

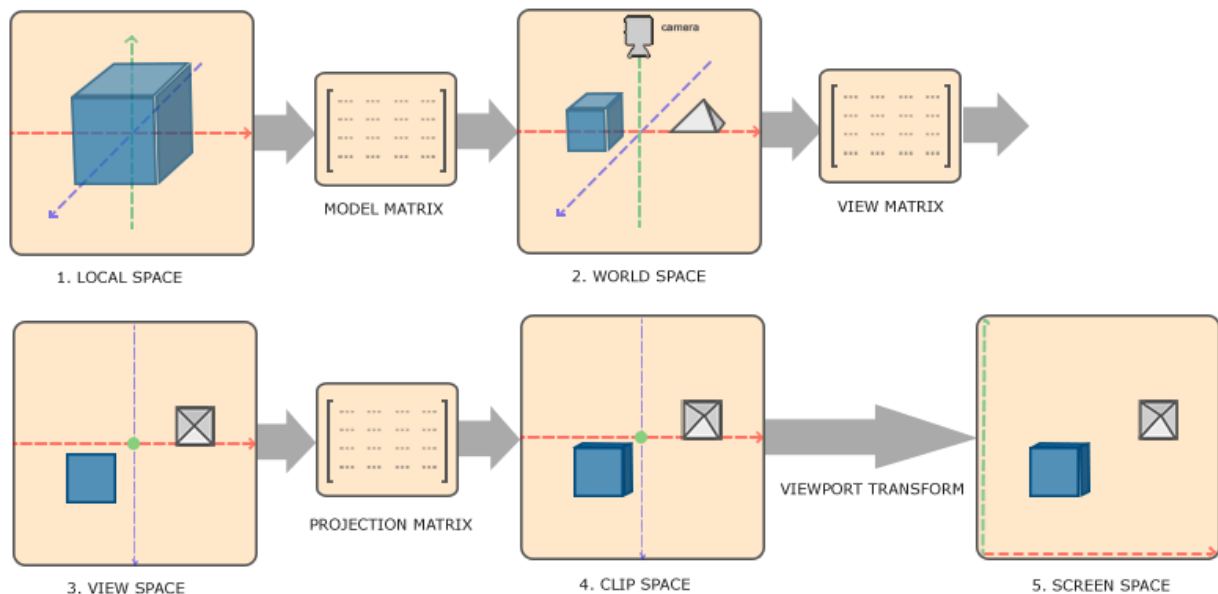
Apunts G

Sistemes de Coordenades (punts)

[Sistemes de Coordenades \(punts\)](#) (1h 6')

Fins arribar a **window space** les coordenades són homogènies, tota coordenada xyz ve acompanyada d'una quarta coordenada w que representa la profunditat, aquesta s'anomena component homogènia. En cas de tenir un punt en coordenades homogènies i es vol obtenir el punt en 3D s'ha de dividir les coordenades x, y i z per la component homogènia w. És a dir el punt en coordenades homogènies (x, y, z, w), en coordenades 3D seria el punt (x/w, y/w, z/w). w ha de ser > 0. En cas de representar vectors, la component homogènia s'igual a 0 i ja està.

Per entendre millor les diferents transformacions i els diferents sistemes de coordenades (Extret de [Learn OpenGL](#)).



Per passar de **object space** a **world space** es fan les transformacions de modelat (modeling transform), on es realitzen les translacions, escalats i rotacions necessàries. Sempre en el ordre següent (modelMatrix): $T \cdot R \cdot S \cdot (x, y, z, 1)$

Tota la resta, és igual, recordar una mica IDI i tal. [Diapos](#)

Introducció a la il·luminació realista

[Introducció a la il·luminació realista](#) (19')

Physically-based rendering

Il·luminació el més semblant a la realitat possible. Intentar simular la interacció de la llum amb les superfícies amb la màxima fidelitat a la física possible.

Photorealistic rendering

Intentar aconseguir una aparença fotorealista. Ombres aproximades, aprofitar limitacions del sistema visual humà (HVS, Human visual system)

Non-Photorealistic rendering (NPR)

Estilo cartoon

Limitacions HVS

- Efectes visuals
- Càustiques (Reflex de la llum en un contenidor de vidre amb aigua)
- Utilitzar imatges com a reflex de certes superfícies (Environment mapping).

Elements de realisme

II·luminació

- Ombres
- Reflexions especulars
- Translucidesa

Detalls

- Geomètrics
- Material
- Color

Subsurface scattering

Penetració de certs rajos de llum en la superfície on incideix (sobretot es nota en superfícies com la pell).

Emissió, reflexió, transmissió

Un fotó prové d'una font d'emissió. Depenent del material on incideix el raig, es pot produir absorció, transmissió o reflexió. En el cas del vidre, alguns dels fotons que hi incideixin es reflectaran i altres es transmetran (passar a través del material). Quan el fotó canvia de medi és transmissió i quan es manté en el mateix és reflexió. Depenent de l'angle amb el que incideix i haurà més o menys especulació. L'angle de reflexió és el mateix que l'incident.

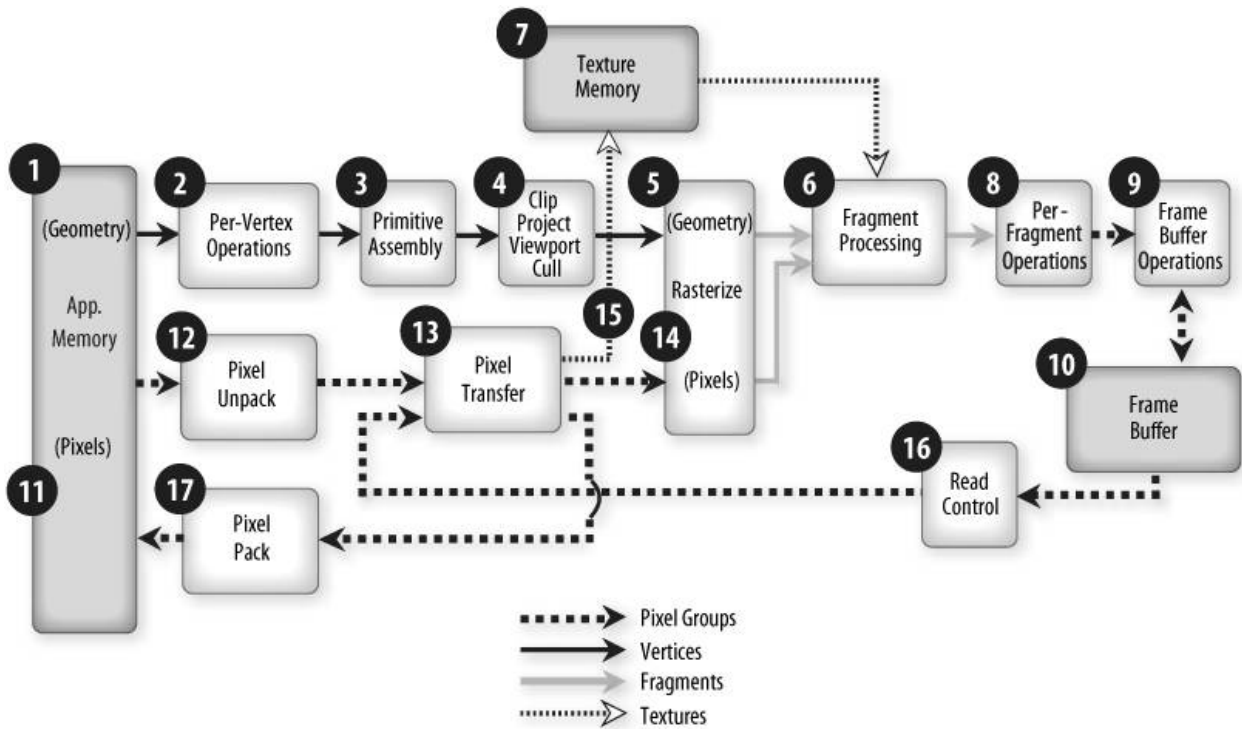
Aplicacions dels gràfics

[Aplicacions dels gràfics](#) (11')

A la [pàgina web de l'assignatura](#) hi ha 3 articles que s'han de llegir per l'examen final ja que hi haurà un petit qüestionari per evaluar les competències transversals.

Pipeline OpenGL

Pipeline OpenGL (58')



1. Geometry. **OBJECT SPACE**

Dibuix de primitives (punts, línies, polígons...)

- Vèrtex a vèrtex: *glBegin*, *glVertex*, *glEnd*(compatibility profile)
- Vertex array object: *glDrawArrays*, *glDrawElements*...

2. Vertex Shader **OBJECT SPACE -> CLIP SPACE**

Depenent de la il·luminació que es vulgui fer, aquest rebrà una informació o una altre.

- Color per vèrtex (no hi ha il·luminació, escàner 3D): vertex, color
- Il·luminació per vèrtex: vertex, normal
- Il·luminació per fragment: vertex, normal

Depenent de la il·luminació que es vulgui fer, aquest enviarà una informació o una altre.

- Color per vèrtex (no hi ha il·luminació, escàner 3D): el mateix color que reb (*frontColor*), i *gl_Position*
- Il·luminació per vèrtex: *frontColor* (amb la il·luminació), *gl_Position*
- Il·luminació per fragment: P (coordenades en eye space), N (normal en eye space)

3. Primitive Assembly **CLIP SPACE**

- Els vèrtex s'agrupen en primitives
- Cada primitiva requereix un clipping diferent.

4. Primitive processing **CLIP SPACE -> WINDOW SPACE**

- Clipping (retallat) a la piràmide de visió. Es retallen els elements que sobresurten del camp de visió, eliminant la part del element que queda fora i reconstruint (afegint nous vèrtex) el element.

