

## Índex

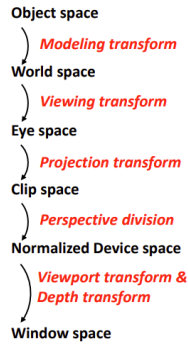
Teoria:.....	3
1. Procés de visualització projectiu:.....	3
Sistemes de coordenades:.....	3
Pipeline OpenGL:.....	5
2. Textures .....	6
Mètodes per a generar coordenades de textura: .....	6
Creació de la textura:.....	7
Filtrat .....	7
Perspective-correct interpolation.....	8
Projective texture mapping.....	9
3. Ombres.....	9
Ombres per projecció (pla).....	10
Stencil Buffer: .....	10
Ombres per projecció (amb Stencil).....	12
Matrius de projecció:.....	12
Volums d'ombra:.....	12
Shadow mapping: .....	13
4. Reflexions especulars: .....	14
Reflexions amb objectes virtuals.....	15
Matriu de reflexió:.....	15
Environment mapping: .....	15
Sphere Mapping: .....	16
Cube Mapping: .....	17
5. Objectes translúcids: .....	17
Refracció: .....	17

Llei de Snell: .....	17
Equacions de Fresnel:.....	18
Aproximació de Schlick: .....	18
Alpha Blending: .....	18
6. Il·luminació global.....	19
Radiometria: .....	19
Flux Radiant ( $\phi$ ): .....	19
Llei de Lambert: .....	19
Intensitat (I): .....	20
Radiància (L):.....	20
BRDF (Bidirectional Reflectance Distribution Functions) .....	20
Radiosity: .....	21
7. Ray Tracing.....	21
Ray-Tracing clàssic .....	22
Path Tracing: .....	22
Distributed Ray Tracing: .....	22
Two-pass Ray Tracing: .....	23
Ray Tracing Clàssic en Detall: .....	23
Intersecció Raig-Geometria: .....	24

# Teoria:

## 1. Procés de visualització projectiu:

Sistemes de coordenades:



- Object Space ( $x_m, y_m, z_m, w_m$ )
  - Sistemes de coordenades del model i de l'objecte.
  - $w_m$  serà 1 en punts i 0 en vectors.
- Modeling Transform:

$$\text{Translate}(t_x, t_y, t_z) \quad T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Scale}(s_x, s_y, s_z) \quad T = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{Rotate}(a, x, y, z) \quad T = \begin{bmatrix} x^2d + c & xyd - zs & xzd + ys & 0 \\ yxd + zs & y^2d + c & yzd - xs & 0 \\ xzd - ys & yzd + xs & z^2d + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{○ } c = \cos(a), s = \sin(a), d = 1 - \cos(a)$$

$$\text{glRotate}^*(a, 1, 0, 0): \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{glRotate}^*(a, 0, 1, 0): \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

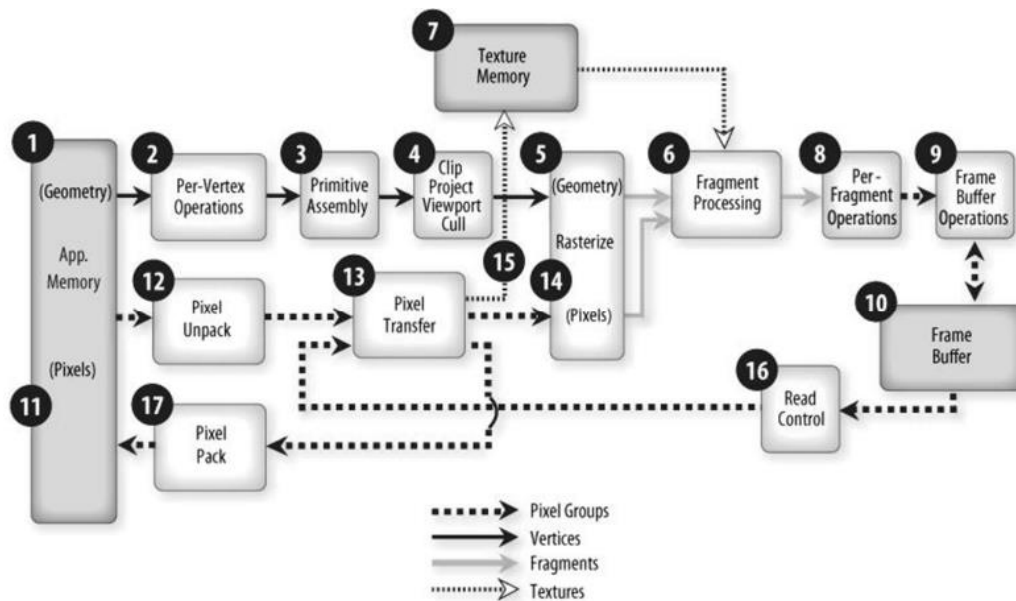
$$\text{glRotate}^*(a, 0, 0, 1): \begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

○

- World Space ( $x_a, y_a, z_a, w_a$ )
  - Sistemes de coordenades del món i de l'aplicació.
- Viewing Transform
  - Es defineix amb crides tipus lookAt, o amb angles d'euler.
- Eye Space ( $x_e, y_e, z_e, w_e$ )
  - Sistemes de Coordenades de l'observador i de la càmera.
  - Si la càmera és perspectiva:  $z_e < 0$  i  $z_e < -z_{\text{near}}$

