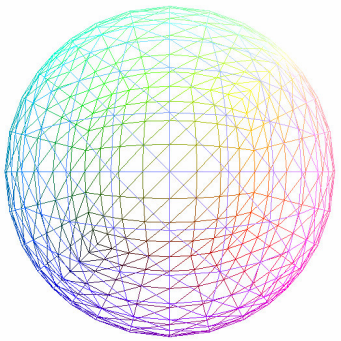
Spherize 2 punts

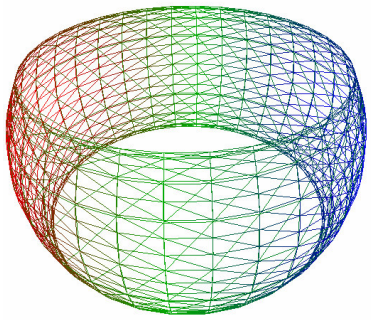
Escriu un **vertex shader** que, abans de transformar cada vèrtex, el projecti sobre una esfera de radi unitat centrada a l'origen del sistema de coordenades del model.

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 centroide estudiat a classe, amb el centroide a l'origen.

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



Cub



Torus

Cow

Identificadors (ús obligatori)

```
spherize.vert
```

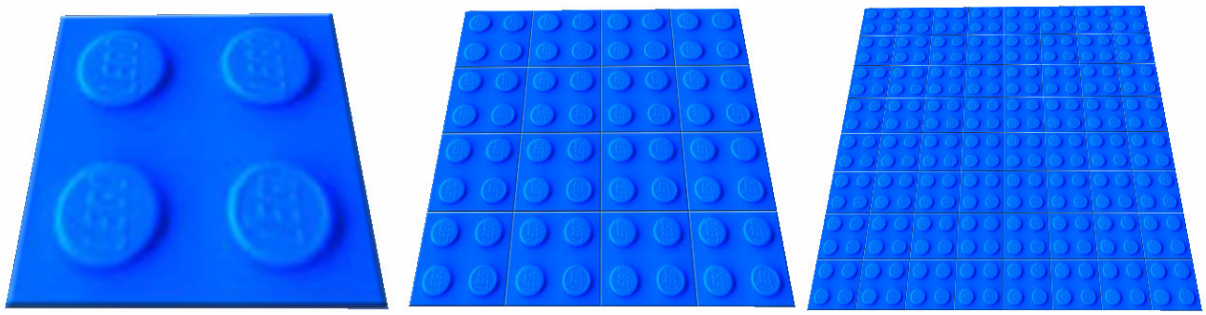
Tiling 3 punts

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.

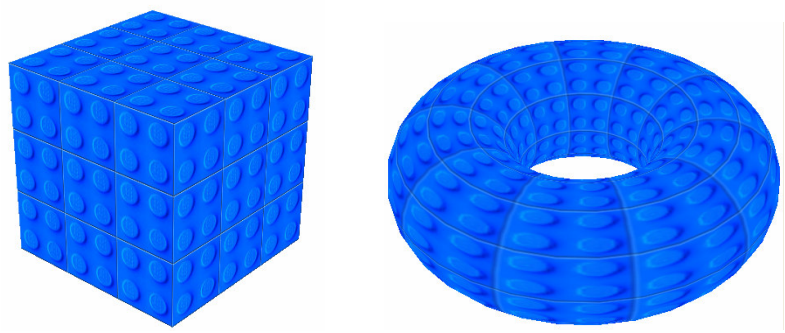
El **vertex shader** farà servir una variable **uniform int tiles** que indicarà el nombre de cops que volem que es repeteixi la textura (en horitzontal i vertical) respecte la parametrització original del model.

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):



Identificadors (ús obligatori)

```
uniform int tiles;

uniform sampler2D sampler;

tiling.vert

tiling.frag
```

Lighting (5) 3 punts

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,

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

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)
{









}
```

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.

Identificadors (ús obligatori)

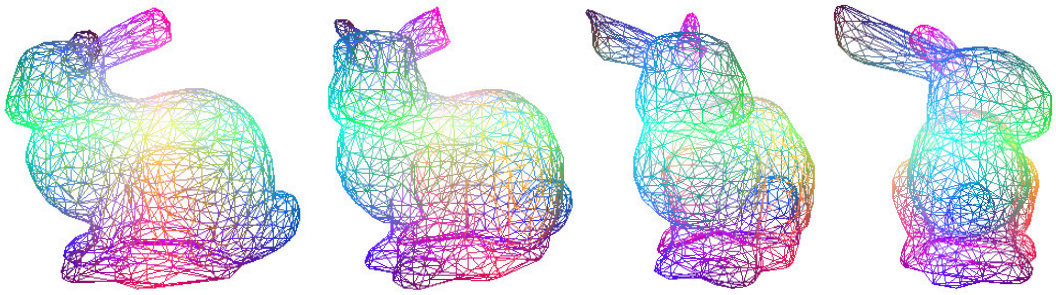
```
uniform bool world;

lighting-5.vert

lighting-5.frag
```

Auto-rotate 2 punt

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:



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:

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

Identificadors (ús obligatori)

```
uniform float speed

auto-rotate.vert
```