TEMA 2. MODELADO DE OBJETOS

Objetivos:

- *Conocer distintas formas de representar y almacenar objetos geométricos 3D*
- *Comprender los fundamentos de las transformaciones geométricas*
- *Ser capaz de componer objetos complejos en función de otros más sencillos*
- *Realizar modelos jerárquicos de forma eficiente*

INTRODUCCIÓN

¿Qué es un modelo? Podríamos decir que es un ente hecho por el hombre que nos permite visualizar de forma fácil otro ente, real o imaginario. De este modo, una maqueta hecha por un arquitecto es un modelo del edificio que tiene pensado construir; o una representación de la molécula de hipoclorito sódico (en 2D o 3D) no es más que la forma que tenemos de ver la estructura de las moléculas de lejía.

La Informática Gráfica trata de representar visualmente datos numéricos, si bien estos datos pueden ser obtenidos a partir de entes reales o de la imaginación del que los creó. En cualquier caso, podemos inferir que el origen de todo es un modelo (geométrico, matemático, físico, volumétrico, etc.).

No todos los modelos son iguales y su representación numérica no es igual, ni su visualización es la misma. Además, hay que tener en cuenta que la realidad es infinita, en precisión y en cantidad, y dicha infinitud no puede guardarse en un ordenador. Por tanto, hemos de tener siempre presente que los modelos computacionales no pueden representar exactamente la realidad.

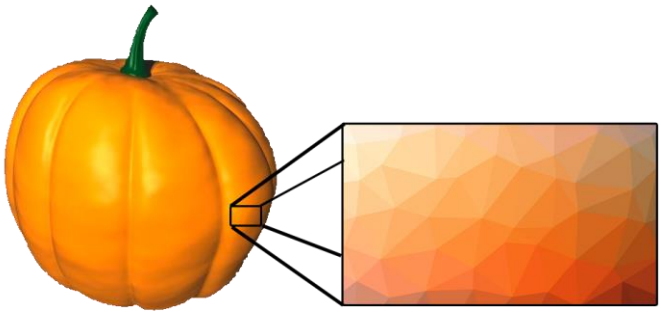


Ilustración 36: Representación de fronteras de una calabaza.

Para representar, de forma aproximada, datos reales, hay dos principales aproximaciones:

- **Modelos basados en fronteras (**Ilustración 36**)**, en los cuales se representa la frontera o superficie del objeto. Para ello se utilizan conjuntos finitos de polígonos planos (caras).
- **Modelos basados en enumeración espacial (**Ilustración e Ilustración 38 **)**, con los que se representan tanto la frontera como el interior de los objetos, o propiedades localizadas en el espacio. Estos modelos, también conocidos como volumétricos, se usan en aplicaciones como la Medicina, la Arqueología o la Paleontología, para visualizar datos obtenidos mediante el TAC (Tomografía Axial Computerizada), georradar, ecografías, u otras técnicas.

Como ejemplo, tomemos una naranja. Si sólo nos interesa verla encima de la mesa, podemos usar un modelo basado en fronteras, que represente la piel. Si por el contrario queremos en cualquier momento cortarla y ver el corte realizado, tendremos que utilizar un modelo basado en enumeración espacial.

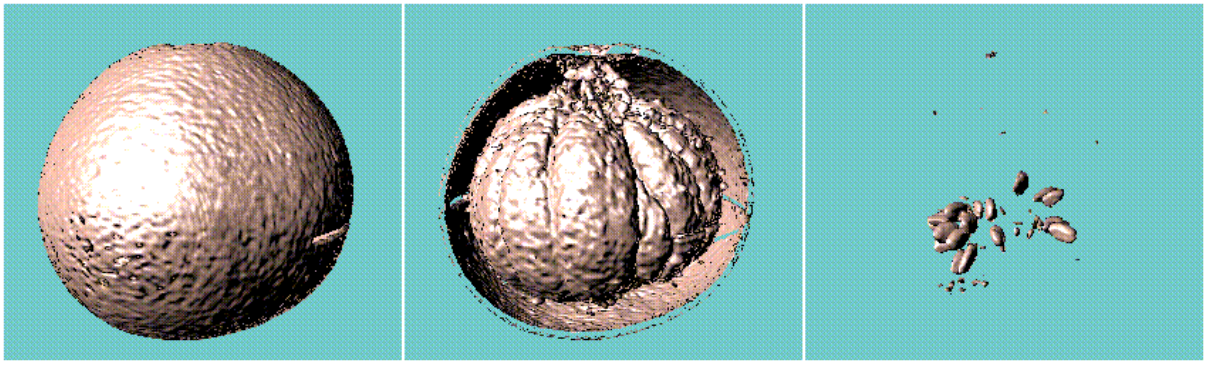


Ilustración 37: Modelo basado en enumeración espacial de una naranja. Con un único modelo se visualiza la piel, los gajos y las semillas. Fuente: http://nis-ei.eng.hokudai.ac.jp/~yamamoto/Gallery/

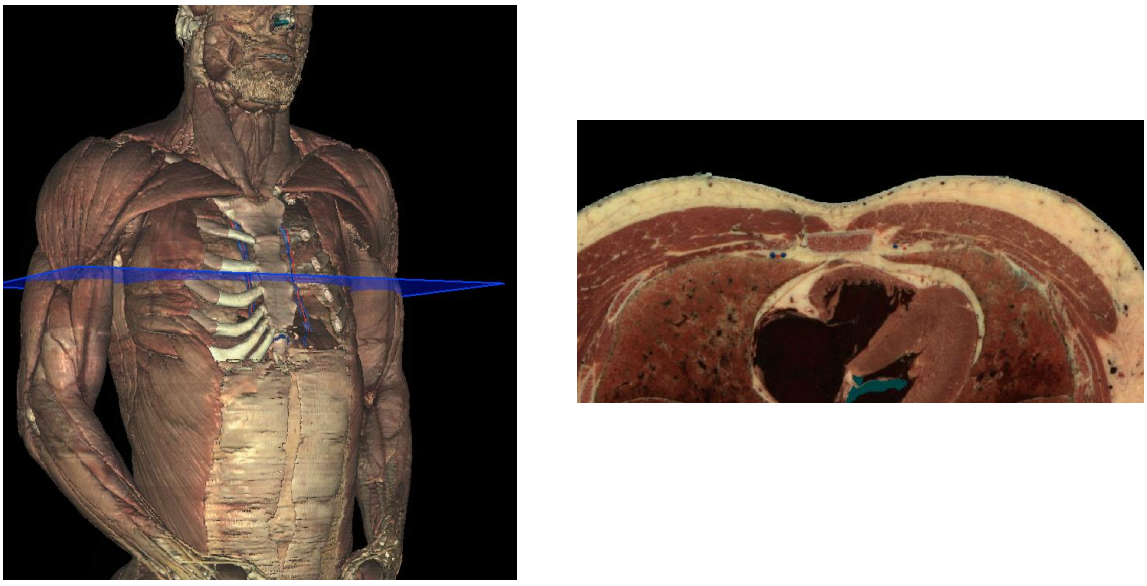


Ilustración 38: Modelo volumétrico del cuerpo humano (VH Dissector de Toltech: http://www.toltech.net/anatomy-software/solutions/vh-dissector-for-medical-education)

MODELOS DE FRONTERAS O SUPERFICIE

En esta asignatura nos vamos a centrar en los modelos de fronteras o de superficie, en inglés *B-rep* (Boundary Representations), dejando los modelos volumétricos para otras asignaturas posteriores de Grado o de Master. Este tipo de modelos permiten una visualización por rasterización de la escena usando OpenGL.

Para representar modelos de fronteras se utilizan polígonos, que vistos todos juntos pueden parecer como un mosaico o una red (según si los vemos como piezas o como líneas que unen los puntos).

Antes de entrar en profundidad en el tema, vamos a comentar algo de terminología:

- **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.

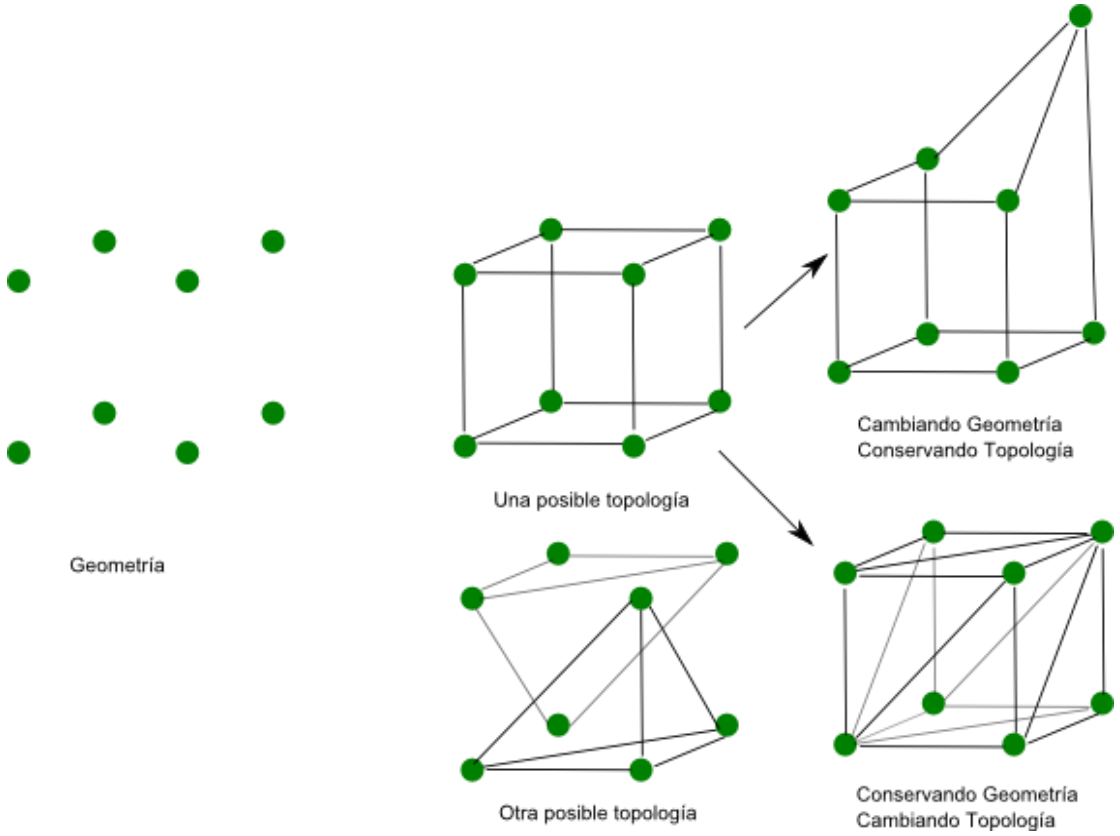


Ilustración 39: Geometría y Topología

En la Ilustración 39 podemos ver cómo geometría y topología no son cosas independientes a la hora de crear un modelo digital 3D, sino que están muy relacionadas. Se muestra cómo con ocho muestras espaciales, ocho vértices, se pueden generar bien un cubo o dos tetraedros, y a su vez, un cubo se puede representar cambiando la topología y conservando la geometria inicial. De igual manera, también se muestra cómo se puede conservar la topología y generar una figura distinta a un cubo.

Viendo la Ilustración 39 también extraemos algunos términos que habríamos de fijar:

- **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. También se puede ver como el segmento que comparten dos caras adyacentes.
- **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. Podemos verlo en la tapa superior del cubo deformado de la Ilustración 39, que no son coplanares sus cuatro puntos. ¿Y qué importancia tiene la

EJERCICIOS

28. *¿Podría afirmar desde el punto de vista topológico que el cubo de la Ilustración 39 se parece mucho más a una esfera que un donut?*

coplanaridad? Como veremos más adelante, uno de los atributos de las caras es su orientación (su normal), y la orientación sólo puede calcularse si es un plano dicha cara.