- Divisió de perspectiva: es divideix (x,y,z) per w . Aconseguir la sensació de profunditat.
- Viewport (x,y) & depth tranform (z) -> window space.
- Backface culling - `glEnable(GL_CULL_FACE)`. Eliminar les cares traseras, les que no es veuen. Per defecte està deshabilitat.

5. Rasterització **WINDOW SPACE**

- Generació dels fragments corresponents a la primitiva retallada. Es generen els fragments on el centre del píxel està a dins de la primitiva.
- Cada fragment té usualment diversos atributs:
 - Coordenades (window space)
 - Color (interpolat)
 - Coordenades de textura (interpolat)

6. Fragment Shader **WINDOW SPACE**

Calcula el color del fragment.

En cas de fer la il·luminació per fragment, també es calcula. Més detall, quants menys fragments més es nota la diferencia amb calcular la il·luminació al vertex shader.

7. Per-Fragment operations ("raster operations") **WINDOW SPACE**

- Pixel ownership test
 - Stencil test
 - Depth test (test Z-buffer)
 - Blending
- Ara ja tenim el color definitiu del fragment.

8. Frame buffer operations **WINDOW SPACE**

- Es modifiquen els buffers que s'hagin escollit amb `glDrawBuffers`
- Es veu afectada per `glColorMask`, `glDepthMask`

9. Frame Buffer **WINDOW SPACE**

Dibuixa els buffers.

Exercici

Donats 4 elements, ordenar:

- `glDrawElements` -> Perspective Division -> Raster -> Stencil
- `glBufferData` -> Write to `gl_Position` -> Raster -> Alpha blending
- Clipping -> Perspective Division -> Raster -> Fragment shader
- Raster -> Fragment Shader -> Stencil test -> Depth test
- Geometry shader -> Raster -> Fragment Shader -> Depth test
- Vertex shader -> Clipping -> Perspective Division -> Raster
- Geometry shader -> Raster -> Fragment Shader -> Stencil test
- Vertex shader -> Perspective Division -> Raster -> `dFdx`, `dFdy` (Operació Fragment Shader)
- Geometry shader -> Raster -> Stencil Test -> Alpha Blending

- Pas a clip space -> Perspective Division -> Raster -> Fragment shader
- glDrawElements -> Pas a Clip Space -> Perspective Division -> Raster

Tots els elements ja ordenats: **IMPORTANT, SEMPRE ENTRA**

glBufferData -> glDrawElements -> Vertex shader -> Write to gl_Position -> Clipping -> Perspective Division -> Geometry shader -> Raster -> Fragment Shader -> Stencil Test -> Depth Test/Alpha blending

Exemple Exercici: Abans o després de raster?

Abans d'un raster:

- Vertex Shader
- Geometry Shader
- Pas a NDC (Normalized Device Coordinates)
- Pas de les coordenades a clip space
- Primitive assembly

Després d'un raster:

- Depth Test
- discard (Fragment Shader)
- Processament del fragment
- Ús de les funcions dFdx, dFdy

Abans & Després d'un raster:

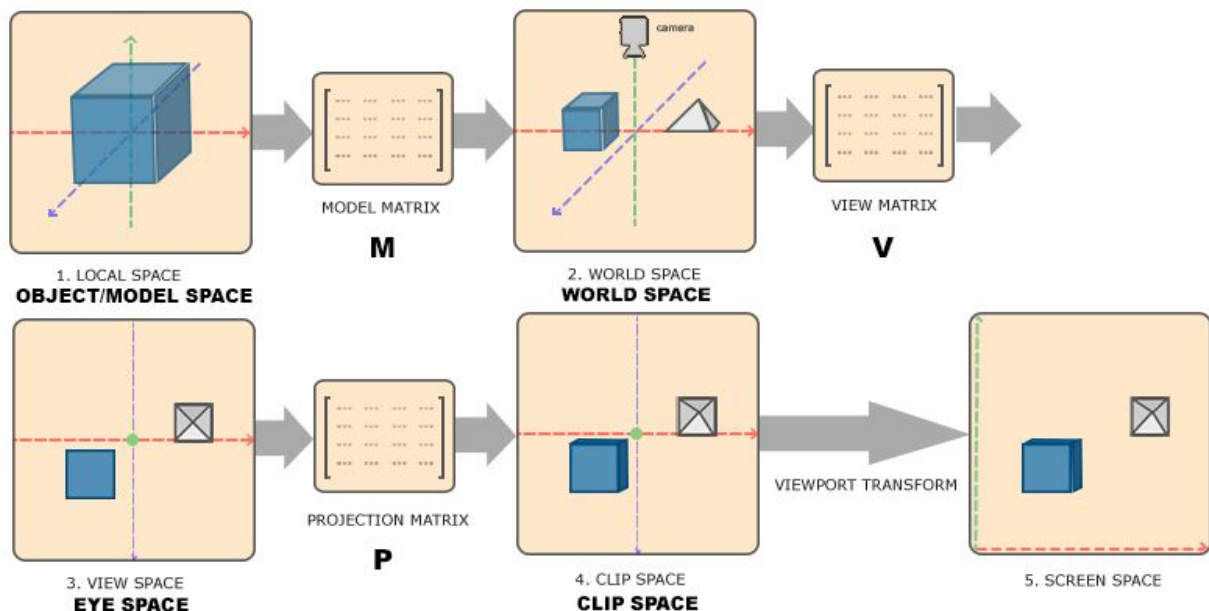
- Operacions que es poden fer en un Vertex o Fragment Shader (Càlculs d'il·luminació; generació de coordenades de textura (sphere mapping...))

dFdx, dFdy: l'espai de coordenades x, y haurà d'estar en Window space.

Matriu/Productes de matrius

SEMPRE ENTRA, Sistemes de Coordenades Origen->Destí

Vèrtex de object space a model space	I
Vèrtex de object space a world space	M
Vèrtex de object space a clip space	$P * V * M$
Vèrtex de world space a eye space	V
Vèrtex de world space a clip space	$P * V$
Vèrtex de eye space a world space	V^{-1}
Vèrtex de eye space a clip space	P
Vèrtex de clip space a object space (ModelViewProjectionMatrixInverse)	$M^{-1} * V^{-1} * P^{-1}$
Vèrtex de clip space a world space	$V^{-1} * P^{-1}$
Normal de model space a eye space	N
Normal de object space a eye space	N



`vec4 P = projectionMatrix * viewMatrix * modelMatrix * vec4(vertex, 1.0);`

`vec3 N = normalMatrix * modelViewMatrix * vec4(normal,0);`

Exercici: Fer la conversió de vec4 P de clip space a object space (assumint que no hi ha transformació de modelat):

`gl_ModelViewProjectionMatrixInverse * P;`

->

`gl_ModelViewMatrixInverse * gl_ProjectionMatrixInverse * P;`

Exercici: Escriu quina matriu 4x4 necessitem per simular la reflexió de l'escena respecte un mirall situat al pla $Y=0$. (Solució: Matriu identitat, on es posa negativa la coordenada que es vol inversa)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Exercici: La projecció ortogonal sobre el pla (a,b,c,d) és:

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

Exercici: La reflexió respecte un pla (a,b,c,d) és:

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

Textures

Introducció

Textures - introducció (31')

Útils per representar detalls superficials deguts a variacions de la superfície i a variacions de les propietats òptiques (reflexivitat o color difús).

Una textura és una taula de 1, 2 o 3 dimensions on cada cel·la (texel) emmagatzema una certa propietat amb 1, 2, 3 o 4 canals. Habitualment s'utilitzen al FS (Fragment Shader).

- Les textures 1D són aquelles que estan formades per un sol vector i que bàsicament és un gradient de color.
- Les textures 2D són aquelles formades per un bitmap, una taula de dues dimensions i s'utilitzen amb dues variables (s,t). Una imatge d'unes pedres o algo així.
- Les textures 3D es treballen amb tres variables (s,t,p). Un cub amb una textura aplicada. Les coordenades (x,y,z) del model es podrien utilitzar com coordenades de la textura.
- Kd (color map): Color del objecte (Coeficiente de reflexivitat difusa del material).
- Ks (gloss map): Reflexivitat especular del material (Una textura d'un metall oxidat).
- Opacitat (opacity map, alpha mask): Ajuda a determinar quines zones són més transparents.
- Normal (normal mapping): S'utilitzen les components dels vectors normals d'una textura com si fossin components d'un color, x,y,z passen a ser R, G, B.
- Desplaçament(Bump mapping): Com el normal mapping però en lloc d'utilitzar 3 canals (R,G,B), utilitza 1.

Mida d'una textura: # texels en cada dimensió. Habitualment w, h són potència de 2.

Espai normalitzat de textura: Les coordenades de textura són s i t, que van de 0 a 1. (0,0) abaix a l'esquerra i (1,1) a dalt a la dreta.

Mapping

Textures - Mapping (21')

Forward mapping: Funcions que diuen a cada texel on va en espai de pantalla. Textura -> Imatge

Inverse mapping: Funcions que diuen de quin texel ha d'agafar el seu color cada punt de la pantalla. Per OpenGL és més útil. Imatge -> Textura

Parametrizació d'una superfície: Utilitzar forward mapping per mappejar un element fàcilment parametrizable (cilindres, esferes, plànols...).

