# Resumen Informática Gráfica

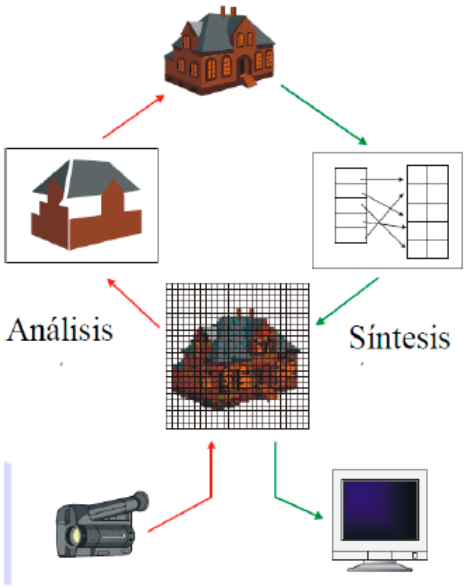
Apuntes extraídos de la guía del profesor Fco. Javier Melero Rus

## Contenido

# Tema 1 – Introducción a la informática gráfica

## 1.1. Definiciones

- **Gráficos por ordenador:** Es la creación, manipulación, análisis e interacción de las representaciones pictóricas de objetos usando ordenadores.
- **Informática gráfica:** es la síntesis pictórica de objetos reales o imaginarios basada en sus modelos de ordenador.
- **Análisis y síntesis de imagen:**
    - **Análisis:** es la extracción de información derivada de sensores y representada gráficamente en formato de dos o tres dimensiones, para lo cual se puede utilizar tanto análisis visual como digital.
    - **Síntesis:** La síntesis gráfica es cuando una figura simplifica su forma original manteniendo el uso de líneas y planos, pero en menor cantidad.




- **Primitiva:** elementos más pequeños que pueden ser visualizados. Normalmente serán triángulos, pero también pueden ser polígonos, puntos, segmentos de recta, círculos, etc.

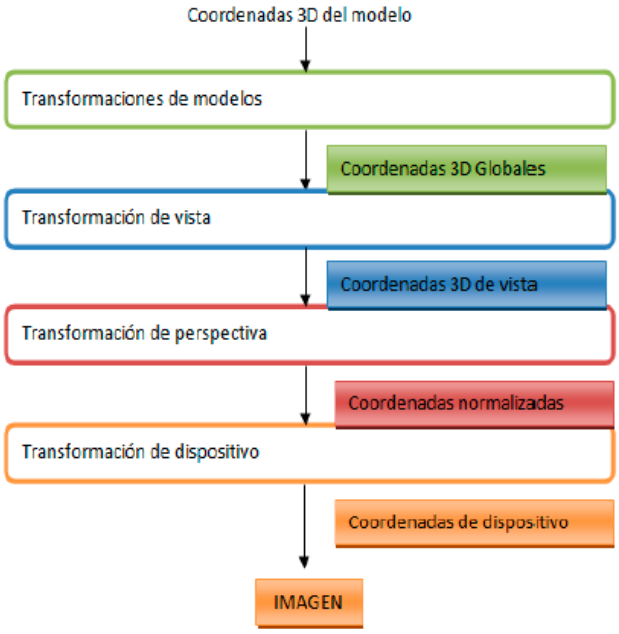
## 1.2. Áreas de la informática gráfica

- **Modelado (modelling):** especificación matemática de las propiedades de forma y apariencia de manera que se puede almacenar en la computadora.
- **Síntesis de imágenes (rendering):** termino heredado del arte que se encarga de la creación sombreado de imágenes a modelos 3D por ordenador.
- **Animación (animation):** es la técnica de crear una ilusión de movimiento a través de una secuencia de imágenes.

- **Realidad Virtual (virtual reality):** intento de sumergir al usuario en un mundo virtual en 3D.
- **Interacción (user interaction):** interfaz entre los dispositivos de entrada y el usuario.
- **Visualización (visualization):** intento de dar a los usuarios una idea a través de la visualización.
- **Digitalización 3D (3D scanning):** uso de tecnología para crear modelos 3D.

## 1.3. Proceso de visualización

1. **Transformaciones del modelo**
   a. Situarlo en la escena.
   b. Cambiarlo de tamaño.
   c. Crear modelos compuestos de otros más simples.
2. **Transformación de vista.**
   a. Poner al observador en la posición deseada.
3. **Transformación de perspectiva.**
   a. Pasar de un mundo 3D a una imagen 2D.
4. **Rasterización.**
   a. Calcular para cada pixel su color, teniendo en cuenta la primitiva que se muestra, su color, material, texturas, luces, etc.
5. **Transformación de dispositivo.**
   a. Adaptar la imagen 2D a la zona de dibujado.




## 1.4. Rasterización y ray-tracing

La **rasterización** es el proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital.

Se puede relatar como:

```
Inicializar el color de todos los pixeles
Para cada primitiva P de la escena a visualizar
     Encontrar el conjunto S de pixeles cubiertos
por P
Para cada pixel s de S:
     Calcular el color de P en s
     Actualizar el color de s
```

Otra opción es el algoritmo de **ray-tracing (trazado de rayos)**, que suele ser más lento, pero consigue resultados más realistas, por lo

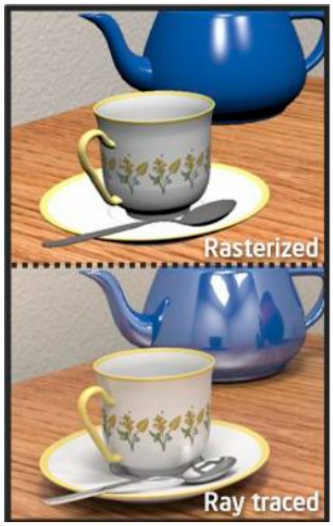



Ilustración 16: Rasterización vs Trazado de Rayos (Intel)

que se utiliza para síntesis de imágenes realistas *off-line*, como la producción de animaciones y efectos especiales en películas o alumnos.

El algoritmo consiste en invertir los bucles de la rasterización:

```
Inicializar el color de todos los pixeles
Para cada pixel s de I:
     Calcular T el conjunto de primitivas que cubren s
     Para cada primitiva P de T
          Calcular el color de P en s
          Actualizar el color de s
```

En el trazado de rayos lo que se hace es seguir el camino que sigue un rayo de luz desde la imagen hasta la fuente de luz, pasando por la cámara y rebotando entre todos los objetos de la escena.

## 1.5. Introducción a OpenGL

OpenGL es una API independiente de plataforma que se utiliza para la visualización 2D/3D basada en **rasterización**.

El objetivo final de OpenGL es generar imágenes mediante la rasterización de escenas 3D, independientemente del lenguaje de programación utilizado, del hardware existente y del sistema operativo que se esté utilizando.

OpenGL no maneja nada relacionado ni con la gestión de ventanas ni con los eventos sobre éstas. OpenGL se centra única y exclusivamente en procesar unos datos, referentes a la escena 3D y generar un **buffer de memoria**, ***framebuffer***, con la imagen 2D rasterizada.

Por tanto, será necesario utilizar una librería de interfaz de usuario distinta a OpenGL para gestionar las ventanas y los eventos, y aquí las posibilidades son casi infinitas: **GLU**, Qt, WxWidgets, Java, FLTK, Tcl-TK, etc.

## 1.6. Primitivas en OpenGL > 2.0

A partir de OpenGL, el estándar propone el uso de buffers para, en lugar de hacer miles de llamadas a **`glVertex`**, realizar una única llamada a **`glDrawArrays()`** o bien de **`glDrawElements()`.**

Con **`glDrawArrays`** la idea que subyace es tener un array con las coordenadas de los vértices, y en el momento de dibujar**, pasar todos en bloque a la tarjeta gráfica**. Si hay más propiedades, como la normal, el color o la coordenada de textura, se almacenan en **otros buffers**, y se activan para ser tenidos en cuenta.

**void glDrawArrays(GLenum mode, GLint first, GLsizei count);**

***PARÁMETROS***

| | |
|---|---|
| **mode.** | Especifica cómo interpreter la secuencia de datos ( GL_POINTS, GL_LINE_STRIP,GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY o GL_PATCHES) |
| **first** | Especifica el índice inicial de los array habilitados |
| **count** | Especifica el número de elementos del array que se van a usar |

```
void Object3D::dibujarSolido(){
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_COLOR_ARRAY);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glVertexPointer(3,GL_FLOAT,0,&vertices[0]);
  glColorPointer(3, GL_FLOAT, 0, &color[0]);
  glDrawElements(GL_TRIANGLES, 3*numLados, GL_UNSIGNED_INT, &lados[0]);
  glDisableClientState(GL_COLOR_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);
}
```

## 1.7. Función redibujado

Un ejemplo sencillo, en C y con GLUT, de función de redibujado es:

```
void draw () {
// limpiar la ventana
glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
// envio de primitivas
glEnableClientState(GL_VERTEX_ARRAY); // designación del vector
de vértices
glVertexPointer(3, GL_FLOAT, 0, vertices); // dibujado de
triángulos con el vector de vértices activos
glDrawArrays(GL_TRIANGLES, 0, 3);.....
// Intercambio de buffers
glutSwapBuffers() ;
}
```

En otras librerías de GUI, el esquema sería similar:

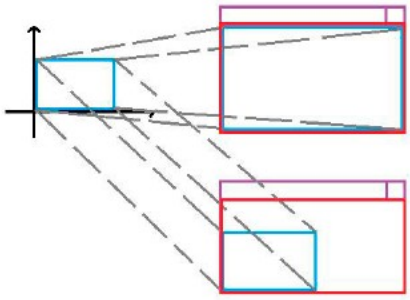
- limpiar la ventana (con `glClear`)
- enviar las primitivas
- intercambiar los buffers

¿Qué es eso de **intercambiar los buffers**? Con OpenGL se puede utilizar, y de hecho se utiliza, un **sistema de doble buffer** para el dibujado, de forma que la rasterización se realiza en una imagen que no se ve, está oculta, hasta que no ejecutamos el intercambio de buffers.

