UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Importar mallas 3D en tiempo de compilación para una aplicación de videojuego en la 3DS

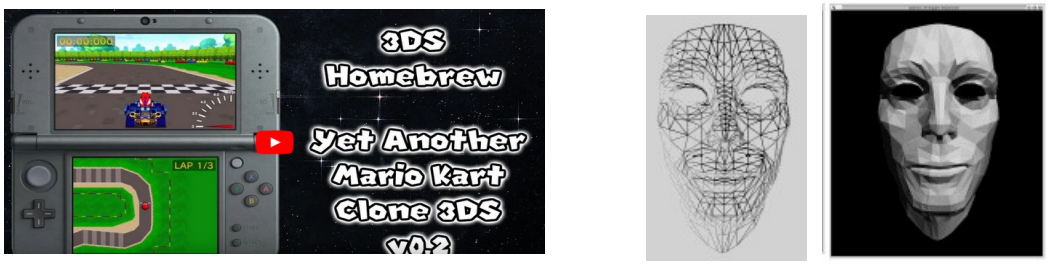
Apellidos, nombre	**Agustí i Melchor, Manuel** (magusti@disca.upv.es)
Departamento	**Departamento de Informática de Sistemas y Computadores (DISCA)**
Centro	Universitat Politècnica de València

1 Resumen de las ideas clave

A la hora de realizar una aplicación, como un videojuego, que puede necesitar mostrar objetos en 3D (como personajes o elementos del mundo), es habitual representarlos a partir de la **malla** de cada objeto, cargarla, posicionarla en la escena y moverla como consecuencia de la interacción con el usuario y con los objetos presentes en la misma. La malla es el conjunto de puntos, en un espacio de coordenadas y sus relaciones espaciales. que definen la superficie externa de un objeto. Los objetos (sus mallas) se pueden crear algorítmicamente, si se conoce una función que los genere (desde un tablero de ajedrez a un paisaje) o a partir del trabajo de una persona que lo "modela", esto es, que lo dibuja en 3D, habitualmente con aplicaciones como Blender[1]. Posteriormente, se pueden **cargar** esos modelos a partir de los ficheros que los contienen. En este punto, caben básicamente dos alternativas: cargar **estáticamente o dinámicamente** los modelos desde fichero**.**

Por un lado, se habla de carga **estática** de objetos 3D al incorporar la malla que los define **en tiempo de compilación** al proyecto; lo que obliga al equipo de desarrollo a disponer de los objetos antes de ejecutar el código, pudiendo realizar optimizaciones, corrección de errores o simplificaciones que aumenten la eficiencia de la aplicación. Así se evita tener que incluir en el código las operaciones necesarias para resolver problemas que puedan aparecer durante la ejecución del programa por errores u omisiones en los ficheros que contienen las mallas. De esta manera, se mantiene la independencia respecto al modelo, pudiendo incorporar rápidamente otro modelo diferente sin más que recompilar y, en un punto de la implementación en que es posible pararse a resolver problemas que pudiera no ser posible dar solución en tiempo de ejecución para la carga de una malla 3D con alguna deficiencia. Para conseguir esto es necesario disponer de un conversor que permita incorporar el modelo al proyecto que se está realizando.

Por su parte, la carga **dinámica** o **mediante código,** es la que en tiempo de ejecución importa una malla desde fichero. Esto implica disponer de librerías que conozcan el detalle de los formatos que se pretende utilizar y permite cambiar los modelos hasta el último momento, el previo a lanzar el ejecutable. Citar como contrapartida que complica la distribución del programa al requerir llevar detrás los ficheros necesarios y las librerías de que dependa; también que hace crecer el tamaño de la aplicación para realizar las validaciones e importar los datos necesarios. Además, puede que no interese que estos ficheros estén accesibles al usuario final, para que no los pueda editar.



a) *b)*

Figura 1: Ejemplos de aplicaciones que utilizan mallas 3D: (a) YAMKC [1] en 3DS, (b) Geoface [2] en OpenGL sobre el computador.

