Informática Gráfica:
Teoría. Tema 2. Modelado de Objetos.

Carlos Ureña

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

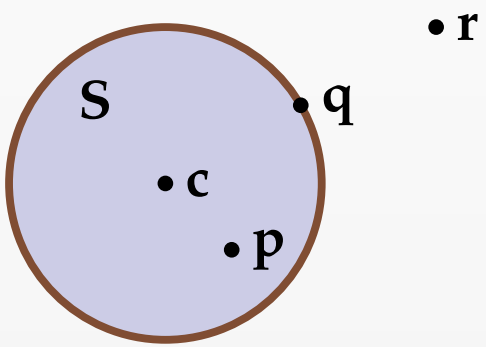
2018-19

Teoría. Tema 2. Modelado de Objetos.
Índice.

Sección 1
Modelos geométricos. Introducción.

Modelos geométricos formales

Un **modelo geométrico** es un modelo matemático que sirve para representar un objeto geométrico que existe en el espacio 3D o 2D. Los modelos geométricos matemáticos más generales posibles son los **subconjuntos de puntos** del espacio $\mathbf{E}$, por ejemplo, una esfera:



$$\mathbf{S} = \{ \mathbf{x} \in \mathbf{E} \text{ t.q. } \|\mathbf{x} - \mathbf{c}\| \leq 1 \}$$
$$\mathbf{p} \in \mathbf{S}$$
$$\mathbf{q} \in \mathbf{S} \quad \mathbf{q} \in \partial\mathbf{S}$$
$$\mathbf{r} \notin \mathbf{S}$$

Cada subconjunto o **región S** es:

- **cerrado** (incluye a su propia **superficie** o **frontera**, $\partial\mathbf{S}$),
- **acotado** (no tiene extensión infinita),
- su superficie es diferenciable (**plana** a escala muy pequeña)

Modelos computacionales

El modelo basado en subconjuntos de puntos del espacio:

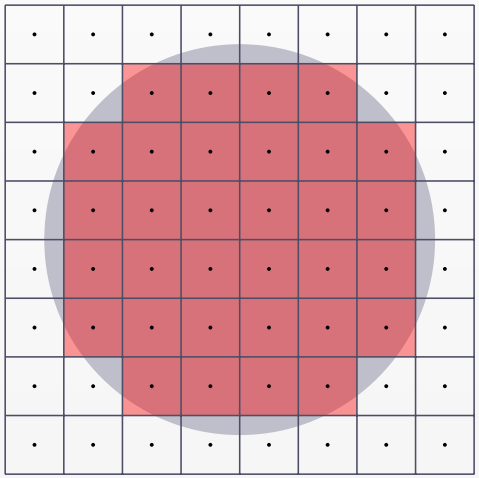
- permite representar matemáticamente cualquier objeto (es el modelo **más general** posible)
- pero **no se puede representar en la memoria** (finita, discreta) de un ordenador

Ante esto hay representaciones aproximadas pero que usan una cantidad finita de memoria (**modelos geométricos computacionales**), se usan básicamente dos opciones :

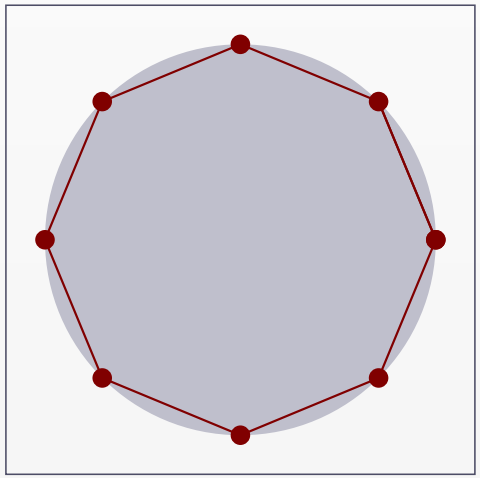
- **enumeración espacial:** para un conjunto grande pero finito de pequeños volumenes (**voxels**), se indica cuales de ellos estan dentro del volumen del objeto y cuales no.
- **modelos de fronteras:** se representa la frontera (la superficie) en lugar de todo el interior, para ello se usan conjuntos finitos de polígonos planos pequeños (**caras**)

Ejemplo 2D de los modelos aproximados

Las dos formas computacionales de modelar son aproximadamente iguales al modelo matemático basado en un conjunto, con un error que disminuye al aumentar la cantidad de memoria usada (la precisión o resolución).



Enumeración espacial



Modelos de fronteras

- Modelos de fronteras: usados en la mayoría de las aplicaciones.
- Enumeración espacial: muy útiles en aplicaciones específicas

Modelos de fronteras 3D

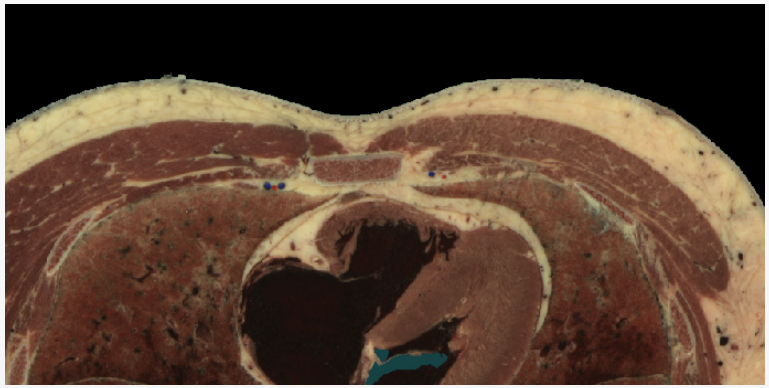
Las **mallas de polígonos** se usan en la mayoría de las aplicaciones:

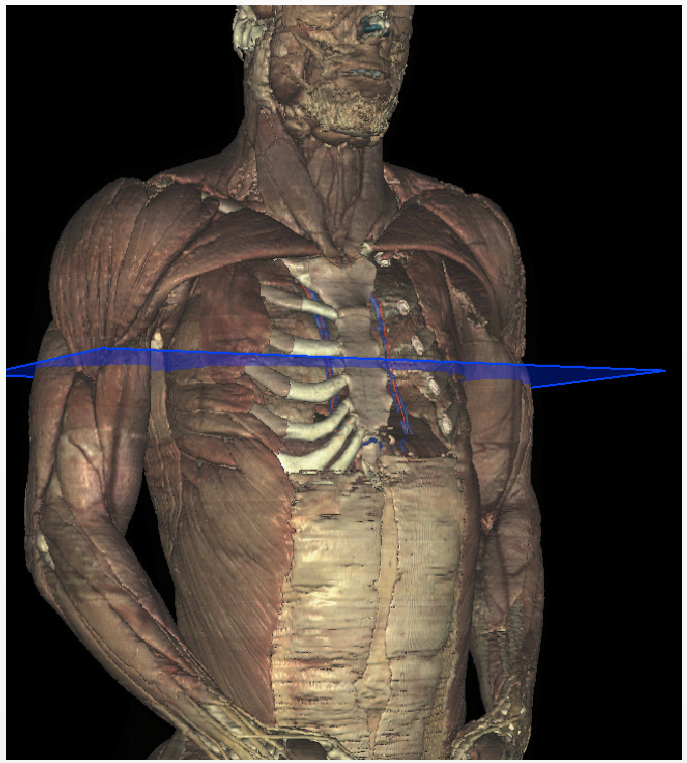
(aunque se use enumeración espacial, se suele generar una malla de polígonos como paso previo a la visualización por rasterización)

Imagen obtenida de: ☞ http://www.renderspirit.com/tutorials/wireframe/

Enumeración espacial 3D

Las **modelos volumétricos** se usan en aplicaciones donde interesa todo el volumen del objeto (p.ej. en la **Tomografía Axial Computerizada**, TAC, para Medicina y Arqueología, o en Geología y Climatología)





Captura de pantalla del software *VH Dissector* de Toltech:

☞ http://www.toltech.net/anatomy-software/solutions/vh-dissector-for-medical-education

Sección 2
Modelos de fronteras. Mallas de polígonos.

2.1. Introducción: elementos y adyacencia.
2.2. Lista de triángulos aislados (TA).
2.3. Mallas como tiras de triángulos (TT).
2.4. Mallas indexadas.
2.5. Representación con estructura de aristas aladas
2.6. Representación y visualización de atributos
2.7. Visualización de mallas en modo diferido

Subsección 2.1
Introducción: elementos y adyacencia.

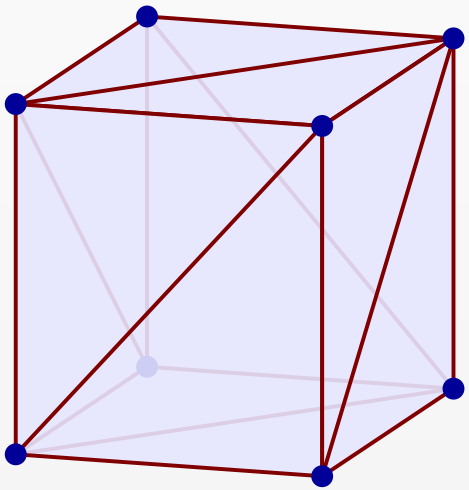
Mallas de polígonos y sus elementos.

Los modelos de fronteras se representan usando **mallas de polígonos** (*polygon meshes*). Básicamente, una malla es un conjunto de **caras** (*faces*) planas (polígonos) que constituyen la frontera de un objeto. En el espacio 3D, las mallas de polígonos:

- permiten modelar la superficie (cerrada) de un objeto 3D con volumen, o bien una superficie bidimensional aislada (una *variedad bidimensional*).
- los segmentos de recta entre dos caras adyacentes se llaman **aristas** (*edges*).
- los puntos extremos de las aristas se llaman **vértices** (*vertices or vertexes*).
- una **escena** se compone de una o más mallas.
- las caras pueden ser polígonos de 3 vértices o más, **todos ellos están en el mismo plano**.
- los más usual es considerar únicamente **mallas de triángulos**

Identificación y relaciones entre elementos.

Todos los elementos (vértices, aristas y caras) tienen entre ellos una relación binaria de adyacencia (culquier par de elementos, del mismo tipo o distintos, pueden ser adyacentes o no)



- Cada vértice se identifica por posición $\mathbf{p}_i$ (un punto en el espacio).
- Cada arista se identifica por los dos vértices de sus extremos (adyacentes a la arista).
- Cada cara se identifica por sus vértices o por sus aristas (adyacentes a la cara).

Marco de referencia de la malla.

La posición de cada vértice se identifica a su vez por sus coordenadas respecto de un marco de referencia cartesiano $\mathcal{R}$ único

- A dicho marco de referencia se le denomina **marco de referencia local de la malla**
- El i-ésimo vértice tiene como posición $\mathbf{p}_i$ y sus coordenadas son $\vec{c}_i = (x_i, y_i, z_i, 1)$, es decir:

$$\mathbf{p}_i \;=\; \mathcal{R}\,\vec{c}_i \;=\; \mathcal{R}\,(\,x_i\,,\,y_i\,,\,z_i\,,1\,)^T$$

- A la tupla $\vec{c}_i$ se le denomina las **coordenadas locales** del vértice i-ésimo.
- No se suele almacenar en memoria la componente w ya que siempre es 1, aunque a veces se podría hacer para acortar algo el tiempo de procesamiento (a costa de usar más memoria).

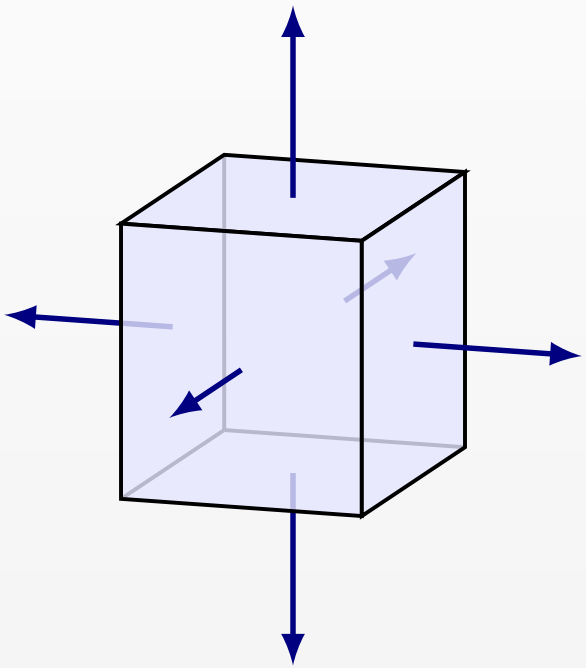
Atributos de las mallas

Como modelos de objetos reales, las mallas suelen incluir más información geométrica o del aspecto del objeto:

- **Normales** (*normals*): vectores de longitud unidad
  - **normales de caras**: vector unitario perpendicular a cada cara, de longitud unidad, apuntando al exterior de la malla si es una malla cerrada. Precalculado a partir del polígono.
  - **normales de vértices**: vector unitario perpendicular al plano tangente a la superficie en la posición del vértice.
- **Colores**: ternas (usualmente RGB) con tres valores entre 0 y 1.
  - **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 (la superficie varía de color de forma continua entre vértices).
- otros atributos (p.ej.: coordenadas de textura o vectores tangente y bitangente en cada vértice)

Normales de caras

Pueden ser útiles cuando el objeto que se modela con la malla está realmente compuesto de caras planas (p.ej., un cubo).



Para un polígono (con dos aristas $\mathbf{e}_i$,$\mathbf{e}_j$, vectores distintos, no nulos), su normal $\mathbf{n}$ se calcula como:

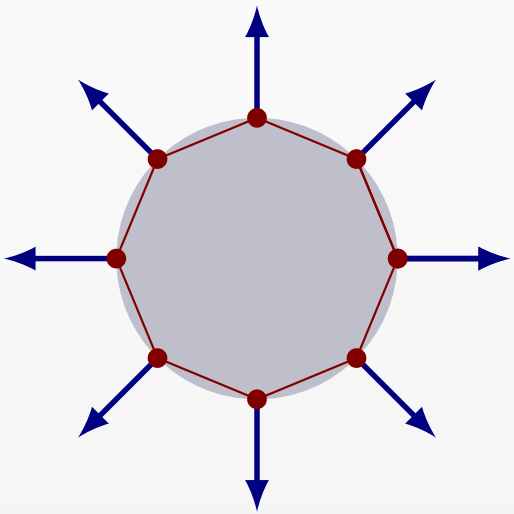
$$\mathbf{n} = \frac{\mathbf{m}}{\|\mathbf{m}\|} \quad \text{donde } \mathbf{m} = \mathbf{e}_i \times \mathbf{e}_j$$

(con $i \neq j$)

En estos casos la normal se puede precalcular y almacenar en la malla para lograr eficiencia en tiempo de visualización.

Normales de vértices

Si se aproxima un objeto curvo, las normales de vértices se puede conocer *a priori* (p.ej. en una esfera) o bien calcularse promediando las normales de las caras adyacentes:



Para un vértice (con k caras adyacentes) su normal $\mathbf{n}$ se calcula como:

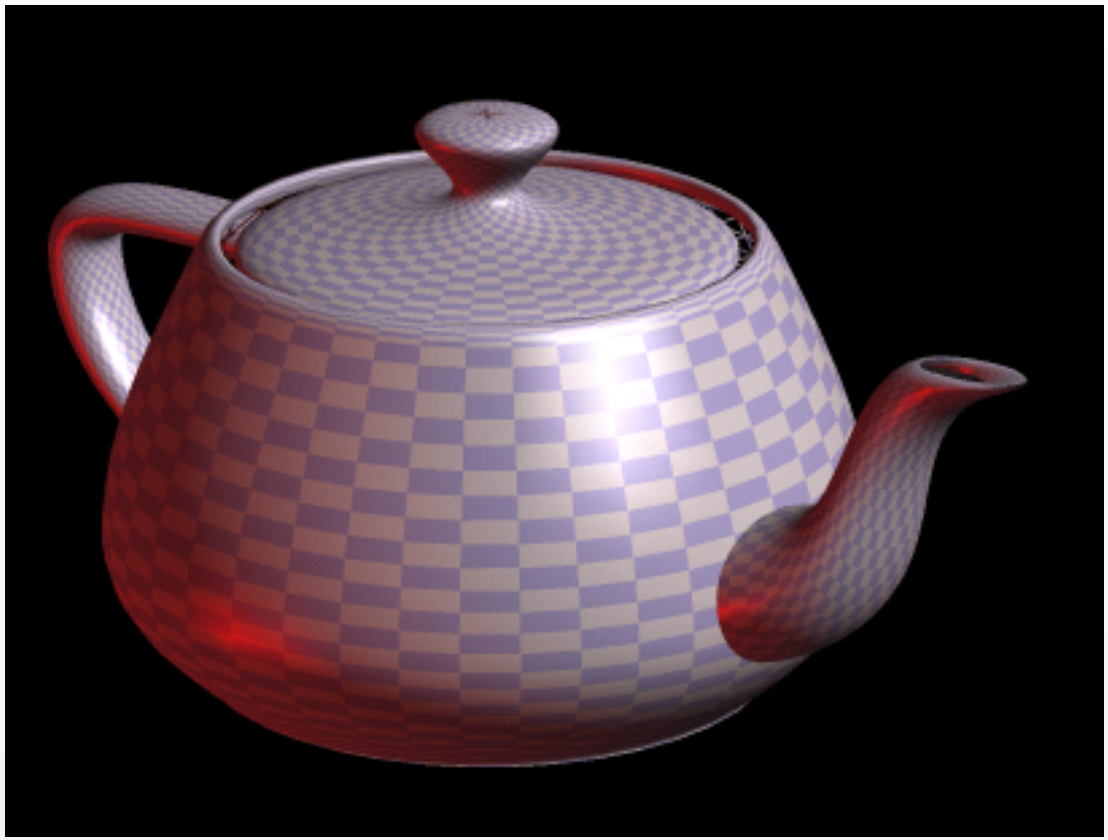
$$\mathbf{n} = \frac{\mathbf{s}}{\|\mathbf{s}\|} \quad \text{donde } \mathbf{s} = \sum_{i=1}^{k} \mathbf{m}_k$$

donde $\mathbf{m}_1, \mathbf{m}_2, \ldots \mathbf{m}_k$ son las normales de las caras adyacentes.

Estas normales también se pueden precalcular y almacenar.

