

# **Apuntes G Examen Final.pdf**



**user\_3666769**



**Gráficos**



**3º Grado en Ingeniería Informática**



**Facultad de Informática de Barcelona (Fib)  
Universidad Politécnica de Catalunya**

UNIVERSITAT POLITÈCNICA DE  
CATALUNYA

GRÀFICS

# Apuntes

*Alex Garcés*



Q1 2023-2024

WUOLAH

# Índice general

<b>1. Proces de Visualización Proyectivo</b>	<b>3</b>
1.1. Coordenadas Homogeneas . . . . .	3
1.2. Sistema de Coordenadas . . . . .	4
1.3. Vectores . . . . .	6
1.3.1. Normales . . . . .	6
1.4. Pipeline OpenGL . . . . .	7
1.4.1. Pipeline Programable . . . . .	9
<b>2. Texturas</b>	<b>10</b>
2.1. Mapping . . . . .	10
2.2. Pipeline - Coordenadas de textura . . . . .	12
2.3. Generación de coordenadas de textura . . . . .	13
2.3.1. Planos S, T . . . . .	13
2.3.2. Superficies auxiliares . . . . .	13
2.4. Creación de texturas . . . . .	15
2.5. Filtrado de texturas . . . . .	16
2.5.1. Magnification Filter . . . . .	16
2.5.2. Minification Filter . . . . .	16
2.5.3. Wrapping . . . . .	17
2.6. Interpolación de coordenadas de textura . . . . .	18
2.6.1. Perspective Correct Interpolation . . . . .	18
2.7. Projective Texture Mapping . . . . .	18
2.8. Aplicaciones de texturas . . . . .	19
2.8.1. Parallax Mapping . . . . .	19
2.8.2. Relief Mapping . . . . .	20
2.8.3. Displacement Mapping . . . . .	20
<b>3. Sombras</b>	<b>21</b>
3.1. Introducción . . . . .	21

3.2. Sombras por proyección . . . . .	22
3.2.1. Matriz de Proyección . . . . .	25
3.3. Shadow Volume . . . . .	27
3.3.1. FrontFace y BackFace . . . . .	29
3.4. Shadow Mapping . . . . .	29
<b>4. Reflexiones Especulares</b>	<b>33</b>
4.1. Reflexiones basadas en objetos virtuales . . . . .	33
4.1.1. Reflexiones (sin <i>Stencil Test</i> ) . . . . .	33
4.1.2. Reflexiones (con <i>Stencil Test</i> ) . . . . .	34
4.1.3. Texturas Dinámicas . . . . .	35
4.2. Environment Mapping . . . . .	36
4.2.1. Sphere Mapping . . . . .	38
4.2.2. Cube Mapping . . . . .	39
<b>5. Objetos Translúcidos</b>	<b>40</b>
5.1. Objetos translúcidos . . . . .	40
5.1.1. Ley de Snell . . . . .	41
5.1.2. Ecuaciones de Fresnell . . . . .	42
5.2. Alpha Blending . . . . .	42
5.3. Cross Product . . . . .	45
<b>6. Iluminación Global</b>	<b>46</b>
6.1. Radiometría . . . . .	47
6.1.1. Calculo de la Radiancia . . . . .	49
6.2. BRDF-BTDF-BSDF . . . . .	49
6.3. Ecuación General del Rendering . . . . .	50
6.4. Light Paths . . . . .	51
<b>7. Ray Tracing</b>	<b>52</b>
7.1. Ray Tracing y variantes . . . . .	52
7.1.1. Ray Casting . . . . .	52
7.1.2. Ray Tracing Clásico . . . . .	52
7.1.3. Path Tracing . . . . .	53
7.1.4. Distributed Ray Tracing . . . . .	53
7.1.5. Two-Path Ray Tracing . . . . .	54
7.2. Ray Tracing Clásico . . . . .	55
7.3. Intersección Rayo-Triángulo . . . . .	57

# Capítulo 1

## Proces de Visualización Proyectivo

Las etapas principales de visualización de una objeto son: Procesamiento de Vertices  $\Rightarrow$  Rasterización  $\Rightarrow$  Imagen. Ahora bien, el pipeline de OpenGL hace muchas cosas por detras.

La información que se pasa al pipeline de OpenGL se divide en dos tipos:

- puntos (vertices)
- vectores (normales)

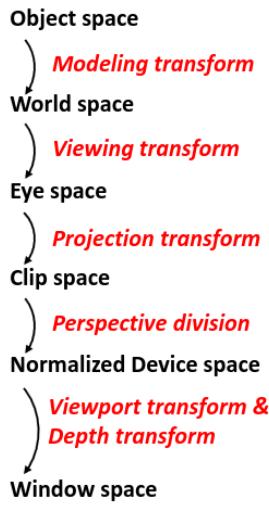
### 1.1. Coordenadas Homogeneas

Las coordenadas homogeneas son coordenadas que nos permiten representar tanto puntos como vectores. También nos permiten representar transformaciones geométricas a partir de matrices que funcionan tanto en puntos como vectores.

Las coordenadas homogeneas destacan por tener una coordenada adicional, en un espacio de tres dimensiones ( $x, y, z, w$ ). En puntos la coordenada adicional actua como una división postergada ( $w \neq 0$ ). En vectores la coordenada homogonea no nos interesa y por tanto su valor será nulo ( $w = 0$ ).

La principal razón de usar coordenadas homogeneas es que las transformaciones no tienen por que ser lineales.

## 1.2. Sistema de Coordenadas



**Object Space** ( $x_m, y_m, z_m, w_m$ ) Sistema de coordenadas con que se ha generado un modelo ( $w_m = 1$ ).

**World Space** ( $x_a, y_a, z_a, w_a$ ) Sistema de coordenadas que representa los objetos dentro de una escena ( $w_a = 1$ ).

Las transformaciones que nos permiten pasar de Object Space a World Space son las *modeling transform* que incluyen transformaciones de

- Translación.
- Escalado.
- Rotación.

**Eye Space** ( $x_e, y_e, z_e, w_e$ ) Sistema de coordenadas que representa la escena des del punto de vista de la cámara ( $w_e = 1$ )

Las transformaciones que nos permiten pasar de World Space a Eye Space son las *viewing transform*. Las transformación se basan en operaciones sobre matrices para orientar y posicionar la cámara son de dos tipos:

---

`lookAt(eye, target, up)`

---

$$T(0, 0, -d) \times R_z(-\phi) \times R_y(\theta) \times R_x(-\psi) \times T(0, 0, -VRP)$$

**Clip Space** ( $x_c, y_c, z_c, w_c$ ) Sistema de coordenadas con la cámara definida con propiedades de visión. El aspect ratio, la zNear, la zFar, si es de perspectiva el FOV, si es axonométrica el frustum de visión.

Estas transformaciones se conocen como las *projection transform* Normalmente ya hay funciones definidas como *perspective*, *frustum*, *ortho*.

---

`perspective(fov, aspect, z_near, z_far)`

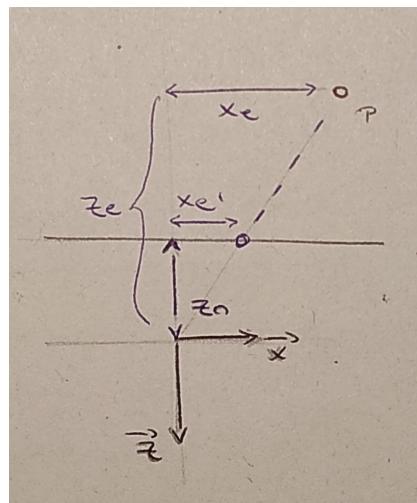
---

En clip space queremos saber con una camara determinada con una optica determinada que puntos van en que parte de la pantalla. Esto se hace en dos pasos:

1. Uso de la projection matrix.
2. División de perspectiva.

Tenemos un punto  $p$  y queremos saber en que punto de la cámara se proyecta. Sea  $z_n$  el parámetro zNear de la cámara calculamos el punto de intersección entre el rayo que sale de  $p$  hacia la cámara y el plano paralelo a esta a distancia  $z_n$ . Una vez tenemos el punto de intersección solo queda encontrar la distancia de  $x$  respecto al origen

$$\frac{x'_e}{z_n} = \frac{x_e}{z_e}$$



En Clip Space  $x_c, y_c, z_c \in [-w_c, w_c]$ . Si la cámara es perspectiva entonces  $w_c = -z_e$

**Normalized Device Space**  $(x_n, y_n, z_n)$  Coordenadas en el rango  $[-1, 1]$ . Un punto situado sobre zNear tiene componente  $z_n = -1$  y uno situado sobre zFar tiene componente  $z_n = 1$ .

Para transformar de ClipSpace a NormalizedDeviceSpace aplicamos la transformación

$$(x_c, y_c, z_c, w_c) \longmapsto (x_c/w_c, y_c/w_c, z_c/w_c)$$

**Window Space**  $(x, y, z)$  Coordenadas en pixeles para  $x$  e  $y$ , y  $z \in [0, 1]$ . Para conseguir estas coordenadas se aplican dos operaciones.

1. Viewport Transformation: transforma las coordenadas  $(x, y)$  a coordenadas en el rango  $([0, width], [0, height])$  en donde  $width$  es el ancho de la ventana y  $height$  es la altura de esta.
2. Depth Range: Transforma las coordenadas  $z$  de  $[-1, 1]$  a  $[0, 1]$

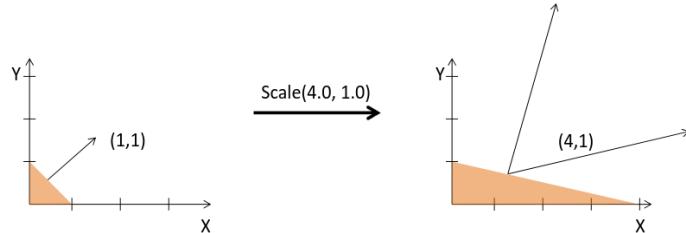
$$z_n \cdot 0,5 + 0,5$$

## 1.3. Vectores

Los vectores se consideran libres y por tanto no están anclados a ningún punto, las transformaciones *modeling transform* no le afectan. Normalmente un vector se transforma hasta que llega a el sistema de coordenadas *Eye Space*. Ahora bien, puede ser que queramos proyectar un vector por pantalla, por tanto ahora no nos sirve que el vector sea libre. Lo que podemos hacer es, sea  $\vec{v}$  el vector en cuestión y  $p$  el punto donde queremos que parta  $\vec{v}$ , podemos definir  $\vec{v}$  como  $\vec{pq}$ . Generamos  $q$  y aplicamos transformaciones geométricas sobre  $p$  y  $q$  hasta llegar a Window Space.

### 1.3.1. Normales

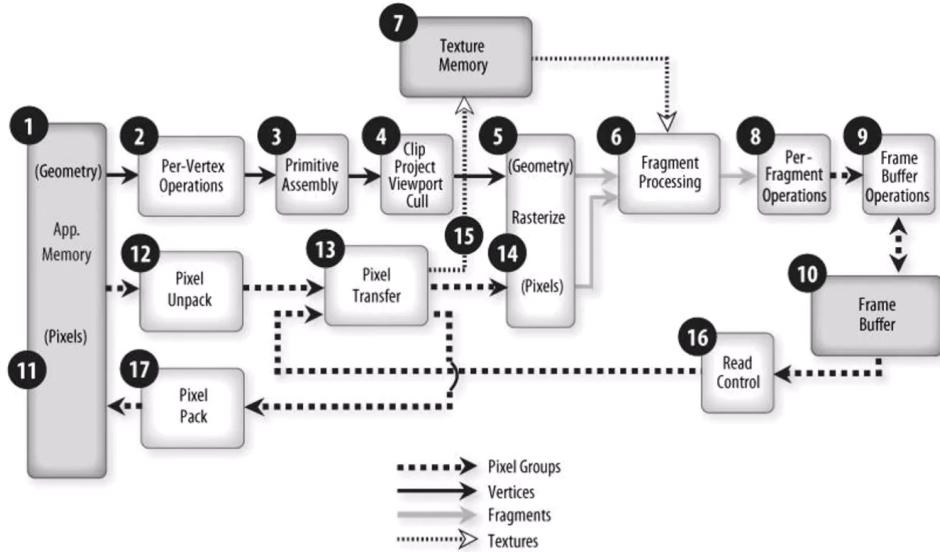
Las transformaciones geométricas con matrices no funcionan bien en vectores normales cuando se aplican escalados no uniformes.



Para poder aplicar correctamente las transformaciones sobre las normales  $\vec{n}$  debemos multiplicar  $\vec{n}$  por una matriz propia. De hecho la matriz en cuestión se trata de la inversa de la transpuesta de la sub-matriz  $3 \times 3$  de la ModelMatrix;

$$\text{normalMatrix} = (\text{modelMatrix}_{3 \times 3})^{-T}$$

## 1.4. Pipeline OpenGL



Desde la CPU existen dos formas de enviar primitivas (vértices, líneas, polígonos):

- Vertices a vértice (modo inmediato) disponible solo en el compatibility profile.
- Usando VertexArrayObjects (VAO).