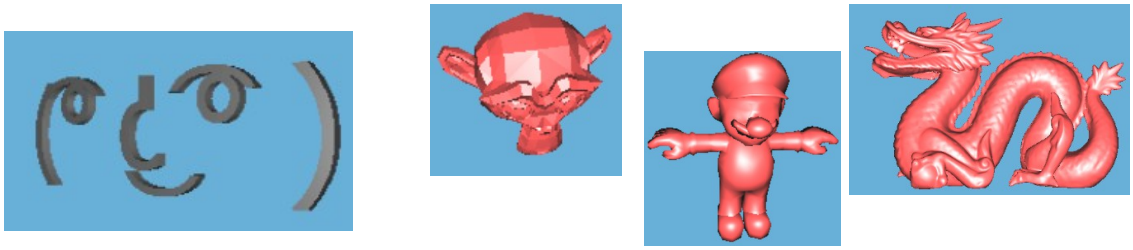
En este artículo vamos a proponer una aplicación, a utilizar en línea de órdenes, para realizar la conversión de ficheros en formato OBJ e incorporarlos a un proyecto de código C para el desarrollo

[1] Véase el sitio web de Blender en <https://www.blender.org/>.

de un videojuego. Ejemplos de uso de estas ideas se pueden ver en la Figura 1, donde podemos ver un momento de la ejecución de YAMKC [1] y uno de los ejemplos clásicos de OpenGL: geoface[2].

2 Objetivos

Como hilo conductor de este artículo, se va a trabajar sobre la idea de cómo un visualizador de mallas 3D (Figura 2), realizado a partir de un ejemplo de código para la 3DS [3], tiene como necesidad obvia poder disponer de mallas que mostrar. El código del conversor de mallas 3D a código C será publicado en GitHub [4] para completar el uso de esta propuesta.



a) *b)*

Figura 2: Ejemplos de salida en pantalla de un visualizador de mallas 3D: (a) original [3] y (b) utilizando otros posibles modelos.

Una vez que el lector haya revisado este artículo con detenimiento y explore el código que se adjunta, dispondrá de una referencia para aplicar en el caso de necesitar importar una malla de un objeto 3D en una aplicación propia. En particular, será capaz de:

- Describir qué es una malla 3D.
- Explorar el código que se referencia para localizar cómo hace la conversión de un formato de fichero 3D a un conjunto de instrucciones en lenguaje C. Estas definen y asignan valores a una estructura de datos que representa una malla 3D.
- Compilar y ejecutar la aplicación mostrada para incorporar un objeto 3D, en formato OBJ, en su propio videojuego, sin modificar manualmente el código.

3 Introducción

Partiremos de un ejemplo de desarrollo (*lenny [3]*) ya existente en la 3DS, que muestra, siempre, el mismo objeto 3D sobre un fondo liso (véase la Figura 2a) en pantalla, a partir de una estructura de datos que le proporciona la información de los puntos que forman el objeto y de sus relaciones espaciales en un espacio tridimensional: la malla 3D. Sobre esa base, se va a realizar un conversor de un formato de fichero de imagen 3D para poder cambiar la malla 3D del objeto que se muestra en esta aplicación.

Este tipo de acciones son habituales en aplicaciones en que se conoce de partida el número y los contenidos exactos de los objetos que se desean utilizar a lo largo de la ejecución de una aplicación.

[2] Ejemplo aparecido en el libro de F. Parke y K. Waters “Computer facial animation” y que se puede encontrar en “OpenGL demos” <https://www.opengl.org/archives/resources/code/samples/glut_examples/demos/demos.html>.

En el caso de los videojuegos, es habitual incrustar (importar antes de la ejecución de la aplicación) los sonidos, las imágenes y los objetos 3D para gestionarlo posteriormente. Así, en el caso de desarrollo para una consola, como la 3DS, todos los ficheros de media se suelen convertir a estructuras de datos inicializadas con los valores de cada clip de sonido, textura de imagen y/o malla de un objeto 3D. De esta forma se libera de la necesidad de incorporar la operativa necesaria para leer cada tipo de fichero y la sobrecarga de los encabezados de cada uno de los ficheros indicados. Con las diferencias propias de cada lenguaje de programación, el contenido de todos estos ficheros van a parar a variables dentro del código, declaradas e inicializadas de una manera más o menos manual. Con el tiempo, la aparición de nuevos formatos, el alto número que se utilizan en aplicaciones de mercado y el uso de compresión de los mismos ha ido propiciando la aparición de herramientas que convierten el audio o la imagen[3] en esas variables de forma automática.