- Projection transform
  - Es defineix amb crides tipus perspective, frustrum, ortho.
- Clip Space ( $x_c, y_c, z_c, w_c$ )
  - Coordenades de clipping.
  - Si un punt és a l'interior del frustrum:
 
$$\begin{array}{l} -w_c \leq x_c \leq w_c \\ -w_c \leq y_c \leq w_c \\ -w_c \leq z_c \leq w_c \end{array}$$
  - Si la càmera és perspectiva  $w_c < -z_c$
- Perspective Division
  - Pas de coordenades homogènies a 3D. Es divideix cada coordenada per la homogènia:  $(x, y, z, w) \rightarrow (x/w, y/w, z/w)$
- Normalized Device Space ( $x_n, y_n, z_n$ )
  - Coordenades Normalitzades.
  - Punt a l'interior del frustrum:
 
$$\begin{array}{l} -1 \leq x_n \leq 1 \\ -1 \leq y_n \leq 1 \\ -1 \leq z_n \leq 1 \end{array}$$
  - Punts situats sobre znear:  $z = -1$
  - Punts situats sobre zfar:  $z = 1$
- Viewport Transform i Depth Transform:
  - glViewport i glDepthRange (default: [0,1])
- Window Space ( $x_d, y_d, z_d$ )
  - Punt a l'interior del frustrum:
 
$$\begin{array}{l} 0 \leq x_d \leq w \\ 0 \leq y_d \leq h \\ 0 \leq z_d \leq 1 \end{array}$$
  - Punts situats al znear:  $z = 0$
  - Punts situats al zfar:  $z = 1$
  - Quan es generin fragments:  $gl\_FragCoord.w = 1/w_c = -1/z_c$

## Pipeline OpenGL:



1: Dibuix de primitives: Les primitives (punts, línies, polígons...) es poden pintar:

- Vèrtex a vèrtex (`glBegin`, `glVertex`, `glEnd`)
- Vertex Array Object (VAO) (`glDrawArrays`, `glDrawElements...`)

2: Pre-vertex operations: Es transformen els vèrtex (modelview i projection) i les normals.  
Es pot calcular la il·luminació del vèrtex.

- Coordenades: Object Space
- Normal: Object Space
- Color: RGBA

3: Primitive assembly: S'agrupen els vèrtexs per formar primitives. Cada una requereix el seu clipping.

- Coordenades: Clip Space
- Color amb il·luminació: RGBA

4: Primitive Processing:

- Clipping a la piràmide de visió.
- Divisió de perspectiva.
- Viewport i depth transform → window space.
- Backface culling (`glEnable(GL_CULL_FACE)`).

#### 5: Rasterització:

- Es generen els fragments.
- Coordenades: Window Space
- Color: Interpolat.
- TextCoord: Interpolades

#### 6: Fragment Processing: Càlcul del color del fragment.

#### 8: Per-fragment operations: Diversos tests:

- Pixel ownership test.
- Scissor test.
- Alpha test.
- Stencil test.
- Depth test (z-buffer).
- Blending.
- Dithering.
- Logical Ops (glLogicOp)

9: Frame buffer operations: Es modifiquen els buffers escollits amb glDrawBuffers. Es veu afectada per glColorMask, glDepthMask...

## 2. Textures

Una textura és una taula de 1, 2 o 3 dimensions on cada cel·la conté una certa propietat amb 1-4 canals. S'utilitzen normalment al FS.

Texel: "píxel" d'una textura. Normalment, l'alçada i amplada són potències de 2.

(s, t): Coordenades de textura. Es mouen en l'interval [0, 1] i són la versió normalitzada de la x i y originals.

Mètodes per a generar coordenades de textura:

Plans S, T:

Treiem un pla S ( $ax+by+cz+dw=0$ ) i un de T ( $a'x+b'y+c'z+d'w=0$ ) a partir de l'objecte a texturitzar i de les vegades que volguem repetir la textura.

## Creació de la textura:

```
// Load Texture (once)
QImage img0("fieldstone.png");
QImage T = img0.convertToFormat(QImage::Format_ARGB32);
glGenTextures( 1, &textureId0);
glBindTexture(GL_TEXTURE_2D, textureId0);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, T.width(), T.height(), 0,
             GL_RGBA, GL_UNSIGNED_BYTE, T.bits());

...
// Bind textures, set uniforms...
g.glActiveTexture(GL_TEXTURE0);
g.glBindTexture(GL_TEXTURE_2D, textureId0);
program->bind();
program->setUniformValue("colorMap", 0);

...
```

## Al FS:

```
uniform sampler2D colorMap;
in vec2 vtexcoord;
...

vec4 color = texture(colorMap, vtexcoord);
...
```

## Filtrat

Magnification:  $\text{preimatge} < \text{texel}$ . Un píxel correspon a  $1/x$  texels ( $\partial u/\partial x = \partial v/\partial y = 1/x$ ,  $\partial v/\partial x = \partial u/\partial y = 0$ ) essent  $x$  el nombre de vegades que volem magnificar la imatge.

Minification:  $\text{preimatge} > \text{texel}$ . Un píxel correspon a  $x$  texels ( $\partial u/\partial x = \partial v/\partial y = x$ ,  $\partial v/\partial x = \partial u/\partial y = 0$ ) essent  $x$  el nombre de vegades que volem minificar la imatge.

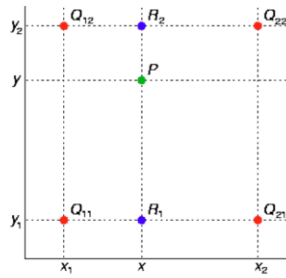
## Magnification filters:

Dues opcions: `GL_NEAREST` i `GL_LINEAR`. (`glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, [GL_NEAREST|GL_LINEAR])`)

**NEAREST:** Simplement, el color escollit és el color del veí més proper. Menys càlcul, però es noten les separacions entre píxels.