Esta estructura reticular de caras, aristas y vértices se denomina normalmente **malla de triángulos**, *triangle mesh* en inglés.

A estas alturas tenemos claro que los vértices no son nada sin su triángulo, y que no se pueden formar triángulos sin vértices. Es por ello por lo que para representar y visualizar modelos 3D de superficies, es fundamental disponer de una estructura de datos que nos guarde tanto la topología como la geometría del modelo. Vamos a examinar algunas posibilidades:

Lista de triángulos aislados: sopa de triángulos

Lo más simple es definir cada triángulo uno a uno, con las coordenadas de cada uno sus vértices:

Triángulo	V1	V2	V3
0	x_{v1} y_{v1} z_{v1}	x_{v2} y_{v2} z_{v2}	x_{v3} y_{v3} z_{v3}
1	x_{v1} y_{v1} z_{v1}	x_{v2} y_{v2} z_{v2}	x_{v3} y_{v3} z_{v3}
…			
n	x_{v1} y_{v1} z_{v1}	x_{v2} y_{v2} z_{v2}	x_{v3} y_{v3} z_{v3}

Para cada triángulo necesitamos 9 valores flotantes, y es muy simple, pero tiene numerosos puntos débiles, siendo los dos principales:

- Si un vértice forma parte de *k* triángulos, sus coordenadas aparecen repetidas *k* veces en la lista. En una malla regular, de media, cada vértice es compartido por seis triángulos, por lo que cada *x,y,z* de cada vértice aparece unas seis veces repetido.

- Hay una falta de información explícita sobre la conectividad entre los elementos, por ejemplo, cuando dos triángulos comparten aristas. En algunas aplicaciones esta información es esencial, y se perdería mucho tiempo comparando uno a uno las coordenadas de los vértices.

```
typedef GLfloat Real ;   //Real->GLfloat flotante de (mínimo) 32 bits
typedef GLuint Natural; //Natural->GLuint entero sin signo de (mínimo) 32 bits

const unsigned int     //Índices enteros de los ejes
      X=0, Y=1, Z= 2 ; //implica orden X,Y,Z en las tuplas de coordenadas

typedef Real Tupla3r[3];    // tupla con 3 valores reales representando coordenadas

typedef Natural Tupla3n[3];   // tupla de 3 enteros no negativos (naturales)
                              // se pueden respresentar indices en tablas


typedef struct {   // Malla de Triángulos aislados
      unsigned long num_tri; // Número total de triángulos
      Tupla3r *tri;           // Lista de Triángulos;
} mallaTA;
```

De esta forma, si una variable mesh es del tipo mallaTA, podría acceder al vértice *i* del triángulo *t*-ésimo de la siguiente forma:

```
mesh.tri[t][i]
```

y su coordenada X sería

```
mesh.tri[t][i][X]
```

Visualización en modo inmediato

Para visualizar en modo inmediato podemos usar glBegin/glEnd según vimos en el tema 1:

```
void Visualizar_BE( mallaTA * mesh ) {
       glBegin( GL_TRIANGLES );
             for( Natural i = 0 ; i < mesh->num_tri ; i++ )
                   for( Natural j = 0 ; j < 3 ; j++ )
                         glVertex3fv( mesh->tri[i][j]);
       glEnd();
 }
```

Otra opción, tambien en modo inmediato, es usar una única llamada a `glDrawArrays` y pasarle el vector de coordenadas:

```
void Visualizar_VA( mallaTA * mesh )  {
       // habilitar el uso de puntero a tabla de coordenadas
       glEnableClientState( GL_VERTEX_ARRAY );
       // proporcionar el puntero (tri) la tabla de coordenadas
       glVertexPointer( 3, GL_FLOAT, 0, mesh->tri );
       glDrawArrays( GL_TRIANGLES, 0, 3*pm->num_tri ) ;
}
```

EJERCICIOS

29. *De las dos aproximaciones mostradas (glBegin/glEnd y glDrawArrays), ¿cuál cree que es más eficiente en términos de tiempo de procesamiento? ¿y en cuanto a transferencia CPU-bus-GPU?*
30. *Documente en sus apuntes el formato de archivo STL. ¿Es eficiente?*

Tiras de triángulos

OpenGL ofrece una primitiva que permite ahorrar el repetir vértices en una secuencia de triángulos cuando ésta se organiza como en la Ilustración 40. La primitiva GL_TRIANGLE_STRIP permite reutilizar los vértices sin tener que indicarlos expresamente, de forma que la secuencia:

```
glBegin(GL_TRIANGLE_STRIP);
 glVertex3fv(v0);
 glVertex3fv(v1);
 glVertex3fv(v2);
 glVertex3fv(v3);
 glVertex3fv(v4);
 glVertex3fv(v5);
 glVertex3fv(v6);
glEnd();
```

genera la salida mostrada en la Ilustración 40.

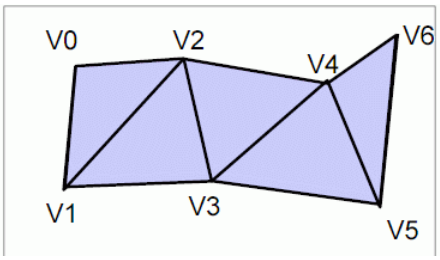


Ilustración 40: Tira de triángulos.

Obviamente, no es fácil sacar de una sola tira toda una malla poligonal, especialmente en casos muy complejos (pensemos en lo difícil que es pelar una manzana toda de un tirón). Hay complejos algoritmos que se encargan de descomponer las mallas de triángulos en tiras lo más largas posibles, pues de esta forma se consigue una menor redundancia en la transmisión de vértices.

EJERCICIOS

31. *¿Cuántas llamadas a glVertex se realizan para una tira de* n *triángulos?*
32. *Defina una estructura de datos que permita almacenar tiras de triángulos de una malla 3D.*

Aunque con esta estructura de datos conseguimos mejorar el rendimiento en cuanto al número de veces que se utiliza cada vértice, seguimos sin disponer de información topológica.

Tablas de vértices y triángulos

Vistas las estructuras anteriores, parece claro que lo que hay que hacer es intentar minimizar la redundancia en la información de los vértices y, además, proporcionar cierto tipo de información topológica.

Se puede entonces usar una estructura que indexe los vértices, y cuya principal información sean los triángulos. Se muestra gráficamente en la Ilustración 41.

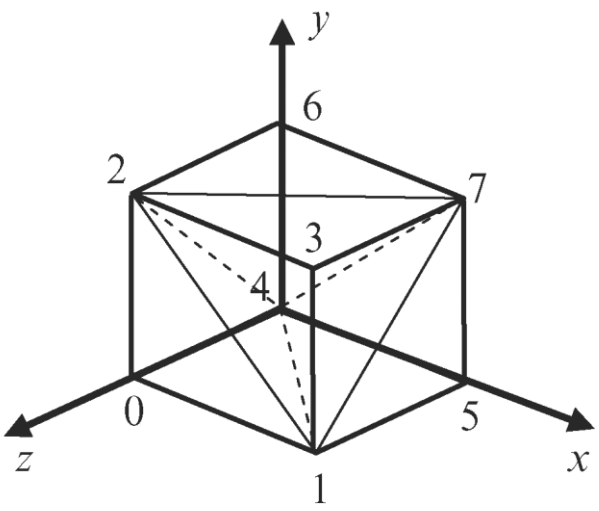


Tabla de vértices		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

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

Ilustración 41 Tabla de Vértices y Triángulos

Podemos entonces tener una estructura que en C sería:

```
struct Malla {
       Natural num_ver;
       Natural num_tri;
       Tupla3r *vertices;
       Tupla3n *triangulos;
};
```

De esta manera, conseguimos tener 3*`num_ver` valores reales contiguos en memoria, correspondientes a los vértices, y 3*`num_tri` valores enteros contiguos (correspondientes a los índices de los vértices que forman los triángulos).

Visualización en modo inmediato

Dada una malla como la descrita anteriormente, se puede visualizar con glBegin/glEnd de la siguiente manera:

```
void  visualizarBE( Malla *mesh )  {
 glBegin( GL_TRIANGLES ) ;
       for( Natural i = 0 ; i < mesh->num_tri ; i++ )
             for( Natural j = 0 ; j < 3 ; j++ )  {
                   //leer el índice del vértice j del triángulo i
                   Natural ind_ver = mesh->triangulos[i][j] ;
                   / / leer y enviar coordenadas del vértice
                   glVertex3fv(mesh->vertices[ind_ver] ) ;
             }
       }
 glEnd() ;
}
```

Dos bucles anidados, y enviar los vértices de uno en uno no parece la mejor forma de visualizar esto, aunque bien es cierto que funciona. Afortunadamente, OpenGL proporciona una función para manejar los índices de vértices, de forma que, proporcionando un vector de vértices y uno de caras (con los índices de los vértices que las forman), se pinte todo.

`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.

Su uso es muy sencillo habiendo preparado los datos como se ha explicado anteriormente. En el caso de un cubo, tan sólo hay que definir el array con vértices, normales, etc. y los índices, que serían:

```
GLubyte indices[] = {0,1,2, 2,3,0,   // 36 indices (los agrupamos visualmente)
                     0,3,4, 4,5,0,
                     0,5,6, 6,1,0,
                     1,6,7, 7,2,1,
                     7,4,3, 3,2,7,
                     4,7,6, 6,5,4};
```

Y en la función de dibujado, se indican el buffer de índices y en lugar de llamar a `glDrawArrays` para pintar, se llama a `glDrawElements` con el puntero a los `índices`.

En el caso de nuestra malla, sería de la siguiente manera:

```
void visualizarVA( Malla *mesh ) {
          glEnableClientState( GL_VERTEX_ARRAY ); // habilitar 'vertex arrays'
     // especificar puntero a tabla de coords. de vértices
     glVertexPointer( 3, GL_FLOAT, 0, mesh->vertices );
     // dibujar usando vértices indexados
     glDrawElements( GL_TRIANGLES, 3*mesh->num_tri,
                     GL_UNSIGNED_INT, mesh->triangulos ) ;
} ;
```

El formato PLY

El formato de archivo PLY fue diseñado por Greg Turk y otros en la Universidad de Stanford a mediados de los 90. Almacena una malla poligonal en un archivo ASCII o binario en formato de tabla de vértices y caras, y permite almacenar no sólo esta información, sino también atributos de las primitivas como el color o las normales. Nótese como en el siguiente código, las caras no son triángulos sino cuadriláteros:

```
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
```

Ilustración 42 Archivo PLY de ejemplo.

Podemos analizar el texto de la Ilustración 42 y ver cómo el archivo PLY consta de las siguientes partes:

- **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 de la Ilustración 42 sus coordenadas X, Y y Z (reales) en ASCII, separadas por espacios. Si se hubiesen definido más `property` en la cabecera, pues habría más números por línea.
- **Lista de caras**, una cara por línea, indicando el número de vértices que tiene y después los índices de los vértices (comenzando en cero para el primer vértice de la tabla de vértices).

Su simplicidad y estandarización hace muy fácil su uso. Si en las prácticas queréis usar un archivo 3D que no está en formato PLY, con la herramienta MeshLab (***meshlab**.sourceforge.net*) podréis exportarlo a PLY (ASCII o binario) sin ningún problema.

