teniu un exemple del resultat esperat:

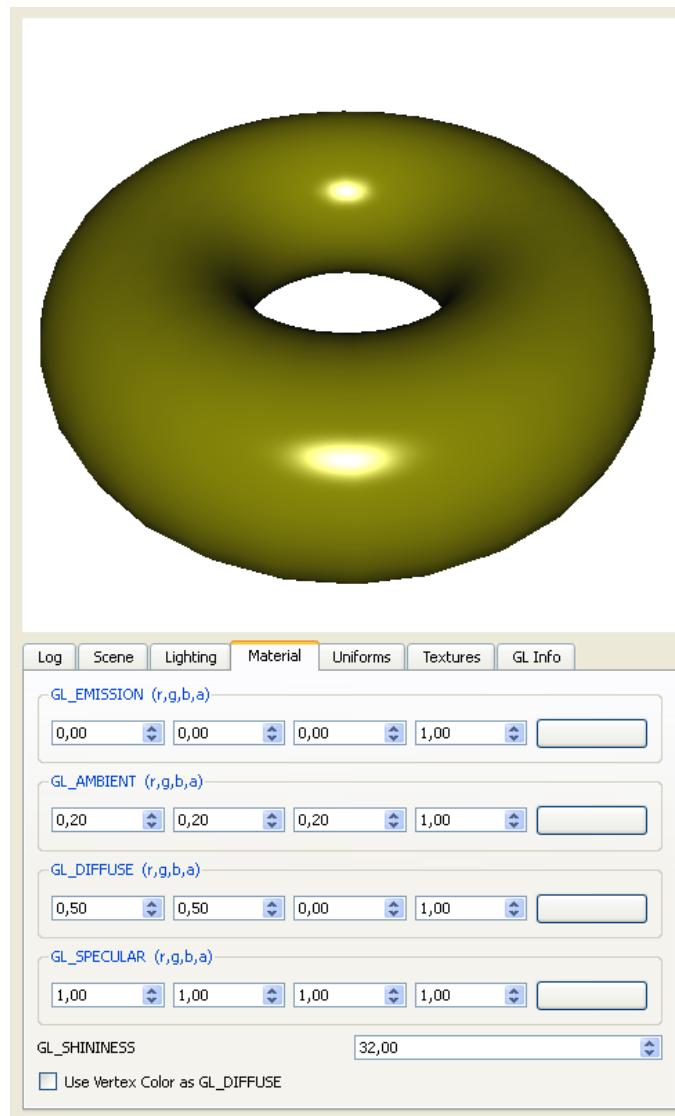


## Lighting (4)

Escriu un **vertex shader** i un **fragment shader** que calculi la il·luminació per fragment fent servir el model de Phong. Només cal que considereu la primera font de llum GL\_LIGHT0.

Necessitarreu 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 de definir aquestes variables varying.

Aquí teniu un exemple del resultat esperat:



---

## Lighting (5) (1<sup>er</sup> control laboratori, 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 dependent 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,

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

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 = pow( RdotV, gl_FrontMaterial.shininess );
    return
        gl_FrontMaterial.emission +
        gl_FrontMaterial.ambient * gl_LightModel.ambient +
        gl_FrontMaterial.ambient * gl_LightSource[0].ambient +
        gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse * Idiff+
        gl_FrontMaterial.specular * gl_LightSource[0].specular * 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.

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

---

## Basic texture

---

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

```
gl_FrontColor = vec4((gl_NormalMatrix * gl_Normal).z);
```

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

```
gl_TexCoord[0] = gl_MultiTexCoord0;
```

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

```
gl_FragColor = gl_Color * texture2D(sampler, gl_TexCoord[0].st);
```

On la variable sampler és una textura 2D:

```
uniform sampler2D sampler;
```

Aquí teniu un exemple del resultat esperat, amb el torus texturat amb Fieldstone.png:



---

## Animate texture

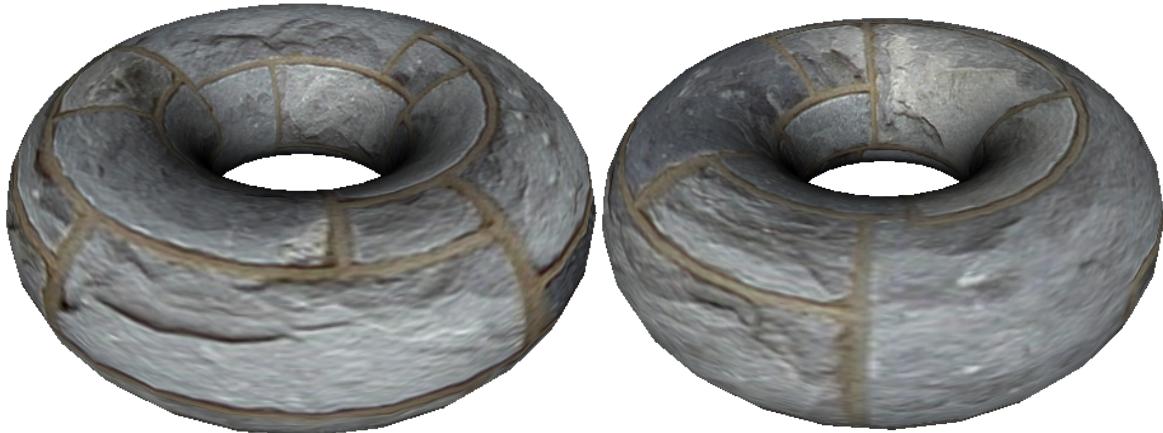
---

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 speed proporcionat per l'usuari).

Feu servir el uniform time que us proporciona el Shader Maker.