**LINEAR:** Interpolació bilineal. El color de la mostra és la mitjana ponderada dels colors dels quatre veïns.

## Bilinear interpolation



$$f(x, y) \approx \frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y_2 - y) + \frac{f(Q_{21})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y_2 - y) + \frac{f(Q_{12})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y - y_1) + \frac{f(Q_{22})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y - y_1).$$

## Minification filters

Cada textura tindrà diverses resolucions (Level of Detail o LOD). El 0 és el que més resolució té i, a partir d'allà, a més nombre, menys resolució. A això se li diu MIPMAPPING.

Es calcula un valor rho:  $f(\partial u / \partial x, \partial v / \partial x, \partial u / \partial y, \partial v / \partial y)$

I es calcula  $\lambda = \log_2(\rho)$ . Aquest serà el LOD escollit.

### Without mipmapping

- GL\_NEAREST // Nearest neighbor sampling on LOD 0
- GL\_LINEAR // Bilinear interpolation on LOD 0

### With mipmapping

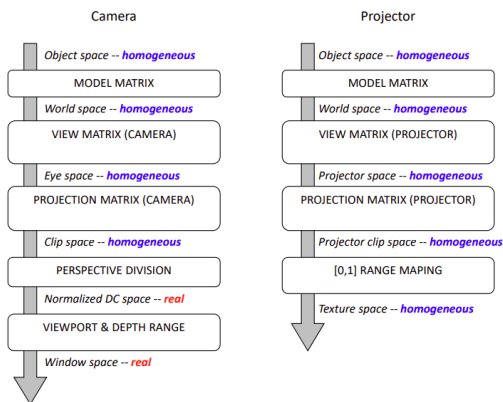
- GL\_NEAREST\_MIPMAP\_NEAREST // Nearest neighbor sampling on LOD  $\text{int}(\lambda)$
- GL\_LINEAR\_MIPMAP\_NEAREST // Bilinear sampling on LOD  $\text{int}(\lambda)$
- GL\_NEAREST\_MIPMAP\_LINEAR //  $c_0$  = nearest neighbor on LOD  $\text{int}(\lambda)$   
//  $c_1$  = nearest neighbor on LOD  $\text{int}(\lambda+1)$   
//  $\text{mix}(c_0, c_1, \text{fract}(\lambda))$
- GL\_LINEAR\_MIPMAP\_LINEAR //  $c_0$  = bilinear sampling on LOD  $\text{int}(\lambda)$   
//  $c_1$  = bilinear sampling on LOD  $\text{int}(\lambda+1)$   
//  $\text{mix}(c_0, c_1, \text{fract}(\lambda))$

## Perspective-correct interpolation

O bé interpolem (s,t) en object, world, eye o clip space, o interpolem (sw, tw, w) en window space, obtenint un texel (s, t, q). Per accedir a la textura: (s/q, t/q).



## Projective texture mapping



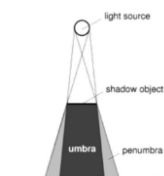
VS: Generem coords de textura:

- Passem el vèrtex de object a window space (viewport 1x1) però sense aplicar la divisió de perspectiva. Calculem (s, t, p, q) (p és la z respecte al projector i q és la homogènia).

$$\circ \begin{bmatrix} 1 & 0 & 0 & 0.5 \\ 0 & 1 & 0 & 0.5 \\ 0 & 0 & 1 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} P_p V_P M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} s \\ t \\ p \\ q \end{bmatrix}$$

FS: Accés a textura (s/q, t/q)

## 3. Ombres



Si la font de llum és puntual → no hi ha penombra.

Si augmenta la mida de la font de llum → Augmenta la penombra i disminueix la umbra.

Si apropem ocluser i receptor → Disminueix la penombra.

## Ombres per projecció (pla)

Per evitar z-fighting:

`glPolygonOffset(factor, units)`: Abans del depth test, es modifica el valor de la z del fragment amb l'equació:  $z' = z + \partial z \cdot \text{factor} + r \cdot \text{units}$ .

$$\partial z = \max(\partial z / \partial x, \partial z / \partial y)$$

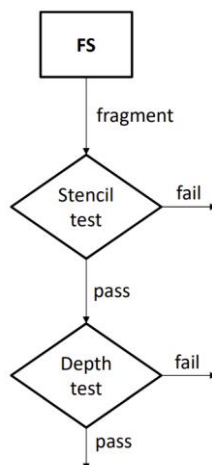
$r$  = valor més petit tal que garantitza un offset  $> 0$

Factor permet introduir un offset variable, i units un de constant.

```
void CastShadows::paintGL(...) {  
    // 1. Dibuixar receptor  
    phongShader->bind();  
    drawReceiver();  
  
    // 2. Dibuixar l'occludor projectat (ombra)  
    phongShader->setUniform("shadow",true);  
    phongShader->setUniform("modelMatrix",...);  
    glEnable(GL_POLYGON_OFFSET_FILL);  
    glPolygonOffset(-1, -1);  
    drawOccluder();  
  
    // 3. Dibuixar occludor  
    phongShader->setUniform("shadow",false);  
    phongShader->setUniform("modelMatrix",identity);  
    glDisable(GL_POLYGON_OFFSET_FILL);  
    drawOccluder();  
}
```



## Stencil Buffer:



El Stencil Buffer guarda, per cada píxel, un enter entre 0 i  $2^n - 1$ .