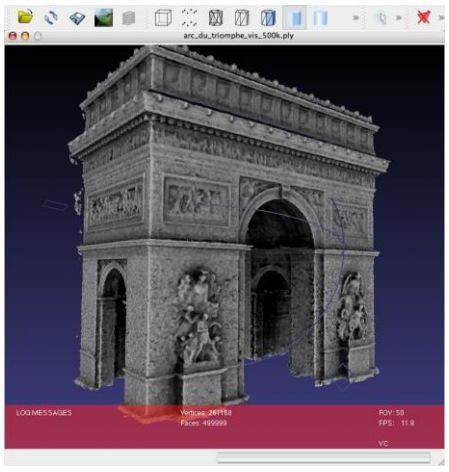


Ilustración 43: Captura de pantalla de MeshLab

Aristas Aladas

En las estructuras anteriores, las posibilidades de navegación por la malla son bastante limitadas, ya que no hay información de adyacencia entre caras ni una secuencia de aristas más o menos coherente.

Ejemplo de operaciones que son costosas de realizar:

- Detectar todas las caras que comparten un mismo vértice
- Dada una arista, detectar las dos caras que la comparten
- Detectar inconsistencias en la orientación de los triángulos.

Las estructuras de datos para mallas poligonales genéricas (no necesariamente de triángulos) están centradas en el concepto de arista, ya que podemos entender los triángulos como una *ilusión óptica* formada por la conectividad entre los vértices, es decir, por la red de aristas.

La estructura de aristas aladas se muestra en la Ilustración 44. Cada arista almacena referencias a los vértices situados en los extremos de la arista, a las dos caras adyacentes a la arista, y a la siguiente y anterior arista en la cara izquierda y derecha. Además, cada vértice y cada cara almacenan una referencia a una de sus aristas.

```
Vértice
Tupla3r p;
Natural arista;
```

```
Arista
Natural V1, V2;
Natural caraIzda,
caraDcha;
Natural aristaSigDcha;
Natural aristaSigIzda;
Natural aristaAntDcha;
Natural aristaAntIzda;
```

```
Cara
Natural arista;
```

Ilustración 44: Estructuras base para crear una lista de aristas aladas.

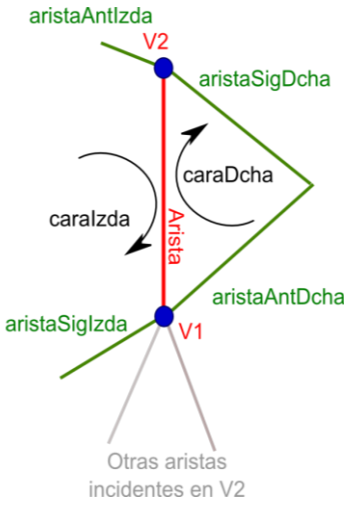


Ilustración 45: Arista alada

Lo que forma la estructura de aristas aladas son tres listas (Vértices, Caras y Aristas), accesibles mediante índices, no la Ilustración 45 que es sólo un gráfico explicativo.

El hecho de almacenar la arista siguiente y anterior permite hacer recorridos por las entradas de la tabla de aristas:

- Dada una arista y una cara adyacente, se pueden saber las aristas de la cara o sus vértices
- Dada una arista y uno de sus vértices, se pueden saber las aristas que inciden en el vértice o los triángulos que comparten ese vértice.

EJERCICIOS

33. Escriba un código en C que calcule el área total de una malla de triángulos almacenada como lista de triángulos y vértices.
34. Defina la estructura de datos en C que almacene la estructura de aristas aladas. Escriba una función en C que la rellene a partir de una lista de triángulos y vértices.

Semiaristas Aladas

La estructura de aristas aladas es bastante elegante y potente, pero tiene una gran pega: hay que estar constantemente comprobando la orientación de la arista antes de decidir el paso a la siguiente arista. Pensemos lo que cambia la estructura de la Ilustración 45 si intercambiamos V1 y V2.

La estructura de ***semiaristas*** aladas elimina la contrariedad anterior dividiendo cada arista en dos semiaristas orientadas en sentido antihorario de la cara a la que pertenece cada una.

```
Vértice
    Tupla3r     p;
    Natural
    semiarista;
```

```
Cara
    Natural
    semiarista;
```

```
Semiarista
Natural V;
Natural cara;
Natural semiaristaSig;
Natural semiaristaAnt;
Natural opuesta;
```

Ilustración 47 Estructuras base para crear una lista de aristas aladas.

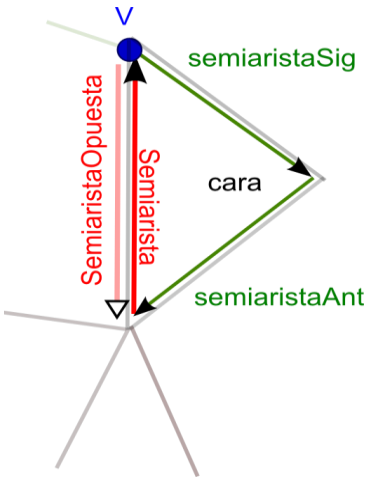
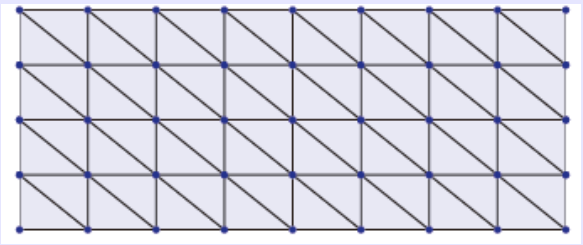


Ilustración 46 Semiarista Alada

A partir de esta estructura de semiaristas aladas han aparecido evoluciones que optimizan aún más el espacio, pero son de menor implantación. Hoy en día, la mayoría de librerías que utilizan mallas de triángulos y conservan la información topológica, usan la estructura de datos de semiaristas aladas.

EJERCICIOS

35. ¿Se podría evitar tener el campo `semiaristaopuesta`? En caso afirmativo, ¿cómo? Si no, ¿por qué?
36. Defina la estructura de datos en C que almacene la estructura de semiaristas aladas.
37. Escriba un código en C que averigüe de forma eficiente el número medio de caras por vértice usando una malla almacenada en la estructura de semiaristas aladas definida en el ejercicio anterior.
38. Considera una malla como la de la figura, en la cual hay *n* columnas y *m* filas de (pares de) triángulos
    - Expresa el número de vértices en función de n y m.
    - Suponiendo que Natural y Real ocupan 4 bytes, calcular el espacio en disco utilizando:
        1. Lista de triángulos
        2. Lista de triángulos y vértices
        3. Aristas aladas
        4. Semiaristas aladas



Atributos de los elementos de los modelos de fronteras

Como modelos de objetos reales, las mallas suelen incluir más información aparte de la geométrica o la topológica. Por ahora nos vamos a centrar en dos importantes para la generación de escenas con cierta calidad estética:

- 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.

Otros atributo habitual es la coordenada de textura.

Cada vértice que procesa OpenGL tiene asociado siempre una normal y un color, y en función de la forma de dibujar que estemos usando, se definen de una forma u otra.

- Si se usa **`glVertex`**, en el momento de ejecutar dicha instrucción se asocia al vértice el color actual y la normal actual, o lo que es lo mismo, la última normal y el último color especificados con las funciones **`glNormal`** y **`glColor`**.
- Si se usan *vertex arrays* (ya sea con **`glDrawArrays`** o con **`glDrawElements`**), se puede especificar un array de colores y otro de normales, donde habrá una entrada por cada vértice:

```
void glNormalPointer(GLenum type, GLsizei stride, const GLvoid * pointer);
void glColorPointer(GLint size, GLenum type,
                    GLsizei stride, const GLvoid * pointer);
```

  Si no se especifican estos array en el momento de dibujar los vértices, se usa el color actual y la normal actual a todos los vértices.

Normales de Caras

Son útiles cuando el objeto que se modela con la malla está realmente compuesto de caras planas, como el cubo de la Ilustración 48.

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).

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

Como las normales de un objeto rígido no cambian en ningún momento, se pueden precalcular y almacenar en la malla.

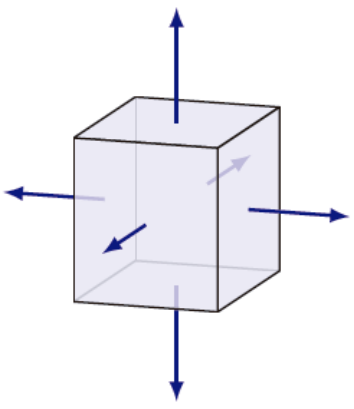


Ilustración 48 Normales de las caras de un cubo.

Normales de Vértices

Si se aproxima un objeto curvo, 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:

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

Donde m_i son las normales de las caras que comparten dicho vértice, calculadas como se ha indicado anteriormente.

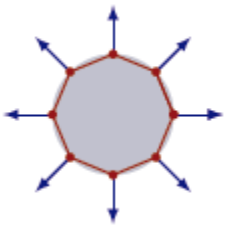


Ilustración 49 Normales en vértices

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.

En la Ilustración 50 se puede ver la diferencia de usar normales por cara y normales por vértice.

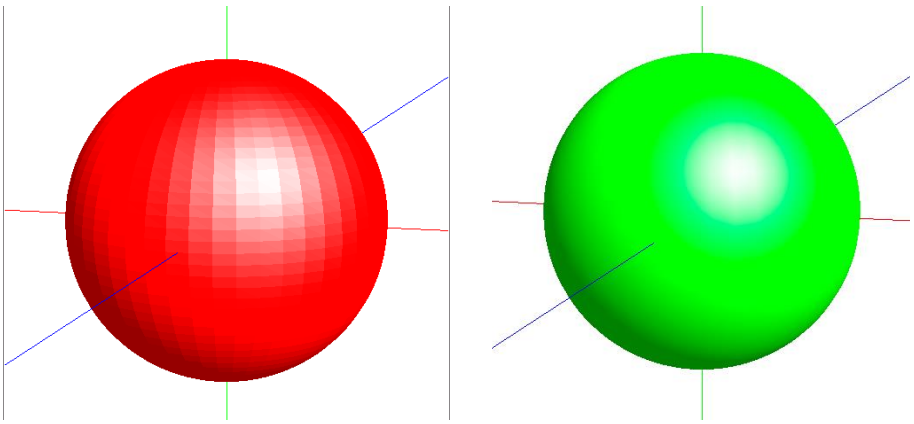


Ilustración 50: Iluminación usando normales de caras (izda) y normales por vértice (dcha.)

Colores de caras y de vértices

Hay casos en los que podemos querer pintar un color homogéneo en cada cara, como en la Ilustración 51, o bien que sepamos los valores de color de cada vértice en concreto y tengamos que interpolar los valores dentro de los polígonos, como en la Ilustración 52 .