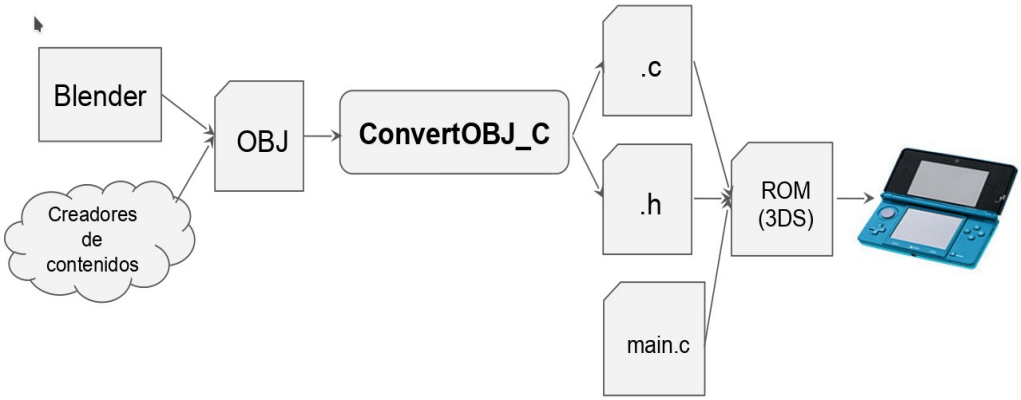


Figura 3: Idea gráfica del proceso de importación de modelos 3D en tiempo de compilación.

Si nos centramos en los objetos 3D, la complejidad de los mismos motiva que sea tanto o más interesante automatizar esa tarea. Para realizar esa tarea se aborda aquí el desarrollo de una aplicación **convertOBJ_C** que, como muestra la Figura 3, será la encargada de recodificar un fichero (en formato OBJ) en un fichero de código C y uno de cabeceras (con extensión .h). De esta forma, se generarán un par de ficheros de código que es posible incluir en el conjunto de ficheros a compilar para obtener una ROM (un ejecutable). Como resultado de la compilación se tendrá importada la malla que se describe en el fichero OBJ (realizada con *Blender* u obtenida desde algún portal en Internet de descarga o de venta de ficheros gráficos) y la aplicación hará el uso que le corresponda a esa malla en una consola real como la 3DS.

En resumen, la aplicación de conversión de un fichero gráfico 3D en formato OBJ a un fichero de código C es, como ya se ha comentado, para implementar un visor (pero podría ser otra aplicación). La aplicación original tiene fijado, en el código, la ruta de la malla a mostrar, como se puede ver en la Figura 2a. Llegados a este punto, es necesario profundizar en el concepto de malla (también llamada malla poligonal o malla 3D) y cómo refleja la forma de representar un objeto gráfico 2D/3D en un computador.

3.1 Mallas 2D y 3D

Ya se ha dicho que una malla es tanto las posiciones de un conjunto de puntos, como las conexiones entre ellos. Esta información es la que se guarda en los ficheros gráficos 2D/3D y es el primer paso

[3] Ejemplos de estas utilidades son, respectivamente: *cwavtool* <https://github.com/PabloMK7/cwavtool> y *magick* (anteriormente *convert*) <https://imagemagick.org/script/convert.php>.

en la representación volumétrica de un objeto en nuestros computadores, como se puede ver en los ejemplos de la Figura 4 realzados con efectos de iluminación y sombreado. Es lo que se conoce, en el campo de los gráficos por computador, como modelo de alambre (*wireframe*).

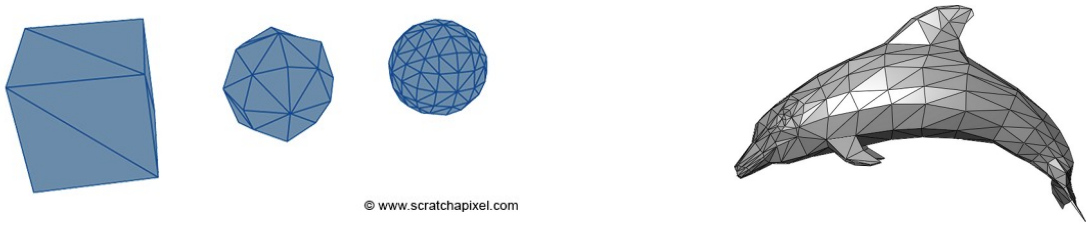
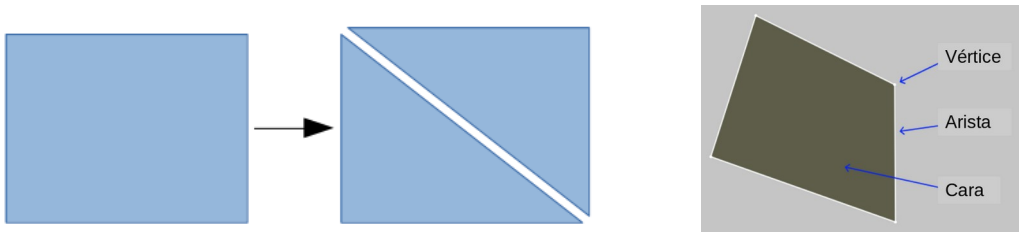


Figura 4: Mallas de modelos 3D complejos [6].

