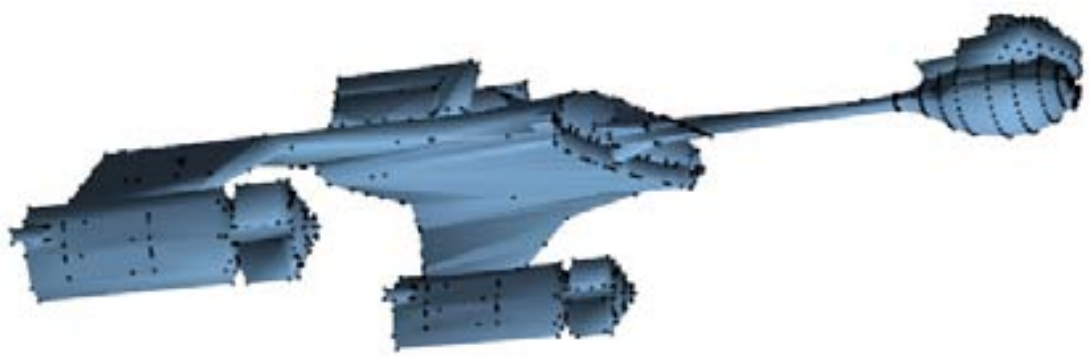


# Tema 2: Modelado

J. Ribelles

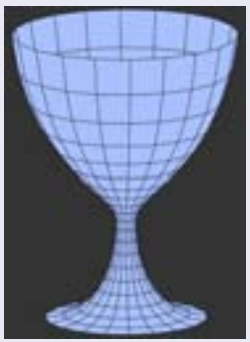
SIE020: Síntesis de Imagen y Animación
Institute of New Imaging Technologies, Universitat Jaume I

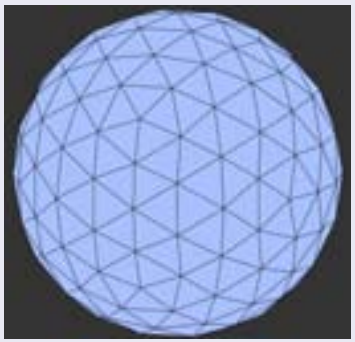
# Contenido

# Introducción

## Modelado

- Se denomina modelo al conjunto de datos que describe a un objeto y que puede ser utilizado por un sistema gráfico para ser visualizado.
- Modelado poligonal es cuando se utilizan polígonos para describir la geometría.
- El triángulo es la primitiva más utilizada.
- OpenGL no proporciona mecanismos para describir o modelar objetos geométricos complejos, sino que proporciona mecanismos para especificar cómo dichos objetos deben ser dibujados.

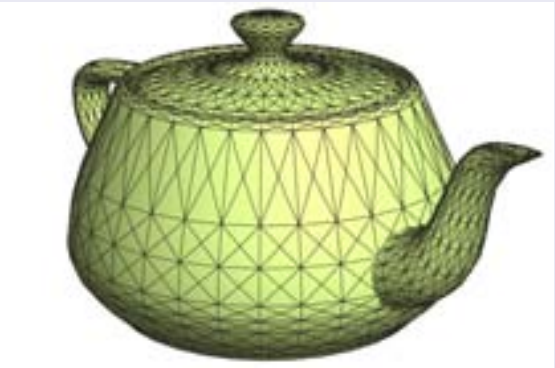




# Modelado poligonal

## Modelado poligonal

- Un modelo polígonal, además de vértices y caras, suele almacenar atributos como el color, la normal o las coordenadas de textura.
- Estos atributos son necesarios para mejorar significativamente el realismo visual.



- La normal es un vector perpendicular a la superficie en un punto, ¿cómo obtenerla?
- Ya que el producto vectorial no es commutativo, hay que establecer un orden y obtener todas las normales de manera consistente.

## Ejercicio

Observa la siguiente descripción poligonal de un objeto. Las líneas que comienzan por *v* se corresponden con los vértices e indican sus coordenadas. El primero es el número 1 y los demás se enumeran de forma consecutiva. Las líneas que comienzan por *f* se corresponden con las caras e indican qué vertices lo forman.

```
v 0 0 0        f 1 3 2
v 0 0 1        f 1 4 3
v 1 0 1        f 1 2 5
v 1 0 0        f 2 6 5
v 0 1 0        f 3 2 6
v 0 1 1        f 3 6 7
v 1 1 1        f 3 4 7
v 1 1 0        f 4 8 7
               f 4 1 8
               f 1 5 8
```

- Dibújalo en papel, ¿qué objeto representa?
- ¿Están todas sus caras definidas en el mismo orden?
- ¿En qué sentido están definidas, horario o antihorario?

