Basic lighting (1)

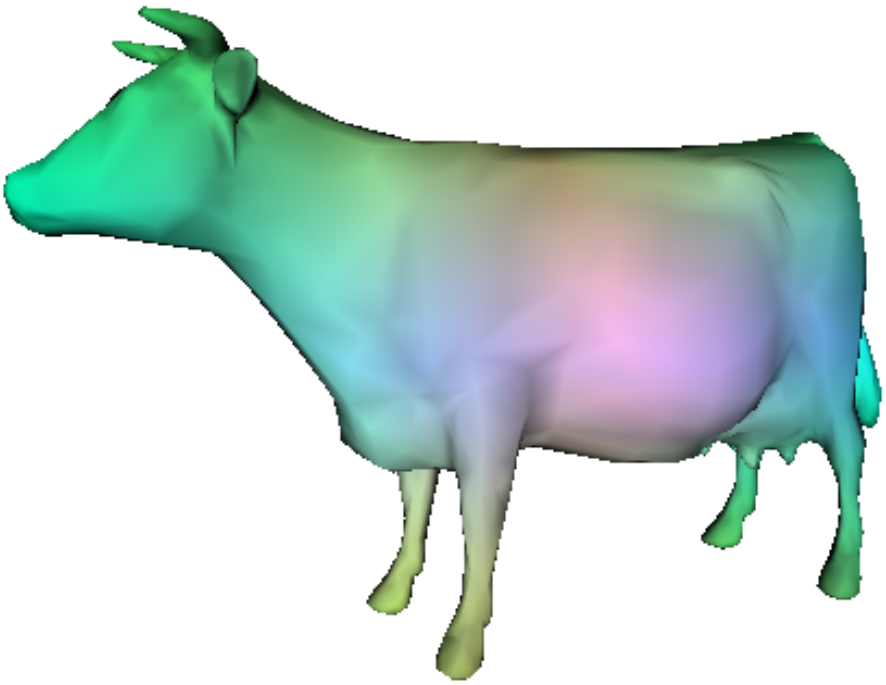
Durant el desenvolupament de shaders sovint ens pot interessar aconseguir un cert efecte d'il·luminació de la forma més senzilla possible.

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)

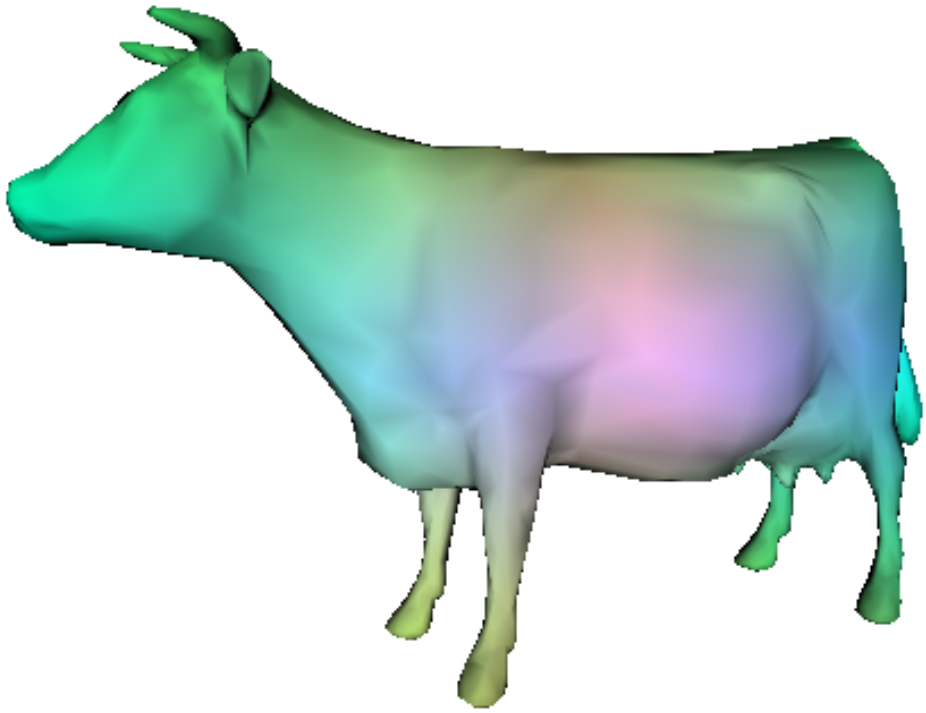
Durant el desenvolupament de shaders sovint ens pot interessar aconseguir un cert efecte d'il·luminació de la forma més senzilla possible.