De vértice a vértice tenemos las funciones

---

`glBegin(), glVertex, glNormal, glEnd`

---

y para VAOs tenemos las funciones

---

`glDrawArrays(mode, first, count)  
glDrawElements(mode, count, type, indices)`

---

Un ejemplo usando vértice a vértice sería

```

1   glBegin(GL_POLYGON);
2   glNormal3f(nx0, ny0, nz0);
```

```

3     glVertex3f(x0, y0, z0);
4     glNormal3f(nx1, ny1, nz1);
5     glVertex3f(x1, y1, z1);
6     ...
7     glEnd();

```

Y uno usando VAOs

```

1 // create buffers
2     glm::vec3 Vertices[...];
3     Vertices[0] = glm::vec3(-1.0, -1.0, 0.0);
4     ...
5     glGenVertexArrays(1, &VAO);
6     glBindVertexArray(VAO);

7
8     glGenBuffers(1, &VBO);
9     glBindBuffer(GL_ARRAY_BUFFER, VBO);
10    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);
11    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

12
13    glBindVertexArray(0);
14 // paintGL
15    glBindVertexArray(VAO);
16    glDrawArrays(GL_TRIANGLES, 0, numVerts);
17    glBindVertexArray(0);

```

Cuando las primitivas salen de la CPU se encuentran en Object Space y entran en el Per-Vertex-Operations donde se le aplican las transformaciones geométricas pertinentes sobre los vertices (model, view, projection) y sobre las normales.

Una vez salen del Per-Vertex-Operations las primitivas se encuentran en Clip Space y entran en Primitive Assembly para que las primitivas se agrupen en función de como hayamos configurado las funciones *glDrawArrays*, *glDrawElements*, etc.

Cuando salen las primitivas (en clip space) ya agrupadas entran en el proceso de ClipProjectViewportCull que consta de 4 pasos:

1. Clipping: Recortado en la pirámide de visión sobre las primitivas que se salen de ella. Las que están parcialmente fuera se eliminan y se generan nuevas primitivas en el borde de la pirámide.
2. Project: Se aplica Perspective Division. Transforma las coordenadas a

Normalized Device Space.

3. Viewport & Depth Transform: Transforma las coordenadas a Window Space.
4. BackFace Culling: si es que está activado.

Cuando las primitivas salen recortadas de ClipProjectViewportCull pasan al proceso de Rasterización en donde se generan fragmentos correspondientes a las primitivas recortadas. OpenGL genera un fragmento para una primitiva si el centro del fragmento está ocupado por esta.

Los fragmentos generados salen del proceso de rasterización y entran en el Pragment Processing donde se calcula el color, además de otras operaciones, sobre los fragmentos.

Los fragmentos salen con sus valores calculados y entran en el proceso de Per-Fragment Operations en donde por cada fragmento se aplican un conjunto de operaciones para determinar si un fragmento es o no visible como son:

- Stencil Test.
- Depth Test.
- Blending.

Estas operaciones no son programables aunque sí configurables.

Finalmente salen los fragmentos visibles y entran en el proceso de Frame Buffer Opeations en donde se modifican los buffers escogidos con *GlDrawBuffers*. También se ve afectado por las llamadas que se hayan hecho a *glColorMask*, *glDepthMask*, *glStencilOp*, etc.

Finalmente se aplica la escritura en el Frame Buffer.

#### 1.4.1. Pipeline Programable

El pipeline programable es la pipeline de openGL que permite programar las operaciones Per-Vertex Operations, en pipeline programable conocido como Vertex Processor, en caso que se active un *vertex program*. Lo mismo ocurre con las operaciones de Per-Fragment Operations, que en pipeline programable se conoce como Fragment Processor, si hay activado un fragment program. el *shader* es el código fuente de un programa y el *program* es el ejecutable del mismo.

# Capítulo 2

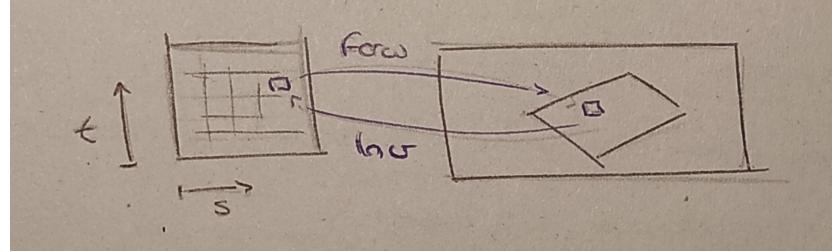
## Texturas

### 2.1. Mapping

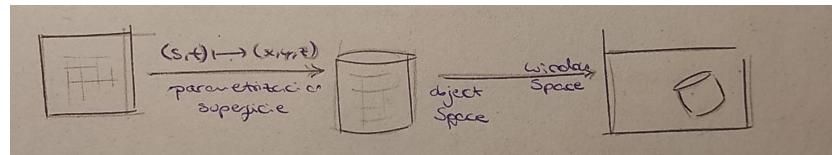
Las funciones de mapping son esas funciones que nos permiten establecer correspondencias entre la textura y la superficie del modelo. Existen 2 tipos de funciones:

- Forward Mapping: Función que asigna al texel de la textura un píxel del modelo.
- Inverse Mapping: Función que le asigna al pixel del modelo un texel de la textura.

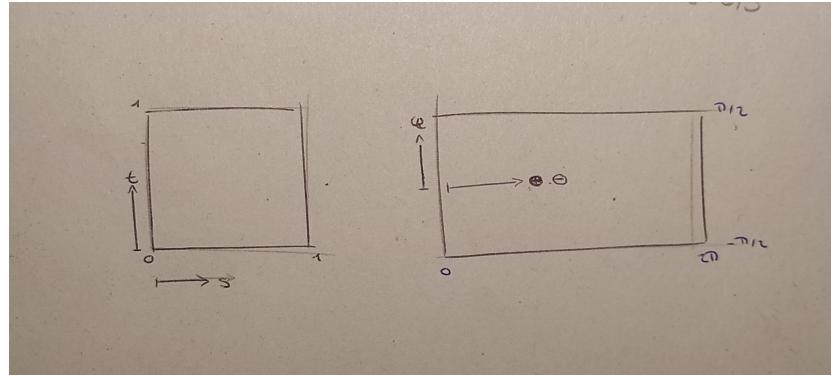
En OpenGL es común aplicar inverse mapping, funciona mejor



La forma mas común de usar forward mapping es sobre objetos simples (cilindros, esferas, ...)



En el caso de forward mapping para una esfera de radio 1 (*S-Mapping*) tenemos coordenadas de textura  $(s, t)$  y queremos transformarlas a coordenadas eulerianas  $(\psi, \theta)$ .



Si nos fijamos en la imagen es tan sencillo como:

$$(\psi, \theta) = \begin{cases} \psi = 2 \cdot \pi \cdot s \\ \theta = \pi(t - 0,5) \end{cases}$$

Una vez las tenemos en coordenadas eulerianas hemos de transformarlas en coordenadas de  $(x, y, z)$  de la esfera. La formula es la siguiente:

$$(x, y, z) = \begin{cases} x = \sin \theta \cdot \cos \psi \\ y = \sin \psi \\ z = \cos \theta \cdot \cos \psi \end{cases}$$

Este tipo de mapping, de rectángulo a esfera, se le suele llamar proyección equirectangular.

En OpenGL cada vértice del objeto suele tener asociada su coordenada de textura. Luego en el proceso de rasterización se interpolan las coordenadas para que cada fragmento las tenga. Una vez los fragmentos tienen su coordenada de textura se aplica inverseMapping para localizar el valor del texel que le pertenece.

```
//VS
in vec2 texCoord;
out vec2 vtexCoord;
void main() {
    vtexCoord = texCoord;
```

```
    gl_Position = modelViewProjectionMatrix * vec4(vertex, 1.0);  
}
```

---

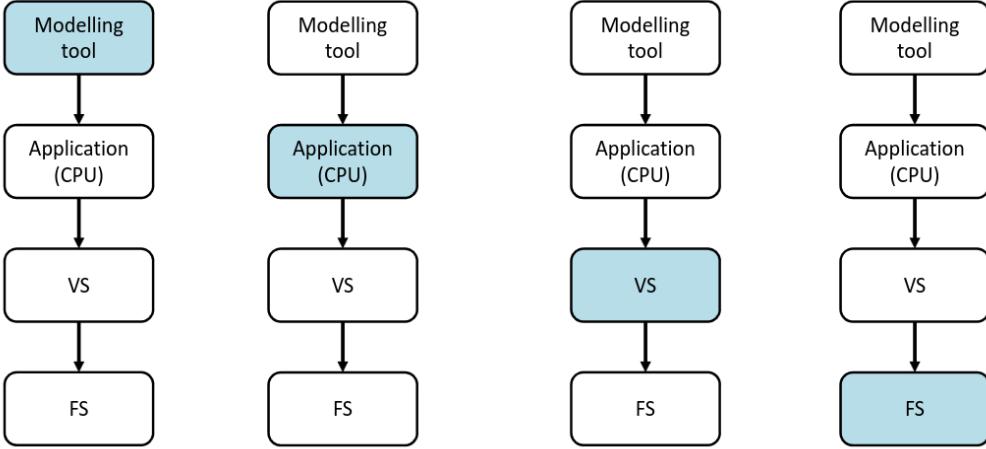
```
//FS  
in vec2 vtexCoord;  
uniform sampler2D colorMap; //textura con el color de la superficie  
void main() {  
    fragColor = texture(colorMap, vtexCoord);  
}
```

---

## 2.2. Pipeline - Coordenadas de textura

Hay 4 fases del pipeline en donde tiene sentido generar coordenadas de textura:

- Modelado (Blender): método habitual, es el que ofrece mas nivel de detalle.
- CPU: Se genera un VBO en donde a partir de los parámetros que se quiera (normalmente a partir de las normales y los vértices del modelo) se computan las coordenadas de textura. Es útil para cuando queremos generar objetos dinámicamente, quads por decir uno.
- Vertice (Vertex Shader): Lo mismo que en la CPU aunque está pensado para casos en que la interpolación es lineal.
- Fragmentos (Fragment Shader): Lo mismo que a nivel de vertice. La única diferencia es que en este caso esta pensado para cuando la interpolación de las coordenadas de fragmento no es lineal.



## 2.3. Generación de coordenadas de textura

### 2.3.1. Planos S, T

Tenemos dos planos  $S(a_s, b_s, c_s, d_s)$  y  $T(a_t, b_t, c_t, d_t)$ . Usando las ecuaciones implícitas del plano podemos llegar a calcular las coordenadas  $(s, t)$ . La idea radica en que  $S$  es el plano que sigue a la coordenada  $s$  y  $T$  el que sigue a la coordenada  $t$  de tal forma que dando las coordenadas  $(x, y, z, w)$  del vértice calcule  $(s, t)$  como:

$$(s, t) = \begin{cases} s = a_s \cdot x + b_s \cdot y + c_s \cdot z + d_s \cdot w \\ t = a_t \cdot x + b_t \cdot y + c_t \cdot z + d_t \cdot w \end{cases}$$

### 2.3.2. Superficies auxiliares

La técnica se basa en usar una superficie auxiliar para acceder a la coordenada de textura. Si dicha superficie auxiliar fuese una esfera necesitaríamos hacerlo dos pasos:

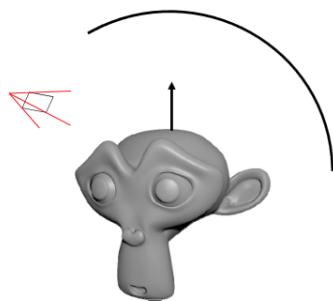
1.  $O$ -Mapping: transformamos las coordenadas  $(x, y, z, 1)$  del modelo en coordenadas  $(x', y', z')$  de la esfera
2.  $S^{-1}$ -Mapping: transformamos las coordenadas  $(x', y', z')$  de la esfera en coordenadas  $(s, t)$  de la textura.

$$\left. \begin{array}{l} x = \sin \theta \cdot \cos \psi \\ y = \sin \psi \\ z = \cos \theta \cdot \cos \psi \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} \psi = \arcsin y \\ \frac{x}{z} = \frac{\sin \theta}{\cos \theta} \cdot \frac{\cos \psi}{\cos \psi} \Rightarrow \frac{x}{z} = \frac{\sin \theta}{\cos \theta} \Rightarrow \theta = \arctan(x/z) \end{array} \right.$$

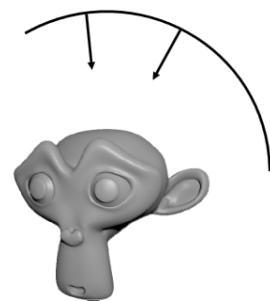
La función arctan devuelve valores en el rango  $[0, \pi]$ . Sucede que necesitamos que  $\theta$  esté en el rango  $[0, 2\pi]$ . Por suerte muchos lenguajes ya tienen funciones de la arco-tangente tales que  $\text{atan2}(x, z) \in [0, 2\pi]$ .

Existen varias implementaciones de *O-Mapping* para encontrar las coordenadas de la esfera:

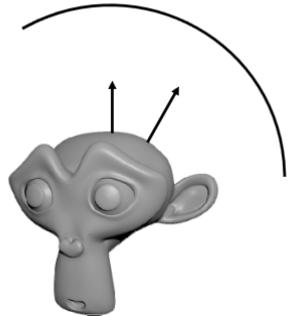
- Reflected View Ray: Usando el vector reflejado de la cámara y el punto de la superficie.
- Intermediate Surface Normal: Usando las normales de la superficie auxiliar.
- Object Normal: Usando las normales del objeto.
- Object Centroid: Se usa el centroide del objeto.