Demandar una finestra OpenGL amb stencil:

- `QOpenGLFormat f;`
- `f.setStencil(true);`
- `QOpenGLFormat::setDefaultFormat(f);`

Obtenir el número de bits del stencil:

- `glGetIntegerv(GL_STENCIL_BITS, &nbits);`

Esborrar stencil:

- `glClearStencil(0);`
- `glClear(GL_STENCIL_BUFFER_BIT);`

Escollir test de comparació:

- `glEnable(GL_STENCIL_TEST);`
- `glStencilFunc(comparació, valorRef, mask);`
  - o `comparació = GL_NEVER, GL_ALWAYS, GL_LESS...`
    - Ex: `GL_LESS: (valorRef & mask) < (valorStencil & mask)`

Operacions a fer a Stencil buffer segons el resultat del test:

- `glStencilOp(fail, zfail, zpass)`
  - o `fail` → op. a fer quan el fragment no passi el Stencil Test
  - o `zfail` → ídem però quan passi el Stencil però no el Depth.
  - o `zpass` → ídem però passa els 2 tests.
- Valors dels paràmetres:
  - o `GL_KEEP`: No facis res, deixa el valor que té al Stencil.
  - o `GL_ZERO`: Posa'l a 0.
  - o `GL_INCR`: Incrementa el valor en 1.
  - o `GL_DECR`: Decrementa el valor en 1.
  - o `GL_INVERT`: Inverteix el valor.
  - o `GL_REPLACE`: Substitueix el seu valor pel de referència.

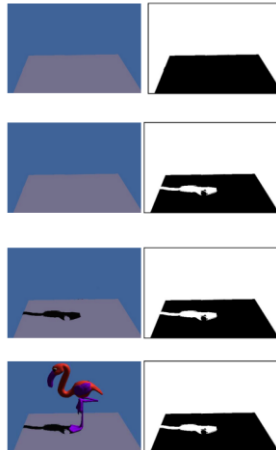
## Ombres per projecció (amb Stencil)

```
// 1. Dibuixa el receptor al color buffer i al stencil buffer
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
dibuixa(receptor);

// 2. Dibuixa ocluser per netejar l'stencil a les zones a l'ombra
glDisable(GL_DEPTH_TEST);
glColorMask(GL_FALSE, ... GL_FALSE);
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
glPushMatrix(); glMultMatrixf(MatriuProjeccio);
dibuixa(occludor);
glPopMatrix();

// 3. Dibuixa la part fosca del receptor
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
glColorMask(GL_TRUE, ... GL_TRUE);
glDisable(GL_LIGHTING);
glStencilFunc(GL_EQUAL, 0, 1);
Dibuixa(receptor);

// 4. Dibuixa l'occludor
glEnable(GL_LIGHTING);
glDepthFunc(GL_LESS);
glDisable(GL_STENCIL_TEST);
Dibuixa(occludor);
```



Matrius de projecció:

Donat un pla (a,b,c,d):

Respecte l'origen:

$$\begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ a & b & c & 0 \end{bmatrix}$$

Respecte punt (x,y,z):

$$\begin{bmatrix} -d - by - cz & xb & xc & xd \\ ya & -d - ax - cz & yc & yd \\ za & zb & -d - ax - by & zd \\ a & b & c & ax - by - cz \end{bmatrix}$$

En la direcció (x,y,z):

$$\begin{bmatrix} by + cz & -bx & -cx & -xd \\ -ay & ax + cz & -cy & -yd \\ -az & -bz & ax + by & -zd \\ 0 & 0 & 0 & ax + by + cz \end{bmatrix}$$

Volums d'ombra:

```
// 1. Dibuixa l'escena al z-buffer
glColorMask(GL_FALSE, ..., GL_FALSE);
dibuixa(escena);

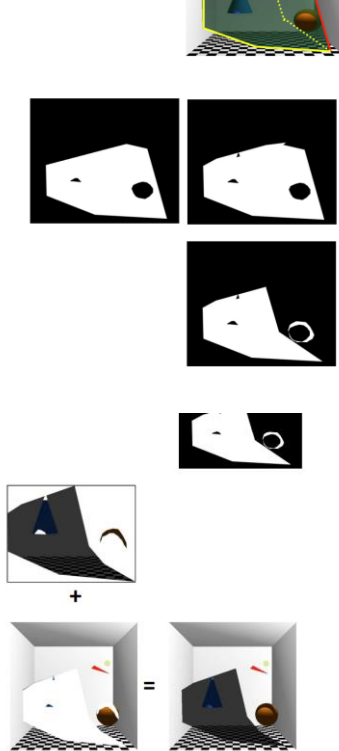
// 2. Dibuixa al stencil les cares frontals del volum
glEnable(GL_STENCIL_TEST);
glDepthMask(GL_FALSE);
glStencilFunc(GL_ALWAYS, 0, 0);
glEnable(GL_CULL_FACE);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
glCullFace(GL_BACK);
dibuixa(volum_ombra);

// 3. Dibuixa al stencil les cares posteriors del volum
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);
glCullFace(GL_FRONT);
dibuixa(volum_ombra);

// 4. Dibuixa al color buffer la part fosca de l'escena
glDepthMask(GL_TRUE);
glColorMask(GL_TRUE, ..., GL_TRUE);
glCullFace(GL_BACK);
glDepthFunc(GL_LEQUAL);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilFunc(GL_EQUAL, 1, 1);
glDisable(GL_LIGHTING);
dibuixa(escena);

// 5. Dibuixem al color buffer la part clara de l'escena
glStencilFunc(GL_EQUAL, 0, 1);
glEnable(GL_LIGHTING);
dibuixa(escena);

// 6. Restaura l'estat inicial
glDepthFunc(GL_LESS);
glDisable(GL_STENCIL_TEST);
```



## Shadow mapping:

### // Setup shadow map (un cop)

```
glActiveTexture(GL_TEXTURE0);
glGenTextures( 1, &textureId);
glBindTexture(GL_TEXTURE_2D, textureId);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32, SHADOW_MAP_WIDTH,
SHADOW_MAP_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
```

### // Pas 1. Actualització del shadow map

#### // 1. Definir càmera situada a la font de llum

```
glViewport( 0, 0, SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT );
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( fov, ar, near, far); // de la càmera situada a la llum!
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
gluLookAt( lightPos, ..., lightTarget, ..., up,...);
```