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).

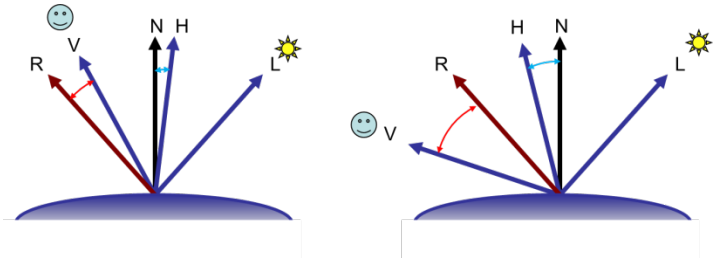
Suposeu que les llums no són SPOT. Per tant la fórmula a aplicar serà la de Blinn-Phong:

$$K_e + K_a(M_a + I_a) + K_dI_d(N \cdot L) + K_sI_s(N \cdot H)^s$$

on

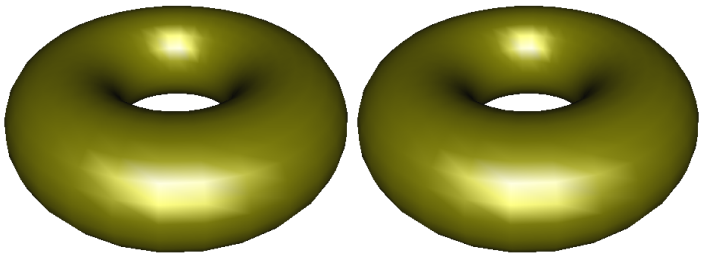
K_e = emissió del material
K_a, K_d, K_s = reflectivitat ambient, difosa i especular del material
s = shininess del material
M_a = llum ambient del model (gl_LightModel.ambient)
I_a, I_d, I_s = propietats ambient, difosa i especular de la llum GL_LIGHT0.
N = vector normal unitari (eye space)
L = vector 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'infinit 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 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)).

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:

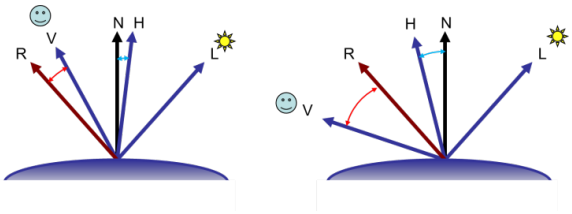
$$K_e + K_a(M_a + I_a) + K_dI_d(N \cdot L) + K_sI_s(R \cdot V)^s$$

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·L)N-L

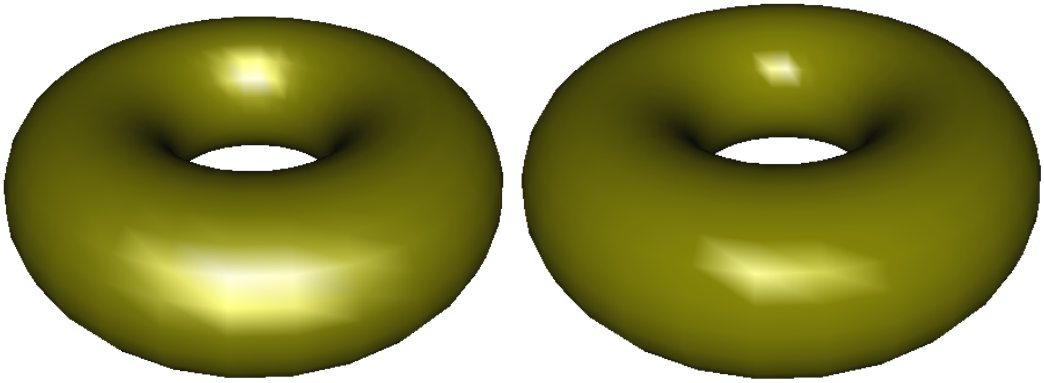
Totes les components estan en coordenades de la càmera.