Mapping esfèric

Input: $s \in [0,1], t \in [0,1]$
Output: $x,y,z \in$ esfera unitat

```
// pas (s, t) → (Θ, Ψ)  
Θ = 2πs;  
Ψ = π(t-0.5);
```

```
// pas esfèriques → (x,y,z)  
x = sin(Θ)cos(Ψ);  
y = sin(Ψ);  
z = cos(Θ)cos(Ψ);
```

Mapping cilíndric

Input: $s \in [0,1], t \in [0,1]$
Output: $x,y,z \in$ cilindre $r=1$ sobre pla XZ

```
// pas (s, t) → (Θ, h)  
Θ = 2πs;  
h = t;
```

```
// pas cilíndriques → (x,y,z)  
x = sin(Θ);  
y = h;  
z = cos(Θ);
```

On generar coordenades de textura

[Textures - On generar coordenades de textura \(8'\)](#)

Es poden generar:

1. En temps de modelat (Amb una eina de modelat, per exemple Blender)
Li passa a l'aplicació VBOs per cada vertex i aquesta li passa al VS les coordenades s,t amb el `vec2 texCoord`.
2. A la CPU, per l'aplicació (Geometry)
No se li importa cap model i les calcula per passar-li al VS el `texCoord`.
3. Al vertex shader
A partir de vertex i la normal utilitza alguna funció per generar les coordenades de textura s,t i passar-li al FS amb el nom de `vertexCoord`.
4. Al fragment shader
El VS li passa el vertex i la normal sota els noms P i N (per exemple) i aquest les calcula.

Per textures complexes és millor generarles lo abans possible.

De vegades l'etapa de rasterització fa necessari que les coordenades de textura en generin en un moment específic.

Generació de coords de textura amb plans S, T

[Textures - Generació de coords de textura amb plans S, T \(20'\)](#)

L'entrada és un vèrtex $(x,y,z,1)$. Els plans S i T els representem com $S=(a_s, b_s, c_s, d_s)$ i $T=(a_t, b_t, c_t, d_t)$.

Si la normal d'un pla qualsevol està definida com $n=(a,b,c)$ l'equació $ax+by+cz+d=0$ defineix el pla en qüestió.

Per obtenir les coordenades de textura s i t es fa el següent, $s = S \cdot \text{vèrtex}$ i $t = T \cdot \text{vèrtex}$, per tant:

- $s = (a_s x, b_s y, c_s z, d_s w)$
- $t = (a_t x, b_t y, c_t z, d_t w)$

Generació de coordenades amb superfície auxiliar i Parametrizació

[Textures - Generació de coordenades amb superf. param.](#) (11')

S'aporfita la facilitat de parametritzar certes superfícies (esferes, cilindres, plànols...). La idea és utilitzar una superfície auxiliar que s'assembli (una mica) a la superfície que es vol parametritzar. Donat un vèrtex qualsevol del model, s'utilitzarà la funció *O-mapping* per associar-lo a un punt de la superfície auxiliar i després s'utilitzarà la inversa de *S-mapping*.

Creació de texture objects

[Textures - Creació de texture objects](#) (5')

Carregar una textura

```
QImage img0("fieldstone.png"); //imatge que es vol utilitzar
QImage T = img0.convertToFormat(QImage::Format_ARGB32); //Convertir la imatge a un format compatible amb OpenGL
glGenTextures( 1, &textureId0);
glBindTextures(GL_TEXTURE_2D, textureId0);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, T.width(), T.height(), 0, GL_RGBA, GL_UNSIGNED_BYTE, T.bits());
```

1. **GL_TEXTURE_2D**: On es guardarà la textura creada
2. **0**: Nivell de detall
3. **GL_RGB**: Format en el que es construirà la textura
4. **T.width()**: Amplada de la imatge
5. **T.height()**: Alçada de la imatge
6. **0**: Marge de la imatge
7. **GL_RGBA**: Determina quantes components es trobarà al buffer
8. **GL_UNSIGNED_BYTE**: Interpreta cada element de color com un enter sense signe (0 - 255)
9. **T.bits()**: Apuntador al buffer amb tots els bits de la imatge, el bitmap

Filtrat de textures

[Textures - Filtrat](#) (29')

- Magnification = Upsampling => Preimatge < texel
- Minification = Downsampling => Preimatge > texel
- Preimatge d'un fragment: : És la regió de la textura associada a la regió rectangular associada al fragment, d'acord amb el mapping definit per les coordenades de textura de la primitiva.

Magnification filters

[Textures - Magnification filters](#) (14')

- **GL_NEAREST**: Canvi molt notable entre els texels al fer zoom
- **GL_LINEAR**: Canvi entre texels més smooth, es fa la mitja dels texels adjacents (interpolació bilineal)



GL_NEAREST



GL_LINEAR






Minification filters

[Textures - Minification filters \(23'\)](#)

Més difícil ja que es pot donar que un fragment hagi de representar tota la textura.

- **Mipmapping:** Downsampling d'una textura. Cada textura està representada amb dieferents resolucions (*level-of-detail*, *LODs*)

Textura inicial = LOD 0. Cada vegada que es divideix entre dos les dimensions de la textura, la nova textura és un quart de l'anterior, és un nivell més de LOD.

	Minification	$\partial u / \partial x, \partial v / \partial y$	LOD
	1x	1	0
	2x	2	1
	4x	4	2
	8x	8	3
	$2^\lambda x$	2^λ	λ

En aquest cas $2^\lambda = \partial u / \partial x$
Per tant: $\lambda = \log_2(\partial u / \partial x)$

Filtres

Filtre	Texels	LODs	Texels Totals
GL_NEAREST_MIPMAP_NEAREST	1	1	1
GL_LINEAR_MIPMAP_NEAREST	4	1	4
GL_NEAREST_MIPMAP_LINEAR	1	2	2
GL_LINEAR_MIPMAP_LINEAR	4	2	8

```
float LOD() //Retorna el paràmetre lambda = el nivell de LOD requerit per MIPMAPPING
{
    vec2 uv = vec2(WIDTH, HEIGHT) * gl_TexCoord[0].st;
    vec2 dx = dFdx(uv);
    vec2 dy = dFdy(uv);
    float rho = max( dot(dx, dx), dot( dy, dy ) );
    return max( 0.5 * log2(rho), 0.0 );
}
```

Codi per escalar textura en shader

$vtexCoord = vec2(x,y)*texCoord$; (Exemple: $vec2(2,-3)$ té 3 imatges en horitzontal i 2 en vertical, més la imatge voltejada en y)

Codi per texturar un objecte (fent escalat):

$gl_TexCoord[0].st = float k * gl_MultiTexCoord0.st$; (Exemple: si $k = 4.0$, obtenim la següent imatge)

Exercici

Tenim una aplicació que no suporta MipMapping, però volem simular el resultat de GL_LINEAR_MIPMAP_LINEAR en GLSL. Completa aquest codi, on lambda és un float amb el nivell de LOD (no necessàriament enter) més adient pel fragment:

```
vec4 sampleTexture(sampler2D sampler, vec2 texCoord, float lambda) {
    vec4 color0 = textureLod(sampler, texCoord, floor(lambda));
    vec4 color1 = textureLod(sampler, texCoord, floor(lambda)+1);
    return mix(color0, color1, fract(lambda));
}
```

Podeu assumir que $textureLod(P,sampler,lod)$ fa un accés bilineal a textura al punt P usant el nivell especificat a lod.

Wrapping

[Textures - Wrapping](#) (10')

Mode pensat per textures que es repeteixen.

```
glTexParameter f (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, mode)
```

Cofigura el mode de wrapping per la coordenada S.

Modes de wrapping

- GL_REPEAT (Mode per defecte): Repetir la textura, agafa la part fraccionaria de la textura.
- GL_CLAMP_TO_EDGE: Exten el últim mig texel de la textura.



GL_REPEAT



GL_MIRRORED_REPEAT



GL_CLAMP_TO_EDGE



GL_CLAMP_TO_BORDER

Combinació del color de la textura

[Combinació del color de la textura](#) (13')

Hi ha diferents modes per determinar le color de la textura.

- REPLACE: $\text{fragColor} = \text{texColor}$. (Quan la textura ja te la il·luminació aplicada)
- MODULATE: $\text{fragColor} = \text{texColor} * \text{frontColor}$. (Quan la textura encara no te il·luminació aplicada)
- DECAL: $\text{fragColor} = \text{mix}(\text{frontColor}, \text{texColor}, \text{texColor.a})$ Només s'utilitza quan tenim textures amb un canal Alfa, RGBA, un canal d'opacitat.

Projective texture mapping

[Projective texture mapping](#) (16')

Hi ha un projector que se'n carrega de projectar la textura.

El VS s'encarregarà de calcular les coordenades dels vèrtex i passarles de object space a window space




$$T(0.5) \cdot S(0.5) \cdot P \cdot V \cdot M \cdot [x \ y \ z \ 1] = [s \ t \ p \ q]$$

- $T(0.5) \cdot S(0.5)$: Viewport transformation
- P: Projective matrix del projector
- V: View matrix del projector
- M: Model matrix del objecte (si en té)
- $[x \ y \ z \ 1]$: vèrtex en object space
- $[s \ t \ p \ q]$: vèrtex en window space però sense haver aplicat encara la divisió de perspectiva.