#### // 2. Dibuixar l'escena

```
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
glPolygonOffset(1,1); glEnable(GL_POLYGON_OFFSET_FILL);
drawScene();
glDisable(GL_POLYGON_OFFSET_FILL);
```

#### // 3. Guardar el z-buffer en una textura

```
glBindTexture(GL_TEXTURE_2D, textureId);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, SHADOW_MAP_WIDTH,
SHADOW_MAP_HEIGHT);
```

#### // Restaurar càmera i viewport

// Generació de coords de textura pel shadow map

// La generació és similar a projective texture mapping

glLoadIdentity();

glTranslated( 0.5, 0.5, 0.5 );

glScaled( 0.5, 0.5, 0.5 );

gluPerspective( fov, ar, near, far);

gluLookAt( lightPos, ... lightTarget, ... up...);

→ La matriu resultant és la que passa les coordenades del vertex (x,y,z,1) de *world space* a *homogeneous texture space* (s,t,p,q)

// VS

uniform mat4 lightMatrix;

out vec4 texCoord;

void main()

{

...

texCoord = lightMatrix\*vec4(vertex,1);

gl\_Position = modelViewProjectionMatrix \* vec4(vertex,1);

}

// FS

...

vec2 st = texCoord.st / texCoord.q;

float trueDepth = texCoord.p / texCoord.q;

float storedDepth = texture(shadowMap, st).r;

float bias = 0.01; // només si no hem usat glPolygonOffset

if (trueDepth - bias <= storedDepth)

fragColor = ... // il·luminat

else

fragColor = ... // a l'ombra

#### 4. Reflexions especulars:

Vector reflectit

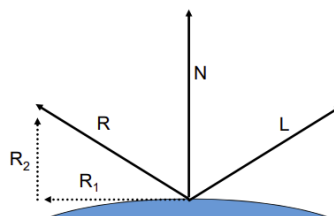
$$R = R_1 + R_2$$

$$R_1 = -L + R_2$$

$$R = 2R_2 - L$$

$$R_2 = (N \cdot L)N$$

$$R = 2(N \cdot L)N - L$$



## Reflexions amb objectes virtuals

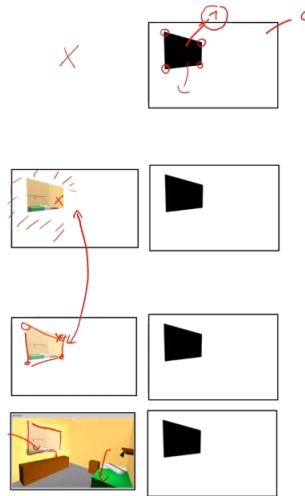
Concepte: Dupliquem l'escena, la invertim i la posem a l'altre costat del mirall.

```
// 1. Dibuixem el mirall a l'stencil buffer
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glDepthMask(GL_FALSE); glColorMask(GL_FALSE...);
dibuixar(mirall);

// 2. Dibuixem els objectes virtuals
glDepthMask(GL_TRUE); glColorMask(GL_TRUE...);
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glPushMatrix(); glMultMatrix(matriu simetria);
glLightfv(GL_LIGHT0, GL_POSITION, pos);
glCullFace(GL_FRONT);
dibuixar(escena);
glPopMatrix();

// 3. Dibuixem el mirall semitransparent
glDisable(GL_STENCIL_TEST);
glLightfv(GL_LIGHT0, GL_POSITION, pos);
glCullFace(GL_BACK);
dibuixar(mirall);

// 4. Dibuixem els objectes reals
dibuixar(escena);
```



## Matriu de reflexió:

Matriu de reflexió respecte un pla (a,b,c,d):

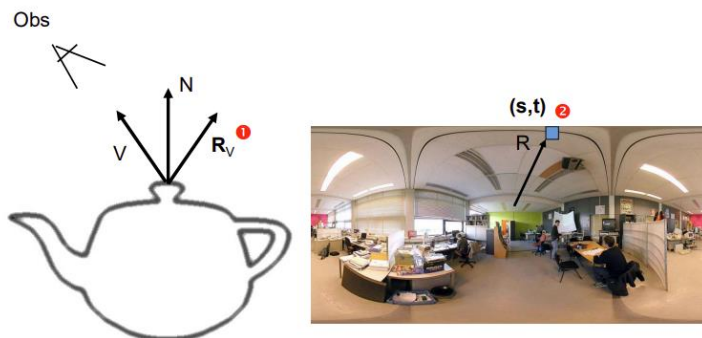
$$\begin{bmatrix} 1-2a^2 & -2ba & -2ca & -2da \\ -2ba & 1-2b^2 & -2cb & -2db \\ -2ca & -2cb & 1-2c^2 & -2dc \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Environment mapping:

Environment map: Fotografia de 360°. Donat una direcció arbitrària R, ens retornarà el color de l'entorn en aquesta direcció:

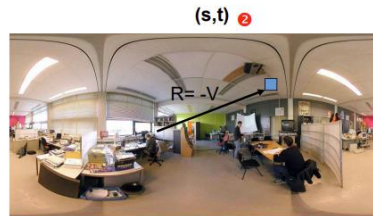
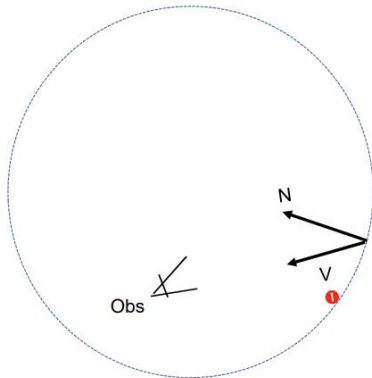
Color = environmentMap(R).