De entre todos los polígonos, el triángulo define el mínimo conjunto de todos los puntos que están el mismo plano y, por lo tanto, los que tendrán el mismo color. Con ellos se pueden construir el resto de polígonos: la Figura 5a muestra un rectángulo construido a partir de dos triángulos. Esto quiere decir que el dibujado de objetos 2D y 3D se basa en mallas[4] formadas por triángulos (definidos en términos de vértices, aristas y caras, véase la la Figura 5b) y que conforman la impresión de un objeto sólido. Cada uno de los triángulos, es una cara (face) donde se aplica el mismo valor de iluminación (color) y está limitada por los vértices (*vertex*) espacialmente más cercanos y sus conexiones o aristas (*edge*).

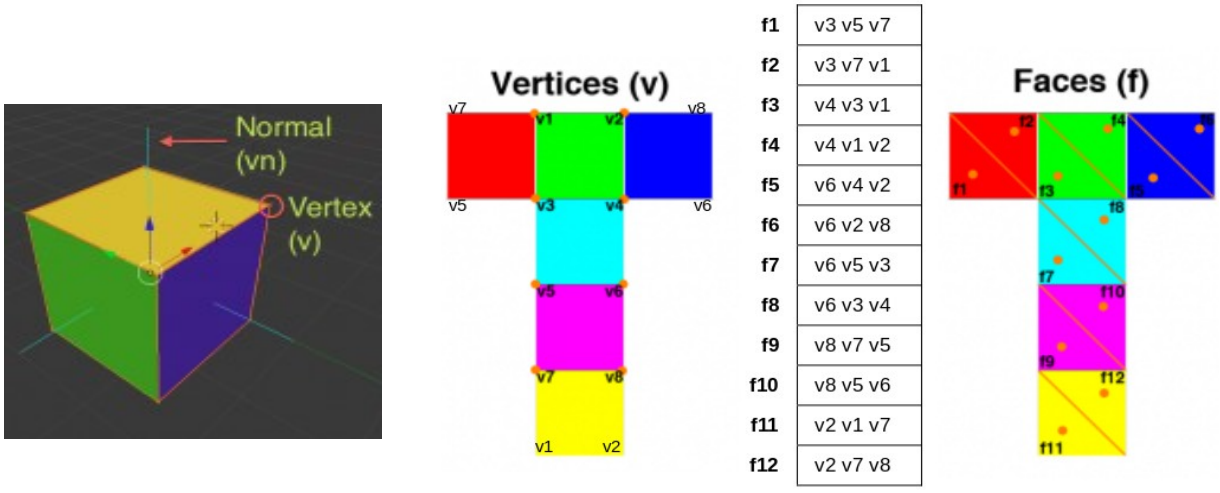


a) *b)*

Figura 5: Dibujo de un rectángulo: (a) su malla asociada con cuadriláteros y triángulos y (b) elementos básicos que la componen. Imagen modificada de [5].

En el caso de la aplicación original del visor, Figura 2a, se muestra un logotipo con 3345 caras (triángulos), lo que le confiere un grado de detalle alto: lo que se aprecia en la curvatura de la superficie y el número de detalles de brillos y sombras. La especificación del modelo ocupa cerca de 3350 líneas de código, frente a menos de doscientas líneas de instrucciones que implementan el visor.

[4] También conocidas como mallas poligonales, mallas 3D o, simplemente, *mesh*.



a) *b)*

Figura 6: Cubo 3D: (a) vista 3D y (b) malla formada por vértices y caras (triángulos) [7].

En un caso más simple, un cubo, que tiene 6 lados, está formado por una malla con 12 caras (triángulos). ¡Veámoslo!. La Figura 6a muestra el cubo en la típica vista en 3D, en la que podemos ver tres lados (cuadriláteros) del cubo y sabemos que hay otros tres, pero están ocultos: así que tenemos los seis lados en total. Como algunos comparten vértices, solo necesitamos ocho vértices para posicionar todos los lados. Si desplegamos los lados del cubo, como muestra la Figura 6b, se observa que algunos vértices aparecen dos veces etiquetados sobre la imagen de "Vertices (v)". Cada lado del cubo se dibuja como dos triángulos o caras en "Faces (f)", que están enumeradas en la tabla de la Figura 6b: cada cara (triángulo) como índice (con valores de f1 a f12) y los tres vértices que lo definen al lado.

