Informática Gráfica

Juan Carlos Torres

Curso 2024/25

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Granada

Disclaimer
You can edit this page to suit your needs. For instance, here we have a no copyright statement, a colophon and some other information. This page is based on the corresponding page of Ken Arroyo Ohori's thesis, with minimal changes.

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

Modelos geométricos 2

2.1 Representaciones poligonales 11
2.2 Visualización con OpenGL 12
2.2.1 Caras traseras 13
2.3 Cálculo de normales de cara 13
2.3.1 Normales de vértice . . . 14
2.4 Sopa de triángulos 14
2.5 Formatos de archivo . . . 15
2.6 Herranmientas: MeshLab 16
2.7 Ejercicios 18

Un modelo en Informática Gráfica es la representación computacional de una escena u objeto. El modelo describe tanto la forma como la apariencia del objeto. La forma del objeto es esencialmente información geométrica. La apariencia del objeto describe como se comporta ante la luz que recibe. En este tema nos centraremos en las propiedades geométricas. Hablamos de modelos geométricos cuando nos centramos en la representación geométrica de los objetos.

Hay diversos métodos de representación de modelos geométricos: basados en la representación de la superficie exterior del objeto; descomposición del espacio ocupado por el objeto en celdas y enumeran las celdas que ocupa (ver Figura 2.1); ecuaciones implícitas; en el proceso de construcción del objeto a partir de primitivas simples con operaciones booleanas (ver Figura 2.2); o en ecuaciones paramétricas 2.3, por citar algunas.

En este capítulo nos centraremos en la representación de la superficie exterior del objeto mediante un conjunto de polígonos (es decir poliedros). Veremos como podemos es posible con esta representación visualizar superficies curvas.

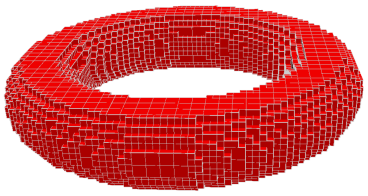

Figura 2.1: Representación de un toroide con un octree.

2.1. Representaciones poligonales

Una superficies poligonal o malla es una representación (discreta) de superficie. Las mallas son una forma común de representar objetos tridimensionales en informática gráfica. Están compuestas por tres elementos básicos: vértices, aristas y caras. Un vértice es un punto en el espacio tridimensional, una arista es una línea que conecta un par de vértices, y una cara, en el caso de las mallas de triángulos, es el área delimitada por tres aristas conectadas.

En la superficie aparecen tres tipos de elementos geométricos (Figura 2.4):

Vértices Un vértice es un punto en el espacio. Está definido por sus tres coordenadas (x, y, z).

Aristas Una arista es un segmento de recta en el espacio delimitado por dos vértices. Se puede representar como un par .

Polígonos Un polígono es una porción de un plano delimitada por una secuencia de aristas conectadas por vértices. Podemos describir el polígono mediante la secuencia ordenada de sus vértices.

Un polígono es simple si sus aristas no consecutivas no se intersectan y las consecutivas se intersectan solo en los vértices comunes. Los polígonos simples pueden ser convexos o concavos. Es **concavo** cuando la recta que contiene alguna de sus aristas divide al polígono en dos, y **convexo** en caso contrario.

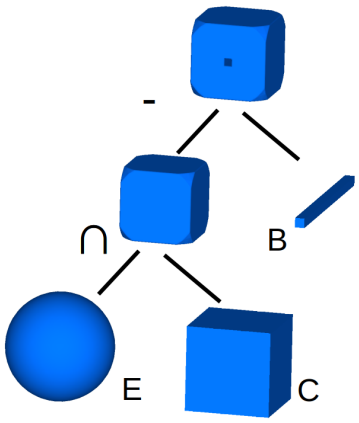

Figura 2.2: Representación de un objeto como una expresión CSG. El objeto se representa como $(E \cap C) - B$.

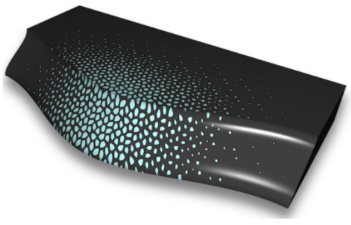

Figura 2.3: Objeto representado mediante volúmenes Bézier paramétricos.