Finalment se li passaran la S i la T al FS com $S=s/q$ i $T=t/q$

Aplicacions: Bump/Normal/Parallax/Relief

Aplicacions: Bump/Normal/Parallax/Relief/Displacement mapping (35')

Tècnica	Info a la textura	Ús de la textura	Coords de textura	View parallax	Self-occlusion	Detailed silhouette	On s'aplica
Color mapping	RGB 3	Kd del material	(s,t) 	-	-	-	FS
Bump mapping	D 1 	Modificar la normal	(s,t)	N	N	N	FS
Normal mapping	Normal 3 	Modificar la normal	(s,t)	N	N	N	FS
Parallax mapping	Normal + D 3+1 o 4	Modificar la normal	(s+d _s , t+d _t)	S	N	N	FS
Relief mapping	Normal + D 3+1 o 4	Modificar la normal; descartar fragments	(s+d _s , t+d _t)	S	S	S	FS!!!
Displacement mapping	D 1	Desplaçar els vèrtexs un cop subdividits els polígons.	(s,t)	S	S	S	CPU GS TCS+TES

- **Color Mapping:** A cada fragment se li aplica el color que li correspon respecte la textura.
- **Bump Mapping/Normal Mapping:** Partint d'un Height field s'assigna un color al fragment, en cas de Bump Mapping es fa mitjançant el desplaçament respecte un pla i en el cas de Normal Mapping mitjançant la normal perturbada (la normal respecte el height field).
- **Parallax Mapping:** Te en compte el moviment de la càmera. Funciona millor que el Bump Mapping i el Normal Mapping però segueix sense ser òptima. Depenent de la distància del fragment a la càmera tindrà un offset o un altre. El problema de Parallax Mapping és la oclusió.
- **Relief Mapping:** Calcula la intersecció del raig de visió amb la superfície del relleu.
- **Displacement Mapping:** Partint del Height field crea molts vèrtex. Pràcticament mateix resultat que Relief Mapping.

Displacement/Bump/Normal mapping

Teoria - Textures - Displacement/Bump/Normal mapping (42')

Exercicis

Exercici: Tenint un model amb coordenades de textura 2D que cobreixen l'interval [-1,1], com es pot fer una projecció sobre aquest espai de textura que ocupi tot el viewport?

```
gl_Position = vec4(gl_TexCoord[0].s, gl_TexCoord[0].t, 0.0, 1.0);
```

Exercici: Fes discard als fragments que cauen a la meitat dreta de la finestra OpenGL, amb mida 800x600:

a) Si P està en clip space //Si $(P.x/P.w) < 0$, esquerra; si $(p.x p.w) > 0$, dreta)
if $(P.x/P.w > 0)$ discard;

b) Si P està en NDC //En X va de -1 a 1, en Y va de -1 a 1
if $(P.x > 0)$ discard;

c) Si P està en window space //En X va de 0 a 800, en Y va de 0 a 600
if $(P.x > 400)$ discard;

Interacció amb escenes 3D

Mode Selecció OpenGL

```
glRenderMode(GL_SELECT);  
...  
for (int i=0; i<n; i++)  
{  
    glPushName(id);  
    glBegin(GL_POLYGON);  
    glVertex3f(...)  
    ...  
    glEnd();  
    glPopName();  
}  
hits = glRenderMode(GL_RENDER);
```

Pregunta examen:

El càlcul de la base ortonormal de l'espai tangent d'un triangle es pot realitzar amb el procés d'ortogonalització de Gram-Schmidt.

Bump Mapping

-Per cada texel del bump map es pot guardar:

- Directament (un height field F) $F(u,v)$ si tenim una textura amb un canal

- Gradient de $F(u,v)$: dF/du , dF/dv si tenim una textura amb dos canals.

-Si s'utilitza un Normal Map amb el Bump Mapping, les coordenades hauran d'estar codificades en Tangent Space.

Codi per calcular una aproximació del gradient d'un heightfield codificat en una textura (útil en bump mapping).

```
float F = texture2D(height,gl_TexCoord[0].st+vec2(0.0,0.0)).r;  
float Fx = texture2D(height,gl_TexCoord[0].st+vec2(eps,0.0)).r;  
float Fy = texture2D(height,gl_TexCoord[0].st+vec2(0.0,eps)).r;  
vec2 dF = 0.05*vec2(Fx-F, Fy-F)/eps;
```

Height field (Heightmap): textura utilitzada en Relief mapping, Parallax mapping, Displacement mapping, Bump mapping i Normal mapping

Exercici: Tenim un normal map on les components RGB de cada texel codifiquen les components (nx',ny',nz') en espai tangent d'un vector unitari en la direcció de la normal pertorbada. Donats els vectors T,B,N (tangent, bitangent i normal) d'una base ortonormal, en eye space, corresponents a un fragment, indica com calcularies la normal pertorbada N' en eye space.

Simplement hem de passar la normal (nx',ny',nz') de tangent space a eye space: $[T \ B \ N] * (nx',ny',nz')$ on $[T \ B \ N]$ és una matriu 3×3 que té per columnes els tres vectors T, B, N .

Exercici: Indica el shader més adient per implementar les següents tècniques:

- Displacement mapping VS/GS

- Relief mapping FS

- Parallax mapping FS

- Shadow mapping FS

<https://learnopengl.com/Advanced-Lighting/Normal-Mapping>

Ombres

Introducció

[Teoria - Ombres - Introducció](#) (17')

- Més realisme
- Indicació visual de profunditat
- Informació adicional
- Hard Shadows: Amb una llum puntual la ombra creada és total, 0% de llum.
- Soft Shadows: En cas de tenir una font de llum formada per diverses llums puntuals, un fluorescent, es crea penombra, parts on arriba llum desde alguns punts de la font però no de tots.

Si la mida de la font de llum augmenta, augmenta la penombra, disminueix la ombra (pot desaparèixer).

Si apropem la font de llum passa el contrari.

Area light: condició per haver zona de penombra en ombres d'una escena

Ombres per projecció (sense stencil)

[Teoria - Ombres - Ombres per projecció \(sense stencil\)](#) (20')

Suposem que el receptor (el terra) és pla i que la llum és puntual.

1. Dibuixem el receptor
2. Dibuixem la ombra, dibuixar l'occludor (l'objecte) projectat. Al ser pla el terra es pot dibuixar fàcilment amb una matriu.

```
glDisable (GL_LIGHTING); // !
glDisable (GL_DEPTH_TEST); // !
glMatrixMode (GL_MODELVIEW);
glPushMatrix ();
glMultMatrixf (MatriuProjeccio); // !
dibuixa (occludor);
glPopMatrix ();
```

3. Dibuixar l'occludor.

```
glEnable (GL_LIGHTING); // !
glEnable (GL_DEPTH_TEST);
dibuixa (emissor);
```

Z-fighting: Al estar a la mateixa y tant el pla com la projecció s'haurà de tenir cura que es prioritzi la projecció.

Stencil Buffer

Teoria - Ombres - Stencil Buffer (13')

És un buffer que forma part del frame buffer, per cada pixel, a més de tenir color i una z també pot tenir un valor de *stencil*.
Serverix bàsicament per fer màscares, substituir una certa part de la textura.

Això te lloc en el Per-fragment operations del Pipeline d'OpenGL, just després del fragment shader. Just abans del Depth test es fa el stencil test.

Depth test: Modify depth values to modify visibility priority of primitives. E.g. give priority to projected object over receiver in cast shadows.

El stencil buffer guarda, per cada pixel, un enter entre $0..2n-1$. ($n = n^\circ$ de pixels)

Demanar una finestra OpenGL amb stencil:

```
QGLformat f;  
f.setStencil(true);  
QGLformat::setDefaultFormat(f);
```

Obtenir el núm. de bits del stencil:

```
glGetIntegerv(GL_STENCIL_BITS, &nbits);
```

Esborrar stencil (no li afecta glStencilFunc(), sí glStencilMask):

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

Establir el test de comparació

```
glEnable(GL_STENCIL_TEST);  
glStencilFunc(comparació, valorRef, mask);
```

- Comparació pot ser: GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL i GL_ALWAYS
 - Ex: GL_LESS: $(valorRef \& mask) < (valorStencil \& mask)$
- valorRef: especifica el valor per el stencil test. valor entre $[0, 2n-1]$. $n = n^\circ$ de bits en el stencil buffer.
- mask: especifica la màscara que s'aplica amb un AND al **valorRef** i al valor de stencil guardat quan es fa el test.

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

```
glStencilOp(fail, zfail, zpass)
```

- fail -> op. a fer quan el fragment no passa el test de stencil
- Zfail -> op. a fer quan passa stencil, pero no passa z-buffer
- Zpass -> op. a fer quan passa stencil i passa z-buffer

Cadascú dels paràmetres anteriors pot ser:

- GL_KEEP: The current value is kept.
- GL_ZERO: The stencil value is set to 0.
- GL_REPLACE: The stencil value is set to the reference value in the glStencilFunc call.
- GL_INCR: The stencil value is increased by 1 if it is lower than the maximum value.
- GL_INCR_WRAP: Same as GL_INCR, with the exception that the value is set to 0 if the maximum value is exceeded.
- GL DECR: The stencil value is decreased by 1 if it is higher than 0.
- GL DECR_WRAP: Same as GL DECR, with the exception that the value is set to the maximum value if the current value is 0 (the stencil buffer stores unsigned integers).
- GL_INVERT: A bitwise invert is applied to the value.

Ombres per projecció, amb stencil

[Teoria - Ombres](#) - [Ombres per projecció - amb stencil](#) (8')

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);
```

Pinta el receptor de color negre en el stencil buffer.

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(ocloror);
glPopMatrix();
```

En els fragments en els quals el stencil buffer té un valor de 1 (negre, tot el terra), es pinten de color 0 (blanc) la projecció del ocluser. Resulta en tot el terra negre excepte la ombra del ocluser que és de color blanc.