Aquí teniu dos frames de l'animació, incrementant simultàniament les coordenades s i t, amb la textura Fieldstone.png:



Teniu un vídeo d'exemple amb speed = 0.1 a animate-texture.mp4

---

## Tiling

(1<sup>er</sup> control laboratori, curs 2011-12, Q2)

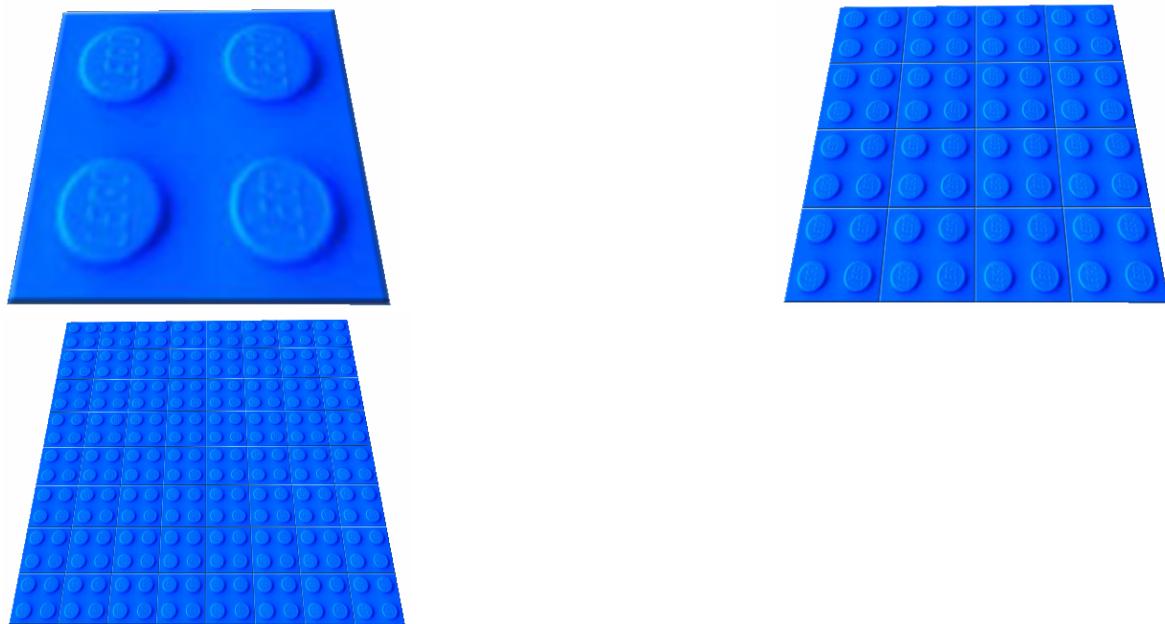
---

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.

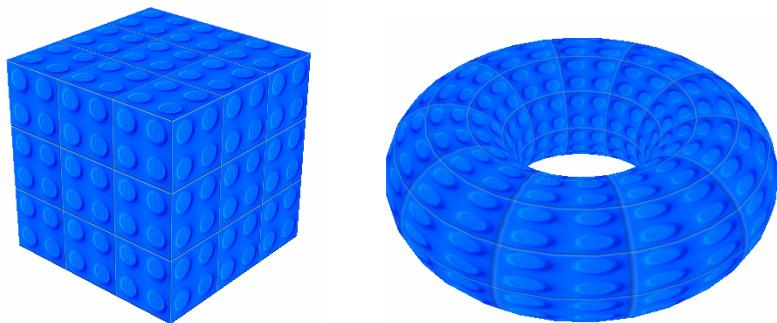
El **vertex shader** farà servir una variable **uniform int tiles** que indicarà el nombre de cops que volem que es repeteixi la textura (en horitzontal i vertical) respecte la parametrització original del model.

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

Amb la textura d'una peça de Lego i el model Plane, s'obtenen aquestes imatges (tiles = 1, 4 i 8):



Resultats amb el cub (tiles=3) i el torus (tiles=10):



---

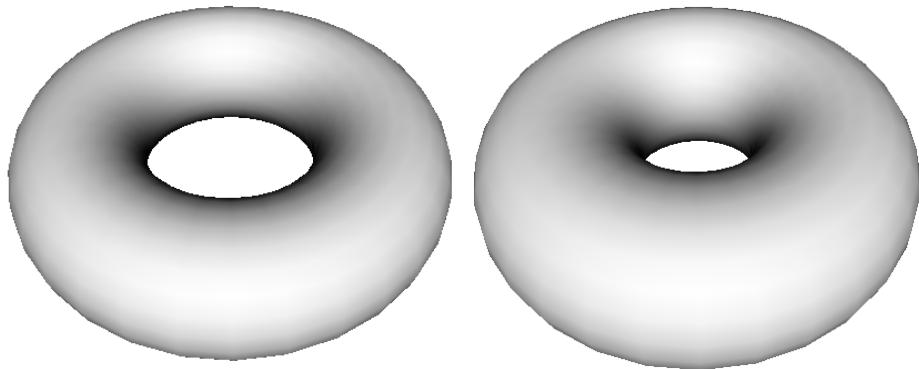
## Animate vertices (1)

---

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 sinusoidal amb una certa amplitud  $A$  i freqüència  $F$  (tots dos uniform float).

Feu servir el uniform time que us proporciona el Shader Maker.

Aquí teniu dos frames de l'animació, amb amplitud 0.1



Teniu un vídeo d'exemple amb amplitud = 0.1 i freqüència = 1Hz a [animate-vertices-1.mp4](#)

---

## Animate vertices (2)

---

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  $A$ , freqüència  $F$  (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.

Feu servir el uniform time que us proporciona el Shader Maker.