## 1.8. Viewport

La función **`glViewport`** permite establecer que parte de la ventana será usada para visualizar.

```
glViewport( izq, abajo, ancho, alto ) ;
```



# Tema 2 – Modelado de objetos

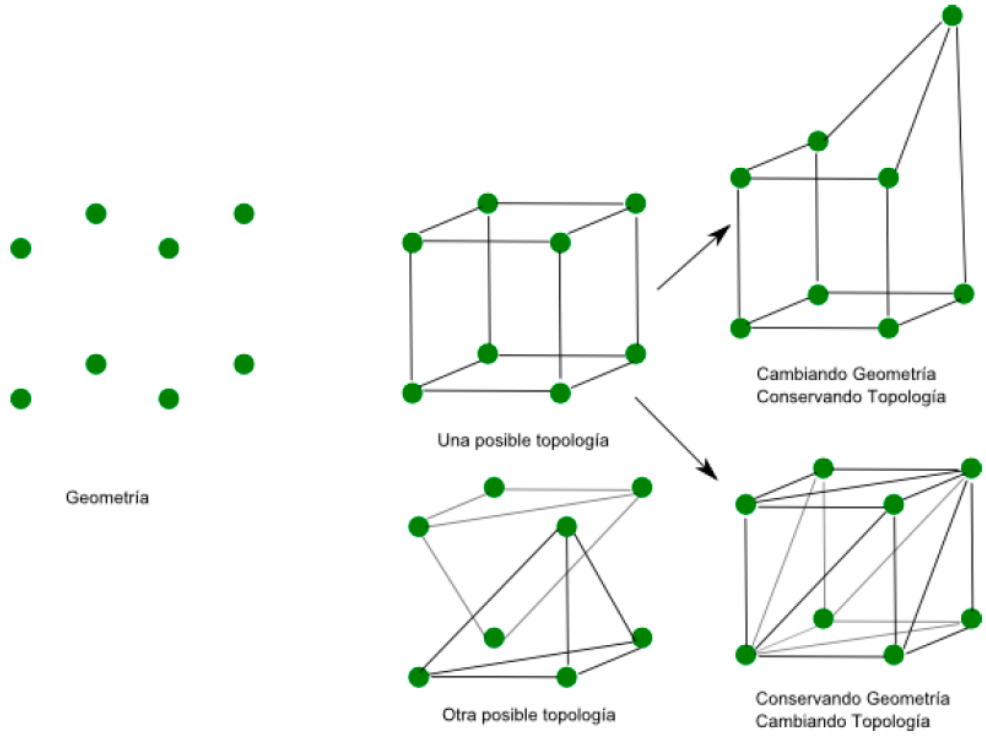
## 2.1. Introducción

Hay dos principales aproximaciones:

- **Modelos basados en fronteras:** en estos se representa la frontera o superficie del objeto. Para ello se utilizan conjuntos finitos de polígonos planos (caras).
- **Modelos basados en enumeración espacial:** se representan tanto la frontera como el interior de los objetos. Se usan en aplicaciones como la medicina, arqueología, paleontología, etc.

## 2.2. Modelos de fronteras o superficie

- **Geometría**. La geometría de un modelo viene dada por las posiciones o coordenadas de aquellos puntos que están sobre la superficie.
- **Topología**. La topología de un modelo es cómo se organiza la geometría para dar lugar a una superficie.

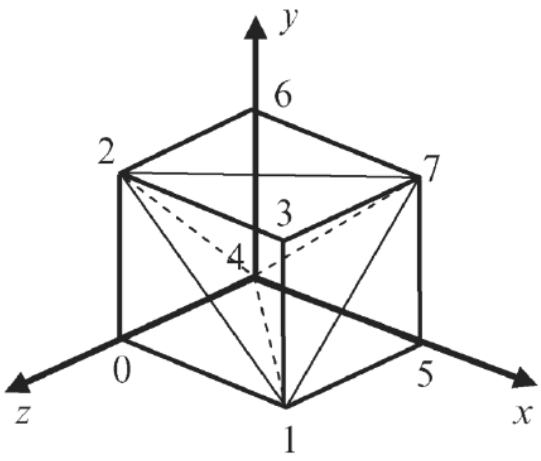



- **Vértice**, posición en el espacio que corresponde con un punto de la superficie del objeto a representar.
- **Arista**, segmento de recta que une dos vértices.
- **Cara**, polígono definido por una secuencia de vértices.

En los gráficos por ordenador, las caras son normalmente **triángulos**, es decir, están formadas por tres vértices. Esto es porque es el único polígono que

garantiza que cualesquiera que sean las posiciones de sus vértices, siempre son **coplanares** (que se encuentran en el mismo plano).

## 2.3. Tabla de vértices y triángulos




| Tabla de vértices | | |
|---|---|---|
| x | y | z |
| 0.0 | 0.0 | 1.0 |
| 1.0 | 0.0 | 1.0 |
| 0.0 | 1.0 | 1.0 |
| 1.0 | 1.0 | 1.0 |
| 0.0 | 0.0 | 0.0 |
| 1.0 | 0.0 | 0.0 |
| 0.0 | 1.0 | 0.0 |
| 1.0 | 1.0 | 0.0 |

| Tabla de triángulos | | |
|---|---|---|
| v0 | v1 | v2 |
| 0 | 1 | 2 |
| 1 | 3 | 2 |
| 2 | 3 | 7 |
| 2 | 7 | 6 |
| 1 | 7 | 3 |
| 1 | 5 | 7 |
| 6 | 7 | 4 |
| 7 | 5 | 4 |
| 0 | 4 | 1 |
| 1 | 4 | 5 |
| 2 | 6 | 4 |
| 0 | 2 | 4 |

```
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const
                    GLvoid * indices);
```

**PARÁMETROS**

| | |
|---|---|
| `mode` | Especifica qué primitivas se van a dibujar: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY o GL_PATCHES |
| `count` | Número de elementos que se van a dibujar |
| `type` | Tipo de los indices ( GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, o GL_UNSIGNED_INT) |
| `indices` | Puntero al vector donde están los índices almacenados. |

```
void Object3D::dibujarSolido(){
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_COLOR_ARRAY);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glVertexPointer(3,GL_FLOAT,0,&vertices[0]);
  glColorPointer(3, GL_FLOAT, 0, &color[0]);
  glDrawElements(GL_TRIANGLES, 3*numLados, GL_UNSIGNED_INT, &lados[0]);
  glDisableClientState(GL_COLOR_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);
}
```

## 2.4. Formato PLY

El formato de archivo PLY fue diseñado por **Greg Turk** y otros en la Universidad de Stanford a mediados de los 90.

```
ply
format ascii 1.0
comment Archivo de  ejemplo del  formato PLY (8 vertices y  6  caras)
element vertex  8
property float  x
property float  y
property float  z
element face  6
property list uchar   int  vertex_index
end_header
0.0   0.0   0.0
1.0   0.0   0.0
1.0   1.0   0.0
0.0   1.0   0.0
0.0   0.0   -1.0
1.0   0.0   -1.0
1.0   1.0   -1.0
0.0   1.0   -1.0
4    0  1  2  3
4    2  1  5  6
4    3  2  6  7
4    3  7  4  0
4    7  6  5  4
4    4  5  1  0
```

- **Cabecera**, donde se describen los atributos presentes y su formato, se indica el número de vértices y caras.
- **Lista de vértices**, un vértice por línea, indicando en el ejemplo sus coordenadas X, Y y Z (reales) en ASCII, separadas por espacios.
- **Lista de caras**, una cara por línea, indicando el número de vértices que tiene y después los índices de los vértices de la tabla de vértices.

## 2.5. Atributos de los elementos de los modelos de fronteras

- **Normales**: vectores de longitud unidad
  - *normales de caras*: vector unitario perpendicular a cada cara, de longitud unidad, apuntando al exterior de la malla. Se precalcula a partir de la información de la cara.
  - *normales de vértices*: vector unitario perpendicular al plano tangente a la superficie en la posición del vértice.
- **Colores**: valores RGB o RGBA
  - *colores de caras*: útil cuando cada cara representa un trozo de superficie de color homogéneo
  - *colores de vértices*: color de la superficie en cada vértice. En este caso se supone que el color varía de forma continua entre los vértices.

Otro atributo habitual es la **coordenada de textura**.

```
void Object3D::calcularNormalesCaras(){
  _vertex3f a, b, c;
  // Calculo
  for(unsigned int i = 0; i < caras.size(); i++){
    a._0 = vertices[caras[i].l2].x - vertices[caras[i].l1].x;
    a._1 = vertices[caras[i].l2].y - vertices[caras[i].l1].y;
    a._2 = vertices[caras[i].l2].z - vertices[caras[i].l1].z;

    b._0 = vertices[caras[i].l3].x - vertices[caras[i].l1].x;
    b._1 = vertices[caras[i].l3].y - vertices[caras[i].l1].y;
    b._2 = vertices[caras[i].l3].z - vertices[caras[i].l1].z;

    c = a.cross_product(b);
    c.normalize();
    normalesCaras.push_back(c);
  }
}

void Object3D::calcularNormalesVertices(){
  // Inicializo a 0 todas las normales de los vertices
  _vertex3f a;
  a._0=0.0;
  a._1=0.0;
  a._2=0.0;;
  for(unsigned int i = 0; i < vertices.size(); i++){
    normalesVertices.push_back(a);
  }

  // Sumo las normales de las caras adyacentes a los vertices
  for(unsigned int i = 0; i < caras.size(); i++){
    normalesVertices[caras[i].l1] += normalesCaras[i];
    normalesVertices[caras[i].l2] += normalesCaras[i];
    normalesVertices[caras[i].l3] += normalesCaras[i];
  }

  // Normalizo las normales de los vertices
  for(unsigned int i = 0; i < normalesVertices.size(); i++){
    normalesVertices[i].normalize();
  }
}
```

```
template <class Type> _vertex3<Type>
_vertex3<Type>::cross_product(const _vertex3<Type> &Vertex1)
{
_vertex3<Type> VertexTemp(*this),VertexTemp1;

VertexTemp1.x=y*Vertex1.z-z*Vertex1.y;
VertexTemp1.y=z*Vertex1.x-x*Vertex1.z;
VertexTemp1.z=x*Vertex1.y-y*Vertex1.x;
return(VertexTemp1);
}
```

```
template <class Type> _vertex2<Type> &
_vertex2<Type>::normalize()
{
Type Module=this->module();

x/=Module;
y/=Module;
return(*this);
}
```

```
double      module(){
    return (sqrt(x*x+y*y+z*z+w*w));
};
```

## 2.6. Normales caras

Para un polígono, su normal se calcula como la normalización del vector resultado del producto vectorial de dos aristas adyacentes ($e_i$ y $e_j$).

$$n = \frac{m}{\|m\|} donde\ m = e_i \times e_j$$

## 2.7. Normales vértices

Las normales de los vértices se pueden aproximar *a priori*, bien porque conozcamos exactamente su valor (en el caso de una esfera) o bien promediando las normales de las caras que comparten dicho vértice.