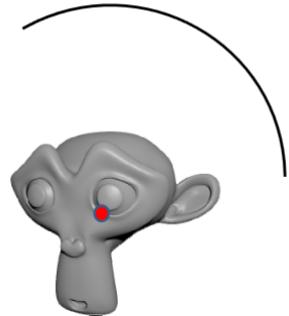
Reflected view ray



Intermediate surface normal



Object Normal



Object centroid

## 2.4. Creación de texturas

```
1 // Load Texture (oncse)
2 QImage img0("fieldstone.png");
3 QImage T = img0.convertToFormat(QImage::Format_ARGB32);
4 glGenTextures( 1, &textureId0); // Generamos un espacio de textura
5 // apuntado por textureId0
6 glBindTexture(GL_TEXTURE_2D, textureId0); // Generamos una textura
7 // (2D) en textureId0
8 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, T.width(), T.height(), 0,
9 GL_RGBA, GL_UNSIGNED_BYTE, T.bits()); // Configuramos la textura
10 // void glTexImage2D(
11 //     GLenum target, // GL_TEXTURE_2D
12 //     GLint level, // nivel de detalle (0 es el mas alto)
13 //     GLint internalformat // numero de componentes de la textura
14 //             // (GL_RGB => (r, g, b))
15 //     GLsizei width, // ancho de la textura
16 //     GLsizei height, // alto de la textura
17 //     GLint border, // siempre a 0
18 //     GLint format, // formato de los datos
19 //     GLenum type, // tipo de datos
20 //     const GLvoid *data // puntero al binario con los valores de textura
21 );
22 // Bind textures, set uniforms...
23 g.glActiveTexture(GL_TEXTURE0); // activamos la textura GL_TEXTURE_0
24 g glBindTexture(GL_TEXTURE_2D, textureId0); // usamos la textura
25 // textureId0
26 program->bind();
27 program->setUniformValue("colorMap", 0); // le pasamos la textura
```

## 2.5. Filtrado de texturas

Buscamos reconstruir la imagen de textura ya que se dan dos problema:

- Magnification: el espacio de la pantalla asignado al texel es mayor que el tamaño del texel.
- Minification: el espacio de la pantalla asignado al texel es menor que el tamaño del texel.

### 2.5.1. Magnification Filter

Para aplicar los filtros hemos de interpretar que el valor de textura para un texel solo es aplicable al centro del mismo, los demás son indeterminados.

- **GL\_LINEAR** (nearest neighbour sampling): Todo punto dentro del texel tiene asociado el valor del centro.

---

```
glTexParameterf(TEXTURE_2D, GL_GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

---

- **GL\_NEAREST** (bilinear interpolation): El valor del fragmento se interpola entre los centros más próximos a él.

---

```
glTexParameterf(TEXTURE_2D, GL_GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

---

### 2.5.2. Minification Filter

Un MipMap es un conjunto de texturas que representan la misma textura pero cada una con un nivel de resolución distinto. Normalmente la resolución de los MipMaps suelen ser potencias de 2. (por ejemplo  $256 \times 256\text{px}$ ,  $128\text{px}$ , ...,  $1 \times 1\text{px}$ ).

Lo que necesitamos es una función que nos calcule un valor  $\lambda$  que indique el LOD (*Level-Of-Detail*) necesario para representar la textura.

Sean  $(u, v)$  las coordenadas de textura en pixel y  $(x, y)$  coordenadas en window

space sabemos que hay una relación directamente proporcional entre el LOD y  $\partial u / \partial x$ ,  $\partial v / \partial y$ ,  $\partial u / \partial y$ ,  $\partial v / \partial x$  (desplazamiento de u y v en función de x e y). De hecho para encontrar la lambda solo tenemos que encontrar una función que a partir de estas derivadas parciales nos devuelva un valor  $rho$  tal que  $\lambda = \log_2(rho)$ .

Existen varias opciones para configurar como actuar ante Minification, sin MipMapping:

- **GL\_LINEAR**
- **GL\_NEAREST**

Estas opciones no suelen funcionar bien ya que solo usa los valores de LOD0 (Nivel de detalle basico (mas alto)). Sin embargo también hay usando Mip-Mapping:

- **GL\_NEAREST\_MIPMAP\_NEAREST**: GL\_NEAREST pero ajustado a LOD  $\lfloor \lambda \rfloor$ .
- **GL\_LINEAR\_MIPMAP\_NEAREST**: GL\_LINEAR pero ajustado a LOD  $\lfloor \lambda \rfloor$ .
- **GL\_NEAREST\_MIPMAP\_LINEAR**: GL\_NEAREST pero interpolando entre LOD  $\lfloor \lambda \rfloor$  y LOD  $\lceil \lambda \rceil$  en función de  $fract(\lambda)$ .
- **GL\_LINEAR\_MIPMAP\_LINEAR**: GL\_LINEAR pero interpolando entre LOD  $\lfloor \lambda \rfloor$  y LOD  $\lceil \lambda \rceil$  en función de  $fract(\lambda)$ .

### 2.5.3. Wrapping

las funciones de Wrapping definen el comportamiento de los fragmentos cuando la coordenada de textura se sale del intervalo  $[0, 1]^2$ .

Hay 2 opciones:

- **GL\_REPEAT**: se hace overflow de la coordenada tal que  $texCoord = (fract(s), fract(t))$
- **GL\_CLAMP\_TO\_EDGE**: Coge la parte derecha del ultimo texel (la mitad del texel) y lo copia indefinidamente a lo largo de la superficie.

La opción por defecto de OpenGL es **GL\_REPEAT**

## 2.6. Interpolación de coordenadas de textura

Se hace en 2 fases del pipeline:

- Recortado de vertices (en clip space)
- Rasterización: Se aplica interpolación lineal sobre todos los valores del vértice.

### 2.6.1. Perspective Correct Interpolation

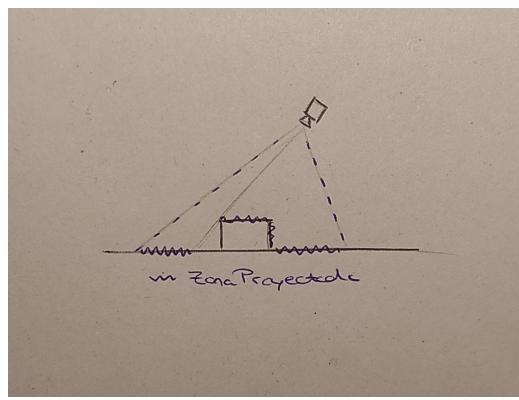
Si se asignasen coordenadas de textura después de la división de perspectiva interpolación sería ser lineal. Sucede que la interpolación es lineal y por tanto hay dos métodos para que la interpolación resulte correcta:

- Interpolar ( $s, t$ ) con las coordenadas de textura en ObjectSpace (sirve tambien EyeSpace, WorldSpace, ClipSpace y cualquier sistema previo a la division de perspectiva).
- interpolamos ( $s, t$ ) con coordenadas  $(s', t', p', q')$  tal que  $(s'/q', t'/q') \in [0, 1]^2$ .

La segunda opción se ve una aplicación a continuación

## 2.7. Projective Texture Mapping

Projective Texture Mapping es una técnica que nos permite proyectar texturas sobre superficies (similar a un proyector). Para ello el *proyector* debe tener su propia ViewMatrix y ProjectionMatrix.



El algoritmo a grandes rasgos seria el siguiente:

En el Vertex Shader pasariamos las coordenadas en ObjectSpace a coordenadas

en ClipSpace  $[-1, 1]^3$  y de ClipSpace a coordenadas en el rango  $[0, 1]^3$ . Sea  $T(x)$  una función de translación y  $S(x)$

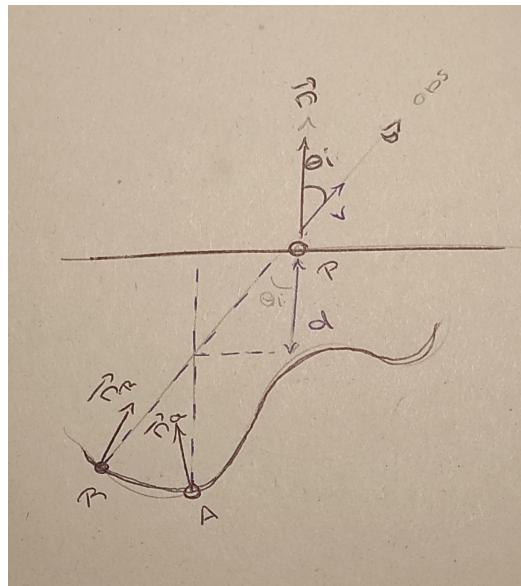
$$T(0'5) \cdot S(0'5) \cdot P_M \cdot V_M \cdot M_M \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} s \\ t \\ p \\ q \end{bmatrix} \text{ t.q. } \begin{bmatrix} s/q \\ t/q \end{bmatrix} \in [0, 1]^2$$

Y en el Fragment Shader accedemos a la textura usando las coordenadas  $(s/q, t/q)$ .

## 2.8. Aplicaciones de texturas

### 2.8.1. Parallax Mapping

Sirve para calcular la normal aproximada que debería tener la superficie en ese punto. Sea  $P$  el punto de la superficie y  $d$  el valor del HeightMap asociado a  $P$  calculamos el angulo incidente  $\theta_i$  entre la normal de  $P$  ( $\vec{n}$ ) y el observador  $\vec{V}$ . Con  $\theta_i$  y  $d$  encontramos la distancia  $\Delta x$ , que se supone es la distancia en el eje  $x$  que está  $P$  del punto  $R$ , que es el punto que vería el observador en ese píxel.



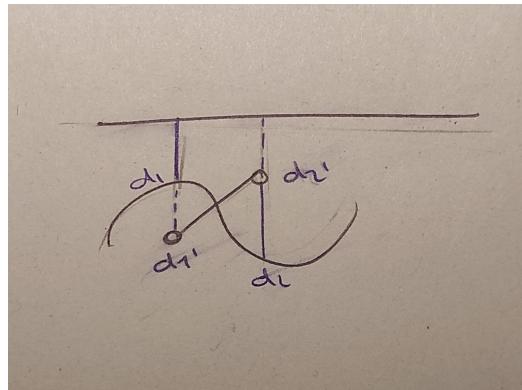
Ahora bien, esto es una aproximación y por tanto hay errores. En el ejemplo de arriba vemos que el punto seleccionado es  $A$  y no  $R$ . También hay otro

error asociado a ParallaxMapping y es que no tiene en cuenta las occlusiones que puede hacer el propio objeto.

### 2.8.2. Relief Mapping

Relief Mapping hace lo que ParallaxMapping pero sin errores. Lo que hace es trazar un rayo desde el observador en dirección a  $P$  y calcular el punto de intersección con la superficie ficticia (la que se supone tiene a ojos de la pantalla).

Para encontrar el punto de intersección  $R$  lo que se hace es ir trazando pequeños segmentos desde el observador de tal forma que, sea  $d1$  y  $d2$  el valor del heightMap en dos puntos del rayo y  $d1'$  y  $d2'$  la distancia de los puntos con la superficie, cuando  $d1 \neq d1'$  y  $d2 \neq d2'$  entonces hay una intersección. Si quiere encontrar el punto cercano entonces lo que hace es aplicar algun tipo de búsqueda dicotómica.



### 2.8.3. Displacement Mapping

Displacement Mapping genera geometría detallada desplazando los una vez subdivididos en segmentos. Es el mas realista pero obliga a generar muchos vértices. Es el mas costoso.

# Capítulo 3

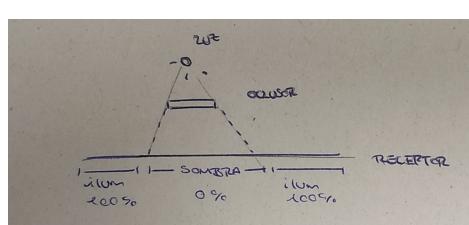
## Sombras

### 3.1. Introducción

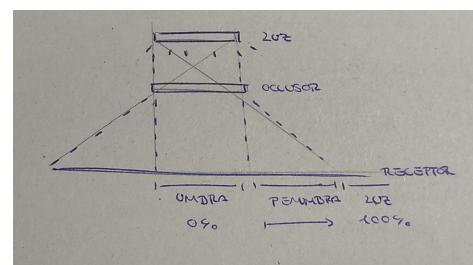
Existen dos tipos de sombras:

- Sombras generadas por luz puntual.
- Sombras generadas por luz no puntual (e.j. luminaria).

En una sombra generada por una luz puntual, la parte que no recibe luz, también conocida como umbra, no recibe irradiancia mientras que la parte que sí la recibe recibe el 100 %. Por otro lado, la luz no puntual además de la zona iluminada y la umbra también hay la penumbra en donde la luz que irradia la zona esta en el rango (0 %, 100 %).