Aquí teniu dos frames de l'animació, amb amplitud 0.1, amb el model Plane:



Teniu un vídeo d'exemple amb amplitud = 0.1 i freqüència = 1Hz a animate-vertices-2.mp4

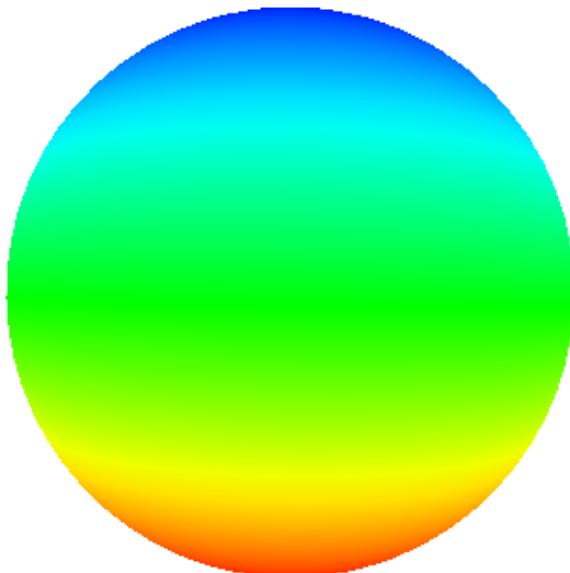
## Gradient de color (1)

Escriu un **vertex shader** que apliqui un gradient de color al model segons la seva coordenada Y en object space. L'aplicació rebrà dos uniforms minY, maxY amb els valors extrems de la coordenada Y del model.

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  $y=\text{minY}$  es pintin de vermell, i els vèrtexs amb  $y=\text{maxY}$  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.

Aquí teniu la imatge que s'espera amb el model de l'esfera:



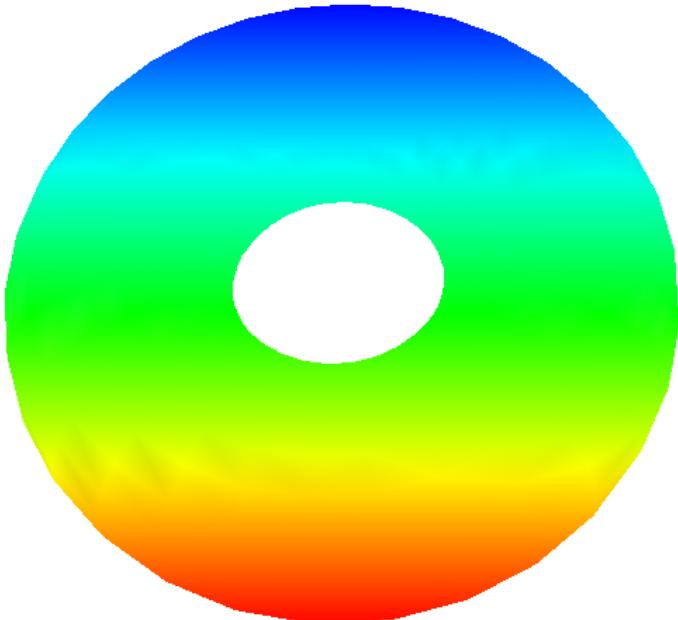
## Gradient de color (2)

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.

Aquí teniu la imatge que s'espera amb el model del torus:



## Foldme

(1<sup>er</sup> control laboratori, curs 2012-13, Q1)

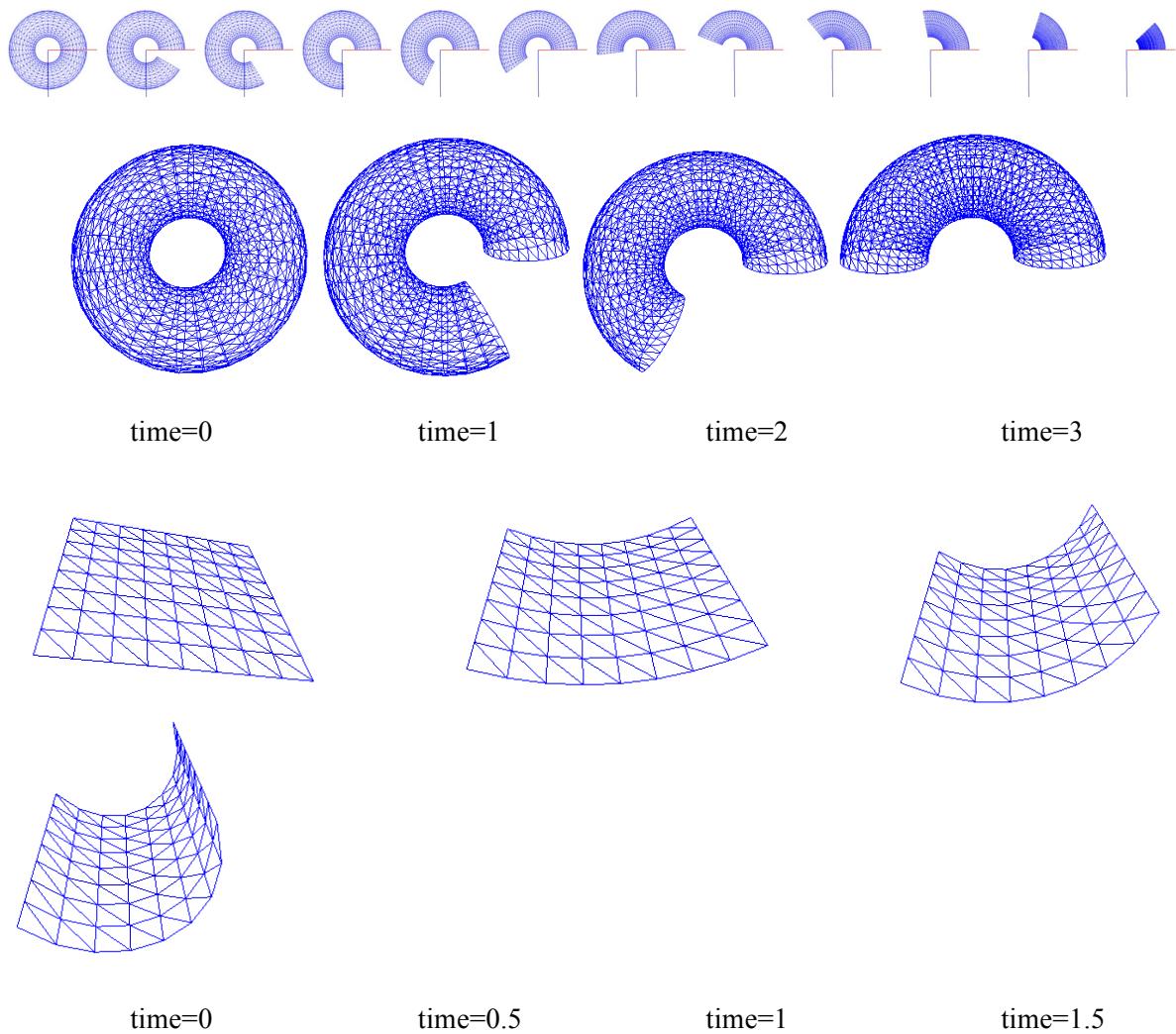
Escriu un **vertex shader** que, abans de passar el vèrtex a clip space, li apliqui una rotació  $R_Y(\varphi)$  respecte l'eix Y. L'angle de rotació  $\varphi$ , en radians, l'heu de calcular com