Colores de caras

A veces es deseable asignar un color distinto a cada cara



Colores de vértices

En algunos casos, es conveniente asignar colores RGB a los vértices. La utilidad más frecuente de esto es hacer interpolación de color en las caras durante la visualización:

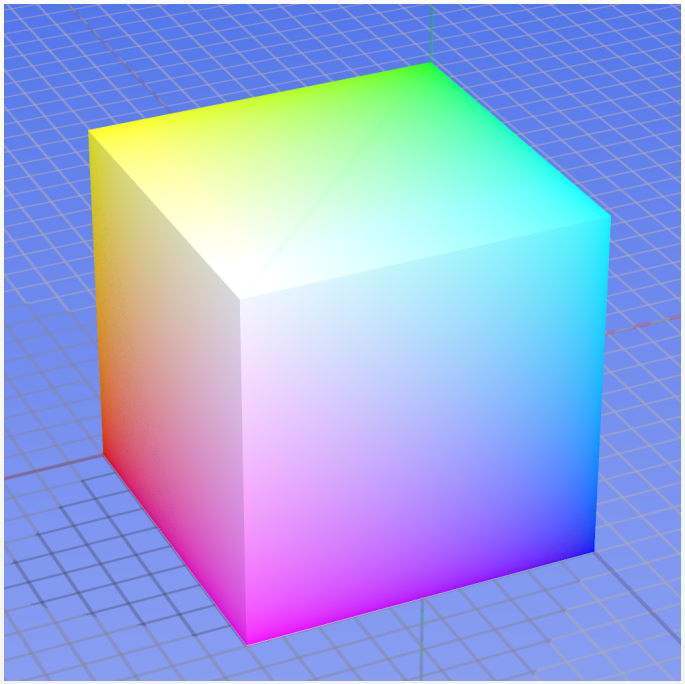


Imagen: ☞ http://en.wikipedia.org/wiki/File:RGB_color_solid_cube.png
(Wikimedia Commons)

Subsección 2.2
Lista de triángulos aislados (TA).

Tabla de triángulos aislados.

La más simple es usar una lista o tabla de **triángulos aislados**. La malla se representa como un vector o lista con una entrada o nodo para cada uno de los n triángulos:

Malla TA (n triángulos)

0	x_0	y_0	z_0
1	x_1	y_1	z_1
2	x_2	y_2	z_2
3	x_3	y_3	z_3
4	x_4	y_4	z_4
5	x_5	y_5	z_5
⋮	⋮		
$3n-3$	x_{3n-3}	y_{3n-3}	z_{3n-3}
$3n-2$	x_{3n-2}	y_{3n-2}	z_{3n-2}
$3n-1$	x_{3n-1}	y_{3n-1}	z_{3n-1}

- Para cada triángulo se almacenan las coordenadas locales de cada uno de sus tres vértices (9 valores flotantes en total).
- La tabla se puede almacenar en memoria con todas las coordenadas contiguas.
- En total, incluye $9n$ valores flotantes.

Valoración.

Esta representación consume muchísima más memoria de la necesaria:

- Si un vértice es adyacente a k triángulos, sus coordenadas aparecen repetidas k veces en la tabla.
- En una malla típica que representa una rejilla de triángulos, los vértices internos (la mayoría) son adyacentes a 6 triángulos, es decir, cada tupla aparece casi 6 veces como media.

Hay una falta de información explícita sobre la conectividad o adyacencia entre elementos (la **topología** de la malla)

- En algunas aplicaciones, esta información es esencial.
- Se puede calcular comparando coordenadas de vértices, pero tendría una complejidad en tiempo cuadrática con el número de vértices.

En algunos casos muy particulares, podría ser útil (mallas con muchos triángulos realmente aislados).

Implementación de mallas: clase base abstracta.

Cualquier objeto que se pueda visualizar se implementará usando una clase concreta derivada de la clase **Objeto3D**, que se define como:

```
class Objeto3D
{
   public:
      virtual void visualizarGL( ContextoVis & cv ) = 0 ;
} ;
```

- Cualquier tipo de objeto que se pueda dibujar con OpenGL llevará asociada una clase concreta que implementará una versión concreta del método **visualizarGL**
- **ContextoVis** es una clase con información de contexto sobre la visualización. Contendrá el modo de visualización:

```
class ContextoVis
{  public:
      unsigned modo_vis ;  // modo de visualización (alambre, solido,...)
      //..... // otros datos.
} ;
```

Implementación de mallas de triángulos aislados

Como ejemplo, podemos usar una clase (**MallaTA**), que contiene el número de triángulos, y un puntero a las coordenadas (contiguas) en memoria. En total, para n_t triángulos, $9n_t$ valores reales:

```
// Coordenadas de los 3 vértices de un triángulo
struct CVerTri
{
   Tupla3f ver[3] ;           // vértices 0,1 y 2
} ;
// Malla de triángulos aislados:
class MallaTA  : public Objeto3D
{
   protected:
      unsigned long          num_tri   ;  // número total de triángulos (n_t)
      std::vector<CVerTri>   tri       ;  // tabla de triángulos (num_tri elementos)
      void visualizarBE() ;                // visualizar (Begin/End, neutro)
      void visualizarDA() ;                // visualizar (DrawArrays, neutro)
   public:
      virtual void visualizarGL( ContextoVis & cv ) ;
} ;
```

Visualización en modo inmediato (begin/end)

Para visualiar en modo inmediato podemos usar **glBegin**/**glEnd**, según vimos en el tema 1:

```
void MallaTA::visualizarBE(  )
{
   glBegin( GL_TRIANGLES );
      for( unsigned i = 0 ; i < num_tri ; i++ )
         for( unsigned j = 0 ; j < 3 ; j++ )
            // enviar coordenadas del vértice j del triángulo i:
            glVertex3fv( tri[i].ver[j] );
   glEnd();
}
```

Inconvenientes:

- Múltiples llamadas a **glVertex**
- Transferencia de coordenadas por el bus del sistema hacia la GPU
- Procesamiento repetido de las mismas coordenadas varias veces en la GPU

Visualización en modo inmediato (*Vertex Arrays*)

La malla completa se puede visualizar con una única llamada a **glDrawArrays** y pasándole el array de coordenadas (***Vertex Array***). Se debe indicar el puntero a una tabla con $3n$ tuplas de coordenadas de vértices contiguas:

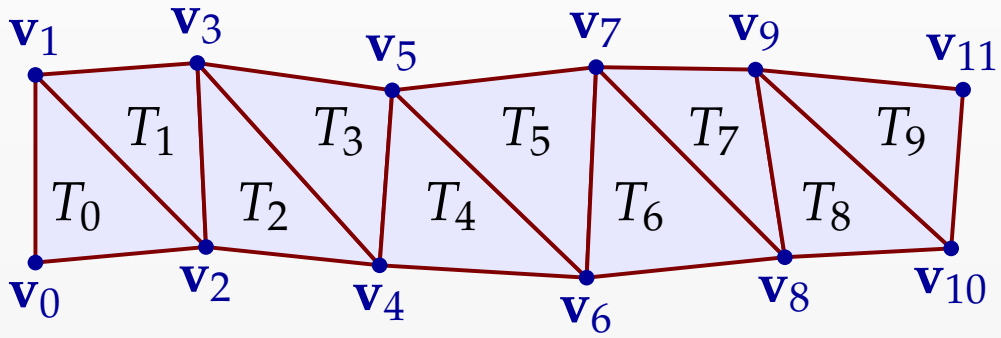
```
void MallaTA::visualizarDA(  )
{
    // habilitar el uso de puntero a tabla de coordenadas
    glEnableClientState( GL_VERTEX_ARRAY );
    // proporcionar el puntero (tri) la tabla de coordenadas de vv.
    // parámetros: (1) long. tuplas (2) tipo de reales (3) separación entre tuplas (4) puntero a tabla
    glVertexPointer( 3, GL_FLOAT, 0, tri.data() );
    // visualización de la malla. parámetros: (1) tipo prim., (2) índice de inicio, (3) núm. prim.
    glDrawArrays( GL_TRIANGLES, 0, num_tri ) ;
    // deshabilitar el puntero a la tabla de vértices
    glDisableClientState( GL_VERTEX_ARRAY );
}
```

Requiere transferencia y procesamiento repetidos (una vez por frame) de coordenadas por el bus del sistema hacia la GPU.

Subsección 2.3
Mallas como tiras de triángulos (TT).

Tiras de triángulos.

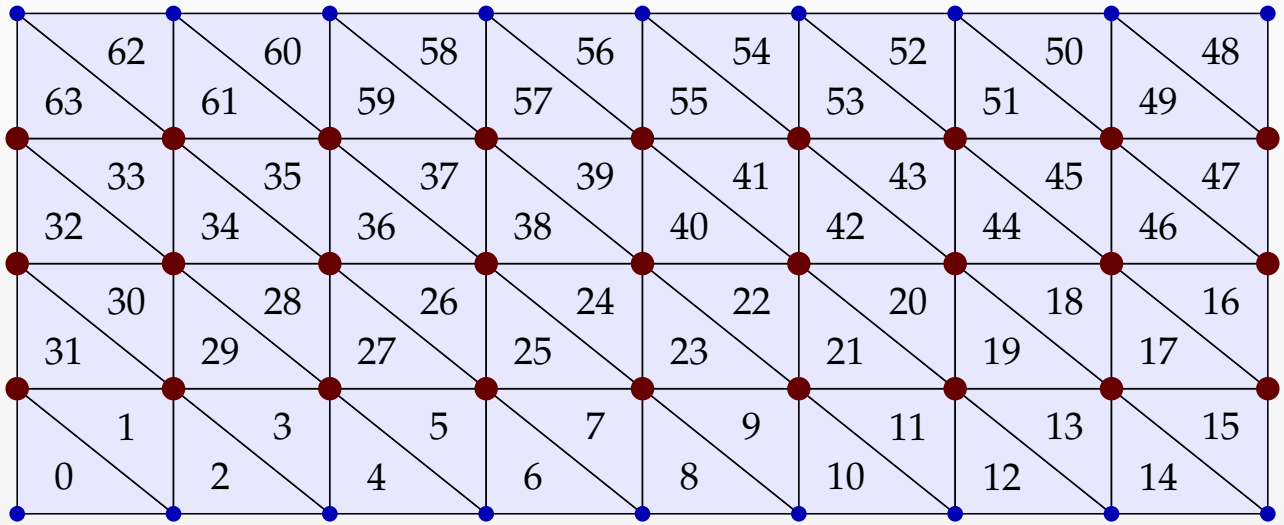
Cada triángulo T_{i+1} en la secuencia es adyacente al anterior T_i, con lo cual T_{i+1} comparte con T_i una arista y dos vértices, vértices cuyas coordenadas no tienen que ser repetidas en memoria:



- Cada tira de n triángulos necesita $n+2$ tuplas de coordenadas de vértices (tres para el primer triángulo y después una más por cada triángulo adicional)
- Se almacena una tabla que en la i-ésima entrada almacena las coordenadas del i-ésimo vértice.

Tiras de triángulos para mallas no simples

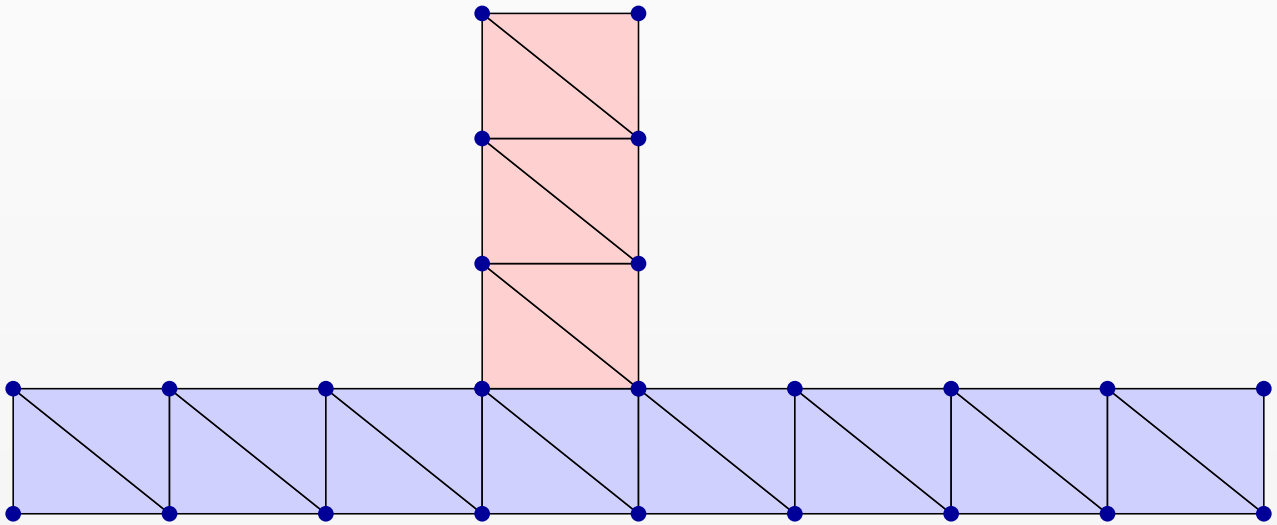
En la mayoría de los casos, las tiras obligan a repetir algunas coordenadas de vértices (aunque en mucho menor grado que los triángulos aislados). Ejemplo de una tira en zig-zag:



- las coords. de los vértices en rojo (grandes) se repiten dos veces.
- las de los vértices en azul (pequeños) aparecen una sola vez.

Mallas con varias tiras

En algunos casos, es inevitable tener que recurrir a más de una tira para una única malla:



Por este motivo la implementación de una malla con tiras debe preveer más de una tira en la misma malla.

Representación en memoria

Una malla es una estructura con varias tiras. La tira número i (con n_i triángulos) es un array con $n_i + 2$ celdas, en cada una están las coordenadas maestras de un vértice.

Tira 0 (n_0 triángulos)

x_0	y_0	z_0
x_1	y_1	z_1
x_2	y_2	z_2
x_3	y_3	z_3
x_4	y_4	z_4
$\vdots$	$\vdots$	$\vdots$
x_{n_0}	y_{n_0}	z_{n_0}
x_{n_0+1}	y_{n_0+1}	z_{n_0+1}
x_{n_0+2}	y_{n_0+2}	z_{n_0+2}

Tira 1 (n_1 triángulos)

x_0	y_0	z_0
x_1	y_1	z_1
x_2	y_2	z_2
x_3	y_3	z_3
x_4	y_4	z_4
$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$
x_{n_1}	y_{n_1}	z_{n_1}
x_{n_1+1}	y_{n_1+1}	z_{n_1+1}
x_{n_1+2}	y_{n_1+2}	z_{n_1+2}

Tira 2 (n_2 triángulos)

x_0	y_0	z_0
$\vdots$	$\vdots$	$\vdots$
x_{n_2+2}	y_{n_2+2}	z_{n_2+2}

Tiras de triángulos: valoración

La mejora de las tiras frente a los triángulos aislados es que usan menos memoria, sin embargo, tiene estos inconvenientes:

- Al requerir probablemente más de una tira de triángulos, la representación es algo más compleja.
- Se necesitan algoritmos (complejos) para calcular las tiras a partir de una malla representada de alguna otra forma. Se intenta optimizar de forma que el número de coordenadas a almacenar sea el menor posible.
- El numero promedio de veces que se repite cada coordenada en memoria es prácticamente siempre superior a la 1, y cercano a 2.
- Esta representación tampoco incorpora información explícita sobre la conectividad.

Implementación

Una malla hecha de tiras de triángulos se puede representar con una estructura similar a **MallaTT**, para cada tira hay una instancia de **TiraTriangulos**

```
// Tira de triángulos (coords. de vértices)
struct TiraTri
{
   unsigned long         num_tri ;   // número de triángulos en esta tira
   std::vector<Tupla3f> ver      ;   // vector de coords. de vert. (num_tri+2 entradas)
} ;
// Malla compuesta de tiras de triángulos
class MallaTT : public Objeto3D
{
   protected:
      unsigned long           num_tiras ;  // numero total de tiras de triángulos
      std::vector<TiraTri>    tiras      ;  // vector de tiras (num_tiras entradas)
      void visualizarBE() ;                 // visualizar (Begin/End, neutro)
      void visualizarDA() ;                 // visualizar (DrawArrays, neutro)
   public:
      virtual void visualizarGL( ContextoVis & cv );
} ;
```

Visualización en modo inmediato

Una malla se podría visualizar en modo inmediato mediante begin/end o mediante un *vertex array*:

```
void MallaTT::visualizarBE( )
{
   for( unsigned i = 0 ; i < num_tiras ; i++ )
   {  glBegin( GL_TRIANGLE_STRIP ) ;
         for( unsigned long j = 0 ; j < tiras[i].num_tri+2 ; j++ )
            // enviar coordenadas del vértice j de la tira i:
            glVertex3fv( tiras[i].ver[j] ) ;
      glEnd() ;
   }
} ;

void MallaTT::visualizarDA(  )
{
   glEnableClientState( GL_VERTEX_ARRAY );
   for( unsigned long i = 0 ; i < num_tiras ; i++ )
   {  glVertexPointer( 3, GL_FLOAT, 0, tiras[i].ver.data() );
      glDrawArrays( GL_TRIANGLE_STRIP, 0, tiras[i].num_tri + 2 ) ;
   }
   glDisableClientState( GL_VERTEX_ARRAY );
} ;
```

Subsección 2.4
Mallas indexadas.

Mallas Indexadas.

Para solucionar los problemas de uso de memoria y tiempo de procesamiento de las soluciones anteriores, se puede usar una estructura con dos tablas:

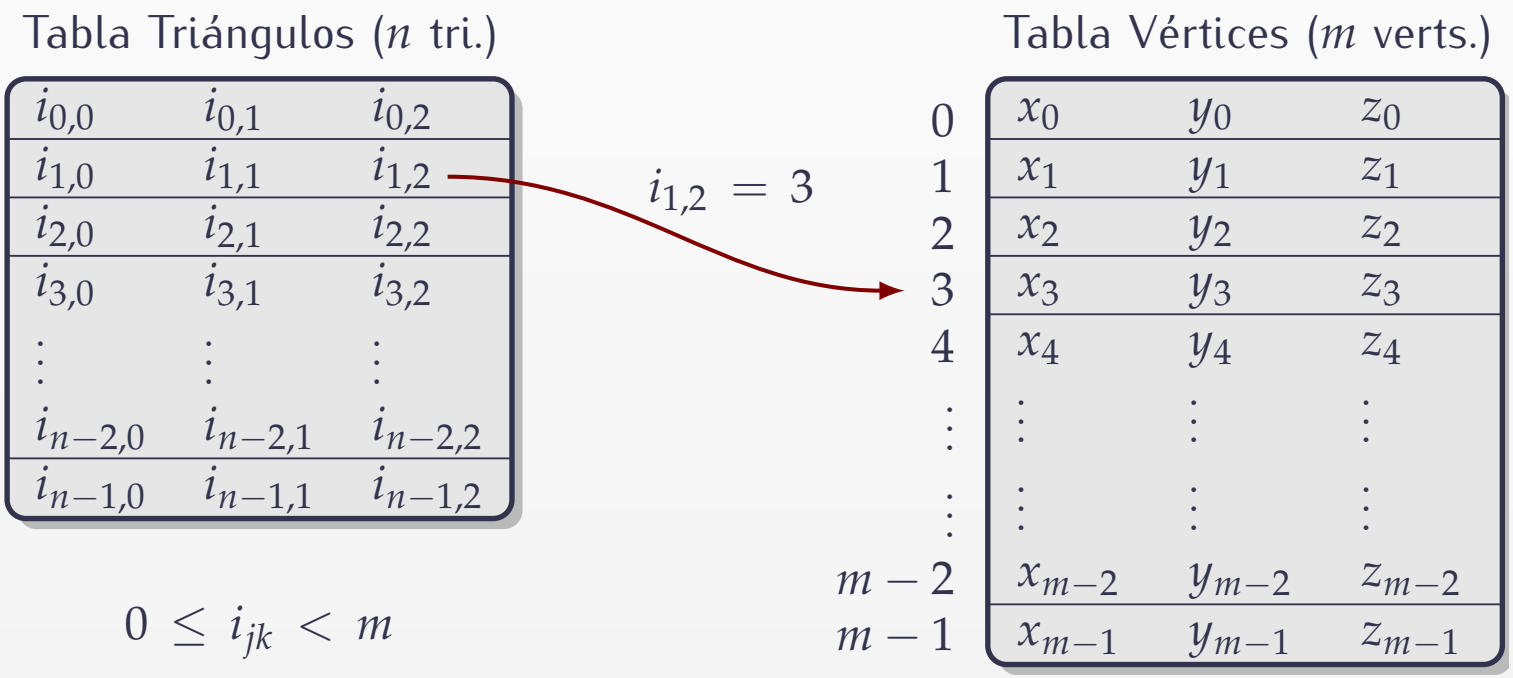
- **Tabla de vértices**: tiene una entrada por cada vértice, incluye sus coordenadas
- **Tabla de triángulos**: tiene una entrada por triángulo, incluye los índices de sus tres vértices en la tabla anterior.