Para un vértice *v*, con *k* caras adyacentes, su normal *n* se calcula como:

$$n = \frac{s}{\|s\|} donde\ s = \sum_{i=1}^{k} m_i$$

Hay que tener claro que **estas normales no son las reales del objeto**, pues se desconoce la orientación exacta de la superficie en dicho punto, pero **son una muy buena aproximación**.

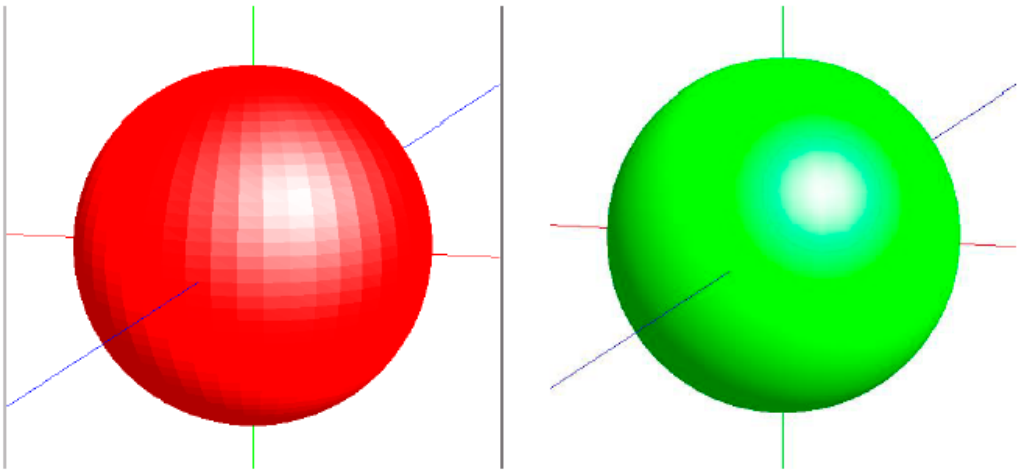


Ilustración 50: Iluminación usando normales de caras (izda) y normales por vértice (dcha.)

Para conseguir el efecto de sombreado plano de la esfera roja de la Ilustración 50, se utiliza, además del valor de la normal, la función **`glShadeModel`**:

- **`glShadeModel(GL_FLAT)`:** dibuja cada triángulo con un único color en todos los píxeles (el color del último vértice de la cara)
- **`glShadeModel(GL_SMOOTH)`:** hace que cada pixel sea generado por interpolación lineal de los colores de los vértices del triángulo.

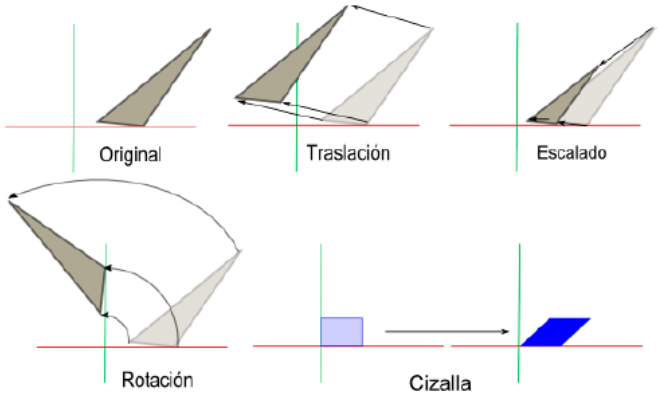
```
void Object3D::dibujarSinSuavizado(){
  glShadeModel(GL_FLAT);
  glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,GL_TRUE);
  glEnable(GL_NORMALIZE);

  if(materialActivo == 0){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialPorDefecto.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialPorDefecto.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialPorDefecto.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialPorDefecto.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialPorDefecto.brillo);
  }
  else if(materialActivo == 1){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialRubi.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialRubi.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialRubi.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialRubi.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialRubi.brillo);
  }
  else if(materialActivo == 2){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialDorado.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialDorado.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialDorado.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialDorado.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialDorado.brillo);
  }
  else if(materialActivo == 3){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialTurquesa.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialTurquesa.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialTurquesa.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialTurquesa.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialTurquesa.brillo);
  }

  glBegin(GL_TRIANGLES);
  for(unsigned int i = 0; i < caras.size(); i++){
    glNormal3fv((GLfloat *) &normalesCaras[i]);
      glVertex3fv((GLfloat *) &vertices[caras[i].l1]);
      glVertex3fv((GLfloat *) &vertices[caras[i].l2]);
      glVertex3fv((GLfloat *) &vertices[caras[i].l3]);
  }
  glEnd();

  glDisable(GL_NORMALIZE);
}
```

```
void Object3D::dibujarConSuavizado(){
  glShadeModel(GL_SMOOTH);
  glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,GL_TRUE);
  glEnable(GL_NORMALIZE);

  if(materialActivo == 0){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialPorDefecto.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialPorDefecto.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialPorDefecto.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialPorDefecto.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialPorDefecto.brillo);
  }
  else if(materialActivo == 1){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialRubi.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialRubi.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialRubi.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialRubi.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialRubi.brillo);
  }
  else if(materialActivo == 2){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialDorado.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialDorado.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialDorado.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialDorado.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialDorado.brillo);
  }
  else if(materialActivo == 3){
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialTurquesa.emision);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialTurquesa.ambiente);
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialTurquesa.difuso);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialTurquesa.especular);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialTurquesa.brillo);
  }

  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_NORMAL_ARRAY);
  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
  glVertexPointer(3,GL_FLOAT,0,&vertices[0]);
  glNormalPointer(GL_FLOAT, 0, &normalesVertices[0]);
  glDrawElements(GL_TRIANGLES, 3*numCaras, GL_UNSIGNED_INT, &caras[0]);
  glDisableClientState(GL_NORMAL_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);

  glDisable(GL_NORMALIZE);
}
```

## 2.8. Transformaciones geométricas usuales en IG

Existen varias transformaciones geométricas simples que son muy útiles en Informática Gráfica:

- **Traslación**. Consiste en desplazar todos los puntos del espacio de igual forma, es decir, en la misma dirección y la misma distancia.
  - glTranslatef( GLfloat dx, GLfloat dy, GLfloat dz )
- **Escalado**. Supone estrechar o alargar las figuras en una o varias direcciones.
  - glScalef( GLfloat sx, GLfloat sy, GLfloat sz )
- **Rotación**. Rotar los puntos un ángulo dado en torno a un eje de rotación.
  - glRotatef( GLfloat a, GLfloat ex, GLfloat ey, GLfloat ez )
- **Cizalla**. Se puede ver como un desplazamiento de todos los puntos en la misma dirección, pero con distintas distancias.



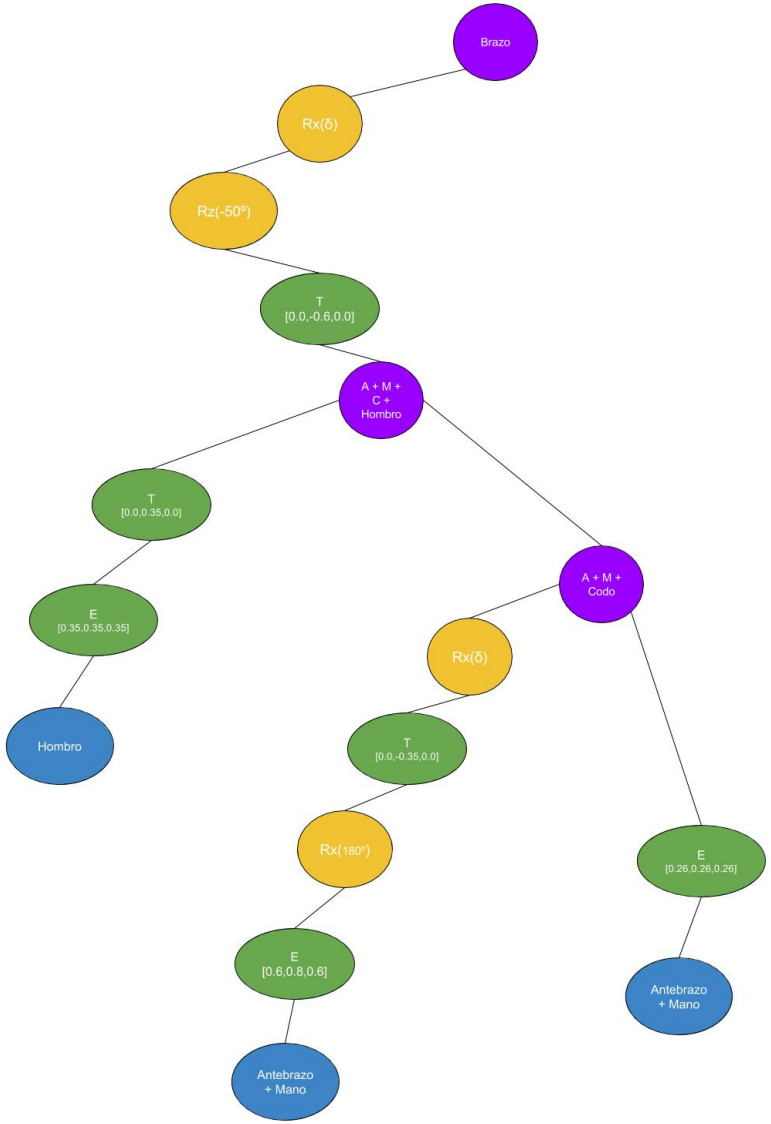

## 2.9. Transformaciones en OpenGL

### 2.9.1. La matriz modelview

OpenGL almacena, como parte de su estado, una matriz 4x4 *M* que codifica una transformación geométrica, y que se llama *modelview matrix* (matriz de modelado y vista). Esta matriz se puede ver como la composición de dos matrices, *V* y *N:*

- *N* es la matriz de modelado que posiciona los puntos en su lugar en coordenadas del mundo.
- *V* es la matriz de vista, que posiciona los puntos en su lugar de coordenadas relativas a la cámara.