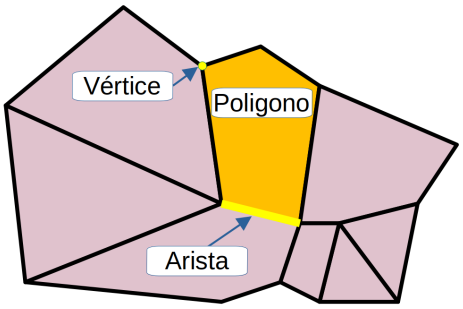

Figura 2.4: Las superficies poligonales están formadas por vértices, aristas y polígonos.

2.2. Visualización con OpenGL

OpenGL utiliza **primitivas** geométricas simples: puntos, líneas, triángulos, cuadriláteros y polígonos, como los bloques constructivos básicos para crear formas más complejas. Por ejemplo, un cuadrado se puede construir usando dos triángulos (o un cuadrilatero), y una esfera se puede aproximar con un gran número de triángulos que aproximan su superficie.

Para dibujar una primitiva (usando legacy mode) se debe llamar a la función **glBegin** indicando como argumento la primitiva que se va a dibujar. A continuación se pasa la información de la (o las) primitivas a dibujar. Los valores que se pueden pasar a **glBegin** para dibujar primitivas simples son los siguiente (Figura 2.5):

GL_POINTS Dibuja un punto en cada uno de los vértices.
GL_LINES Dibuja lineas uniendo cada par de vértices.
GL_TRIANGLES Dibuja los triangulos formados cada terna de vértices.
GL_QUADS Dibuja cuadrilateros formados por cada grupo de cuatro vértices.
GL_POLYGON Dibuja el polígono formado por la secuencia de vértices completa. Los vértices deben definir un polígono convexo simple.

Los vértices a dibujar se pasan uno a uno con llamadas a la función **glVertex3f**. Se termina con una llamada a **glEnd**. El ejemplo 2.1 se dibuja dos triángulos vecinos a través de la arista (P_0,P_1):

Listing 2.1: Dibujo de dos triángulos

```
1  glBegin(GL_TRIANGLES);
2    // Primer triangulo
3    glVertex3f(x0,y0,z0);
4    glVertex3f(x1,y1,z1);
5    glVertex3f(x2,y2,z2);
6
7    // Segundo triangulo
8    glVertex3f(x1,y1,z1);
9    glVertex3f(x0,y0,z0);
10   glVertex3f(x3,y3,z3);
11 glEnd();
```

Si se utilizan estas primitivas para dibujar una superficie continua los vértices compartidos por varias primitivas será necesario repetirlos, lo que implica transferirlos a la GPU y procesarlos varias veces. Para reducir estas situaciones se usan primitivas compuestas, que agrupan líneas, triángulos o cuadriláteros (Figura 2.6):

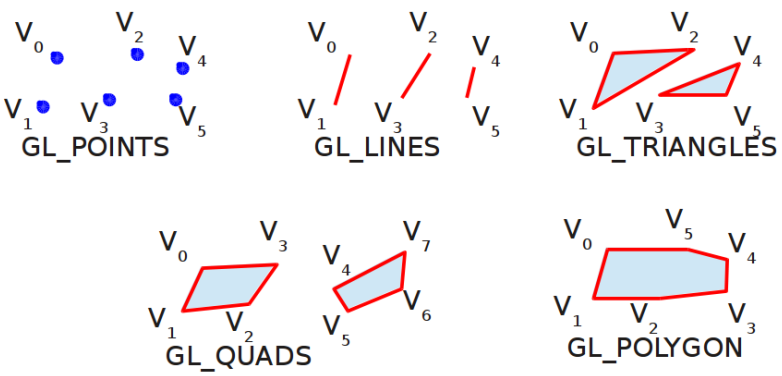

Figura 2.5: Resultado de pasar los mismos 6 vértices como diferentes primitivas.