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

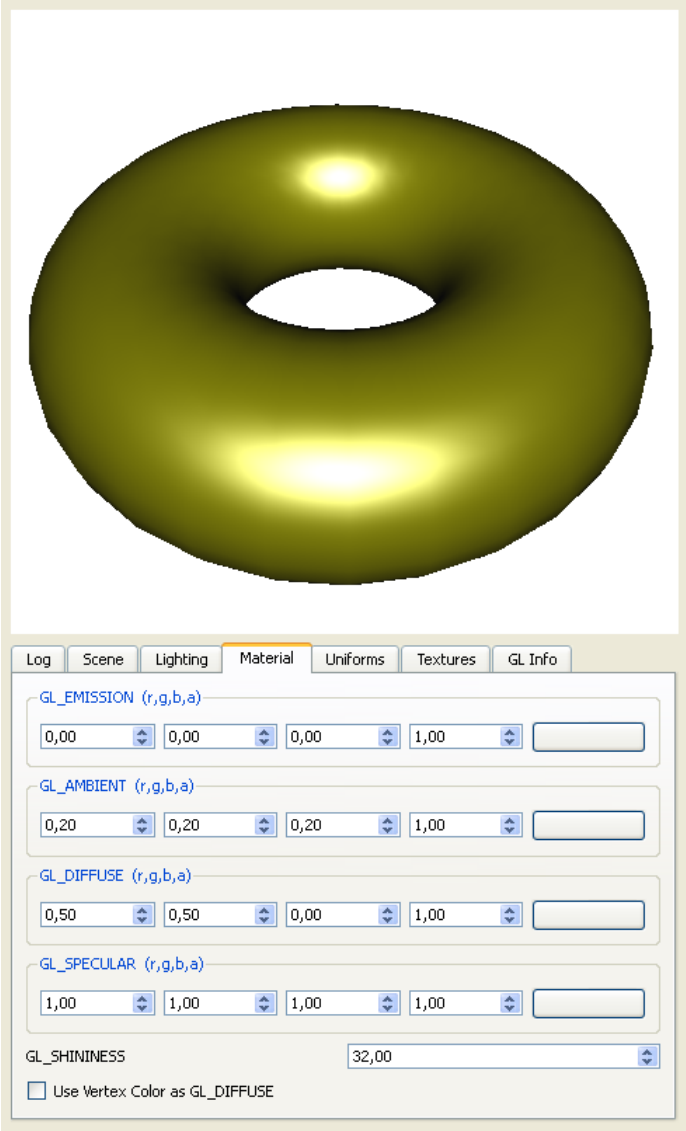


Lighting (3)

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.

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 de definir aquestes variables varying.

Aquí teniu un exemple del resultat esperat:

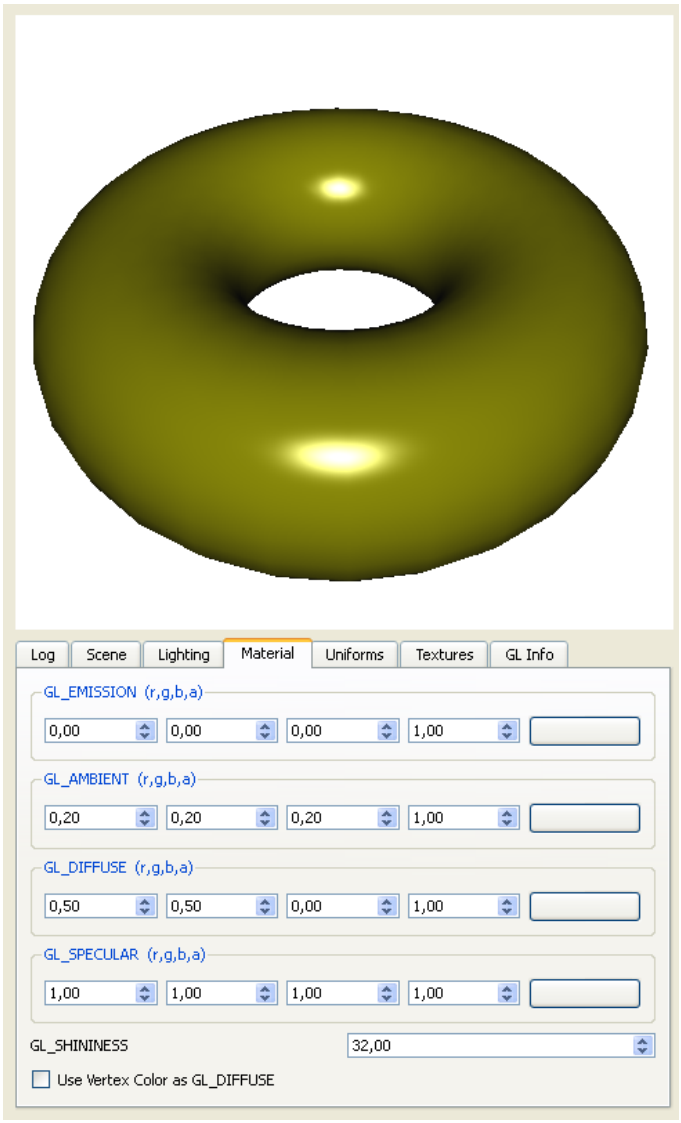


Lighting (4)

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.

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 de definir aquestes variables varying.

Aquí teniu un exemple del resultat esperat:



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:

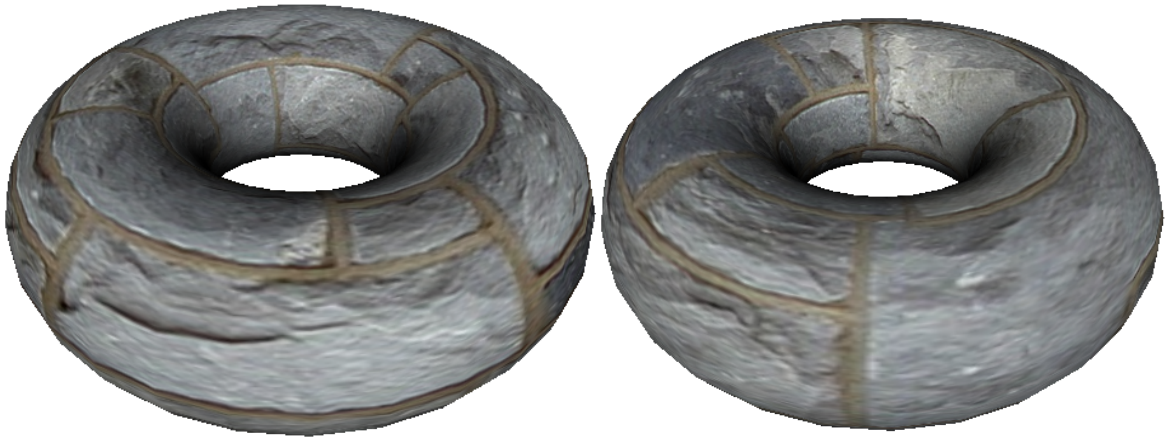
Animate texture

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:



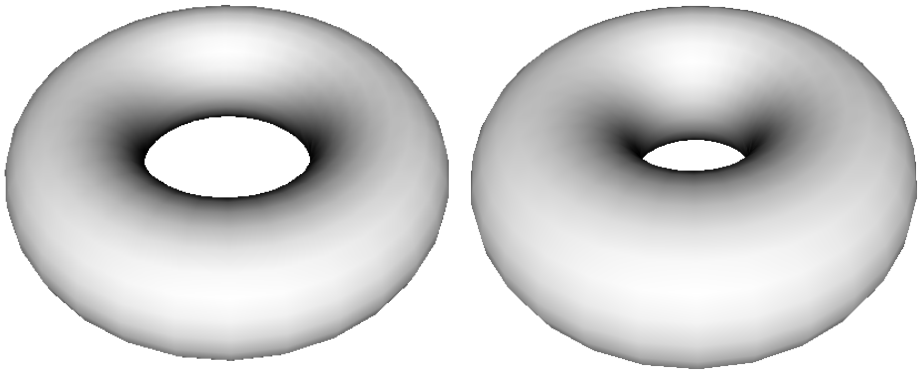
Teniu un vídeo d'exemple amb speed = 0.1 a animate-texture.mp4