## 2.10. Modelos jerárquicos. Representación y visualización.




```
void Brazo::dibujarSolido(){
    // Brazo completo
    glPushMatrix();
    glScalef(5,5,5);
        glRotatef(grHombro,1.0,0.0,0.0);
        glRotatef(-50.0,0.0,0.0,1.0);
        glTranslatef(0.0,-0.6,0.0);
        // Antebrazo + Mano + Codo + Hombro
        glPushMatrix();
            // Hombro
            glPushMatrix();
                hombro.modifyActualColor(Color(1.0,0.5,0.0)); // Color naranja camiseta
                glTranslatef(0.0,0.35,0.0);
                glScalef(0.35,0.65,0.35);
                hombro.dibujarSolido();
            glPopMatrix();
            // Antebrazo + Mano + Codo
            glPushMatrix();
                //Antebrazo + Mano
                glPushMatrix();
                    antebrazo_mano.modifyActualColor(Color(1.0,0.80,0.0)); // Color amarillo piel
                    glRotatef(grCodo,1.0,0.0,0.0);
                    glTranslatef(0.0,-0.35,0.0);
                    glRotatef(180.0,1.0,0.0,0.0);
                    glScalef(0.6,0.8,0.6);
                    antebrazo_mano.dibujarSolido();
                glPopMatrix();
                // Codo
                glPushMatrix();
                    codo.modifyActualColor(Color(1.0,0.80,0.0)); // Color amarillo piel
                    glScalef(0.26,0.26,0.26);
                    codo.dibujarSolido();
                glPopMatrix();
            glPopMatrix();
        glPopMatrix();
    glPopMatrix();
}
```

# Tema 3 – Visualización, iluminación y texturas

## 3.1. Visualización

Lo que tradicionalmente entendemos como "cámara" en el mundo real, en realidad supone **dos** tareas:

- Posicionarla y orientarla en el mundo, lo que se gestiona con la matriz *modelview*
- Generar la imagen, esto es, aplicar la transformación de perspectiva. Para ello influyen muchos parámetros, como la distancia focal, el tipo de lente, etc.

En el caso de los gráficos por ordenador, además de posicionar la cámara y realizar las proyecciones adecuadas, hay otra **serie de algoritmos que** se han de ejecutar y que **en el mundo físico pasan inadvertidos**:

- Ignorar partes ocultas.
- Ignorar elementos fuera del campo de visión.
- Ignorar objetos o partes de objetos obstruidos por otros.

En los entornos gráficos donde se generan imágenes por rasterización, intervienen **tres elementos fundamentales** para simular este fenómeno físico:

- Luces
- Materiales
- Texturas

### 3.1.1. La Cámara: Transformaciones de vista y proyección

Una cámara, o un observador, tienen una **posición en el espacio y una orientación**. Estos parámetros son definidos mediante la transformación de vista.

```
void gluLookAt(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,
               GLdouble atX, GLdouble atY, GLdouble atZ,
               GLdouble upX, GLdouble upY, GLdouble upZ);
```

Esta función posiciona al observador de forma que **el *ojo* está en la posición *eye*, mirando hacia el punto *at* y con el sentido "hacia arriba" definido por el vector *up***. Normalmente este vector es (0,1,0) cuando queremos ver la escena en una posición natural.

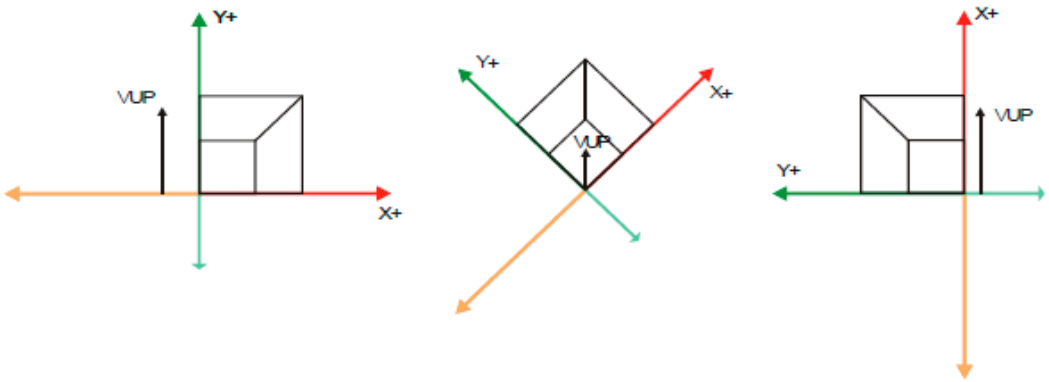



*Ilustración 74 Visualización de una escena con UP (0,1,0), (1,1,0) y (1,0,0).*

### 3.1.2. El volumen de visualización (frustum)

No es más que **la parte de la escena virtual que será considerada** a la hora de generar la imagen 2D.

Este volumen de visualización es una pirámide truncada que se define en el sistema de coordenadas de vista, y es atravesada por el eje Z de este sistema de coordenadas. Los parámetros que definen este volumen de visualización son:

- **near**, distancia del observador al plano más cercano. Todo lo que esté más cerca que ese plano, se ignora.
- **far**, distancia del observador al plano más lejano. Lo que esté más alejado en la escena, se ignora.
- **bottom**, distancia desde el centro del plano cercano (eje Z de la cámara) hasta el borde inferior del frustum en su parte más cercana a la cámara (valor mínimo de coordenada Y en el S.C. de vista).
- **top**, distancia desde el centro del plano cercano (eje Z de la cámara) hasta el borde superior del frustum en su parte más cercana a la cámara (valor máximo de Y en el S.C. de vista).
- **right**, distancia desde el centro del plano cercano (eje Z de la cámara) hasta el borde derecho del frustum en su parte más cercana a la cámara (valor máximo de coordenada X en el S.C. de vista)
- **left**, distancia desde el centro del plano cercano (eje Z de la cámara) hasta el borde izquierdo del frustum en su parte más cercana a la cámara (valor mínimo de coordenada X en el S.C. de vista).

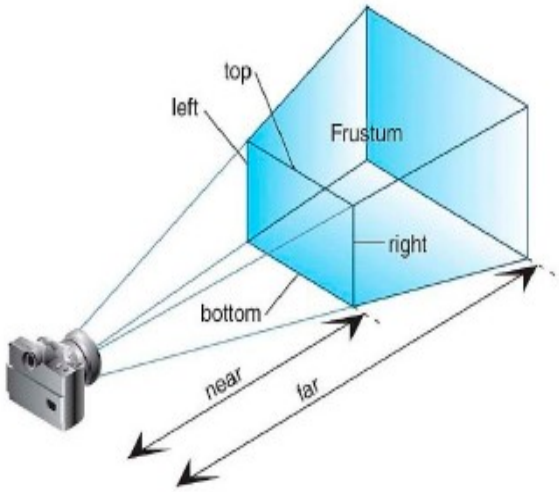



*Ilustración 79: Parámetros del volumen de visualización (frustum)*

### 3.1.3. La transformación de la Proyección

Los dos tipos de proyecciones que vienen por defecto en OpenGL son la **perspectiva con un punto de fuga y la ortográfica**.

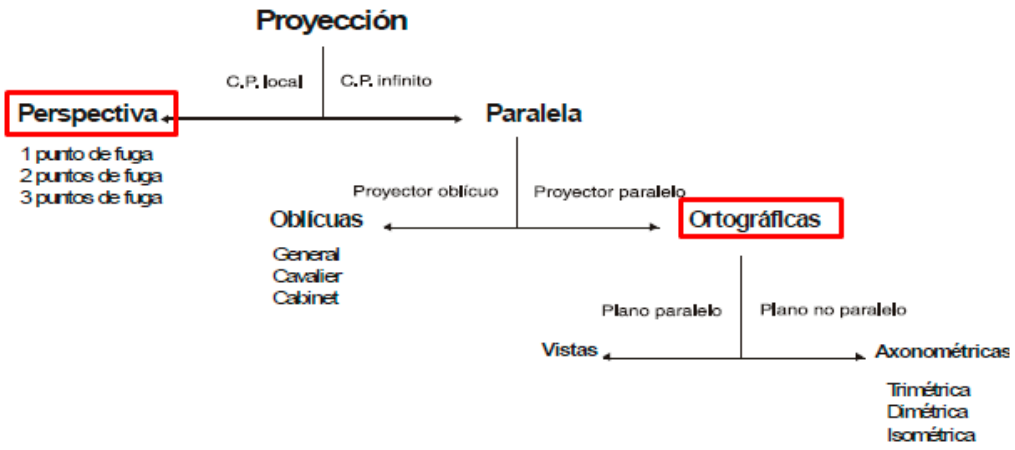



*Ilustración 82: Tipos de proyecciones*

Elementos que forman una operación de proyección:

- El ***proyector*** es una recta que pasa por el punto a y el centro de proyección.
- El ***centro de proyección*** es un punto por donde pasan, convergen, todos los proyectores
- El ***plano de proyección*** es el plano 2D donde se forma la imagen, mediante la intersección de los proyectores con este plano.