# Polígonos y OpenGL

## Primitivas geométricas

- El punto, el segmento de línea y el triángulo.
- Cada primitiva se define especificando sus respectivos vértices.
  - Dibujo de puntos:
    - GL_POINTS
  - Dibujo de líneas:
    - Segmentos sueltos: GL_LINES
    - Secuencia o tira de segmentos: GL_LINE_STRIP
    - Secuencia cerrada de segmentos: GL_LINE_LOOP
  - Triángulos
    - Triángulos sueltos: GL_TRIANGLES
    - Tira de triángulos: GL_TRIANGLE_STRIP
    - Abanico de triángulos: GL_TRIANGLE_FAN
- Las primitivas geométricas se forman agrupando vértices, y estas se agrupan a su vez para definir objetos de mayor complejidad.

## Modelado

- Solemos asociar el concepto de vértice con las coordenadas.
- El concepto de vértice es más general entendiéndose como una agrupación de datos llamados atributos.
- Los atributos más utilizados son la posición, la normal y el color.
- El programador pueda incluir como atributo cualquier información.
- Como OpenGL requiere que la información que vaya a visualizarse se disponga en vectores...

```
struct vertice
{
  GLfloat coordenadas[3];
  GLfloat color[3];
}
Vertices[nVertices];

struct triangulo
{
  GLuint indices[3];
}
Triangulos[nTriangulos];
```

## Visualización

- Tres pasos previos:
  1. Almacenar el modelo poligonal en *buffer objects*.
  2. Obtener los índices de las variables del *Shader* que representan los atributos de los vértices.
  3. Especificar para cada atributo dónde y cómo se encuentran almacenados así como habilitar los vectores correspondientes.
- Dibujar indicando tipo de primitiva y número de elementos.

Listado 1: Paso 1

```
enum {bufferVertices, bufferTriangulos, nBuffers}
GLuint buffers[nBuffers];

glGenBuffers (nBuffers, buffers);

glBindBuffer (GL_ARRAY_BUFFER, buffers[bufferVertices]);
glBufferData (GL_ARRAY_BUFFER, nVertices*sizeof(vertice), Vertices, GL_STATIC_DRAW);

glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, buffers[bufferTriangulos]);
glBufferData (GL_ELEMENT_ARRAY_BUFFER, nTriangulos*sizeof(triangulo), Triangulos,
      GL_STATIC_DRAW);
```

# Listado 2: Paso 2

```
const char *vertexShaderSource =
{
  "#version 140\n"
  ""
  "in  vec3 posicion;"
  "in  vec3 color;"
  "out vec4 nuevoColor;"
  ""
  "void main()"
  "{"
  "    nuevoColor  = vec4 (color, 1.0);"
  "    gl_Position = vec4 (posicion, 1.0);"
  "}"
};

const char *fragmentShaderSource =
{
  "#version 140\n"
  ""
  "in  vec4 nuevoColor;"
  "out vec4 colorFragmento;"
  ""
  "void main(void)"
  "{"
  "    colorFragmento = nuevoColor;"
  "}"
};

... // compila y enlaza el Shader

GLuint iPosicion = glGetAttribLocation (program, "posicion");
GLuint iColor    = glGetAttribLocation (program, "color");
```

# Listado 3: Paso 3 y dibujado

```
glBindBuffer (GL_ARRAY_BUFFER, buffers[bufferVertices]);

glVertexAttribPointer (iPosicion, 3, GL_FLOAT, GL_FALSE,
                       sizeof(vertice), (GLvoid*)0);
glVertexAttribPointer (iColor,    3, GL_FLOAT, GL_FALSE,
                       sizeof(vertice), (GLvoid*)(sizeof(GLfloat)*3));

glEnableVertexAttribArray (iPosicion);
glEnableVertexAttribArray (iColor);

glBindBuffer   (GL_ELEMENT_ARRAY_BUFFER, buffers[bufferTriangulos]);

glDrawElements (GL_TRIANGLES, nTriangulos*3, GL_UNSIGNED_INT, 0);
```