Si queremos un color por cara, en OpenGL debemos dejar el color con el mismo valor para todos los vértices del triángulo, p.ej:

```
glColor3f(1.0,1.0,1.0);
glBegin(GL_TRIANGLES);
 glVertex3fv(v0);
 glVertex3fv(v1);
 glVertex3fv(v2);
glEnd();
```

Ilustración 51: Color por cara

Para el color por vértice, hemos de cambiar el color antes de indicar cada coordenada de vértice:

```
glBegin(GL_TRIANGLES);
 glColor3f(1.0,0.2,0.6);
 glVertex3fv(v0);
 glColor3f(0.2,1.0,1.0);
 glVertex3fv(v1);
 glColor3f(0.8,0.3,1.0);
 glVertex3fv(v2);
glEnd();
```

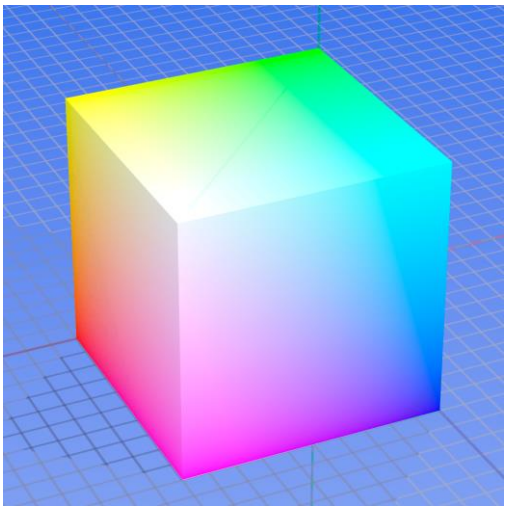


Ilustración 52: Color por vértice

Para conseguir el efecto de interpolación de color en la Ilustración 52, o el 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.

```
typedef struct {
 Natural num_tri, num_ver ; // Número total de triángulos y vértices
 Tupla3n *tri; // Lista de Triángulos;
 Tupla3r *ver;       // Lista de Vértices;
 Tupla3r *nv, *cv;   // Listas de Normales y Colores por vértice;
 Tupla3r *nc, *cc; // Lista de Normales y Color por Caras;
} Malla;

void Visualizar_BE_AtrCara( Malla * mesh ) {
 glBegin( GL_TRIANGLES ) ;
   for( Natural i = 0 ; i < mesh->num_tri ; i++ ) {
        if ( mesh->cc != NULL )
              glColor3fv( mesh->cc[i] );
        if ( mesh->nc != NULL )
              glNormal3fv( mesh->nc[i]);

        for( Natural j = 0 ; j < 3 ; j++ )      {
           Natural iv = mesh->tri[i][j] ;
           glVertex3fv( mesh->ver[iv]) ;
        }
   }
  glEnd();
}

void Visualizar_BE_AtrVertice( Malla * mesh ) {
 glBegin( GL_TRIANGLES ) ;
   for( Natural i = 0 ; i < mesh->num_tri ; i++ ) {
        for( Natural j = 0 ; j < 3 ; j++ )      {
           Natural iv = mesh->tri[i][j] ;
           if ( mesh->cc != NULL )
               glColor3fv( mesh->cv[i] );
           if ( mesh->nc != NULL )
               glNormal3fv( mesh->nv[i]);

           glVertex3fv( mesh->ver[iv]) ;
        }
   }
glEnd();
}
```

Código 1 Código para visualizar con glBegin/glEnd normales y colores de vértices y caras.

```
void Visualizar_VA_AtrVertice( Malla *mesh ) {
 if ( mesh->nv != NULL )  {
   glEnableClientState( GL_NORMAL_ARRAY ); // habilitar uso de array de normales
   glNormalPointer( GL_FLOAT, 0, mesh->nv); / / especifica puntero a normales
  }

  if ( mesh->cv != NULL ) {
    glEnableClientState( GL_COLOR_ARRAY ); / / habilitar uso de array de col.
    glColorPointer( 3, GL_FLOAT, 0, mesh->cv); // especifica puntero a colores
  }

  glEnableClientState( GL_VERTEX_ARRAY );
  glVertexPointer( 3, GL_FLOAT, 0, mesh->ver );

  glDrawElements( GL_TRIANGLES, 3*mesh->num_tri, GL_UNSIGNED_INT, mesh->tri);
}
```

Código 2: Visualización con colores y normales por vértice usando vertex array.

En los trozos de código anteriores podemos ver ejemplos de cómo se visualiza con `glBegin/glEnd` y con *vertex array* usando estos atributos de vértices.

EJERCICIOS

39. Escriba un código en C que calcule la normal de un triángulo, dados sus tres vértices.
40. Escriba un código en C que, dada una malla de triángulos, calcule la normal en los vértices, si:
    - La malla está definida como lista de caras y vértices,
    - La malla está definida con una estructura de semiaristas aladas.
41. ¿Tiene influencia en el valor de la normal en el vértice el área de los triángulos que lo comparten?
42. Escribe el código para crear una copia de una malla de triángulos, de forma que en la copia los vértices estén cada uno desplazado en la dirección de su normal una distancia d, siendo d un parámetro Real de la función.
43. Escribe el código para visualizar una malla con los triángulos de un color sólido y las aristas en modo alambre, a la vez. ¿Qué problemas se pueden encontrar?

TRANSFORMACIONES GEOMÉTRICAS

Concepto de Transformación Geométrica

Los modelos que hemos visto hasta ahora tenían definida su geometría con respecto a un sistema de coordenadas que podríamos denominar *local*, pues de hecho sólo le afecta a dicho modelo.

Sin embargo, en una escena con varios modelos, todos los vértices deben aparecer referidos a un único sistema de referencia común. Dicho sistema, denominado **sistema de referencia de la escena, o del mundo** (*world coordinate system*), hace que las coordenadas de los vértices no sean las que originalmente se dotaron al objeto, sino que se expresan en **coordenadas del mundo.**

Una de las principales ventajas es que un objeto se puede definir una vez, pero se puede instanciar varias veces en una o distintas escenas.

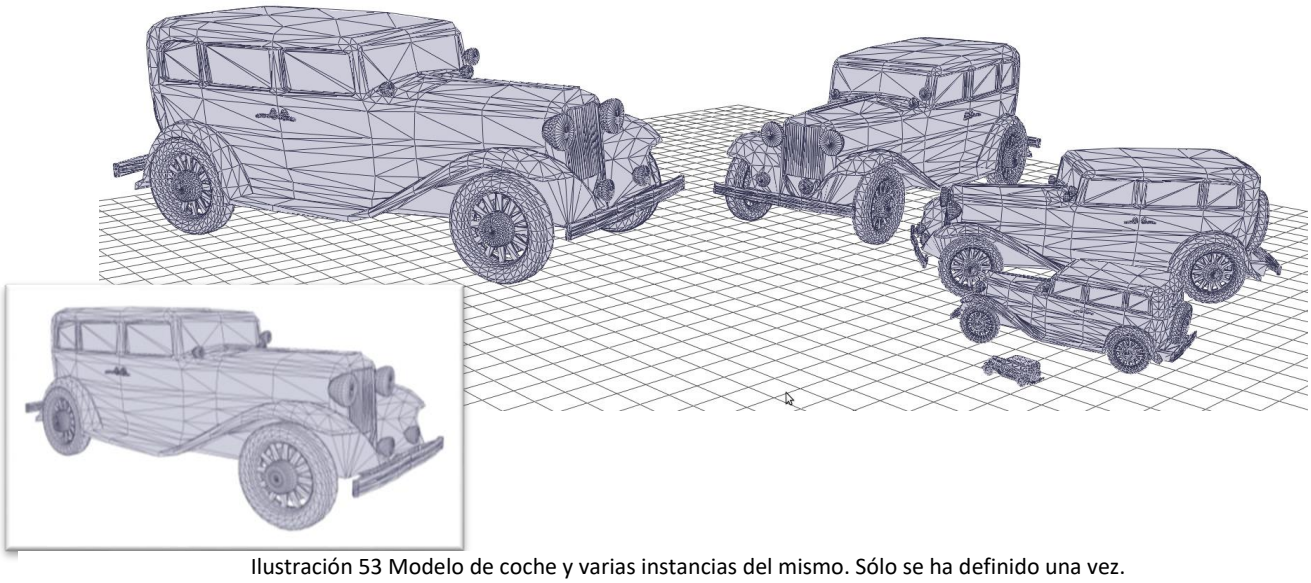


Ilustración 53 Modelo de coche y varias instancias del mismo. Sólo se ha definido una vez.

Para lograr una escena como la de la Ilustración 53, es necesario modificar la posición de los vértices del coche original, y podemos ver que hay no sólo cambios de tamaño, sino también de orientación y de posición. Esta modificación no es más que calcular sus coordenadas del mundo a partir de las coordenadas locales.

Para esto se usan las **transformaciones geométricas.**

Una transformación geométrica T, se define matemáticamente como una aplicación que asocia a cualquier punto o vector **p** otro punto o vector **q,** y escribimos

q = T p

que se lee "**q** es T aplicado a **p"**

Una transformación T cambia las coordenadas de los puntos sobre los que actúa.

Transformaciones usuales en Informática Gráfica

Existen varias transformaciones geométricas simples que son muy útiles en Informática Gráfica para la definición de escenas y animaciones, y que constituyen la base de otras transformaciones:

- **Traslación**. Consiste en desplazar todos los puntos del espacio de igual forma, es decir, en la misma dirección y la misma distancia.
- **Escalado**. Supone estrechar o alargar las figuras en una o varias direcciones.
- **Rotación**. Rotar los puntos un ángulo dado en torno a un eje de rotación.
- **Cizalla**. Se puede ver como un desplazamiento de todos los puntos en la misma dirección, pero con distintas distancias.

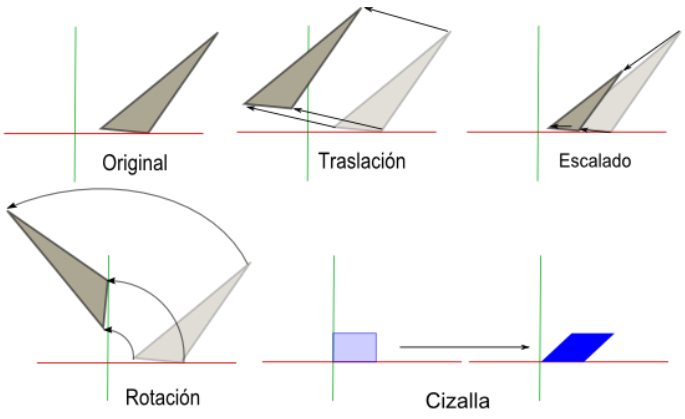
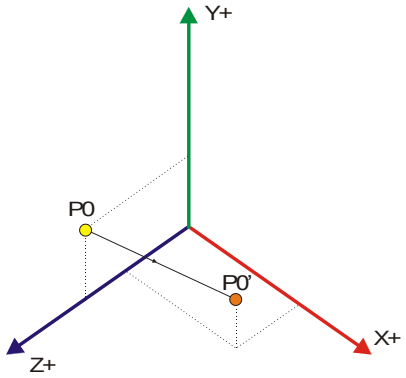


Ilustración 54: Transformaciones geométricas

Se puede ver el efecto de estas transformaciones en la Ilustración 54.

Traslación

Si $\boldsymbol{d}$ es un vector, la transformación de traslación T[$\boldsymbol{d}$] en el espacio 3D es una transformación geométrica que desplaza cada punto según el vector director $\boldsymbol{d}$.



Si definimos la traslación como una función *f(x,y,z)* (resumido sería *T[dx,dy,dz]*) que afecta a las coordenadas del punto *p*, podemos definirla en 3D cómo

$$p'_x = p_x + dx$$
$$p'_y = p_y + dy$$
$$p'_z = p_z + dz$$

En la Ilustración 55 podemos ver cómo el modelo *casa* se ve afectado por las traslaciones T[0,6,0] (centro) y T[5,4,-1] (derecha).

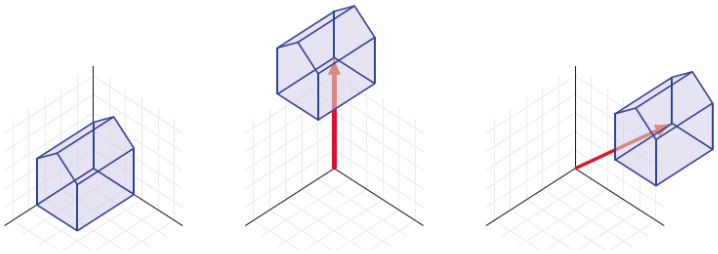


Ilustración 55: Dos transformaciones de traslación sobre el modelo casa.

Escalado

La transformación de escalado viene determinada por tres valores reales ($\boldsymbol{e_x}$, $\boldsymbol{e_y}$, $\boldsymbol{e_z}$). Equivale a un cambio de escala o tamaño con centro en el origen del sistema de coordenadas, es decir, que el origen queda inalterable.