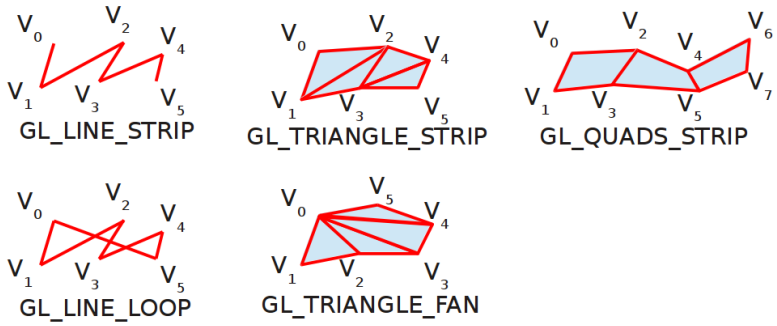

Figura 2.6: Resultado de pasar los mismos 6 vértices como diferentes primitivas complejas.

GL_LINE_STRIP Dibuja una poligonal.
GL_LINE_LOOP Dibuja un polígono.
GL_TRIANGLE_STRIP Dibuja una cinta de triángulos uniendo cada vértice con los dos anteriores.
GL_TRIANGLE_FAN Dibuja un abanico de triángulos. El primer vértice es común a todos los triángulos.
GL_QUADS_STRIP Dibuja una cinta de cuadriláteros.

2.2.1. Caras traseras

Cada polígono tiene dos caras (si te lo imaginas recortado en una cartulina serían las dos caras de la cartulina). Si el polígono forma parte de la superficie de un objeto solo se verá una de las caras de los polígonos [1]. Para reducir el número de elementos a dibujar, y por tanto el tiempo de dibujo, las caras traseras no suelen dibujarse. La identificación de las dos caras de los polígonos se realiza comprobando el orden de recorrido de los vértices en la proyección del polígono en pantalla. Por este motivo los vértices deben darse siempre en sentido antihorario (**CCW** Counter Clockwise) mirandolo desde la cara visible [2].

1: Salvo que la cámara entre dentro del objeto

2: Este convenio puede cambiarse.

2.3. Cálculo de normales de cara

El color con el que se debe visualizar cada polígono depende del color de su color y de como le incide la luz. Para calcular la iluminación es necesario conocer las normales, que son vectores de módulo uno que indican la dirección perpendicular de la superficie (veremos la relación entre ambas mas adelante). Por este motivo se debe especificar la normal de cada primitiva que se dibuja. Las normales se indican con la función **glvertex3f**.

En el ejemplo 2.2 se se asigna normal $(1.0, 0.0, 0.0)$ al primer triángulo y $(0.0, 0.0, 1.0)$ al segundo.

El cálculo de las normales de un triángulo con vértices $\mathbf{v}_0$, $\mathbf{v}_1$ y $\mathbf{v}_2$ se realiza calculando el producto vectorial de dos de sus aristas, por ejemplo $\mathbf{a} = \mathbf{v}_1 - \mathbf{v}_0$ y $\mathbf{b} = \mathbf{v}_2 - \mathbf{v}_0$

$$\mathbf{n} = (\mathbf{b} \times \mathbf{a}) / \| \mathbf{n} \|$$

El vector $\mathbf{n}$ apunta según la regla de la mano derecha (Figura 2.7).

Listing 2.2: Dibujo con normales

```
1   glBegin(GL_TRIANGLES);
2   // Primer triangulo
3   glNormal3f(1.0,0.0,0.0);
4   glVertex3f(x0,y0,z0);
5   glVertex3f(x1,y1,z1);
6   glVertex3f(x2,y2,z2);
7
8   // Segundo triangulo
9   glNormal3f(0.0,0.0,1.0);
10  glVertex3f(x1,y1,z1);
11  glVertex3f(x0,y0,z0);
12  glVertex3f(x3,y3,z3);
13  glEnd();
14
```

Para que la orientación de las normales sea consistente la orientación de todos los triángulos debe ser la misma.

2.3.1. Normales de vértice

Como veremos mas adelante, para visualizar superficies suaves se calcula la iluminación en los vértices y se interpola el color en el interior de los polígonos (ver figura 2.8). Para ello se debe especificar la normal en cada vértice[3]. Si la ecuación de la superficie que se está modelado es conocida se puede calcular la normal analíticamente.

En caso contrario se puede calcular la normal en los vértices promediando las normales de las caras (ver Figura 2.9):

$$\mathbf{n} = \frac{\sum_{i=1}^{k} \mathbf{n}_i}{\| \sum_{i=1}^{k} \mathbf{n}_i \|}$$

siempre que

$$\| \sum_{i=1}^{k} \mathbf{n}_i \| \neq 0$$

siendo n_i la normal en la cara i de las k que comparten el vértice.