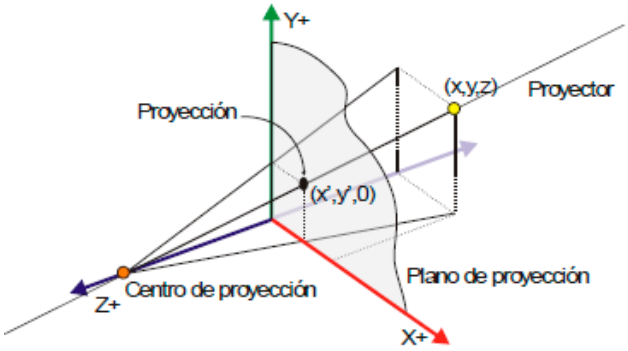



*Ilustración 83 Elementos de la proyección*

Lo que determinará si estamos ante uno u otro tipo de proyección es la configuración de estos elementos:

- Una ***proyección perspectiva*** se genera con **un único centro de proyección** en un punto concreto **y finito del eje Z** de la cámara.
- Una ***proyección ortográfica*** se consigue **ubicando el centro de proyección en el infinito del eje Z** de la cámara, de forma que todos los proyectores son paralelos

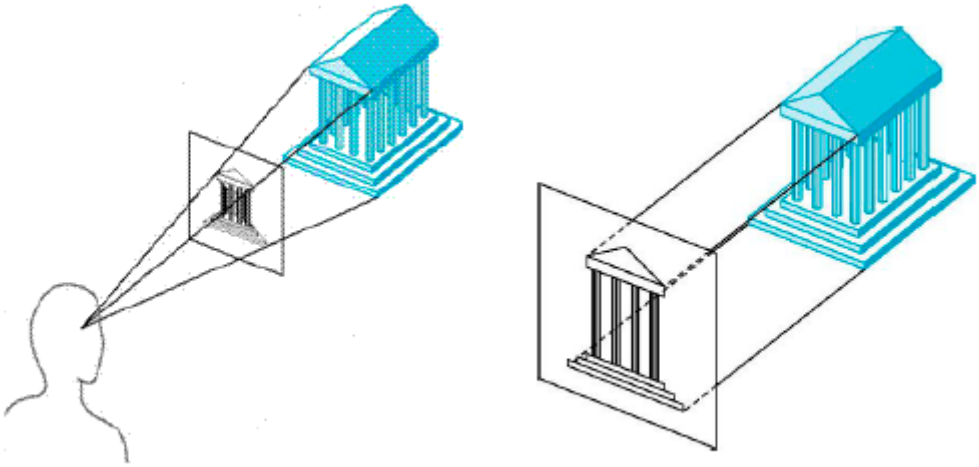


*Ilustración 85 Proyección perspectiva (izda) vs Proyeccion Ortográfica o Paralela (dcha)*

### 3.1.4. Definición de la matriz de proyección en OpenGL

En la máquina de estados de OpenGL hay una **matriz de proyección (projection matrix)** que puede ser manipulada mediante varias llamadas.

La función que primero hay que invocar antes de manipular esta matriz de proyección es **glMatrixMode, con el parámetro GL_PROJECTION**. Esto **indica a OpenGL que todas las operaciones sobre matrices** que se apliquen desde ese instante hasta que se cambie el glMatrixMode, **se aplicarán sobre la matriz de proyección**. Tras esta llamada, **inicializamos al valor identidad**:

*glMatrixMode(GL_PROJECTION);*

*glLoadIdentity();*

**Si queremos usar** en nuestra escena una **proyección perspectiva**, usaremos **glFrustum:**

*void glFrustum(GLdouble left, GLdouble right,*

*GLdouble bottom, GLdouble top,*

*GLdouble near, GLdouble far);*

**Si queremos usar** en nuestra escena una **proyección ortográfica**, usaremos **glOrtho** con los mismos parámetros que glFrustum.

*void glOrtho(GLdouble left, GLdouble right,*

*GLdouble bottom, GLdouble top,*

*GLdouble near, GLdouble far);*

### 3.1.5. Eliminación de partes ocultas

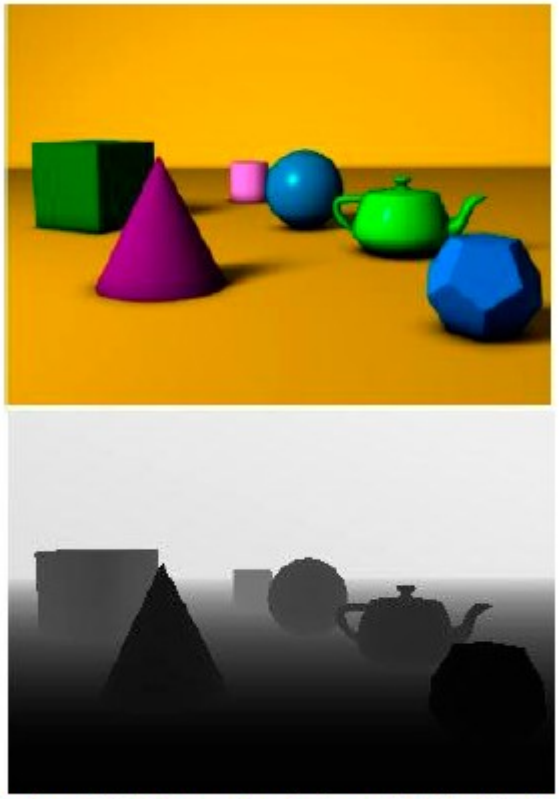


*Ilustración 89: Escena (arriba) y su Z-Buffer (abajo)*

Si un proyector atraviesa varias primitivas en su camino al centro de proyección, parece evidente que el pixel proyectado tendrá el color de la primitiva más cercana al plano de proyección, pero, **¿cómo se calcula cuál es el más cercano?**

**Una opción es pintar las primitivas siguiendo un orden**, desde la más alejada hasta la más cercana, y así nos garantizamos que la última en pintar cada pixel será la más cercana. Pero esto es **muy poco eficiente**.

Lo más usado es el **algoritmo de ZBuffer**, que se representa visualmente en la Ilustración 89. Para cada primitiva que se dibuje, **se almacena en el pixel del ZBuffer su valor de profundidad** normalizado dentro del volumen de visualización (**0 lo más cercano al plano near, y 1 lo más cercano al plano far**). De esta forma, si el valor de Z de la primitiva es menor que el actualmente existente en esa posición del buffer, se sustituye el color del pixel por ese nuevo valor y se actualiza el ZBuffer.

```
glutInitDisplaymode( GLIT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
glEnable(GL_DEPTH_TEST);
```

Cada vez que se dibuja, igual que limpiamos el buffer de color, hay que limpiar el ZBuffer:

```
glClear(GL_DEPTH_BUFFER);
```

## 3.2. Iluminación

### 3.2.1. La luz

La luz es una radiación es una radiación electromagnética, un tipo de ondas que se propagan por el espacio, similar a las ondas que se usan para las comunicaciones móviles, wifi, radio y televisión. La propiedad física que corresponde a la percepción de color es la **longitud de onda**.

El **sistema visual humano** puede percibir esta radiación sólo cuando su longitud de onda está aproximadamente **entre 390 y 750 nanómetros**.

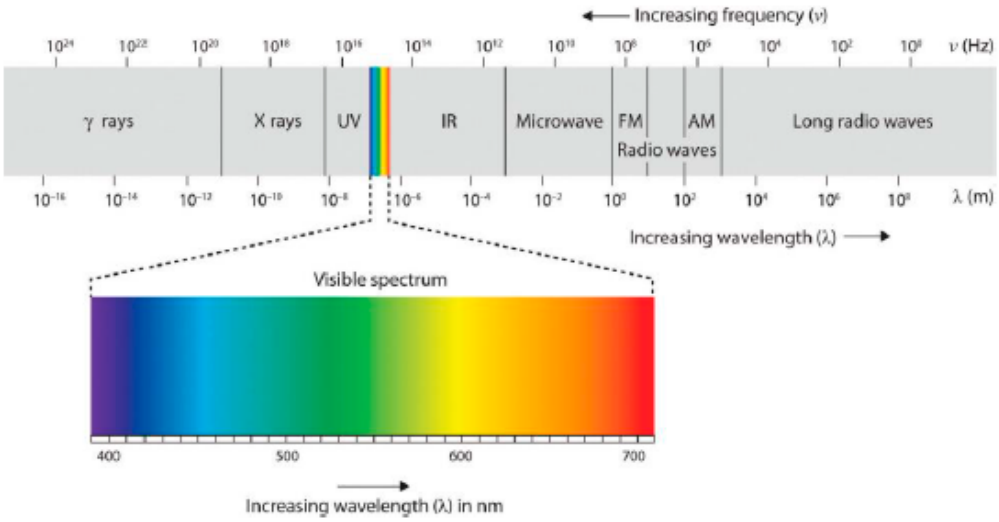



*Ilustración 90: Espectro electromagnético.*

La radiación se puede describir de forma idealizada como un **flujo** en el espacio **de partículas puntuales** llamadas **fotones**. Cada uno de estos fotones tiene una *energía radiante*, que depende de su longitud de onda.