En esta solución:

- No se repiten coordenadas de vértices: se ahorra memoria y se puede visualizar sin repetir cálculos (se repiten índices enteros).
- Hay información explícita de la topología (conectividad): se almacenan explícitamente los vértice adyacentes a un triángulo y se pueden calcular fácilmente el resto de adyacencias.

Estructura de datos

La tabla de triángulos (para n triángulos), almacena un total de $3n$ índices de vértices (enteros sin signo), y la de vértices $3m$ valores reales:



Implementación de las mallas indexadas

Usaremos una clase de nombre **MallaInd**:

```
class MallaInd :  public Objeto3D
{
   protected:
     unsigned   num_ver         ;  // número de vértices (n_v)
     unsigned   num_tri         ;  // número de triángulos (n_t)
     std::vector<Tupla3f> ver ;  // tabla de vértices (num_ver entradas)
     std::vector<Tupla3i> tri ;  // tabla de triángulos (num_tri entradas)
     void visualizarBE() ;          // visualizar (Begin/End, neutro)
     void visualizarDE() ;          // visualizar (DrawElements, neutro)
     .......
   public:
     virtual void visualizarGL( ContextoVis & cv ) ;
} ;
```

Incluye un total de $3n_v$ valores reales contiguos en memoria (coordenadas de vértices), y $3n_t$ valores naturales (índices de vértices), también contiguos.

Visualización en modo inmediato (begin/end)

Si se usa **glVertex**, entonces es necesario enviar a la GPU varias veces cada tuplas de coordenadas, que es también procesada varias veces:

```
void MallaInd::visualizarBE(  )
{
   glBegin( GL_TRIANGLES ) ;
      for( unsigned i = 0 ; i < num_tri ; i++ )
         for( unsigned j = 0 ; j < 3 ; j++ )
         {  // leer el índice del vérice j del triángulo i
            unsigned ind_ver = tri[i](j) ;   // ind_ver < num_ver
            // leer y enviar coordenadas del vértice
            glVertex3fv( ver[ind_ver] ) ;
         }
      }
   glEnd() ;
}
```

Esta solución, por tanto, es **muy ineficiente** en tiempo.

Vis. en modo inmediato (*vertex array*)

Para evitar el problema descrito, se pueden enviar todas las coordenadas de vértices y los índices con una sola llamada a **`glDrawElements`**:

```
void MallaInd::visualizarDE(  )
{
    // habilitar uso de un array de vértices
   glEnableClientState( GL_VERTEX_ARRAY );
    // especificar puntero a tabla de coords. de vértices
   glVertexPointer( 3, GL_FLOAT, 0, &(ver[0]) );
    // dibujar usando vértices indexados
    // params.: (1) tipo de primitivas (2) número de índices
    // (3) tipo de índices (4) puntero a tabla de triáng.
   glDrawElements( GL_TRIANGLES, 3*num_tri, GL_UNSIGNED_INT, tri.data() ) ;
    // deshabilitar el array de vértices
   glDisableClientState( GL_VERTEX_ARRAY );
} ;
```

- Las coordenadas de cada vértice se envían a y se procesan una sola vez en la GPU (es más rapido).
- Cada índice de vértice debe ser enviado varias veces.

Archivos con mallas indexadas: el formato PLY

El formato PLY fue diseñado por Greg Turk y otros en la Univ. de Stanford a mediados de los 90. Codifica una malla indexada en un archivo ASCII o binario (usaremos la versión ASCII). Tiene tres partes:

- Cabecera: describe los atributos presentes y su formato, se indica el número de vértices y caras, ocupa varias líneas.
- Tabla de vértices: un vértice por línea, se indican sus coordenadas X,Y y Z (flotantes) en ASCII, separadas por espacios
- Tabla de caras: una cara por línea, se indica el número de vértices de la cara, y después los índices de los vértices de la cara (comenzando en cero para el primer vértice de la tabla de vértices).
- El formato ex extensible de forma que un archivo puede incluir otros atributos (p.ej., colores de vértices).

Su simplicidad hace fácil usarlo.

Ejemplo de archivo PLY

```
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
0.0  0.0  2.0
0.0  1.3  1.0
0.0  1.4  0.0
1.1  0.0  0.0
1.0  0.0  2.0
1.0  0.8  1.5
0.5  1.0  0.0
4  0 1 2 3
4  7 6 5 4
4  0 4 5 1
4  1 5 6 2
4  2 6 7 3
4  3 7 4 0
```

Tabla de vértices y tiras de triángulos

Es posible representar una malla como una tabla de vértices y varias tiras de triángulos. Cada tira almacena índices de vértices en lugar de coordenadas de vértices.

- Las coordenadas no se repiten en memoria.
- Se repiten en memoria los índices de vértices, pero menos veces que con tabla de vértices y triángulos.
- El modelado usando tiras es más complejo.

Se pueden visualizar usando **`glDrawElements`** con **`GL_TRIANGLE_STRIP`** como tipo de primitivas:

- Las coordenadas de vértice se envían y se procesan una sola vez
- Los índices de vértices se envían repetidos, pero solo un par de veces de media aprox.

Tabla de aristas

En una malla indexada podría ser conveniente (para ganar tiempo de procesamiento en ciertas aplicaciones) almacenar explicitamente las aristas (ahora esa info. está implícita en la tabla de triángulos). Se puede hacer usando un **tabla de aristas**:

- Contiene una entrada por cada arista.
- En cada entrada hay dos índices (naturales) de vértices (los dos vértices en los extremos de la arista).
- El orden de los vértices en cada entrada es irrelevante, pero las aristas no deben estar duplicadas.

La disponibilidad de esta tabla permite, por ejemplo, dibujar en modo alambre con begin/end sin repetir dos veces el dibujo de aristas adyacentes a dos triángulos.

Subsección 2.5
Representación con estructura de aristas aladas

Motivación

La estructura de malla indexada permite, por ejemplo, consultar con tiempo en $O(1)$ si un vértice es adyacente a un triángulo, pero:

- Para consultar si dos vértices son adyacentes, hay que buscar en la tabla de triángulos si los vértices aparecen contiguos en alguno: esto requiere un tiempo en $O(n_t)$.
- No se guarda información de las aristas. Las consultas relativas a aristas se resuelven también en $O(n_t)$.
- En general, las consultas sobre adyancencia son costosas en tiempo.

Para poder reducir los tiempos de cálculo de adyacencia a $O(1)$, se puede usar más memoria de la estrictamente necesaria para la malla indexada. Veremos la estructura de **aristas aladas** (para mallas que encierran un volumen, es decir: siempre hay **dos caras adyacentes a una arista**, no necesariamente triangulares)

Estructura de aristas aladas: tabla de aristas.

Una malla se puede codificar usando una tabla de vértices (**`tver`**) (similar a la de la mallas indexadas), mas una tabla de aristas (**`tari`**). Esta última:

- Tiene una entrada por cada arista , que guarda los dos índices de vértices:
  - **`vi`** = índice de **vértice inicial**
  - **`vf`** = índice de **vértice final**

  (es indiferente cual se selecciona como inicial y cual como final).
- Se asume que cada arista es adyacente a dos triángulos (la malla encierra un volumen). Se guarda:
  - **`ti`** = índice del **triángulo a la izquierda**
  - **`td`** = índice del **triángulo a la derecha**

  (izquierda y derecha entendidos según se recorre la arista en el sentido que va desde vértice inicial al final)
- Esto permite consultas directas sobre adyacencia **arista-vértice** y **arista-triángulo** (basta consultar la entrada de la tabla correspondiente a la arista).

Aristas siguiente y anterior

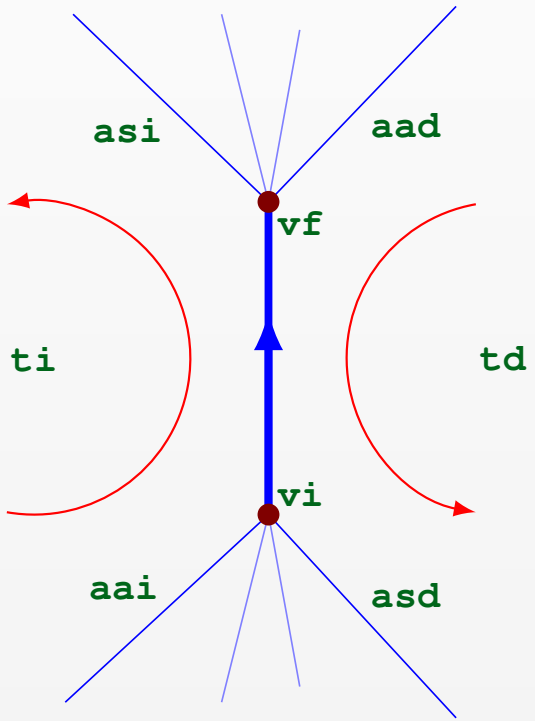
Además de los datos anteriores, se guarda, para cada arista, los índices de otras cuatro adyacentes a ella.

- Si se recorren las aristas del triángulo de la izquierda aparecerá la arista en cuestión entre otras dos, cuyos índices se guardan en la tabla:
  - **`aai`** = índice de la **arista anterior**
  - **`asi`** = índice de la **arista siguiente**

  (el recorrido de las aristas se hace en sentido anti-horario cuando se observa el triángulo desde el exterior de la malla).
- Igualmente, se guardan los dos índices de arista anterior y siguiente relativas al recorrido anti-horario del triángulo de la derecha:
  - **`aad`** = índice de la **arista anterior** (derecha)
  - **`asd`** = índice de la **arista siguiente** (derecha)

Índices asociados a una arista

Índices (valores naturales) en la entrada correspondiente a la arista vertical en una malla (no necesariamente de triángulos):



Uso de las aristas siguiente y anterior.

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

- Dada una arista y un triángulo adyacente (el izquierdo o el derecho), se pueden obtener estas listas:
  - aristas adyacentes al triángulo.
  - vértices adyacentes al triángulo
- Dada una arista y un vértice adyacente (el inicial o el final), se pueden obtener estas listas:
  - aristas que inciden en el vértice (esto permite resolver fácilmente adyacencias **arista-arista**)
  - triángulos adyacentes al vértice.

Con toda esta información (los 8 valores), se puede resolver directamente cualquier adyacencia que involucre una arista al menos (consultando su entrada). Aun no podemos resolver el resto.

Tablas adicionales. Uso.

Para poder resolver todo tipo de consultas en $O(1)$, hay que tener dos tablas nuevas, que llamamos **`taver`** y **`tatri`**:

- **`taver`** = tabla de **aristas de vértice**: para cada vértice, se debe almacenar el índice de una arista adyacente a dicho vértice (cualquiera de ellas).
  - dado un vértice, permite recuperar todas las aristas, vértices y triángulos adyacentes.
  - por tanto, permite consultas de adyacencia **vértice-vértice** y **vértice-triángulo**.
- **`tatri`** = tabla de **aristas de triángulo**: para cada triángulo, se almacena el índice de una arista adyacente al mismo (cualquiera de ellas).
  - dado un triángulo, permite recuperar todas las aristas, vértices y triángulos adyacentes,
  - por tanto, permite consultas de adyacencia **triángulo-triángulo** y **vértice-triángulo** (esta última se puede hacer de dos formas).

Subsección 2.6
Representación y visualización de atributos

Visualización de atributos de vértice en OpenGL

La librería OpenGL contempla explicitamente la asignación de atributos a vértices:

- **Colores**: se le asigna una tupla RGB a cada vértice
- **Normales**: se le asigna un vector XYZ (tupla de 3 coordenadas) a cada vértice, usualmente estará normalizado, pero no necesariamente.
- **Coordenadas de textura**: es un par de reales (se verá más adelante)
- **Otros atributos**: cuando se usa programación del cauce gráfico, se puede asociar un conjunto de atributos arbitrarios a cada vértice, atributos que son procesados por los *fragment shaders* (con semántica no prefijada, sino establecida por el programador).

A modo de ejemplo, veremos como visualizar especificando colores y normales en mallas con tablas de vértices y triángulos (la utilidad de las normales se verá más adelante).

Representación de atributos en mallas indexadas.

La representación de mallas indexadas permite añadir fácilmente atributos de caras y de vértices (normales y colores, p.ej.). Para cada atributo, se añade una tabla con un número de entradas igual al número de triángulos o vértices:

```
class MallaInd : public Objeto3D          // Malla indexada
{
  protected:
    ........
    std::vector<Tupla3f> nor_tri ;  // normales de triángulos (num_tri entradas)
    std::vector<Tupla3f> nor_ver ;  // normales de vértices (num_ver entradas)
    std::vector<Tupla3f> col_tri ;  // colores de triángulos (num_tri entradas)
    std::vector<Tupla3f> col_ver ;  // colores de vértices (num_ver entradas)

  public:
    .......
} ;
```

Si una malla no tiene algún atributo, el correspondiente vector estará vacío.

Asociación de atributos a vértices en OpenGL

Cada vértice que procesa OpenGL tiene asociado **siempre** una **normal** y un **color**.

- Si se usa **`glVertex`**, en el momento de la llamada OpenGL le asocia al vértice el **color actual** y la **normal actual**
  - Las funciones **`glNormal`** y **`glColor`** permiten cambiar el color actual
  - Se pueden invocar antes de cada llamada a **`glVertex`**, o bien antes de cada primitiva (el atributo se replicará en todos sus vértices)
- Cuando se usan *vertex arrays* (ya sea con **`glDrawArrays`** o bien con **`glDrawElements`**, en modo inmediato o con VBOs):
  - Se puede especificar un array de colores y/o otro de normales (tienen una entrada por vértice)
  - Si no se especifican, en el momento en que se dibujan OpenGL les asigna el color actual y la normal actual a **todos los vértices**.

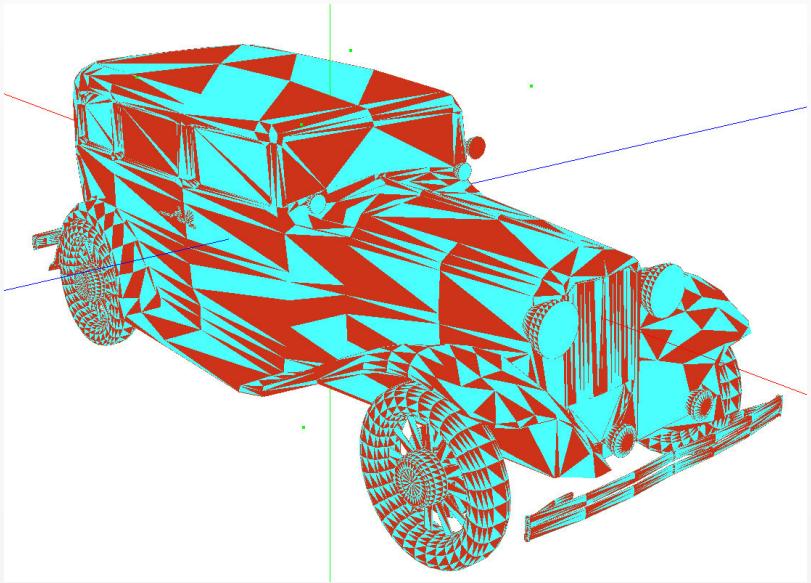
Atributos de caras en OpenGL

En principio, OpenGL no permite asociar explícitamente atributos a caras durante la visualización (ya dicha asociación que siempre se hace a nivel de vértices), sin embargo:

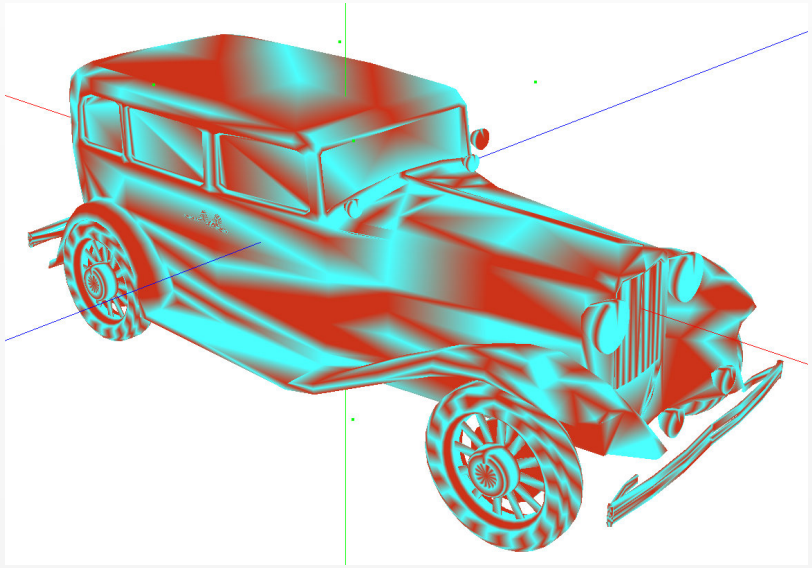
- En mallas de **triángulos aislados**, se puede replicar un color o una normal para todos los vértices de un triángulo, ya que los vértices no se comparten entre triángulos.
- Cuando se usa visualización **`glBegin`**/**`glEnd`**, también se pueden replicar unos atributos en todos los vértices de una cara, cambiandolos antes de comenzar el envío de cada cara.
- Por tanto: si se usa **`glDrawElements`** para visualización de mallas indexadas, no es posible visualizar asociando atributos a las caras.

Ejemplo de `glShadeModel`

Aquí vemos el efecto de cambiar el modo de sombreado para una malla indexada visualizada con **`glBegin`**/**`glEnd`** (se le asocian colores alternos a caras y a vértices).