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

on  $s$  és la coordenada de textura  $s$  del vèrtex, i  $time$  és el uniform que ens proporciona ShaderMaker amb el temps en segons. Observeu que, tal i com està definit l'angle  $\varphi$ , 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. Aquí teniu alguns exemples (en wireframe i color de fons blanc) amb el torus i el plane:



---

## Oscillate

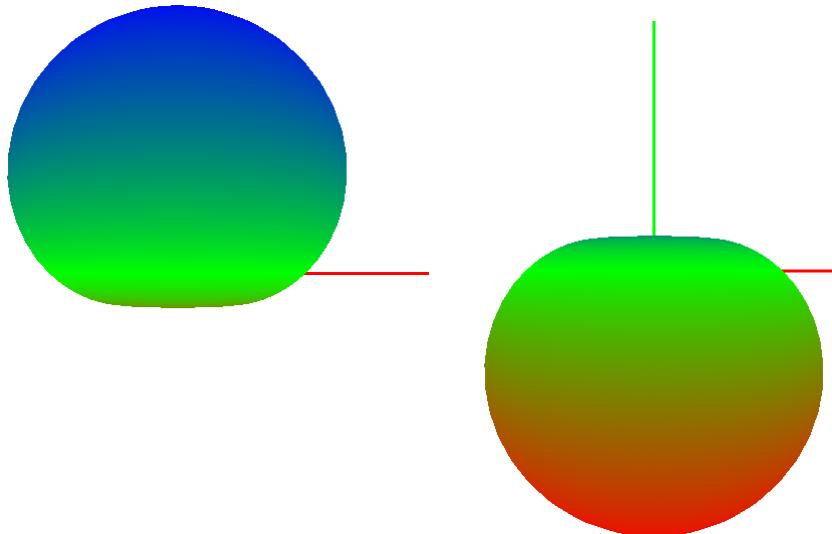
(1<sup>er</sup> control laboratori, curs 2012-13, Q1)

---

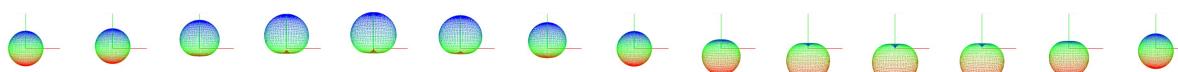
Escriu un vertex shader que pertorbi cada vèrtex del model en la direcció de la seva normal, desplaçant-lo una distància  $d$  que varii sinusoidalment amb amplitud igual a la component  $y$  del vèrtex i període  $2\pi$  segons. Per calcular la amplitud s'agafarà la component  $y$  en eye space si el **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 **gl\_Color**.

Aquí teniu els resultats esperats amb la esfera, després de  $\pi/2$  i  $3\pi/2$  segons (per aquesta vista concreta el resultat no depèn del uniform eyespace) .



Un cicle complet:



---

## Auto-rotate

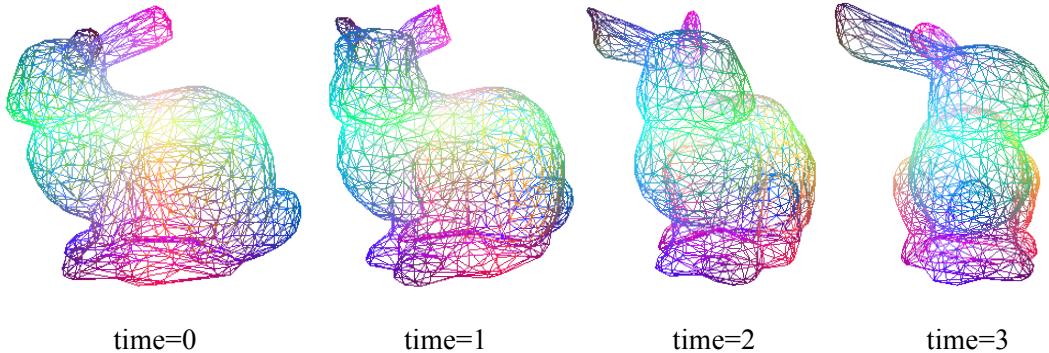
(1<sup>er</sup> control laboratori, curs 2011-12, Q2)

---

El shader maker té una opció per auto-rotar la llum, però no per auto-rotar el model.