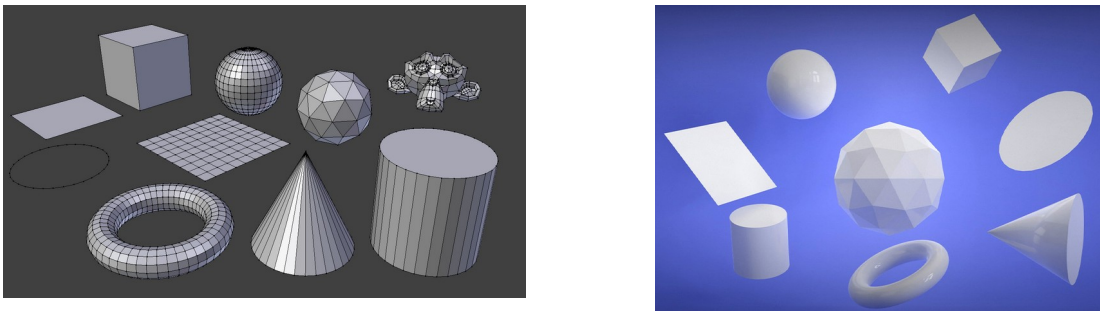


Figura 7: Diferentes objetos 3D construidos con mallas poligonales y con efectos de iluminación: (a) Blender <https://docs.blender.org/manual/en/latest/modeling/meshes/primitives.html> y (b) 3D Studio <https://3dstudio.co/es/polygon-mesh/>.

Ya casi está vista toda la información para pintar una malla, solo un comentario sobre las "normales" o "Normal (vn)" de la Figura 6a. Este término se utiliza para indicar una dirección perpendicular a algo. En nuestro caso sirve para indicar cuál es la parte exterior de cada triángulo y hacer cálculos entre la dirección del iluminante y la de cada triángulo para determinar el color y brillo de los puntos de cada polígono. La Figura 7 muestra algunos ejemplos donde podemos ver cómo las caras (triángulos o cuadriláteros) de las mallas muestran estos efectos.

4 Desarrollo

¡Ya lo tenemos! Ahora solo es cuestión de ver cómo pasar la información de la malla (que estará contenida en un fichero gráfico) a la estructura de datos que utilice la aplicación que vayamos a "alimentar" con los modelos 3D. Hay que concretar cómo espera los datos el hardware, cómo los recoge el formato de fichero y cómo es la aplicación… Y los tres tienen muchas variantes, así que vamos a contextualizar con los objetivos marcados.

4.1 Estructura 3D en memoria del hardware gráfico

Existen ya aplicaciones que hacen este tipo de conversiones, como *obj2opengl*[5] o mtl2opengl[6], y lo hacen pensando en un interfaz de OpenGL y una tarjeta gráfica genérica. En nuestro caso nos hemos propuesto desarrollar una herramienta que lleve a cabo esa conversión a las estructuras de datos de una aplicación propia que se ejecutará sobre el hardware gráfico de la 3DS (una GPU PICA200) que, en su forma más simple (sin definir el color del modelo), es una lista de vértices que se guarda contigua en memoria, por ejemplo, en una estructura de tipo vector.

Malla = secuencia o conjunto de posiciones

Punto #1 Triang. #1	Punto #2 Triang. #1	Punto #3 Triang. #1	…	Punto #1 Triang. #T	Punto #2 Triang. #T	Punto #3 Triang. #T

Cada posición de la malla

v_X	v_Y	v_Z	vn_X	vn_Y	vn_Z

Figura 8: La malla como secuencia de puntos o posiciones espacialmente conectadas.

Esta estructura se muestra esquemáticamente en la Figura 8 y está compuesta por:

- Tantos elementos como posiciones o puntos tiene la malla, que son tantos como número de triángulos (*face* o caras) multiplicado por tres, puesto que cada tres posiciones consecutivas del vector definen los tres vértices de un triángulo en la malla.
- En cada elemento del vector está la información de un punto (o posición) del modelo codificado en seis valores, tres para las coordenadas del vértice y tres para la normal del mismo. Las coordenadas de vértices ('v') se repetirán en alguna otra posición, puesto que los vértices se comparten entre caras vecinas, pero la parte de orientación (vn) de la posición será diferente si no están en el mismo plano.

En el caso de un cubo tendríamos una malla de doce caras (utilizando triángulos, o seis si se compusieran con cuadriláteros), por lo tanto 36 (12*3) posiciones (o puntos), de seis valores (de tipo real) cada elemento de la malla. Veamos ahora cómo lo extraemos de un fichero gráfico.

[5] Véase este código para convertir modelos 3D de ficheros OBJ a vectores en C/C++en formato compatible con la función *glDrawArrays* de OpenGL ES en <https://github.com/Hbehrens/obj2opengl>.