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);
```

Tots els fragments en que el valor del stencil és 0 passen el test i es dibuixen pintats de color negre. Alsehores tenim el terra amb la ombra del ocluser.

4. Dibuixa l'occludor

```
glEnable(GL_LIGHTING);  
glDepthFunc(GL_LESS);  
glDisable(GL_STENCIL_TEST);  
Dibuixa(occludor);
```

Només cal pintar l'occludor.

Per evitar pintar una ombra fora de lloc, s'usa la versió amb stencil buffer per limitar el dibuix de l'ombra a la part ocupada pel receptor.

Matrius de projecció per ombres

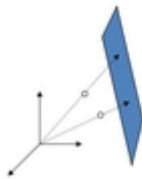
[Teoria - Ombres - Matrius de projecció](#) (10')

L'objectiu és trobar la projecció de l'occludor al pla desde la direcció de la llum. Per aconseguir-ho s'ha de calcular el punt d'intersecció del pla amb el vector que va de la llum fins un punt de l'occludor.

S'ha de substituir el punt d'intersecció ($P' = (x,y,z)$) a l'equació del pla $ax + by + cz + d$ i que doni 0.

Projecció respecte l'origen

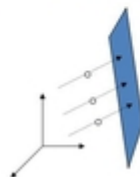
- Donats els coeficients (a,b,c,d) d'un pla, la matriu de projecció respecte l'origen és:



$-d$	0	0	0
0	$-d$	0	0
0	0	$-d$	0
a	b	c	0

Projecció en la direcció (x,y,z)

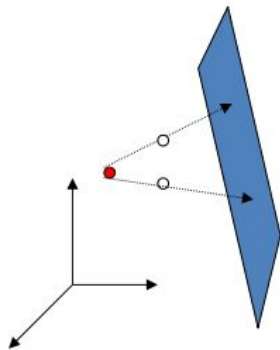
- Donats els coeficients (a,b,c,d) d'un pla, la matriu de projecció en la direcció del vector (x,y,z) és:



$by + cz$	$-bx$	$-cx$	$-dx$
$-ay$	$ax + cz$	$-cy$	$-dy$
$-az$	$-bz$	$ax + by$	$-dz$
0	0	0	$ax + by + cz$

Projecció respecte punt (x,y,z)

- Donats els coeficients (a,b,c,d) d'un pla, la matriu de projecció respecte un punt (x,y,z) és:



$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -(d+ax+by+cz) & 0 & 0 & 0 \\ 0 & -(d+ax+by+cz) & 0 & 0 \\ 0 & 0 & -(d+ax+by+cz) & 0 \\ a & b & c & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$-d - by - cz$	xb	xc	xd
ya	$-d - ax - cz$	yc	yd
za	zb	$-d - ax - by$	zd
a	b	c	$-ax - by - cz$

Shadow Volumes

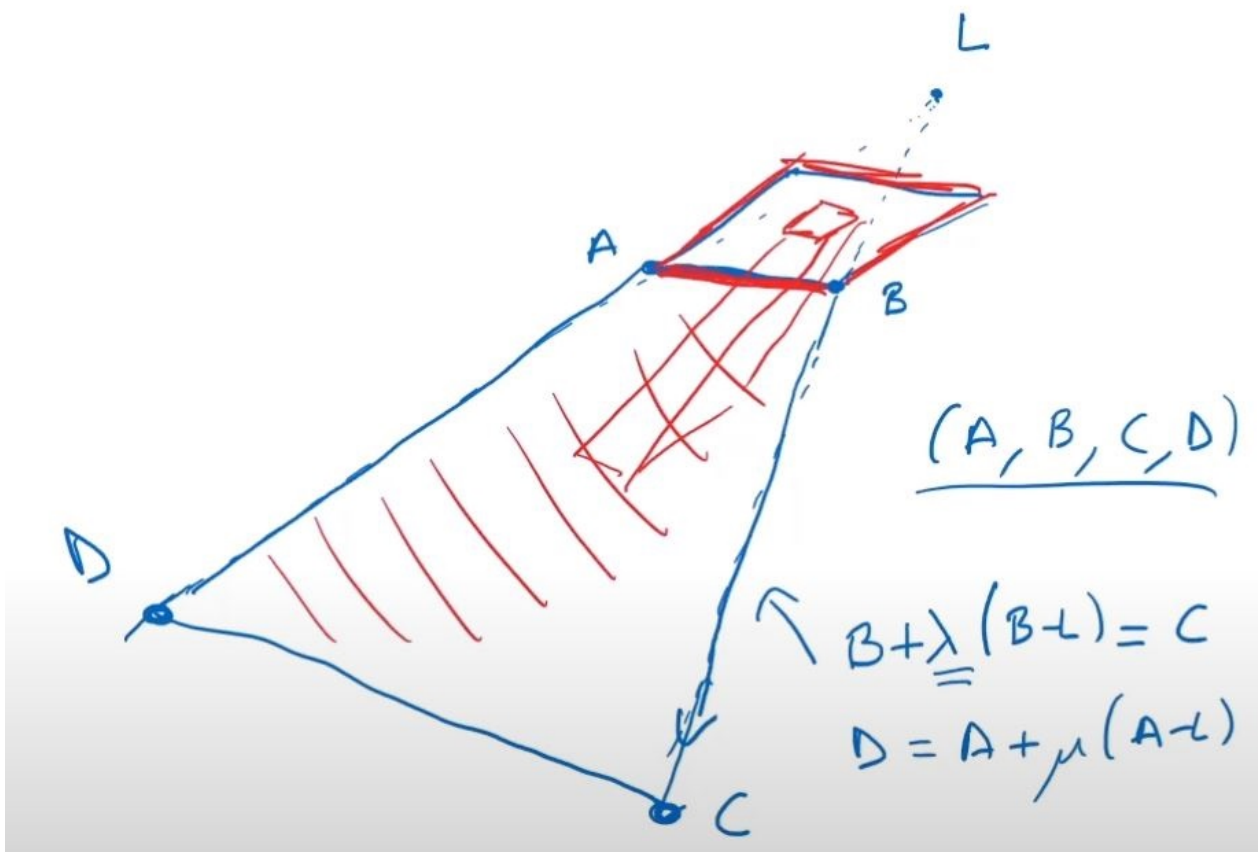
Teoria - Ombres - Shadow Volumes (45')

- El receptor no ha de ser pla com en els casos anteriors.
- L'occludor ha de ser sencill.
- Llum puntual

Generació de Shadow Volume (SV)

$$SV(O, L) = \{P \in \mathbb{R}^3 \mid \overline{PL} \cap O\}$$

- Aresta Contorn:** Aquelles arestes tal que una de les dues cares que la formen no és frontface, és una backface.



Per saber si un punt es troba a l'interior del shadow volume es llança un raig en qualsevol direcció (cap a la càmera per exemple) i comptar quantes vegades intersecciona amb els límits del shadow volume, si intersecciona un nombre senar de vegades, està a dins. També es pot comptar quantes vegades creua cares frontface i backface, si creua més cares frontface que backface el punt és interior.

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);
```

Exercici

Què fa la funció: void glDepthMask(GLboolean foo)?

Activa/desactiva l'escriptura en el depth buffer.

Frontface / Backface test

[Teoria - Front face / back face test](#) (6')

Per saber si una cara és frontface o backface es calcula la distància del pla a la càmera o a la llum (Q).

$$dist(Q,pla) = aqx + bqy + cqz + d$$

Si $dist(Q,pla) > 0 \Rightarrow$ frontface, sino backface

Un altre manera es calcular el producte entre el vector Q-P (P és un vèrtex del pla que es vol testear) i la normal del pla. Si el resultat és major a 0, frontface, sino Backface

Shadow Mapping

[Teoria - Ombres - Shadow Mapping](#) (56')

La tècnica més utilitzada ja que és molt simple i té poc overhead. Tant el receptor com l'occludor poden ser arbitraris. Llum puntual i unidireccional (*spotlight*) cap problema, llum omnidireccional (sol, per exemple) dona problemes.

Es visualitza l'escena desde una càmera situada a la font de llum i s'utilitza el depth map, anomenat ara shadow map.

Al VS es fan els mateixos càlculs que amb projective texture mapping:

$$T(0.5) \cdot S(0.5) \cdot P \cdot V \cdot M \cdot [x \ y \ z \ 1] = [s \ t \ p \ q]$$

El FS obtindrà per una banda una z calculada amb $shadowMap(s/q, t/q)$ (*storedDepth*) i per l'altre la z en window space (p/q) (*trueDepth*).

Si per un punt $shadowMap(s/q, t/q) < (p/q)$ aleshores el punt estarà a la ombra o ocult.

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);
```

Shadow mapping – 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);
```

Shadow mapping – Pas 2. 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)

Shadow mapping - VS

```
uniform mat4 lightMatrix;
...
void main()
{
    ...;
    gl_TexCoord[0] = lightMatrix*gl_Vertex;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Shadow mapping - FS

```
vec2 st = gl_TexCoord[0].st / gl_TexCoord[0].q;
float trueDepth = gl_TexCoord[0].p / gl_TexCoord[0].q;
float storedDepth = texture2D(shadowMap, st).r;
if (trueDepth <= storedDepth)
    gl_FragColor = ... ;// iluminat
else
    gl_FragColor = ... ;// a l'ombra
```

Evitar Z-fighting

glPolygonOffset(GLfloat factor, GLfloat units): abans del depth test, es modifica el valor de la z del fragment (per defecte en [0,1]), 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

Dz representa quant **tangencial és la primitiva a la direcció de visió**.

- factor: permet introduir un offset variable (depén de la inclinació del polígon)
- units: permet introduir un offset constant

offset = dz · factor + r · units

glPolygonOffset is useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges.

Reflexions especulars

Introducció

[Teoria - Reflexions - Introducció](#) (26')

Considerem un punt d'una superfície amb una certa normal i una llum puntual de la qual li arriba un raig de llum (L).

Reflexió difosa

La probabilitat que aquest raig es reflecteixi en qualsevol direcció d'una semiesfera centrada al punt, és la mateixa. No importa on estigui l'observador, veurà el punt del mateix color.

Reflexió especular

El raig que arriba al punt es reflectirà en la direcció oposada respecte la normal (R_L), l'angle reflexat = l'angle incident. El vector de reflexió R és coplanar a N i a L (estàn al mateix pla). Depenent de la posició de l'observador es veurà un color o un altre.

Calcular el vector reflectit

Tenim R (raig reflectit), N (normal de la superfície) i L (raig de llum). R es pot descomposar en R_1 (en direcció a N) i R_2 (perpendicular a N).

$$R = R_1 + R_2 \Rightarrow -L + 2N(N \cdot L)$$

$$R_1 = -L + R_2$$

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

Reflexió basada en objectes virtuals

[Teoria - Reflexions - Objectes virtuales](#) (51')

El mirall (el terra, per exemple) ha de ser pla.

La idea és dibuixar l'objecte que es reflecteix dues vegades, la real i la reflectida en el mirall (virtual). Això es pot fer de diverses maneres.

Modelat

Duplicar l'escena simètricament respecte el mirall i treure un mirall, que hi hagi un forat i es pugui veure la part duplicada que fa l'efecte de reflexió. Es fa en temps de modelat, per tant si un objecte es desplaça, el seu duplicat no. Per aconseguir un resultat realista també es duplica la font de llum.

1. Dibuixar els objectes en posició virtual