Si definimos el escalado como una función f(x,y,z) (resumido sería E[$\boldsymbol{e_x}$, $\boldsymbol{e_y}$, $\boldsymbol{e_z}$]) que afecta a las coordenadas del punto p, podemos definirla en 3D cómo

$$f_x(p) = p'_x = e_x p_x$$
$$f_y(p) = p'_y = e_y p_y$$
$$f_x(p) = p'_z = e_z p_z$$

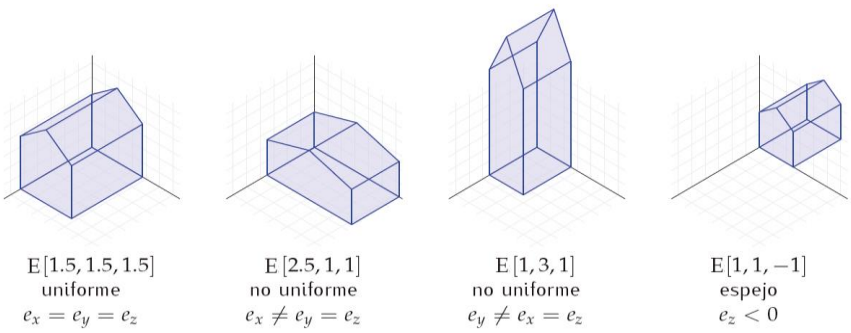


Ilustración 56 Diversas operaciones de escalado.

Cuando los tres valores de escala, e_x, e_y, e_z tienen el mismo valor, se dice que es un **escalado uniforme**. Los escalados uniformes conservan las proporciones y los ángulos de los objetos, y se puede abreviar escribiendo E[e].

La función de escalado puede fácilmente representarse en forma matricial, con la siguiente expresión:

$$[p'_x \quad p'_y \quad p'_z] = [p_x \quad p_y \quad p_z] \begin{bmatrix} e_x & 0 & 0 \\ 0 & e_y & 0 \\ 0 & 0 & e_z \end{bmatrix}$$

En la Ilustración 56 vemos diversas transformaciones de escalado sobre el modelo *casa*. Nótese cómo la operación de *espejo* no es más que un escalado con valor negativo en el eje no incluido en el plano espejo.

Rotación

Rotación en 2D

En un marco de referencia R=[x,y,O] (dos ejes, X e Y, con origen en O), podemos construir la transformación de rotación R[α] en torno al origen O un ángulo α radianes con las siguientes funciones:

$$f_x(p) = p'_x = p_x \cos(\alpha) - p_y \sin(\alpha)$$

$$f_y(p) = p'_y = px \sin(\alpha) + py \cos(\alpha)$$

En la Ilustración 57 podemos ver cómo afecta la rotación a la figura de la casa.

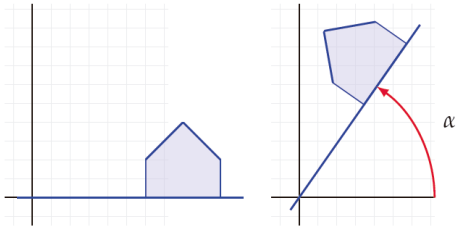


Ilustración 57: Rotación 2D con respecto al origen de coordenadas.

Esta operación también se puede expresar matricialmente como:

$$[p'_x \quad p'_y] = [p_x \quad p_y] \begin{bmatrix} \cos(\alpha) & sin(\alpha) \\ -\sin(\alpha) & cos(\alpha) \end{bmatrix}$$

Rotación en 3D

En 3D hay tres posibles rotaciones básicas, una en torno a cada eje del sistema de coordenadas. Las llamaremos $R_x[\alpha]$, $R_y[\alpha]$ y $R_z[\alpha]$, y visualmente se pueden ver en la Ilustración 58.

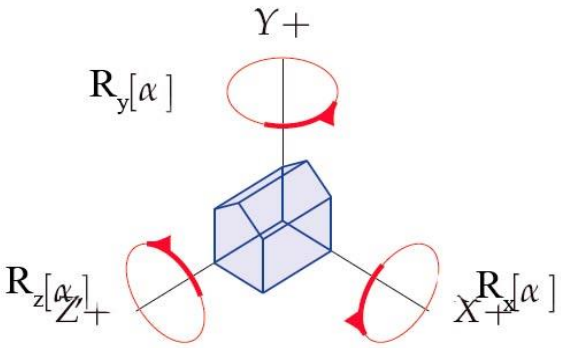


Ilustración 58: Rotaciones en 3D

Como características tenemos:

- Cada rotación modifica dos coordenadas y deja la del eje que se usa para la rotación intacta.
- Son rotaciones siempre en sentido anti-horario para α>0.

Al ser tres rotaciones distintas, que afectan a ejes distintos, ya no hay una única fórmula de la rotación, sino que hemos de definir tres funciones distintas. Estas funciones se pueden expresar de la siguiente manera:

$R_x[\alpha] = f(p)$	$R_y[\alpha] = f(p)$	$R_z[\alpha] = f(p)$
$f_x(p) = p' = p_x$ $f_y(p) = p'_y = \cos(\alpha)\, p_y - \sin(\alpha)\, p_z$	$f_x(p) = p' = \cos(\alpha)\, p_x + \sin(\alpha)\, p_z$ $f_y(p) = p'_y = p_y$	$f_x(p) = p' = \cos(\alpha)\, p_x - \sin(\alpha)\, p_y$ $f_y(p) = p'_y = \sin(\alpha)\, p_x + \cos(\alpha)\, p_y$

$f_z(p)=p'_z= sin(\alpha)\ p_y + cos(\alpha)\ p_z$	$f_z(p)=p'_z= -sin(\alpha)\ p_x + cos(\alpha)\ p_z$	$f_z(p)=p'_z= p_z$

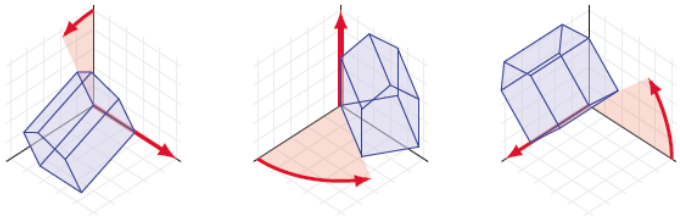


Ilustración 59. Visualización de rotaciones en 3D
$R_x[\pi/8]$ (izda), $R_y[2\pi/3]$ (centro), $R_z[\pi/2]$ (dcha.)

Estas rotaciones también se pueden expresar matricialmente, siendo la matriz transformadora en cada caso la siguiente:

$R_x[\alpha]$	$R_y[\alpha]$	$R_z[\alpha]$
$\begin{bmatrix}1 & 0 & 0\\ 0 & cos(\alpha) & sin(\alpha)\\ 0 & -sin(\alpha) & cos(\alpha)\end{bmatrix}$	$\begin{bmatrix}cos(\alpha) & 0 & -sin(\alpha)\\ 0 & 1 & 0\\ sin(\alpha) & 0 & cos(\alpha)\end{bmatrix}$	$\begin{bmatrix}cos(\alpha) & sin(\alpha) & 0\\ -sin(\alpha) & cos(\alpha) & 0\\ 0 & 0 & 1\end{bmatrix}$

De forma que la rotación en torno al eje X de un punto p sería:

$$[p'_x \quad p'_y \quad p'_z] = [p_x \quad p_y \quad p_z]\begin{bmatrix}1 & 0 & 0\\ 0 & cos(\alpha) & sin(\alpha)\\ 0 & -sin(\alpha) & cos(\alpha)\end{bmatrix}$$

EJERCICIOS

44. Calcule los nuevos valores de la posición en el espacio del punto [3,0,0] aplicándole las rotaciones del ejercicio anterior, por separado, y además tras las siguientes secuencias:
    - $R_x[90] \rightarrow R_y[-90] \rightarrow R_z[200]$
    - $R_z[200] \rightarrow R_y[-90] \rightarrow R_x[90]$

    Para estas secuencias, calcule tras cada rotación el valor temporal del punto y su posición final.

Composición de Transformaciones

Una transformación C puede obtenerse como composición de otras dos transformaciones: A (primero) y B (después) como se muestra en la Ilustración 60 . En ese caso, se escribe $C = B \cdot A$

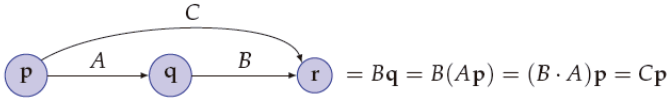


Ilustración 60: Composición de transformaciones.

La composición es, en general, **no conmutativa**, y se puede extender a 3 o más transformaciones:

$$T_4(\ T_3(\ T_2(\ T_1(p)))) = (T_4 \cdot T_3 \cdot T_2 \cdot T_1\ )p$$

La composición es asociativa, y se interpreta de derecha a izquierda, es decir, es como si se aplicara primero T_1, luego T_2, a continuación T_3 y finalmente T_4.

EJERCICIOS

45. Para comprobar la no conmutatividad de las transformaciones geométricas, calcule los nuevos valores de la posición en el espacio del punto p= [3,0,0] aplicándole las siguientes secuencias:
    - $P'=(T_3 \bullet T_2 \bullet T_1)p$
    - $P'=(T_1 \bullet T_2 \bullet T_3)p$
    - $P'=(T_3 \bullet T_1 \bullet T_2)p$

    Siendo
    - $T_1 = R_x[90]$
    - $T_2 = T[-9,4,3]$
    - $T_3 = R_z[45]$

    Para estas secuencias, calcule tras cada transformación de la composición el valor temporal del punto y su posición final.

Composición de transformaciones como producto de matrices

Supongamos una secuencia $T_1, T_2, \ldots, T_n$ de n transformaciones. Si consideramos la transformación compuesta

$$C = T_n \cdot T_{n-1} \cdot \ldots \cdot T_2 \cdot T_1$$

Entonces, la matriz M_c asociada a C será el producto de las matrices M_i asociadas a cada una de las T_i:

$$M_c = M_n M_{n-1} \ldots M_2 M_1$$

Esta propiedad es fundamental, pues nos permitirá obtener matrices de transformaciones compuestas mediante multiplicación de matrices, que recordemos las vistas hasta ahora:

$$E[e_x, e_y, e_z] = \begin{bmatrix} e_x & 0 & 0 \\ 0 & e_y & 0 \\ 0 & 0 & e_z \end{bmatrix}$$

$$R_x[\alpha] \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{bmatrix} \qquad R_y[\alpha] \quad \begin{bmatrix} cos(\alpha) & 0 & -sin(\alpha) \\ 0 & 1 & 0 \\ sin(\alpha) & 0 & cos(\alpha) \end{bmatrix} \qquad R_z[\alpha] \quad \begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 \\ -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Pero, ¿qué ocurre con la Traslación? No hemos visto ninguna matriz… porque no se puede expresar de forma matricial con una matriz 3x3.

Por tanto, parece que con estas matrices no podremos usar composición de matrices… Pero si recuerda, al principio de la asignatura comentamos que OpenGL tiene una única matriz de transformación, donde se va acumulando. ¿Cómo lo hace? En 4D, con las coordenadas homogéneas.

Coordenadas homogéneas

Las coordenadas homogéneas nos permiten representar de la misma manera puntos y vectores en el espacio afín.

El espacio afín 2D (x,y) es un subespacio del espacio proyectivo 3D (X,Y,W), definido como los puntos con coordenada W = 1.

De la misma manera, el espacio afín 3D (x,y,z) es un subespacio del espacio proyectivo 4D (X,Y,Z,W), definido como los puntos con coordenada W=1.

Dicha esta introducción matemática, podemos expresar cualquier punto p(x,y,z) como

$$P=[x\ y\ z\ 1]$$

en coordenadas homogéneas 4D.