[6] Otro conversor de ficheros OBJ y MTL en vectores compatibles con el API de OpenGL ES en dispositivos iOS en <https://github.com/ricardo-rendoncepeda/mtl2opengl>.

4.2 Contenido de la malla de un fichero gráfico en formato OBJ

OBJ es el formato aquí escogido por su larga tradición en este campo (desde su invención por *Wavefront Technologies* [8] (pasando por *Silicon Graphics*, *Alias* y *Autodesk*) por su naturaleza abierta (puede leerse con un sencillo editor de texto ASCII).

Está compuesto por una serie de bloques, como muestra la Figura 9. Cada línea empieza con la letra que identifica los valores de esa línea: vértices (v), coordenadas de textura (vt), normales (vn) y caras (faces) definidas por tres puntos (triángulos); para cada punto se especifica v/vt/vn. Las coordenadas de textura, que tienen dos parámetros (U y V) permiten asociar a un punto el color de un "material". Este es [8] otro fichero (generalmente con el mismo nombre que el OBJ, pero con extensión MTL), pero pueden ser más de uno. Los ficheros MTL contienen los parámetros de la iluminación ambiente y uno o más ficheros de imágenes 2D que modelan, como un mapa, esa iluminación.

```
# List of vertices (v) with XYZ coordinates.
v  1.0x   1.0y   1.0z
v  ...X   ...Y   ...Z

# List of texture coordinates (vt) with UV coordinates.
vt 0.5u   0.5v
vt ...U   ...V

# List of normals (vn) with XYZ coordinates.
vn 0.0x   1.0y   0.0z
vn ...X   ...Y   ...Z

# List of faces (f) with v, vt, and vn data (three points ABC per face).
f  1v/1vt/1vn   2v/2vt/1vn   3v/3vt/1vn
f  v/vt/vnA     v/vt/vnB     v/vt/vnC
```

Figura 9: Estructura interna de un fichero OBJ sin texturas [7].

Así que se habrán de extraer las líneas 'f' y para cada una de ellas, los valores de vértice (v) y normal (vn), dejando fuera los valores de textura (vt) que se guardan con cada punto del triángulo.

4.3 Estructura de la aplicación

La tarea a realizar es, básicamente, una lectura de un fichero de texto, del que extraer la información deseada y generar dos nuevos ficheros con esa información extraída en el formato que se necesita. Básicamente la aplicación recodifica la información contenida en el fichero OBJ, para generar las estructuras de datos en lenguaje C en un fichero y las cabeceras para utilizarlas en otro fichero. Así que se va recorrer el fichero OBJ, guardando los vértices (v) y las normales (vn), hasta encontrar las definiciones de las caras (f) definidas como triángulos, que harán referencia a los vértices y las normales leídos.

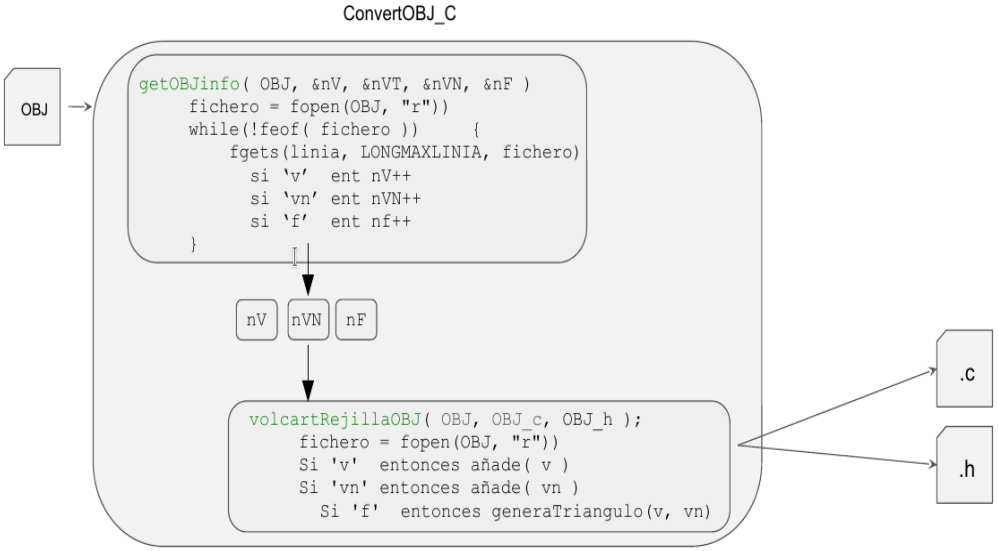


Figura 10: Estructura de la aplicación convertOBJ_C.