```
glPushMatrix();
glMultMatrix(matriu_simetria)
glLightfv(GL_LIGHT0, GL_POSITION, pos);
glCullFace(GL_FRONT);
dibuixar(escena);
glPopMatrix();
```

La matriu de simetria es calcula mitjançant el mirall de l'escena i el pla en el que es troba. Suposem un mirall amb normal (1,0,0) que es troba en el pla x=0, aleshores la matriu simetria seria (-1, 1, 1, 1).

2. Dibuixar el mirall semi-transparent

```
glLightfv(GL_LIGHT0, GL_POSITION, pos);
glCullFace(GL_BACK);
dibuixar(mirall);
```

3. Dibuixar els objectes en posició real

```
dibuixar(escena);
```

Aquest mètode pot comportar problemes com per exemple voler modelar una habitació amb un mirall que comparteix paret amb una altre habitació. Qualsevol forat de la paret es considera mirall. Una solució és utilitzar `glClipPlane()`. Una altre solució és utilitzar un stencil buffer i un stencil test.

Reflectits (amb stencil test)

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. Dibuixar els objectes en posició real

```
dibuixar(escena);
```

Textures dinàmiques

Consisteix en dibuixar els objectes en posició virtual, crear una textura, dibuixar el mirall utilitzant la textura creada i finalment dibuixar el objectes en posició real.

Matriu de reflexió

Tenim un punt P que es reflexa en un mirall que es troba en un pla i té una normal n. El que busquem és la posició del punt P reflexat en el mirall (P'), és a dir el punt simètric de P utilitzant el pla com a pla de simetria.

- Pla: $ax + by + cz + d = 0$
- $P = (px, py, pz)$

Primer necessitem la distància del pla al punt P, $\text{dist}(P, \text{pla}) = apx + bpy + cpz + d$.

- $P' = P - 2\text{dist}(P, \text{pla})N$

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

Environment mapping

[Teoria - Environment Mapping](#) (65')

Donada una direcció arbitrària R, ens retorna el color de l'entorn en direcció R.

`color = EnvironmentMap (R)`

Per calcular el color d'un punt, s'han de considerar el view vector (V, que va del punt la càmera), la normal de la superfície on es troba el punt (N), el vector de reflexió (R_V)

El VS passarà el punt P i la normal N al FS en eye space o world space. Aleshores el FS calcula el color.

- V (en world space) = normalitzar(obs - P)
- V (en eye space, obs = 0) = normalitzar(-P)
- $R_V = 2(N \cdot V)N - V$ / $R_V = \text{reflect}(-V, N)$
- `fragColor = EnvironmentMap(R_V)`

Hi ha una altra manera d'utilitzar Environment Mapping, que és com entorn (background).

Sphere Mapping

Amb un sphere map és possible capturar els 360 de l'entorn de l'esfera.

L'objectiu és passar del vector R_V a les coordenades de textura u,v que van de -1 a 1 i després, d'aquestes a s,t que van de 0 a 1.

Assumim que la càmera és axonomètrica, de manera que els vectors V que van de l'esfera a la càmera són constants. La normal N és igual al vector P que va del centre de l'esfera al punt del qual es vol calcular el color.

Al considerar els vector com constants i en direcció a la càmera, com que la càmera sempre mira en direcció -Z, $V = (0,0,1)$, per tant:

- $R_V = 2(N \cdot (0,0,1))N - (0,0,1) = 2n_z(n_x, n_y, n_z) - (0,0,1) = (2n_zn_x, 2n_zn_y, 2n_z^2 - 1)$

$$R = (2n_zn_x, 2n_zn_y, 2n_z^2 - 1) = (r_x, r_y, r_z)$$

$$p_z = n_z = \sqrt{(r_z + 1)/2}$$

$$p_x = n_x = r_x / 2n_z \rightarrow u = r_x$$

$$p_y = n_y = r_y / 2n_z \rightarrow v = r_y$$

$$s = ((r_x / 2n_z) + 1) / 2$$

$$t = ((r_y / 2n_z) + 1) / 2$$

Si els càlculs es fan en eye space no veurem cap canvi al girar l'esfera, en canvi si els fem en world space, al girar l'esfera girarà l'entorn reflectit. S'utilitza més eye space ja que tot i que no es pot moure, la qualitat del que es veu és millor i no s'aprecia tant la distorsió.

Cube Mapping

Amb cube mapping en canvi (un cub format per 6 imatges), hi ha molta menys distorsió y es pot fer tant en eye space com amb world space.

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, ...);
```

2. Activació Cube mapping

```
glEnable(GL_TEXTURE_CUBE_MAP_EXT);
```

3. Generació de coordenades de textura

```
glTexGenfv(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT);
glTexGenfv(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT);
glTexGenfv(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
```

GLSL

```
uniform samplerCube samplerC;
...
vec3 R;
...
vec4 color = textureCube(samplerC, R);
```

Objectes Translúcids

Quan un raig de llum canvia de medi (aire -> vidre, vidre -> aire) s'anomena transmissió.

Major densitat del medi major índex de refracció, tot i que també depèn de la longitud d'ona de la llum.

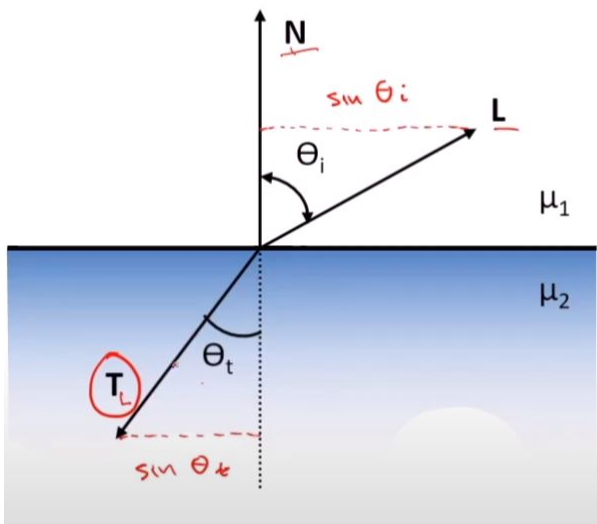
Llei de Snell

La llei de Snell és una fórmula simple utilitzada per a calcular l'angle de refracció de la llum en travessar la superfície de separació entre dos medis de propagació de la llum (o qualsevol ona electromagnètica) amb índex de refracció diferent.

Les magnituds que afecten a la direcció del raig transmès són:

- Velocitat de propagació de la llum als medis
- Longitud d'ona de la llum
- Angle d'incidència

$$\sin(\theta_t) = \frac{\mu_1}{\mu_2} \sin(\theta_i) = \mu \sin(\theta_i)$$



Si passem d'aire a vidre, el vidre és més dens, per tant:

$$\mu_2 > \mu_1$$

$$\mu = \frac{\mu_1}{\mu_2} < 1$$

$$\sin(\theta_t) \leq \sin(\theta_i)$$

Quant major l'angle de refracció més proper serà el vector de transmissió a la normal, en canvi si és petit, el vector de transmissió s'allunyarà de la normal.

L'angle crític és aquell en el que tota la llum es reflexa i cap es transmet. Això passa si l'angle d'incidència és molt gran.

$$\theta_c = \text{asin}\left(\frac{\mu_2}{\mu_1}\right)$$

Equacions de Fresnel

Assumim que:

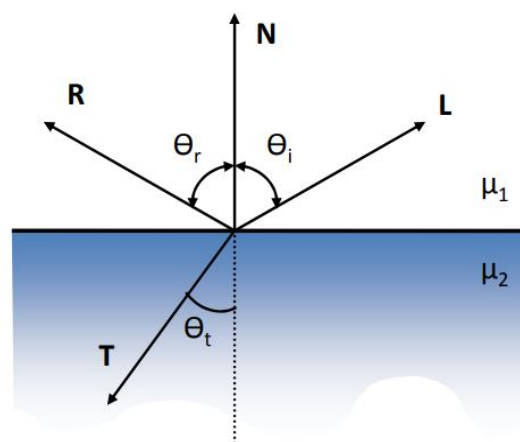
- El comportament és especular pur, no hi ha dispersió.
- No hi ha absorció, $R+T=1$
- El material no és conductor, és dielèctric.

Equacions de Fresnel

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

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

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



- R_s : llum no polaritzada
- R_p : llum polaritzada

Al passar a un medi més dens, major l'angle d'incidència, més reflexió. En canvi, al passar a un medi menys dens es passa de transmetre una mica a reflexió total a partir de l'angle crític.

Aproximació de Schlick

$$R = f + (1 - f)(1 - LN)^5$$

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

Alpha Blending

Teoria - Alpha blending (27')

A cada component de RGB es guardarà també un component **A** que ens indicarà la opacitat.

S'han de pintar els polígons en un determinat ordre, dels més llunyans als més propers, Back-to-front.