Ús per reflexions especulars:



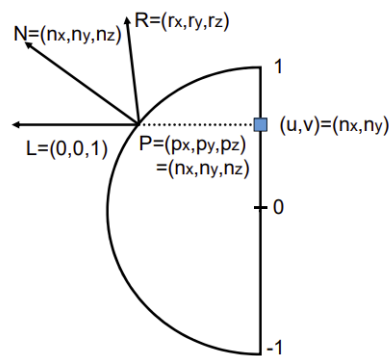
A GLSL podem usar `vec3 reflect(-V, N)` per calcular  $R_v$ .

Ús com a entorn:



Sphere Mapping:

$$R = (2n_z n_x, 2n_z n_y, 2n_z^2 - 1)$$



```
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);
    return texture(sampler, st);
}
```



## Cube Mapping:

### // 1. Creació de les sis textures

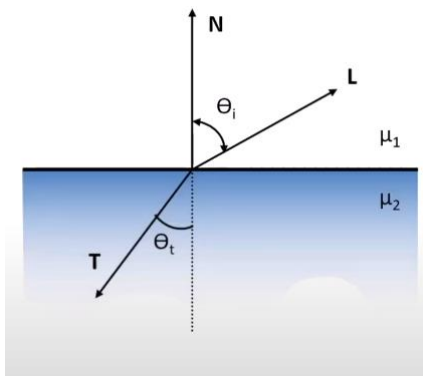
```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT, ...);  
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, ...);  
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, ...);  
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, ...);  
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, ...);  
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT, ...);
```

```
uniform sampler2D sampler;  
...  
fragColor = texture(sampler, vtexCoord);
```

```
uniform samplerCube samplerC;  
...  
vec3 R;  
...  
fragColor = textureCube(samplerC, R);
```

## 5. Objectes translúcids:

### Refracció:



<b>Buit</b>	<b>1.0</b>
<b>Aire</b>	<b>1.0003</b>
Gel	1.31
<b>Aigua a 20° C</b>	<b>1.33</b>
Alcohol	1.36
<b>Cristall</b>	<b>1.52</b>
Safir	1.77
<b>Diamant</b>	<b>2.417</b>

### Llei de Snell:

- N, L i T són coplanars.
- $\sin(\theta_t) = \frac{\mu_1}{\mu_2} \sin(\theta_i)$
- Si  $\mu_2 > \mu_1$ , 2 és més dens que 1. T s'apropa a N. ( $\theta_t$  decreix)
- Si  $\mu_2 < \mu_1$ , 2 és menys dens que 1. T s'allunya de N. ( $\theta_t$  creix)

Equacions de Fresnel:

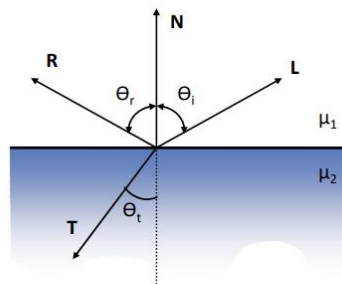
Assumim que:

- El comportament és especular pur.
- No hi ha absorció de llum.  $R+T = 1$ .
- Material no conductor, dielèctric.

$$R = \frac{R_s + R_p}{2}$$

$$R_s = \left( \frac{\sin(\theta_t - \theta_i)}{\sin(\theta_t + \theta_i)} \right)^2$$

$$R_p = \left( \frac{\tan(\theta_t - \theta_i)}{\tan(\theta_t + \theta_i)} \right)^2$$



$$T = 1 - R.$$

Aproximació de Schlick:

$$R = f + (1 - f)(1 - L \cdot N)^5$$

$$f = \frac{(1 - \mu)^2}{(1 + \mu)^2}$$

$$\mu = \frac{\mu_1}{\mu_2}$$

Alpha Blending:

Mètode back-to-front. Primer fem el que hi ha darrere, i anem cap al front.

```
glEnable(GL_BLEND);
```

```
(1) glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```