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 sinusoïdal 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



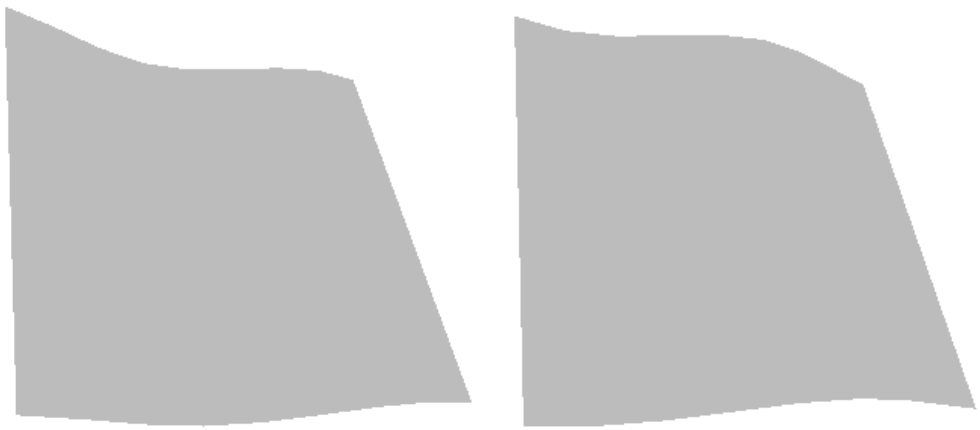
Teniu un vídeo d'exemple amb amplitud = 0.1 i freqüència = 1Hz a animate-vertices-1.mp4

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=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.

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

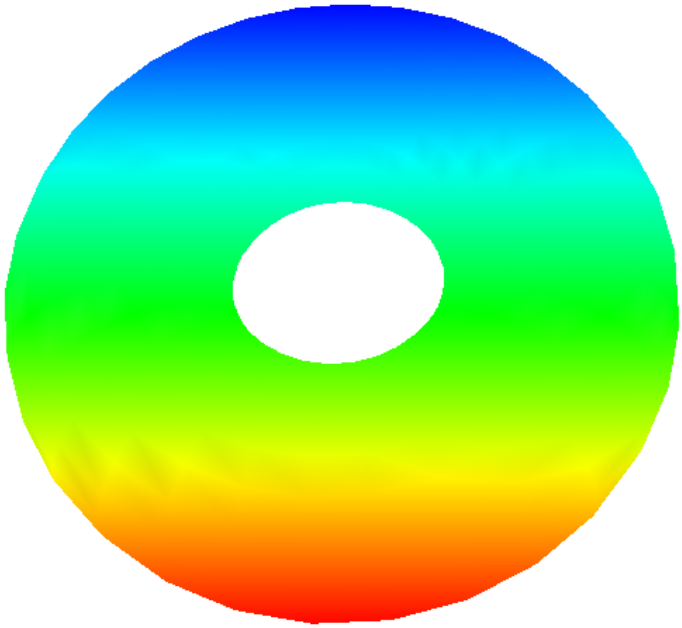
Gradient de color (2)

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:

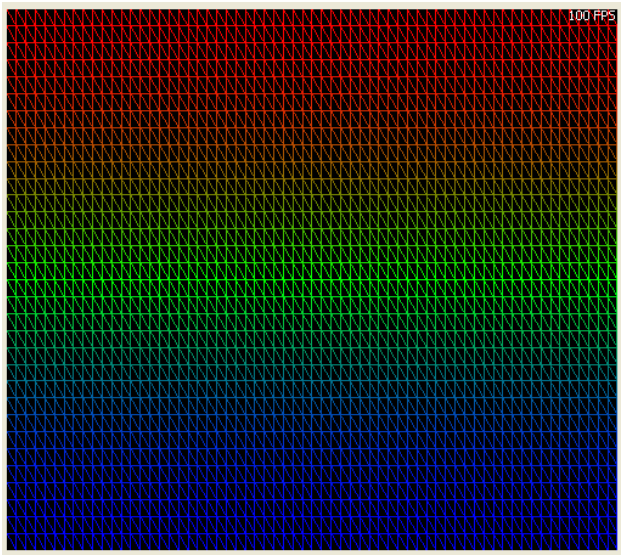


UV unfold

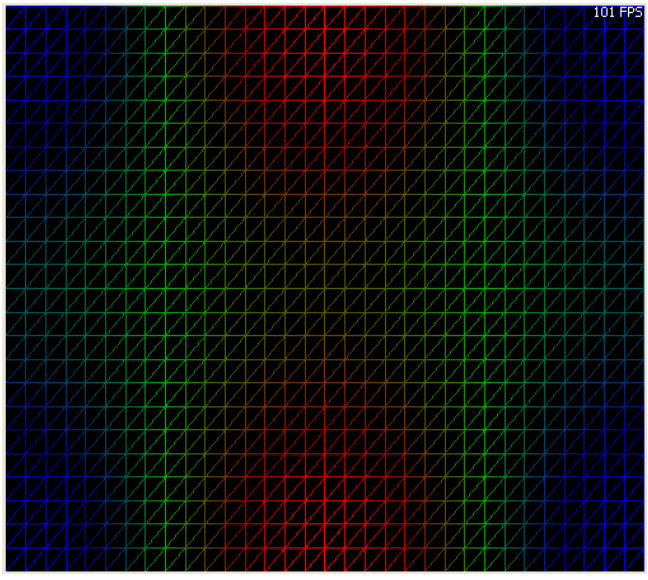
Escriu un **vertex shader** que mostri el model projectat sobre l'espai de textura (assumint que té coordenades de textura 2D definides). Atès que l'espai de textura no està acotat, l'usuari proporcionarà el rectangle de l'espai de textura que vol visualitzar, mitjançant els seus extrems Min, Max (cadascú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