Para ello, la Figura 10 muestra el esquema ampliado de la Figura 3, en el que vemos con mayor detalle el algoritmo que implementa la aplicación:

- Primero, la función *getOBJInfo*, comprueba los valores que contiene el fichero de partida y si hubiera algún posible error. Después, ha de determinar el número de vértices (nV), de normales (nVN) y de caras (nF) y los valores de vértices, normales y de caras ((en las estructuras de datos 'v', 'vn' y 'f' respectivamente) que contiene el modelo 3D. Para ello, recibe la ruta al fichero OBJ, lee el fichero línea a línea y devuelve el número de cada tipo de elemento que forma la malla.
- En segundo lugar, la función *volcarRejillaOBJ*, será la encargada de recodificar la información de los triángulos que componen la malla en el código en lenguaje C. Para ello, la Figura 11 muestra, de forma esquemática, el formato con que se genera la estructura de datos principal que sirve de entrada para el visor de objetos. Observe que se han relacionado las estructuras de datos de vértices (v), normales (vn) y caras (f) que se han explicado en el apartado 3.1 Mallas 2D y 3D. Para ello, recibe la ruta del fichero OBJ, vuelve a leer el fichero línea a línea y devuelve la estructura de datos esperada en forma de código en lenguaje C.

Malla formada por triángulos

v1[f[1]]]_X	v1[f[1]] _Y	v1[f[1]]_Z	vn1[f[1]]_X	vn1[f[1]]_Y	vn1[f[1]]_Z
v2[f[1]]_X	v2[f[1]]_Y	v2[f[1]]_Z	vn2[f[1]]_X	vn2[f[1]]_Y	vn2[f[1]]_Z
v3[f[1]]_X	v3[f[1]]_Y	v3[f[1]]_Z	vn3[f[1]]_X	vn3[f[1]]_Y	vn3[f[1]]_Z
...					
v[f[T-2]]_X	v[f[T-2]]_Y	v[f[T-2]]_Z	vn[f[T-2]]_X	vn[f[T-2]]_Y	vn[f[T-2]]_Z
v[f[T-1]]_X	v[f[T-1]]_Y	v[f[T-1]]_Z	vn[f[T-1]]_X	vn[f[T-1]]_Y	vn[f[T-1]]_Z
v[f[T]]_X	v[f[T]]_Y	v[f[T]]_Z	vn[f[T]]_X	vn[f[T]]_Y	vn[f[T]]_Z

Punto #1
Punto #2
Punto #3
Triángulo, cara o Face #1

...

Punto #1
Punto #2
Punto #3
Triángulo, cara o Face #T

Figura 11: Esquema de la estructura de datos que soportar a la malla 3D y su contenido: cada fila contiene toda la información de un punto o vértice y tres consecutivos forman un triángulo.

4.4 Algunos resultados

Se puede utilizar *Blender* como modelador 3D, generar el modelo y exportarlo como malla triangular o descargar ficheros desde repositorios de OpenGL o similares. Veamos algunos ejemplos, véase la Figura 12, con ficheros que se han recopilado de aplicaciones OpenGL y de catálogos con licencias libres en Internet.

El código, del conversor, lo compilaremos y ejecutaremos con:

```
$ gcc convertOBJ_toC.c -o convertOBJ_toC
```

Y, para cada objecto que deseemos mostrar habrá que recompilar el ejemplo modificado de *lenny*, sustituyendo "fichero" por el nombre del OBJ y recompilando, cada vez, de la forma:

```
$ convertOBJ_toC fichero.obj fichero.c fichero.h
$ make
```