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:



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

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

---

## Spherize

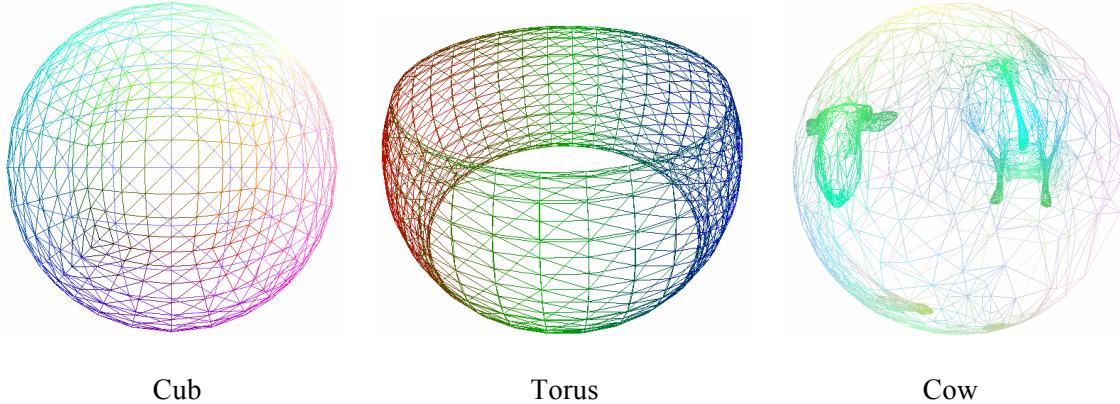
(1<sup>er</sup> control laboratori, curs 2011-12, Q2)

---

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). Aquesta projecció és similar a l'O-mapping del centroïde estudiat a classe, amb el centroïde a l'origen.

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



---

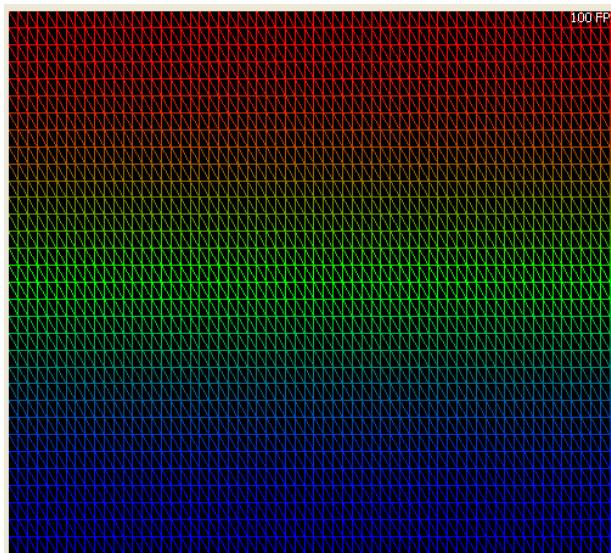
## UV unfold

---

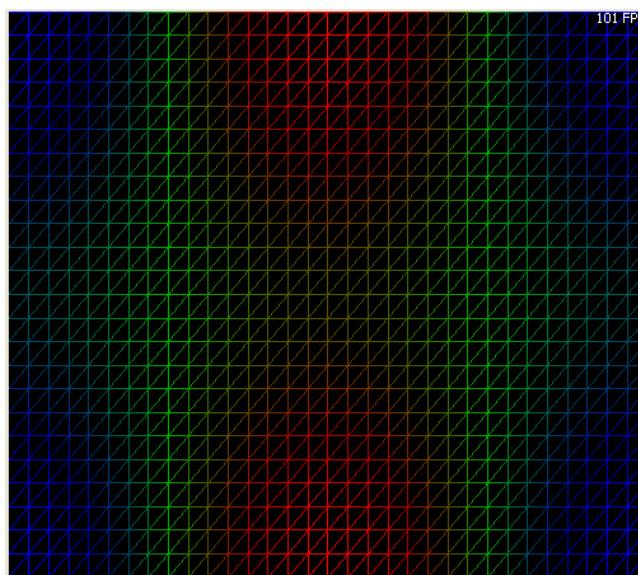
Escriu un **vertex shader** que mostri el model projectat sobre l'espai de textura (assumint que té coordenades de textura 2D definides). Atès que l'espai de textura no està acotat, l'usuari proporcionarà el rectangle de l'espai de textura que vol visualitzar, mitjançant els seus extrems Min, Max (cadascú serà un uniform vec2).

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

Aquí tens un exemple del resultat esperat (amb wireframe rendering) amb el model Sphere, amb Min=(0,0) i Max=(1,1):



I un altre exemple, amb el model Torus i Min=(0, 0.5), Max=(1.0, 1.5):