El color del fragment després de sortir del FS se'n diu *src_color*. És aleshores, després del FS que s'aplica l'alpha blending.

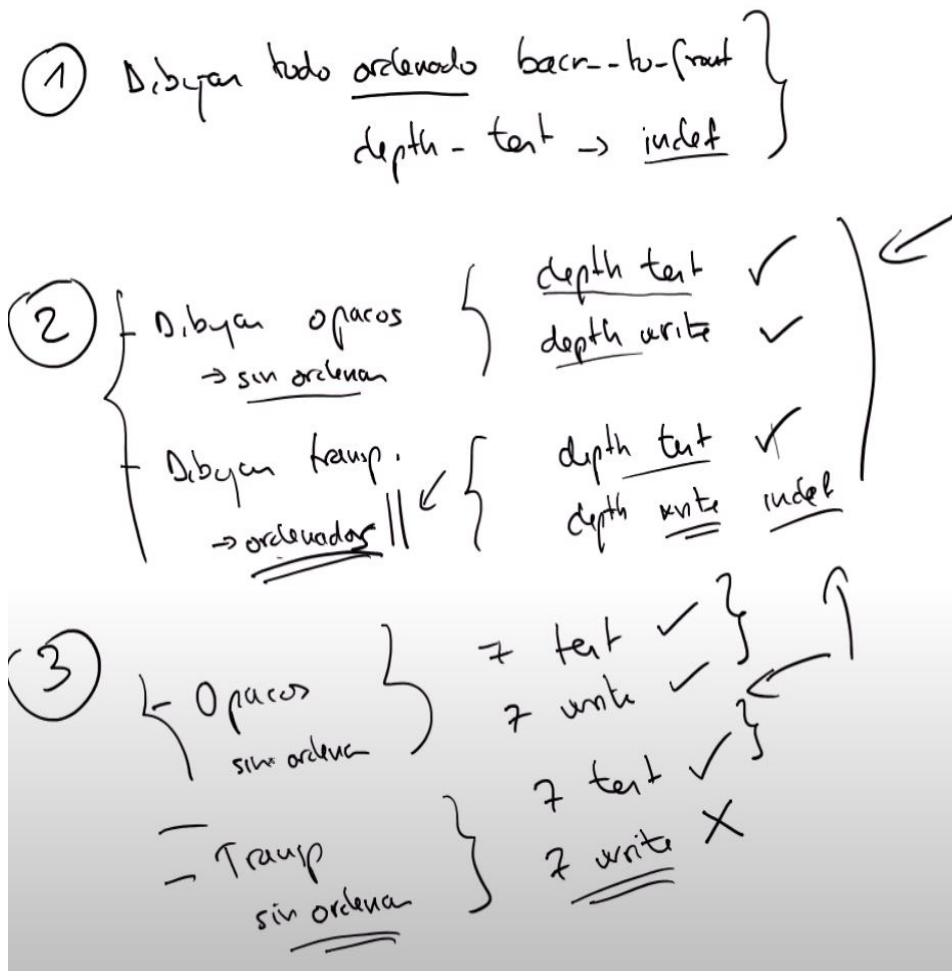
S'agafa el *src_color* i del color buffer s'agafa el color d'allò que s'havia dibuixat abans (per això lo de back-to-front) que se'n diu *dst_color* (destination color) i es passen a la funció de Blending, aquesta no es pot programar però sí que podem manipular quin pes tindrà cada color.

`glEnable(GL_BLEND)` must be used before to enable blending.

`glBlendFunc(GLenum sfactor, GLenum dfactor)`: defines the operation of blending for all draw buffers when it is enabled.

Color = **sfactor** * *src_color* + **dfactor** * *dst_color*

Opcions respecte el z-buffer



- 1: Més costosa X
- 2: OK
- 3: Més òptima però pot quedar cutre

Exemple exercici

Tenim:

- Fragment amb color RGBA del pixel (i,j) = (1.0, 0.5, 0.0, **0.8**) - SOURCE
- El color RGBA del pixel (i,j) al buffer de color = (1.0, 0.5, 1.0, 1.0) - DESTINATION

Si tenim activat alpha blending amb la crida:

`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`

Quin serà el color RGBA resultant al buffer de color (assumint que passa tots els tests)?

$R = GL_SRC_ALPHA * S + GL_ONE_MINUS_SRC_ALPHA * D$

$$0.8 * 1.0 + 0.2 * 1.0 = 1.0$$

$$0.8 * 0.5 + 0.2 * 0.5 = 0.5$$

$$0.8 * 0.0 + 0.2 * 1.0 = 0.2$$

$$0.8 * 0.8 + 0.2 * 1.0 = 0.84 \text{ (1.0, 0.5, 0.2, 0.84)}$$

Il·luminació global

Introducció

Teoria - Introducció a la il·luminació global (14')

Quan l'observador mira un punt, reb la llum d'aquest punt, el qual està reflexant la llum que li arriba o fins i tot emitint la llum que se li transmet en cas de ser una superfície translúcida. La il·luminació pot ser:

- Local: Llum directa, llum que va directament de la font de llum al punt. Els model d'il·luminació local només tenen en compte aquest tipus d'il·luminació.
- Global: Llum indirecta, llum que arriba al punt després d'haver tingut com a mínim una interacció amb una altre superfície.

Radiometria

Teoria - Radiometria (30')

- Radiometria: Disciplina que estudia la mesura de les radiacions electromagnètiques. (Qualsevol longitud d'ona)
- Fotometria: Disciplina que estudia la mesura de les radiacions visibles. (Longitud d'ona visible)

Resum

Sím.	Radiomet.	Fotometria	Definició	Ús
Φ	Fluxe (W)	Fluxe (lm)	Energia que travessa una superfície per unitat de temps	Energia total que emet una font de llum
E	Irradiancia (W/m²)	Iluminància (lux=lm/m²)	Fluxe per unitat d'àrea	Llum que incideix en un punt, des de qualsevol direcció
I	Intensitat (W/sr)	Intensitat (cd=lm/sr)	Fluxe per unitat d'angle sòlid	Distribució direccional d'una llum puntual
L	Radiància W/(sr·m²)	Luminància (cd/m²)	Fluxe per unitat d'àrea i unitat d'angle sòlid	Energia que travessa un punt en una determinada direcció

BRDF / BTDF / BSDF

Teoria - BRDFs (14')

BRDF: Bidirectional refelctance distribution function

BTDF: Bidirectional transmission distribution function

BSDF: Bidirectional scattering distribution function

BRDF

$fr(p, w_o, w_i)$

- p : punt de la superfície que s'està considerant
- w_o : direcció de sortida (vector del punt a l'observador)
- w_i : direcció d'entrada (vector del punt a la font de llum)
- $fr(p, w_o, w_i) \geq 0$
- $fr(p, w_o, w_i) = fr(p, w_i, w_o)$

$$\int_{\Omega} fr(p, \omega_o, \omega') \cos(\Theta) d\omega' \leq 1$$

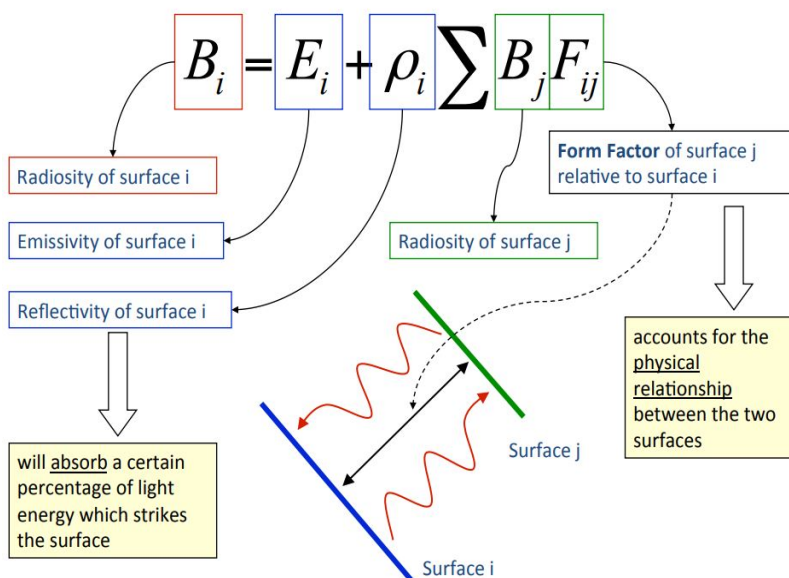
Eqüació general del rendering

Teoria - Eqüació general del rendering (25')

$$L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} fr(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t) (\omega_i \cdot n) d\omega_i$$

- λ és la longitud d'ona de la llum.
- Ω és la semiesfera centrada al punt x i alineada amb la normal n que representa totes les possibles direccions en les que pot arribar llum a x .
- "Omega i" fa de light vector L del model d'il·luminació de Lambert.
- "fr" és el BRDF. Els seus 3 primers paràmetres:
 - Posició
 - Direcció d'entrada
 - Direcció de sortida

Eqüació de radiositat



Radiositat com a sistema d'equacions

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1-\rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

Light Paths

Tots els possibles camins del raigs de llum desde que s'emeten d'una font de llum fins que arriben al sensor. Cada camí de llum s'etiqueta amb una combinació que consisteix en:

- L (D|S)* E
 - L: Llum
 - D: Interacció amb una superfície difusa
 - S: Interacció amb una superfície especular
 - E: Eye, observador, la càmera

Phong Lightning

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

Equació d'obscuràncies

$$W(P, N) = \frac{1}{\pi} \int_{\Omega} \rho(d(P, \omega)) \cdot (N \cdot \omega) d\omega$$

Què representa ρ ? Una funció que decreix segons la distància

Com hauria de ser ρ per obtenir oclusió ambient? Funció constant $\rho = 1$

Ray Tracing

[Teoria - RayTracing i variants \(29'\)](#)

Ray-casting

La idea és no utilitzar rasterització i en el seu lloc llançar raigs de la càmera a la escena. Per això, el primer que es fa es recórrer tots els pixels de la imatge. Utilitza il·luminació local, no hi ha càlcul recursiu. Per calcular el color del pixel, es calcula el color de la component del raig reflectit en la superfície en direcció a la càmera.