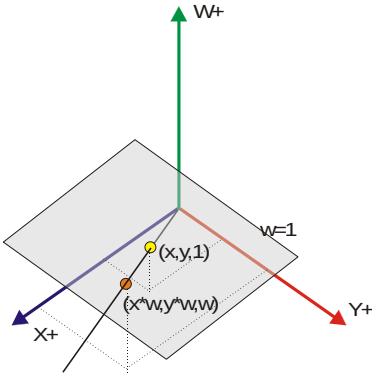


Ilustración 61: En coordenadas homogéneas, una recta 3D se convierte en un mismo punto en el plano 2D w=1

De hecho, en realidad la conversión es

$P^{3D}[x,y,z]=[P^{4D}{}_x / P^{4D}{}_w, P^{4D}{}_y / P^{4D}{}_w, P^{3D}{}_z / P^{4D}{}_w]$

Para cada coordenada, su valor es el resultado de dividir por w el valor en el espacio de coordenadas homogéneas. P.ej.

Si $P^{4D}=[3,4,6,2]$, el punto en el espacio afín 3D P^{3D} es $P^{3D}=[1.5,2,3]$

¿Y esto qué implicación tiene? Pues que todas las matrices anteriormente presentadas como 3x3 pueden expresarse, invariantemente como 4x4:

$$E[e_x,e_y,e_z,1]=\begin{bmatrix} e_x & 0 & 0 & 0 \\ 0 & e_y & 0 & 0 \\ 0 & 0 & e_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x[\alpha]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) & 0 \\ 0 & -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y[\alpha]\begin{bmatrix} cos(\alpha) & 0 & -sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\alpha) & 0 & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z[\alpha]\begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 & 0 \\ -sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y como novedad, ahora ya sí se puede expresar la transformación de traslación como una matriz 4x4

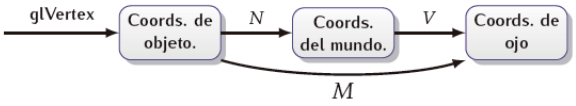
$$T[d_x,d_y,d_z,1]=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x & d_y & d_z & 1 \end{bmatrix}$$

¡Ya podemos realizar composición de matrices para todas las transformaciones geométricas!

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.



La matriz modelview M se aplica a todos los puntos indicados con glVertex.

Especificación de la matriz de modelado

La matriz modelview se puede especificar en OpenGL mediante estos pasos:

1. Hacer la llamada `glMatrixMode(GL_MODELVIEW)` para indicar que las siguientes operaciones operan sobre la matriz modelview M
2. Usar `glLoadIdentity` para hacer M igual a la matriz identidad
3. Usar `gluLookAt` u otras para componer una matriz de vista V
4. Usar una o varias llamadas para componer la matriz de modelado N:
   - `glRotatef( GLfloat a, GLfloat ex, GLfloat ey, GLfloat ez )`
   - `glTranslatef( GLfloat dx, GLfloat dy, GLfloat dz )`
   - `glScalef( GLfloat sx, GLfloat sy, GLfloat sz )`
   - `glMultMatrixf( GLfloat * A ) ;`

Y al final, se tiene $M = V \cdot N$. OpenGL construye M componiendo las matrices que se le proporcionan en los pasos 3 y 4.

Generación de la matriz modelview por composición

OpenGL va componiendo la matriz *modelview* con las instrucciones que se le indiquen durante la ejecución, de forma que si tenemos el siguiente código:

```
glMatrixMode(GL_MODELVIEW); // indicamos que, a partir de aquí,
                            // se modifica la matriz modelview, M.
glLoadIdentity() ; / / hacemos M:= I (matriz identidad)
glMultMatrix( T3 ); / / hacemos M:=M·T3
glMultMatrix( T2 ); / / hacemos M:=M·T2
glMultMatrix( T1 ); / / hacemos M:=M·T1
```

Al final de ejecutar estas instrucciones, la matriz *modelview* será

$$M = T_3 \cdot T_2 \cdot T_1$$

Es decir, el efecto de T_1 seguido de T_2 seguido de T_3

EJERCICIOS

46. Escribe funciones en C para manipulación de transformaciones y para aplicar transformaciones a tuplas de 3 reales.

`Componer( Transformacion *res, Transformacion *m1, Transformacion *m2 )`
Compone dos matrices de transformación m1 y m2, y guarda el resultado en res

`Aplicar( Real res[4], Transformacion *m, Real coo[4] )`
Aplica la matriz de transformación m a la tupla de 4 reales coo, y almacena el resultado en res.

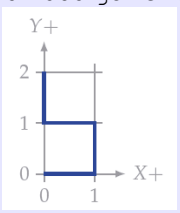
`Rotacion( Transformacion *res, Real a, Real ex, Real ey, Real ez )`
Crea una matriz de rotación de a grados entorno al eje definido por (ex, ey, ez). Escribe la matriz resultado en res.

`Traslacion( Transformacion *res, Real dx, Real dy, Real dz )`
Escribe en res la matriz de traslación asociada al vector de desplazamiento (dx, dy, dz).

`Escalado( Transformacion *res, Real sx, Real sy, Real sz )`
Escribe en res la matriz de escalado con factores de escala sx,sy y sz.

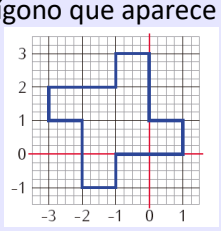
Siendo `typedef Real Transformacion[4][4];`

47. Escribe una función en OpenGL, llamada `gancho` para dibujar con OpenGL la figura:



Cada segmento recto tiene longitud unidad, y el extremo inferior está en el origen. Use para ello `glBegin`, `glVertex2f` y `glEnd`

48. Usando exclusivamente la función `gancho` del problema anterior, construye otra función `gancho_x4` para dibujar el polígono que aparece en la figura:



Hay que tener en cuenta que la figura se puede obtener exclusivamente usando rotaciones de la original, pero en esas rotaciones el centro no es el origen.

49. Documente en sus apuntes: ¿Cómo se hace una rotación con respecto a cualquier punto?
50. Escribe el pseudocódigo OpenGL otra función `gancho_2p` para dibujar la figura delejercicio 52, pero escalada y rotada de forma que sus extremos coincidan con dos puntos arbitrarios p0 = (x0, y0) y p1 = (x1, y1), que se pasan como parámetro a dicha función.

MODELOS JERÁRQUICOS. REPRESENTACIÓN Y VISUALIZACIÓN.

Hasta ahora todos los modelos que hemos usado en las ilustraciones son figuras simples o cargadas desde un archivo externo.

Pero la realidad es más compleja, y no siempre se puede conocer la geometría exacta de cada uno de los elementos que componen los objetos como el de la Ilustración 62. Ese, o el de la Ilustración 63, son claros ejemplos de modelos jerárquicos.

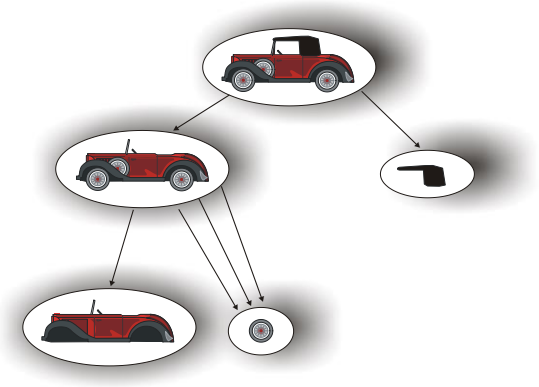


Ilustración 62: Un coche compuesto de tres elementos simples.

Podemos decir que los objetos pueden clasificarse en:

- *objetos simples*, aquellos que no están compuestos de otros más simples
- *objetos compuestos*, es decir, objetos que se forman mediante instanciación y transformación de otros *objetos simples*.

Pensemos en modelar un árbol. La forma más simple puede ser usar un cilindro para el tronco y una esfera para la copa. El cilindro y la esfera serían los objetos simples que forman el objeto compuesto *árbol*.

De la misma manera, por generalización, podríamos pensar que una ***Escena*** es un conjunto de instancias distintas de objetos geométricos, modificados mediante transformaciones de rotación, escalado o traslación. Cada una de estas transformaciones sirve para situar al objeto, que fue definido con respecto a su propio sistema de referencia, en un sistema de coordenadas común a toda la escena, el *marco de referencia del mundo*.

De hecho, un mismo objeto puede instanciarse varias veces, aplicándosele en cada caso una transformación distinta.

En la Ilustración 63 podemos ver a la izquierda un brazo robótico, y en el centro una abstracción del mismo. Podemos ver a la derecha como en realidad estamos usando la misma figura "triángulo" tres veces, pero con distinto tamaño y en distintas posiciones.

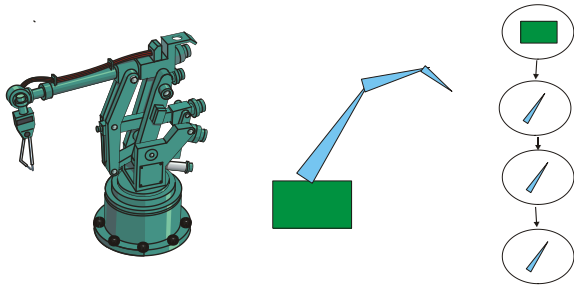


Ilustración 63: Modelo Jerárquico móvil

Representación de modelos jerárquicos.

La estructura que representa este tipo de objetos, o incluso las escenas, se denomina **grafo de escena**. Un grafo de escena es un *grafo dirigido acíclico*, donde:

- cada objeto compuesto es un subgrafo dentro del grafo
- cada objeto simple es un nodo terminal
- cada arco une dos nodos, y estos nodos pueden ser transformaciones geométricas, nodos terminales o nodos *grupo* que permiten unir elementos del grafo.

En la Ilustración 64 se puede ver un grafo de escena con un objeto simple y dos objetos compuestos, pero en realidad se visualizan un total de cuatro elementos, ya que C2 es dibujado con dos transformaciones distintas T2 y T3. Es lo que se llama ***instanciación.***

En este grafo podemos ver como hay:

- pares de hermanos instanciados con la misma transformación
- nodos instanciados más de una vez
- nodos terminales a los que se llega por varias rutas, lo que supone distintas instancias del nodo.

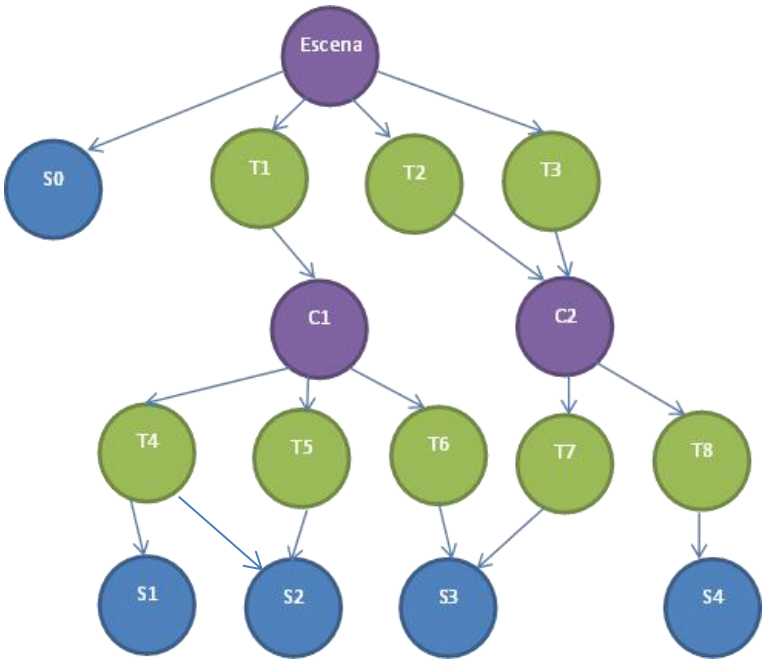


Ilustración 64 Un grafo de escena cualquiera.