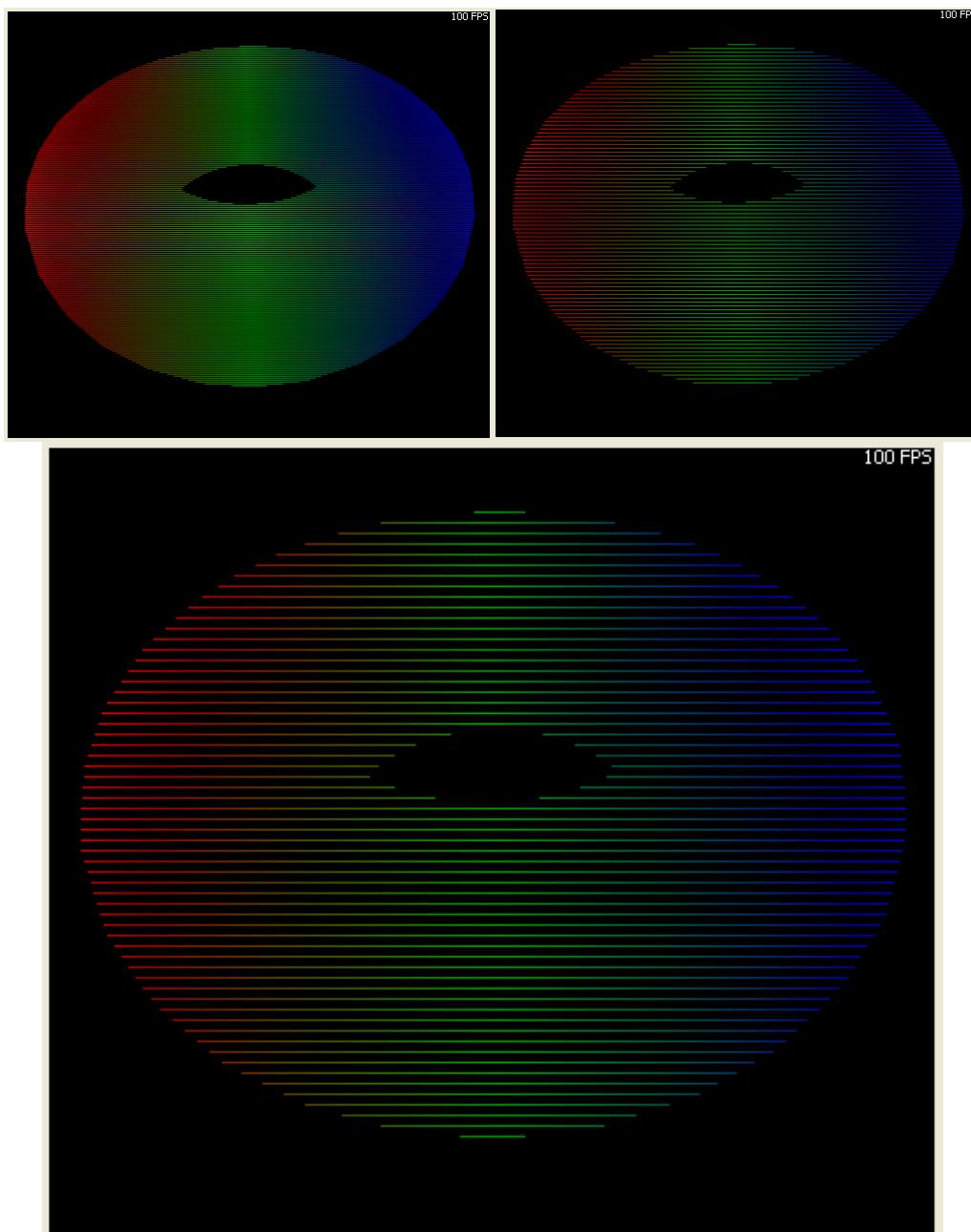


## CRT display

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).

Aquí teniu dos exemples amb el torus, amb  $n=\{2, 4, 6\}$ .

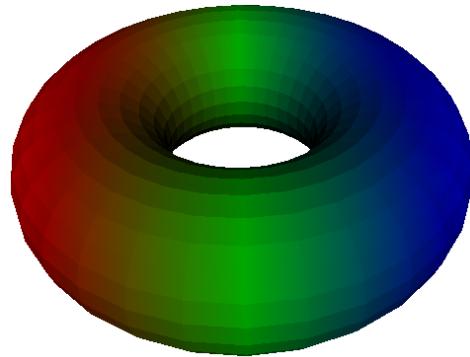


## Calculant la normal al fragment shader

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 `gl_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, amb el torus. 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

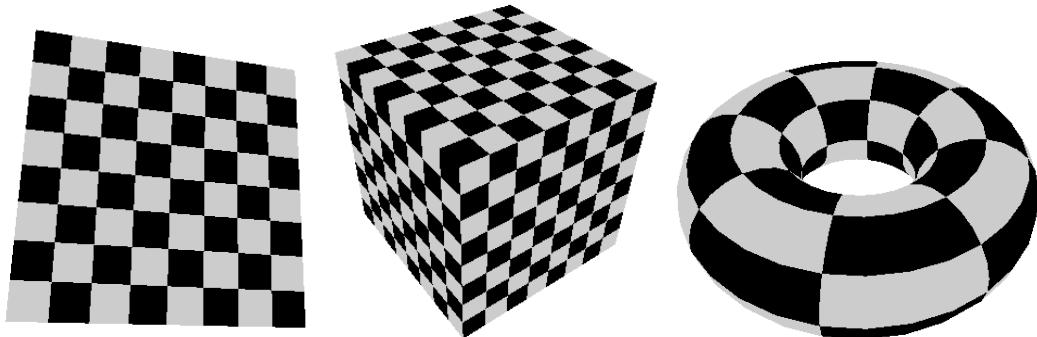
Available only in the fragment shader, `dFdx` and `dFdy` return the partial derivative of expression `p` in `x` and `y`, respectively. Deviations are calculated using local differencing. It is assumed that the expression `p` is continuous and therefore, expressions evaluated via non-uniform control flow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

## 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, que podeu consultar al fragment shader amb `gl_TexCoord[0]`.

Aquí teniu la imatge que s'espera amb els models del pla, cub i torus:



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.

---

## Checkerboard (2)

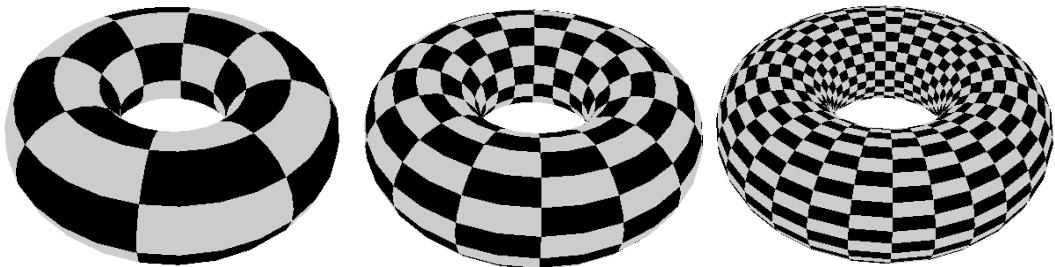
---

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, que podeu consultar al fragment shader amb `gl_TexCoord[0]`.

La part de l'espai de textura entre el (0,0) i el (1,1) estarà ocupada pel tauler de  $N \times N$  cel·les, on  $N$  és un uniform proporcionat per l'aplicació. Podeu assumir que aquest tauler es repeteix indefinidament en totes dues direccions.