Cuando se utilizan normales de vértice se debe llamar a **glNormal3f** antes de cada **glVertex3f** para indicar la normal de vértice.

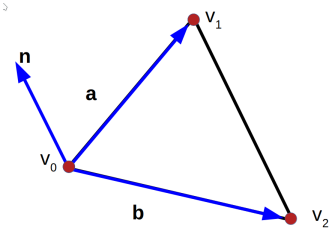

Figura 2.7: Cálculo de la normal de un triángulo

3: Además se le debe indicar a OpenGL usando la llamada a la función de OpenGL glShadeModel(GL_SMOOTH), para volver a dibujar caras planas se usa glShadeModel(GL_FLAT).

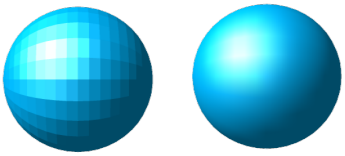

Figura 2.8: Poliedro visualizado con normales de cara (izquierda) y de vértice (derecha)

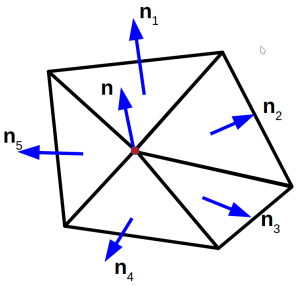

Figura 2.9: Cálculo de la normal de vértice

2.4. Sopa de triángulos

Podemos representar los objetos que aparecen en la escena como una lista de triángulos, en la que de cada triángulo se almacenan sus tres vértices y cada vértice tiene tres coordenadas (Figura 2.10 y listado 2.3).

A esta representación se le suele llamar **sopa de triángulos** ya que es un **conjunto** de triángulos almacenados de forma independiente. Los triángulos pueden ser adyacentes, en este caso los vértices aparecen duplicados.

Para visualizar el modelo se recorre la lista dibujando cada triángulo Tal como se ha comentado en la sección 2.2.1 los vértices deben estar ordenados de forma consistente en sentido antihorario (CCW).

Se puede calcular la normal de cada triángulo al dibujarlo o tenerla almacenada en la estructura (en este caso los triángulos tendrían tres valores **float** adicionales; **nx, ny, nz**. Suponiendo que está almacenada, el código de dibujo sería el siguiente:

```
glBegin(GL_TRIANGLES);
  for(i=0;i<Mesh.n;i++){
    glNormal3f(Mesh.t[i].nx, Mesh.t[i].ny, Mesh.t[i].nz);
    glVertex3f(Mesh.t[i].a.x,Mesh.t[i].a.y,Mesh.t[i].a.z);
    glVertex3f(Mesh.t[i].b.x,Mesh.t[i].b.y,Mesh.t[i].b.z);
    glVertex3f(Mesh.t[i].c.x,Mesh.t[i].c.y,Mesh.t[i].c.z);
  }
glEnd();
```

	Vertice 1			Vertice 2			Vertice 3		
	X	Y	Z	X	Y	Z	X	Y	Z
0	-6,17	-4,31	5,84	-2,61	2,84	7,15	-5,49	0,09	-2,98
1	-8,54	4,18	1,17	-3,61	7,05	-7,78	5,42	8,87	0,34
2	6,17	-6,29	-3,09	-1,83	-0,22	3,77	8,75	9,54	5,00
3	-8,63	-5,76	-9,86	9,10	-6,73	-1,92	8,77	9,34	-3,45
4	6,56	-6,84	-1,76	4,51	8,90	2,48	5,68	4,97	0,67
5	-0,22	-4,65	-3,58	-1,86	-6,11	-5,15	-3,72	2,29	-5,41
6	2,41	-1,14	6,55	0,73	-0,94	8,15	3,08	-1,24	2,03
7	-8,54	5,67	-6,90	4,66	2,93	6,12	9,39	1,78	-3,97
8	-7,03	4,12	7,35	9,48	9,14	-7,92	3,75	-3,00	-9,18
9	-6,53	-8,56	1,82	-5,90	-9,34	5,09	5,98	-1,60	5,36
10	-0,65	-2,55	7,15	0,64	-1,28	8,80	6,99	5,47	2,53
11	-8,44	7,93	5,24	-5,82	9,34	8,07	-9,94	-8,73	-8,24
12	7,05	-2,88	6,63	6,50	-8,76	-9,25	-2,06	9,62	1,20
13	-6,95	4,77	-2,41	3,78	4,58	9,82	0,04	9,43	-0,72
14	-1,97	-3,69	5,53	8,75	9,22	6,58	-6,88	-0,55	4,31

Figura 2.10: Representación de una sopa de triángulos.