`GL_FLAT`
(**`glColor`** antes de cada triángulo)



`GL_SMOOTH`
(**`glColor`** antes de cada vértice)

Atributos de caras con *begin/end*

Una función como esta puede usarse para visualizar mallas indexadas con atributos de caras usando **glBegin/glEnd**:

```
void MallaInd::visualizarBEAtrTri( )
{
  glBegin( GL_TRIANGLES ) ;
  for( unsigned i = 0 ; i < num_tri ; i++ )
  {
     if ( col_tri.size() > 0 )        // si hay normales de t. disponibles
        glColor3fv( col_tri[i] );     // color act.= color i-ésimo triángulo
     if ( nor_tri.size() > 0 )        // si hay colores de t. disponibles
        glNormal3fv( nor_tri[i] );    // normal act. = normal i-ésimo triángulo

     for( unsigned j = 0 ; j < 3 ; j++ )
     {  unsigned iv = tri[i](j) ;     // iv = índice de vértice.
        glVertex3fv( ver[iv] ) ;      // leer y enviar coords. de vért.
     }
  }
  glEnd() ;
}
```

Atributos de vértices con *begin/end*

El mismo esquema se puede usar para visualizar cambiando el color y la normal actual antes de cada vértice (usando las tablas de atributos de vértices), en lugar de antes de cada cara:

```
void MallaInd::visualizarBEAtrVer( )
{
   glBegin( GL_TRIANGLES ) ;
   for( unsigned i = 0 ; i < num_tri ; i++ )
   for( unsigned j = 0 ; j < 3 ; j++ )
   {
      unsigned iv = tri[i](j) ;          // iv = índice de vértice.

      if ( col_ver.size() > 0 )          // si hay normales de v. disponibles:
         glColor3fv( col_ver[iv] );      // color act.= color i-ésimo vértice
      if ( nor_ver.size() > 0 )          // si hay colores de v. disponibles:
         glNormal3fv( nor_ver[iv] );     // normal act. = normal i-ésimo vért.

      glVertex3fv( pm->ver[iv] ) ;       // enviar coords. de vértice
   }
   glEnd() ;
}
```

Atributos de vértices con *vertex arrays*

Cuando se usa **glDrawElements**, se pueden especificar vectores de atributos de vértices. OpenGL asocia a cada vértice su atributo durante la visualización:

```
void MallaInd::visualizarDEAtrVer( )
{
 if ( nor_ver.size() > 0 ) // si hay normales de v. disponibles:
 {  glEnableClientState( GL_NORMAL_ARRAY );          // habilitar normales
    glNormalPointer( GL_FLOAT, 0, nor_ver.data() );  // fija puntero a normales
 }
 if ( col_ver.size() > 0 ) // si hay colores de v. disponibles:
 {  glEnableClientState( GL_COLOR_ARRAY );           // habilitar colores
    glColorPointer( 3, GL_FLOAT, 0, col_ver.data() ); // fija puntero a colores
 }
 // visualizar usando el método ya explicado:
 visualizarDE() ;
 // deshabilitar arrays:
 glDisableClientState( GL_NORMAL_ARRAY );
 glDisableClientState( GL_COLOR_ARRAY );
}
```

Subsección 2.7
Visualización de mallas en modo diferido

Visualización en modo diferido.

La visualización en modo inmediato supone que en cada *frame* hay que volver a enviar (vía el bus del sistema) muchos bytes (lento). Para evitarlo se puede usar visualización en **modo diferido**:

- Una malla se envía una vez a la GPU.
- Se puede visualizar en más de un *frame* sin volverla a transferir.
- Solo es necesaria la transferencia cuando se modifica la malla.

OpenGL dispone de dos métodos para visualización en modo diferido

- ***Display Lists***: permite enviar cualquier orden OpenGL a la GPU para ser ejecutada más tarde (obsoletas en OpenGL 3.0 y eliminadas en OpenGL 3.1)
- ***Vertex Buffer Objects*** (VBO): permiten enviar las mallas con todos sus atributos, para su posterior visualización.

Vertex Buffer Objects

Un **Vertex Buffer Object** (VBO) es **un bloque contiguo de la memoria de la GPU**, que se usa para almacenar tablas de vértices, caras y otras, previamente a su visualización:

- Una vez copiados los datos a la GPU desde la RAM, la visualización se hace mucho más rápido, sin necesidad de enviarlos en cada cuadro de una animación.
- Cada VBO se identifica en el programa con un valor de tipo `GLuint`, proporcionado por OpenGL cuando se crea. Se llama **identificador** del VBO.
- Cada tabla de datos (vértices, caras, otras) se identifica por un VBO (donde reside) y un *offset* (desplazamiento) dentro del VBO, donde comienza dentro del bloque del VBO.
- Un VBO puede contener una o varias tablas (entrelzadas o no), en nuestro caso usaremos un VBO por tabla, por simplicidad.

Uso de *Vertex Buffer Objects* para mallas

Las diferentes tablas que forman una malla se pueden incluir en VBOs, cada una en un VBO. Existen órdenes OPenGL para:

- Crear nuevos VBOs en la GPU, obteniendo sus identificadores enteros (crearemos dos: uno para la tabla de triángulos y otro para la de vértices)
- Transferir triángulos y vértices desde RAM hacia los VBOs en la GPU.
- Activar un VBO para especificar donde está el puntero a los vértices, (con **`glVertexPointer`**). Nuestro *offset* será 0.
- Visualizar los VBOs (con **`glDrawElements`**), indicando un offset en la GPU (que será cero también).

Los identificadores de VBOs formarán parte del objeto malla.

Creación de un VBO

La creación de un VBO tiene como parámetros el tamaño y la dirección en RAM desde donde copiar. Se obtiene como resultado un identificador entero positivo:

```
GLuint VBO_Crear( GLuint tipo, GLuint tamanio, GLvoid * puntero )
{
   assert( tipo == GL_ARRAY_BUFFER || tipo == GL_ELEMENT_ARRAY_BUFFER );
   GLuint id_vbo ;                    // resultado: identificador de VBO
   glGenBuffers( 1, & id_vbo );       // crear nuevo VBO, obtener identificador
   glBindBuffer( tipo, id_vbo );      // activar el VBO usando su identificador
   glBufferData( tipo, tamanio, puntero, GL_STATIC_DRAW );
                                      // transferencia RAM -> GPU
   glBindBuffer( tipo, 0 );           // desactivación del VBO (activar 0)
   return id_vbo ;                    // devolver el identificador resultado
}
```

el tipo permite diferenciar si el VBO contiene datos de vértices (**GL_ARRAY_BUFFER**) o bien si contiene una tabla de triángulos (índices de vértices) (**GL_ELEMENT_ARRAY_BUFFER**).

Creación de VBOs para mallas indexadas

Para el caso de mallas indexadas, es necesario crear al menos dos VBOs, uno para la tabla de vértices y otro la de triángulos. En la malla se almacenan los identificadores de VBOs:

```
void MallaInd::crearVBOs(  )
{
    // crear VBO conteniendo la tabla de vértices
    id_vbo_ver = VBO_Crear( GL_ARRAY_BUFFER, tam_ver, ver.data() );

    // crear VBO con la tabla de triángulos (índices de vértices)
    id_vbo_tri = VBO_Crear( GL_ELEMENT_ARRAY_BUFFER, tam_tri, tri.data() );
}
```

Identificadores de VBOs y su tamaño

Los datos que se añaden a la estructura **MallaInd** (sección **protected**) son estos:

```
class MallaInd : public Objeto3D
{   .....
   GLuint   id_vbo_ver ;  // identificador del VBO con la tabla de vértices
   GLuint   id_vbo_tri ;  // identificador del VBO con la tabla de triángulos

   unsigned tam_ver ;  // tamaño en bytes de la tabla de vértices
   unsigned tam_tri ;  // tamaño en bytes de la tabla de triángulos
   ....
} ;
```

el cálculo de **tam_ver** y **tam_tri** puede hacerse con:

```
tam_ver = sizeof(float)*3L*num_ver ;
tam_tri = sizeof(unsigned)*3L*num_tri ;
```

Visualización con VBOs para mallas ind.

La malla se puede visualizar muchas veces sin volver a enviar los datos a la GPU:

```
void MallaInd::visualizarVBOs( )
{
   // especificar formato de los vértices en su VBO (y offset)
   glBindBuffer( GL_ARRAY_BUFFER, id_vbo_ver ); // act. VBO
   glVertexPointer( 3, GL_FLOAT, 0, 0 );     // formato y offset (=0)
   glBindBuffer( GL_ARRAY_BUFFER, 0 );       // desact VBO.
   glEnableClientState( GL_VERTEX_ARRAY ); // act. uso VA

   // visualizar con glDrawElements (puntero a tabla == NULL)
   glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, id_vbo_tri );
   glDrawElements( GL_TRIANGLES, 3L*num_tri, GL_UNSIGNED_INT, NULL ) ;
   glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, 0 );

   // desactivar uso de array de vértices
   glDisableClientState( GL_VERTEX_ARRAY );
}
```

Creación de VBOs para atributos de vértices

Los colores y normales de vértices se pueden almacenar en sus propios VBOs, de una forma parecida como se hace con los vértices:

```
// crear VBO con los colores de los vértices
if ( col_ver.size() > 0 )
  id_vbo_col_ver = VBO_Crear( GL_ARRAY_BUFFER, tam_ver, col_ver.data() );

// crear VBO con las normales de los vértices
if ( nor_ver.size() > 0 )
  id_vbo_nor_ver = VBO_Crear( GL_ARRAY_BUFFER, tam_ver, nor_ver.data() );
```

este código requiere guardar en la malla **id_vbo_col_ver** y **id_vbo_nor_ver** (los identificadores de VBOs, ambos de tipo **GLuint**).

Visualización de atributos de vértices con VBOs

La visualización teniendo en cuenta los atributos de vértices almacenados en sus VBOs puede hacerse como sigue:

```
void MallaInd::visualizarVBOsAtrVer( )
{
   if ( col_ver.size() > 0 )
   {  glBindBuffer( GL_ARRAY_BUFFER, id_vbo_col_ver );  // act. VBO col.v.
      glColorPointer( 3, GL_FLOAT, 0, 0 );     // formato y offset de colores
      glEnableClientState( GL_COLOR_ARRAY );  // activa uso de colores de v.
   }
   if ( nor_ver.size() > 0 )
   {  glBindBuffer( GL_ARRAY_BUFFER, id_vbo_nor_ver );  // act. VBO nor.v.
      glNormalPointer( GL_FLOAT, 0, 0 );        // formato y offset de normales
      glEnableClientState( GL_NORMAL_ARRAY );  // activa uso de normales
   }
   visualizarVBOs( );   // visualización con glDrawElements

   glDisableClientState( GL_COLOR_ARRAY );   // desact. array de colores
   glDisableClientState( GL_NORMAL_ARRAY );  // desact. array de normales
}
```

Sección 3
Transformaciones geométricas

3.1. Concepto de transformación geométrica (TG).
3.2. Transformaciones usuales en IG.
3.3. Representación y operaciones sobre matrices.
3.4. Transformaciones en OpenGL.
3.5. Gestión de la matriz de modelado en C/C++/GLSL.

Subsección 3.1
Concepto de transformación geométrica (TG).

Coordenadas del mundo e instanciación de objetos.

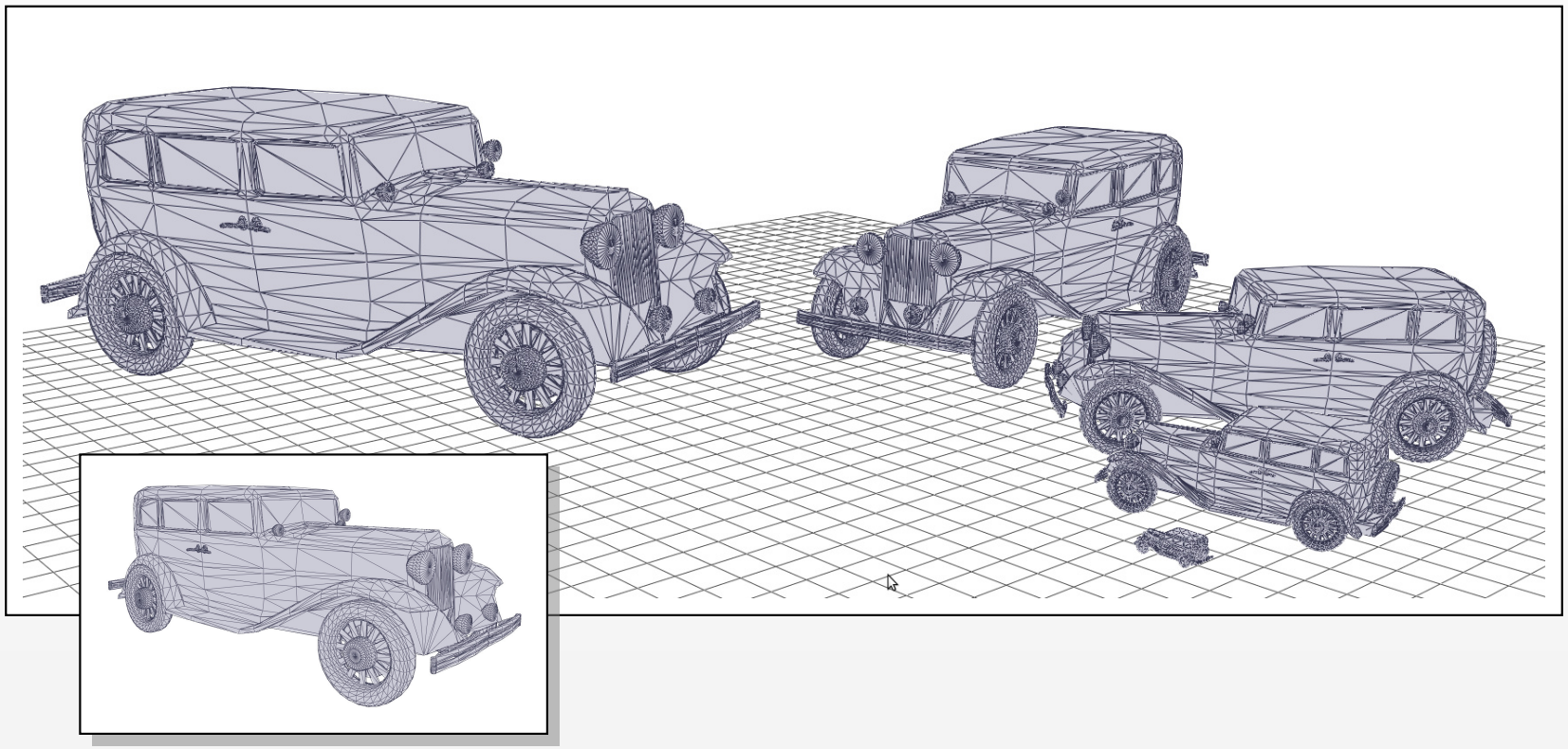
Las mallas que hemos visto en la sección anterior tienen las coordenadas de sus vértices definidas respecto de un sistema de referencia local (coordenadas maestras o locales) , pero :

- En una escena con varias mallas (o, en general, varios objetos) todos los vértices deben aparecer referidos a un único sistema de referencia común
- Dicho sistema es el llamado **marco de coordenadas de la escena**, o **marco de coordenadas del mundo**, (***world coordinate system***), y es un marco cartesiano.
- Las coordenadas de los vértices, respecto de dicho sistema de referencia, se llaman **coordenadas del mundo** (*world coordinates*)
- Esto permite separar la definción de los objetos (en coordenadas maestra), de su uso en una escena concreta, lo cual es usual en la industria de la infografía 3D actualmente.

Un objeto se define una vez pero se puede **instanciar** muchas veces en una o distintas escenas.

Instanciación de una malla en una escena

A modo de ejemplo, una malla indexada se podría instanciar varias veces en distintas posiciones, orientaciones y tamaños:



Transformación geométrica

Para lograr la instanciación, hay que modificar la posición de los vértices de cada objeto, o, lo que es lo mismo, calcular sus coordenadas del mundo a partir de las coordenadas locales o maestras. Para esto se usan transformaciones geométricas

- Lo más frecuente es que esas transformaciones sean transformaciones afines
- En este tema veremos las transformaciones afines más comunes:
  - Rotaciones
  - Traslaciones
  - Escalado (uniformes y no uniformes)
  - Cizallas

Las transformaciones geométricas se usan para instanciar objetos, o, en general, modificar su posición, orientación y tamaño.

Subsección 3.2
Transformaciones usuales en IG.

Transformaciones geométricas usuales en IG

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

En lo que sigue veremos con algo de más detalle estas transformaciones básicas, junto con algunas formas útiles de componerlas.

Transformaciones afines: propiedades.

Es fácil verificar formalmente que una transformación afín tiene las siguientes propiedades:

- Transforma líneas rectas en líneas rectas (y planos en planos)
- Transforma dos líneas paralelas en otras dos líneas paralelas.
- Conserva los ratios entre las longitudes de dos segmentos en dos líneas paralelas.
- No conserva el área o volumen de los objetos.
- No conserva las distancias.
- No conserva los ángulos entre líneas o planos.

Transformación de traslación en 3D

Si $\vec{d}$ es un vector de un espacio afín, la transformación de **traslación** por $\vec{d}$ en dicho espacio es la transformación afín que desplaza cada punto según el vector $\vec{d}$, pero no afecta a los vectores, es decir, para cualquier punto $\dot{p}$ y vector $\vec{v}$:

$$T(\dot{p}) \equiv \dot{p} + \vec{d} \qquad\qquad T(\vec{v}) \equiv \vec{v}$$

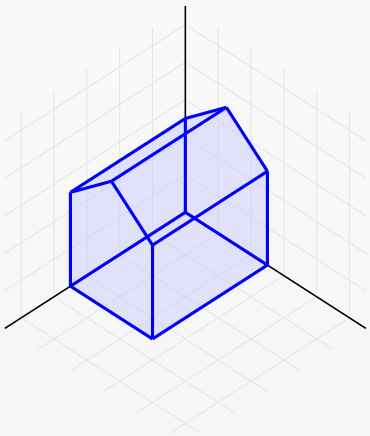
Si $\vec{d} = \mathcal{R}(d_x, d_y, d_z, 0)$, entonces T transforma el punto (o vector) $\mathcal{R}(x, y, z, w)^t$ en el punto (o vector) $\mathcal{R}(x', y', z', w)^t$, donde:

$$\begin{aligned} x' &= x + d_x\, w \\ y' &= y + d_y\, w \\ z' &= z + d_z\, w \end{aligned}$$

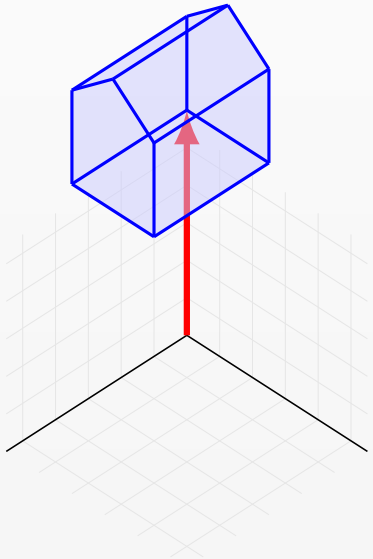
La correspondiente matriz (en $\mathcal{R}$) la llamamos $\mathrm{D}\left[d_x, d_y, d_z\right]$

Ejemplos de traslaciones

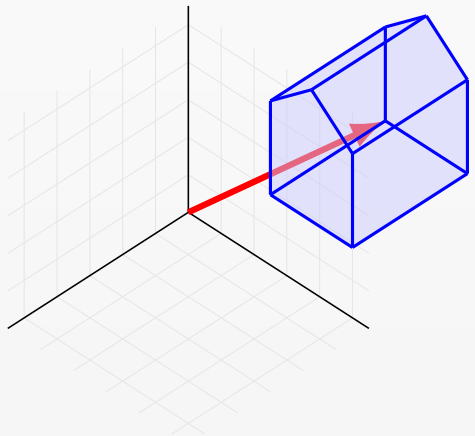
Aquí vemos a modo de ejemplo varias transformaciones de traslación aplicadas a los puntos de una figura sencilla.