```
(peso_srcColor, peso_destinColor)
```

```
(2) glBlendFunc(GL_SRC_ALPHA, GL_ONE)
```

```
color = s_factor*src_color(r, g, b, a) + d_factor*dst_color
```

En el cas de (1):  $\text{color} = a_s \cdot \text{src\_color} + (1 - a_s) \cdot \text{dst\_color}$

En el de (2):  $\text{color} = a_s \cdot \text{src\_color} + \text{dst\_color}$

## 6. II·luminació global

Radiometria:

Sím.	Radiomet.	Fotometria	Definició	Ús
$\Phi$	Fluxe (W)	Fluxe (lm)	Energia que travessa una superfície per unitat de temps	Energia total que emet una font de llum
E	Irradiància ( $\text{W}/\text{m}^2$ )	Iluminància ( $\text{lux} = \text{lm}/\text{m}^2$ )	Fluxe per unitat <b>d'àrea</b>	Llum que incideix en un punt, des de qualsevol direcció
I	Intensitat ( $\text{W}/\text{sr}$ )	Intensitat ( $\text{cd} = \text{lm}/\text{sr}$ )	Fluxe per unitat <b>d'angle sòlid</b>	Distribució direccional d'una llum puntual
L	Radiància ( $\text{W}/(\text{sr} \cdot \text{m}^2)$ )	Luminància ( $\text{cd}/\text{m}^2$ )	Fluxe per unitat <b>d'àrea</b> i unitat <b>d'angle sòlid</b>	Energia que travessa un punt en una determinada direcció

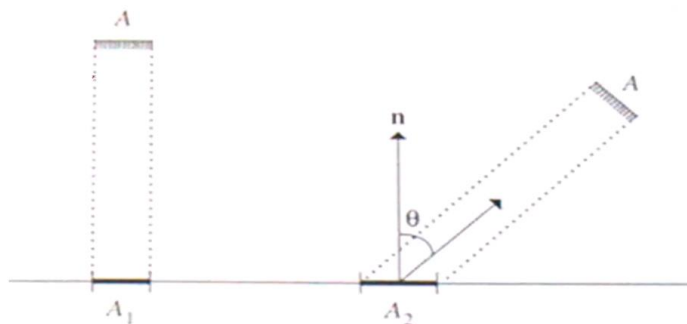
Flux Radiant ( $\phi$ ):

Quantitat d'energia que travessa una superfície per unitat de temps ( $E/t$ )  $\rightarrow$  (W) o Lumens en fotometria.

Irradiància (E):

Densitat de flux radiant. Flux radiant per unitat d'àrea.  $\text{W}/\text{m}^2$  o Lux (lx) en fotometria.

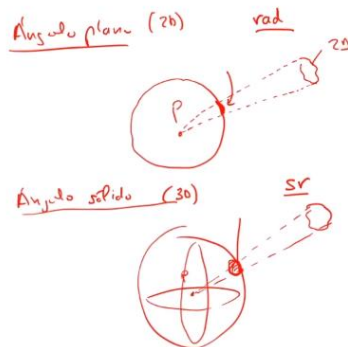
Llei de Lambert:



$$A_1 = A; E = \frac{\phi}{A} \quad A_2 = \frac{A}{\cos(\theta)}; E = \frac{\phi}{A_2} = \frac{\phi}{A} \cos(\theta) = \frac{\phi}{A} (N \cdot L)$$

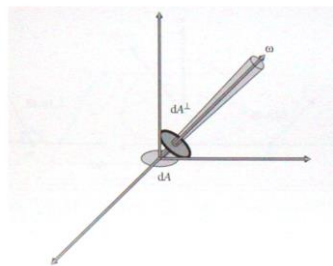
Intensitat (I):

Flux per unitat d'angle sòlid. Unitats: W/sr. Sr és a l'interval  $[0, 4\pi]$ .



Radiància (L):

Flux per unitat d'àrea i d'angle sòlid.



$$L = \frac{d^2\phi}{d\omega dA^\perp}$$

BRDF (Bidirectional Reflectance Distribution Functions)

És una funció  $f(p, \omega_o, \omega_i)$ . Per a un punt qualsevol d'una superfície i una direcció d'entrada i una altra de sortida determinades (que no tenen per què ser coplanars), BRDF ens retorna, de la quantitat de la radiància que incideix en p en la direcció  $\omega_i$ , la que en surt per  $\omega_o$ .

$$f(p, \omega_o, \omega_i) = \frac{L_o(p, \omega_o)}{L_i(p, \omega_i) \cdot \cos(\theta_i)}$$

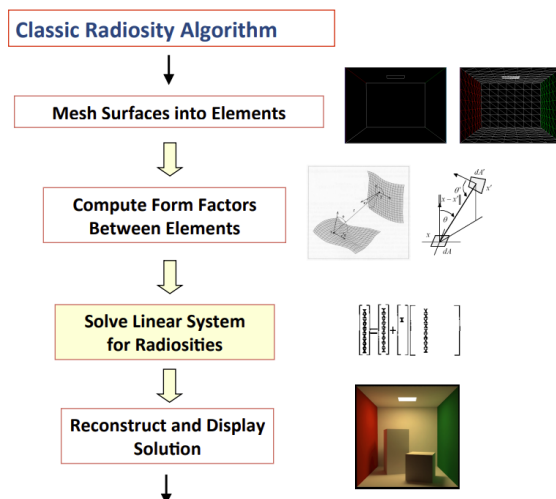
Propietats:

- $f(p, \omega_o, \omega_i) \geq 0$ .
- $f_r(p, \omega_o, \omega_i) = f_r(p, \omega_i, \omega_o)$ .
- $\int_{\Omega} f_r(p, \omega_o, \omega') \cdot \cos(\theta) d\omega' \leq 1$  (Consevació de l'energia)

Una BRDF analítica és la de Phong, o la de Lambert.

## Radiosity:

- Suposem que totes les superfícies i fonts de llums són difuses i tenen un comportament uniforme en tota la superfície.
- Subdividim en pedaços per a mantenir uniformitat.
- Versió discretitzada de l'equació de Rendering:
  - $B_i = E_i + \rho_i \sum B_j F_{ij}$
  - On:
    - $B_i$  és la radiositat de la superfície i
    - $E_i$  és l'emissivitat de la superfície i
    - $\rho_i$  és la reflectivitat de la superfície i
    - $B_j$  és la radiositat de la superfície j
    - $F_{ij}$  és el factor de forma de la superfície j relatiu a la superfície i (relació física entre ambdues).



## 7. Ray Tracing

Tenim el Ray Tracing clàssic, el Path Tracing, Distributed Ray Tracing i Two-Pass Ray Tracing.

Camins suportats per il·luminació local: LDE, LSE.

Camins suportats per RT clàssic:  $LDS * E$  (\* = diversos)

Camins suportats per Path Tracing:  $L(D|S) * E$

Camins suportats per Distributed RT:  $LDS * E$  (però la S pot simular reflexions especulars perfectes i imperfectes).

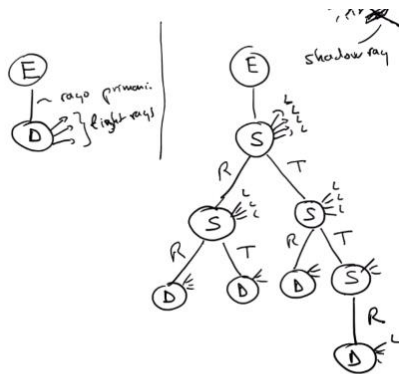
Camins suportats per Two-pass RT:  $LS*DS*E$  ( $LS*D$  light pass i  $DS*E$  eye pass)

### Ray-Tracing clàssic

L'observador llença rajos cap a l'escena. El primer raig s'anomena "raig primari". Es forma un arbre amb els rajos llençats. És com un Ray casting però recursiu.

Un cop un raig impacta amb una superfície, llancem un Shadow Ray cap a cada font de llum per veure si hi ha algun objecte opac que s'interposi entre el focus de llum i el punt. (superfície difusa).

Si la superfície és especular, el raig rebota en ella. Això fa que, un cop el raig impacta, tinguem de sortida un raig reflectit i un de transmès. Això fa que, en cas que els rajos no parin de col·lisionar amb superfícies especulars, es vagin creant en forma d'arbre més i més, de forma recursiva. Per cada hit, llancem també un Shadow Ray per cada font de llum.



Problema? No suporta gaires camins diferents.

### Path Tracing:

Per cada píxel es llancen  $R$  rajos. Si fa hit, llençarem un altre raig (que pot ser transmès o reflectit, depenent si estem amb una superfície Especular o Difusa). Cada hit també comporta llençar els Shadow Rays. Això ho fem fins arribar a un nombre màxim de rebots.

Problema? L'estimador té massa variància, fet que fa que les imatges tinguin molt soroll.

### Distributed Ray Tracing:

Llancem un raig primari a partir de l'observador i si el hit el fa amb una difusa, s'acaba.

Si fa hit amb una especular, aquest cop llancem múltiples rajos des d'aquesta, tots ells reflexions.

Two-pass Ray Tracing:

Pas 1: Light Pass: Llancem rajos però des de la font de llum. Mentre anem trobant-nos superfícies especulars, seguim llençant rajos. Quan arribem a una superfície difusa, emmagatzemem "l'energia" que porta el "fotó" del raig i la superfície en qüestió.

Pas 2: Eye Pass: Per cada píxel, llenço un raig des de l'observador i, quan arribo a una difusa, a part de llençar els Shadow Rays, miro quina energia tenia emmagatzemada en punts propers al de col·lisió.

Ray Tracing Clàssic en Detall:

Color d'un punt P:

$$I(P) = I_D(P) + I_R(P) + I_T(P) \text{ on:}$$

$I_D(P)$  és el color degut a la llum DIRECTA del focus.

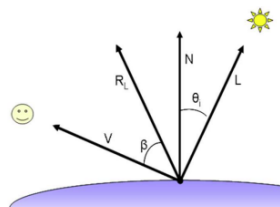
$I_R(P)$  és el color degut a la llum INDIRECTA que es reflecteix a P en direcció cap a l'observador.

$I_T(P)$  és el color degut a la llum INDIRECTA que es transmet a P en direcció cap a l'observador.

$$I(P) = I_D(P) + I_R(P) + I_T(P)$$

$$I_D(P) = K_a I_a + K_d \sum I_L \cos(\theta_i) + K_s \sum I_L \cos^n(\beta)$$

- $\cos(\theta_i) = N \cdot L$
- $\cos(\beta) = R \cdot V$
- El sumatori només considera les fonts de llum no ocluides (ombres)



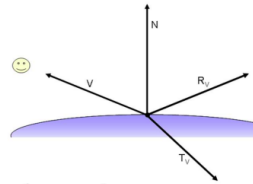
$$\cos(\theta_i) = N \cdot L$$

$$\cos^n(\beta) = (R \cdot V)^n$$

$$I(P) = I_D(P) + I_R(P) + I_T(P)$$

$$I_R(P) = K_R L_R$$

$$I_T(P) = K_T L_T$$



- $K_R$   $K_T$  coeficients empírics de reflexió/transmissió especular
- $L_R$  = llum que incideix en P en la direcció  $R_v$
- $L_T$  = llum que incideix en P en la direcció  $T_v$

Es calculen recursivament, traçant un nou raig reflectit i un altre transmès

Algorisme:

```
acció rayTracing
  per i en [0..w-1] fer
    per j en [0..h-1] fer
      raig:=raigPrimari(i, j, camera);
      color:=traçarRaig(raig, escena, μ);
      setPixel(i, j, color);
    fper
  fper
facció
```

```
funció traçar_raig(raig, escena, μ)
  si profunditat_correcta() llavors
    info:=calcula_interseccio(raig, escena)
    si info.hi_ha_interseccio() llavors
      color:=calcular_I0(info,escena); // I_D
      si es_reflector(info.obj) llavors
        raigR:=calcula_raig_reflectit(info, raig)
        color+= K_R*traçar_raig(raigR, escena, μ) //I_R
      fsi
      si es_transparent(info.obj) llavors
        raigT:=calcula_raig_transmès(info, raig, μ)
        color+= K_T*traçar_raig(raigT, escena, info.μ) //I_T
      fsi
    sino color:=colorDeFons
  fsi
  sino color:=Color(0,0,0); // o colorDeFons
  fsi
  retorna color
ffunció
```

Intersecció Raig-Geometria:

El test d'intersecció té 2 possibles resultats:

- La recta no intersecciona el poliedre.
- La recta intersecciona 2 cops la superfície del poliedre: Una d'entrada i una de sortida.

Les interseccions d'entrada sempre són amb plans front-face respecte la direcció del raig:

$$V \cdot N < 0.$$

Les de sortida són back-face respecte la direcció del raig:  $V \cdot N > 0$ .

L'algorisme de Haines calcula la intersecció d'entrada i de sortida.

Per cada pla  $\pi$  del poliedre:



- Calculem la intersecció de la recta amb el pla.
- Actualitzem  $\lambda_{\text{near}}$  o  $\lambda_{\text{far}}$ .
- Si  $\lambda_{\text{near}} > \lambda_{\text{far}}$  no tenim intersecció.