Luz Puntual



Luz no Puntual

Existen distintos tipos de algoritmos para generar sombras:

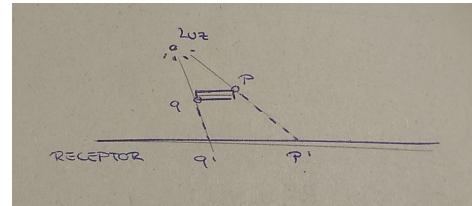
- Sombras con proyección
- Shadow Volume

- Shadow Mapping
- Sombras precalculadas con LightMap

### 3.2. Sombras por proyección

La idea detrás de sombras por proyección se puede resumir en proyectar el oclusor sobre el plano del receptor. Las precondiciones de la técnica son:

- La luz es puntual.
- El receptor es una superficie.



#### Versión sin Stencil Test

El algoritmo de sombras por proyección sin stencil es:

1. Dibujar el receptor.
2. Dibujar la sombra proyectada.

Como el receptor es un plano la proyección se puede calcular como una matriz  $4 \times 4$ .

3. Dibujar el oclusor.

Al estar la sombra en el mismo plano que el receptor pueden haber problemas de *z-fighting* en el zBuffer. Existen varias formas de solucionarlo. La más simple aunque no recomendada es desactivar el *Depth Test* al momento de dibujar la sombra.

---

```

1 // 1. Dibujar receptor
2   dibujar(receptor)
3 // 2. Dibujar el oclusor proyectado
4   glDisable(GL_LIGHTING); // Desactivamos la iluminación para que se
                           // pinte como una sombra
5   glDisable(GL_DEPTH_TEST); // Evitamos z-fighting
6   glMatrixMode(GL_MODELVIEW);
7   glPushMatrix(); // duplicamos la modelViewMatrix
8   glMultMatrix(matrizProyeccion); // Multiplicamos la modelViewMatrix
                                   // por la matriz de proyección
9   dibujar(oclusor);
10
11

```

```

12     glPopMatrix(); // Retiramos la modelViewMatrix modificada
13
14 // 3. Dibujar el oclusor
15     glEnable(GL_LIGHTING);
16     glEnable(GL_DEPTH_TEST);
17     dibujar(oclusor);

```

---

Sin embargo hay otra forma de garantizar que la sombra se vea por encima del receptor y es usando la función *glPolygonOffset(factor, units)*. *glPolygonOffset* traslada la componente *z* de los vértices en una distancia *ε* en la dirección que queremos. La translación de la componente *z* se puede expresar como:

$$z' = z + \delta z \cdot factor + r \cdot units$$

- *units*: offset constante
- *factor*: offset variable en el ángulo de *z*
- *r*: unidad mínima por la cual no hay *z-fighting*

---

```

glPolygonOffset(1, 1); // offset positivo, aleja la z de la cámara
glPolygonOffset(-1, -1); // offset negativo, acerca la z a la cámara

```

---

```

1 // 1. Dibujar receptor
2     dibujar(receptor)
3 // 2. Dibujar el oclusor proyectado
4     glDisable(GL_LIGHTING);
5     glEnable(GL_POLYGON_OFFSET_FILL); // Activamos glPolygonOffset
6     glPolygonOffset(-1, -1) // Acercamos la sombra a la cámara
7     glMatrixMode(GL_MODELVIEW);
8     glPushMatrix();
9     glMultMatrix(matrizProyección);
10    dibujar(oclusor);
11    glPopMatrix();
12
13 // 3. Dibujar el oclusor
14     glEnable(GL_LIGHTING);
15     glDisable(GL_POLYGON_OFFSET); // Desactivamos glPolygonOffset
16     glEnable(GL_DEPTH_TEST);
17     dibujar(oclusor);

```

---

Este método tiene un problema. La sombra se proyecta sobre el plano de la superficie, si la sombra se sale de la superficie del receptor esta se vera igualmente.

## Versión con Stencil Test

Para solucionar el problema de la versión sin Stencil Test está la versión con Stencil Test.

Para configurar el test de stencil tenemos la función *glStencilFunc*:

---

```
glStencilFunc(comp, ref, mask);
```

---

- *comp*: comparación que hará el test de Stencil.

El valor de *comp* puede ser, entre otros: GL\_ALWAYS, GL\_NEVER, GL\_EQUAL, GL\_LESS, GL\_LESS\_EQUAL, ...

- *ref*: valor que le pasaremos al test para evaluar.
- *mask*: mascara de bits que se aplica a *ref*.

Para configurar qué hacer en caso que superemos el test de Stencil (o el test de zBuffer) tenemos la función *glStencilOp*:

---

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

---

- *fail*: falla el test de Stencil.
- *zfail*: pasa el test de Stencil pero falla el zTest.
- *zpass*: pasa el test de Stencil y el zTest.

Los valores que pueden tener los tres parametros son: GL\_KEEP, GL\_ZERO, GL\_INCR, GL\_DECR, GL\_INVERT y GL\_REPLACE.

El algoritmo con Stencil se puede resumir en:

1. Dibujamos el receptor en el colorBuffer y en el StencilBuffer.
2. Limpiamos del StencilBuffer los pixels donde haya sombra.
3. Volvemos a dibujar el receptor.

#### 4. Dibujamos el oclusor.

```
1 // 1. Dibujar el receptor
2 glEnable(GL_STENCIL_TEST);
3 glStencilFunc(GL_ALWAYS, 1, 1); // Pasa siempre el StencilTest
4 glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
5 dibujar(receptor);
6 // 2. Limpiar del StencilBuffer la sombra
7 glDisable(GL_DEPTH_TEST);
8 glColorMask(GL_FALSE, ..., GL_FALSE); // desactivamos la escritura
9 // en el ColorBuffer
10 glStencilFunc(GL_EQUAL, 1, 1); // Pasamos el test si es zona a 1
11 glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO); // Limpiamos el StencilBuffer
12 glPushMatrix(); glMultMatrix(matrizProyección);
13 dibujar(oclusor);
14 glPopMatrix();
15 // 3. Dibujar la parte sombreada del receptor
16 glEnable(GL_DEPTH_TEST)
17 glDepthFunc(GL_LEQUAL); // las zetas son las mismas
18 glColorMask(GL_TRUE, ..., GL_TRUE); // Reactivamos la impresion en
19 // el ColorBuffer
20 glDisable(GL_LIGHTING); // Se nos dibuja oscuro
21 glStencilFunc(GL_EQUAL, 0, 1);
22 dibujar(receptor);
23 // 4. Dibujar el oclusor
24 glEnable(GL_LIGHTING);
25 glDeptFunc(GL_LESS); // Reestablecemos el depthTest
26 glDisable(GL_STENCIL_TEST);
27 dibujar(oclusor)
```

### 3.2.1. Matriz de Proyección

Para calcular la matriz de proyección hemos de pensar que el punto proyectado  $p'$  es aquel que está tanto en la recta que forman el punto original y la fuente de luz  $\vec{LP}$  como en el plano que representa la superficie  $(a, b, c, d)$ .

## Matriz de proyección respecto el origen

Dados los coeficientes del plano  $(a, b, c, d)$  y una luz situada en el origen el punto proyectado es aquel tal que:

$$p' \in \text{plano} \Rightarrow a \cdot p'_x + b \cdot p'_y + c \cdot p'_z + d = 0$$

$$p' \in \vec{OP} \Rightarrow p' = \lambda \cdot p$$

Por tanto deducimos:

$$\lambda = \frac{-d}{a \cdot p'_x + b \cdot p'_y + c \cdot p'_z} = \frac{-d}{\vec{n} \cdot p}$$

$$p' = \frac{-d}{\vec{n} \cdot p} \cdot p$$

Y la matriz de proyección se calcula como:

$$\begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ a & b & c & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} -d \cdot p_x \\ -d \cdot p_y \\ -d \cdot p_z \\ a \cdot p'_x + b \cdot p'_y + c \cdot p'_z \end{bmatrix}$$

## Matriz de proyección respecto un punto $(x, y, z)$

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -(d+ax+by+cz) & 0 & 0 & 0 \\ 0 & -(d+ax+by+cz) & 0 & 0 \\ 0 & 0 & -(d+ax+by+cz) & 0 \\ a & b & c & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -d - by - cz & xb & xc & xd \\ ya & -d - ax - cz & yc & yd \\ za & zb & -d - ax - by & zd \\ za & zb & xc & -ax - by - cz \end{bmatrix}$$

## Matriz de proyección respecto una dirección $(x, y, z)$

$$\begin{bmatrix} by + cz & -bx & -cx & -dx \\ -ay & ax + cz & -cy & -dy \\ -az & -bz & ax + by & -dz \\ 0 & 0 & 0 & ax + by + cz \end{bmatrix}$$

### 3.3. Shadow Volume

Precondiciones:

- El oclusor es **sencillo**.
- La luz es puntual.

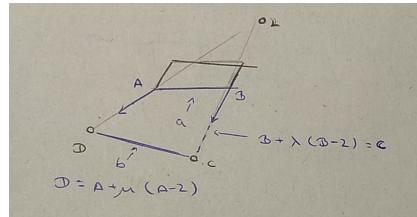
El Shadow Volume entre un objeto oclusor  $o$  y una fuente de luz  $f$  se define como el conjunto  $SV(o, f)$  de todos los puntos  $p$  del espacio tales que el segmento entre  $p$  y  $f$  intersecta con  $o$ .

Pese a que  $SV(o, f)$  es semi-infinito solo nos importan los puntos que están dentro de la escena y el frustum de visión.

Para poder generar el volumen de sombra nos es necesario conocer las aristas contorno del oclusor respecto la luz. Una arista es una arista contorno si, de las dos caras que tienen la arista, una cara es *Back Face* y la otra es *Front Face* respecto la cámara.

$$a \in \text{aristaContorno} \iff |\{c \in \text{BackFace} | a \in c\}| = 1 \quad \& \quad |\{c \in \text{FrontFace} | a \in c\}| = 1$$

Una vez obtenemos las aristas contorno generamos las caras laterales del volumen de sombra.



Cara Lateral de un Shadow Volume

Ya conseguidas las caras del Shadow Volume tocaría mirar si un punto está o no dentro del volumen de sombras. Una forma de determinarlo es lanzar un rayo arbitrario desde el punto y calcular el número de intersecciones con el polígono. Cuando el número de intersecciones es par se cumple que el punto está fuera del polígono, cuando es impar está fuera.

$$p \in o \iff \exists \vec{v} \text{ t.q } \#\text{intersecciones}(p + \lambda \vec{v}, o) \in \mathbb{Z}$$

Ya que cualquier rayo nos sirve vamos a optar por lanzar el rayo en dirección al observador. Contar el numero de intersecciones puede llegar a ser muy costoso. Una alternativa menos costosa puede darse usando el test de Stencil. Según el observador salimos si atravesamos una cara *FrontFace* y entramos cuando atravesamos una cara *BackFace*.

$$|intersecciones(FrontFace)| - |intersecciones(BackFace)| > 0 \Rightarrow p \in SV(o, f)$$

El algoritmo de Shadow Volume a grandes rasgos se basaria en:

1. Dibujar la escena en el ZBuffer.
2. Dibujar las caras frontales del shadow volume, por cada una aumentar en uno el valor del píxel en el StencilBuffer (si pasa todos los tests).
3. Dibujar las caras traseras del shadow volume, por cada una restar uno al valor del píxel en el StencilBuffer.
4. Dibujar la parte sombreada de la escena (la que tiene el stencilBuffer a 1).
5. Dibujar la parte iluminada de la escena.

---

```

1 // 1. Dibujar la escena al z-buffer
2     glColorMask(GL_FALSE, ..., GL_FALSE); Desactivamos pintar por color
3     dibujar(escena);
4 // 2.Dibujar en el stencil les caras frontales del volumen
5     glEnable(GL_STENCIL_TEST); // Activamos test de Stencil
6     glDepthMask(GL_FALSE); // Desactivamos escritura en el DepthBuffer
7     glStencilFunc(GL_ALWAYS, 0, 0); // Pasamos el test siempre
8     glEnable(GL_CULL_FACE); // Activamos FaceCulling
9     glStencilOp(GL_KEEP, GL_KEEP, GL_INCR); // Incrementamos si pasamos
10                                // todos los tests
11    glCullFace(GL_BACK); // Definimos BackFaceCulling
12    dibujar(shadowVolume);
13 // 3.Dibujar en el StencilBuffer las caras traseras del volumen
14    glStencilOp(GL_KEEP, GL_KEEP, GL_DECR); // Decrementamos si pasamos
15                                // todos los tests
16    glCullFace(GL_FRONT); // Definimos FrontFaceCulling
17    dibujar(shadowVolume);
18 // 4. Dibujamos en el colorBuffer la zona ensombrecida
19    glDepthMask(GL_TRUE); // Reactivamos pintado el DepthBuffer
20    glColorMask(GL_TRUE, ... , GL_TRUE);      // Reactivamos pintado

```

```

21 // en el colorBuffer
22 glCullFace(GL_BACK); // Reestablecemos BackFaceCulling
23 glDepthFunc(GL_LEQUAL); // Test ZBuffer (regla <=) ya que ya estan
24 // las mismas coordenadas en el ZBuffer
25 glColorMask(GL_KEEP, GL_KEEP, GL_KEEP); // No tocamos el stencil buffer
26 glStencilFunc(GL_EQUAL, 1, 1); // Solo pasamos el stencilTest si
27 // el valor del Buffer es 1
28 glDisable(GL_LIGHTING); // Desactivamos luz para que haya sombra
29 dibujar(escena);
30 // 5. Dibujamos en el color buffer la zona iluminada
31 glColorMask(GL_KEEP, GL_KEEP, GL_KEEP); // Solo pasamos el stencilTest si
32 // el valor del Buffer es 0
33 glEnable(GL_LIGHTING); // Activamos la luz
34 dibujar(escena);
35 // 6. Reseteamos todo
36 glDepthFunc(GL_LESS); // Norma por defecto del DepthTest
37 glDisable(GL_STENCIL_TEST); // Desactivamos StencilTest

```

### 3.3.1. FrontFace y BackFace

Hay 2 formas de determinar si una cara es frontFace o backFace respecto a un punto.