En un **punto p** de la superficie de un objeto podemos **medir la densidad de energía radiante** en un instante de tiempo **de los fotones de una longitud de onda concreta (λ)** que pasan en una **determinada dirección (v).** Este valor **L(λ,p,v) se denomina radiancia**, y es lo que **determina el tono de color y el brillo.**

El **brillo** o intensidad dependerá de la *cantidad de fotones*. El **color** dependerá de la *distribución de las longitudes de onda* de dichos fotones.

### 3.2.2. El espacio RGB

**Cualquier color reproducible en un dispositivo** se puede **representar por una terna (r,g,b)** con los **tres valores comprendidos entre 0 y 1**. El valor **0 indica que el correspondiente color no aparece** y el valor **1 significa la máxima potencia para ese color**.

### 3.2.3. Modelo de iluminación simplificado

La radiación electromagnética visible se genera en las fuentes de luz, que pueden ser:

- Fuentes naturales: sol o estrellas, fuego, objetos incandescentes, órganos de algunos animales, etc…
- Fuentes artificiales o luminarias: filamentos incandescentes, tubos fluorescentes, LEDs, etc.

OpenGL realiza en su forma más básica diferentes simplificaciones:

1. Las fuentes de luz son **puntuales o unidireccionales**, y hay un número finito de ellas (**hasta 8**)
2. No se considera la luz incidente en una superficie que no provenga directamente de las fuentes de luz. Para suplir esos fotones que andan "rebotando" por la escena, se crea una **radiancia ambiente** constante.
3. Los **objetos** o polígonos son **totalmente opacos** (no hay transparencias ni materiales translúcidos).
4. No se consideran sombras arrojadas, es decir**, un objeto no impide la trayectoria de la luz**.
5. **El espacio entre los objetos no dispersa la luz** (ésta tiene igual densidad independientemente de la distancia a la fuente)
6. En lugar de considerar todas las longitudes de onda posibles, se **usa el modelo RGB**.

*Ilustración 95 Iluminación compleja (izda.) vs Iluminación simplificada (dcha.)*

Para este modelo simplificado, podemos considerar dos tipos genéricos de reflexiones en la superficie: la reflexión **especular** y la **difusa.**

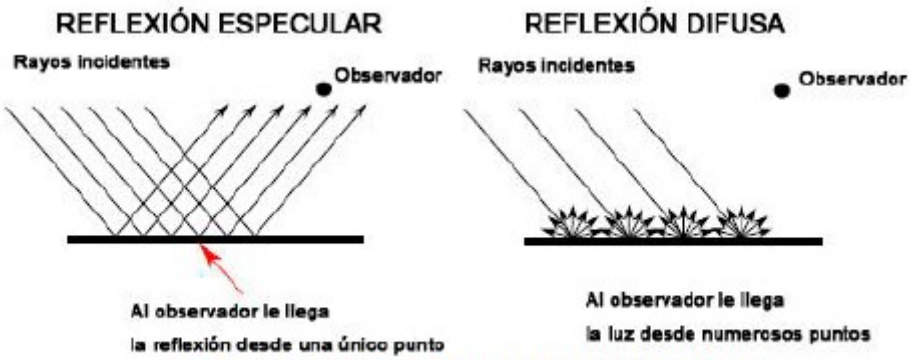



*Ilustración 96: Reflexión Especular vs Reflexión Difusa*

### 3.2.4. Fuentes de luz

En el modelo de la escena se pueden incluir un conjunto de n fuentes de luz, pudiendo ser cada una de ellas de **dos tipos**:

- **Fuentes de luz posicionales**, que ocupan un punto en el espacio y emiten en todas direcciones.
- **Fuentes de luz direccionales**, que están en un punto a distancia infinita y tienen un vector director en el que se emiten los fotones.

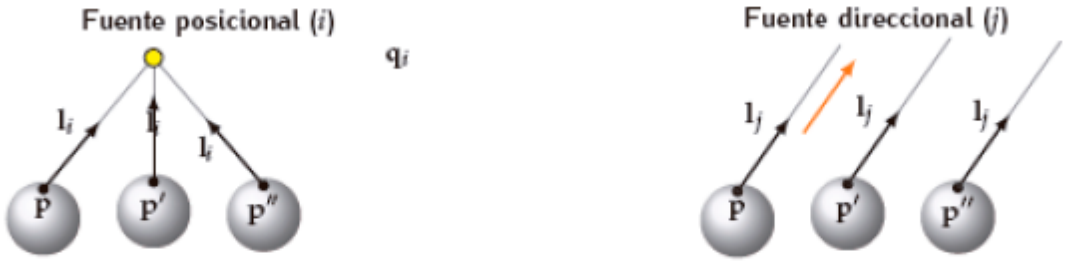



*Ilustración 97 Fuente de luz posicional vs direccional*

### 3.2.5. Normales

La **iluminación en un punto p depende la orientación de la superficie en dicho punto**. Esta **orientación está caracterizada por el vector normal np** asociado a dicho punto.

En modelos de fronteras, puede calcularse de varias formas:

- Considerar que **np es la normal del triángulo (caras) que contiene a p.**
- Aproximar el valor de **np en cada vértice como la media de las normales de los triángulos que lo comparten**.

```
void Object3D::calcularNormalesCaras(){
  _vertex3f a, b, c;
  // Calculo
  for(unsigned int i = 0; i < caras.size(); i++){
    a._0 = vertices[caras[i].l2].x - vertices[caras[i].l1].x;
    a._1 = vertices[caras[i].l2].y - vertices[caras[i].l1].y;
    a._2 = vertices[caras[i].l2].z - vertices[caras[i].l1].z;

    b._0 = vertices[caras[i].l3].x - vertices[caras[i].l1].x;
    b._1 = vertices[caras[i].l3].y - vertices[caras[i].l1].y;
    b._2 = vertices[caras[i].l3].z - vertices[caras[i].l1].z;

    c = a.cross_product(b);
    c.normalize();
    normalesCaras.push_back(c);
  }
}
```

```
void Object3D::calcularNormalesVertices(){
  // Inicializo a 0 todas las normales de los vertices
  _vertex3f a;
  a._0=0.0;
  a._1=0.0;
  a._2=0.0;;
  for(unsigned int i = 0; i < vertices.size(); i++){
    normalesVertices.push_back(a);
  }

  // Sumo las normales de las caras adyacentes a los vertices
  for(unsigned int i = 0; i < caras.size(); i++){
    normalesVertices[caras[i].l1] += normalesCaras[i];
    normalesVertices[caras[i].l2] += normalesCaras[i];
    normalesVertices[caras[i].l3] += normalesCaras[i];
  }

  // Normalizo las normales de los vertices
  for(unsigned int i = 0; i < normalesVertices.size(); i++){
    normalesVertices[i].normalize();
  }
}
```

### 3.2.6. Materiales

Debemos tener claro que **el color no es una simple terna RGB**, sino que **está compuesto de dos elementos**: *las características de la luz incidente y del material.*

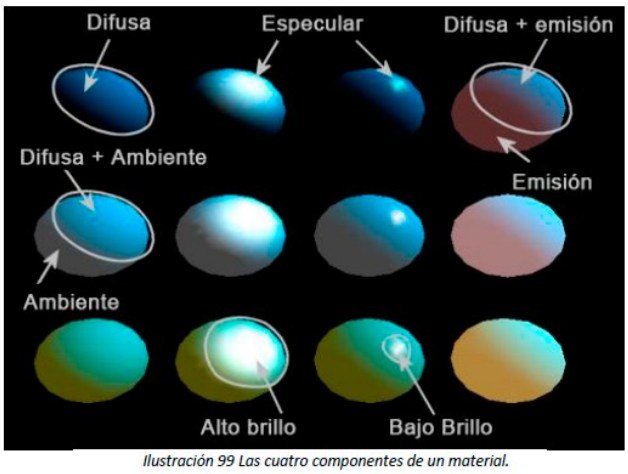



*Ilustración 99 Las cuatro componentes de un material.*

Un material se puede definir mediante **cinco componentes**:

- Color **difuso ($M_D$)**, que indica el color "base" del material.
- Color **especular ($M_S$)**, que indica el color de los brillos.
- Color **ambiente ($M_A$)**, que se refiere al comportamiento del objeto cuando no le incide directamente ninguna fuente de luz.
- Color de **emisión ($M_E$)**, y se refiera a un comportamiento "artificial" que hace que se vea el objeto incluso cuando no hay fuentes de luz activas, pero sin ser él una fuente de luz (algo así como las *pegatinas fluorescentes*).
- **Brillo (e)**, es decir, la cantidad de luz que es rebotada de forma especular.

```
// Material Rubi
materialRubi.emision[0] = 0.0; materialRubi.emision[1] = 0.0; materialRubi.emision[2] = 0.0; materialRubi.emision[3] = 1.0;
materialRubi.ambiente[0] = 0.1745; materialRubi.ambiente[1] = 0.01175; materialRubi.ambiente[2] = 0.01175; materialRubi.ambiente[3] = 1.0;
materialRubi.difuso[0] = 0.61424; materialRubi.difuso[1] = 0.04136; materialRubi.difuso[2] = 0.04136; materialRubi.difuso[3] = 1.0;
materialRubi.especular[0] = 0.727811; materialRubi.especular[1] = 0.626959; materialRubi.especular[2] = 0.626959; materialRubi.especular[3] = 1.0;
materialRubi.brillo = 65.3;
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, materialRubi.emision);
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialRubi.ambiente);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialRubi.difuso);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialRubi.especular);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, materialRubi.brillo);
```

### 3.2.7. Iluminación en OpenGL

Es necesario usar la orden **glEnable/glDisable** para activar o desactivar la funcionalidad de iluminación:

*glEnable(GL_LIGHTING); // activa el modelo de iluminación local*

*glDisable(GL_LIGHTING); / / desactiva el modelo de iluminación local*

Esta funcionalidad está **por defecto desactivada** en el estado inicial de OpenGL.