Por ejemplo, el nodo terminal (objeto simple) S3 se ve afectado por la siguiente secuencia de transformaciones:

- T1·T6
- T2·T7
- T3·T7

Nótese que las transformaciones se leen desde el nodo raíz.

Ejemplos de grafos

Vamos a ver unos cuantos ejemplos en 2D de cómo se compondrían modelos jerárquicos, o escenas, basándonos en los objetos simples de la Ilustración 65.

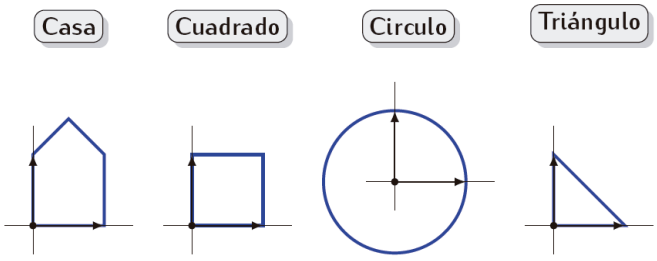


Ilustración 65. Objetos simples para los ejemplos

Todo se puede hacer con la combinación adecuada de transformaciones e instanciaciones de estos objetos simples.

Estos objetos simples tienen como dimensión básica la unidad: el círculo es de radio 1, el cuadrado de lado 1, el triángulo rectángulo de catetos longitud 1 y la casa de base y altura de muros 1.

Ejemplo Construcción de una casa

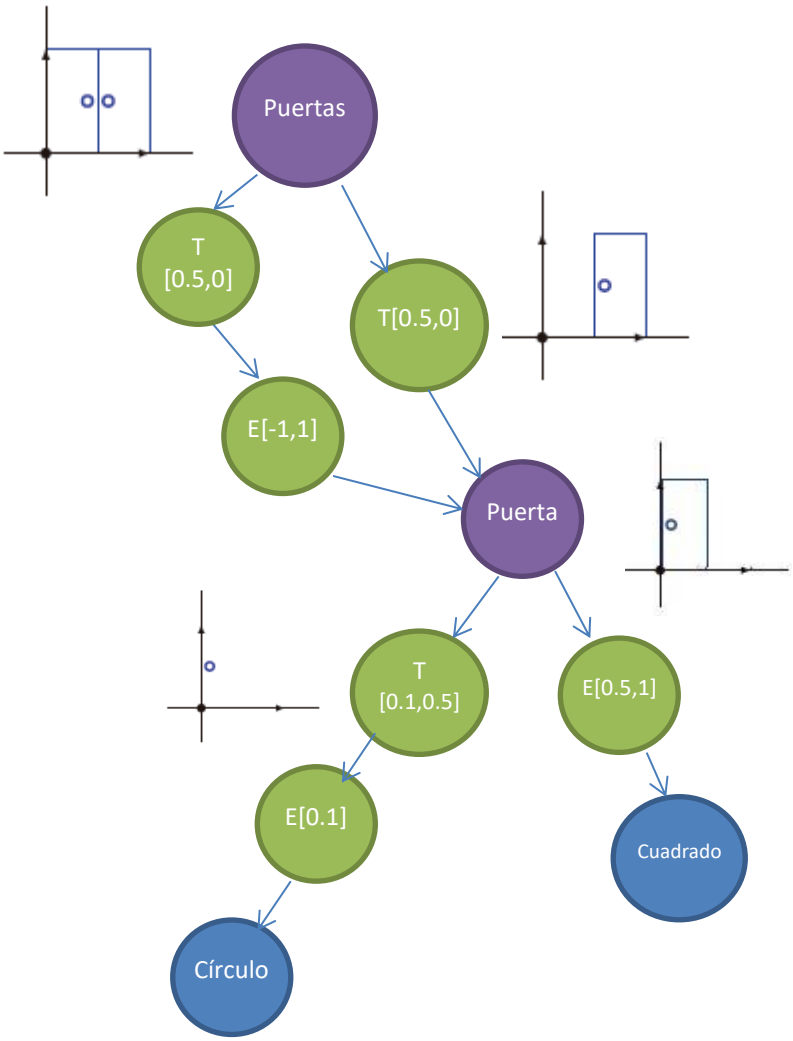


Ilustración 66: Grafo de escena de una puerta con dos hojas a partir de un círculo y un cuadrado.

En la Ilustración 66 se muestra el grafo de escena que permite construir una puerta con dos hojas, con sus correspondientes pomos a partir de un cuadrado y un círculo.

Se acompañan a los nodos unas ilustraciones de apoyo que muestran los pasos intermedios a partir de los cuales se va generando la imagen completa.

En la Ilustración 67 se sigue la misma metodología para construir una ventana a partir de dos cuadrados y un triángulo.

Finalmente, en la Ilustración 68 se muestra el grafo de escena de una fachada, sustituyendo subgrafos por los nodos raíz de las ilustraciones 66 y 67.

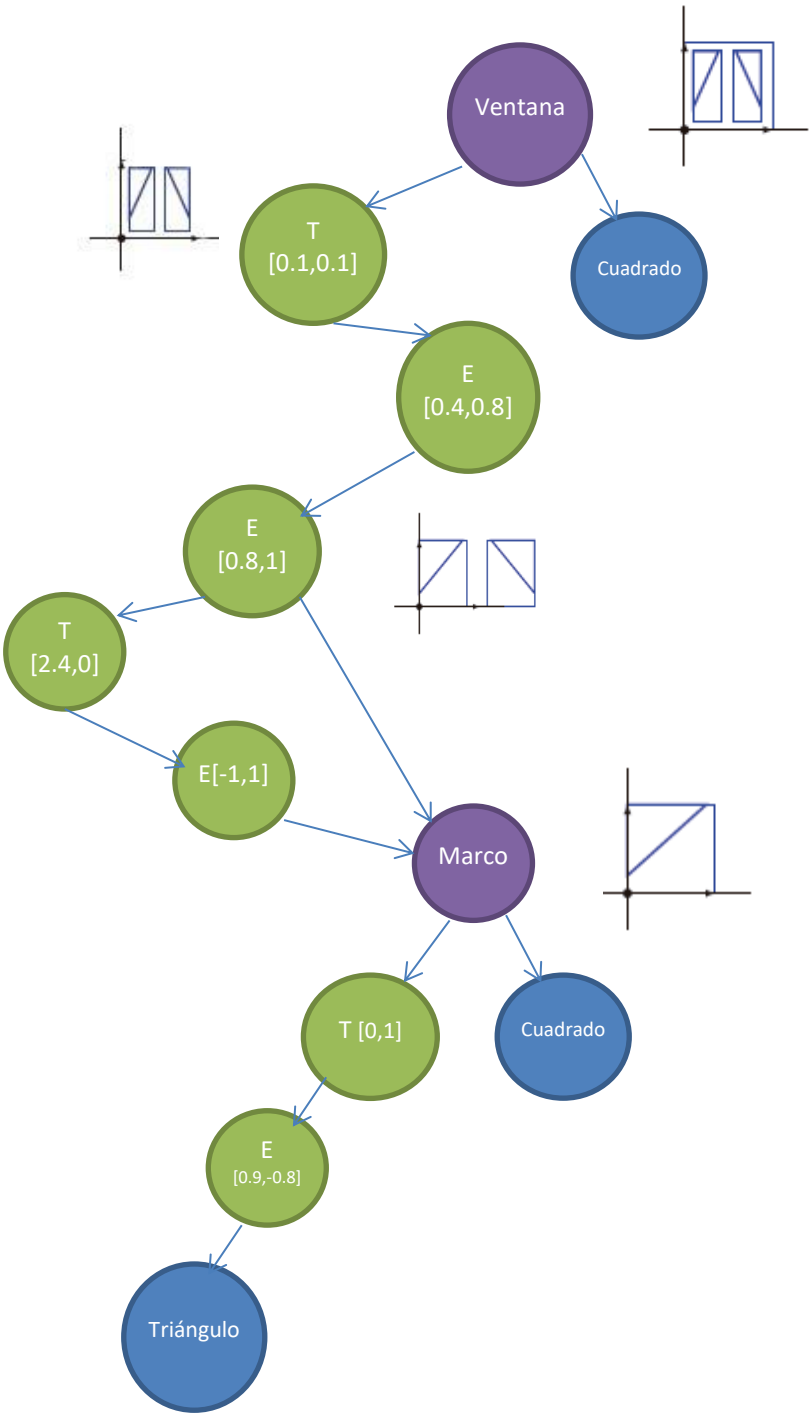


Ilustración 67: Grafo de escena de una ventana con dos hojas a partir de un triángulo y un cuadrado.

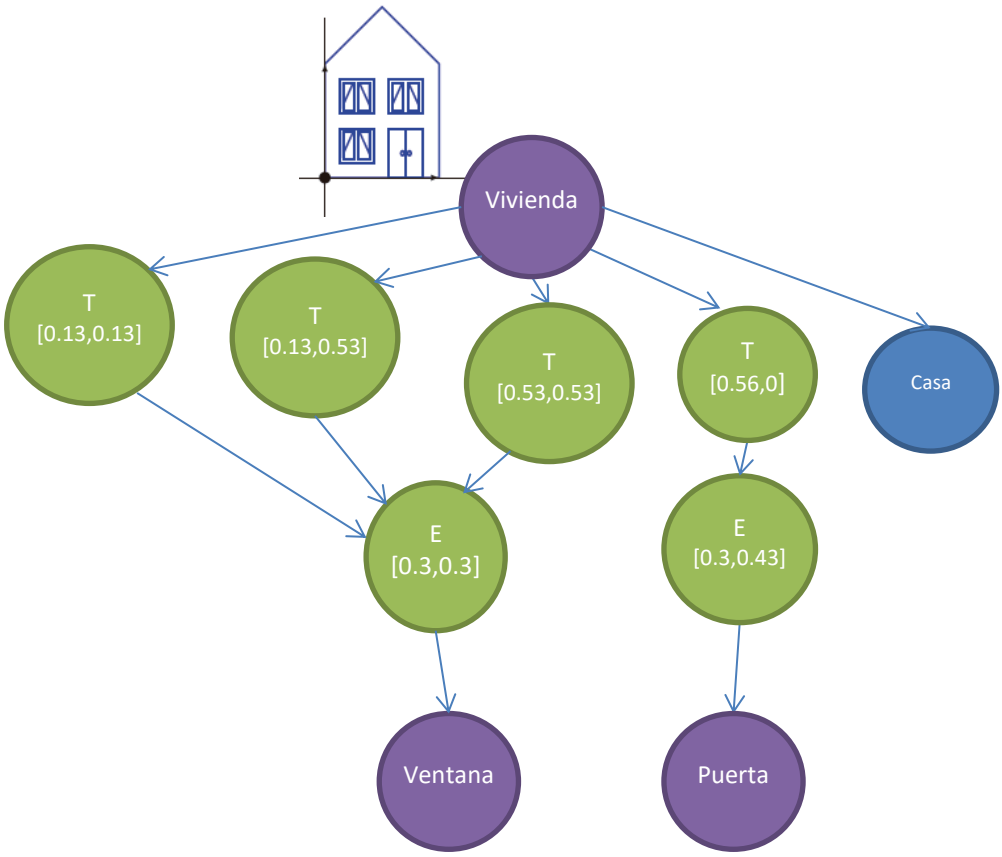


Ilustración 68: Grafo de escena de una vivienda a partir de la figura "casa", tres ventanas y una puerta.

Visualización de modelos jerárquicos en OpenGL

La visualización de modelos jerárquicos en OpenGL se basa en operaciones que permiten guardar y recuperar la matriz *modelview*. Se puede hacer de dos formas:

- realizando operaciones de multiplicación de matrices, como realmente ocurre en la tarjeta gráfica, y usar la matriz identidad para cada inicio de recorrido del grafo.
- usando el mecanismo de pila de OpenGL para recuperar versiones temporales de las matrices.