Ray Tracing clàssic

[Teoria - RayTracing clàssic \(35'\)](#)

Es llançen raigs del observador a l'escena.

Al impactar el raig amb una superfície difusa, es calcula la radiància de la superfície en direcció a la càmera, i a més es llançen shadow rays del punt a la font de llum per calcular la contribució de la il·luminació directa.

En cas d'impactar en una superfície especular, s'ha de calcular el raig reflectit (i en cas de ser un material com el vidre, també el raig transmès), aquest raig reflectit pot impactar a la vegada en una altra superfície especular i així successivament. Amb això podríem arribar a tenir un arbre binari, recursivitat que acabaria en cas d'arribar a una superfície difusa.

- $LDS * E$
- $LS * E$
- Suporta light paths

Color amb que un determinat observador veu un cert punt P:

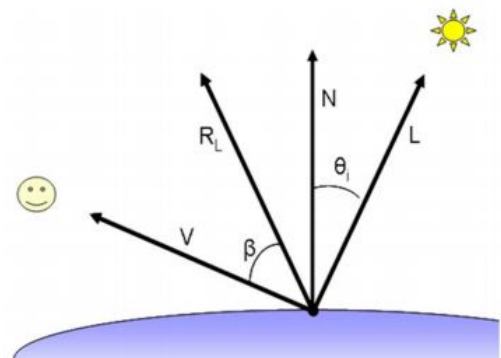
$$I(P) = I_D(P) + I_R(P) + I_T(P)$$

- $I_D(P)$: color degut a la **llum directa** dels focus.
- $I_R(P)$: color degut a la **llum indirecta que es reflecteix** a P en direcció cap a l'observador.
- $I_T(P)$: color degut a la **llum indirecta que es transmet** des de P en direcció cap a l'observador.

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

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

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

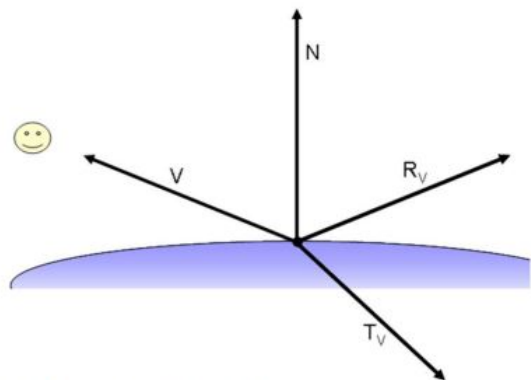


En cas que la superfície sigui especular també s'han de tenir en compte $I_R(P)$ i $I_T(P)$

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

$$I_R(P) = K_R L_R$$

$$I_T(P) = K_T L_T$$



- 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

```
def rayTracing():
    for i in [0..w-1]:
        for j in [0..h-1]:
            raig = raigPrimari(i, j, camera)
            color = tracarRaig(raig, escena, μ)
            setPixel(i, j, color)
```

```
def tracar_raig(raig, escena, μ):
    if profunditat_correcta():
        info = calcula_interseccio(raig, escena)

        if info.hi_ha_interseccio():
            color = calcular_ID(info, escena) # ID

            if es_reflector(info.obj):
                raigR = calcula_raig_reflectit(info, raig)
                color += KR*tracar_raig(raigR, escena, μ) #IR

            if es_transparent(info.obj):
                raigT = calcula_raig_transmes(info, raig, μ)
                color += KT*tracar_raig(raigT, escena, info.μ) #IT

            else color = colorDeFons

        else color = Color(0,0,0); # o colorDeFons

    return color
```

FS

```
void main() {
    vec3 obs = gl_ModelViewMatrixInverse[3].xyz;
    vec3 dir = normalize(pos - obs);
    Ray ray = Ray(obs,dir);
    gl_FragColor = trace(ray, 1.0);
}

vec4 trace(Ray ray, float mu) {
    if( intersectScene(ray, Phit, Nhit,Kd, Kr, Kt)){
        Ray shadowRay = Ray(Phit, normalize(lightPos-Phit));
        bool inShadow = intersectScene(shadowRay, aux, aux,aux4, aux4, aux4);
        if (inShadow) shadow = 0.2; else shadow = 1.0;
        color+= shadow*light(Nhit, -ray.dir, lightPos-Phit, Kd, vec4(1.0));

        vec3 R = reflect(ray.dir, Nhit);
        color += Kr*trace1(Ray(Phit, R), mu);

        if (mu==1.0) muHit=1.5;
        else { muHit = 1.0; Nhit*=-1.0;}
        vec3 T = refract(ray.dir, Nhit, mu/muHit);
        if (length(T)>0.0) color+=Kt*trace1(Ray(Phit, T), muHit);
    }
    else color+=samplePanorama(ray.dir);
    return color;
}
```

Path Tracing

Per cada pixel es llançen R raigs.

Es llança un primer raig, reaccionarà igual que amb ray tracing clàssic però en aquest cas s'agafarà un camí del arbre, no es calcularan tots. Per cada raig llançat s'agafarà un camí diferent. Resulta en una imatge amb molt de soroll. No és òptim.

- $LS \cdot DS \cdot E$
- Suporta light paths

Distributed Ray Tracing

Igual que ray tracing clàssic però al impactar en una superfície especular, en lloc de llançar un raig de reflexió, llança diversos i resulta en un arbre n-ari.

Two-pass ray tracing

Pas 1

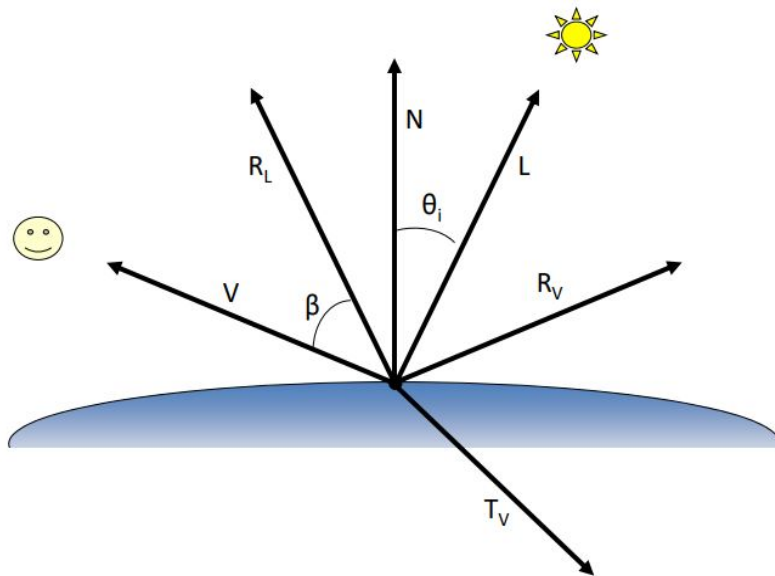
Es llançen raigs desde la font de llum, cada raig surt amb una determinada energia. Al impactar amb una superfície especular té el mateix comportament que el ray tracing clàssic, però al impactar en una superfície difusa, es guarda l'energia amb la que ha arribat el raig.

Pas 2

Mateix comportament en amb ray tracing clàssic però al arribar a una superfície difusa es té en compte l'energia dels raigs que han arribat als punts propers.

- $LS \cdot DS \cdot E$
- $(LS \cdot E)$
- Suporta light paths

Exercici



Exercici: Amb la notació de la figura, indica, en el cas de Ray-tracing:

- Quin vector té la direcció del shadow ray?
 - L
- Si el punt pertany a un mirall, quina és la direcció del raig reflectit que cal traçar?
 - R_V???
- Quin vector depèn de l'índex de refracció?
 - T_V
- Quin vector és paral·lel al raig primari?
 - V
- Quin vector cal usar per indexar un cube map, en el cas de cube mapping?
 - R_V
- Quins dos vectors determinen la contribució de Phong?
 - R_L i V
- Quins dos vectors determinen la contribució de Lambert?
 - L i N

Intersecció raig triangle

VBOs

Exercici: Tenim un cub format per 6 cares i 8 vèrtexs.

a) Si volem que cada corner (vèrtex associat a una única cara) tingui la normal de la cara a la que pertany, quants vèrtexs necessitem en el VBO del cub?

$6 \text{ cares} * 4 \text{ vèrtexs per cada cara} = 24 \text{ vèrtexs}$

b) Si volem que cada vèrtex del cub tingui com a normal el promig de les normals de les cares que incideixen al vèrtex, quants vèrtexs necessitem (com a mínim) en el VBO del cub?

8 vèrtexs (mínim)

glLightfv
glBegin
glNormal3f
glVertex3f
glEnd
glTranslate
glTranslated
glScalef
glScaled
glRotatef
gluLookat
gluPerspective
glFrustum
glOrtho
glViewport
glDepthRange
glDrawArrays
glDrawElements
glDrawBuffers
glColorMask
glDepthMask
glRenderMode
glPopName
glGetDoublev
glGetIntegerv
glReadPixels
gluUnProject
glMatrixMode
glLoadIdentity
glMultMatrixf
glMatrixMode
glTexCoord3f
glTexGen
glPushMatrix
glDisable
glEnable
glClearStencil
glClear
glStencilFunc
glStencilOp
PRIMITIVES
GL_POINT
GL_LINES
GL_POLYGON