### 3.2.8. Colores vs iluminación

Depende de si la evaluación del modelo de iluminación local está activada o no lo está:

- Con la **iluminación desactivada**, el color de las primitivas dibujadas depende de una terna RGB del estado interno, que se modifica con *glColor*.
- Con la **iluminación activada**, el color resultante se calcula a partir de las características de las luces y los materiales, en lugar del especificado con glColor.

En su **estado interno**, OpenGL **mantiene un conjunto de ternas RGB** que constituyen los **parámetros más importantes** del modelo de iluminación local. Son los siguientes:

- Una **luz ambiente global**, que "viene de la nada". Se especifica con:
  *GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };*
  *glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);*
- Para las **luces**:
  - **SiA, SiD, SiE** componentes **ambiental, difusa y especular** de la i-ésima fuente de luz.
  - **qi, li posición o dirección** de la i-ésima fuente de luz.
- para los **materiales**, en el momento de dibujar una primitiva se consulta:
  - **ME emisividad** del material.
  - **MA,MD,MS reflectividad difusa, ambiente y especular** del material**.**
  - **e** exponente de la componente pseudo-especular.

### 3.2.9. Definición de fuentes de luz: tipos y atributos

#### *3.2.9.1. Activación y desactivación*

Las implementaciones de OpenGL están obligadas a gestionar un número **mínimo de 8 fuentes de luz** (GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7). Cada una de estas fuentes de luz puede activarse y desactivarse de forma individual, con:

```
glEnable(GL_LIGHTi) ; / / activa la i-esima fuente de luz
glDisable(GL_LIGHTi) ; / / desactiva la i-esima fuente de luz
```

Solo las fuentes activas intervienen en el modelo de iluminación local (**por defecto están todas desactivadas**).

#### *3.2.9.2. Configuración de colores de una fuente*

Se hace con la función **glLightf**.

```
const float
caf[4] = { ra, ga, ba, 1.0 }, / / color ambiental de la fuente
cdf[4] = { rd, gd, bd, 1.0 }, / / color difuso de la fuente
csf[4] = { rs, gs, bs, 1.0 }; / / color especular de la fuente
glLightfv( GL_LIGHTi, GL_AMBIENT, caf ) ; / / hace SiA := (ra, ga, ba)
glLightfv( GL_LIGHTi, GL_DIFFUSE, cdf ) ; / / hace SiD := (rd, gd, bd)
glLightfv( GL_LIGHTi, GL_SPECULAR, csf ) ; / / hace SiS := (rs, gs, bs)
```

#### *3.2.9.3. Configuración de posición/dirección de una fuente.*

**La posición (en luces posicionales) o dirección (en direccionales)** se especifica con una llamada a **glLightfv**, de esta forma:

```
const GLfloat posf[4] = { px, py, pz, 1.0 } ;  // (x,y,z,w)
glLightfv( GL_LIGHTi, GL_POSITION, posf );

const GLfloat dirf[4] = { vx, vy, vz, 0.0 } ; // (x,y,z,w)
glLightfv( GL_LIGHTi, GL_POSITION, dirf );
```

El **valor de w**, el cuarto valor del vector, determina el **tipo de fuente de luz**. Si es 0, es una luz direccional (p.ej. el sol). Si w es distinto de cero, estamos en una luz posicional (p.ej. una bombilla de una lámpara), e irradia en todas direcciones.

### 3.2.10. Observador local o en el infinito

La función **glLightModel** permite configurar este comportamiento.

Si la **proyección es ortogonal**, (el observador está en el infinito), es necesario hacer esta llamada:

*glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE );*

Si la **proyección es perspectiva**, se dice que el observador es local (no está en el infinito), y en este caso debemos indicar:

*glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );*

## 3.3. Texturas

**Una textura no es más que una imagen que representa las modificaciones de los parámetros anteriormente indicados**. Se puede interpretar como una **función T** que asocia a cada **punto s de un dominio D** (usualmente [0, 1] x [0, 1]) **un valor para un parámetro del modelo de iluminación global** (típicamente MD y MA. La función T determina como varía el parámetro en el espacio.

La función T puede estar representada en memoria como una **matriz de pixels RGB**, a cuyos pixels se les llama ***texels* (texture elements)**. A esta imagen se le llama imagen de textura.

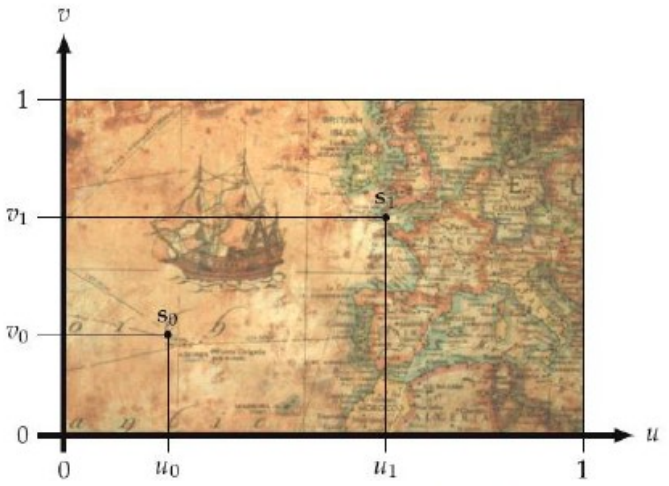



*Ilustración 105 Textura 2D*

$$T(\mathbf{s}_1) = T(u_1, v_1) = (r_1, g_1, b_1)$$

$$T(\mathbf{s}_0) = T(u_0, v_0) = (r_0, g_0, b_0)$$

### *3.3.1. Coordenadas de textura*

Para poder **aplicar una textura a la superficie de un objeto**, es necesario hacer corresponder cada **punto p = (x, y, z)** de su superficie con un **punto sp=(u, v)** del dominio de la textura:

Debe existir una **función f** tal que (u, v) = f (x, y, z). Entonces decimos que (u, v) son las coordenadas de textura del punto p.

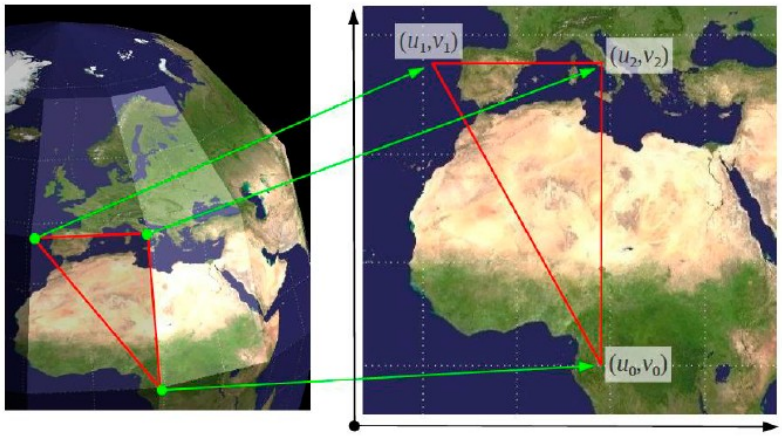



*Ilustración 106 Coordenadas de textura de los vértices de un triángulo.*

La asignación de coordenadas de textura se puede hacer de **dos formas**:

- **Asignación explícita a vértices**: Las coordenadas forman parte de la definición del modelo de escena, y son un dato de entrada en forma de un vector o tabla de coordenadas de textura de vértices (v0, u0), (v1, u1), . . . (un-1, vn-1). Se puede hacer:
  - **manualmente en objetos sencillos**, o bien
  - de **forma asistida usando software para CAD** (p.ej. 3D Studio).
- **Asignación procedural**: La función $f$ se implementa como un algoritmo `CoordText(p)` que se invoca para calcular las coordenadas de textura, de forma que acepta un punto p como parámetro y devuelve el par $(u, v) = f(p)$ con las coordenadas de textura de $p$. Esta asignación procedural se puede hacer de dos formas
  - ***Asignación procedural a vértices***, es decir, usamos una función (coordText(vi)) para realizar la asignación por vértices, pero una vez creado el vector de coordenadas de textura, se comporta igual que en el caso de la asignación explícita a vértices.
  - ***Asignación procedural a puntos***, en cuyo caso `coordText(p)` es invocado cada vez que hay que calcular el color de un punto $p$ de la superficie durante los cálculos del modelo de iluminación local.

Los tipos de funciones f son de lo más diversos, por lo que **nos centraremos sólo en tres aproximaciones**, por ser las más usadas:

- **Asignación procedural por coordenadas paramétricas.**

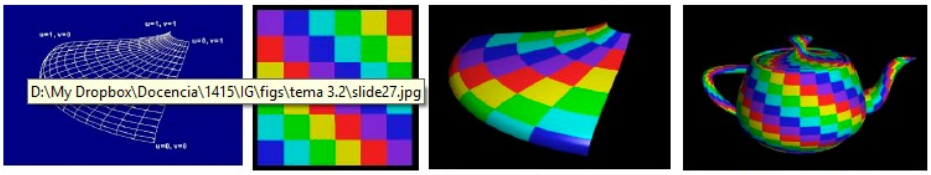



*Ilustración 110 Parche paramétrico, al que aplicándose la textura (b), resulta un parche texturizado. A la derecha, tetera formada por varios parches parametrizados.*

*Ilustración 111 Aplicación de coordenadas cilíndricas sobre varios objetos.*

**Asignación procedural por coordenadas cilíndricas.**

- **Asignación procedural por coordenadas esféricas.**

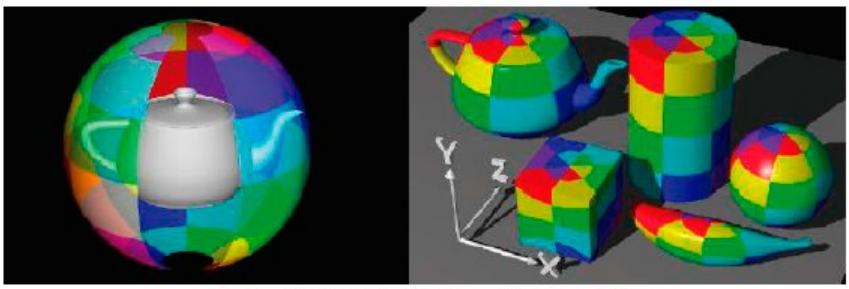



*Ilustración 112 Aplicación de coordenadas esféricas.*

### 3.3.2. Activación y desactivación.

Las ordenes glEnable y glDisable se pueden usar para activar o desactivar toda la funcionalidad de OpenGL relacionada con las texturas, que se encuentra **inicialmente desactivada**:

*glEnable( GL_TEXTURE_2D ) ; / / habilita texturas*

*glDisable( GL_TEXTURE_2D ) ; / / deshabilita texturas*

Cuando se habilitan las texturas hay una textura activa en cada momento, que se consulta cada vez que un polígono se proyecta en un pixel, antes de calcular el color de dicho pixel:

- con **iluminación desactivada**, el color de la textura sustituye al especificado con glColor.