Está representación tiene varios inconvenientes:

Tiene información redundante. Las coordenadas de los vértices se repiten para cada triángulo que contiene el vértice [4].

Pueden aparecer fisuras por errores de representación. Las diversas copias de las coordenadas de cada vértice, que están almacenadas como float, pueden no coincidir.

Hacer operaciones es complejo. La mayor parte de las operaciones que se pueden realizar con la superficie requieren poder buscar elementos vecinos. Con esta representación es muy costoso encontrar los triángulos adyacentes a uno dado.

Para resolver los dos primeros problemas evitando la redundancia de vértices se pueden indexar los vértices en lugar de copiar en cada triángulo[5]. De esta forma la representación constará de una lista de vértices y una lista de triángulos, en la que cada triángulo contiene los índices de sus vértices. De esta forma cada vértice se almacena una sola vez y cada triángulo apunta a sus tres vértices. La representación en C usando arrays se muestra en 2.4. La definición de la representación en C++ puede ser la siguiente:

```
typedef struct {
  float x,y,z;
  } Point;  //Coordenadas del vertice

typedef struct  {
  int v0,v1,v2;
  } Index;  //Indices de los tres vertices de un triangulo

class Malla : Objeto3D {
  protected:
    std::vector < Point > vert;   // Lista de vertices
    std::vector < Index > tri;    // lista de triangulos
    std::vector < Point > normal; // normales de los triangulos

  int nV, nT;   // numeros de vertices y triangulos

  public:
    Malla();
    ...
    };
```

Listing 2.3: Estructura de datos de una sopa de triángulos

```
typedef struct {
  float x, y, z;
  } vertex;

typedef struct {
  vertex a, b, c;
  } triangle;

typedef struct {
  int n;
  triangle* t;
  } mesh;
```

4: En una malla regular cada vértice está en un promedio de 6 triángulos

5: En la lección siguiente veremos como resolver el problema de la búsqueda de triángulos vecinos.

Listing 2.4: Estructura indexada para una sopa de triángulos

```
typedef struct {
  float x, y, z;
  } vertex;

typedef struct{
  int nv,nt;
  float vertex[MaxVertex][3];
  int triangle[MaxTriangle
    ][3];
  } Mesh;
```

2.5. Formatos de archivo

Para almacenar y transferir mallas de triángulos se suelen usar diversos formatos de archivo. Entre los más comunes están:

PLY Es un formato abierto, sencillo y configurable, diseñado por la Universidad de Stanford.

OBJ Es un formato simple que almacena información básica de vértices y caras. Es popular debido a su simplicidad y facilidad de uso en diferentes plataformas y herramientas.

STL Ampliamente utilizado en la impresión 3D. STL puede almacenar mallas de triángulos utilizando sólo la información de las coordenadas de los vértices de cada triángulo.

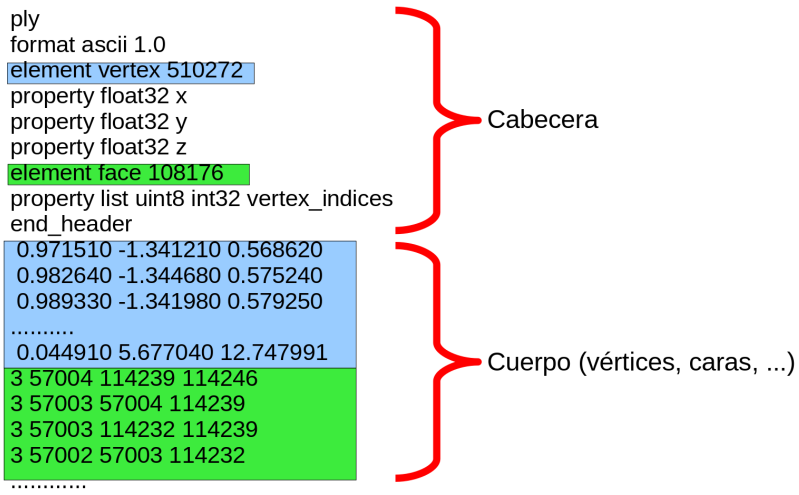

Figura 2.11: Estructura de un archivo PLY.