$\mathrm{D}[0,0,0]$



$\mathrm{D}[0,1.2,0]$



$\mathrm{D}[0.7,0.6,-0.5]$

Transformación de escalado en 3D

Esta transformación afín es relativa a un sistema de referencia $\mathcal{R}$. Viene determinada por una tupla de valores reales $\mathbf{e} = (e_x, e_y, e_z)$.

- Equivale a un cambio de escala o tamaño con centro en el origen de $\mathcal{R}$ (deja el origen de $\mathcal{R}$ sin modificar).
- Afecta igual a puntos y a vectores.

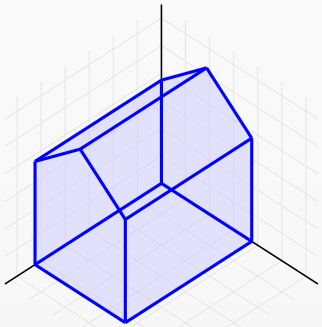
Esta transformación transforma el punto (o vector) $\mathcal{R}(x, y, z, w)^t$ en $\mathcal{R}(x', y', z', w)^t$, donde:

$$\begin{aligned} x' &= e_x\, x \\ y' &= e_y\, y \\ z' &= e_z\, z \end{aligned}$$

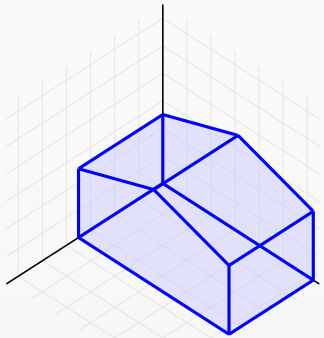
A la correspondiente matriz (en $\mathcal{R}$) la llamamos $\mathrm{E}\left[e_x, e_y, e_z\right]$

Ejemplos de transformaciones de escalado

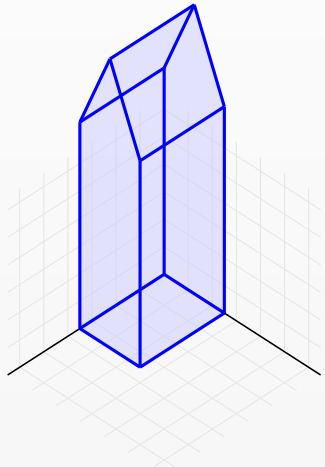
Aquí vemos varios ejemplos de la transformación de escalado.



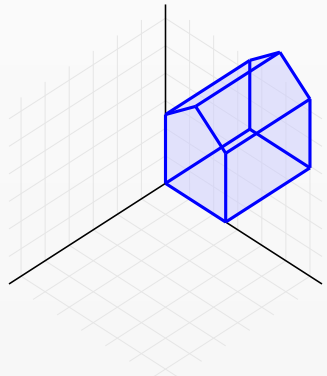
$\mathrm{E}[1.5, 1.5, 1.5]$
uniforme
$e_x = e_y = e_z$



$\mathrm{E}[2.5, 1, 1]$
no uniforme
$e_x \neq e_y = e_z$



$\mathrm{E}[1, 3, 1]$
no uniforme
$e_y \neq e_x = e_z$



$\mathrm{E}[1, 1, -1]$
espejo
$e_z < 0$

Transformaciones de cizalla

Las **transformaciones de cizalla** (*shear*) son también relativas a un sistema de referencia $\mathcal{R} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$:

- Son como las traslaciones en la dirección de un eje, pero usando una distancia proporcional a la coordenada de otro eje.
- Se definen con tres parámetros: un real y dos ejes. Afectan igual a puntos y vectores.

Llamamos C[..] a las correspondientes matrices (resp. de $\mathcal{R}$). Hay 6 posibles cizallas en 3D, aquí las expresiones de 3 de ellas:

$\mathrm{C_{xy}}[a]$:

$$\begin{aligned} x' &= x + ay \\ y' &= y \\ z' &= z \end{aligned}$$

$\mathrm{C_{yx}}[b]$:

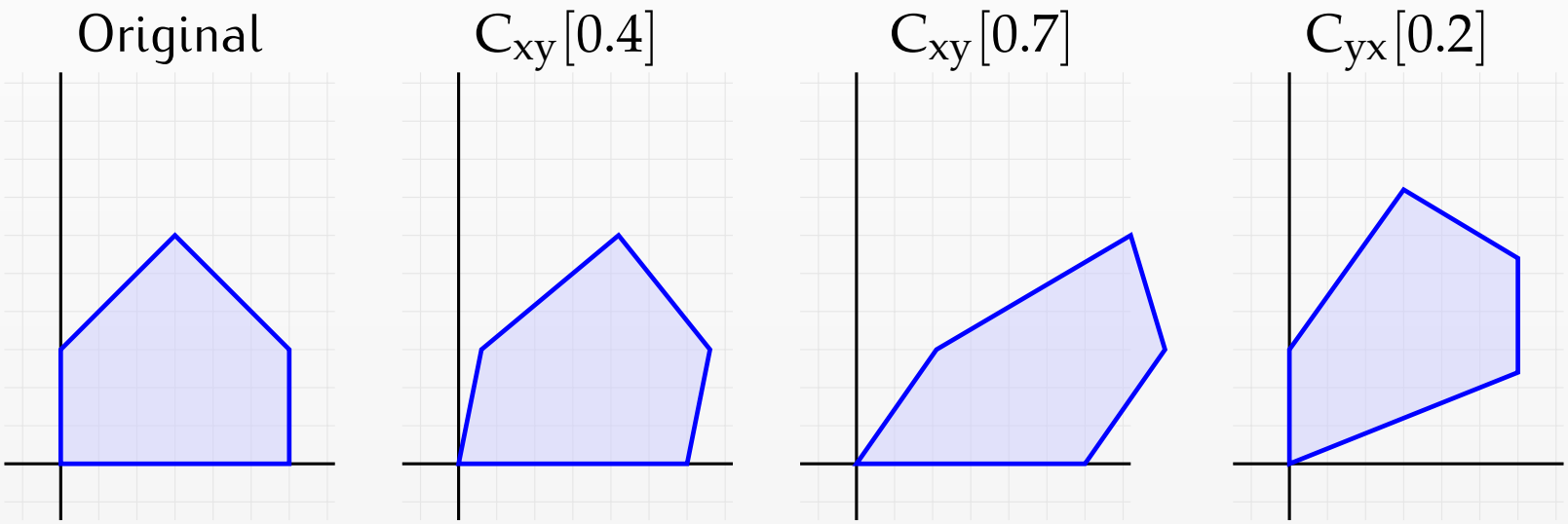
$$\begin{aligned} x' &= x \\ y' &= y + bx \\ z' &= z \end{aligned}$$

$\mathrm{C_{zy}}[c]$:

$$\begin{aligned} x' &= x \\ y' &= y \\ z' &= z + cy \end{aligned}$$

Ejemplos de transformaciones de cizalla.

En el caso 2D, hay dos posibles cizallas. Aquí las vemos actuando sobre una figura simple:



Transformaciones de rotación

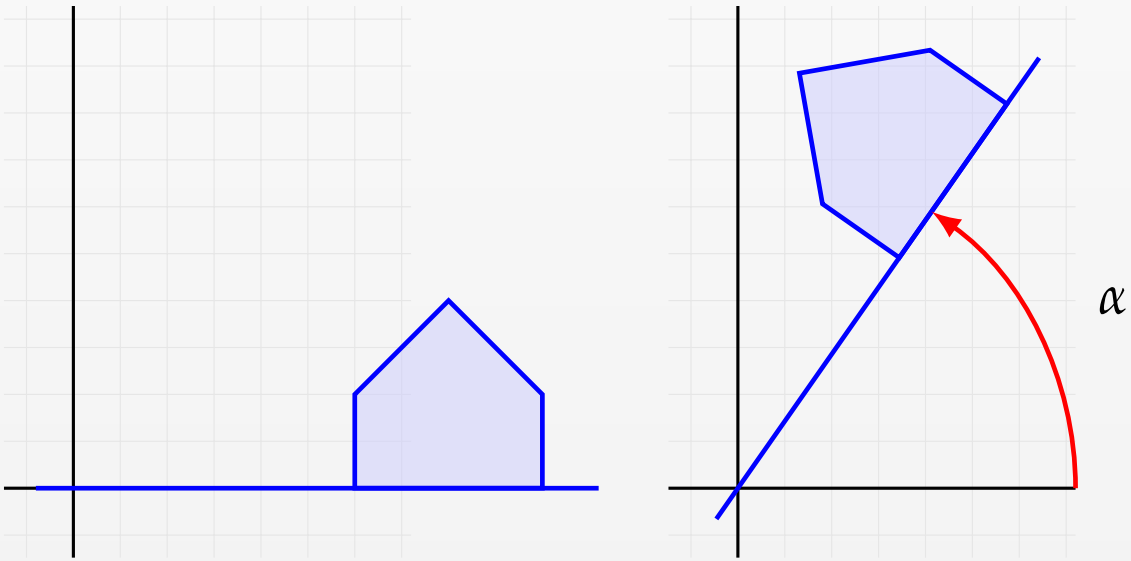
Una transformación de **rotación**, en un espacio afín 2D o 3D es una transformación afín que conserva las distancias entre puntos y tiene al menos un punto que no varía:

- En 2D, una rotación viene determinada (en un marco $\mathcal{R}$) por un ángulo θ. El punto $\dot{o}$ (el origen de $\mathcal{R}$ no varía). A la correspondiente matriz la llamamos $\mathrm{R}[\theta]$.
- En 3D, una rotación viene determinada (en un marco $\mathcal{R}$) por un vector $\vec{v} = \mathcal{R}\mathbf{v}$ (el origen de $\mathcal{R}$ y el vector $\vec{v}$ definen la línea que llamamos **eje de rotación**), más un ángulo θ (un real). Los puntos en el eje no varían. A la correspondiente matriz la llamamos $\mathrm{R}[\theta, \mathbf{v}]$.
- Las rotaciones afectan igual a los puntos y los vectores.

Rotación en 2D, respecto de un marco $\mathcal{R}$

En un marco de referencia $\mathcal{R} = [\vec{x}, \vec{y}, \dot{o}]$, la transformación (un ángulo α radianes) se define por estas dos expresiones

$$\begin{aligned} x' &= x \cos\alpha - y \sin\alpha \\ y' &= x \sin\alpha + y \cos\alpha \end{aligned}$$



Transformaciones de rotación en 3D.

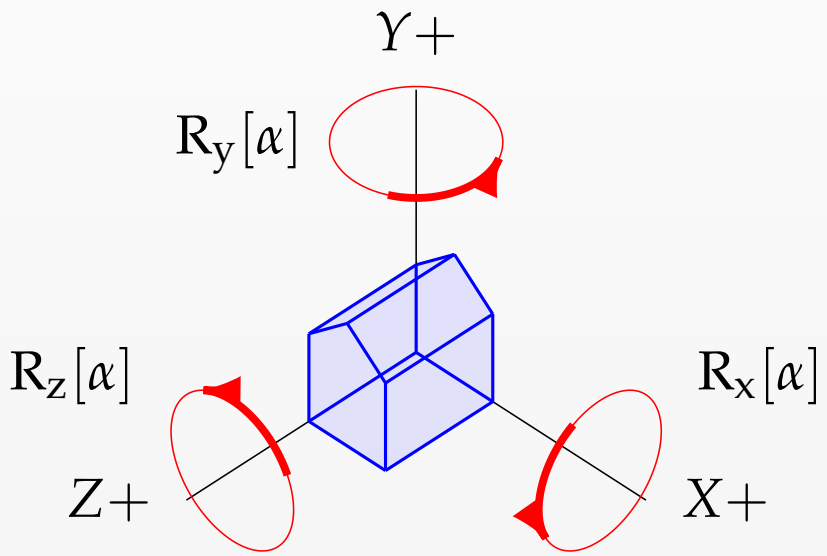
En un espacio afín consideramos un marco $\mathcal{R} = [\vec{x}, \vec{y}, \vec{z}, \dot{o}]$ y una transformación de rotación de θ radianes, con eje en el vector $\vec{v}$

- La rotación es en sentido anti-horario cuando $\theta > 0$ (según se mira hacia $\dot{o}$ desde la punta del vector $\vec{v}$).
- Llamamos rotaciones **elementales** a las que tienen como eje los versores de $\mathcal{R}$. Hay 3 rotaciones elementales (una por eje). A las correspondientes matrices las llamamos:

$$\begin{aligned} \mathrm{R_x}[\theta] &\equiv \mathrm{R}[\theta, (1,0,0)] \\ \mathrm{R_y}[\theta] &\equiv \mathrm{R}[\theta, (0,1,0)] \\ \mathrm{R_z}[\theta] &\equiv \mathrm{R}[\theta, (0,0,1)] \end{aligned}$$

Transformaciones de rotación en 3D.

En la figura, los círculos simbolizan indican como se rotan los puntos bajo cada rotación elemental:



- Cada rotación modifica dos coordenadas y deja la otra.
- Son rotaciones **siempre en sentido anti-horario** (para $\alpha > 0$ cuando se mira hacia el origen desde la rama positiva del eje de giro).

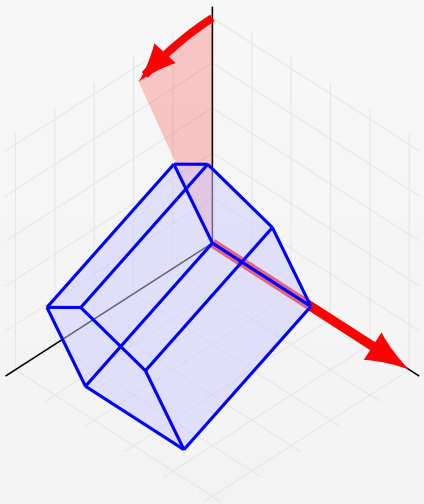
Expresiones de las rotaciones elementales

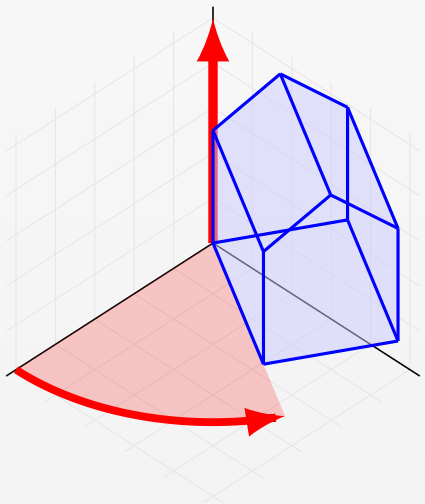
Definiendo $c \equiv \cos\alpha$ y $s \equiv \sin\alpha$, las expresiones son:

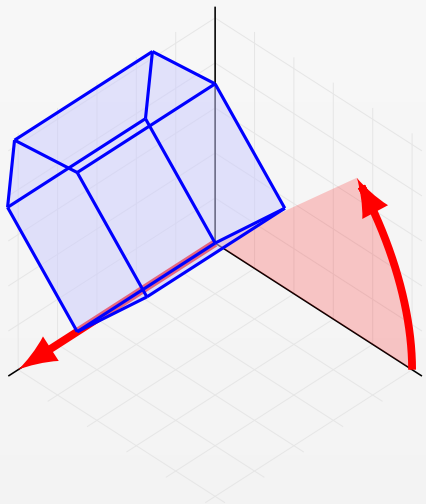
$$\mathrm{R_x}[\alpha] \qquad \mathrm{R_y}[\alpha] \qquad \mathrm{R_z}[\alpha]$$

$$\begin{aligned} x' &= x \\ y' &= cy - sz \\ z' &= sy + cz \end{aligned} \qquad \begin{aligned} x' &= cx + sz \\ y' &= y \\ z' &= -sx + cz \end{aligned} \qquad \begin{aligned} x' &= cx - sy \\ y' &= sx + cy \\ z' &= z \end{aligned}$$

$\mathrm{R_x}[23^o]$ $\mathrm{R_y}[60^o]$ $\mathrm{R_z}[45^o]$





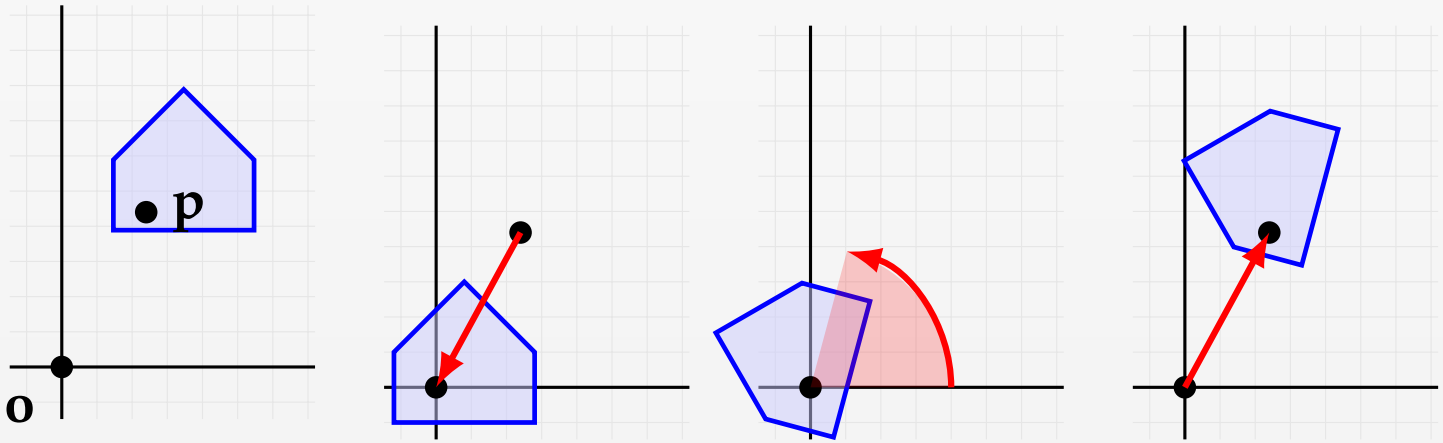


Rotaciones sobre puntos arbitrarios

Un ejemplo de composición es la rotación entorno a un punto $\dot{p} = \mathcal{R}\mathbf{p}$ distinto del origen, que se consigue componiendo (1) traslación al origen de $\mathcal{R}$ (por el vector $\dot{o} - \dot{p}$), (2) rotación por α (en torno al origen $\dot{o}$) y (3) traslación inversa (por $\dot{p} - \dot{o}$). La correspondiente matriz es el producto de las tres (de derecha a izquierda)

$$\mathrm{R}[\alpha, \mathbf{p}] \quad = \quad \mathrm{D}[\mathbf{p}] \cdot \mathrm{R}[\alpha] \cdot \mathrm{D}[-\mathbf{p}]$$

las transformaciones usadas se aprecian en esta figura:



Rotaciones de eje arbitrario

En 3D las rotaciones distintas de las elementales, que tienen como eje de rotación un vector $\vec{e} = \mathcal{R}(e_x, e_y, e_z)$ se definen (en el marco $\mathcal{R}$) por estas expresiones:

$$\begin{array}{rcllllll} x' & = & (h_x e_x + c)\, x & + & (h_x e_y - s e_z)\, y & + & (h_x e_z + s e_y)\, z \\ y' & = & (h_y e_x + s e_z)\, x & + & (h_y e_y + c)\, y & + & (h_y e_z - s e_x)\, z \\ z' & = & (h_z e_x - s e_y)\, x & + & (h_z e_y + s e_x)\, y & + & (h_z e_z + c)\, z \end{array}$$

donde:

- $c \equiv \cos\alpha, \quad s \equiv \sin\alpha.$
- $h_x \equiv (1-c)e_x, \quad h_y \equiv (1-c)e_y, \quad h_z \equiv (1-c)e_z$

Matrices de transformación 3D usuales

$$\mathrm{E}[s_x, s_y, s_z] = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathrm{D}[d_x, d_y, d_z] = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathrm{C_{xy}}[a] = \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathrm{R_x}[\alpha] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathrm{R_y}[\alpha] = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathrm{R_z}[\alpha] = \begin{pmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde: $c \equiv cos(\alpha)$ y $s \equiv sin(\alpha)$, y α es el ángulo de rotación, en radianes

Subsección 3.3
Representación y operaciones sobre matrices.