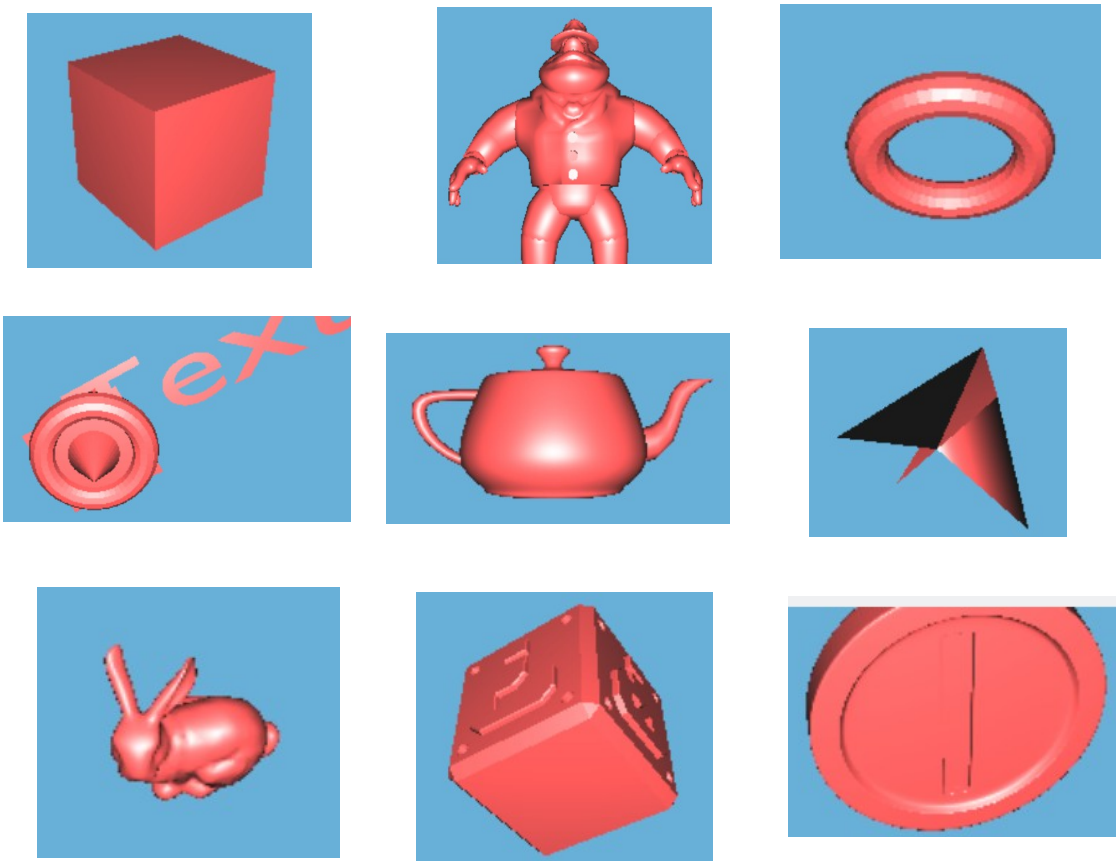


Figura 12: Capturas de la ejecución cambiando el modelo 3D.

5 Conclusión y cierre

A lo largo de este objeto de aprendizaje hemos visto qué trabajo ha de realizar una herramienta para facilitar el importar un modelo 3D a una aplicación propia de manera integrada en nuestro código.

Esta ha sido planteada como una operación de lectura de la información de un tipo de fichero gráfico (OBJ) y su elaboración para generar convertirla en las instrucciones en lenguaje C, que generen y rellenen los campos de una estructura de datos optimizada y creada para visualizar una malla 3D formada por triángulos. El código del conversor de mallas 3D a código C será publicado en GitHub [4] para completar el uso de esta propuesta.

Como resultado de la lectura con detenimiento de este documento y la exploración del código publicado, el lector podrá explorar el uso de mallas para mostrar objetos 3D en pantalla en la plataforma 3DS.

Si te animas, estimado lector, tienes repositorios muy interesantes como "The base mesh"[7]. Anímate a descargar alguno y comprobar que el conversor funciona generando los ficheros en código C que lo representan.

Si te parece que se puede seguir trabajando en esta línea, te animo a ampliar las opciones de conversión para añadir color a estos modelos.

6 Bibliografía y referencias

[1] PabloML7. (2021). Yet Another Mario Kart Clone 3DS (v0.2). <https://www.youtube.com/watch?v=LcKwGymVBEw>.

[2] Agustí Melchor, M. (2017). Ejemplos de aplicaciones 3D interactivas con OpenGL. <http://hdl.handle.net/10251/83395>.

[3] devkitPro / 3ds examples. <https://github.com/devkitPro/3ds-examples/tree/master/graphics/gpu/lenny>.

[4] Repositorio del conversor en GitHub <https://github.com/magusti/3DS/cargarMalla3D_estatica>.

[5] Hello Triangle. <https://learnopengl.com/Getting-started/Hello-Triangle>.

[6] Hektor Docs. Triángulos, vértices y mallas. <https://docs.hektorprofe.net/graficos-3d/06-triangulos-vertices-mallas/>.

[7] R. Rendon Cepeda. (2013) How To Export Blender Models to OpenGL ES Part 1/3 <https://www.kodeco.com/2604-how-to-export-blender-models-to-opengl-es-part-1-3>.

[8] Wavefront .obj file – Wikipedia.pdf' <https://en.wikipedia.org/wiki/Wavefront_.obj_file>.

[7] La puede encontrar en la URL https://thebasemesh.com/.