La primera es calculando si el punto está o no en el semi-espacio positivo del plano. Sea  $p$  el punto y  $\pi = (a, b, c, d)$  el plano, la distancia entre el punto y el plano se calcula como  $dist(p, \pi) = a \cdot p_x + b \cdot p_y + c \cdot p_z + d$ . Si  $dist(p, \pi) > 0$  entonces  $p$  está en el semi-espacio positivo, en caso que  $dist(p, \pi) < 0$  entonces  $p$  se encuentra en el semi-espacio negativo.

La otra forma es calcular el grado incidente entre, sea  $q$  un punto del plano, el ángulo incidente entre la normal del plano  $\vec{n}$  y  $\vec{pq}$ . Si  $\vec{n} \cdot \vec{pq} > 0$  entonces hablamos de una cara frontFace, y si  $\vec{n} \cdot \vec{pq} < 0$  entonces hablamos de una cara backFace.

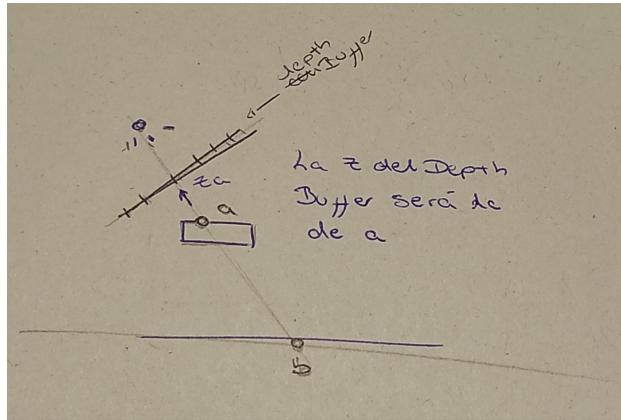
En los dos casos no hemos visto que ocurre cuando el valor es 0. Esto es porque en cualquiera de ellos significa que  $p$  es coplanar a  $\pi$  y por tanto la cámara no podrá mostrarlo.

## 3.4. Shadow Mapping

Shadow Mapping es la técnica más utilizada ya que no distingue entre oclu-sor y receptor. La precondición de la técnica es que la luz sea puntual y, para

Shadow Mapping básico, funcione como un spotlight (no sea unidireccional).

Supongamos que tenemos una cámara situada en la fuente de luz, entonces en la cámara no se mostrará sombra alguna. Todos los puntos de la escena serán puntos iluminados y el resto de elementos que no se muestran ya que **no han pasado el DepthTest** estarán a la sombra. Lo que podemos hacer es en vez de visualizar la escena a color, veremos el valor de las z's y guardaremos estos en una textura. Por tanto tendremos un DepthMap  $\approx$  ShadowMap de la escena con la cámara en la fuente de luz.



El algoritmo se divide en dos pasos:

1. Dibujar la escena y guardarse el DepthBuffer en una textura.
2. Redibujar la escena y usar shaders para implementar ShadowMapping.

Para un punto  $p = (x, y, z, w)$  debemos preguntarnos si está o no a la sombra. Para saberlo una opción es, a partir de las coordenadas de clip space (NDC)  $\in [-1, 1]^3$ , transformarlas a coordenadas de textura  $\in [0, 1]^3$  y así poder consultar si la coordenada  $z$  es o no la del ShadowMap. Sea  $T$  una matriz de translación y  $S$  una matriz de escalado podemos expresar la función que nos devuelve las coordenadas de textura como:

$$T(0,5) \cdot S(0,5) \cdot \text{modelViewProjectionMatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} s \\ t \\ p \\ q \end{bmatrix}$$

en donde  $s/q$  es la coordenada  $s$  de textura,  $t/q$  es la coordenada  $t$  de textura y  $p/q$  el valor  $z$  que queremos comparar. Cabe decir que esto se hace en el vertexShader.

En el fragment shader compararemos el valor  $p/q$  de nuestro punto  $p$  con el valor  $z$  que se encuentra en la coordenada  $(s/q, t/q)$  de la textura. Si tienen un valor parecido entonces será el punto iluminado, en caso contrario estará a la sombra.

---

```
if (texture(shadowMap, vec2(s/q, t/q)) >= p/q) ilumina();
else ensombrece(); // Asumimos que no hay error de aproximación
```

---

El código mas a detalle sería, la parte en openGL:

```
1 // 1. Definimos la escena con la camara en el punto de luz
2     glViewport( 0, 0, SHADOW_MAP_WIDTH, SHADOW_MAP_HEIGHT); // Tamaño de
3                                         // la textura
4     glMatrixMode(GL_PROJECTION);
5     glLoadIdentity(); // Cargamos una matriz identidad en la pila
6     gluPerspective( fov, ar, near, far);      // projectionMatrix de la
7                                         // fuente de luz
8     glMatrixMode( GL_MODELVIEW );
9     glLoadIdentity();
10    gluLookAt( lightPos, ..., lightTarget, ..., up, ...); // modelView-
11                                         // Matrix de la
12                                         // fuente de luz
13 // 2. Dibujamos la escena
14     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ); // Limpiamos la
15                                         // pantalla
16     glPolygonOffset(1,1); glEnable(GL_POLYGON_OFFSET_FILL); // Alejamos la
17                                         // sombra un poco para
18                                         // que no haya error
19                                         // de aproximacion
20     dibujar(escena);
21     glDisable(GL_POLYGON_OFFSET_FILL); // Desactivamos el glPolygonOffset
22 // 3. Guardamos la textura
23     glBindTexture(GL_TEXTURE_2D, textureId);
24     glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0, SHADOW_MAP_WIDTH,
25     SHADOW_MAP_HEIGHT);
26 // Restaurar cámara y viewport
27     glViewport(0, 0, SCENE_WIDTH, SCENE_HEIGHT);
28     glMatrixMode(GL_PROJECTION);
29
```

---

---

```

1 // Generación de coords de textura para el shadow map
2 // La generación es similar a projective texture mapping
3     glLoadIdentity();
4     glTranslated( 0.5, 0.5, 0.5 );
5     glScaled( 0.5, 0.5, 0.5 );
6     gluPerspective( fov, ar, near, far);
7     gluLookAt( lightPos, ... lightTarget, ... up... );

```

---

En el Vertex Shader:

---

```

uniform mat4 lightMatrix;
out vec4 texCoord;
void main()
{
    ...
    texCoord = lightMatrix*vec4(vertex,1);
    gl_Position = modelViewProjectionMatrix * vec4(vertex,1);
}

```

---

y en el FragmentShader:

---

```

...
    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 = ... // iluminat
    else
        fragColor = ... // a l'ombra
...

```

---

Al usar texturas ShadowMapping puede tener errores de resolución y dibujar las sombras como dientes de sierra.

# Capítulo 4

## Reflexiones Especulares

### 4.1. Reflexiones basadas en objetos virtuales

#### 4.1.1. Reflexiones (sin *Stencil Test*)

El algoritmo a nivel básico se basa en 3 pasos:

1. Dibujar los objetos en posición virtual. Para ello usamos una matriz de reflexión.
2. Dibujar el espejo (*quad*) semi-transparente (*Alpha Bleeding*). Este paso es opcional.
3. Dibujar la escena con los objetos en su posición real.

```
1 // 1. Dibuixar els objectes en posició virtual
2     glPushMatrix();
3     glMultMatrix(matriu_simetria)
4     glLightfv(GL_LIGHT0, GL_POSITION, pos); // Ponemos la luz en posicion
5                                     // virtual
6     glCullFace(GL_FRONT); // Los vertices ahora estan en ClockWise
7     dibuixar(escena);
8     glPopMatrix();
9 // 2. Dibuixar el mirall semi-transparent
10    glLightfv(GL_LIGHT0, GL_POSITION, pos); // Volvemos la luz
11                      // a su posicion normal
12    glCullFace(GL_BACK); // Los vertices vuelven a estar en
13                      // CounterClockWise
14    dibuixar(mirall);
```

```

15 // 3. Dibuixar els objects en posició real
16     dibuixar(escena);

```

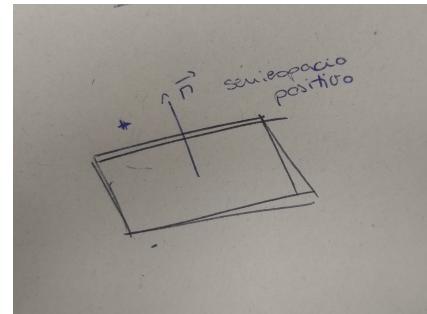
Se puede calcular la matriz de reflexión a partir de la normal  $\vec{n} = (a, b, c)$ . Supongamos que tenemos un espejo en una pared en el punto  $x = 0$ . La normal del espejo se puede expresar como  $\vec{n} = (1, 0, 0)$ . El plano perpendicular a  $\vec{n}$  de la forma  $(a, b, c, d)$  Sabemos que es  $(1, 0, 0, 0)$ . De aquí la matriz de reflexión es una de escalado que nos invierte el valor de  $x$ .

$$\text{Reflexión} = S(-1, 1, 1) = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Esta técnica tiene limitaciones:

- Toda la escena va a ser reflejada.
- No diferencia entre espejos y agujeros

Una posible solución sería pintar solo los objetos virtuales dentro de un plano recortado (*gl\_ClipPlane()*) eliminando todos los elementos que estén en el semi-espacio positivo del plano.



#### 4.1.2. Reflexiones (con *Stencil Test*)

La idea es usar *Stencil Test* para limitar los objetos virtuales en la región limitada por el espejo. Si la pared tiene agujeros no se verán objetos reflejados tras este. El algoritmo se basa en:

1. Dibujar el espejo en el *Stencil Buffer*.
2. Dibujar los objetos en su posición virtual
3. Dibujar el espejo semi-transparente
4. Dibujar los objetos en su posición real

En el primer paso vamos a desactivar el *Color Buffer*, pues no queremos pintar nada en escena, y activamos el *Stencil Buffer* para que cuando dibujemos en el *Stencil Buffer* este se ponga a 1. Si el espejo pasa el test del *Z-Buffer*

entonces se dibujará en el *Stencil Buffer*.

En el segundo paso como solo queremos imprimir los objetos virtuales activaremos el *Color Buffer* y solo dejaremos que se pinten si en el test de *Stencil* tienen el bit a 1.

En OpenGL el código se ve de la siguiente forma:

```
1 // 1.Dibujamos el espejo en el Stencil Buffer
2     glEnable(GL_STENCIL_TEST);
3     glStencilFunc(GL_ALWAYS, 1, 1); // Pone los elementos del buffer
4                                     // a 1 siempre
5     glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE)    // Si pasa el stencil y
6                                     // zBuffer pone el
7                                     //buffer a 1
8     glDepthMask(GL_FALSE); glColorMask(GL_FALSE); // No pintamos ni en el
9                                     // zBuffer ni el el
10                                    // ColorBuffer
11     Dibujar(espejo);
12 // 2.Dibujamos los objetos virtuales
13     glDepthMask(GL_TRUE); glColorMask(GL_TRUE); // Ahora si pintamos
14     glStencilFunc(GL_EQUAL, 1, 1)   // Pasamos el StencilTest si
15                                     // el pixel es = 1
16     glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
17     glPushMatrix(); glMultMatrix(matrizReflexion);
18     glLightfv(GL_LIGHT0, GL_POSITION, pos);
19     glCullFace(GL_FRONT);
20     dibujar(escena);
21     glPopMatrix();
22 // 3.Dibujamos el espejo
23     glDisable(GL_STENCIL_TEST); // Desactivamos el stencil test
24     glLightfv(GL_LIGHT0, GL_POSITION, pos);
25     glCullFace(GL_BACK);
26     dibujar(espejo);
27 // 4.Dibujamos los objetos reales
28     dibujar(escena)
```

#### 4.1.3. Texturas Dinámicas

La idea de la implementación es la siguiente:

1. Dibujamos los objetos en posición virtual.
2. Creamos una textura de toda la escena (o de solo la región del espejo).

### 3. Dibujamos el espejo usando *Texture Mapping*

```
vec2 size = vec2(width, height);
fragColor = texture(colorMap, g1_FragCoord.xy/size);
```

### 4. Dibujamos la escena con los objetos reales.

Sabemos que con una textura cualquiera las coordenadas  $(s, t) \in [0, 1]^2$ . Por tanto dividimos las coordenadas en *window space* por el ancho y alto de la pantalla.

Con texturas podemos aprovechar que si en una escena estamos cambiando las propiedades de un objeto pero no afecta a la escena reflejada en el espejo no tendremos que rehacer la textura.