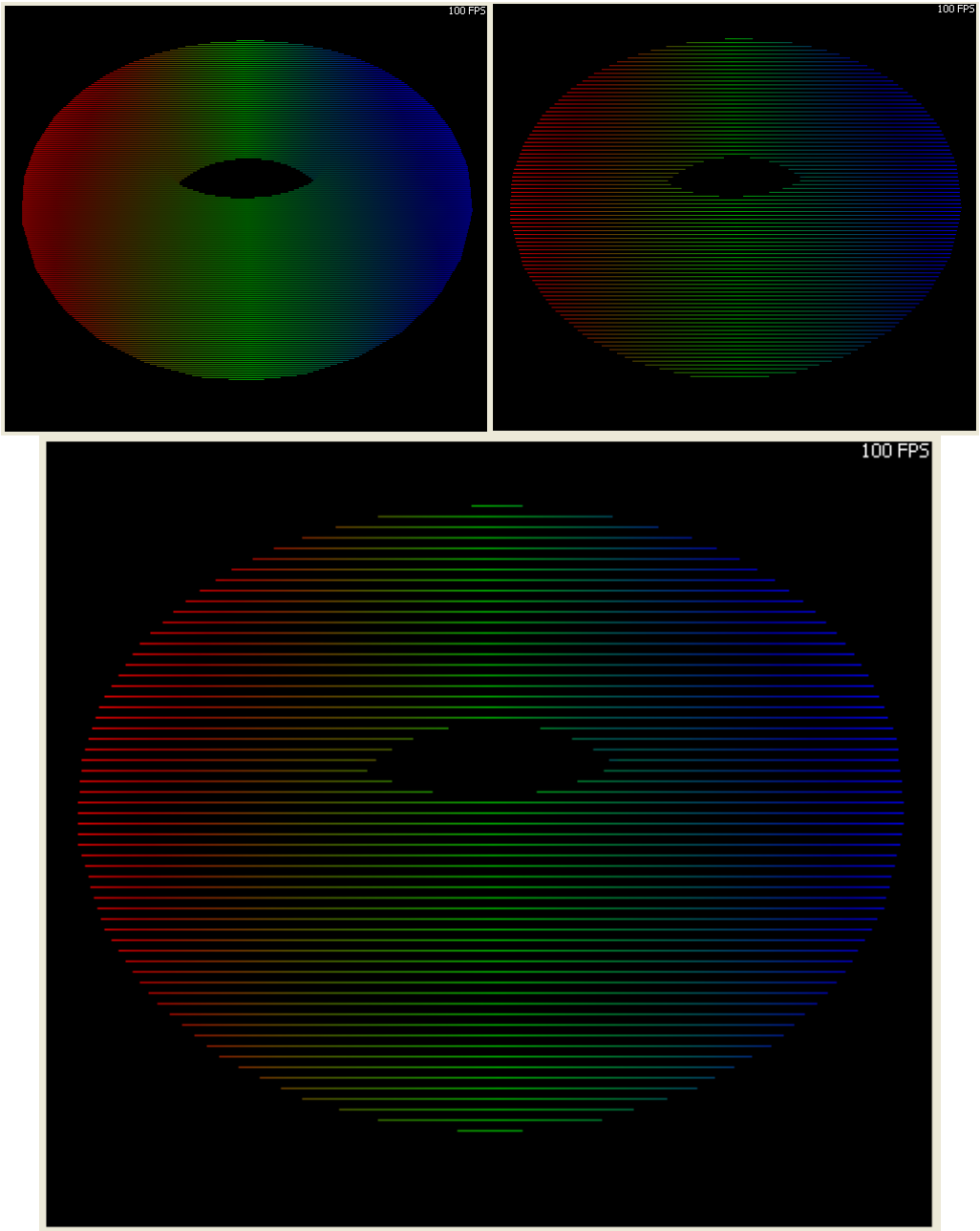


CRT display

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}.

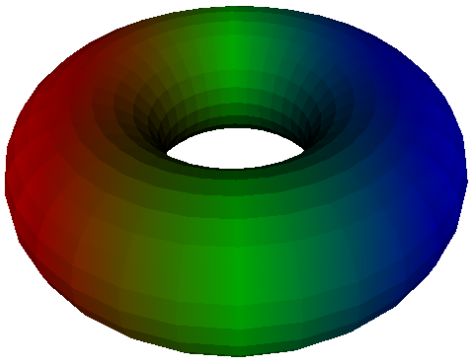


Calculant la normal al fragment shader

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

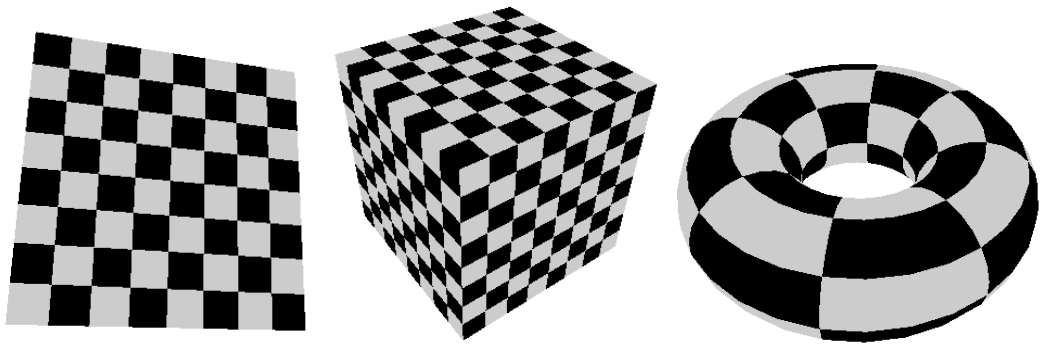
Available only in the fragment shader, dFdx and dFdy return the partial derivative of expression p in x and y, respectively. Deviatives are calculated using local differencing. It is assumed that the expression p is continuous and therefore, expressions evaluated via non-uniform control flow may be undefined.

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.