FBX Es un formato más complejo que puede almacenar no solo mallas, sino también animaciones, texturas, y otros datos relevantes para proyectos de gráficos avanzados y videojuegos.

Los archivos PLY se pueden almacenar en binario o en Ascii, que ocupan mas espacio pero permiten la edición con un editor de textos. Los archivos ply almacenan superficies poligonales indexadas, están organizados en dos áreas (2.11), en la cabecera se identifica el tipo de archivo (en la figura ply almacenado en ascii) y se detalla la información contenida en el archivo. En la figura el modelo tiene 510272 vértices (almacenados como float32 con coordenadas x,y,z) y 108176 caras (cada cara tiene el número de vértices y los identificadores de los vértices). El cuerpo del archivo tiene la lista de vértices (la posición del vértice en la lista es su identificador) y la lista de caras.

Para leer un archivo PLY se debe leer el número de los elementos de la cabecera y a continuación leer y cargar en la estructura de datos cada uno de los vértices. Seguidamente se lee cada cara y se almacena en la lista de caras. En este proceso se pueden detectar errores por duplicación de elementos comprobando que el elemento que se acaba de leer no está ya en la estructura de datos y se pueden calcular y almacenar las normales.

2.6. Herranmientas: MeshLab

MeshLab es un software de código abierto utilizado para el procesamiento y edición de mallas 3D[6] diseñado por el Visual Computing Lab del ISTI-CNR [Cig+08]. Es especialmente útil para la limpieza de modelos 3D, la reconstrucción de superficies a partir de puntos 3D y la visualización de mallas.

6: Se puede descargar de `https://www.meshlab.net/#download`

En este curso es especialmente útil para:

- Visualizar modelos, cambiando el modo de visualización.
- Editar modelos: eliminar triángulos o vértices, remallado del modelo.
- Corregir errores: eliminar elementos duplicados, corregir topología, tapar fisuras.

- Generar modelos simples. Puede crear algunos objetos predefinidos (esferas, poliedros simples, terrenos) y calcular operaciones booleanas entre modelos.
- Convertir el formato de los modelos. Nos permite leer archivos en diferentes formatos y exportarlos en un formato diferente.
  Importa: PLY, STL, OFF, OBJ, 3DS, COLLADA, PTX, V3D, PTS, APTS, XYZ, GTS, TRI, ASC, X3D, X3DV, VRML, ALN.
  Exporta: PLY, STL, OFF, OBJ, 3DS, COLLADA, VRML, DXF, GTS, IDTF, X3D

Para cargar un modelo en MeshLab basta con arrastrar el archivo sobre el icono de meshlab o usar la opción

```
File > Import Mesh
```