Por último queda decir que para un plano  $(a, b, c, d)$  la matriz de reflexión se calcula como:

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

## 4.2. Environment Mapping

Environment Mapping permite reflexión en superficies de cualquier tipo. Dada una dirección unitaria arbitraria  $R \in \mathbb{R}^3$ , *Environment Map* nos devuelve el color del entorno.

$$color = environmentMap(R)$$

$$R = (R_x, R_y, R_z)$$

$$R = (\theta, \psi, 1)$$

*Environment Map* puede ser tanto una textura procedural como una textura. En el caso de la figura siguiente, para  $R_x \approx 1$  o  $\theta \approx 90^\circ$  nos encontraríamos alrededor del 75% de la textura. para  $R_z \approx 1$  nos encontraríamos en uno de los extremos.

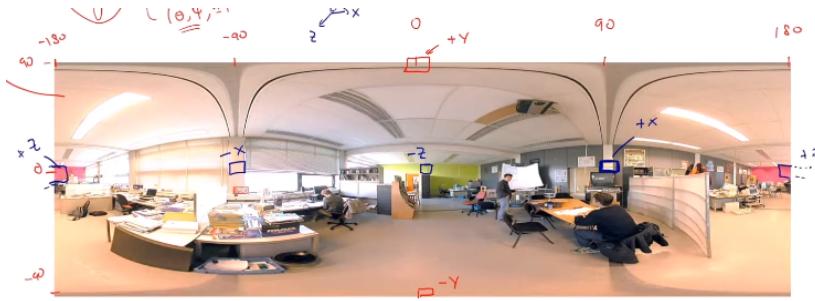
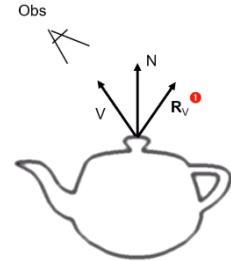


Figura 4.1: Posible textura para Environment Mapping

Las texturas para *Environment Mapping* se representan con texturas esféricas, cúbicas y equirectangulares.

Para un objeto especular que usa *environment mapping*, para calcular el color de textura vamos a interpretar la coordenada de textura como la posición de la fuente de luz en iluminación local. Por ello vamos a tener que encontrar el vector reflejado del punto de la superficie y el observador y así determinar la coordenada de textura. El *Vertex Shader* pasaría las coordenadas del punto de la superficie y su normal (en *eye space*) y el *Fragment Shader* se encargaría de encontrar el vector reflejado.




---

```

uniform vec3 obs;
in vec3 p;
void main() {
    vec3 V = normalize(obs - p);
    vec3 R_v = 2*dot(N, V)*N - V; // R_v = reflect(-V, N);
    fragColor = environmentMap(R_v);
}

```

---

*Environment Mapping* también se usa para representar el color de fondo de la escena (cielos, montañas lejanas, etc.). En este caso por cada fragmento se calcula el *view vector* (vector  $\vec{V}$  que va del punto al observador) y se invierte.

---

```

uniform vec3 obs;
in vec3 P;
void main() {
    vec3 V = normalize(obs - p);
    vec3 R = -V;

```

---

---

```

fragColor = environmentMap(R);
}

```

---

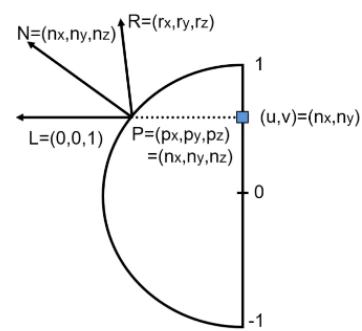
#### 4.2.1. Sphere Mapping

La función *environmentMap* nos calcula las coordenadas  $(s, t) \in [0, 1]^2$ . Por ahora trabajaremos con las coordenadas  $(u, v) \in [-1, 1]^2$  y luego transformaremos los intervalos para traducirlo a  $(s, t)$ .

$$environmentMap(R) : \mathbb{R}^3 \mapsto [0, 1]^2$$

Siendo que nos dan el vector unitario  $R$ , que la normal de un punto  $P$  a una esfera es  $P - \vec{\text{centro}}$ , y que el centro del *Sphere Map* lo suponemos  $(0, 0)$  tenemos que encontrar la normal de  $P$  para encontrar  $(u, v)$ . Viendo la imagen nos podemos dar cuenta de la siguiente relación:

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



De esto deducimos que  $P$  se puede calcular como:

$$\begin{aligned} P_x &= n_x = \frac{r_x}{2 \cdot n_z} \\ P_y &= n_y = \frac{r_y}{2 \cdot n_z} \\ P_z &= n_z = \sqrt{\frac{r_z + 1}{2}} \end{aligned}$$

Esta función se puede ver en *glsl* de la siguiente forma:

---

```

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);
}

```

---

Con *Sphere Mapping* en *Eye Space* no veremos cambio alguno al girar la

esfera mientras que en *World Space* si gira el entorno reflejado. Se suele usar mas *Eye Space* ya que la calidad es mejor y no se aprecia tanto la distorsión.

#### 4.2.2. Cube Mapping

*Cube Mapping*, al ser un cubo formado por 6 caras, muestra la reflexión con menor distorsión y se puede hacer tanto en *Eye Space* como en *World Space*.

```
1 // Creación de las seis texturas
2     glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT, ...);
3     glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X_EXT, ...);
4     glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y_EXT, ...);
5     glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_EXT, ...);
6     glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z_EXT, ...);
7     glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_EXT, ...);
```

```
1 // Activación de Cube Mapping
2 glEnable(GL_TEXTURE_CUBE_MAP_EXT);
```

```
1 // Generación de coordenadas de textura
2     // coordenadas en la forma (s, t, r, q)
3     glTexGenfv(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT); // s
4     glTexGenfv(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT); // t
5     glTexGenfv(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_EXT); // r
6     glEnable(GL_TEXTURE_GEN_S);
7     glEnable(GL_TEXTURE_GEN_T);
8     glEnable(GL_TEXTURE_GEN_R);
```

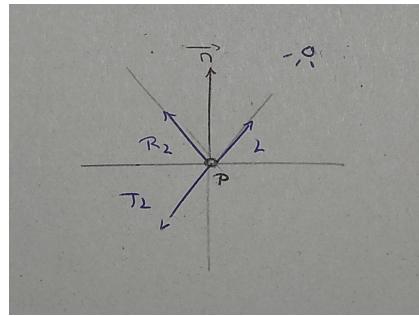
```
uniform samplerCube samplerC;
...
vec3 R;
...
vec4 color = textureCube(samplerC, R);
```

# Capítulo 5

## Objetos Translúcidos

### 5.1. Objetos translúcidos

Con los objetos translúcidos buscamos poder definir la transmisión de luz que pasa de un medio a otro.



Reflexión y Transmision

Cuando un rayo incide se pueden dar dos casos:

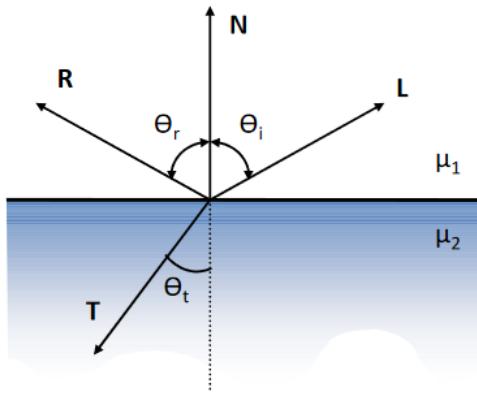
- Transmisión: La luz se transmite de un medio a otro en una dirección preferente.
- Difusión (*Scattering*): La luz se transmite de un medio a otro en cualquier dirección de forma uniforme.

Supondremos a partir de ahora que estamos hablando de transmisión.

También tenemos que tener en cuenta el concepto de refracción en lo referente al medio  $\mu$ .

### 5.1.1. Ley de Snell

Cuando un rayo con un ángulo  $\theta_i$  se transmite de un medio  $\mu_1$  a un medio  $\mu_2$  podemos saber con que ángulo  $\theta_t$  saldrá aplicando la Ley de Snell.



Sean  $N$  y  $L$  dos vectores unitarios y  $\theta_i$  el ángulo entre ellos, podemos expresar la separación entre ellos como  $\sin \theta_i$ . De aquí podemos deducir que la separación entre  $T_L$  y el opuesto de  $N$  como:

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

- Cuando  $N$  y  $L$  son colineares la separación entre ellos es nula,  $\sin \theta_i = 0$  y por tanto  $\sin \theta_t = 0$
- Cuando  $\mu_2$  es mas denso que  $\mu_1$  ( $\mu_2 > \mu_1$ ) entonces  $\sin \theta_t < \sin \theta_i$ .
- Cuando  $\mu_2$  es menos denso que  $\mu_1$  ( $\mu_2 < \mu_1$ ) entonces  $\sin \theta_t > \sin \theta_i$ .

A partir de cierto ángulo la luz ya no se puede refractar, este ángulo es conocido como ángulo crítico y es aquel en que  $\theta_t = 90^\circ$

$$\begin{aligned}\sin \theta_t &= 1 \\ &= \mu \cdot \sin \theta_c \\ \sin \theta_c &= \frac{\sin \theta_i}{\mu} \\ &= \frac{1}{\mu} = \frac{\mu_2}{\mu_1}\end{aligned}$$

### 5.1.2. Ecuaciones de Fresnell

Las ecuaciones de Fresnell nos permiten determinar que proporción de un rayo incidente se refleja y que porcentaje se transmite. Para poder aplicarlas tenemos que asumir:

- La transmisión es especular pura (no hay difusión).
- La absorción de la luz es nula ( $\%_R + \%_T = 100\%$ )

Sea  $R$  el porcentaje de luz que se refleja y  $T$  el porcentaje de luz que se transmite,  $T = 1 - R$ .

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

Vemos que las ecuaciones de Fresnell acaban dependiendo de:

- $\theta_i$
- $\theta_t$  que depende a su vez de: 
$$\begin{cases} \theta_i \\ \mu_1 \\ \mu_2 \end{cases}$$

### Aproximación de Slick

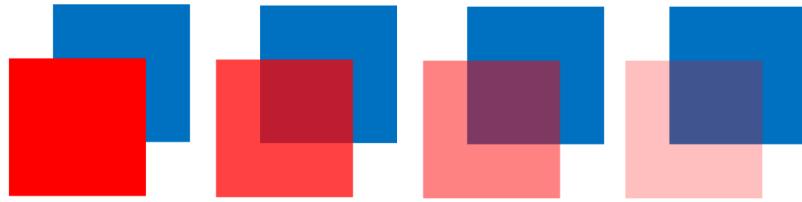
La aproximación de Slick es una función polinómica que aproxima las ecuaciones de Fresnell. Al no requerir de funciones trigonométricas es la mas utilizada por las aplicaciones de renderizado.

$$R = f + (1 - f) \cdot (1 - L \cdot N)^5$$
$$f = \frac{(1 - \mu)^2}{(1 + \mu)^2}$$

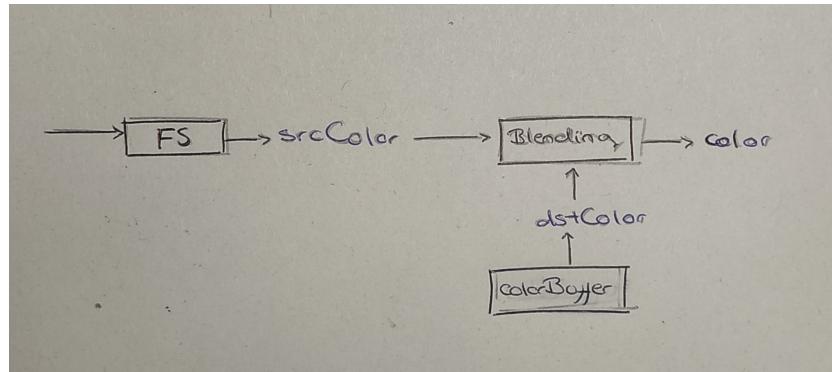
## 5.2. Alpha Blending

Alpha Blending se usa para representar los objetos semitransparentes. Para determinar el nivel de opacidad de un objeto se suele usar el componente *alpha*

del color en formato  $(r, g, b, a)$ , siendo  $a$  la componente *alpha*. En la imagen vemos que la componente *alpha* del quad rojo va disminuyendo y haciéndose cada vez más transparente.



En Alpha Blending, después de la rasterización, el Fragment Shader saca el color *srcColor* y aplica la función de blending sobre el color que tiene el pixel en el ColorBuffer, que llamaremos *dstColor*.



El color de salida de la función de Blending se calcula de la forma

$$color = sFactor \cdot srcColor + dFactor \cdot dstColor$$

en donde *sFactor* es el peso que tiene *srcColor* y *dFactor* el peso que tiene *dstColor*.

En OpenGL se usa la función ***glBlendFunc(sFactor, dFactor)*** para determinar los valores de *sFactor* y *dFactor*. Para el caso en que queramos la función de Blending similar a

---

```
float a = srcColor.a;
vec4 color = mix(srcColor, dstColor, 1 - a);
```

---

haremos lo siguiente:

```

1  glEnable(GL_BLEND); // Activamos Blending
2  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
3      // sFactor = alpha
4      // dFactor = 1 - alpha

```

El orden en que se pintan los polígonos es importante ya que el *Depth Buffer* no tiene en cuenta si un objeto es o no opaco. La forma correcta de implementar Alpha Blending para que no de errores es usando el metodo *Back-To-Front* en el que ordenamos los objetos en función de la distancia de la componente *z* respecto a la cámara antes de pintarlos de tal forma que primero se pinten los elementos mas alejados.