Supongamos el modelo jerárquico de la Ilustración 69:

- Para el objeto A usamos la matriz de modelado M=T1·T2
- Para el objeto B usamos la matriz de modelado M=T1·T3
- Para ambos, queremos usar una matriz de vista V (no dibujada en la ilustración)

Esta situación es típica cuando A y B están en distintas ramas de un subárbol del grafo de escena, como es el caso.

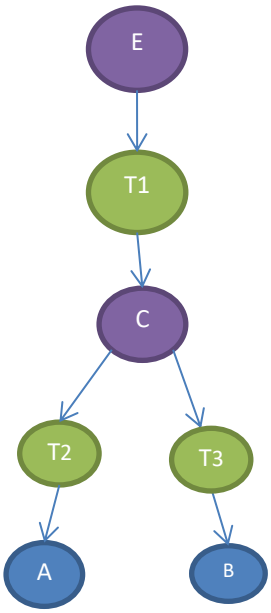


Ilustración 69: Modelo Jerárquico

Para visualizar con OpenGL el grafo de la Ilustración 69 será necesario usar el siguiente código:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity() ;   // M := Identidad
glMultMatrixf( V );  // M := M·V
glMultMatrixf( T1 ); // M := M·T1
glMultMatrixf( T2 ); // M := M·T2
A.dibujar();         // visualizar A con M == V·T1·T2

glLoadIdentity() ; // M := Identidad
glMultMatrixf( V ); // M := M·V
glMultMatrixf( T1 ); // M := M·T1
glMultMatrixf( T3 ); // M := M·T3
B.dibujar();         // visualizar B con M == V·T1·T3
```

Como podemos ver, por un lado estamos usando multiplicación de matrices, pero además es que es necesario repetir operaciones para conseguir la pila completa de visualización.

Otra forma alternativa es la siguiente:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity() ; // M := Identidad
glMultMatrixf( V ); // M := M·V
glMultMatrixf( T1 ); // M := M·T1
glPushMatrix() ; // C := M (guarda una copia de M en C)
      glMultMatrix( T2 ); // M := M·T2
      A.dibujar(); // visualizar A con M == V·T1·T2
glPopMatrix() ; // M := C (restaura copia de M)
glPushMatrix() ; // C := M (guarda una copia de M en C)
      glMultMatrix( T3 ); // M := M·T3
      B.dibujar(); // visualizar B con M == V·T1· T3
glPopMatrix() ; // M := C (restaura copia de M)
```

Internamente, OpenGL tiene un mecanismo de pilas para las transformaciones geométricas, que guarda en la pila el estado actual. Cuando se realiza un *glPushMatrix* lo que se hace es duplicar el tope de la pila, de forma que se almacenan dos copias de la matriz actual. La que queda en el tope es la única que se consulta y se ve afectada por las operaciones de transformación, mientras que la que hay debajo sólo será utilizada tras realizar un *glPopMatrix*

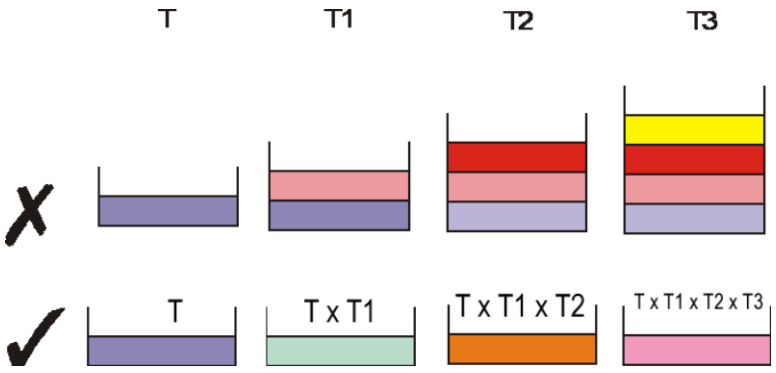


Ilustración 70: La pila de transformaciones cuando se realizan multiplicaciones de matrices no incrementa de tamaño.

El código entre *push* y *pop* es neutro con respecto a M, lo que quiere decir que tras hacer un *pop*, la matriz M vale exactamente lo que valía cuando se hizo *push.* Lógicamente los *push* y *pop* han de estar balanceados, a lo que puede ayudar realizar una indentación en el código que permita visualizar rápidamente los distintos niveles de la pila.

Un programa que visualice un grafo de escena puede implementarse trasladando dicho grafo a código de forma sencilla:

- Cada nodo es una secuencia de llamadas entre operaciones *push* y *pop*
- Una entrada correspondiente a un objeto simple supone una llamada al método *dibuja()* de dicha malla
- Una entrada correspondiente a un objeto complejo supone:
  - Una secuencia de llamadas correspondientes a dicho subgrafo, entre *push* y *pop*
  - Una llamada a un método *dibuja()* de dicho objeto que contendrá una secuencia de operaciones entre *push* y *pop.*

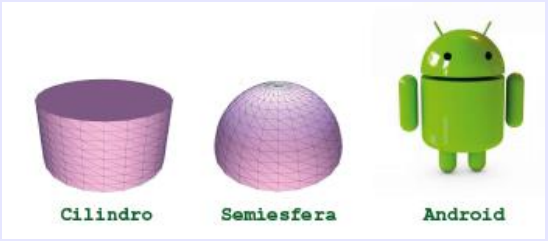
Por ejemplo, la fachada de la Ilustración 68 podría programarse como:

```
Vivienda::dibuja(){
        glPushMatrix();
              Casa.dibuja();
              glPushMatrix(); // instrucciones puerta
                    glTranslatef(0.56,0.0,0.0);
                    glScalef(0.3,43,1.0);
                    Puerta.dibuja();
              glPopMatrix();
              glTranslatef(0.13,0.13,0.0);
              miVentana.dibuja() ;
              glTranslatef(0.13,0.53,0.0);
              miVentana.dibuja() ;
              glTranslatef(0.53,0.53,0.0);
              miVentana.dibuja() ;
        glPopMatrix();
}

Ventana::dibuja(){
        glPushMatrix();
              glScalef(0.3,0.3,1.0);
              unaVentana.dibuja();
        glPopMatrix();
}
```

EJERCICIOS

51. Supón que dispones de dos objetos simples: Semiesfera y Cilindro, con su correspondiente función dibujar(). La semiesfera (en coordenadas maestras) tiene radio unidad, centro en el origen y el eje vertical en el eje Y. Igualmente el cilindro tiene radio y altura unidad, el centro de la base está en el origen, y su eje es el eje Y. Con estas dos primitivas queremos construir la figura símbolo de Android
    - Diseña el grafo de escena correspondiente, ten en cuenta que hay objetos compuestos que se pueden instanciar más de una vez (cada brazo o pierna se puede construir con un objeto compuesto de dos semiesferas en los extremos de un cilindro).
    - Escribe el código OpenGL para visualizarlo, usando transformaciones y push/pop de la matriz modelview.



Grafos parametrizables

Un grafo de escena puede tener nodos parametrizados, de forma que los valores de la transformación no sean necesariamente los mismos siempre. Por ejemplo:

- Controlar una transformación:
  - Ángulo de rotación
  - Factor de escala en una dimensión
  - Distancia de traslación en una dirección dada
- Controlar las dimensiones del objeto
- Otros valores, como los puntos de control de un objeto deformable.

De esta forma, un mismo grafo de escena parametrizado se traduce en objetos con distinta geometría para distintos valores concretos de los parámetros. Este número de parámetros denota los ***grados de libertad*** del modelo jerárquico.

Estos grados de libertad, estos modelos jerárquicos parametrizables, permiten la realización de partes móviles, como es el caso de la Ilustración 63, insertando un nodo transformación parametrizado para que modifique los elementos móviles.

EJERCICIOS

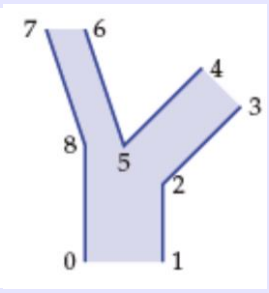
52. Escribe una segunda versión del grafo de escena del ejercicio anterior, de forma que las transformaciones estén parametrizadas por dos valores reales (α y β) que expresan el ángulo de rotación del brazo izquierdo y derecho (respectivamente), en torno al eje que pasa por los centros de las dos semiesferas superiores de los brazos. Asimismo, habrá otro parámetro (ϕ) que es el ángulo de rotación de la cabeza (completa: con los ojos y antenas) entorno al eje vertical que pasa por su centro (cuando estos ángulo valen 0, el androide está en reposo y tiene exactamente la forma de la figura del problema anterior).
    - Escribe el método `Androide::dibuja(float a, float b, float c)` para visualizar el androide parametrizado de esta forma. El método Androide::dibuja() debe aceptar como parámetros los tres ángulos.
53. Supón que dispones de una clase llamada `figuraSimple` cuyo método `dibuja()` pinta con OpenGL la figura que aparece a la izquierda (un cuadrado de lado unidad con la esquina inferior izquierda en el origen, con un triángulo inscrito). Esta función llama a `glVertex` para especificar las posiciones de los vértices de los polígonos de dicha figura. Usando exclusivamente llamadas a dicha función, construye otra clase `figuraCompleja` con un método `dibuja()` que pinta la figura de la derecha. Para lograrlo puedes usar manipulación de la pila de la matriz *modelview* (`glPushMatrix` y `glPopMatrix`), junto con `glTranslatef` y `glScalef`.

EJERCICIOS

54. Crea un método `dibujaRec()` en la clase `figuraCompleja` que dibuje la figura siguiente. Dicho método contendrá una única llamada a `figuraSimple::dibuja()`, que se ejecutará 15 veces.

55. Escribe el código OpenGL de un método Tronco::dibuja() que pinte la figura que aparece a aquí. El código dibujará el polígono relleno de color azul claro, y las aristas que aparecen de color azul oscuro (ten en cuenta que no todas las aristas del polígono relleno aparecen).

Índice de vértice	Coordenadas de vértice
0	(+0.0, +0.0)
1	(+1.0, +0.0)
2	(+1.0, +1.0)
3	(+2.0, +2.0)
4	(+1.5, +2.5)
5	(+0.5, +1.5)
6	(+0.0, +3.0)
7	(−0.5, +3.0)
8	(+0.0, +1.5)



56. Escribe el método `Arbol::dibuja()` que pinte la figura siguiente, mediante llamadas al método `Tronco::dibuja()`. En este árbol, cada tronco conecta dos copias del mismo más pequeñas, situadas en sus ramas. El tronco raíz es idéntico al del ejercicio 60. En total hay 63 troncos de 6 niveles o tamaños distintos. Diseña el método usando recursividad, de forma que el número de niveles sea un parámetro del método.

BIBLIOGRAFÍA

[Foley97] **Computer Graphics: Principles and Practice in C**; Foley, Van Dam, Feiner y Hughes. de. Addison-Wesley , 1997.

[Shirley09] **Fundamentals of Computer Graphics;** P. Shirley. AK Peters, 2009

[Angel08] **Interactive Computer Graphics: A Top Down Approach** (5ª Ed); E. Angel. Addison Wesley, 2008

[Hearn10] **Computer Graphics with Open GL** (4ª Ed) ; D. Hearn, P. Baker, W. Carithers; Prentice Hall, 2010

Recursos online (principalmente cursos en otras universidades):

- http://www.xmission.com/~nate/tutors.html
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-837-computer-graphics-fall-2003/
- http://www.student.cs.uwaterloo.ca/~cs488/
- http://www4.ujaen.es/~demiras/igv_app
- http://www.opengl-tutorial.org/