Aquí teniu la imatge que s'espera amb el model del torus, per  $N=8, 16$  i  $32$ :



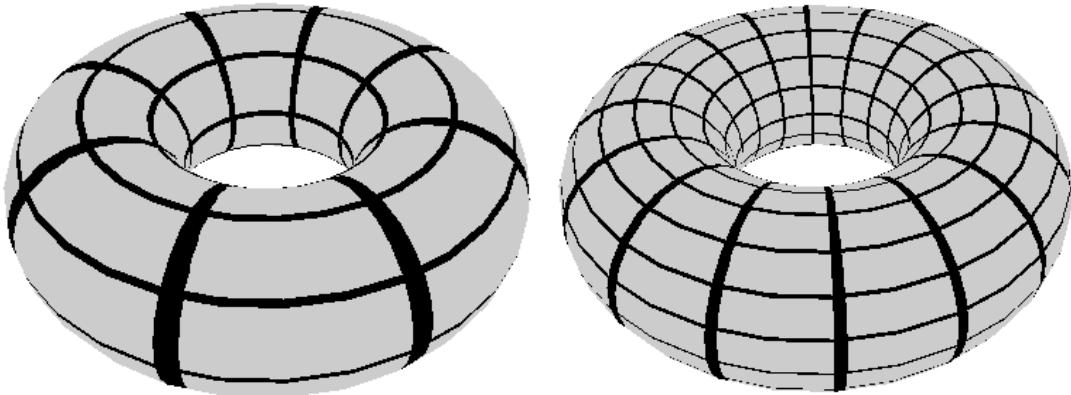
## Checkerboard (3)

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, que podeu consultar al fragment shader amb `gl_TexCoord[0]`.

La part de l'espai de textura entre el (0,0) i el (1,1) estarà ocupada pel tauler de  $N \times N$  cel·les, on  $N$  és un uniform proporcionat per l'aplicació. Podeu assumir que aquest tauler es repeteix indefinidament en totes dues direccions.

Aquí teniu la imatge que s'espera amb el model del torus, per  $N=8$  i 16:



---

## Senyera

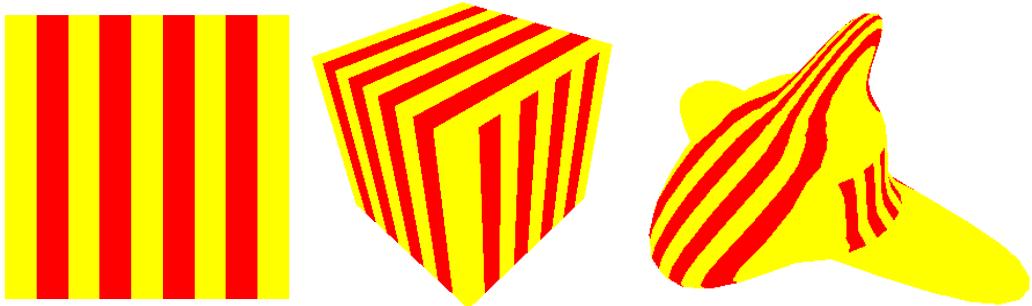
(1<sup>er</sup> control laboratori, curs 2012-13, Q1)

---

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.

Aquí teniu exemples amb el plane, cub i boid:



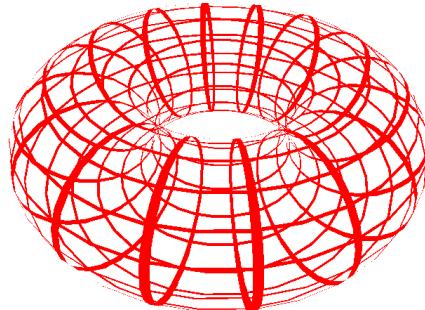
---

## Checkerboard (4)

---

Escriu un **fragment shader** similar al demanat a l'exercici anterior, 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.

Aquí teniu la imatge que s'espera amb el model del torus, per N=16:



---

## Uncover

(1<sup>er</sup> control laboratori, curs 2012-13, Q1)

---

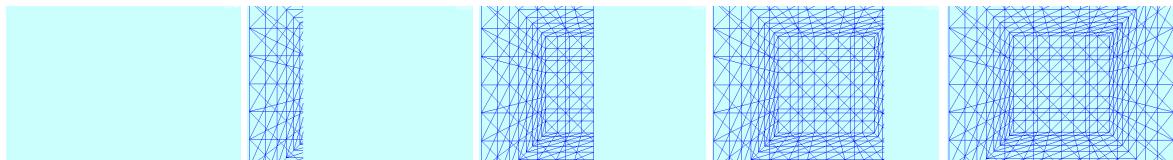
Una transició típica als programes d'edició de video/presentacions és descobrir el contingut de forma progressiva, per exemple d'esquerra a dreta. Escriu un **vertex shader** i un **fragment shader** que conjuntament simulin aquesta transició. Per aconseguir aquest efecte, cal que el FS comenci descartant 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 ( $\text{time}=0$ ), el FS descartarà tots els fragments
- Al cap d'un segon ( $\text{time}=1$ ), serà visible només la meitat esquerra del viewport,
- A partir dels dos segons ( $\text{time}>=2$ ), no es descartarà cap fragment.

El FS assignarà als fragments no descartats el **color blau** (no cal cap càcul 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.



---

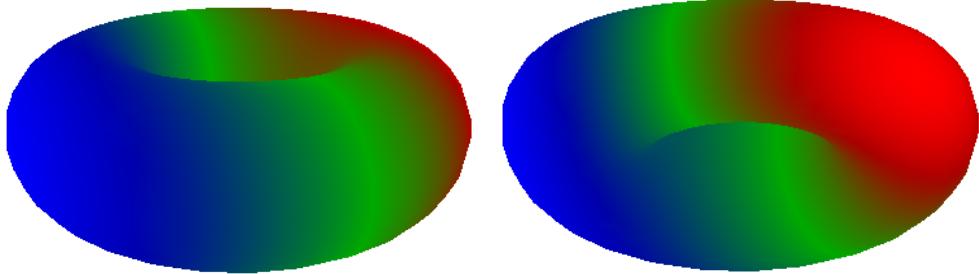
## Reverse Z (1)

---

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 llunyanas a l'observador.