En caso que no los ordenemos nos pueden dar varios errores como pueden ser:

- Se dibuje antes el objeto transparente, si *ZTest* está activado el objeto, que el opaco estando el opaco detras. El objeto opaco no se pintara que que *DepthTest* no tiene en cuenta que sea opaco.
- Se dibuje antes el objeto transparente antes que el opaco, si *ZTesting* está desactivado. *srcColor* será el color del objeto opaco y *dstColor* el del objeto transparente.

Para imprimir los objetos de forma correcta hay 3 metodos:

1. Back-To-Front:

- No requiere de ZBuffer ni ZTesting.
- Es muy costoso.

2. Dibujar primero los objetos opacos sin ordenar y luego dibujamos los transparentes ordenados
- |                    |   |
|--------------------|---|
| DepthTest activado | { |
| DepthMask activado |   |
- |                       |   |
|-----------------------|---|
| DepthTest activado    | { |
| DepthMask indiferente |   |
3. Dibujar primero los objetos opacos sin ordenar y luego dibujamos los transparentes desordenados
- |                    |   |
|--------------------|---|
| DepthTest activado | { |
| DepthMask activado |   |
- |                       |   |
|-----------------------|---|
| DepthTest activado    | { |
| DepthMask desactivado |   |

La opción 3 es la mas eficiente aunque da errores de srcColor y dstColor. Por otro lado la opción 2 es la mejor ya que no es tan costosa como la Back-To-Front pero da el mismo resultado.

### 5.3. Cross Product

Sea  $\vec{a} = (a_1, a_2, a_3)$  y  $\vec{b} = (b_1, b_2, b_3)$  se define el cross product entre ellos como:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \hat{i} \cdot \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} - \hat{j} \cdot \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} + \hat{k} \cdot \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}$$

El calculo de un determinante  $2 \times 2$ :

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21}$$

# Capítulo 6

## Iluminación Global

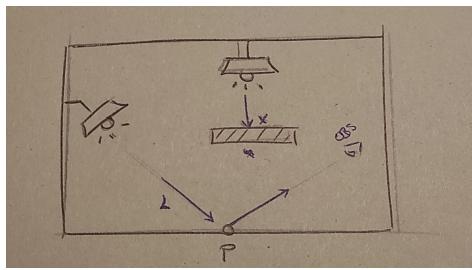
Sea  $p$  un punto cualquiera de una superficie, el color que le llega al observador de  $p$  es la luz propagada de  $p$  hacia el observador. Esta luz depende de tres factores:

- Luz reflejada de  $p$  hacia el observador.
- Luz refractada si  $p$  es un objeto translúcido.
- Reflectividad de  $p$  (características de su material)

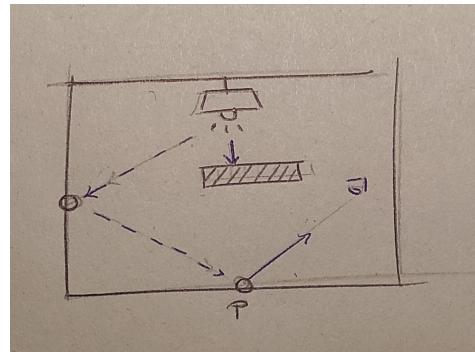
Para calcular la luz que le llega al observador de  $p$  hay dos métodos: iluminación local e iluminación global.

La iluminación local solo tiene en cuenta la luz que procede directamente de la fuente (no hay rebotes); el seguimiento de la luz sería de la forma Emisor  $\Rightarrow p \Rightarrow$  Observador donde el emisor puede ser tanto una fuente de luz como una textura (p.e. Environment Mapping). Un modelo conocido de iluminación local es el de Phong.

La iluminación global es esa que además de tener en cuenta la luz directa también tiene en cuenta la luz indirecta (luz que llega a  $p$  después de haber sufrido al menos un rebote o interacción con alguna superficie). En este caso un emisor puede ser tanto una fuente de luz como una textura (p.e. Environment Mapping) como cualquier superficie de la escena. De técnicas de iluminación global hay dos grupos: RayTracing, que funciona mejor para superficies especulares, y Radiosity, que funciona mejor para superficies difusas.



Iluminación Local



Iluminación Global

## 6.1. Radiometría

La radiometría se define como el estudio de la medida de las ondas electromagnéticas. La fotometría se define como el estudio de la medida de la radiación visible (luz).

Dadas estas dos definiciones nos podemos preguntar *¿Cómo se mide la luz?*. Lo que hemos visto hasta ahora los modelos de iluminación global usan valores *RGB* para medir la luz. Esto es poco realista ya que no existe una cota superior en lo que a energía puede emitir la luz. Así que veremos como se mide la luz según la física.

**Flujo Radiante  $\phi$**  : Cantidad de energía que atraviesa una superficie por unidad de tiempo. Sirve para medir la cantidad de energía que emite una fuente de luz en un determinado tiempo.

- Radiometría  $W$
- Fotometría  $lm$  (lumens)

**Irradiancia  $E$**  : Densidad de flujo (Flujo/Área) de una superficie. Mide la cantidad de luz que incide en una superficie, sin distinguir de donde llega.

- Radiometría  $W/m^2$
- Fotometría  $lm/m^2 = lx$  (lux)

Sea  $p$  un punto de la superficie y un emisor con flujo  $\phi$  y área  $A$ , medimos la irradiancia en  $p$  como  $\phi/A$  si el área del emisor es perpendicular con la normal de  $p$ . En caso contrario dependerá del ángulo con que incida

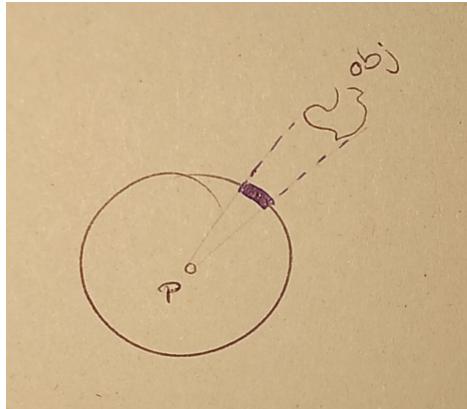
$A$  sobre  $p$ . Digamos que incide con un ángulo  $\theta_i$  entonces el área de incidencia en  $p$  lo podemos definir como  $A / \cos \theta_i$  y por tanto vemos que cuanto mayor es el ángulo de incidencia mayor será el área de incidencia y menor será la irradiancia.

$$E_2 = \frac{\phi}{A / \cos \theta_i} = \frac{\phi}{A} \cdot \cos \theta_i = \frac{\phi}{A} \cdot (N \cdot L)$$

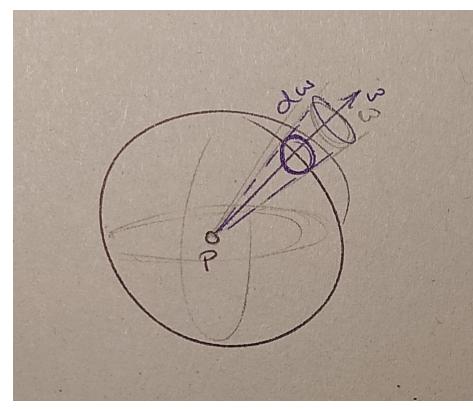
**Luminosidad  $I$**  : Flujo radiante por unidad de ángulo sólido. Describe la distribución direccional de una fuente de luz (solo sirve para luces puntuales)

- Radiometría  $W/sr$
- Fotometría  $lm/sr$

Sea  $p$  un punto cualquiera del espacio y  $obj$  un objeto cualquiera del mismo. Imaginemos una circunferencia sobre  $p$  en donde queremos medir cuanto ocupa  $obj$  de la circunferencia. Este valor medido en radianes es lo que se conoce como angulo plano (2D). Ahora extrapolemos esto a un entorno en 3 dimensiones e imaginemos que en vez de tener un objeto  $obs$  lo que tenemos es un vértice  $\vec{w}$  unitario con origen en  $p$  y lo que queremos saber es el área incidente de un cierto cono alineado con  $\vec{w}$ . Esta proporción de área se conoce como ángulo sólido y comprende valores en el rango  $[0, 4\pi]$ .



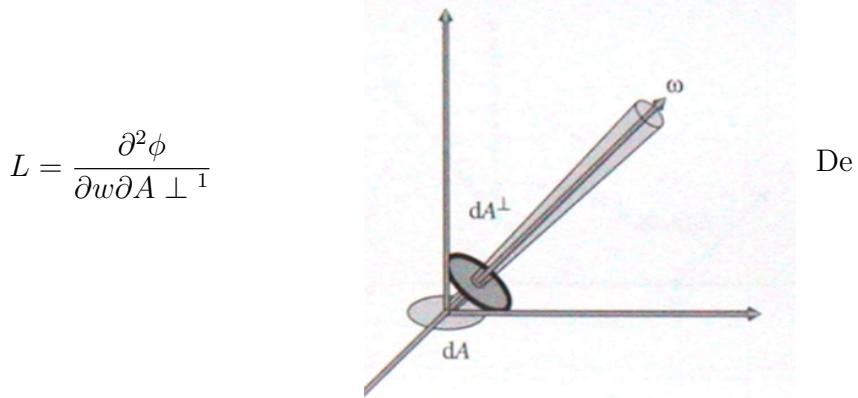
Ángulo Plano (2D)



Ángulo Sólido (3D)

**Radiancia  $L$**  : Flujo por unidad de área y unidad de ángulo sólido. Cantidad de energía que sale/llega de un punto en una determinada dirección (color

de un rayo de luz).



radiancia hay 3 tipos:

1. Radiancia Emitida  $L_e(p, w)$ : Radiancia que emite  $p$  en dirección  $w$ .  
 $L_e(p, w) = 0$  a no ser que  $p$  sea una fuente de luz.
2. Radiancia Incidente  $L_i(p, w)$ : Radiancia que llega a  $p$  desde una dirección  $w$ . Sea  $q$  el punto desde donde se emite la onda,  $\vec{w} = \vec{pq}$ .
3. Radiancia Saliente  $L_o(p, w)$ : Radiancia que sale de  $p$  en dirección  $w$ .

### 6.1.1. Calculo de la Radiancia

Sea  $\Omega$  el conjunto de todas las direcciones  $w_i$  que inciden sobre el punto  $p$  en la semiesfera positiva alineada con la normal  $\vec{n}$  de  $p$ , definimos la irradiancia de  $p$  como

$$\begin{aligned} E(p) &= \int_{\Omega} L_i(p, w) \cdot \cos \theta_i \partial w \\ &= \int_{\Omega} L_i(p, w) \cdot (N \cdot L) \partial w \\ &= \int_{\Omega} L_i(p, w) \partial w \perp \end{aligned}$$

## 6.2. BRDF-BTDF-BSDF

Se define BRDF como la función de distribución bidireccional que tiene en cuenta la reflectividad. Sea  $p$  un punto de una superficie,  $w_i$  una dirección inci-

---

<sup>1</sup> $A \perp$  equivale al área proyectada por  $A$

dente en  $p$  y  $w_o$  una dirección saliente de  $p$  BRDF nos indica cuantos porcentaje de radiancia incidente en  $p$  desde  $w_i$  sale en dirección  $w_o$ .

$$f_R(p, w_i, w_o) = \frac{L_o(p, w_o)}{L_i(p, w_i) \cdot \cos \theta_i}$$

Se cumple que:

- $f_R(p, w_i, w_o) \geq 0$
- $f_R(p, w_i, w_o) = f_R(p, w_o, w_i)$
- $\int_{\Omega} f_R(p, w', w_o) \cdot \cos \theta_i \partial w'$

Se define BTDF como la función de distribución bidireccional basada en la transmisión y BSDF como aquella función que combina tanto BRDF como BTDF.

### 6.3. Ecuación General del Rendering

La ecuación general del rendering expresa la radiancia saliente de un punto. Si el medio de propagación de la luz no se tiene en cuenta (p.e. aire) entonces  $L_o(p, w_o)$  es igual al color del pixel.

La siguiente ecuación no sirve para calcular  $L_o(p, w_o)$  pero sabemos que

$$L_o(p, w_o) = L_e(p, w_o) + \int_{S^2} f(p, w_i, w_o) \cdot L_i(p, w_i) \cdot \cos \theta_i \partial w_i$$

en donde:

- $L_e(p, w_o) = 0$  si  $p$  no es una fuente de luz
- $S^2$  representa todas las direcciones unitarias posibles de una esfera
- $f(p, w_i, w_o)$  es el porcentaje calculado por BSDF
- $L_i(p, w_i) \cdot \cos \theta_i$  es la irradiancia  $E(p)$  6.1 de  $p$ .

La ecuación cumple:

- La ecuación dentro de la integral no puede resolverse analíticamente.
- La ecuación permite clasificar los algoritmos para iluminación global.

- La ecuación es recursiva.

Por último cabe decir que  $L_o(p, w_o)$  se ha de calcular para cada longitud de onda  $\lambda$  (normalmente las longitudes de onda *RGB*).