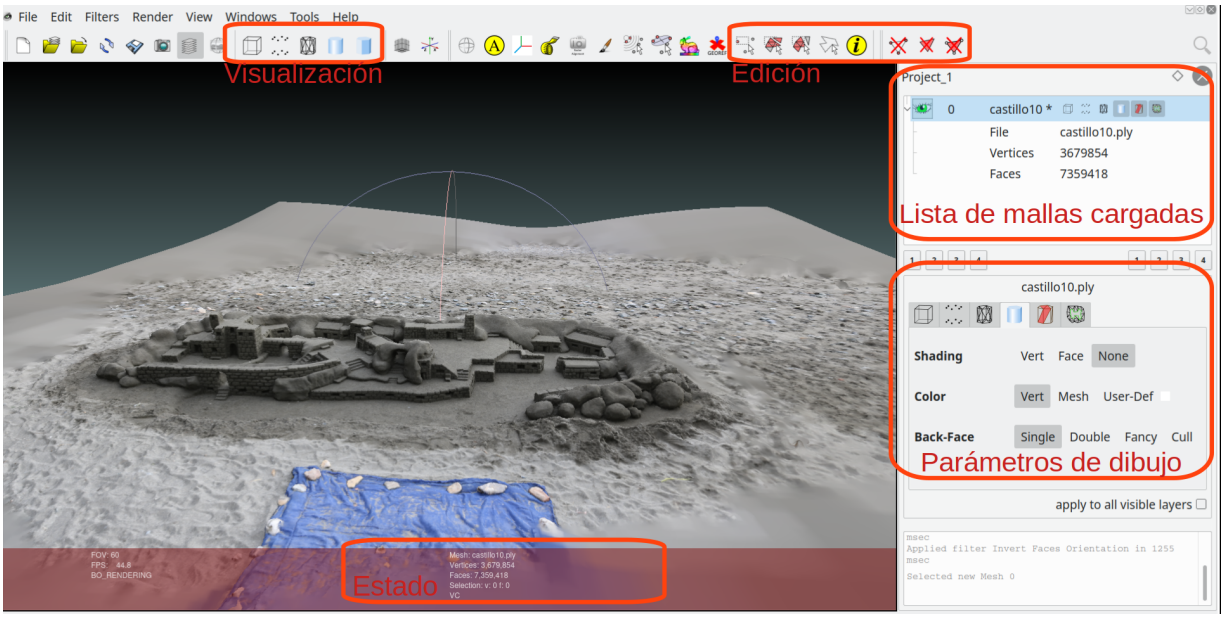

Figura 2.12: Interfaz de usuario de MeshLab.

La Figura 2.12 muestra la interfaz de MeshLab. En ella destacamos los iconos de control de la **visualización**, que controlan que se visualiza (caja envolvente [7], puntos, aristas o polígonos); La zona de edición, que permite seleccionar y borrar vértices o polígonos; A la derecha el panel de proyecto, en el que se muestra la lista de mallas cargadas, permitiendo seleccionarlas y ocultarlas; Y debajo el panel de parámetros de dibujo de la malla seleccionada, con el que podemos controlar como se visualiza cada tipo de elemento de la malla seleccionada.

En todo momento hay una malla seleccionada, que se destaca en el panel del proyecto con un sombreado celeste (ver Figura 2.12), todas las operaciones se realizan con la malla seleccionada, independientemente de que sea o no visible.

Los parámetros de las mallas se pueden consultar en la barra de estado o en el propio panel del proyecto.

La mayor parte de las operaciones se encuentran en el menú **Filters**. Hay encontraremos las operaciones de creación de mallas **Create New Mesh Layers**, que nos permiten crear mallas simples predefinidas.

7: La caja envolvente es el mínimo paralelepípedo alineado con los ejes que contiene todos los vértices.

Para cambiar el formato de un modelo deberemos abrirlo con MeshLab y usar la orden

```
File > Export Mesh As
```

En el panel que aparece deberemos indicar el nombre del archivo y el formato. Por último aparecerá un panel de selección de información a incluir en el archivo.

2.7. Ejercicios

1. Calcular el espacio ocupado por una sopa de triángulos en función del número de vértices para un sistema de 64 bits.
2. obtener cara vecinas de una dada
3. Repetir el cálculo para una estructura indexada.
4. Generar el archivo PLY ascii de un tetraedro con MeshLab.
5. Crear un archivo PLY para representar una pirámide de base cuadrada con un editor de textos. Compruebalo abriendolo con MeshLab.
6. Busca 3 modelos diferentes en formato PLY y comprueba la relación entre número de vértices y de triángulos.
7. Escribe el código OpenGL necesario para dibujar una pirámide de base cuadrada.
8. Diseña un algoritmo para convertir una sopa de triángulos en una estructura indexada.
9. Calcula la complejidad del algoritmo del ejercicio anterior.
10. ¿Que procesamiento habría que hacer al leer una sopa de triángulos para evitar las fisuras por errores de representación?
11. ¿Pueden aparecer estos errores en una malla indexada?
12. Diseña un algoritmo para determinar si un polígono 2D es simple.
13. Diseña un algoritmo para determinar si un polígono 2D es cóncavo o convexo.
14. Diseña un algoritmo para determinar si un polígono definido en el espacio es plano.