Representación de transf. en memoria.

Para representar una matriz en memoria, es cómodo almacenar los 16 valores de forma contigua, y de tal manera que se puedan acceder usando el índice de fila y de columna. Para ello se puede usar el tipo de datos **Matriz4f**:

```
#include <matrizg.hpp>
// declaraciones de matrices
Matriz4f m,m1,m2,m3 ;  float a,b,c ; unsigned f = 0 , c = 1 ;
// accesos (comprobados) de lectura (var = matriz(fila,columna))
a = m(1,2) ;
b = m(f,c);
// accesos (comprobados) de escritura (matriz(fila,columna) = expr)
m(1,2) = 34.6 ;
m(f,c) = 0.0 ;
// multiplicación o composición de matrices
m1 = m2*m3 ;
// multiplicación de matriz 4x4 por tupla de 4 floats (y de 3, añadiendo 1)
Tupla4f c4( 1.0,2.0,3.0,4.0 ) ;  Tupla3f c3( 1.0,1.0,3.0);
m1 = m2*c4 ;  m1 = m2 * c3
// conversión a puntero a 16 flotantes (float *) (formato OpenGL)
float * pm = m ;   const float * pcm = m ;
// escritura en la salida estándar (varias líneas)
cout << m << endl ;
```

Matrices más usuales

También podemos construir funciones C++ para obtener las matrices más usuales:

```
// devuelve la matriz identidad
Matriz4f MAT_Ident( ) ;

// devuelven una matriz de traslación por dx,dy,dz (o d[X],d[Y],d[Z])
Matriz4f MAT_Traslacion( const float d[3] ) ;
Matriz4f MAT_Traslacion( const float dx, const float dy ,
                         const float dz ) ;

// devuelve una matriz de escalado por s_x,s_y,s_z
Matriz4f MAT_Escalado( const float sx, const float sy,
                       const float sz ) ;

// devuelve una matriz de rotación de eje arbitrario (ex,ey,ez)
Matriz4f MAT_Rotacion( const float ang_gra, const float ex,
                       const float ey, const float ez ) ;
```

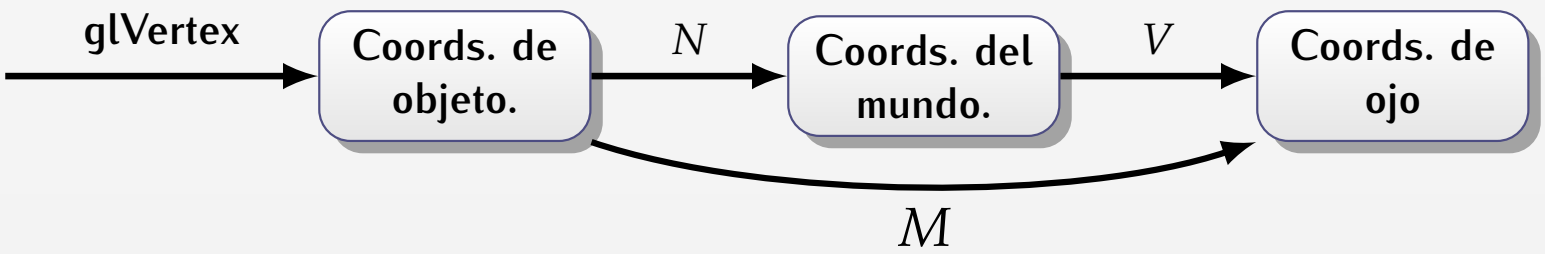
Subsección 3.4
Transformaciones en OpenGL.

La matriz *Modelview* de OpenGL.

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**). Dicha matriz se puede ver como la composición de otras dos, V y N:

- $N \equiv$ **matriz de modelado** (*modeling matrix*): posiciona los puntos en su lugar en coordenadas del mundo.
- $V \equiv$ **matriz de vista** (*view matrix*): posiciona los puntos en su lugar en coordenadas relativas a la cámara

La matriz *modelview* M se aplica a todos los puntos generados 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 actuan sobre la matriz *modelview* M.
2. usar **`glLoadIdentity`**`()` para hacer M igual a la matriz identidad ($M:=$Ide).
3. usar **`gluLookAt`** u otras para componer una matriz de vista V en M ($M:=M\cdot V$)
4. usar una (o varias) llamadas para componer una matriz de modelado con N (llamadas a las funciones **`glMultMatrix`**, **`glScale`**, **`glRotate`**, o **`glTranslate`**) ($M:=M\cdot N$)

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

Llamadas útiles para la matriz de modelado.

OpenGL incorpora estas llamadas (también con `d` en lugar de `f`)

```
// hace M := M · R[α, e_x, e_y, e_z] (α en grados)
glRotatef( GLfloat α, GLfloat e_x, GLfloat e_y, GLfloat e_z )

// hace M := M · D[d_x, d_y, d_z].
glTranslatef( GLfloat d_x, GLfloat d_y, GLfloat d_z )

// hace M := M · E[s_x, s_y, s_z].
glScalef( GLfloat s_x, GLfloat s_y, GLfloat s_z )

// hace M := M A, donde A = (a_ij) es una matriz (por columnas)
GLfloat A[16] = {  a_0,0, a_1,0, a_2,0, a_3,0,  // columna 0
                   a_0,1, a_1,1, a_2,1, a_3,1,  // columna 1
                   a_0,2, a_1,2, a_2,2, a_3,2,  // columna 2
                   a_0,3, a_1,3, a_2,3, a_3,3   // columna 3
                 } ;
glMultMatrixf( GLfloat * A ) ;
```

Geneación de *modelview* por composición.

En general, el código permite ir componiendo matrices con la matriz modelview:

```
glMatrixMode(GL_MODELVIEW);  // indicamos que, a partir de aquí,
                             // se modifica la matriz modelview, M.
glLoadIdentity() ;   // hacemos M := Ide (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:

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

es decir, el efecto de M es igual al efecto de T_1 seguido de T_2 seguido de T_3

Un ejemplo de un programa.

Suponemos que **VisObjeto()** dibuja un objeto cuyos vértices (en coordenadas de objeto) están entre -1 y 1 en todos los ejes.

El siguiente trozo de código haría que dicho objeto primero se viese rotado en el eje X un ángulo igual a 45 grados y después desplazado al punto $(5,6,7)$:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity() ;
                  // M = Ide
gluLookAt( 15.0,15.0,15.0, 5.0,6.0,7.0, 0.0,1.0,0.0 ) ;
                  // M := M · V (V ≡ matriz 'look-at')
glTranslatef( 5.0, 6.0, 7.0 ) ;
                  // M := M · D[5,6,7]
glRotatef( 45.0,  1.0, 0.0, 0.0 ) ;
                  // M := M · R[45º,x]
VisObjeto() ;
                  // visualizar con M ≡ V · D[5,6,7] · R[45º,x]
```

Subsección 3.5
Gestión de la matriz de modelado en C/C++/GLSL.

Gestión de matrices

La funcionalidad de OpenGL relacionada con la gestión de de matrices está declarada **obsoleta** en OpenGL versión 3.1 y **eliminada** de OpenGL 3.3 y posteriores (igualmente, ha sido eliminada de OpenGL ES 2.0).

- Es necesario escribir el código para la gestión de las matrices (o usar alguna librería a tal efecto).
- En nuestro caso, podemos usar el tipo `Matriz4f`, con las operaciones ya vistas.

Vamos a ver ejemplos para gestionar desde la aplicación la matriz de modelado, sin usar la funcionalidad de OpenGL

- Esto requiere pasar la matriz de modelado al vertex shader como un parámetro.
- Veremos los mecanismos de paso de parámetros en GLSL.

Parámetros en el vertex shader

El vertex shader sencillo que ya vimos puede extenderse para aceptar un parámetro de tipo matriz de 16 flotantes (un parámetro **uniform**).

```
// declarar parámetro uniform tipo matriz 4x4
uniform mat4 matrizModelado ;

void main()
{
   gl_FrontColor = gl_Color ;
   // aplicar la matriz de modelado antes que las de OpenGL:
   gl_Position = gl_ModelViewProjectionMatrix
                 * matrizModelado * gl_Vertex ;
}
```

- Un parámetro es **uniform** cuando su valor es constante en todos los vértices y píxeles de una primitiva
- El uso de **gl_ModelViewProjectionMatrix** también es obsoleto, veremos como evitarlo.

Fijar valores de los parámetros uniform

En la aplicación (CPU) es necesario dar valores a los parámetros uniform:

- Cada parámetro uniform (de cualquier tipo) en un programa se identifica por un valor entero no negativo que se denomina su **localización** (**location**), único en el programa.
- Dando el nombre del parámetro, es posible **obtener su localización** con una llamada a **`glGetUniformLocation`** (si no existe o no se usa se devuelve `-1`).
- Conocida la localización de un *uniform*, es posible en cualquier momento **asignarle valor** mediante una llamada a una versión de **`glUniform`**
- Para cada tipo de dato (enteros,flotantes, vectores, matrices, etc....) hay una versión de **`glUniform`**.

Fijar valores de los parámetros uniform

Para facilitar la tarea de asignarle valores a los uniform, podemos usar esta función auxiliar (sirve para matrices tipo **Matriz4x4f** en la aplicación)

```
void asignarUniform( GLuint idProg, const char * nombre,
                     const Matriz4f & mat )
{
   using namespace std ;
   const int loc = glGetUniformLocation(idProg,nombre);
   if ( loc != -1 )
     glUniformMatrix4fv( loc,1,GL_FALSE, mat );
   else
     cout << "advertencia: matriz uniform '"
          << nombre
          << "' no esta declarada en el programa "
          << " (o no se usa en la salida)" << endl  ;
}
```

Ejemplo de un programa.

Ahora vemos un ejemplo equivalente al anterior :

```
// asignar V (view matriz) con OpenGL:
glMatrixMode( GL_MODELVIEW );
glLoadIdentity() ;
gluLookAt( 15.0,15.0,15.0, 5.0,6.0,7.0, 0.0,1.0,0.0 ) ;

// crear y asignar 'matrizModelado':
const Matriz4f
  matrizModelado =
      MAT_Traslacion( 5.0, 6.0, 7.0 ) *
      MAT_Rotacion( 45.0,  1.0, 0.0, 0.0 ) ;

asignarUniform(idProg,"matrizModelado",matrizModelado);

// visualizar usando los shaders:
glUseProgram( idProg );
VisObjeto() ;  // visualizar con M ≡ V · D[5,6,7] · R[45º,x]
```

Sección 4
Modelos jerárquicos. Representación y visualización.

Subsección 4.1
Modelos Jerárquicos y grafo de escena.

La escena como vector de objetos.

En una escena típica actual se incluyen muchas instancias distintas de muchas mallas u **objetos geométricos** (una escena S es una serie de n objetos $S = \{O_1, O_2, \ldots, O_n\}$)

- Cada objeto O_i se incluye con una transformación T_i.
- Un mismo objeto puede instanciarse más de una vez ($O_i = O_j$), pero con distintas transformaciones ($T_i \neq T_j$).
- Cada transformación sirve para situar al objeto (definido en su propio marco de referencia $\mathcal{R}_i$) en su lugar en relación al marco de referencia de la escena (es el **marco de referencia del mundo**, lo llamamos $\mathcal{W}$).
- La matriz de T_i sirve para convertir desde coordenadas relativas a $\mathcal{R}_i$ a coordenadas relativas a $\mathcal{W}$.

Jerarquías: objetos simples y compuestos. DAGs

A pesar de ser versátil, el esquema anterior es complejo de usar para escenarios muy complejos. En estos casos se usan **modelos jerárquicos**.

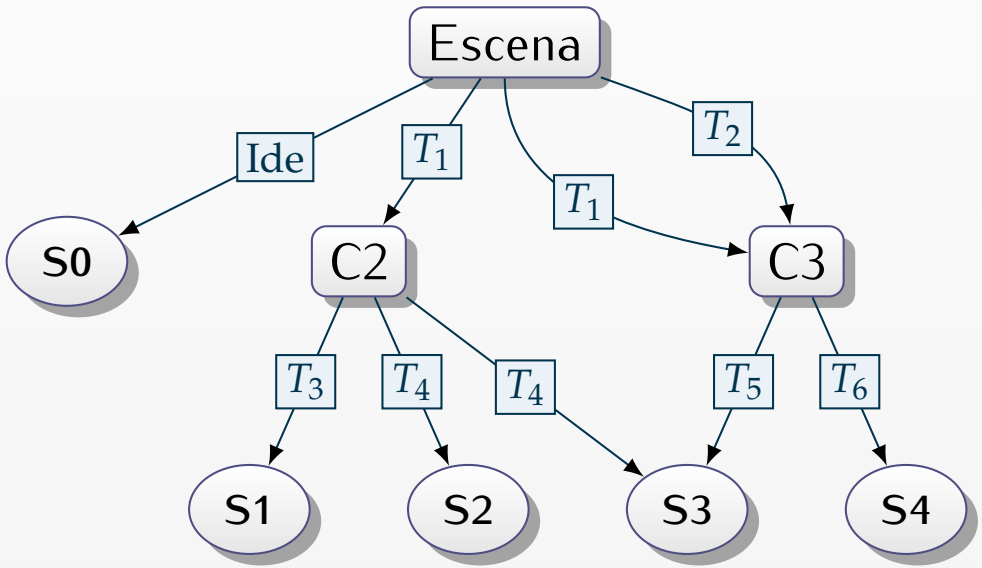
Cada objeto O_i del esquema anterior puede ser de **dos tipos** de objetos geométricos:

- **Objeto simple:** el objeto O_i es una malla u otros objetos que no están compuestos de otros objetos más simples.
- **Objeto compuesto:** el objeto O_i es una sub-escena, es decir, está compuesto de varios objetos que se instancian mediante diferentes transformaciones.
- Una escena ahora es un único objeto, que puede ser simple o compuesto (esto último es lo más frecuente).

Una escena es, por tanto, una estructura de tipo **Grafo Dirigido Acíclico** (*Directed Acyclic Graph*) o **DAG**.

Grafo de escena

La estructura que representa una de estas escenas se denomina **Grafo de Escena** (*Scene Graph*).



- cada objeto compuesto se identifica con un **nodo no terminal**
- cada objeto simple se identifica con un **nodo terminal**
- cada arco se etiqueta con una **transformación geométrica**.

Instanciaciones en el grafo

En el grafo anterior:

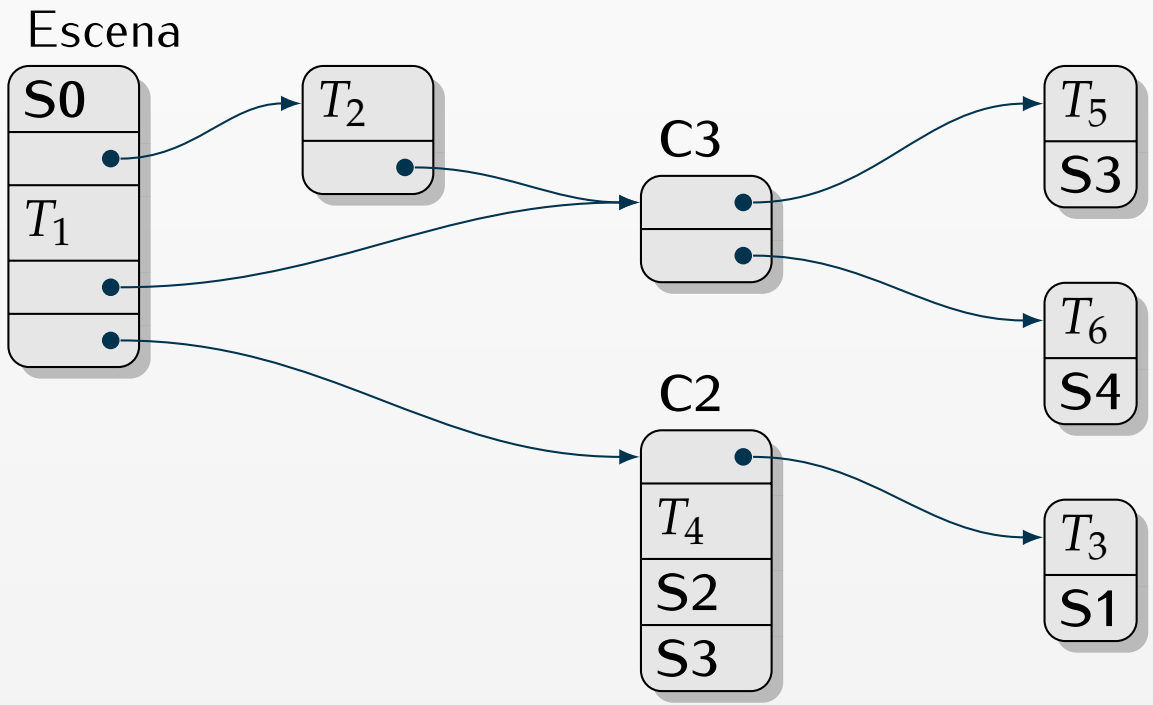
- Algunas transformaciones pueden ser la matriz identidad (Ide)
- Algunos pares de hermanos aparecen instanciados con la misma transformación.
- Algunos nodos (simples o compuestos) aparecen instanciados más de una vez (en el mismo o distinto padre)
- La transformaciones por las que se instancia a un nodo en la escena (marco $\mathcal{W}$), se obtienen **siguiendo todos los caminos posibles desde la raiz al nodo**. A modo de ejemplo: S3 aparece instanciado 3 veces, con estas transformaciones:
  - $T_1 \cdot T_4$
  - $T_1 \cdot T_5$
  - $T_2 \cdot T_5$

  (nótese que el efecto de la transformaciones debe leerse desde el nodo hacia la raíz).