## 6.4. Light Paths

Los Light Paths usan una notación que nos permite identificar todos los posibles caminos de los fotones desde que son emitidos hasta que llegan a un sensor. Un LightPath se escribe como:

$$LP = L(D|S) * E$$

En donde  $L$  simboliza la luz/fuente,  $D$  simboliza un objeto difuso,  $S$  un objeto especular y  $E$  un observador. Si un objeto es tanto especular como difuso el LightPath lo interpreta como

$$LDSE + LSSE$$

En función del Light Path podemos escoger cual algoritmo de iluminación global es mejor:

- DD: Radiosidad
- DS
- SD
- SS: Ray Tracing.

# Capítulo 7

## Ray Tracing

### 7.1. Ray Tracing y variantes

#### 7.1.1. Ray Casting

Ray Casting es un paradigma que se basa en lanzar rayos desde la cámara hacia la escena. Es un método iterativo, de iluminación local.

---

Ray Casting

---

```
for all  $y \in [vyM \dots vym]$  do
    for all  $x \in [vxm \dots vxM]$  do
         $R \leftarrow throwRay(x, y, Obs)$ 
         $\forall f \in face : findIntersection R-f$ 
        sortIntersections
         $(p, w_o) \leftarrow firstIntersection$ 
        color( $x, y$ ) :=  $L_o(p, w_o)$ 
    end for
end for
```

---

#### 7.1.2. Ray Tracing Clásico

Similar a Ray Casting. Se lanzan rayos desde el observador, en sentido contrario a la propagación de la luz.

El rayo que lanza la cámara e interseca con una superficie de la escena se conoce como rayo primario. El impacto entre el rayo y la superficie se conoce como *Hit*. Dependiendo de si el objeto es difuso o especular se aplican distintos criterios.

Cuando el objeto  $D$  es difuso no se aplica recursividad, calculamos iluminación local. Por cada fuente de luz  $f$  lanzamos un rayo  $D + \lambda D\vec{f}$  desde  $D$  a la fuente para determinar si hay un punto opaco que se interpone entre la fuente y  $D$ . Si es así el objeto opaco hace sombra y por tanto no tenemos en cuenta esa fuente de luz. Estos rayos se conocen como *Shadow Ray*.

Cuando el objeto  $S$  es especular, además de calcular los shadow rays, lanzamos un rayo  $R_S$  reflejado por  $S$  y, si  $S$  es transmisor/translúcido/transparente, lanzamos un rayo transmitido  $R_T$ . Si  $R_S$  o  $R_T$  chocan con algún objeto especular se repite recursivamente el proceso hasta encontrarse con un objeto difuso. Cuando se encuentra el objeto difuso se extrae el LightPath del camino. Deducimos que Ray Tracing Clásico solo permite Light Paths de la forma  $LD(S)^*E$

### 7.1.3. Path Tracing

Path Tracing se basa en lanzar  $r \in \mathbb{N}$  rayos desde la cámara en cada píxel de la pantalla. Entonces, la cámara lanza un rayo primario y cuando hace Hit, ya sea con un objeto difuso o especular lanza un rayo saliente. Si el objeto solo es especular el rayo será un rayo reflejado pero si además es translúcido se elegirá lanzar un rayo reflejado o transmitido al azar. Se aplica este algoritmo recursivamente hasta llegar a una profundidad deseada. Los shadow rays se siguen lanzando. Se deduce que PathTracing permite describir cualquier tipo de camino aunque está limitado por su profundidad.

$$L(D|S)^\lambda E$$

$$\lambda = \text{profundidad}$$

Path Tracing al reflejar los rayos en objetos difusos puede llegar a haber mucho ruido

### 7.1.4. Distributed Ray Tracing

Distributed Ray Tracing funciona parecido a Ray Tracing Clásico. Lanzamos un rayo primario. Si nos encontramos una superficie difusa lanzamos los shadow rays y devolvemos la radiancia  $L_o(p, w_o)$ . En caso que nos encontremos una superficie especular lanzaremos, en vez de uno,  $r \in \mathbb{N}$  rayos distribuidos en un cierto alineado con la dirección de reflexión preferente (si el objeto es translúcido ademas podemos lanzar  $r$  rayos distribuidos en un cierto cono ali-

neado con la dirección preferente de transmisión). Este metodo nos permite simular reflexiones imperfectas.

Al ser similar a Ray Tracing Clásico solo podemos simular Light Paths de la forma  $LD(S)^*E$ .

### 7.1.5. Two-Path Ray Tracing

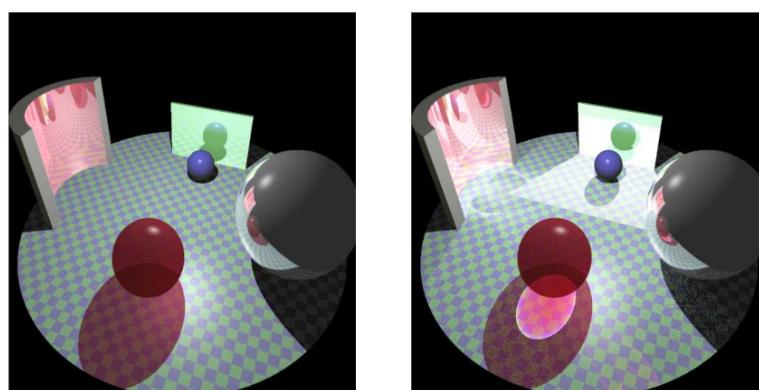
Two-Path Ray Tracing sigue el metodo de terminar la simulación de rayos cuando se llega a una superficie difusa. Pero corre dos tipos de simulaciones.

1. Rayos lanzados desde la cámara (Ray Tracing Clásico).
2. Rayos lanzados desde la fuente de luz.

Lo primero que se simula son los rayos lanzados desde la fuente de luz. Cuando el rayo incide sobre una superficie difusa  $D$  de almacena la energía que llega a  $D$  en una estructura de datos cualquiera ( $k-d\text{-tree}$ ). Luego desde la cámara se lanzan los rayos y cuando a una se llega a una superficie difusa, además de calcular los shadow rays, consultamos en la estructura de datos la energía acumulada en los puntos cercanos la irradiancia acumulada. Los Light Paths son de la forma  $L(S)^*D(S)^*E$ .

Two-Path Ray Tracing es de los métodos más inteligentes ya que explota al máximo las superficies especulares (que es para lo que está Ray Tracing) manteniendo la simplicidad del algoritmo de Ray Tracing Clásico.

## Classic vs Two-pass ray-tracing

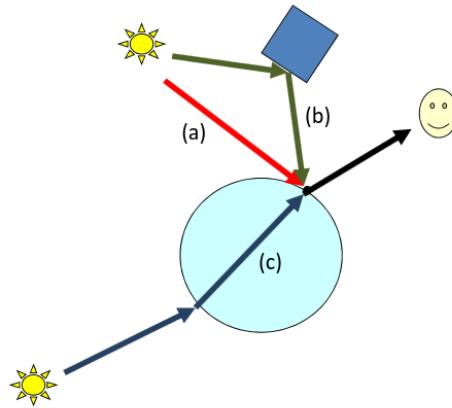


## 7.2. Ray Tracing Clásico

En Ray Tracing el color que ve el observador en un punto  $p$  se define como

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

- $I_D(p)$ : Luz directa que llega a  $p$ . (a)
- $I_r(p)$ : Luz indirecta que se refleja en  $p$  hacia el observador. (b)
- $I_T(p)$ : Luz indirecta que se transmite en  $p$  hacia el observador. (c)

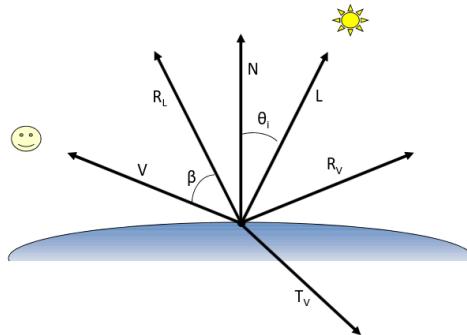


La contribución de la luz directa se define como

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

en donde

- $\cos \theta_i = N \cdot L$ .
- $\cos \beta = V \cdot R_L$ .



Los sumatorios solo se aplican a luces no ocluidas, que no tienen un oclusor que genere sombra en  $p$ . Y la contribución de la luz indirecta como

$$I_R(p) = K_R \cdot L_R$$

$$I_T(p) = K_T \cdot L_T$$

En donde

- $K_R, K_T$ : coeficientes de reflexión/transmisión especular.
- $L_R$ : Luz que incide en  $P$  en dirección  $R_V$
- $L_T$ : Luz que incide en  $P$  en dirección  $T_V$

$L_R$  y  $L_T$  se calculan recursivamente trazando un nuevo rayo.

El algoritmo de Ray Tracing es: Para trazar un rayo se aplica un algoritmo

---

#### Ray Tracing

---

```

for all  $(x, y) \in pixel$  do
     $R \leftarrow rayoPrimario(x, y, obs)$ 
     $color \leftarrow trazarRayo(R, scene, \mu)$        $\triangleright \mu$ : indice de refracción del medio
     $setPixel(x, y, color)$ 
end for

```

---

tal que:

1. Si el límite de número de rebotes ha sido superado se devuelve el color de fondo, o un color cualquiera que implique algo parecido, así evitamos errores de estimación.
2. Lanzamos un rayo y calculamos si hay una intersección con la escena. Si no la hay devolvemos el color de fondo.
3. Si el objeto es especular se traza otro rayo alineado con la dirección preferente de reflexión.
4. Si el objeto es transmisor se traza otro rayo alineado con la dirección preferente de transmisión adaptado al nuevo medio  $\mu_T$ .
5. calculamos la luz directa y devolvemos el color resultante.

---

Trazar Rayo(rayo, escena,  $\mu$ )

---

```
if profundidadCorrecta() then
    info ← calculaInterseccion(rayo, escena)
    if info.existeInterseccion() then
        color ← calculaID(info, escena)                                ▷  $I_D$ 
        if info.obj.esReflector() then
            rayoR ← calculaRayoReflejado(info, rayo)
            color ← color +  $K_R \cdot \text{trazarRayo}(rayoR, \text{escena}, \mu)$       ▷  $I_R$ 
        end if
        if info.obj.esTransparente() then
            rayoT ← calculaRayoTransmitido(info, rayo,  $\mu$ )
            color ← color +  $K_T \cdot \text{trazarRayo}(rayoT, \text{escena}, \text{info.}\mu)$   ▷  $I_T$ 
        end if
    else
        color ← colorFondo
    end if
else
    color ← colorFondo
end if
return color
```

---

En el Fragment Shader llamariamos a calcular *trazarRayo*:

---

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

---

### 7.3. Intersección Rayo-Triángulo

Tenemos un rayo  $p + \lambda\vec{v}$  y un triangulo  $(A, B, C)$  y queremos saber si el rayo intersecciona con el triángulo. la intersección se puede calcular en dos pasos:

En el primer paso se calcula si el rayo interseca con el plano del triangulo. Para ello se extraen las ecuaciones del plano y el triángulo de tal forma que

un punto en el rayo se expresa como

$$q \in p + \lambda \vec{v} \iff \begin{cases} q_x = p_x + \lambda v_x \\ q_y = p_y + \lambda v_y \\ q_z = p_z + \lambda v_z \end{cases}$$

y un punto en el plano con la normal  $\vec{n} = (a, b, c)$  como

$$a \cdot q_x + b \cdot q_y + c \cdot q_z + d = 0$$

y se deduce que un punto del rayo aparece en el plano cuando

$$a \cdot (p_x + \lambda v_x) + b \cdot (p_y + \lambda v_y) + c \cdot (p_z + \lambda v_z) + d = 0$$

Si se aislá  $\lambda$

$$\lambda = \frac{- \cdot (a \cdot p_x + b \cdot p_y + c \cdot p_z + d)}{a \cdot v_x + b \cdot v_y + c \cdot v_z} = \frac{-\text{dist}(p, \text{plano})}{\vec{n} \cdot \vec{v}}$$

Si  $\vec{n} \cdot \vec{v} \neq 0$  existe una intersección. Si  $\vec{n} \cdot \vec{v} \neq 0$  entonces el rayo y el plano son paralelos y habrá intersección si  $\text{dist}(p, \text{plano}) \neq 0$ , no habrá en caso contrario.

En el segundo paso queda calcular si el punto de intersección con el plano pertenece al triángulo. Sea  $(A, B, C)$  el triángulo,  $q$  es un punto del triángulo si

$$\begin{aligned} q &= A + \beta \cdot \overrightarrow{BA} + \gamma \cdot \overrightarrow{CA} \\ &= (1 - \beta - \gamma) \cdot A + \beta \cdot B + \gamma \cdot C \\ &= \alpha \cdot A + \beta \cdot B + \gamma \cdot C \end{aligned}$$

$\alpha, \beta$  y  $\gamma$  son coordenadas baricéntricas del triángulo. Entonces definimos que un punto existe en el triángulo si

$$p \in \Delta(A, B, C) \iff \begin{cases} \alpha, \beta, \gamma \in [0, 1] \\ \alpha + \beta + \gamma = 1 \end{cases}$$

El truco para encontrar si un punto  $p$  está en el triángulo es ver que el valor de la coordenada baricéntrica es igual al porcentaje de área del sub-triángulo

opuesto al punto en  $p$ .

$$\text{alpha} = \frac{\text{area}\Delta(p, B, C)}{\text{area}\Delta(A, B, C)}$$