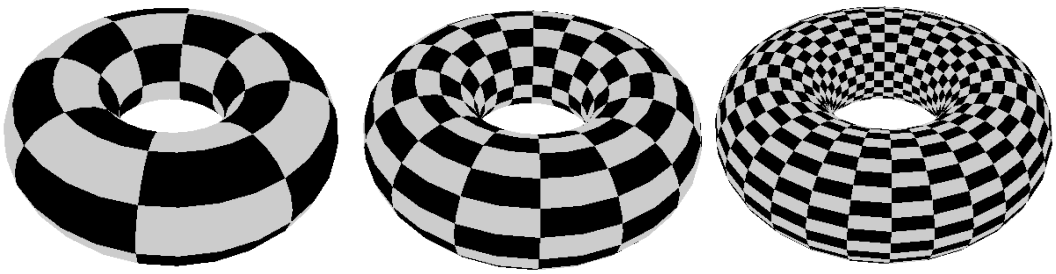
Checkerboard (2)

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*x*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:



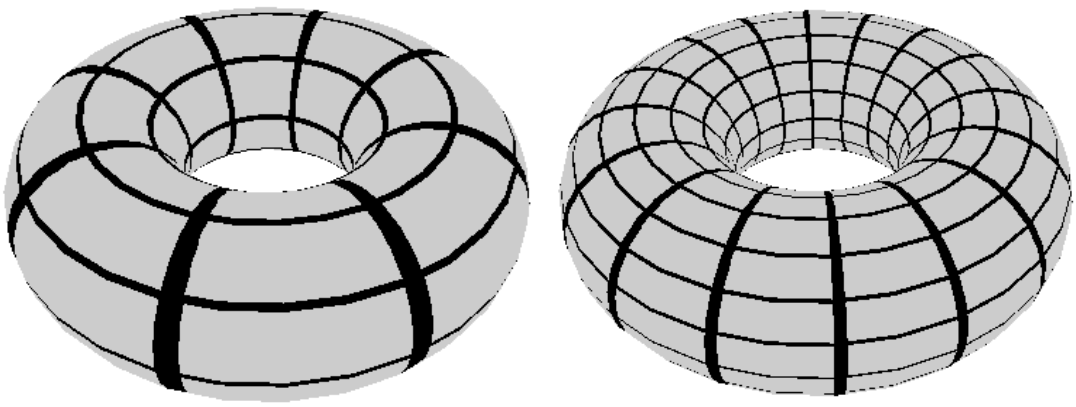
Checkerboard (3)

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*x*N* cel·les, on *N* és un uniform proporcionat per l'aplicació. Podeu assumir que aquest tauler es repeteix indefinidament en totes dues direccions.

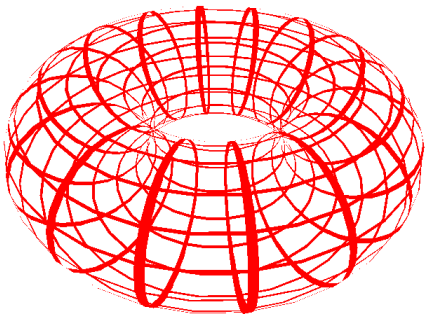
Aquí teniu la imatge que s'espera amb el model del torus, per N=8 i 16:



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:

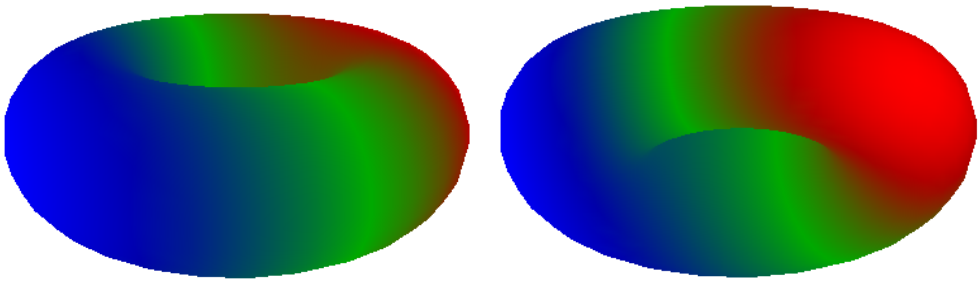


Reverse Z (1)

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 llunyanes 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 llunyanes a l'observador.

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