Subsección 4.2
Grafos tipo PHIGS. Ejemplo.

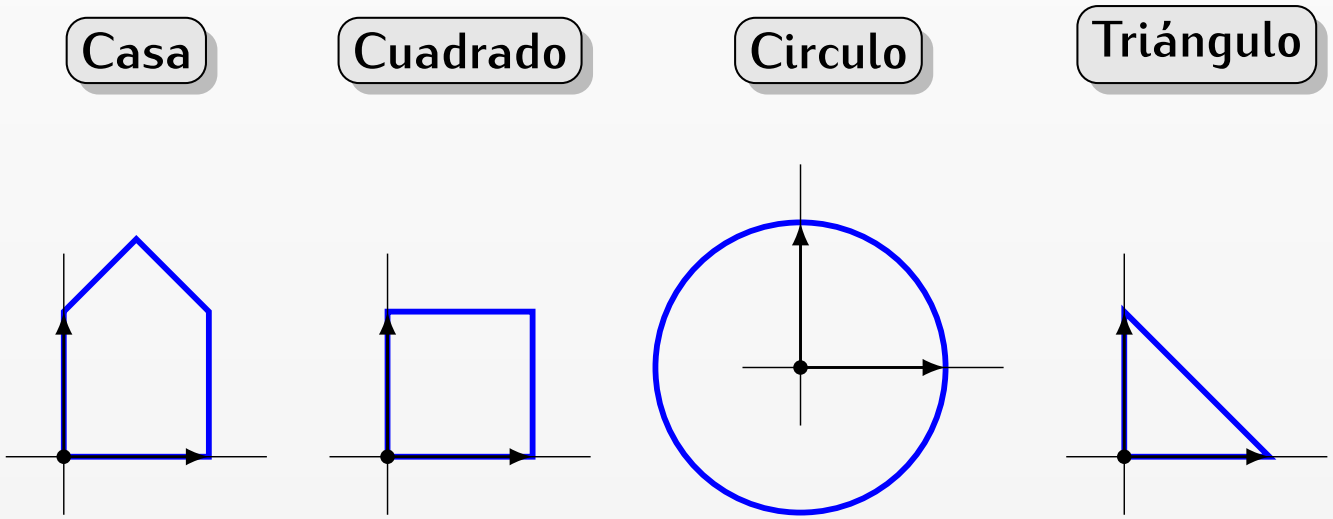
Notación inspirada en PHIGS para grafos de escena

Por su mayor simplicidad y proximidad a la implementación OpenGL, usaremos una notación inspirada en el antiguo estándar PHIGS. El grafo anterior se puede expresar de forma equivalente así:



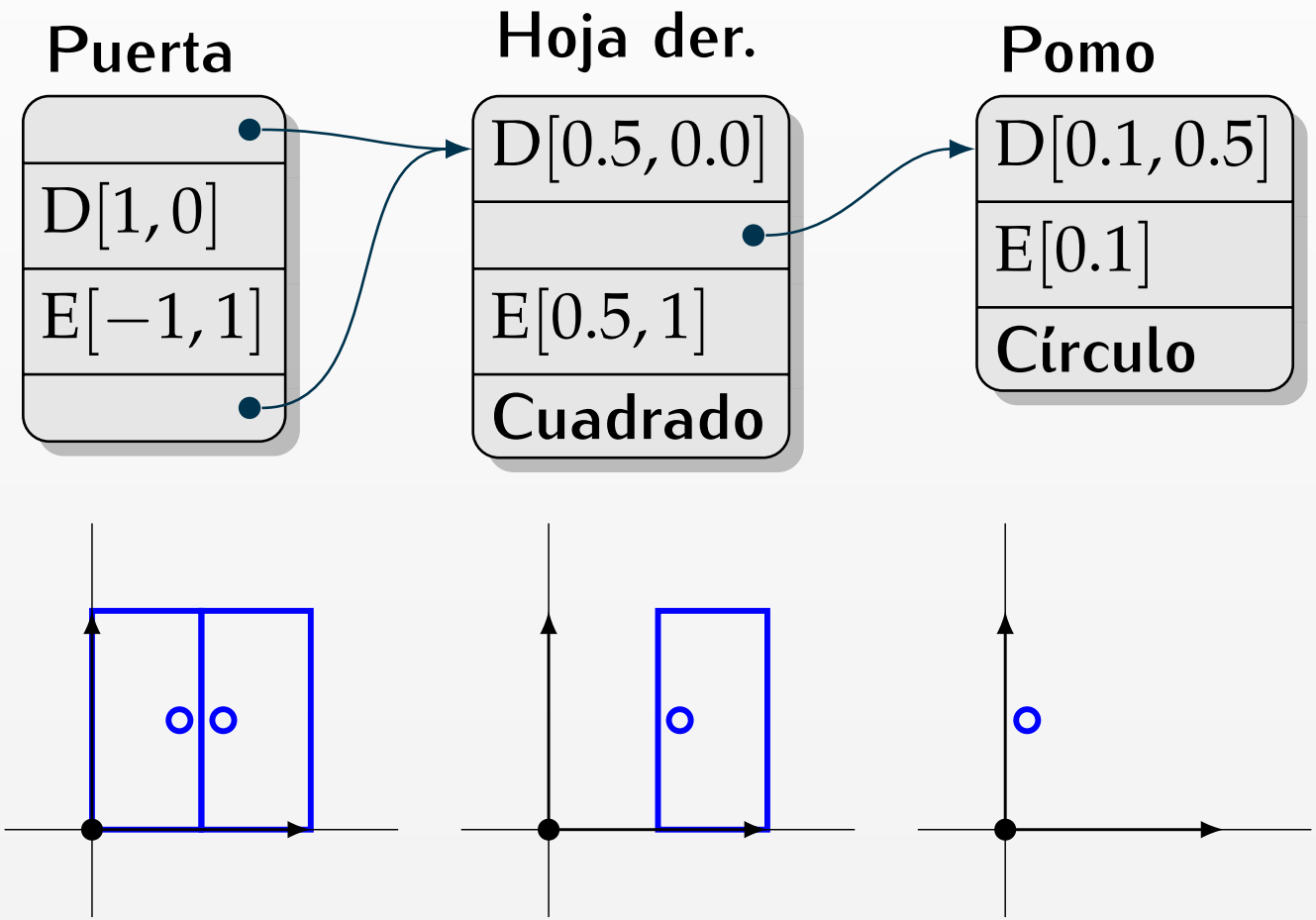
Ejemplo en 2D: objetos simples

Una escena puede estar formada por ún único objeto simple, en ese caso el grafo tiene un único nodo con una sola entrada (vemos tres ejemplos en 2D)



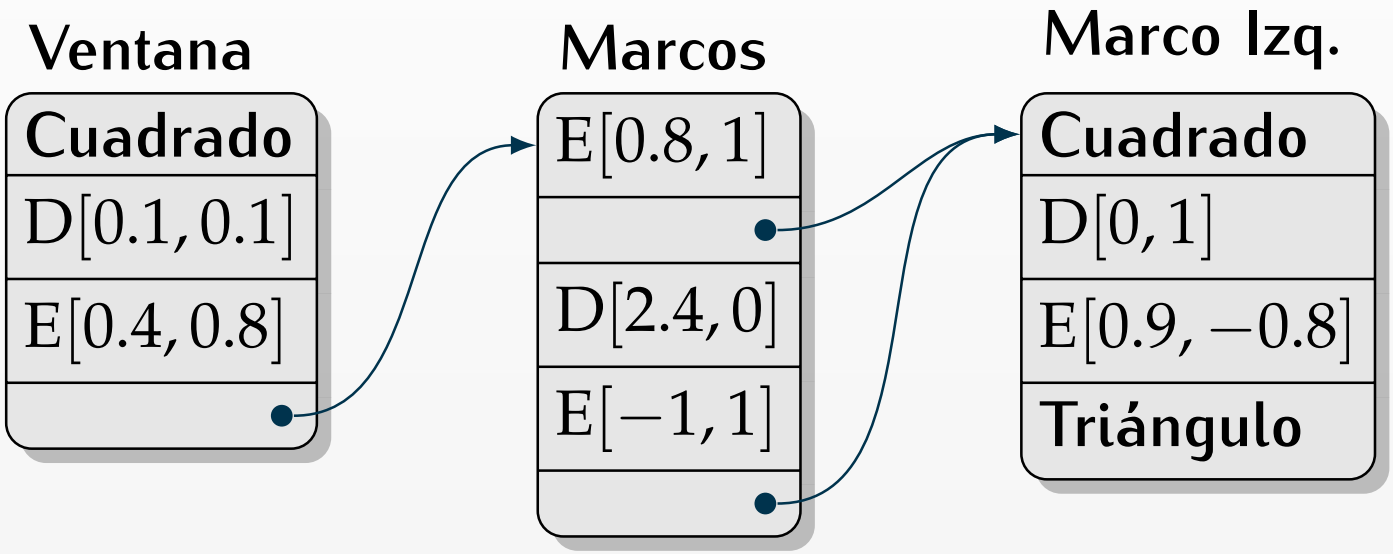
Objeto compuesto: Puerta

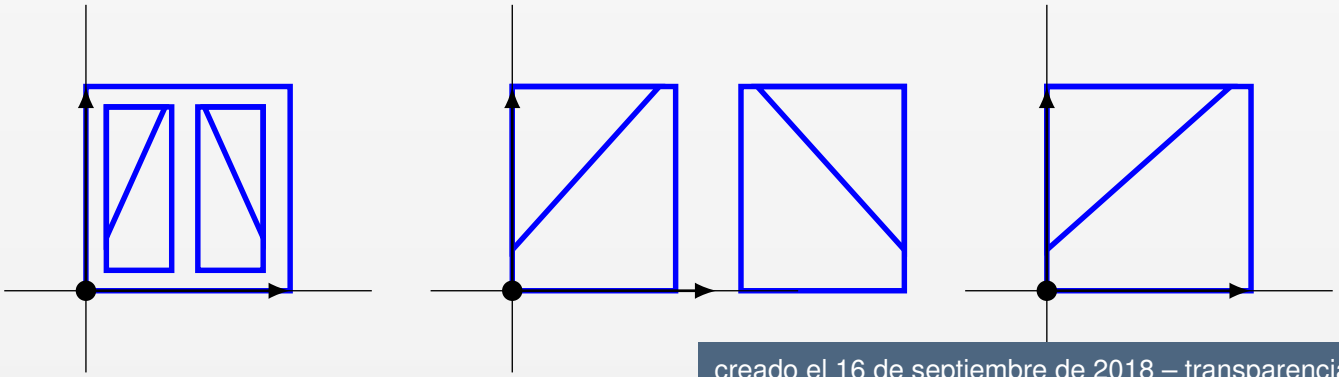
Este grafo define una puerta a partir de cuadrados y círculos:



Objeto compuesto: Ventana

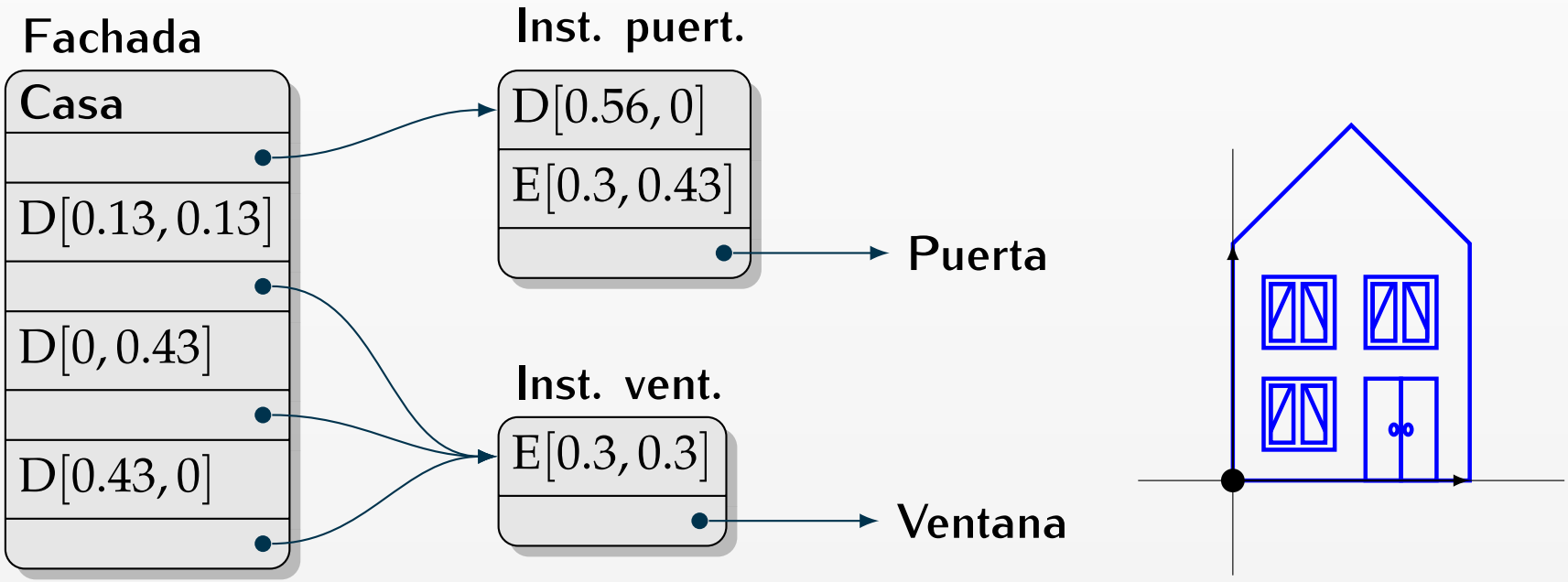
Una ventana hecha de cuadrados y triángulos:





Objeto compuesto: Fachada

Los dos objetos compuestos creados se pueden reutilizar en un objeto de más alto nivel:



Subsección 4.3
Representación de grafos.

Representación de grafos 3D

Cada nodo del grafo de escena es un tipo especial de **`Objeto3D`** con una lista o vector de entradas. Cada entrada puede ser de dos tipos:

- Un objeto 3D: se almacena un puntero a un **`Objeto3D`** (puede ser otro nodo, una malla o cualquier otro tipo de objeto).
- Una transformación: se usa un puntero a una matriz (**`Matriz4f`**).

Una escena se puede representar usando un puntero al nodo raíz. Un nodo puede verse como un objeto 3D compuesto de otros objetos y transformaciones.

Entradas de los nodos

Cada entrada del nodo puede ser una instancia de esta clase:

```
// Entrada del nodo del Grafo de Escena
struct EntradaNGE
{
   unsigned char tipoE ;  // 0=objeto, 1=transformacion
   union
   {  Objeto3D *  objeto ;  // ptr. a un objeto (propietario)
      Matriz4f *  matriz ;  // ptr. a matriz 4x4 transf. (prop.)
   } ;
   // constructores (uno por tipo)
   EntradaNGE( Objeto3D * pObjeto ) ;   // (copia solo puntero)
   EntradaNGE( const Matriz4f & pMatriz ); // (crea copia)
} ;
```

Los objetos 3D tipo nodo del grafo

Los nodos del grafo son básicamente vectores de entradas.

```
class NodoGrafoEscena : public Objeto3D
{
   protected:
     std::vector<EntradaNGE> entradas ;   // vector de entradas
   public:
     // visualiza usando OpenGL
     virtual void visualizarGL( ContextoVis & cv ) ;
     // añadir una entrada (al final).
     void agregar( EntradaNGE * entrada );// genérica
     // construir una entrada y añadirla (al final)
     void agregar( Objeto3D * pObjeto );         // objeto
     void agregar( const Matriz4f & pMatriz ); // matriz
} ;
```

Creación de estructuras

Para crear la estructura se pueden crear clases concretas derivadas de **`NodoGrafoEscena`**:

- Los constructores de dichas clases se encargan de crear las entradas.
- Cada constructor crea los sub-objetos de forma recursiva, así como las transformaciones necesarias.
- Si la estructura es de árbol, la liberación de la memoria puede hacerse recursivamente, si es un grafo acíclico, dicha liberación puede ser más complicada.

Ejemplo de creación

Suponiendo el mismo ejemplo de la casa:

- Las clases **`Cuadrado`** y **`Triangulo`** son clases derivadas de **`Objeto3D`**, son la primitivas de las que partimos (suponemos que incluyen un método para visualizar un cuadrado y un triángulo, respectivamente).
- A modo de ejemplo, vamos a ver como construir las clases **`Ventana`**, **`Marcos`** y **`MarcoIzq`**.
- La clase **`Fachada`** se podría construir de forma similar.

Para cada clase se define un constructor que crea la estructura.

- El constructor añade las entradas (mediante **`agregar`**) en el orden adecuado.
- Llama recursivamente los constructores de los sub-objetos.