- con **iluminación activada**, el color de la textura sustituye a las reflectividades del material (usualmente a la **difusa y la ambiental**).

*Todas las operaciones de texturas requieren que esta funcionalidad esté activada*.

### 3.3.3. Textura activa

Para cambiar el identificador de textura activa podemos hacer:

```
glBindTexture( GL_TEXTURE_2D, idTex ); // activa textura con id 'idTex'
```

# Tema 4 – Interacción y animación

## 4.1. Interacción

Un sistema gráfico interactivo es un **sistema que responde de forma directa a las acciones del usuario**. Por ejemplo:

- Sistemas CAD.
- Videojuegos.
- Simuladores.

Un sistema gráfico interactivo **debe contener al menos un dispositivo de entrada y uno de salida**, por lo que al menos necesitaremos un teclado y un monitor, aunque hoy en día casi no podemos sobrevivir sin tener un ratón. A estos dispositivos podemos añadirles un diverso conjunto de dispositivos de entrada:

- Joystick
- Pantalla táctil
- Tableta digitalizadora
- Phantom hápticos

Y por supuesto, **dispositivos de salida** de lo más avanzados:

- Cuevas de realidad virtual
- Cascos de visualización inmersiva (p.ej. oculus Rift®)
- Gafas de Realidad aumentad (p.ej. Google Glass©)
- Pantallas estéreo
- Dispositivos móviles
- Hologramas


*Ilustración 118 Flujo de interacción*

El tiempo total del ciclo debe **ser inferior a 0.03 segundos en aplicaciones interactivas no hápticas,** e **inferior a 0.001 segundos en aplicaciones hápticas**.

A lo largo del proceso de interacción es conveniente proporcionar al usuario un mecanismo de **realimentación**, que le permita conocer los pasos seguidos por el sistema. Entre las técnicas de realimentación encontramos:

- Mostrar el estado del sistema (p.ej. modo de cámara)
- Indicar la herramienta activa
- Resaltar elementos seleccionables
- Cambios de cursor según acción permitida

Todo **sistema gráfico que se considere interactivo** deberá **permitir leer posiciones y seleccionar** componentes delo modelo geométrico. La lectura de posiciones permite generar puntos en coordenadas del mundo, mientras que la selección permite identificar elementos de la escena.

Toda esta interacción se puede realizar no sólo con los clásicos ratones y teclado, sino con cualquier otro tipo de dispositivos físicos avanzados, pero al final, lo que tenemos en cualquiera de los casos es una coordenada (x,y) de la pantalla. Esto no es más que una abstracción del **dispositivo hardware** a un **dispositivo lógico**.

Hay dos tipos de dispositivos lógicos principales:

- **Locator**, que lee posiciones (X,Y) de la pantalla. Su sistema de coordenadas es el de la pantalla, por lo que habrá que realizar la transformación inversa de la visualización para obtener la coordenada del mundo a la que se corresponde.
- **Pick**, que lee identificadores de componentes del modelo geométrico. Es un dispositivo lógico que consulta la escena y devuelve el objeto sobre el que está el cursor.

### 4.1.1. Gestión de eventos.

Los eventos son **fundamentales en cualquier sistema interactivo**. Los eventos son **generados por las acciones del usuario o de forma automática por el sistema**. Cada librería de interfaces de usuario dispone de su propio sistema gestor de eventos, podemos destacar:

- Event Listeners, en Java
- Callbacks en GLUT
- Signal/Slot en QT

#### *4.1.1.1. Gestión de eventos en GLUT*

GLUT gestiona las entradas en **modo evento**, esto es, tiene un buffer donde se van almacenando las peticiones realizadas por el usuario. **El gestor de eventos** de GLUT **va leyendo ese buffer** y, si el evento tiene asociada una función que lo procesa (un callback), ejecuta dicha función y reacciona al evento recibido. Si no hay un callback para el evento, se ignora

Los callback más usados en *glut* son:

- **glutDisplayFunc**: Redibujado.
- **glutMouseFunc**: Pulsación de botones del ratón.
- **glutMotionFunc**: Movimiento del ratón mientras se pulsa un botón.
- **glutPassiveMotionFunc**: Movimiento del ratón sin pulsar botones.
- **glutReshapeFunc**: Cambio de tamaño de la ventana.
- **glutKeyFunc**: Pulsación de tecla.
- **glutIdleFunc**: Ausencia de eventos externos.
- **glutTimerFunc**: Temporizador.

#### *4.1.1.2. Eventos de pulsación de botones del ratón.*

Si asociamos el **callback** con la función adecuada, llamada **raton**:

*glutMouseFunc( raton );*

siendo

*void raton( GLint button, GLint state, GLint x, GLint y )*

podemos conocer mucha información:

- **button** devuelve que botón se accionó:
  - GLUT_LEFT_BUTTON,
  - GLUT_MIDDLE_BUTTON,
  - GLUT_RIGHT_BUTTON
- **state** es si se ha pulsado o soltado:
  - GLUT_UP,
  - GLUT_DOWN
- **x e y** determinan la posición del cursor en coordenadas de pantalla.

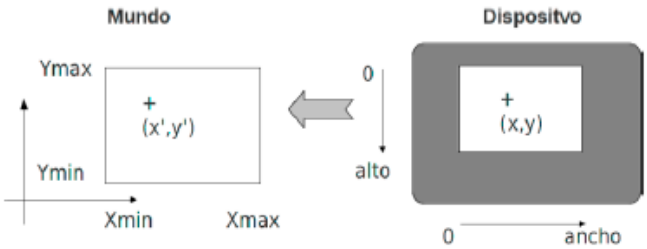
### 4.1.2. Posicionamiento

La posición que da el dispositivo lógico de posicionamiento en pantalla **es una coordenada 2D en el sistema de coordenadas de la pantalla**, pero si queremos conocer a qué punto de coordenadas del mundo corresponde, **hay que realizar la correspondiente conversión**. Para ello se puede:

- invertir la transformación de visualización (si estamos en un entorno 2D)
- restringir el posicionamiento a un plano
- utilizar un cursor 3D
- utilizar vistas con tres proyecciones paralelas
- invertir la transformación de visualización (si el dispositivo de entrada es 3D)

#### *4.1.2.1. Posicionamiento 2D*

Es el caso más fácil, y se da por ejemplo en los sistemas de edición de imágenes (Photoshop©, GIMP, etc.). En este contexto, la proyección es ortogonal y la conversión es casi inmediata.




$$x' = X_{min} + x * (X_{max} - X_{min})/ancho$$

$$y' = Y_{max} - y * (Y_{max} - Y_{min})/alto$$

#### *4.1.2.2. Posicionamiento 3D*

La introducción de posiciones 3D, utilizando dispositivos de entrada 2D, requiere el uso de **técnicas especiales.**

Existen varias técnicas para generar los desplazamientos en las tres direcciones, a partir de los del dispositivo 2D. La más simple es utilizar un botón, o tecla, para conmutar la interpretación del movimiento del dispositivo, del plano X-Y al X-Z.

### 4.1.3. Control de cámara

La cámara tiene demasiados parámetros para controlarlos todos interactivamente usando un ratón. Habitualmente **se controla de forma interactiva un número reducido de parámetros**, seleccionados dependiendo del tipo de aplicación:

- En **visualización de modelos** estamos en una cámara que mira siempre al mismo punto y gira en torno a él como si estuviéramos recorriendo una esfera. Es lo que se denomina una **cámara orbital.**

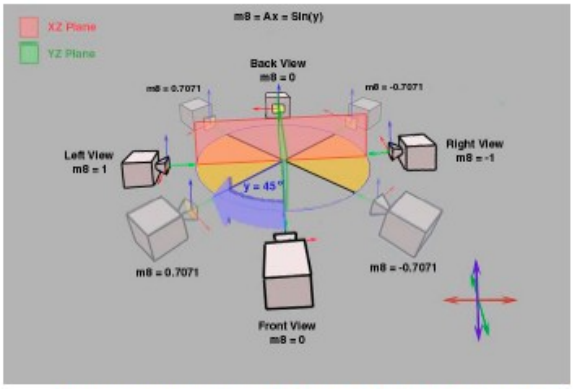



*Ilustración 121 Cámara orbitando alrededor de un punto.*

- En la **exploración de escenarios** estamos en una cámara **en primera persona**, de forma que la cámara está situada en un punto fijo del sistema de coordenadas de un objeto. Una cámara en **tercera persona** es una cámara que sigue a un objeto, sobre la cual no hay control salvo el del avatar que se mueve.

*Ilustración 122 Cámara en primera persona.*

*Ilustración 123 Cámara en tercera persona.*

### 4.1.4. Selección

La selección permite al usuario **referenciar componentes de un modelo geométrico** y es esencial en cualquier aplicación gráfica que requiera la edición de un modelo.

El proceso de selección suele realizarse como un posicionamiento y una búsqueda en el modelo geométrico, para lo que es necesario tener identificados a los objetos de la escena de alguna manera.