Aquí teniu un exemple amb el model del torus, sense el vertex shader, i amb el vertex shader:



---

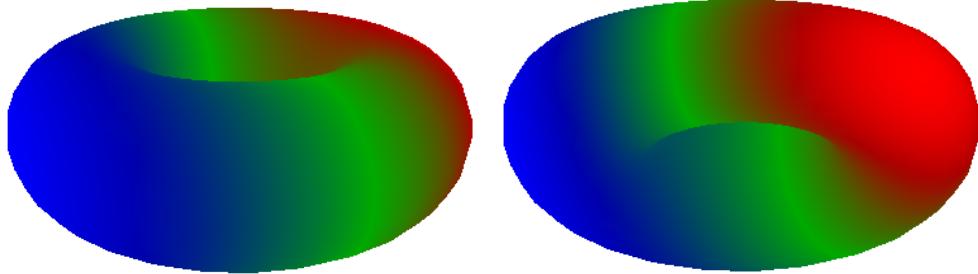
## Reverse Z (2)

---

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 resultat és que s'invertirà la visibilitat de les cares, sent visibles les cares més llunyanas a l'observador.

Aquí teniu un exemple amb el model del torus, sense el fragment shader, i amb el fragment shader:



---

## Projective texture mapping (2<sup>on</sup> control laboratori, curs 2011-12, Q2)

---

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

El **vertex shader** haurà d'escriure en `gl_TexCoord[0]` 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):



---

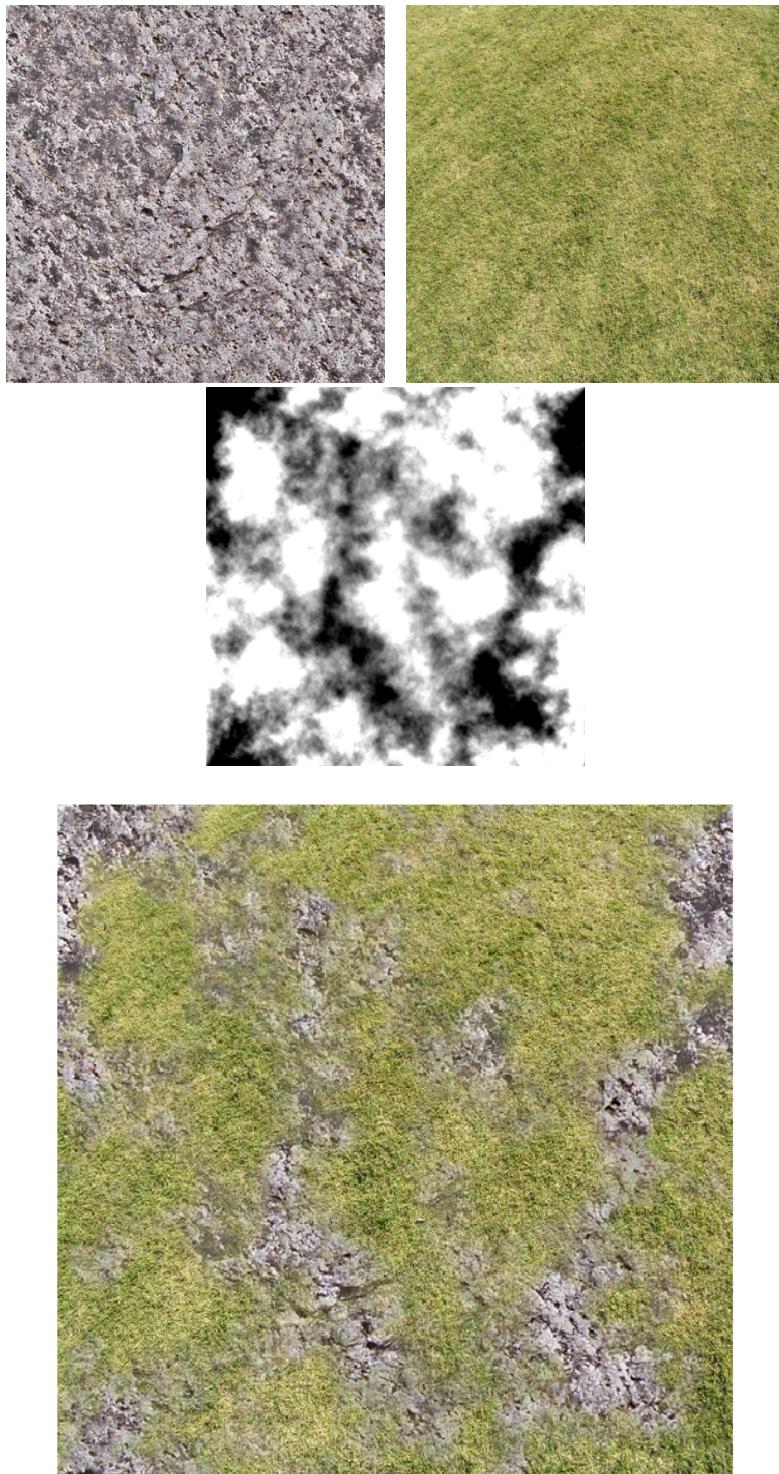
## Texture splatting

(2<sup>on</sup> 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** per ShaderMaker 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 ShaderMaker.

No feu servir cap tipus d'il·luminació.

El resultat esperat amb les textures anteriors i l'objecte Torus és aquest:



## Sphere Mapping (2<sup>on</sup> control laboratori, curs 2011-12, Q2)

Escriviu un **vertex shader** i un **fragment shader** per ShaderMaker 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 texture2D(sampler, st);
}
```

---

## Skymap

(2<sup>on</sup> 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 texture2D(sampler, st);
}
```

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