Ejemplo de creación

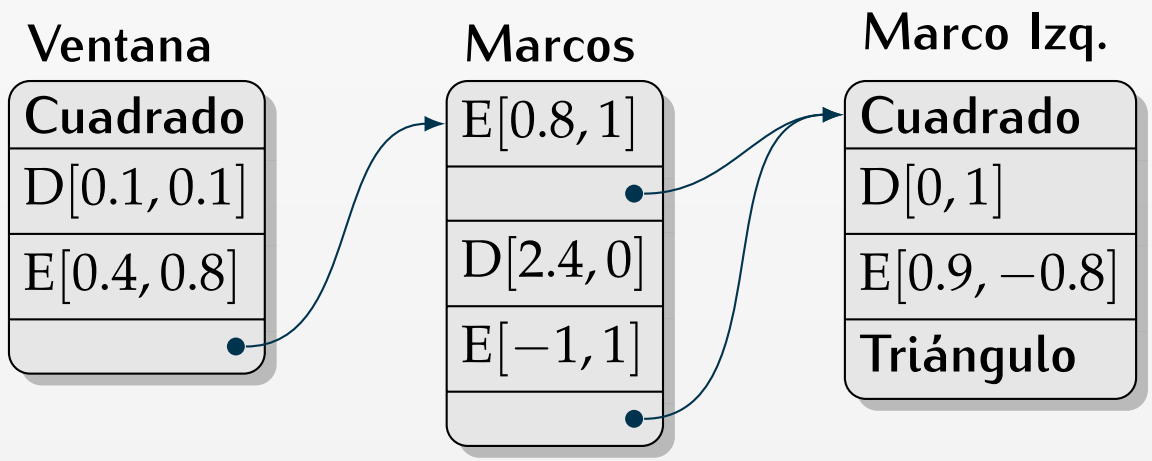
La declaración de las clases se puede hacer así:

```
// clase para el nodo del grafo etiquetado como Ventana
class Ventana : public NodoGrafoEscena
{  public:
   Ventana() ;  // constructor
} ;
// clase para el nodo del grafo etiquetado como Marcos
class Marcos : public NodoGrafoEscena
{  public:
   Marcos() ;  // constructor
} ;
// clase para el nodo del grafo etiquetado como Marco Izq.
class MarcoIzq : public NodoGrafoEscena
{  public:
   MarcoIzq() ;  // constructor
} ;
```

Implementación de constructores (1)

La implementación de los constructores se podría hacer así:

```
Ventana::Ventana()
{
 agregar( new Cuadrado );                          // Cuadrado
 agregar( MAT_Traslacion( 0.1,1.0,0.0 ) );         // D[0.1,0.1]
 agregar( MAT_Escalado( 0.4,0.8,1.0 ) );           // E[0.4,0.8]
 agregar( new Marcos );                            // Marcos
}
```



Implementación de constructores (2)

```
MarcoIzq::MarcoIzq()
{
 agregar( new Cuadrado );                    // Cuadrado
 agregar( MAT_Traslacion( 0.0,1.0,0.0 ) ); // D[0,1]
 agregar( MAT_Escalado( 0.9,-0.8,1.0 ) );  // E[0.9,−0.8]
 agregar( new Triangulo );                   // Triangulo
}
Marcos::Marcos()
{
 agregar( MAT_Escalado( 0.8,0.1,1.0 ) );     // E[0.8,0.1]
 agregar( new MarcoIzq );                    // Marco Izq.
 agregar( MAT_Traslacion( 2.4,0.0,0.0 ) ); // D[2.4,0]
 agregar( MAT_Escalado( -1.0,1.0,1.0 ) );   // E[−1,1]
 agregar( new MarcoIzq );                    // Marco Izq.
}
```

Subsección 4.4
Visualización de grafos en OpenGL.

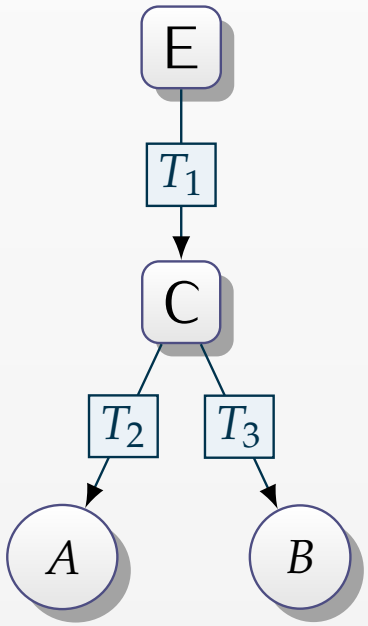
Visualización de varios objetos

La visualización de grafos en OpenGL se basa en operaciones que permiten guardar y recuperar la matriz modelview.

Supongamos que queremos visualizar dos objetos A y B:

- Para el objeto A usamos la matriz de modelado $N_A = T_1 \cdot T_2$
- Para el objeto B usamos la matriz de modelado $N_B = T_1 \cdot T_3$
- Para ambos queremos usar un matriz de vista V.

Esta situación es típica si A y B están en distintas ramas de un sub-árbol en un grafo de escena.



Duplicación de matrices

Para hacer la visualización con OpenGL se debe usar este código:

```
glMatrixMode( GL_MODELVIEW );

glLoadIdentity() ;      // M := Ide
glMultMatrixf( V );     // M := M · V
glMultMatrixf( T1 );    // M := M · T1
glMultMatrixf( T2 );    // M := M · T2
VisualizarObjeto( A );  // visualizar A con M == V · T1 · T2

glLoadIdentity() ;      // M := Ide
glMultMatrixf( V );     // M := M · V
glMultMatrixf( T1 );    // M := M · T1
glMultMatrixf( T3 );    // M := M · T3
VisualizarObjeto( B );  // visualizar B con M == V · T1 · T3
```

Es decir: **es necesario replicar llamadas para acumular en M las matrices V y T_1.**

Guardar y recuperar la matriz Modelview

Para evitar esas repeticiones (es lento), se dispone de dos instrucciones que permiten en guardar la matriz de OpenGL modelview M y restaurarla después:

```
glMatrixMode( GL_MODELVIEW );
glLoadIdentity() ;      // M := Ide
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
   VisualizarObjeto( A );     // 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( T2 );        // M := M · T3
   VisualizarObjeto( B );     // visualizar B con M == V · T1 · T3
glPopMatrix() ;               // M := C (restaura copia de M)
```

Anidamiento de *push* y *pop*

El código entre *push* y *pop* es **neutro** en cuanto a la matriz M (es decir: no la modifica)

- es cierto **incluso cuando push y pop se anidan**
- para lo cual OpenGL **necesita tener en su estado una pila LIFO P de matrices** de transformación modelview (inicialmente vacía), en lugar de una sola matriz C:

```
// al inicio tope == 0
glMatrixMode( GL_MODELVIEW ); // push y pop actuarán sobre M
   ...
   glPushMatrix() ;    // P[tope]=M ; tope=tope+1
      ....
   glPopMatrix() ;     // tope=tope-1 ; M=P[tope]
   ....
```

(lógicamente, los *push* y *pop* deben estar balanceados)

Visualización de grafos de escena

Un programa que siempre visualiza el mismo grafo de escena (tipo PHIGS) puede implementarse trasladando dicho grafo a código de forma sencilla:

- Cada nodo se implementa con una secuencia de llamadas, entre operaciones *push* y *pop* (no modifica M)
- Una entrada correspondiente a un nodo simple (una malla), supone una llamada al procedimiento para visualizarla
- Las entradas correspondientes a transformaciones suponen acumular la correspondiente matriz en modelview
- Una entradas correspondiente a una referencia a otro nodo B puede traducirse en:
  - Una secuencia de llamadas correspondientes a B entre push y pop
  - Una llamada a un procedimiento específico del nodo B (esto mejor si el nodo B está referenciado desde más de un sitio, para no repetir código).

Ejemplo: el nodo *Fachada*

```
void Fachada()
{
 glPushMatrix();
   Casa();
   glPushMatrix(); // inst.puerta
     glTranslatef(0.56,0.0,0.0);
     glScalef(0.3,43,1.0);
     Puerta();
   glPopMatrix();
   glTranslatef(0.13,0.13,0.0);
   InstVentana() ;
   glTranslatef(0.0,0.43,0.0);
   InstVentana() ;
   glTranslatef(0.43,0.0,0.0);
   InstVentana() ;
 glPopMatrix();
}
```

```
void InstVentana()
{
 glPushMatrix();
   glScalef(0.3,0.3,1.0);
   Ventana();
 glPopMatrix();
}
```

Visualización de grafos

El método de visualización anterior presenta dos inconvenientes:

- El modelo jerárquico está prefijado en el código, que solo puede visualizar dichos modelos. Cambios en el modelo requieren cambios en el código.
- Cada vez que se visualizan los objetos es necesario recalcular las matrices (en el caso de las rotaciones, supone evaluar senos y cosenos).

A continuación veremos como visualizar con OpenGL los grafos de escena que se han introducido en esta sección.

- Esto permite visualizar grafos almacenados en archivos, creados con herramientas de diseño asistido. Las matrices solo se calculan una vez.

El método *visualizar* en grafos.

El método **visualizarGL** dibuja recursivamente estructuras completas.

```
void NodoGrafoEscena::visualizarGL( ContextoVis & cv )
{
 glMatrixMode( GL_MODELVIEW ); // operaremos sobre la modelview
 glPushMatrix() ;              // guarda modelview actual
 // recorrer todas las entradas del array que hay en el nodo:
 for( unsigned i = 0 ; i < entradas.size() ; i++ )
   if( entradas[i].tipoE == 0 )// si la entrada es sub-objeto:
     entradas[i].objeto->visualizarGL( cv ) ; // visualizarlo
   else                        // si la entrada es transformación:
   {  glMatrixMode( GL_MODELVIEW );    // modo modelview
      glMultMatrixf( *(entradas[i].matriz) ); // componerla
   }
 glMatrixMode( GL_MODELVIEW ); // operaremos sobre la modelview
 glPopMatrix();                // restaura modelview guardada
}
```

Subsección 4.5
Grafos parametrizables.

Modelos parametrizables

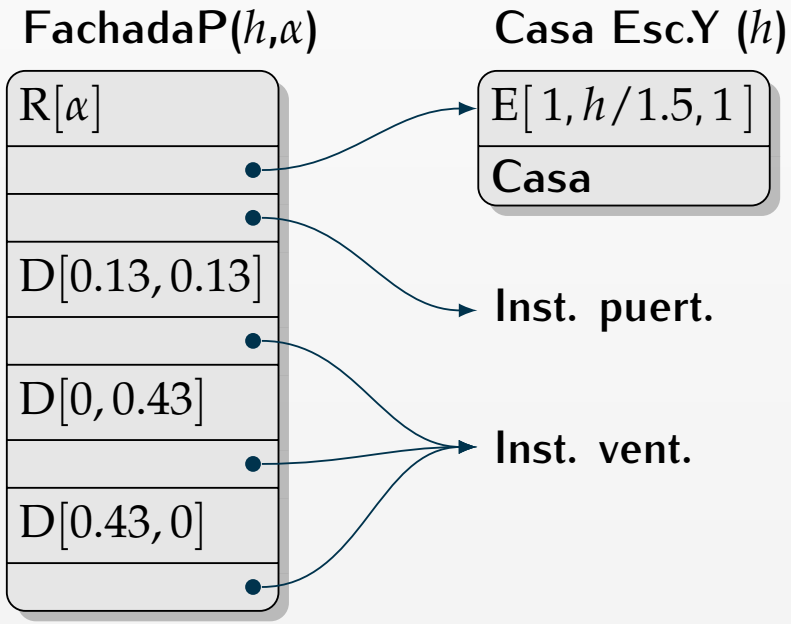
Un grafo de escena puede estar parametrizado respecto de ciertos valores reales:

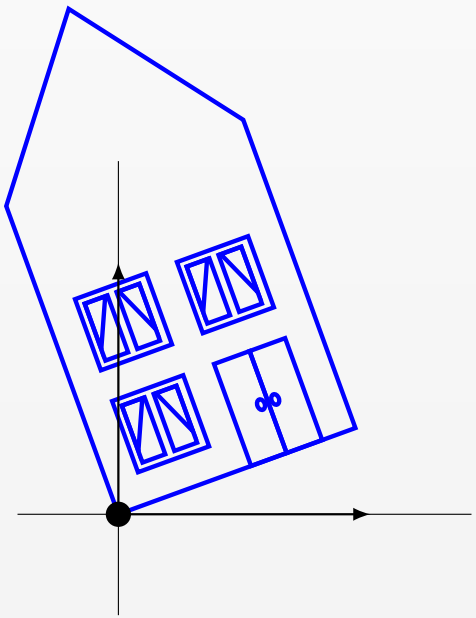
- Dichos valores puede controlar, por ejemplo, un transformación
  - Ángulo de rotación.
  - Factor de escala en una dimensión.
  - Distancia de traslación en una dirección dada.
- En otros casos pueden definir las dimensiones de un objeto.
- U otros valores, como por ejemplo la posición de puntos de control en objetos deformables.

Un mismo grafo de escena parametrizado se traduce en objetos con distinta geometría para distintos valores concretos de los parámetros.

Grafos parametrizados

Un grafo de escena puede venir definido por uno más parámetros (**grados de libertad**), usualmente valores reales. P.ej.: el objeto compuesto **FachadaP** admite dos parámetros: altura de **Casa** (h) y ángulo de rotación de (α):





Visualización de FachadaP

Usando exclusivamente código, se le añaden parámetros a las funciones para representar los grados de libertad:

```
void FachadaP( float h, float alpha )
{
 glPushMatrix();
   glRotatef(alpha,0,0,1);
   CasaEscY(h);
   InstPuerta() ;
   glTranslatef(0.13,0.13,0.0);
   InstVentana() ;
   glTranslatef(0.0,0.43,0.0);
   InstVentana() ;
   glTranslatef(0.43,0.0,0.0);
   InstVentana() ;
 glPopMatrix();
}
```

```
void CasaEscY( float h )
{
 glPushMatrix();
   glScalef(1,h/1.5,1);
   Casa();
 glPopMatrix();
}
```

Representacion de grafos parametrizables

Una transformación parametrizable en un grafo puede representarse con una clase derivada de **NodoGrafoEscena**, en la cual se añade:

- Un método para cambiar el valor de cada parámetro o conjunto de parámetros.
- Una variable de instancia con el valor del parámetro (si necesario).

A modo de ejemplo, la clase **FachadaP** podría quedar así:

```
class FachadaP : public NodoGrafoEscena
{
  protected:
   float h, alpha ;
  public:
   FachadaP( float h_inicial, float alpha_inicial );
   void fijarH( float h_nuevo ) ;
   void fijarAlpha( float alpha_nuevo );
} ;
```

Cambiar el valor de un parámetro

Estos métodos recalculan las matrices correspondientes del grafo en función de los nuevos valores de los parámetros.

```
void FachadaP::fijarAlpha( float alpha_nuevo )
{  // guardar nuevo valor de 'alpha'
   alpha = alpha_nuevo ;
   // insertar nueva matriz en la primera entrada del array
   *(entradas[0].matriz) = MAT_Rotacion( alpha_nuevo, 0,0,1 );
}
void FachadaP::fijarH( float h_nuevo )
{  // guardar nuevo valor (opcional)
   h = h_nuevo ;
   // cambiar matriz en el nodo hijo (segunda entrada, tipo CasaEscY)
   CasaEscY * hijo = (CasaEscY *) entradas[1].objeto ;
   hijo->fijarH( h_nuevo );
}
// falta: definir fijarH para nodo tipo CasaEscY
```

Subsección 4.6
Gestion de una pila de matrices.

Gestión de la pila

En las versiones de OpenGL que no disponen de las funciones `Push/Pop` es necesario usar funciones C/C++ para gestionar explicitamente la pila de transf. desde la aplicación.

- Dichas funciones gestionan una **matriz actual** y una **pila lifo** con las copias guardadas.
- Se permite **hacer push/pop** sobre la pila.
- Se permite **componer con la matriz actual** matrices de rotación, traslación, escalado, etc...
- Es necesario pasar como parámetro la matriz actual al vertex shader (debe existir una función para **consular la matriz actual**

Para todo esto, podemos usar de nuevo el tipo **`Matriz4f`**.

La clase para pilas LIFO de matrices

Puede tener esta forma que se indica aquí:

```
class PilaLIFOMatrices
{
   public:
   Matriz4f                actual ;   // matriz actual (inicialmente identidad)
   std::vector<Matriz4f> pila ;       // matrices apiladas (inicialmente vacía)

   // métodos:
   PilaLIFOMatrices() ;       // constructor
   void push( ) ;             // apilar la matriz actual en el tope
   void pop() ;               // restaurar matriz actual desde tope, y eliminar tope
   void asignaIdent() ;       // hacer matriz actual igual a la identidad

   // métodos de composición (actual = actual*matriz )
   void cMatriz( const Matriz4f & mat ) ;
   void cTraslacion( const float dx, const float dy, const float dz ) ;
   void cEscalado  ( const float sx, const float sy, const float sz ) ;
   void cRotacion  ( const float ang_gra,
                     const float ex, const float ey, const float ez ) ;
} ;
```

Visualización con pilas LIFO

El ejemplo de visualización procedural del objeto casa se puede adaptar para usar una pila lifo de matrices, en lugar de usar la funcionalidad de OpenGL

- Las funciones que visualizan cada sub-objeto aceptan como parámetro una pila de matrices.
- Las operaciones de apilamiento, desapilamiento y composición se hacen sobre dicha pila.
- La pila se pasa como parámetro a los sub-objetos de cada objeto.

Ejemplo sencillo (1)

Vemos, por ejemplo, una versión de la función que visualiza la puerta del ejemplo anterior:

```
void Puerta( PilaLIFOMatrices & pila )
{
   pila.push();
      Hoja_der( pila );
      pila.cTraslacion(1.0,0.0,0.0);
      pila.cEscalado(-1.0,1.0,1.0);
      Hoja_der( pila );
   pila.pop();
}
```

- Se asume que los vertex/fragment shaders están activados
- Se usan las funciones que hemos creado en lugar de las incluidas en OpenGL.

Ejemplo sencillo (2)

```
void Fachada( PilaLIFOMatrices & pila )
{
  pila.push();
   pila.push();
      Casa_mu( pila );
      pila.push();
      pila.cTraslacion(0.56,0.0,0.0);
      pila.Escalado(0.3,43,1.0);
      Puerta( pila );
   pila.pop();
   pila.cTraslacion(0.13,0.13,0.0);
   InstVentana( pila ) ;
   pila.cTraslacion(0.0,0.43,0.0);
   InstVentana( pila ) ;
   pila.cTraslacion(0.43,0.0,0.0);
   InstVentana( pila ) ;
  pila.pop();
}
```

Asignación de la matriz en nodos terminales

Antes de visualizar las primitivas:

- Es necesario actualizar la matriz de modelado `matrizModelado` (parámetro uniform del vertex shader).
- Se puede hacer con **`asignarUniform`** y **`pila.actual`**.

Aquí vemos una variante de la función **`Casa`** de antes:

```
void Casa_mu( PilaLIFOMatrices & pila )
{
   asignarUniform( idProg, "matrizModelado", pila.actual );
   Casa() ;  // visualiza las prims. (sin apilar más transf.)
}
```

Visualización de un grafo usando una pila.

Los grafos de escena se pueden visualizar usando una pila. La pila de matrices *modelview* a usar puede ser una componente del contexto de visualización.

- Por tanto, dicho contexto contendrá el modo de visualización, la pila, y posiblemente otros datos:
- Puede declararse así:

```
class ContextoVis
{
  public:
     // modo de visualización (alambre, sólido, etc....)
     unsigned modo_vis ;
     // pila la modelview
     PilaLIFOMatrices pilaModelview ;
     // otros datos:
     //.....
} ;
```

Visualización de grafos usando una pila

El método **visualizarGL** sería igual que antes pero reemplazando la funcionalidad de OpenGL para **push**/**pop**:

```
void NodoGrafoEscena::visualizarGL( ContextoVis & cv )
{
   cv.pilaModelview.push() ;
   for( unsigned i = 0 ; i < entradas.size() ; i++ )
     if( entradas[i].tipoE == 0 ) // sub-objeto: visualizarlo
       entradas[i].objeto->visualizarGL( cv ) ;
     else // transformación: componerla
       cv.pilaModelview.cMatriz( *(entradas[i].matriz) );
   cv.pilaModelview.pop() ;
}
```

las primitivas o las mallas deben usar la matriz **cv.actual** para fijar la matriz *modelview* del *shader*.

Fin de la presentación.